SPARC
RISC

USER'S GUIDE

hyperSPARC Edition

ROSS
TECHNOLOGY, INC.

SPARC RISC USER'S GUIDE

ROSS Technology, Inc.

A Fujitsu Limited Company

ROSS Technology, Inc. • 5316 Hwy. 290 West • Austin, TX 78735 • PH 512-892-7802 • FAX 512-892-3036

Third Edition – September 1993

Products bearing the SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.
SPARC is a registered trademark of SPARC International, Inc.
hyperSPARC is a trademark of SPARC International, Inc. used under permission by ROSS Technology, Inc.

Numerous features of hyperSPARC technology are covered by US Patent No. 5,226,142 and other patents pending.

ROSS Technology, Inc. is a subsidiary of Fujitsu Limited

© ROSS Technology, Inc., 1993. The information contained herein is subject to change without notice. ROSS Technology, Inc. assumes no responsibility for the use of any circuitry other than circuitry embodied in a ROSS Technology, Inc. product. Nor does it convey or imply any license under patent or other rights. "ROSS Technology does not authorize its products for use as critical components in life support systems where a malfunction or failure of the product may reasonably be expected to result in significant injury to the user. The inclusion of ROSS Technology products in life support systems applications implies that the manufacturer assumes all risk of such use and in so doing indemnifies ROSS Technology against all damages."

Table of Contents

Chapter 1: Introduction

1.1 SPARC Architecture Features 1-2
1.1.1 Load/Store Architecture 1-2
1.1.2 Register Windows 1-2
1.1.3 Instruction Set 1-3
1.1.4 Arithmetic /Logical /Shift Instructions 1-3
1.1.5 Control Transfer Instructions 1-3
1.1.6 Read/Write Control Register Instructions 1-3
1.1.7 Floating-Point-Operate and Coprocessor-Operate Instructions 1-3

1.2 hyperSPARC Overview 1-4
1.2.1 hyperSPARC Design Features 1-5

1.3 CY7C600 Overview 1-9
1.3.1 Partitioning 1-9

Chapter 2: SPARC Programming Environment

2.1 Programming Model 2-2
2.1.1 Supervisor/User Modes 2-2
2.1.2 Register Windows 2-3
2.1.3 Floating-Point Register File (FREGS) 2-7

2.2 SPARC Control/Status Registers 2-7
2.2.1 Integer Unit Control/Status Registers 2-7
2.2.2 FPU Control/Status Registers 2-12
2.2.3 Cache Controller/MMU Control 2-14

2.3 SPARC Data Types 2-14
2.3.1 Data Organization In Registers 2-14
2.3.2 Data Organization In Memory 2-16
2.3.3 SPARC Floating-Point Data Types 2-16

2.4 SPARC Instruction Set 2-20
2.4.1 Instruction Formats 2-20
2.4.2 Addressing 2-22
2.4.3 Instruction Types 2-25
2.4.4 SPARC Instruction OP Codes 2-39
2.4.5 SPARC Exception Model 2-51

Chapter 3: RT620 hyperSPARC Central Processing Unit

3.1 RT620 hyperSPARC CPU 3-1

3.2 Integer Data Path (IDP) 3-4
3.2.1 Arithmetic and Logical Unit (ALU) 3-4
3.2.2 Load and Store Unit (LSU) 3-4
3.2.3 Program Counter Unit (PCU) 3-5
3.2.4 Special Register Unit (SRU) 3-6

3.3 Instruction Fetch Unit (IFETCH) 3-6

3.4 Instruction Scheduler (ISCHED) 3-8
3.4.1 Single Instruction Launch Group 3-10
3.4.2 Multiple Instruction Launch Group 3-10
3.4.3 Interlocks and Dependencies 3-11
3.4.4 Special Features 3-13

3.5 Floating-Point Unit (FPU) 3-15
3.5.1 Floating-Point Instruction Decode-Schedule-and-Dispatch Controller (FPSCHED) 3-17
3.5.2 Floating-Point Execution Units 3-23

3.6 Instruction Cache (ICACHE) 3-24
3.6.1 Virtual Addresses 3-25
3.6.2 ICACHE Hits 3-25
3.6.3 Instruction Cache Control Register 3-26
3.6.4 On-Chip Instruction Cache Read/Write Diagnostic Support 3-26

3.7 hyperSPARC Signal Descriptions 3-30
3.7.1 hyperSPARC CPU Pinouts 3-30
3.7.2 hyperSPARC Intra-Module Bus (IMB) 3-33
3.7.3 hyperSPARC CPU Bus Timing Waveforms 3-36

3.8 Instruction Pipelines 3-58
3.8.1 Instruction Fetch Timing 3-58
3.8.2 Integer Instruction Pipeline 3-61
3.8.3 ALU Instructions 3-62
3.8.4 Load Instructions 3-66
3.8.5 Store Instructions 3-66
3.8.6 Atomic Load-Store Instructions 3-66
3.8.7 Branch Instructions 3-68
3.8.8 CALL Instructions 3-68
3.8.9 JMPL/RETT/FLUSH Instructions 3-68
3.8.10 Save/Restore Instructions 3-70
3.8.11 Read/Write Special Register Instructions 3-72
3.8.12 Special Feature Pipelines 3-72
3.8.13 Floating-Point Instruction Pipelines 3-75

3.9 Traps and Interrupts 3-77
3.9.1 Machine State at Reset 3-78
3.9.2 Exception Pipeline 3-79
3.9.3 Trap Operation 3-79

3.9.4 Error Mode 3-79
3.9.5 Trap Priorities 3-80
3.9.6 Precise Traps 3-80
3.9.7 Interrupting Traps (Asynchronous) 3-81
3.9.8 Floating-Point Unit Traps (Deferred Traps) 3-82
3.9.9 IU-FPU Exception Flush Logic 3-100
3.9.10 RT620 Method for Handling Traps and Error Mode 3-100

Chapter 4: RT625 hyperSPARC Cache Controller, Memory Management, and Tag Unit

4.1 RT625 hyperSPARC CMTU 4-1

4.2 RT625 Memory Management Unit 4-4
4.2.1 Translation Lookaside Buffer (TLB) 4-4
4.2.2 Table Walk 4-8
4.2.3 Page Table Pointer (PTP) 4-11
4.2.4 Page Table Entry (PTE) 4-12
4.2.5 Page Table Pointer Cache (PTPC) 4-13

4.3 RT625 MMU Operation Modes 4-16
4.3.1 MMU Invalidate and Probe Operations 4-17

4.4 RT625 Cache Controller 4-18
4.4.1 RT625 Cache Modes 4-18
4.4.2 RT625 Cache Controller 4-19
4.4.3 RT625 Software Cache Flushing Operations 4-34
4.4.4 RT625 Cacheable/Non-Cacheable Memory Accesses 4-34
4.4.5 RT625 MBus Cacheable (MC) Bit 4-35
4.4.6 RT625 LDST (Atomic Load-Store Instruction) Cycles 4-35
4.4.7 RT625 Cache Byte Write Enables 4-36
4.4.8 Cache Data Forwarding 4-37

4.5 RT625 Registers 4-37
4.5.1 RT625 System Control Register (SCR) 4-37
4.5.2 RT625 Context Table Pointer Register (CTPR) 4-39
4.5.3 RT625 Context Register (CXR) 4-39
4.5.4 RT625 Reset Register (RR) 4-39
4.5.5 RT625 Root Pointer Register (RPR) 4-39
4.5.6 RT625 Instruction access PTPs (IPTP0, IPTP1) 4-40
4.5.7 RT625 Data Access PTPs (DPTP0, DPTP1) 4-40
4.5.8 RT625 Index Tag Registers (ITR0, ITR1) 4-40
4.5.9 RT625 TLB Replacement Control Register (TRCR) 4-40
4.5.10 RT625 Synchronous Fault Status Register (SFSR) 4-41
4.5.11 RT625 Synchronous Fault Address Register (SFAR) 4-42
4.5.12 RT625 Asynchronous Fault Status Register (AFSR) 4-42
4.5.13 RT625 Asynchronous Fault Address Register (AFAR) 4-42

4.6 RT625 Block Copy and Block Fill 4-43
4.6.1 RT625 Block Copy 4-43
4.6.2 RT625 Block Fill 4-43

4.7 RT625 Diagnostic Support 4-44
4.7.1 RT625 MMU TLB Entries 4-44
4.7.2 RT625 Cache Tag Entries 4-45
4.7.3 CDU Cache Data Entries 4-45

4.8 RT625 Reset **4-45**
4.8.1 Power-On Reset (RSTIN) 4-45
4.8.2 Watchdog Reset (WDR) 4-46
4.8.3 Software Internal Reset (SIR) 4-46

4.9 RT625 ASI and Register Mapping **4-46**

4.10 Synchronous Faults **4-48**
4.10.1 Synchronous Fault Cases 4-51

4.11 RT625 Pinouts **4-56**
4.11.1 Pin Description 4-56
4.11.2 Pin Virtual Bus (Intra-Module Bus) Operation 4-62

Chapter 5: RT627 hyperSPARC Cache Data Unit

5.1 RT627 Pinouts **5-4**

5.2 RT627 Access Waveforms **5-5**

Chapter 6: CY7C601/611 Integer Unit

6.1 CY7C601/CY7C611 Register Set **6-2**
6.1 CY7C601/CY7C611 Cycle Per Instruction (CPI) 6-3

6.2 Signal Description **6-5**
6.2.1 Memory Subsystem Interface Signals 6-7
6.2.2 Floating-Point/Coprocessor Interface Signals 6-11
6.2.3 Interrupt and Control Signals 6-14
6.2.4 Power and Clock Signals 6-15

6.3 Pipeline And Instruction Execution Timing **6-16**
6.3.1 Stages 6-17
6.3.2 Multicycle Instructions 6-18
6.3.3 Pipeline Freezes 6-22
6.3.4 Traps 6-22

6.4 Bus Operation And Timing **6-22**
6.4.1 Instruction Fetch 6-25
6.4.2 Load 6-25
6.4.3 Load with Interlock 6-25
6.4.4 Load Double 6-26
6.4.5 Store 6-27
6.4.6 Store Double 6-27
6.4.7 Atomic Load-Store 6-28
6.4.8 Floating-Point Operations 6-29
6.4.9 Bus Arbitration 6-30
6.4.10 Load with Cache Miss 6-31
6.4.11 Store with Cache Miss 6-32
6.4.12 Memory Exceptions 6-34
6.4.13 Floating-Point Exceptions 6-39
6.4.14 Interrupts 6-39
6.4.15 Reset Condition 6-40
6.4.16 Error Condition 6-40

6.5 Exception Model 6-42
6.5.1 Reset 6-42
6.5.2 Synchronous Traps 6-42
6.5.3 Interrupts (Asynchronous Traps) 6-45
6.5.4 Floating-Point/Coprocessor Traps 6-46
6.5.5 Trap Operation 6-47

6.6 Coprocessor Interface 6-49
6.6.1 Protocol 6-49
6.6.2 Register Model 6-50
6.6.3 Exceptions 6-51

6.7 CY7C611 Integer Unit for Embedded Control 6-52

Chapter 7: CY7C602 Floating-Point Unit

7.1 CY7C602 Functional Description 7-2

7.2 Floating-Point/Integer Unit Interface 7-4
7.2.1 CY7C602 Instruction Fetch and Execution 7-5
7.2.2 Instruction Pipeline Flush 7-10

7.3 CY7C602 Programming Model 7-14
7.3.1 CY7C602 Registers 7-14
7.3.2 CY7C602 Floating-Point Instructions 7-18
7.3.3 CY7C602 Internal Operation 7-19
7.3.4 CY7C602 Exception Cases 7-21

7.4 CY7C602 Signal Descriptions 7-22
7.4.1 Integer Unit Interface Signals 7-22
7.4.2 Coprocessor Interface Signals 7-23
7.4.3 System/Memory Interface Signals 7-24

Chapter 8: CY7C604/CY7C605 Cache Controller and Memory Management Unit

8.1 Memory Management Unit 8-3
8.1.1 Translation Lookaside Buffer (TLB) 8-4
8.1.2 Table Walk 8-9
8.1.3 Page Table Pointer (PTP) 8-10
8.1.4 Page Table Entry (PTE) 8-11
8.1.5 Page Table Pointer Cache (PTPC) 8-12

8.2 MMU Operation Modes 8-15
8.2.1 MMU Flush and Probe Operations 8-16

8.3 CY7C604 / CY7C605 Cache Controllers 8-17
8.3.1 CY7C604/605 Cache Modes 8-18
8.3.2 CY7C604 Cache Controller 8-18
8.3.3 CY7C605 Cache Controller 8-23
8.3.4 CY7C604/CY7C605 Cache Control Signals 8-35
8.3.5 CY7C604/605 Write Buffer 8-37
8.3.6 CY7C604/605 Read Buffer 8-38
8.3.7 CY7C604/605 Cache Flushing Operations 8-38
8.3.8 CY7C604/605 Cacheable/Non-Cacheable Memory Accesses 8-39
8.3.9 CY7C604/605 MBus Cacheable (MC) Bit 8-39
8.3.10 CY7C604/605 LDST (Atomic Load-Store Instruction) cycles 8-40
8.3.11 CY7C604/605 Cache Byte Write Enables 8-40

8.4 CY7C604 / CY7C605 Registers 8-41
8.4.1 CY7C604 System Control Register (SCR) 8-41
8.4.2 CY7C605 System Control Register (SCR) 8-42
8.4.3 CY7C604/605 Context Table Pointer Register (CTPR) 8-44
8.4.4 CY7C604/605 Context Register (CXR) 8-44
8.4.5 CY7C604/605 Reset Register (RR) 8-44
8.4.6 CY7C604/605 Root Pointer Register (RPR) 8-45
8.4.7 CY7C604/605 Instruction access PTP (IPTP) 8-45
8.4.8 CY7C604/605 Data access PTP (DPTP) 8-45
8.4.9 CY7C604/605 Index Tag Register (ITR) 8-45
8.4.10 CY7C604/605 TLB Replacement Control Register (TRCR) 8-46
8.4.11 CY7C604/605 Synchronous Fault Status Register (SFSR) 8-46
8.4.12 CY7C604/605 Synchronous Fault Address Register (SFAR) 8-47
8.4.13 CY7C604/605 Asynchronous Fault Status Register (AFSR) 8-47
8.4.14 CY7C604/605 Asynchronous Fault Address Register (AFAR) 8-47

8.5 CY7C604 / CY7C605 Multichip Configuration 8-48
8.5.1 System Initialization 8-48
8.5.2 Cache Configurations 8-51

8.6 CY7C604/605 Diagnostic Support 8-52
8.6.1 CY7C604/605 MMU TLB Entries 8-52
8.6.2 CY7C604/605 Cache Tag Entries 8-53
8.6.3 CY7C604/605 Cache Data Entries 8-53

8.7 CY7C604/605 Reset 8-54
8.7.1 Power-On Reset (POR) 8-54
8.7.2 Watchdog Reset (WDR) 8-54
8.7.3 Software Internal Reset (SIR) 8-54
8.7.4 Software External Reset (SER) (CY7C604 only) 8-55
8.7.5 CY7C604/605 Reset in Multichip Configuration 8-55

8.8 CY7C604/605 ASI and Register Mapping 8-55

8.9 Synchronous Faults 8-56
8.9.1 Synchronous Fault Cases 8-59

8.10 CY7C604/605 Pin Definitions **8-64**

8.11 Virtual Bus Operation **8-71**

Chapter 9: CY7C157 Cache Storage Unit

9.1 Description Of Part **9-2**

9.2 Operation **9-2**

9.3 Signal Descriptions **9-2**

Chapter 10: SPARC MBus CPU Modules

10.1 hyperSPARC Modules **10-1**
10.1.1 hyperSPARC Module Description 10-2
10.1.2 hyperSPARC Module Design 10-2
10.1.3 hyperSPARC System Design Considerations 10-3

10.2 CYM600X Modules **10-4**
10.2.1 CYM600X Module Design 10-5
10.2.2 System Design Considerations 10-5

10.3 CYM6111 Multi-Die Package CPU **10-6**
10.3.1 Multi-Die Packaging Technology 10-6

Chapter 11: MBus Operation

11.1 MBus Principles **11-1**
11.1.1 MBus Level 1 Overview 11-1
11.1.2 MBus Level 2 Overview 11-2
11.1.3 MBus Physical Signal Summary 11-3
11.1.4 MBus Multiplexed Signal Summary 11-7
11.1.5 MBus Address Cycle 11-7
11.1.6 MBus Data Cycle 11-10
11.1.7 MBus Transactions 11-11
11.1.8 MBus Acknowledgement Cycles 11-19
11.1.9 MBus Arbitration 11-21
11.1.10 MBus Configuration Address Map 11-22
11.1.11 MBus Transaction Timing 11-24

Chapter 12: SPARC Instruction Set

12.1 Assembly Language Syntax **12-1**
12.1.1 Register Names 12-1
12.1.2 Special Symbol Names 12-2
12.1.3 Values 12-2
12.1.4 Label 12-2
12.1.5 Instruction Mnemonics 12-3

12.2 Definitions **12-4**

ADD	Add	12-8
ADDcc	Add and modify icc	12-9
ADDX	Add with Carry	12-10
ADDXcc	Add with Carry and modify icc	12-11
AND	And	12-12
ANDcc	And and modify icc	12-13
ANDN	And Not	12-14
ANDNcc	And Not and modify icc	12-15
Bicc	Integer Conditional Branch	12-16
CALL	Call	12-18
CBccc	Coprocessor Conditional Branch	12-19
CPop	Coprocessor Operate	12-21
FABSs	Absolute Value Single	12-22
FADDd	Add Double	12-23
FADDq	Add Quad	12-24
FADDs	Add Single	12-25
FBfcc	Floating-Point Conditional Branch	12-26
FCMPd	Compare Double	12-28
FCMPEd	Compare Double and Exception if Unordered	12-29
FCMPEq	Compare Quad and Exception if Unordered	12-30
FCMPEs	Compare Single and Exception if Unordered	12-31
FCMPq	Compare Quad	12-32
FCMPs	Compare Single	12-33
FDIVd	Divide Double	12-34
FDIVq	Divide Quad	12-35
FDIVs	Divide Single	12-36
FdMULq	Multiply Double to Quad	12-37
FdTOi	Convert Double to Integer	12-38
FdTOq	Convert Double to Quad	12-39
FdTOs	Convert Double to Single	12-40
FiTOd	Convert Integer to Double	12-41
FiTOq	Convert Integer to Quad	12-42
FiTOs	Convert Integer to Single	12-43
FLUSH	Instruction Cache Flush	12-44
FMOVs	Move	12-45
FMULd	Multiply Double	12-46
FMULq	Multiply Quad	12-47
FMULs	Multiply Single	12-48
FNEGs	Negate	12-49
FqTOd	Convert Quad to Double	12-50
FqTOi	Convert Quad to Integer	12-51
FqTOs	Convert Quad to Single	12-52
FsMULd	Multiply Single with Double	12-53
FSQRTd	Square Root Double	12-54
FSQRTq	Square Root Quad	12-55
FSQRTs	Square Root Single	12-56
FsTOd	Convert Single to Double	12-57

FsTOi	Convert Single to Integer	12-58
FsTOq	Convert Single to Quad	12-59
FSUBd	Subtract Double	12-60
FSUBq	Subtract Quad	12-61
FSUBs	Subtract Single	12-62
JMPL	Jump and Link	12-63
LD	Load Word	12-64
LDA	Load Word from Alternate space	12-65
LDC	Load Coprocessor Register	12-66
LDCSR	Load Coprocessor State Register	12-67
LDD	Load Doubleword	12-68
LDDA	Load Doubleword from Alternate space	12-69
LDDC	Load Doubleword Coprocessor	12-70
LDDF	Load Doubleword Floating-Point	12-71
LDF	Load Floating-Point Register	12-72
LDFSR	Load Floating-Point State Register	12-73
LDSB	Load Signed Byte	12-74
LDSBA	Load Signed Byte from Alternate space	12-75
LDSH	Load Signed Halfword	12-76
LDSHA	Load Signed Halfword from Alternate space	12-77
LDSTUB	Atomic Load-Store Unsigned Byte	12-78
LDSTUBA	Atomic Load-Store Unsigned Byte in Alternate space	12-79
LDUB	Load Unsigned Byte	12-80
LDUBA	Load Unsigned Byte from Alternate space	12-81
LDUH	Load Unsigned Halfword	12-82
LDUHA	Load Unsigned Halfword from Alternate space	12-83
MULScc	Multiply Step and modify icc	12-84
OR	Inclusive-Or	12-85
ORcc	Inclusive-Or and modify icc	12-86
ORN	Inclusive-Or Not	12-87
ORNcc	Inclusive-Or Not and modify icc	12-88
RDASR	Read Ancillary State Register	12-89
RDPSR	Read Processor State Register	12-90
RDTBR	Read Trap Base Register	12-91
RDWIM	Read Window Invalid Mask Register	12-92
RDY	Read Y Register	12-93
RESTORE	Restore caller's window	12-94
RETT	Return from Trap	12-95
SAVE	Save caller's window	12-97
SDIV	Signed Divide	12-98
SDIVcc	Signed Divide (modify icc)	12-99
SETHI	Set High 22 bits of r-Register	12-100
SLL	Shift Left Logical	12-101
SMUL	Signed Multiply	12-102
SMULcc	Signed Multiply (modify icc)	12-103
SRA	Shift Right Arithmetic	12-104
SRL	Shift Right Logical	12-105
ST	Store Word	12-106

STA	Store Word into Alternate space	12-107
STB	Store Byte	12-108
STBA	Store Byte into Alternate space	12-109
STC	Store Coprocessor Register	12-110
STCSR	Store Coprocessor State Register	12-111
STD	Store Doubleword	12-112
STDA	Store Doubleword into Alternate space	12-113
STDC	Store Doubleword Coprocessor	12-114
STDCQ	Store Doubleword Coprocessor Queue	12-115
STDF	Store Doubleword Floating-Point	12-116
STDFQ	Store Doubleword Floating-Point Queue	12-117
STF	Store Floating-Point Register	12-118
STFSR	Store Floating-Point State Register	12-119
STH	Store Halfword	12-120
STHA	Store Halfword into Alternate space	12-121
SUB	Subtract	12-122
SUBcc	Subtract and modify icc	12-123
SUBX	Subtract with Carry	12-124
SUBXcc	Subtract with Carry and modify icc	12-125
SWAP	Swap r-Register with memory	12-126
SWAPA	Swap r-Register with memory in Alternate space	12-127
TADDcc	Tagged Add and modify icc	12-128
TADDccTV	Tagged Add (modify icc) Trap on Overflow	12-129
Ticc	Trap on integer condition codes	12-130
TSUBcc	Tagged Subtract and modify icc	12-132
TSUBccTV	Tagged Subtract (modify icc) Trap on Overflow	12-133
UDIV	Unsigned Divide	12-134
UDIVcc	Unsigned Divide (modify icc)	12-135
UMUL	Unsigned Multiply	12-136
UMULcc	Unsigned Multiply (modify icc)	12-137
UNIMP	Unimplemented instruction	12-138
WRASR	Write Ancillary State Register	12-139
WRPSR	Write Processor State Register	12-140
WRTBR	Write Trap Base Register	12-141
WRWIM	Write Window Invalid Mask Register	12-142
WRY	Write Y Register	12-143
XNOR	Exclusive-Nor	12-144
XNORcc	Exclusive-Nor and modify icc	12-145
XOR	Exclusive-Or	12-146
XORcc	Exclusive-Or and modify icc	12-147

Appendix A: hyperSPARC Software Notes

A.1 Reset Considerations **A-1**
A.1.1 Miscellaneous Notes A-2

A.2 Compiler Optimization Notes **A-2**
A.2.1 Software pipelining: A-2
A.2.2 Loop Unrolling: A-3
A.2.3 Inter-procedure Optimizations A-3
A.2.4 Operating Systems Notes A-4

Glossary **G-1**

Index **I-1**

List of Tables

Chapter 2: SPARC Programming Environment

Table 2–1. Register Addressing 2-3
Table 2–2. Single-Precision Floating-Point Format 2-18
Table 2–3. Double-Precision Floating-Point Format 2-19
Table 2–4. Quad-Precision Floating-Point Format 2-20
Table 2–5. op field Coding 2-22
Table 2–6. op2 Field Coding 2-22
Table 2–7. Standard SPARC ASI Assignments 2-25
Table 2–8. Load and Store Instructions 2-26
Table 2–9. Arithmetic/Logical/Shift Instructions 2-27
Table 2–10. Control Transfer Instructions 2-29
Table 2–11. Control Transfer Instruction Characteristics 2-30
Table 2–12. Bicc and Ticc Condition Codes 2-31
Table 2–13. FBfcc Condition Codes 2-31
Table 2–14. CBccc Condition Codes 2-31
Table 2–15. Delayed Control Transfer Instruction Example 2-32
Table 2–16. Effect of Annul Bit Reset (a=0) 2-33
Table 2–17. Effect of Annul Bit Set (a=1) 2-33
Table 2–18. Effect of Annul Bit on Delay Instruction 2-34
Table 2–19. Delayed Control Transfer Couple Instruction Sequence 2-35
Table 2–20. Execution of Delayed Control Transfer Couples 2-35
Table 2–21. Read/Write Control Register Instructions 2-37
Table 2–22. Floating-Point-Operate and Coprocessor-Operate Instructions 2-37
Table 2–23. Miscellaneous Instructions 2-37
Table 2–24. Load and Store Instruction Opcodes 2-39
Table 2–25. Arithmetic/Logical/Shift Instruction Opcodes 2-41
Table 2–26. Control Transfer Instruction Opcodes 2-43
Table 2–27. Bicc and Ticc Condition Codes 2-43
Table 2–28. FBcc Condition Codes 2-44
Table 2–29. CBccc Condition Codes 2-44
Table 2–30. Read/Write Control Register Instruction Opcodes 2-44
Table 2–31. Floating-Point /Coprocessor Instruction Opcodes 2-45
Table 2–32. Miscellaneous Instruction Opcodes 2-46
Table 2–33. Instruction Opcode Numeric Listing 2-46

Chapter 3: RT620 hyperSPARC Central Processing Unit

Table 3–1. Instruction Grouping 3-9
Table 3–2. Instruction Combinations Eligible for Simultaneous Execution 3-11
Table 3–3. CACHE Line Privilege Match 3-26
Table 3–4. FLUSH instructions 3-29
Table 3–5. Integer Unit Cycle Per Instruction (CPI) 3-62

Table 3–6. Typical FPU Instruction Cycle Times 3-77
Table 3–7. RT620 Supported Exceptions 3-80
Table 3–8. Pipeline Stage Exception Recognition 3-81
Table 3–9. FIT Field of FSR 3-82
Table 3–10. FP Pipeline Stage Exception Recognition 3-83
Table 3–11. IU Actions Upon Reset, Trap, and Error Mode Events 3-101
Table 3–12. FPU Actions Upon Reset, Trap, And Error Mode Events 3-101

Chapter 4: RT625 hyperSPARC Cache Controller, Memory Management, and Tag Unit

Table 4–1. Short Translation Bits ST(1:0) 4-5
Table 4–2. Access-Level Protection Bits-ACC < 2:0 > 4-7
Table 4–3. Page Table Entry Type 4-12
Table 4–4. MMU Operation Modes 4-16
Table 4–5. TLB Entry Invalidation 4-18
Table 4–6. MBus Snooping Transactions 4-31
Table 4–7. Cache Flush operations 4-34
Table 4–8. Cacheable/Non-Cacheable Accesses 4-35
Table 4–9. State Table for MC (Memory Cacheable) Bit 4-35
Table 4–10. Cache Byte Write Enables 4-36
Table 4–11. TLB Entry Address Mapping 4-44
Table 4–12. Cache Tag Entry Address Mapping 4-45
Table 4–13. Power-On Reset States 4-46
Table 4–14. RT625 Register Address Mapping 4-47
Table 4–15. Standard SPARC ASI Assignments 4-47
Table 4–16. SPARC Fault Cases 4-48
Table 4–17. OW Bit States 4-49
Table 4–18. Fault Register Level Field 4-49
Table 4–19. Fault Register Access Type Field 4-49
Table 4–20. Fault Register Fault Type Field 4-50
Table 4–21. Fault Type (FT) for PTE[ET] = 2 4-50
Table 4–22. Fault Register Error Priorities 4-50
Table 4–23. Transaction Status Bit Encoding 4-60

Chapter 6: CY7C601/611 Integer Unit

Table 6–1. CY7C601/CY7C611 Instruction CPI 6-3
Table 6–2. CY7C601 External Signal Summary 6-6
Table 6–3. ASI Assignments 6-8
Table 6–4. SIZE Bit Encoding 6-11
Table 6–5. Internally Generated Opcodes 6-17
Table 6–6. Externally Generated Synchronous Exception Traps 6-42
Table 6–7. Trap Type and Priority Assignments 6-48
Table 6–8. Signal Differences Between CY7C601 and CY7C611 6-52
Table 6–9. CY7C611 Signal Summary 6-53

Chapter 7: CY7C602 Floating-Point Unit

Table 7–1. FPop execution 7-6
Table 7–2. Load instruction execution 7-6

Table 7–3. Store instruction execution 7-6
Table 7–4. FHOLD Resource/Operand Dependency Cases 7-13
Table 7–5. Floating-Point Status Register Summary 7-17
Table 7–6. Floating-Point Load and Store Instruction Cycle Count 7-19
Table 7–7. Floating-Point Operate (FPops) Instruction Cycle Count 7-19
Table 7–8. FCC(1:0) Condition Codes 7-23

Chapter 8: CY7C604/CY7C605 Cache Controller and Memory Management Unit

Table 8-1. Short Translation Bits - ST(1:0) 8-6
Table 8-2. Access-Level Protection Bits—ACC(2:0) 8-6
Table 8-3. Page Table Entry Type 8-11
Table 8-4. MMU Operation Modes 8-15
Table 8-5. TLB Entry Flushing 8-17
Table 8-6. MBus Snooping Transactions 8-35
Table 8-7. Cache Flush Operations 8-38
Table 8-8. Cacheable / Non-Cacheable Accesses 8-39
Table 8-9. State Table for MC (Memory Cacheable) Bit 8-40
Table 8-10. Byte Write Enables 8-41
Table 8-11. TLB Entry Address Mapping 8-52
Table 8-12. Cache Tag Entry Address Mapping 8-53
Table 8-13. CY7C604/605 Power-On Reset States 8-54
Table 8-14. CY7C604/605 Register Address Mapping 8-55
Table 8-15. Standard ASI Assignments 8-56
Table 8-16. OW Bit States 8-57
Table 8-17. Fault Register Level Field 8-58
Table 8-18. Fault Register Access Type Field 8-58
Table 8-19. Fault Register Fault Type Field 8-58
Table 8-20. Fault Type (FT) for PTE [ET] = 2 8-59
Table 8-21. Fault Register Error Priorities 8-59

Chapter 11: MBus Operation

Table 11–1. MBus Signal Summary 11-3
Table 11–2. Transaction Status Bit Encoding 11-5
Table 11–3. Multiplexed Signal Summary 11-7
Table 11–4. TYPE Encodings 11-8
Table 11–5. SIZE Encodings 11-9
Table 11–6. Bus Status Encoding 11-12

Chapter 12: SPARC Instruction Set

Table 12–1. Instruction Description Notations 12-4
Table 12–2. Instruction Set Summary 12-6

List of Figures

Chapter 1: Introduction

Figure 1–1. hyperSPARC CPU Block Diagram 1–4
Figure 1–2. RT620 Instruction Pipeline Example 1–7
Figure 1–3. RT620 Pipeline Stages 1–8
Figure 1–4. Architectural Partitioning—Uniprocessor System 1–10
Figure 1–5. Architectural Partitioning—Multiprocessors 1–10

Chapter 2: SPARC Programming Environment

Figure 2–1. SPARC Register Models 2-1
Figure 2–2. Overlapping Windows 2-3
Figure 2–3. Registers as Seen by a Procedure 2-5
Figure 2–4. FPU Register File 2-7
Figure 2–5. Processor State Register 2-8
Figure 2–6. Window Invalid Mask 2-10
Figure 2–7. Trap Base Register 2-11
Figure 2–8. Y Register 2-11
Figure 2–9. ICCR Register 2-12
Figure 2–10. Floating-Point Status Register (FSR) 2-12
Figure 2–11. Processor Data Types 2-15
Figure 2–12. Byte Operand Load and Store 2-16
Figure 2–13. Data Organization in Memory 2-16
Figure 2–14. Single-Precision Floating-Point Format 2-18
Figure 2–15. Double-Precision Floating-Point Format 2-19
Figure 2–16. Quad-Precision Data Organization in Registers 2-20
Figure 2–17. Quad-Precision Data Organization in Memory 2-20
Figure 2–18. Instruction Format Summary 2-21
Figure 2–19. Address Generation 2-24
Figure 2–20. Tagged Data Example 2-28
Figure 2–21. Ticc Trap Address Generation 2-31
Figure 2–22. Delayed Control Transfer 2-34
Figure 2–23. Delayed Control Transfer Couples 2-36

Chapter 3: RT620 hyperSPARC Central Processing Unit

Figure 3–1. RT620 hyperSPARC CPU Block Diagram 3-2
Figure 3–2. Integer Unit Blocks 3-5
Figure 3–3. hyperSPARC Register Model 3-6
Figure 3–4. RT620 State Transition Diagram 3-7
Figure 3–5. Floating-Point Unit Block Diagram 3-16
Figure 3–6. Floating-Point Unit State Transition Diagram 3-16
Figure 3–7. Floating-Point Queue (FPQ) 3-18
Figure 3–8. Forwarding between two FP Instructions 3-20
Figure 3–9. Forwarding from Load to FP Instruction 3-21

Figure 3–10. Forwarding from FP Instruction to Store . . . 3-22
Figure 3–11. ICACHE Line Organization . . . 3-24
Figure 3–12. Virtual Address Format . . . 3-24
Figure 3–13. ICACHE Organization . . . 3-25
Figure 3–14. ICCR Register . . . 3-26
Figure 3–15. RT620 Signals . . . 3-30
Figure 3–16. hyperSPARC CPU with 256-Kbyte Cache . . . 3-35
Figure 3–17. Instruction Access with Instruction Cache Hit . . . 3-37
Figure 3–18. Read Accesses with External Cache Hit . . . 3-38
Figure 3–19. Write Accesses with External Cache Hit . . . 3-39
Figure 3–20. Atomic Read-Write Accesses with External Cache Hit . . . 3-40
Figure 3–21. Write Access Followed by Read Access with External Cache Hit . . . 3-41
Figure 3–22. Write Access Followed by Write Access with External Cache Hit . . . 3-42
Figure 3–23. Read Access with External Cache Miss . . . 3-43
Figure 3–24. Write Access with External Cache Miss . . . 3-45
Figure 3–25. Atomic Read-Write Access with External Cache Miss . . . 3-47
Figure 3–26. Data Read Access with Memory Exception . . . 3-49
Figure 3–27. Instruction Read Access with Memory Exception . . . 3-51
Figure 3–28. Reset Timing . . . 3-53
Figure 3–29. Interrupt Timing . . . 3-55
Figure 3–30. Error Timing . . . 3-57
Figure 3–31. Timing for Instruction Fetch with ICACHE Hit . . . 3-59
Figure 3–32. Timing for Instruction Fetch with ICACHE Miss and e-Cache Hit . . . 3-59
Figure 3–33. Timing for Instruction Fetch with ICACHE Miss and e-Cache Miss . . . 3-60
Figure 3–34. Timing for Instruction Fetch with ICACHE Disabled and e-Cache Hit . . . 3-60
Figure 3–35. Timing for Instruction Fetch with ICACHE Disabled and e-Cache Miss . . 3-61
Figure 3–36. Typical Integer Instruction Pipeline . . . 3-61
Figure 3–37. Typical ALU Instruction Pipeline . . . 3-63
Figure 3–38. Typical INTEGER MULTIPLY Instruction Pipeline . . . 3-63
Figure 3–39. Typical INTEGER DIVIDE Instruction Pipeline . . . 3-64
Figure 3–40. Typical LOAD Instruction Pipelines . . . 3-65
Figure 3–41. Typical STORE Instruction Pipeline . . . 3-65
Figure 3–42. Typical Atomic Load-Store Instruction Pipeline . . . 3-67
Figure 3–43. Typical Branch Instruction Pipeline . . . 3-67
Figure 3–44. Typical CALL Instruction Pipeline . . . 3-68
Figure 3–45. Typical JMPL Instruction Pipeline . . . 3-69
Figure 3–46. Typical RETT Instruction Pipeline . . . 3-69
Figure 3–47. Typical FLUSH Instruction Pipeline . . . 3-70
Figure 3–48. Typical Save/Restore Instruction Pipeline . . . 3-71
Figure 3–49. Typical Read/Write Special Register Instruction Pipeline . . . 3-71
Figure 3–50. Fast Constant Instruction Pipeline . . . 3-72
Figure 3–51. Fast Index Instruction Pipeline . . . 3-73
Figure 3–52. Sequential Branch Instruction Pipeline . . . 3-74
Figure 3–53. Fast Branch Instruction Pipeline . . . 3-75
Figure 3–54. Typical FPU Instruction Pipeline . . . 3-76
Figure 3–55. Typical Multiple Cycle FPU Instruction Pipeline . . . 3-77
Figure 3–56. Exception Pipeline . . . 3-78

Figure 3–57. Floating-Point Exception Pipeline 3-83
Figure 3–58. Floating-Point Exception during Forwarding to Store 3-84

Chapter 4: RT625 hyperSPARC Cache Controller, Memory Management, and Tag Unit

Figure 4–1. 128-Kbyte Cache Memory Sub-System 4-2
Figure 4–2. 256-Kbyte Cache Memory Sub-System 4-3
Figure 4–3. Translation Lookaside Buffer (TLB) 4-5
Figure 4–4. Address Comparison 4-6
Figure 4–5. TLB Replacement and Locking 4-8
Figure 4–6. Four-Level Table Walk (4-Kbyte Addressing) 4-10
Figure 4–7. Three-Level Table Walk (256-Kbyte Addressing) 4-11
Figure 4–8. Page Table Pointer 4-11
Figure 4–9. Page Table Entry Format 4-12
Figure 4–10. Page Table Pointer Cache 4-14
Figure 4–11. Table Walk Algorithm 4-15
Figure 4–12. MMU Invalidate Address Format 4-17
Figure 4–13. RT625 Cache Tag Entry 4-21
Figure 4–14. RT625 Cache TAG (CTAG) Comparison (128-Kbyte Cache) 4-21
Figure 4–15. RT625 Cache TAG (CTAG) Comparison (256-Kbyte Cache) 4-22
Figure 4–16. Copy-Back Invalid 4-24
Figure 4–17. Copy-Back Exclusive Clean 4-25
Figure 4–18. Copy-Back Shared Clean 4-26
Figure 4–19. Copy-Back Exclusive Modified 4-28
Figure 4–20. Copy-Back Shared Modified 4-29
Figure 4–21. Write-Through Invalid 4-30
Figure 4–22. Write-Through Valid 4-31
Figure 4–23. RT625 Write Buffers 4-33
Figure 4–24. RT625 Write Buffer (copy-back mode) 4-33
Figure 4–25. RT625 Read Buffer (copy-back mode) 4-34
Figure 4–26. CBWE Byte Assignments 4-36
Figure 4–27. RT625 System Control Register (SCR) 4-37
Figure 4–28. RT625 Context Table Pointer Register 4-39
Figure 4–29. RT625 Context Register 4-39
Figure 4–30. RT625 Reset Register 4-39
Figure 4–31. RT625 Root Pointer Register 4-40
Figure 4–32. RT625 Instruction Access PTP Registers 4-40
Figure 4–33. RT625 Data Access PTP Registers 4-40
Figure 4–34. RT625 Index Tag Registers 4-40
Figure 4–35. RT625 Replacement Control Register 4-41
Figure 4–36. RT625 Synchronous Fault Status Register 4-41
Figure 4–37. Synchronous Fault Address Register 4-42
Figure 4–38. RT625 Asynchronous Fault Status Register 4-42
Figure 4–39. RT625 Asynchronous Fault Address Register 4-42
Figure 4–40. TLB Entry Format 4-44
Figure 4–41. RT625 Cache Tag Entry Format 4-45
Figure 4–42. RT625 Pinout 4-57

Chapter 5: RT627 hyperSPARC Cache Data Unit

Figure 5–1. 256-Kbyte Cache Subsystem 5-1
Figure 5–2. RT627 Block Diagram 5-3
Figure 5–3. Read Access Followed by Read Access 5-5
Figure 5–4. Write Access Followed by Write Access 5-6
Figure 5–5. Read Followed by Write Access Followed by Read Access 5-7
Figure 5–6. Non-Cacheable Read Access (illustrates the use of CROE) 5-8

Chapter 6: CY7C601/611 Integer Unit

Figure 6–1. Integer Unit Block Diagram 6-1
Figure 6–2. SPARC Register Model 6-2
Figure 6–3. CY7C601/CY7C611 External Signals 6-5
Figure 6–4. Processor Instruction Pipeline 6-16
Figure 6–5. Pipeline with All Single-Cycle Instructions 6-16
Figure 6–6. Pipeline with One Double-Cycle Instruction (Load) 6-18
Figure 6–7. Pipeline with One Triple-Cycle Instruction (Store) 6-19
Figure 6–8. Pipeline with Hardware Interlock (Load) 6-20
Figure 6–9. Pipeline During Branch Instruction 6-21
Figure 6–10. Branch with Annulled Delay Instruction 6-21
Figure 6–11. Pipeline Frozen During Bus Arbitration 6-22
Figure 6–12. Pipeline Operation for Taken Trap (Internal) 6-23
Figure 6–13. Data Bus Contents During Data Transfers 6-24
Figure 6–14. Instruction Fetch 6-25
Figure 6–15. Load Single Integer Timing 6-25
Figure 6–16. Load Single with Interlock Timing 6-26
Figure 6–17. Load Double Integer Timing 6-26
Figure 6–18. Store Single Integer Timing 6-27
Figure 6–19. Store Double Integer Timing 6-28
Figure 6–20. Atomic Load-Store Timing 6-29
Figure 6–21. Floating-Point Operation Timing 6-30
Figure 6–22. Bus Arbitration Timing 6-31
Figure 6–23. Load with Cache Miss Timing 6-32
Figure 6–24. Store with Cache Miss Timing 6-33
Figure 6–25. Load with Memory Exception Timing 6-35
Figure 6–26. Store with Memory Exception Timing 6-37
Figure 6–27. Floating-Point Exception Handshake Timing 6-39
Figure 6–28. Asynchronous Interrupt Timing 6-39
Figure 6–29. Power-On Reset Timing 6-40
Figure 6–30. Error/Reset Timing 6-41
Figure 6–31. Best-Case Interrupt Response Timing 6-44
Figure 6–32. Worst-Case Interrupt Response Timing 6-45
Figure 6–33. Coprocessor Register Model 6-51

Chapter 7: CY7C602 Floating-Point Unit

Figure 7–1. CY7C602 Functional Block Diagram 7-1
Figure 7–2. CY7C602 Block Diagram 7-3
Figure 7–3. CY7C601 – CY7C602 Hardware Interface 7-4

Figure 7–4. CY7C602 Address/Instruction Pipe 7-5
Figure 7–5. Instruction Fetch (Cache Hit) 7-7
Figure 7–6. Instruction Fetch (Cache Miss on A2) 7-8
Figure 7–7. Floating-Point Instruction Dispatching 7-9
Figure 7–8. Floating-Point Compare (FCMP) Execution 7-9
Figure 7–9. Floating-Point Instruction Pipeline During A Trap 7-11
Figure 7–10. Effect of FLUSH on LDF Instruction 7-11
Figure 7–11. Effect of FLUSH on STF Instruction 7-11
Figure 7–12. Effect of FLUSH on FPop Instruction 7-12
Figure 7–13. Effect of FLUSH on FCMP Instruction 7-12
Figure 7–14. f-Register Organization 7-14
Figure 7–15. f-Register Addressing 7-15
Figure 7–16. Floating-Point Status Register 7-16
Figure 7–17. FPU Operation Modes 7-20
Figure 7–18. Floating-Point Exception Handshake 7-21

Chapter 8: CY7C604/CY7C605 Cache Controller and Memory Management Unit

Figure 8–1. Virtual 64-Kbyte Cache 8-2
Figure 8–2. Translation Lookaside Buffer (TLB) 8-4
Figure 8–3. Address Comparison 8-5
Figure 8–4. TLB Replacement and Locking 8-8
Figure 8–5. Four-Level Table Walk (4-Kbyte Addressing) 8-8
Figure 8–6. Three-Level Table Walk (256-Kbyte Addressing) 8-10
Figure 8–7. Page Table Pointer 8-11
Figure 8–8. Page Table Entry Format 8-12
Figure 8–9. Page Table Pointer Cache 8-13
Figure 8–10. Table Walk Algorithm 8-14
Figure 8–11. MMU Flush Address Format 8-17
Figure 8–12. CY7C604 Cache Tag 8-20
Figure 8–13. CY7C604 Write-Through with No Write Allocate 8-21
Figure 8–14. CY7C604 Copy-Back with Write Allocate 8-21
Figure 8–15. CY7C605 Processor Virtual Cache Tag (PVTAG) Comparison 8-24
Figure 8–16. CY7C605 Cache Tag Entries 8-25
Figure 8–17. CY7C605 MBus Physical Cache Tag (MPTAG) Comparison 8-25
Figure 8–18. Copy-back Invalid 8-27
Figure 8–19. Copy-back Exclusive Clean 8-28
Figure 8–20. Copy-back Shared Clean 8-29
Figure 8–21. Copy-back Exclusive Modified 8-30
Figure 8–22. Copy-back Shared Modified 8-32
Figure 8–23. Write-through Invalid 8-33
Figure 8–24. Write-through Valid 8-34
Figure 8–25. Write Buffers (Write-through Mode or Non-cacheable Write) 8-37
Figure 8–26. Write Buffer (Copy-Back Mode) 8-37
Figure 8–27. Read Buffer (Copy-Back Mode) 8-37
Figure 8–28. CBWE Byte Assignments 8-40
Figure 8–29. CY7C604 System Control Register (SCR) 8-42
Figure 8–30. CY7C605 System Control Register (SCR) 8-43

Figure 8–31. CY7C604/605 Context Table Pointer Register 8-44
Figure 8–32. CY7C604/605 Context Register 8-44
Figure 8–33. CY7C604/605 Reset Register 8-44
Figure 8–34. CY7C604/605 Root Pointer Register 8-45
Figure 8–35. CY7C604/605 Instruction Access PTP Register 8-45
Figure 8–36. CY7C604/605 Data Access PTP Register 8-45
Figure 8–37. CY7C604/605 Index Tag Register 8-45
Figure 8–38. CY7C604/605 TLB Replacement Control Register 8-46
Figure 8–39. CY7C604/605 Synchronous Fault Status Register 8-47
Figure 8–40. CY7C604/605 Synchronous Fault Address Register 8-47
Figure 8–41. CY7C604/605 Asynchronous Fault Status Register 8-47
Figure 8–42. CY7C604/605 Asynchronous Fault Address Register 8-48
Figure 8–43. Dual–CY7C604 Multichip Configuration 8-49
Figure 8–44. Dual–CY7C605 Multichip Configuration 8-50
Figure 8–45. Examples of Multichip Addressing 8-51
Figure 8–46. TLB Entry Format 8-52
Figure 8–47. CY7C604 Cache Tag Entry Format 8-53
Figure 8–48. CY7C605 Cache Tag Entry Format 8-53
Figure 8–49. CY7C604 and CY7C605 I/O Signals 8-64

Chapter 9: CY7C157 Cache Storage Unit

Figure 9–1. CY7C157 Block Diagram 9-1

Chapter 11: MBus Operation

Figure 11–1. Level 2 MBus Cache State Diagram 11-2
Figure 11–2. Byte Organization 11-4
Figure 11–3. MBus Address Cycle 11-8
Figure 11–4. MBus Data Ordering 11-11
Figure 11–5. MBus Burst Transaction Example 11-11
Figure 11–6. MBus Read Transaction 11-13
Figure 11–7. MBus Write Transaction 11-13
Figure 11–8. MBus Coherent Read Transaction - MIH not asserted 11-15
Figure 11–9. MBus Coherent Read Transaction - MIH asserted 11-15
Figure 11–10. MBus Coherent Invalidate Transaction 11-16
Figure 11–11. MBus Coherent Read and Invalidate Transaction - MIH not asserted 11-17
Figure 11–12. MBus Coherent Read and Invalidate Transaction - MIH asserted 11-17
Figure 11–13. MBus Coherent Write and Invalidate Transaction 11-18
Figure 11–14. MBus Configuration Address Map 11-18
Figure 11–15. MBus Write-through Coherent Write and Invalidate Transaction ($\overline{MRDY}$ in A+2) 11-19
Figure 11–16. MBus Configuration Address Map 11-23
Figure 11–17. MBus Port Register Format 11-23
Figure 11–18. Initial MBus Arbitration 11-25
Figure 11–19. MBus Mastership Transfer 11-25
Figure 11–20. MBus Arbitration with Multiple Requests 11-26
Figure 11–21. MBus Single-Cycle – Read Transaction 11-27

Figure 11–22. MBus Single-Cycle – Write Transaction 11-27
Figure 11–23. MBus Burst-Cycle – Read Transaction 11-28
Figure 11–24. MBus Burst-Cycle – Read Transaction (Slow memory) 11-28
Figure 11–25. MBus Burst-Cycle – Write Transaction 11-29
Figure 11–26. MBus Burst-Cycle – Write Transaction (Slow memory) 11-30
Figure 11–27. MBus Locked Transaction 11-31
Figure 11–28. MBus Relinquish and Retry 11-32
Figure 11–29. MBus Retry 11-32
Figure 11–30. MBus Error (Bus Error) 11-33
Figure 11–31. MBus Error (Timeout) 11-33
Figure 11–32. MBus Error (Uncorrectable) 11-34
Figure 11–33. MBus Coherent Read – Shared Data 11-35
Figure 11–34. MBus Coherent Read – Owned Data (CY7C605) (Slow Memory) 11-37
Figure 11–35. MBus Coherent Read – Owned Data (CY7C605) (Fast Memory) 11-39
Figure 11–36. MBus Coherent Read – Owned Data (RT625) (Slow Memory) 11-41
Figure 11–37. MBus Coherent Read – Owned Data (RT625) (Fast Memory) 11-43
Figure 11–38. MBus Coherent Write and Invalidate 11-45
Figure 11–39. MBus Coherent Write and Invalidate (RT625) (Block Copy/Fill) 11-47
Figure 11–40. MBus Coherent Invalidate 11-49
Figure 11–41. MBus Coherent Read and Invalidate – Shared Data 11-50
Figure 11–42. MBus Coherent Read and Invalidate – Owned Data (CY7C605) (Slow Memory) 11-52
Figure 11–43. MBus Coherent Read and Invalidate – Owned Data (CY7C605) (Fast Memory) 11-54
Figure 11–44. MBus Coherent Read and Invalidate – Owned Data (RT625) (Slow Memory) 11-56
Figure 11–45. MBus Coherent Read and Invalidate—Owned Data (RT625) (Fast Memory) 11-58

Chapter 12: SPARC Instruction Set

Figure 12–1. SPARC Instruction Mnemonic Summary 12-3
Figure 12–2. Instruction Description 12-4

ROSS TECHNOLOGY, INC.

Chapter 1

Introduction

SPARC, an acronym for Scalable Processor Architecture, is an open RISC architecture with multiple semiconductor implementations from a number of vendors. SPARC is an architecturally driven standard, with binary compatibility of software between processor versions ensured by enforcing compliance to the architecture standard. The open architecture approach offered by SPARC allows all its participants to make creative contributions in developing their versions of SPARC processor. This results in a vastly greater number of technical contributions than would be possible for a closed architecture held and defined by only one group. This architectural freedom has allowed the SPARC architecture to expand into process technologies such as CMOS gate arrays, full-custom CMOS, BiCMOS, and GaAs faster than any other RISC architecture. This same freedom allows SPARC vendors to make micro-architectural enhancements to their SPARC implementations while maintaining absolute binary compatibility. The final result of this open architecture approach is that it provides the customer with a wider range of price/performance and technology options that cannot be matched by less innovative and restricted licensing policies. In addition, the various SPARC vendors also participate in standard second-sourcing agreements.

The inclusion of the word "scalable" in the acronym for SPARC emphasizes its importance in the philosophy of the architecture. "Enforced compatibility" has been embraced to ensure migration of the architecture as semiconductor technology improves. Scalability allows SPARC to be re-implemented without complication as semiconductor process technology evolves. This allows SPARC to continually be offered in higher clock speeds and technologies than other RISC architectures, providing rapid performance improvements as process technology continues to be refined. Other RISC processors have complicated their micro-architectures with features that create an unnecessary burden for the hardware designer. These features provide only a minimal performance improvement, but greatly complicate hardware design and cost. ROSS SPARC microprocessors do not require multiple-phase clocks, de-multiplexing of the processor's address or data buses or many of the other problems that affect hardware complexity and cost. This provides ROSS SPARC based designs with the advantages of excellent performance, low design costs, a high degree of manufacturability, and increased reliability due to simplicity of design.

ROSS Technology provides two families of SPARC processors: the RT600 hyperSPARC family, and the original CY7C600 family. The hyperSPARC is the second-generation SPARC RISC processor family, consisting of the RT620 Central Processing Unit (CPU), the RT625 Cache Controller, Memory Management, and Tag Unit (CMTU) and the RT627 Cache Data Unit (CDU). The CY7C600 is the first-generation SPARC processor family, which consists of the CY7C601 Integer Unit (IU), the CY7C611 Integer Unit for embedded control, the CY7C602 Floating-Point Unit (FPU), the CY7C604 Cache Controller and Memory Management Unit (CMU), the CY7C605 Cache Controller and Memory Management Unit for Multiprocessing (CMU-MP), and the CY7C157 Cache Storage Unit (CSU). The RT600 hyperSPARC and the CY7C600 processor families provide a range of cost-performance choices, allowing the designer to match the level of processor technology to the requirements of the system.

The hyperSPARC is a high-performance, superscalar processor representing a milestone in the development of SPARC. The RT620 hyperSPARC CPU is a superscalar RISC processor featuring an integrated integer unit/floating-point unit with an 8-Kbyte 2-way set-associative instruction cache. The superscalar RT620 supports dual-instruction launch for a majority of instruction combinations, and features several significant

pipeline enhancements in both the integer unit and floating-point unit data paths. The demand on memory required by hyperSPARC is supported by the 64-bit Intra-Module Bus (IMB), which provides a high-bandwidth interface to the hyperSPARC cache system. The RT625 CMTU and RT627 Cache Data Units complete the hyperSPARC CPU, providing 128 or 256 Kbytes of cache with complete hardware support for MBus-based multiprocessor systems.

The CY7C600 family is ROSS Technology's first-generation SPARC processor family. The CY7C600 family includes two Integer Units (the CY7C601 and the CY7C611), an FPU (the CY7C602), two Cache Controller and Memory Management Units (the CY7C604 and the CY7C605), and a high-speed synchronous SRAM Cache Storage Unit (CY7C157). The CY7C601 is the primary processing engine for the CY7C600 CPU, and is designed to work as part of a tightly-coupled SPARC CPU. The CY7C602 is an efficient FPU designed to operate in tandem with the CY7C601. The cache control and memory management functions of the CY7C600 CPU may be provided by the CY7C604 or the CY7C605, depending upon system requirements. The CY7C604 is an efficient uniprocessor cache controller and memory management unit (MMU). The CY7C605 is a SPARC cache controller and memory management unit that provides full hardware support for Level-2 MBus multiprocessing systems. Both cache controllers are designed to use the CY7C157, a synchronous, self-timed cache RAM custom designed for the CY7C600 family. The CY7C611 is a SPARC Integer Unit derived from the CY7C601, and is designed for high-performance, cost-sensitive embedded control applications.

1.1 SPARC Architecture Features

1.1.1 Load/Store Architecture

SPARC uses a register Load/Store architecture, that performs all operations using either immediate operands or operands stored in internal registers or status registers. Data is fetched from memory to an internal register by the use of a load instruction, and is transferred from the register file to memory by the use of a store instruction. Register Load/Store architectures are a superior method of operand access in terms of minimizing memory bus traffic and maximizing data locality to the CPU. In addition, the Load/Store paradigm allows the definition of an instruction set with a high degree of orthogonality. This is accomplished by separating data access operations from logical data operations. This instruction set orthogonality provides a logical separation of tasks, resulting in greater simplicity in the implementation of the processor. The implied simplicity of the Load/Store architecture results in efficient pipelining and thereby lends itself to high-performance superscalar designs.

1.1.2 Register Windows

ROSS SPARC microprocessors contain a large, 32-bit-wide register file that is divided into multiple windows that are controlled by internal hardware. Each window contains 24 working registers and has access to eight global registers. Combined with SPARC's register-to-register architecture, this file operates effectively as a compiler-directed, copy-back data cache, considerably reducing data bus traffic. Load instructions enter data into this cache, and store instructions "copy back" information when it needs to be replaced into main memory.

The register file is managed as a circular stack, with the first and last windows overlapping each other. Each window overlaps the previous window and succeeding window by eight registers, making the window mechanism ideal for passing parameters in procedure calls. Results left in the overlapping registers by a calling routine automatically become available operands for the called routine as the window moves, and vice versa. This parameter passing technique eliminates the need for the loads and stores to memory required by machines using a stack during procedure calls.

1.1.3 Instruction Set

SPARC instructions fall into five basic categories: Load/Store, arithmetic/logical/shift, control transfer, read/write control register, and floating-point/coprocessor-operate.

1.1.3.1 Load and Store Instructions

Load and store instructions are the only way to access memory or external registers. Addresses are calculated using the contents of two registers or one register and a constant. The destination may be an integer unit, floating-point unit, or coprocessor register, that either supplies or receives the data. In order to greatly speed up memory accesses, halfword, word, and doubleword data must be aligned on their corresponding boundaries. If they are not, a trap is generated when an access is attempted.

Whenever an address is sent to the address bus, the processor also generates eight bits of address space identifier (ASI). The ASI pins identify to the external system which of the 256 possible address spaces is to be accessed. For most load or store operations, one of four standard ASI values is asserted. These four ASI values indicate whether the processor is in user or supervisor mode, and whether the access is an instruction or data reference.

The address space identifier is intended for use by the system operating software. Consequently, the instructions that specify a particular ASI value (Load/Store Alternate) are privileged and can only be executed in the supervisor mode. Many of the ASI bit patterns are assigned for accessing various features of the cache controller and memory management unit. A large block of address spaces is reserved for the designer to implement as desired.

1.1.4 Arithmetic/Logical/Shift Instructions

These instructions compute a result using two source operands and place the result in a destination register. In addition to standard arithmetic operations, the SPARC instruction set includes tagged arithmetic operations. Tagged arithmetic instructions assume that the least-significant two bits of the operands are tags, and set a condition code bit if they are not zero. Tagged instructions are used with artificial intelligence languages such as LISP to indicate the data type of the operands. The use of tagged arithmetic instructions allows languages such as LISP and Prolog to run significantly faster than on RISC machines without this type of instruction.

1.1.5 Control Transfer Instructions

Control transfer instructions include jumps, calls, branches, and traps. Transfer of control to the new address is usually delayed until after execution of the next instruction immediately following the JUMP, CALL or Branch, etc., so that the transfer does not create a hole or bubble in the instruction pipeline. It is the compiler's or assembly language programmer's job to attempt to place a useful instruction in this delay slot.

1.1.6 Read/Write Control Register Instructions

These include instructions to read and write the contents of various SPARC control registers. The source (read) or destination (write) is implied by the instruction name, or by an alternate state register number provided as an instruction operand.

1.1.7 Floating-Point-Operate and Coprocessor-Operate Instructions

This category includes floating-point calculations, floating-point register operations, and instructions involving computations or other operations in the second coprocessor.

Floating-point-operate (FPop) instructions execute concurrently with integer instructions and possibly with other floating-point instructions. Concurrent execution is also possible with the coprocessor-operate instructions if they are implemented.

The CY7C601 supports a coprocessor interface, thus allowing the user to provide a customized processing engine to work in tandem with the CY7C601 Integer Unit and the CY7C602 Floating-Point Unit. This interface is not supported on the CY7C611 Integer Unit or the hyperSPARC RT620.

Coprocessor-operate (CPop) instructions are defined by the coprocessor itself. In the SPARC instruction set, they are specified by the CPop instruction. The SPARC architecture will accommodate 1024 coprocessor-operate instructions.

Floating-point and coprocessor loads and stores are not classified as operate instructions; they belong to the "load and store" category previously discussed.

1.2 hyperSPARC Overview

The hyperSPARC is designed as a tightly-coupled chipset to be utilized in a SPARC MBus module. Each hyperSPARC CPU supports either 128 or 256 Kbytes of second-level cache, and each module may contain one or two CPUs. The chipset is comprised of the RT620 Central Processing Unit (CPU), the RT625 Cache Controller, Memory Management, and Tag Unit (CMTU), and two or four RT627 Cache Data Units (CDUs) for 128 Kbytes or 256 Kbytes of second-level cache, respectively. The chipset can be configured for uniprocessing (Level 1 MBus) or multiprocessing (Level 2 MBus). *Figure 1–1* represents a block diagram of the hyperSPARC chipset.

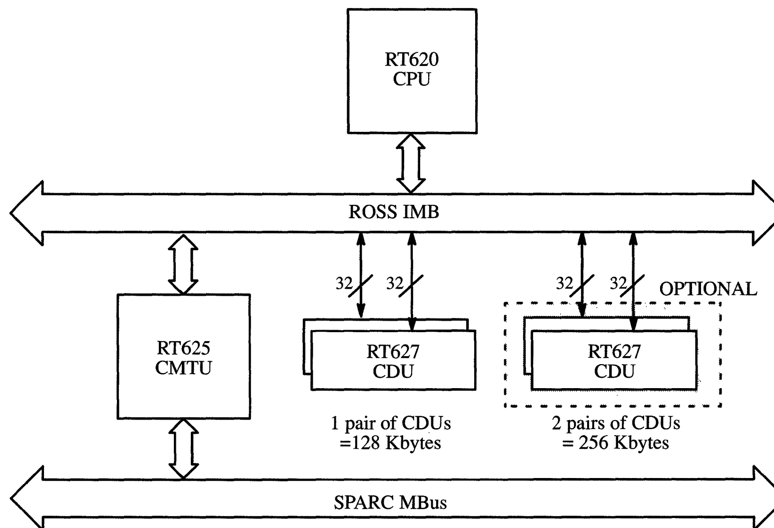


Figure 1–1. hyperSPARC CPU Block Diagram

The RT620 is the primary processing unit in hyperSPARC. This chip is comprised of an integer unit, a floating-point unit, and an 8-Kbyte, 2-way set-associative instruction cache. The integer unit contains the ALU and separate Load/Store data path, constituting two of the chip's four execution units. There is also the floating-point unit and a Branch/Call unit (for processing control transfer instructions). Two instructions are fetched every clock cycle. In general, as long as these two instructions require different execution units and

have no data dependencies, they can be launched simultaneously. Two floating-point instructions may also be simultaneously dispatched, due to the inclusion of a floating-point instruction pre-queue. The RT620 contains two register files: 136 integer registers configured as eight register windows, and 32 separate floating-point registers in the floating-point unit.

The hyperSPARC's second-level cache is built around the RT625 CMTU, a combined cache controller and memory management unit that supports shared-bus multiprocessing. The cache controller portion supports 128-Kbytes or 256-Kbytes of cache, made up of two or four RT627 CDUs. The cache is direct-mapped with 4-Kbytes of cache tags. The cache is physically tagged and virtually indexed so that the RT625's cache coherency logic can quickly determine snoop hits and misses without stalling the RT620's access to the cache. Both copy-back and write-through caching modes are supported.

The SPARC reference MMU of the RT625 provides a 64-entry, fully set-associative TLB that supports 4096 contexts. The RT625 contains a read buffer (32 bytes deep) and a write buffer (64 bytes deep) for buffering the 32-byte cache lines in and out of the second-level cache, and synchronization logic for interfacing the virtual IMB to the SPARC MBus.

The RT627 is a custom-designed 16-Kbyte X 32-bit SRAM. It is organized as four arrays of 16-Kbyte SRAM with byte write logic, registered inputs, and data-in and data-out latches. The RT627s provide a zero-wait-state cache to the CPU with no pipeline penalty (i.e., stalls) for loads and stores that hit the cache. The RT627 requires no glue logic for interfacing to the RT620 (CPU) and the RT625 (CMTU).

1.2.1 hyperSPARC Design Features

The microarchitecture of hyperSPARC boasts classic RISC and superscalar features for improving instruction processing throughput. In addition, hyperSPARC also employs architectural enhancements that differentiate it from other next-generation microprocessor designs. The following sections highlight of some of hyperSPARC's most important attributes.

1.2.1.1 High Frequency of Operation

Fundamentally, hyperSPARC is built for speed. In order to facilitate high clock frequencies, particular attention is paid to the 6-stage integer and floating-point pipelines, keeping them simple and well-balanced. Each stage of the pipelines is carefully partitioned in order that the number of gates per stage is similar, thus more easily lending itself to process shrinks for scaling to higher clock rates.

The hyperSPARC allows processor clock rates to be increased independently of the external bus (MBus). The hyperSPARC chipset was partitioned to allow synchronous or asynchronous operation through synchronization logic contained in the RT625. This decoupling of the CPU bus from the external bus allows scaling of hyperSPARC's clock frequency independent of the memory and I/O subsystems. This provides longer product life cycles since upgrades to higher performance hyperSPARC modules require no hardware changes to the underlying system design.

1.2.1.2 Instruction Scheduling

Instruction scheduling and dispatching is a critical portion of any superscalar design. Optimal instruction scheduling involves both minimizing pipeline stalls (costly enough for any RISC machine, but even more costly when multiple instructions are being held) and minimizing conditions that prevent simultaneous instruction launching.

All superscalar microprocessors are not the same. The ability to fetch and launch multiple instructions is only as good as the number of times this feature is actually taken advantage of. Compilers can help reduce

the occurrences of instructions that cannot be launched together by scheduling instructions accordingly. However, the software can only optimize for the hardware; the microprocessor design must be intelligently partitioned to provide opportunities for software enhancements.

The hyperSPARC is partitioned into four execution units in order to facilitate parallel processing of major instruction types. These execution units are the Load/Store unit, Branch/Call unit, integer unit, and floating-point unit. The floating-point unit is actually comprised of an instruction queue and two parallel pipelines, an adder and a multiplier.

The hyperSPARC fetches two instructions every clock cycle and evaluates them for simultaneous launch. hyperSPARC's primary scheduler is invoked for this evaluation and determines the hardware resources required for processing the instructions, as well as any data dependencies. This critical juncture in the instruction processing path exposes the strengths and weaknesses of microarchitectures.

Poorly architected designs require more frequent splits of instruction groupings. Internal constraints, such as bus design/bandwidth and number of read/write register ports, sometimes prevent the scheduler from having any opportunity to launch multiple instructions together. These design constraints manifest themselves in many ways, especially restricting simultaneous launch based on the types of instructions, and/or order of the instructions, within groupings. The result is frequent sequential (instead of simultaneous) instruction launches.

hyperSPARC's ability to launch multiple instructions simultaneously is not restricted by the order and type† of instructions within groupings. Sequential launch is required, of course, for cases involving resource conflicts or data dependencies. But unlike some other superscalar designs, any instruction can occupy any position in the grouping and still be considered for simultaneous launch.

hyperSPARC also provides special support for the launching of floating-point instructions. The hyperSPARC floating-point unit employs two queues: a *pre-queue* and *post-queue*. The post-queue maintains information on instructions currently in execution in either the floating-point adder or multiplier units. This information includes the instruction type, address, and stage of the pipeline for any given clock. Information of this type is required for exception handling to recover the instructions aborted when the floating-point pipeline is flushed due to a trap.

ROSS, however, extends this principle to a floating-point pre-queue, which holds the same information for up to four instructions that are pending execution. The significance of this pre-queue is that it allows floating-point instructions to be sent to the pre-queue from the normal integer instruction stream. The hyperSPARC scheduler is capable of fetching and *dispatching* any two floating-point instructions at a time, sending both to the pre-queue *in the same clock cycle*. If the floating-point unit is not busy, one of the two floating-point instructions bypasses the pre-queue and begins final instruction decode and execution immediately. The integer pipeline proceeds uninterrupted to fetch, decode, and execute more instructions in the next clock cycle (*Figure 1–2*). This is made possible with hyperSPARC's dual-level instruction decoding, which off-loads final instruction decode to the floating-point unit. In hyperSPARC, integer multiplies and divides are executed in the integer ALU, removing this workload from the floating-point unit.

† There is a group of infrequently occurring instructions that must be launched sequentially (e.g., JMPL, RETT, FCMP, FLUSH, etc., and privileged instructions such as RDY and WRY), but they represent only a small fraction of most executable programs.

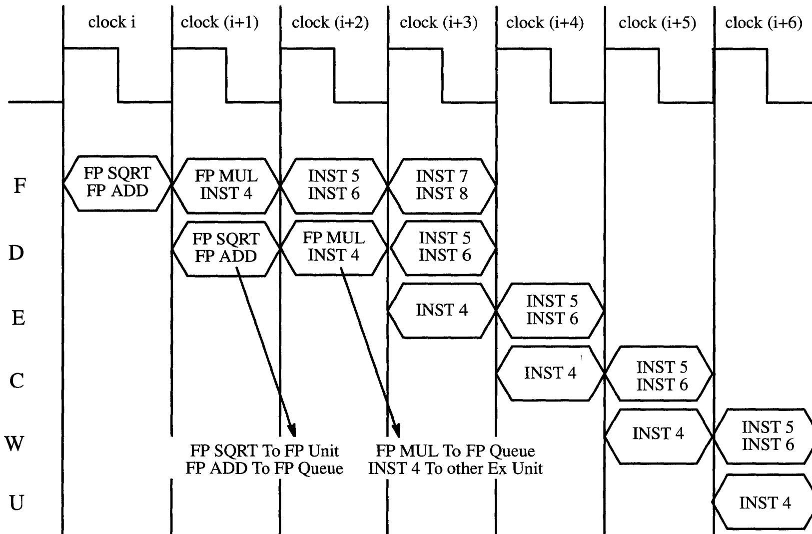


Figure 1–2. RT620 Instruction Pipeline Example

1.2.1.3 Multiprocessing

Multiprocessing is a key to dramatically higher performance from existing silicon technology. hyperSPARC provides full hardware support for tightly-coupled multiprocessing system architectures.

hyperSPARC provides a high-performance snoop mechanism to facilitate efficient data transfers between processors. In a write-invalidate protocol, such as the one implemented in Level 2 MBus, caches residing on a shared bus must check, or "snoop," each address request to shared memory space. If a cache owns the cache line at the address being requested, it can respond to the request by copying the data to memory (which later forwards the data to the requesting cache) or supply the data directly to the requesting processor (*direct data intervention*). In the case of a direct data intervention transfer, the cache supplying the data must prevent memory from obtaining the bus and responding to the request.

The SPARC Architecture allows a window of MBus clock cycles within which a cache must assert the Memory InHibit ($\overline{\text{MIH}}$) signal if it owns the requested cache line (i.e., there is a snoop hit). That window is A + 2 cycles to A + 7 cycles, the "A" representing the cycle in which the address of the cache line being requested is placed on the MBus. The hyperSPARC responds on snoop hits with $\overline{\text{MIH}}$ in the A + 3 cycle. This means that memory is free to respond beginning A+4. Using the full window allowed by MBus would impose a three-cycle penalty for every memory access. Responding this quickly, even though the MBus specification offers more relaxed timing, enables a very high-performance memory subsystem to be built around hyperSPARC.

1.2.1.4 Cache Architecture

hyperSPARC was designed as a tightly-coupled chipset in order to achieve optimal performance between processor and cache. The ROSS designers' understanding of the CPU's relationship with the cache is demonstrated in the RT620's design, which imposes only a one-cycle primary cache (instruction cache) miss penalty. A pipeline stage is allotted in the RT620 for accessing the second-level cache so that no additional stall in CPU throughput is realized if the on-chip cache is missed and the second-level cache is hit.

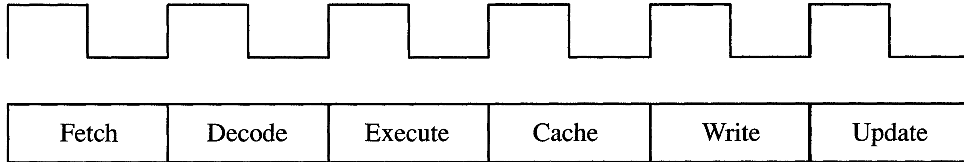


Figure 1–3. RT620 Pipeline Stages

The RT620 utilizes a six-stage pipeline, as shown in *Figure 1–3*. The first three stages are typical RISC pipeline stages: Fetch, Decode, and Execute. The fourth stage of the pipeline is the Cache stage, which is a built-in recognition of the latency of accessing the second-level cache for data access.

Instruction fetches will cause the RT620 to initiate two accesses: one to the on-chip 8-Kbyte instruction cache, and, at the same time, one to the second-level cache. If the address for the instruction is found within the on-chip cache, the access to the second-level cache is cancelled and the instruction is available at the Decode stage of the pipeline. If there is a miss on the internal cache, and a hit on the second-level cache, the instruction is available after a one-cycle miss penalty built into the pipeline. The significance of this design is that it allows the pipeline to proceed uninterrupted as long as the instruction accesses hit either the on-chip cache or the second-level cache, which has been found to be about 90% and 98% of the time, respectively, for typical workstation applications. Since the integer and floating-point pipelines mirror these six stages for reasons of architectural balance and ease of exception handling, this design enables the RT620 to achieve its high throughput rate at speeds that would otherwise not be possible.

The RT627 Cache Data Units utilize a unique single-stage pipeline and data forwarding similar to that used in microprocessor designs. This pipeline design allows the RT627 to keep up with the data rate of the processor, which requires latching and writing data into the RAM core within a short period. Writes into the RT627 are buffered by latching the address and data during the write cycle. The RT620 is then free to perform read operations. The write into the RAM core is delayed until the next write access. Each write access operation provides the opportunity to write the previous write data into RAM without incurring a timing penalty.

The obvious drawback of this approach is the possibility of a read of the data being held in the latches before the RAM core is updated. The RT627 addresses this problem by using data forwarding. A comparitor checks the address of the pending write with the incoming read address. If a match occurs, data is forwarded from the input data latches directly to output pins, bypassing the RAM core. In this way, the most recent data is provided by the RT627 CDUs.

1.2.1.5 Special hyperSPARC Features

There are a number of subtle but clever design features implemented in hyperSPARC that improve performance for common functions required of the CPU. One such feature is the RT620 CPU's Fast Constant/Index/Branch capability.

Fast Constant, for example, represents a commonly occurring combination of two ALU instructions that are used to generate 32-bit constants. Specifically, the SETHI and OR instruction pair is used frequently to create the 22 high-order and 10 low-order bits, respectively (in fact, the current SPARC compilers generate these two instructions from the pseudo-instruction SET for sufficiently large constants.) When hyperSPARC's scheduler encounters this instruction pair, it launches them for execution in parallel, as if they were a single instruction. Thus, an operation that normally takes two cycles (i.e., the setting of the high- and low-order bits for the designated register) is reduced to one cycle. Fast Index works similarly, combining the SETHI and LD instruction pair commonly used to generate a 32-bit base address for array indexing.

Fast Branch is a feature which avoids waiting for condition codes to be set by an ALU instruction before initiating a Branch Target Fetch. This feature allows a branch and an associated ALUcc instruction to be

launched simultaneously. The hyperSPARC uses a branch-taken prediction strategy, and fetches the branch target address. This reduces the number of cycles between branch resolution and target instruction Fetch and Execute if the branch is taken, or continued instruction processing if the branch is not taken.

Block Copy and Block Fill are special features of the RT625 CMTU. These are software-initiated operations to increase the performance of data movement in and out of main memory. Taking full advantage of the RT625's read and write buffers, these block manipulation functions allow data to be moved to or from main memory without having to be brought into cache. This not only saves the latency of filling the cache, but also allows the RT620 to continue processing at the same time.

Block Copy copies an entire 32-byte block of data from a cache or main memory location to another location in main memory. This is particularly useful when copying files, databases or other large memory blocks to other memory locations. If data is being copied from main memory to another location in main memory, for example, it is first read into the read buffer, transferred to the write buffer, and then written to the specified memory location in memory. Block Copy saves more than 10 clock cycles that would be encountered if the block were read into, and then out of, cache.

Block Fill copies into the specified memory location the doubleword embedded in the special block fill STA instruction. The Block Fill works similarly to the Block Copy, only the read transaction is not required since the source data comes from the processor. The specified doubleword pattern is written throughout the 32-byte block of memory, which is very useful in initializing large blocks of memory. The alternative solution would require a cache line in main memory to be brought into cache, initialized with a data pattern, and then written back out to main memory.

1.3 CY7C600 Overview

The CY7C600 chip set is a 32-bit custom CMOS implementation of the SPARC architecture. Designed by ROSS Technology, Inc., the chip set is implemented in 0.8-μm CMOS technology. The chip set is in production and is available at 40 MHz. The CY7C600 family includes the CY7C601 Integer Unit, the CY7C602 Floating-Point Unit (FPU), the CY7C604 Cache Controller and MMU (CMU), the CY7C605 Cache Controller and Memory Management Unit for Multiprocessing (CMU-MP), and the CY7C157 Cache Storage Unit (CSU). This CPU includes a SPARC Reference MMU and a 64-Kbyte cache, and directly interfaces to a 64-bit physical bus capable of a bandwidth approaching 320 Mbytes per second at 40 MHz. The five-chip CY7C600 CPU requires no glue logic, and provides maximum computing performance with minimal design effort.

1.3.1 Partitioning

The CY7C600 family is designed to offer a complete solution for high-performance computer and controller applications. The CY7C601 Integer Unit and the CY7C602 FPU together comprise the full SPARC instruction set architecture. Additional family members include the CY7C604 CMU for uniprocessor applications, the CY7C605 CMU-MP, and the CY7C157 CSU.

Figure 1–4 and *Figure 1–5* illustrate how CY7C600 family devices connect to each other in both single-processor and multiprocessor applications. The CY7C601's second coprocessor interface is not shown in these diagrams. The function of this second coprocessor (CP) is defined by the system designer, but its interface to the CY7C601 is identical to that of the CY7C602 FPU coprocessor.

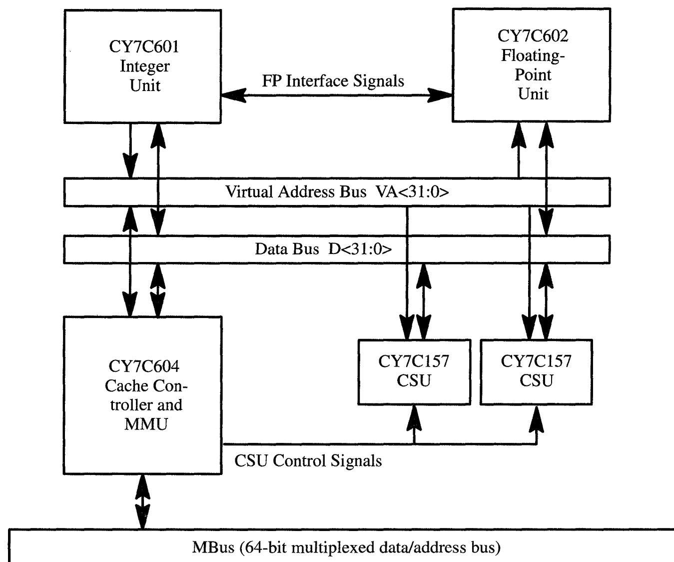


Figure 1–4. Architectural Partitioning—Uniprocessor System

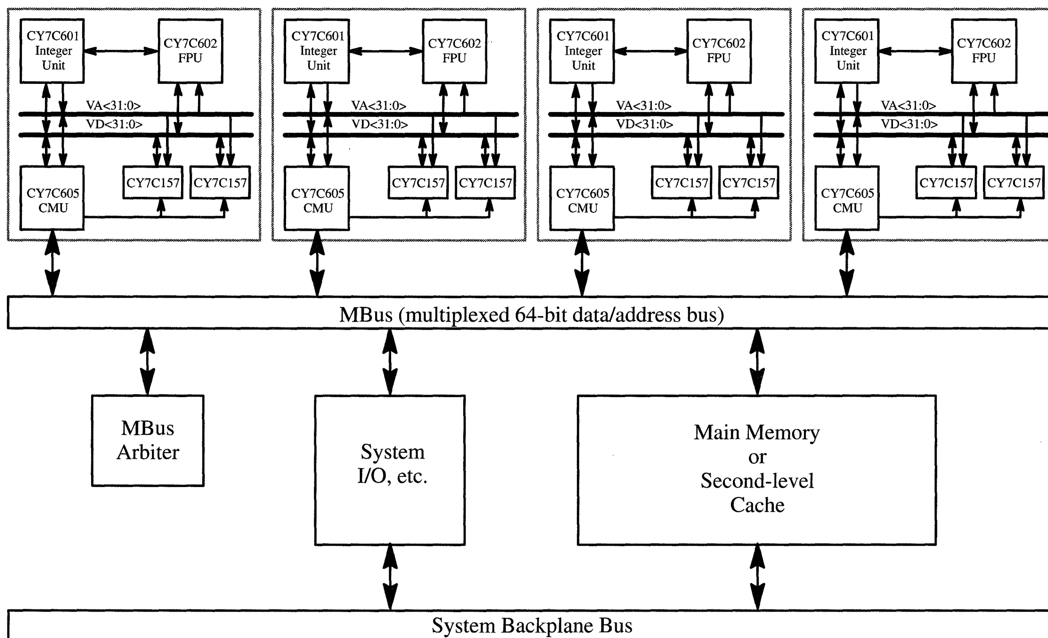


Figure 1–5. Architectural Partitioning—Multiprocessors

1.3.1.1 The CY7C601 Integer Unit

The CY7C601 is the primary processing engine in the SPARC architecture, executing all instructions except for specific floating-point and coprocessor operations. The CY7C602 FPU does its floating-point calculations concurrently with the CY7C601 Integer Unit. The architecture also allows for concurrent operation through the use of an optional second coprocessor.

Significant features of the CY7C601 include:

- Full binary compatibility with entire SPARC application software base
- Architectural efficiency that sustains 1.25 to 1.5 clocks per instruction
- Large-windowed register file
- Tightly-coupled floating-point interface
- User/supervisor modes for multitasking
- Semaphore instructions and alternate address spaces for multiprocessing
- Tagged arithmetic instructions to support artificial intelligence software

1.3.1.1.1 Traps and Exceptions

The CY7C601 supports a full set of traps and exceptions. A table-based set of trap vectors supports 128 hardware and 128 software trap types, both synchronous (error conditions and instructions) and asynchronous (interrupts and reset). The CY7C601 supports a very fast interrupt time of 4 to 7 clocks, depending upon the contents of the instruction pipeline.

1.3.1.1.2 Multitasking

Multitasking is supported with user and supervisor modes. Certain privileged instructions can only be executed while the CY7C601 is in supervisor mode, ensuring that user programs cannot accidentally alter the state of the machine. Supervisor mode is only accessible by using a hardware interrupt or by executing a trap instruction.

1.3.1.1.3 Multiprocessing

The CY7C601 supports multiprocessing with two instructions for implementing semaphores in memory. Atomic Load/Store unsigned byte loads a byte from memory, then sets the memory location to all ones. The SWAP instruction exchanges the contents of a register and a memory location. Both of these instructions are "atomic," meaning they are uninterruptable.

1.3.1.2 CY7C602 Floating-Point Unit

The CY7C602 FPU provides high-performance, IEEE STD-754-1985-compatible single- and double-precision floating-point calculations for CY7C600 systems, and is designed to operate concurrently with the CY7C601. All address and control signals for memory accesses by the CY7C602 are supplied by the CY7C601. Floating-point instructions are addressed by the CY7C601, and are simultaneously latched from the data bus by both the CY7C601 and CY7C602. Floating-point instructions are concurrently decoded by the CY7C601 and the CY7C602, but do not begin execution in the CY7C602 until after the instruction is enabled by a signal from the CY7C601. Pending and currently executing FP instructions are placed in an on-chip queue while the CY7C601 continues to execute non-floating-point instructions.

The CY7C602 has a 32-bit x 32-bit data register file for floating-point operations. The contents of these registers are transferred to and from external memory under control of the CY7C601 using floating-point Load/Store instructions. Addresses and control signals for data accesses during a floating-point load or store are supplied by the CY7C601, while the CY7C602 supplies or receives data. Although the CY7C602 operates concurrently with the CY7C601, a program containing floating-point computations generates results as if the instructions were being executed sequentially.

1.3.1.3 CY7C157 Cache Storage Unit

The CY7C157 is a 16-Kbyte x 16-bit high-performance CMOS static RAM designed specifically as a cache memory for CY7C600 systems. It incorporates registered address and write-enable inputs, latched data inputs and outputs, and a self-timed write mechanism—features that have greatly simplified the design of cache memories for the CY7C600 family.

1.3.1.4 CY7C604/CY7C605 Cache Controller and Memory Management Units

The CY7C604 and CY7C605 are combined cache controller and memory management units designed specifically to support the CY7C601. The CY7C604 and CY7C605 provide control for a 64-Kbyte direct-mapped virtual cache and provide a SPARC reference standard MMU for virtual to physical address translation. The CY7C604 and CY7C605 directly interface with the CY7C600 family, requiring no glue logic for a 64-Kbyte cache system. The CY7C604 and CY7C605 use two CY7C157 Cache Storage Units to implement a 64-Kbyte cache system using only three chips. Cache tag memory is provided as an on-chip feature of the CY7C604/CY7C605, thereby reducing hardware complexity for a CY7C604- or CY7C605-based system.

The CY7C604 is optimized for uniprocessor systems, providing cache locking and cache expandability to 256 kilobytes using additional CY7C604s. The cache locking feature of the CY7C604 allows deterministic response from the cache system, an important feature for real-time systems. The SPARC reference MMU, supported on both the CY7C604 and the CY7C605, provides translation of a 4-Gbyte virtual address space to a 64-Gbyte physical address space. Both the CY7C604 and the CY7C605 provide a 64-entry fully associative translation lookaside buffer (TLB), used in translating virtual addresses to physical addresses. TLB entries may be locked, excluding critical TLB entries from replacement and thereby preventing unnecessary table walks. Table walking (required to obtain additional virtual to physical address translations not stored in the TLB) for the CY7C604 and CY7C605 is implemented in hardware, providing a substantial time savings over software table walk routines.

The SPARC MMU section of the CY7C604/CY7C605 is designed for the efficient support of multitasking operating systems. CY7C604/CY7C605 TLB and cache tag entries allow a maximum of 4096 different context tags to identify tasks within an operating system. The SPARC MMU implemented in the CY7C604/CY7C605 provides extensive memory access level protection (User/Supervisor and read/write/Execute), including an execute-only memory access level. The ability to mark memory accesses as execute-only provides a security feature that can be used to protect proprietary features of a software system from unauthorized scrutiny. The CY7C604 and CY7C605 MMU also support multilevel address mapping, allowing software to select a region of 4 Kbytes, 256 Kbytes, 16 Mbytes, or 4 Gbytes to be addressed by a single TLB entry. This feature allows efficient utilization of TLB entries, which in turn reduces the number of table walks caused by system software.

The CY7C605 is an extension of the CY7C604 designed for use in multiprocessor systems. The CY7C605 provides a dual cache tag memory, which allows the CY7C605 to perform bus snooping while it simultaneously supports cache accesses by the CY7C601. The CY7C605 implements a cache coherency protocol based on the IEEE Futurebus, which has been recognized as a superior protocol for maintaining consistency

of shared data in a multiprocessing system. The CY7C605 supports direct data intervention, which is the capability of a CY7C605-based cache to directly supply modified data to another requesting cache without first requiring main memory to be updated. This feature provides a significant performance advantage over cache systems that must update main memory in order to supply modified data to another cache. In addition to direct data intervention, the CY7C605 also supports memory reflection. Memory reflection allows a memory system to automatically update itself during a direct data intervention operation. This feature allows a multiprocessing system to update both a requesting cache and main memory in a single bus operation.

Both the CY7C604 and the CY7C605 are specifically designed to support secondary cache systems. The use of common secondary caching provides the advantage of increased cache performance for each processing node of a multiprocessor system without the expense of large caches for each node. This approach provides a direct upgrade path to the next generation of high-integration SPARC processors, and also allows a system to be upgraded from uniprocessor to multiprocessor by modifying the operating system and replacing the CY7C604 with the CY7C605.

The CY7C604 and CY7C605 support the SPARC MBus standard bus interface. The MBus is a peer level, high-speed, 64-bit, multiplexed address and data bus which supports a full peer-level protocol (i.e., multiple bus masters). The CY7C604/605 MBus supports data transfers in transaction sizes of 1, 2, 4, 8, or 32 bytes. These data transfers are performed in either burst or non-burst mode, depending upon size. Data transactions larger than eight bytes (one doubleword) are transferred in burst mode, which consists of an address phase followed by four data phases. Non-burst transactions consist of an address phase followed by one data phase, and are used for data transactions of eight or fewer bytes. Bus mastership is granted and controlled by an external bus arbiter. The bus arbiter sets bus priorities, and grants access to a bus master.

The MBus is divided into two levels of implementation: Level 1 and Level 2. Level 1, implemented on the CY7C604, is the uni-processor version of MBus. Level 1 is a subset of Level 2, which is the multiprocessor version of MBus. The CY7C605 supports Level 2 MBus. Level 2 MBus includes the IEEE Futurebus (MOSEI) cache coherency protocol, which has been recognized in the industry as a superior method of supporting multiprocessing systems. Level 2 MBus defines five cache states for describing cache line status. Transactions on the MBus are monitored or "snooped" by the CY7C605 and other bus agents on the Level 2 MBus to maintain ownership and modified status for each cache line. Transactions on the Level 2 MBus are made with respect to the cache line ownership and modified status to ensure consistency for shared data images.

The Level 2 MBus supports direct data intervention, which allows a cache system with the up-to-date version of a cache line to directly supply the data to another cache system without having to first update main memory. Direct data intervention provides a significant performance improvement over systems that do not support this feature. In addition, the CY7C605 provides support for memory systems with reflective memory controllers. A memory system with reflective memory control can recognize a cache-to-cache data transaction and automatically update itself without delaying the system. Another system concept supported by the CY7C605 is secondary caching. Secondary caching provides a performance advantage over systems directly using main memory, and provides an economic advantage over systems using large caches for each processing node.

SPARC Programming Environment

The purpose of this chapter is to provide a general description of the programming environment for ROSS Technology SPARC products. The SPARC Architecture Specification, available from SPARC International, describes the features and requirements of a SPARC compatible system. This architecture specification is specific enough to ensure software compatibility between SPARC systems, yet allows for design flexibility within the specification. Therefore, this chapter describes the SPARC programming environment as implemented on the RT600 hyperSPARC and CY7C600 SPARC families of processors.

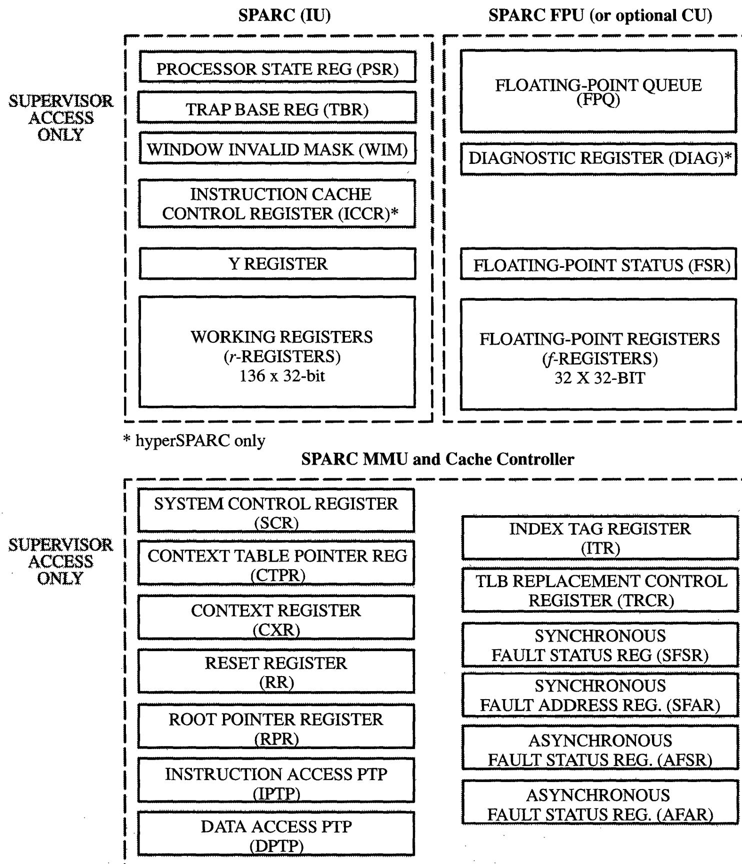


Figure 2–1. SPARC Register Models

2.1 Programming Model

The SPARC register model, register window mechanism, processor states, supervisor/user modes, control/status registers, and data types are described in detail in this section. The concepts and properties explained here are central to an understanding of the SPARC operation.

Figure 2–1 represents the SPARC register set available to the programmer. The register sets for the hyperSPARC RT600 and the CY7C600 families are identical with the following exceptions:

- The instruction cache control register (ICCR) has been added to the RT620 Central Processing Unit (CPU) in order to allow control of the on-chip instruction cache.
- In order to provide a diagnostic interface for chip testing, the diagnostic register (DIAG) has been added to the RT620. This register is for manufacturing test purposes, and is not intended for user access.
- The floating point unit (FPU) portion of the RT620 has a four-entry pre-queue and a three-entry post-queue, as compared to the CY7C602 FPU three-entry queue.
- The features of the hyperSPARC RT625 Cache Controller, Memory Management, and Tag Unit (CMTU) have been enhanced over the previous CY7C604 and CY7C605 CMUs. Therefore, control registers such as the system control register (SCR), reset register (RR), synchronous fault status register (SFSR), and the asynchronous fault status register (AFSR) have been changed.

SPARC registers can be divided into two general classifications: working registers and control/status registers. Working registers are those used for data and addressing operations. They are called *r*-registers for the integer unit (IU), or *f*-registers in the floating-point unit. The various control/status registers record status or control the state of a processor or memory management unit (MMU).

The 136 *r*-registers of the integer unit are divided into eight register windows, as described in the next section. The 32 *f*-registers of the floating-point unit are a directly addressed register file (referred to as *freg 0..freg 31*) and are discussed in *section 2.1.3*. The various control/status registers for the IU, the FPU, and the cache controller/MMU are discussed in their respective sections.

All registers for SPARC are 32-bits in length, although floating-point double-precision instructions allow an adjacent and aligned floating-point data register pair to be accessed as a single 64-bit register. Also note that while all control registers are 32-bits in length, some of the bit fields may be designated as *reserved.* Reserved bits are non-writable, and are returned as zero when the register is read. It is good programming practice to write zeros into a reserved bit field when writing to a control register of this type. This practice avoids upgrade problems with later hardware versions.

2.1.1 Supervisor/User Modes

In support of multitasking, SPARC employs a supervisor/user model of operation. The processor is in supervisor mode when the S bit in the processor state register (PSR) is set, and in user mode when S is reset (see *Section 2.2.1.2*). The state of this bit determines whether or not privileged instructions may be used. Privileged instructions restrict control register access to supervisor software, preventing user programs from accidentally altering the state of the machine.

A program running in user mode may enter supervisor mode by encountering a software or hardware trap. A return to user mode is accomplished by executing a return from trap (RETT) instruction, which restores the state of the S bit to what it was before the trap was taken. A commonly used trap return is the JMPL, RETT delayed control transfer couple (refer to *Section 2.4.3.4.4*). This restores both the PC and nPC (see *Section 2.2.1.1*) and the previous state of the S bit.

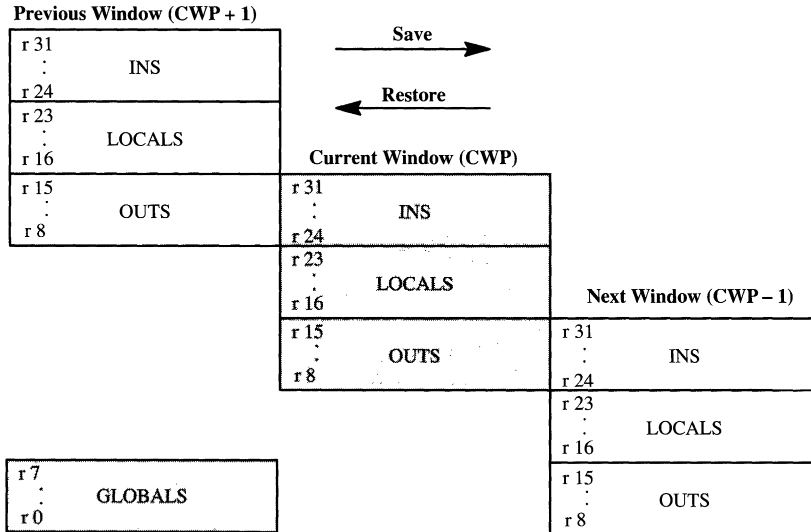


Figure 2–2. Overlapping Windows

2.1.2 Register Windows

The SPARC architecture uses a "windowed" register file model in which the file is divided up into groups of registers called windows. This windowed register model simplifies compiler design, speeds procedure calls, and efficiently supports A/I programming languages such as Prolog, LISP and Smalltalk.

The 136 *r*-registers of the RT620 and CY7C601 are 32-bits wide and are divided into a set of 128 window registers and a set of eight global registers. The 128 window registers are grouped into eight sets of 24 *r*-registers called windows. One of these eight windows is selected by setting the Current Window Pointer (CWP), a 5-bit field in the processor state register (PSR) (see *Section 2.2.1.2*). Within each window, the programmer can directly access 24 windowed *r*-registers by register number (as described in *Table 2–1*). The eight global registers may be accessed regardless of the window selected by the CWP.

Table 2–1. Register Addressing

Register Number	Alternate Register Number	Register Group Name
r[24] to r[31]	i[0] to i[7]	ins
r[16] to r[23]	l[0] to l[7]	locals
r[8] to r[15]	o[0] to o[7]	outs
r[0] to r[7]	g[0] to g[7]	globals

The windowed register file is implemented as a circular stack, with the highest numbered window joined to the lowest. For the eight windows implemented in SPARC, window 7 adjoins window 0.

Note that each window shares its *ins* and *outs* with adjacent windows (refer to *Figure 2–2*). *Outs* from a previous window (CWP+1) are the *ins* of the current window, and the *outs* of the current window are the *ins* of the next window (CWP–1). While only adjacent windows share *ins* and *outs*, *globals* are shared by all windows. A window's *locals*, on the other hand, are not shared at all, belonging only to that window.

An alternative approach to understanding SPARC window registers is to note that the Current Window Pointer (CWP) acts as an index pointer within the stack of 128 window registers. Changing the Current Window Pointer by one offsets the *r*-register addressing by 16. Since 24 *r*-registers can be addressed with each CWP value, incrementing or decrementing the CWP results in an eight register overlap in register addressing. This overlap of window register addressing creates the in-out feature of the windowed registers.

Programming Note:

After power-on reset, the state of the Current Window Pointer and the WIM register (see *Section 2.2.1.3*) are undefined. The power-on reset trap routine must initialize the CWP and WIM register for correct operation.

2.1.2.1 Parameter Passing Using Register Windows

Register window overlap provides an efficient means of passing parameters during procedure calls and returns. One method of implementing a procedure call that takes advantage of the overlap is to have the calling procedure move the parameters to be passed into its *outs* registers, then execute a CALL instruction. A SAVE instruction then decrements the CWP to activate the next window. The calling procedure's *outs* become the called procedure's *ins*, making the passed parameters directly accessible.

When a called procedure is ready to return results to the procedure that called it, those results are moved into its *ins* registers and it then executes a return, usually with a JMPL instruction. A RESTORE instruction increments the CWP to activate the previous window. The called procedure's *ins* are still the calling procedure's *outs*; thus the results are available to the calling procedure. Note that the terms *ins* and *outs* are defined relative to calling, not returning.

If the calling procedure must pass more parameters than can be accommodated by the *outs* and *globals*, the additional parameters must be passed on the memory stack. One method of handling the stack pointer is to dedicate an *out* register in the current window to hold the stack pointer (see *Figure 2–3*). After a CALL, this pointer (which is now in an *ins* register) can be used as the frame pointer for the called procedure. The SAVE instruction, in addition to decrementing the CWP, also performs an ADD using registers from the current window and placing the result in a register in the next window. This feature can be used to set a new stack pointer for the called procedure from the old pointer in the calling procedure. RESTORE also performs an ADD, using registers in the current window and placing the result in the previous window.

2.1.2.2 Window Overflow and Underflow

No matter how many windows a register file has, it is possible that at some point the program will try to use more than are available. Since the register file is a circular stack, something must be done to prevent overwriting the oldest window as the stack wraps around.

Window management is provided by using bits in the window invalid mask (WIM) register to mark windows that will trigger an underflow or overflow trap (see *Section 2.2.1.3*). If a SAVE instruction points the CWP to a marked window, a window overflow trap is generated. This means that only seven of the eight windows are available for calls, because the last window must be saved for the trap handler. However, since a typical overflow trap handler would transparently save one or more of the oldest windows to memory, the program sees an apparently infinite number of windows.

The CWP is automatically decremented upon encountering a trap. This happens without generating another window overflow trap, regardless of the state of the WIM register. By setting at least one window as masked by the WIM register, the system is assured of at least one window for use by the trap handler.

in	r31	(i7)	return address
	r30	(FP)	frame pointer
	r29	(i5)	incoming param reg 5
	r28	(i4)	incoming param reg 4
	r27	(i3)	incoming param reg 3
	r26	(i2)	incoming param reg 2
	r25	(i1)	incoming param reg 1
	r24	(i0)	incoming param reg 0
local	r23	(l7)	local 7
	r22	(l6)	local 6
	r21	(l5)	local 5
	r20	(l4)	local 4
	r19	(l3)	local 3
	r18	(l2)	local 2
	r17	(l1)	local 1
	r16	(l0)	local 0
out	r15	(o7)	temp
	r14	(SP)	stack pointer
	r13	(o5)	outgoing param reg 5
	r12	(o4)	outgoing param reg 4
	r11	(o3)	outgoing param reg 3
	r10	(o2)	outgoing param reg 2
	r9	(o1)	outgoing param reg 1
	r8	(o0)	outgoing param reg 0
global	r7	(g7)	global 7
	r6	(g6)	global 6
	r5	(g5)	global 5
	r4	(g4)	global 4
	r3	(g3)	global 3
	r2	(g2)	global 2
	r1	(g1)	global 1
	r0	(g0)	0
floating point	f31		floating-point value
	:		:
	f0		floating-point value

Figure 2–3. Registers as Seen by a Procedure

A RESTORE instruction causes a window underflow trap if it attempts to restore to a window invalidated by the WIM register. Execution of a return from trap (RETT) instruction under the same circumstances will also generate an underflow trap. SAVE, RESTORE, and RETT always check the WIM register before completing their actions.

For example, if the procedure using the window 0 executes a CALL and SAVE sequence and the WIM bit 7 is set, a window overflow trap occurs. The overflow trap handler may safely use only the *locals* of w7, because w7's *ins* are w0's *outs* and w7's *outs* are w6's *ins*.

Active window = 0	CWP = 0
Previous window = 1	CWP+1 = 1
Next window = 7	CWP-1 = 7
Trap window = 7	WIM = $10000000_{(base\ 2)}$

The overflow trap handler is responsible for saving one or more of the least recently used windows to the memory stack. Simulations of register file management methods show that saving and restoring one window at a time is the simplest and most effective algorithm for handling overflow and underflow. The stack pointer to the window-save area must be aligned to a word boundary in valid memory and, for efficiency, should be doubleword aligned. This is because it is faster to load and store doublewords than to load and store words.

A linear sequence of doubleword loads and stores is also used to speed up context switches. In a context switch, only the windows containing valid data are saved, and on average this is about half the number of *r*-register windows, minus one for the reserved trap window.

2.1.2.2.1 Alternate Register Window Usage

Although the windowing layout is particularly well suited to procedure calls and returns, hardware does not force their use for that purpose alone. Except for the eight-register overlap and the partial fixing of the function of several registers by the instruction set (see *Section 2.1.2.3*), register windows can be viewed and manipulated as needed to fit the application at hand.

For example, the register set can be treated as a flat register file. Access to any particular register in any window is obtained by writing its window value into the Current Window Pointer located in the processor state register. Moreover, windows naturally segment registers into blocks that could be dedicated to specific purposes and accessed through the CWP. Register saving and parameter passing could be done with a standard push/pop stack in memory, although this would substantially increase bus traffic.

For real-time and embedded controller systems, where Fast Context switching may be more important than procedure calling, the register file can easily be divided into banks of registers separated by trap handling windows set up by the WIM register (see *Section 2.2.1.3*). Switching from one register bank to another is accomplished by writing to the CWP field of the processor state register. *Globals* are accessible by all processes.

2.1.2.3 Special Registers

In general, the window registers seen at any given time can be used in any manner desired, while keeping in mind that windows overlap at both ends. However, the instruction set does fix the use of r[0] and partially fixes the use of r[15].

Global register r[0] always returns the value 0 when read, making the most frequently used constant easily available at all times. In addition, when addressed as a destination operand, r[0] discards the value written to it.

The CALL instruction writes its own address into register r[15] (*out* register 7) of the calling procedure's window. If a SAVE instruction then activates a new window, r[15] of the old window becomes r[31] (*in* register 7) of the new window and serves as the return address to the calling procedure. However, if the register is needed for some other purpose, the return address can be saved to a stack or simply overwritten.

Two other registers are also used by hardware to save information during a trap. Registers r[17] and r[18] (*locals* 1 and 2) of the trap window (not the trapping procedure's window) are used to save the contents of the program counters (PC and nPC) at the time the trap is taken. Because the trap window *locals* are all a trap handler is allowed to use (unless it saves to the system stack), this limits the trap handler's usable registers to six.

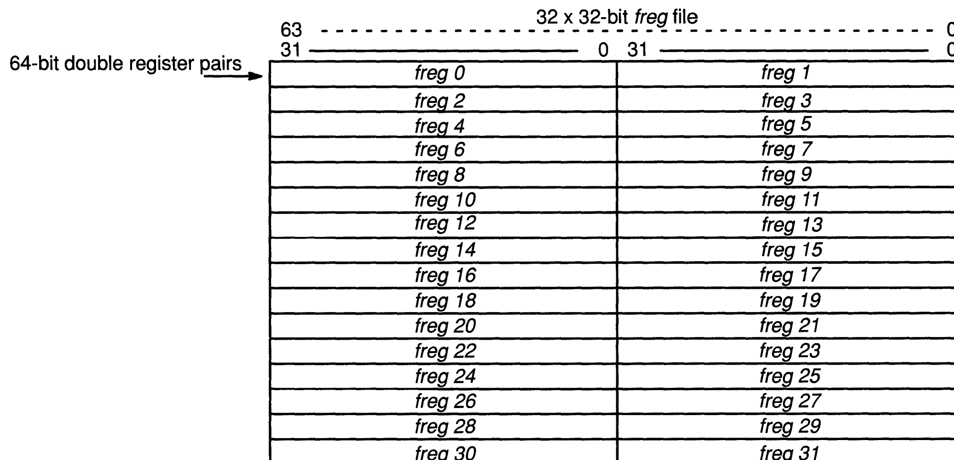


Figure 2–4. FPU Register File

2.1.3 Floating-Point Register File (FREGS)

In addition to the 132 *r*-registers, SPARC defines a set of 32-bit floating-point data registers, referred to as *f-registers*. The RT620 and CY7C602 fp register files each provide a set of 32 *f*-registers. These registers can be accessed as 32 registers containing single precision (32-bit) data types or as 16 pairs of registers containing double precision (64-bit) data types. Double precision register pairs are always addressed as adjacent even-odd registers.

2.2 SPARC Control/Status Registers

Control/status registers provide the software control interface for the SPARC CPU. Control/status registers are divided into three groups: integer unit, floating-point unit, and cache controller/MMU. For the RT620, both the integer unit and floating-point unit control/status registers reside in the same processor. The CY7C601 uses a separate floating-point unit. The cache controller/MMU is a separate unit for both the hyperSPARC and the CY7C600 processor families.

With the exceptions of the Y register and the FPU Status register (FSR), control/status registers are generally restricted to supervisor-mode access. The following sections describe the control/status registers for the integer unit, and the FPU. Control/status registers for ROSS SPARC cache controller and memory management units are described in *Chapter 4* (RT625) and *Chapter 8* (CY7C604/CY7C605).

2.2.1 Integer Unit Control/Status Registers

This section describes the control and status registers for the integer unit portion of the RT620 and the CY7C601 Integer Unit. These registers are identical for both families of processors, with minor exceptions noted.

The two program counters (PC and nPC) are accessed indirectly using such instructions as a CALL, JMPL, software trap (Ticc), and return from trap (RETT). The processor state register (PSR), window invalid mask (WIM), trap base register (TBR), and multiply-step register (Y), are all read/write registers. Read/write instructions that access the PSR, WIM, and TBR are privileged and thus may only be used in supervisor mode.

2.2.1.1 Program Counters (PC and nPC)

The program counter (PC) contains the address of the instruction currently being executed by the SPARC processor, and the next program counter (nPC) holds the address (PC + 4) of the next instruction to be executed (assuming there is no control transfer and a trap does not occur). The nPC is necessary to implement delayed control transfer couples, wherein the instruction that immediately follows a control transfer may be executed before control is transferred to the target address (see *Section 2.4.3.4*). Having both the PC and nPC available to the trap handler allows a trap handler to choose between retrying the instruction causing the trap (after the trap condition has been eliminated) or resuming program execution after the trap causing instruction.

2.2.1.2 Processor State Register (PSR)

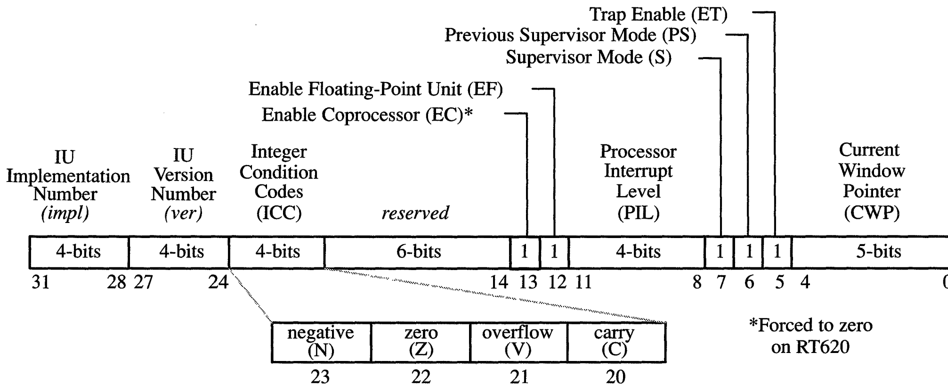


Figure 2–5. Processor State Register

The processor state register is the primary status and control register for the RT620 or the CY7C601, containing fields that report the status of processor operations or control processor operations. Instructions that modify its fields include SAVE, RESTORE, Ticc, RETT, and any instruction that modifies the condition code field (*icc*). Any hardware or software action that generates a trap modifies the S, PS, and ET fields. The PSR may be read or written directly using the privileged instructions RDPSR and WRPSR. Figure 2–5 illustrates the fields of the PSR. Field names given in *italics* in *Figure 2–5* are read-only and cannot be modified with the WRPSR instruction. The PSR is made up of the following fields:

impl* — *Implementation: PSR(31:28) contain the processor's implementation number. The implementation number for ROSS Technology SPARC is "0001." This field, along with the version field, is provided to allow software to identify the processor manufacturer and version. WRPSR does not modify this field.

ver* — *Version: PSR(27:24) contain the ROSS SPARC processor version number. WRPSR does not modify this field. The version numbers for ROSS SPARC processors are '0001' for the CY7C601 and '1111' for the RT620.

ICC* — *Integer Condition Codes: PSR(23:20) hold the integer unit's condition codes. These bits are modified by arithmetic and logical instructions whose names end with the letters *cc* (for example, ANDcc), and can be overwritten by the WRPSR instruction. The Bicc and Ticc instructions base their control transfer on these bits, which are defined as follows:

N — Negative: PSR(23) indicates whether the ALU result was negative for the last icc-modifying instruction. This bit is set to '1' to indicate a negative result.

Z — Zero: PSR(22) indicates whether the ALU result was zero for the last icc-modifying instruction. This bit is set to '1' to indicate a zero result.

V — Overflow: PSR(21) indicates whether an arithmetic overflow occurred during the last *icc*-modifying instruction. This bit is set to '1' to indicate an arithmetic overflow. The overflow bit is also set if a tagged operation (TADDcc, TSUBcc, etc.) is performed on non-tagged operands (refer to *Section 2.4.3.2.3*). Logical instructions that modify the icc field always set the overflow bit to 0.

C — Carry: PSR(20) indicates whether an arithmetic carry out of result bit 31 occurred from the last icc-modifying addition or if a borrow into bit 31 resulted from the last icc-modifying subtraction. This bit is set to '1' to indicate a carry/borrow occurrence. Logical instructions that modify the icc field always set the carry bit to 0.

reserved: PSR(19:14) are reserved. This field is fixed at 0; a write to any bit in this field is ignored.

EC — Coprocessor Enabled: PSR(13) determines whether a coprocessor is enabled or disabled. This bit is set to '1' to enable the coprocessor interface. This feature is supported only on the CY7C601. Consequently, this bit is fixed to zero in the RT620, and writes to this bit are ignored for this processor.

CY7C601 note:

If the coprocessor is either disabled or enabled but not present, a CPop, CBccc, or coprocessor Load/Store instruction will cause a coprocessor-disabled trap. When the CP is disabled, it retains that state until it is re-enabled or reset. Even when disabled, the coprocessor can continue to execute instructions if it contains a queue.

EF — Enable Floating-Point Unit: PSR(12) determines whether the FPU is enabled or disabled. This bit is set to '1' to enable the FPU. If the FPU is either disabled or enabled but not present, an FPop, FBfcc, or floating-point Load/Store instruction will cause a floating-point-disabled trap. When disabled, the FPU retains that state until it is re-enabled or reset. Even when disabled, it can continue to execute any instructions in its queue.

Note that if the EF bit in the PSR is set to zero while there are instructions executing in the floating point unit, the behavior is undefined. Software should wait for all instructions to complete and recognize all pending floating-point exceptions before disabling the FPU.

PIL — Processor Interrupt Level: PSR(11:8) identify the processor's external interrupt priority level. The processor will only accept external interrupts whose interrupt level is greater than the value in PIL or whose interrupt level is 15, which denotes a non-maskable interrupt. Note that a PIL = 0x0 denotes no interrupt request. Bit 11 of the PIL is the MSB and bit 8 is the LSB.

S — Supervisor: PSR(7) determines whether the processor is in supervisor or user mode. This bit is set to '1' to indicate supervisor mode. Because WRPSR is privileged and only available in the supervisor mode, supervisor mode can only be entered by a software or hardware trap.

PS — Previous Supervisor: PSR(6) holds the value that was in the S bit at the time the most recent trap was taken.

ET — Enable Traps: PSR(5) determines whether traps are enabled. This bit is set to '1' to enable traps. If traps are disabled, all asynchronous (or interrupting) traps are ignored. If a synchronous (also called precise) or floating-point/coprocessor trap occurs while traps are disabled, the SPARC processor halts and enters error mode (for a description of error mode, see *Section 3.9* for RT620 and *Section 6.5.5* for CY7C601).

CWP — Current Window Pointer: PSR(4:0) contain a pointer to the currently active register file window. CWP is decremented by traps and the SAVE instruction, and is incremented by RESTORE and RETT instructions.

Programming Note:

If it is necessary for the software to manually disable traps, care must be taken when changing the ET bit from enabled (ET=1) to disabled (ET=0), since the RDPSR, WRPSR instruction sequence is interruptible. A suggested solution is to write all interrupt trap handlers so that before they return program control to the interrupted supervisor routine, they restore the PSR to the value it had before the interrupt was taken. This will guarantee a correct result when the interrupted RDPSR, WRPSR sequence continues. The only PSR bit that cannot be restored is the PS bit, which is overwritten when the trap is taken.

An alternative to the RDPSR–WRPSR sequence is to generate a "trap instruction" trap with a Ticc instruction. A taken trap automatically sets ET to 0, disabling further traps.

2.2.1.3 Window Invalid Mask Register (WIM)

This register designates which window(s) will cause generation of an underflow or overflow trap when pointed to by the CWP as the result of a SAVE, RESTORE, or RETT instruction. The WIM register does not affect register window access except in the case of the SAVE, RESTORE, or RETT instructions.

Each bit in the WIM register (see *Figure 2–6*) corresponds to a window; if a bit is set to 1, the window corresponding to that bit is marked as invalid. If a SAVE, RESTORE, or RETT instruction would cause the CWP to point to a window whose WIM bit equals 1, a window overflow (SAVE) or window underflow (RESTORE, RETT) trap is generated. The overflow trap prevents previous windows from being overwritten, while an underflow trap is used to restore previous windows from memory. No traps are generated if an invalidated window is accessed by directly changing the Current Window Pointer (CWP) of the PSR through the use of a WRPSR instruction.

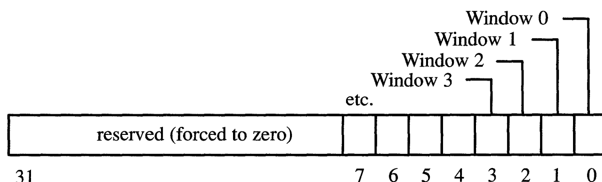


Figure 2–6. Window Invalid Mask

In general practice, a WIM bit is usually set by the operating system software to identify the boundary between the oldest and newest window. The practice of invalidating a window to separate the end of the windowed register file from the beginning has the effect of reserving a window for use by a trap handler in the case of a full *r*-register file. As a taken trap always causes the processor to switch to a new window, an invalidated window is necessary to prevent over-writing the oldest window in the *r*-register file. In order to prevent over-writing the *in* registers of the oldest register window when the *r*-register file is full, a trap handler should be restricted to using the *local* registers of a window.

WIM is read by the RDWIM instruction, and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeros and are unaffected by writes.

NOTE: The WIM register is NOT cleared during reset. It must be initialized by software.

2.2.1.4 Trap Base Register (TBR)

When a trap occurs, the program counter (PC) is loaded with the contents of the trap base register. The TBR contains two fields that together constitute a pointer into the trap table, which in turn contains the trap han-

dler address (see *Figure 2–7*). RDTBR can read the entire register; however, the WRTBR instruction can write only to the trap base address field. Only hardware can write to the trap type field, and bits 0 through 3 are zeros and are unaffected by a write. The trap type field can be directly manipulated using the Ticc instruction. For more information on trap operation, see *Section 2.4.5*.

TBA — Trap Base Address: TBR(31:12) contain the most-significant 20 bits of the trap table address. This field applies to all trap types except reset, which forces address 0. The TBA is software controlled.

tt — Trap Type: TBR(11:4) comprise the trap type field, an eight-bit value that provides an offset into the trap table based on the type of trap being taken (see *Section 2.4.5.4.2*). This field retains its value until the next trap is taken.

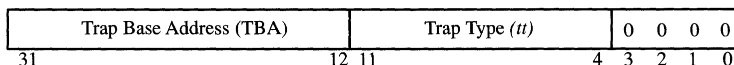


Figure 2–7. Trap Base Register

2.2.1.5 Y Register

The Y register is a 32-bit register used by integer multiply and divide instructions to create 64-bit results. It is used by multiply instructions (such as MULSE UMUL, SMUL, UMULcc, and SMULcc) and integer divide instructions (such as UDIV, SDIV, UDIVcc, and SDIVcc) to hold the 32 most significant bits of an operation. The Y register is also used by the multiply step instruction. The Y register is also used by the multiply step instruction (MULScc) to create 64-bit products. The Y register may also be accessed using the non-privileged RDY and WRY instructions. Refer to *Chapter 12, SPARC Instruction Set* for a full description of these instructions and their operation.

Integer multiplication instructions write the 32 most significant bits of the product into the Y register and the 32 least significant bits into the destination register. The integer division instructions (UDIV, SDIV, UDIVcc, and SDIVcc) use the Y register to hold the 32 most significant bits of a 64-bit dividend.

Note that integer multiply and divide instructions are supported only by the RT620. In the CY7C601, these instructions will cause an unimplemented instruction trap. The CY7C601 requires the user to provide software to support integer multiplication. The MULScc instruction is provided to support software implementation of integer multiplication.

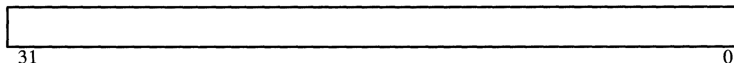


Figure 2–8. Y Register

2.2.1.6 On-chip Instruction Cache Control Register (ICCR) (RT620 only)

This control register has been added to the RT620 to support the on-chip instruction cache. Access to this register is privileged (i.e., only accessible if the PSR supervisor bit is set). It is accessed using the RDASR (read ancillary state register) and WRASR (write ancillary state register) instructions. If the PSR supervisor bit is not set and a read or write to the register is attempted, a privilege exception occurs. The appropriate rd (destination register) or rs1 (source register 1) field must be set to 31 (0x1F) in order to access this register.

Figure 2–9. ICCR Register

The instruction cache enable (IBE, ICCR < 0 >) bit enables ICACHE accesses if it is set and disables accesses if it is cleared.

The flush trap disabled (FTD, ICCR < 1 >) bit determines whether an ICACHE sub-block (packet) will be flushed (invalidated) or an unimplemented flush trap (tt=25) will be taken when a flush instruction is executed.

When FTD = 1, if a flush instruction is executed and an ICACHE hit occurs, the packet corresponding to the address in the ICACHE line is invalidated. When FTD = 0, if an attempt is made to execute a flush instruction, an unimplemented flush exception is taken. The purpose of the FTD bit is to support self-modifying code in a symmetric multiprocessing environment.

Upon power-on reset, the instruction cache is disabled (the IBE bit is cleared) and flush traps are enabled (the FTD bit is cleared). Also during power on, all entries in the instruction cache are invalidated. Writes to bits other than the FTD and IBE bits are ignored. Bits other than FTD and IBE are forced to 0.

2.2.2 FPU Control/Status Registers

The following is a description of the SPARC floating-point status register as implemented in the RT620 and the CY7C602 Floating Point Unit (FPU).

2.2.2.1 Floating-Point Status Register (FSR)

The FSR contains fields which report the status of and control FPU operations. All fp operations can modify the status fields of the FSR. The fields are represented in *Figure 2–10.*

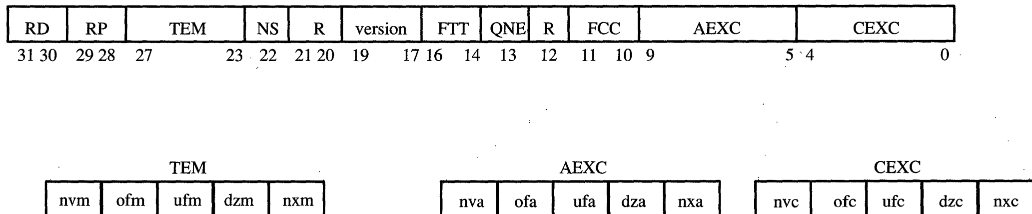


Figure 2–10. Floating-Point Status Register (FSR)

RD —*Rounding Direction:* FSR(31:30) define the rounding direction used by the FPU during a fp arithmetic operation.

00	round to nearest (tie-even)
01	round to zero
10	round to + ∞
11	round to – ∞

U — *Unused:* FSR (29: 28) and FSR(12) are unused. Writes to these fields (via ldfsr) are ignored. The fields read as zero. For future compatibility, software should only issue a ldfsr instruction with zeros in these bits.

TEM — ***Trap Enable Mask:*** FSR(27:23) enable traps caused by fp instructions. These bits are ANDed (1 = enable, 0 = disable) with the bits of the CEXC (current exception) field to determine whether to force an fp trap. This field can be read and written using stfsr and ldfsr respectively.

bit	trap enable mask
27	invalid operation trap mask
26	overflow trap mask
25	underflow trap mask
24	divide by zero trap mask
23	inexact trap mask

NS — ***Non-Standard Floating-Point:*** FSR(22) enables non-standard mode of operation in the FPU (0 = standard mode, 1 = non-standard mode). This bit can be read and written using stfsr and ldfsr respectively.

R — ***Reserved:*** FSR(21:20) are reserved. Writes to these bits are ignored. The bits read as zero. For future compatibility, software should only issue a ldfsr instruction with zeros in these bits.

version: FSR(19:17) is fixed to the appropriate value below. Any write to this field is ignored. Reading the field always returns the fixed value.

FPU	version
RT620	'000'
CY7C602	'011'

FTT — ***Floating-Point Trap Type:*** FSR(16:14) identifies the fp trap type of the current fp exception and is updated on every fp operation. Any write to this field via a LDFSR instruction is ignored. Note that a **stfsr** instruction does not clear the ftt field.

value	trap type
0	none
1	IEEE exception
2	unfinished fp instruction
3	unimplemented fp instruction
4	sequence Error
5	hardware Error
6, 7	reserved

In the hyperSPARC CPU, the hardware error trap type (ftt = 5) is not supported.

qne — ***FPQ Not Empty:*** FSR(13) signals whether the FPQ is empty (0 = empty, 1 = not empty). This bit can be read using the STFSR instruction. Writes to this bit are ignored.

FCC — ***Floating-Point Condition Codes:*** FSR(11:10) report the fp condition codes. This field can be read and written using stfsr and ldfsr respectively.

fcc value	condition
0	equal
1	less than
2	greater than
3	unordered

AEXC —*Accumulated Exceptions:* FSR(9:5) reports the accumulated exceptions that are masked by the TEM field. All masked exceptions are ORed with the contents of the AEXC field and accumulated as status. This field can be read and written using STFSR and LDFSR respectively.

bit	accrued exception type
9	accrued invalid exception
8	accrued overflow exception
7	accrued underflow exception
6	accrued divide by zero exception
5	accrued inexact exception

CEXC — *Current Exceptions:* FSR(4:0) reports the current IEEE fp exceptions. This field is updated on the completion of every fp instruction. This field can be read and written using STFSR and LDFSR respectively.

bit	accrued exception type
4	current invalid exception
3	current overflow exception
2	current underflow exception
1	current divide by zero exception
0	current inexact exception

2.2.3 Cache Controller/MMU Control/Status Registers

The cache control and MMU functions for ROSS SPARC CMU products are provided by the hyperSPARC RT625 CMTU, the CY7C604 CMU, or the CY7C605 CMU-MP. ROSS CMU products adhere to the SPARC reference MMU architecture specifications, and programming features for the MMU functions are identical for each CMU. However, due to progressive feature enhancements in cache management, the cache related programming features change somewhat with each CMU. The differences between these registers has been minimized. However, it is recommended that the programmer refer to the chapter corresponding to the CMU utilized in the target system. Please refer to *Chapter 4* for detailed information on the hyperSPARC RT625, or *Chapter 8* for information on the CY7C604 and CY7C605.

2.3 SPARC Data Types

SPARC supports ten data types (eleven with quad-precision floating-point, see *Section 2.3.3.2.3*). SPARC is a big-endian architecture (refer to *Section 2.3.2*). Integer types include byte, unsigned byte, halfword, unsigned halfword, word, unsigned word, doubleword, and tagged data (see *Figure 2–11*). ANSI/IEEE 754-1985 floating-point types include single- and double-precision. A byte is eight bits wide, halfwords are 16 bits, words and single-precision floating-point are 32 bits, doublewords and double-precision floating-point are 64 bits.

2.3.1 Data Organization In Registers

The organization of the ten data types when loaded into registers is shown in *Figure 2–11.*

When moving memory data to or from the registers, byte operands are always loaded to or extracted from the lower eight bits of a register. On a Load, bits 8 through 31 are sign-extended for a byte or zero-extended for an unsigned byte. halfwords are always loaded to or extracted from the lower 16 bits of a register. Bits 16 through 31 are sign-extended for a halfword or zero-extended for an unsigned halfword during a Load.

All 32 bits of a signed or unsigned word are loaded from or stored to memory. Stores of byte and halfword data are not sign-extended. Tagged data is handled as an unsigned word. doubleword operands load to and store from two contiguous registers, r[n] and r[n+1], with r[n] containing the most significant word. *Figure 2–12* illustrates the relationship between the way data is stored in memory and the way it is loaded into registers.

For single-precision, floating-point operands, bit 31 contains the sign bit, bits 23 through 30 contain the eight bits of exponent, and bits 0 through 22 contain the 23-bit fraction. Double-precision operands require a register pair, with the upper-order register (r[n]) containing the sign bit, 11-bit exponent, and the high-order bits of the fraction. The lower-order register (r[n+1]) contains the low-order bits of the fraction. Total fraction size is 52 bits.

When loading doublewords or double-precision operands from memory to the working registers (either *r* or *f*), the destination register must be at an even address or the hardware will force such an address. For example, an attempted load double to register r[9] would be forced to r[8], so that the most significant word would be loaded in r[8] and the least significant word in r[9]. A load double to r[0] would result in the loss of the most significant word.

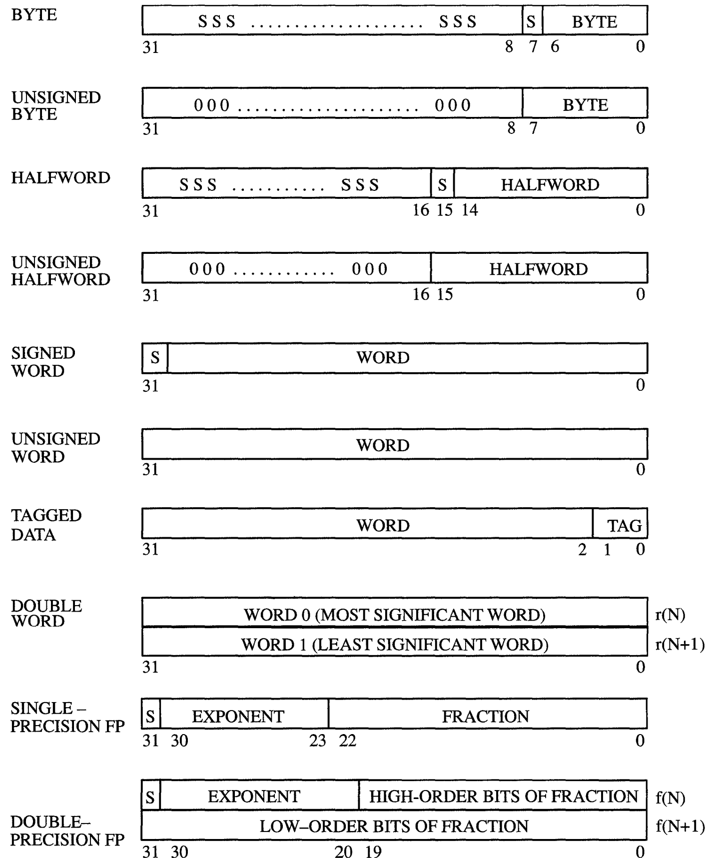


Figure 2–11. Processor Data Types

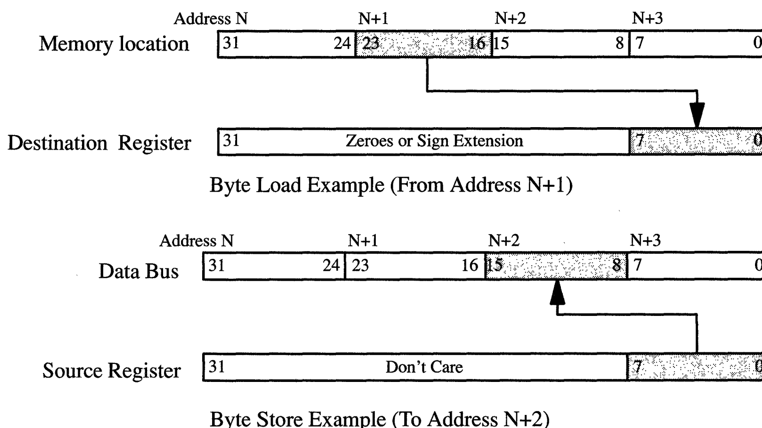


Figure 2–12. Byte Operand Load and Store

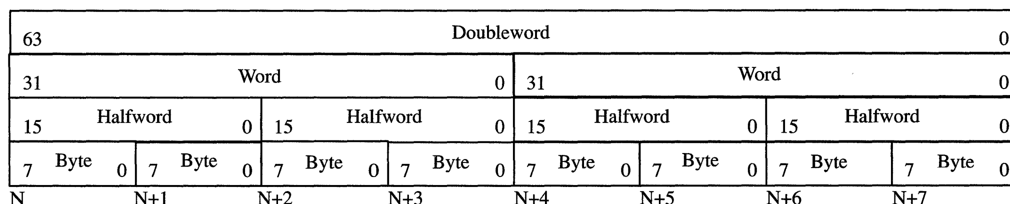


Figure 2–13. Data Organization in Memory

2.3.2 Data Organization In Memory

Organization and addressing of data in memory follows the "big-endian" convention wherein lower addresses contain the higher-order bytes (see *Figure 2–13*). For a stored word, address N corresponds to the most significant byte of the word, and address N+3 corresponds to the least significant byte. The address of a halfword, word, or doubleword is also the address of its most significant byte. A halfword datum must be located on a halfword boundary (address bit <0> = 0), which is evenly divisible by 2. Similarly, a word must be located on a word boundary (address bits <1:0> = 0) evenly divisible by 4, and a doubleword must be located on a doubleword boundary (address bits <2:0> = 0) evenly divisible by 8. Attempting to access misaligned data will generate a memory_address_not_aligned trap.

2.3.3 SPARC Floating-Point Data Types

ROSS Technology SPARC FPUs are compliant to IEEE Std. 754-1985 for floating-point arithmetic. Accuracy of the results of its operations are within ± $^1/_2$ LSB, as specified by the IEEE standard. The following sections describe the IEEE format as implemented on the RT620 and CY7C602.

2.3.3.1 IEEE Definitions

The following terms are used extensively in describing the IEEE-754 floating-point data formats. This section is directly quoted from the *IEEE Standard for Binary Floating-Point Arithmetic*.

biased exponent	The sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative. (Note in the remainder of this section, the term "exponent" refers to a biased exponent.)
binary floating-point number	A bit string characterized by three components: a sign, a signed exponent and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent.
Denormalized	Denormalized numbers are those numbers whose magnitude is smaller than the smallest magnitude representable in the format. They have a zero exponent and a denormalized non-zero fraction. Denormalized fraction means that the hidden bit is zero.
denormalized number	(DNRM) A non-zero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero. (Denormalized numbers are also referred to as subnormal in this text.)
fraction	The field of the significand that lies to the right of its implied binary point.
NaN	Not a number, a symbolic entry encoded in floating-point format. They are used to signal invalid operations and as a way of passing status information through a series of calculations. NaNs arise in one of two ways: they can be generated upon an invalid operation or they may be supplied by the user as an input operand. NaN is further subdivided into two categories: quiet and signaling. Signaling NaNs signal the invalid operation exception whenever they appear as operands. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions.
Normalized	Most calculations are performed on normalized numbers. For single-precision, they have a biased exponent range of 1 to 255, which results in a true exponent range of -126 to $^{+}127$. The normalized number type implies a normalized significand (hidden bit is 1).
significand	The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.
true exponent	The component of a binary floating-point number that normally signifies the integer power to which 2 is raised in determining the value of the represented number.
Zero	The IEEE zero has all fields except the sign field equal to zero. The sign bit determines the sign of zero (i.e., the IEEE format defines a +0 and a –0).

2.3.3.2 IEEE Floating-point Data Formats

The RT620 and CY7C602 directly support single- and double-precision floating-point data formats. Quad-precision (or extended-precision) formats are defined as part of the SPARC architecture, but are not directly executed by the RT620 or CY7C602. Single-, double-, and quad-precision formats are described in this section.

2.3.3.2.1 Single-Precision Floating-Point

Single-precision floating-point data are 32-bits wide and consist of three fields: a single sign bit (s), an eight-bit biased exponent (e), and a 23-bit fraction (f). *Figure 2–14* illustrates the single-precision floating-point format.

Table 2–2. Single-Precision Floating-Point Format

s = sign (1) e = biased exponent (8) f = fraction (23)		
normalized number (0 < e < 255):		$(-1)^S * 2^{e-127} * 1.f$
subnormal (e=0):	$f \neq 0$	$(-1)^S * 2^{-126} * 0.f$
zero (e=0):	f = 0	$(-1)^S * 0$
signaling NaN:	$f \neq 0$	s = u e = 255 (max) f = .0uuu–uu (at least one bit must be non-zero)
quiet NaN:	$f \neq 0$	s = u; e = 255 (max) f = .1uuu–uu
infinity:	f = 0	s = 0 or 1, depending upon sign; e = 255 (max) f = .00–00 (all zeros)

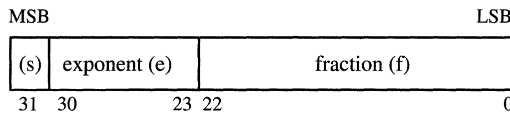


Figure 2–14. Single-Precision Floating-Point Format

2.3.3.2.2 Double-Precision Floating-Point

Double-precision floating-point data are 64-bits wide and consist of three fields: a single sign bit (s), an eleven-bit biased exponent (e), and a 52-bit fraction (f). *Figure 2–15* illustrates the double-precision floating-point format.

Table 2–3. Double-Precision Floating-Point Format

s = sign (1) e = biased exponent (11) f = fraction (52)		
normalized number (0 < e < 2047):		$(-1)^S * 2^{e-1023} * 1.f$
subnormal (e=0):	$f \neq 0$	$(-1)^S * 2^{-1022} * 0.f$
zero (e=0):	f = 0	$(-1)^S * 0$
signaling NaN:	$f \neq 0$	s = u e = 2047 (max) f = .0uuu–uu (at least one bit must be nonzero)
quiet NaN:	$f \neq 0$	s = u e = 2047 (max) f = .1uuu–uu
infinity:	f = 0	s = 0 or 1, depending upon sign; e = 2047 (max) f = .00–00 (all zeros)

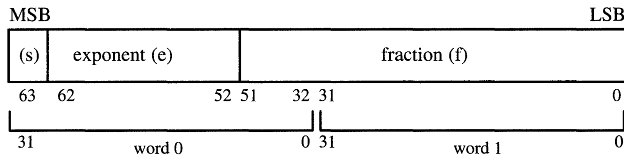


Figure 2–15. Double-Precision Floating-Point Format

2.3.3.2.3 Quad Precision

The SPARC architecture supports another data type, a quad-precision floating-point type with a width of 128 bits (see Table 2–4). For the present, however, the CY7C602 FPU and RT620 hyperSPARC do not implement quad-precision Floating-Point-operate (FPop) instructions, so they must be emulated in software. A quad-precision format FPop will generate a floating-point-exception trap if execution is attempted.

When loaded to the working registers, extended-precision operands require a register quadruple (see Figure 2–16). The upper-order register (r[N]) contains the sign bit, a 15-bit exponent, and the high order 16 bits of the fraction. The next register (r[N+1]) contains the next 32 bits of the fraction, register (r[N+2]) holds the next 32 bits of the fraction, and register (r[N+3]) the low-order 32 bits. As with double-precision operands, when loading a quad-precision operand, the destination register must be at an even address or the hardware will force an even address.

The memory address of a quad-precision datum is also the address of its most significant byte (see Figure 2–17). A quad-precision datum must be located on a quad-precision boundary (address bits <3:0> = 0), which is evenly divisible by 16.

Table 2–4. Quad-Precision Floating-Point Format

s = sign (1) e = biased exponent (15) f = fraction (112 bits) u = undefined	
normalized number (0 < e < 32767): subnormal number (e = 0) (f · 0): zero (s = 0; e = 0) (f · 0):	$(-1)^s$ x $2^{e-16383}$ x 1.f $(-1)^s$ x 2^{-16383} x 0.f $(-1)^s$ x 0
signaling NaN (e=32767): f · 0	s = u e = 32767 (max); f = .0uu–uu (at least one bit must be non-zero)
quiet NaN (e=32767): f · 0	s = u e = 32767 (max); f = .1uu–uu
infinity: f= 0	s = 1, e = 32767 (max); f = .000–00 (all zeroes)

QUAD PRECISION FP

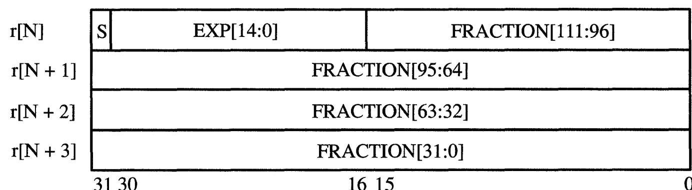


Figure 2–16. Quad-Precision Data Organization in Registers

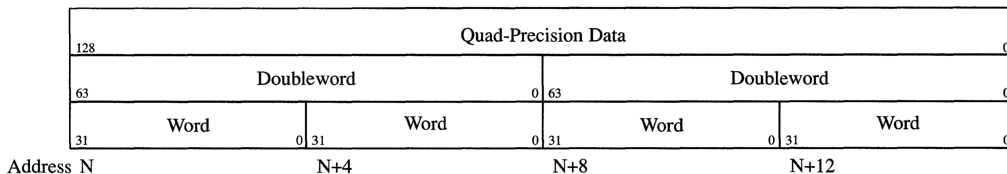


Figure 2–17. Quad-Precision Data Organization in Memory

2.4 SPARC Instruction Set

This section describes the SPARC instruction set as defined by the SPARC architecture. Included are subsections on instruction formats, addressing, instruction types, and an op code summary. *Chapter 12, SPARC Instruction Set*, contains a description of the assembly language syntax and a complete set of instruction definitions.

2.4.1 Instruction Formats

There are only three basic instruction formats plus three subformats. Format 1 is used for the CALL instruction, format 2 for the SETHI and Branch instructions, and format 3 for the remaining integer and

floating-point/coprocessor instructions. *Figure 2–18* shows each format with its fields, bit positions, and the instructions that use that format. All instructions are one word long and aligned on word boundaries in memory. For most instructions, operands are located in source registers (represented by *rs1* and *rs2*). The remaining instructions use one source register plus a displacement or immediate operand contained within the instruction itself.

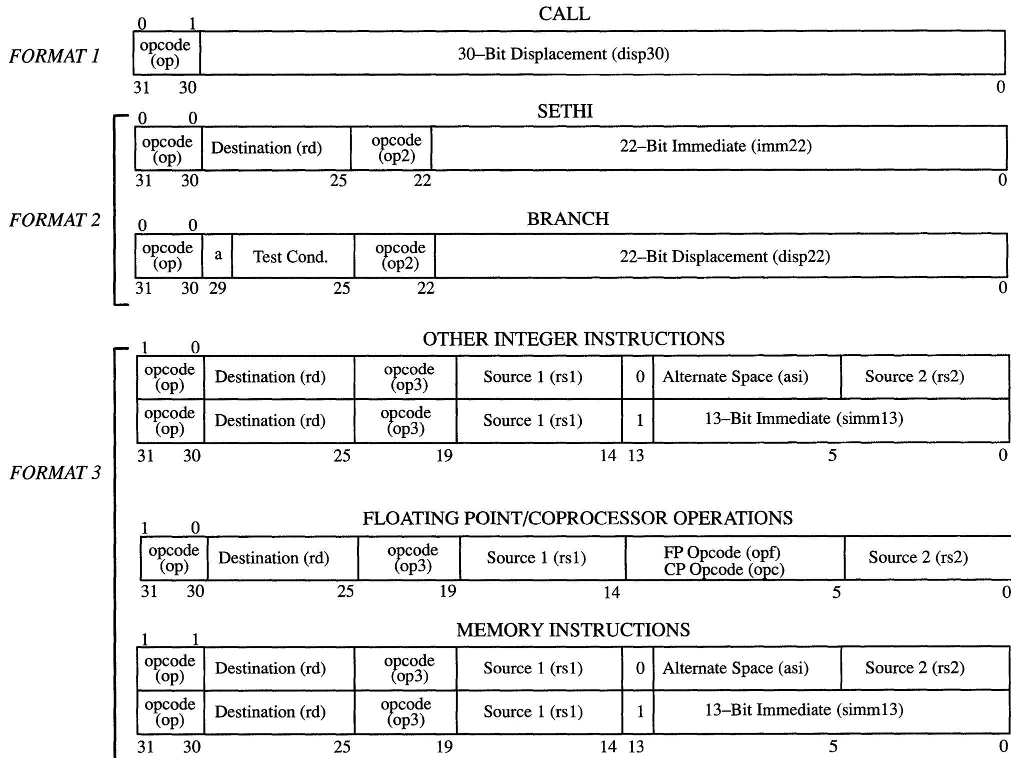


Figure 2–18. Instruction Format Summary

a The *a* (annul) bit is used in Branch instructions to control the execution of the delay instruction that immediately follows a control transfer instruction (see *Section 2.4.3.4.3*).

asi The address space identifier (ASI) is an eight-bit field used in Load/Store alternate instructions. Refer to *Section 2.4.2.6*.

cond This field identifies the condition code used for a Branch instruction.

disp22 This field contains the 22-bit displacement value used for PC-relative addressing for a taken Branch. It is sign extended to full-word size when used.

disp30 This field contains the 30-bit displacement used for the PC-relative addressing of a CALL instruction.

i The *i* (immediate) bit determines whether the second ALU operand (for non-FPop instructions) will be r[rs2] (i = 0), or a sign-extended *simm13* (i = 1).

imm22 This field contains the 22-bit constant used by the SETHI instruction.

op The *op* field selects the instruction format as shown in *Table 2–5*.

op2 The *op2* field (*Table 2–6*) contains the instruction opcode for format 2 instructions (op=0).

op3 The 6-bit *op3* field contains the instruction opcode for a format 3 instruction (op = 2 or 3).

opc The 9-bit *opc* identifies a coprocessor-operate (CPop) instruction. The relationship between the *opc* field and CPop instructions is described in *Section 2.4.3.6.*

opf The 9-bit *opf* identifies a floating-point-operate (FPop) instruction. The relationship between the *opf* field and FPop instructions is described in *Section 2.4.3.6.*

rd The *r*-register (or *r*-register pair) or *f*-register (or *f*-register pair) specified in the *rd* field serves as the source during store instructions. For all other instructions, the identified register (register pair) serves as the destination. Note that r[0] as a source supplies the value 0, and as a destination causes the result to be discarded. Note that *rd* must be an *r*-register for integer instructions and must be an *f*-register for floating-point instructions.

rs1 The 5-bit *rs1* field identifies the register containing the first source operand. The source is an *r*-register for integer instructions, an *f*-register for floating-point instructions, or a *c* register for coprocessor instructions.

rs2 The 5-bit *rs2* field identifies the register containing the second source operand. The source is an *r*-register for integer instructions, an *f*-register for floating-point instructions, or a *c* register for coprocessor instructions.

simm13 This field holds the 13-bit immediate value used as the second ALU operand when i = 1. It is sign-extended to full-word size when used.

Table 2–5. op field Coding

op Value	Instruction
00	Bicc, FBfcc,CBccc, SETHI
01	CALL
10	Integer/FP
11	Memory

Table 2–6. op2 Field Coding

op2 Value	Instruction
000	UNIMPlemented
010	Bicc
100	SETHI
110	FBfcc
111	CBccc

Unused (reserved) bit patterns which are used in the *op, op2, op3,* or *i* (wrong bit used) fields of instructions will cause an illegal_instruction trap. Fields that are not used for a particular instruction are ignored and so will not cause a trap, regardless of the bit pattern placed in that field. Unused or reserved bit patterns used in the *opf* or *opc* fields of a floating-point or coprocessor instruction cause an fp exception or a cp exception.

2.4.2 Addressing

SPARC supports four address modes: two register, register plus 13-bit immediate, 13-bit immediate, and program-counter relative. Memory address generation is done only for load and store instructions and is byte

oriented. Program counter-relative addressing is generated only for calls and branches and is word-boundary oriented because it is addressing instructions. Register-indirect addressing applies to jumps, returns, and traps and is also word-boundary oriented. Address generation is illustrated in *Figure 2–19*.

2.4.2.1 Two Register

Two-register addressing uses the *rs1* and *rs2* fields (instruction format 3) to specify two source registers whose 32-bit contents are added together to create a memory address. This is a Load/Store (or register-indirect) addressing mode.

2.4.2.2 Register Plus 13-Bit Immediate

This addressing mode is used where an immediate value is required as one of the sources. The address is generated by adding the 32-bit source register specified by *rs1* (format 3) to a 13-bit, sign-extended immediate value contained in the instruction. This is a Load/Store (or register-indirect) addressing mode.

2.4.2.3 13-Bit Immediate

Immediate addressing is a special case of register-plus-immediate addressing. In this case, the *rs1*-specified register is r[0] (whose value is 0), which means the address is generated using only the 13-bit immediate value. Use of this special case allows absolute addressing of the upper and lower 4 Kbytes of a memory (or instruction) space with the 13-bit immediate value. Immediate addressing is the simplest method of addressing because no registers need be set up beforehand.

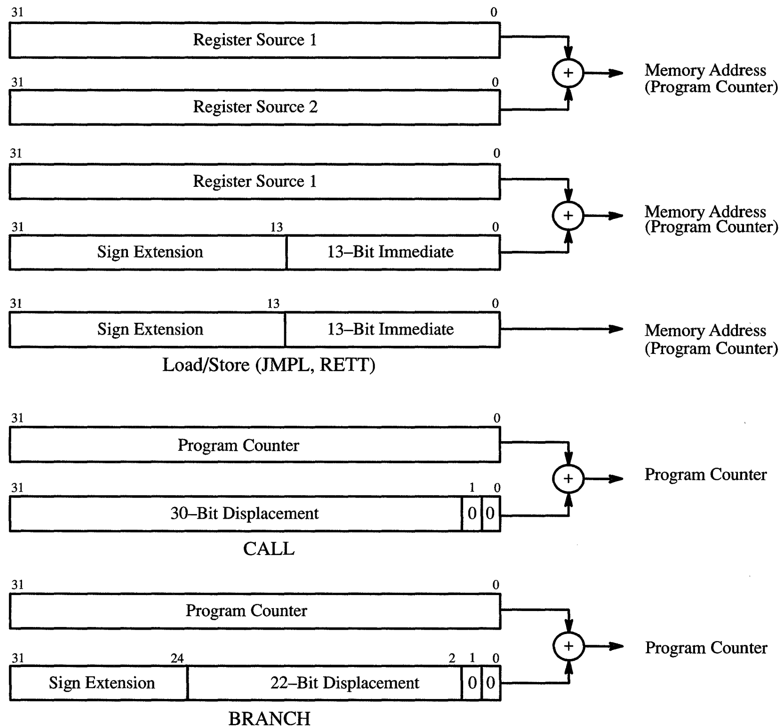


Figure 2–19. Address Generation

2.4.2.4 CALL

Address generation for the CALL instruction is program counter-relative, that is, the target address is based on the program counter (PC). The PC refers to the PC of the CALL instruction; the PC is not replaced with the nPC (SPARC is a delayed-control-transfer architecture, see *Section 2.4.3.4*) until the effective address is calculated with the CALL instruction's PC (see *Figure 2–19*).

An address is generated by adding this PC value to the 30-bit word displacement contained in the CALL instruction. The displacement is formed by appending two zeros to the 30-bit value from the instruction. This allows control transfers to any word-boundary location in the virtual memory instruction space. The result of the address generation becomes the new nPC.

2.4.2.5 Branch

Branch instructions also use PC-relative addressing, but in this case, the value added to the PC is a sign-extended 22-bit word displacement. Again, the displacement is formed by appending two zeros to the 22-bit value contained in the Branch instruction and then sign extending out to 32 bits. This allows a branching range of 8 Mbytes on word boundaries. The generated address becomes the new nPC.

Table 2–7. Standard SPARC ASI Assignments

ASI	Address Space
0x08	User Instruction
0x0A	User Data
0x09	Supervisor Instruction
0x0B	Supervisor Data

2.4.2.6 ASI

In addition to the 32 bits of address output by the processor, an additional eight bits of address space identifier (ASI) is sent to system memory (by means of the ASI(7:0) bus) during a memory access. These ASI bits define 256 alternate 32-bit address spaces, which may or may not overlap depending upon the designer's implementation.

The SPARC architecture defines four ASI values for user instructions, user data, supervisor instructions, and supervisor data (see *Table 2–7*). The ASI value is supplied on ASI external signals for each instruction Fetch and each data access encountered. These four ASI values all map to the same 32-bit address space, and are used to implement access-level protection. ASI values are commonly used to identify user/supervisor accesses, to identify special protected memory accesses such as boot PROM, and to access resources such as Cache controller/MMU (CMU) control registers, TLB entries, cache tag entries, etc.. Alternate ASIs (those other than the standard ASIs listed in *Table 2–7*) can be asserted by the use of alternate ASI load and store instructions (refer to *Section 2.4.3.1.1* or to *Chapter 12, SPARC Instruction Set*).

ROSS SPARC assigns a number of these ASI values to the CMU, and others are reserved for future assignment. Nevertheless, nearly 80 are left unassigned for use by the system. Refer to *Section 4.9* (RT625) or *Section 8.8* (CY7C604/605) for ASI assignments reserved for ROSS SPARC CMUs.

2.4.3 Instruction Types

SPARC instructions fall into six functional categories: Load/Store, arithmetic/logical/shift, control transfer, read/write control register, floating-point-operate/coprocessor-operate, and miscellaneous. For complete information on each instruction, see *Chapter 12*.

2.4.3.1 Load and Store

Load and store instructions (see *Table 2–8*) move bytes, halfwords, words, and doublewords between the byte-addressable main memory and a register in either the IU, FPU, or CP. They are the only instructions that access data memory.

Load and store instructions use two-register, register-plus-immediate, and immediate addressing modes. In addition to the 32-bit address, SPARC also generates an eight-bit address space identifier.

2.4.3.1.1 ASI

The address space identifier is used by the external system to ascertain which of the 256 available address spaces to access for the load or store being executed. Access to these alternate spaces can be gained directly by using the "load from alternate space" and "store to alternate space" instructions. These instructions use two-register addressing and the *asi* field in instruction format 3. The address space specified in the *asi* field

overrides the automatic ASI assignment made by the processor, giving access to such resources as system control registers that are invisible to the user. Because the ASI is intended for use by the system operating software, the alternate space instructions are privileged and can only be executed in supervisor mode.

Table 2–8. Load and Store Instructions

Name		Operation	
LDSB	(LDSBA*)	Load Signed Byte	(from Alternate Space)
LDSH	(LDSHA*)	Load Signed Halfword	(from Alternate Space)
LDUB	(LDUBA*)	Load Unsigned Byte	(from Alternate Space)
LDUH	(LDUHA*)	Load Unsigned Halfword	(from Alternate Space)
LD	(LDA*)	Load Word	(from Alternate Space)
LDD	(LDDA*)	Load Doubleword	(from Alternate Space)
LDF		Load Floating-Point	
LDDF		Load Double Floating-Point	
LDFSR		Load Floating-Point Status	
LDC		Load Coprocessor	
LDDC		Load Double Coprocessor	
LDCSR		Load Coprocessor Status Register	
STB	(STBA*)	Store Byte	(into Alternate Space)
STH	(STHA*)	Store Halfword	(into Alternate Space)
ST	(STA*)	Store Word	(into Alternate Space)
STD	(STDA*)	Store Doubleword	(into Alternate Space)
STF		Store Floating-Point	
STDF		Store Double Floating-Point	
STFSR		Store Floating-Point Status Register	
STDFQ*		Store Double Floating-Point Queue	
STC		Store Coprocessor	
STDC		Store Double Coprocessor	
STCSR		Store Coprocessor State Register	
STDCQ*		Store Double Coprocessor Queue	
LDSTUB	(LDSTUBA*)	Atomic Load-Store Unsigned Byte	(in Alternate Space)
SWAP	(SWAPA*)	Swap *r*-register with Memory	(in Alternate Space)

* denotes supervisor instruction

2.4.3.1.2 Multiprocessing Instructions

In addition to alternate address spaces, SPARC provides two uninterrupible instructions, SWAP and atomic Load-Store unsigned byte (LDSTUB), to support tightly coupled multiprocessing.

The SWAP instruction exchanges the contents of an *r*-register with a word from a memory location without allowing asynchronous traps or other memory accesses during the exchange.

The LDSTUB instruction reads a byte from memory into an *r*-register and then overwrites the memory byte to all ones. As with SWAP, LDSTUB prevents asynchronous traps and other memory accesses during its execution. LDSTUB is used to construct semaphores.

Multiple processors attempting to simultaneously execute SWAP or LDSTUB to the same memory location are guaranteed that the competing instructions will execute in serial order.

2.4.3.2 *Arithmetic/Logical/Shift*

This class of instructions performs a computation on two source operands and writes the result into a destination register (r[rd]). One of the source operands is always a register, r[rs1], and the other depends on the state of the instruction's "i" (immediate) bit. If i = 0, the second operand is register r[rs2]. If i = 1, the operand is the 13-bit, sign-extended constant in the instruction's *simm13* field. SETHI is a special case because it is a single-operand instruction.

Table 2–9. Arithmetic/Logical/Shift Instructions

Name		Operation	
ADD	(ADDcc)	Add	(and modify icc)
ADDX	(ADDXcc)	Add with Carry	(and modify icc)
TADDcc	(TADDccTV)	Tagged Add and modiify icc	(and Trap on oVerflow)
SUB	(SUBcc)	Subtract	(and modify icc)
SUBX	(SUBXcc)	Subtract with Carry	(and modify icc)
TSUBcc	(TSUBccTV)	Tagged Subtract and modify icc	(and Trap on oVerflow)
MULScc		Multiply Step and modify icc	
UMUL	(UMULcc)	Unsigned Integer Multiply	(and modify icc)
SMUL	(SMULcc)	Signed Integer Multiply	(and modify icc)
UDIV	(UDIVcc)	Unsigned Integer Division	(and modify icc)
SDIV	(SDIVcc)	Signed Integer Division	(and modify icc)
AND	(ANDcc)	And	(and modify icc)
ANDN	(ANDNcc)	And Not	(and modify icc)
OR	(ORcc)	Inclusive Or	(and modify icc)
ORN	(ORNcc)	Inclusive Or Not	(and modify icc)
XOR	(XORcc)	Exclusive Or	(and modify icc)
XNOR	(XNORcc)	Exclusive Nor	(and modify icc)
SLL		Shift Left Logical	
SRL		Shift Right Logical	
SRA		Shift Right Arithmetic	
SETHI		Set High 22 Bits of *r*-register	

For most arithmetic and logical instructions, there is both a version that modifies the integer condition codes and one that does not (see *Table 2–9*).

Shift instructions shift left or right by a distance specified in either a register or an immediate value in the instruction.

The multiply step instruction, MULScc, is used to generate the signed or unsigned 64-bit product of two 32-bit integers. For more information on MULScc, refer to its definition in *Chapter 6*.

2.4.3.2.1 Register r[0]

Because register r[0] reads as a 0 and discards any result written to it as a destination, it can be used with some instructions to create syntactically familiar pseudoinstructions. For example, an integer COMPARE instruction is created using the SUBcc (subtract and set condition codes) with r[0] as its destination. A TEST instruction uses SUBcc with r[0] as both the destination and one of the sources. A register-to-register MOVE

is accomplished using an ADD or OR instruction with r[0] as one of the source registers. A negation is done with SUB and r[0] as one source. If the assembler being used supports psuedoinstructions, it translates the psuedoinstruction into the equivalent instruction in the native assembly language. Refer to your assembly language manual for details.

2.4.3.2.2 SETHI

SETHI is a special instruction that can be combined with another arithmetic instruction (such as an OR immediate) to construct a 32-bit constant. SETHI loads a 22-bit immediate value into the upper 22 bits of the destination register and clears the lower 10 bits. The arithmetic immediate instruction which follows is used to load the lower 10 bits. Note that the 13-bit immediate value gives a 3 bit overlap with the 22-bit SETHI value. SETHI can also be combined with a load or store instruction to construct a 32-bit memory address.

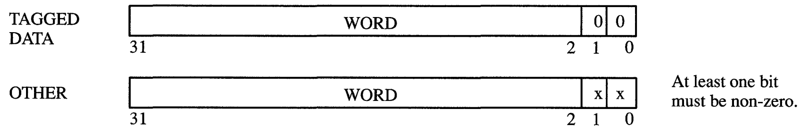


Figure 2–20. Tagged Data Example

2.4.3.2.3 Tagged Arithmetic

The tagged arithmetic instructions are useful for languages that employ tags, such as LISP, Smalltalk, or Prolog. For efficient support of such languages, the SPARC architecture defines tagged data as a data type. Tagged data are assumed to be 30 bits wide with the tag bits (the least two significant bits) set to zero (see *Figure 2–20*). A tagged add (TADDcc) or subtract (TSUBcc) will set the overflow bit if either of the operands has a nonzero tag or if a normal overflow occurs.

Tagged add or subtract instructions are normally followed by a conditional Branch. If the overflow bit is set during a tagged add or subtract operation, control is commonly transferred to a routine that checks the operand types. In order to expedite this software construct, the SPARC architecture provides two trap on overflow instructions: TADDccTV and TSUBccTV, which automatically trap if the overflow bit is set during their execution.

2.4.3.3 Control Transfer

Control transfer instructions are those that change the values of the PC and nPC. These include conditional Branches (Bicc, FBfcc, CBccc), a call (CALL), a jump (JMPL), conditional traps (Ticc), and a return from trap (RETT). Also included are the SAVE and RESTORE instructions, which don't transfer control but are used to save or restore windows during a call to a new procedure or a return to a calling procedure (see *Table 2–10*).

For SPARC processors, control transfer is usually delayed so that the instruction immediately following the control-transfer instruction (called the delay instruction) can be executed before control transfers to the target address. The delay instruction is always fetched. However, the annul or *a* bit in conditional Branch instructions can cause the instruction to be annulled (i.e., prevent execution) if the Branch is not taken (or always annulled in the case of BA, FBA, and CBA). If a Branch is taken, the delay instruction is always executed (except for BA, FBA, and CBA, see *Section 2.4.3.4.3*). *Table 2–11* shows the characteristics of each control transfer type.

Program Counter Relative

PC-relative addressing computes the target address by adding a displacement to the program counter. See *Section 2.4.2.*

Register-Indirect

Register-indirect addressing computes the target address as either r[rs1] + r[rs2] if i = 0, or r[rs1] + *simm13* if i = 1. See *Section 2.4.2.*

Delayed

A control-transfer instruction is delayed if it transfers control to the target address after a one-instruction delay. See *Section 2.4.3.4.*

Annul Bit

In an instruction with an annul bit, the delay instruction that follows may be annulled. See *Section 2.4.3.4.3.*

2.4.3.3.1 Branching and the Condition Codes

The condition code bits in the *icc*, *fcc*, and *ccc* fields, are located (respectively) in the processor state register (PSR), floating-point state register (FSR), and coprocessor state register (CSR). The integer condition code bits are modified by arithmetic and logical instructions whose names end with the letters *cc*, or they may be written directly with WRPSR. The floating-point condition codes are modified by the floating-point compare instructions, FCMP and FCMPE, or directly with the STFSR instruction. Modification of the coprocessor condition codes is done directly with STCSR or by operations defined by the particular coprocessor implementation.

Except for Branch Always (BA) and Branch Never (BN), a Bicc instruction evaluates the integer condition codes as specified in the *cond* field. If the tested condition evaluates as true, the branch is taken, causing a PC-relative delayed transfer to the address [(PC + 4) + sign extnd(disp22)]. If the evaluation result is false, the branch is not taken. For BA and BN, there is no evaluation; the result is simply forced to true for BA and false for BN.

Table 2–10. Control Transfer Instructions

Name	Operation
SAVE	SAVE caller's window
RESTORE	RESTORE caller's window
Bicc	Branch on integer condition codes
FBfcc	Branch on floating-point condition codes
CBccc	Branch on coprocessor condition codes
CALL	Call
JMPL	Jump and Link
RETT	Return from Trap
Ticc	Trap on integer condition codes

Table 2–11. Control Transfer Instruction Characteristics

Instructions	Addressing Mode	Delayed	Annul Bit
Conditional Branch	Program Counter Relative	yes	yes
CALL	Program Counter Relative	yes	yes
JUMP	Register Indirect	yes	no
Return	Register Indirect	yes	no
Trap	Register Indirect	no	no

If the Branch is not taken, then the annul bit is checked. If the "a" bit is set, the delay instruction is annulled. If "a" is not set, the delay instruction is executed. If the Branch is taken, the annul bit is ignored and the delay instruction is executed. For more information on delayed control transfer and the annul bit, see *Section 2.4.3.4.*

If its annul field is 0, a BN instruction acts like an NOP. If its annul field is 1, the following (delay) instruction is annulled (not executed). In neither case does a transfer of control take place.

BA, on the other hand, always branches, so the annul bit would normally be ignored. But for BA, FBA, and CBA, the effect of the annul bit is changed. See *Section 2.4.3.4.3* for details.

As illustrated in *Table 2–12*, Bicc and Ticc instructions test for the same conditions and use the same *cond* field codes during their evaluations. The FPfcc instruction operates in the same way as a Bicc, except it tests floating-point condition codes. A CBccc instruction behaves in the same manner as an FBfcc, except it tests the CCC<1:0> signals supplied by the coprocessor (see *Table 2–14*). Both FBN and CBN behave in the same way as BN. Note that all coprocessor instructions cause an unimplemented trap on the RT620.

2.4.3.3.2 Trap Instructions

The "Trap on integer condition codes" (Ticc) instruction evaluates the condition codes specified by its *cond* (condition) field. If the result is true, a trap is immediately taken (no delay instruction). If the condition codes evaluate to false, Ticc executes as a NOP. Once the Ticc is taken, it identifies which software trap type caused it by writing its trap number + 128 (the offset for trap instructions) into the *tt* field of the Trap Base Register (TBR), as illustrated in *Figure 2–21.*

The trap number is the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign extnd(simm13)" if the *i* field is one. The processor then disables traps (ET=0), saves the state of S into PS, decrements the CWP, saves PC and nPC into the *locals* r[17] and r[18] (respectively) of the new window, enters supervisor mode (S=1), and writes the trap base register to the PC and TBR + 4 to nPC.

Ticc can be used to implement kernel calls, breakpointing, and tracing. It can also be used for run-time checks, such as out-of-range array indices, integer overflow, etc.

Return from a trap is accomplished using the delayed control transfer couple, JMPL, RETT. RETT first increments the CWP by one, calculates the return address (using register-indirect addressing), and then checks for a number of trap conditions before it allows a return. An illegal_instruction trap is generated if traps are enabled (ET=1) when RETT is executed. If ET=0, RETT checks for other trap conditions and generates a reset trap and enters error mode for the following conditions: if S=0; if the new CWP would cause a window underflow; or if the return address is not word aligned. If none of these conditions exists, RETT enables traps (ET=1), restores the previous supervisor state to the S bit, and writes the target address into the nPC.

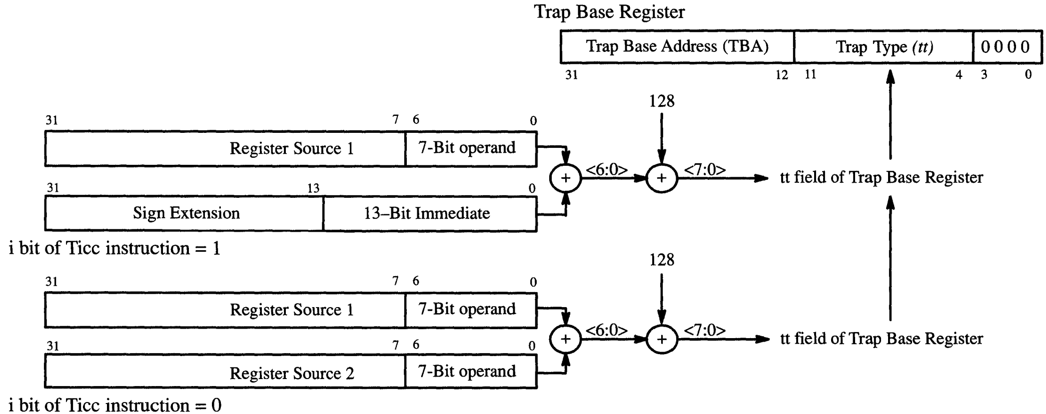


Figure 2–21. Ticc Trap Address Generation

Table 2–12. Bicc and Ticc Condition Codes

Cond.	Test	Cond.	Test
0000	Never	1000	Always
0001	Equal to	1001	Not equal to
0010	Less than or equal	1010	Greater than
0011	Less than	1011	Greater than or equal to
0100	Less than or equal to, unsigned	1100	Greater than, unsigned
0101	Carry set (less than, unsigned)	1101	Carry clear (greater than or equal to, unsigned)
0110	Negative	1110	Positive
0111	Overflow set	1111	Overflow clear

Table 2–13. FBfcc Condition Codes

Cond.	Test	Cond.	Test
0000	Never	1000	Always
0001	Not equal to	1001	Equal to
0010	Less than or greater than	1010	Unordered or equal to
0011	Unordered or less than	1011	Greater than or equal to
0100	Less than	1100	Unordered or greater than or equal to
0101	Unordered or greater than	1101	Less than or equal to
0110	Greater than	1110	Unordered or less than or equal to
0111	Unordered	1111	Ordered

Table 2–14. CBccc Condition Codes

Opcode	Cond.	CCC[1:0] Test	Opcode	Cond.	CCC[1:0] Test
CBN	0000	Never	CBA	1000	Always
CB123	0001	1 or 2 or 3	CB0	1001	0
CB12	0010	1 or 2	CB03	1010	0 or 3
CB13	0011	1 or 3	CB02	1011	0 or 2
CB1	0100	1	CB023	1100	0 or 2 or 3
CB23	0101	2 or 3	CB01	1101	0 or 1
CB2	0110	2	CB013	1110	0 or 1 or 3
CB3	0111	3	CB012	1111	0 or 1 or 2

2.4.3.3.3 Calls and Returns

Calling a subroutine or procedure can be done in one of two ways. A CALL instruction computes its target address using a PC-relative displacement of 30-bits, or the Jump and link (JMPL) instruction uses register-indirect addressing (the sum of two registers or the sum of a register and a 13-bit signed immediate value) to compute its target address. Either instruction allows control transfer to any arbitrary instruction address.

Control transfer to a procedure that requires its own register window is done with either a CALL or JMPL instruction and a SAVE instruction. A procedure that does not need a new window, a so-called "leaf" routine, is invoked with only the CALL or JMPL.

The CALL instruction stores its return address (the current PC) into *outs* register r[15]. When the new window is activated, this becomes *ins* register r[31] (see *Figure 2–2*). The JMPL instruction stores its return address (the contents of PC, which is the link) into the *r*-register specified in the destination field, *rd*.

The primary purpose of the SAVE instruction is to "save" the caller's window by decrementing the Current Window Pointer (CWP) by one, thereby activating the next window and making the current window into the previous window. SAVE also performs a normal ADD, using source registers from the caller's window, but writing the result into a destination register in the new window. This can be used to set a new stack pointer from the previous one (see *Section 2.1.2.1*).

Return from a procedure requiring its own window is done with a RESTORE and a JMPL instruction. A leaf procedure returns by executing a JMPL only. The target address for the return is normally that of the instruction following the CALL's or JMPL's delay instruction; that is, the return address + 8. The RESTORE instruction restores the caller's window by incrementing the CWP by one, causing the previous window to become the current window. As with SAVE, RESTORE performs an ADD using source registers from the called (new) window and writing the result into the calling (previous) window.

Both SAVE and RESTORE compare the new CWP against the window invalid mask (WIM) to check for window overflow or underflow. They may also be used to atomically change the CWP while establishing a new memory stack pointer in an *r*-register.

2.4.3.4 Delayed Control Transfer

Traditional architectures usually execute the target instruction of a control transfer immediately after the control transfer instruction. However, in a pipelined RISC architecture, this type of transfer requires flushing the instruction that follows the control transfer instruction. To avoid creating a hole or bubble in the pipeline, SPARC processors delay execution of the target instruction until the instruction following the control transfer instruction is executed. The instruction in this delay slot is called the delay instruction.

Table 2–15. Delayed Control Transfer Instruction Example

PC	nPC	Instruction
8	12	Non-control transfer
12	16	Control transfer (target = 40)
16	40	Non-control transfer (delay instruction) (Transfers control to 40)
40	44	...

Table 2–16. Effect of Annul Bit Reset (*a*=0)

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (*a*=0) 40	Not Taken
16	20	Delay slot instruction	Executed
20	24	...	Executed

Table 2–17. Effect of Annul Bit Set (*a*=1)

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (*a*=1) 40	Not Taken
16	20	Delay slot inst. (annulled)	Not Executed
20	24	...	Executed

2.4.3.4.1 PC and nPC

The program counter (PC) contains the address of the instruction currently being executed by the SPARC processor, and the next program counter (nPC) holds the address (PC + 4) of the next instruction to be executed (assuming a control transfer or a trap does not occur).

Most instructions end by copying the contents of the nPC into the PC and then they either increment nPC by four or write a computed control transfer target address into nPC. At this point, the PC points to the instruction that is about to begin execution and the nPC points to the instruction that will be executed after that, i.e., the second instruction after the currently executing instruction. It is the existence of the nPC that allows the execution of the delay instruction before transfer of control to the target instruction.

2.4.3.4.2 Delay Instruction

The instruction pointed to by the nPC when the PC is pointing to a delayed-control-transfer instruction is called the delay instruction. Normally, this is the next sequential instruction in the code stream. However, if the instruction that preceded the delayed control transfer was itself a delayed control transfer, the target of the preceding control transfer becomes the delay instruction (where the nPC will point). For more on delayed control transfer couples, see *Section 2.4.3.4.4*.

Table 2–15 shows the order of execution for a simple (not back-to-back) delayed control transfer. The order of execution is 8, 12, 16, 40. If the delayed-control-transfer instruction is not taken, the order would becomes 8, 12, 16, 20.

2.4.3.4.3 Annul Bit

The *a* (annul) bit is only available on conditional Branch instructions (Bicc, FBfcc, and CBccc), where it changes the behavior of the delay instruction. If *a* is set on a conditional Branch instruction (except BA, FBA, and CBA) and the Branch is not taken, the delay instruction is annulled (not executed). An annulled instruction does not affect the state of the SPARC processor, and does not allow a trap to occur during an annulled instruction. If the Branch is taken, the *a* bit is ignored and the delay instruction is executed. *Table 2–16* and *Table 2–17* show the effect of the annul bit when it is reset or set.

The "Branch Always" instructions (BA, FBA, and CBA) are a special case. If the *a* bit is set in these instructions, the delay instruction is annulled, even though the Branch is taken. Effectively, this gives a "traditional"

non-delayed Branch. When a = 0 in a "Branch Always" instruction, it behaves the same as any other conditional Branch; the delay instruction is executed. *Figure 2–22* displays the effect the *a* bit has on any Branch for either the set or reset state. *Table 2–18* summarizes the effect the annul bit has on the execution of delay instructions.

Table 2–18. Effect of Annul Bit on Delay Instruction

a bit	Type of Branch	Delay instruction executed?
a = 1	Always	No
	Conditional, taken	Yes
	Conditional, not taken	No
a = 0	Always	Yes
	Conditional, taken	Yes
	Conditional, not taken	Yes

2.4.3.4.4 Delayed Control Transfer Couples

The occurrence of two back-to-back, delayed control transfer instructions is called a delayed control transfer couple, which the processor handles differently from a simple control transfer. An instruction sequence containing a delayed control transfer couple is shown in *Table 2–19*, and the order of execution for the six different cases of back-to-back, delayed control transfer instructions is shown in *Table 2–20.*

The delay slot instruction for a delayed control transfer instruction is the instruction fetched after the delayed control transfer instruction. For most cases, this instruction is located immediately in the code listing after the delayed control transfer instruction. However, in the case of a delayed control transfer couple, the target instruction of the first delayed control transfer instruction is the delay slot instruction for the second delayed control transfer instruction, since that target instruction is the next instruction to be fetched. The delay slot instruction for the second delayed control transfer instruction is the next instruction loaded into the instruction pipeline after the second delayed control transfer instruction.

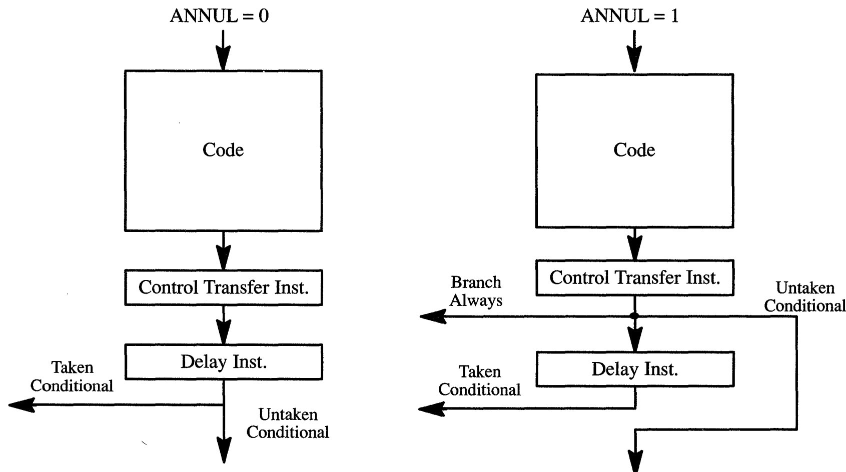


Figure 2–22. Delayed Control Transfer

In the following tables, "delayed control transfer instruction" is abbreviated to "DCTI." A "Non-DCTI" may be either a non-control transfer instruction or a control transfer that is not delayed (i.e., a Ticc). Where the annul bit is not indicated, it may be either 0 or 1.

Case 1 of *Table 2–20* includes the "JMPL, RETT" couple, which is the normal method of returning from a trap handler. The JMPL, RETT couple ensures correct values of PC and nPC are restored upon exiting the trap routine, even in the case of a trap caused by a delay slot instruction (see *Section 2.4.3.4.2*). The case of a trap caused by a delay slot instruction is one where the nPC will not be PC + 4, thus requiring both PC and nPC to be restored. The JMPL, RETT couple allows the choice of re-executing the trapped instruction or executing the instruction following the trap occurrence. Refer to the RETT instruction description in *Chapter 12* for further information.

Table 2–19. Delayed Control Transfer Couple Instruction Sequence

Address	Instruction	Target
8:	Non DCTI	
12	DCTI	4
16	DCTI	60
20	Non DCTI	
24	...	
...	...	
40	Non DCTI	
44	...	
...	...	
60	Non DCTI	
64	...	
...	...	

Table 2–20. Execution of Delayed Control Transfer Couples

Case	DCTI at Location 12	DCTI at Location 16	Order of Execution
1	DCTI Unconditional	DCTI Taken	12,16,40,60,64...
2	DCTI Unconditional	B*cc(a=0) Untaken	12,16,40,44...
3	DCTI Unconditional	B*cc(a=1) Untaken	12,16,44,48...(40 annulled)
4	DCTI Unconditional	B*A(a=1)	12,16,60,64...(40 annulled)
5	B*A(a=1)	any CTI	12,40,44...(16 annulled)
6	B*cc	DCTI	Not Supported

Definitions:
B*A .... BA,FBA, or CBA
B*cc .... Bicc,FBicc, or CBicc (except B*A)
DCTI Unconditional ..... CALL,JMPL,RETT, or B*A(a=0)
DCTI Taken .... CALL,JMPL,RETT,B*cc taken, or B*A(a=0)

Cases 1–5 described in *Table 2–20* are illustrated in *Figure 2–23*. In case 1, the first DCTI is fetched at address 12 and the target address is calculated while the delay slot instruction is fetched. The delay slot instruction for the first DCTI (located at address 16) is another DCTI, which also has a delay slot. The target address of the first DCTI has been calculated by the time the first delay slot instruction has been fetched, and the target instruction is fetched at address 40. The target instruction is the instruction located in the instruction pipeline after the second DCTI, and therefore it is the delay slot instruction for the second DCTI. The target instruction for the second DCTI (address 60) is fetched after the delay slot instruction for the second DCTI (which is also the target address for the first DCTI) has been fetched.

Case 2 differs from case 1 in that the second DCTI is conditional, and is not taken. In case 2, the instruction at address 40 (target for DCTI #1) is the delay slot instruction for the second DCTI. Since the second DCTI does not cause a Branch, the instruction fetch continues to address 44.

Case 3 is an interesting case in which the target instruction of the first DCTI is annulled by the second DCTI. This causes the instruction at address 40 to be annulled. Since the second DCTI is an untaken conditional Branch, instruction Fetch continues after the annulled target instruction (address 44).

Case 4 illustrates a DCTI followed by a Branch Always instruction with the annul bit set. This causes the target instruction of the first DCTI (address 40) to be annulled, and program control is transferred to the target of the second DCTI at address 60.

Case 5 illustrates the case where the second DCTI is annulled by the annul bit of the first DCTI. The second DCTI, since it is annulled, has no effect on instruction Fetch. This case is identical to the case of any other annulled delay slot instruction.

When the first instruction of a delayed control transfer couple is a conditional Branch, control transfer is undefined (case 6). If such a couple is executed, the location where execution continues is within the same address space but is otherwise undefined. Execution of this sequence does not change any other aspect of the processor state.

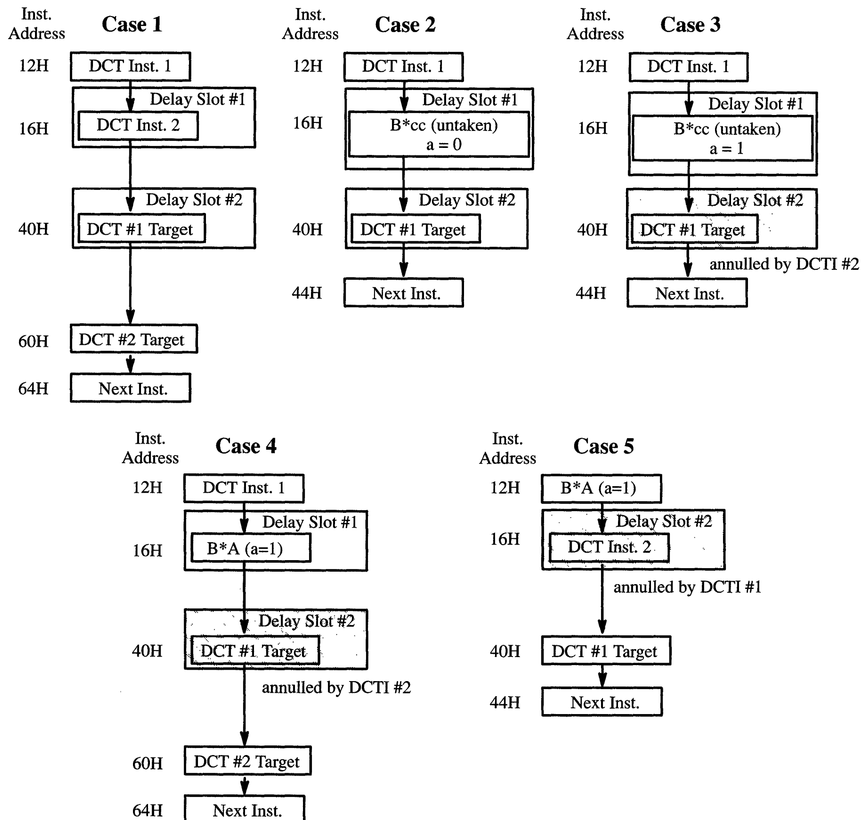


Figure 2–23. Delayed Control Transfer Couples

Table 2–21. Read/Write Control Register Instructions

Name	Operation	Cycles
RDY	Read Y Register	1
RDPSR*	Read Processor State Register	1
RDWIM*	Read Window Invalid Mask	1
RDTBR*	Read Trap Base Register	1
WRY	Write Y Register	1
WRPSR*	Write Processor State Register	1
WRWIM*	Write Window Invalid Mask	1
WRTBR*	Write Trap Base Register	1

* denotes supervisor instruction

Table 2–22. Floating-Point-Operate and Coprocessor-Operate Instructions

Name	Operation	Cycles
FPop	Floating-Point Operations	1 to launch
CPop	Coprocessor Operations	1 to launch

Table 2–23. Miscellaneaous Instructions

Name	Operation	Cycles
UNIMP	Unimplemented Instruction	1
FLUSH	Instruction Cache Flush	1

2.4.3.5 Read/Write Control Registers

This class of instruction reads or writes the contents of the various control registers (see *Table 2–21*). The source (read) or destination (write) is implied by the instruction name. Read/write instructions are provided for the PSR, WIM, TBR, FSR, CSR, and the Y register. Reads and writes to the PSR, WIM, and TBR are privileged and are available in supervisor mode only.

2.4.3.6 Floating-Point-Operate

Floating-point calculations are accomplished with floating-point-operate instructions (FPops), which are register-to-register instructions that compute some result as a function of one or two source operands (see *Table 2–22*). The result is always placed in a destination register (i.e., source operands are not overwritten). The source and destination registers for floating-point instructions must be *f*-registers. If the EF bit of the PSR is not set, executing a floating-point instruction will generate a fp disabled trap. An fp disabled trap is also generated if the CY7C602 FPU is not present in a CY7C601-based system.

Because the FPU instructions execute concurrently with the integer unit, when a floating-point exception occurs, the PC does contain the address of an FPop instruction, but not the one that caused the exception. However, the front entry of the floating-point queue contains the offending instruction and its address.

Floating-point Load/Store instructions are not operate instructions; they fall under the Load/Store instruction category (see *Section 2.4.3.1*).

2.4.3.7 Coprocessor-Operate (CY7C601 only)

For CY7C601 systems, the coprocessor-operate instructions (CPops) are executed by an attached coprocessor. Coprocessor instructions use the *c registers* located in the coprocessor's register file as source and

destination registers. If there is no attached coprocessor, or if the coprocessor interface is not supported as is the case with the RT620, attempted execution of a coprocessor instruction generates a cp disabled trap.

Coprocessor Load/Store instructions are not operate instructions; they fall under the Load/Store instruction category (see *Section 2.4.3.1*).

If the coprocessor executes instructions concurrently with the CY7C601, the architecture supports a coprocessor queue that functions in the same fashion as the floating-point queue.

2.4.3.8 FLUSH

The FLUSH instruction is used to flush a word from an internal instruction cache. The instruction is described in *Chapter 12, SPARC Instruction Set.* The RT620 provides an 8-Kbyte instruction cache, which is described in *Section 4.6.* Refer to *Section 3.6.4.3* for information on the effect of the FLUSH instruction on the RT620.

The CY7C601 does not incorporate an internal instruction cache, so FLUSH would normally execute as an NOP. However, if the CY7C601 is supported by an external instruction cache or buffer, FLUSH causes an illegal instruction trap if the $\overline{\text{IFT}}$ signal is LOW (refer to *Section 6.2.1.9* for information on the IFT signal for the CY7C601).

2.4.4 SPARC Instruction OP Codes

This section contains tables that give a complete list of the instruction opcodes, both by functional groups and in ascending numeric order.

2.4.4.1 Load and Store Instructions

Table 2–24. Load and Store Instruction Opcodes

	Opcodes with Format						
Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
LD	1 1	rd	000000	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDA	1 1	rd	010000	rs1	i =0	asi	rs2
LDC	1 1	rd	110000	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDCSR	1 1	rd	110001	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDD	1 1	rd	000011	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDDA	1 1	rd	010011	rs1	i =0	asi	rs2
LDDC	1 1	rd	110011	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDDF	1 1	rd	100011	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDF	1 1	rd	100000	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDFSR	1 1	rd	100001	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDSB	1 1	rd	001001	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDSBA	1 1	rd	011001	rs1	i =0	asi	rs2
LDSH	1 1	rd	001010	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDSHA	1 1	rd	011010	rs1	i =0	asi	rs2
LDSTUB	1 1	rd	001101	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDSTUBA	1 1	rd	011101	rs1	i =0	asi	rs2
LDUB	1 1	rd	000001	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDUBA	1 1	rd	010001	rs1	i =0	asi	rs2
LDUH	1 1	rd	000010	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDUHA	1 1	rd	010010	rs1	i =0	asi	rs2
ST	1 1	rd	000100	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
	Opcodes with Format						
STA	1 1	rd	0 1 0 1 0 0	rs1	i =0	asi	rs2
STB	1 1	rd	0 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STBA	1 1	rd	0 1 0 1 0 1	rs1	i =0	asi	rs2
STC	1 1	rd	1 1 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STCSR	1 1	rd	1 1 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STD	1 1	rd	0 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDA	1 1	rd	0 1 0 1 1 1	rs1	i =0	asi	rs2
STDC	1 1	rd	1 1 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDCQ	1 1	rd	1 1 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDF	1 1	rd	1 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDFQ	1 1	rd	1 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STF	1 1	rd	1 0 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STFSR	1 1	rd	1 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STH	1 1	rd	0 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STHA	1 1	rd	0 1 0 1 1 0	rs1	i =0	asi	rs2
SWAP	1 1	rd	0 0 1 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SWAPA	1 1	rd	0 1 1 1 1 1	rs1	i =0	asi	rs2

2.4.4.2 Arithmetic/Logical/Shift Instructions

Table 2–25. Arithmetic/Logical/Shift Instruction Opcodes

	Opcodes with Format						
Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
ADD	1 0	rd	0 0 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ADDcc	1 0	rd	0 1 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ADDX	1 0	rd	0 0 1 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ADDXcc	1 0	rd	0 1 1 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
AND	1 0	rd	0 0 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ANDcc	1 0	rd	0 1 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ANDN	1 0	rd	0 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ANDNcc	1 0	rd	0 1 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
MULScc	1 0	rd	1 0 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
OR	1 0	rd	0 0 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ORcc	1 0	rd	0 1 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ORN	1 0	rd	0 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ORNcc	1 0	rd	0 1 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SLL	1 0	rd	1 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt
SDIV	1 0	rd	0 0 1 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SDIVcc	1 0	rd	0 1 1 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SMUL	1 0	rd	0 0 1 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SMULcc	1 0	rd	0 1 1 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SRA	1 0	rd	1 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt
SRL	1 0	rd	1 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt

	Opcodes with Format						
Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
SUB	1 0	rd	0 0 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SUBcc	1 0	rd	0 1 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SUBX	1 0	rd	0 0 1 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SUBXcc	1 0	rd	0 1 1 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TADDcc	1 0	rd	1 0 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TADDccTV	1 0	rd	1 0 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TSUBcc	1 0	rd	1 0 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TSUBccTV	1 0	rd	1 0 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UDIV	1 0	rd	0 0 1 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UDIVcc	1 0	rd	0 1 1 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UMUL	1 0	rd	0 0 1 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UMULcc	1 0	rd	0 1 1 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XNOR	1 0	rd	0 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XNORcc	1 0	rd	0 1 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XOR	1 0	rd	0 0 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XORcc	1 0	rd	0 1 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

	31 30	29 25	24 22	21 0
SETHI	0 0	rd	1 0 0	imm22

2.4.4.3 Control Transfer Instructions

Table 2–26. Control Transfer Instruction Opcodes

Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
	Opcodes with Format						
JMPL	1 0	rd	1 1 1 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
RESTORE	1 0	rd	1 1 1 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
RETT	1 0	unused	1 1 1 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SAVE	1 0	rd	1 1 1 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

	31 30	29	28 25	24 22	21 0
Bicc	0 0	a	cond	0 1 0	disp22
CBccc	0 0	a	cond	1 1 1	disp22
FBfcc	0 0	a	cond	1 1 0	disp22

	31 30	29	28 25	24 19	18 14	13	12 5	4 0
Ticc	1 0	R*	cond	1 1 1 0 1 0	rs1	i =0	unused (zero)	rs2
						i =1	simm13	
CALL	0 1	disp30						

*R = reserved.

Table 2–27. Bicc and Ticc Condition Codes

Cond.	Test
0000	Never
0001	Equal to
0010	Less than or equal to
0011	Less than
0100	Less than or equal to, unsigned
0101	Carry set (less than, unsigned)
0110	Negative
0111	Overflow set
1000	Always
1001	Not equal to
1010	Greater than
1011	Greater than or equal to
1100	Greater than, unsigned
1101	Carry clear (greater than or equal, unsigned)
1110	Positive
1111	Overflow clear

Table 2–28. FBfcc Condition Codes

Cond.	Test
0000	Never
0001	Not equal
0010	Less than or greater than
0011	Unordered or less than
0100	Less than
0101	Unordered or greater than
0110	Greater than
0111	Unordered
1000	Always
1001	Equal
1010	Unordered or equal
1011	Greater than or equal
1100	Unordered or greater than or equal
1101	Less than or equal
1110	Unordered or less than or equal
1111	Ordered

Table 2–29. CBccc Condition Codes

Opcode	Cond.	CCC[1:0] Test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

2.4.4.4 Read/Write Control Register Instructions

Table 2–30. Read/Write Control Register Instruction Opcodes

Mnemonic	Opcodes with Format						
	31 30	29 25	24 19	18 14	13 0		
RDASR*	1 0	rd	1 0 1 0 0 0	rs1	unused (zero)		
RDPSR	1 0	rd	1 0 1 0 0 1	unused	unused (zero)		
RDTBR	1 0	rd	1 0 1 0 1 1	unused	unused (zero)		
RDWIM	1 0	rd	1 0 1 0 1 0	unused	unused (zero)		
	31 30	29 25	24 19	18 14	13	12 5	4 0
WRASR*	1 0	rd	1 1 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRPSR	1 0	reserved	1 1 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRTBR	1 0	reserved	1 1 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRWIM	1 0	reserved	1 1 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

* WRASR and RDASR are general case instructions that include the previous WRY and RDY instructions. WRY and RDY are indicated by setting rd = 0. Setting rd = 30 accesses the DIAG register; setting rd = 31 accesses the ICCR register (both supported by RT620 only).

2.4.4.5 *Floating-Point/Coprocessor Instructions*

Table 2–31. Floating-Point /Coprocessor Instruction Opcodes

	Opcodes with Format					
Mnemonic	31 30	29 25	24 19	18 14	13 5	4 0
CPop1	1 0	rd	1 1 0 1 1 0	rs1	OPC	rs2
CPop2	1 0	rd	1 1 0 1 1 1	rs1	OPC	rs2
FABSs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 1 0 0 1	rs2
FADDd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 0	rs2
FADDq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 1	rs2
FADDs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 0 1	rs2
FCMPd	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 0	rs2
FCMPq	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 1	rs2
FCMPs	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 0 1	rs2
FCMPEd	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 0	rs2
FCMPEq	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 1	rs2
FCMPEs	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 0 1	rs2
FDIVd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 0	rs2
FDIVq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 1	rs2
FDIVs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 0 1	rs2
FdMULq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 1 0 1 1 1 0	rs2
FdTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 1 0	rs2
FdTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 1 0	rs2
FdTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 1 0	rs2
FiTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 0 0	rs2
FiTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 0 0	rs2
FiTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 0 0	rs2
FMOVs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 0 0 0 1	rs2
FMULd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 0	rs2
FMULq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 1	rs2
FMULs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 0 1	rs2
FNEGs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 0 1 0 1	rs2
FqTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 1 1	rs2
FqTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 1 1	rs2
FqTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 1 1	rs2
FsMULd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 1 0 1 0 0 1	rs2
FSQRTd	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 1 0	rs2
FSQRTq	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 1 1	rs2
FSQRTs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 0 1	rs2
FsTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 0 1	rs2
FsTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 0 1	rs2
FsTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 0 1	rs2
FSUBd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 0	rs2
FSUBq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 1	rs2
FSUBs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 0 1	rs2

2.4.4.6 *Miscellaneous Instructions*

Table 2–32. Miscellaneous Instruction Opcodes

Mnemonic	Opcodes with Format 31 30	29 25	24 19	18 14	13	12 5	4 0
FLUSH	1 0	unused	1 1 1 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UNIMP	0 0	reserved	0 0 0	const22			

2.4.4.7 *Opcodes In Ascending Numeric Order*

Table 2–33. Instruction Opcode Numeric Listing

Mnemonic	Opcodes with Format 31 30	29	25	24 22	21 19	18 14	13	12 5	4 0
UNIMP	0 0	reserved		0 0 0	const22				
Bicc	0 0	a	cond	0 1 0	disp22				
SETHI	0 0	rd		1 0 0	imm22				
FBfcc	0 0	a	cond	1 1 0	disp22				
CBccc	0 0	a	cond	1 1 1	disp22				
CALL	0 1	disp30							
ADD	1 0	rd		0 0 0 0 0 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
AND	1 0	rd		0 0 0 0 0 1		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
OR	1 0	rd		0 0 0 0 1 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
XOR	1 0	rd		0 0 0 0 1 1		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
SUB	1 0	rd		0 0 0 1 0 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
ANDN	1 0	rd		0 0 0 1 0 1		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
ORN	1 0	rd		0 0 0 1 1 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
XNOR	1 0	rd		0 0 0 1 1 1		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
ADDX	1 0	rd		0 0 1 0 0 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
UMUL	1 0	rd		0 0 1 0 1 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
SMUL	1 0	rd		0 0 1 0 1 1		rs1	i =0	unused (zero)	rs2
							i =1	simm13	
SUBX	1 0	rd		0 0 1 1 0 0		rs1	i =0	unused (zero)	rs2
							i =1	simm13	

Mnemonic	Opcodes with Format 31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
UDIV	1 0	rd	0 0 1 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SDIV	1 0	rd	0 0 1 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ADDcc	1 0	rd	0 1 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ANDcc	1 0	rd	0 1 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ORcc	1 0	rd	0 1 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XORcc	1 0	rd	0 1 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SUBcc	1 0	rd	0 1 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ANDNcc	1 0	rd	0 1 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ORNcc	1 0	rd	0 1 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
XNORcc	1 0	rd	0 1 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
ADDXcc	1 0	rd	0 1 1 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UMULcc	1 0	rd	0 1 1 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SMULcc	1 0	rd	0 1 1 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SUBXcc	1 0	rd	0 1 1 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
UDIVcc	1 0	rd	0 1 1 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SDIVcc	1 0	rd	0 1 1 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TADDcc	1 0	rd	1 0 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TSUBcc	1 0	rd	1 0 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TADDccTV	1 0	rd	1 0 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
TSUBccTV	1 0	rd	1 0 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
MULScc	1 0	rd	1 0 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SLL	1 0	rd	1 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt

Mnemonic	Opcodes with Format						
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
SRL	1 0	rd	1 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt
SRA	1 0	rd	1 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	unused (zero)	shcnt
RDY	1 0	rd	1 0 1 0 0 0	unused	I*	unused (zero)	
RDPSR	1 0	rd	1 0 1 0 0 1	unused	I*	unused (zero)	
RDWIM	1 0	rd	1 0 1 0 1 0	unused	I*	unused (zero)	
RDTBR	1 0	rd	1 0 1 0 1 1	unused	I*	unused (zero)	
WRASR*	1 0	unused	1 1 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRPSR	1 0	unused	1 1 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRWIM	1 0	unused	1 1 0 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
WRTBR	1 0	unused	1 1 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
FPOP1	1 0	rd	1 1 0 1 0 0	rs1	OPF		rs2
FMOVs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 0 0 0 1		rs2
FNEGs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 0 1 0 1		rs2
FABSs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 0 0 1 0 0 1		rs2
FSQRTs	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 0 1		rs2
FSQRTd	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 1 0		rs2
FSQRTq	1 0	rd	1 1 0 1 0 0	unused	0 0 0 1 0 1 0 1 1		rs2
FADDs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 0 1		rs2
FADDd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 0		rs2
FADDq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 1		rs2
FSUBs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 0 1		rs2
FSUBd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 0		rs2
FSUBq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 1		rs2
FMULs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 0 1		rs2
FMULd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 0		rs2
FMULq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 1		rs2
FDIVs	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 0 1		rs2
FDIVd	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 0		rs2
FDIVq	1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 1		rs2
FiTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 0 0		rs2
FdTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 1 0		rs2
FqTOs	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 0 1 1 1		rs2
FiTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 0 0		rs2
FsTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 0 1		rs2
FqTOd	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 0 1 1		rs2

* WRASR and RDASR are general case instructions that include the previous WRY and RDY instructions. WRY and RDY are indicated by setting rd = 0. Setting rd = 30 accesses the DIAG register; setting rd = 31 accesses the ICCR register (both supported by RT620 only).

	Opcodes with Format						
Mnemonic	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
FiTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 0 0		rs2
FsTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 0 1		rs2
FdTOq	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 0 1 1 1 0		rs2
FsTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 0 1		rs2
FdTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 1 0		rs2
FqTOi	1 0	rd	1 1 0 1 0 0	unused	0 1 1 0 1 0 0 1 1		rs2
FPOP2	1 0	rd	1 1 0 1 0 1	rs1	OPF		rs2
FCMPs	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 0 1		rs2
FCMPd	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 0		rs2
FCMPq	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 1		rs2
FCMPEs	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 0 1		rs2
FCMPEd	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 0		rs2
FCMPEq	1 0	unused	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 1		rs2
CPop1	1 0	rd	1 1 0 1 1 0	rs1	OPC		rs2
CPop2	1 0	rd	1 1 0 1 1 1	rs1	OPC		rs2
JMPL	1 0	rd	1 1 1 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
RETT	1 0	unused	1 1 1 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
Ticc	1 0	R* cond	1 1 1 0 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
FLUSH	1 0	unused	1 1 1 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
SAVE	1 0	rd	1 1 1 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
RESTORE	1 0	rd	1 1 1 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LD	1 1	rd	0 0 0 0 0 0	rs1	i =0	asi	rs2
					i =1	simm13	
LDUB	1 1	rd	0 0 0 0 0 1	rs1	i =0	asi	rs2
					i =1	simm13	
LDUH	1 1	rd	0 0 0 0 1 0	rs1	i =0	asi	rs2
					i =1	simm13	
LDD	1 1	rd	0 0 0 0 1 1	rs1	i =0	asi	rs2
					i =1	simm13	
ST	1 1	rd	0 0 0 1 0 0	rs1	i =0	asi	rs2
					i =1	simm13	
STB	1 1	rd	0 0 0 1 0 1	rs1	i =0	asi	rs2
					i =1	simm13	
STH	1 1	rd	0 0 0 1 1 0	rs1	i =0	asi	rs2
					i =1	simm13	
STD	1 1	rd	0 0 0 1 1 1	rs1	i =0	asi	rs2
					i =1	simm13	

R*=reserved

Mnemonic	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
			Opcodes with Format				
LDSB	1 1	rd	0 0 1 0 0 1	rs1	i =0	asi	rs2
					i =1	simm13	
LDSH	1 1	rd	0 0 1 0 1 0	rs1	i =0	asi	rs2
					i =1	simm13	
LDSTUB	1 1	rd	0 0 1 1 0 1	rs1	i =0	asi	rs2
					i =1	simm13	
SWAP	1 1	rd	0 0 1 1 1 1	rs1	i =0	asi	rs2
					i =1	simm13	
LDA	1 1	rd	0 1 0 0 0 0	rs1	i =0	asi	rs2
LDUBA	1 1	rd	0 1 0 0 0 1	rs1	i =0	asi	rs2
LDUHA	1 1	rd	0 1 0 0 1 0	rs1	i =0	asi	rs2
LDDA	1 1	rd	0 1 0 0 1 1	rs1	i =0	asi	rs2
STA	1 1	rd	0 1 0 1 0 0	rs1	i =0	asi	rs2
STBA	1 1	rd	0 1 0 1 0 1	rs1	i =0	asi	rs2
STHA	1 1	rd	0 1 0 1 1 0	rs1	i =0	asi	rs2
STDA	1 1	rd	0 1 0 1 1 1	rs1	i =0	asi	rs2
LDSBA	1 1	rd	0 1 1 0 0 1	rs1	i =0	asi	rs2
LDSHA	1 1	rd	0 1 1 0 1 0	rs1	i =0	asi	rs2
LDSTUBA	1 1	rd	0 1 1 1 0 1	rs1	i =0	asi	rs2
SWAPA	1 1	rd	0 1 1 1 1 1	rs1	i =0	asi	rs2
LDF	1 1	rd	1 0 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDFSR	1 1	rd	1 0 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDDF	1 1	rd	1 0 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STF	1 1	rd	1 0 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STFSR	1 1	rd	1 0 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDFQ	1 1	rd	1 0 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDF	1 1	rd	1 0 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDC	1 1	rd	1 1 0 0 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDCSR	1 1	rd	1 1 0 0 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
LDDC	1 1	rd	1 1 0 0 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STC	1 1	rd	1 1 0 1 0 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STCSR	1 1	rd	1 1 0 1 0 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

Mnemonic	Opcodes with Format							
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
STDCQ	1 1	rd	1 1 0 1 1 0	rs1	i =0	unused (zero)	rs2
					i =1	simm13	
STDC	1 1	rd	1 1 0 1 1 1	rs1	i =0	unused (zero)	rs2
					i =1	simm13	

2.4.5 SPARC Exception Model

SPARC supports three types of traps: *precise*, *deferred*, and *interrupting*. A precise trap (also referred to as a synchronous trap) is induced by a particular instruction and occurs before the processor state is changed by the trap-inducing instruction. All instructions preceding the trap inducing instruction have completed execution.

Deferred traps are caused by a trap-inducing instruction, but the processor state may have been changed by other instructions executing after the trap-inducing instruction. Deferred traps are generated by a unit other than the integer unit executing instructions in parallel with the IU, such as the internal FPU within a RT620 or the CY7C602 FPU in a CY7C601 system.

Interrupting traps occur when an external event interrupts the processor. They are not related to any particular instruction and occur between the execution of instructions. Memory exceptions, RESET, and interrupt requests belong to this group of traps. Interrupting traps are also sometimes referred to as an asynchronous traps.

Several details concerning the handling of exceptions are processor dependent. Detailed information on trap handling is given in *Section 3.9* (RT620) and *Section 6.5.5* (CY7C601).

2.4.5.1 Precise Traps

Precise traps are caused by the actions of an instruction, with the trap stimulus occurring either internally to the SPARC processor or from an external signal which was provoked by the instruction. These traps are taken immediately and the instruction that caused the trap is aborted *before* it changes any state in the processor. All instructions in the SPARC pipeline that preceded the trap-inducing instruction complete execution before the trap is recognized.

2.4.5.1.1 Internal/Software

Precise traps generated by internal hardware are associated with an instruction. The trap condition is detected during the Execute stage of the instruction and the trap is taken immediately, before the instruction can complete.

illegal instruction trap

An illegal instruction trap occurs:

- when the UNIMP instruction is encountered,
- when an unimplemented instruction is encountered (excluding FPops and CPops),
- in any of the situations below where the continued execution of an instruction would result in an illegal processor state:
  1. Writing a value to the PSR's CWP field that is greater than the number of implemented windows (with a WRPSR)
  2. Executing an alternate space instruction with its *i* bit set to 1
  3. Executing a RETT instruction with traps enabled (ET=1)

Unimplemented floating-point and unimplemented coprocessor instructions do not generate an illegal instruction trap. They generate fp exception and cp exception traps, respectively.

privileged instruction

This trap occurs when a privileged instruction is encountered while the PSR's supervisor bit is reset (S=0).

fp disabled

An fp disabled trap is generated when an FPop, FBfcc, or floating-point Load/Store instruction is encountered while the PSR's EF bit =0. In the case of the CY7C601, this trap can also be generated by these instructions if no FPU is present.

cp disabled (CY7C601 only)

A cp disabled trap is generated when a CPop, CBccc, or coprocessor Load/Store instruction is encountered while the PSR's EC bit =0, or if no coprocessor is present ($\overline{CP}$ input signal =1).

window overflow

This trap occurs when the continued execution of a SAVE instruction would cause the CWP to point to a window marked invalid in the WIM register.

window underflow

This trap occurs when the continued execution of a RESTORE instruction would cause the CWP to point to a window marked invalid in the WIM register. The window underflow trap type can also be set in the PSR during a RETT instruction, but the trap taken is a reset. Refer to *Chapter 12* for the instruction definition for RETT.

memory address not aligned

Memory address not aligned trap occurs when a load or store instruction generates a memory address that is not properly aligned for the data type or if a JMPL instruction generates a PC value that is not word aligned (low-order two bits nonzero).

tag overflow

This trap occurs if execution of a TADDccTV or TSUBccTV instruction causes the overflow bit of the integer condition codes to be set. Refer to the instruction definitions of TADDccTV and TSUBccTV in *Chapter 12, SPARC Instruction Set* for details.

trap instruction

This trap occurs when a Ticc instruction is executed and the trap conditions are met. There are 128 programmable trap types available within the trap instruction trap (see *Chapter 12, Ticc Instruction*).

2.4.5.2 Deferred Traps

Deferred traps differ from precise traps in that the trap-inducing instruction may be followed by other instructions that may change the state of the processor. For ROSS SPARC, deferred traps may be caused by a trap-inducing FP or CP operation. Note that FP and CP register loads and stores are considered a type of load and store instruction, and therefore are not a deferred trap.

The servicing of a deferred trap is facilitated by the use of a deferred-trap queue. This queue contains the trap-inducing instruction and its address, as well as other FP (and/or CP, in the case of the CY7C601) instructions that may have been fetched. This allows the deferred-trap handling routine to recover from the deferred trap by emulation or re-execution of the trap-inducing instruction. The topics of floating-point queues are covered in *Section 3.5.1.1* for the RT620 and *Section 7.3.1.2* for the CY7C602 FPU. The topic of floating-point traps for the RT620 is further addressed in *Section 3.9.8*.

2.4.5.3 Interrupting Traps

Interrupting traps occur in response to the interrupt level inputs, memory exceptions, or reset. They differ from both precise traps and deferred traps in that they are not necessarily caused by a particular instruction, or may be due to an exception caused by a previous instruciton. Interrupting traps also differ from the other trap types in that they are not required to provide a means of emulating an instruction (such as the FP queue) that caused an interrupting trap.

2.4.5.3.1 Interrupt Requests

Interrupt requests are made to the SPARC processor by means of a group of interrupt level signals (MIRL(3:0) for the RT620 and IRL(3:0) for the CY7C601). The interrupt level on these lines are compared against the four-bit processor interrupt level (PIL) field in the processor state register (PSR) of the integer unit. If the interrupt level is greater than the setting of the PIL, an interrupting trap is taken by the processor. Note that (M)IRL(3:0) = '0000' denotes no interrupt request and '1111' denotes a non-maskable interrupt request.

For processor-specific information on interrupts, refer to *Section 3.9.7* for the RT620 or *Section 6.5.3* for the CY7C601.

2.4.5.3.2 Reset

The reset trap is a special case of the external asynchronous trap type. It is asynchronous because it is triggered by asserting the $\overline{\text{RESET}}$ input signal. Upon recognizing the $\overline{\text{RESET}}$ signal, the RT620 or CY7C601 enters reset mode and stays there until the $\overline{\text{RESET}}$ line is deasserted. After reset, the processor enters Execute mode and then begins the execute trap procedure. The processor modifies the enable traps bit (ET=0), and the supervisor bit (S=1). It then sets the PC to 0 (rather than changing the contents of the TBR), the nPC to 4, and transfers control to location 0. *All other PSR fields, and all other registers retain their values from the last Execute mode.*

If the processor was reset from error mode, then the normal actions of a trap have already been performed, including setting the *tt* field to reflect the cause of the error mode. Because this field is not changed by the reset trap, a post-mortem can be conducted on what caused the error mode. The processor enters error mode whenever a synchronous trap occurs while traps are disabled.

Note: Upon power-up reset the state of all registers other than the PSR are undefined.

2.4.5.3.3 Memory Exceptions

Memory exceptions are signaled to the processor by the cache controller. In general, these may be due either to a cache access exception or to an MBus exception that is signalled by the responding MBus unit.

Memory exceptions are signalled to the RT620 by the RT625 CMTU using the $\overline{\text{IMEXC}}$ signal. In a similar fashion, the CY7C604/605 CMU uses the $\overline{\text{MEXC}}$ signal to signal memory exceptions to the CY7C601. The IU (either RT620 or CY7C601) This interface and other details concerning memory exceptions are covered in the chapters specific to these processors.

2.4.5.4 Trap Operation

Once a trap is taken, the following operations take place:

- Further traps are disabled (asynchronous traps are ignored; synchronous traps force an error mode).
- The S bit of the PSR is copied into the PS bit; the S bit is then set to 1.
- The CWP is decremented by one (modulo the number of windows) to activate a trap window. This happens regardless of the contents of the WIM register, which is ignored upon entering a trap.
- The PC and nPC are saved into r[17] and r[18], respectively, of the trap window.
- The *tt* field of the TBR is set to the appropriate value.
- If the trap is not a reset, the PC is written with the contents of the TBR and the nPC is written with TBR + 4. If the trap is a reset, the PC is set to address zero and the nPC to address four.

The SPARC architecture does not automatically save the PSR into memory during a trap. Instead, it saves the volatile S bit into the PSR itself and the remaining fields are either altered in a reversible manner (ET and CWP), or should not be altered in the trap handler until the PSR has been saved to memory.

2.4.5.4.1 Trap Addressing

The trap base register (TBR) is made up of two fields, the trap base address (TBA) and the trap type (*tt*). The TBA contains the most-significant 20 address bits of the trap table, which is in external memory. The trap type field, which was written by the trap, not only uniquely identifies the trap, it also serves as an offset into the trap table when the TBR is written to the PC. The TBR address is the first address of the trap handler. However, because the trap addresses are only separated by four words (the least-significant four bits of TBR are zero), the program must jump from the trap table to the actual address of the particular trap handler.

Of the 256 trap types allowed by the 8-bit *tt* field, half are dedicated to hardware traps (0-127), and half are dedicated to programmer-initiated traps (Ticc). For a Ticc instruction, the processor must calculate the *tt* value from the fields given in the instruction, while the hardware traps can be set from a table such as the one below. The *tt* field remains valid until another trap occurs. Refer to the Ticc instruction definition in *Chapter 12* for further information.

2.4.5.4.2 Trap Types and Priority

Each type of trap is assigned a priority. When multiple traps occur, the highest priority trap is taken, and lower priority traps are ignored. In this situation, a lower priority trap must either persist or be repeated in order to be recognized and taken. Refer to *Table 3–7* for a trap priority listing for the RT620 or *Table 6–7* for a trap priority listing for the CY7C601.

2.4.5.4.3 Return From Trap

On returning from a trap with the RETT instruction, the following operations take place:

- The CWP is incremented by one (modulo the number of windows) to re-activate the previous window.
- The return address is calculated.
- Trap conditions are checked. If traps have already been enabled (ET=1), an illegal instruction trap is taken. If traps are still disabled but S=0, or the new CWP points to an invalid window, or the return address is not properly aligned, then an error mode/reset trap is taken.
- If no traps are taken, then traps are re-enabled (ET=1).
- The PC is written with the contents of the nPC, and the nPC is written with the return address.
- The PS bit is copied back into the S bit.

The last two instructions of a trap handler should be a JMPL followed by a RETT. This instruction couple causes a non-delayed control transfer back to the trapped instruction or to the instruction following the trapped instruction, whichever is desired. See the RETT instruction definition for details.

RT620 hyperSPARC Central Processing Unit

The hyperSPARC RT620 is a second-generation SPARC RISC processor, providing a highly pipelined, superscalar Central Processing Unit (CPU) solution for high-performance computing systems. Features of the RT620 include:

- Dual instruction launch capability for a majority of instruction combinations
- Integrated integer unit (IU) and floating-point unit (FPU)
  - Integer unit and floating-point unit feature enhanced pipelines for increased performance
- Two-way set associative 8-Kbyte instruction cache (ICACHE)
- 64-bit high-speed Intra-Module Bus (IMB) for high memory bandwidth
  - 3.3V IMB logic for increased speed and reduced power consumption
- Hardware support for integer multiply and divide instructions
- Compliant to SPARC Instruction Set Architecture Version 8

The RT620 is designed as part of a tightly coupled hyperSPARC CPU system, consisting of the RT620, the RT625 Cache Controller, Memory Management Unit and Tag Unit (CMTU), and two or four RT627 Cache Data Units (CDUs). The hyperSPARC CPU system provides the RT620 with a 128- or 256-Kbyte cache, a SPARC reference Memory Management Unit (MMU), and SPARC MBus with full multiprocessing support.

3.1 RT620 hyperSPARC CPU

The RT620 is divided into several different functional blocks: the integer data path (IDP), the floating-point data path (FPDP), the instruction fetch (IFETCH) unit, an on-chip (execute-only) instruction cache, the Instruction Scheduler (ISCHED), the floating-point instruction decode-schedule-and-dispatch controller (FPSCHED), and the Intra-Module Bus Interface Unit (IBIU). These blocks are illustrated in *Figure 3–1.*

Instructions are fetched for the RT620 by the instruction fetch (IFETCH) block under the control of the integer unit. In order to minimize delays due to instruction cache misses, instructions are simultaneously requested from the instruction cache and the IBIU. Instructions are supplied to the IFETCH by the instruction cache if a cache hit occurs, or from external memory through the IBIU in the case of a cache miss. The IBIU instruction fetch request from external cache is nullified if an internal instruction cache hit occurs.

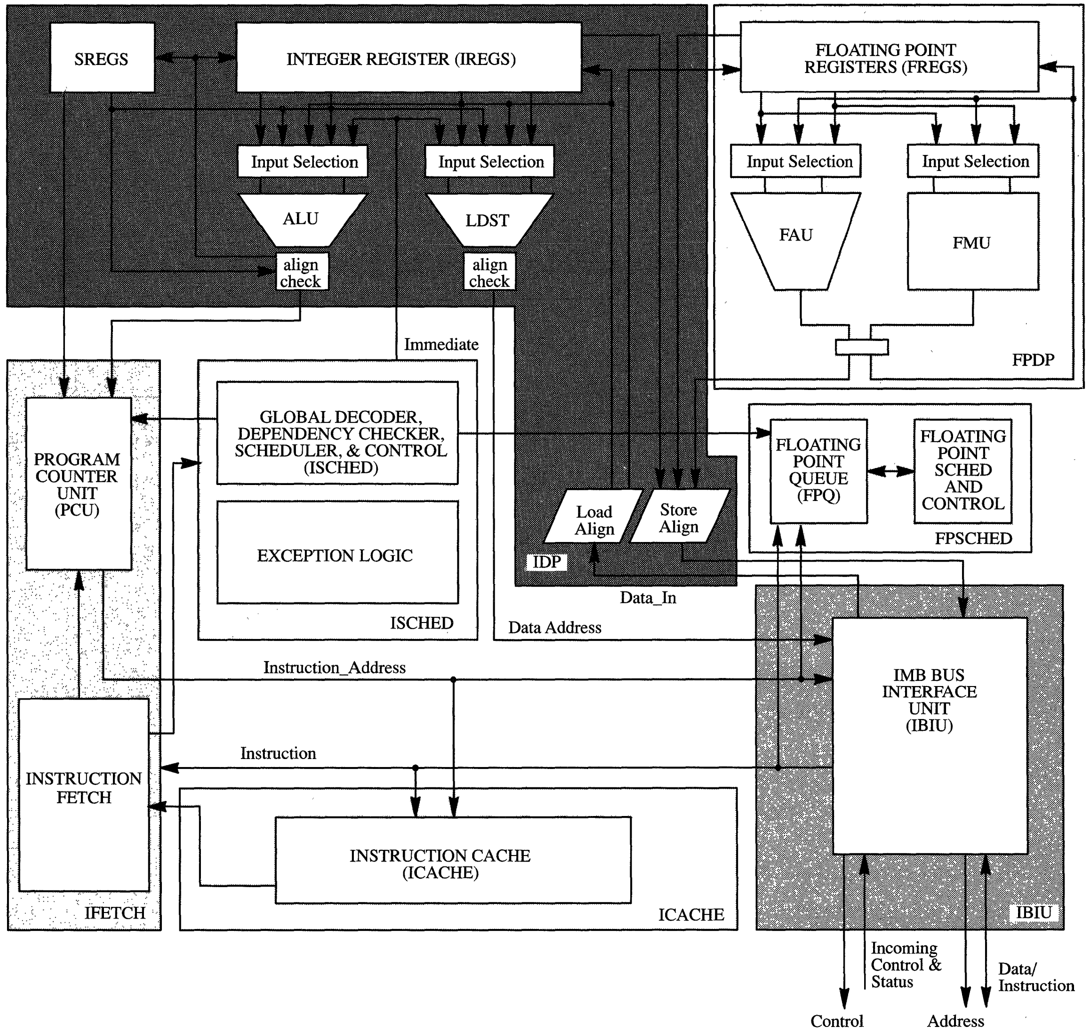


Figure 3–1. RT620 hyperSPARC CPU Block Diagram

Once an instruction pair is fetched, it is globally decoded by the ISCHED block. The instruction pair is scheduled for multiple or single instruction launch depending upon the instruction combination and the execution resources required. If an instruction pair (also called an instruction *packet*) cannot be executed together, the packet is split and the individual instructions are executed singly and in order. Floating-point instructions are dispatched to the floating-point unit by the ISCHED, where they are locally decoded by the FPSCHED unit. Local decoding of the integer instructions is performed by a sub-block of the integer data path (IDP). *Figure 3–1* illustrates the functional blocks of the RT620.

IDP — The integer data path is comprised of several units. The arithmetic and logic unit (ALU) handles integer arithmetic, logical, and shift instructions. The load and store unit (LSU) handles instructions that load and store data between memory and registers. This includes the loading and storing of both

integer and floating-point (fp) data. The program counter unit (PCU) maintains the program counter and performs condition code evaluation. The special register unit (SRU) handles instructions which read and write the SPARC special registers (SREGS). The integer register file (IREGS) is also contained in the integer unit data path.

FPDP — The floating-point unit is comprised of the floating-point queue (FPQ), the floating-point arithmetic unit (FAU), the Floating-Point Multiplier Unit (FMU), the floating-point register file (FREGS), and the floating-point status register (FSR). The FPDP handles all SPARC fp data operations. Note that fp load and store operations, which are considered a subset of standard load and store instructions, are handled by the integer unit. The FPDP generates results which are fully compatible with the ANSI/IEEE 754-1985 standard.

ISCHED — The Instruction Scheduler performs several key control functions. It provides global instruction decoding, identifying which execution unit resources are required and determining whether sequential or simultaneous execution is possible. It determines whether data forwarding can be performed and whether instruction dispatches (also called "launches") need to be delayed due to data dependencies. It initiates instruction launch and it identifies and controls interrupt and trap handling.

FPSCHED — The floating-point instruction scheduler performs key control functions for the floating-point unit. When the integer unit detects fp instructions in the Decode stage, it off-loads these instructions to the fp functional units and continues processing. Therefore, functional blocks exist which perform necessary decode, scheduling and control for fp operations. The FPSCHED provides floating-point instruction queue control (FPQC), performs fp instruction decoding, and performs fp data dependency and data forwarding resolution. It provides the integer unit with signals that indicate when fp load or store instructions should be delayed due to data dependence on instructions in the FPQ. It also initiates fp instruction launch.

IFETCH — The instruction fetch unit consists of two major functional blocks: the program counter unit (PCU) and the instruction fetch controller (IFETCHC).

The program counter unit (PCU) calculates the address of the next instruction to be fetched. It handles instructions which cause program control transfer such as CALL and Branch. This unit handles both integer and fp Branch instructions.

The hyperSPARC CPU fetches two instructions (a *packet*) at a time. The instructions within the packets are referred to as *slots*. The first instruction in the packet is referred to as *slot-a* and the second instruction in the packet is referred to as *slot-b*. In each clock cycle, the CPU attempts to launch both the instructions in the packet. If the instruction packet cannot be launched together, the instructions are launched singly and in order.

In order to fully support the instruction fetch and launch mechanism, the CPU's on-chip instruction cache supports non-aligned packet boundary accesses. The instruction cache also interfaces with the 64-bit data path of the IBIU block and the external cache subsystem.

ICACHE — The on-chip instruction cache stores 8 Kbytes of instructions (corresponding to 2048 instructions). Its inclusion follows the Harvard architectural approach, reducing contention for the bus during memory accesses. This approach allows parallel access to instructions and data. While instructions are fetched from the on-chip ICACHE, the external bus can simultaneously access data from memory.

When a conflict over bus usage between a data and instruction access does arise (e.g., in the case where an instruction cache miss occurs), the data access receives priority over the instruction access.

When an on-chip instruction cache miss occurs, only one extra clock cycle is required to fetch the instruction from the external cache. This is accomplished by simultaneously accessing the on-chip instruction cache and the external cache subsystem. If an on-chip instruction cache hit does occur while the external access is in progress, the external access is canceled.

IBIU — The IBIU provides the interface between the CPU and the other hyperSPARC chips. The IBIU samples incoming control signals and propagates controls to appropriate functional blocks. The IBIU is responsible for generating memory access control signals to the cache memory subsystem (e.g., address, size of data, access-type, etc.). Data and instructions are read from memory, and data is written to memory, through the IBIU.

3.2 Integer Data Path (IDP)

The integer data path is comprised of five major data units: the arithmetic and logical unit (ALU), the load and store unit (LSU), a set of special control registers (SRU), and a register file for integer data (IREG). *Figure 3–2* illustrates the logical blocks of the integer unit, which are described in this section.

3.2.1 Arithmetic and Logical Unit (ALU)

The ALU handles all the SPARC integer operation instructions. The following instructions are executed by the arithmetic and logical unit:

integer arithmetic instructions: ADD, ADDcc, ADDX, ADDXcc, SUB, SUBcc, SUBX, SUBXcc, MULScc, UMUL, UMULcc, SMUL, SMULcc, SRL, SRA, SLL, SETHI, TADDcc, TADDccTV, TSUBcc, TSUBccTV, SAVE, RESTORE.

integer logical instructions: AND, ANDcc, ANDN, ANDNcc, OR, ORcc, ORN, ORNcc, XOR, XORcc, XNOR, XNORcc.

special register unit read/write instructions: RDY, RDASR, RDPSR, RDWIM, RDTBR, WRY, WRASR, WRPSR, WRWIM, WRTBR.

fetch control instructions: JMPL, RETT, FLUSH

These instructions are described in detail in *Chapter 12, SPARC Instruction Set.*

3.2.2 Load and Store Unit (LSU)

The load and store unit (LSU) handles all the SPARC integer and fp load and store instructions. It is comprised of two sub-blocks (represented in *Figure 3–2*), the Load/Store adder and the Load/Store alignment shifter.

The LSU performs two basic functions:

- It calculates the data effective virtual address.
- It aligns data which is being loaded and/or stored. Therefore, the LSU uses add, shift, and memory alignment exception checking logic.

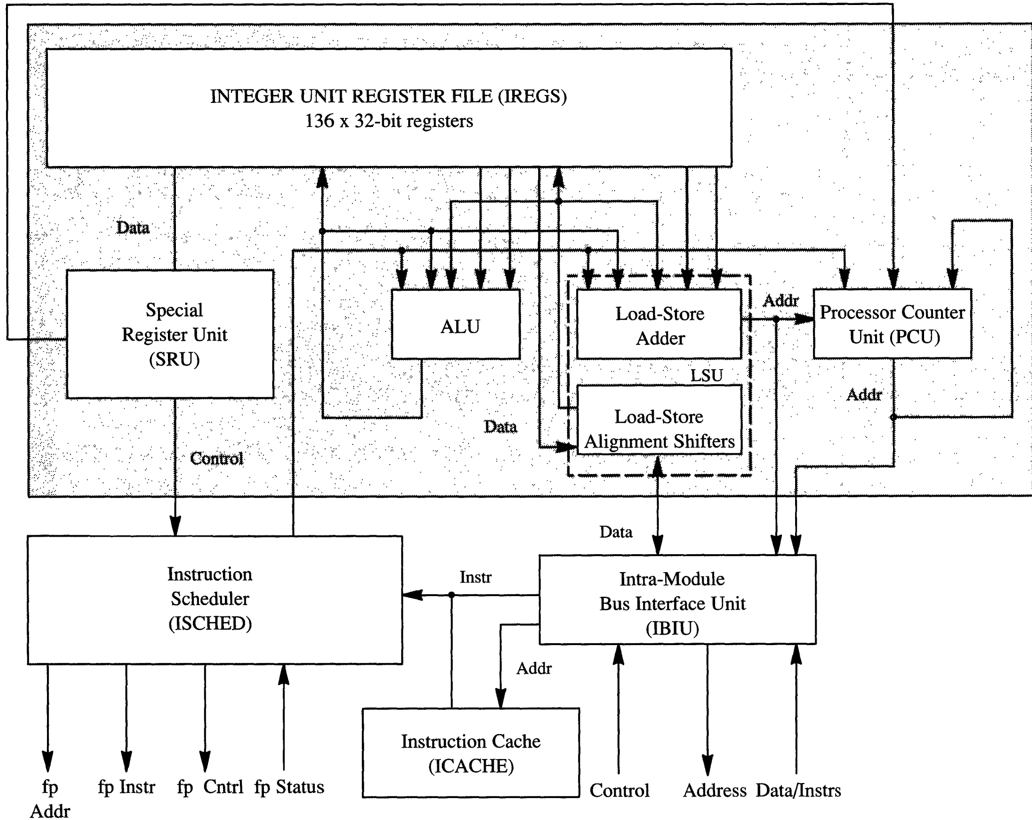


Figure 3–2. IDP Blocks

The following instructions are executed by the load and store unit:

- Integer load and store instructions: LDUB, LDSB, LDUH, LDSH, LD, LDD, LDUBA, LDSBA, LDUHA, LDSHA, LDA, LDDA, STB, STH, ST, STD, LDSTUB, SWAP, STBA, STHA, STA, STDA, LDSTUBA, SWAPA.
- fp load and store instructions: LDFSR, LDF, LDDF, STFSR, STF, STDF, STDFQ.

These instructions are described in detail in *Chapter 12, SPARC Instruction Set.*

3.2.3 Program Counter Unit (PCU)

The program counter unit (PCU) performs three functions. It selects the address for the next instruction from several possible sources. It performs condition code evaluation. It also maintains the program counter (PC) through the successive stages of the execution pipeline.

The PCU executes the following SPARC integer and fp instructions:

- Branch and CALL instructions: Bicc, FBfcc, and CALL.

These instructions are described in full detail in *Chapter 12, SPARC Instruction Set.*

The PCU is also involved in the execution of the Ticc, JMPL, RETT, and FLUSH instructions through the propagation of instruction addresses to the ICACHE and the IBIU units.

3.2.4 Special Register Unit (SRU)

The special register unit is comprised of the control/status registers defined for Version 8 of the SPARC Architecture. These special registers are identical to those implemented in the CY7C601, with the addition of the instruction cache control register (ICCR) and the diagnostic (DIAG) register. A more detailed description of integer unit control/status registers is given in *Section 2.2.1*.

3.2.5 Internal Register File (IREGS)

The general register model for the hyperSPARC is given in *Figure 3–3*. The *r*-registers (or IREGS) are the working register set for the SPARC integer unit. All *r*-registers are 32-bits in length. The 136 *r*-registers supported by the RT620 are divided into 128 windowed *r*-registers and eight global registers. The 128 windowed *r*-registers are divided into eight overlapping windows of twenty-four *r*-registers. The twenty-four *r*-registers that comprise a window are further subdivided into three groups of eight registers, referred to as the *in*, *out*, and *local* registers. The eight *r*-register windows overlap in a manner such that the *in* registers of one window are the *out* registers of the previous window. *Local r*-registers are not shared with another window, but are private to that window. The current window in use by the processor is pointed to by the current window pointer (CWP), a field within the processor state register. In addition to the *r*-register window, there are eight global registers that are accessible regardless of the current window pointer.

More information on the IREGS and the SPARC register model is given in *Chapter 2*.

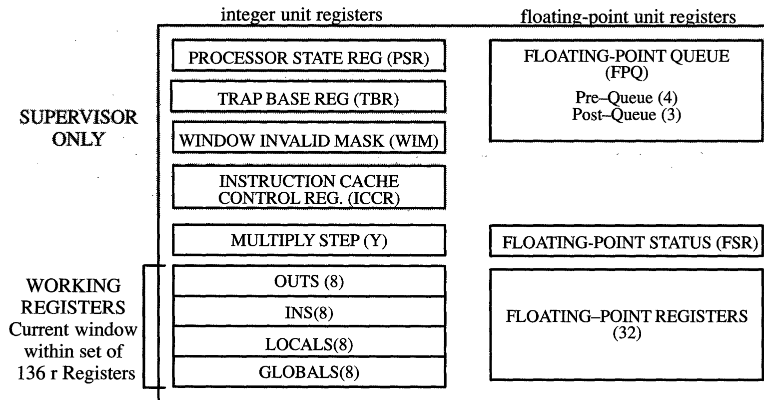


Figure 3–3. hyperSPARC Register Model

3.3 Instruction Fetch Unit (IFETCH)

Before discussing the details of these blocks, it is helpful to first reach an understanding of the state machine which governs the behavior of the integer unit. The behavior of the entire hyperSPARC CPU follows that of the integer unit. *Figure 3–4* shows the four states of the integer unit: RESET, EXECUTE, HOLD, and ERROR. During reset, the reset line ($\overline{\text{PRST}}$) will be asserted. Until reset is deasserted, the CPU remains in the RESET state. When the reset signal is deasserted, the CPU enters the EXECUTE state. This is the normal mode of operation.

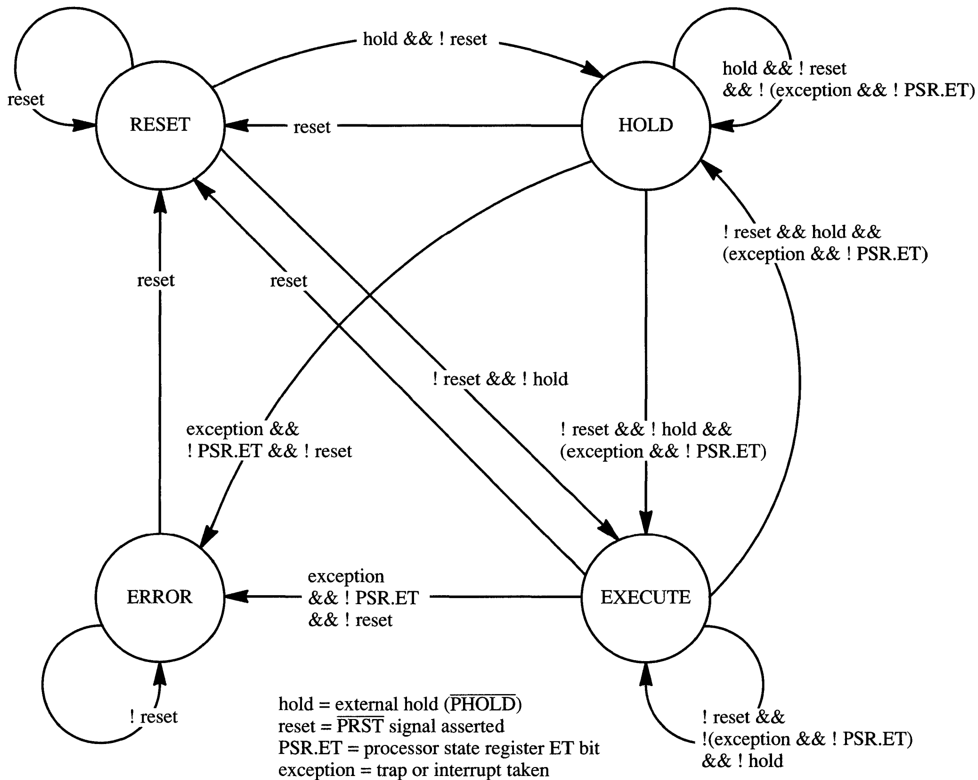


Figure 3–4. RT620 State Transition Diagram

After reset, the processor remains in the EXECUTE state unless (i) a precise or deferred trap occurs while traps are disabled, or (ii) for some reason, reset is asserted, or (iii) an access to the external cache subsystem has resulted in an external hold signal being asserted (PHOLD). If condition (i) occurs, the CPU enters ERROR state and remains in ERROR state until reset is asserted again. If condition (ii) occurs, RESET state is entered until the reset signal is again deasserted. If condition (iii) occurs, the CPU enters HOLD state and remains in the HOLD state until either a reset is asserted or the external hold is deasserted. If the reset is asserted, RESET state is entered as in the previous cases. If the external hold is deasserted, the CPU returns to EXECUTE state.

While the integer unit remains in EXECUTE mode, instruction fetches may occur. The process of instruction fetch involves (i) generating an address for instruction fetch, (ii) locating and retrieving that instruction, and (iii) placing the instruction in a cache from which it can be decoded and executed.

Once the address for the next instruction packet is available, the instruction must be located. The PCU will send the instruction address to the IBIU and the ICACHE simultaneously. If the ICACHE is enabled and the instruction is located in the ICACHE, the ICACHE will return the instruction packet to IFETCH (Instruction FETCH) and signal the IBIU to cancel the request for data from the external cache subsystem. Otherwise, the IBIU continues with the request for an instruction fetch to the external cache subsystem.

If the instruction is available in the external cache, the external cache subsystem will return the instruction to IFETCH through the IBIU. Otherwise, a hold signal (PHOLD) will be generated by the external cache

subsystem controller (RT625). The RT625 asserts the hold signal until the cache line corresponding to the instruction has been fetched from main memory and stored in the external cache subsystem. The RT625 fetches the requested instruction data first and then the rest of the words in the cache line. The RT625 generates a data strobe signal when the requested instruction is available on the Intra-Module Bus; it deasserts the hold immediately (i.e., it does not wait until the entire external cache line is filled).

3.4 Instruction Scheduler (ISCHED)

Rather than attempting to execute all instruction combinations simultaneously, the SPARC instruction set has been divided into two major groups to simplify the detection of conditions for executing simultaneous instructions. These two groups are instructions which must be executed sequentially, and instructions which are eligible for simultaneous execution.

Table 3–1. Instruction Grouping

Group-Id	Group-Name	Instructions
A	Load/Store	LDSB, LDSH, LD, LDUB, LDUH, LDD LDF, LDDF ST, STB, STH, STD STF, STDF SWAP, LDSTUB
B	ALU (arithmetic/logic)	ADD, AND, OR, XOR, SUB, ANDN, ORN, XNOR ADDX SUBX ADDcc, ANDcc, ORcc, XORcc, SUBcc, ANDNcc, ORNcc, XORNcc ADDXcc, SUBXcc TADDcc, TSUBcc, MULScc SLL, SRL, SRA, SETHI
C	FP instruction (FP add/multiply except fcmp)	FADDs, FADDd, FSUBs, FSUBd FTO (fp data conversions) FMULs, FSMULd, FMULd FDIVs, FDIVd, FSQRTs, FSQRTd FMOVs, FNEGs, FABSs fp instruction (quad precision)
D	(reserved)	(reserved)
E	Bcc (Branch control transfer)	Bicc, FBfcc
F	Single-step	CALL, JMPL, RETT, SAVE, RESTORE TADDccTV, TSUBccTV UMUL, UMULcc, SMUL, SMULcc UDIV, UDIVcc, SDIV, SDIVcc RDY, RDPSR, RDWIM, RDTBR WRY, WRPSR, WRWIM, WRTBR LDSBA, LDSHA, LDA, LDUBA, LDUHA, LDDA, LDFSR STA, STBA, STHA, STDA, STDFQ, STFSR, SWAPA, LDSTUBA Ticc, FLUSH, CPOPS, CBccc, illegal, LDC, LDDC, LDCSR, STC, STDC, STCSR, STDCQ FCMPs, FCMPd, FCMPEs, FCMPEd

In order to simplify this process of group classification, instruction decoding has been broken down into two levels. These two levels are global (scheduling) decoding and local (execution unit) decoding.

During the global level of Decode, each instruction in the Decode buffer is characterized as either belonging to a group which must always be launched one at a time (referred to as an *fgroup*) or belonging to a group which is eligible for simultaneous launch (sometimes referred to as a *non-fgroup*). The instruction groups eligible for simultaneous launch can be conceptualized as being mapped to one of four execution units.

During global Decode, resource conflicts, data dependencies and operand forwarding are detected and resolved. After group characterization is performed by the instruction decoder, local decoding is performed. If an instruction is recognized as a floating-point instruction, local decoding for that instruction is performed by the FPSCHED.

3.4.1 Single Instruction Launch Group

During global Decode, each instruction in the instruction packet is classified as either a single launch instruction, or a multiple launch instruction. If either instruction belongs to the single step group, then it must be executed by itself. In other words, single step instructions cause the execution of the packet to be split (i.e., to be executed in sequence). The instructions in this group are:

- CALL, JMPL, RETT, SAVE, RESTORE, TADDccTV, TSUBccTV, UMUL, UMULcc, SMUL, SMULcc, UDIV, UDIVcc, SDIV, SDIVcc, Ticc, FLUSH, RDPSR, RDTBR, RDWIM, RDY, RDASR, WRPSR, WRTBR, WRWIM, WRY, WRASR, LDSBA, LDSHA, LDUBA, LDUHA, LDA, LDDA, LDSTUBA, SWAPA, LDFSR, STFSR, STDFQ, STBA, STHA, STA, STDA, FCMP, (CBccc, LDC, LDDC, LDCSR, STC, STDC, STCSR, STDCQ).
- all other coprocessor instructions (*cop1* and *cop2*).†
- any other unimplemented instruction or reserved opcode detected at global Decode time (except for unimplemented fp instructions).

All other instructions are eligible to be executed simultaneously.

Special Case: There is a special case regarding single step instructions when a write special register or load floating-point status register instruction has been executed. The execution of write special register or load floating-point status register instructions cause packet splitting to be enforced for the next three instructions regardless of the group to which the three succeeding instructions belong.

3.4.2 Multiple Instruction Launch Group

If the instruction scheduler detects a packet which does not contain a single launch instruction, does not contain two instructions of the same group (for an exception to this, see *Table 3–2*) does not have an intra-packet data conflict, and does not have an inter-packet data conflict, then the instruction scheduler will launch both instructions in parallel.

If there is an intra-packet data conflict, the instruction scheduler will split the packet. If there is an inter-packet data conflict, the instruction scheduler will delay the slot-b instruction if the conflict exists only for the slot-b instruction. The scheduler will delay both instructions if the conflict exists for the slot-a instruction. The concept of intra-packet and inter-packet dependencies is discussed in *Section 3.4.3* under the notes for case vii.

As mentioned previously, the group *f* instructions force packet splitting. The following table lists combinations of instructions in the Decode packet that are eligible for simultaneous execution.

† The RT620 does not support a coprocessor interface. The RT620 enters a coprocessor disabled trap upon encountering a coprocessor instruction.

Table 3–2. Instruction Combinations Eligible for Simultaneous Execution

Slot-a	Slot-b
group A	group B
group A	group C
group A	group E
group B	group A
group B [2]	group B [2]
group B	group C
group B	group E

Slot-a	Slot-b
group C	group A
group C	group B
group C	group C [1]
group C	group E
group E	group A
group E	group B
group E	group C

Notes: 1. Assuming the FPQ is not full, from the perspective of the instruction scheduler, the combination of two group C instructions in a packet behaves as though both instructions are launched simultaneously.

2. A special case exists called Fast Constant (refer to section 3.4.4.1) which permits two group B instructions in a packet to be executed at the same time. Otherwise, two group B instructions in an instruction packet must be executed singly.

3.4.3 Interlocks and Dependencies

There are situations where certain activities of the processor are temporarily suspended. These suspensions come in two forms:

1. A total freeze on activities in both the integer unit and the floating-point unit (an interlock caused by an external hold being applied by the external memory subsystem or $\overline{\text{IMBNA}}$ asserted and data access required.)
2. A delay in launching additional instructions (i.e., a delay caused by the lack of available computing resources or available data to continue launching instructions).

More specifically, instruction launch delays can be caused by:

i. A miss in the ICACHE.

ii. Contention for integer register file read ports.
Whenever a store instruction is executed, the pipeline maintains this information and the scheduler checks to see if a store instruction is in the Execute stage. If a fetched instruction requires the Load/Store adder (e.g., LD, ST, LDSTUB or SWAP), that instruction is delayed until the store in execute has advanced to the Cache stage. The reason for this is that a store instruction actually has three source operands. The first two operands (rs1 and rs2, or rs1 and the signed immediate field) are used to calculate the effective address. The rd field contains the third source, that is the register containing the data to be written to memory. Since the integer register file has only two read ports which can be used by the LSU, the third operand access must be deferred to the Execute stage of the Store instruction. When a Load or Store instruction is in Decode, it requires access to the register ports to obtain its effective address source operands. This access conflicts with that of the store instruction which is in the Execute stage. Therefore, the second Load-Store instruction will be delayed.

A JMPL, RETT or FLUSH that follows a store instruction must be delayed for one clock to avoid bus contention. A JMPL, RETT or FLUSH instruction that follows a LDSTUB or SWAP instruction will be delayed for two clocks to avoid bus contention.

iii. Contention for execution units.

iv. Arbitrary internal delays.

There are cases involving Bicc, FBfcc, CALL, JMPL, and RETT that result in delays until the next instruction is available. For the cases of JMPL and RETT, these instructions are both delayed control transfer instructions. Because it is possible for these control transfer instructions to come alone or in pairs and instructions are fetched two at a time, the location of a delayed control transfer instruction (slot-a vs. slot-b) and availability of delay slot instructions determine when a change in the flow of program control takes effect. When a Bicc, FBfcc, CALL, JMPL or RETT instruction is in Decode and the delay slot instruction is not available, the control transfer instruction will be delayed until the delay slot instruction is available.

A specified delay (called an "internal nop" or *inop*) is inserted after any JMPL or RETT instruction. This provides sufficient time to generate the new target instruction address and only fetch the delay slot instruction packet. An internal nop (inop) has the same effect as a nop instruction; there is no operation and no program accessible machine state is altered.

In the case of FLUSH instructions, the processor must perform an ICACHE lookup and a Writeback (FLUSH). This requires that three inops be inserted after any FLUSH instruction before a new instruction fetch can be performed. Also note that a FLUSH instruction is delayed until any instruction or data access that is in progress is completed.

v. The presence of an exception that has not yet trapped.

The scheduler also tracks the occurrence of exceptions that are detected at various pipeline stages. If an exception has been detected, the pipeline stages advance until the trap is taken; but additional instructions are neither fetched nor launched until the trap is taken.

vi. A full fp pre-queue while additional fp instructions are "waiting" in the DBUF (Decode buffer).

The floating-point unit provides the scheduler with status information regarding free space in the fp pre-queue. When fp instructions exist in the integer unit decoder and there are no free entries in the pre-queue for off-loading fp instructions, additional instruction fetch and launch cannot continue. Instruction fetch and launch resumes when instructions in the floating-point unit launch, thus making a free entry available in the fp pre-queue.

vii. Data dependencies (integer or fp).

Data dependencies need to be checked in two directions. The first involves checking for data dependencies between instructions in the same packet (called "intra-packet" dependencies). The second direction involves dependency checking between instructions which have not yet been launched and instructions currently advancing through the pipeline stages but have not yet written results back to the register file (called "inter-packet" dependencies). Dependency checking covers both the integer unit and floating-point unit and is limited to detecting register conflicts.

When intra-packet dependencies are detected the second instruction will be delayed at least one clock. For inter-packet comparisons, either one instruction will be delayed (if only the slot-b instruction is involved) or both instructions will be delayed (if the slot-a instruction is involved).

viii. FP compare instructions in the pre-queue and an FBfcc instruction in the instruction (not fp) decoder.

The fp compare instructions and ldfsr are the only fp instructions that can affect the fp condition codes. The fp Branch is delayed until new fp condition codes are available (in the ex2 stage of an fp compare instruction and the Update stage of a LDFSR instruction).

ix. An LDFSR or a STFSR in the instruction decoder and a non-empty FPQ.

x. Integer multiply instructions cause further instruction launch to be held for 18 cycles.
The integer multiply instructions require 17 clock cycles to complete execution. Instruction launches can continue after cycle 17.

xi. Execution of an integer divide instruction causes further instruction launch to be held for 37 cycles.
The integer divide instructions require 37 cycles to complete execution. Instruction launches can continue after the execution is complete.

Instruction Execution Notes:

Integer Unit Forwarding: ALU results are computed in one clock. The results move through a series of buffers as the instruction moves through each pipeline stage. If an instruction in Decode stage requires the result from an ALU operation before the result is written back to the destination register, this result is *forwarded* to the instruction in Decode stage as an operand input. However, if the instruction in Decode stage requires the result of a load operation in the Execute stage or Cache stage, a delay will occur until the operand is available. In the floating-point unit, since more than one clock is required to execute an instruction, forwarding is not possible for intermediate pipeline stages, so delays are performed in much the same way as for load instructions.

Integer Unit Dependency Checking: Since dependency checking needs to be performed for both the integer unit and floating-point unit, responsibility for the checks is divided between the two units. In the cases where inter-packet dependencies are resolved by the FPSCHED, dependencies detected by the FPSCHED are signaled to the integer unit scheduler. For more information on fp instruction scheduling, see *Section 3.5.1.*

There are two possible outcomes for inter-packet dependencies. When the slot-a instruction encounters a data dependency, both the slot-a and slot-b instructions are delayed. When the slot-a instruction does not encounter a data dependency and the slot-b instruction does encounter a data dependency, the slot-a instruction is launched and only the slot-b instruction is delayed. This preserves the execution order of programs and minimizes the complexity of the exception handling logic.

3.4.4 Special Features

There are three special cases in which multiple instruction launch is allowed that otherwise would have to be executed sequentially. These three cases are called *Fast Constant*, *Fast Index*, and *Fast Branch.*

3.4.4.1 Fast Constant

When there is a need to construct 32-bit constants, SPARC typically accomplishes this by executing two ALU instructions. For example:

```
sethi %hi(const), %rx
or %rx, %lo(const), %rx
```

In these instruction sequences, no ADD or logical OR need actually be performed. These operations can be accomplished through simple concatenation. Therefore, when the SETHI is in slot-a and the add/or is in slot-b, it is possible to construct the constant in one step rather than two sequential steps by simply concatenating the high and low immediate fields of the instructions on the operand 2 input bus to the ALU. During a Fast Constant, register %g0 (constant zero) is selected as the ALU input operand 1 and the concatenated 32-bit constant is selected as the input operand 2. The operation performed by the ALU will be an OR operation. The result can then be passed through the ALU.

3.4.4.2 Fast Index

There is a need to rapidly establish base addresses for array indexing, especially for programs which use complex data structures. This is typically accomplished by executing two instructions sequentially, for example :

```
sethi %hi(const), %rx
ld [ %rx + %ry], %rz
  or
sethi %hi(const), %rx
ld [%rx + imm], %rz
  or
sethi %hi(const), %rx
st %rz, [%rx + %ry]
  or
sethi %hi(const), %rx
st %rz, [%rx + imm]
```

In this instance, although the two instructions do not require the same execution unit, there appears to be a data dependency. An add is performed and the results of the SETHI need to be written to the register file. However, when the SETHI is in slot-a and the load or store (including a fp load or store) is in slot-b, it is possible to provide the "shifted" constant to both the operand 2 input of the ALU and the operand 1 input of the LSU in one step rather than two sequential steps. The ALU processes the SETHI in the normal fashion. The LSU calculates the effective address by adding the operand 1 input (generated by the instruction decoder, not the fp decoder) with the second operand (specified by the instruction).

In other words, instruction pairs of the type:

```
sethi %hi(const), %rx
ldd [%rx + imm], %rz
```

are executed as

```
sethi %hi(const), %rx
ldd [%hi(const) + imm], %rz
```

3.4.4.3 Fast Branch

Ordinarily, fetching a condition-code-altering instruction and conditional Branch in the same packet would cause single instruction launch to occur. Consider the case where slot-a contains an ALU instruction which sets condition codes (e.g., ANDcc) and slot-b contains a Branch on Condition Code (e.g., BNE). Note that Branch Always (e.g., BA) and Branch Never (e.g., BN) do not depend on condition codes.

Ordinarily, the Branch instruction that uses the condition codes cannot be launched at the same time as the ALU instruction that sets the condition codes. This is because the time at which they are needed to evaluate whether the branch should be taken or not has already passed by the time the condition codes are set. Also, since the condition codes are not written back to the PSR immediately, a dependency issue exists.

Delays are avoided for most intra-packet and inter-packet branch condition code dependencies through the application of two techniques:

1. condition code forwarding
2. special handling of Branch under the Fast Branch situation

Condition code forwarding works similarly to operand forwarding. That is, if condition codes are required the clock after an ALUcc instruction has been launched, the newly generated condition codes are provided even though the PSR has not yet been updated.

In the special case where the ALUcc and Bicc instructions are contained in the same packet (i.e., there is an intra-packet condition code dependency and the delay slot instruction is available in the Fetch stage), the situation is referred to as a Fast Branch condition. An assumption is made that code generation is biased such that the Branch will be taken more often than not taken. Rather than split the packet and potentially waste a clock cycle, the new target address fetch is attempted. If the Branch is taken, the new target is available by the time the delay slot instruction has been executed. Otherwise, the delay slot instruction packet is executed and the original instruction stream is restored. *Section 3.8.12.3* describes the pipeline operation for the Fast Branch case.

Fast Branch does not apply to floating-point Branch instructions. Fast Branch also does not apply when the slot-a instruction is a TADDccTV, TSUBccTV, UMULcc, SMULcc, UDIVcc, or SDIVcc instruction, because these instructions force packet splitting.

Note that when a UMULcc, SMULcc, UDIVcc or SDIVcc instruction is being executed, the condition codes will NOT be available until after the instruction completes its Execute stages. The condition codes can be forwarded through the Cache and Writeback stages for these instructions. *Sections 3.8.3.1* and *3.8.3.2* provide a description of the pipeline stages for multiply and divide instructions.

3.5 Floating-Point Unit (FPU)

The hyperSPARC RT620 features a high-performance pipelined floating-point unit capable of launching one fp operation per cycle. The floating-point unit (illustrated in *Figure 3–5*) is comprised of a floating-point queue (FPQ), a floating-point register file, a floating-point status register (FSR), and two execution units. The two execution units are a 64-bit floating-point arithmetic unit (FAU) and a 64- x 32-bit Floating-Point Multiplier Unit, which enable single-precision addition, subtraction, and multiplication operations to execute in one cycle. To further enhance performance, the floating-point unit utilizes condition code forwarding to the integer unit to allow one-cycle FP compares.

The general operation of the floating-point unit is illustrated by *Figure 3–6,* which represents the high level state transitions for the floating-point unit. There are four states through which the floating-point unit transitions:

1. *EXECUTE.* This is the normal mode of operation of the floating-point unit.
2. *EXCEPTION PENDING.* The floating-point unit enters this state when an exception takes place in the floating-point unit on which a trap should be taken. It remains in this state until the integer unit acknowledges the exception.
3. *EXCEPTION.* The floating-point unit enters this state when a pending exception is acknowledged by the integer unit. In this state, only fp store instructions can be executed. The floating-point unit remains in this state as long as the queue not empty (qne) bit in the FSR is not clear or if an instruction other than a fp store instruction is executed.
4. *FLOATING-POINT UNIT FREEZE.* The floating-point unit enters this state when the integer unit signals a hold. The hold could be due to an external hold (e.g., cache miss) or an internal hold. All activities in the floating-point unit are frozen in this state.

Floating-Point Status Register

Floating-point register file
32-x32-bits
3 read ports: OPA, OPB, STORE
2 write ports: RESULT, LOAD

Floating-Point Queue and Controller

pre-launch queue

instr. A [32-bits]

instr. B [32-bits]

instr. A address [32-bits]

control [51-bits]

entry 4
entry 3
entry 2
entry 1

instr. decode, scheduling and control logic

Execute
Execute2
Round

post-launch queue

mux

store reg.

load reg.

OPA mux

OPB mux

Floating-point Add

Floating-point Multiply

result mux

Load/Store Bus (64 bits)

Figure 3–5. Floating-Point Unit Block Diagram

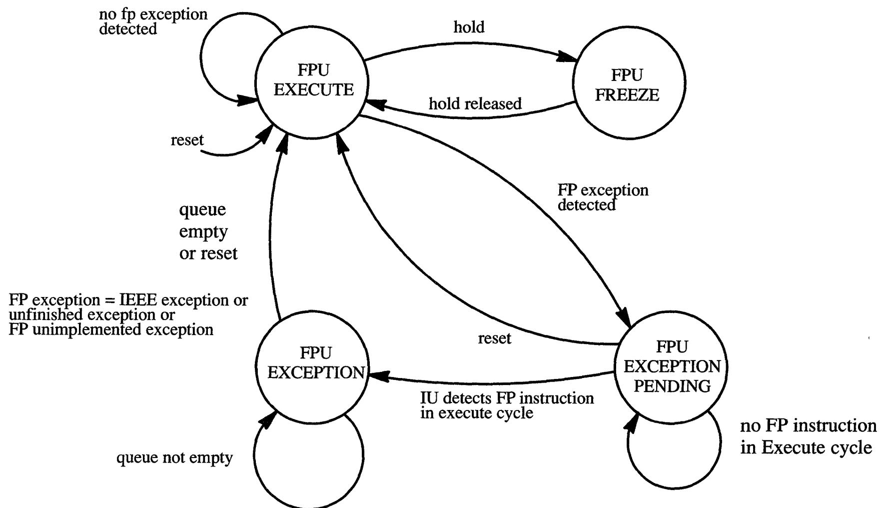


Figure 3–6. Floating-Point Unit State Transition Diagram

3.5.1 Floating-Point Instruction Decode-SCHEDule-and-Dispatch Controller (FPSCHED)

The IFETCH and ISCHED blocks provide instruction fetch and global decoding for the RT620. All instructions recognized as a floating-point instruction are forwarded to the floating-point instruction scheduler for local fp instruction decoding and fp instruction launch. The task of the FPSCHED is largely performed by the floating-point queue (FPQ) and the floating-point queue control (FPQC) blocks. The FPQ stores both instructions awaiting execution and those in the process of execution. The FPQC provides control for the FPQ, as well as local fp instruction decoding and execution scheduling. The following sections describe the FPQ and FPQC.

3.5.1.1 Floating-Point Queue (FPQ)

The floating-point queue (FPQ) is divided into two parts, a pre-queue and a post-queue, as illustrated in *Figure 3–7*. The post-queue consists of three queue entries corresponding to the three stages of the fp execution pipeline (Execute1, Execute2, and Round). The post-queue tracks instructions which have begun execution until an exception is detected or result generation is completed.

In order to support exception handling, the post-queue retains both the instruction address and a copy of the instruction as it passes through successive stages of the fp execution pipeline. Since the floating-point unit and integer unit pipelines operate somewhat independently, the exception detected by the floating-point unit is delayed with regards to the integer unit pipeline. The address of the exception causing fp instruction is used by trap handlers to determine the point in the instruction stream where the exception occurred.

The pre-queue is a performance enhancement which largely eliminates stalls of the RT620 due to the execution of multiple-cycle floating-point unit instructions. The pre-queue contains four entries, and behaves like an auxiliary set of instruction fetch buffers. When a series of fp instructions is fetched, the fp instructions are deposited in the pre-queue until execution, thereby allowing the integer unit to continue fetching and processing additional instructions.

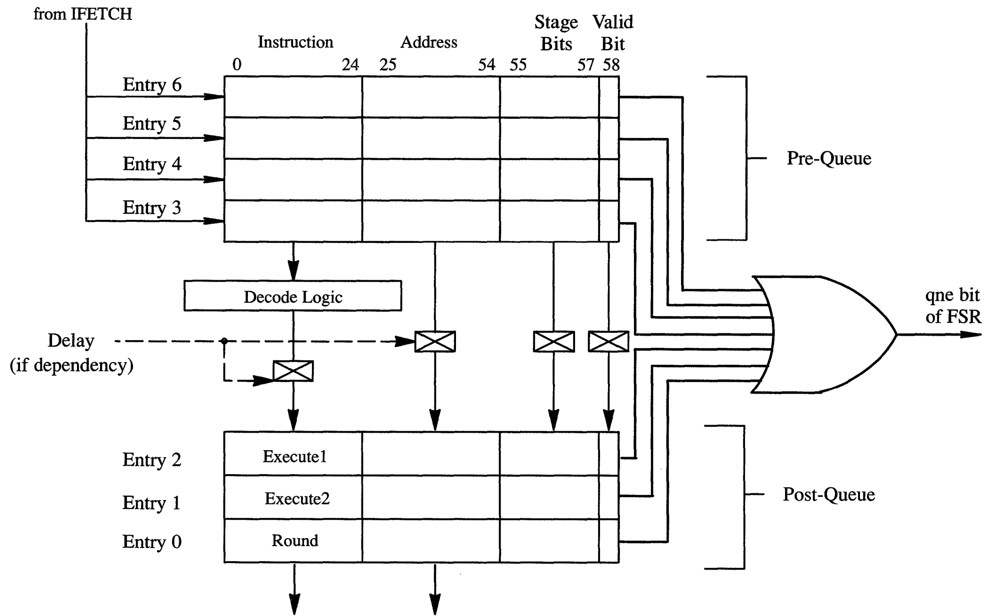


Figure 3–7. Floating-Point Queue (FPQ)

3.5.1.2 Floating-Point Queue Control (FPQC)

The FPQC provides the following functions :

- It provides FPQ management, including: queue advance, queue load and queue store.
- It decodes and launches fp instructions.
- It selects the appropriate fp operands for fp instructions. This includes forwarding any fp operands.
- It performs dependency checking against other fp instructions before launching an fp instruction.
- It interacts with the integer unit to perform dependency checking between fp instructions and fp Load and Store instructions.
- It directs loads and stores to and from the fp register file and the floating-point status register (FSR), and stores from the FPQ.
- It maintains the state of the FSR. The FPQC also forwards the floating-point condition codes (fcc) to the integer unit.
- It performs exception handling based on the status of the fp operations reported by the fp computational units and the FPQ.

3.5.1.2.1 FPQ Management

The floating-point unit unit operates in a pipelined manner. One fp instruction is launched per cycle assuming no constraints exist. There are three stages of execution for each fp instruction: Execute1, Execute2, and

Round. As the fp instruction advances through the pipeline stages, the corresponding entries in the queue also advance. When an fp instruction is launched for execution, it enters entry 2 in the FPQ (refer to *Figure 3–7*). This corresponds to the Execute1 stage of the fp instruction. When the instruction advances to the Execute2 stage, entry 2 is advanced to entry 1. Similarly, when the instruction advances to the Round stage, entry 1 advances to entry 0. When the instruction completes execution, it drops out of entry 0 and space is available for a new instruction. If an fp instruction requires more than three cycles for execution, the instruction is held in queue entry 2 for the duration of the extra cycles.

When an fp instruction is decoded in the integer unit, it is sent into the queue on the next cycle (if the queue is not full). If the pre-queue is empty, then the instruction enters post-queue entry 2 directly and goes into the Execute1 stage. If the instruction cannot be launched, it is stored in the pre-queue. Various conditions may preclude a launch:

- If there are instructions already present in the pre-queue, the new instruction cannot be launched. It is stored in the first available pre-queue entry (closest to entry 2).
- If the fp instruction in entry 2 requires multiple execution cycles, then entry 2 is not available for a new fp instruction and the new instruction must be delayed (see *Section 3.8.13*).
- Dependency constraints may prevent a launch even if the pre-queue is empty and entry 2 is available. In this case, the instruction is stored in entry 3.
- If an fp exception was detected and is awaiting acknowledgment from the integer unit, the floating-point unit will be in the floating-point unit exception pending state (refer to the state diagram in *Figure 3–6*). All fp execution is halted until the floating-point unit is returned to the Execute state.

As instructions are launched from the pre-queue, the other instructions in the pre-queue are advanced in a FIFO manner. As is apparent from the above discussion, instructions in the pre-queue are always launched for execution from entry 3. It is not possible to have an invalid queue entry between two valid queue entries in the pre-queue or vice-versa. It is possible to have holes in the post-queue if instruction launch is delayed due to dependencies or an instruction spends extended cycles in the Execute2 stage.

The queue entries are emptied as follows:

1. The fp instructions finish normal execution; the instructions are flushed out of entry 0 and the queue advances.
2. Instruction(s) in the queue are flushed due to an integer exception.
3. A STDFQ instruction is issued which causes a queue entry to be stored out to memory.

The FPQC provides queue-not-empty (qne) status information to the FSR. The qne bit in the FSR indicates if the queue is empty (qne is clear). From the programmer's perspective, the qne bit in the FSR indicates whether or not there are valid entries in the pre-queue and post-queue, but it does not indicate the number of valid entries or their *location*. The FPQC also provides the integer unit with two signals which indicate if the queue is full or has only 1 entry left. If the queue is full, the integer unit must stall upon fetching an fp instruction and hold them until the queue can accommodate the fetched instruction. If there is only 1 entry left and both slot-a and slot-b contain fp instructions, then the integer unit must split the packet and hold the slot-b instruction in the instruction fetch buffer. The integer unit provides the FPQC with a control signal to FLUSH instructions in the queue under certain exception conditions.

3.5.1.3 Dependency Checking before FP Instruction Launch

The FPQC is responsible for performing dependency checking on an fp instruction before it launches the instruction. The FPQC performs two kinds of dependency checks:

1. It checks for dependency against any instructions still executing in the post-queue.
2. It checks for dependency against any load instructions still executing and which occurred earlier in the instruction stream.

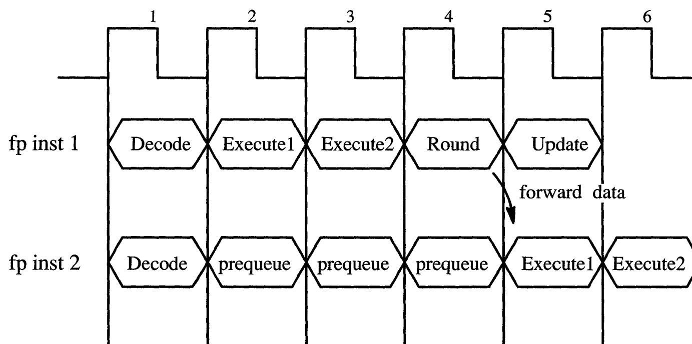
3.5.1.4 Floating-Point Data Forwarding

Forwarding can be classified into two types:

1. forwarding data to an fp instruction waiting to be launched.
2. forwarding data to a store instruction.

3.5.1.4.1 Forwarding to fp instruction

If an fp instruction (fp inst 2) is dependent on data which will be generated by an fp instruction in the post-queue (fp inst 1), the data is directly forwarded to fp inst 1 when it becomes available; fp inst 2 does not have to wait for the data to be written into the register file. However, as stated earlier, the precisions of the two instructions should be the same. *Figure 3–8* shows the timing for a case where two fp instructions are launched in parallel with an empty queue. The second instruction from the packet uses the result from the first instruction as one of its operands. The result from the first instruction is forwarded at the end of the Round stage and the second instruction enters execution in the next cycle.

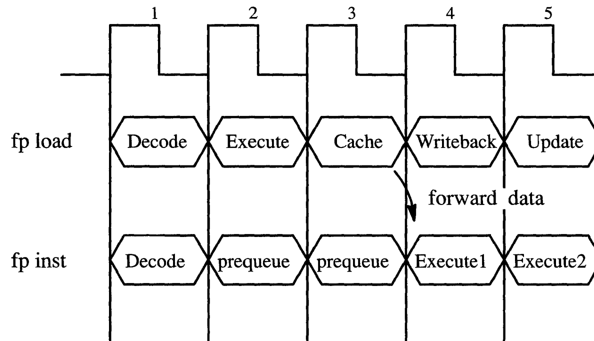


Case of fp instruction forwarding data to another fp instruction:

```
fp inst 1: fadds %x, %y, %z
fp inst 2: fadds %z, %a, %b
```

Note: fp inst 1 and fp inst 2 are in the same packet and the queue is empty

Figure 3–8. Forwarding between two FP Instructions



Case of fp load followed by fp instruction.

fp load: `ld [scr], %z`
fp inst: `fadds %z, %a, %b`

Note: The load instruction and the fp instruction are in the same packet and are launched in the same cycle.

Figure 3–9. Forwarding from Load to FP Instruction

3.5.1.4.2 Forwarding to Load Instruction

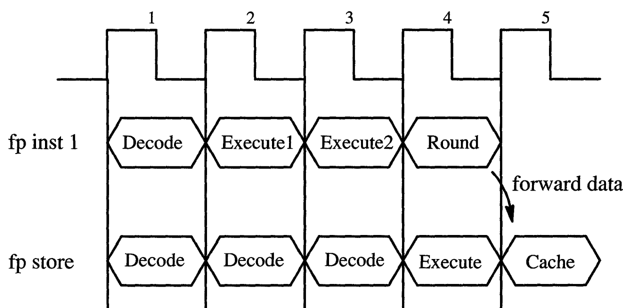
A fp instruction may also be dependent on data which is being loaded via an fp load instruction. Data from the Cache or Writeback stages of an fp load can be directly forwarded to the fp instruction. Since the instruction scheduler keeps track of the fp load execution, the FPQC communicates with the instruction scheduler to handle forwarding.

Figure 3–9 illustrates the case of an fp load instruction followed by an fp instruction with both the instructions launched in parallel. The data from the fp load instruction is forwarded from the Cache stage (when it becomes available) to the fp instruction so that the fp instruction enters execution in the next cycle. Until then, the fp instruction waits in the pre-queue.

3.5.1.4.3 Forwarding to Store Instruction

The fp store instruction requires its data operand to be available during the Cache stage of the fp store. If an fp store instruction is currently in Execute and its data is being generated by an fp instruction in entry 0, then the data is directly forwarded to the store in the next cycle (refer to *Figure 3–10*).

In this example, the fp instruction and the fp store instruction are in slot-a and slot-b respectively of the instruction cache. The source of the fp store operand is the result of the fp instruction. Since an intra-packet conflict exists, the instruction scheduler will split the packet and only launch the fp instruction. In subsequent cycles, the FPQC performs the inter-packet dependency checking. The FPQC signals the instruction decoder to delay the fp store instruction until the fp instruction enters the Round stage. The data from the fp instruction is forwarded to the fp store instruction in the Cache stage. Note that the fp store instruction is launched in the cycle in which the fp instruction enters the Round stage (before the fp instruction finishes computation of the result). Refer to *Section 3.9.8* regarding the case of an fp exception during this type of forwarding operation.



Case of fp instruction followed by fp store :

fp instruction: `fadds %x, %y, %z`
fp store: `st %z, [dest]`

Note: The fp instruction and the store instruction are in the same packet.

Figure 3–10. Forwarding from FP Instruction to Store

3.5.1.5 Floating-Point Load/Store Control

Control for fp load and fp store instructions is handled by the FPQC. The instructions include LDFSR, STFSR, LDF, STF, LDDF, STDF and STDFQ.

Load Instructions: The integer unit is responsible for generating the address for load instructions. The FPQ does the appropriate alignment of the received data. Data from the LDST bus is loaded into an internal register in the Writeback stage. It is then written from the internal register into the register file or FSR in the next (Update) stage. Thus the pipeline for fp loads is identical with the pipeline for integer unit loads.

A LDFSR instruction is always launched by itself. The scheduler forces inops for three cycles after launching a LDFSR, allowing enough time for the FSR update to take place.

Store Instructions (except STDFQ): When the instruction scheduler detects an fp store instruction, it signals the FPQC to provide the required data. The FPQC puts the data on the Load/Store bus during the Cache stage of the store. For a single precision store, the data is replicated on the upper and lower half of the Load/ Store bus.

A STFSR instruction is always launched by itself. STFSR instructions are held in Decode until all floating-point unit instructions have completed execution.

STDFQ Instructions: Each STDFQ instruction stores out a valid queue entry from the FPQ. The entries are stored out in the sequence they entered the queue (i.e., in a FIFO manner). The post-queue entries are stored out directly from their respective queue locations and the queue is not advanced. For example, if post-queue entries 0 and 1 are valid and a STDFQ is executed, then post-queue entry 0 will be stored out on the next cycle and will be marked invalid. On the next STDFQ, the next valid entry, post-queue entry 1, will be stored out and marked invalid. If all the post-queue entries are invalid and one or more pre-queue entries are valid, then the pre-queue is advanced on each STDFQ and the entries are stored out from entry 3. For example, if only pre-queue entries 3 and 4 are valid, then entry 3 will be stored out on a STDFQ, entry 4 will be advanced to entry 3 and entry 4 will be marked invalid and so on.

If a STDFQ instruction is executed when the queue is empty, an fp sequence error trap occurs (see *Section 3.9.8*). If the floating-point unit is in EXECUTE mode, a STDFQ instruction will be held in the instruction

cache until the queue becomes empty or the floating-point unit enters into EXCEPTION_PENDING mode. The state of the queue, empty or not empty, is indicated by the qne bit in the FSR.

A STDFQ instruction is always launched by itself (*f*group instruction). The scheduler forces inops for three cycles after launching a STDFQ allowing enough time for the qne bit in the FSR to be updated.

3.5.1.6 Floating-Point Unit Condition Codes:

The floating-point condition codes are reflected in the fcc field of the FSR. Since fp Branch instructions are handled by the integer unit, there are two lines that carry the current fcc values to the integer unit. The FPQC also sends another signal indicating if the fcc values are valid. When an fp compare instruction enters the queue, the FPQC marks the fcc values as invalid until valid condition codes are available.

The FPQC also forwards the fcc values from a fcmp instruction to the integer unit; it does not wait until the FSR is updated. Valid fcc values are available from the floating-point arithmetic unit in the Execute2 stage and are forwarded to the integer unit so that an fp Branch can be launched in the next cycle. The fcc values are not forwarded in the following cases:

- If there is a valid instruction in queue entry 0 or 1 and the instruction is an unimplemented instruction.
- If any of the queue entries 2–6 contains a compare instruction.
- If the compare instruction itself caused an exception.

3.5.2 Floating-Point Execution Units

The two floating-point execution units for the RT620 hyperSPARC are the floating-point arithmetic unit (FAU) and the floating-point multiplier unit (FMU). These execution units are used to implement all fp instructions of single and double precision, including all fp compare and type conversion instructions. Instructions involving extended (or quad) precision data types are not implemented, and will produce an unimplemented fp instruction trap when encountered by the floating-point unit.

The FAU handles all the SPARC fp arithmetic, compare and convert instructions. Appropriate conversion, alignment, addition, rounding, and IEEE-754 standard fp exception detection is performed by this unit in order to execute the assigned instructions. These instructions follow the typical fp execution pipeline. This pipeline involves Fetch, Decode, Execute1, Execute2, Round, and Update stages.

The FMU handles the SPARC fp multiply, divide, and square root instructions. Appropriate conversion, alignment, addition, rounding, and IEEE-754 standard fp exception detection is performed by this unit in order to execute the assigned instructions. Most of these instructions follow a multiple cycle fp execution pipeline. For additional information on the fp execution pipeline, see *Section 3.8.13*.

The floating-point functions of hyperSPARC conform to the IEEE floating-point arithmetic standard 754-1985. In addition to standard operation, the RT620 floating-point unit allows de-normalized operands to be treated as zero's when non-standard mode is enabled.

The following instructions are executed by the floating-point arithmetic unit:

FAU instructions: FMOVs, FABSs, FNEGs, FADDs, FADDd, FSUBs, FSUBd, FiTOs, FiTOd, FsTOi, FsTOd, FdTOi, FdTOs, FCMPs, FCMPd, FCMPEs, and FCMPEd.

The following instructions are executed by the Floating-Point Multiplier Unit:

FMU instructions: FMULs, FMULd, FsMULd, FDIVs, FDIVd, FSQRTs and FSQRTd.

Floating-point unit instructions are described in detail in *Chapter 12, SPARC Instruction Set.*

3.5.2.1 Non-standard mode

The floating-point unit may perform computations in one of two modes: standard and non-standard. Standard mode reflects adherence to the IEEE-754 standard. Non-standard mode reflects a slight departure from the standard mode for handling denormalized numbers. In non-standard mode, all denormalized operands are rounded to zero. This feature is occasionally desirable in some computation routines, and it has the advantage of avoiding underflow. Standard mode is achieved by clearing the NS bit (bit 22) in the FSR. Non-standard mode is achieved by setting the NS bit.

3.6 instruction Cache (ICACHE)

The RT620 instruction cache is organized as a two-way set associative cache, organized as two sets of 128 line entries. Each line entry consists of four instruction packets (each packet contains two instructions), a 20-bit tag, a supervisor bit, and four valid bits to indicate the validity of each instruction packet in the ICACHE line (refer to *Figure 3–11*). Up to 2048 instructions can be held in the instruction cache.

Each instruction fetch for the RT620 is simultaneously requested from both the ICACHE, and the IBIU, which in turn places the request on the RT620 address lines. In this manner, if an ICACHE miss occurs and the external cache contains the instruction, the miss penalty is only one clock cycle. If an ICACHE hit occurs, the $\overline{\text{PNULL}}$ signal is used to cancel the request already on the bus for the external cache subsystem.

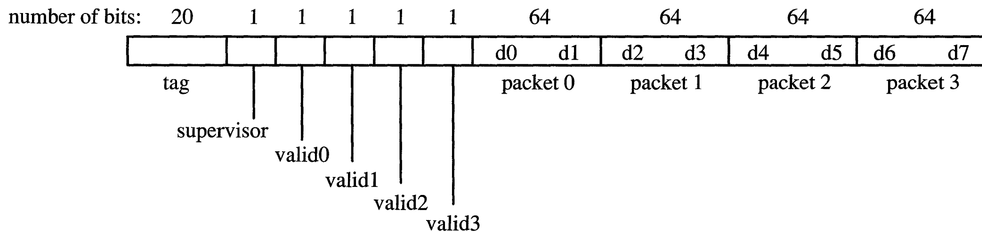


Figure 3–11. ICACHE Line Organization

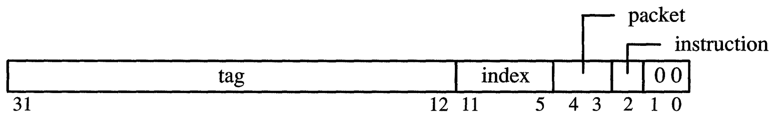


Figure 3–12. Virtual Address Format

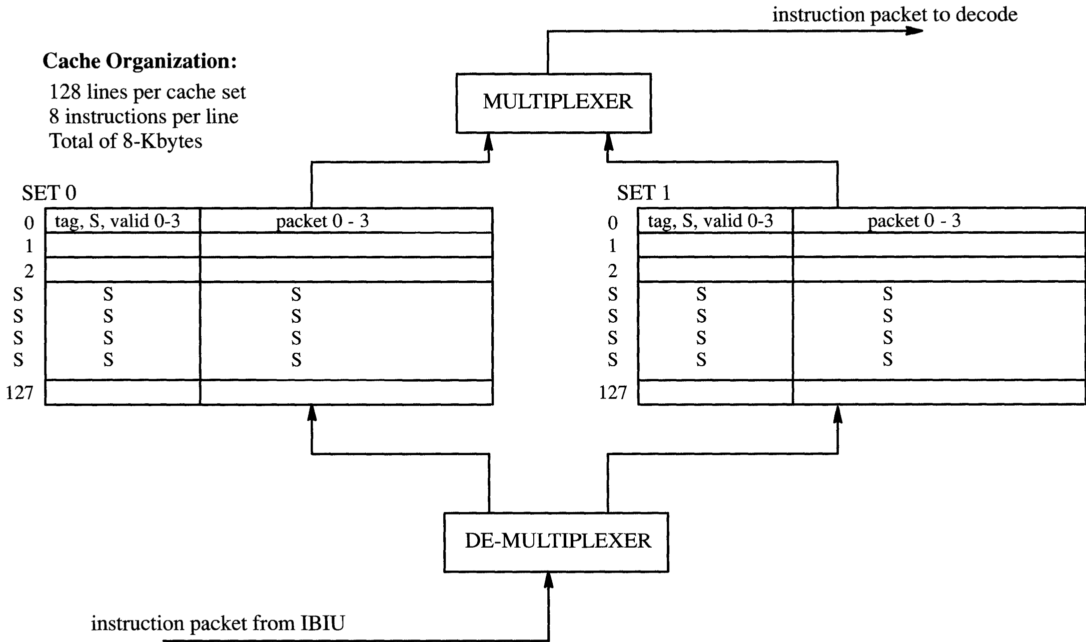


Figure 3–13. ICACHE Organization

3.6.1 Virtual Addresses

Figure 3–12 illustrates the fields of the virtual address as they are decoded by the ICACHE. The two sets of the cache are directly indexed by bits 11 – 5 of the virtual address. These seven index bits select one of the 128 lines in both cache sets. Bits 31 through 12 of the virtual address are used for tag comparisons with the two cache lines selected by the index bits. Bits 4 and 3 are used to determine which packet is being accessed for the indexed cache line. Bit 2 is used to determine which instruction within a packet is being accessed for the indexed cache line.

Alternatively, the encoding derived from bits 4 through 2 (binary values 000 through 111) also identify instructions within the cache line as d0 through d7, respectively. From this alternative viewpoint, packet 0 corresponds to instructions d0 and d1, packet 1 corresponds to instructions d2 and d3, packet 2 corresponds to instructions d4 and d5, and packet 3 corresponds to instructions d6 and d7.

3.6.2 ICACHE Hits

An *ICACHE hit* is generated when there is a tag match, and a privilege match and a packet match or a partial packet match. A *ICACHE miss* takes place if there is no ICACHE hit.

- A *tag match* occurs when the tag field from the virtual address compares successfully against one of the tags in the two sets of the indexed cache line (determined from the index bits of the address). If there is no tag match, a *tag mismatch* occurs, which is sufficient to cause a cache miss.
- A *privilege match* is determined by the comparison of the supervisor bit in the PSR against the access privilege for the indexed cache line as shown in the table below. If there is no privilege

match, there is a *privilege mismatch.* A privilege mismatch is sufficient to cause an ICACHE miss.

Table 3–3. Cache Line Privilege Match

S bit in PSR	S bit in cache line	privilege match
1	ignored	1 (always matched)
0	0	1
0	1	0

- The S bit in the cache line is set at the time the instruction packet is fetched and written into the ICACHE. The S bit is set to 1 if the instruction fetch access is privileged (i.e., the PSR supervisor bit is set) and cleared otherwise.
- A *packet match* occurs when:
  - The access to the cache line is aligned on the packet boundary of the cache line (i.e., when A2 = 0) and the valid bit for that packet is set.
  - The access to the cache line is on an odd-word boundary (A2 = 1) and valid bits for both the packets (in which the two instructions being accessed are contained) are set.
- Instruction fetch from off-chip sources is performed on doubleword boundaries. However, instructions from the ICACHE can be fetched on word boundaries. The instructions to be accessed from the indexed cache line are selected by the A4, A3 and A2 bits. Therefore, it is possible for the selected instructions to straddle two packets of the cache line.

When a cache miss is generated, the instruction access request will be placed on the IMB through the IBIU. If the miss was caused by a privilege mismatch, the RT625 CMTU will detect the attempted access to the privileged address space. This can cause a memory access exception to be generated if the external access is also tagged as a privileged address space.

When a partial packet hit occurs as the result of an odd instruction word access, (i.e., the instruction packet requested straddles a doubleword boundary), the available instruction is forwarded to the integer unit, instead of delaying the packet until the even instruction word is available.

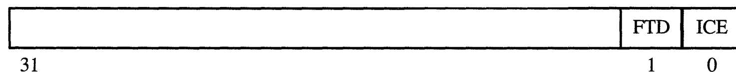


Figure 3–14. ICCR Register

3.6.3 Instruction Cache Control Register

The ICACHE control register (ICCR) is an ancillary state register provided for control of the on-chip instruction cache. Access to this register is privileged (i.e., only accessible if the PSR supervisor bit is set). It is accessed using the RDASR (read ancillary state register) and WRASR (write ancillary state register) instructions. If the PSR supervisor bit is not set and a read or write to the register is attempted, a privilege exception occurs. The appropriate rd (destination register) or rs1 (source register 1) field must be set to 31 (0x1f) in order to access this register.

The instruction cache enable (ICE, ICCR < 0 >) bit enables ICACHE accesses if it is set and disables accesses if it is cleared.

The flush trap disabled (FTD, ICCR < 1 >) bit determines whether an ICACHE sub-block (packet) will be flushed (invalidated) or an unimplemented flush trap (tt=25) will be taken when a FLUSH instruction is executed.

When FTD = 1, if a FLUSH instruction is executed and an ICACHE hit occurs, the packet corresponding to the address in the cache line is invalidated. When FTD = 0, if an attempt is made to execute a FLUSH instruction, an unimplemented FLUSH exception is taken. The purpose of the FTD bit is to support self-modifying code in a symmetric multiprocessing environment.

Upon power-on reset, the instruction cache is disabled (the ICE bit is cleared) and Flush Traps are enabled (the FTD bit is cleared). Also during power on, all entries in the instruction cache are invalidated. Writes to bits other than the FTD and ICE bits are ignored. Bits other than FTD and ICE are forced to 0.

A *partial cache line hit* occurs when one of the two sets has a tag match and a supervisor match, and there are one or more valid packets in that line. A *partially valid cache line* is one in which there is no tag match but one or more packets are valid in that line. When an ICACHE miss occurs and the ICACHE is enabled, the new instruction fetched from the external cache must be written into the ICACHE. The replacement strategy is based on determining whether there is equal eligibility for replacement between the two cache sets or not.

CASE 1: In the case where neither set has any valid packets (i.e., all valid bits in the line are clear), they are equally eligible for replacement. The replacement bit (see below) determines the set into which the newly fetched packet is written.

CASE 2: In the case where one of the two sets has a tag match and there are one or more valid packets in that line, the line is partially valid and the packet must be written into this set.

CASE 3: In the case where there is no tag match in either set but one set has one or more valid packets, then the set containing valid packets must be preserved and the newly fetched packet must be written into the other set.

CASE 4: Finally, in the case where there is no tag match and both sets' cache lines contain one or more valid packets, the replacement bit determines into which set the newly fetched instruction is writtten.

Whenever an ICACHE line is being overwritten with a new tag, a new supervisor bit, and its first valid packet, the remaining packet valid bits must be cleared. This must be done to avoid generating false ICACHE hits during subsequent instruction fetches.

The replacement bit is set and cleared as follows:

- When there is an ICACHE hit, the replacement bit does not change.
- If there is an ICACHE miss for the instruction fetch but there is a partial cache line hit. In this case, the replacement bit is forced to the set in which the partial cache line hit occurred, and replacements are now performed in that set.
- If there is an ICACHE miss and no packets are valid in the current set, the replacements will occur in the set indicated by the replacement bit. The replacement bit is unchanged.
- If there is an ICACHE miss and no tag match but any packets are valid in this set, then the replacement bit changes to the opposite set, and replacements are performed in that set.
- The replacement bit is cleared on reset.

3.6.4 On-Chip Instruction Cache Read/Write Diagnostic Support

There are several instruction cache diagnostic commands defined which allow access to the tag/valid/supervisor bits and the contents of the ICACHE blocks (lines). These diagnostic instructions are only accessible when the instruction cache is disabled. Access to the ICACHE is made using load and Store Alternate instructions. The two following subsections describe these actions.

3.6.4.1 ICACHE Tag Access Address Mapping for Diagnostic Support

For ICACHE tag access, use LDA and STA instructions (word size transactions only) with ASI = 0x0c.

Virtual Address Bits:

31 – 13	12	11 – 5	4 – 0
0000...0000	set number	tag entry number	XXXXX

	VA	Entry		VA	Entry
Set 0	0x0000000X	entry 0	Set 1	0x0000100X	entry 0
	0x0000002X	entry 1		0x0000102X	entry 1
	0x0000004X	entry 2		0x0000104X	entry 2
	...			...	
	0x00000fcX	entry 126		0x00001fcX	entry 126
	0x00000feX	entry 127		0x00001feX	entry 127

3.6.4.2 ICACHE Sub-Block Access (Data) Address Mapping for Diagnostic Support

For ICACHE data access, use LDDA and STDA instructions (doubleword size transactions only) with ASI = 0x0d.

Virtual Address Bits:

31 – 13	12	11 – 5	4 3	2 – 0
0000...0000	set number	tag entry number	sub-block	000

	VA	Entry		VA	Entry
Set 0	0x00000000	entry 0: sub-block 0	Set 1	0x00001000	entry 0: sub-block 0
	0x00000008	entry 0: sub-block 1		0x00001008	entry 0: sub-block 1
	0x00000010	entry 0: sub-block 2		0x00001010	entry 0: sub-block 2
	0x00000018	entry 0: sub-block 3		0x00001018	entry 0: sub-block 3
	0x00000020	entry 1: sub-block 0		0x00001020	entry 1: sub-block 0
	... ...			... ...	
	0x00000fe0	entry 127: sub-block 0		0x00001fe0	entry 127: sub-block 0
	0x00000fe8	entry 127: sub-block 1		0x00001fe8	entry 127: sub-block 1
	0x00000ff0	entry 127: sub-block 2		0x00001ff0	entry 127: sub-block 2
	0x00000ff8	entry 127: sub-block 3		0x00001ff8	entry 127: sub-block 3

3.6.4.3 ICACHE FLUSH Instructions

Two types of FLUSH instructions are available for the RT620: the FLUSH instruction, and Store Alternate (STA) instructions using ASI values reserved for FLUSH actions. *Table 3–4* lists the various FLUSH instruction combinations and the corresponding actions.

FLUSH instructions may be used to flush individual instruction packets, individual ICACHE lines, or the entire ICACHE. In order to flush an ICACHE line, all valid bits in the line must be cleared. Flushing an ICACHE line by flushing individual instruction packets requires a sequence of four FLUSH instructions, each of which corresponds to an instruction packet in a particular ICACHE line.

Flushing an entire ICACHE line involves executing a Store Alternate word (STA) instruction using an ASI of 0x10 – 0x14 (which flushes both the ICACHE and external cache), or ASI = 0x18 – 0x1C (which flushes

the ICACHE only). The effective address tag must match the tag in the indexed entry in one of the two sets in order for the line flush to occur. Only the ICACHE line in the matching set will be flushed.

The entire ICACHE may be invalidated by using a Store Alternate word instruction with an ASI of 0x31. Both sets of the ICACHE are affected, and the effective address for the FLUSH instruction is ignored.

Table 3–4. FLUSH instructions

Instruction	PSR.S	ICCR.FTD	ICCR.ICE	ICACHE_ hit	Memory Misalign	Action	Memory Access Required?	PNULL
FLUSH	x	1	1	0	x	nop	no	na
				1	x	FLUSH	no	na
			0	x	x	nop	no	na
		0	x	x	x	unimp flush (tt = 0x25)	no	na
STA ASI= 0x10 . . 0x14 (on-chip and external cache)[2]	1	x	1	0	0	nop	yes	no
					1	unpredictable[1]		
				1	0	flush_line	yes	no
					1	unpredictable[1]		
			0	x	0	nop	yes	no
					1	unpredictable[1]		
	0	x	x	x	x	priv violation (tt = 0x3)	no	na
STA ASI = 0x18 . . 0x1c (on-chip)[2]	1	x	1	0	0	nop	no	na
					1	unpredictable[1]		
				1	0	flush_line	no	na
					1	unprecitable[1]		
			0	x	0	nop	no	na
					1	unpredictable[1]		
	0	x	x	x	x	priv violation (tt = 0x3)	no	na
STA ASI = 0x31[1]	1	x	x	x	x	flush all	no	na
	0	x	x	x	x	priv violation (tt = 0x3)	no	na
LDA/STA ASI = 0xc LDDA/STDA ASI = 0xd[3]	1	x	0	x	x	read/wrt test	no	na
			1	x	x	unpredictable[1]		
	0	x	x	x	x	priv violation (tt = 0x3)	no	na

Notes:

1. Use of these cases is discouraged. Programmers should avoid unpredictable cases by prohibiting the use of misaligned addresses. In the RT620 implementation, the results are unpredictable and not guaranteed to behave in any specific way.
2. Defined only for the STA instruction. Other instructions (such as LDUBA, LDSBA, etc.) generate unpredictable results.
3. LDDA/STDA with ASI = 0xd causes the data area of the ICACHE to be read/written; LDA /STA with ASI = 0xc causes the tag area of the ICACHE to be read/written; other types of loads and stores (e.g., Atomic LDSTUB, STH, etc.) result in unpredictable results.

Programming Note: The diagnostic ICACHE access instructions (LDA or STA with ASI = 0xc, and LDDA or STDA with ASI = 0xd) will result in unpredictable results in the following cases:

1. The instruction is executed while ICACHE is enabled.
2. The instruction is one of three instructions following the execution of a WRPSR or WRASR *%iccr* instruction with ICACHE disabled.

3.7 hyperSPARC Signal Descriptions

3.7.1 hyperSPARC CPU Pinouts

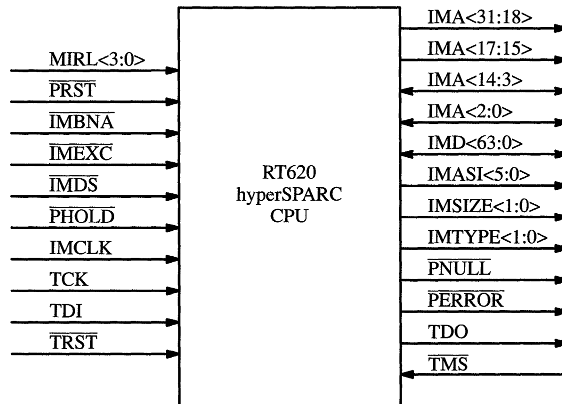


Figure 3–15. RT620 Signals

IMCLK — (input) IMB Clock

This is the basic clock for all the Intra-Module components. All the Intra-Module signals are driven and sampled on only the rising edge of the IMCLK. The RT620 uses only the IMCLK clock (not the MBus clock). The rising edge of IMCLK defines the beginning of each pipeline stage. The processor cycle is equal to a full clock cycle.

IMA < 31:18 > — (output, bi-state) Intra-Module Address Bus

IMA < 17:15 > — (output, tri-state) Intra-Module Address Bus

IMA < 14:3 > — (input/output, tri-state) Intra-Module Address Bus

IMA < 2:0 > — (input/output, bi-state) Intra-Module Address Bus

The 32-bit address bus carries instruction or data address during a fetch or Load/Store operation. Addresses are sent out unlatched and must be latched external to the RT620. The address on the IMA < 31:0 > is a virtual address.

IMD < 63:0 > — (input/output, tri-state) Intra-Module Data Bus

These pins form a 64-bit bi-directional data bus that serves as the interface between the CPU and memory.

Store data is sent out unlatched (and is latched by the RT625). Alignment for load and store instructions is performed by the RT620. Double words are aligned on 8-byte boundaries, words on 4-byte boundaries, and halfwords on 2-byte boundaries.

IMASI < 5:0 > — (output, bi-state) Intra-Module Address Space Identifier

These 6 bits constitute the address space identifier (ASI). The ASI identifies the memory address space to which the instruction or data access is being directed. The IMASI bits are sent out unlatched simultaneously with the address (and are latched by the RT625).

The following table describes IMASI generation:

Bus Activity	IMASI
instruction fetch	001000 (binary) + supervisor bit
Load/Store access	001010 (binary) + supervisor bit
Load/Store alternate	ASI immediate field

IMSIZE < 1:0 > — (output, bi-state) Intra-Module Access SIZE

These two pins indicate the SIZE of the current access. Instruction accesses are always doubleword size. Because the data bus is 64-bits wide, doubleword accesses can be performed in a single access. The value of the size bits during a given cycle relates only to the address which appears on pins IMA<31:0>. The IMSIZE<1:0> bits are sent out unlatched (and are latched by the RT625). Size values are defined as follows:

Bus Activity	SIZE < 1:0 >
instruction fetch	11 (binary)
Load/Store double	11 (binary)
Load/Store word	10 (binary)
Load/Store halfword	01 (binary)
Load/Store byte	00 (binary)

IMTYPE < 1:0 > — Intra-Module Access TYPE

- IMTYPE < 1 > — (output, bi-state)
- IMTYPE < 0 > — (output, tri-state)

These two pins indicate the current access TYPE. Instruction accesses are always type read. The value of the type bits during a given cycle relates only to the address which appears on pins IMA<31:0>. The IMTYPE bits are sent out unlatched (and are latched by the RT625). Type values are defined as follows.

IMTYPE < 1 >	IMTYPE < 0 >	Meaning
0	0	Normal write (store)
0	1	Normal read (load or instruction fetch)
1	0	Locked write (atomic store)
1	1	Locked read (atomic load)

$\overline{\text{PHOLD}}$ — (input) processor HOLD

This signal is used by the CMTU to hold the processor and prevent pipeline advancement. When the $\overline{\text{PHOLD}}$ input signal is asserted, all pipelines in the RT620 are frozen.

$\overline{\text{IMDS}}$ — (input) Intra-Module Data Strobe

This signal is generated by the CMTU to strobe the data into the CPU whenever the valid data is available while the processor is being held. The data strobe is valid only if $\overline{\text{PHOLD}}$ is asserted during the cycle that precedes the $\overline{\text{IMDS}}$ occurrence.

$\overline{\text{IMEXC}}$ — (input) Intra-Module Exception

This signal is generated by the CMTU to indicate an exception condition to the CPU.

$\overline{\text{IMBNA}}$ — (input) Intra-Module Bus Not Available

This signal is generated by the CMTU to indicate that the CMTU is using the Intra-module Bus. This feature allows data in a missed cache line to be filled as a background task while the processor continues executing instructions from the on-chip instruction cache. The processor can resume instruction execution once the CMTU has provided the data which caused the processor hold.

$\overline{\text{PNULL}}$ — (output, bi-state) Processor Nullify

The RT620 asserts $\overline{\text{PNULL}}$ to indicate that the current external cache access is being nullified. Asserting $\overline{\text{PNULL}}$ nullifies the access already latched by the external cache subsystem.

$\overline{\text{PNULL}}$ is generated under the following conditions:

1. An on-chip instruction cache hit occurred.
2. An on-chip instruction cache miss occurred and a Fast Branch was not taken.
3. An exception is pending.
4. No instruction fetch is pending and no load or store instruction is pending.

$\overline{\text{PERROR}}$ — (output, bi-state) Processor ERROR

This pin is asserted when the processor is in the ERROR mode, which occurs when a synchronous trap is encountered while traps are disabled (i.e., the PSR's ET bit = 0). The only way to restart a processor which is in the error mode state is to trigger a reset by asserting the $\overline{\text{PRST}}$ signal.

$\overline{\text{PRST}}$ — (input) Processor Reset

Assertion of this pin will reset the RT620. $\overline{\text{PRST}}$ must be asserted for a minimum of eight processor clock cycles. After $\overline{\text{PRST}}$ is de-asserted, the processor starts fetching from virtual address 0. $\overline{\text{PRST}}$ is latched by the RT620 before it is used.

MIRL < 3:0 > — (input, asynchronous) MBus Interrupt Request Levels

MIRL<3:0> indicate the interrupt request level. If traps are enabled, this value is compared against the PSR processor interrupt level field to determine if the interrupt should be acknowledged. MIRL<3:0> are synchronized by the RT620 for two clocks.

The state of these four pins defines the external interrupt level. MIRL < 3:0 > = 0000 indicates that no external interrupts are pending and is the normal state of the MIRL pins. MIRL < 3:0 > = 1111 signifies a non-maskable interrupt.

TCK — Test Clock

This is a test access port (TAP) clock signal that is independent of the IMCLK. The changes on TAP input signals ($\overline{\text{TMS}}$ and TDI) are clocked into the TAP controller, instruction register, or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) also occur on the rising edge of TCK.

TDI — Test Data Input.

TDI is a TAP input that is clocked into the selected register (instruction or data) on a rising edge of TCK. The TDI input has a built-in pull-up resistor which ensures that an un-terminated or open input is seen by the test logic as a "1."

TDO — Test Data Output.

TDO is a TAP serial data output. The contents of the selected register (instruction or data) are shifted out of TDO on the falling edge of TCK. TDO is set to high-impedance except when scanning of data is in progress.

$\overline{\text{TMS}}$ — Test Mode Select.

This control input is clocked into the TAP controller on the rising edge of TCK. The $\overline{\text{TMS}}$ input has a built-in pull-up resistor which ensures that an un-terminated or open input is seen by the test logic as a '1.'

$\overline{\text{TRST}}$ — Test Reset.

$\overline{\text{TRST}}$ initializes the state of the instruction register bits and the TAP controller state machine.

3.7.2 Intra-Module Bus (IMB)

The IMB is a high-speed 64-bit synchronous processor bus specifically designed for interface with the hyperSPARC external cache (refer to *Figure 3–16*). The hyperSPARC external cache (referred to as the *e-Cache*) is comprised of two or four RT627 CDUs controlled by one RT625 CMTU. The IMB is a synchronous data bus designed to transfer one 64-bit data word upon every processor clock (IMCLK) cycle. All data and control signals are sampled on the rising IMCLK edge.

During normal operation, the RT620 asserts IMA<31:0>, IMASI<5:0>, IMSIZE<1:0>, and IMTYPE<1:0> for all instruction fetches, regardless of whether the instruction cache is enabled. If the instruction is found in the ICACHE, the instruction request is nullified by the RT620 by asserting the $\overline{\text{PNULL}}$ signal. This causes the RT625 to deassert the cache read output enable ($\overline{\text{CROE}}$) signal, which disables the outputs of the cache RAM. $\overline{\text{PNULL}}$ remains asserted as long as the RT620 ICACHE contains the required instructions. If the ICACHE misses on an instruction request, $\overline{\text{PNULL}}$ is released and the RT625 asserts $\overline{\text{CROE}}$. Asserting $\overline{\text{CROE}}$ allows the e-Cache to respond to the address supplied by the RT620.

If the RT625 determines that the address supplied on the IMB is not in the e-Cache, the RT625 asserts the signal $\overline{\text{PHOLD}}$, which freezes the RT620 processor pipeline. An example of this is illustrated in the timing diagram *Figure 3–23* on page 3-43. In addition to $\overline{\text{PHOLD}}$, the RT625 asserts the $\overline{\text{IMBNA}}$ signal in order to force the RT620 to release the IMB address and data buses, and the IMTYPE<0> signal. This allows the RT625 to drive the IMB to transfer data into the RT627 Cache Data Unit. $\overline{\text{PHOLD}}$ is held until the RT625 has fetched the data originally requested by the RT620.

Upon assertion of the $\overline{\text{IMBNA}}$ signal, the RT620 tri-states IMA<17:0>, IMD<63:0>, and IMTYPE<0>. These signals are directly connected to the RT627 CDUs, and therefore must be tri-stated in order for the

RT625 to transfer data. IMA<31:18> and IMTYPE<1> are direct inputs to the RT625, and remain driven when $\overline{\text{IMBNA}}$ is asserted.

The hyperSPARC supports data forwarding during cache line fetches. If the RT620 is stopped with $\overline{\text{PHOLD}}$ due to a Cache read miss, $\overline{\text{PHOLD}}$ is released when the requested data is placed on the IMB by the RT625. In addition, the $\overline{\text{IMDS}}$ data strobe is asserted for the clock edge upon which the missed data is valid on the IMB. This allows the RT620 to latch the data on the IMB as it is transferred to the RT627 CDUs. The $\overline{\text{IMBNA}}$ signal remains asserted by the RT625 until the entire cache line is transferred to the cache.

Memory exceptions are signaled by a one-clock assertion of the $\overline{\text{IMEXC}}$ by the RT625. This is illustrated in *Figure 3–27* on page 3-51. The RT620 responds by asserting the $\overline{\text{PNULL}}$ signal, thereby nullifying the instruction or data requests following the memory exception. The RT620 enters a trap routine and begins fetching trap instructions after flushing the pipeline.

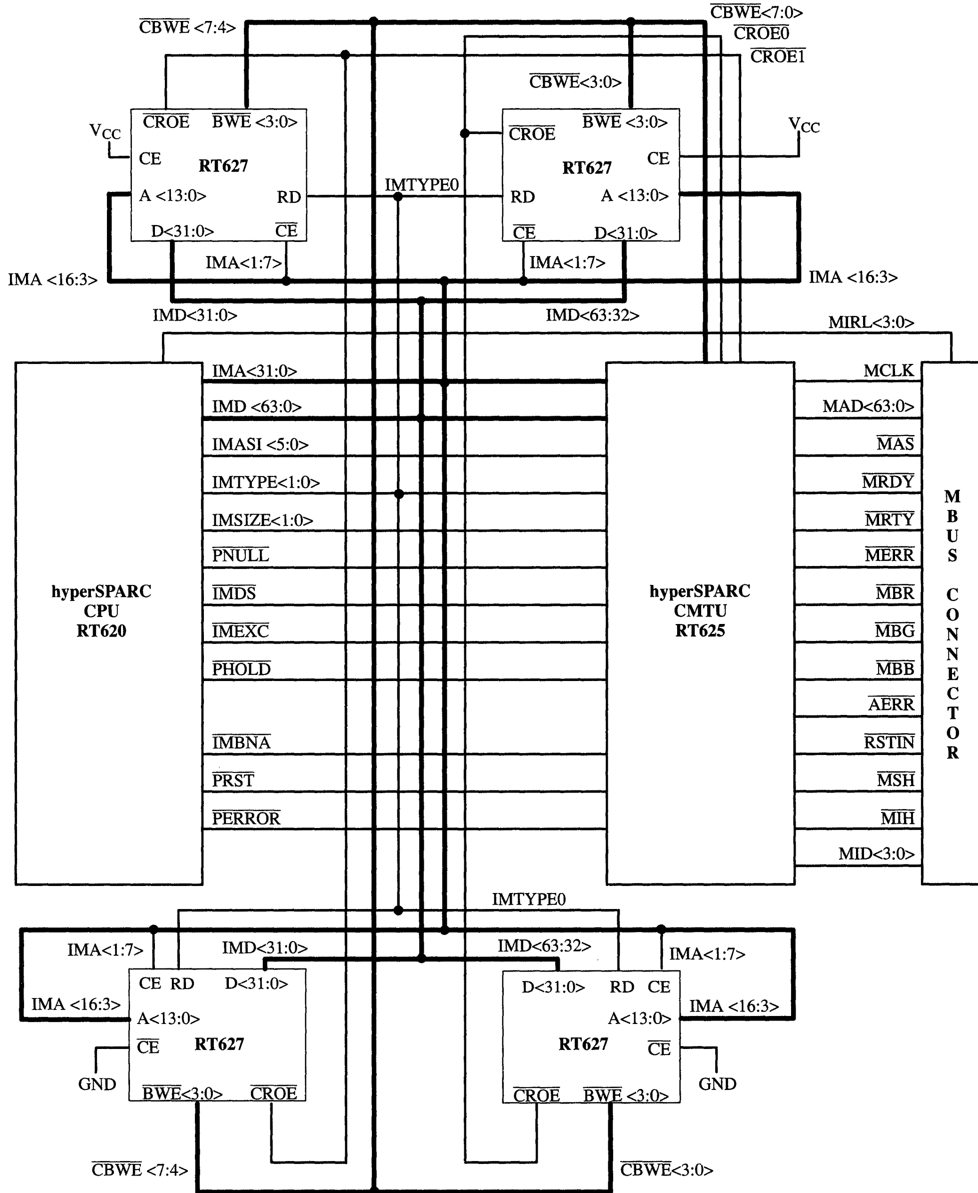


Figure 3–16. hyperSPARC CPU with 256-Kbyte Cache

3.7.3 hyperSPARC CPU Bus Timing Waveforms

This subsection supplies a sampling of bus timing diagrams related to read, write, atomic read-write, memory exceptions, reset, interrupt, traps, and error mode. These diagrams represent a subset of possible combinations of bus activities.

3.7.3.1 List of Figures: page

Figure 3–17. Instruction Access with Instruction Cache Hit 3-37
Figure 3–18. Read Accesses with External Cache Hit 3-38
Figure 3–19. Write Accesses with External Cache Hit 3-39
Figure 3–20. Atomic Read-Write Accesses with External Cache Hit 3-40
Figure 3–21. Write Access Followed by Read Access with External Cache Hit 3-41
Figure 3–22. Write Access Followed by Write Access with External Cache Hit 3-42
Figure 3–23. Read Access with External Cache Miss 3-43
Figure 3–24. Write Access with External Cache Miss 3-45
Figure 3–25. Atomic Read-Write Access with External Cache Miss 3-47
Figure 3–26. Data Read Access with Memory Exception 3-49
Figure 3–27. Instruction Read Access with Memory Exception 3-51
Figure 3–28. Reset Timing 3-53
Figure 3–29. Interrupt Timing 3-56
Figure 3–30. Error Timing 3-57

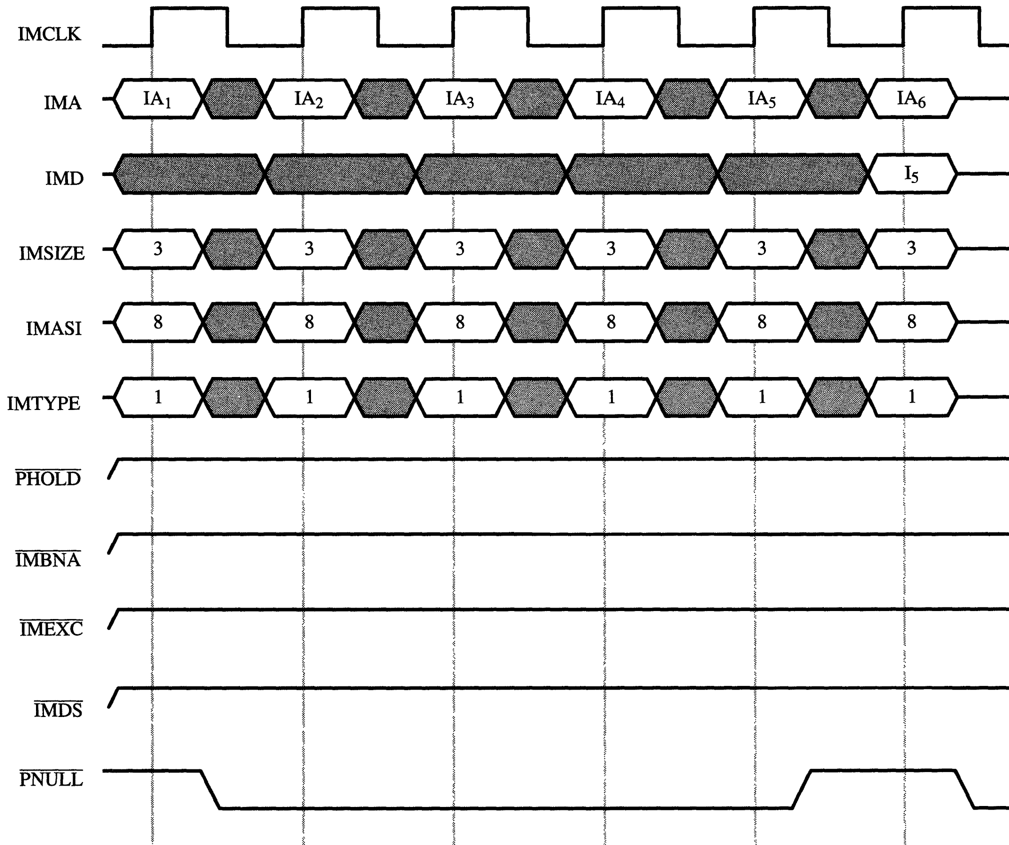


Figure 3–17. Instruction Access with Instruction Cache Hit*

* This series of read activities corresponds to . . .
four user instruction reads (Fetches) that results in an internal instruction cache hit
followed by one user instruction read (Fetch) that results in an external cache hit.

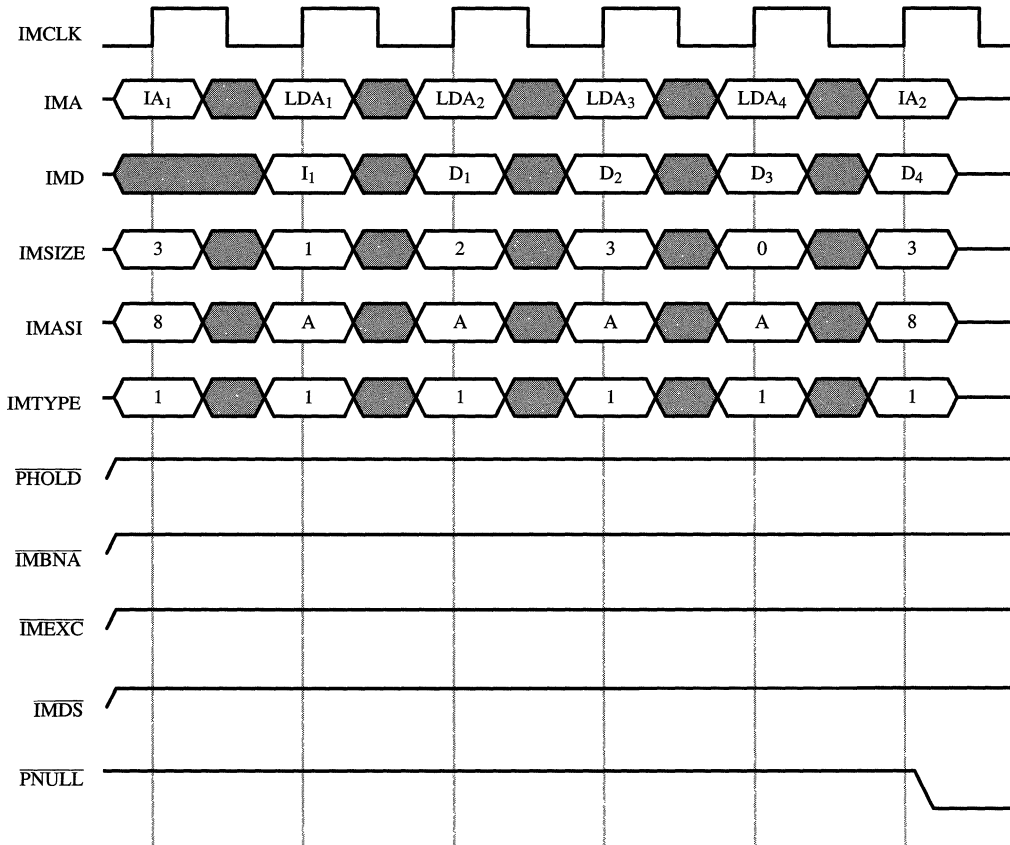


Figure 3–18. Read Accesses with External Cache Hit*

* This series of read activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user halfword read (e.g., LDSH or LDUH) that results in an external cache hit followed by
a user word read (e.g., LD or LDF) that results in an external cache hit followed by
a user doubleword read (e.g., LDD or LDDF) that results in an external cache hit followed by
a user byte read (e.g., LDSB or LDUB) that results in an external cache hit followed by
a user instruction read

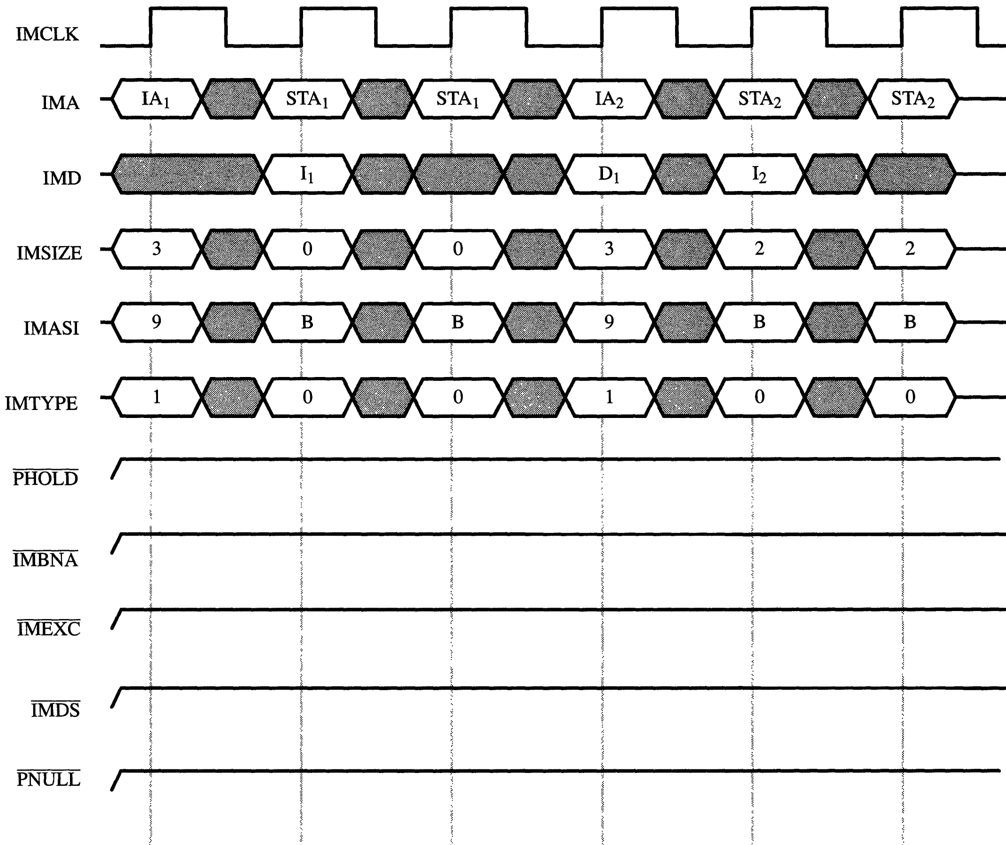


Figure 3–19. Write Accesses with External Cache Hit*

* This series of read/write activities corresponds to . . .
a supervisor instruction read (Fetch) that results in an external cache hit followed by
a supervisor byte write (e.g., STB) that results in an external cache hit followed by
a supervisor instruction read (Fetch) that results in an external cache hit followed by
a supervisor word write (e.g., ST or STF) that results in an external cache hit.

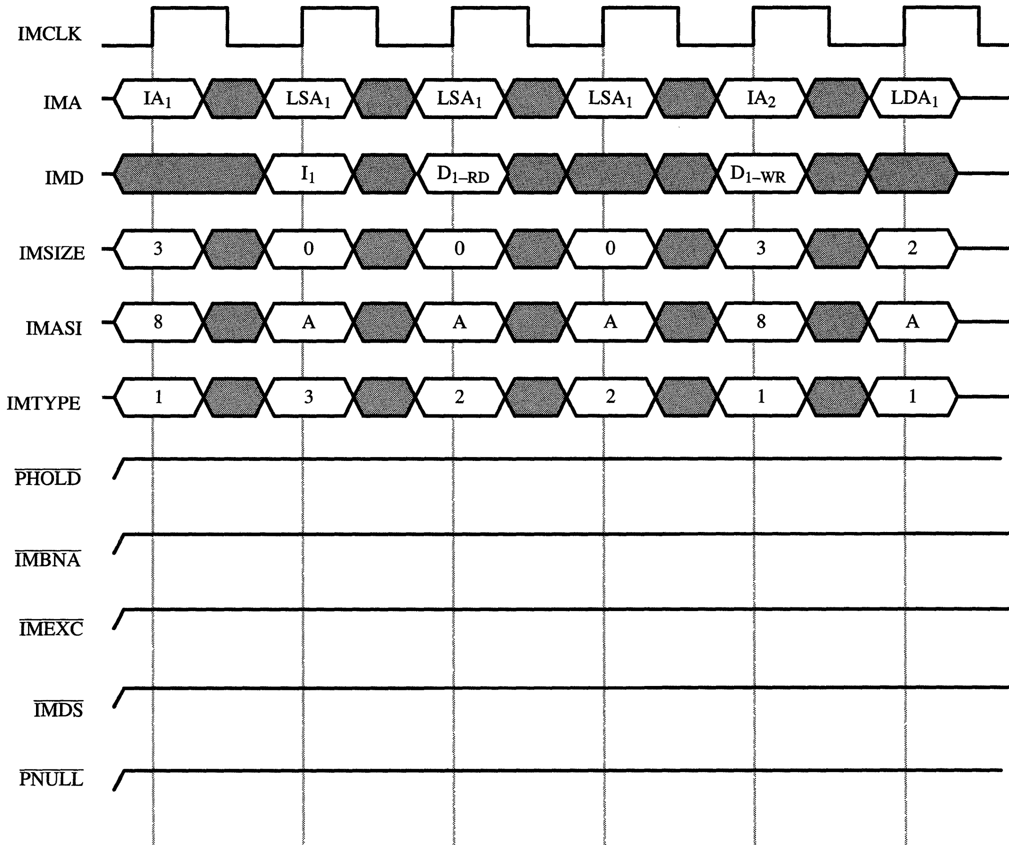


Figure 3–20. Atomic Read-Write Accesses with External Cache Hit*

* This series of read/write activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user atomic read/write (e.g., LDSTUB) that results in an external cache hit followed by
a user instruction read (Fetch) that results in an external cache hit followed by
a user word read (e.g., LDD or LDDF).

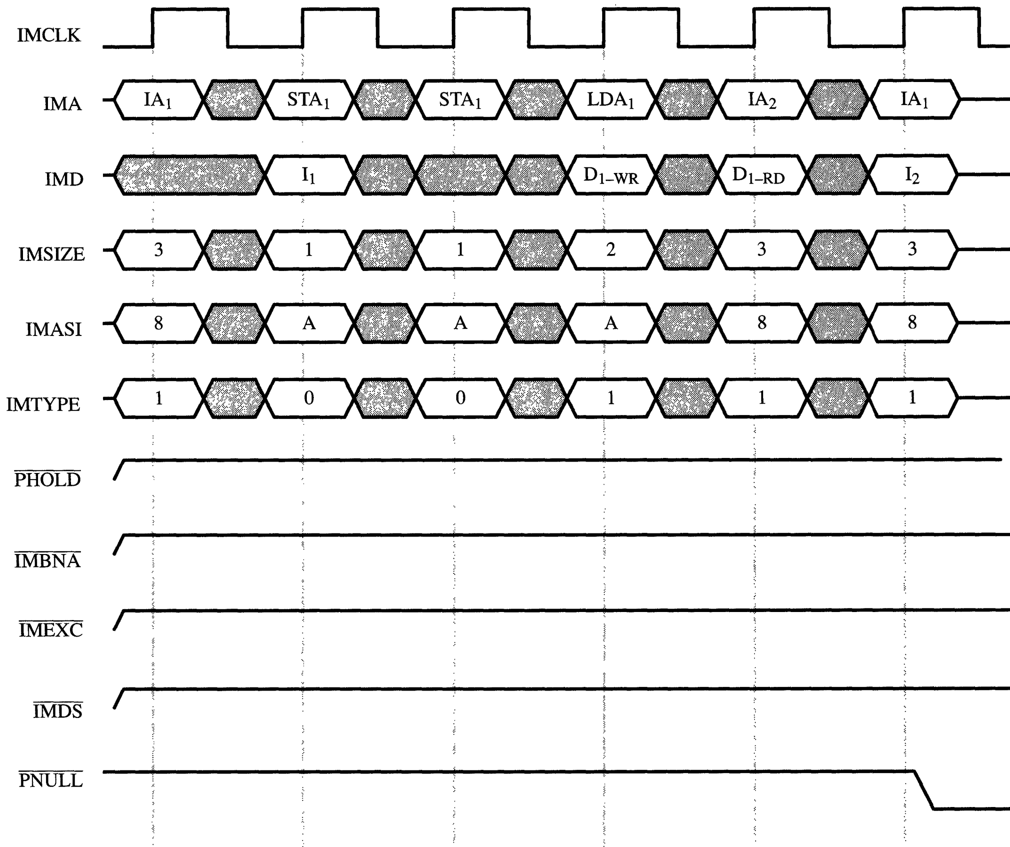


Figure 3–21. Write Access Followed by Read Access with External Cache Hit*

* This series of read activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user halfword write (e.g., STH) that results in an external cache hit followed by
a user word read (e.g., LD or LDF) that results in an external cache hit followed by
two user instruction reads (Fetches).

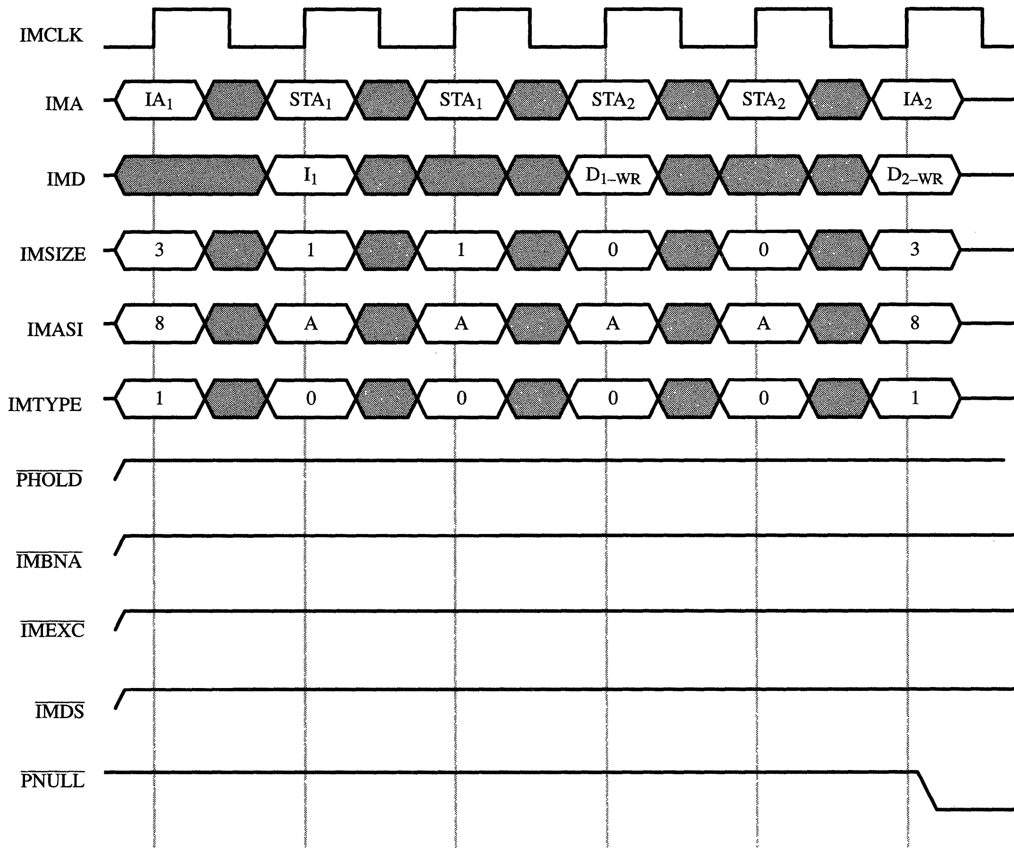


Figure 3–22. Write Access Followed by Write Access with External Cache Hit*

* This series of read activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user halfword write (e.g., STH) that results in an external cache hit followed by
a user byte write (e.g., STB) that results in an external cache hit followed by
a user instruction read.

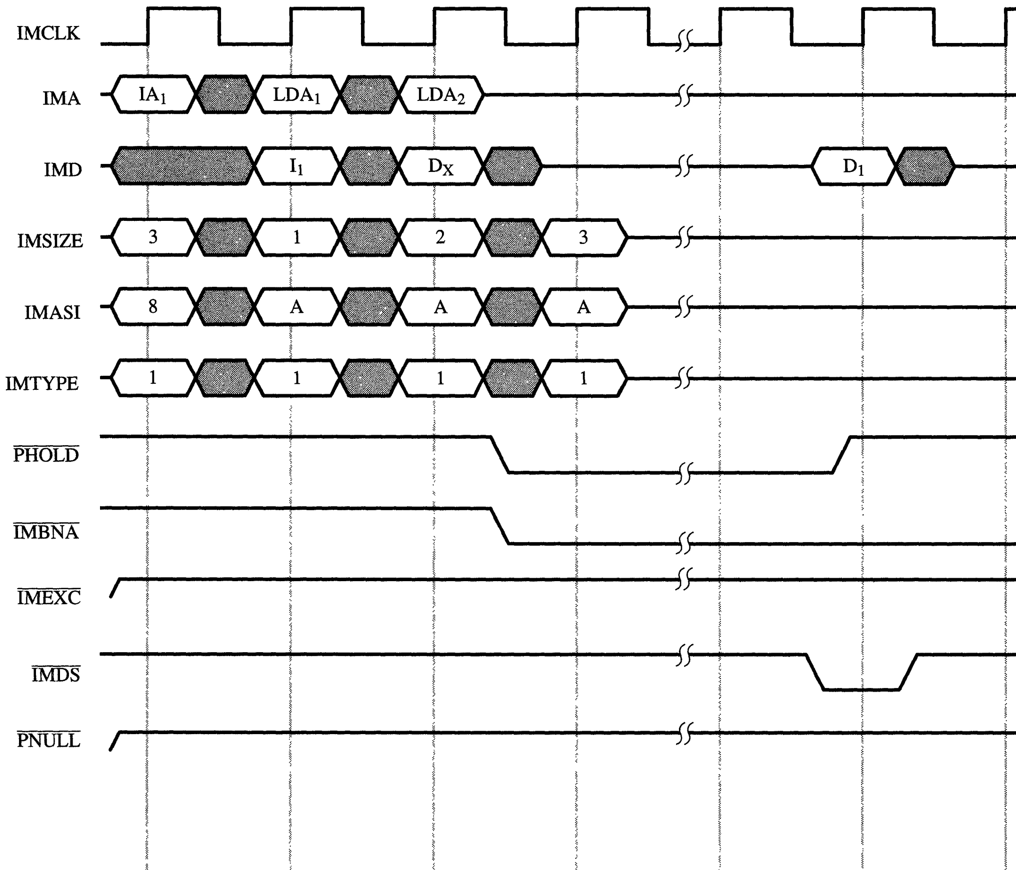


Figure 3–23. Read Access with External Cache Miss* **(page 1 of 2)**

* This series of read activities corresponds to . . .
  a user instruction read (Fetch) that results in an external cache hit followed by
  a user halfword read (e.g., LDSH) that results in an external cache miss and a cache line fill
  (where the halfword is latched by the processor as soon as it is available) followed by
  a user word read (e.g., LD or LDF) that results in an external cache hit followed by
  a user doubleword read (e.g., LDD or LDDF) that results in an external cache hit followed by
  a user byte read (e.g., LDSB or LDUB) that results in an external cache hit followed by
  a user instruction read (Fetch)

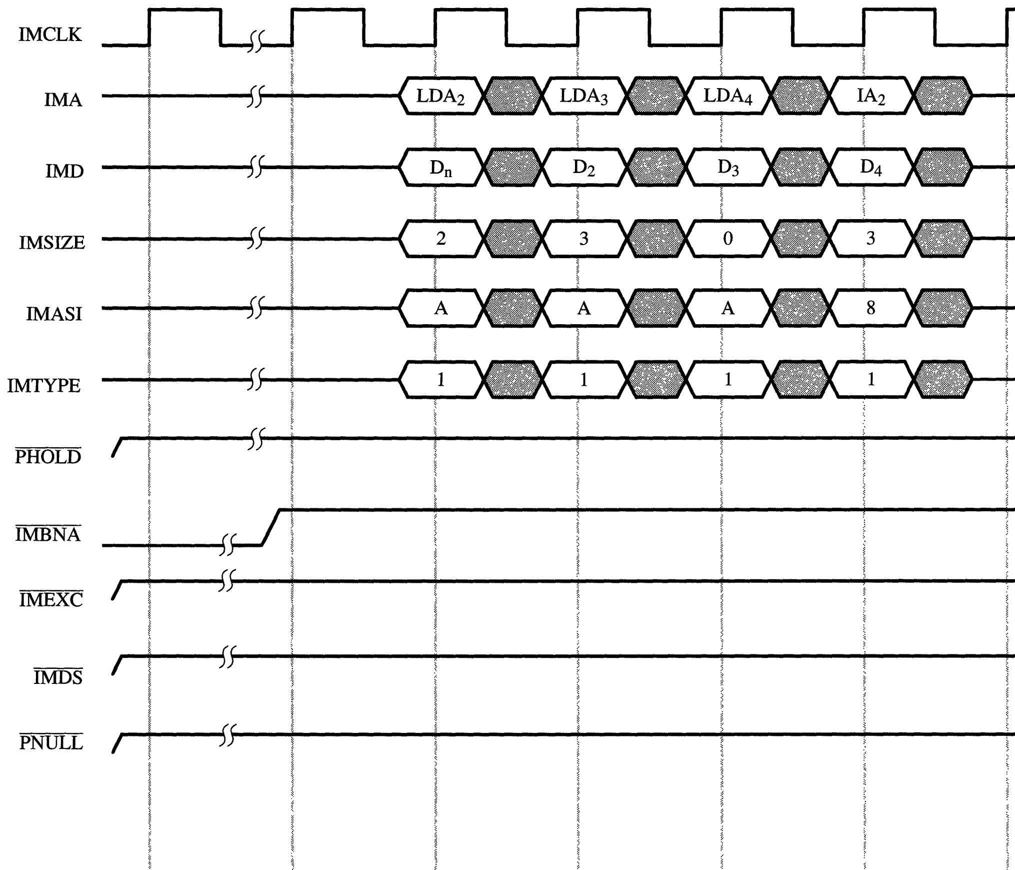


Figure 3–23. Read Access with External Cache Miss (page 2 of 2)

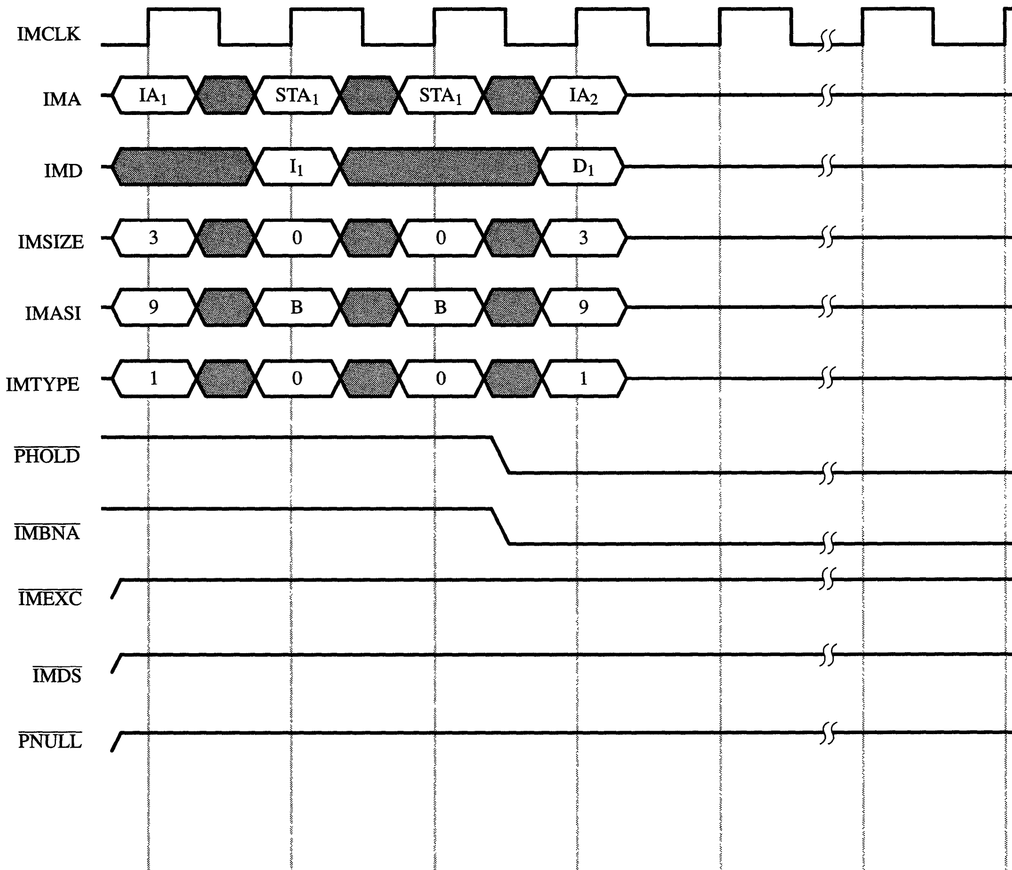


Figure 3–24. Write Access with External Cache Miss* (page 1 of 2)

* This series of read/write activities corresponds to . . .
a supervisor instruction read (Fetch) that results in an external cache hit followed by
a supervisor byte write (e.g., STB) that results in an external cache miss and a cache line fill followed by
a supervisor instruction read (Fetch) that results in an external cache hit followed by
a supervisor word write (e.g., ST or STF) that results in an external cache hit.

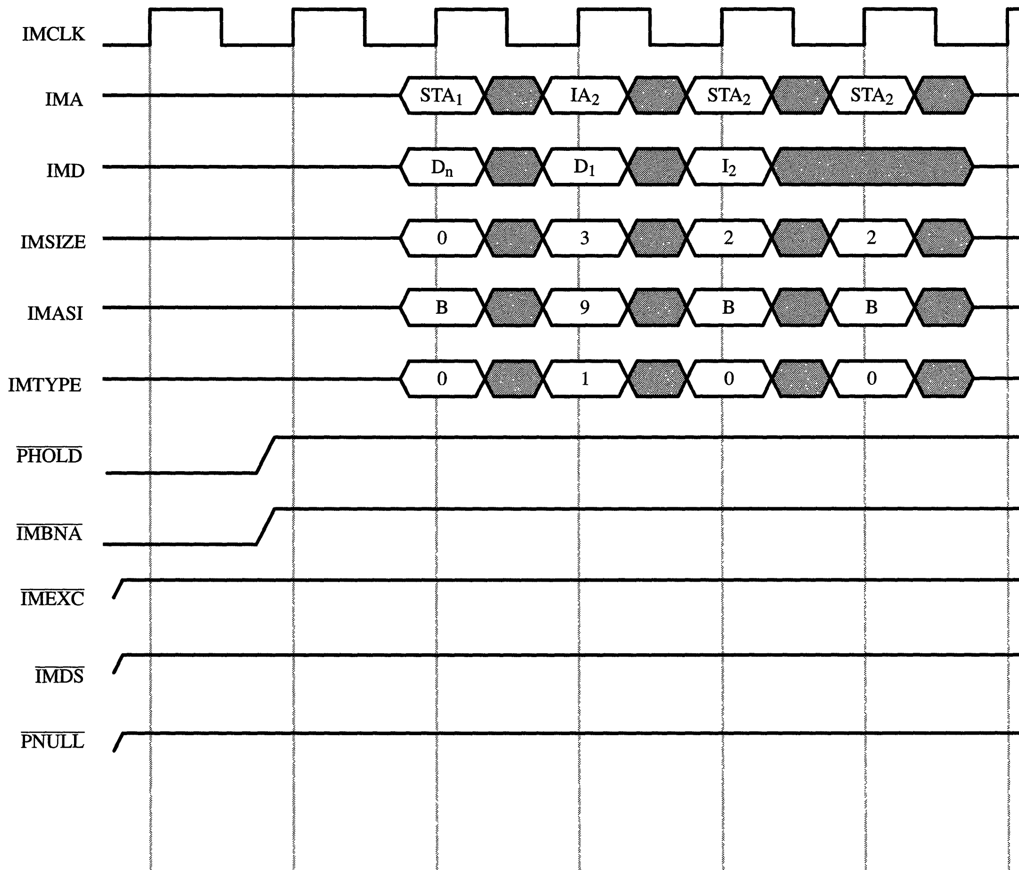


Figure 3–24. Write Access with External Cache Miss (page 2 of 2)

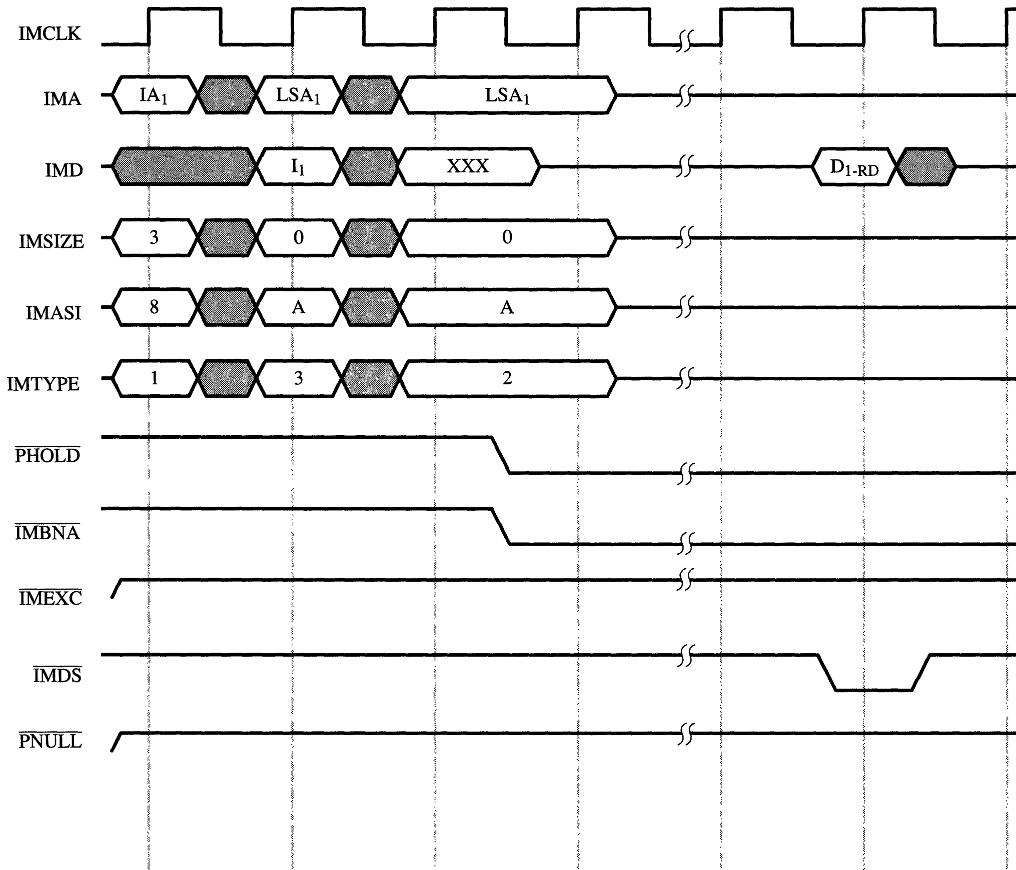


Figure 3–25. Atomic Read-Write Access with External Cache Miss* (page 1 of 2)

* This series of read/write activities corresponds to . . .
  a user instruction read (Fetch) that results in an external cache hit followed by
  a user atomic read/write (e.g., LDSTUB) that results in an external cache miss and a cache line fill
  (where the byte is latched by the processor as soon as it is available) followed by
  two user instruction reads (Fetches) that result in external cache hits.

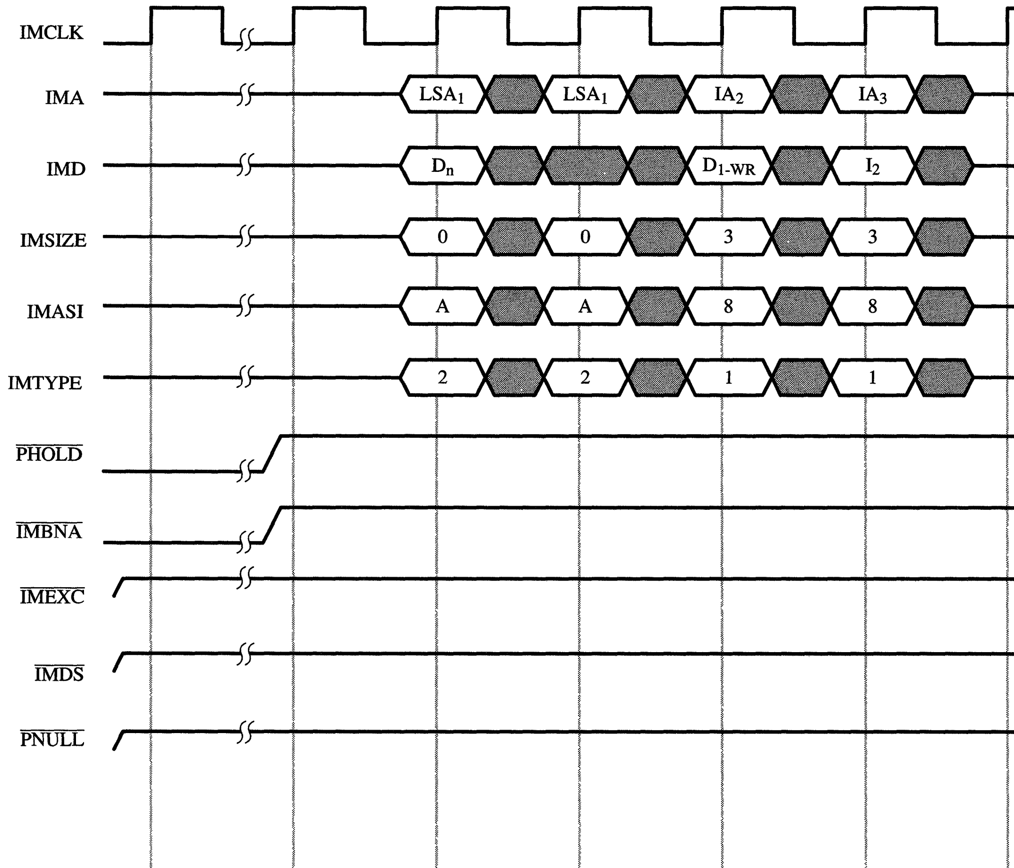


Figure 3–25. Atomic Read-Write Access with External Cache Miss (page 2 of 2)

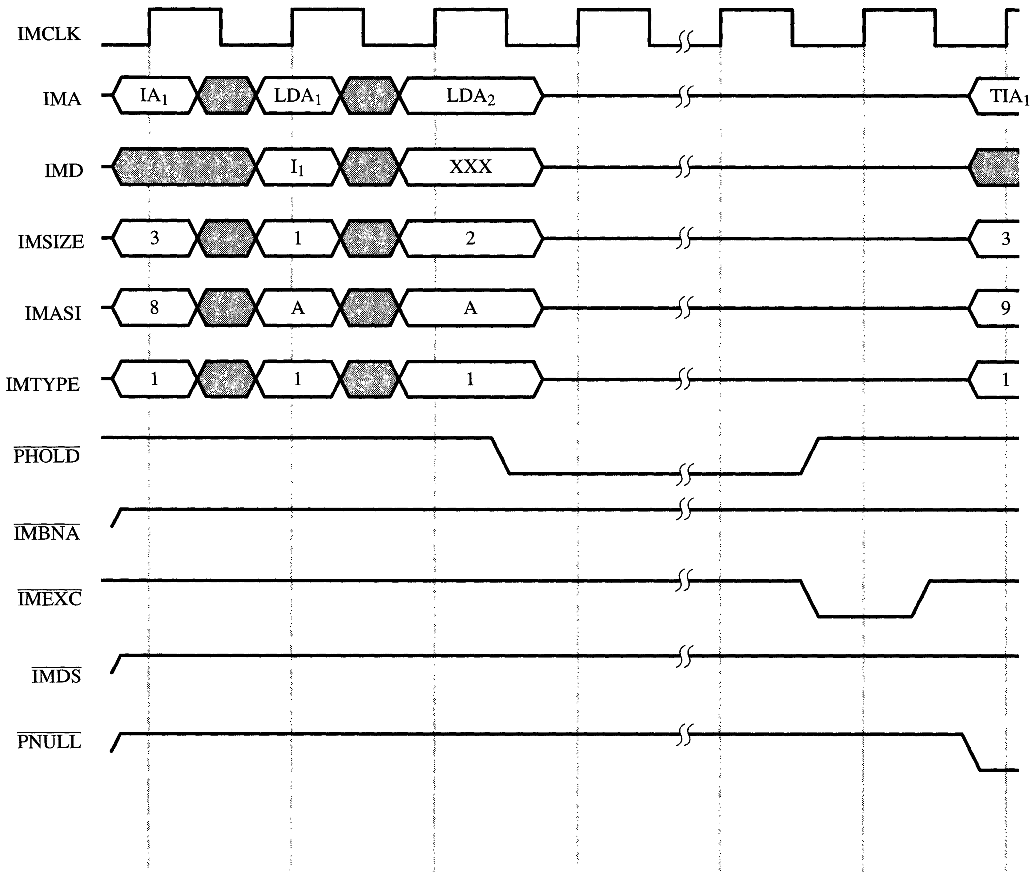


Figure 3–26. Data Read Access with Memory Exception* **(page 1 of 2)**

* This series of read activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user halfword read (e.g., LDSH) that results in an external cache miss and
a memory exception followed by
a sequence of supervisor instruction reads (Fetches) corresponding to the exception
trap handler that are fetched from main memory.

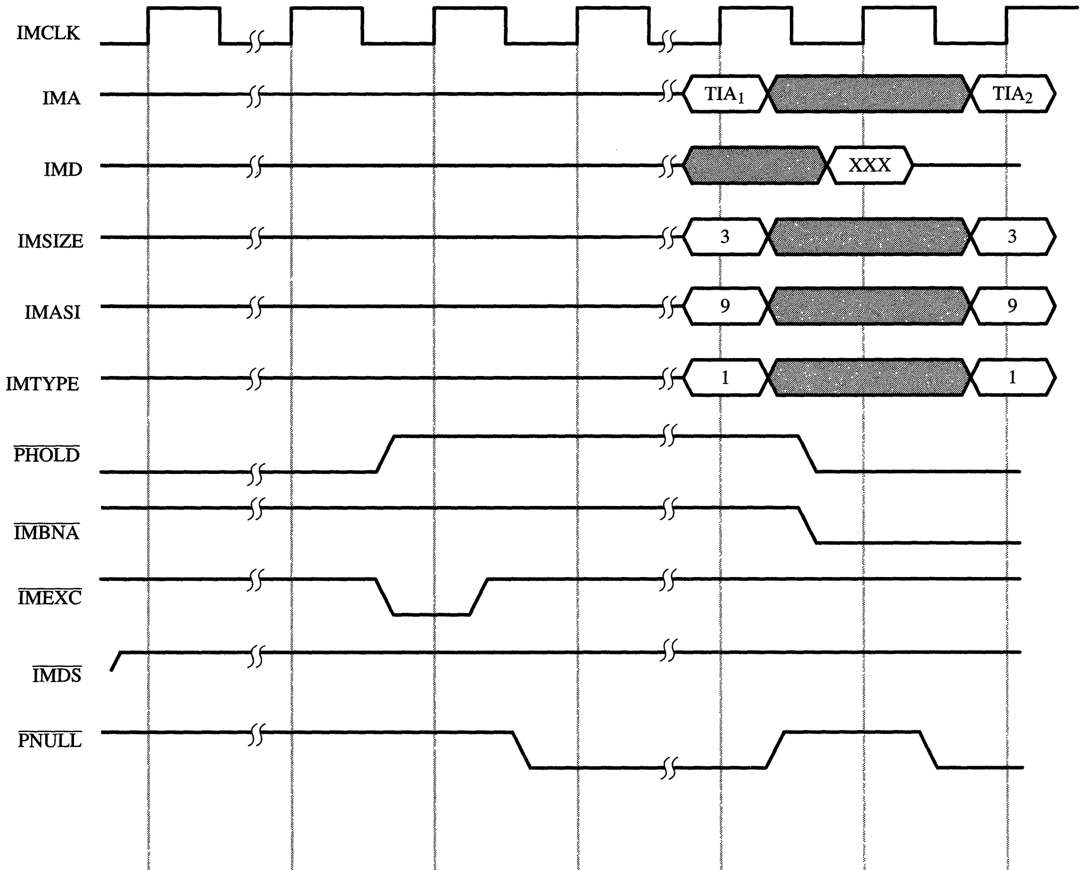


Figure 3–26. Data Read Access with Memory Exception (page 2 of 2)

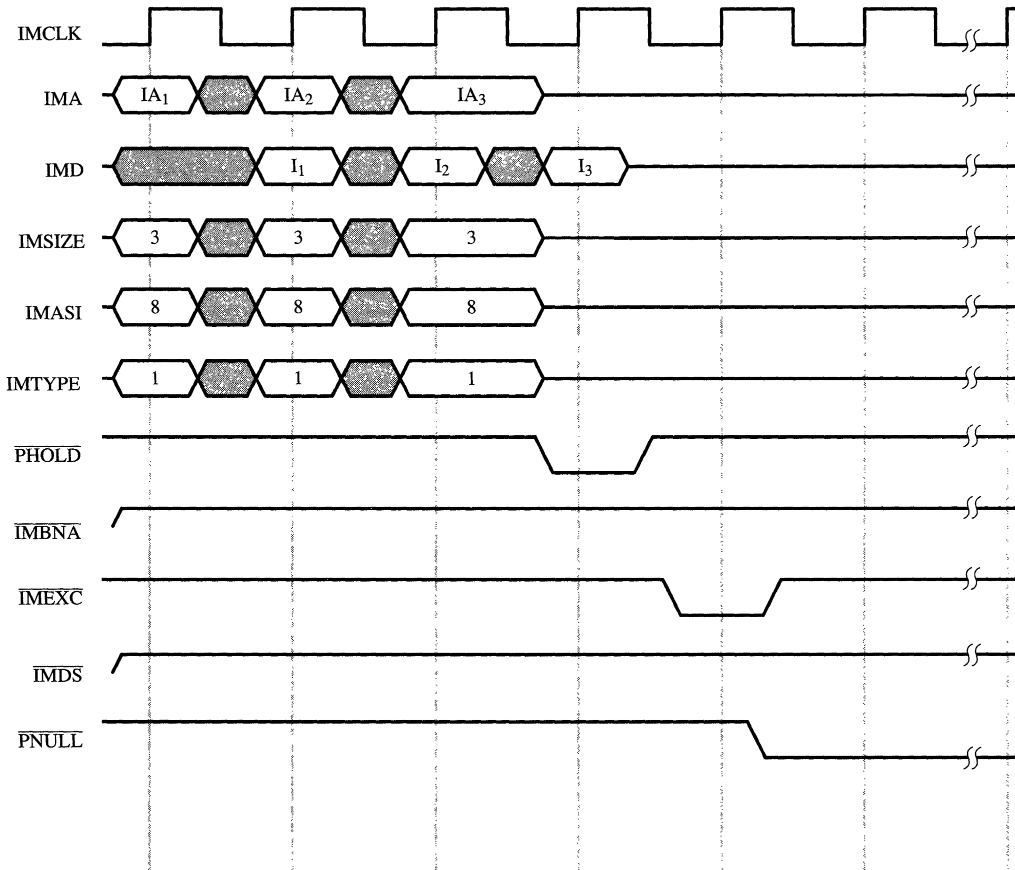


Figure 3–27. Instruction Read Access with Memory Exception* (page 1 of 2)

* This series of read activities corresponds to . . .
a user instruction read (Fetch) that results in an external cache hit followed by
a user instruction read (Fetch) that results in an external cache miss and a memory exception followed by
a user instruction "pre-Fetch" followed by
a sequence of supervisor instruction reads (Fetches) corresponding to the exception trap handler that are fetched from main memory.

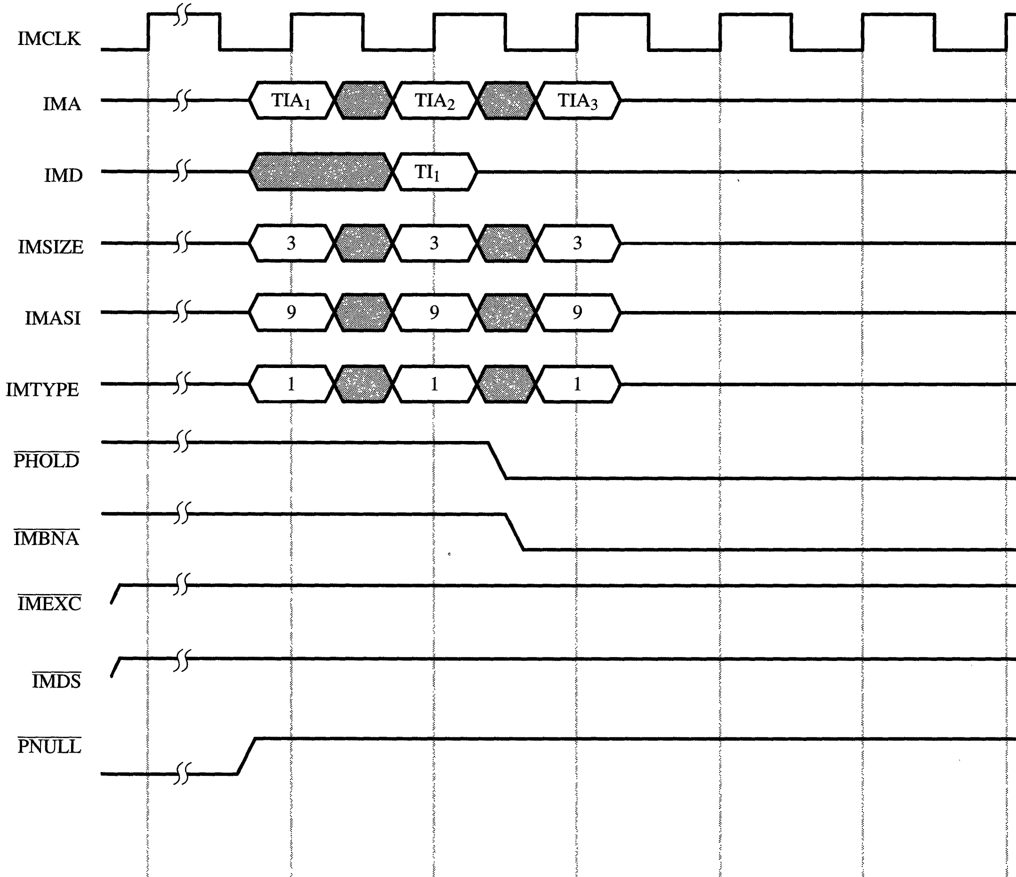


Figure 3–27. Instruction Read Access with Memory Exception (page 2 of 2)

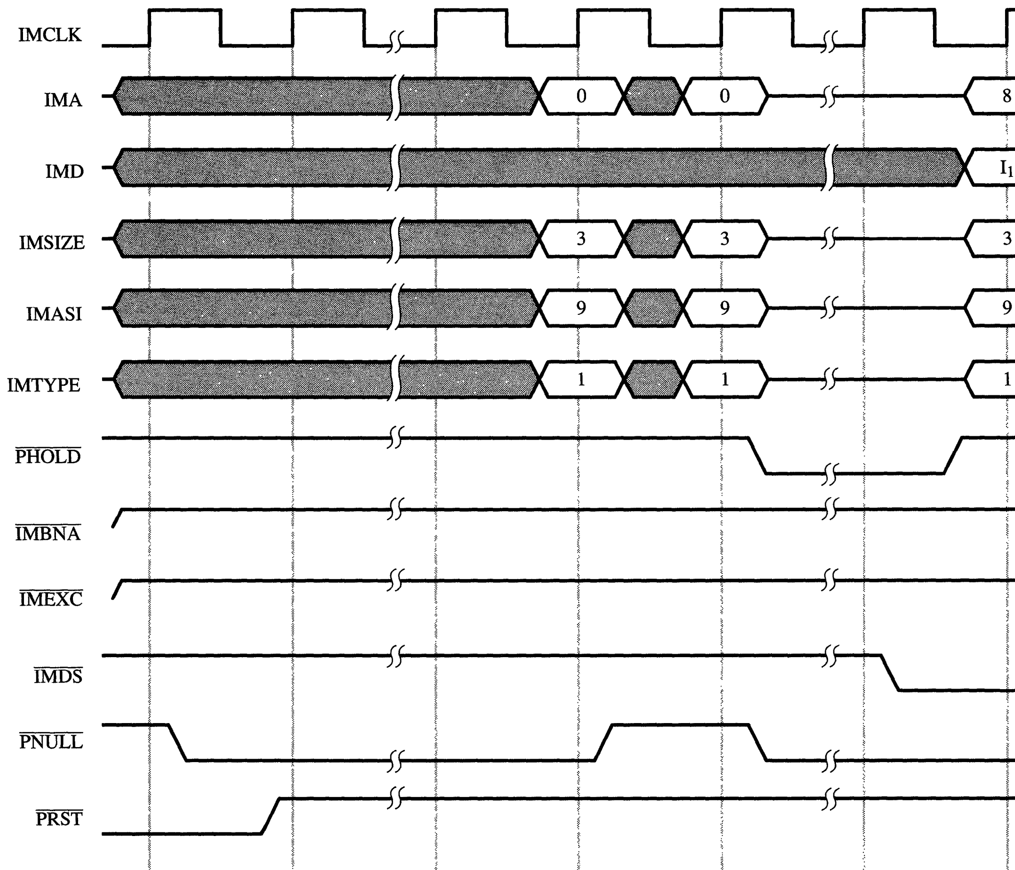


Figure 3–28. Reset Timing* **(page 1 of 2)**

* These bus activities correspond to . . .
a reset that is held for at least eight clocks followed by
a series of supervisor instruction reads (Fetches) that corresponds to
the reset boot program beginning at address 0.

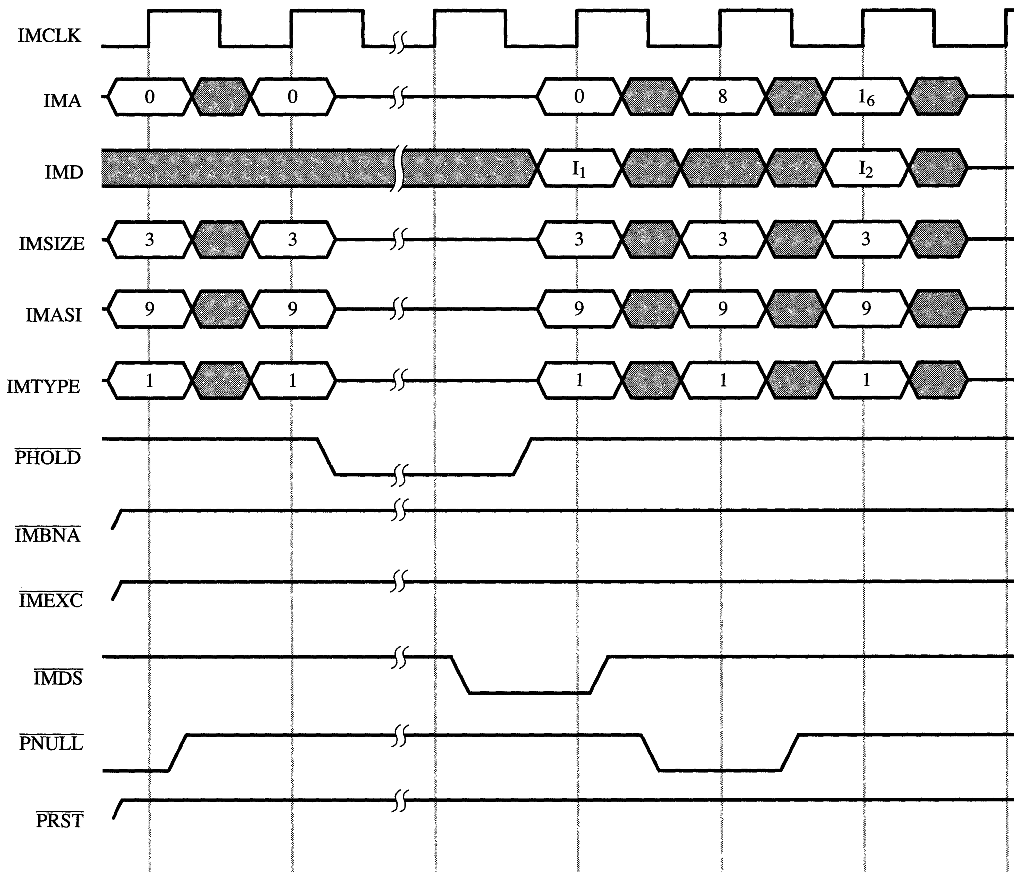


Figure 3–28. Reset Timing (page 2 of 2)

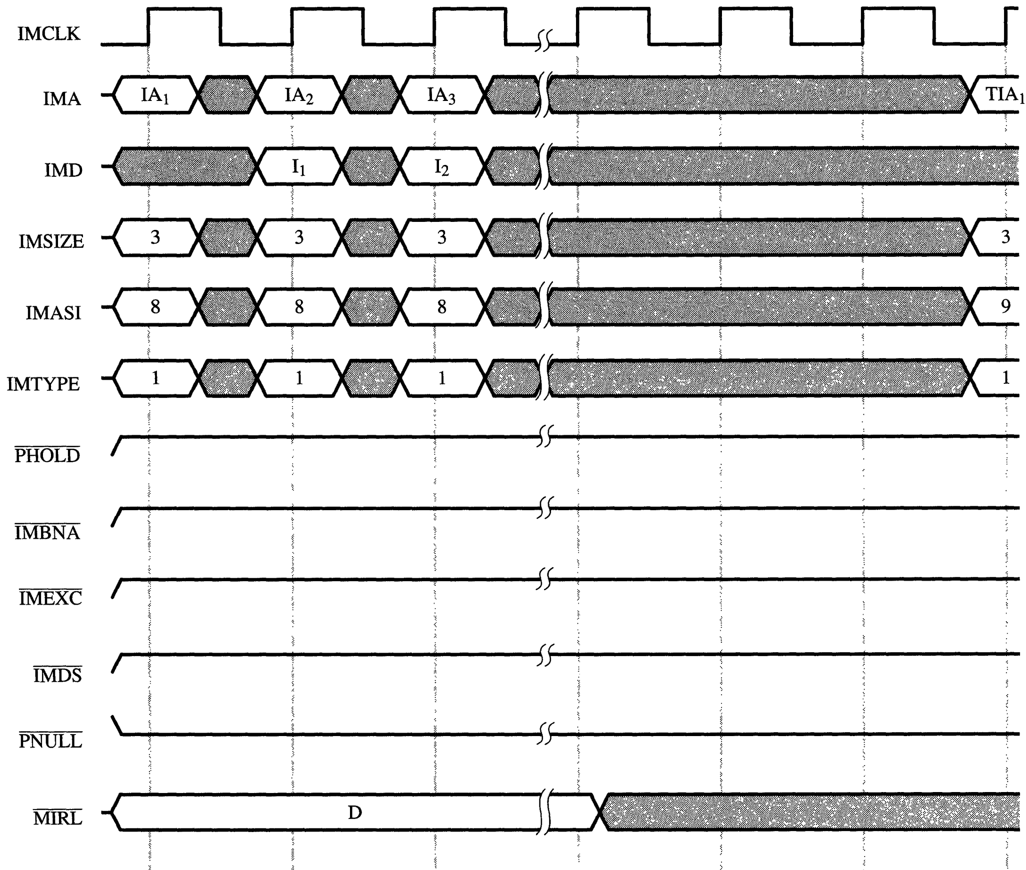


Figure 3–29. Interrupt Timing* **(part 1 of 2)**

* This series of read activities corresponds to . . .
an external interrupt coinciding with
a series of user instructions reads (Fetches) with ICACHE hits followed by
a series of supervisor instruction reads (Fetches) of the interrupt handler.

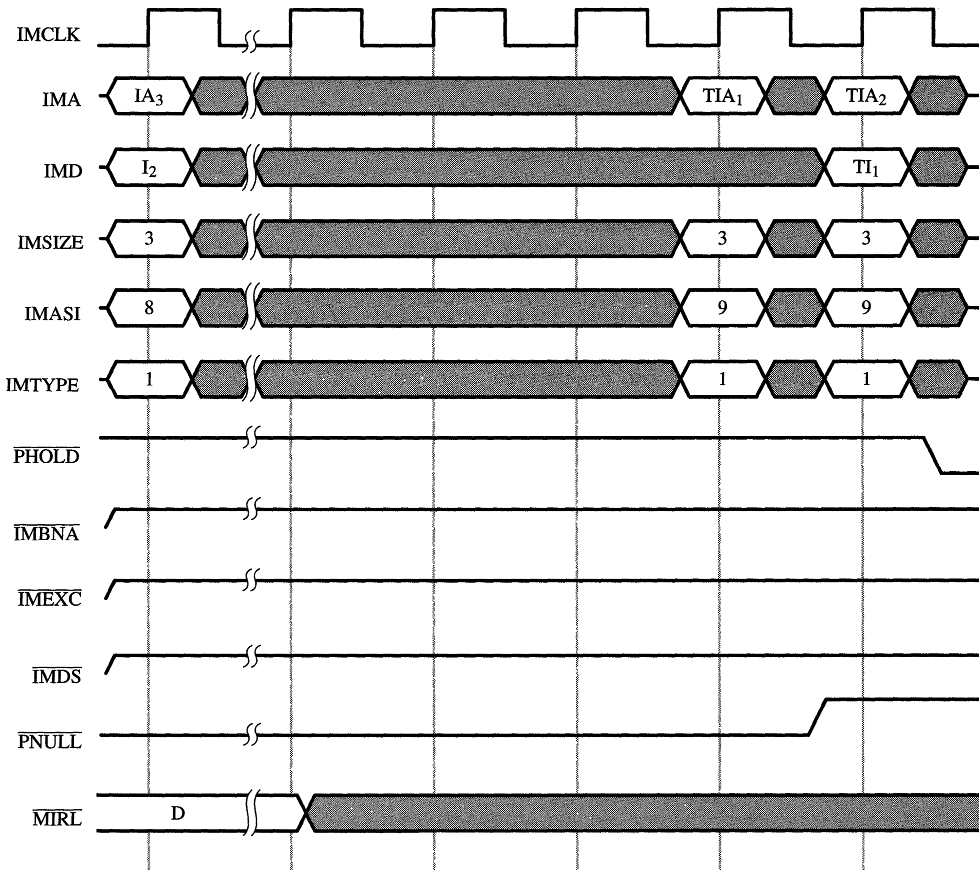


Figure 3–29. Interrupt Timing (part 2 of 2)

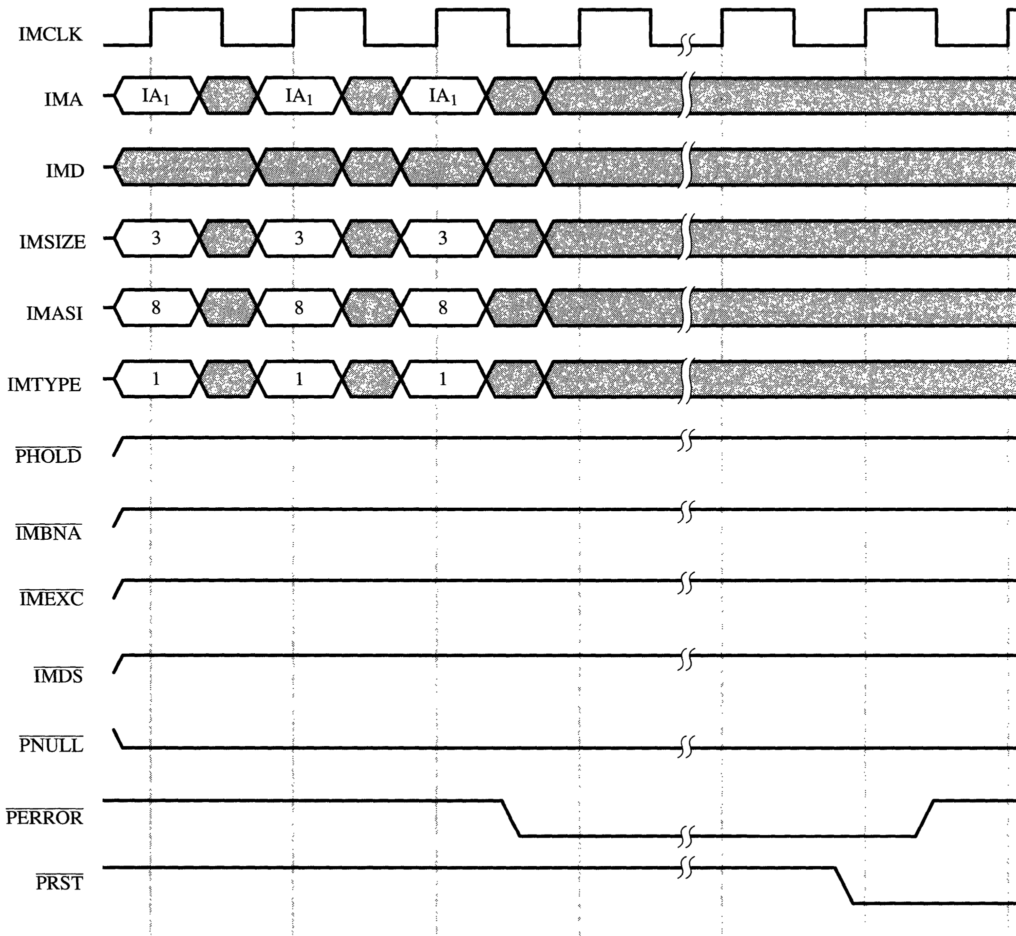


Figure 3–30. Error Timing* (page 1 of 2)

* This series of bus activities corresponds to . . .
  an instruction read (Fetch) that is cancelled due to an exception advancing through the pipeline while traps are disabled, which results in the generation of the processor error signal.
  the processor is reset externally
  the boot program is fetched beginning at address 0.

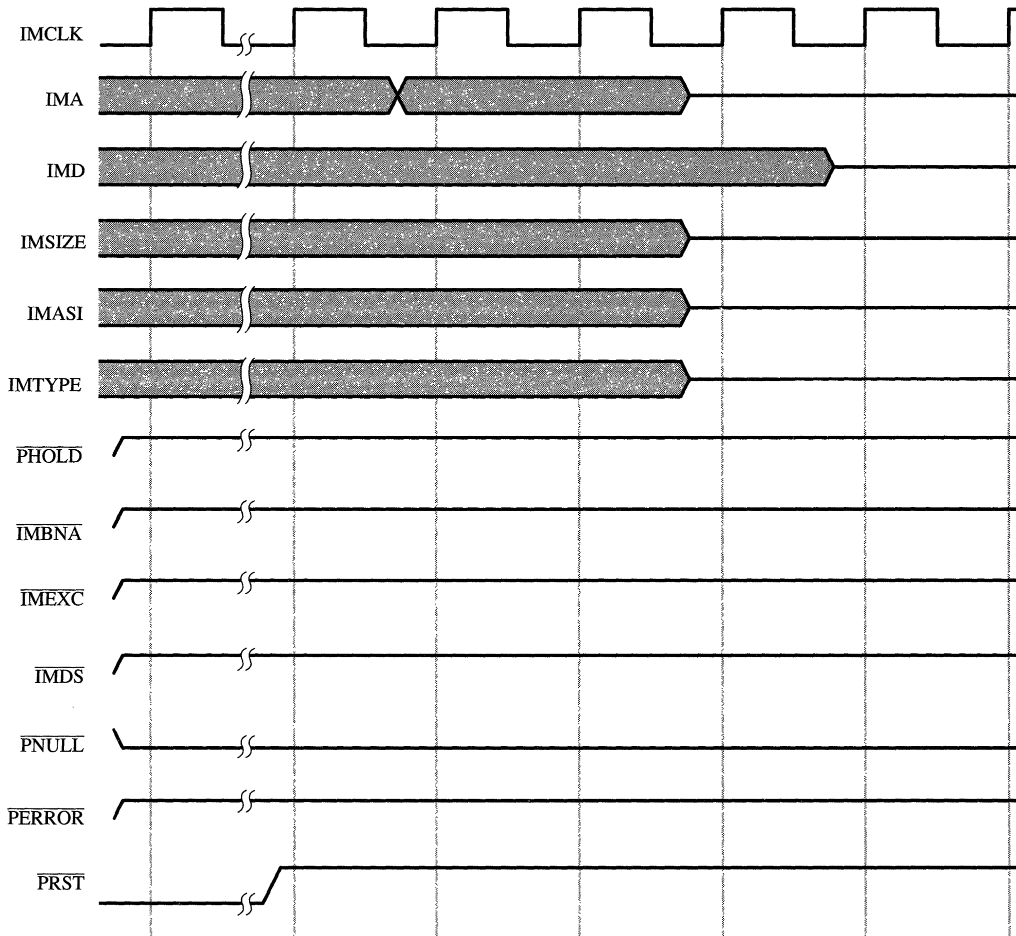


Figure 3–30. Error Timing (page 2 of 2)

3.8 Instruction Pipelines

The hyperSPARC CPU has a highly pipelined microarchitecture. Except for a few cases, the pipelines for the integer unit and floating-point unit are uniform. This facilitates multiple instruction launch and simplifies recovery from instruction execution exceptions. The following sections describe the RT620 instruction pipelines under various conditions.

3.8.1 Instruction Fetch Timing

Every instruction execution begins with an instruction fetch. The instruction can be fetched from one of three locations: the on-chip instruction cache, external cache (e-Cache), or main memory. *Figure 3–31, Figure 3–32* and *Figure 3–33* describe each of these cases when the ICACHE is enabled. *Figure 3–34* and *Figure 3–35* show timing of instruction fetch with e-Cache hit and e-Cache miss with the ICACHE disabled.

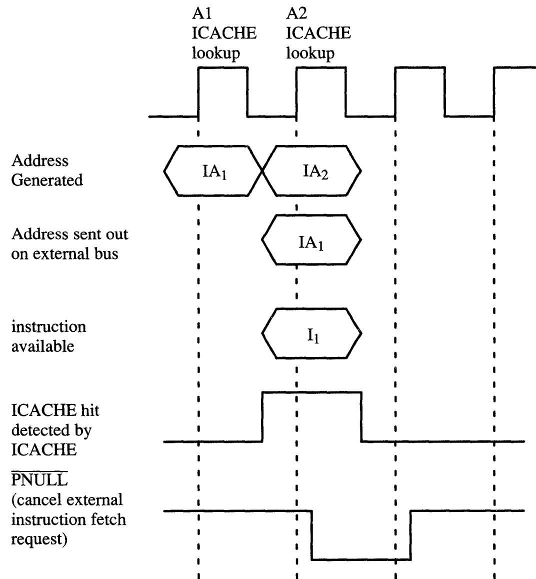


Figure 3–31. Timing for Instruction Fetch with ICACHE Hit

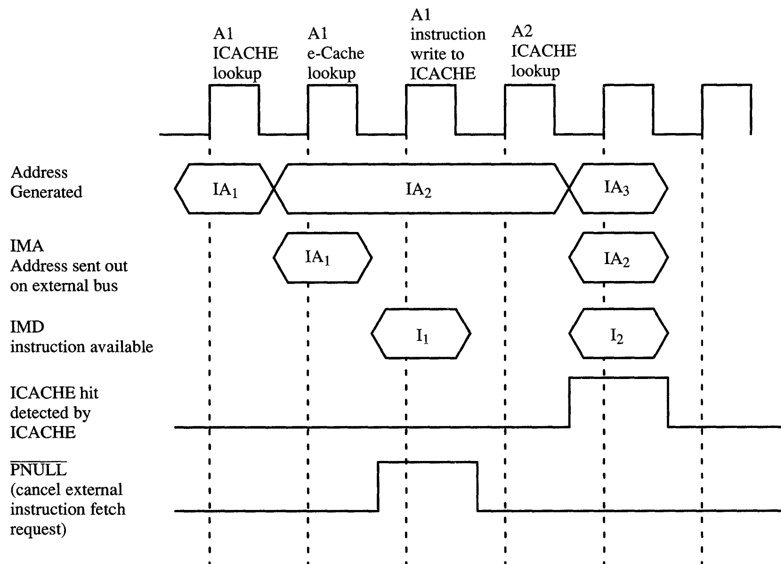


Figure 3–32. Timing for Instruction Fetch with ICACHE Miss and e-Cache Hit

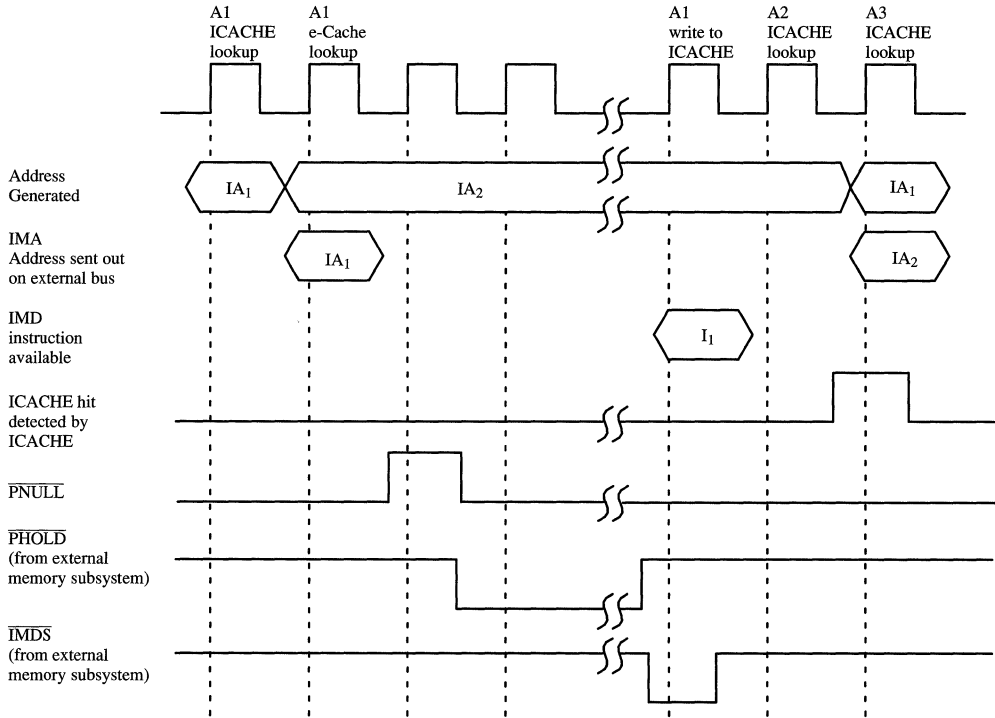


Figure 3–33. Timing for Instruction Fetch with ICACHE Miss and e-Cache Miss

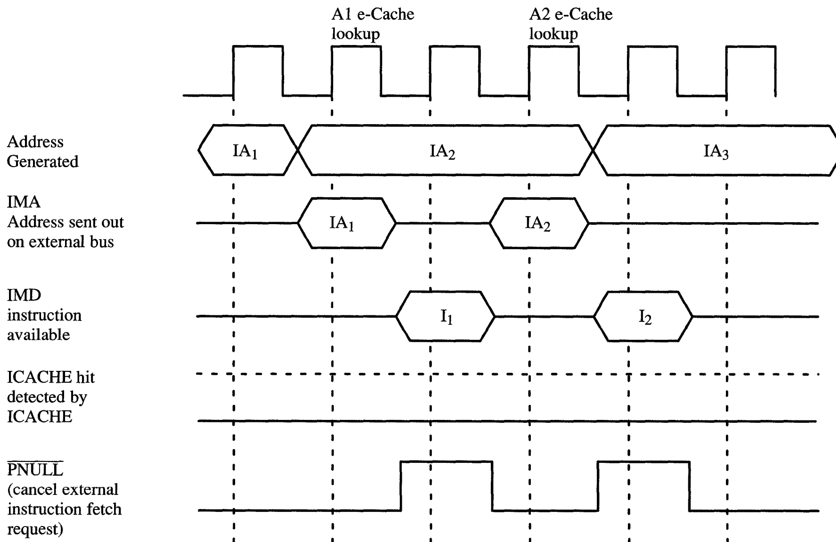


Figure 3–34. Timing for Instruction Fetch with ICACHE Disabled and e-Cache Hit

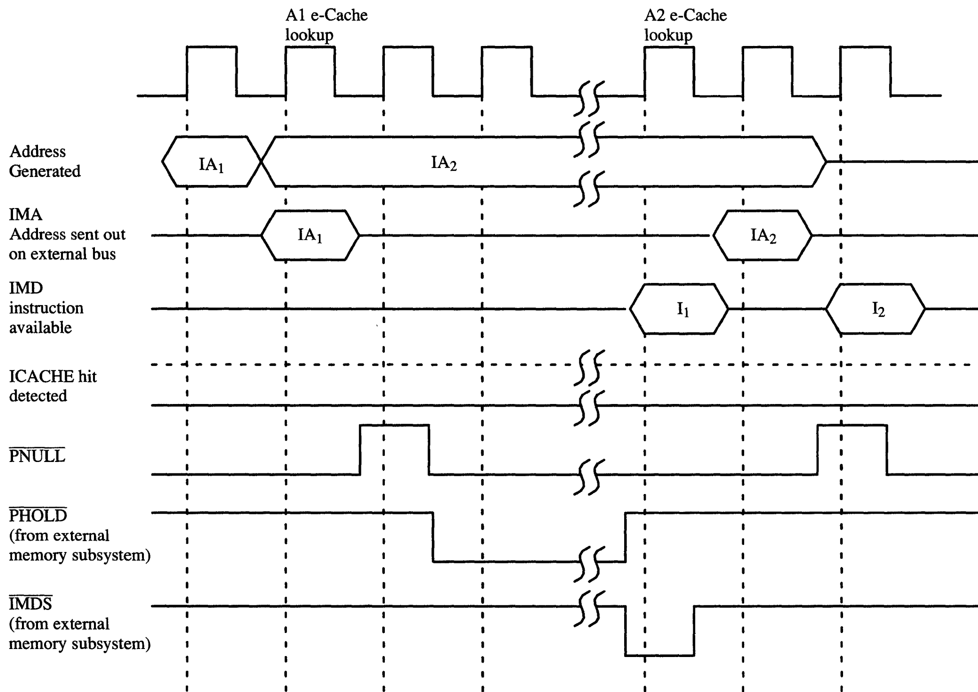


Figure 3–35. Timing for Instruction Fetch with ICACHE Disabled and e-Cache Miss

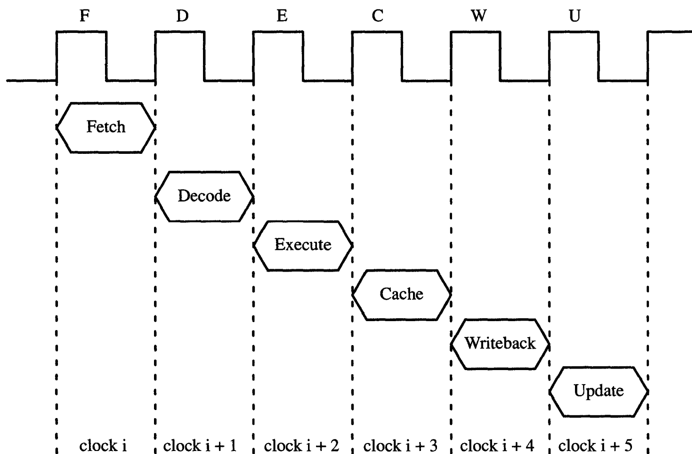


Figure 3–36. Typical Integer Instruction Pipeline

3.8.2 Integer Instruction Pipeline

The instructions in the Arithmetic Logic Unit (ALU), Load/Store Unit (LSU), and Program Counter Unit (PCU) groups follow a 6-stage pipeline denoted as: Fetch (F), Decode (D), Execute (E), Cache (C), Write-

back (W) and Update (U). All instructions share common instruction fetch activities and common global Decode activities. The local Decode and other Execute, Cache, and Writeback stage activities vary from instruction to instruction. *Figure 3–36* describes the typical integer instruction pipeline stages.

Note that there are Execute, Cache, and Writeback stage forwarding mechanisms for the ALU and there are Cache and Writeback stage forwarding mechanisms for the LSU. The Update stage is when the register file is actually written.

The following text and illustrations are intended to convey basic timing information associated with the pipeline stage activities for each of the major integer instructions. The functional activities are slightly different for load, store, and Atomic Load-Store instructions, so each is treated as a separate case.

Table 3–5. Integer Unit Cycle Per Instruction (CPI)*

Instruction	CPI
Single-cycle ALU instructions: ADD, AND, OR, XOR, SUB, ANDN, ORN, XNOR, ADDX SUBX, ADDcc, ANDcc, ORcc, XORcc, SUBcc, ANDNcc, ORNcc, XORNcc, ADDXcc, SUBXcc, TADDcc, TSUBcc, MULScc, SLL, SRL, SRA, SETHI	1
Integer multiply instructions: UMUL, UMULcc, SMUL, SMULcc	17
Integer divide instructions: UDIV, UDIVcc, SDIV, SDIVcc	37
Branch instructions: Bicc, FBfcc	1**
Control transfer instructions: CALL, JMPL, RETT, SAVE, RESTORE, Ticc	1
Control register access instructions: RDY, RDPSR, RDWIM, RDTBR, WRY, WRPSR, WRWIM, WRTBR	1
Load instructions: LDSB, LDSH, LD, LDUB, LDUH, LDD, LDF, LDDF ***	1
Store instructions: ST, STB, STH, STD, STF, STDF ***	2
Atomic Load-Store instructions: SWAP, LDSTUB	3
Miscellaneous: UNIMP, FLUSH	1

* These CPI values assume the worst case situation of single instruction launch. The CPI decreases if dual instruction launch occurs. Also, pipeline enhancements such as Fast Branch or Fast Constant affect CPI.

** Assuming the delay slot is filled with a useful instruction and single instruction launch.

*** These instruction groups also include the alternate space version of the instruction (for example: LDA, STA).

3.8.3 ALU Instructions

Each ALU instruction proceeds through a series of processing activities. *Figure 3–37* shows processing activities for typical ALU instructions. Note that integer multiply and divide instructions follow a slightly different pipeline. The first stage activity is the instruction fetch which involves looking up the address of the next ALU instruction in the ICACHE and making it available for Decode. During the second stage (Decode), the ALU instruction is globally decoded during the first half of the clock. By the end of the second half of the clock, local Decode is performed. During local Decode, appropriate control signals for the ALU execution unit are set up and operand access is performed. The decision to launch or delay the instruction is also made at this time. Operand access involves retrieving operand data either from the register file, obtaining results currently in the pipeline (through forwarding), or extracting immediate data from the instruction itself.

During the Execute stage, the ALU operation result is generated. If the instruction causes updates to the condition codes, the new condition codes are generated during the Execute stage. There are several exceptions that can occur when certain integer instructions are executed (e.g., TADDccTV). These exceptions are detected during the Decode or Execute stages (tag overflow, or illegal instruction).

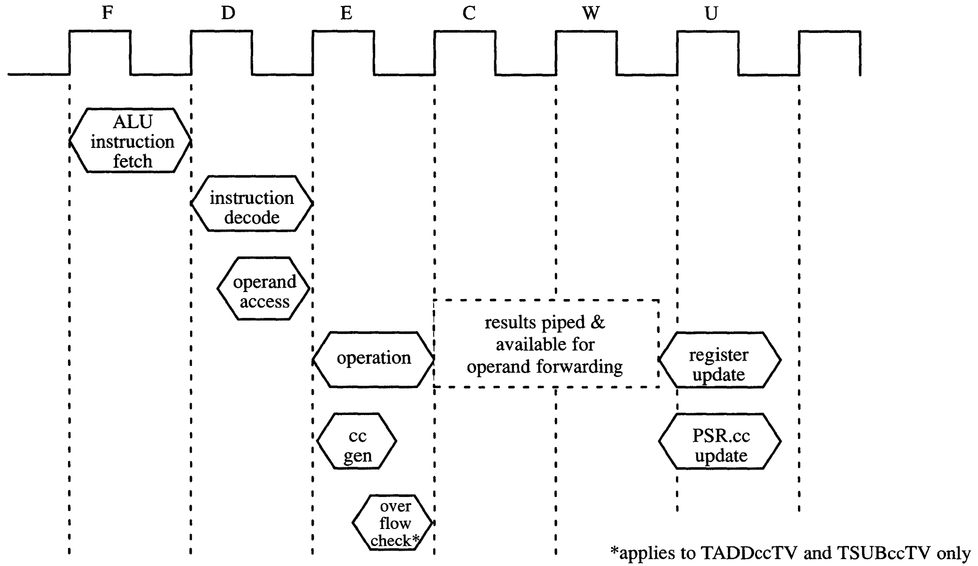


Figure 3–37. Typical ALU Instruction Pipeline

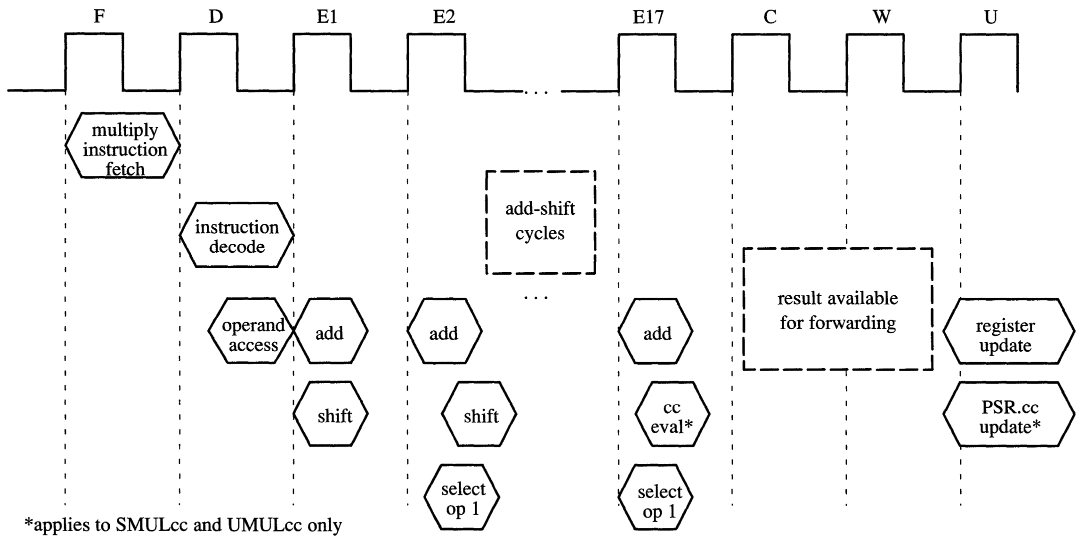


Figure 3–38. Typical Integer Multiply Instruction Pipeline

3.8.3.1 *Integer Multiply Instructions*

The Fetch and Decode stage activities for integer multiply are identical to those for other ALU instructions. Integer multiply instructions require 17 Execute stages to generate results. Each stage of the 17 clock cycles involves addition of inputs 1 and 2, shifting the multiplier, updating the intermediate partial product and selecting the next input 1. If the instruction updates condition codes, the new condition codes are generated once the last addition has been performed. The results and condition codes are forwarded and additional instructions can be launched after Execute stage 17.

3.8.3.2 *Integer Divide Instructions*

The Fetch and Decode stage activities for integer divide are identical to those for other ALU instructions. Integer divide instructions require more than one Execute stage to generate results. For a signed divide, the two's complement of the dividend input is generated during the next two cycles if it is negative. These cycles are idle cycles for other cases. In the next cycle, the dividend and divisor are compared to check for overflow. If there is no overflow, the non-restoring algorithm is executed for 32 cycles. Each iteration consists of a shift followed by an add. Following the last add, for a signed divide, overflow is again checked for during the later half of the cycle and the two's complement of the quotient is generated during the next cycle if necessary. If the instruction updates condition codes, the new condition codes are generated once the last addition has been performed for an unsigned divide or following the two's complement (if necessary) for the signed divide. The results and condition codes are forwarded and additional instructions can be launched after Execute stage 37.

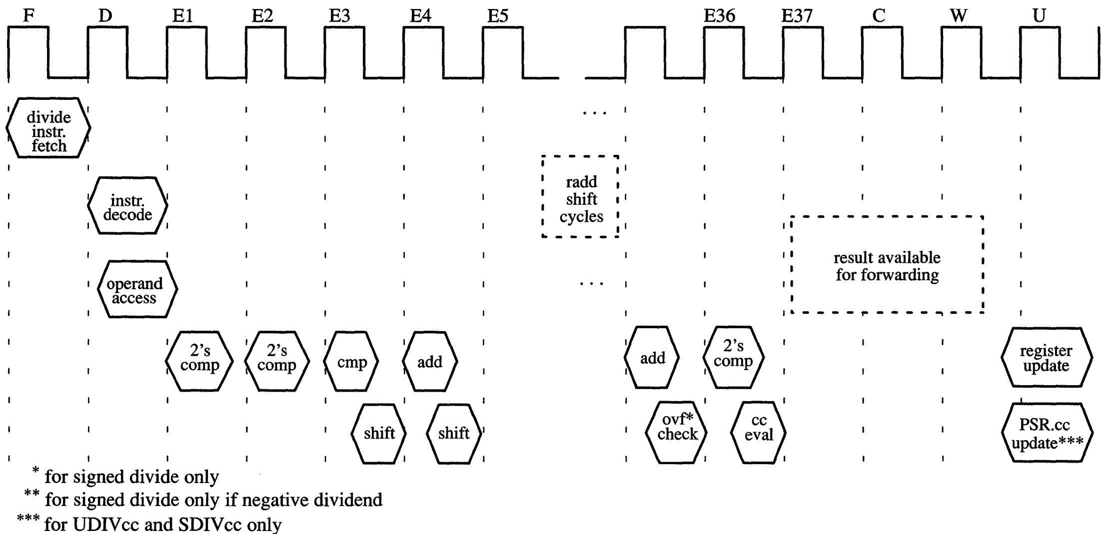
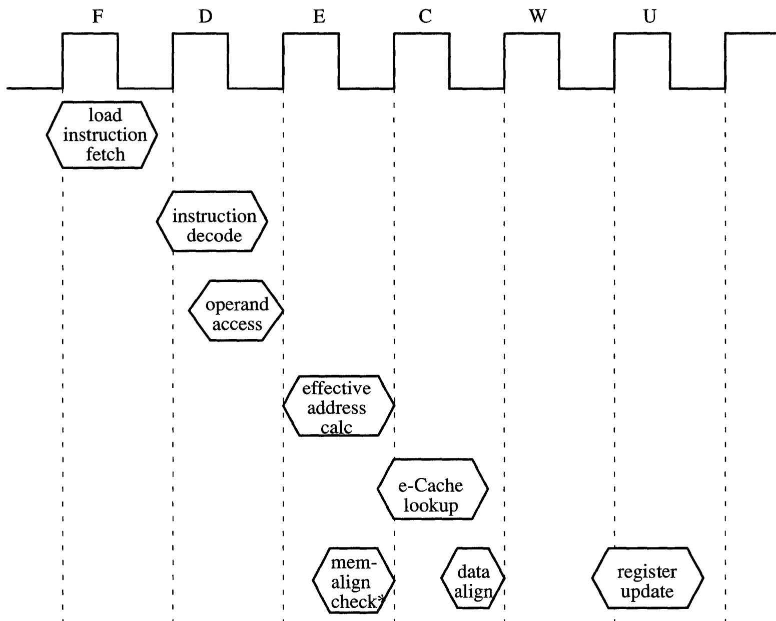


Figure 3–39. Typical Integer Divide Instruction Pipeline



*memory alignment checks are NOT performed for LDSB and LDUB instructions

Figure 3–40. Typical Load Instruction Pipelines

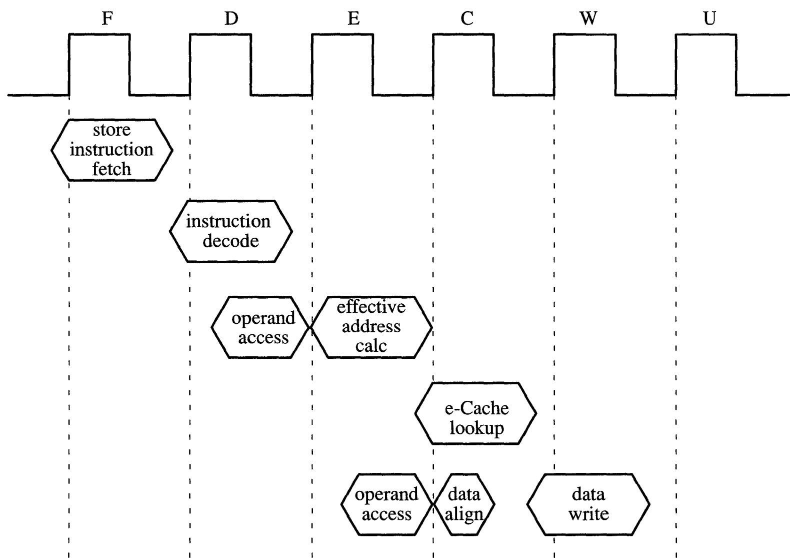


Figure 3–41. Typical Store Instruction Pipeline

3.8.4 Load Instructions

Load instructions come in several varieties, including LD, LDF and LDFSR (word), LDD and LDDF (double), LDUB (unsigned byte), LDSB (signed byte), LDUH (unsigned halfword), and LDSH (signed half). Each integer unit Load also has a corresponding alternate address space form. *Figure 3–40* illustrates the pipeline stages which Load instructions encounter.

Global Decode, local Decode and operand access are performed during the Decode stage. These activities are similar to those employed for ALU instructions. During Decode, it is possible to detect that a privilege violation or illegal instruction has been encountered for the alternate address space form of Load instructions.

During the Execute stage, the effective address is calculated. After this, memory alignment is checked (e.g., if a word data fetch is required but the address calculated does not have "00" in its two least significant bits, a memory alignment exception must be generated). The address is placed on the IMB address lines during the rising edge of the Cache stage (along with the appropriate IMSIZE, IMASI, and IMTYPE 0 signals). If a memory alignment exception has occurred, the memory access is nullified by asserting PNULL on the cycle following the address. This provides the cache management unit with the data address to lookup in the external cache for this data access request.

Assuming there is a hit for the data in the external cache, the data is delivered prior to the Writeback stage. Data alignment is performed during the Cache stage and the data is updated in the register file after the Writeback stage.

3.8.5 Store Instructions

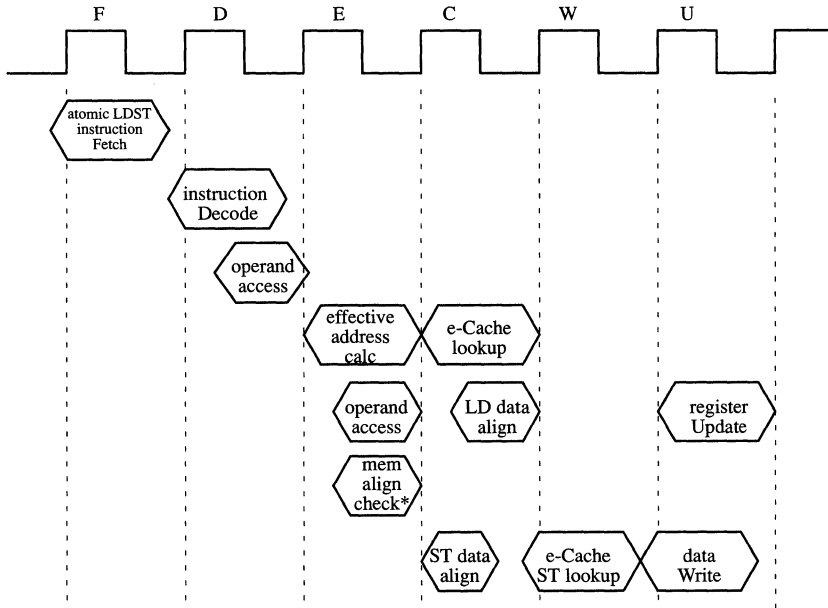
Store instructions include all access size and alternate address space forms available for Load instructions. As with load instructions, alternate address forms of the store instruction must be decoded so as to detect privilege violations and illegal instructions. *Figure 3–41* shows timing for a typical Store instruction.

The effective address is calculated and memory alignment checks are performed during the Execute stage. If a memory alignment exception is detected (i.e., access not word aligned), a memory alignment exception is generated. During the execute stage, the source register is accessed for the purpose of writing its contents to memory. During the Cache stage, the address is placed on the IMB (along with the appropriate IMSIZE, IMASI, and IMTYPE < 0 > signals) and data alignment is performed on the source data. If a memory alignment exception has occured, PNULL is asserted in the cycle following the address cycle in order to nullify the memory access.

In the case of a Load instruction, the cache management unit latches the address from the CPU and lookup does not require more than one clock (assuming an external cache hit occurs). In the case of store instructions, the address must be held on the IMB through the Cache and Writeback stages to guarantee Writeback to the data cache (one cycle is required to determine cache-hit and access and protection checks, and a second is required to perform the Writeback).

3.8.6 Atomic Load-Store Instructions

The Atomic Load-Store instructions are non-interruptible sequences of a load memory access followed by a store data access. Therefore, the Atomic Load-Store instruction timing follows that shown in *Figure 3–42*. One difference (which is not obvious from *Figure 3–42*) between an ordinary Load and Store instruction pair and an Atomic Load-Store instruction is that the IMTYPE < 1 > (LOCK) signal is asserted by the IBIU while an Atomic Load-Store instruction is in progress.



* memory alignment check applies only to SWAP and SWAPA instructions

Figure 3–42. Typical Atomic Load-Store Instruction Pipeline

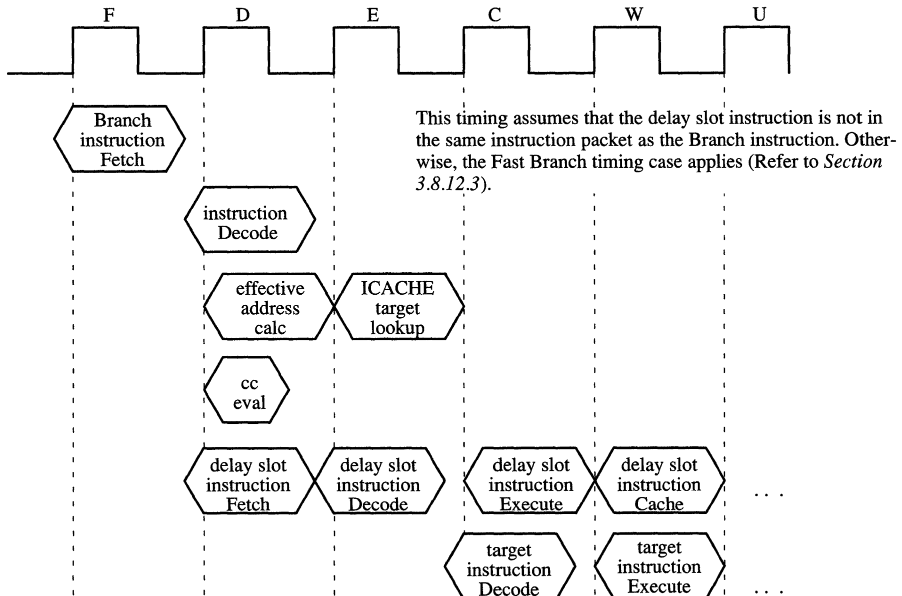


Figure 3–43. Typical Branch Instruction Pipeline

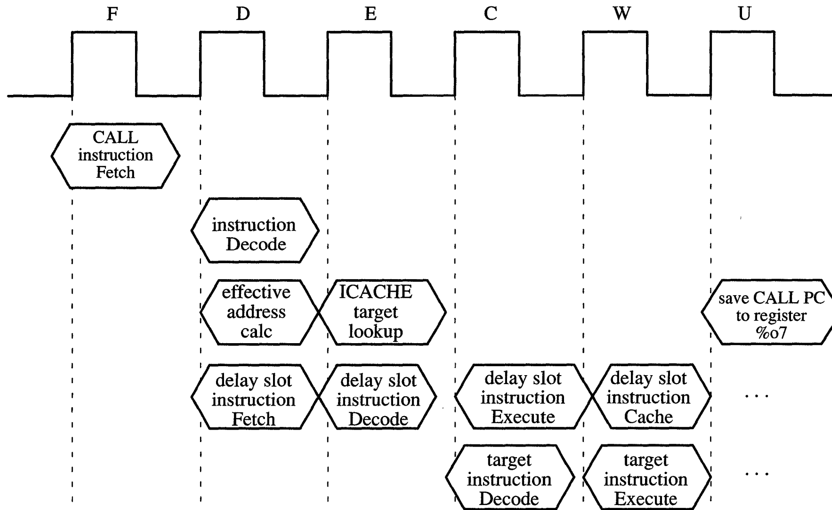


Figure 3–44. Typical CALL Instruction Pipeline

3.8.7 Branch Instructions

Branch instruction timing departs significantly from that applied to ALU, Load, and Store instructions. *Figure 3–43* illustrates Branch instruction timing for integer and fp branches (for the non-Fast Branch case). The Fetch stage is performed in the usual fashion. Three activities occur during Decode. First, the condition codes are evaluated to determine whether the branch will be taken or not. Second the effective address of the target must be calculated using the current PC and the displacement extracted from the instruction by the instruction decoder. Finally, since SPARC permits nullification of delay slot instructions, this must be determined during Decode. The first two activities proceed in parallel. The third activity cannot be determined until the condition codes have been evaluated except in the special case of a *Fast Branch.* Refer to *Section 3.8.12.3* for pipeline descriptions for Fast Branch events.

If the branch is taken, the new target address is placed on the address lines (both for ICACHE and for the IMB along with appropriate IMSIZE, IMASI, and IMTYPE < 0 > signals) and the new instruction stream Fetch begins. If the branch is not taken, instruction fetch continues along the previous instruction stream.

3.8.8 CALL Instructions

The CALL instruction pipeline activities are shown in *Figure 3–44*. The CALL instruction is very similar to branches except in three respects. First, CALL belongs to the single instruction launch group, so a "packet split" always occurs (the concepts of "instruction groups" and "packet splits" are discussed in *Section 3.4*). Second, no condition code evaluation is performed because for CALL instructions, the "branch" is always taken. Finally, the CALL instruction requires that the PC of the CALL instruction be recorded in the register file as described in *Chapter 12, SPARC Instruction Set.*

3.8.9 JMPL/RETT/FLUSH Instructions

Although JMPL, RETT, and FLUSH perform significantly different functions, they are all *f* group instructions and require generation of target addresses. They follow nearly identical timing. *Figure 3–45, Figure 3–46* and *Figure 3–47* illustrate the JMPL/RETT/FLUSH instruction timings.

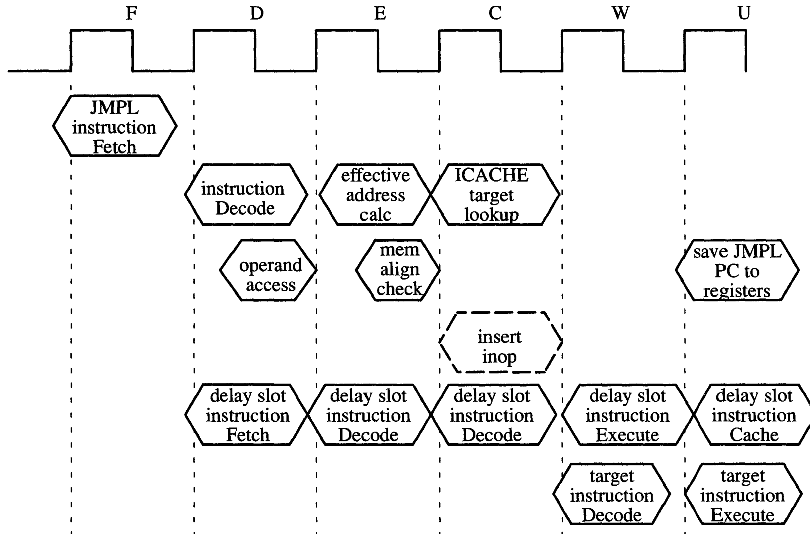


Figure 3–45. Typical JMPL Instruction Pipeline

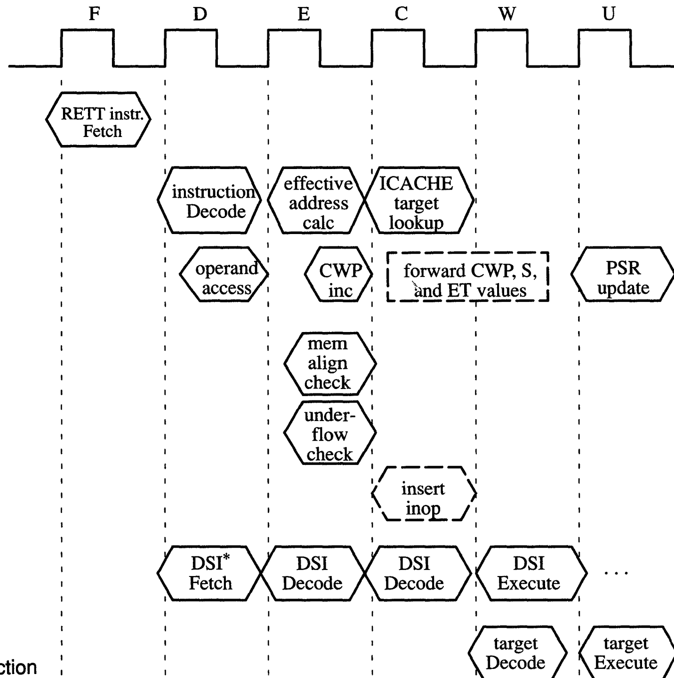


Figure 3–46. Typical RETT Instruction Pipeline

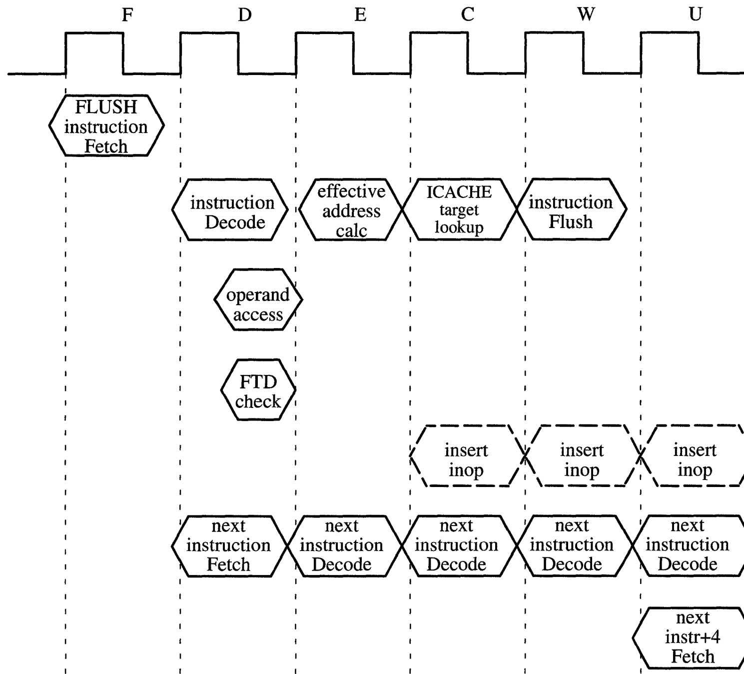


Figure 3–47. Typical FLUSH Instruction Pipeline

The Decode, Execute, and Cache stages of a JMPL/RETT/FLUSH instruction are similar to those of the load instruction. An operand access is performed, an effective address is calculated from these operands, and memory alignment errors are checked. For all these instructions, the address is supplied for lookup in the instruction cache and external caches. The differences between these instructions are shown in the timing diagrams, and include:

- JMPL requires saving the instruction PC back to the register file.
- RETT performs window underflow checks and updates several fields in the PSR.
- FLUSH does not update the program counter with the target address but either takes an exception or flushes the corresponding packet entry in the cache line when an ICACHE hit is detected, depending on the state of the Instruction Cache Control Register (ICCR). When determining which packet of an ICACHE line is to be flushed, the last 3 bits of the address are ignored. Therefore, no misalignment exception can occur for a FLUSH instruction.

3.8.10 SAVE/RESTORE Instructions

The SAVE and RESTORE instructions are very similar to ordinary ALU instructions with two exceptions: (i) they belong to the single instruction launch group and (ii) they modify the Current Window Pointer (CWP). SAVE instructions decrement the CWP and check for window overflows; RESTORE instructions increment the CWP and check for window overflows. The concepts of “instruction groups” are discussed in *Section 3.4. Figure 3–48* illustrates timing for SAVE and RESTORE instructions.

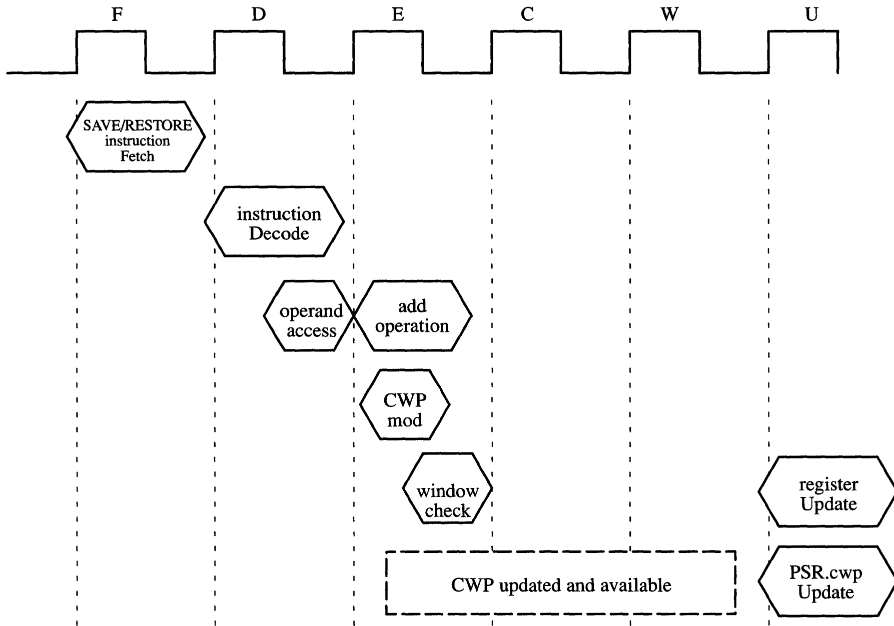
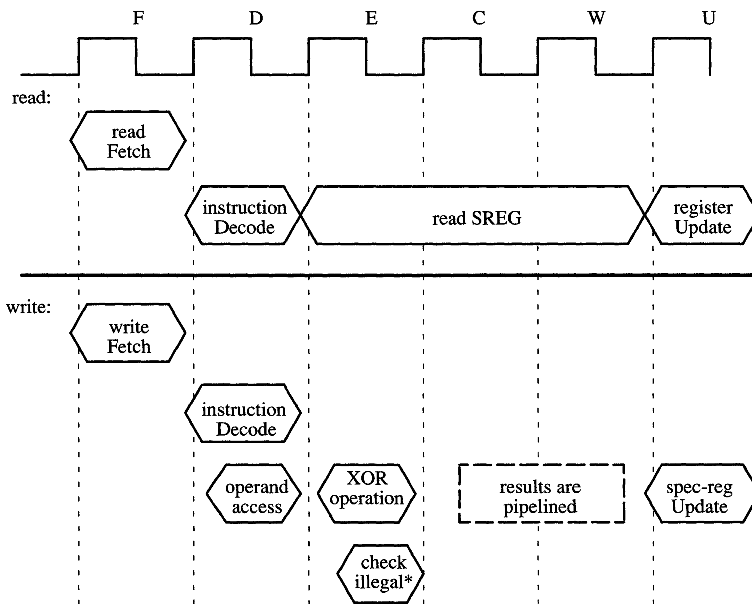


Figure 3–48. Typical SAVE/RESTORE Instruction Pipeline



* for WRPSR only

Figure 3–49. Typical Read/Write Special Register Instruction Pipeline

3.8.11 Read/Write Special Register Instructions

The Read and Write Special Register instructions move information between the integer register file and SPARC special control registers. *Figure 3–49* illustrates the pipeline activities for both instructions. All of these instructions are privileged instructions other than RDY and WRY. Exception detection is performed during the Decode stage except for the special case of WRPSR which is checked in the Execute stage to confirm that the CWP points to a valid window.

The Read Special Register instructions load the special register values into the ALU writeback result buffer used for forwarding during the Writeback stage. The register file is updated (as usual) during the clock following the Writeback stage (i.e., in the Update stage).

The write special register instructions behave like most ALU instructions. During the Decode stage, a register access is made. During the Execute stage an XOR is performed on the operands. During the Update stage the specified special register is updated.

3.8.12 Special Feature Pipelines

There are several special features of the RT620 which exploit multiple instruction launch and provide performance speedups. Additional information regarding scheduling activities associated with these instruction combinations can be located in the corresponding subsections in *Section 3.4*.

3.8.12.1 Fast Constant Pipeline

When a specific combination of two instructions used to generate a 32-bit constant is contained in an instruction packet, it is possible to execute both instructions in parallel. *Figure 3–50*(a) shows the typical sequential instruction execution used to generate a 32-bit constant. *Figure 3–50*(b) shows parallel execution provided by the RT620.

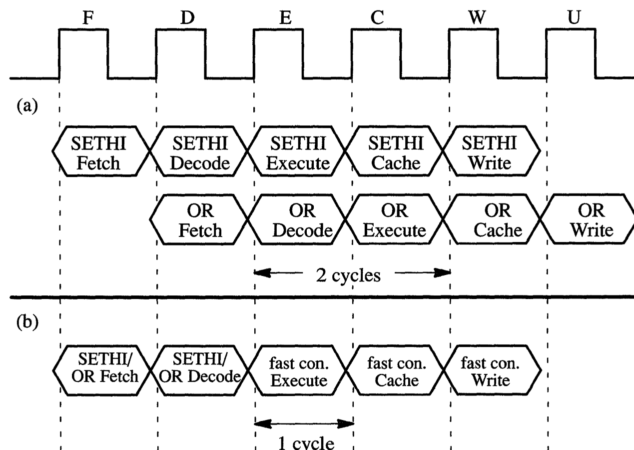


Figure 3–50. Fast Constant Instruction Pipeline

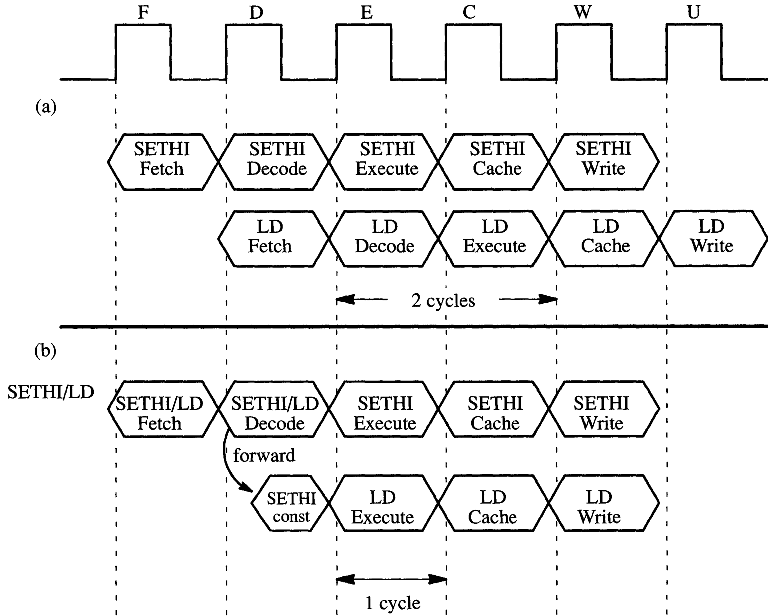


Figure 3–51. Fast Index Instruction Pipeline

3.8.12.2 Fast Index Pipeline

When a specific combination of two instructions used to generate a 32-bit memory index is contained in an instruction packet, it is possible to execute both instructions in parallel. *Figure 3–51*(a) shows the typical sequential instruction execution used to generate a 32-bit base address for array indexing. *Figure 3–51*(b) shows the parallel execution for base address construction provided by the RT620. Again, the speedup factor is two times that of sequential execution.

3.8.12.3 Fast Branch

Fast Branch is a feature which avoids waiting for the outcome of an ALU instruction setting condition codes in order to initiate a Branch instruction target fetch. This situation occurs when an integer Branch is immediately preceded by an ALUcc-type instruction in the same packet. In such a case, it is possible to launch both instructions simultaneously. *Figure 3–52* shows sequential execution of two instructions which make up a *Fast Branch* pair. Notice that the cycles per instruction execution (CPI) count is 4 cycles ÷ 4 instructions = 1.0 CPI.

Figure 3–53 shows timing results when Fast Branch is applied. When the branch is taken, a one clock speed up occurs and the CPI now becomes 3 cycles ÷ 4 instructions = 0.75 CPI. When the branch is not taken, some logic must be exercised to fetch instructions from the original instruction stream. Even when the delay slot instruction (DSI) and the instruction which follows the DSI (DSI+4) cannot be executed in parallel, the 0.75 CPI still holds (i.e., the performance is still better than that for sequential execution).

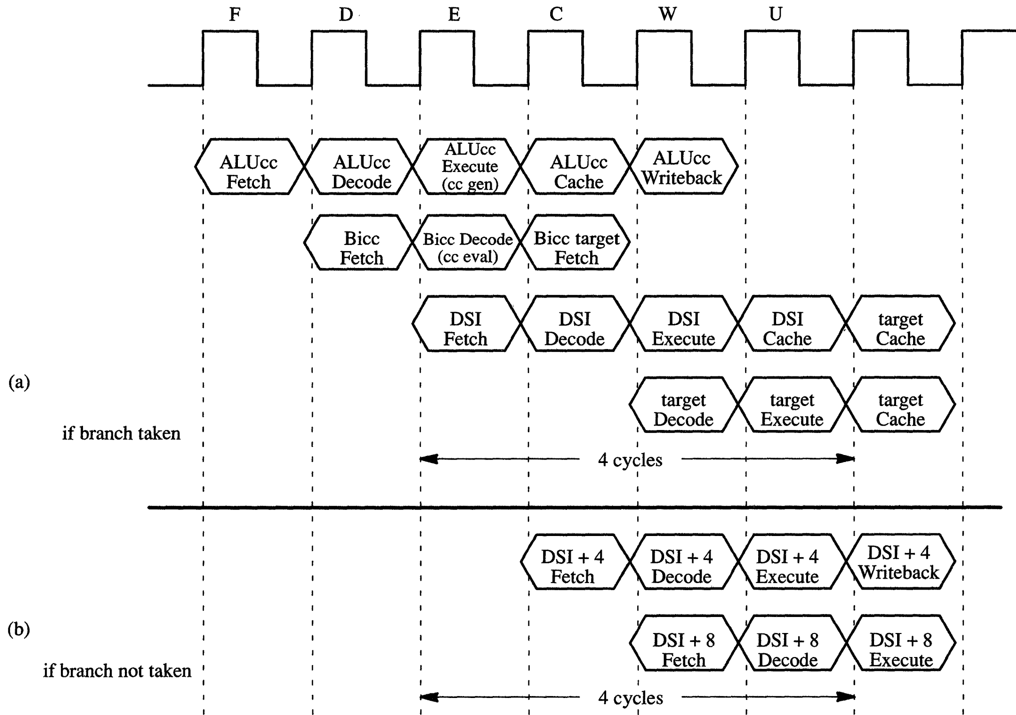


Figure 3–52. Sequential Branch Instruction Pipeline

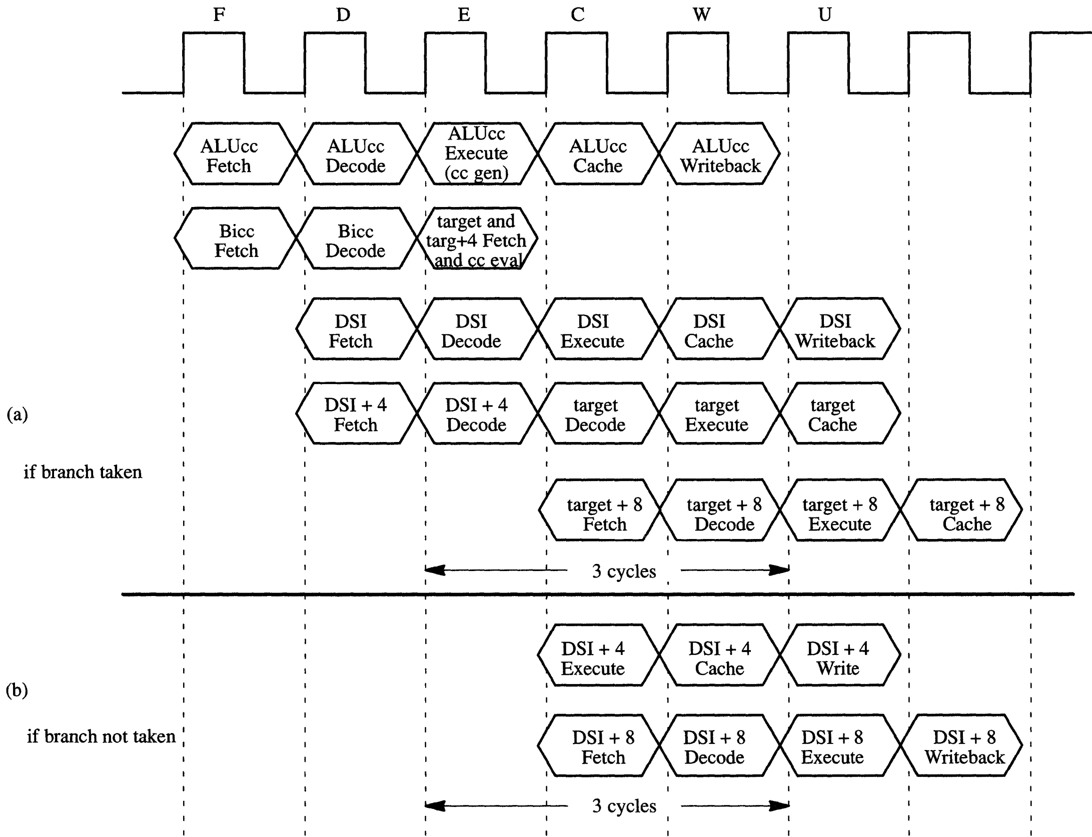


Figure 3–53. Fast Branch Instruction Pipeline

3.8.13 Floating-Point Instruction Pipelines

There are two types of pipeline instructions executed by the floating-point unit. These are *typical* floating-point unit instructions and *multiple-cycle* floating-point unit instructions. The typical floating-point unit instructions follow a standard six-stage sequence (both for single and double precision computations). Multiple cycle instructions require iterative algorithms to produce results and require multiple execution stages. The number of execution stages depends on the type of fp operation and the precision of the required result. All instructions in the Floating-Point Arithmetic Unit (FAU) group and two instructions in the Floating-Point Multiplier Unit (FMU) group follow the typical six-stage pipeline. These six stages are: Fetch (F), Decode (D), Execute1 (E1), Execute2 (E2), Round (R), and Update (U).

Note that there is a Round stage forwarding mechanism in both the FAU and FMU units.

Figure 3–54 illustrates the typical floating-point unit instruction pipeline. Instructions are fetched in the same way as integer instructions but they must be decoded locally by the floating-point unit decoder. As with ALU and LSU instructions, during the Decode stage, the fp register operands are accessed during the decode stage. During the Execute1 stage, operands are checked for validity and converted to the internal representation. Fractions and exponents are aligned for instruction execution. Several IEEE exceptions are determined at this time by detecting operands which are NaN, Denormalized, or Infinity.

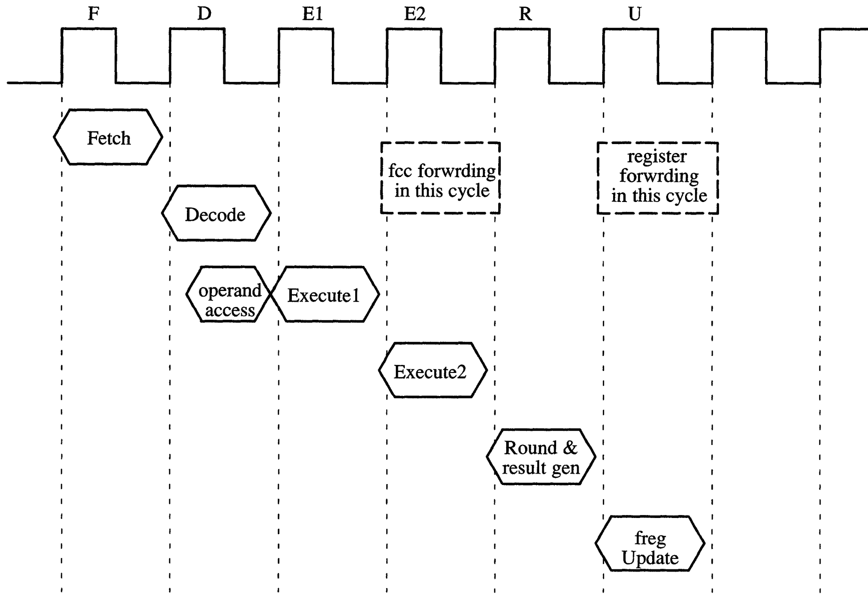


Figure 3–54. Typical Floating-Point Unit Instruction Pipeline

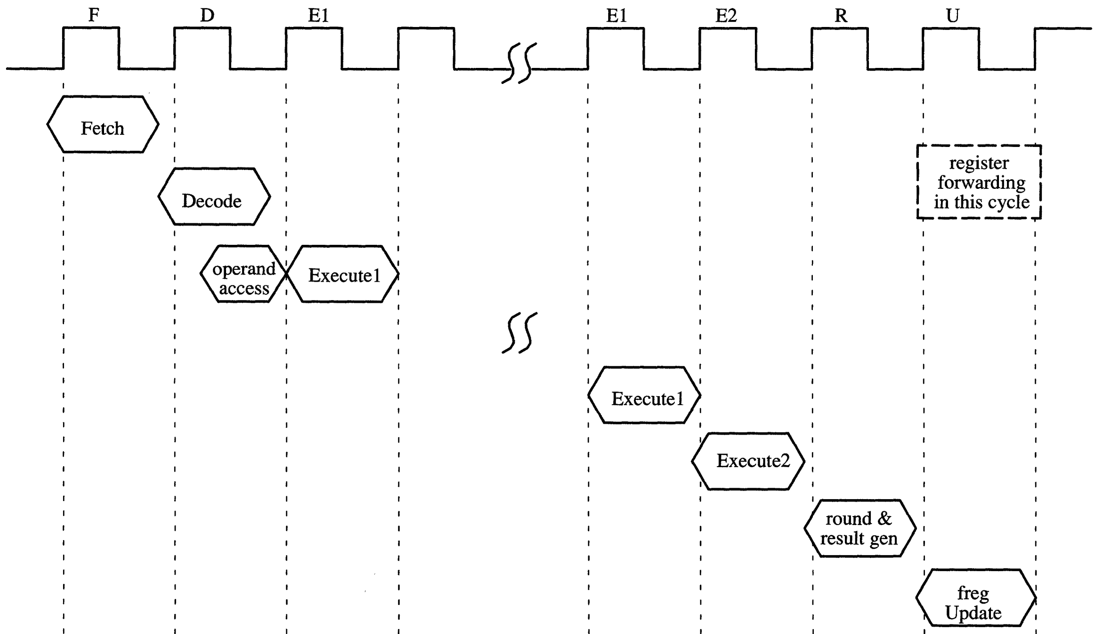


Figure 3–55. Typical Multiple Cycle Floating-Point Unit Instruction Pipeline

Once the operands have been correctly aligned for execution (e.g. for adds and subtracts, a common exponent must be established), the instruction is actually performed. IEEE rounding, result generation, and additional exception detection on the result is performed during the Round stage. Assuming no exception has been generated, the register file can be updated during the Update stage.

Note that the fp condition codes can be forwarded at the beginning of the Execute2 stage and that results can be forwarded at the beginning of the Update stage. *Table 3–6* provides instruction execution performance for the typical floating-point unit instructions.

For multiple cycle instructions, results cannot be computed in two execution stages and one Round stage except if the operation involves special case operands in which the result can be forced. The algorithms required to generate results require a series of iterations using intermediate results. *Figure 3–55* shows the basic multiple cycle fp instruction pipeline. Other than the iteration associated with the execution stages, it is identical to the basic fp instruction pipeline.

Note that when a multiple cycle instruction is in the fp post queue, it occupies the Execute1 stage (post queue entry 2) until the last (nth) iteration of its algorithm.

Table 3–6. Typical Floating-Point Unit Instruction Cycle Times

Instruction	CPI	**Instruction**	**CPI**	Instruction	CPI
FABSs	1	FDIVd	12*	FMULs	1
FADDs	1	FiTOs	1	FsMULd	1
FADDd	1	FiTOd	1	FMULd	1*
FCMPs	1	FsTOi	1	FNEGs	1
FCMPd	1	FsTOd	1	FSQRTs	11*
FCMPEs	1	FdTOi	1	FSQRTd	17*
FCMPEd	1	FdTOs	1	FSUBs	1
FDIVs	8*	FMOVs	1	FSUBd	1

* An additional cycle is required if the instruction in the table above is followed by a FMULs, FMULd, FsMULd, FDIVs, FDIVd, FSQRTs or FSQRTd.

3.9 Traps and Interrupts

The RT620 supports three categories of traps. These are *precise* traps, *deferred* traps, and *interrupting* traps. Precise traps correspond to exceptions induced by a particular instruction which occur before the instruction has changed t he program-visible state of the machine. When a precise trap occurs (except in the case of a power-on reset trap), several conditions must hold. These are: (1) the instructions before the trap inducing instruction will have completed execution, (2) the instructions after the trap inducing instruction remain unexecuted, and (3) the PC and nPC point to the trap inducing instruction and to the instruction which was to be executed next.

Deferred traps correspond to traps induced by a particular instruction, but may occur after the program-visible state has changed. For SPARC architectures, this trap type occurs in floating-point (fp) exceptions, where the trap inducing instruction is executed, but the deferred trap is not taken until one or more instructions after the trap-inducing instruction. Associated with the deferred trap there must exist the following: (1) an instruction that provokes a potentially outstanding deferred-trap exception to be taken as a trap, (2) the ability to resume execution of the trapped instruction stream and (3) privileged instructions that access the state required for the supervisor to emulate the deferred trap-inducing instructions and resume execution of the trapped instruction stream (i.e., access the deferred trap queue). The deferred trap must be taken before the execution of any instruction which depends on the trap-inducing instruction.

Interrupting traps are controlled by the enable trap (ET) and processor interrupt level (PIL) level fields of the PSR and do not correspond to precise traps or deferred traps. These traps were previously referred to as asynchronous traps. These interrupting exceptions are generated by external hardware.

3.9.1 Machine State at Reset

The reset trap is triggered asynchronously by asserting the external $\overline{\text{PRST}}$ signal. When the hyperSPARC CPU recognizes the $\overline{\text{PRST}}$ signal, it enters reset mode and stays there until the $\overline{\text{PRST}}$ line is deasserted. The processor then enters Execute mode and executes the trap procedure. The specific actions taken during reset are:

- ET (enable trap) bit in the PSR is set to 0.
- S (supervisor) bit in the PSR is set to 1.
- PC is set to 0 and nPC is set to 4.
- FTD bit in the ICCR is set to 0 and the ICE bit is set to 0.

All other fields in the above programmer-accessible registers and all other registers are unspecified on a power-up reset; if the reset is not a power-up reset, they retain their values from the last execution mode.

Internally, on a reset trap

- All the valid bits in the ICACHE are cleared.
- The valid bits in the instruction fetch buffers are cleared.
- The valid bits of all the FPQ entries are cleared. Because all FPQ entries are cleared, the qne bit will also be initially cleared.
- Control is then transferred to location 0.

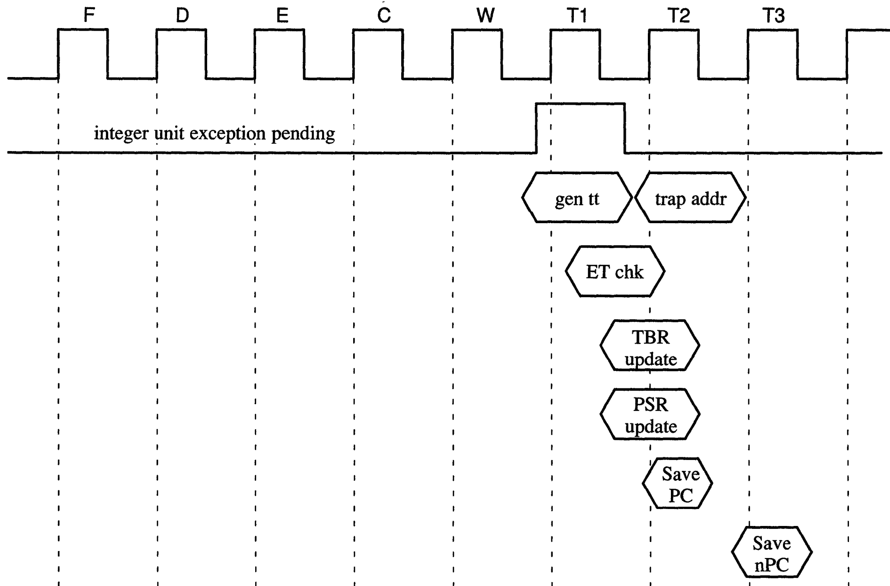


Figure 3–56. Exception Pipeline

3.9.2 Exception Pipeline

A trap is always taken at the end of the Writeback stage. Instructions which preceded the instruction which caused the trap are allowed to complete register Update before the trap is recognized. At the end of this Writeback stage, three additional pipeline stages are "appended" to support trap handling. These stages are referred to as T1, T2, and T3. *Figure 3–56* shows these stages along with the activities performed during each stage. Note that if T1 is entered while the *trap enable*d (ET) bit in the PSR is cleared, the processor enters ERROR mode.

There is a special case involving an ALU instruction in slot-a paired with a load instruction in slot-b. If both instructions are launched together and the load instruction causes a data access trap, the ALU instruction is allowed to complete its register Update.

3.9.3 Trap Operation

SPARC traps are enabled or disabled with the enable traps (ET) bit in the processor state register (PSR). If traps are enabled, the following occurs upon recognizing a trap:

- The current window pointer (CWP) in the PSR is decremented, thus changing the CWP. Note that this is done without regard to the WIM register and without checking for window overflow.
- The existing user/supervisor mode is preserved in the PS bit: S → PS.
- The user/supervisor mode is changed to supervisor: 1 → S.
- The RT620 automatically saves the PC in r[17] (local register 1) of the current window.
- nPC is automatically saved in r[18] (local register 2) of the current window.
- The ET bit of the PSR is cleared, thereby disabling traps.
- The trap type is automatically entered into the *tt* (trap type) field of the trap base register (TBR).
- If the trap is a reset trap, control is transferred to address 0: 0 → PC, 4 → nPC.
- If the trap is not a reset trap, control is transferred to the address specified by the TBR: TBR → PC, TBR+4 → nPC.

The trap table pointed to by the TBR has a space of four instructions for each trap vector specified by the TBR. This available slot of four instructions is generally used to jump to a trap handler routine.

If the enable trap bit is cleared (ET = 0) when a trap is recognized by the processor, the RT620 enters error mode and halts execution. If ET = 0 and an interrupt or deferred exception occurs, it is ignored.

3.9.4 Error Mode

SPARC processors enter error mode or state when a trap occurs while traps are disabled (when the ET bit of the PSR is set to zero). Upon encountering this event, the processor halts execution and asserts the $\overline{\text{PERROR}}$ signal. Standard trap actions, such as decrementing CWP and saving the program counters (PC and nPC) in r[17] and r[18] do not occur. The *tt* field of the TBR is not written except in the case of a RETT instruction that traps while ET = 0. In this singular case, the *tt* field is written to indicate the exception type induced by the RETT instruction. The method of handling error mode is implementation dependent, but it is typically handled by causing the processor to trigger an external reset.

3.9.5 Trap Priorities

Each trap type is assigned a priority; priority 1 is the highest priority and priority 31 the lowest. If two traps occur simultaneously, the highest priority trap is taken. The RT620 avoids the case of two traps occuring with the same priority by disallowing simultaneous launch of instructions which can cause exceptions at the same priority level.

The trap table (used to direct traps to the appropriate trap handler) is divided into two halves. The first half is dedicated to hardware exceptions and the second half is dedicated to software exceptions. Hardware exceptions are detected by the processor logic. Software traps are accessed through the Ticc instruction. *Table 3–7* lists exceptions supported by the RT620:

Table 3–7. RT620 Supported Exceptions

Exception	Priority	tt
reset	1	n/a
instruction access exception	5	0x01
privileged instruction	6	0x03
illegal instruction	7	0x02
floating-point disabled	8	0x04
coprocessor disabled	8	0x24
unimplemented flush	8	0x25
window overflow	9	0x05
window underflow	9	0x06
memory address not aligned	10	0x07
floating-point exception	11	0x08
data access exception	13	0x09
tag overflow	14	0x0a
division by zero	15	0x2a
trap instruction	16	0x80 through 0xff
interrupt level 15	17	0x1f
interrupt level 14	18	0x1e
interrupt level 13	19	0x1d
interrupt level 12	20	0x1c
interrupt level 11	21	0x1b
interrupt level 10	22	0x1a
interrupt level 9	23	0x19
interrupt level 8	24	0x18
interrupt level 7	25	0x17
interrupt level 6	26	0x16
interrupt level 5	27	0x15
interrupt level 4	28	0x14
interrupt level 3	29	0x13
interrupt level 2	30	0x12
interrupt level 1	31	0x11

3.9.6 Precise Traps

Precise trap exception conditions are detected during any one of the integer instruction pipeline stages. The priority encoding of these exceptions is performed at each pipeline stage as the instruction progresses. *Table 3–8* identifies the pipeline stage at which each exception is detected during instruction execution.

Table 3–8. Pipeline Stage Exception Recognition

Decode	Execute	Writeback
instruction access	illegal instruction	data access
privilege violation	window overflow	
illegal instruction	window underflow	
floating-point unit disabled	memory not aligned	
CP disabled	tag overflow	
floating-point exception	floating-point exception	floating-point exception
trap instruction (ticc)	interrupt	
unimplemented flush	divide-by-zero	

A pending fp exception is recognized in Decode if the instruction is a FBfcc, in Execute if the instruction is an fp Load or an FPop, and in Writeback if the instruction is an fp store. Note that an fp exception is always taken regardless of the exception priority if the instruction is valid (for example, if an fp Store recognizes an fp exception and there is a misaligned exception on the Store, the fp exception trap is taken even though a misaligned trap has higher priority).

In the RT620, an illegal instruction exception is signaled when an integer instruction is not implemented.

An illegal instruction exception is recognized in Decode except for an illegal instruction exception due to a WRPSR instruction with an invalid CWP which is recognized in Execute.

3.9.7 Interrupting Traps (Asynchronous)

For interrupt requests, the MIRL<3:0> is compared against the processor interrupt level (PIL) of the processor status register (PSR). If traps are enabled (ET = 1) and if MIRL<3:0> is greater than the PIL, or if MIRL<3:0> = 0x15 (a non-maskable interrupt), the interrupt trap is taken if no higher priority traps are outstanding.

Note that an interrupt will be masked when:

- traps are disabled (ET = 0) or
- the interrupt level on the external lines (MIRL<3:0>) is less than or equal to the PIL in the PSR (unless MIRL<3:0> = 15, which is a non-maskable interrupt).

Interrupts are sampled two successive times at a sampling rate of one-half of the IMCLK frequency before a new MIRL<3:0> value is recognized by the processor. The value of MIRL<3:0> must be held until the interrupt trap is taken. This is usually accomplished by coding the interrupt trap routine to access a user-defined system interrupt control register and clear the interrupt.

3.9.8 Floating-Point Unit Traps (Deferred Traps)

There are four types of fp exceptions:

Unimplemented fp instruction: This exception is signaled when the floating-point unit local decoder determines that the opcode bit pattern is invalid or the instruction is a SPARC fp instruction that is not supported in hardware in the RT620 (e.g., FADDx).

Unfinished fp instruction: This exception is signaled when the floating-point unit is operating in standard mode and any of the inputs is a denormalized number or when the output is a tiny number before rounding, with the following exceptions:

1. One operand is a NaN, or during an add or subtract operation.
2. One operand is a NaN, or during a multiply or divide operation.
3. The operand is a negative denormalized number during a square root operation.
4. The instruction is an absolute value (FABSs), move (FMOVs), negate (FNEGs), compare (FCMPs, FCMPd, FCMPEs, FCMPEd, or convert floating-point to/from integer (FdTOi, FsTOi, FiTOs, FiTOd).

Sequence Error: This exception is signaled when the floating-point unit is in exception (TRAP) mode and an fp instruction other than an fp store is decoded. This exception is also signaled when a STDFQ instruction is issued when the floating-point unit is not in EXCEPTION mode.

IEEE Exceptions: This is a class of exceptions defined by the IEEE-754 standard. The specific exception is signaled through the *cexc* field of the FSR.

The fp exceptions are signaled through the *ftt* field in the FSR as shown below.

Table 3–9. FTT Field of FSR

ftt Field	Exception Type
000	none
001	IEEE
010	Unfinished
011	Unimplemented
100	Sequence error
101	Hardware error (not supported in RT620)
110	reserved
111	reserved

The exceptions are detected by the FPQC in the Round stage, or previous pipeline stages in which case they advance through the fp pipeline stages until they reach the Round stage. All the exceptions cause fp deferred traps except those corresponding to IEEE exceptions. For a particular IEEE exception, the IEEE exception will cause a deferred trap only if the corresponding bit in the *trap enable* mask field in the FSR is set. On recognizing a trap condition, the floating-point unit enters exception pending mode and asserts the fp exception pending signal. The next fp instruction to enter the instruction decoder will cause the integer unit to acknowledge the fp exception and take an fp trap as shown in *Figure 3–57*. When the floating-point unit receives the exception acknowledgement signal, it goes into EXCEPTION mode as discussed in *Section 3.5*.

As defined in the SPARC architecture, when an fp trap occurs:

- the destination register is unchanged.
- the FSR *fcc* field is unchanged.
- the FSR *aexc* field is unchanged.
- the FSR *cexc* field is unchanged except for an IEEE-754 exception; in this case, the *cexc* contains exactly one bit (which is 1) corresponding to the exception which caused the trap.

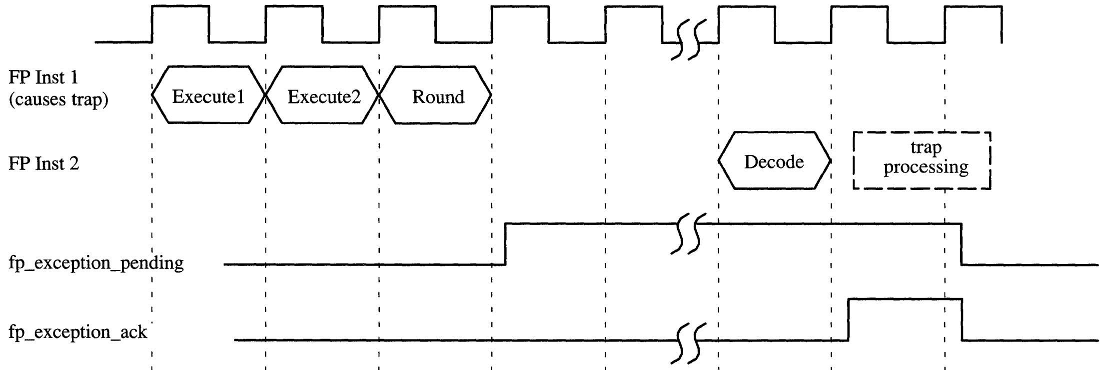


Figure 3–57. Floating-Point Exception Pipeline

When an fp exception is acknowledged by the integer unit, an fp exception trap handler is invoked and the floating-point unit enters exception mode (also referred to as trap mode). Typically, this trap handler executes a sequence of store double floating-point queue (STDFQ) instructions to empty the queue of its address and instructions. The FPQ is emptied starting at post-queue entry 0 and working back through to pre-queue entry 6, skipping invalid entries. If any fp instruction other than a store instruction is attempted, a sequence error occurs and the processor remains in exception mode. When the FPQ is emptied, the floating-point unit returns to normal execution mode.

The instruction decoder checks to see if there is an fp exception pending in different stages depending on the instruction being decoded as given in the following table.

Table 3–10. FP Pipeline Stage Exception Recognition

Instruction	Stage
FBfcc	Decode
FPop	Execute
fp Load	Execute
fp Store	Writeback

FP exceptions are checked in the Decode stage if the instruction is an fp Branch to maintain correct PCs to save in the trap handler.

In case of FPop instructions, fp exception is checked for in the Execute stage. If this were done in the Decode stage, then in the case where the exception pending signal from the floating-point unit arrives in the Execute cycle, the integer unit would assume that the instructions had been successfully launched. However, the floating-point unit would not accept the instructions, and the instructions would then be "lost."

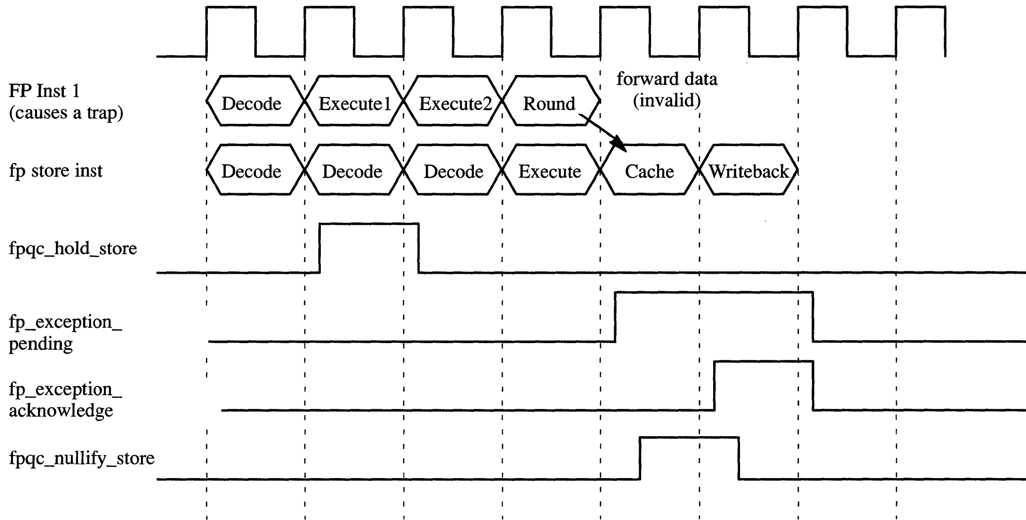


Figure 3–58. Floating-Point Exception during Forwarding to Store

In case of fp Load instructions, fp exceptions are checked in the Execute stage. This is because no dependency checks are done for a load in Decode stage against the instruction in entry 0 (the Round stage of the instruction); if the instruction should cause an fp exception pending in the next cycle, it would be detected in the Execute cycle of the Load.

The instruction decoder checks for the exception pending signal in the Writeback stage of the Store instruction. This case is illustrated in *Figure 3–58*. The Store instruction is waiting for data to be forwarded from the fp instruction. However, the Store instruction anticipates the completion of the fp instruction and goes into the Execute stage in the same cycle that the fp instruction enters Round stage. If the fp instruction takes a trap, then the exception pending signal arrives too late for the instruction decoder to recognize the exception while the Store is in Execute. Since the data forwarded is invalid in the case of an exception, the Store instruction needs to be cancelled and the exception has to be recognized. The Store instruction is cancelled by the FPQC which sends the cancel signal (in the next cycle) to the IBIU if there is an fp Store in Execute and there is an exception pending. The instruction decoder checks for an exception pending signal from the floating-point unit in the Writeback stage of the Store and takes the fp exception only if the exception pending signal was also asserted in the previous cycle (if the exception pending signal was asserted starting in the Writeback stage, then the fp exception is not recognized on the Store instruction since it is too late to annul the Store).

3.9.8.1 IEEE Exceptions

There are five RT620 supported IEEE exceptions: *inexact*, *underflow*, *overflow*, *invalid*, and *divide by zero*.

The conditions under which these exceptions occur are described in the IEEE-754 standard. However, a few details of the IEEE–754 standard are implementation dependent. Also, the RT620 does not handle denormalized operands in all cases. The behavior of the RT620 in the case of IEEE–754 exceptions is described in the following sections. Note that there cannot exist more than two exception cases simultaneously. If there are two exception cases, one exception will be an inexact exception. In the standard mode, two exception cases can arise when an overflow exception occurs; in the non-standard mode, an inexact exception can arise along with any other exception.

In non-standard mode, denormalized numbers at the input or output are substituted by zero if the case is such that an unfinished exception would result in the standard mode. Detailed description of the behavior of each individual instruction in standard and non-standard mode is given in *Section 3.9.8.6*.

NXM refers to the inexact bit in the term, *NX* indicates that an inexact trap is taken (the *nx* bit is set in the *cexc*), and *nx* means that the inexact bit is set in the *cexc* and ORed into the *aexc* but no trap is taken. Similar abbreviations are used for the other exception types.

3.9.8.2 Inexact exception

3.9.8.2.1 Standard mode:

The inexact exception can arise due to two causes:

1. There is an overflow condition as discussed in *Section 3.9.8.3.3*.
2. The result is inexact as defined by the IEEE standard and there is no other exception.

3.9.8.2.2 Non-Standard Mode:

In addition to the exception cases in the standard mode, an inexact exception also occurs if denormalized numbers at the input or output are substituted by zero; the inexact exception will trap only if NXM = 1 and no other higher priority IEEE exception trap occurs as discussed in the tables below.

3.9.8.3 Underflow Exception

3.9.8.3.1 Standard Mode:

In the standard mode of operation, there is no underflow exception; an unfinished trap is taken if the result underflows.

3.9.8.3.2 Non-Standard Mode:

If tininess is detected in the result of an operation (before rounding), then an underflow condition exists. The state of the inexact and underflow *trap enable* mask bits in the FSR decides the course of action taken.

If both the inexact and the underflow traps are disabled, then the uf and nx bits in the *cexc* field of the FSR are set and also ORed into the aexc, and the result is replaced by zeros *with the same sign.* If the uf and/or the nx traps are/is enabled, then the trap which is taken is prioritized (remember that only one bit in the *cexc* should be set when an fp trap due to an IEEE exception is taken).

The following table indicates the priority between the IEEE exceptions when there is a denormalized result with the floating-point unit in non-standard mode (Note: XX denotes bit set and trap taken, xx denotes bit set only).

Trap Enable State	FSR Update (NS = 1)
UFM = 1, NXM = x	UF
UFM = 0, NXM = 1	NX
UFM = 0, NXM = 0	nx, uf

3.9.8.3.3 Overflow Exception:

Overflow behaves identically in the non-standard mode and the standard mode. The following table defines the behavior when an overflow exception occurs (Note: XX denotes bit set and trap taken, xx denotes bit set only).

Trap Enable State	FSR Update
OFM = 1, NXM = x	OF
OFM = 0, NXM = 1	NX
OFM = 0, NXM = 0	nx, of

3.9.8.4 Invalid Exception

3.9.8.4.1 Standard Mode:

This exception occurs as specified in the IEEE 754-1985 standard.

3.9.8.4.2 Non-Standard Mode:

In the non-standard mode, in addition to the standard-mode exceptions, if the exception occurs due to substitution of denormalized numbers, the exceptions are prioritized as follows (Note: XX denotes bit set and trap taken, xx denotes bit set only).

Trap Enable State	FSR Update (NS = 1)
NVM = 1, NXM = x	NV
NVM = 0, NXM = 1	NX
NVM = 0, NXM = 0	nx, nv

3.9.8.5 Divide by Zero Exception

3.9.8.5.1 Standard Mode:

This exception occurs as specified in the IEEE 754-1985 standard.

3.9.8.5.2 Non-Standard Mode:

In the non-standard mode, in addition to the standard-mode exceptions, if the exception occurs due to substitution of denormalized numbers, the exceptions are prioritized as follows (Note: XX denotes bit set and trap taken, xx denotes bit set only).

Trap Enable State	FSR Update (NS = 1)
DZM = 1, NXM = x	DZ
DZM = 0, NXM = 1	NX
DZM = 0, NXM = 0	nx, dz

3.9.8.6 Result Generation

3.9.8.6.1 Standard Mode Result Generation

The following sequence of tables provides the results generated by the floating-point unit for various combinations of input operands when the processor is operating in standard mode and the corresponding *trap*

enable mask bit for an IEEE exception is cleared. However, in the case of an unfinished exception, the trap is always taken.

In the following tables, terms are abbreviated as follows:

Inum an integer.

Norm a normalized number.

DeNorm a denormalized number.

QNaN a quiet NaN.

SNaN a signaling NaN.

QNaN*n* a quiet NaN which appears at the rs n source operand input.

SNaN*n* a signaling NaN which appears at the rs n source operand input.

QSNaN*n* a quiet NaN produced by the NaN transformation (described below) on a signaling NaN from rs n.

QQNaN*n* a quiet NaN produced by the NaN transformation (described below) on a quiet NaN from rs n.

UNFNSH an unfinished exception occurs (trap is taken and destination is unchanged).

NX an IEEE inexact exception occurs.

NV an IEEE invalid exception occurs.

OF an IEEE overflow exception occurs.

DZ an IEEE divide-by-zero exception occurs.

UF an IEEE underflow exception occurs (only in non-standard mode).

= the fcc is set to 0 (equal).

< the fcc is set to 1 (smaller).

> the fcc is set to 2 (greater).

? the fcc is set to 3 (unordered).

RN round to nearest.

RP round to $+\infty$.

RM round to $-\infty$.

RZ round to zero.

$\mathbf{0}_D$ zero delivered at the output in non-standard mode when the output result is a denorm.

$\mathbf{R}_{IEEE}$ number delivered at the output in case of an overflow as specified by IEEE-754.

NaN transformation: The most significant bits of the operand fraction are copied to the most significant bits of the result fraction. When converting to a narrower format, excess low order bits are discarded. When converting to a wider format, excess lower order bits of the result fraction are set to 0. The quiet bit (most significant bit of the result fraction) is always set to 1, thus the NaN transformation always produces a quiet NaN.

FMOVs:

rs2	result
+ Norm	+ Norm
– Norm	– Norm
+ De Norm	+ DeNorm
– DeNorm	– DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNaN2 (1)
+ 0	+ 0
– 0	– 0
+ ∞	+ ∞
– ∞	– ∞

Note 1: The input is delivered to the output without any change.

FABSs:

rs2	**result**
+ Norm	+ Norm
– Norm	– Norm
+ De Norm	+ DeNorm
– DeNorm	+ DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNaN2 (1)
+ 0	+ 0
– 0	+ 0
+ ∞	+ ∞
– ∞	+ ∞

Note 1: If the sign bit is 0, the input is delivered to the output without any change.
If the sign bit is 1, the input is delivered to the output with the sign bit cleared.

FNEGs:

rs2	**result**
+ Norm	– Norm
– Norm	+ Norm
+ De Norm	– DeNorm
– DeNorm	+ DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNzN2 (1)
+ 0	– 0
– 0	+ 0
+ ∞	–∞
– ∞	+∞

Note 1: The input is delivered to the output with the sign bit inverted.

FADDs, FADDd:

rs1/rs2	+ Norm	– Norm	+/– DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	UNFNSH	QSNaN2 NV	QNaN2	+ Norm	+ Norm	+ ∞	– ∞
– Norm	(2)	(3)	UNFNSH	QSNaN2 NV	QNaN2	+ Norm	– Norm	+ ∞	– ∞
+/– DeNorm	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	UNFNSH	UNFNSH	+ ∞	– ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ Norm	– Norm	UNFNSH	QSNaN2 NV	QNaN2	+ 0	0 (5)	+ ∞	– ∞
– 0	+ Norm	– Norm	UNFNSH	QSNaN2 NV	QNaN2	0 (5)	–0	+ ∞	– ∞
+ ∞	+ ∞	+ ∞	+ ∞	QSNaN2 NV	QNaN2	+ ∞	+ ∞	+ ∞	QNaN NV (4)
– ∞	– ∞	– ∞	– ∞	QSNaN2 NV	QNaN2	– ∞	– ∞	QNaN NV (4)	– ∞

Note 1: outputs possible: + Norm, R_{IEEE} & OF & NX, + Norm & NX.
Note 2: outputs possible: rs1 ≠ –rs2: +/– Norm, UNFNSH, +/– Norm & NX.
rs1 = rs2: + 0 or – 0 depending on rounding mode (see (5)).
Note 3: outputs possible: – Norm, R_{IEEE} & OR & NX, – Norm & NX.
Note 4: QNaN delivered at output = 7ff..f
Note 5: returns + 0 in rounding modes RN, RZ, and RP; returns –0 in rounding mode RM.

FSUBs, FSUBd:

rs1/rs2	+ Norm	– Norm	+/– DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(3)	UNFNSH	QSNaN2 NV	QNaN2	+ Norm	+ Norm	– ∞	+ ∞
– Norm	(2)	(1)	UNFNSH	QSNaN2 NV	QNaN2	– Norm	– Norm	+ ∞	– ∞
+/– DeNorm	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	UNFNSH	UNFNSH	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	– Norm	+ Norm	UNFNSH	QSNaN2 NV	QNaN2	0 (5)	+ 0	– ∞	+ ∞
– 0	– Norm	+ Norm	UNFNSH	QSNaN2 NV	QNaN2	– 0	–0 (5)	– ∞	+ ∞
+ ∞	+ ∞	+ ∞	+ ∞	QSNaN2 NV	QNaN2	+ ∞	+ ∞	QNaN NV(4)	+ ∞
– ∞	– ∞	– ∞	– ∞	QSNaN2 NV	QNaN2	– ∞	– ∞	–∞	QNaN NV(4)

Note 1: output possible: rs1 ≠ –rs2: +/– Norm, UNFNSH, +/– Norm & NX.
rs1 = rs2: + 0 or – 0 depending on rounding mode (see (5)).
Note 2: output possible: – Norm, R_{IEEE} & OF & NX, – Norm & NX.
Note 3: output possible: + Norm, R_{IEEE} & OF & NX, + Norm & NX.
Note 4: QNaN delivered at output = 7ff..f
Note 5: returns + 0 in rounding modes RN, RZ, and RP; returns –0 in RM.

FsTOi, FdTOi:

rs2	result
+ Norm	(1)
– Norm	(2)
+ De Norm	+ 0 NX
– DeNorm	– 0 NX
SNaN2	NV (3)
QNaN2	NV (3)
+ 0	+ 0
– 0	– 0
+ Inf	NV (4)
– Inf	NV (5)

Note 1: outputs possible: +Inum, NV if output overflows (see (3)), +Inum & NX.
Note 2: outputs possible: – Inum, NV if output overflows (see (3)), – Inum & NX.
Note 3: if sign bit is 0, deliver 7ff..f; if sign bit is 1, deliver 800..0
Note 4: deliver 7ff..f at output.
Note 5: deliver 800..0 at output.

FiTOs, FiTOd:

rs2	result
+ 0	+ 0
+ Inum	(1)
– Inum	(2)

Note 1: outputs possible: FITOS: + Norm, +Norm & NX.
FITOD: + Norm
Note 2: outputs possible: FITOS: – Norm, – Norm & NX.
FITOD: – Norm.

FSTOD	
rs2	result
+ Norm	+ Norm
– Norm	– Norm
+ De Norm	UNFNSH
– DeNorm	UNFNSH
SNaN2	QSNaN2 NV
QNaN2	QQNaN2
+ 0	+ 0
– 0	– 0
+ ∞	+ ∞
– ∞	– ∞

FDTOS	
rs2	result
+ Norm	(1)
– Norm	(2)
+ De Norm	UNFNSH
– DeNorm	UNFNSH
SNaN2	QSNaN2 NV
QNaN2	QQNaN2
+ 0	+ 0
– 0	– 0
+ ∞	+ ∞
– ∞	– ∞

Note 1: outputs possible: + Norm, + Norm & NX, R_{IEEE} & OF & NX, UNFNSH.
Note 2: outputs possible: – Norm, – Norm & NX, R_{IEEE} & OF & NX, UNFNSH.

FMULs, FMULd:

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	+ 0	– 0	+ ∞	– ∞
– Norm	(2)	(1)	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	– 0	+ 0	– ∞	+ ∞
+DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	+ 0	– 0	+ ∞	– ∞
– DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	– 0	+ 0	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QNaN2	+ 0	– 0	QNaN NV (3)	QNaN NV (3)
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QNaN2	– 0	–0	QNaN NV(3)	QNaN NV (3)
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	+ ∞	– ∞
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	– ∞	+ ∞

Note 1: outputs possible:+ Norm, UNFNSH, R_{IEEE} & OF & NX, + Norm & NX.
Note 2: outputs possible: – Norm, UNFNSH, R_{IEEE} & OF & NX, – Norm & NX.
Note 3: QNaN delivered at output = 7ff..f

FsMULd:

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	+ Norm	– Norm	UNFNSH	UNFNSH	QSNaN2 NV	QQNaN2	+ 0	– 0	+ ∞	– ∞
– Norm	– Norm	+ Norm	UNFNSH	UNFNSH	QSNaN2 NV	QQNaN2	– 0	+ 0	– ∞	+ ∞
+DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QQNaN2	+ 0	– 0	+ ∞	– ∞
– DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QQNaN2	– 0	+ 0	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QQNaN1	QQNaN1	QQNaN1	QQNaN1	QSNaN2 NV	QQNaN2	QQNaN1	QQNaN1	QQNaN1	QQNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QQNaN2	+ 0	– 0	QNaN NV (1)	QNaN NV (1)
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QQNaN2	– 0	+ 0	QNaN NV (1)	QNaN NV (1)
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QQNaN2	QNaN NV (1)	QNaN NV (1)	+ ∞	– ∞
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QQNaN2	QNaN NV (1)	QNaN NV (1)	– ∞	+ ∞

Note 1: QNaN delivered at output = 7ff..f

FDIVs, FDIVd:

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	+ ∞ DZ	– ∞ DZ	+ 0	– 0
– Norm	(2)	(1)	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	– ∞ DZ	+ ∞ DZ	– 0	+ 0
+DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	+ ∞ DZ	– ∞ DZ	+ 0	– 0
– DeNorm	UNFNSH	UNFNSH	UNFNSH	UNFNSH	QSNaN2 NV	QNaN2	– ∞ DZ	+ ∞ DZ	– 0	+ 0
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	+ 0	– 0
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	– 0	+ 0
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QNaN2	+ ∞	– ∞	QNaN NV (3)	QNaN NV (3)
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QNaN2	– ∞	+ ∞	QNaN NV (3)	QNaN NV (3)

Note 1: outputs possible: + Norm, UNFNSH, R_{IEEE} & OF & NX, + Norm & NX.
Note 2: outputs possible: – Norm, UNFNSH, R_{IEEE} & OF & NX, – Norm & NX
Note 3: QNaN delivered at output= 7ff..f

FSQRTs, FSQRTd:

rs2	result
+ Norm	(1)
– Norm	QNaN NV (2)
+ DeNorm	UNFNSH
– DeNorm	QNaN NV(2)
SNaN2	QSNaN2 NV
QNaN2	QNaN2
+ 0	+ 0
– 0	– 0
+ ∞	+ ∞
– ∞	QNaN NV(2)

Note 1: outputs possible: + Norm, + Norm & NX.
Note 2: QNaN delivered = 7ff..f

FCMPs, FCMPd:

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	>	>	>	? NV	?	>	>	<	>
– Norm	<	(1)	<	<	? NV	?	<	<	<	>
+DeNorm	<	>	(1)	>	? NV	?	>	>	<	>
– DeNorm	<	>	<	(1)	? NV	?	<	<	<	>
SNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
QNaN1	?	?	?	?	? NV	?	?	?	?	?
+ 0	<	>	<	>	? NV	?	=	=	<	>
– 0	<	>	<	>	? NV	?	=	=	<	>
+ ∞	>	>	>	>	? NV	?	>	>	=	>
– ∞	<	<	<	<	? NV	?	<	<	<	=

Note 1: output possible: >, < , =

FCMPEs, FCMPEd:

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	>	>	>	? NV	? NV	>	>	<	>
– Norm	<	(1)	<	<	? NV	? NV	<	<	<	>
+DeNorm	<	>	(1)	>	? NV	? NV	>	>	<	>
– DeNorm	<	>	<	(1)	? NV	? NV	<	<	<	>
SNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
QNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
+ 0	<	>	<	>	? NV	? NV	=	=	<	>
– 0	<	>	<	>	? NV	? NV	=	=	<	>
+ ∞	>	>	>	>	? NV	? NV	>	>	=	>
– ∞	<	<	<	<	? NV	? NV	<	<	<	=

Note 1: output possible: >, < , =

3.9.8.6.2 Non-Standard Mode Result Generation

The following sequence of tables provides the results generated by the floating-point unit for various combinations of input operands when the processor is operating in non-standard mode and the corresponding *trap enable* mask bit for an IEEE exception is cleared. Note that the results are the same as in the standard-mode except in the cases when an unfinished exception would be taken. In such cases, denorms at the input are substituted by zero (nx flag is set). If the result is a denorm, it too is substituted by zero (uf and nx are set).

FABSs (NS):

rs2	result
+ Norm	+ Norm
– Norm	+ Norm
+ DeNorm	+ DeNorm
– DeNorm	+ DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNaN2 (1)
+ 0	+ 0
– 0	+ 0
+ ∞	+ ∞
– ∞	+ ∞

Note 1: If the sign bit is 0, the input is delivered to the output without any change.
Note 2: If the sign bit is 1, the input is delivered to the output with the sign bit cleared.

FMOVs (NS):

rs2	result
+ Norm	+ Norm
– Norm	– Norm
+ De Norm	+ DeNorm
– DeNorm	– DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNaN2 (1)
+ 0	+ 0
– 0	– 0
+ ∞	+ ∞
– ∞	– ∞

Note 1: The input is delivered to the output without any change.

FNEGs (NS):

rs2	result
+ Norm	– Norm
– Norm	+ Norm
+ DeNorm	– DeNorm
– DeNorm	+ DeNorm
SNaN2	SNaN2 (1)
QNaN2	QNaN2 (1)
+ 0	– 0
– 0	+ 0
+ ∞	– ∞
– ∞	+ ∞

Note 1: The input is delivered to the output with the sign bit inverted.

FADDs, FADDd (NS):

rs1/rs2	+ Norm	– Norm	+ DeNorm	– DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	+ Norm NX	+ Norm NX	QSNaN2 NV	QNaN2	+ Norm	+ Norm	+ ∞	– ∞
– Norm	(2)	(3)	– Norm NX	– Norm NX	QSNaN2 NV	QNaN2	+ Norm	– Norm	+ ∞	– ∞
+DeNorm	+ Norm NX	– Norm NX	+0 NX	(5) NX	QSNaN2 NV	QNaN2	+ 0 NX	(5) NX	+ ∞	– ∞
– DeNorm	+ Norm NX	– Norm NX	(5) NX	– 0 NX	QSNaN2 NV	QNaN2	(5) NX	– 0 NX	+ ∞	– ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ Norm	– Norm	+ 0 NX	(5) NX	QSNaN2 NV	QNaN2	+ 0	(5)	+ ∞	– ∞
– 0	+ Norm	– Norm	(5) NX	– 0 NX	QSNaN2 NV	QNaN2	(5)	–0	+ ∞	– ∞
+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	QSNaN2 NV	QNaN2	+ ∞	+ ∞	+ ∞	QNaN NV (4)
– ∞	– ∞	– ∞	– ∞	– ∞	QSNaN2 NV	QNaN2	– ∞	– ∞	QNaN NV (4)	– ∞

Note 1: outputs possible: + Norm, R_{IEEE} & OF & NX, + Norm & NX.
Note 2: outputs possible:
rs1 ≠ –rs2: +/– Norm, UNFNSH, +/– Norm & NX.
rs1 = rs2: + 0 or – 0 depending on rounding mode (see (5)).
Note 3: outputs possible: – Norm, R_{IEEE} & OF & NX, – Norm & NX.
Note 4: QNaN delivered at output = 7ff..f
Note 5: returns + 0 in rounding modes RN, RZ, and RP; returns –0 in rounding mode RM.

FSUBs, FSUBd (NS):

rs1/rs2	+ Norm	– Norm	+ DeNorm	– DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(3)	+ Norm NX	+ Norm NX	QSNaN2 NV	QNaN2	+ Norm	+ Norm	– ∞	+ ∞
– Norm	(2)	(1)	– Norm NX	– Norm NX	QSNaN2 NV	QNaN2	– Norm	– Norm	+ ∞	– ∞
+ DeNorm	– Norm NX	+ Norm NX	(5) NX	+ 0 NX	QSNaN2 NV	QNaN2	(5) NX	+ 0 NX	– ∞	+ ∞
– DeNorm	– Norm NX	+ Norm NX	–0 NX	(5) NX	QSNaN2 NV	QNaN2	– 0 NX	(5) NX	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	– Norm	+ Norm	(5) NX	+ 0 NX	QSNaN2 NV	QNaN2	(5)	+ 0	– ∞	+ ∞
– 0	– Norm	+ Norm	–0 NX	(5) NX	QSNaN2 NV	QNaN2	– 0	(5)	– ∞	+ ∞
+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	QSNaN2 NV	QNaN2	+ ∞	+ ∞	QNaN NV(4)	+ ∞
– ∞	– ∞	– ∞	– ∞	– ∞	QSNaN2 NV	QNaN2	– ∞	– ∞	–∞	QNaN NV(4)

Note 1: output possible:
rs1 ≠ –rs2: +/– Norm, UNFNSH, +/– Norm & NX, +/– O_D & UF & NX.
rs1 = rs2: + 0 or – 0 depending on rounding mode (see (5)).
Note 2: output possible: – Norm, – Norm & NX, R_{IEEE} & OF & NX, – Norm & NX.
Note 3: output possible: + Norm, + Norm & NX, R_{IEEE} & OF & NX, + Norm & NX.
Note 4: QNaN delivered at output = 7ff..f
Note 5: returns + 0 in rounding modes RN, RZ, and RP; returns –0 in RM.

FsTOi, FdTOi (NS):

rs2	result
+ Norm	(1)
– Norm	(2)
+ DeNorm	+ 0 NX
– DeNorm	– 0 NX
SNaN2	NV (3)
QNaN2	NV (3)
+ 0	+ 0
– 0	– 0
+ Inf	NV (4)
– Inf	NV (5)

Note 1: outputs possible: +Inum, NV if output overflows (see (3)), +Inum & NX.
Note 2: outputs possible: – Inum, NV if output overflows (see (3)), – Inum & NX.
Note 3: If sign bit is 0, deliver 7ff..f; if sign bit is 1, deliver 800..0
Note 4: Deliver 7ff..f at output.
Note 5: Deliver 800..0 at output.

FiTOs, FiTOd (NS):

rs2	result
+ 0	+ 0
+ Inum	(1)
– Inum	(2)

Note 1: outputs possible:
FITOS: + Norm, +Norm & NX.
FITOD: + Norm

Note 2: outputs possible:
FITOS: – Norm, – Norm & NX.
FITOD: – Norm.

FSTOD (NS):			FDTOS (NS):	
rs2	result		rs2	result
+ Norm	+ Norm		+ Norm	(1)
– Norm	– Norm		– Norm	(2)
+ DeNorm	+ 0 NX		+ DeNorm	+ 0 NX
– DeNorm	– 0 NX		– DeNorm	– 0 NX
SNaN2	QSNaN2 NV		SNaN2	QSNaN2 NV
QNaN2	QQNaN2		QNaN2	QQNaN2
+ 0	+ 0		+ 0	+ 0
– 0	– 0		– 0	– 0
+ ∞	+ ∞		+ ∞	+ ∞
– ∞	– ∞		– ∞	– ∞

Note 1: outputs possible: + Norm, + Norm & NX, R_{IEEE} & OF & NX, $+0_D$ & UF & NX.
Note 2: outputs possible: – Norm, – Norm & NX, R_{IEEE} & OF & NX, -0_D & UF & NX.

FMULs, FMULd (NS):

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	+ 0 NX	– 0 NX	QSNaN2 NV	QNaN2	+ 0	– 0	+ ∞	– ∞
– Norm	(2)	(1)	– 0 NX	+ 0 NX	QSNaN2 NV	QNaN2	– 0	+ 0	– ∞	+ ∞
+DeNorm	+ 0 NX	– 0 NX	+ 0 NX	– 0 NX	QSNaN2 NV	QNaN2	+ 0	– 0	+ ∞	– ∞
– DeNorm	– 0 NX	+ 0 NX	– 0 NX	+ 0 NX	QSNaN2 NV	QNaN2	– 0	+ 0	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QNaN2	+ 0	– 0	QNaN NV (3)	QNaN NV (3)
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QNaN2	– 0	+0	QNaN NV(3)	QNaN NV (3)
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	+ ∞	– ∞
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	– ∞	+ ∞

Note 1: outputs possible: + Norm, + Norm & NX, 0_D & UF & NX, R_{IEEE} & OF & NX.
Note 2: outputs possible: – Norm, – Norm & NX, -0_D & UF & NX, R_{IEEE} & OF & NX.
Note 3: QNaN delivered at output = 7ff..f

FsMULd (NS):

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	+ Norm	– Norm	+ 0 NX	– 0 NX	QSNaN2 NV	QQNaN2	+ 0	– 0	+ ∞	– ∞
– Norm	– Norm	+ Norm	– 0 NX	+ 0 NX	QSNaN2 NV	QQNaN2	– 0	+ 0	– ∞	+ ∞
+DeNorm	+ 0 NX	– 0 NX	+ 0 NX	– 0 NX	QSNaN2 NV	QQNaN2	+ 0	– 0	+ ∞	– ∞
– DeNorm	– 0 NX	+ 0 NX	– 0 NX	+ 0 NX	QSNaN2 NV	QQNaN2	– 0	+ 0	– ∞	+ ∞
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QQNaN1	QQNaN1	QQNaN1	QQNaN1	QSNaN2 NV	QQNaN2	QQNaN1	QQNaN1	QQNaN1	QQNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QQNaN2	+ 0	– 0	QNaN NV (1)	QNaN NV (1)
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QQNaN2	– 0	+ 0	QNaN NV (1)	QNaN NV (1)
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QQNaN2	QNaN NV (1)	QNaN NV (1)	+ ∞	– ∞
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QQNaN2	QNaN NV (1)	QNaN NV (1)	– ∞	+ ∞

Note 1: QNaN delivered at output = 7ff..f

FDIVs, FDIVd (NS):

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	(2)	+ ∞ DZ, NX	– ∞ DZ, NX	QSNaN2 NV	QNaN2	+ ∞ DZ	– ∞ DZ	+ 0	– 0
– Norm	(2)	(1)	– ∞ DZ, NX	+ ∞ DZ, NX	QSNaN2 NV	QNaN2	– ∞ DZ	+ ∞ DZ	– 0	+ 0
+DeNorm	+ 0 NX	– 0 NX	QNaN NV, NX (3)	QNaN NV, NX (3)	QSNaN2 NV	QNaN2	+ ∞ DZ	– ∞ DZ	+ 0	– 0
– DeNorm	– 0 NX	+ 0 NX	QNaN NV, NX (3)	QNaN NV, NX (3)	QSNaN2 NV	QNaN2	– ∞ DZ	+ ∞ DZ	– 0	+ 0
SNaN1	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN2 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV	QSNaN1 NV
QNaN1	QNaN1	QNaN1	QNaN1	QNaN1	QSNaN2 NV	QNaN2	QNaN1	QNaN1	QNaN1	QNaN1
+ 0	+ 0	– 0	+ 0	– 0	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	+ 0	– 0
– 0	– 0	+ 0	– 0	+ 0	QSNaN2 NV	QNaN2	QNaN NV (3)	QNaN NV (3)	– 0	+ 0
+ ∞	+ ∞	– ∞	+ ∞	– ∞	QSNaN2 NV	QNaN2	+ ∞	– ∞	QNaN NV (3)	QNaN NV (3)
– ∞	– ∞	+ ∞	– ∞	+ ∞	QSNaN2 NV	QNaN2	– ∞	+ ∞	QNaN NV (3)	QNaN NV (3)

Note 1: outputs possible: + Norm, + Norm & NX, $+ 0_D$ & UF & NX, R_{IEEE} & OF & NX.
Note 2: outputs possible: – Norm, – Norm & NX, $+ 0_D$ & UF & NX, R_{IEEE} & OF & NX.
Note 3: QNaN delivered at output= 7ff..f

FCMPs, FCMPd (NS):

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	>	>	>	? NV	?	>	>	<	>
– Norm	<	(1)	<	<	? NV	?	<	<	<	>
+DeNorm	<	>	(1)	>	? NV	?	>	>	<	>
– DeNorm	<	>	<	(1)	? NV	?	<	<	<	>
SNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
QNaN1	?	?	?	?	? NV	?	?	?	?	?
+ 0	<	>	<	>	? NV	?	=	=	<	>
– 0	<	>	<	>	? NV	?	=	=	<	>
+ ∞	>	>	>	>	? NV	?	>	>	=	>
– ∞	<	<	<	<	? NV	?	<	<	<	=

Note 1: output possible: >, < , =

FCMPEs, FCMPEd (NS):

rs1/rs2	+ Norm	– Norm	+DeNorm	–DeNorm	SNaN2	QNaN2	+ 0	– 0	+ ∞	– ∞
+ Norm	(1)	>	>	>	? NV	? NV	>	>	<	>
– Norm	<	(1)	<	<	? NV	? NV	<	<	<	>
+DeNorm	<	>	(1)	>	? NV	? NV	>	>	<	>
– DeNorm	<	>	<	(1)	? NV	? NV	<	<	<	>
SNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
QNaN1	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV	? NV
+ 0	<	>	<	>	? NV	? NV	=	=	<	>
– 0	<	>	<	>	? NV	? NV	=	=	<	>
+ ∞	>	>	>	>	? NV	? NV	>	>	=	>
– ∞	<	<	<	<	? NV	? NV	<	<	<	=

Note 1: output possible: >, < , =

3.9.9 Integer Unit-Floating-Point Unit Exception Flush Logic

When an integer instruction is executing and an exception occurs, it is possible for following fp instructions to have already entered the fp pre-queue. To avoid the problem of incorrectly "re- executing" fp instructions that follow the trapping integer unit instruction, logic has been added to the FPQ to track the program counter progress on the integer unit side and flush appropriate instructions from the FPQ.

3.9.10 RT620 Method for Handling Traps and Error Mode

3.9.10.1 Integer Unit

Table 3–11 describes how the processor affects critical fields and registers during reset traps, non-reset traps, and error mode.

If *tt* were overwritten on reset or error mode, the initial trap type might be lost and the preserved information would not correspond to the initial trap.

Since neither reset nor error mode update the CWP, TBR.tt, r[17], and r[18] registers, it is possible for the reset trap handler to locate and identify the initial trap that was being processed when error mode occurred.

Do not confuse reset due to power-on with reset due to error mode. The state of CWP, TBR.tt, r[17], and r[18] registers are undefined. The RT625 logs enough information for boot code to recognize whether it entered into reset routine due to power-on or error (i.e., double fault). When reset is asserted by the RT625, the RT620 treats reset identically whether the reset trap was due to power on from the module "power-on" logic or asserted in order to clear error mode.

Table 3–11. Integer Unit Actions Upon Reset, Trap, and Error Mode Events

On Reset	On Non-Reset Trap	On Error Mode
ET ← 0	ET ← 0	ET ← n/c*
PS ← n/c	PS ← S	PS ← n/c
S ← 1	S ← 1	S ← n/c
CWP ← n/c	CWP ← (CWP –1)mod	CWP ← n/c
r[17] ← n/c	r[17] ← exc PC	r[17] ← n/c
r[18] ← n/c	r[18] ← exc nPC	r[18] ← n/c
PC ← 0	PC ← TBR	PC ← n/c
nPC ← 4	nPC ← TBR + 4	nPC ← n/c
TBR.tt ← n/c	TBR.tt ← trap type	TBR.tt ← n/c**

* indicates the ET bit must be zero in order to enter error mode.

** indicates RETT with and exception can cause tt to be updated when error mode is entered. In the case of RETT, tt can be set to privilege violation, misaligned address, or window underflow traps when ET = 0 and a corresponding exception occurred. No other exceptions can cause tt to be updated when ET = 0 and a corresponding exception occurred. No other exceptions can cause tt to be updated when error mode occurs.

3.9.10.2 Floating-Point Unit

Table 3–12 describes how the processor affects critical fields and registers in the floating-point unit during reset traps, non-reset traps, and error mode

If the integer unit enters error mode with fp instructions executing in the queue, the floating-point unit will complete all pending instructions in the queue and signal exceptions as appropriate. The integer unit will ignore all floating-point unit signals in error mode state.

Table 3–12. Floating-Point Unit Actions Upon Reset, Trap, And Error Mode Events

On Reset	On Non-Reset Trap	On Error Mode
FSR.qne ← 0	FSR.qne ← unchanged	FSR.qne ← unchanged
rest of FSR ← undefined	rest of FSR ← unchanged	rest of FSR ← unchanged
FREGS ← undefined	FREGS ← unchanged	FREGS ← unchanged
Queue Entries ← invalid	Queue entries may be flushed if the trap is not an fp_exception trap.	Queue entries ← unchanged

RT625 hyperSPARC Cache Controller, Memory Management, and Tag Unit

The hyperSPARC RT625 Cache Controller, Memory Management, and Tag Unit (CMTU) consists of a Memory Management unit (MMU), a cache controller with cache tag memory, and support for the SPARC Reference MBus interface. The CMTU is designed as an integral part of the hyperSPARC family to provide a high-performance solution for cache, virtual memory, and multiprocessing support. Features of the RT625 include:

- 64-entry translation lookaside buffer (TLB)
- 32-byte read buffer and 64-byte write buffer
- Uniprocessor and multiprocessor system support
- 128-Kbyte and 256-Kbyte cache size support
- Support for memory systems with reflective memory controllers
- MBus Level 2 cache coherency protocol
- Supports the SPARC MBus reference standard interface

The RT625 is designed as part of a tightly coupled hyperSPARC CPU system which includes the RT625 CMTU, RT620 CPU, and two or four RT627 Cache Data Units (CDUs).

4.1 RT625 hyperSPARC CMTU

The RT625 CMTU combines two major functions: memory management and cache control. In addition, the RT625 provides the asynchronous interface between the system bus (MBus) and the rest of the hyper-SPARC CPU.

The MMU portion of the RT625 provides translation from a 32-bit virtual address range (4 Gigabytes) to a 36-bit physical address (64 Gigabytes), as provided in the SPARC Reference MMU specification. Virtual address translation is further extended with the use of a context register, which is used to identify up to 4096 contexts or tasks. The TLB entries contain context numbers to identify tasks or processes. This minimizes unnecessary TLB entry replacement during task switching.

The MMU features a 64-entry translation lookaside buffer. The TLB acts as a cache for address mapping entries used by the MMU to map a virtual address to a physical address. These mapping entries, referred to as page table entries or PTEs, allow one of four levels of address mapping. A PTE can be defined as the address mapping for a single 4-Kbyte page, a 256-Kbyte region, a 16-Mbyte region, or a 4-Gbyte region. The TLB entries are lockable, allowing important TLB entries to be excluded from replacement.

The MMU performs its address translation task by comparing a virtual address supplied by the hyperSPARC RT620 Central Processing Unit (CPU) through the Intra-Module Bus to the address tags in the TLB entries. If the virtual address and the value of the context register match a TLB entry, a TLB "hit" occurs. When this occurs, the physical address stored in the TLB is used to translate the virtual address to a physical address. The Access Type (read/write of data or instruction) and privilege level (user/supervisor) are checked during translation. If a TLB hit occurs but access-level protection is violated, the MMU signals an exception and the operation ends.

If the virtual address or context does not match any valid TLB entry, a TLB "miss" occurs. This causes a table walk to be performed by the MMU. The table walk is a search performed by the MMU through the address translation tables stored in main memory. The MMU searches through several levels of tables for the PTE corresponding to the virtual address. Upon finding the PTE, the MMU translates the address and selects a TLB entry for replacement, where it then stores the PTE.

Two sizes of cache are supported: 128-Kbyte and 256-Kbyte. The cache is "virtually indexed" and "physically tagged." The term "virtually indexed" refers to the direct addressing of the cache line by the RT620 CPU with the Intra-Module Address Bus (IMA <31:0>). The 128-Kbyte cache is organized into 4096 lines of 32 bytes each. IMA <16:5> select the cache line, and IMA < 4:3 > select the 64-bit word of the cache line, as illustrated in *Figure 4–1*. The 256-Kbyte cache is organized into 4096 lines with two sub-blocks, each sub-block being 32 bytes (cache sub-block approach). Address bits IMA < 17:6 > select the cache line, address bit IMA < 5 > selects the sub-block and address bits IMA < 4:3 > select the 64-bit word of the cache line, as illustrated in *Figure 4–2*.

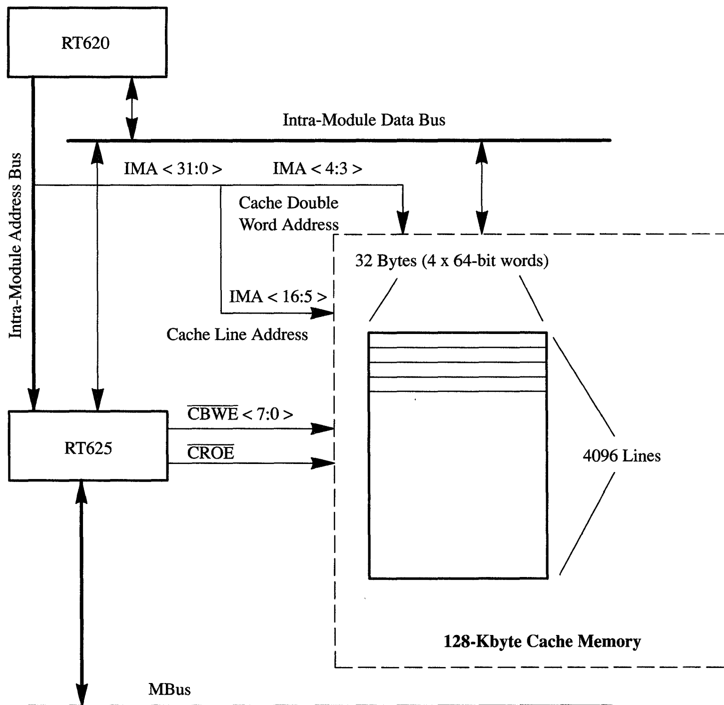


Figure 4–1. 128-Kbyte Cache Memory Sub-System

The cache tag entries reflect the physical address, hence the cache is said to be "physically tagged." The RT625 provides access control for the cache by checking the physical address (translated through the TLB)

against the cache tags. If the physical address matches the cache tag for the cache line addressed, a *cache hit* occurs and the access is enabled. If the physical address does not match the cache tag for the cache line, a *cache miss* occurs and the cache controller accesses main memory for the required data. The RT625's cache directory can be accessed from both the processor and MBus. Due to the inclusion of 8 Kbytes of on-chip instruction cache in the RT620 and an efficient arbitration mechanism to access cache tags from both the processor and MBus, the performance degradation due to a single cache directory compared with a dual cache directory is insignificant. The RT625 supports the MBus Level 2 cache coherency protocol, which is modeled after the acclaimed IEEE Futurebus.

The RT625 cache controller supports two modes of caching: write-through with no write allocate and copy-back with write allocate. Write-through mode is a simpler style of cache management that causes write accesses to the cache to be written through to main memory upon each write access. The advantage of this method is that the cache always remains coherent with main memory. Its disadvantage is that each write to the cache is echoed to main memory, which increases traffic on the system bus. Another disadvantage to write-through is that the processor is delayed by the time required to arbitrate the system bus and write the data to main memory. However, in the case of the RT625, this disadvantage is significantly offset by the inclusion of write buffers.

Copy-back cache mode causes write accesses to be written to the cache only. This causes the cache line to become modified. Modified cache lines are automatically written back to main memory only when the cache line is no longer needed. Copy-back mode is a more complex mode of cache management, but provides substantial system performance improvements over write-through due to decreased traffic on the system bus.

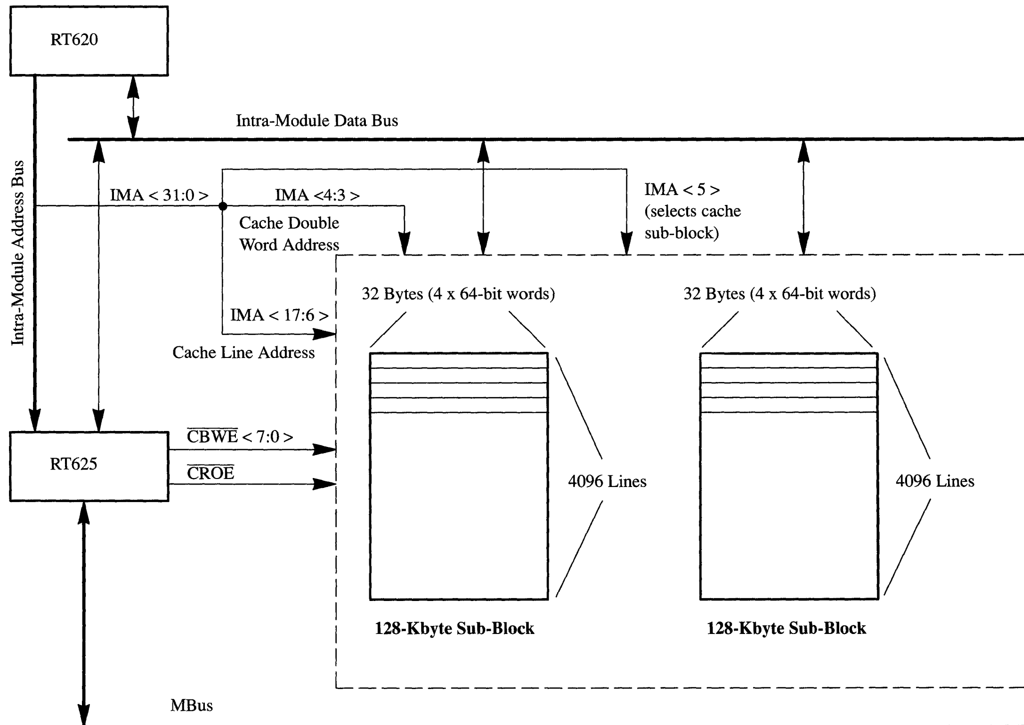


Figure 4–2. 256-Kbyte Cache Memory Sub-System

A 64-byte write buffer and a 32-byte read buffer are provided in the RT625 to fully buffer the transfer of a cache line. This feature allows the RT625 to simultaneously read a cache line from main memory as it is flushing a modified cache line from the cache. The write buffer can store up to eight doubleword accesses and avoids stalling the RT620 on writes to main memory by storing the write data until the physical bus becomes available. The write buffer writes the data to memory as a background task.

The RT625 supports the SPARC MBus reference standard interface. The MBus is a peer-level, high-speed, 64-bit, multiplexed address and data bus that supports a full peer-level protocol (i.e., multiple bus masters). The RT625 MBus supports data transfers in transaction sizes of 1, 2, 4, 8, or 32 bytes. These data transfers are performed in either burst or non-burst mode, depending upon the size. Data transactions larger than eight bytes (one doubleword) are transferred in burst mode, which consists of an address phase followed by four data phases. Non-burst transactions consist of an address phase followed by one data phase, and are used for data transactions of eight or fewer bytes. Bus mastership is granted and controlled by an external bus arbiter. The bus arbiter sets bus priorities, and grants access to a bus master. Additional information on the MBus can be found in the SPARC MBus Interface Specification.

The RT625 provides support for memory systems with reflective memory controllers. A memory system with reflective memory control can recognize a cache-to-cache data transaction and automatically update itself without delaying the system.

4.2 RT625 Memory Management Unit

This section describes the SPARC Reference MMU as implemented in the RT625.

The MMU provides virtual to physical address translation with the use of an on-chip translation lookaside buffer (TLB). The TLB is in reality a full address translation cache for address translation entries stored from tables in main memory. These entries, referred to as page table entries or PTEs, contain the mapping information used by the MMU to translate the virtual addresses. Addresses presented to the MMU for translation are compared against the set of PTEs stored in the TLB. All entries in the TLB are simultaneously accessed through the use of content addressable memory (CAM) technology. If a match for the virtual address and context is found in a valid TLB entry and the access protection is not violated, a TLB hit occurs and the address is translated. A virtual address and context that matches a valid TLB entry but violates the memory access protections will cause the RT625 to generate a memory exception to the RT620. If the TLB entries do not match the address and context, or the TLB entry is invalid, then a TLB miss occurs. The MMU responds to the TLB miss by initiating a table walk to find the correct PTE stored in main memory for the virtual address.

The MMU uses a tree-structured table walk algorithm to find page table entries not found in the TLB. The table walk is a search through a series of up to four tables in main memory for the PTE corresponding to a virtual address. These tables are: the context table, the Level 1 table, the Level 2 table and the Level 3 table. The table walk uses the context table pointer register as a base register and the context number as an offset to point to an entry in the context table. At any address, the MMU finds either a PTE, which terminates its search, or a page table pointer (PTP). A PTP is a pointer used in conjunction with a field in the virtual address to select an entry in the next level of tables. The table walk continues searching through levels of tables as long as PTPs are found pointing to the next table. The table walk terminates if a PTE is found at any level; an exception is generated if a PTE is not found after accessing the Level 3 table. An exception is also generated if the table walk finds an invalid or reserved entry in the page tables. Upon finding the PTE, the RT625 stores it in an available TLB entry and translates the corresponding virtual address. The table walk processing is implemented in the RT625 hardware. It is self-initiated, and is transparent to the user.

4.2.1 Translation Lookaside Buffer (TLB)

The RT625 uses a 64-entry fully associative TLB for address translation. The TLB consists of two sections: a virtual section and a physical section, as shown in *Figure 4–3*. The virtual section is compared against the

virtual address and the contents of the context register. A content addressable memory (CAM) is used as the virtual section of the TLB. The CAM provides simultaneous comparison of all 64 TLB entries with the current virtual address and context. The physical section of the TLB is a RAM array, and its entries are addressed by a valid compare output from a CAM entry. If a CAM entry matches the virtual address and context, the corresponding RAM entry in the TLB provides the physical address for use by the RT625.

The virtual section of a TLB entry consists of 20 bits of virtual address (VA < 31:12 >) and a 12-bit context number (CXN <11:0>). The physical section of a TLB entry consists of a 24-bit physical page number (PPN < 35:12 >), a cacheable bit (C), a modified bit (M), a three-bit field for page access-level protection (ACC < 2:0 >), a two-bit short translation field (ST < 1:0 >), and one valid bit (V).

As described by the SPARC Reference MMU specification, bits 31 through 12 of the virtual address are translated to an expanded physical address using bits 35 through 12. The translation of these bits depends upon the ST field of the TLB entry (or PTE) and the MMU operation mode (refer to *Section 4.3*). Bits 11 through 0 of the virtual address are not translated, and are defined as the page offset.

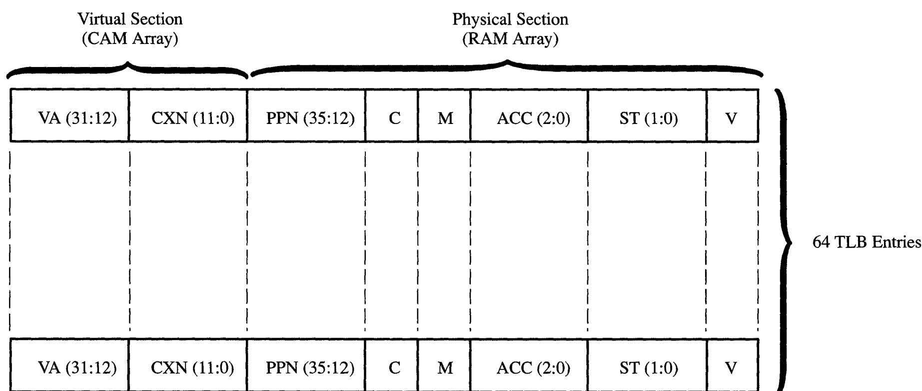


Figure 4–3. Translation Lookaside Buffer (TLB)

Table 4–1. Short Translation Bits ST(1:0)

ST1	ST0	Address Mapping
0	0	4-Kbyte (page size)
0	1	256-Kbyte
1	0	16-Mbyte
1	1	4-Gbyte

A TLB entry (PTE) can be defined to map a virtual address into one of four sizes of addressing regions using the ST field. The four sizes of addressing regions are: 4-Kbyte, 256-Kbyte, 16-Mbyte, or 4-Gbyte. *Table 4–1* illustrates the values assigned to the ST field.

The value of the short translation bits affects both the addresses generated using the TLB entry and the virtual addresses allowed to match with the TLB entry. The virtual address supplied by the RT620 is divided into four fields: index 1, index 2, index 3 and page offset, as illustrated in *Figure 4–4*. For ST = (1,1) (4-Gbyte addressing range), only the context register is used to match a TLB entry. Setting ST = (1,1) essentially causes the CAM array to ignore the index 1, 2, and 3 fields of the virtual address. Consequently, the address generated using the TLB entry only supplies the upper four bits of the 36-bit physical address. Index 1, 2, and 3 fields, along with the page offset, are passed along to the physical address unchanged.

The three remaining values of the ST field "turn on" comparison of the three index fields. The index fields that are required to match a TLB entry also become the fields that are replaced by the TLB entry during virtual to physical translation. Setting ST = (1,0) (16-Mbyte addressing region) requires the TLB to match the context and index 1 fields of the virtual address to the TLB entry. The TLB entry with ST = (1,0) will supply the upper four address bits and replace the index 1 field of the virtual address with a physical address field. The index 2, 3, and page offset fields are passed along to the physical address from the virtual address. Setting ST = (0,1) and (0,0) adds index 2 and index 3 fields to the comparison, respectively. Setting ST = (0,0) causes the TLB to require matching of the context, index 1, 2, and 3, and will replace all but the page offset when translating the virtual address.

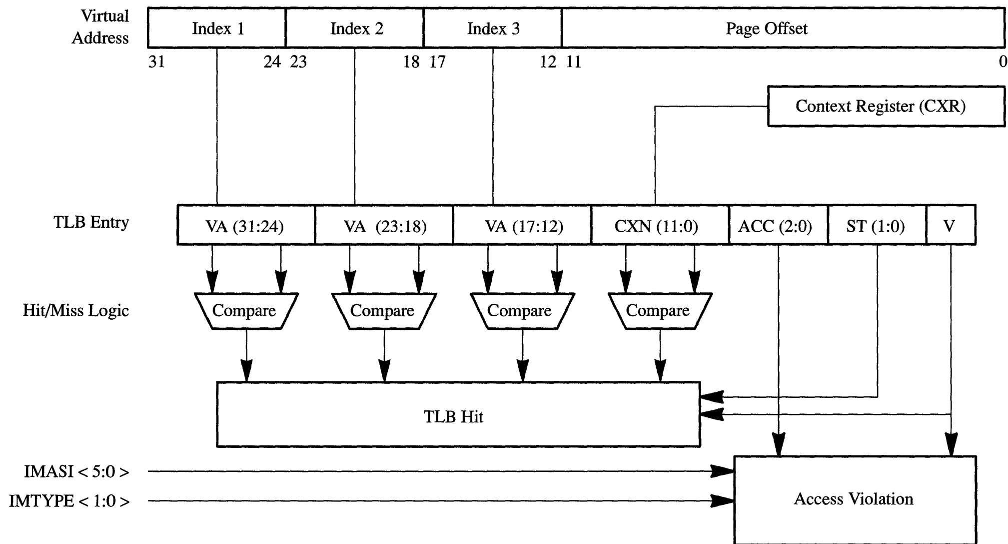


Figure 4–4. Address Comparison

Physical addresses are generated using the contents of the PPN field of the TLB entry. The portion of the PPN field used to map the virtual address to a physical address is dependent upon the ST(1:0) bit field, as described above. If a 4-Kbyte linear addressing range is specified by the ST(1:0) bits, then the entire 24-bit field is used as the upper 24 bits of the physical address. When a 256-Kbyte linear addressing range is specified, the upper 18 bits of the PPN field < 35:18 > are used in the physical address. The remaining bits of the physical address are supplied from the virtual address. The upper 12 bits of the PPN field < 35:24 > are used for a 16-Mbyte addressing region. If a 4-Gbyte region is selected, only the upper four bits of the PPN field < 35:32 > are used in the address translation. The page offset field of the virtual address is always used as the lower twelve bits of the physical address.

The cacheable bit (C) indicates whether the memory addressed by the TLB entry is cacheable or not. If the MMU is enabled, the value of the C bit is output on the MC pin (MAD) of the MBus during the address phase of a transaction. The MBus is described in the SPARC MBus interface specification.

The modified bit (M) in the TLB is set when the RT620 modifies the memory page. This bit may be checked by an operating system to determine the modified status of a memory area.

Table 4–2. Access-Level Protection Bits-ACC < 2:0 >

ACC	User Access	Supervisor Access
0	Read Only	Read Only
1	Read/Write	Read/Write
2	Read/Execute	Read/Execute
3	Read/Write/Execute	Read/Write/Execute
4	Execute Only	Execute Only
5	Read Only	Read/Write
6	No Access	Read/Execute
7	No Access	Read/Write/Execute

The access-level protection (ACC) bits are described in *Table 4–2*. The ACC bits define the access-level protection for the addressing region controlled by the TLB entry. Access-level protection is checked during a TLB access. If a TLB hit occurs but access-level protection is violated, the MMU generates a synchronous fault and the operation terminates (see *Section 4.10, Synchronous Faults*).

The valid bit (V) reports the valid status of the TLB entry. These bits are cleared upon power-on reset ($\overline{\text{RSTIN}}$ asserted) to invalidate the TLB entries. These bits are also cleared on a TLB entry invalidation.

Programmer's Note: When loading the TLB entries under software control (i.e., TLB entries loaded by the central processing unit with ASI = 6), care must be taken to ensure that multiple TLB entries cannot map to the same virtual address. This may inadvertently occur when combining TLB entries that map different sizes of addressing regions. For example, a 4-Kbyte region described by a TLB entry could be included in a TLB entry for a 16-Mbyte region. Violation of this restriction will result in an invalid output from the TLB. It is recommended that when writing to the TLB using ASI = 6, the CAM should be written first followed by write to the RAM. Note that this case cannot happen when the TLB entries are automatically loaded by the RT625 during a table walk, as the TLB is checked for a "hit" first.

4.2.1.1 TLB Look-up

A virtual address to be translated by the RT625 is compared against each entry in the TLB as shown in *Figure 4–4*. If a TLB hit (match) occurs and access-level requirements are satisfied, then the TLB outputs the physical address and the cacheable bit. This physical address is output by the RT625 onto the MBus if the cache has been disabled or if the page is non-cacheable. If the cache controller is enabled and a cache miss occurs, the physical address of the cache miss is used to access the new cache line in main memory for cache line replacement.

The short translation bits specify a linear address mapping range of 4-Kbyte, 256-Kbyte, 16-Mbyte, or 4-Gbyte for each TLB entry. The short translation bits also determine the index fields of the virtual address that are matched with the TLB entry to determine a TLB hit. For a TLB entry with a linear address range of 4-Kbyte, index fields 1, 2, and 3 of the virtual address and the context register are compared against the TLB entry. A TLB entry with a 256-Kbyte linear addressing range requires a match of the context and of the index 1 and index 2 fields. A 16-Mbyte linear addressing range requires a match of the index 1 field and the context. The 4-Gbyte linear address mapping requires only a context match to produce a TLB hit.

If the modified (M) bit is not set in a TLB entry, write, Load-Store accesses, Block Fills, cache flush and the destination write part of Block Copy that match the TLB entry and meet all access-level requirements will cause a table walk. (see *Section 4.2.2, Table Walk*) The table walk sets the modified bit in the page table pointer entry for the memory region. This information is used by an operating system to ensure that modified regions of memory are stored in alternate memory media (typically a disk drive) before they are overwritten during memory page swap operations.

If there is a matched entry, but the access-level requirements are not satisfied, then a synchronous address fault exception is asserted. Context number matching is not required if the access-level field (ACC) is either 6 or 7 and the memory access is a supervisor mode access (ASI = 9,B H). This produces a means of mapping the kernel of an operating system into the same virtual address locations of every context.

The TLB ignores access-level checking during MMU probe and software cache flush operations.

4.2.1.2 TLB Entry Replacement and Locking

The RT625 supports a random replacement algorithm to replace a TLB entry during TLB miss processing; however, the RT625 will attempt to fill all invalid entries before selecting any valid entry for replacement. The random replacement algorithm is implemented by using a counter to point to one of the 64 TLB entries. A 6-bit replacement counter (RC) is used to point to the next TLB entry to be replaced as shown in *Figure 4–5*. Upon encountering a TLB miss, the RT625 uses the counter value to address a TLB entry to be replaced. The hardware automatically replaces an entry pointed to by the replacement counter (RC) during TLB miss processing.

Locking of TLB entries is supported with a 6-bit initial replacement counter (IRC). The number of locked entries is specified by setting the value of the IRC. The value of the IRC is used as a counter preset for the replacement counter. Once the replacement counter (RC) reaches the maximum value, it wraps to the IRC value. Upon power-on reset ($\overline{RSTIN}$ asserted), both the IRC and RC are initialized to zero.

Locked TLB entries can be changed (read/write) only through the alternate space Load/Store instructions with ASI = 6 (see *Section 4.7., Diagnostics Support*) These locked entries will not participate in the random replacement algorithm during TLB miss processing. The IRC should be initialized to the number of lockable entries by writing to the TLB replacement control register (TRCR).

Programming Note: When changing the IRC, the RC should also be written with the same value. This ensures that the RC is always pointing to the replacement area of the TLB.

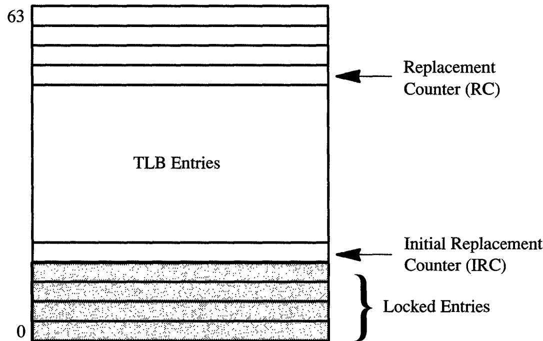


Figure 4–5. TLB Replacement and Locking

4.2.1.3 TLB Entries (TLBs)

Both the virtual and physical sections of each TLB entry can be accessed (read/write) through single Load or Store instructions. Software has the option to write and to lock high-usage or high-priority TLB entries to optimize system response time (Refer to *Section 4.7.1, MMU TLB Entries* for more details.)

4.2.2 Table Walk

The RT625 supports tree-structured, 4-level table walk processing (including the context table level) as shown in *Figure 4–6*. All of the virtual to physical address mapping tables are located in physical memory.

These tables are accessed in the case of a TLB miss, a write, Load-Store operation, Block Fills, cache flush or destination write part of Block Copy operation with a cleared M (modified) bit in the TLB entry.

Upon starting a table walk, the RT625 walks through a series of tables to find a page table entry (PTE). The page table entry contains the physical page number, the access-level permission, cacheable, modified, and referenced bits for the address generating the table walk. (Refer to *Section 4.2.4* for information on PTEs.) A table walk caused by a TLB miss causes the RT625 to update an available TLB entry with the new PTE. A table walk forced by a write or Load-Store operation on an unmodified memory region causes the RT625 to set the modified bit in the page table entry and in the TLB entry.

The table walk begins with an access to the context table. The RT625 uses the context table pointer register (CTPR) as a base register to point to the beginning of the context table. The context register (CXR) is used as an index register to point to the table entry. The upper 22 bits of the CTPR are concatenated with the twelve bits of the CXR to provide a 36-bit address. The lowest two bits of all addresses pointing to a page table entry or pointer are always forced to zero.

If a PTE is found at the context table level, the table walk terminates. The PTE is stored in the TLB and, if necessary, the modified bits and/or the referenced bits are updated. (Refer to *Section 4.2.4* for information on page table entries and the modified and reference bits.) If a page table entry is not found, then a Page Table Pointer (PTP) must be located at the address pointed to in the context table. (See *Sections 4.2.3* and *4.2.4* for more information on PTPs and PTEs.) The page table pointer is used as the base address for the next table.

If a PTE is not found, the table walk continues by accessing the Level 1 table using the PTP as a base address and the index 1 field from the virtual address as an index pointer. The index 1 field (virtual address (31:24)) is used to select an entry in the Level 1 table. If a page table entry is not found at this location, a page table pointer stored at this entry is used as the base address for the Level 2 table. The index 2 field (virtual address < 23:18 >) is used to select an entry in the Level 2 table. If the entry in the Level 2 table is not a page table entry, it is used as the base address for the Level 3 table. The index 3 field (virtual address < 17:12 >) is used to select an entry in the Level 3 table, which must be a page table entry.

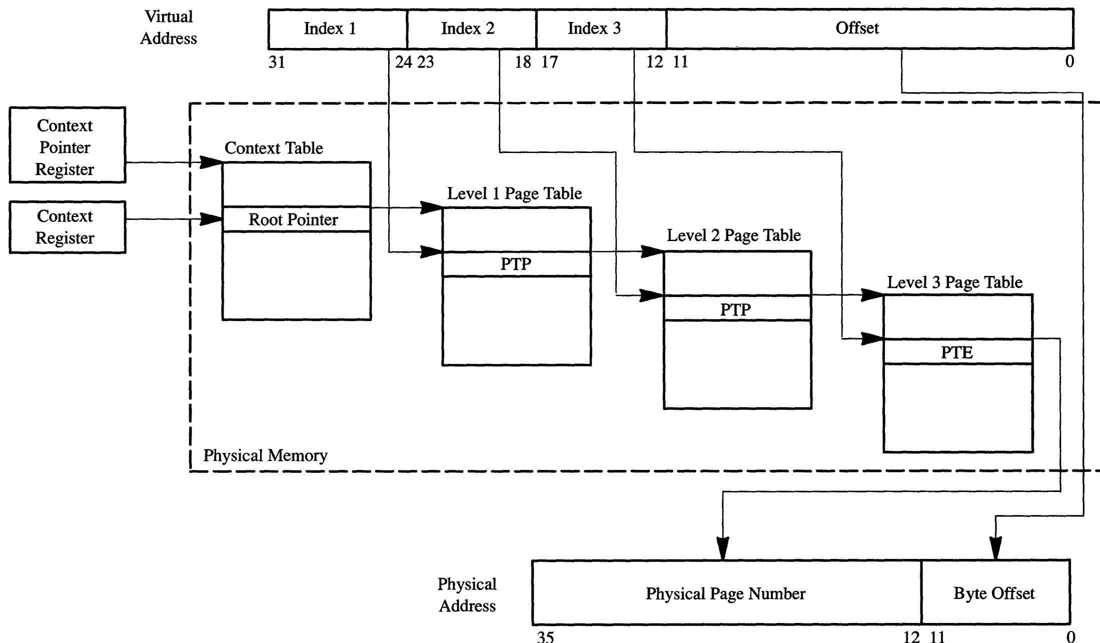


Figure 4–6. Four-Level Table Walk (4-Kbyte Addressing)

If a page table entry is not found after the Level 3 table access, a synchronous fault exception is asserted. A synchronous fault exception is also generated if an invalid or reserved entry is found at any level of the table walk. The table walk terminates immediately when an exception is generated.

The level at which the table walk terminates is related to the size of addressing region associated with the entry. A table walk that finds its page table entry in the context table corresponds to an addressing region of 4 Gbytes. Each level deeper into the table walk corresponds to a smaller size of address mapping. A PTE for a 16-Mbyte addressing region will be found in a Level 1 table. A 256-Kbyte PTE will be found in a Level 2 table. Only an addressing region of 4-Kbyte will require a table walk of four levels to find the correct page table entry).

An example of a table walk for a 256-Kbyte linear address space is shown in *Figure 4–7*. The value of the short translation bits are related to the level at which the table walk terminates. The short translation bits decrease from (1,1) for a table walk with a context table PTE to (0,0) for a table walk with a Level 3 table PTE. (refer to *Table 4–1*.)

Each table walk access is performed as a non-burst transaction on the MBus. The MBus busy ($\overline{\text{MBB}}$) signal is asserted from the beginning of the table walk to the end of the table walk process. This locks the MBus and prevents another bus master from gaining the bus until the table walk is complete. The MLOCK bit in the address phase of the MBus transaction will be set indicating a locked transaction. During these transactions, the C bit in the SCR register is output on the MC signal of the MBus. There will be write transactions during the table walk only if the referenced bit (R) and/or the modified bit (M) has to be set in the page tables.

If there is an invalid page table entry (ET = 0) at any level, an invalid address error exception occurs and the table walk terminates immediately. If an external Bus Error occurs or a reserved entry (ET = 3) is detected or a PTP entry is detected in Level 3, a translation error exception occurs, and the table walk terminates im-

mediately. If an access-level protection occurs, the table walk is terminated and a protection/privilege violation exception is asserted.

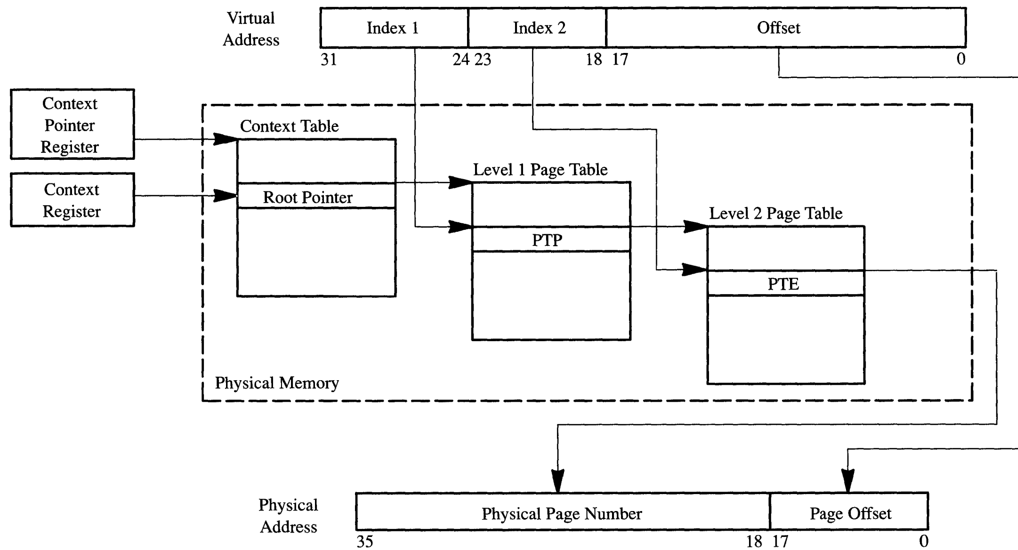


Figure 4–7. Three-Level Table Walk (256-Kbyte Addressing)

The referenced bit (R) and the modified bit (M) are set according to the Access Type. In order to record the exceptions in the synchronous fault status registers properly, the table walk hardware must indicate the fault type and the level at which the fault occurred (refer to *Section 4.10* for more details). For access-level checking during the table walk, Load-Store cycles are treated as write cycles. The table walk state diagram is shown in *Figure 4–11*. During MMU probe and software cache flush, the table walk controller ignores access-level checking.

4.2.3 Page Table Pointer (PTP)

A Page Table Pointer (PTP), as shown in *Figure 4–8*, may be found in the context, Level 1, or Level 2 tables. The PTP is used in conjunction with an index field of the virtual address to point to the next level of table in a table walk. The PTP found at the context level is called the root pointer. Bits 31 through 6 of the root pointer are output on bits 35 through 10 of the MBus (MAD < 35:10 >) and are concatenated with the eight bits of the index 1 field of the virtual address to access the entry in the first level page table. The lowest two bits of the address are equal to zero, as addressing is aligned on word boundaries.

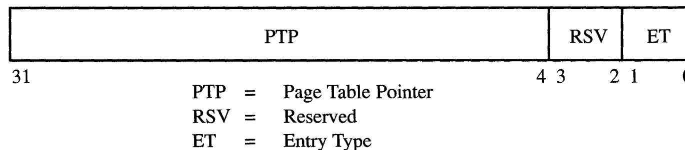


Figure 4–8. Page Table Pointer

Similarly, bits 31 through 4 of the PTP in Level 1 or Level 2 tables are output on bits 35 through 8 of the MBus (MAD < 35:8 >). The index 2 or index 3 fields are concatenated with the PTP to yield the address

of the next table entry. The ET field (see *Table 4–3*) describes the entry type: invalid, page table pointer, or page table entry.

Table 4–3. Page Table Entry Type

ET	Entry Type
0	Invalid
1	Page Table Pointer
2	Page Table Entry
3	Reserved

In order to reduce the penalty for a TLB miss, the root pointer from the context level table and four PTPs from the Level 2 table are cached in the PTP cache. The PTPs from the two most recent data and from the two most recent instruction misses using a four-level table walk are cached for later use. The TLB checks the PTP cache upon a TLB miss, and uses the cached PTP to access the Level 3 table if an entry matches the access. The PTP cache is discussed in more detail in *Section 4.2.5*.

4.2.4 Page Table Entry (PTE)

The Page Table Entry (PTE) is shown in *Figure 4–9* and may be found in the context, Level 1, Level 2 or Level 3 tables. The page table entry contains the address mapping information used by the MMU to translate a range of virtual addresses to physical addresses.

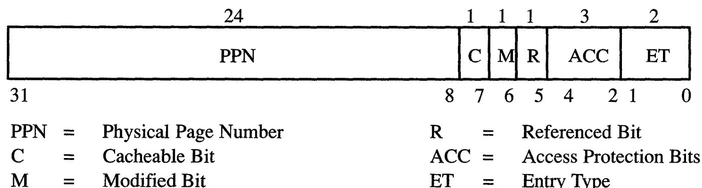


Figure 4–9. Page Table Entry Format

The level of the table in which the PTE is found is related to the addressing range associated with the PTE. A PTE found in the context table will map a 4-Gbyte addressing region. A Level 1 PTE will map a 16-Mbyte addressing region. A Level 2 PTE corresponds to a mapping region of 256 Kbytes. A Level 3 PTE maps a 4-Kbyte addressing region. The addressing region mapped to the PTE determines how many bits in the PPN field of the PTE are used to form the physical address. PTE < 31:28 > from a context level table PTE are output on bits 35 through 32 of the physical address bus (MAD < 35:32 >) to offer 4 Gbytes of linear address mapping. Similarly, PTE < 31:20 > from a Level 1 table PTE are asserted on bits 35 through 24, and provides 16 Mbytes of linear addressing. PTE < 31:14 > from a Level 2 table PTE are asserted on bits 35 through 18, and PTE < 31:8 > from a Level 3 table PTE are asserted on bits 35 through 12 to offer 256 Kbytes and 4 Kbytes of linear address mapping, respectively. The remainder of the PPN field not used for address translation is reserved. The remaining physical address bits not specified by the PPN field are supplied from the virtual address.

The ACC bits describe the access-level and privilege protection assigned to the PTE. These bits are described in *Table 4–2*. The referenced (R) bit is set in the PTE when the RT625 has read the value of the PTE in a table walk. The RT625 automatically sets this bit upon access of the PTE. The modified (M) bit is set upon a write, Load-Store access, Block Fills, cache flush or destination write part of Block Copy of a previously unmodified memory region. This information is commonly used by an operating system to flag regions of memory that must be written to mass storage before being replaced by another memory page.

The cacheable (C) bit indicates whether or not the memory region addressed by the PTE is allowed to be cached. It may be used, for example, to prevent caching of memory-mapped input/output devices.

The ET field, described in *Table 4–3*, is used by the RT625 to determine the type of table entry during a table walk. The ET field is set to 2 to indicate a PTE, and is set to 1 to indicate a PTP. If the RT625 encounters a table entry with ET=0 during a table walk, the RT625 generates an invalid address error. The RT625 generates a translation error if ET=3 (reserved) is encountered in a table entry during a table walk.

4.2.5 Page Table Pointer Cache (PTPC)

In order to reduce the penalty for a TLB miss, the RT625 supports a five-PTP entry page table pointer cache. The page table pointer cache (PTPC) caches the most recently used PTPs, as shown in *Figure 4–10*. The five entries are: the root pointer register (RPR), the two instruction access Level 2 PTPs (IPTP0, IPTP1), and the two data access Level 2 PTPs (DPTP0, DPTP1). The IPTP0 and DPTP0 registers are referenced by the index tag register (ITR0). The IPTP1 and DPTP1 registers are referenced by the index tag register (ITR1). These entries are cached during table walk processing for a TLB miss.

The root pointer for a context is cached in the RPR. The RPR remains valid until the context register (CXR) or the context table pointer register (CTPR) value is changed. The instruction access PTP registers contain the two latest Level 2 PTPs for instruction accesses. These PTPs are cached from the two most recent TLB misses requiring a three- or four-level table walk for instruction accesses. The data access PTP registers contain the two latest Level 2 PTPs for data accesses. These PTPs are cached from the two most recent three- or four-level table walk for data accesses. A LRU algorithm is used to decide which PTPs (either IPTP0 or IPTP1 for an instruction access, DPTP0 or DPTP1 for a data access) to replace when a four-level table walk occurs. Refer to *Section 4.5* for more information on these registers.

Index tag registers (ITR0, ITR1) are used to reference the IPTP and DPTP registers. The ITAG and DTAG fields of the index tag register are used by the RT625 to compare against an address generating a TLB miss. Once a Level 2 page table pointer is cached for an instruction or a data access, the same PTPs are used if the index 1 and index 2 fields of the virtual address match the index 1 and index 2 tag fields of the ITAG or DTAG. The IPTP and DPTP registers are updated only if a TLB miss occurs that does not match the ITAG or DTAG and also generates a table walk that accesses Level 3 of the page tables.

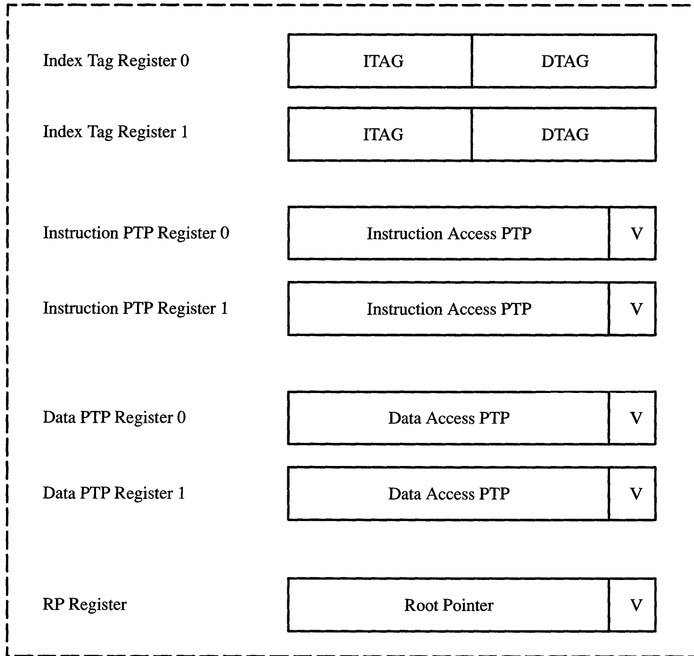


Figure 4–10. Page Table Pointer Cache

Once a root pointer is cached for a particular context, the same root pointer can be used as long as the context is not changed. If the table walk finds a context level or Level 1 or Level 2 entry PTE (i.e., is not a four-level table walk), then no caching of Level 2 pointers is performed.

Whenever the context is changed, the entire PTPC (all five entries) is invalidated. Upon power-on reset, all the PTPC entries are invalidated. When the context pointer register (CTPR) is written, the page table pointer cache is invalidated by clearing the V bits in the IPTPs, DPTPs, and RPR registers. Any TLB invalidate operation invalidates the IPTP and DPTP registers of the PTP Cache. The PTPC entries are also updated for MMU probe, software cache flush and Block Copy/Fill operations.

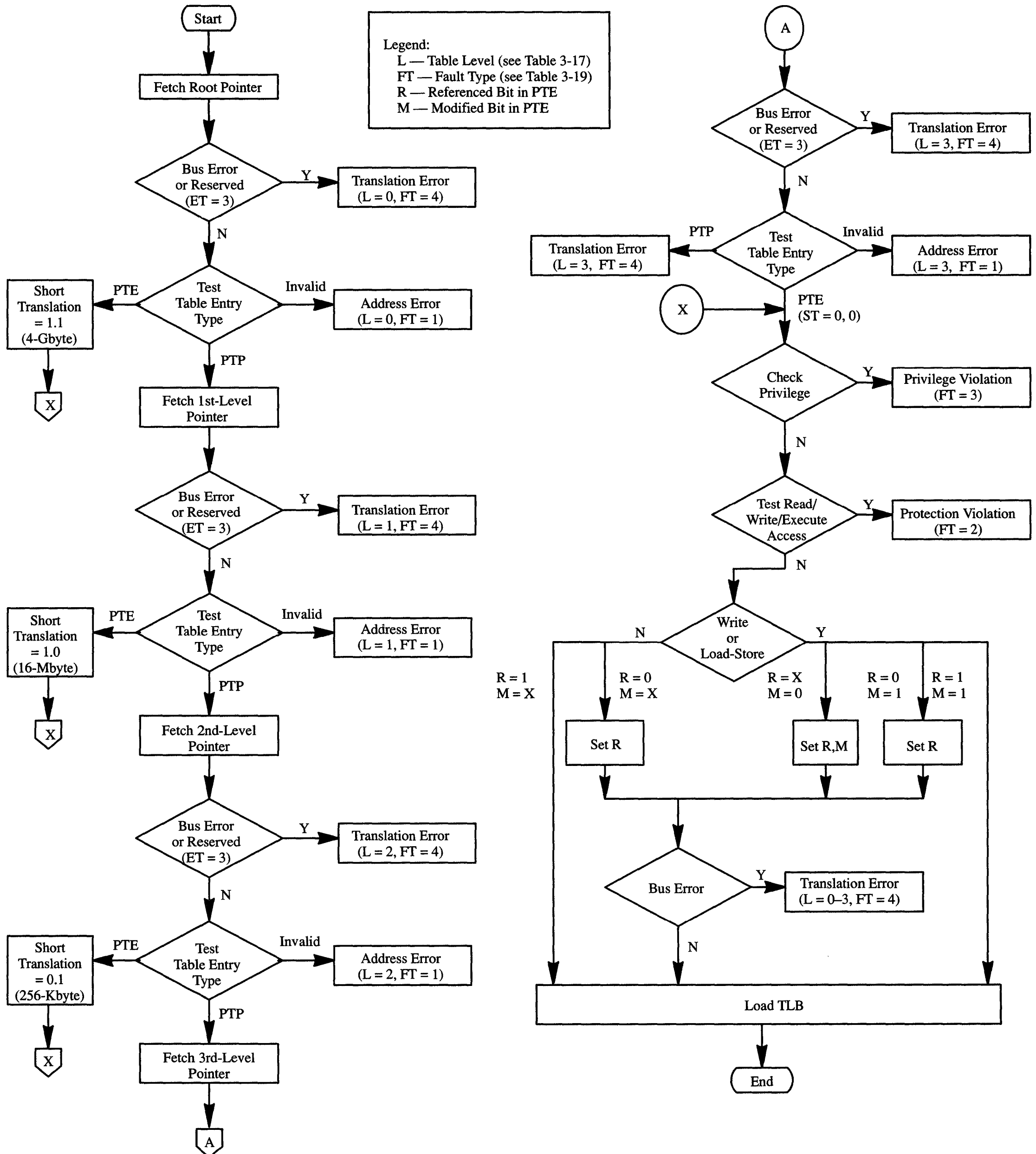


Figure 4–11. Table Walk Algorithm

4.3 RT625 MMU Operation Modes

This section describes the different modes of operation of the RT625, the conditions under which they occur, and what information is reflected on the pins. The operation mode for the MMU (and cache controller) is controlled by the system control register (SCR). Please refer to *Section 4.5.1* for further information on the SCR.

The following symbols are used throughout the chart:

Symbol	Meaning
MC(MAD(43))	MBus Cacheable Indicator signal (Refer to *Section 4.11, Pin Definitions*)
ASI	Address Space Identifier code for current access from RT620
SCR[C]	Cacheable bit of SCR
X	Not Defined or Don't Care
UN	Unassigned ASI
RES	Reserved ASI and ASI
PA	Physical Address
VA	Virtual Address
BM, ME, CE	Bits in System Control Register (SCR)
PTE[C]	Cacheable bit of page table entry

Table 4–4. MMU Operation Modes

MMU Operation Modes								
	Conditions				Results			
Mode	ASI	BM	ME	CE	Physical Addressing		Caching	MC
UN, RES	UN, RES	X	X	X	Ignore	Ignore	Ignore	N/A
By-pass	20-2F	X	X	X	PA < 35:32 > = ASI < 3:0 >	PA < 31:0 > = VA <31:0 >	Not Cached	0
Pass-Through	8, 9, A, B	0	0	X	PA < 35:32 > = 0	PA < 31:0 > = VA <31:0>	Not Cached	SCR [C]
Boot (Instr. access)	8, 9	1	X	X	PA < 35:28 >= FF	PA <27:0 > = VA <27:0>	Not Cached	SCR [C]
Boot (Data access)	A, B	1	0	X	PA < 35:32 > =0	PA < 31:0 > = VA <31:0 >	Not Cached	SCR [C]
Translation 1 (Data Access and Cache Disabled)	A, B	X	1	0	PA < 35:12 > = PTE < 31:8 >[1]	PA <11:0 > = VA < 11:0 >[1]	Not Cached	PTE [C]
Translation 2 (Data Access and Cache Enabled)	A, B	X	1	1	PA < 35:12 > = PTE < 31:8 >[1]	PA <11:0 > = VA < 11:0 >[1]	Cached if PTE[C] = 1	PTE [C]
Translation 3 (Instruction Access and Cache Disabled)	8, 9	0	1	0	PA < 35:12 > = PTE < 31:8 >[1]	PA <11:0 > = VA < 11:0 >[1]	Not Cached	PTE [C]
Translation 4 (Instruction Access and Cache Enabled)	8, 9	0	1	1	PA < 35:12 > = PTE < 31:8 >[1]	PA <11:0 > = VA < 11:0 >[1]	Cached if PTE[C] = 1	PTE [C]

[1]Concatenation field sizes vary depending upon the short translation (ST) bits to provide 4-Gbyte, 16-Mbyte, 256-Kbyte or 4-Kbyte of linear address mapping. Refer to *Section 4.2.1* for further details.

The MMU provides three types of operating modes: boot modes, direct-access modes, and translation modes. Two boot modes are defined for the MMU, one for data accesses, and one for instruction accesses. The boot modes force the upper eight bits of the physical address to FF H for instruction accesses. The upper four bits are forced to zero for data accesses.

The direct access modes allow the central processing unit to access the main memory without address translation by the MMU. These modes include: by-pass, and pass-through. The lower 32 bits of the physical address are supplied directly from the virtual address bus. This mode allows the central processing unit to access the boot mode memory (if supported in the system) without changing the state of the SCR.

Bypass mode allows complete access to the main memory space. The MMU is not enabled, and the lower four bits of the ASI are used as the upper bits of the physical address. The remaining 32 bits are supplied

directly from the virtual address bus. The state of the SCR does not have to be modified. This mode is mapped into the ASI space as ASI = 20 - 2F H.

Pass-through mode describes the RT625 operation with the MMU disabled. The upper four address bits of the physical address are forced to zero. This mode requires the boot mode (BM) and MMU enable (ME) bits of the SCR to be cleared.

The translation modes are considered to be the normal operating modes of the MMU. This group includes four modes of translation operations: Translation 1–4. Translation 1 and 2 are the non-cached and cached data access modes, respectively. Translation 3 and 4 are the non-cached and cached instruction access modes. The cached and non-cached modes are identical in results for both data and instruction accesses, with the exception that the data access modes ignore the BM bit of the SCR. This feature allows the system to enable the MMU for data accesses, yet still access instructions from the boot memory space without changing the BM bit.

Note: The SPARC architecture supports the concept of address space identifiers (ASI), which provide an extension of the standard addressing space. These bits are used to enable special addressing modes, or to provide access to registers and other features of the RT625. Refer to *Section 4.9, RT625 ASI and Register Mapping* for more information.

4.3.1 MMU Invalidate and Probe Operations

4.3.1.1 Invalidate Operations

The invalidate operation allows software invalidation of selected entries in the TLB. TLB entries are invalidated by executing a Store Alternate ASI instruction using ASI = 3 H and supplying a virtual address in the format shown in *Figure 4–12*. The context number is given by the context register (CXR). All TLB entries that match the virtual address, context, and TLB invalidate type will be invalidated simultaneously. The invalidate type is specified in bits 11–8 of the virtual address for the invalidate operation.

Virtual Address Format:

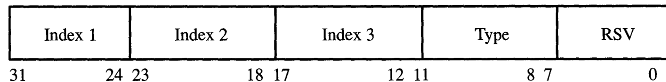


Figure 4–12. MMU Invalidate Address Format

The RT625 supports five different types of TLB invalidate operations. These types are: page, segment, region, context, and entire invalidate. The five types of invalidate operations are listed in *Table 4–5*, and define the address comparison required to match a TLB entry for invalidation. The short translation (ST) bits in the TLB entries are ignored for TLB matching. All TLB entries matching the compare criterion of the invalidate type are invalidated, including those locked by the IRC.

Table 4–5. TLB Entry Invalidation

Type	Invalidate	Compare Criterion
0	Page	Context (or ACC = 6, 7), Index 1, Index 2, and Index 3
1	Segment	Context (or ACC = 6, 7), Index 1, and Index 2
2	Region	Context (or ACC = 6, 7) and Index 1
3	Context	Context (user pages with ACC = 0 to 5)
4	Entire	None
5 to F	Reserved	

4.3.1.2 Probe Operation

The probe operation allows testing the TLB and page tables for a PTE entry corresponding to a virtual address. The operation is initiated by executing a Load Alternate ASI instruction with ASI = 3 H, the appropriate virtual address, and the context number. The context is specified by the context register. Upon starting a probe operation, the TLB is probed first. If there is a TLB hit, it returns the 32-bit physical section of the matched entry. The returned entry fields are formatted such that it is identical to a PTE (see *Section 4.2.4*, for PTE format information). If a matching entry could not be found in the TLB, a table walk is started and an appropriate 32-bit value (PTE) is returned and loaded into the TLB.

A probe operation causes the reference bit (R) to be set in the PTE by means of a table walk. When a probe operation hits the TLB, the R bit is always returned as set.

The context register and access-level protection checking are ignored for TLB matching and during the probe operation table walk. The table walk hardware checks for invalid address error and translation error exceptions and records appropriate fields in the SFSR register as in the normal table walk process. If a bus error occurs or an invalid or reserved entry is detected during the table walk, a 32-bit zero value is returned as status. If a zero value is returned, the UC, TO, BE, L, and FT fields of the SFSR are updated accordingly, but the operation does not cause an exception to the RT620.

4.4 RT625 Cache Controller

The RT625 cache controller is designed to accommodate both uniprocessor and multiprocessor system requirements. The RT625 provides bus snooping and an efficient style of cache coherency protocol.

4.4.1 RT625 Cache Modes

The RT625 cache can be programmed in two cache modes: either write-through with no write allocate or copy-back with write allocate. The two cache modes differ in how they treat cache write accesses. Write-through cache mode causes write hits to the cache to be written to both cache and main memory. Write cache misses in write-through mode only update main memory and do not modify the cache.

A write access in copy-back mode only modifies the cache. The writing of the modified cache line to main memory is deferred until the cache line is no longer required. Copy-back cache mode has the advantage of reducing traffic on the system bus. Bus traffic is reduced since all updates to memory are deferred and are performed subsequently only as absolutely required. In addition, all such data transfers are made utilizing the more efficient burst mode. The following sections describe the two cache modes in detail.

4.4.1.1 RT625 Write-through Mode with No Write Allocate

For write-through cache mode, write access cache hits cause both the cache and main memory to be updated simultaneously. A write access cache miss causes only main memory to be updated (no write allocate). write-through caching mode normally requires a processor to be held during a write miss while the data is written to main memory. The RT625 provides write buffers to prevent this delay in most cases. The write buffers store the write access and write the data to main memory as a background task. (Refer to *Section 4.4.2.8* for further information on the write buffers.)

During read access cache hits, the cached data is read out and supplied to the RT620. In the case of a read access cache miss, a cache line is fetched from main memory to load into the cache and the required data is supplied to the RT620 while data is being loaded into the cache.

4.4.1.2 RT625 Copy-back Mode with Write Allocate

When the RT625 cache is configured for copy-back mode, only the cache is updated on write access cache hits (i.e., main memory is not updated). The modified bit of the cache tag for the cache line is set on a copy-back write access (write hit or after a write miss is corrected). During write access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache and only the cache is updated. If the selected cache line is modified, it has to be flushed out to update main memory. The RT625 flushes the modified cache line from the cache and stores it into its write buffer; at the same time, it fetches the new cache line from main memory and stores it into the read buffer. After the modified cache line has been flushed into the write buffer, the new cache line is stored into the cache memory from the read buffer and the modified cache line is written out from the write buffer into main memory.

During read access cache hits, the cached data is read out and supplied to the RT620. During read access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache. If the selected cache line is modified, the selected cache line is flushed out to the RT625 write buffer, and a new cache line is fetched from main memory and stored into the read buffer. The new cache line is then stored in the cache from the read buffer, while the modified cache line stored in the write buffer is written out to main memory.

4.4.2 RT625 Cache Controller

The cache controller provides cache memory access control for a 128-Kbyte or 256-Kbyte direct-mapped cache. The cache is virtually indexed and physically tagged. The cache controller performs its task by comparing memory accesses against the address and status entries in a cache tag memory. The RT625 provides cache TAG (CTAG) memory for access comparison. Cache memory accesses from the processor (after undergoing translation through the TLB) and bus snooping operations are compared against the CTAG memory. The RT625 cache tag avoids all conflict between processor accesses and bus snooping accesses without requiring a duplicate set of tags for snooping.

The cache controller is designed to use two (or four) RT627 Cache Data Units (CDUs) for the cache memory. Each CDU is a 16-Kbyte x 32 SRAM with on-chip address and data latches and timing control. Two or four RT627s and one RT625 comprise an entire 128-Kbyte or 256-Kbyte cache system with physical bus interface and read and write buffers.

4.4.2.1 128-Kbyte Cache Sub-System

The cache is organized as 4096 cache lines of 32 bytes each. The RT625 has 4096 cache tag entries in the CTAG, one entry in each cache tag memory per cache line. Addressing for the cache indexing is provided directly from the Intra-Module Bus. The address field IMA < 16:5 > selects one of the 4096 lines of the

cache. This address field also selects the cache tag entry in the CTAG dedicated to the selected cache line. A cache hit occurs when the translated physical address matches with the physical address stored in the selected cache tag entry in CTAG. The lowest five bits of the address bus (IMA < 4:0 >) select one or more of the 32 bytes in the cache line. Cache data replacement is always performed by replacing cache lines (32 bytes).

4.4.2.2 256-Kbyte Cache Sub-System

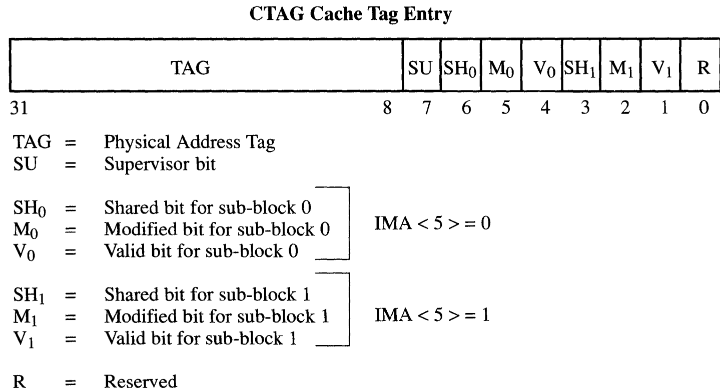
The cache is organized as 4096 cache lines of two sub-blocks, each sub-block containing 32 bytes. The RT625 has 4096 cache tag entries in the CTAG, one entry in each cache tag memory per cache line. Addressing for the cache indexing is provided directly from the Intra-Module Bus. The address field IMA < 17:6 > selects one of the 4096 lines of the cache. This address field also selects the cache tag entry in the CTAG dedicated to the selected cache line. A cache hit occurs when the translated physical address matches with the physical address stored in the selected cache tag entry in CTAG. The address bit IMA < 5 > selects one of the two sub-blocks. The lowest five bits of the address bus (IMA < 4:0 >) select one or more of the 32 bytes in the cache sub-block. Cache data replacement is always performed by replacing cache sub-blocks (32 bytes).

The cache is designed to provide data with every read access asserted on the virtual bus, regardless of the cache controller. The RT625 controls cache read accesses by holding the RT620 with $\overline{\text{PHOLD}}$ if a cache hit is not detected by the cache controller. The cache controller then reads the new cache line from main memory, and supplies the correct data to the RT620. The correct data is latched into the RT620 by strobing the $\overline{\text{IMDS}}$ signal. The RT620 is released from the hold state and execution proceeds normally. Thus, the RT625 performs the rest of the cache line fill (if needed) as a background task without holding the CPU.

Writes to the cache are controlled by the RT625, which decodes the lowest three bits of the virtual address, the IMSIZE < 1:0 > signal, and checks for a cache hit to enable the correct cache byte write enable signals. If a cache write hit occurs, the RT625 decodes the correct $\overline{\text{CBWE}}$ signals for the write access, and outputs these to the cache data unit (CDU) write enables. If the cache mode is set to write-through (see *Section 4.4.1*, Cache Modes), the write data is also written to main memory. If a write cache miss occurs for write-through cache mode, the data is written to main memory and the cache is not updated. If the write cache miss occurs during copy-back cache mode, the cache line is fetched from main memory. If the cache line stored in the cache when the write cache miss occurred has been modified, the old cache line is written to main memory and is replaced by the new cache line fetched from main memory. After the cache line has been replaced, the write access is enabled by the RT625. If both sub-blocks are dirty for a cache miss, they are both flushed to memory as two 32-byte block writes.

4.4.2.3 RT625 Cache Tag

The cache tag (CTAG) array consists of 4096 direct-mapped physical address cache tag entries. *Figure 4–13* shows the layout of a CTAG entry. Each entry in the cache consists of 24 bits of physical address (PA < 35:12 >), one supervisor bit (SU), two shared bits (SH_0 and SH_1), two modified bits (M_0 and M_1) and two valid bits (V_0 and V_1). V_0, SH_0 and M_0 are the valid, shared and modified bits for sub-block 0 of the cache line while V_1, SH_0 and M_0 are the valid, shared and modified bits for sub-block 1 of the cache line. The valid bit is set or cleared to indicate the validity of the cache sub-block. The shared bit (SH) for a cache sub-block is set when bus snooping indicates that the cache sub-block is shared. The modified bit for a cache sub-block is set when this cache is the owner of that cache sub-block in the shared memory image system.



Note: The address bit IMA < 5 > is used to select status bits (even for 128-Kbyte cache subsystem).

Figure 4–13. RT625 Cache Tag Entry

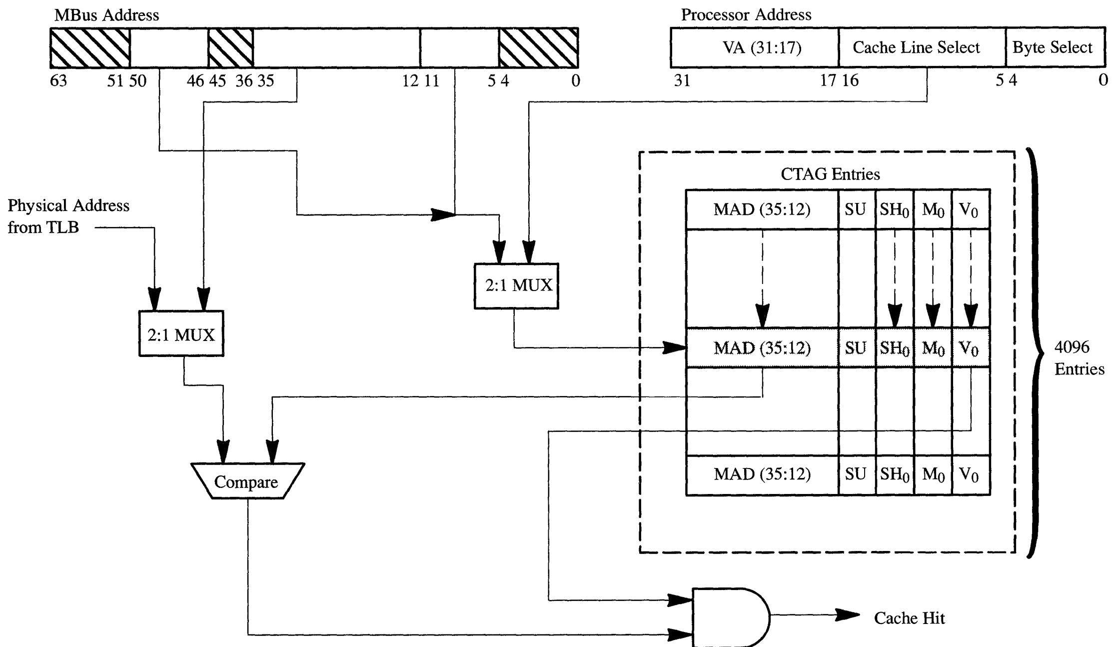


Figure 4–14. RT625 Cache TAG (CTAG) Comparison (128-Kbyte Cache)

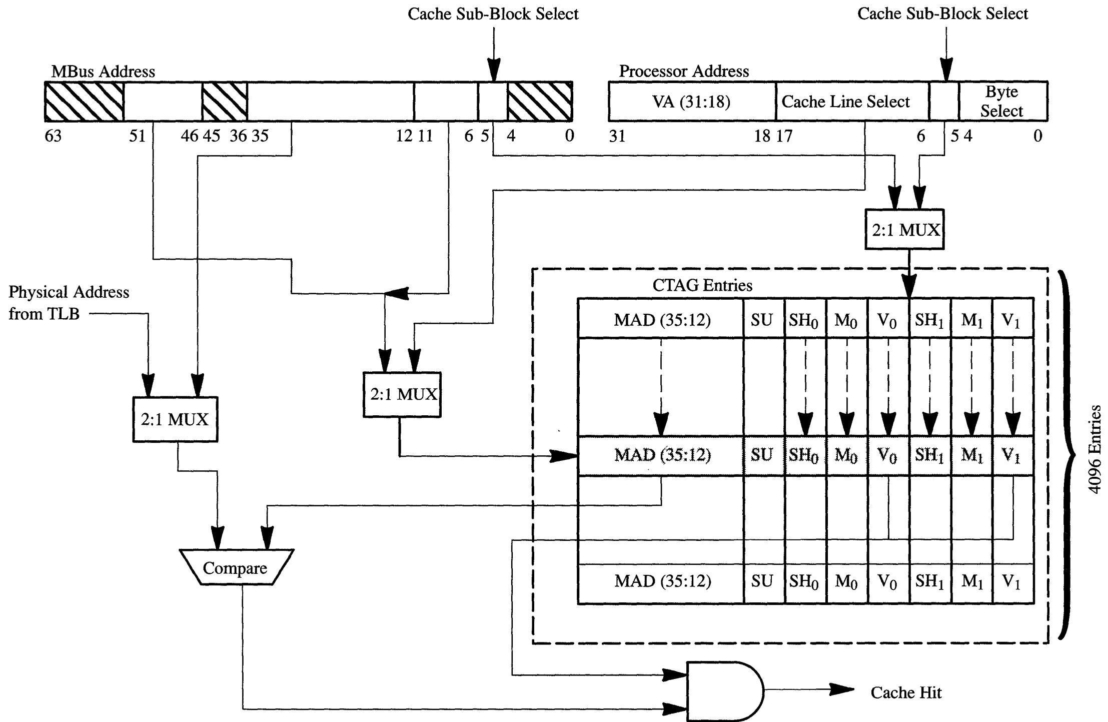


Figure 4–15. RT625 Cache TAG (CTAG) Comparison (256-Kbyte Cache)

The 4096 CTAG entries are virtual address indexed. From the processor side, the cache line select field, IMA < 16:5 > in case of 128-Kbyte cache or IMA < 17:6 > in case of 256-Kbyte cache, is used to select a cache line entry and its corresponding cache tag entry. The translated physical address is compared against the physical address of the selected cache tag entry. If a match occurs, then a cache hit is generated. If a match is not found, then a cache miss is generated. To complete an access successfully, the cache tag and the TLB must be hit with appropriate access-level permission.

From the MBus side, the index field for CTAG, as supplied by the MBus, is formed by concatenating the superset virtual address bits < 16:12 > (MAD < 50:46 >) in case of 128-Kbyte cache or < 17:12 > (MAD < 51:46 >) in case of 256-Kbyte cache with physical address bits < 11:5 > (MAD < 11:5 >) (refer to *Figure 4–14* and *Figure 4–15*). On power-on reset ($\overline{RSTIN}$ asserted), all cache tag entries are invalidated (all V bits are cleared).

4.4.2.4 RT625 Multiprocessing Support

The RT625 is designed to support multiprocessing systems. The RT625 accomplishes this by providing features necessary to maintain cache coherency with a second-level memory system (typically main memory or a secondary cache) and other caching systems on the shared bus.

The RT625 supports two modes of caching: write-through and copy-back. Operation in write-through caching mode causes main memory to be modified with each write access to the cache. This avoids the issue of lack of coherency between the individual cache systems and main memory, but greatly increases memory bus traffic. The effect of this increased bus traffic is a degradation in the performance of a multiprocessor

system as the processing nodes compete for memory bus bandwidth. This problem is greatly reduced when copy-back caching mode is used.

Operation in copy-back mode causes all changes to a cache line to be held until the line is flushed from the cache. This minimizes bus traffic to only those transactions necessary to maintain the cache. However, by allowing the cache line to be modified without updating main memory, a problem arises when other processing nodes require an up-to-date copy of that memory location. The problem of modified cache lines is solved by the enforcement of a cache coherency protocol.

The RT625 implements a cache coherency protocol specified by the Level 2 SPARC MBus standard. In this protocol, each cache line is described by one of five states: Invalid (I), Exclusive Clean (EC), Exclusive Modified (EM), Shared Clean (SC), and Shared Modified (SM). The following describes these five cache states:

Invalid (I): Cache line is not valid.

Exclusive Clean (EC): Only this cache module has a valid copy of this cache line, other than the next level of memory (main memory or secondary cache). No other cache module on the same level of memory has a valid copy of this cache line.

Exclusive Modified (EM): Only this cache module has a valid copy of this cache line. This cache module is the OWNER of the cache line, and has the responsibility to update the next level of memory (main memory or secondary cache) and also to supply data if any other cache references this memory location.

Shared Clean (SC): The same cache line may exist in more than one cache module. The next level of memory may or may not contain a valid copy of this cache line, depending upon whether this cache line has been modified in any other cache.

Shared Modified (SM): The same cache line may exist in more than one cache module, but this cache module is the OWNER of the cache line. The next level of memory does not have a valid copy of this cache line, and this cache module has the responsibility to update the next level of memory and to supply any other cache that may reference this same memory location.

These five states are described by three state bits (valid (V), shared (SH), and modified(M)) in each cache tag entry (refer to *Figure 4–13*).

Under write-through cache mode, only the valid bit applies to cache tag entries,(i.e., the entry is either valid or invalid). The shared and modified bits are not set by the RT625 in write-through mode.

4.4.2.5 RT625 Cache State Transitions

The following sections describe the five cache line states (Invalid, Exclusive Clean, Exclusive Modified, Shared Clean, and Shared Modified) and the transitions these states undergo due to transactions on both the Intra-Module Bus and the MBus. Each numbered transition in a section corresponds to a numbered transition on the state diagram for that section. Note that state transitions are dependent upon both the cache transaction and the state of the MBus signals: memory shared ($\overline{MSH}$), and memory inhibit ($\overline{MIH}$).

All processor transactions described in this section affect the processor serviced by the RT625. All coherent transactions affect all bus agents on the MBus with a copy of the shared cache line.

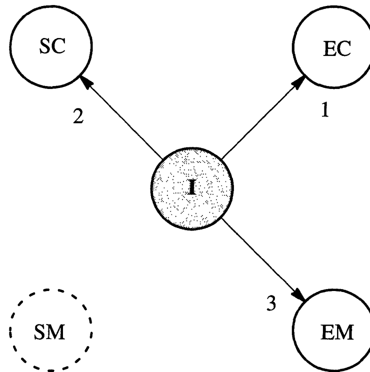


Figure 4–16. Copy-back Invalid

4.4.2.5.1 Copy-back Invalid

Processor Read Miss: RT625 issues a Coherent Read transaction on the MBus. The RT625 will read the cache line from the second-level memory and then load it into the CDUs.

1. If $\overline{\text{MSH}}$ = HIGH, then Invalid changes to Exclusive Clean.
2. If $\overline{\text{MSH}}$ = LOW, then Invalid changes to Shared Clean.

Processor Write Miss: RT625 issues a Coherent Read and Invalidate transaction on the MBus. The RT625 reads the cache line from the second-level memory and loads it into the CDUs. Then the processor data is written into the cache in the cycle following the last cache line entry.

3. The new cache line is marked as Exclusive Modified.

4.4.2.5.2 Copy-back Exclusive Clean

Processor Read Hit: The RT625 will supply data to the RT620 immediately.

1. The tag entry is Exclusive Clean: NO STATE CHANGE.

Processor Read Miss: The RT625 will issue a Coherent Read transaction on the MBus. The RT625 will read the cache line from the second-level memory and then load it into the CDUs.

2. If $\overline{\text{MSH}}$ = HIGH, then Exclusive Clean.
3. If $\overline{\text{MSH}}$ = LOW, then Exclusive Clean changes to Shared Clean.

Processor Write Hit: The RT625 will update the cache immediately with the RT620 data.

4. Exclusive Clean changes to Exclusive Modified.

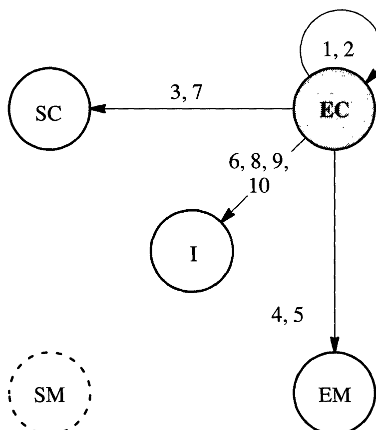


Figure 4–17. Copy-back Exclusive Clean

Processor Write Miss: The RT625 will issue a Coherent Read and Invalidate transaction on the MBus. The RT625 will read the cache line from the second-level memory and then load it into the CDUs. Then the processor data is written into the cache in the cycle following the last cache line entry.

5. The new cache line is marked as Exclusive Modified.

Software Flush Hit (Store Alternate instruction with ASI = 10H to 14H; see Section 4.4.3): The RT625 will invalidate the cache tag entry.

6. Exclusive Clean is changed to Invalid.

Coherent Read Hit: During the A+3 cycle of the MBus Coherent Read transaction, the RT625 will assert $\overline{\text{MSH}}$ and change the state of the cache line from Exclusive Clean to Shared Clean.

7. Assert $\overline{\text{MSH}}$; Exclusive Clean is changed to Shared Clean.

Coherent Read and Invalidate Hit: The cache tag entry in the RT625 is invalidated.

8. Exclusive Clean is changed to Invalid.

Coherent Invalidate Hit: The cache tag entry in the RT625 is invalidated.

9. Exclusive Clean is changed to Invalid.

Coherent Write and Invalidate Hit: The RT625 invalidates the cache tag entry.

10. Exclusive Clean is changed to Invalid.

4.4.2.5.3 Copy-back Shared Clean

Processor Read Hit: The RT625 will supply data immediately to the RT620.

1. The tag entry is Shared Clean: NO STATE CHANGE.

Processor Read Miss: The RT625 will issue a Coherent Read transaction on the MBus. The RT625 will read the cache line from the second-level memory and load it into the CDUs.

2. If $\overline{MSH}$ = HIGH, then entry is marked as Exclusive Clean.
3. If $\overline{MSH}$ = LOW, then entry is marked Shared Clean;

Processor Write Hit: The RT625 issues a Coherent Invalidate transaction on the MBus. The RT625 will wait till it receives $\overline{MRDY}$ for the Coherent Invalidate transaction and then update the cache with the processor data.

4. Shared Clean is changed to Exclusive Modified in the cache tag entry.

Processor Write Miss: The RT625 will issue a Coherent Read and Invalidate transaction on the MBus. The RT625 will read the cache line from the second-level memory and then load the data into the CDUs. The processor data is written into the cache in the cycle following the last cache line entry.

5. The new cache line is marked as Exclusive Modified in the cache tag entry.

Software Flush Hit: The RT625 will invalidate the cache tag entry.

6. Shared Clean is changed to Invalid in the cache tag entry.

Coherent Read Hit: During the A+3 cycle of the MBus Coherent Read transaction, the RT625 will assert $\overline{MSH}$.

7. Assert $\overline{MSH}$; Shared Clean in the cache tags.

Coherent Read and Invalidate Hit: The cache tag entry is invalidated.

8. Shared Clean is changed to Invalid in the cache tag entry.

Coherent Invalidate Hit: The cache tag entry is invalidated.

9. Shared Clean is changed to Invalid in the cache tags.

Coherent Write and Invalidate Hit: The cache tag entry is invalidated.

10. Shared Clean is changed to Invalid in the cache tags.

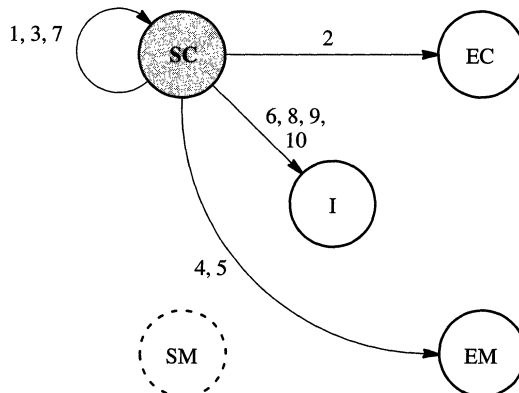


Figure 4–18. Copy-back Shared Clean

4.4.2.5.4 Copy-back Exclusive Modified

Processor Read Hit: The RT625 will supply data to the processor immediately.

1. Exclusive Modified in the cache tags: NO STATE CHANGE.

Processor Read Miss: The RT625 will initiate a Coherent Read transaction followed by a block write transaction of the previously modified cache line. The RT625 will read the cache line from the second-level memory and load the data into the CDUs. The modified cache line has to be written to update the second-level memory. The MBus busy ($\overline{MBB}$) signal is asserted from the beginning of the Coherent Read transaction to the end of the write transaction on the MBus.

2. If $\overline{MSH}$ = HIGH, the cache tag entry is changed from Exclusive Modified to Exclusive Clean.
3. If $\overline{MSH}$ = LOW, the cache tag entry is changed from Exclusive Modified to Shared Clean.

Processor Write Hit: The RT625 will update the cache immediately with the processor data.

4. Exclusive Modified remains as Exclusive Modified in the cache tags.

Processor Write Miss: The RT625 will initiate a Coherent Read and Invalidate transaction followed by a block write transaction of the previously modified cache line. The RT625 will read the cache line from the second-level memory and load it into the CDUs. The processor data is written into the CDU in the cycle following the last cache line entry into the cache. The modified cache line must be written into the second-level memory in order to update the memory. The $\overline{MBB}$ signal is asserted from the beginning of the Coherent Read and Invalidate transaction to the end of the write transaction on the MBus.

5. The cache tag entry remains Exclusive Modified.

Software Flush Hit: The RT625 initiates a block write transaction on the MBus. The RT625 writes the modified cache line to update the second-level memory and then invalidates the cache tag entry.

6. Exclusive Modified is changed to Invalid in the cache tag entry.

Coherent Read Hit: During the A+3 cycle of the Coherent Read transaction on the MBus, the RT625 asserts both the $\overline{MSH}$ and $\overline{MIH}$ signals. This RT625 is the OWNER of the cache line, and is responsible for supplying the data for the Coherent Read transaction on the MBus.

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the RT625 changes the state of the cache tag entry from Exclusive Modified to Shared Clean.
8. If the memory reflection (MR) bit of the SCR is cleared, the RT625 changes the state of the cache tag entry from Exclusive Modified to Shared Modified.

Coherent Read and Invalidate Hit: During the A+3 cycle of a Coherent Read and Invalidate transaction on the MBus, the RT625 asserts $\overline{MIH}$ signals. This RT625 is the OWNER of the cache line, and is responsible for supplying the data for the Coherent Read transaction on the MBus. The cache tag entry is invalidated.

9. Exclusive Modified is changed to Invalid in the cache tag entry.

Coherent Invalidate Hit: The cache tag entry in the RT625 is invalidated.

10. Exclusive Modified is changed to Invalid in the cache tag entry.

Coherent Write and Invalidate Hit: The cache tag entry is invalidated.

11. Exclusive Modified is changed to Invalid in the cache tag entry.

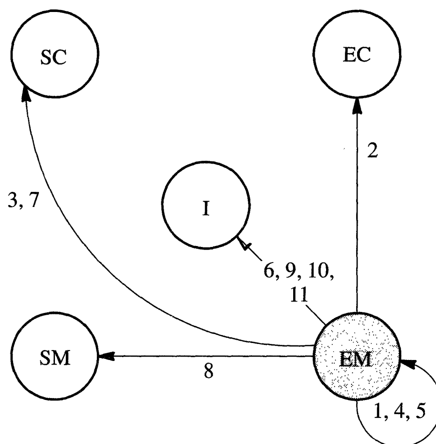


Figure 4–19. Copy-back Exclusive Modified

4.4.2.5.5 Copy-back Shared Modified

Processor Read Hit: The RT625 will supply data immediately to the RT620.

1. Shared Modified in the cache tags: NO STATE CHANGE.

Processor Read Miss: The RT625 will initiate a Coherent Read transaction followed by a block write transaction of the previously modified cache line. The RT625 will read the cache line from second-level memory and load the data into the CDUs. The modified cache line has to be written to update the second-level memory. The $\overline{\text{MBB}}$ signal is asserted from the beginning of the Coherent Read transaction to the end of the write transaction on the MBus if the bus remains granted between the accesses.

2. If $\overline{\text{MSH}}$ = HIGH, the cache tag entry is changed from Shared Modified to Exclusive Clean.
3. If $\overline{\text{MSH}}$ = LOW, the cache tag entry is changed from Shared Modified to Shared Clean.

Processor Write Hit: The RT625 initiates a Coherent Invalidate transaction on the MBus. The RT625 will wait until it receives $\overline{\text{MRDY}}$ for the Coherent Invalidate transaction and then update the cache with the processor data.

4. The cache tag entry is changed from Shared Modified to Exclusive Modified.

Processor Write Miss: The RT625 will initiate a Coherent Read and Invalidate transaction followed by a block write transaction of the previously modified cache line. The RT625 will read the cache line from second-level memory and load it into the CDUs. The processor data is written into the CDUs in the

cycle following the last cache line entry into the cache. The modified cache line must be written into second-level memory in order to update the memory. The $\overline{\text{MBB}}$ signal is asserted from the beginning of the Coherent Read and Invalidate transaction to the end of the write transaction on the MBus if the bus remains granted between the accesses.

5. The cache tag entry is changed from Shared Modified to Exclusive Modified.

Software Flush Hit: The RT625 initiates a block write transaction on the MBus. The RT625 will write the modified cache line to update second-level memory and then it invalidates the cache tag entry.

6. Shared Modified is changed to Invalid in the cache tag entry.

Coherent Read Hit: During the A+3 cycle of the Coherent Read transaction on the MBus, the RT625 asserts both the $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals. This RT625 is the OWNER of the cache line, and is responsible for supplying the data for the Coherent Read transaction on the MBus.

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the RT625 changes the state of the cache tag entry from Shared Modified to Shared Clean.
8. If the MR bit of the SCR is not set, then the cache tag entry remains Shared Modified.

Coherent Read and Invalidate Hit: During the A+3 cycle of a Coherent Read and Invalidate transaction on the MBus, the RT625 asserts $\overline{\text{MIH}}$ signals. This RT625 is the OWNER of the cache line, and is responsible for supplying the data for the Coherent Read transaction on the MBus. The cache tag entry is invalidated.

9. Shared Modified is changed to Invalid in the cache tag entry.

Coherent Invalidate Hit: The cache tag entry in the RT625 is invalidated.

10. Shared Modified is changed to Invalid in the cache tag entry.

Coherent Write and Invalidate Hit: The cache tag entry is invalidated.

11. Shared Modified is changed to Invalid in the cache tag entry.

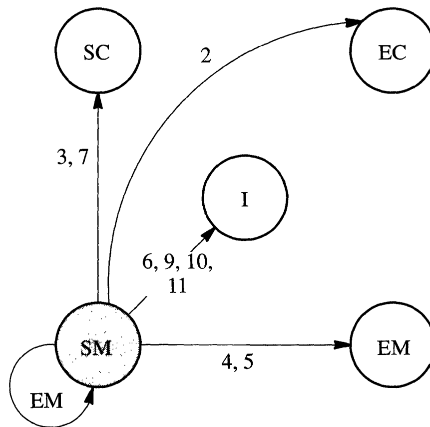


Figure 4–20. Copy-back Shared Modified

4.4.2.5.6 Write-through Invalid

Processor Read Miss: The RT625 issues a block read transaction on the MBus. The RT625 will read the cache line from second-level memory and load the data into the CDUs.

1. The cache tag entry is changed from Invalid to valid.

Processor Write Miss: The RT625 will issue a write-buffered Coherent Write and Invalidate transaction on the MBus.

2. The cache tag entry remains Invalid.

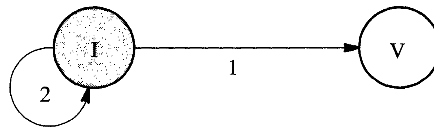


Figure 4–21. Write-through Invalid

4.4.2.5.7 Write-through Valid

Processor Read Hit: The RT625 will supply data to the RT620 immediately.

1. The cache tag entry remains valid: NO STATE CHANGE.

Processor Read Miss: The RT625 issues a block read transaction on the MBus. The RT625 will read the cache line from second-level memory and load the data into the CDUs.

2. The entry remains valid.

Processor Write Hit: The RT625 issues a write-buffered Coherent Write and Invalidate transaction on the MBus. The RT625 will write data into the cache.

3. The entry remains valid.

Processor Write Miss: The RT625 issues a write-buffered Coherent Write and Invalidate transaction on the MBus. The RT625 will not write to the cache.

4. The entry remains valid.

Software Flush Hit: The RT625 invalidates the cache tag entry.

5. The entry changes from valid to Invalid.

Coherent Read Hit: During the A+3 cycle of the MBus Coherent Read transaction, the RT625 asserts $\overline{\text{MSH}}$.

6. Assert $\overline{\text{MSH}}$; the entry remains valid.

Coherent Read and Invalidate Hit: The RT625 invalidates the cache tag entry.

7. The entry changes from valid to Invalid.

Coherent Write and Invalidate Hit: The RT625 invalidates the cache tag entry.

8. The entry changes from valid to Invalid.

Coherent Invalidate Hit: The RT625 invalidates the tag entry.

9. The entry changes from valid to Invalid.

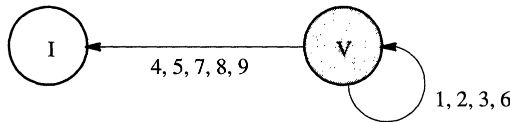


Figure 4–22. Write-through Valid

4.4.2.5.8 RT625 Bus Snooping

The RT625 bus snooper watches MBus transactions and snoops into the CTAG array for coherent transactions, as listed in Table 4–6.

Table 4–6. MBus Snooping Transactions

Cache Mode	Transaction Type	Snoop
Copy-back	Coherent Read & Invalidate	yes
	Coherent Write & Invalidate	yes
	Coherent Read	yes
	Coherent Invalidate	yes
	Read	no
	Write	no
Write-through	Coherent Read & Invalidate	yes*
	Coherent Write & Invalidate	yes
	Coherent Read	yes*
	Coherent Invalidate	yes*
	Read	no
	Write	no

*These transactions are not generated by the RT625 for normal accesses. However, the RT625 will snoop these transactions if generated by another bus master.

4.4.2.6 RT625 Address Aliasing

Two or more virtual addresses mapped to the same physical address is known as *aliasing*. This must be detected to maintain data consistency in a virtual cache system. The SPARC Reference system software convention permits the use of aliases in address spaces that are modulo with respect to the system's underlying cache size. This convention ensures that the aliased entry maps to the same cache line address for each RT625 in the multiprocessor system. With this convention, the existence of address aliasing is automatically prevented in the RT625 cache system.

4.4.2.7 RT625 Cache Control Signals

The RT625 controls the cache through control signals supplied to the RT620 and to the cache data units. The signals used by the cache controller to control the RT620 consist of $\overline{\text{PHOLD}}$, $\overline{\text{IMDS}}$ and $\overline{\text{IMBNA}}$. $\overline{\text{PHOLD}}$

is used to stall the RT620 until the RT625 can service the RT620 memory access request, such as during cache miss processing or during table walks. IMDS is used by the RT625 to strobe data into the RT620. IMBNA is used to obtain the Intra-Module Bus ownership from the RT620.

The signals used to control the cache data unit consist of the cache byte write enable (CBWE) signals and cache read output enable (CROE) signals. CROE is asserted low to enable the output of the cache data units during a cache read. CBWE < 7:0 > is asserted low to enable writing to the cache data units. The multiple CBWE signals allow the cache controller to enable byte, halfword, word or doubleword writes to the cache data unit. Byte or halfword or word or doubleword reads are handled by the RT620, which reads an entire 64-bit doubleword and internally discards unwanted bytes.

During a cache read miss, the RT625 halts the RT620 by asserting PHOLD. The RT625 also asserts IMBNA, which is used to disable the RT620 data bus and address bus output drivers and obtain the ownership of the IMB to access the cache data unit. The cache controller fetches the new cache line from main memory, asserting CBWE < 7:0 > and the cache line addresses to write the data into the cache. While writing data into the cache data unit, when the missed read data doubleword is put on the data bus, the RT625 toggles the data strobe (IMDS) signal and deasserts PHOLD. Toggling IMDS forces the RT620 to latch the data on the data bus. The cache read miss terminates by deasserting the IMBNA signal. Read misses are handled in the same manner for both copy-back and write-through modes of caching.

Cache write misses for write-through mode generally do not affect the operation of the RT620 due to the presence of write buffers in the RT625 (refer to the following section on the write buffer). In the case of a write miss, the write data is written to the write buffer instead of the cache memory. The write buffer writes the data to memory as a background task. The RT620 is stalled for a write-through write miss only if the write buffer is full. This occurs when the RT620 overruns the eight doubleword buffers in the write buffer. In this case, PHOLD is asserted until space is made available in the write buffer as it writes its contents into main memory.

On a write miss, if the cache mode is copy-back and the cache line is clean, the cache line is replaced in a similar manner as in the cache read miss described above. PHOLD is asserted to stall the RT620 and IMBNA is asserted to force the RT620 off the data and address buses. A new cache line is read from main memory, and the cache is updated by writing the data into the cache. This is accomplished by supplying the cache addresses, cache line data from main memory, and asserting the CBWE signals to write the data. The write cache miss terminates by deasserting IMBNA and PHOLD, which causes the missed write data and address to reappear on their respective buses. The RT625 then strobes CBWE < 7:0 > according to the address and IMSIZE < 1:0 > signals to write the data into the cache and allows the processor to return to execution.

If the cache line is modified, the modified cache line is read out of the cache and stored into the write buffer during the same time the new cache line is fetched from main memory and stored in the read buffer (refer to the following sections on write and read buffers). PHOLD is asserted and IMBNA asserted to force the RT620 into a halted and inactive state. The RT625 asserts CROE and the cache addresses to flush the modified cache line into the write buffer. The RT625 then writes the new cache line into the cache from the read buffer while simultaneously writing the modified cache line into main memory from the write buffer. This is accomplished by supplying the cache addresses for the cache line data, and asserting the CBWE < 7:0 > signals to write the data into the cache. The copy-back write miss for a modified cache line terminates by releasing IMBNA and PHOLD to allow the missed write data and address to reassert on the data and address buses. The RT625 asserts the appropriate CBWE < 7:0 > signals to write the data into the cache.

4.4.2.8 RT625 Write Buffer

The RT625 supports eight write buffers on-chip, as shown in *Figure 4–23*. In write-through mode, each buffer can store 64-bit data, which efficiently supports store double operations. A physical address tag is associated with each of the eight buffers in write-through mode. Upon a write access, the write buffers are

loaded with the data to be written to main memory. This allows the RT620 to continue operation without stalling due to memory access delays on the physical bus. Note that the WBE (Write Buffer Enable) bit of the SCR, or system control register (refer to *Section 4.5.1*), must be set to enable the write buffer.

PA0	V	Double Word
PA1	V	Double Word
PA2	V	Double Word
PA3	V	Double Word
PA4	V	Double Word
PA5	V	Double Word
PA6	V	Double Word
PA7	V	Double Word

Write-through mode or non-cacheable write

Figure 4–23. RT625 Write Buffers

In copy-back mode, the same buffers are configured to store a 32- or 64-byte cache line with a single physical address as shown in *Figure 4–24*. This allows for faster cache line flushes during modified cache line replacement. The modified cache line is flushed into the write buffer as the new cache line is simultaneously fetched from main memory. In either case, the contents of the buffers are transferred to main memory as a background task. For a 128-Kbyte cache system or if only one sub-block of the cache line has been modified in a 256-Kbyte cache system, only a 32-byte cache line has to be flushed out. On power-on reset ($\overline{RSTIN}$ asserted), all of the write buffers are invalidated.

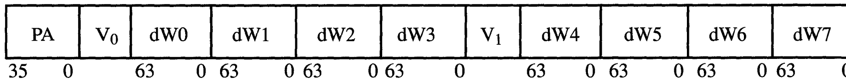


Figure 4–24. RT625 Write Buffer (Copy-back mode)

Non-cacheable writes use the eight write buffers in the same manner as write-through cache transaction, even if copy-back mode is enabled. However, a copy-back cache line and non-cacheable data cannot simultaneously occupy the write buffer. Store and Atomic Load-Store accesses on exclusive cache line hits have to wait until the write buffers are empty. This is necessary to maintain store ordering in multiprocessor systems. Otherwise modified data may be snooped by another processor while an earlier Store is still in the write buffer.

The RT625 requests MBus ownership as soon as one of the write buffers is valid. For each write buffer transfer, the RT625 re-arbitrates the MBus again. A modified cache-line flush is considered as one transaction except if sub-blocks are enabled and both are dirty (in this case, the cache flushing is done as two transactions). When the bus is still granted to the RT625 (i.e., bus parking), the RT625 can transfer the data immediately without any bus re-arbitration (so there are no dead clocks between transactions). Once all of the write buffers are full, further writes from the RT620 are held until a buffer is empty. If there is a read access cache miss, the RT620 is held until all of the write buffers are written back into main memory in order to maintain data consistency. After the write buffers are cleared, the RT625 resumes the task of fetching the cache line for the cache read miss.

4.4.2.9 RT625 Read Buffer

The RT625 provides a read buffer of 32 bytes (one cache line) in order to support simultaneous writing of a modified cache line to main memory and reading of a new cache line from main memory into the cache under copy-back mode. The read buffer is shown in *Figure 4–25*. The read buffer is invalidated on power-on reset.

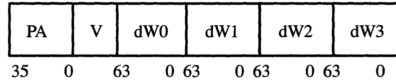


Figure 4–25. RT625 Read Buffer (copy-back mode)

4.4.3 RT625 Software Cache Flushing Operations

The RT625 supports five different levels of cache flush operations, as illustrated in *Table 4–7*. The cache flush operations are dependent upon the cache mode and state. Flushing under copy-back cache mode for a modified cache line means flushing the cache line into main memory and invalidating the cache tag entry. If the cache line is clean (copy-back mode), or is in write-through cache mode, flushing only invalidates the cache tag entry.

Table 4–7. Cache Flush operations

Cache Flush	ASI	Compare
PAGE	10 H	PA [35:12]
SEGMENT	11 H	None
REGION	12 H	None
CONTEXT	13 H	None
USER	14 H	User (S = 0)

Unlike a TLB flush operation, all cache flushing operations flush only one cache line at a time. The cache line selected for operation is indexed as in normal cache access operations (IMA < 16:5 > in case of 128-Kbyte or IMA < 17:6 > in case of 256-Kbyte). The page cache flush virtual address is translated through the TLB to obtain the physical address and then compared with the selected cache tag entry. If the cache flush operation does not match the cache tag entry, no action occurs. The five types of cache flush operations are page flush, segment flush, region flush, context flush and user flush as illustrated in *Table 4–7*. Segment, region and context are unconditional flushes and their virtual address need not be translated through the TLB. The virtual address need not be translated through the TLB in the case of a user flush either; if the cache line is marked as user (SU = 0), it will be flushed. These different levels of cache flush are mapped with the ASI bits. The Store Alternate space instructions for the RT620 must be used to assert the ASI value that corresponds to the level of cache flush operation desired. The combination of the ASI and a Store operation using the virtual address specifies the cache flush operation and the cache line to be matched for flushing. Since, for the page flush, the virtual address undergoes translation through the TLB, a table walk may be required if a TLB miss occurs. During address translation for a cache flush (through the TLB or a table-walk if there is TLB miss), access-level checking is not performed. In a 256-Kbyte cache sub-system, the flush command applies to both sub-blocks; if both sub-blocks are dirty, the cache flush is done as two transactions on the MBus.

4.4.4 RT625 Cacheable/Non-Cacheable Memory Accesses

Pages that are declared as non-cacheable (C = 0 in the page table entry (PTE)) are not cached in the cache data unit and, as such, there are no associated cache tag entries in the RT625. For data consistency and implementation reasons, the RT625 assumes the following cycles are also non-cacheable:

- table walk accesses

- boot mode accesses (except user/supervisor data accesses when the MMU is enabled and the cache is enabled)
- pass-through mode accesses
- by-pass mode accesses
- accesses while the cache is disabled
- accesses when MMU is disabled (ME bit of SCR = 0)
- write-through Atomic Load-Store are non-cacheable.

Table 4–8 shows the RT625 operation conditions for cacheable and non-cacheable accesses. Refer to the section on MMU operation modes for additional information.

Table 4–8. Cacheable/Non-Cacheable Accesses

Access	Condition
Not cached	ASI = 20–2F H (by-pass)
	ASI = UN, RES (unassigned/reserved)
	BM = 1 and ME = x and CE = x and ASI = 8, 9 H
	BM = x and not (ME = 1 and CE = 1 and PTE[C]= 1)
	Table walk cycles
	Write-through Atomic Load-Store
Cached	BM = 0 and ME = 1 and CE = 1 and ASI= 8, 9, A, B H and PTE[C] = 1
	BM = 1 and ME = 1 and CE = 1 and ASI = A,B H and PTE [C] = 1

4.4.5 RT625 MBus Cacheable (MC) Bit

One of the RT625 output signals is a MBus cacheable bit, which is embedded in the MBus address phase as MAD < 43 >. The MBus cacheable bit indicates the cacheable status of a memory access by the RT625. This information is consistent with the cache visibility philosophy of the RT625 and is made available for use by a secondary cache tag array.

When the MMU is enabled, the MC bit is set by the state of the C bit in the corresponding PTE entry. When the MMU function of the RT625 is disabled, the C bit of the SCR register sets the value of the MC bit. The C bit of the SCR register is loaded by the RT620, and it defines the cacheable status of memory accesses when the MMU is disabled. *Table 4–9* illustrates the state of the MC bit for various RT625 operation conditions.

Table 4–9. State Table for MC (Memory Cacheable) Bit

MC	Condition
0	ASI = 20–2F H
Not Applicable	ASI = UN, RES
SCR[C]	Not one of the above and ME = 0 or Not one of the above and (BM = 1 and ASI = 8, 9 H) or Not one of the above and table walk
PTE[C]	Not one of the above

4.4.6 RT625 LDST (Atomic Load-Store Instruction) Cycles

When the cache is in copy-back mode, LDST cycles are treated as normal memory accesses and are cached according to the C bit of the PTE associated with the access.

LDST operations on the physical bus (MBus) are repeated if interrupted by a *Relinquish and Retry* before the Load operation of the LDST has been completed. However, if the *Relinquish and Retry* occurs after the load operation has completed, only the Store operation of the LDST is repeated.

4.4.7 RT625 Cache Byte Write Enables

The RT625 supports eight separate byte write enables (CBWE < 7:0 >) to control write accesses to the cache data units (CDUs). These signals are generated using the lower three bits of the virtual address (IMA < 2:0 >) and size (IMSIZE < 1:0 >) information during write accesses.

The decoding of the IMSIZE < 1:0 > and VA < 2:0 > bits is shown in *Table 4–10.* The CBWE0 signal controls the most significant byte (MSB), which is located at a doubleword aligned address N. CBWE7 controls the least-significant byte, located at address N+7.

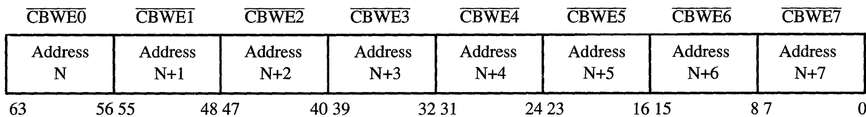


Figure 4–26. CBWE Byte Assignments

Table 4–10. Cache Byte Write Enables

IMSIZE < 1:0 >	IMA < 2:0 >	CBWE7	CBWE6	CBWE5	CBWE4	CBWE3	CBWE2	CBWE1	CBWE0
00	000	1	1	1	1	1	1	1	0
00	001	1	1	1	1	1	1	0	1
00	010	1	1	1	1	1	0	1	1
00	011	1	1	1	1	0	1	1	1
00	100	1	1	1	0	1	1	1	1
00	101	1	1	0	1	1	1	1	1
00	110	1	0	1	1	1	1	1	1
00	111	0	1	1	1	1	1	1	1
01	000	1	1	1	1	1	1	0	0
01*	001*	X	X	X	X	X	X	X	X
01	010	1	1	1	1	0	0	1	1
01*	011*	X	X	X	X	X	X	X	X
01	100	1	1	0	0	1	1	1	1
01*	101*	X	X	X	X	X	X	X	X
01	110	0	0	1	1	1	1	1	1
01*	111*	X	X	X	X	X	X	X	X
10	000	1	1	1	1	0	0	0	0
10*	001*	X	X	X	X	X	X	X	X
10*	010*	X	X	X	X	X	X	X	X
10*	011*	X	X	X	X	X	X	X	X
10	100	0	0	0	0	1	1	1	1
10*	101*	X	X	X	X	X	X	X	X
10*	110*	X	X	X	X	X	X	X	X
10*	111*	X	X	X	X	X	X	X	X
11	000	0	0	0	0	0	0	0	0
11*	001*	X	X	X	X	X	X	X	X
11*	010*	X	X	X	X	X	X	X	X
11*	011*	X	X	X	X	X	X	X	X
11*	100*	X	X	X	X	X	X	X	X
11*	101*	X	X	X	X	X	X	X	X
11*	110*	X	X	X	X	X	X	X	X
11*	111*	X	X	X	X	X	X	X	X

*Denotes an illegal combination of IMSIZE < 1:0 > and IMA < 2:0 >. CBWE values are undefined for illegal combinations.

4.4.8 Cache Data Forwarding

The RT625 supports cache data forwarding on cache read miss accesses both in write-through and copy-back modes. (Note that the CWR bit of the SCR (refer to *Section 4.5.1*) must be set to enable this feature.) The RT625 puts out a request for the data at the doubleword address for which it detected a cache miss on the MBus. As soon as the requested data is available, the RT625 strobes the data into the CPU while loading the data into the cache and releases the CPU from the pipeline hold state. Thus the CPU can continue execution while the rest of the cache line fill continues as a background task. If the RT620 has requested a double word, the RT625 will assert $\overline{\text{IMDS}}$ whenever there is valid data on the IMB as it is filling the cache line. In this case, the RT620 can use the $\overline{\text{IMDS}}$ signal to latch the data if necessary. The address placed on the MBus is such that the requested data comes first and then wraps around on the cache line boundary if necessary in order to complete the cache line fill.

Due to forwarding on cache read miss accesses, the errors (UC,TO,BE) on the MBus cannot be signaled properly if the error occurs after the acknowledge of the first data transfer (of the cache line). In such a case, the RT625 marks the cache line as Invalid. Thus, if the CPU were to continue accessing the other words in this same cache line, eventually the error would occur on the first data transfer and the error would be signaled to the CPU synchronously. Also, due to the same reason, the R&R and *Retry* cannot be acknowledged properly if the error occurs after the acknowledge of the first data transfer and an asynchronous error results. The AFAR and AFSR are updated and the System Error (SE) bit in the AFSR (bit 13) is set.

4.5 RT625 Registers

This section describes the control and data registers for the RT625.

All values in all control registers are read/write (with the exception of the *implementation* and *version* fields of the SCR). Control registers are accessible by use of the alternate space Load or Store instructions with ASI = 4 (AFAR and AFSR are read-only registers). Please refer to *Section 4.9*, ASI and Register Mapping, for more information on register addressing.

Programmer's Note: To ensure software compatibility with future versions of the RT625, reserved fields in a register should be written as zeros and masked out when read.

4.5.1 RT625 System Control Register (SCR)

The system control register (SCR), as shown in *Figure 4–27*, defines the operation modes for the cache controller and MMU. Refer to *Section 4.3* for additional information on the operation modes of the MMU. The following describes the functions of the bit fields in the SCR.

IMPL	VER	RSV	CWR	SE	WBE	MID (3:0)	BM	C	CS	MR	CM	RSV	CE	RSV	NF	ME
31 28	27 24	23 22	21	20	19	18 15	14	13	12	11	10	9	8	7 2	1	0

IMPL	= Specific Implementation of the MMU	C	= Cacheable (when MMU disabled)
VER	= Version of Specific Implementation (typically mask revision)	CS	= Cache Size
		MR	= Memory Reflection
CWR	= Cache Wrap	CM	= Cache Mode
SE	= Snoop Enable	CE	= Cache Enable
WBE	= Write Buffer Enable	NF	= No Fault
MID (3:0)	= Module Identifier	ME	= MMU Enable
BM	= Boot Mode	RSV	= Reserved

Figure 4–27. RT625 System Control Register (SCR)

IMPL, VER The implementation number (SCR < 31:28>) and the version number (SCR < 27:24 >) fields are hardwired; they are read-only fields and writes to these fields are ignored. The assignments for the RT625 are: implementation number field = 0001, version number field = 0111.

CWR *Cache Wrap Enable* (SCR < 21 >) indicates if cache wrapping is enabled. This bit is set to 1 to enable cache wrapping.

SE *Snoop Enable* (SCR < 20 >) indicates if the cache will be snooped for MBus Coherent transactions. This bit is set to 1 to enable snooping.

WBE *Write Buffer Enable* (SCR < 19 >) indicates if the write buffers are enabled. This bit is set to 1 to enable the write buffers.

MID < 3:0 > *Module Identification number* (SCR < 18:15 >) identifies the processor module during transactions on the MBus. This four bit module identification number is embedded in the MBus address phase of all MBus transactions initiated by the RT625. These bits always reflect the hardwired module ID input pins. Writes to them by software are ignored.

BM *Boot-mode bit* (SCR < 14 >) indicates the system is in boot mode. This bit is set to 1 to indicate boot mode. This bit is automatically set upon power-on reset.

C *Cacheable bit* (SCR < 13 >) indicates whether the access is cacheable or not when the MMU is disabled (this bit is independent of the CE bit, see *Section 4.4.4, Cacheable/Non-cacheable Memory Accesses* for more details.) This bit is set to 1 if accesses on the physical bus (with the MMU disabled) are to be considered cacheable.

CS *Cache Size* (SCR < 12 >) CS = 0 indicates a 128-Kbyte cache subsystem. CS = 1 indicates a 256-Kbyte cache subsystem.

MR *Memory Reflection* (SCR < 11 >) MR = 1 indicates that the main memory system on the MBus supports memory reflection. MR affects the status of the CTAG bits as described in *Section 4.4.2.5*.

CM *Cache-mode bit* (SCR < 10 >) indicates whether the cache is operating under write-through no write allocate policy or copy-back write allocate policy. This bit is set to 1 to enable copy-back cache mode. Setting this bit to 0 will enable write-through cache mode.

CE *Cache-enable bit* (SCR < 8 >) indicates whether the cache is enabled or not. This bit is set to 1 to enable the cache controller.

NF *No-fault bit* (SCR < 1 >) prevents accesses other than supervisor instruction accesses from signaling faults to the RT620. When the NF bit is set, exception-generating logic (in both the TLB and the table walk) does not indicate faults to the RT620 (via $\overline{\text{IMEXC}}$) unless the access is a supervisor instruction access (ASI 9), but status and address information is recorded in the SFSR and SFAR registers as in normal access operations. When the NF bit is not set, the RT625 will signal an exception for all faults.

ME *MMU-enable bit* (SCR < 0 >) indicates whether the MMU is enabled or not. This bit is set to 1 to enable the MMU.

Upon power-on reset, all writable control bits except the BM bit are cleared. This sets the RT625 into the following state: snoop disabled (SE = 0), write buffers disabled (WBE = 0), cache disabled (CE = 0), write-through mode (CM = 0), non-cacheable (C = 0), boot-mode enabled (BM = 1), cache size (CS = 0), memory reflection disabled (MR = 0), no fault disabled (NF = 0), and MMU disabled (ME = 0).

4.5.2 RT625 Context Table Pointer Register (CTPR)

The context table pointer points to the context table in physical memory. The table is indexed by the contents of the context register. The context table pointer appears on bits 35 through 14 of the MBus (MAD < 35:14 >) during the first fetch of TLB miss processing. Once the root pointer is cached in the page table pointer cache (PTPC) no fetching of the root pointer is required until the context is changed.

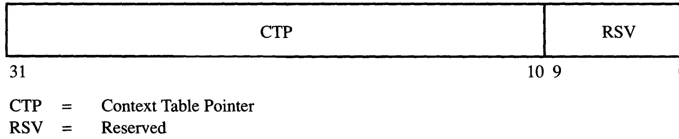


Figure 4–28. RT625 Context Table Pointer Register

4.5.3 RT625 Context Register (CXR)

The context register defines a virtual address space associated with the current process. The CXR is a twelve-bit register, which supports 4096 contexts. This register is used to define the current context. Nearly all the RT625 operations are dependent upon matching the value of this register to a TLB entry.

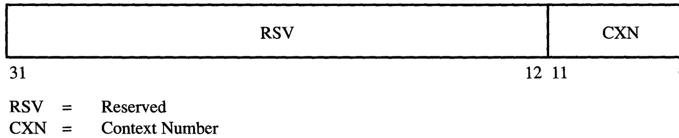


Figure 4–29. RT625 Context Register

4.5.4 RT625 Reset Register (RR)

The RR register contains information regarding whether watchdog reset (WDR) or software internal reset (SIR) occurred. This is a read/write register, and setting the software internal reset bit (SIR) causes the reset. Refer to *Section 4.8* for more details on reset processing. Upon power-on reset, the WDR and SIR bits in the RR will be cleared. Reading the RR will also clear these bits.

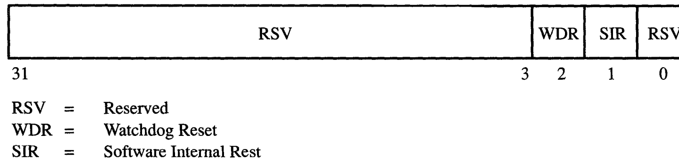
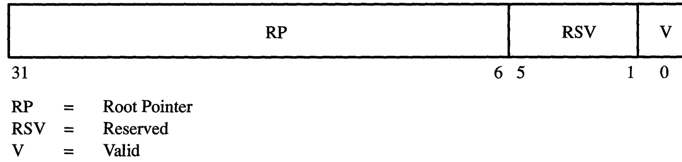


Figure 4–30. RT625 Reset Register

4.5.5 RT625 Root Pointer Register (RPR)

The RPR is the context level page table pointer (PTP) and is cached in the page table pointer cache. Refer to *Section 4.2.5* for information on the page table pointer cache.

On power-on reset, the V bit is cleared. When the current context is changed by writing to the context register (CXR), the V bit of the RPR is cleared. The V bit is also cleared when the CTPR register is written.

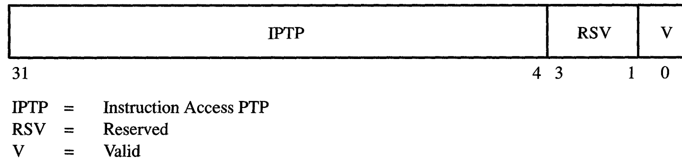


RP = Root Pointer
RSV = Reserved
V = Valid

Figure 4–31. RT625 Root Pointer Register

4.5.6 RT625 Instruction access PTPs (IPTP0, IPTP1)

The IPTPs are the instruction access Level 2 table page table pointers (PTPs) and are part of the page table pointer cache. On power-on reset, the V bit is cleared.

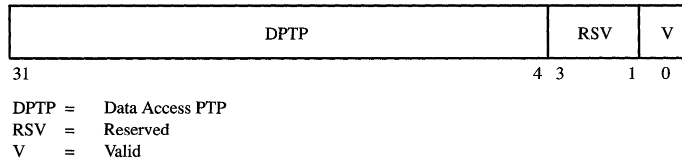


IPTP = Instruction Access PTP
RSV = Reserved
V = Valid

Figure 4–32. RT625 Instruction Access PTP Registers

4.5.7 RT625 Data Access PTPs (DPTP0, DPTP1)

The DPTPs are the data access Level 2 table page table pointers (PTPs) and are registers in the page table pointer cache. On power-on reset, the V bit is cleared.

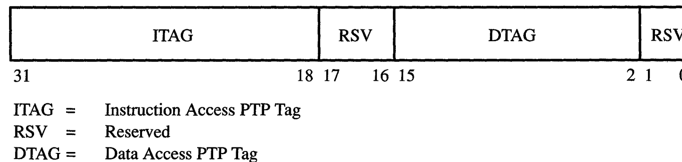


DPTP = Data Access PTP
RSV = Reserved
V = Valid

Figure 4–33. RT625 Data Access PTP Registers

4.5.8 RT625 Index Tag Registers (ITR0, ITR1)

The ITRx contains the tag (index1 and index2) fields of the IPTPx and DPTPx entries. Refer to *Section 4.2.5* for information on the PTP cache.



ITAG = Instruction Access PTP Tag
RSV = Reserved
DTAG = Data Access PTP Tag

Figure 4–34. RT625 Index Tag Registers

4.5.9 RT625 TLB Replacement Control Register (TRCR)

The TRCR contains the Replacement Counter (RC) and Initial Replacement Counter (IRC) fields as shown in *Figure 4–35*. These fields are used in order to support random replacement and to support locking capabil-

ities of the TLB. Refer to *Section 4.2.1.2* for information on TLB entry locking. Upon power-on reset, both the RC and IRC fields are initialized to zero.

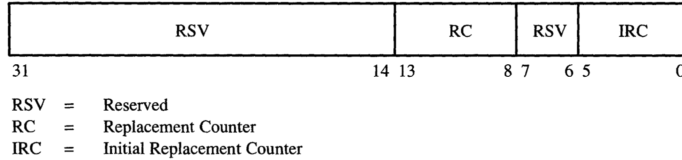


Figure 4–35. RT625 Replacement Control Register

4.5.10 RT625 Synchronous Fault Status Register (SFSR)

The Synchronous Fault Status Register, illustrated in *Figure 4–36*, contains fault-associated information for synchronous faults. Synchronous faults are faults that occur during a central processing unit access of memory. Synchronous faults include almost all possible faults for the RT625. This type of fault is synchronous to the operations of the RT620. For the RT625, this fault type covers all cases except those caused by delayed writes of data stored in the write buffers. The delayed write faults are asynchronous to the operation of the RT620, and are named asynchronous faults.

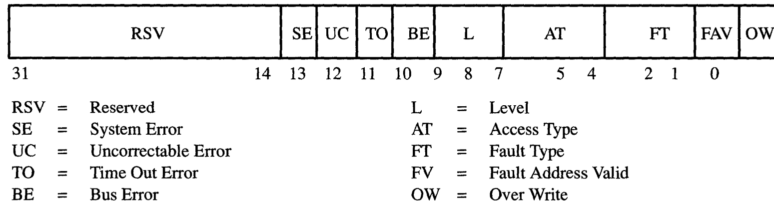


Figure 4–36. RT625 Synchronous Fault Status Register

An example of a synchronous fault is a privilege violation fault caused by attempting an unauthorized memory access. These faults are discussed in detail in *Section 4.10*. Upon encountering a synchronous fault, the RT625 asserts the $\overline{\text{EXC}}$ signal. Synchronous faults are the only exception type that assert the $\overline{\text{EXC}}$ signal.

The Uncorrectable Error (UE), Timeout Error (TO), and Bus Error (BE) bits report error status as encoded in the $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MRDY}}$ signals. The System Error (SE) bit in the SFSR register is set if R&R or *retry* occurs after the acknowledge for the first data transfer has been received during a write or Atomic LDST miss.

The level (L) bits describe the level in a table walk process at which the fault occurred (if applicable). These bits are described in *Table 4–18*.

The Access Type bits (AT < 2:0 >) describe the Access Type that caused the fault. This field specifies user/supervisor access and whether the access is a Load or Store of data or instruction. The AT bits are described in *Table 4–19* in the section on synchronous faults. The fault type bits (FT) describe the fault type, and are illustrated in *Table 4–20*. The fault address valid bit is set when the address in the synchronous fault address register (SFAR) is a valid fault address. The over-write (OW) bit is set in the case of a double fault where the fault status stored in the SFSR does not correspond with the fault first trapped on by the RT620. This is discussed in detail in *Section 4.10, Synchronous Faults.*

Upon power-on reset, the SE, UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.

4.5.11 RT625 Synchronous Fault Address Register (SFAR)

The synchronous fault address register contains the faulted virtual address. (see *Figure 4–37)*

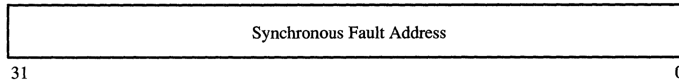


Figure 4–37. Synchronous Fault Address Register

4.5.12 RT625 Asynchronous Fault Status Register (AFSR)

Asynchronous faults are those faults caused by a delayed memory access initiated by the RT625. This type of error can only be caused by a delayed write to main memory initiated by the write buffer, or a *Relinquish and Retry*, or a *Retry* on a cache miss read which occurs after the first data transfer (the SE bit in the AFSR is set in this case). Asynchronous faults cause the $\overline{\text{AERR}}$ signal to be asserted, which can be used as an interrupt to the RT620.

The UC, TO, and BE bits are identical to those in the SFSR. They are set by the information encoded into the $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$,, and $\overline{\text{MRDY}}$ signals of the MBus. The asynchronous fault address bits provide the upper four bits of the physical address not captured in the asynchronous fault address register (AFAR), which is a 32-bit register.

The asynchronous fault occurred (AFO) bit is set when an asynchronous fault is encountered. Once the asynchronous fault occurred bit is set, no further asynchronous faults are recorded until the AFO bit is cleared, which is accomplished by reading the asynchronous fault address register (see *Figure 4–38*). The SE, UC, TO, and BE bits in the AFSR are undefined except when the AFO bit is set. The AFO bit is cleared upon power-on-reset. Reading the AFAR will also clear the AFO bit.

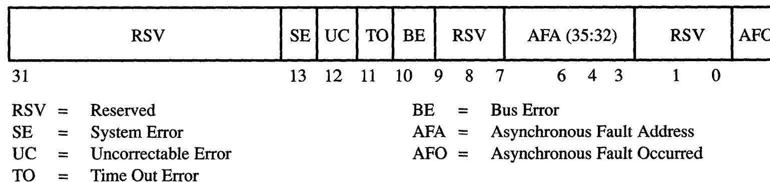


Figure 4–38. RT625 Asynchronous Fault Status Register

4.5.13 RT625 Asynchronous Fault Address Register (AFAR)

The AFAR contains bits 31 through 0 of the physical address for asynchronous faults (Bus Errors) (see *Figure 4–39*). Asynchronous faults can occur during

- delayed write accesses.
- background cache line flush operations in copy-back mode.
- delayed writes during Block Copy or Block Fill.
- an R&R or *Retry* during a cache read miss after the first data transfer has completed.

The address in the AFAR is concatenated with the four AFA bits in the AFSR to define the entire 36-bit physical address.

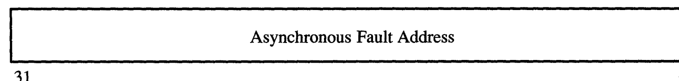


Figure 4–39. RT625 Asynchronous Fault Address Register

4.6 RT625 Block Copy and Block Fill

Block Copy and Block Fill operations are performed by the software to increase the performance of data movement to and from main memory. The RT625 provides support for both of these block manipulation functions. Block Copy and Block Fill will work only on cache line boundaries.

4.6.1 RT625 Block Copy

Block Copy copies an entire 32-byte block of data from a cache or main memory location to another location in main memory. A Block Copy is performed when a STA instruction with ASI equal to 0x17 is detected (the RT625 does not check the size of the Store Alternate instruction; it is the responsibility of the software to ensure that only a single-word size Store Alternate instruction with an ASI of 0x17 is issued). In copy-back mode if

- both the source and destination blocks are cacheable, the RT625 will perform a MBus Coherent Read followed by a Coherent Write and Invalidate.
- the source is cacheable and the destination is non-cacheable, it performs a Coherent Read followed by a normal block write.
- the source is non-cacheable and the destination is cacheable, it performs a normal block read followed by a Coherent Write and Invalidate.
- both are non-cacheable, the RT625 does a normal block read followed by a normal block write.

In case of write-through mode, the RT625 does a normal block read followed by a Coherent Write and Invalidate. The virtual address of the Coherent Read transaction (source) comes from the address of the STA operation (i.e., read address = "r[rs1] + r[rs2]"). The data portion of the STA defines the virtual address of the Coherent Write and Invalidate transaction destination (i.e., write address = r[rd]).

A table walk may be needed prior to starting the Coherent Read transaction if the source address is missed in the TLB. The Coherent Read transaction is performed like a cache read miss and the incoming data will be placed in the read buffer. It may be necessary to perform another table walk prior to starting the Coherent Write and Invalidate transaction if the destination address is missed in the TLB. Then the Coherent Write and Invalidate transaction is performed. If the RT625 that is performing the Block Copy operation "owns" the source block, it still performs the Coherent Read transaction. If the RT625 which is performing the Block Copy operation holds the destination block, it will invalidate that block for the Coherent Write and Invalidate transaction. During the address translation of both the source and destination for Block Copy operation, access level checking is performed in the TLB and during table walk (if required). The referenced and modified bits are updated (if required) as in normal table walk operations. The source access is treated as a normal supervisor data read access and the destination access is treated as a normal supervisor data write access for protection checking. If a synchronous exception occurs on the source or destination, the SFAR and the SFSR hold the proper information for the trap handler. The MBus is locked by asserting $\overline{\text{MBB}}$ during the source and destination transactions (cacheable or non-cacheable).

4.6.2 RT625 Block Fill

Block Fill operation is very similar to Block Copy operation except that the Coherent Read operation is not required since the source data comes from the processor. When a STDA instruction with ASI equal to 0x1F is executed with the size of the operation equal to a double word, the RT625 will perform the Block Fill operation (the RT625 does not check the size of the Store Alternate instruction; it is the responsibility of the software to ensure that only a Store Double Alternate instruction with an ASI of 0x1f is issued so that valid data

is used for filling the block). The store address defines the destination virtual address. The store data defines the pattern which should be filled in that 32-byte block.

A table walk may be needed prior to starting the Coherent Write and Invalidate transaction if the destination address is missed in the TLB. If the RT625 which is performing the Block Fill operation holds the destination block, it will invalidate that block for the Coherent Write and Invalidate transaction. If the block is non-cacheable, then a normal block write is performed. In write-through mode, a Coherent Write and Invalidate is performed. During the address translation of the destination for Block Fill operation, access level checking is performed in the TLB and during table walk (if required). The reference and modified bits are updated (if required) as in normal table walk operations.

4.7 RT625 Diagnostic Support

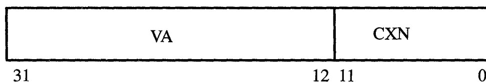
4.7.1 RT625 MMU TLB Entries

TLB entries can be accessed with a Load or Store Alternate instruction with the TLB entry address and ASI = 6H. This feature is supported for diagnostic purposes and to provide the RT620 with access to locked TLB entries. The virtual and physical sections of each entry in the TLB can be accessed by the RT620 as a word read or write. The address mapping for the TLB entries is shown in *Table 4–11*. The format of CAM word and RAM word entries in the TLB are shown in *Figure 4–40*.

Table 4–11. TLB Entry Address Mapping

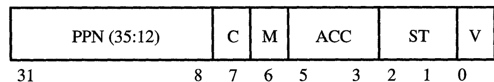
Address	TLB Entry Register
00 H	Entry 0 RAM Word
08 H	Entry 0 CAM Word
10 H	Entry 1 RAM Word
18 H	Entry 1 CAM Word
20 H	Entry 2 RAM Word
28 H	Entry 2 CAM Word
• • •	• • •
3E0 H	Entry 62 RAM Word
3E8 H	Entry 62 CAM Word
3F0 H	Entry 63 RAM Word
3F8 H	Entry 63 CAM Word
400–FFFFFFF8 H	Reserved

TLB Entry CAM Word Format



VA = Virtual Address
CXN = Context Number

TLB Entry RAM Word Format



PPN = Physical Page Number
C = Cacheable Bit
M = Modified Big
ACC = Access Protection Bits
ST = Short Translation Type
V = Valid

Figure 4–40. TLB Entry Format

4.7.2 RT625 Cache Tag Entries

RT625 CTAG entries are accessed using word LDST Alternate instructions with the cache tag entry address and ASI = 0x0E. Each tag entry can be read as a Load single or can be written as a Store single by the RT620. The address mapping for the Cache Tag entries is shown in *Table 4–12*. The RT625 CTAG entry format is illustrated in *Figure 4–41*.

Table 4–12. Cache Tag Entry Address Mapping

128K-byte		256K-byte	
Address	**Cache Tag Entry**	**Address**	**Cache Tag Entry**
0000x H	0	0000x H	0
0002x H	1	0004x H	1
0004x H	2	0008x H	2
0006x H	3	000cx H	3
• • •	• • •	• • •	• • •
1FFEx H	4095	3FFCx H	4095

(X = Don't Care)

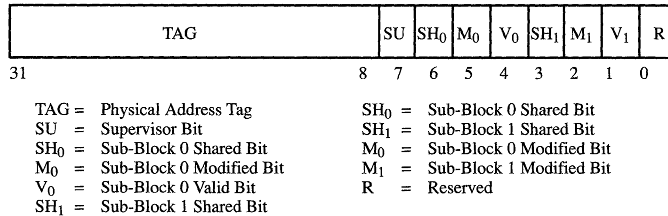


Figure 4–41. RT625 Cache Tag Entry Format

4.7.3 CDU Cache Data Entries

Cache data entries can be accessed from the cache data unit by using a LDST Alternate instruction asserting the virtual address and ASI = 0F H. The RT625 causes a forced hit from the cache tag during these accesses. All data widths are supported for a read or write to the CDU.

4.8 RT625 Reset

4.8.1 Power-On Reset ($\overline{RSTIN}$)

Upon power-on reset, the entire system is forced into a defined state. The TLB and the cache tag in the RT625 are invalidated, all valid bits in control registers are cleared, and certain bits in the AFSR and SFSR are cleared as described in the previous sections. The RT625 asserts $\overline{PRST}$ to the RT620 for as long as $\overline{RSTIN}$ is asserted. $\overline{RSTIN}$ must be asserted for a minimum of 16 clocks. The bits in the Reset Register (RR) are cleared.

Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. The SCR fields in the RT625 will have the following state after a power-on reset:

Table 4–13. Power-On Reset States

SCR field	POR state	SCR field	POR state
IMPL	Unchanged	CS	0
VER	Unchanged	CM	0
SE	0	CE	0
WBE	0	NF	0
MID<3:0>	see note 1	ME	0
BM	1	MR	0
C	0		

Note 1. MID<3:0> value is latched from the MID input pins

4.8.2 Watchdog Reset (WDR)

When the RT620 encounters a trap while traps are disabled, the RT620 enters into an error state, asserts the PERROR signal, and then halts. The only way to restart the RT620 in the error state is to assert its PRST signal. The RT625 does this by performing a watchdog reset, which asserts the PRST signal for 16 clock cycles. The TLB and the cache tag in the RT625 are not invalidated. The WDR (RR < 2 >) bit in the RR register is set. All SCR fields except boot mode (BM) are unchanged. BM is set to 1 after a watchdog reset. The RT625 also asserts AERR during watchdog reset. AERR is cleared when the Reset Register is read.

4.8.3 Software Internal Reset (SIR)

The operating system can reset the RT620 by setting the SIR bit in the Reset Register. The RT625 asserts PRST for 16 clock cycles to reset the RT620. The TLB and cache tag are not invalidated. All SCR fields except BM are unchanged, and BM is set to 1 after a software internal reset. The contents of the Reset Register are unchanged and the SIR bit will remain set.

4.9 RT625 ASI and Register Mapping

The RT625 uses the address space identifier bus (IMASI < 5:0 >) to provide the RT620 with access to the RT625's internal registers and resources, such as the cache tag and the TLB. The RT625 also uses the ASI bus to map restricted memory access functions, such as bypass memory addressing modes. Register access to the RT625 requires using a Load or Store Alternate instruction with ASI = 04 H in addition to the register address, given in *Table 4–14*. *Table 4–15* illustrates the ASI mapping for the RT625.

Table 4–14. RT625 Register Address Mapping

VA (15:8)	RT625 Registers
00 H	System Control Register (SCR)
01 H	Context Table Pointer Register (CTPR)
02 H	Context Register (CXR)
03 H	Synchronous Fault Status Register (SFSR)
04 H	Synchronous Fault Address Register (SFAR)
05 H	Asynchronous Fault Status Register (AFSR)
06 H	Asynchronous Fault Address Register (AFAR)
07 H	Reset Register (RR)
08–0F H	Reserved
10 H	Root Pointer Register (RPR)
11 H	Instruction Access PTP (IPTP0)
12 H	Data Access PTP (DPTP0)
13 H	Index Tag Register (ITR0)
14 H	TLB Replacement Control Register (TRCR)
15 H	Instruction Access PTP (IPTP1)
16 H	Data Access PTP (DPTP1)
17 H	Index Tag Register (ITR1)
18–FF H	Reserved

Table 4–15. Standard SPARC ASI Assignments

ASI	Function	ASI	Function
0 H	Reserved	14 H	Flush Combined Cache Line (user)*
1 H	Reserved	15 H	Reserved
2 H	Reserved	16 H	Reserved
3 H	MMU Invalidate/Probe*	17 H	Block Copy*
4 H	MMU Registers*	18 H	Flush RT620 Instruction Cache Cache Line (page)
5 H	MMU Diagnostics Instruction only TLB	19 H	Flush RT620 Instruction Cache Cache Line (segment)
6 H	MMU Diagnostics Instruction/Data TLB*	1A H	Flush RT620 Instruction Cache Cache Line (region)
7 H	MMU Diagnostics I/O TLB	1B H	Flush RT620 Instruction Cache Cache Line (context)
8 H	User Instruction*	1C H	Flush RT620 Instruction Cache Cache Line (user)
9 H	Supervisor Instruction*	1D H	Reserved
A H	User Data*	1E H	Reserved
B H	Supervisor Data*	1F H	Block Fill*
C H	Cache Tag for Instruction Cache	20–2F H	MMU Bypass Physical Address*
D H	Cache Data for Instruction Cache	30 H	Unassigned
E H	Cache Tag Combined (instruction/data) Cache (CTAG)*	31 H	Flush Entire CY7C260 Instruction Cache
F H	Cache Data for Combined Cache*	32–3F H	Unassigned
10 H	Flush Combined Cache Line (page)*	40–6F H	Reserved
11 H	Flush Combined Cache Line (segment)*	70–7F H	Unassigned
12 H	Flush Combined Cache Line (region)*	80–FF H	Reserved
13 H	Flush Combined Cache Line (context)*		

*indicates functions supported by the RT625

4.10 Synchronous Faults

Synchronous faults are grouped into three classes: instruction access faults, data access faults, and translation table access faults. The translation table access faults are further divided into translation instruction access faults and translation data access faults. The SPARC architecture causes the timing and priority of these fault classes to be handled differently. Due to delays caused by the instruction pipeline, the RT620 can possibly encounter a second fault before the RT620 enters a trap to correct the first. Depending upon the class of fault encountered, the status and address of a fault may be allowed to overwrite information for a previous fault that has not yet generated a trap. This potential condition requires a trap handler that can correct the various combinations of fault conditions. This section describes these potential fault conditions.

The case of multiple faults occurring presents a problem in reporting the correct fault status. This problem is solved by use of an overwrite (OW) bit in the SFSR and by prioritizing which types of faults may overwrite a previous fault. The OW bit signals the trap handler that the status and address stored in the fault registers are not valid for the trap that the RT620 has entered. The SFSR logic sets the OW bit according to a state sequence based on the fault handling of the RT620 and the type of faults encountered.

Since the RT620 delays entering a trap handler for an instruction fault, a trap caused by another fault will overwrite the trap information for the initial instruction fault. If the initial instruction trap is entered before the second fault trap is entered, the OW bit will be set. This is because the first trap reading the fault status registers will have the fault data for the second trap. The OW bit is set only if the status information stored in the SFSR does not correspond to the trap that will be executed first by the RT620. The setting of the OW bit is entirely based upon the types of faults and their order of occurrence. *Table 4–17* illustrates the possible fault cases and their effect on OW.

The RT620 delays a trap caused by an instruction access fault until that instruction reaches the Writeback stage. However, since data accesses are not pipelined, the RT620 jumps to a trap immediately upon encountering a data access fault.

Faults are allowed to overwrite another fault status depending upon priority. An instruction fault is allowed to overwrite only another instruction fault. It is not allowed to overwrite either a data fault or a translation fault. Data faults may overwrite an instruction fault, but not a translation fault. Data faults cannot overwrite another data fault. Translation faults may overwrite any other type of fault, but cannot be overwritten. A translation fault cannot overwrite another translation fault.

All multiple fault cases are recoverable by re-executing the instruction or access that caused the fault whose status has been overwritten. If an instruction access fault occurs and the OW bit is set, the system software must determine the cause by probing the MMU and/or memory.

Table 4–16. SPARC Fault Cases

for Read Access	Before First Data	After First Data
Error *Retry* or R&R	Synchronous Fault Access Will *Retry*	Nothing Asynchronous Fault
For Write/LDST	**Before First Data**	**After First Data**
Error *Retry* or R&R	Synchronous Fault Access Will *Retry*	Synchronous Fault Synchronous Fault

Table 4–17. OW Bit States

Previous Fault	Current Fault	Update SFSR	OW
none	instruction	yes	0
none	data	yes	0
none	translate instruction	yes	0
none	translate data	yes	0
instruction	instruction	yes	1
instruction	data	yes	0
instruction	translate instruction	yes	1
instruction	translate data	yes	0
data	instruction	no	0
data	data	no	0
data	translate instruction	yes	1
data	translate data	yes	1
translate instruction	instruction	no	0
translate instruction	data	no	0
translate instruction	translate instruction	no	0
translate instruction	translate data	no	0
translate data	instruction	no	0
translate data	data	no	0
translate data	translate instruction	no	0
translate data	translate data	no	0

Table 4–18. Fault Register Level Field

L	Level
0	Entry in Context Field
1	Entry in Level 1 Table
2	Entry in Level 2 Table
3	Entry in Level 3 Table

Table 4–19. Fault Register Access Type Field

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

Upon encountering a synchronous fault, the SFSR records the Bus Error status (System Error, Bus Error, timeout, and Uncorrectable Error) when a Bus Error occurs during memory accesses. The Access Type (AT) field, illustrated in *Table 4–19*, defines the type of access that caused the fault. The fault type field FT (see *Table 4–20*) defines the type of the current fault.

Table 4–20. Fault Register Fault Type Field

FT	Fault Type
0	None
1	Invalid Address Error
2	Protection Error
3	Privilege Violation Error
4	Translation Error
5	Bus Access Error
6	Not Generated
7	Reserved

A translation table access fault (FT = 4) occurs if an MMU page table access causes an external System Error. This also occurs if a reserved entry type (ET = 3 in the PTE) is found in any level of the table walk. A translation table access fault also can occur if a page table pointer (PTP) is found in Level 3, instead of a PTE. If the page table entry is invalid (ET = 0 in the PTE), the fault type is an invalid address error (FT = 1). *Table 4–21* illustrates the fault type (FT) assigned for valid TLB entries or PTE entries (ET = 2) that cause a fault condition. These fault conditions are either a protection error (read/write of data or instruction) or a privilege violation (user/supervisor access) error.

Table 4–21. Fault Type (FT) for PTE[ET] = 2

	ACC							
AT	0	1	2	3	4	5	6	7
0	0	0	0	0	2	0	3	3
1	0	0	0	0	2	0	0	0
2	2	2	0	0	0	2	3	3
3	2	2	0	0	0	2	0	0
4	2	0	2	0	2	2	3	3
5	2	0	2	0	2	0	2	0
6	2	2	2	0	2	2	3	3
7	2	2	2	0	2	2	2	0

The fault address valid bit (FAV) is set to one if the content of the synchronous fault address register is valid.

If multiple fault types apply to the same fault occurrence, the highest priority fault is recorded. The highest fault priority is a translation fault (priority 2), as shown in *Table 4–22*. Priority 1 is reserved for an internal fault.

Table 4–22. Fault Register Error Priorities

Priority	Error
1	Internal Error
2	Translation Error
3	Invalid Address Error
4	Privilege Violation Error
5	Protection Error
6	Bus Access Error

Upon power-on reset, the SE, UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.

4.10.1 Synchronous Fault Cases

The following 20 cases describe the combinations of fault cases that can occur:

Case 1: *Instruction fault.* The RT620 trap is delayed until the RT620 tries to execute the instruction.

The trap is taken immediately if the instruction access is actually a data access that is interpreted by the RT625 as an instruction access due to asserting ASI = 8 or 9 with a Load Alternate instruction. In this case, the trap handlers cannot probe main memory using the PC of the instruction. If the instruction is a Load Alternate instruction, the trap handler has to calculate the effective address to probe. The SFAR has the valid address in this case.

Case 1: Instruction Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus access error occurred (external Bus Error: UC or TO or BE is set)
AT	2, 3	Load/Execute from User/Supervisor instruction space
L	0, 1, 2, 3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 2: *Data fault.* The RT620 trap is taken immediately.

Case 2: Data Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus Error occurred (external Bus Error: UC or TO or BE is set)
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 3: *Translation fault on instruction access.* The RT620 trap is delayed until the RT620 tries to execute the instruction, or is taken immediately if the access is data due to a Load Alternate instruction.

Case 3: Translation Fault on Instruction Access		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	Translation error occurred (Bus Error or ET = 3 or PTP in Level 3 during table walk)
AT	2, 3	Load/Execute from User/Supervisor instruction space
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 4: *Translation fault on data access.* The RT620 trap is taken immediately.

Case 4: Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	Translation error occurred (Bus Error or ET = 3 or PTP in Level 3 during table walk)
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 5: *Instruction fault followed by instruction fault.* The RT620 traps on the first instruction fault.

If the second instruction fault is due to a load access with ASI 8,9 (Load Alternate), it overwrites the fault associated information of the first fault. In this case the SFAR has a valid address for the data access of the Load Alternate instruction.

The fault address of the first fault can be obtained from the PC in the RT620 for the trap handler with the exception of the following case.

It is possible that a data access is interpreted by the RT625 as an instruction access because of the use of a Load or Store Alternate instruction with ASI = 8, 9. Before the RT620 takes the trap on the data access fault (which is recorded as an instruction fault in the RT625), another instruction fault may occur. The second instruction will overwrite the data access fault information, because it is recorded as an instruction fault in the RT625. In this case, the PC of the instruction (for the first fault) is not the faulted address; also, the SFAR does not contain the fault address of the first fault and the trap handler has to calculate the effective address to probe.

Case 5: Instruction Fault followed by Instruction Fault		
OW	1	
FAV	1	SFAR has valid address for second instruction fault
FT	1, 2, 3, 5	Fault type of second fault
AT	2, 3	Access Type of second fault
L	0, 1, 2, 3	Level at which second fault occurred during table walk (only valid with FT = 1)

Case 6: *Instruction fault followed by data fault.* The RT620 traps on the data fault.

The history of the instruction fault is lost, but the same fault can be obtained again, once the return from the trap handler of the data fault is completed.

Case 6: Instruction Fault then Data Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1, 2, 3, 5	Fault type of data fault
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 7: *Instruction fault followed by translation fault on instruction access.* The RT620 traps on the instruction fault.

The fault address of the instruction fault can be obtained from the PC in the RT620 for the trap handler with the exception of the following case.

A data access fault can be recorded as an instruction fault if a Load Alternate instruction with ASI = 8, 9 causes a fault. Before the RT620 takes the trap on the data access fault (which is recorded as an instruction fault in the RT625), a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 7: Instruction Fault then Translation Fault on Instruction Access		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2, 3	Load/Execute from User/Supervisor instruction space
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 8: *Instruction fault followed by translation fault on data access.* The RT620 will trap on the data fault.

Case 8: Instruction Fault then Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 9: *Data fault followed by instruction fault.* The instruction fault cannot overwrite the data fault. The instruction fault will occur again, once the return from the data fault trap handler is completed. The RT620 will trap on the data fault.

Case 9: Data Fault then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1, 2, 3, 5	Fault type of data fault
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 10: *Data fault followed by data fault.* The information for the first data fault is saved.

Case 10: Data Fault then Data Fault		
OW	0	
FAV	1	SFAR has valid address for first data fault
FT	1, 2, 3, 5	Fault type of first data fault
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which first data fault occurred during table walk (only valid with FT = 1)

Case 11: *Data fault followed by translation fault on instruction access.* The RT620 traps on the data fault.

Before the RT620 takes the trap on the data access fault, a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 11: Data Fault then Translation Fault on Instruction Access		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2, 3	Execute from User/Supervisor instruction space
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 12: *Data fault followed by translation fault on data access.* The RT620 traps on the data fault.

Before the RT620 takes the trap on the data access fault, a translation fault may occur due to a data access. This will overwrite the data access fault information.

Case 12: Data Fault then Translation Fault on Data Access		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 13: *Translation fault on instruction access followed by instruction fault.* The RT620 traps on the translation fault. The instruction fault cannot overwrite the translation fault.

Case 13: Translation Fault on Instruction Access then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2, 3	Load/Execute from User/Supervisor instruction space
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 14: *Translation fault on instruction access followed by data fault.* The RT620 traps on the data fault. The data fault cannot overwrite the translation fault.

Case 14: Translation Fault on Instruction Access then Data Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2, 3	Execute from User/supervisor instruction space
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 15: *Translation fault on instruction access followed by translation fault on instruction access.* The RT620 traps on first translation fault. The second translation fault cannot overwrite the first translation fault.

Case 15: Translation Fault on Instruction Access then Translation Fault on Instruction Access		
OW	0	
FAV	1	SFAR has valid address for first translation fault
FT	4	
AT	2, 3	Load/Execute from User/supervisor instruction space
L	0, 1, 2, 3	Level at which first translation fault occurred during table walk

Case 16: *Translation fault on instruction access followed by translation fault on data access.* The RT620 traps on the second translation fault. The second translation fault does not overwrite the first translation fault.

Case 16: Translation Fault on Instruction Access then Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for first translation fault
FT	4	
AT	2, 3	Execute from User/supervisor instruction space
L	0, 1, 2, 3	Level at which first translation fault occurred during table walk

Case 17: *Translation fault on data access followed by instruction fault.* The RT620 will trap on the translation fault.

Case 17: Translation Fault on Data Access then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 18: *Translation fault on data access followed by data fault.* The RT620 will trap on the translation fault.

Case 18: Translation Fault on Data Access then Data Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 19: *Translation fault on data access followed by translation fault on instruction access.* The RT620 will trap on the first translation fault.

Case 19: Translation Fault on Data Access then Translation Fault on Instruction Access		
OW	0	
FAV	1	SFAR has valid address for data translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

Case 20: *Translation fault on data access followed by translation fault on data access.* The RT620 will trap on the first translation fault.

Case 20: Translation Fault on Data Access then Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for first data translation fault
FT	4	
AT	0, 1, 4, 5, 6, 7	
L	0, 1, 2, 3	Level at which translation fault occurred during table walk

4.11 RT625 Pinouts

4.11.1 Pin Description

CLKMODE — (input) Clock MODE

During reset

0 = bypass mode

1 = normal synchronizer mode

After reset if bypass mode is selected

0 = synchronize at this IMCLK edge

1 = do not synchronize at this IMCLK edge

There are two simple modes. If the clocks are asynchronous, then the CLKMODE pin is tied to V_{CC} and the synchronizers are used. If the clocks are the same frequency, then they must be guaranteed out of phase and the CLKMODE pin is tied to ground. This causes the synchronizers to be bypassed and every IMCLK edge is available for synchronizing.

IMCLK — (input) Intra-Module Bus (IMB) clock

This is the basic clock for all the Intra-Module components. All the Intra-Module signals are driven and sampled on only the rising edge of the IMCLK. The RT625 uses the IMCLK clock to interface with CPU and CDU units.

IMA < 31:18 > — (input) Intra-Module Address Bus.

This is part of the virtual address sent out by the RT620 during a fetch or Load/Store operation. This address is latched by the RT625.

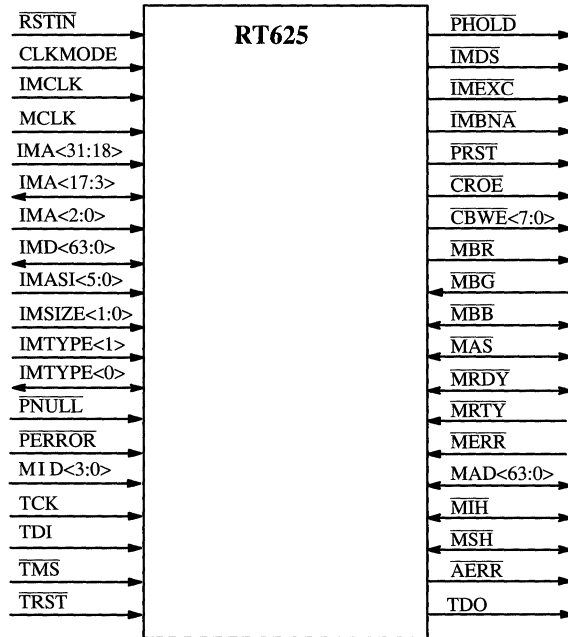


Figure 4–42. RT625 Pinout

IMA < 17:3 > — (input/output, tri-state) Intra-Module Address Bus

This bus acts as input when the RT625 is latching an address from the RT620. The RT625 puts the cache data address on this bus when it needs to write data into the cache data unit.

IMA < 2:0 > — (input) Intra-Module Address Bus

This is part of the virtual address sent out by the RT620 during a fetch or Load/Store operation and is latched by the RT625.

IMD < 63:0 > — (input/output, tri-state) Intra-Module Data Bus

These pins form a 64-bit bi-directional data bus that carries data between the RT625 and the other IMB components.

IMASI < 5:0 > — (input) Intra-Module Address Space Identifier

These 6 bits constitute the Address Space Identifier (ASI). The ASI identifies the memory address space to which the instruction or data access is being directed. The IMASI bits are sent out unlatched along with the address by the RT620 and are latched by the RT625.

IMSIZE < 1:0 > — (input) Intra-Module Access Size

These two pins indicate the SIZE of the current access. Instruction accesses are always doubleword size. Because the data bus is 64-bits wide, doubleword accesses can be performed in a single access.

The value of the size bits during a given cycle relates only to the address which appears on pins IMA < 31:0 >. The IMSIZE < 1:0 > bits are latched by the RT625. Size values are defined as follows:

IMSIZE < 1:0 >	Size
00	Byte
01	Half Word
10	Word
11	Double Word

The size bits for the different bus transactions are given below.

Bus Activity	IMSIZE <1:0 >
Instruction fetch	11
Load/Store double	11
Load/Store word	10
Load/Store halfword	01
Load/Store byte	00

IMTYPE < 1 > — (input) Intra-Module Access Type

IMTYPE < 0 > — (input/output, tri-state) Intra-Module Access Type

These two pins indicate the current access TYPE. Instruction accesses are always type read. The value of the type bits during a given cycle relates only to the address which appears on pins IMA < 31:0 >. The IMTYPE bits are latched by the RT625. Type values are defined as follows.

IMTYPE<1 >	IMTYPE <0 >	Meaning
0	0	Normal write (Store)
0	1	Normal read (Load or instruction fetch)
1	0	Locked write (Atomic Store)
1	1	Locked read (Atomic Load)

PHOLD — (output) Processor Hold

This signal is used by the RT625 to hold the processor.

IMDS — (output) Intra-Module Data Strobe

This signal is used by the RT625 to strobe the data into the CPU when the valid data becomes available.

IMEXC — (output) Intra-Module Exception

This signal is used by the RT625 to indicate a synchronous exception condition to the CPU.

IMBNA — (output) Intra-Module Bus Not Available

This pin is used by the RT625 to indicate to the CPU that the RT625 is using the Intra-Module Bus. This allows data in a missed cache line to be filled as a background task while the processor continues executing instructions from the on-chip instruction cache. The processor can resume instruction execution once the RT625 has provided the data which caused the processor hold.

PNULL — (input) Processor Nullify Signal

The processor PNULL asserts to indicate that the current external cache access is being nullified. This supports highly pipelined accesses on the Intra-Module Bus and allows access nullification at a later stage. Asserting PNULL nullifies the access already latched by the external cache subsystem. For example, in the case of load data access which gets an external cache miss, the RT620 asserts PNULL for one cycle due to data forwarding.

PERROR — (input) Processor Error

This pin is asserted when the processor is in the ERROR mode. The only way to restart a processor that is in the error mode state is to trigger a reset by asserting the PRST signal.

PRST — (output) Processor Reset

Assertion of this pin will reset the processor. PRST must be asserted for a minimum of sixteen processor clock cycles.

CROE — (output) Cache RAM Output Enable.

Assertion of this pin enables the output buffers of the cache RAMs (CDUs).

CBWE < 7:0 > — (output) Cache RAM Byte Write Enables.

The RT625 enables these signals appropriately to enable writes into the cache RAMs (CDUs).

MCLK — (input) MBus Clock.

All the MBus signals are driven and sampled on only the rising edge of the MCLK. The RT625 uses the MCLK clock to interface with the rest of the components on the MBus.

MAD < 63:0 > — (input/output, tri-state) MBus Address Data Bus.

During the address phase, MAD < 35:0 > contains the physical address. The remaining signals contain the transaction specific information. During the data phase, MAD < 63:0 > contains the data of the transfer.

MAS — (input/output, tri-state) MBus Address Strobe.

This signal is asserted by the bus master during the very first cycle of a bus transaction (the "address phase" or the "address cycle").

MRDY — (input/output, tri-state) MBus Ready Strobe.

This signal is one of three bits used to encode the transaction status. The encoding with only MRDY low indicates that valid data has been transferred.

MRTY — (input) MBus Retry Strobe.

This signal is one of three bits used to encode the transaction status. The encoding with only MRTY low indicates that the slave wants the master to abort the current transaction immediately and start over.

MERR — (input) MBus Error Strobe.

This signal is one of three used to encode the transaction status as shown in *Table 4–23*. The encoding with only MERR low indicates that a Bus Error (or other system implementation specific) error occurred.

Table 4–23. Transaction Status Bit Encoding

MER	MRDY	MRTY	Action
H	H	H	Idle Cycle
H	H	L	*Relinquish and Retry*
H	L	H	Data Strobe
H	L	L	Undefined L1, Reserved L2
L	H	H	ERROR1 → Bus Error
L	H	L	ERROR2 → Time Out
L	L	H	ERROR3 → Uncorrectable
L	L	L	*Retry*

MBR —(output) MBus Request.

This signal is asserted by a MBus master to acquire bus ownership. There is one unique MBR signal per master.

MBG — (input) MBus Grant.

This signal is asserted by the external arbiter when the particular MBus master is granted the bus. There is one unique MBG signal per master.

MBB — (input/output, tri-state) MBus Busy.

This signal is asserted as an output during the entire transaction, from and including the assertion of MAS, to the assertion of the last MRDY or first other acknowledgement which terminates the transaction. The potential bus master samples this signal in order to acquire bus ownership as soon as the current master releases the bus.

RSTIN — (input) Reset Input.

This signal should reset all logic on a module to its initial state and ensure that all MBus signals are inactive or tri-state as appropriate.

AERR — (output, open drain) Asynchronous Error.

This signal is asserted by the module as a level to indicate that an asynchronous error was detected by the module.

MIH — (input/output, tri-state) Memory Inhibit.

It is asserted by the owner of a cache block to inform main memory that the current Coherent Read or Coherent Read and Invalidate request should be ignored. This is because the owner will supply the cache block.

$\overline{\text{MSH}}$ — (input/output, open drain) MBus Shared.

This signal is asserted by an RT625 in response to a Coherent Read request of a cache block if it has a valid copy of that cache block.

$\overline{\text{MID}}$ <3:0 > — (input) Module Identifier.

These pins carry the hardwired module identifier to the module. This four bit module identification number is embedded in the MBus address phase of all MBus transactions initiated by the RT625.

The following RT625 signals support the hyperSPARC Test Access Port (TAP) interface. For more information, see the hyperSPARC TAP Interface Specification available from ROSS Technology.

TCK — (input) Test Clock.

This is nominally a free-running clock signal. The changes on TAP input signals ($\overline{\text{TMS}}$ and TDI) are clocked into the TAP controller, instruction register or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) also occur on the rising edge of TCK.

TDI — (input) Test Data Input.

TDI is clocked into the selected register (instruction or data) on a rising edge of TCK. The TDI input must have a built-in pull-up resistor of a value which ensures that an un-terminated input is seen by the test logic as a high signal level.

TDO — (output) Test Data Output.

The contents of the selected register (instruction or data) are shifted out of TDO on the falling edge of TCK. TDO drivers must be set to high-impedance except when scanning of data is in progress.

$\overline{\text{TMS}}$ — (input) Test Mode Select.

This control input is clocked into the TAP controller on the rising edge of TCK. The $\overline{\text{TMS}}$ input must have a built-in pull-up resistor of a value which ensures that an unterminated input is seen by the test logic as a high signal level.

$\overline{\text{TRST}}$ — (input) Test Reset.

$\overline{\text{TRST}}$ initializes the state of the instruction register bits and the TAP controller state machine.

4.11.2 Virtual Bus (Intra-Module Bus) Operation

The following diagrams illustrate the RT625 virtual bus operation: .... **page**

Figure 4–43. RT625 Cache Read Miss, Synchronous Clocks 4-63
Figure 4–44. RT625 Cache Read Miss, Asynchronous Clocks 4-65
Figure 4–45. Write Cache Miss, Copy-back, One Modified Block, Asynchronous Clocks 4-68
Figure 4–46. Write Cache Miss, Copy-back, No Modified Data, Synchronous Clocks 4-71
Figure 4–47. Write Through Write, Synchronous Clocks 4-73
Figure 4–48. Write Through Write, Asynchronous Clocks 4-74
Figure 4–49. RT625 Block Fill, Synchronous Clocks 4-75
Figure 4–50. RT625 Block Fill, Asynchronous Clocks 4-77
Figure 4–51. RT625 Block Copy, Synchronous Clocks 4-79
Figure 4–52. RT625 Block Copy, Asynchronous Clocks 4-82

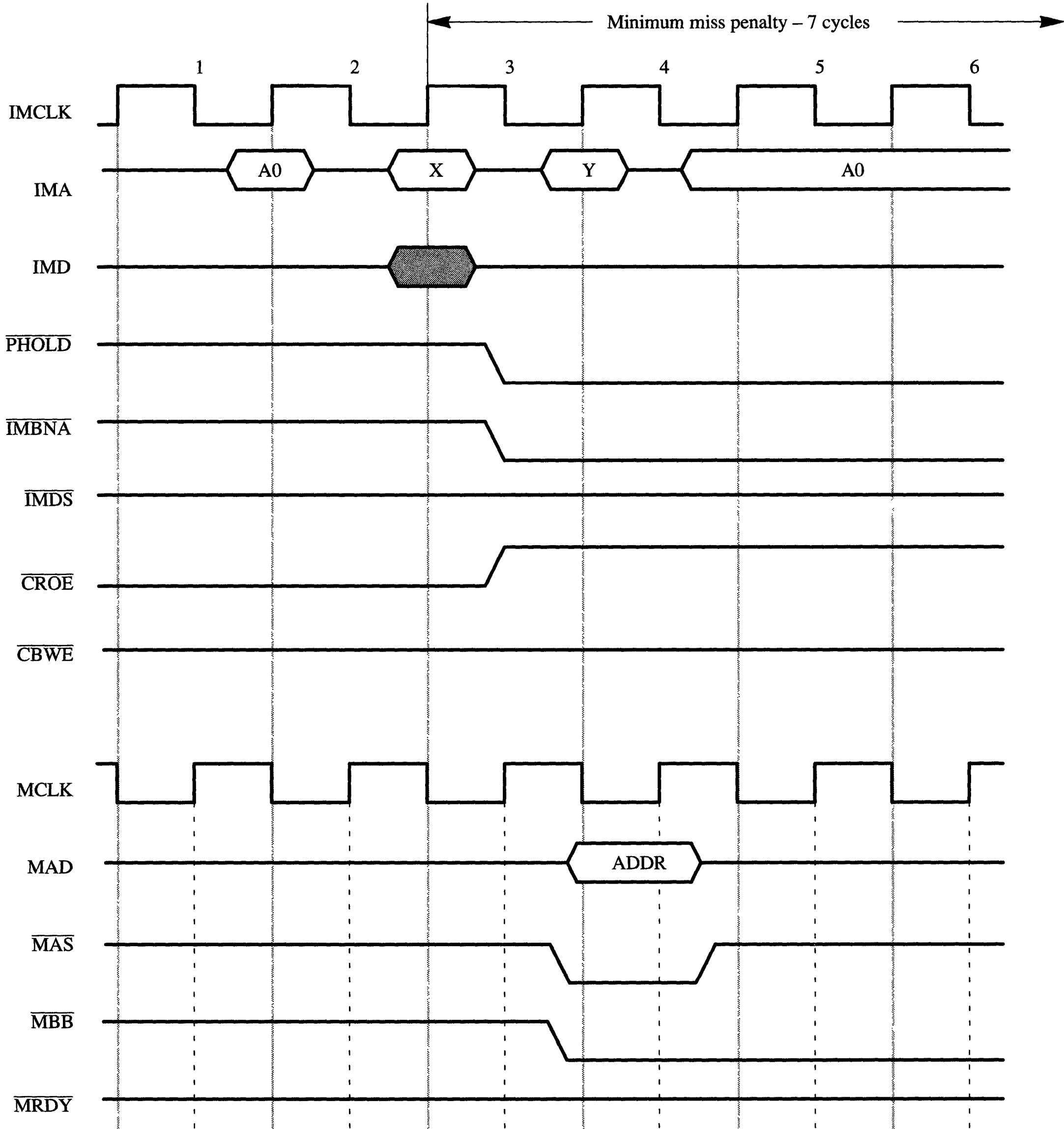


Figure 4–43. RT625 Cache Read Miss, Synchronous Clocks (page 1 of 2)

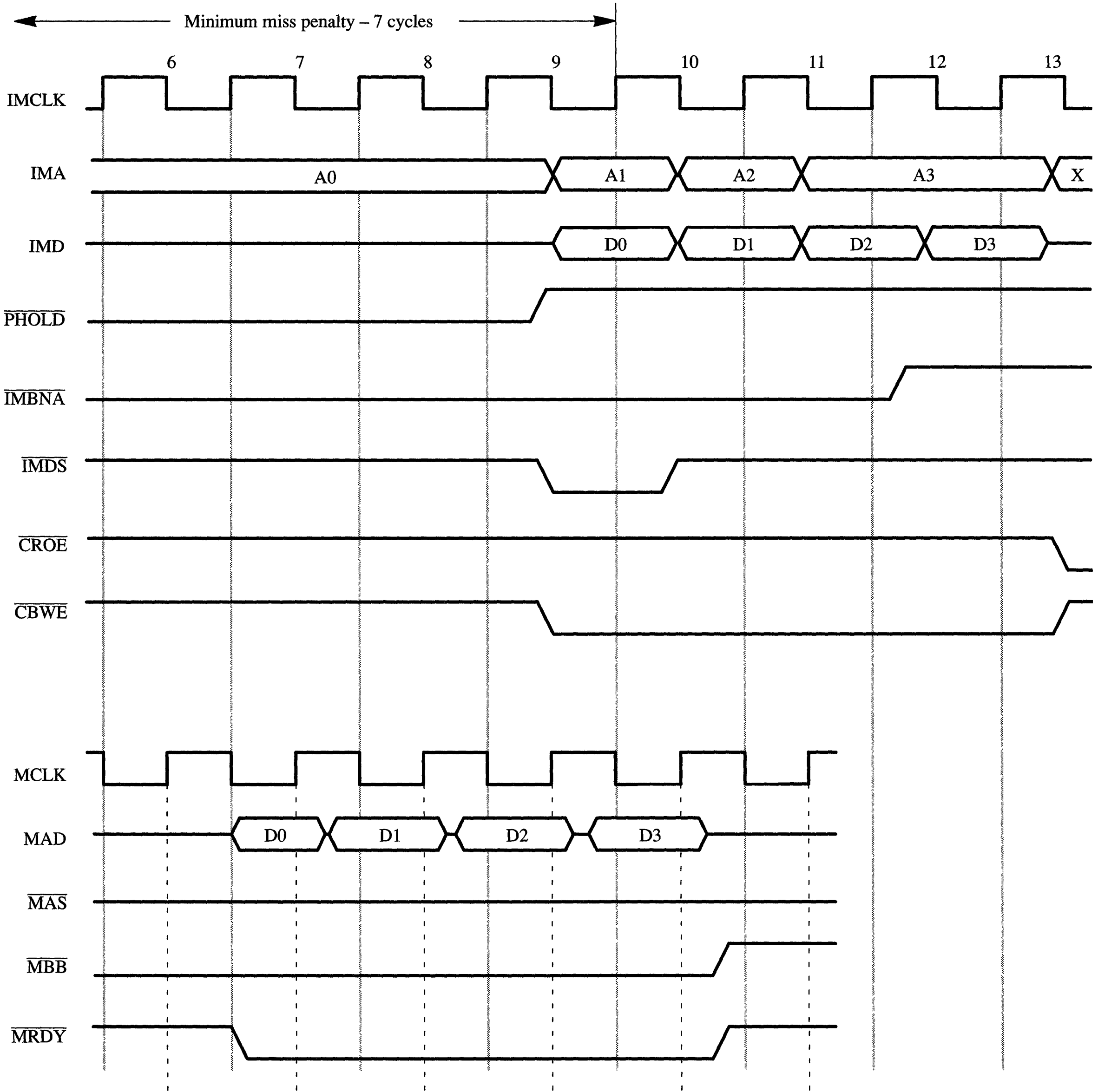
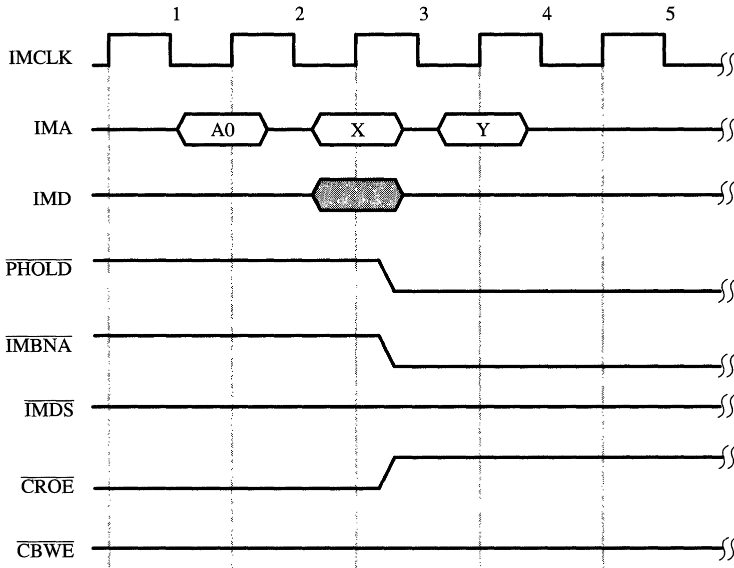


Figure 4–43. RT625 Cache Read Miss, Synchronous Clocks (page 2 of 2)



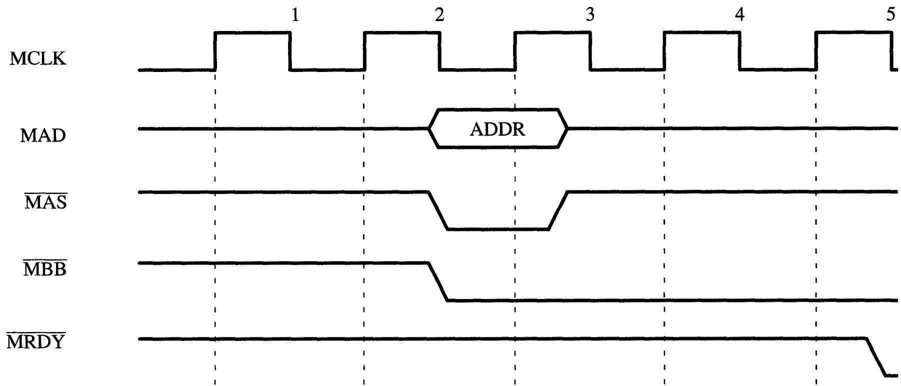


Figure 4–44. RT625 Cache Read Miss, Asynchronous Clocks* (page 1 of 3)

* This timing diagram is only an example. The sequence of writes to the CDUs may or may not have wait states depending on the ratio of IMCLK and MCLK frequencies and on the relative timing of the IMCLK and MCLK rising edges. Each double-word will be written on the fourth IMCLK rise after the data appears on MAD.

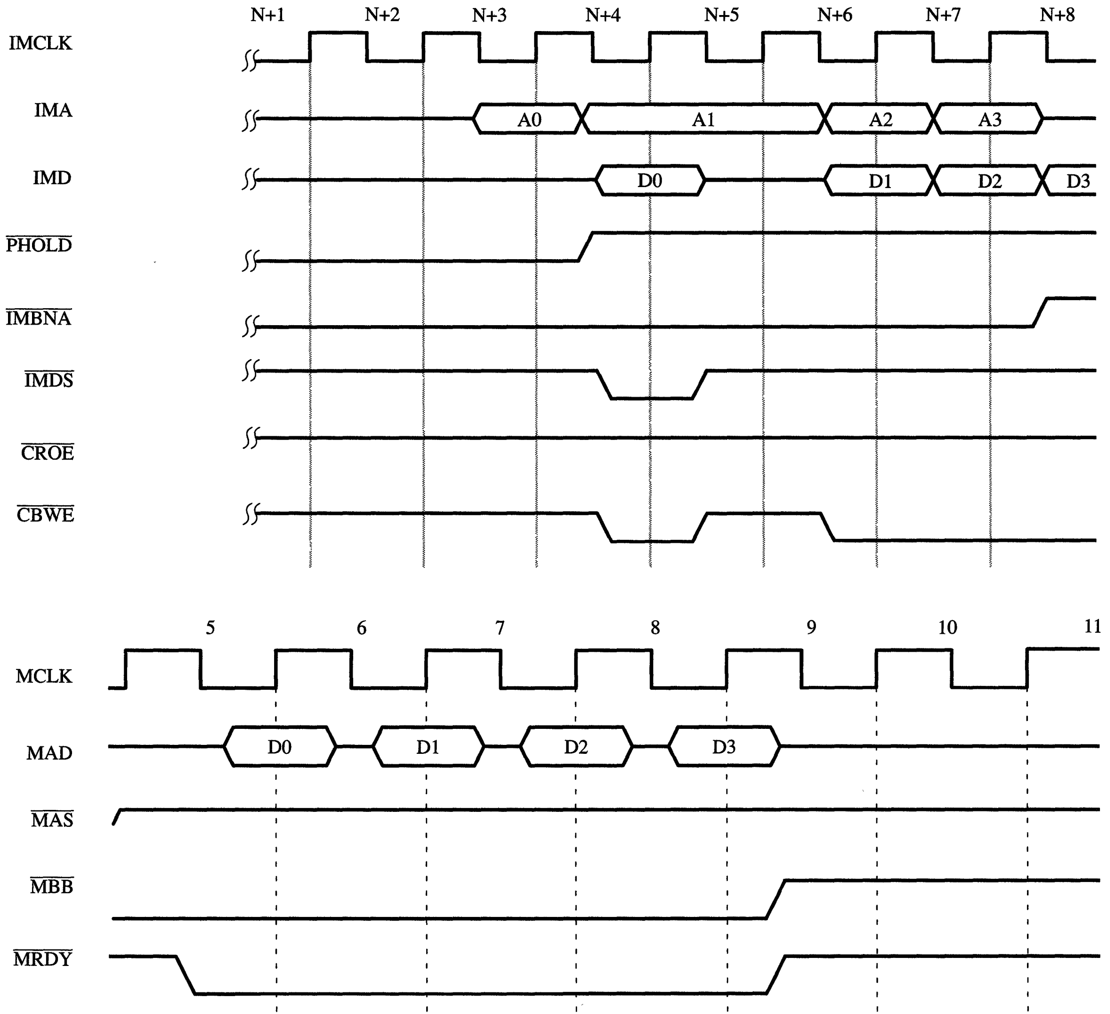


Figure 4–44. RT625 Cache Read Miss, Asynchronous Clocks (page 2 of 3)

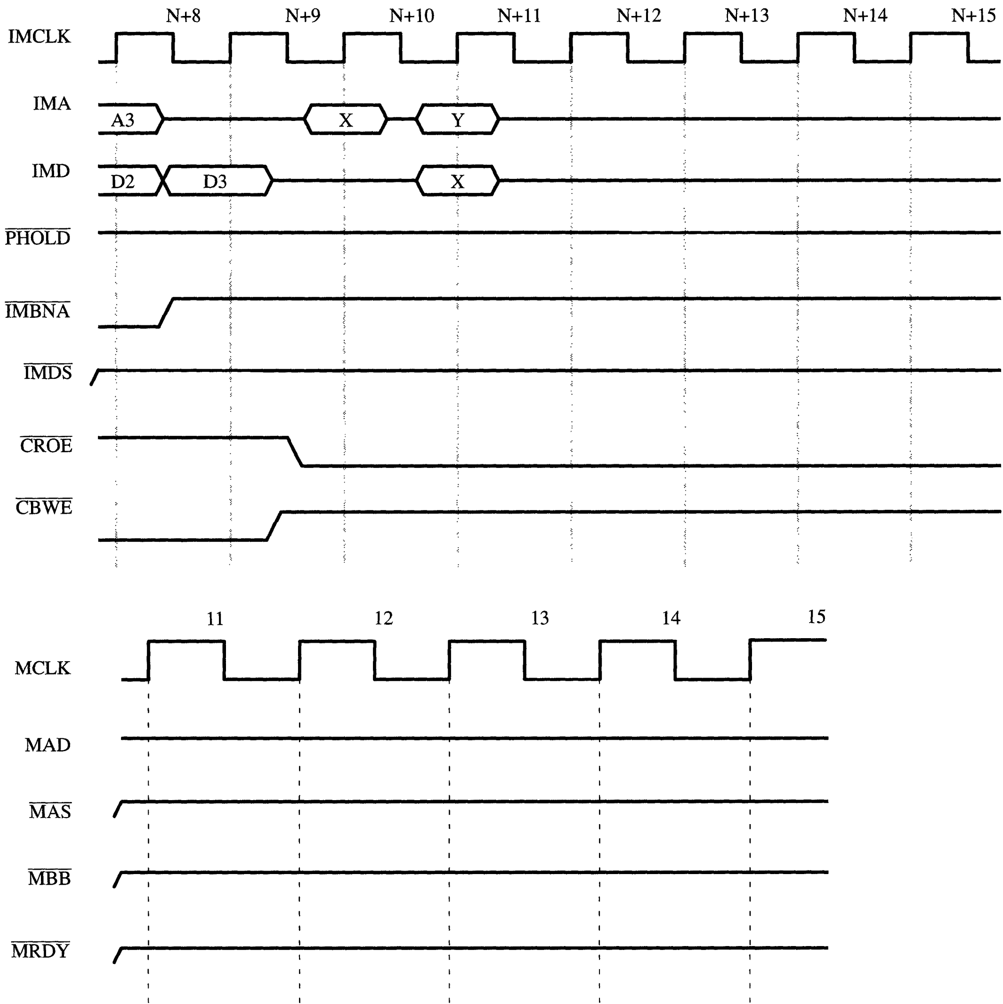


Figure 4–44. RT625 Cache Read Miss, Asynchronous Clocks (page 3 of 3)

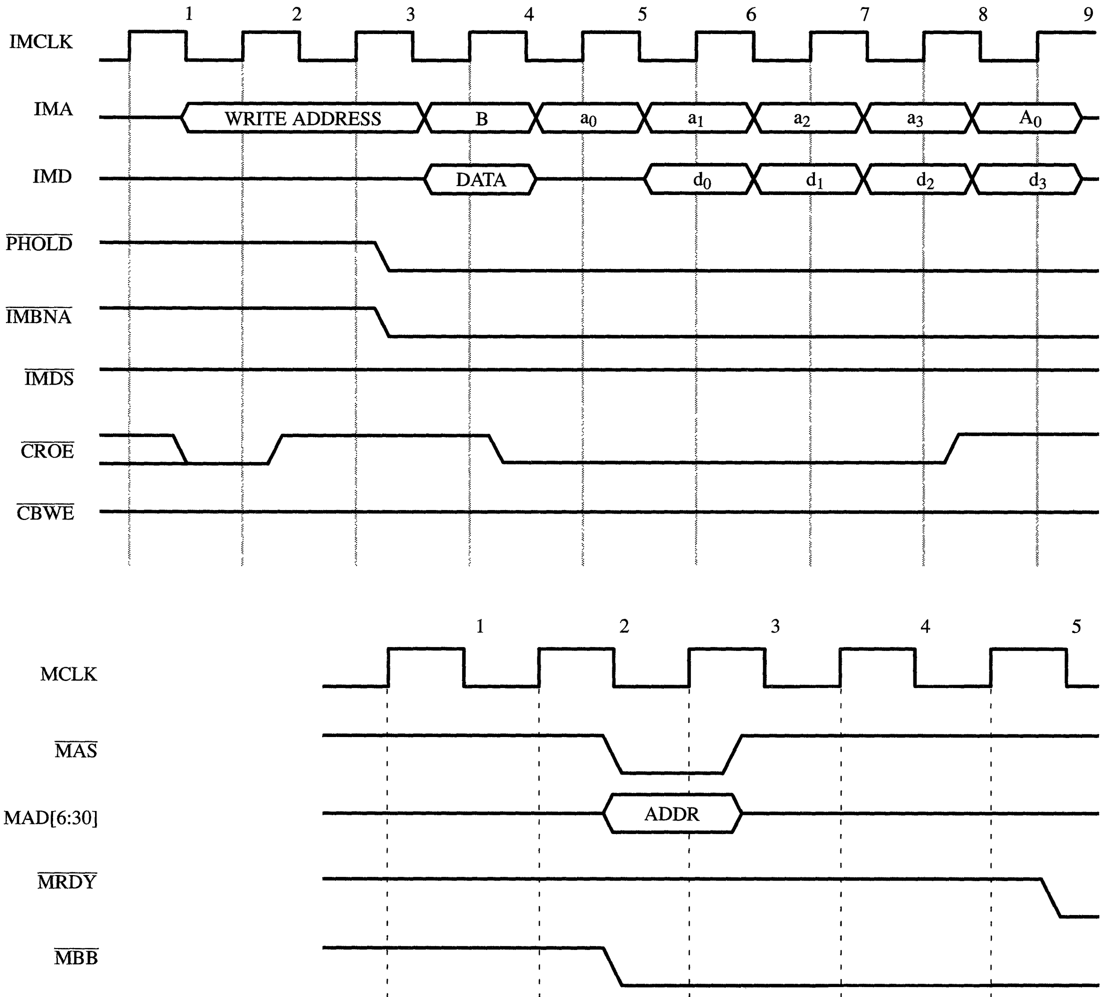


Figure 4–45. Write Cache Miss, Copy-back, One Modified Block, Asynchronous Clocks*
(page 1 of 3)

* If two sub-blocks are modified, both must be loaded into the write buffer before the new cache data can be written into the CDUs. If the memory responds quickly, the copy-back may still be finishing when the new data arrives.

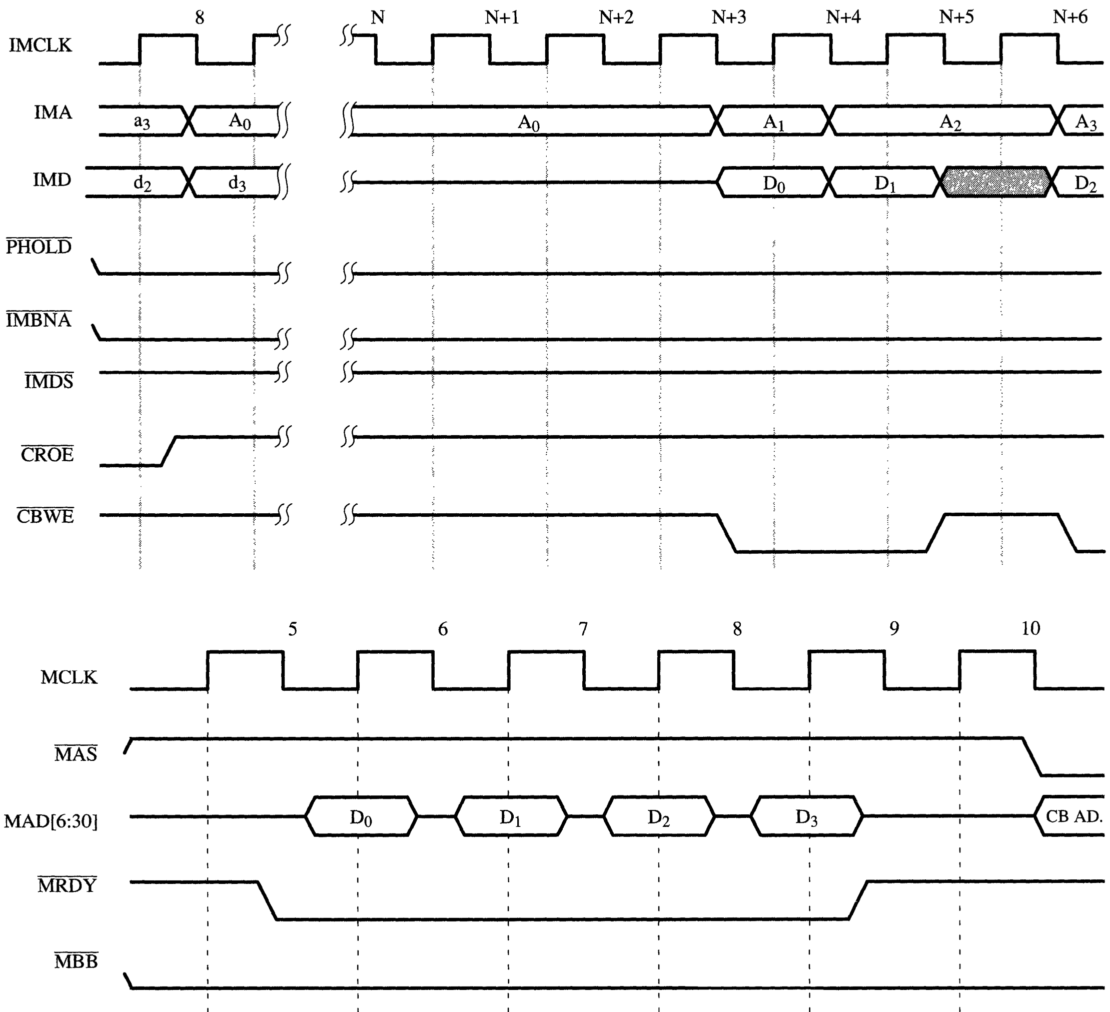
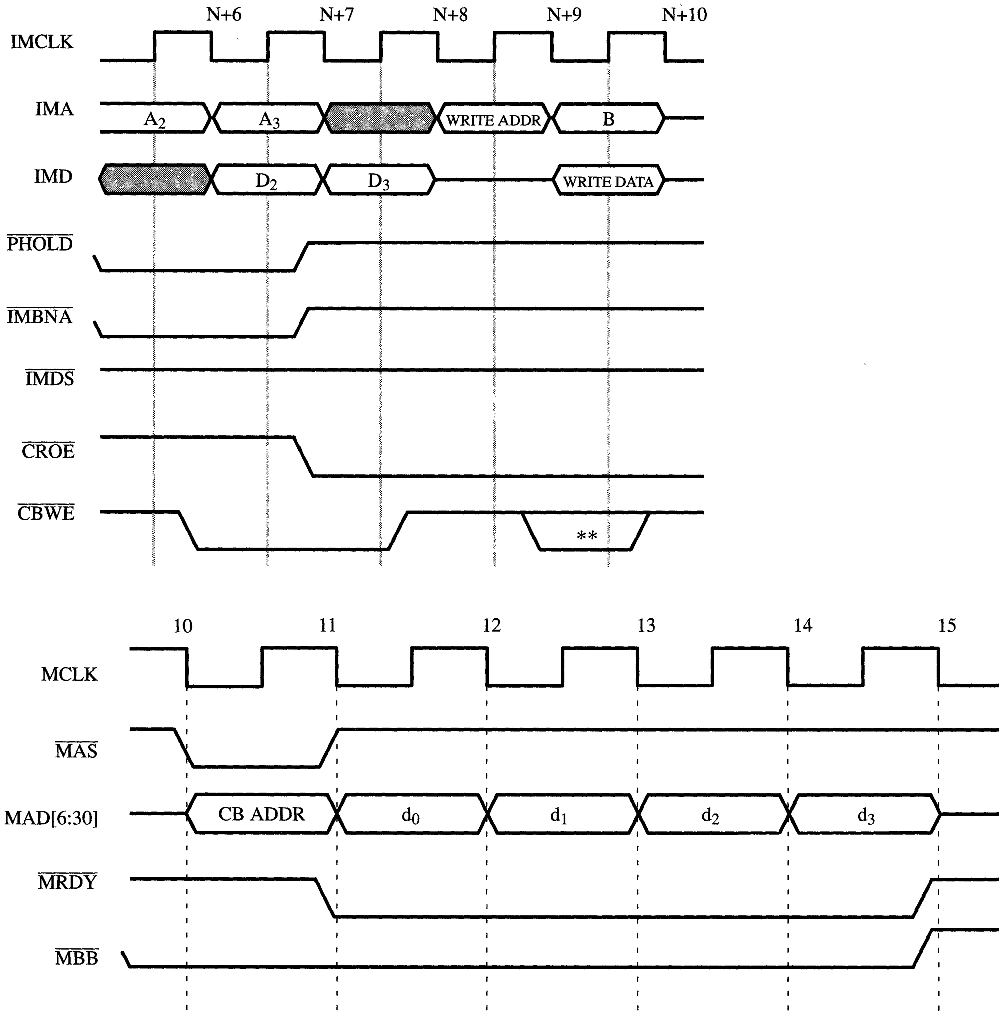


Figure 4–45. Write Cache Miss, Copy-back, One Modified Block, Asynchronous Clocks (page 2 of 3)



** CWBE is asserted here for the bytes selected by the IMA and IMSIZE pins.

Figure 4–45. Write Cache Miss, Copy-back, One Modified Block, Asynchronous Clocks (page 3 of 3)

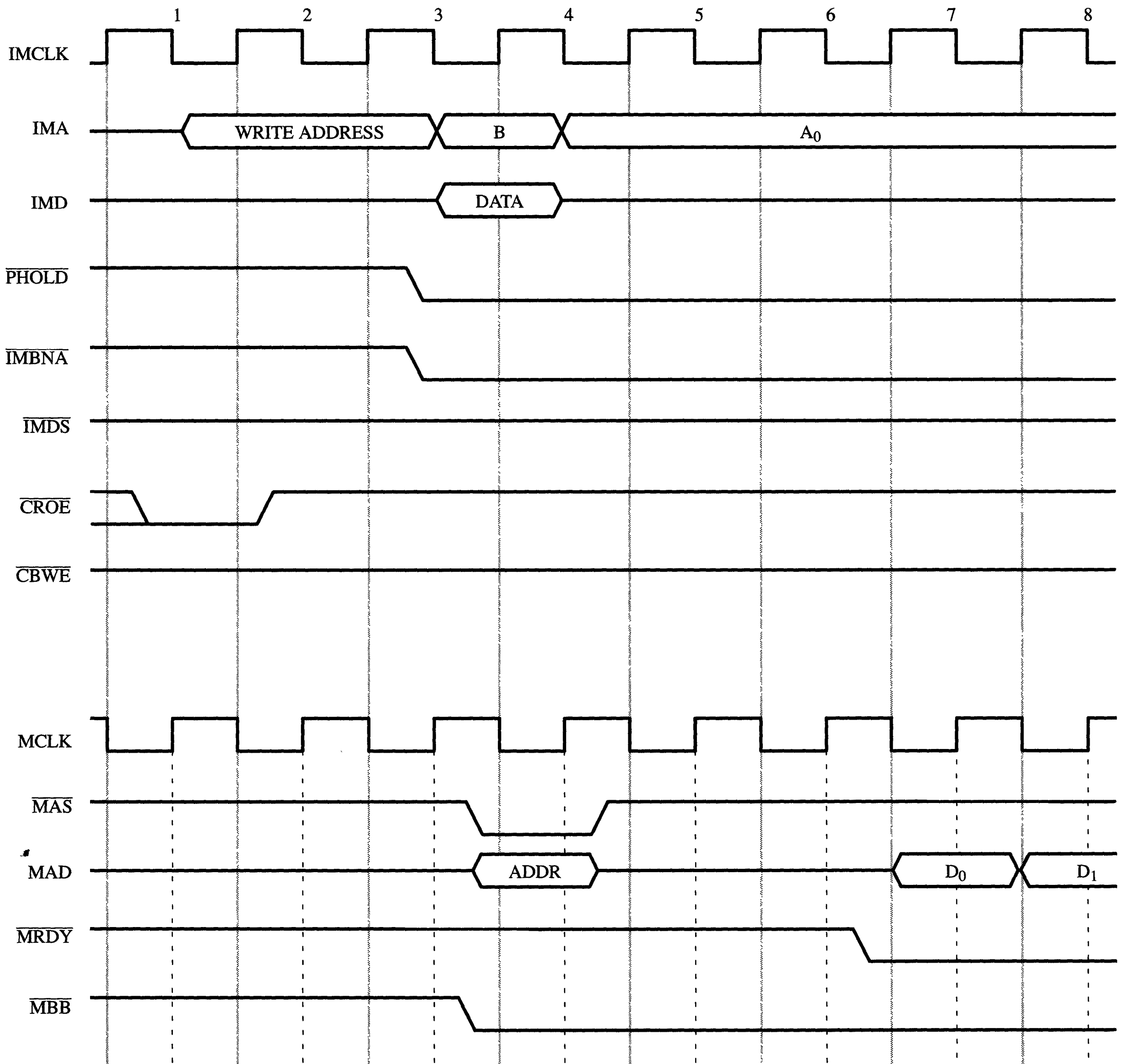
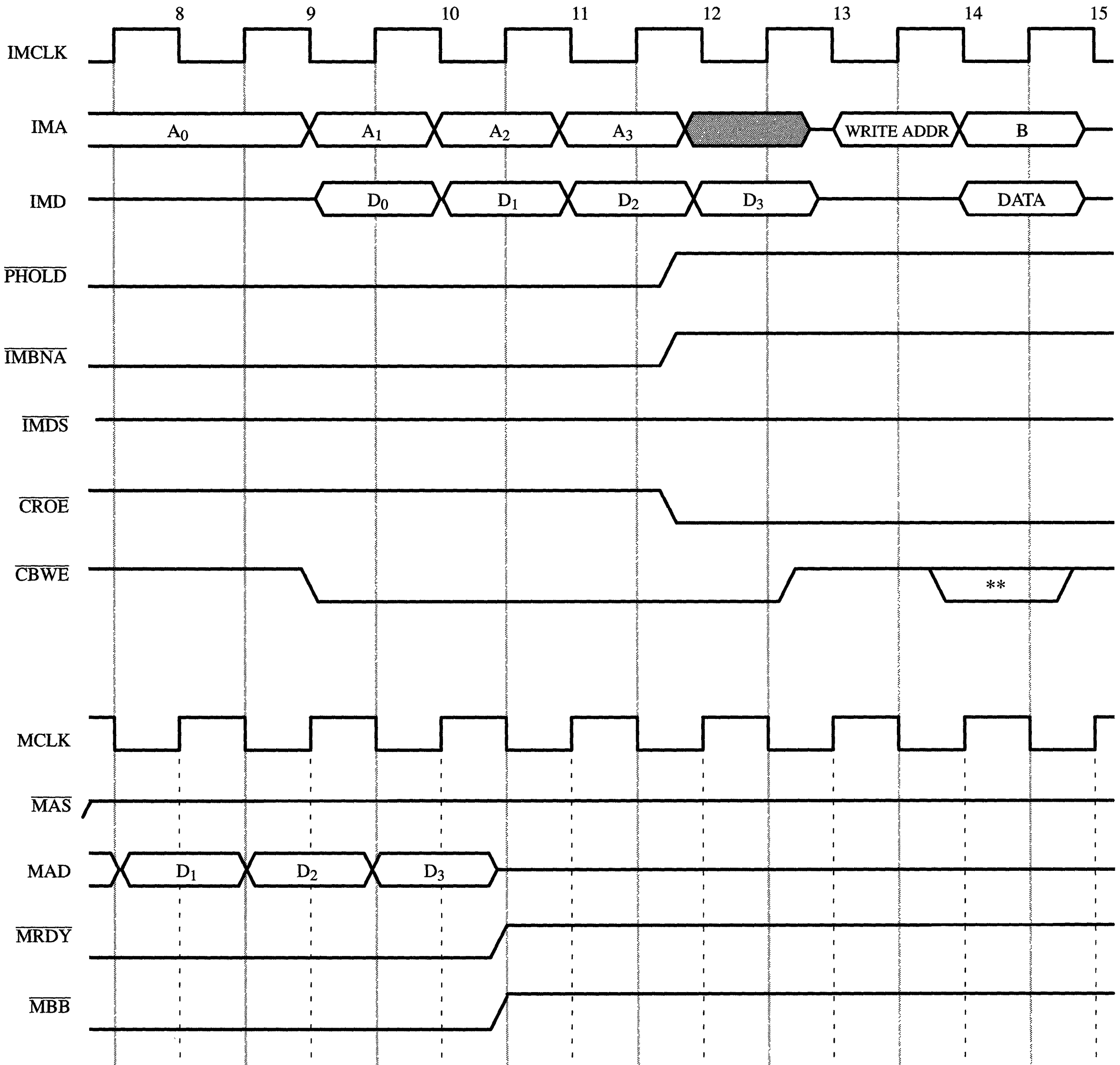
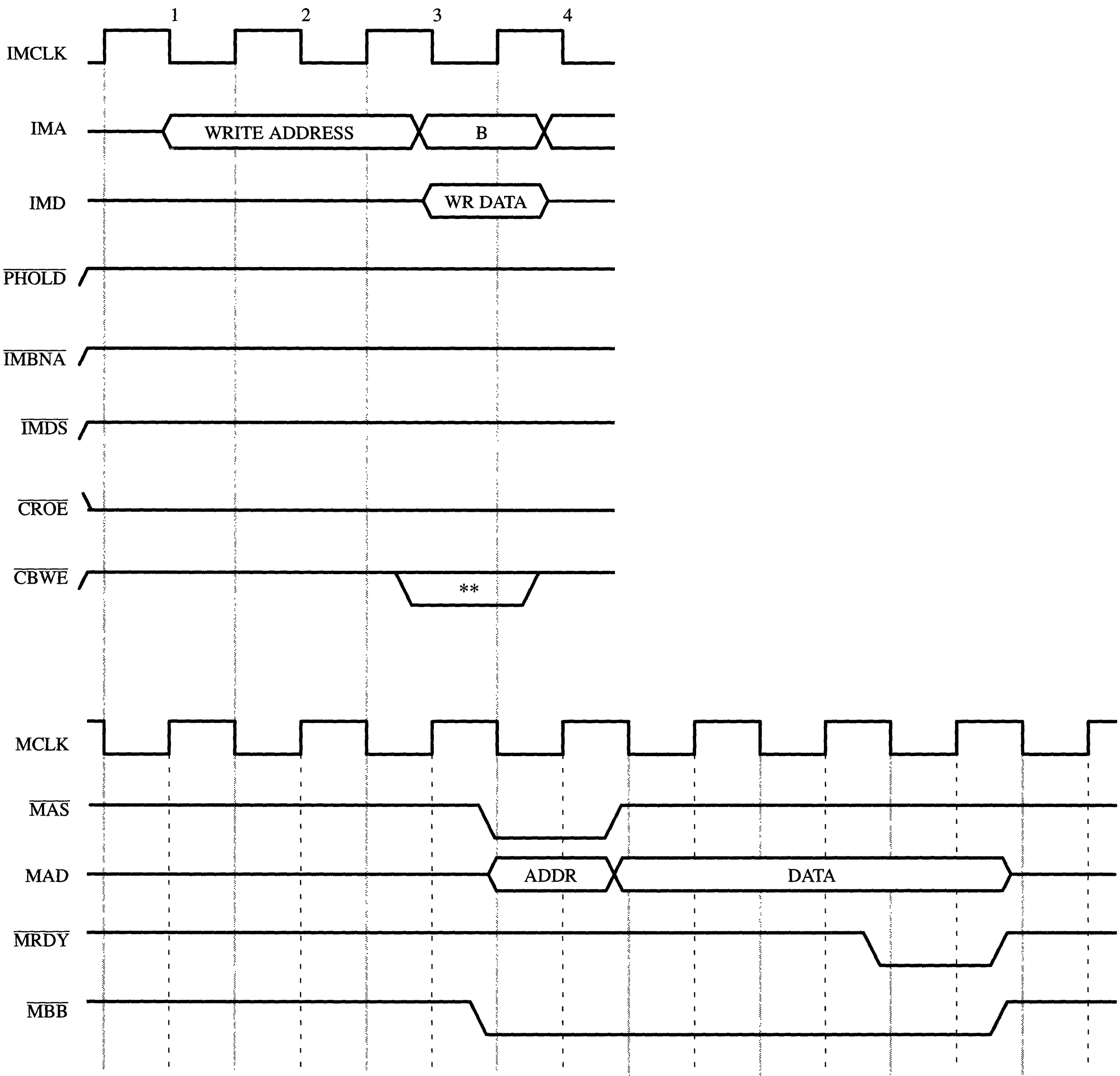


Figure 4–46. Write Cache Miss, Copy-back, No Modified Data Synchronous Clocks (page 1 of 2)



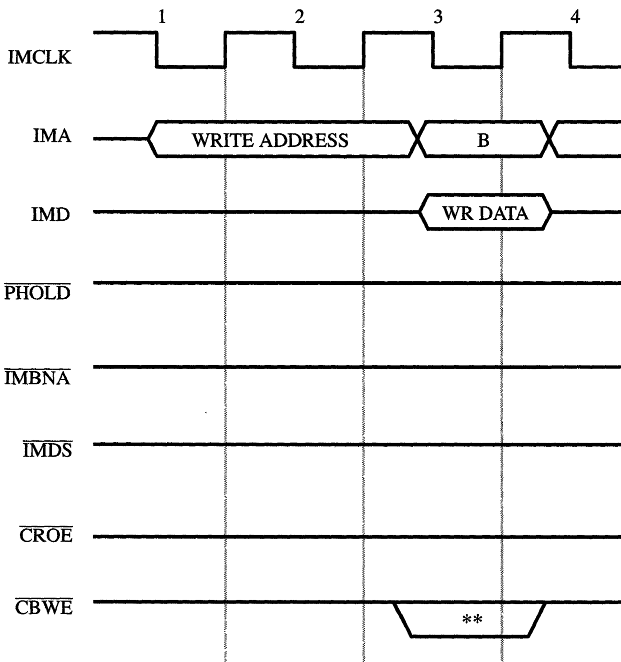
** $\overline{\text{CWBE}}$ is asserted only for the bytes selected by IMA and IMSIZE.

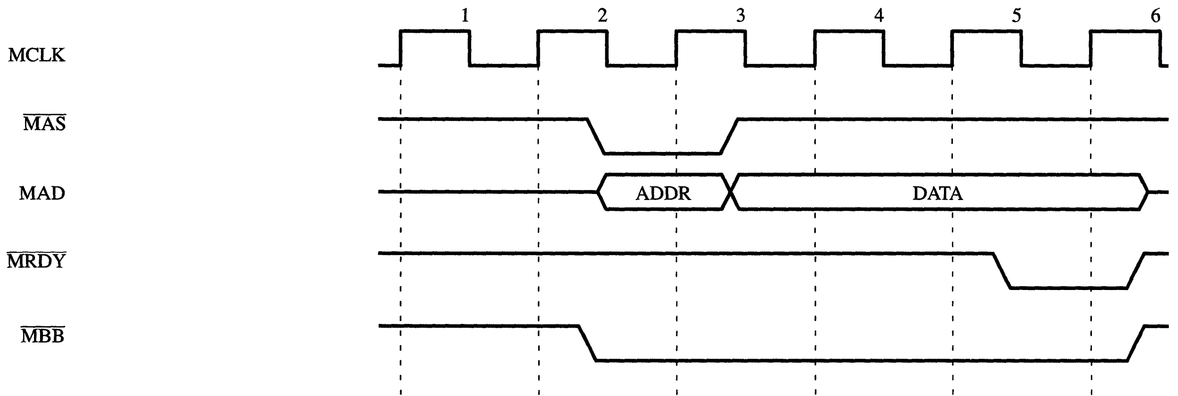
Figure 4–46. Write Cache Miss, Copy-back, No Modified Data Synchronous Clocks (page 2 of 2)



** CWBE is asserted only if the cache hits and only for the bytes selected by IMA and IMSIZE.

Figure 4–47. Write Through Write, Synchronous Clocks





** CWBE is asserted only if the cache hits and only for the bytes selected by IMA and IMSIZE.

Figure 4–48. Write Through Write, Asynchronous Clocks

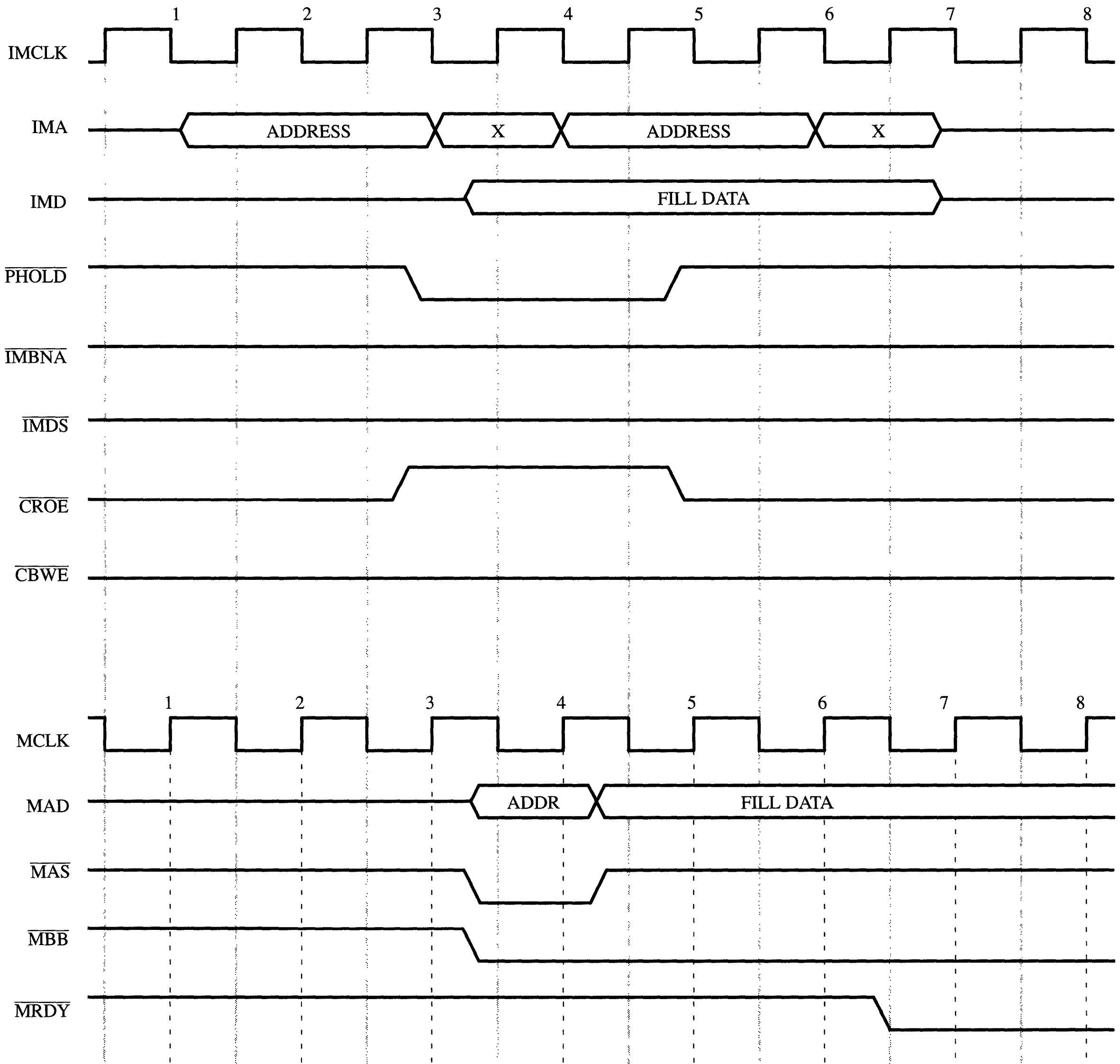


Figure 4–49. RT625 Block Fill, Synchronous Clocks (page 1 of 2)

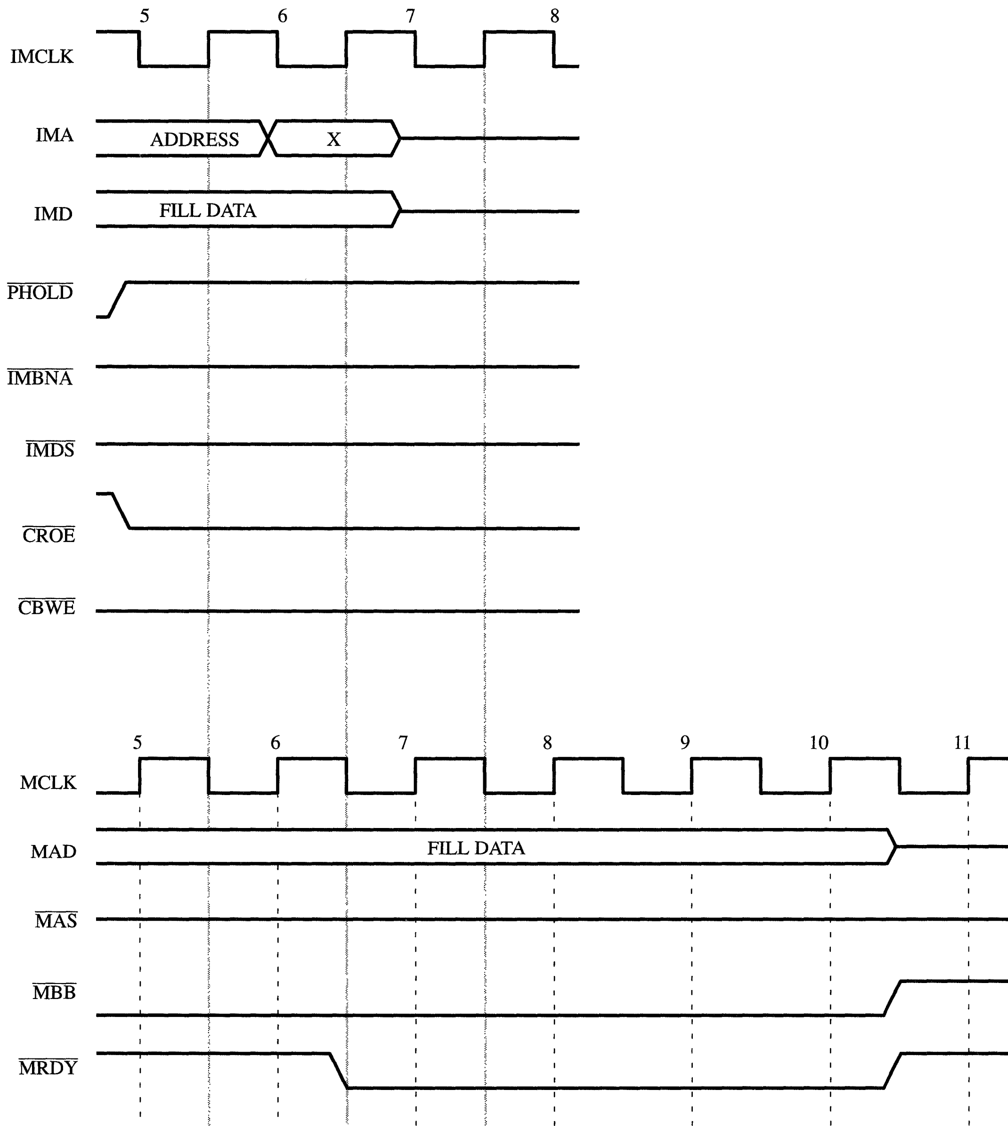
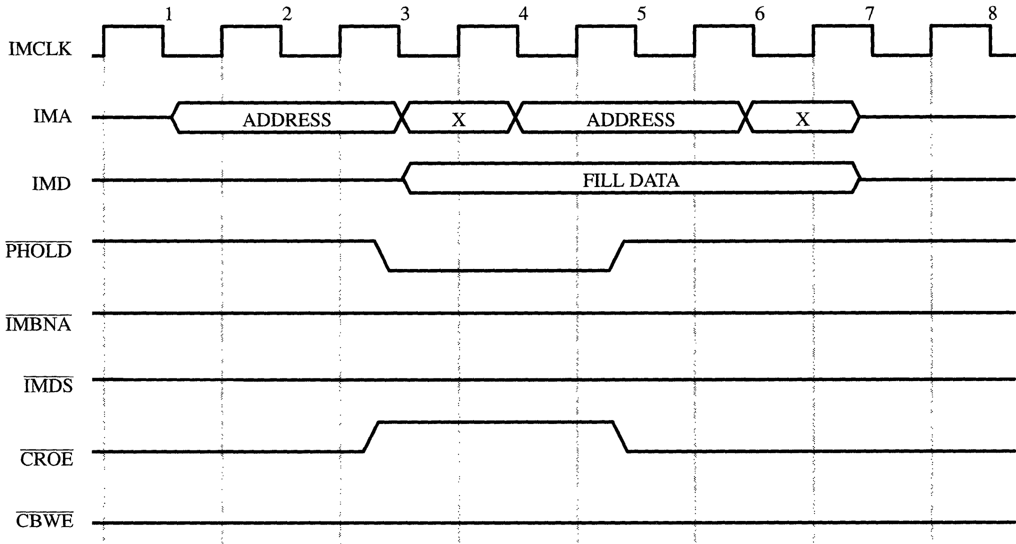


Figure 4–49. RT625 Block Fill, Synchronous Clocks (page 2 of 2)



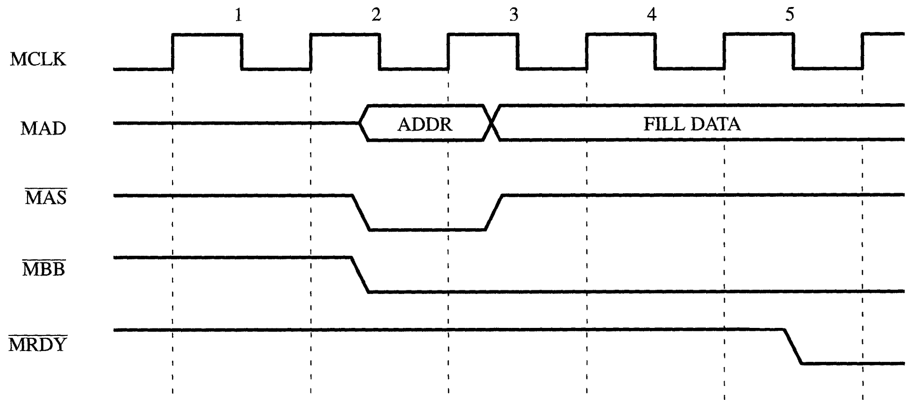
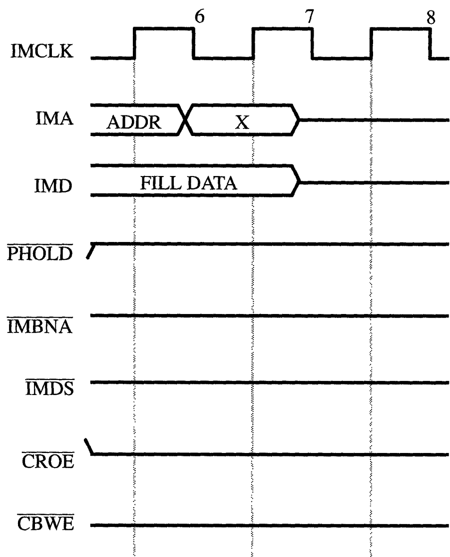


Figure 4–50. RT625 Block Fill, Asynchronous Clocks (page 1 of 2)



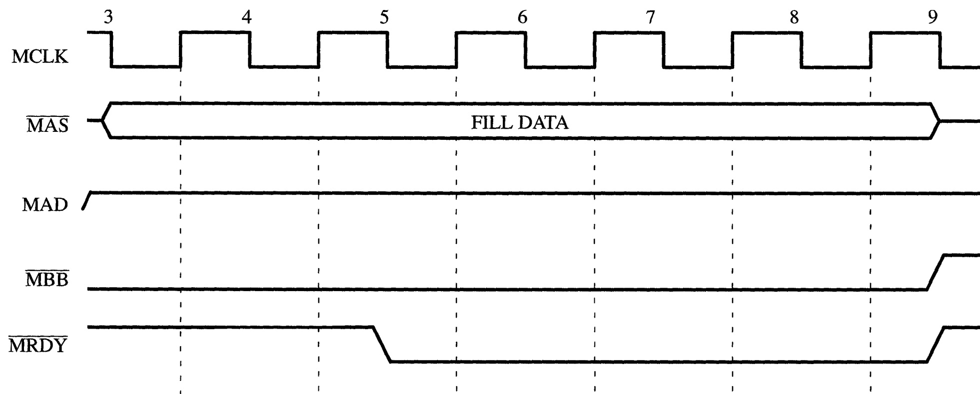


Figure 4–50. RT625 Block Fill, Asynchronous Clocks (page 2 of 2)

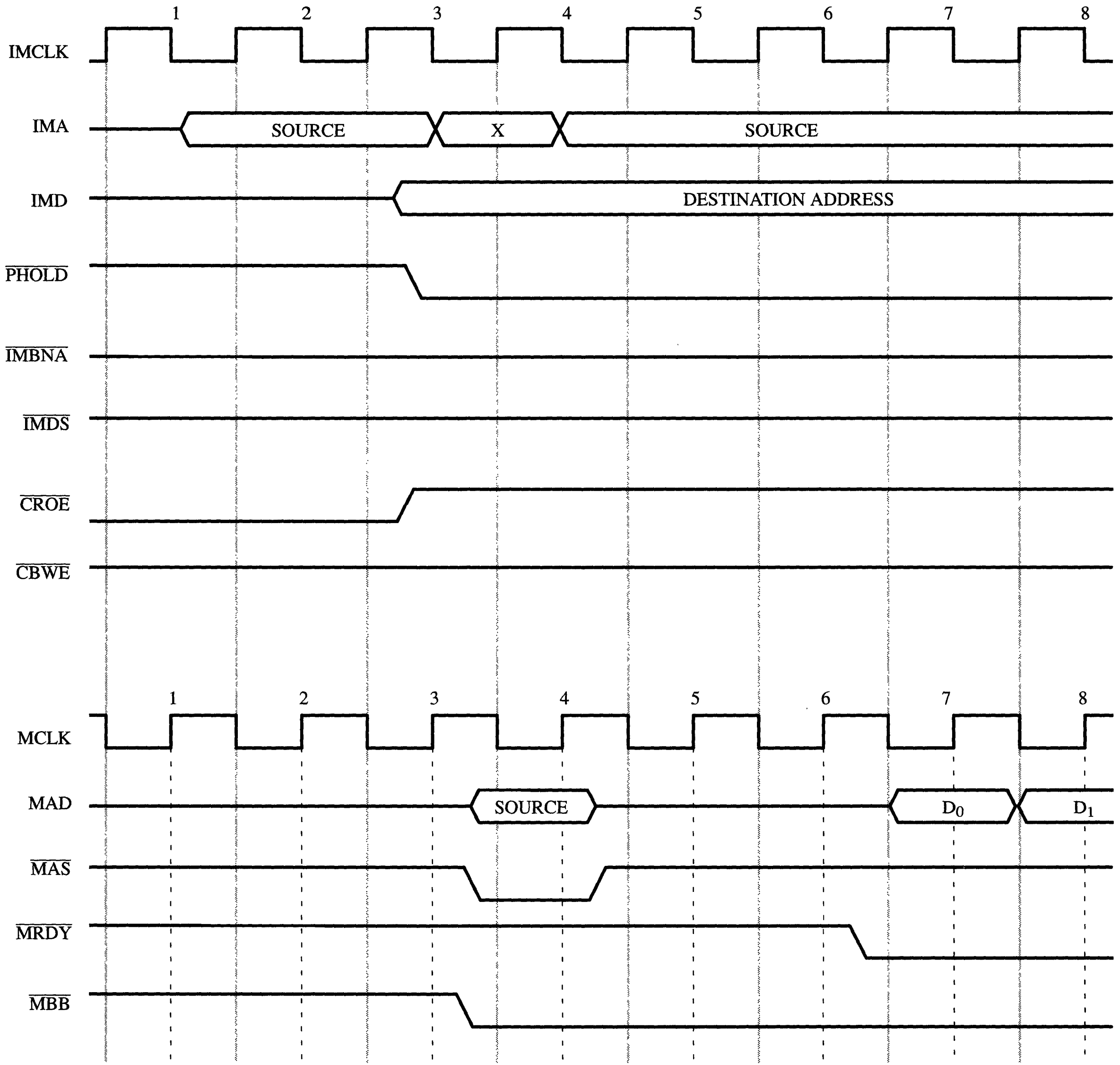


Figure 4–51. Block Copy, Synchronous Clocks (page 1 of 3)

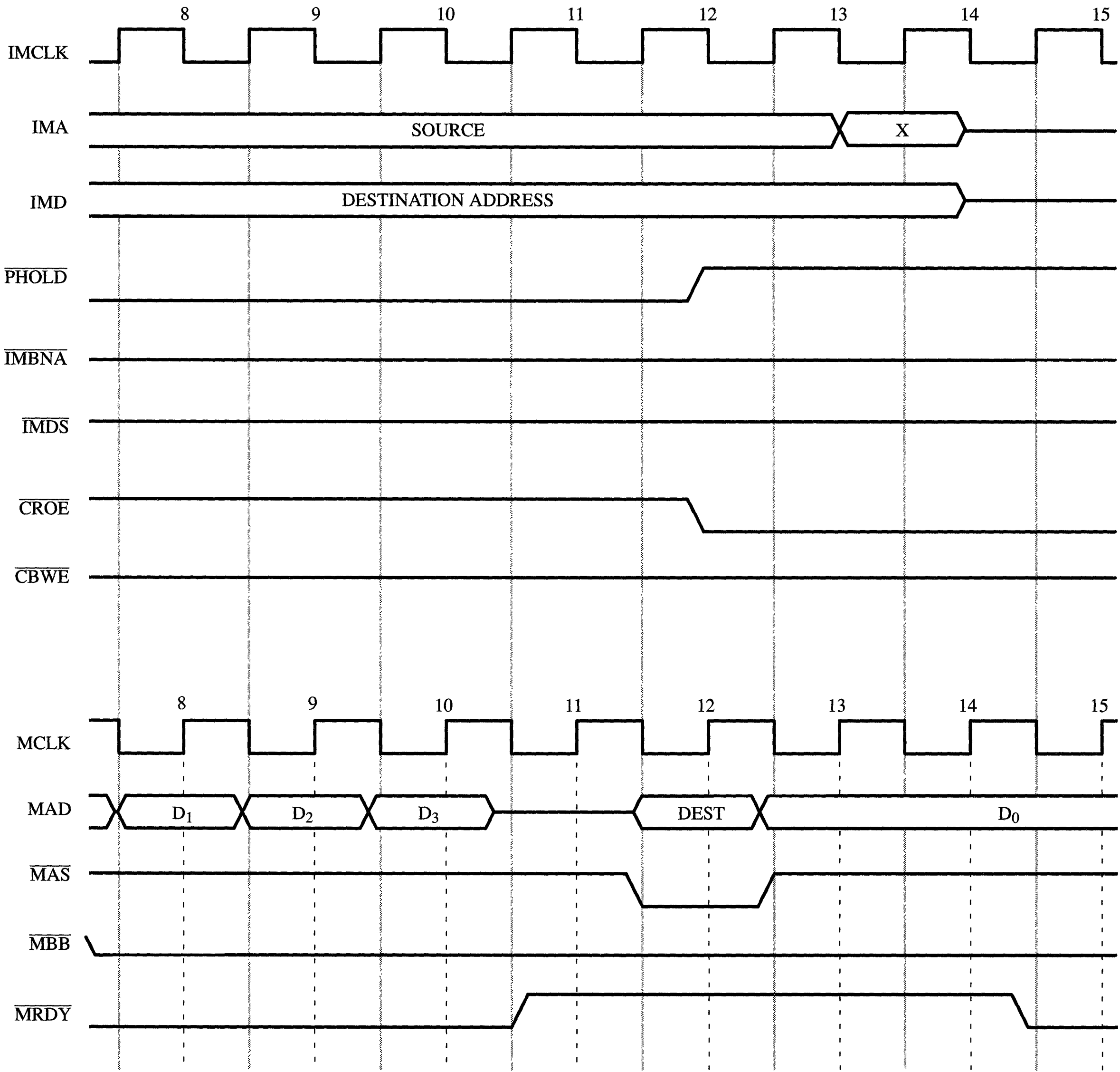


Figure 4–51. Block Copy, Synchronous Clocks (page 2 of 3)

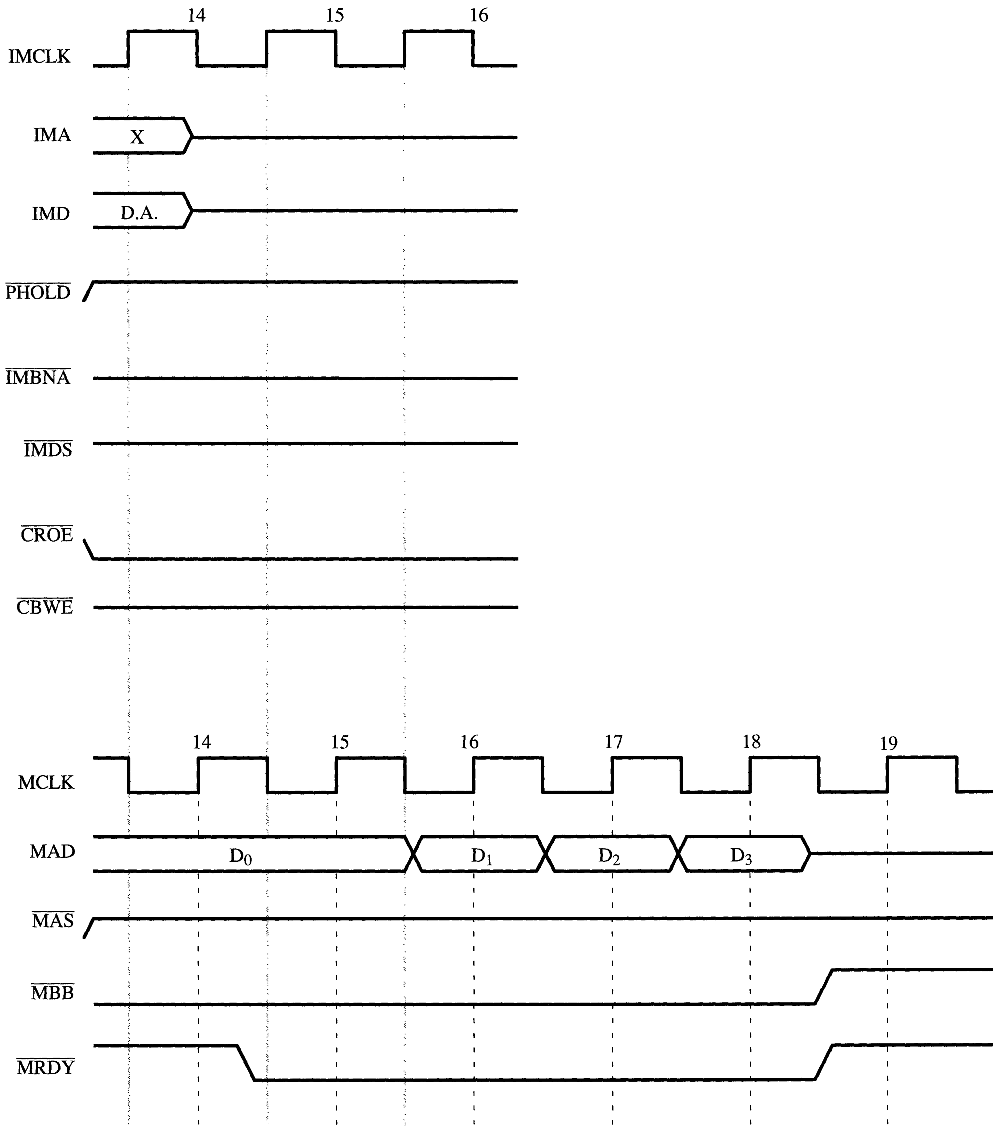
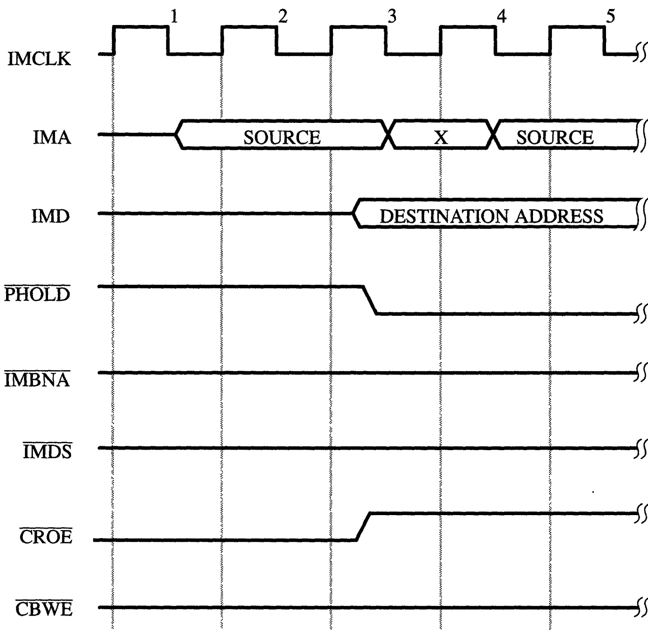


Figure 4–51. Block Copy, Synchronous Clocks (page 3 of 3)



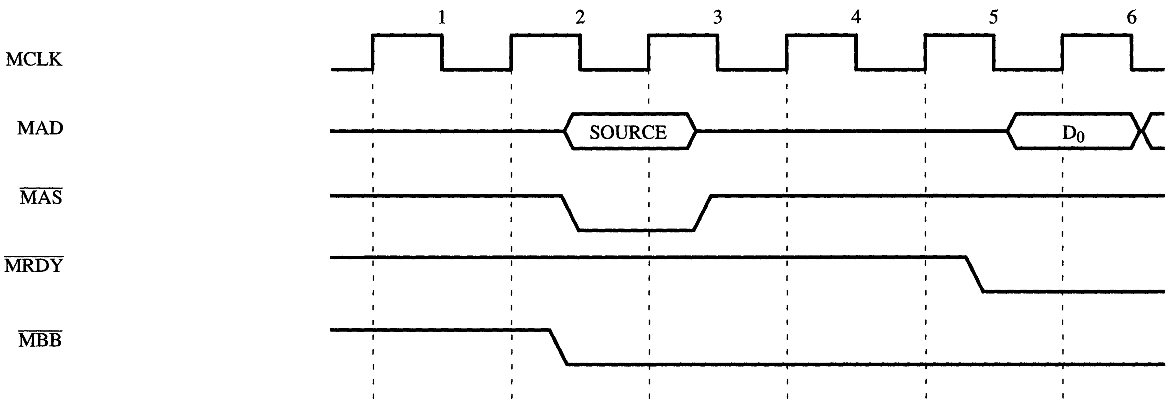


Figure 4–52. Block Copy, Asynchronous Clocks (page 1 of 3)

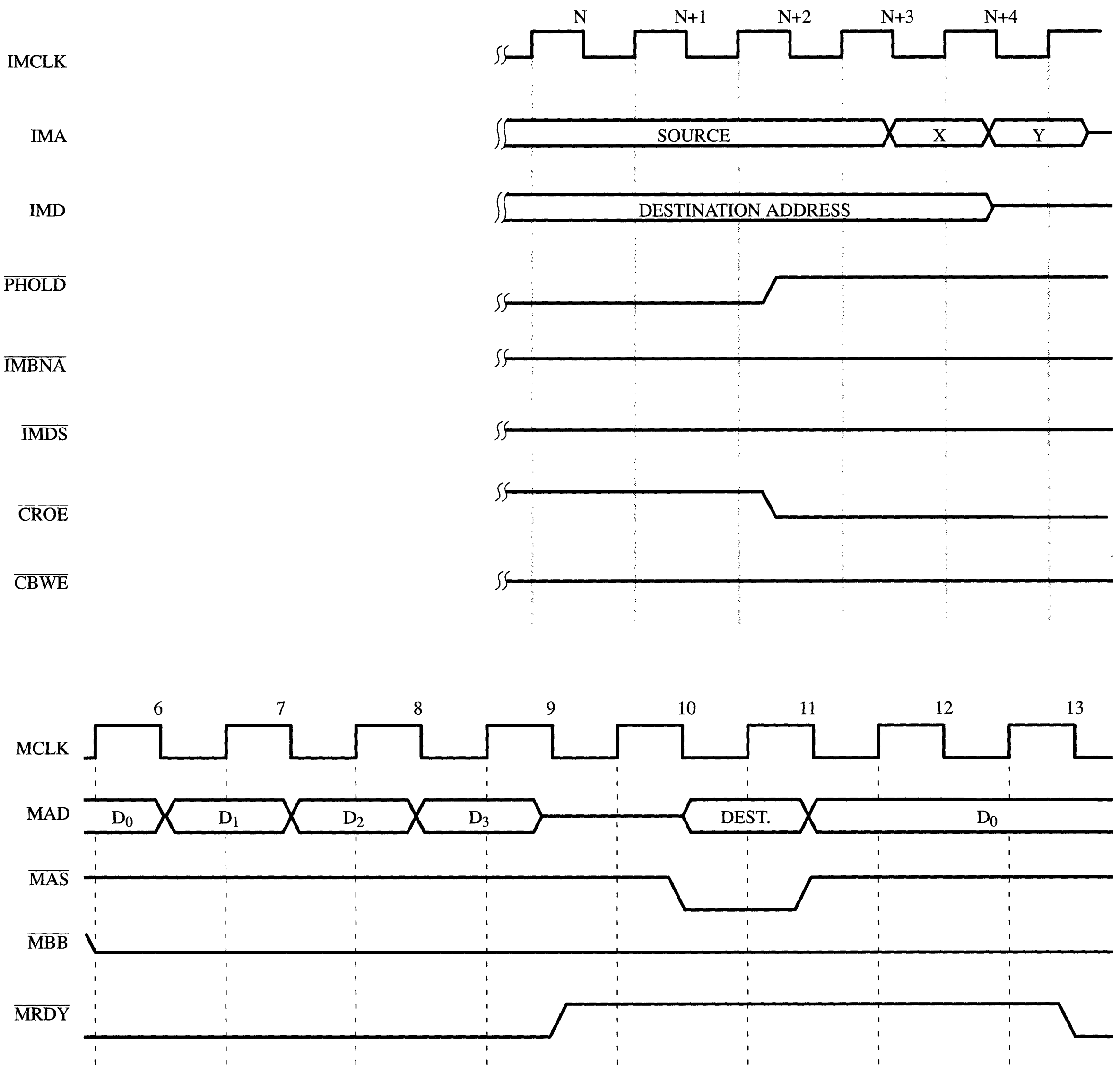
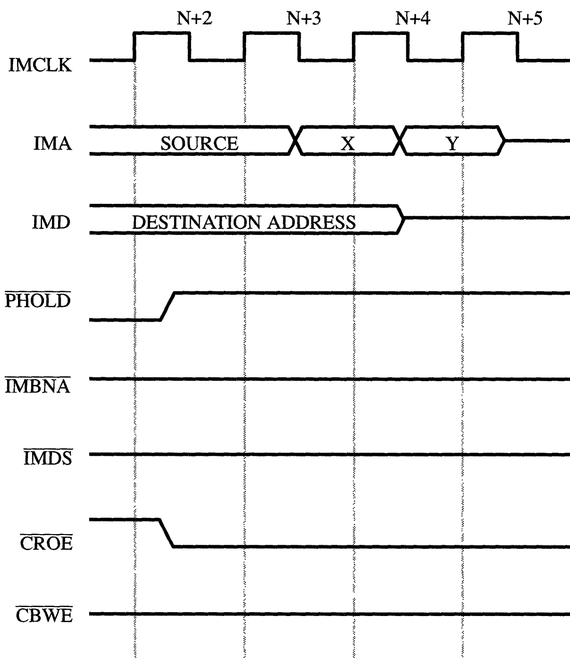


Figure 4–52. Block Copy, Asynchronous Clocks (page 2 of 3)



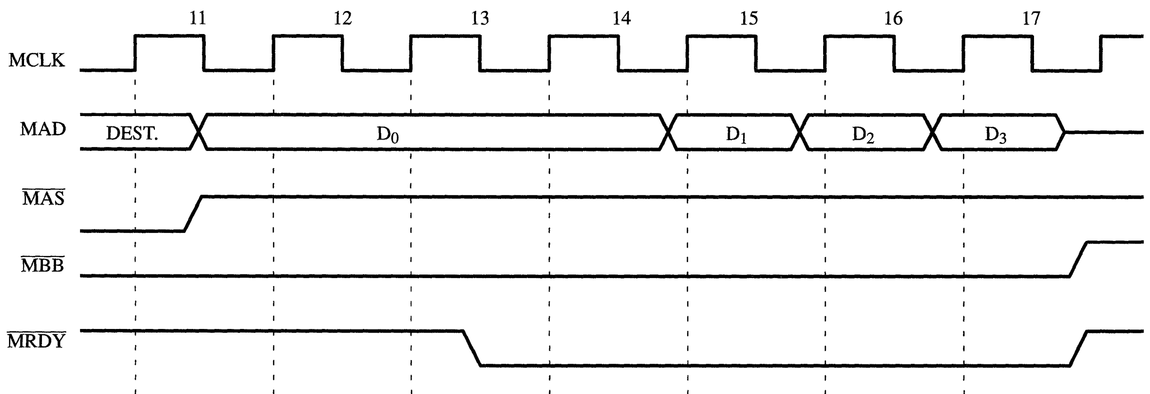


Figure 4–52. Block Copy, Asynchronous Clocks (page 3 of 3)

RT627 hyperSPARC Cache Data Unit

Figure 5–1 illustrates the hyperSPARC Central Processing Unit (CPU) configured with four RT627 Cache Data Units (CDUs), constituting a 256-Kbyte cache subsystem. The RT627s are specialty SRAMS that are specifically designed to interface with the RT620 and the RT625.

The RT627 is a 524,288 bit synchronous SRAM. The device integrates a 4-Kbyte x 32-bit SRAM core with advanced peripheral circuitry. All the control inputs to the device are synchronous.

In order to minimize external interface logic, the RT627 contains a one-deep write-buffer pipeline, byte write logic, registered inputs, data-in and data-out latches, and data forwarding logic from the write-buffer. Because it is designed specifically for the hyperSPARC family of devices, the RT627 CDU requires no glue logic to interface with the CPU and the RT625 CMTU. All relevant pins on each device connect direct to one another.

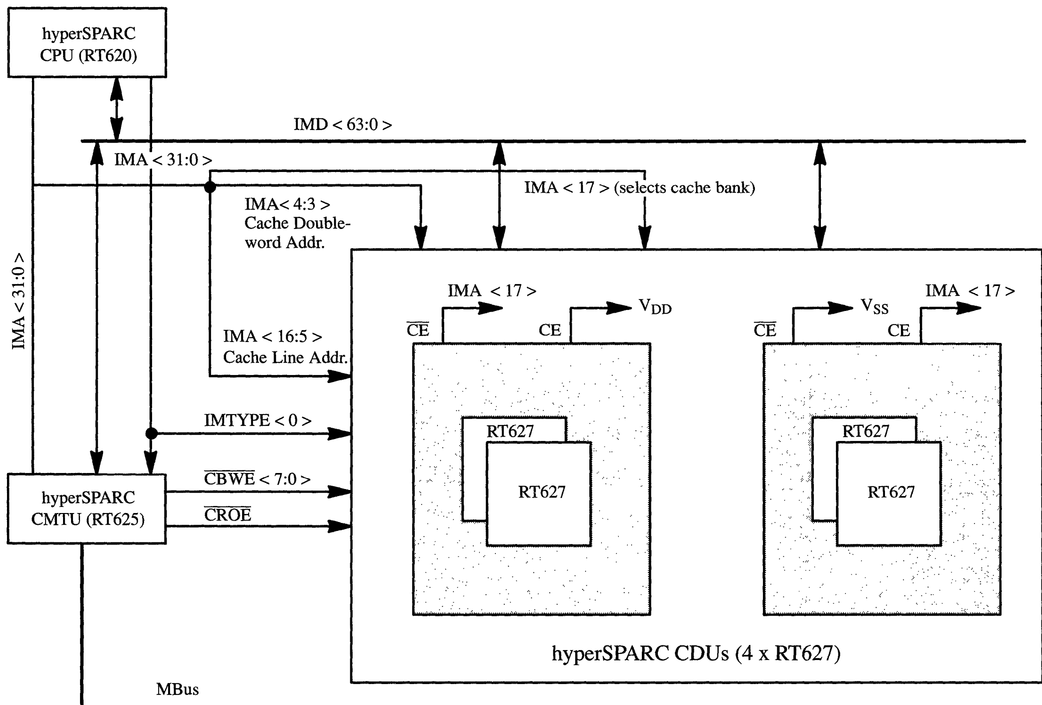


Figure 5–1. 256-Kbyte Cache Subsystem

The two chip enables ($\overline{CE}$, CE) are included in order to support 256-Kbyte cache subsystem with one CMTU and no glue logic. The read RD signal indicates whether the CPU or CMTU is reading (read access) the data from the cache or writing (write access) data into it. The read access appears for only one cycle on the Intra-Module Bus whereas the write access appears for two cycles on the Intra-Module Bus if the access is from the processor. As a result, the RD signal is low for two cycles in write accesses from the processor. However, the RT627 views the write access as a single cycle pipelined write access (the first cycle of the write access is utilized by the CMTU to do address translation and access level checking). The CDU output enable ($\overline{CROE}$) is used in conjunction with the RD pin to control the output buffers of the Cache Data Units. The four cache byte write enables ($\overline{CBWE} < 3:0 >$) are provided to allow individually writeable bytes.

The RT627 includes a pipelined stage write-buffer for high performance (see *Figure 5–2*). Address register-1 and data input latches offer this pipeline stage. Writing into the RAM core is delayed until the next write access. In order to allow forwarding, a comparator is included to compare the address of the write-buffer and the incoming read address. Valid bits, V_0, V_1, V_2 and V_3, are included to indicate if the appropriate bytes are to be written into the RAM core or not.

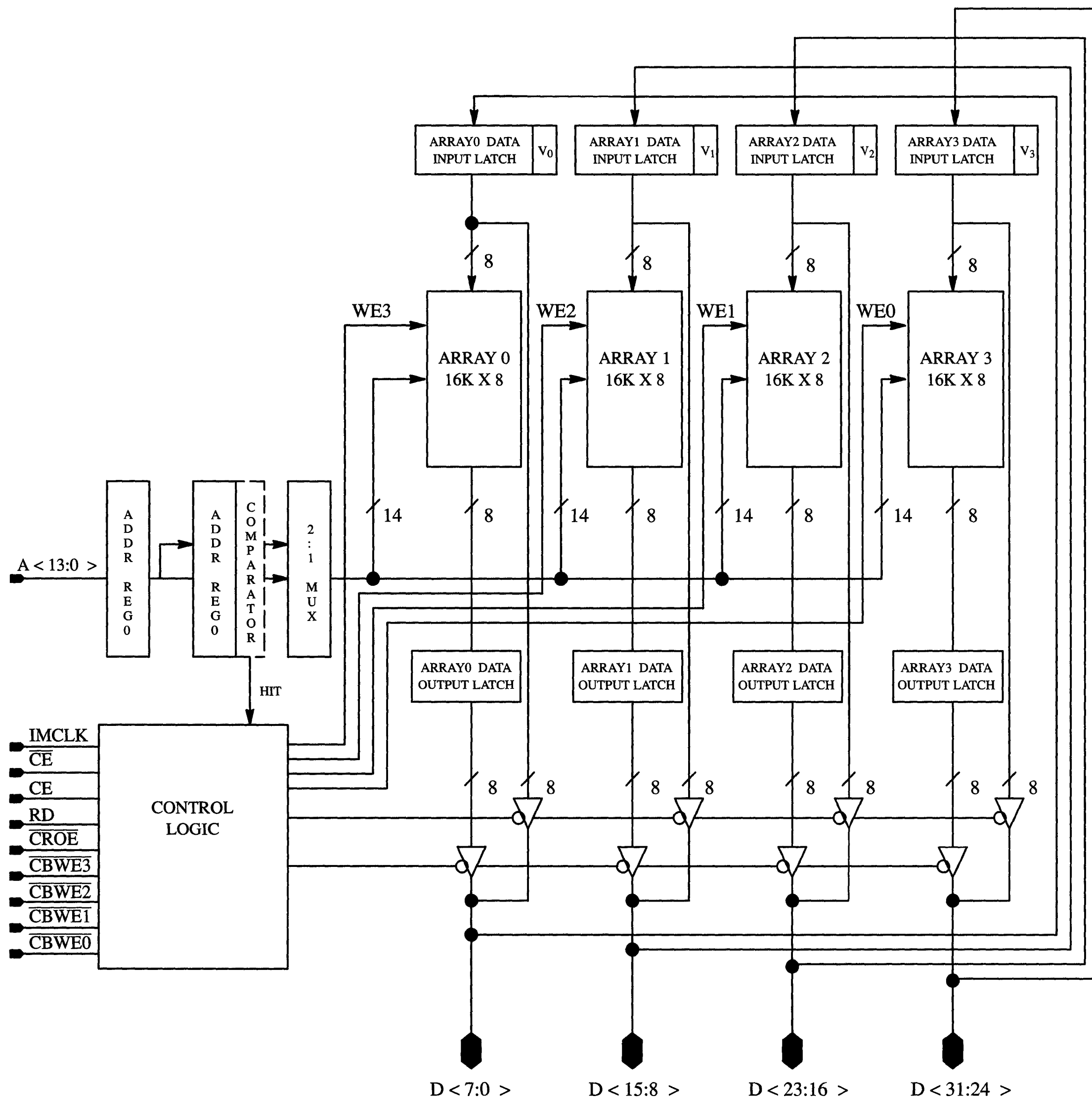


Figure 5–2. RT627 Block Diagram

5.1 RT627 Pinouts

IMCLK— (input) Intra-Module Clock

This is the basic clock for all the Intra-Module Bus components. All the Intra-Module Bus signals are driven and sampled only on the rising edge of the IMCLK. The RT627 uses only the IMCLK clock (not the MBus clock). The RAM cycle is equal to a full clock cycle of IMCLK.

IMA < 13:0 > — (input) Intra-Module Address Bus

These signals directly interface to the address bus of the Intra-Module Bus. The addresses sent out by the RT620 and the RT625 are unlatched and so the RT627 will latch the address.

IMD < 31:0 > — (input/output, tri-state) Intra-Module Data Bus

These signals directly interface with the bi-directional data bus of the Intra-Module Bus. The data bus is driven by the RT620 only during the execution of STORE instructions and the store cycle of Atomic Load-Store instructions. The data bus is driven by the RT625 during cache line fills. Store data sent out by the CYC620 and the RT625 are unlatched and so the RT627 will latch the data. Alignment for Load and Store instructions is performed by the CPU. Doublewords are aligned on 8-byte boundaries, words on 4-byte boundaries, and halfwords on 2-byte boundaries.

$\overline{\text{CE}}$, CE — (input) Chip Enable

These signals are included in order to support the 256-Kbyte cache subsystem with one CMTU without any glue logic. By appropriately connecting an additional Intra-Module Address Bus signal (IMA < 17 >) to these signals, two 128-Kbyte cache banks can be formed for 256-Kbyte cache subsystem as shown in *Figure 5–1*. Because the addresses sent out by the RT620 and the RT625 are unlatched, the RT627 will latch $\overline{\text{CE}}$ and CE inputs.

RD — (input) Read Access

This signal acts as an advanced access type information signal. This signal is useful in high performance systems to determine whether to turn On/Off the output drivers of the RAMs. This signal is directly connected to Intra-Module access TYPE signal (IMTYPE < 0 >). The value of this signal during a given cycle relates only to the address which appears on pins IMA < 31:0 >. Because the IMTYPE bits sent out by the RT620 and the RT625 are unlatched, the RT627 will latch RD input.

$\overline{\text{CROE}}$ — (input) Cache RAM Output Enable

this Output Enable signal is used in conjunction with the advanced RD signal to control the output drivers of the bidirectional data lines. During read accesses, if the RD is HIGH and $\overline{\text{CROE}}$ is asserted (i.e., is LOW) the CDU will drive the data bus. During read accesses, if the $\overline{\text{CROE}}$ is deasserted (i.e., is HIGH) the CDU will not drive the data bus. During write accesses, the CDU will drive the data lines.

RD	$\overline{\text{CROE}}$	
0	X	Data lines not driven
1	0	Data lines driven
1	1	Data lines not driven

$\overline{\text{CBWE}}$ < 3:0 > — (input) Cache RAM Byte Write Enables

The $\overline{\text{CBWE}}$ < 3:0 > signals control data writes into the RAMs. $\overline{\text{CBWE}}$ < 0 > controls byte write on data lines IMD < 31:24 >, $\overline{\text{CBWE}}$ < 1 > controls byte write on data lines IMD < 23:16 >, $\overline{\text{CBWE}}$ < 2 > controls byte write on data lines IMD < 15:8 > and $\overline{\text{CBWE}}$ < 3 > controls byte write on data lines IMD < 7:0 >.

5.2 RT627 Access Waveforms

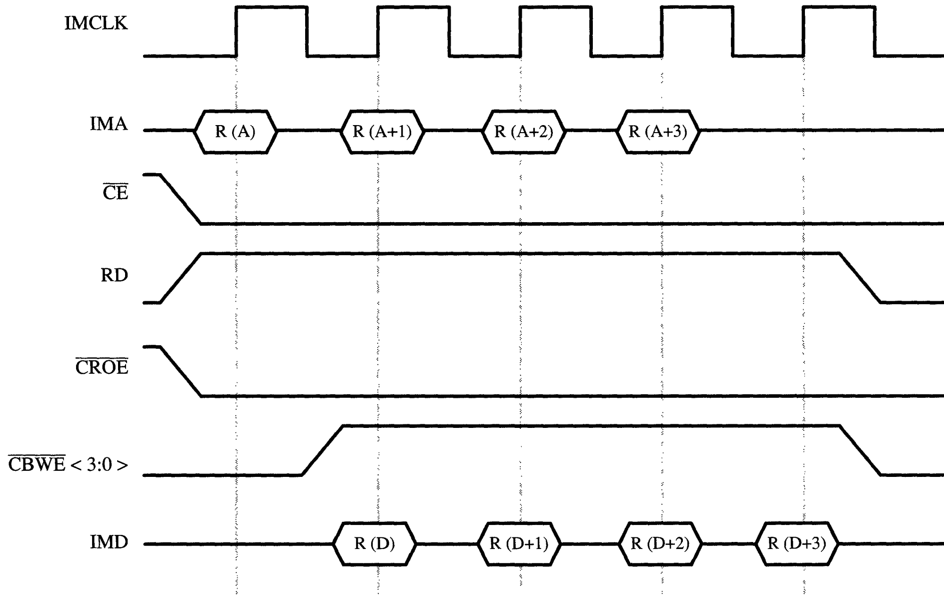


Figure 5–3. Read Access followed by Read Access

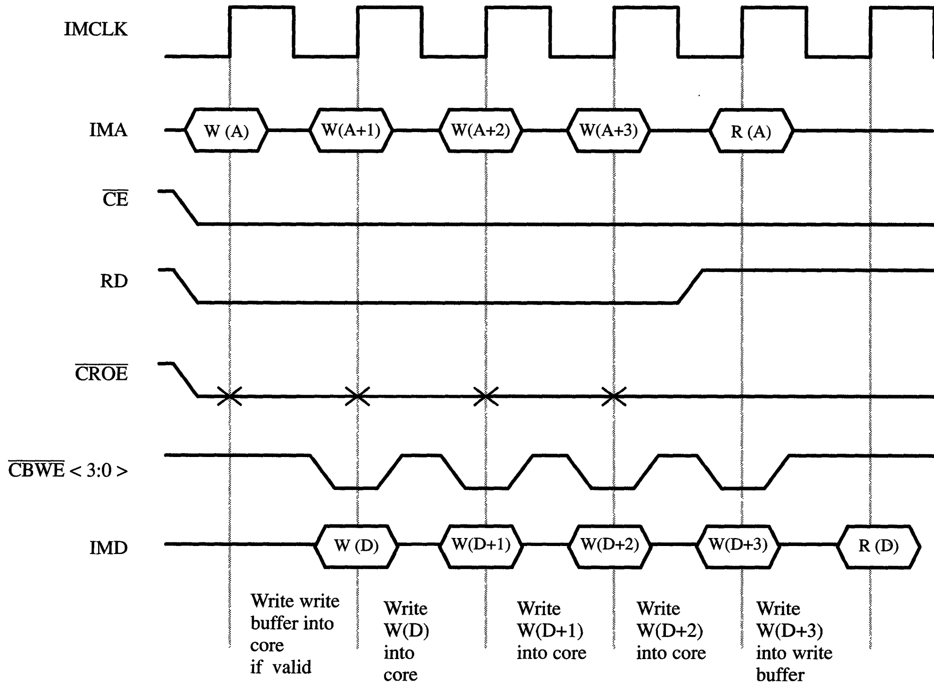


Figure 5–4. Write Access Followed by Write Access

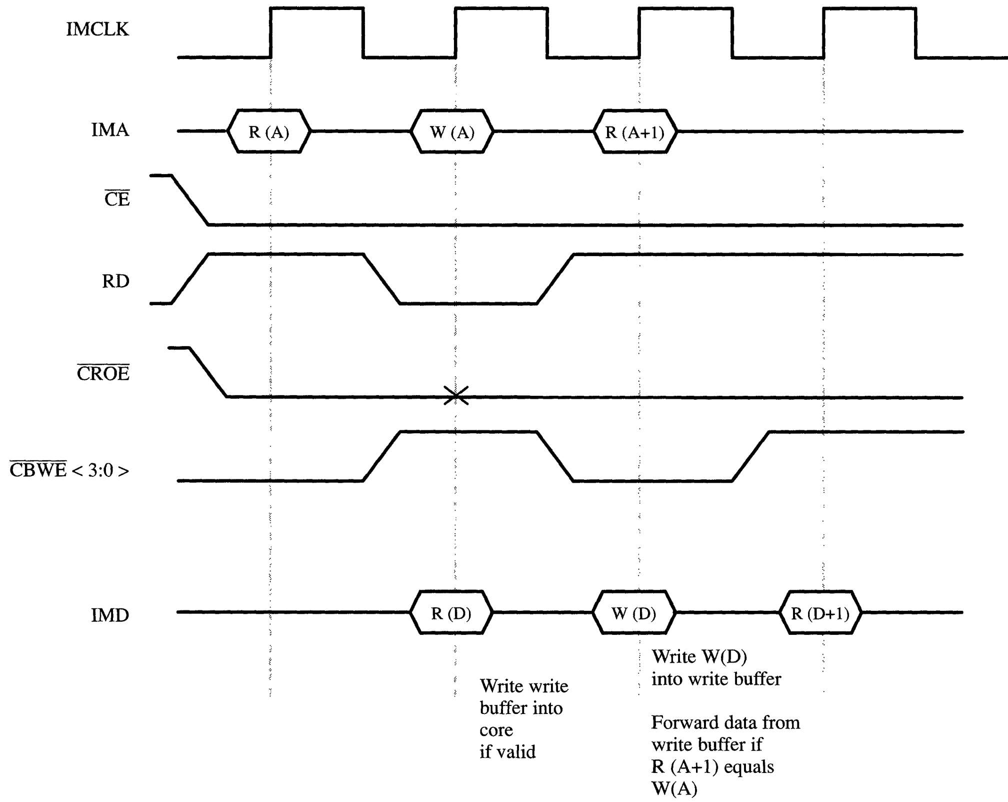
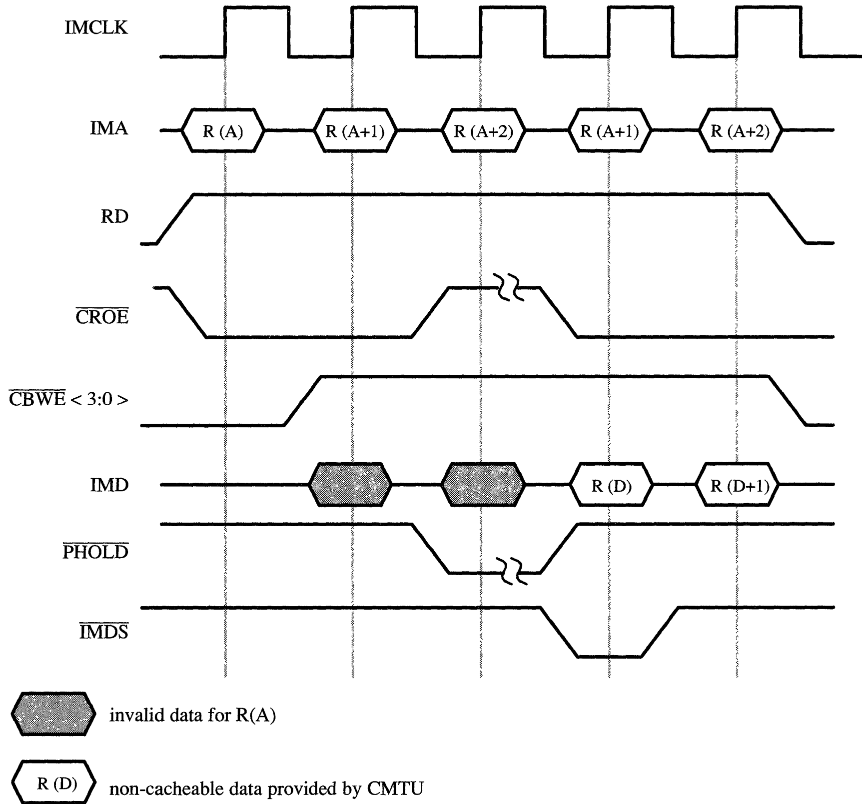


Figure 5–5. Read followed by Write Access followed by Read Access



PHOLD and IMDS are signals from the CMTU to the CPU and not related to the CDU.

Figure 5–6. Non-Cacheable Read Access (illustrates the use of CROE)

CY7C601/CY7C611 Integer Unit

This chapter describes the workings of the CY7C601 Integer Unit (IU) and the CY7C611 Embedded Controller Integer Unit. Descriptions and explanations given for the CY7C601 also apply to the CY7C611 Integer Unit, except for those differences noted in *Section* 6.7.

The CY7C600-family Integer Units are based on the SPARC 32-bit RISC architecture, which defines a processor capable of execution at a rate approaching one instruction per clock cycle. The CY7C601 supports a tightly-coupled floating-point unit (FPU) and a second, system-specific coprocessor, all three of which may operate concurrently. The CY7C611 supports an FPU in the same manner as the CY7C601, but does not support the coprocessor interface. The CY7C601 executes all instructions except floating-point-operate and coprocessor-operate instructions.

A block diagram of the CY7C601/CY7C611 is shown in *Figure 6–1*. The processor is organized around the ALU and the shift unit. These are both two-operand units, accepting 32-bit information from either source 1 or source 2 of the register file, the program counters, or the instruction decoder. ALU or shift unit results may be passed to the register file, address bus, program counters, control registers, or back to themselves.

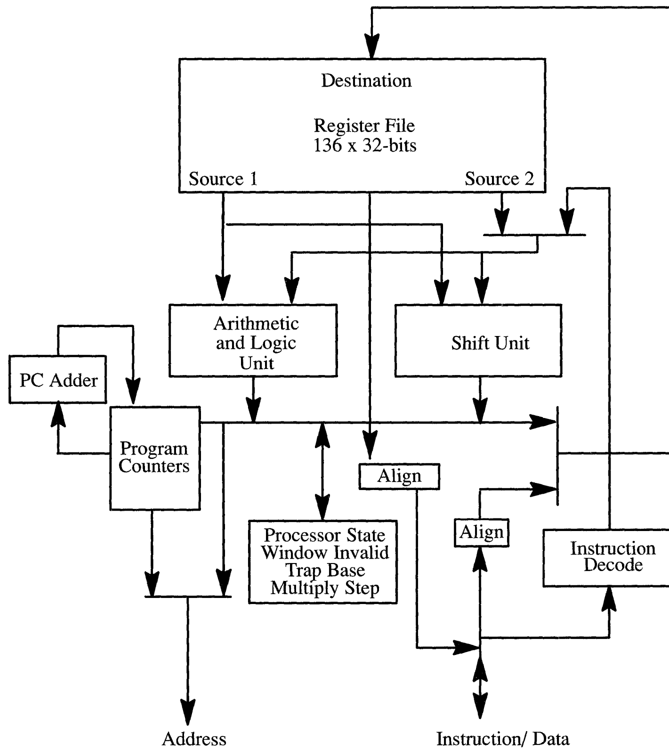


Figure 6–1. Integer Unit Block Diagram

The standard version of the Integer Unit, the CY7C601, contains a 136-bit x 32-bit register file divided into eight overlapping windows. It is supplied in 207-pin PGA and 208-pin QFP packages, which allows 32-bit address and data buses, an eight-bit ASI bus, a number of control lines, and provides both floating-point and coprocessor interfaces.

The CY7C611 Embedded Controller IU is internally the same as the CY7C601, but it is externally optimized for board-space-sensitive controller applications. By eliminating some external pins, the CY7C611 fits into a 160-pin PQFP package. In the smaller package, the address bus is modified to 24 bits, the ASI bus to 3 bits, and the coprocessor interface and five control lines are omitted.

6.1 CY7C601/CY7C611 Register Set

The following is a brief description of the SPARC register set. Detailed programming information for ROSS Technology SPARC processors is given in *Chapter 2,* SPARC Programming Environment.

The general register model for the CY7C600 family is given in *Figure 6–2*. The CY7C601/CY7C611 register set consists of the processor state register (PSR), the trap base register (TBR), the window invalid mask (WIM), the multiply step register (Y register), and 136 *r*-registers. These registers are described in detail in *Section 2.2*.

The *r*-registers are the working register set for the SPARC Integer Unit. All *r*-registers are 32-bits in length. The 136 *r*-registers supported by the CY7C601/CY7C611 are divided into 128 windowed *r*-registers and eight global registers. The 128 windowed *r*-registers are divided into eight overlapping windows of 24 *r*-registers. The twenty-four *r*-registers that comprise a window are further subdivided into three groups of eight registers, referred to as the *in*, *out*, and *local* registers. The eight *r*-register windows overlap in a manner such that the *in* registers of one window are the *out* registers of the previous window. *Local* *r*-registers are not shared with another window, but are private to that window. The current window in use by the processor is pointed to by the Current Window Pointer (CWP), a field within the processor state register. In addition to the *r*-register window, there are eight global registers that are accessible regardless of the Current Window Pointer.

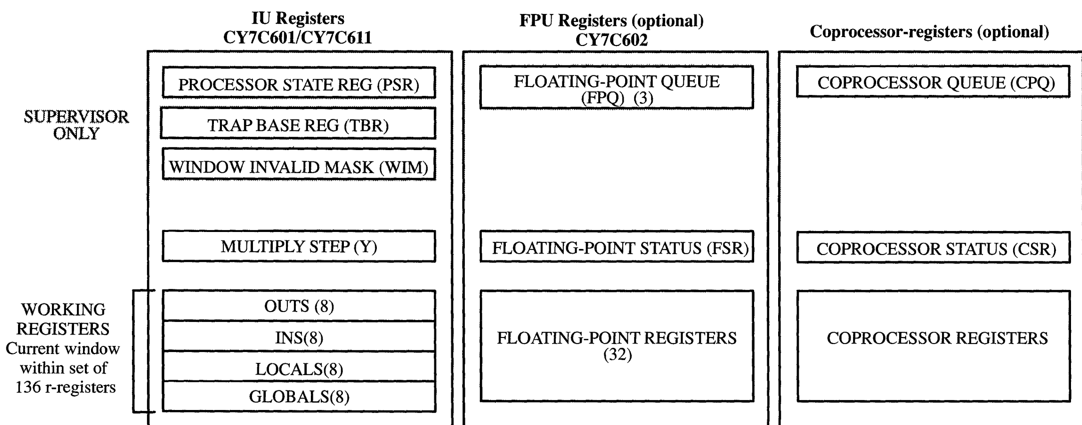


Figure 6–2. SPARC Register Model

6.1.1 CY7C601/CY7C611 Cycle Per Instruction (CPI)

Table 6–1. CY7C601/CY7C611 Instruction CPI

Name		Operation		Cycles
Load/Store Instructions				
LDSB	(LDSBA)[1]	Load Signed Byte	(from Alternate Space)	2
LDSH	(LDSHA)[1]	Load Signed Halfword	(from Alternate Space)	2
LDUB	(LDUBA)[1]	Load Unsigned Byte	(from Alternate Space)	2
LDUH	(LDUHA)[1]	Load Unsigned Halfword	(from Alternate Space)	2
LD	(LDA)[1]	Load Word	(from Alternate Space)	2
LDD	(LDDA)[1]	Load Doubleword	(from Alternate Space)	3
LDF		Load Floating-Point		2
LDDF		Load Double Floating-Point		3
LDFSR		Load Floating-Point Status		2
LDC		Load Coprocessor		2
LDDC		Load Double Coprocessor		3
LDCSR		Load Coprocessor Status Register		2
STB	(STBA)[1]	Store Byte	(into Alternate Space)	3
STH	(STHA)[1]	Store Halfword	(into Alternate Space)	3
ST	(STA)[1]	Store Word	(into Alternate Space)	3
STD	(STDA)[1]	Store Doubleword	(into Alternate Space)	4
STF		Store Floating-Point		3
STDF		Store Double Floating-Point		4
STFSR		Store Floating-Point Status Register		3
STDFQ[1]		Store Double Floating-Point Queue		4
STC		Store Coprocessor		3
STDC		Store Double Coprocessor		4
STCSR		Store Coprocessor State Register		3
STDCQ[1]		Store Double Coprocessor Queue		4
LDSTUB	(LDSTUBA)[1]	Atomic Load-Store Unsigned Byte	(in Alternate Space)	4
SWAP	(SWAPA)[1]	Swap *r*-register with Memory	(in Alternate Space)	4

Name		Operation		Cycles
Arithmetic/Logical/Shift Instructions				
ADD	(ADDcc)	Add	(and modify icc)	1
ADDX	(ADDXcc)	Add with Carry	(and modify icc)	1
TADDcc	(TADDccTV)	Tagged Add and modiify icc	(and Trap on oVerflow)	1
SUB	(SUBcc)	Subtract	(and modify icc)	1
SUBX	(SUBXcc)	Subtract with Carry	(and modify icc)	1
TSUBcc	(TSUBccTV)	Tagged Subtract and modify icc	(and Trap on oVerflow)	1
MULScc		Multiply Step and modify icc		1
AND	(ANDcc)	And	(and modify icc)	1
ANDN	(ANDNcc)	And Not	(and modify icc)	1
OR	(ORcc)	Inclusive Or	(and modify icc)	1
ORN	(ORNcc)	Inclusive Or Not	(and modify icc)	1
XOR	(XORcc)	Exclusive Or	(and modify icc)	1
XNOR	(XNORcc)	Exclusive Nor	(and modify icc)	1
SLL		Shift Left Logical		1
SRL		Shift Right Logical		1
SRA		Shift Right Arithmetic		1
SETHI		Set High 22 Bits of *r*-register		1
Control Transfer Instructions				
SAVE		SAVE caller's window		1
RESTORE		RESTORE caller's window		1
Bicc		Branch on integer condition codes		1 (2)
FBfcc		Branch on floating-point condition codes		1 (2)
CBccc		Branch on coprocessor condition codes		1 (2)
CALL		Call		1 (2)
JMPL		Jump and link		2 (2)
RETT		Return from Trap		2 (2)
Ticc		Trap on integer condition codes		1 (3)
Read/Write Control Register Instructions				
RDY		Read Y Register		1
RDPSR (1)		Read Processor State Register		1
RDWIM (1)		Read Window Invalid Mask		1
RDTBR (1)		Read Trap Base Register		1
WRY		Write Y Register		1
WRPSR (1)		Write Processor State Register		1
WRWIM (1)		Write Window Invalid Mask		1
WRTBR (1)		Write Trap Base Register		1
Miscellaneous Instructions				
FLUSH		Instruction Cache FLUSH		1
UNIMP		Unimplemented Instruction		1 (4)

Notes: 1. denotes supervisor instruction
2. assumes delay slot is filled with a useful instruction
3. A Ticc instruction requires 4 cycles if the trap is taken
4. The UNIMP instruction causes an unimplemented instruction trap

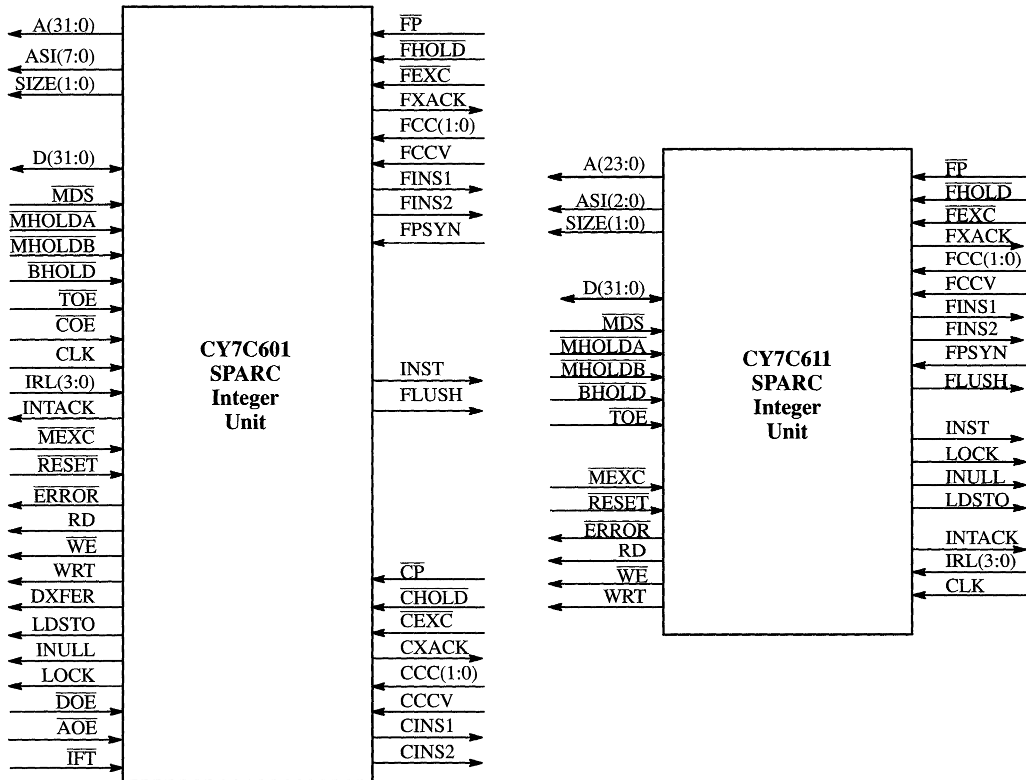


Figure 6–3. CY7C601/CY7C611 External Signals

6.2 Signal Description

This section provides a description of the CY7C601/CY7C611's external signals. Functionally, the IU's external signals can be divided into four categories: memory subsystem interface, floating-point/coprocessor interface, interrupt and control signals, and power and clock signals.

Signals that are active LOW are marked with an overscore; all others are active HIGH. *Table 6–2* summarizes the signals described in this section. *Table 6–2* provides a summary of the external signals for the CY7C601. *Table 6–9* in *Section 6.7* provides a summary of external signals for the CY7C611.

Note: In the descriptions below, and in this manual in general, when a signal is **asserted** it is active, and when it is **deasserted** it is inactive. When a signal is HIGH, it is a logical 1; when it is LOW, it is a logical 0. This is true regardless of whether it is asserted or deasserted.

Table 6–2. CY7C601 External Signal Summary

Pin Name	Description	Signal Type	Active
Memory Subsystem Interface Signals:			
A<31:0>	Address Bus	Three-State Output	
$\overline{AOE}$	Address Output Enable	Input	LOW
ASI<7:0>	Address Space Identifier	Three-State Output	
$\overline{COE}$	Control Output Enable	Input	LOW
$\overline{BHOLD}$	Bus Hold	Input	LOW
D<31:0>	Data Bus	Three-State BiDir.	
$\overline{DOE}$	Data Output Enable	Input	LOW
DXFER	Data Transfer	Three-State Output	HIGH
$\overline{IFT}$	Instruction Cache Flush Trap	Input	LOW
INULL	Integer Unit Nullify Cycle	Three-State Output	HIGH
LDST	Atomic Load-Store	Three-State Output	HIGH
LOCK	Bus Lock	Three-State Output	HIGH
$\overline{MDS}$	Memory Data Strobe	Input	LOW
$\overline{MEXC}$	Memory Exception	Input	LOW
$\overline{MHOLDA}$	Memory Bus Hold A	Input	LOW
$\overline{MHOLDB}$	Memory Bus Hold B	Input	LOW
RD	Read Access	Three-State Output	HIGH
SIZE<1:0>	Bus Transaction Size	Three-State Output	
$\overline{WE}$	Write Enable	Three-State Output	LOW
WRT	Advanced Write	Three-State Output	HIGH
Floating-Point / Coprocessor Interface Signals:			
CCC<1:0>	Coprocessor Condition Codes	Input	
CCCV	Coprocessor Condition Codes Valid	Input	HIGH
CEXC	Coprocessor Exception	Input	LOW
$\overline{CHOLD}$	Coprocessor Hold	Input	LOW
CINS1	Coprocessor Instruction in Buffer 1	Three-State Output	HIGH
CINS2	Coprocessor Instruction in Buffer 2	Three-State Output	HIGH
$\overline{CP}$	Coprocessor Unit Present	Input	LOW
CXACK	Coprocessor Exception Acknowledge	Three-State Output	HIGH
FCC<1:0>	Floating-Point Condition Codes	Input	
FCCV	Floating-Point Condition Codes Valid	Input	HIGH
$\overline{FEXC}$	Floating-Point Exception	Input	LOW
$\overline{FHOLD}$	Floating-Point Hold	Input	LOW
FINS1	Floating-Point Instruction in Buffer 1	Three-State Output	HIGH
FINS2	Floating-Point Instruction in Buffer 2	Three-State Output	HIGH
FLUSH	Floating-Point/Coprocessor Instruction Flush	Three-State Output	HIGH
$\overline{FP}$	Floating-Point Unit Present	Input	LOW

Pin Name	Description	Signal Type	Active
FXACK	Floating-Point Exception Acknowledge	Three-State Output	HIGH
INST	Instruction Fetch	Three-State Output	HIGH
Interrupt and Control Signals:			
IRL<3:0>	Interrupt Request Level	Input	
INTACK	Interrupt Acknowledge	Three-State Output	HIGH
RESET	Reset	Input	LOW
ERROR	Error State	Three-State Output	LOW
FPSYN	Floating-Point Synonym Mode	Input	HIGH
TOE	Test Mode Output Enable	Input	LOW
Power and Clock Signals:			
CLK	Clock	Input	
VCCI	Main internal VCC	Input	
VCCO	Output driver VCC	Input	
VCCT	Input circuit VCC	Input	
VSSI	Main internal VSS	Input	
VSSO	Output driver VSS	Input	
VSST	Input circuit VSS	Input	

The following sections describe the external signals for the CY7C601 and CY7C611. Signals that are modified for the CY7C611 are listed in brackets, such as *[A<23:0>]*. Signals not available on the CY7C611 are denoted as ***[Not available on CY7C611]***.

6.2.1 Memory Subsystem Interface Signals

Memory interface signals consist of the address lines (40 bits), bidirectional data lines (32 bits), transaction size lines (2 bits), and various control signals.

6.2.1.1 A<31:0>—Address Bus (output) [A<23:0>]

The 32-bit address bus carries instruction or data addresses during a Fetch or Load/Store operation. Addresses are sent out unlatched and must be latched external to the CY7C601/611. The address bus is three-stated when the AOE or TOE signal is deasserted (HIGH).

6.2.1.2 AOE—Address Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for the address bus, A<31:0>, and the ASI bus, ASI<7:0>, and is the normal condition. Deassertion of AOE three-states the output drivers and should only be done when the bus is granted to another bus master (i.e., when either BHOLD or MHOLDA/B is asserted).

6.2.1.3 ASI<7:0>—Address Space Identifier (output) [ASI<2:0>]

These 8 bits constitute the address space identifier (ASI), which identifies the memory address space to which the instruction or data access is being directed. The ASI bits are sent out unlatched—simultaneously with the memory address—and must be latched externally. The ASI pins are three-stated when the AOE or TOE signal is deasserted (HIGH). Encoding of the ASI bits is shown in *Table 6-3*. Additional ASI assignments for the SPARC architecture are listed in *Table 8-15*.

Table 6–3. ASI Assignments

CY7C601 Address Space Identifier (ASI)	CY7C611 Address Space Identifier (ASI)	Address Space
00001000 (08 H)	000 (0 H)	User Instruction
00001010 (0A H)	010 (2H)	User Data
00001001 (09 H)	001 (1 H)	Supervisor Instruction
00001011 (0B H)	011 (3 H)	Supervisor Data

6.2.1.4 BHOLD—Bus Hold (input)

BHOLD is asserted when an external bus master wants control of the data bus. Assertion of this signal will freeze the processor pipeline, so after deassertion of BHOLD, external logic must guarantee that the data at all inputs to the CY7C601/611 is the same as it was before BHOLD was asserted. This signal is tested on the falling edge (midpoint) of a cycle and must be valid and stable at the processor for the duration of the specified set-up time prior to the falling edge of CLK. All HOLD signals are latched in the CY7C601/611 (transparent latch with clock high) before they are used. Because MDS and MEXC are recognized by the CY7C601/611 but do not revert to the previous state as in the case of MDS and MEXC assertion with MHOLD active, BHOLD should only be used for bus access requests by an external device. BHOLD should not be asserted when LOCK is asserted.

6.2.1.5 COE—Control Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for SIZE<1:0>, RD, WE, WRT, LOCK, LDST, and DXFER outputs, and is the normal condition. Deassertion of COE three-states these output drivers and should only be done when the bus is granted to another bus master (i.e., when either BHOLD or MHOLDA/B is asserted).

6.2.1.6 D<31:0>—Data Bus (bidirectional)

These pins form a 32-bit bidirectional data bus that serves as the interface between the Integer Unit and memory. The data bus is only driven by the CY7C601/611 during the execution of integer store instructions and the store cycle of atomic-load-store instructions. Similarly, the CY7C602 FPU drives the data bus only during the execution of floating-point store instructions.

Store data is sent out unlatched and must be latched externally before it is used. Once latched, store data is valid during the second data cycle of a store single access, the second and third data cycle of a store double access, and the third data cycle of an atomic-load-store access.

Alignment for load and store instructions is performed by the processor. Doublewords are aligned on 8-byte boundaries, words on 4-byte boundaries, and halfwords on 2-byte boundaries. If a doubleword, word, or halfword load or store instruction generates an improperly aligned address, a memory address not aligned trap will occur. Instructions and operands are always expected to reside in a 32-bit wide memory. D<31> corresponds to the most significant bit of the most significant byte of a 32-bit word going to or from memory.

6.2.1.7 DOE—Data Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for the data bus, D<31:0>, and is the normal condition. Deassertion of DOE three-states the data bus output drivers and should only be done when the bus is granted to another bus master (i.e., when either BHOLD or MHOLDA/B is asserted).

6.2.1.8 DXFER—Data Transfer (output) [Not available on CY7C611]

DXFER is used to differentiate between the addresses being sent out for instruction fetches and the addresses of data fetches. DXFER is asserted by the processor during the address cycles of all bus data transfer cycles, including both cycles of store single and all three cycles of store double and atomic load-store. DXFER is sent out unlatched and must be latched externally before it is used.

6.2.1.9 IFT—Instruction Cache Flush Trap (input) [Not available on CY7C611]

The state of this pin determines whether or not execution of the FLUSH instruction generates a trap. If IFT=0, then execution of FLUSH causes an illegal instruction trap. If IFT=1, then FLUSH executes like a NOP with no side effects.

6.2.1.10 INULL—Integer Unit Nullify Cycle (output)

The processor asserts INULL to indicate that the current memory access is being nullified. It is asserted in the same cycle in which the address being nullified is active (though no longer on the address bus, the address is held in the external address latches). INULL is used to prevent a cache miss (in systems with cache memory) and to disable memory exception generation for the current memory access. This means that MDS and MEXC should not be asserted for a memory access in which INULL=1. INULL is a latched output and should not be latched externally. If a floating-point unit (FPU) or coprocessor is present in the system, INULL should be ORed with the FNULL and CNULL signals to generate a final NULL signal.

INULL is asserted under the following conditions:

1. During the second data cycle of any store instruction (including atomic load-store) to nullify the second occurrence of the store address.
2. On all traps, to nullify the third instruction fetch after the trapped instruction. For reset, it nullifies the error-producing address.
3. On a load in which the hardware interlock is activated.
4. JMPL and RETT instructions.

6.2.1.11 LDST—Atomic Load-Store (output)

This signal is used to identify an atomic load-store to the system and is asserted by the Integer Unit during all the data cycles (the load cycle and both store cycles) of atomic load-store instructions. LDST is sent out unlatched and must be latched externally before it is used.

6.2.1.12 LOCK—Bus Lock (output)

LOCK is asserted by the processor when it needs to retain control of the bus (address and data) for multiple cycle transactions (load double, store single and double, atomic load-store). The bus will not be granted to another bus master as long as LOCK is asserted. Note that BHOLD should not be asserted in the processor clock cycle that follows a cycle in which LOCK is asserted. LOCK is sent out unlatched and must be latched externally before it is used.

6.2.1.13 MDS—Memory Data Strobe (input)

MDS is asserted by the memory system to enable the clock to the Integer Unit's instruction register (during an instruction fetch) or to the load result register (during a data fetch) while the pipeline is frozen with an

MHOLDA/B. In a system with cache, MDS is used to signal the processor when the missed data (cache miss) is ready on the data bus. In a system with slow memories, MDS tells the processor when the read data is available on the bus. During a cache line replacement, MDS may be asserted anywhere within the MHOLD cycle and deasserted before MHOLD is released. For example, if a cache miss occurs on word 2 of a 4-word cache line, MDS should only be driven active while word 2 is being replaced in the cache.

MDS is also used to strobe in the MEXC memory exception signal. MDS may only be asserted when the pipeline is frozen with MHOLDA/B. The CY7C601/611 samples MDS with an on-chip transparent latch before it is used.

6.2.1.14 MEXC—Memory Exception (input)

Assertion of this signal by the memory system initiates an instruction access exception or data access exception trap and indicates to the CY7C601/611 that the memory system was unable to supply a valid instruction or data. If MEXC is asserted during an instruction fetch cycle, it generates an instruction access exception trap. If asserted during a data cycle, it generates a data access exception trap.

MEXC is used as a qualifier for the MDS signal, and must be asserted when both MHOLDA/B and MDS are already asserted. If MDS is applied without MEXC, the CY7C601/611 accepts the contents of the data bus as valid. If MEXC accompanies MDS, an exception is generated and the data bus content is ignored.

MEXC is latched in the processor on the rising edge of CLK and is used in the following cycle. MEXC must be deasserted in the same clock cycle in which MHOLDA/B is deasserted.

6.2.1.15 MHOLD(A/B)—Memory Holds (inputs)

MHOLDA is used to freeze the clock to both the integer and floating-point units during a cache miss (for systems with cache memory) or when accessing a slow memory. The processor pipeline is frozen while MHOLDA is asserted and the memory subsystem interface signals (see *Table 6–2* for a list of these signals), except for INULL, revert to and maintain the value they had at the rising edge of the clock in the cycle in which MHOLDA was asserted. This signal is tested on the falling edge (midpoint) of a cycle and must be valid and stable at the processor for the duration of the specified set-up time prior to the falling edge of CLK.

MHOLDB behaves in the same fashion as MHOLDA, and either can be used to stop the processor during a cache miss or memory exception. The pipeline is actually frozen by a "final" hold signal that is the logical OR of all hold signals (MHOLDA, MHOLDB, and BHOLD). All HOLD signals are latched in the CY7C601/611 (transparent latch with clock high) before they are used.

Note that MHOLD must be driven HIGH while RESET is LOW.

6.2.1.16 RD—Read Access (output)

RD is sent out during the address portion of an access to specify whether the current memory access is a read (RD=1) or a write (RD=0) operation. RD is set to "0" only during the address cycles of store instructions. For atomic load-store instructions, RD is "1" during the load address cycle and "0" during the two store address cycles. It is sent out unlatched by the Integer Unit and must be latched externally before it is used.

RD is used in conjunction with SIZE<1:0>, ASI<7:0>, and LDST to determine the type and to check the read/write access rights of bus transactions. It may also be used to turn off the output drivers of data RAMs during a store operation.

6.2.1.17 SIZE<1:0>—Bus Transaction Size (outputs)

The coding on these pins specifies the size of the data being transferred during an instruction or data fetch. The value of the size bits during a given cycle relates only to the memory address which appears on pins

A<31:0> simultaneously with the size outputs. It does not apply to data which may be on the data bus during that same cycle.

Size bits are sent out unlatched and must be latched external to the CY7C601/611 before they are used. SIZE<1:0> remains valid during the data address cycles of loads, stores, load doubles, store doubles, and atomic load-stores. Encoding of the size bits is shown in *Table 6-4*. For example, during an instruction fetch, SIZE<1:0> is set to "10," because all instructions are 32 bits long. For doubleword instructions, SIZE<1:0> is "11" for all data address cycles.

Table 6–4. SIZE Bit Encoding

SIZE<1>	SIZE<0>	Data Transfer Type
0	0	Byte
0	1	Halfword
1	0	Word
1	1	Word (Load/Store Double)

6.2.1.18 $\overline{\text{WE}}$—Write Enable (output)

$\overline{\text{WE}}$ is asserted by the Integer Unit during the cycle in which the store data is on the data bus. For a store single instruction, this is during the second store address cycle; the second and third store address cycles of store double instructions, and the third load-store address cycle of atomic load-store instructions. It is sent out unlatched and must be latched externally before it is used. To avoid writing to memory during memory exceptions, $\overline{\text{WE}}$ must be externally qualified by the $\overline{\text{MHOLDA/B}}$ signals.

6.2.1.19 WRT—Advanced Write (output)

WRT is an early write signal, asserted by the processor during the first store address cycle of integer single or double store instructions, the first store address cycle of floating-point single or double store instructions, and the second load-store address cycle of atomic load-store instructions. WRT is sent out unlatched and must be latched externally before it is used.

6.2.2 Floating-Point/Coprocessor Interface Signals

The IU incorporates a dedicated group of pins that act as direct-connect interfaces between the Integer Unit and both the floating-point unit and the coprocessor. Using these connections, no external circuits are required to interface the IU to the FPU and coprocessor. The interfaces consist of the following signals:

6.2.2.1 CCC<1:0>—Coprocessor Condition Codes (input) [Not available on CY7C611]

These lines represent the current condition code bits from the coprocessor state register (CSR), qualified by the CCCV signal. When CCCV=1, these bits are valid. During the execution of a CBccc instruction, the processor uses CCC<1:0> to determine whether or not to take the branch. These bits are latched by the processor before they are used.

6.2.2.2 CCCV—Coprocessor Condition Codes Valid (input) [Not available on CY7C611]

This signal is a specialized hold used to synchronize coprocessor compare instructions with coprocessor branch instructions. It is asserted (the normal condition) whenever the CCC<1:0> bits are valid. A coproces-

sor would deassert CCCV (CCCV=0) as soon as a coprocessor compare instruction enters the coprocessor queue, unless an exception is detected. Deasserting CCCV freezes the Integer Unit pipeline, preventing any further compares from entering the pipeline. CCCV is reasserted when the compare is completed and the coprocessor condition codes are valid, thus ensuring that the condition codes match the proper compare instruction. CCCV is latched in the CY7C601 before it is used.

6.2.2.3 $\overline{CEXC}$—Coprocessor Exception (input) [Not available on CY7C611]

$\overline{CEXC}$ is used to signal the Integer Unit that a coprocessor exception has occurred. $\overline{CEXC}$ must remain asserted until the CY7C601 takes the trap and acknowledges the FPU exception via the CXACK signal. Although coprocessor exceptions can occur at any time, they are taken by the CY7C601 only during the execution of a subsequent CPop, a CBfcc instruction, or a coprocessor load or store instruction. A coprocessor implementation should deassert $\overline{CHOLD}$ if it detects an exception while $\overline{CHOLD}$ is asserted. In such a case, $\overline{CEXC}$ should be asserted one cycle before $\overline{CHOLD}$ is deasserted. $\overline{CEXC}$ is latched in the CY7C601 before it is used.

6.2.2.4 $\overline{CHOLD}$—Coprocessor Hold (input) [Not available on CY7C611]

This signal is asserted by the coprocessor if a situation arises in which it cannot continue execution. The coprocessor checks all dependencies in the Decode stage of the instruction and asserts $\overline{CHOLD}$ (if necessary) in the next cycle. If the Integer Unit receives a $\overline{CHOLD}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{CHOLD}$ are resolved, the coprocessor deasserts $\overline{CHOLD}$, releasing the instruction pipeline. Because $\overline{MDS}$ and $\overline{MEXC}$ are recognized by the CY7C601/611 but do not revert to the previous state as in the case of $\overline{MDS}$ and $\overline{MEXC}$ assertion with $\overline{MHOLD}$ active, $\overline{MDS}$ and $\overline{MEXC}$ should not used with $\overline{CHOLD}$ to strobe exceptions into the CY7C601. $\overline{CHOLD}$ is latched in the CY7C601 before it is used.

The conditions under which the coprocessor asserts $\overline{CHOLD}$ are implementation dependent.

6.2.2.5 CINS1—Coprocessor Instruction in Buffer 1 (output) (Not available on CY7C611)

CINS1 is asserted by the Integer Unit during the Decode stage of the coprocessor instruction that is in the D1 buffer of the coprocessor chip. The coprocessor uses this signal to begin decoding and execution of the D1 instruction, and to latch it into its Execute-stage register. CINS1 and CINS2 are never asserted in the same cycle.

6.2.2.6 CINS2—Coprocessor Instruction in Buffer 2 (output) (Not available on CY7C611)

CINS2 is asserted by the Integer Unit during the Decode stage of the coprocessor instruction that is in the D2 buffer of the coprocessor chip. The coprocessor uses this signal to begin decoding and execution of the D2 instruction, and to latch it into its Execute-stage register. CINS1 and CINS2 are never asserted in the same cycle.

6.2.2.7 $\overline{CP}$—Coprocessor Unit Present (input) [Not available on CY7C611]

When pulled low, $\overline{CP}$ indicates that a coprocessor is available to the system. It is normally pulled up to VDD through a resistor, and then grounded by connection to the coprocessor. The Integer Unit will generate a cp disabled trap if $\overline{CP}$=1 during the execution of an CPop, CBfcc, or coprocessor load or store instruction.

6.2.2.8 CXACK—Coprocessor Exception Acknowledge (output) [Not available on CY7C611]

CXACK is asserted by the Integer Unit to inform the coprocessor that a trap has been taken for the currently asserted $\overline{CEXC}$ signal. Receipt of the asserted CXACK causes the coprocessor to deassert $\overline{CEXC}$, which in turn causes the to deassert CXACK. CXACK is a latched output and should not be latched externally.

6.2.2.9 FCC<1:0>—Floating-Point Condition Codes (input)

These lines represent the current condition code bits from the FPU's floating-point state register (FSR), qualified by the FCCV signal. When FCCV=1, these bits are valid. During the execution of an FBfcc instruction, the processor uses FCC<1:0> to determine whether or not to take the branch. These bits are latched by the processor before they are used.

6.2.2.10 FCCV—Floating-Point Condition Codes Valid (input)

This signal is a specialized hold used to synchronize FPU compare instructions with floating-point branch instructions. It is asserted (the normal condition) whenever the FCC<1:0> bits are valid. The CY7C602 deasserts FCCV (FCCV=0) as soon as a floating-point compare instruction enters the floating-point queue, unless an exception is detected (see *Section 7.2.1.2.1*). Deasserting FCCV freezes the Integer Unit pipeline, preventing any further compares from entering the pipeline. FCCV is reasserted when the compare is completed and the floating-point condition codes are valid, thus ensuring that the condition codes match the proper compare instruction. FCCV is latched in the CY7C601/611 before it is used.

6.2.2.11 $\overline{\text{FEXC}}$—Floating-Point Exception (input)

$\overline{\text{FEXC}}$ is used to signal the Integer Unit that a floating-point exception has occurred. $\overline{\text{FEXC}}$ must remain asserted until the CY7C601/611 takes the trap and acknowledges the FPU exception via the FXACK signal. Although floating-point exceptions can occur at any time, they are taken by the CY7C601/611 only during the execution of a subsequent FPop, an FBfcc instruction, or a floating-point load or store instruction. The CY7C602 deasserts $\overline{\text{FHOLD}}$ if it detects an exception while $\overline{\text{FHOLD}}$ is asserted. In such a case, $\overline{\text{FEXC}}$ is asserted one cycle before $\overline{\text{FHOLD}}$ is deasserted. FEXC is latched in the CY7C601/611 before it is used.

6.2.2.12 $\overline{\text{FHOLD}}$—Floating-Point Hold (input)

This signal is asserted by the CY7C602 if a situation arises in which the FPU cannot continue execution. The FPU checks all dependencies in the Decode stage of the instruction and asserts $\overline{\text{FHOLD}}$ (if necessary) in the next cycle. If the Integer Unit receives an $\overline{\text{FHOLD}}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{\text{FHOLD}}$ are resolved, the FPU deasserts $\overline{\text{FHOLD}}$, releasing the instruction pipeline. $\overline{\text{FHOLD}}$ is latched in the CY7C601/611 before it is used.

An $\overline{\text{FHOLD}}$ is asserted if (1) the FPU encounters an STFSR instruction with one or more FPops pending in the queue, (2) if either a resource or operand dependency exists between the FPop being decoded and any FPops already being executed, or (3) if the floating-point queue is full.

6.2.2.13 FINS1—Floating-Point Instruction In Buffer 1 (output)

FINS1 is asserted by the Integer Unit during the Decode stage of the floating-point instruction that is in the D1 buffer of the floating-point unit (see *Section 7.2*). The FPU uses this signal to begin decoding and execution of the D1 instruction, and to latch it into its Execute-stage register. FINS1 and FINS2 are never asserted in the same cycle and both are ignored if (1) FLUSH is asserted, (2) any HOLD is asserted, (3) or if FCCV or CCCV is deasserted.

6.2.2.14 FINS2—Floating-Point Instruction In Buffer 2 (output)

FINS2 is asserted by the Integer Unit during the Decode stage of the floating-point instruction that is in the D2 buffer of the floating-point unit (see *Section 3.1*). The FPU uses this signal to begin decoding and execu-

tion of the D2 instruction, and to latch it into its Execute-stage register. FINS1 and FINS2 are never asserted in the same cycle and both are ignored if (1) FLUSH is asserted, (2) any HOLD is asserted, (3) or if FCCV or CCCV is deasserted.

6.2.2.15 FLUSH—Floating-Point/Coprocessor Instruction Flush (output)

This signal is asserted by the Integer Unit whenever it takes a trap. FLUSH is used by the FPU (or coprocessor) to flush the instructions in its instruction caches. These instructions, as well as the instructions annulled in the CY7C601/611's pipeline, are restarted after the trap handler is finished. If the trap was not caused by a floating-point (or coprocessor) exception, instructions already in the floating-point (or coprocessor) queue may continue their execution. If the trap was caused by a floating-point (or coprocessor) exception, the fp (or cp) queue must be emptied before the FPU (coprocessor) can resume execution.

6.2.2.16 $\overline{FP}$—Floating-point Unit Present (input)

When pulled low, $\overline{FP}$ indicates that a floating-point unit is available to the system. It is normally pulled up to VDD through a resistor, and then grounded by connection to the FPU. The Integer Unit will generate an fp disabled trap if $\overline{FP}$=1 during the execution of an FPop, FBfcc, or floating-point load or store instruction.

6.2.2.17 FXACK—Floating-Point Exception Acknowledge (output)

FXACK is asserted by the Integer Unit to inform the floating-point unit that a trap has been taken for the currently asserted $\overline{FEXC}$ signal. Receipt of the asserted FXACK causes the FPU to deassert $\overline{FEXC}$, which in turn causes the CY7C601/611 to deassert FXACK. FXACK is a latched output and should not be latched externally.

6.2.2.18 INST—Instruction Fetch (output)

The INST signal is asserted by the Integer Unit whenever a new instruction is being fetched. It is used by the floating-point unit or coprocessor to latch the instruction currently on the data bus into an FPU or coprocessor instruction cache. SPARC-compatible floating-point units and coprocessors have two instruction caches (D1 and D2) to save the last two fetched instructions (see *Section 7.2*). When INST is asserted, a new instruction enters buffer D1 and the instruction that was in D1 moves to buffer D2. INST is a latched output and should not be latched externally.

6.2.3 Interrupt and Control Signals

The following signals are used by the Integer Unit to control and to receive input from external events.

6.2.3.1 $\overline{ERROR}$—Error State (output)

This signal is asserted when the Integer Unit enters the error mode state. This happens if a synchronous trap occurs while traps are disabled (the PSR's ET bit =0). Before it enters the error mode state, the CY7C601/611 saves the PC and nPC and sets the trap type (tt) for the trap causing the error mode into the TBR. It then asserts the $\overline{ERROR}$ signal and halts. The only way to restart a processor which is in the error mode state is to trigger a reset by asserting the $\overline{RESET}$ signal.

6.2.3.2 FPSYN—Floating-point Synonym Mode (input)

This is a mode pin which will be used to allow execution of additional instructions in future designs. For the CY7C601/611, it should be kept grounded.

6.2.3.3 INTACK—Interrupt Acknowledge (output)

INTACK (interrupt acknowledge) is a latched output that is asserted by the Integer Unit when an external interrupt is *taken*, not when it is sampled and latched.

6.2.3.4 IRL<3:0>—Interrupt Request Level (input)

The state of these pins defines the external interrupt level (IRL). IRL<3:0>=0000 indicates that no external interrupts are pending and is the normal state of the IRL pins. IRL<3:0>=1111 signifies a nonmaskable interrupt. All other interrupt levels are maskable by the processor interrupt level (PIL) field of the processor state register (PSR). The Integer Unit uses two on-chip synchronizing latches to sample these signals, and a given level must remain valid for two consecutive cycles to be recognized. External interrupts should be latched and prioritized by external logic before they are passed to the CY7C601/611. Logic must also keep an interrupt valid until it is taken and acknowledged. External interrupts can be acknowledged by system software or by the CY7C601/611's interrupt acknowledge (INTACK) signal.

6.2.3.5 $\overline{RESET}$—Integer Unit Reset (input)

Assertion of this pin will reset the Integer Unit. $\overline{RESET}$ must be asserted for a minimum of eight processor clock cycles. After $\overline{RESET}$ is deasserted, the Integer Unit starts fetching from address 0. $\overline{RESET}$ is latched by the CY7C601/611 before it is used.

6.2.3.6 $\overline{TOE}$—Test Mode Output Enable (input)

When *deasserted*, this signal will three-state all Integer Unit output drivers. Thus, in normal operation, this pin should always be asserted (tied to ground). Deassertion of $\overline{TOE}$ isolates the CY7C601/611 from the system for debugging purposes.

6.2.4 Power and Clock Signals

The signals listed below provide clocking and power to the Integer Unit.

6.2.4.1 CLK—Clock (input)

CLK is a 50%-duty-cycle clock used for clocking the Integer Unit's pipeline registers. The rising edge of CLK defines the beginning of each pipeline stage and a processor cycle is equal to a full clock cycle.

6.2.4.2 VCCO, VCCI, VCCT—Power (inputs)

These pins provide +5V power to various sections of the processor. Power is supplied on three different buses to provide clean, stable power to each section: output drivers, main internal circuitry, and the input circuits. VCCO pins supply the output driver bus; VCCI pins supply main internal circuitry bus; and VCCT pins supply the input circuit bus. See *Section 7.1* for pin identification.

6.2.4.3 VSSO, VSSI, VSST—Ground (inputs)

These pins provide ground return for the power signals. Ground is supplied on three different buses to match the power signals to each section: VSSO pins for the output driver bus; VSSI pins for the main internal circuitry bus; and VSST pins for the input circuit bus. See *Section 7.1* for pin identification.

6.3 Pipeline And Instruction Execution Timing

One of the major contributing factors to the CY7C601/CY7C611's high performance is an instruction execution rate approaching one instruction per clock cycle. To achieve that rate of execution, the CY7C601/CY7C611 employs a four-stage instruction pipeline that permits multiple instructions to be operated on at the same time.

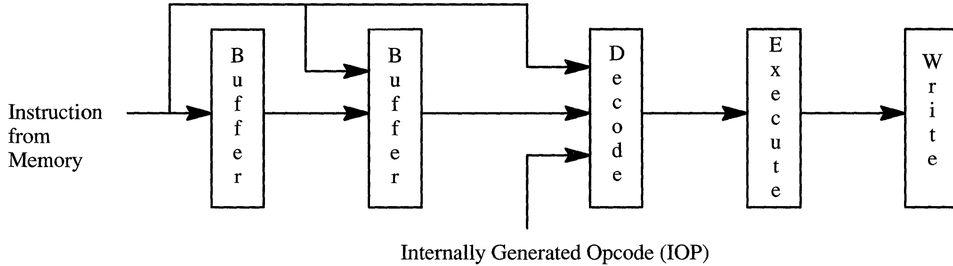


Figure 6–4. Processor Instruction Pipeline

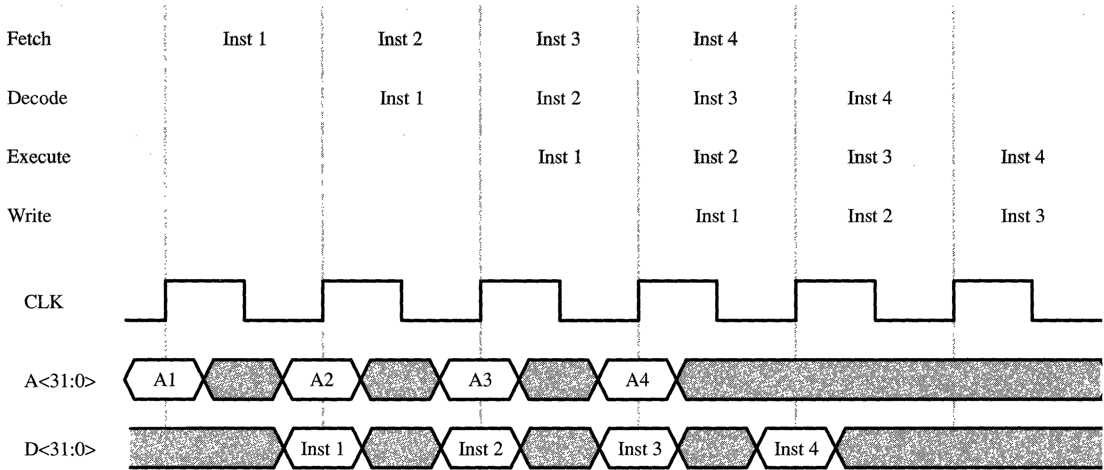


Figure 6–5. Pipeline with All Single-Cycle Instructions

6.3.1 Stages

Instruction execution is broken into four stages corresponding to the stages of the pipeline:

1. **Fetch**—The processor outputs the instruction address to fetch the instruction.
2. **Decode**—The instruction is placed in the instruction register and decoded. The processor reads the operands from the register file and computes the next instruction address.
3. **Execute**—The processor executes the instruction and saves the results in temporary registers. Pending traps are prioritized and internal traps taken during this stage.
4. **Write**—If no trap is taken, the processor writes the result to the destination register.

All four stages operate in parallel, working on up to four different instructions at a time. A basic "single-cycle" instruction enters the pipeline and completes in four cycles. By the time it reaches the Write stage, three more instructions have entered and are moving through the pipeline behind it. So, after the first four cycles, a single-cycle instruction exits the pipeline and a single-cycle instruction enters the pipeline on every cycle (see *Figure 6–5*).

Of course, a "single-cycle" instruction actually takes four cycles to complete, but they are called single cycle because with this type of instruction the processor can complete one instruction per cycle after the initial four-cycle delay.

6.3.1.1 Internal Opcodes

Instructions that require extra cycles automatically insert internal opcodes (IOPs) into the Decode stage as they move into the Execute stage. These internal opcodes are unique to the instruction that generates them. They move all the way through the pipeline, performing functions specific to the instruction that created them. For example, in *Figure 6–6*, the data load in cycle four can be thought of as the fetch for the IOP that starts in cycle three; together they make a complete four-cycle instruction that balances out the pipeline. JMPL and RETT also generate an IOP, but have no external data cycle.

Multicycle instructions may generate up to three IOPs to complete execution. *Table 6–5* lists the instructions that require IOPs and the number generated.

Because instructions continue to be fetched even though IOPs occupy the Decode stage, a two-stage prefetch buffer is used to hold instructions until they can move into the Decode stage (see *Figure 6–4*). This enables the processor to fully utilize the data bus bandwidth and still keep the pipeline full. Only two buffers are required because a maximum of two cycles are available for instruction fetching for any multi-cycle instruction.

Table 6–5. Internally Generated Opcodes

Instruction	Number of Internal Opcodes
Single Loads	1
Double Loads	2
Single Stores	2
Double Stores	3
Atomic Load-Store	3
Jump	1
Return from Trap	1

6.3.2 Multicycle Instructions

Multicycle instructions are those that take more than four cycles (one bus cycle plus the three pipeline cycles) to complete. A double-cycle instruction takes five cycles (two bus cycles), a triple-cycle instruction takes six cycles (three bus cycles), and so on.

In most cases, the extra cycles required by multicycle instructions result from data bus usage (e.g., a data load or store to memory) that prevents the processor from fetching the next instruction during those cycles. In *Figure 6–6*, the Fetch of instruction Inst 3 is delayed by one cycle for the data load, and in *Figure 6–7*, the store sequence delays the Inst 3 Fetch by two cycles.

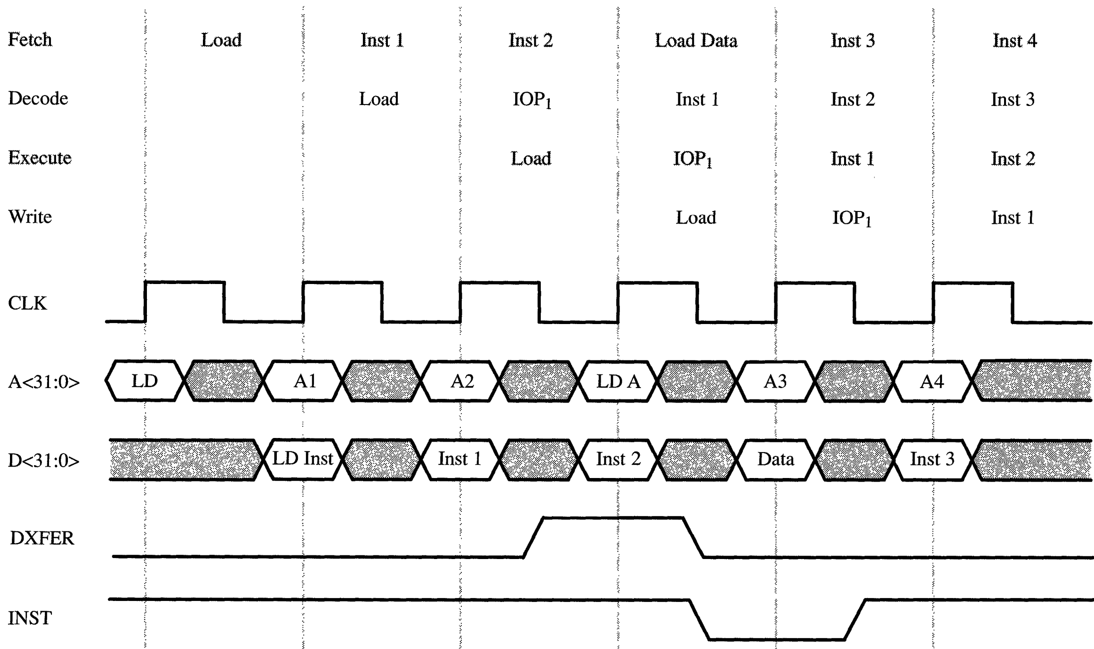


Figure 6–6. Pipeline with One Double-Cycle Instruction (Load)

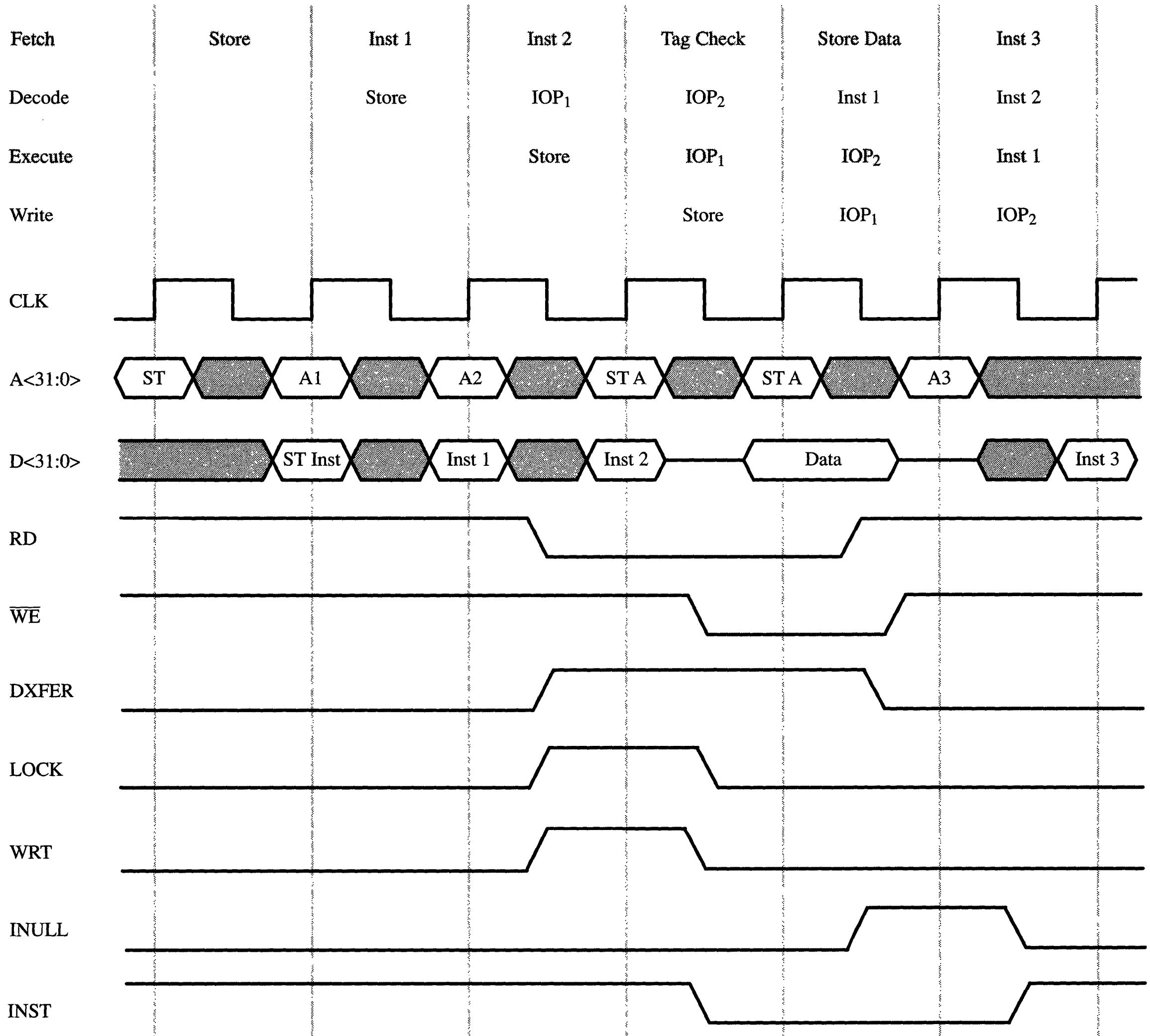


Figure 6–7. Pipeline with One Triple-Cycle Instruction (Store)

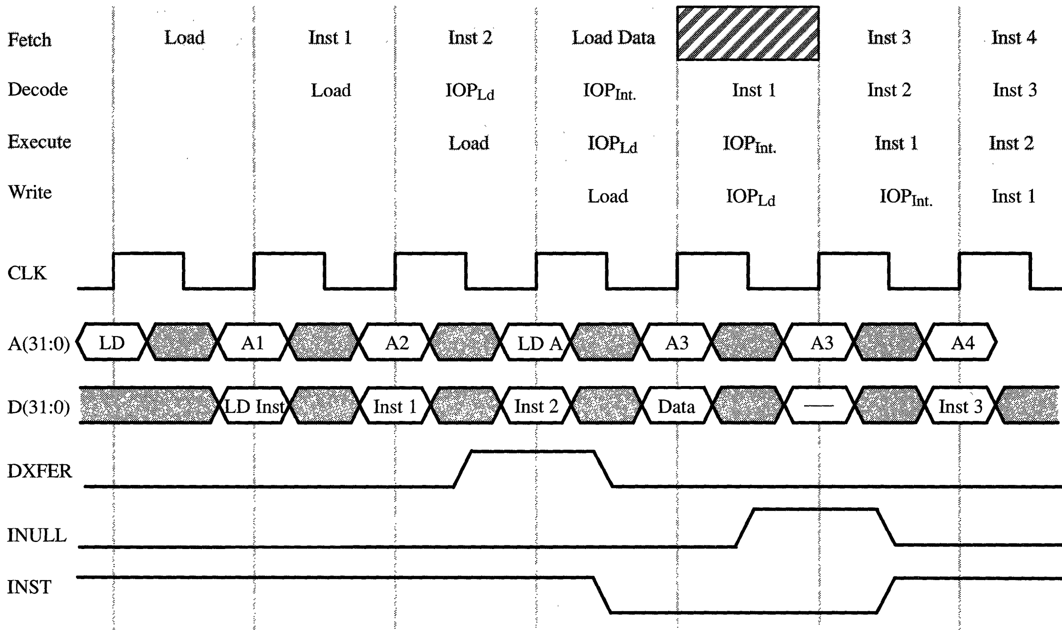


Figure 6–8. Pipeline with Hardware Interlock (Load)

6.3.2.1 Register Interlocks

The pipeline holds several instructions at any given time, so it is possible that an instruction may try to use the contents of a particular register which is in the process of being updated by a previous instruction. Special bypass paths in the pipeline of the CY7C601/CY7C611 make the correct data available to subsequent instructions for all internal register to register operations, but cannot solve the problem of loads to the registers from external memory. For this case, interlock hardware prevents an instruction following a load instruction from reading the register being loaded until the load is complete (see *Figure 6–8*). This also applies to a a CALL instruction with a delay slot instruction using r[15] and a JMPL with a delay slot instruction using the same register specified as the r[rd] of the JMPL. To maximize performance, compilers and assembly language programmers should avoid loads followed immediately by instructions using the loaded register's contents.

6.3.2.2 Branching

The CY7C601/CY7C611's delayed-control-transfer mechanism allows branches (taken or untaken) to occur without creating a bubble in the pipeline (see *Figure 6–9*). Special parallel hardware enables the processor to evaluate the condition codes and calculate the effective branch address during the Decode stage rather than the Execute stage, so that only one delay instruction is required between the branch and the target instruction (or the next instruction, if the branch is not taken). Refer to *Section 2.4.3.3* for a discussion on branching.

If the compiler or programmer cannot place an appropriate instruction in the delay instruction slot, the delay instruction can be annulled by setting the branch instruction's *a* bit. The result is shown in *Figure 6–10*.

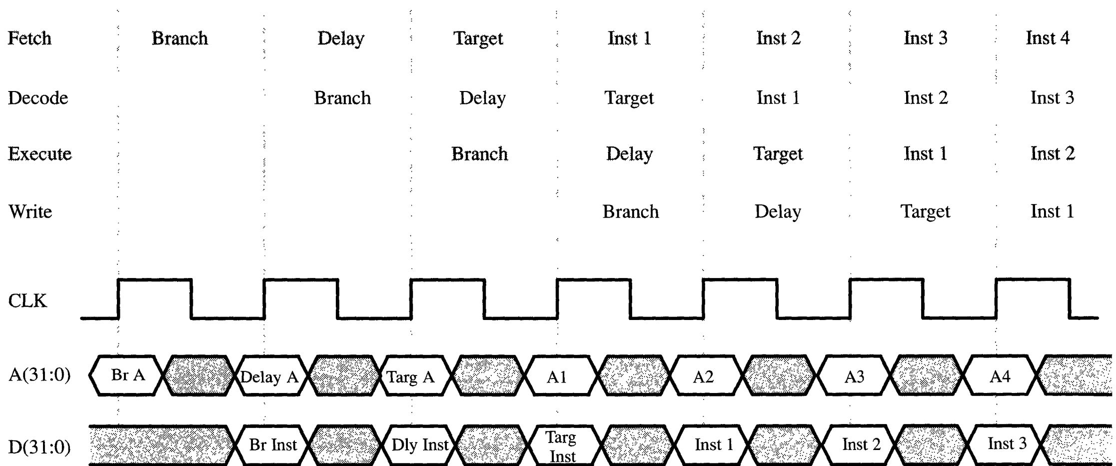


Figure 6–9. Pipeline During Branch Instruction

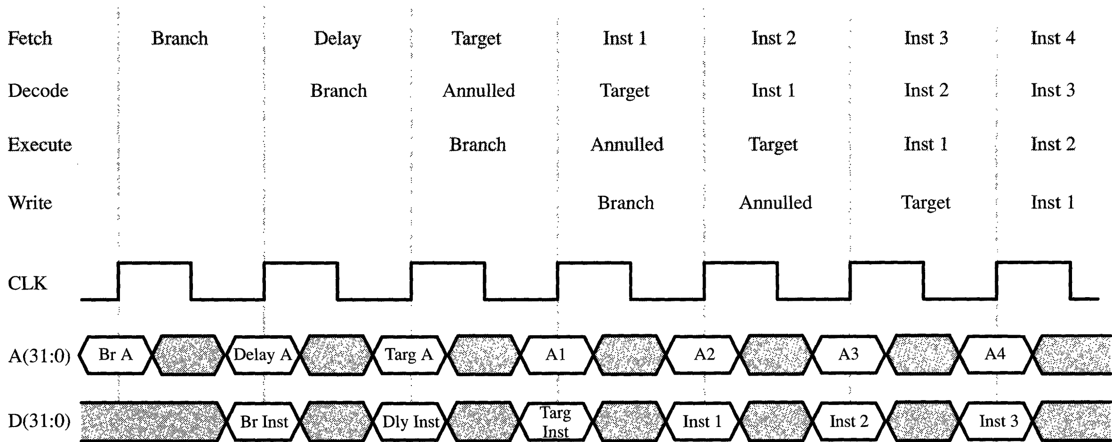


Figure 6–10. Branch with Annulled Delay Instruction

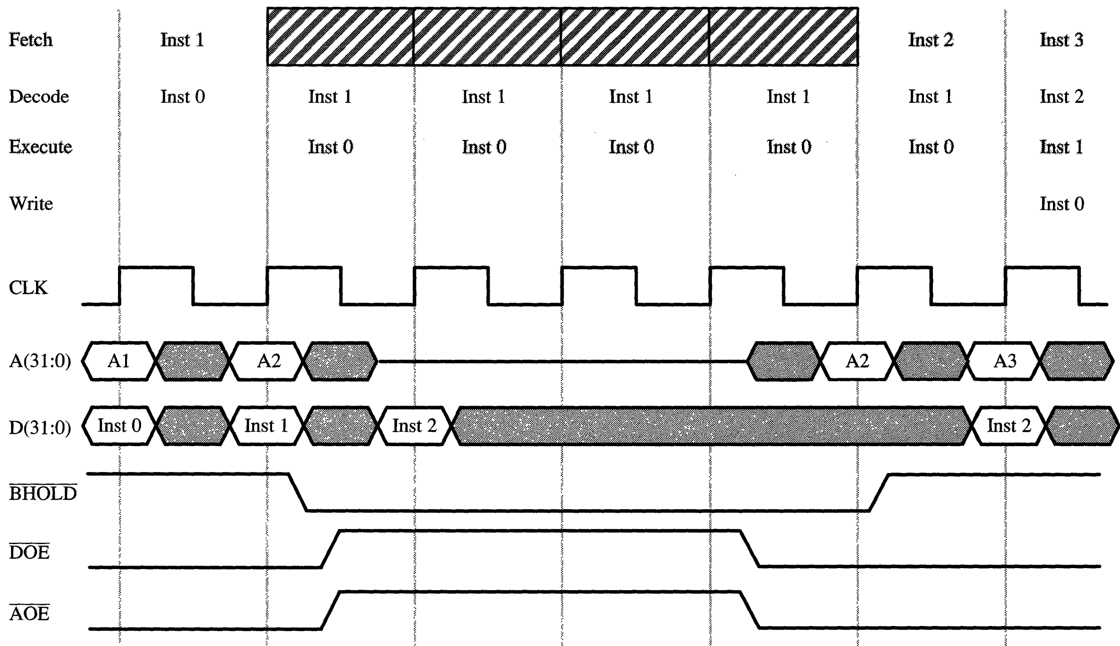


Figure 6–11. Pipeline Frozen During Bus Arbitration

6.3.3 Pipeline Freezes

Whenever the processor receives an externally generated hold input, such as $\overline{\text{MHOLDA/B}}$ or $\overline{\text{BHOLD}}$, the instruction pipeline is frozen. How long it is frozen depends on the type of hold and the external hardware generating the hold. *Figure 6–11* shows the pipeline frozen by a $\overline{\text{BHOLD}}$ as the result of bus arbitration initiated by another bus master in the system.

6.3.4 Traps

Figure 6–12 shows the pipeline operation when an internally generated trap is taken. Instructions in the pipeline after detection of the trap are annulled and the first instruction of the trap target routine is executed in the fourth cycle following detection.

6.4 Bus Operation And Timing

This section covers standard and non-standard bus operations. Standard operations include instruction fetch, load integer, load double integer, load floating-point, load double floating-point, store integer, store double integer, store floating-point, store double floating-point, atomic load-store unsigned byte, and floating-point operations (FPops). Non-standard operations include bus arbitration, cache misses, exceptions, and the reset and error conditions. Coprocessor loads, coprocessor stores, and coprocessor operations are identical in timing to their floating-point counterpart, and are not repeated as a separate case in this section.

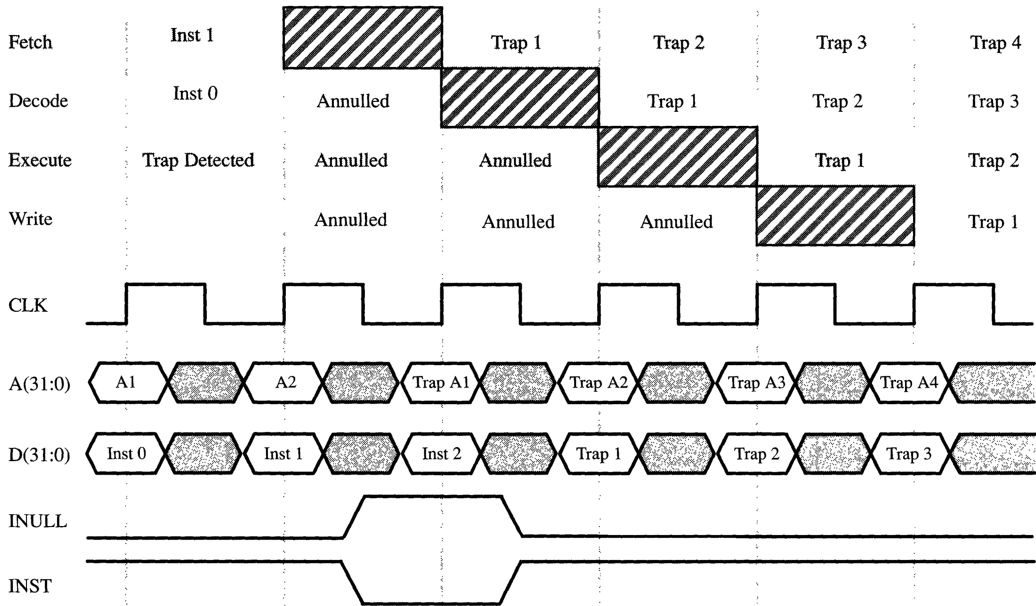


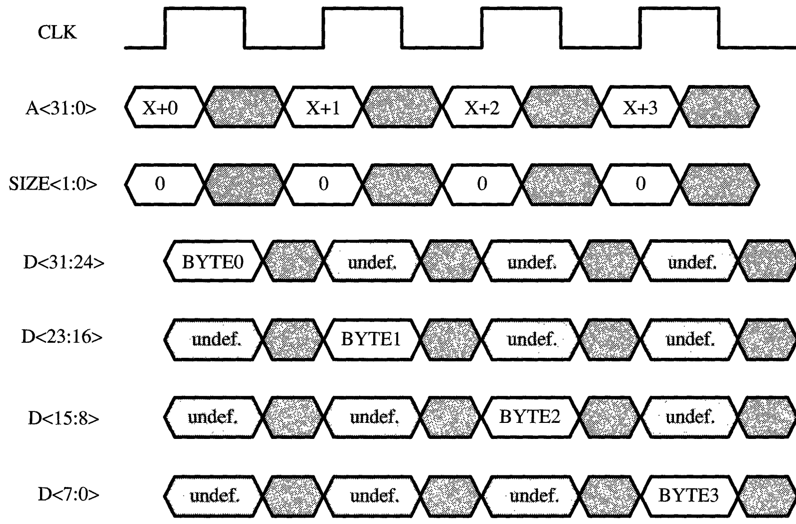
Figure 6–12. Pipeline Operation for Taken Trap (Internal)

Each of the following sections describes a type of bus transaction along with appropriate timing diagrams. The timing diagrams show multiple instructions being fetched for the pipeline. Instruction addresses are sent out in the cycle before the instruction fetch. Instruction fetch cycles begin with the instruction address latched by the memory at the beginning of the Fetch cycle and end with the instruction supplied by the memory. Instruction Decode begins with the latching of the instruction at rising clock edge of the cycle after the Fetch cycle. If the instruction is multi-cycle, or execution requires an interlock, IOPs are inserted into the pipeline at the Decode stage and propagate through the pipeline like a fetched instruction.

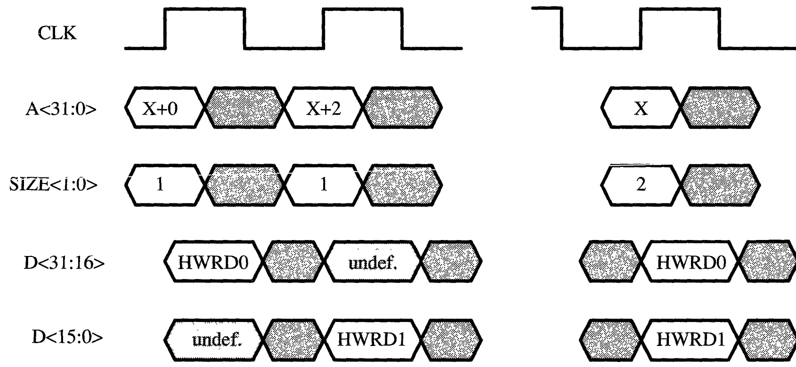
The cross-hatched areas shown in the traces are periods in which the signal is not guaranteed to be asserted or deasserted; in other words, undefined.

In general, signals are valid at the beginning of a cycle, i.e., on the rising edge of the clock. In support of the CY7C601/CY7C611's high-speed operation, many signals are sent out unlatched. Refer to *Section 6.2* for further details on CY7C601/CY7C611 signals.

The processor automatically aligns byte (and halfword) transfers. *Figure 6–13* shows the relationship between the data transferred during byte, halfword, and word operations and the pins of the data bus. For byte and halfword data transfers, the CY7C601/CY7C611 repeats the byte or halfword on each eight-bit or 16-bit section of the bus. In other words, the undefined portions of the bus illustrated in *Figure 6–13* are actually a repeat of the data driven onto the bus. However, this feature is not specified in the SPARC Architecture Reference, and may not be supported on other SPARC processors.



Byte Data Alignment



Halfword Data Alignment **Word Data Alignment**

X = word boundary address

Note: This illustration depicts data alignment and is not intended to illustrate a timing case.

Figure 6–13. Data Bus Contents During Data Transfers

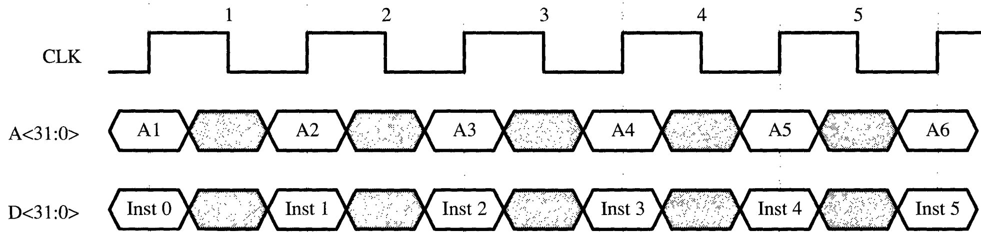


Figure 6–14. Instruction Fetch

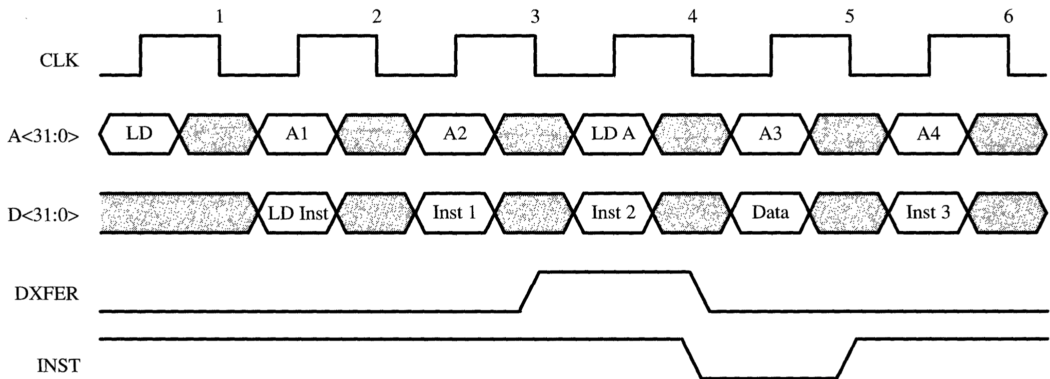


Figure 6–15. Load Single Integer Timing

6.4.1 Instruction Fetch

The instruction Fetch cycle is that cycle in which both the instruction address and the data (the instruction itself) are active on their respective buses (see *Figure 6–14*). The instruction address on A<31:0> is actually sent out in the previous cycle, but is held into the Fetch cycle. It should be latched externally. The instruction is returned on the data bus at the very end of the Fetch cycle and is held into the Decode cycle. It is latched into the on-chip instruction register at the beginning of the Decode cycle.

6.4.2 Load

Figure 6–15 shows the timing for a load single integer instruction. Because the bus is used for a data fetch in the fifth cycle, this is a double-cycle instruction. Note that DXFER is active in the cycle in which the load data address is sent out, while INST is inactive in the cycle in which the load data is on the data bus.

6.4.3 Load with Interlock

In a load with interlock situation, the instruction following the load tries to use the contents of the load's destination register before the load data is available. This requires the insertion of an IOP into the Decode

stage of the pipeline in the fourth cycle, which must be matched by a null bus cycle in the Fetch stage to balance the pipeline (see *Figure 6–16*).

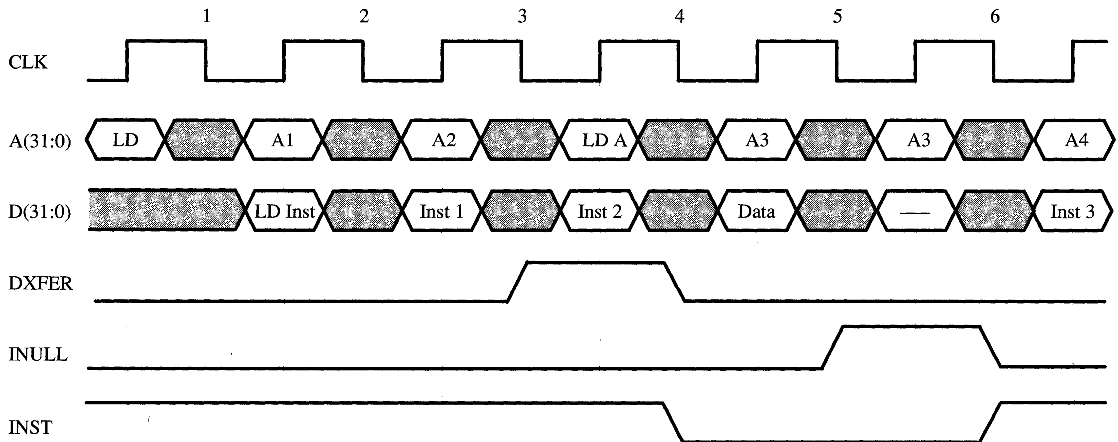


Figure 6–16. Load Single with Interlock Timing

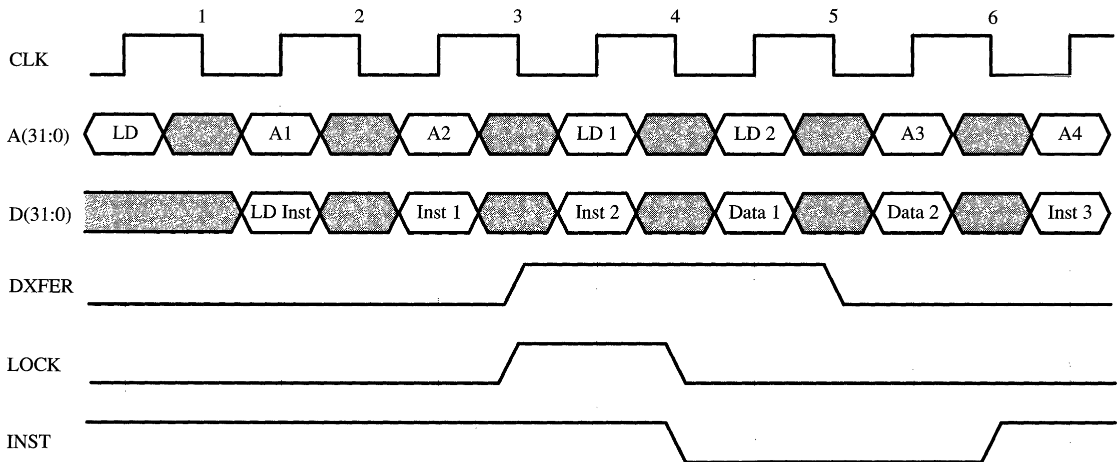


Figure 6–17. Load Double Integer Timing

6.4.4 Load Double

The timing for a load double integer is shown in *Figure 6–17*. The timing is essentially the same as a load single except for the additional data fetch in the fifth cycle. That makes load double a triple-cycle instruction. The most-significant word is fetched in cycle four and the least-significant word in cycle five. Note that the size bits are set to 11 during the address portion of both loads and that the bus is locked to allow the completion of both loads without interruption.

Load single and load double floating-point instructions look identical to their integer counterparts except that the FINS1/FINS2 signal is active for floating-point operations.

6.4.5 Store

Store transactions involve more bus activity than loads, as shown in the store single integer timing in *Figure 6–18*. Store single is a triple-cycle instruction because it includes an extra tag check cycle in which to check an external cache for the store address. This extra cycle also gives the processor and the memory system time to three-state the data bus and turn it around for the store. The store address is sent out again in the fifth cycle to complete the data transfer. Note that the store data is generated by the processor off the falling edge of CLK and is therefore only available at the very end of the first data cycle.

Note also that INULL is active during the second application of the store address. If there is a cache miss on the tag check cycle, INULL prevents an additional miss the second time the address is sent out in the store cycle. Because it is a triple cycle instruction, LOCK is asserted to retain control of the buses.

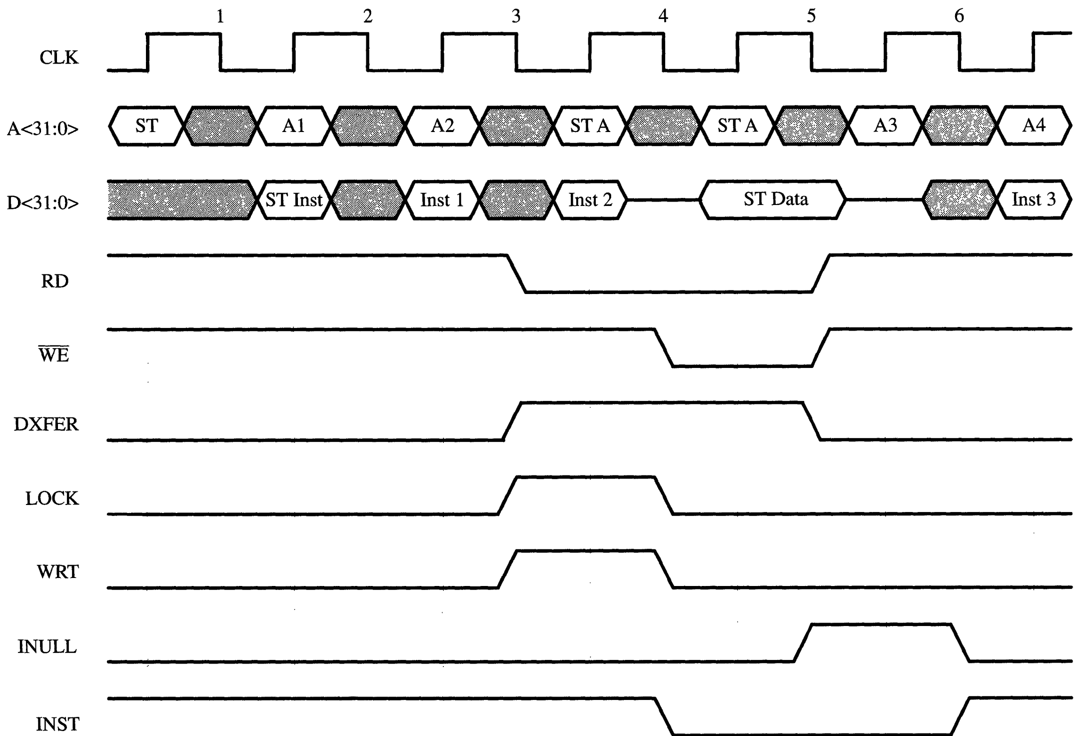


Figure 6–18. Store Single Integer Timing

6.4.6 Store Double

The timing for a store double integer is shown in *Figure 6–19*. The timing is essentially the same as store single except for the additional store cycle in the sixth cycle, making it a four-cycle instruction. The

most-significant word is stored in cycle five and the least-significant word in cycle six. Note that the size bits are set to 11 during the address portion of all three data cycles and that the bus is locked to allow the completion of both stores without interruption. INULL is not active for the address of the least-significant store because there cannot be a miss on this cycle if there was not one on the tag check cycle, unless the cache line is less than two words.

Store single and store double floating-point instructions look identical to their integer counterparts except that the FINS1/FINS2 signal is active for floating-point operations.

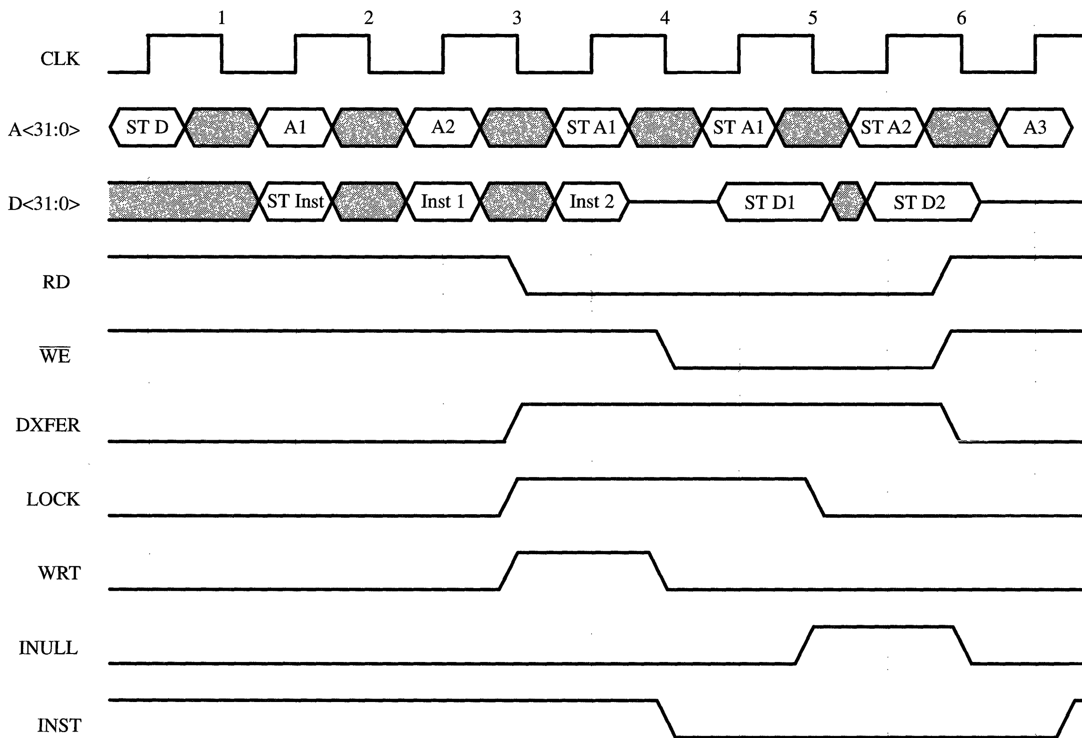


Figure 6–19. Store Double Integer Timing

6.4.7 Atomic Load-Store

Atomic transactions consist of two or more steps that are indivisible; once the sequence begins in the instruction pipeline, it cannot be interrupted. Because atomic operations are four-cycle instructions, the CY7C601/CY7C611 asserts LOCK for as long as necessary to make sure that no interruption occurs on the bus. *Figure 6–20* applies to the atomic operations load-store unsigned byte (LDSTUB, LDSTUBA) and word swap (SWAP, SWAPA). Note that, as with any store, INULL is active on the second occurrence of the store address.

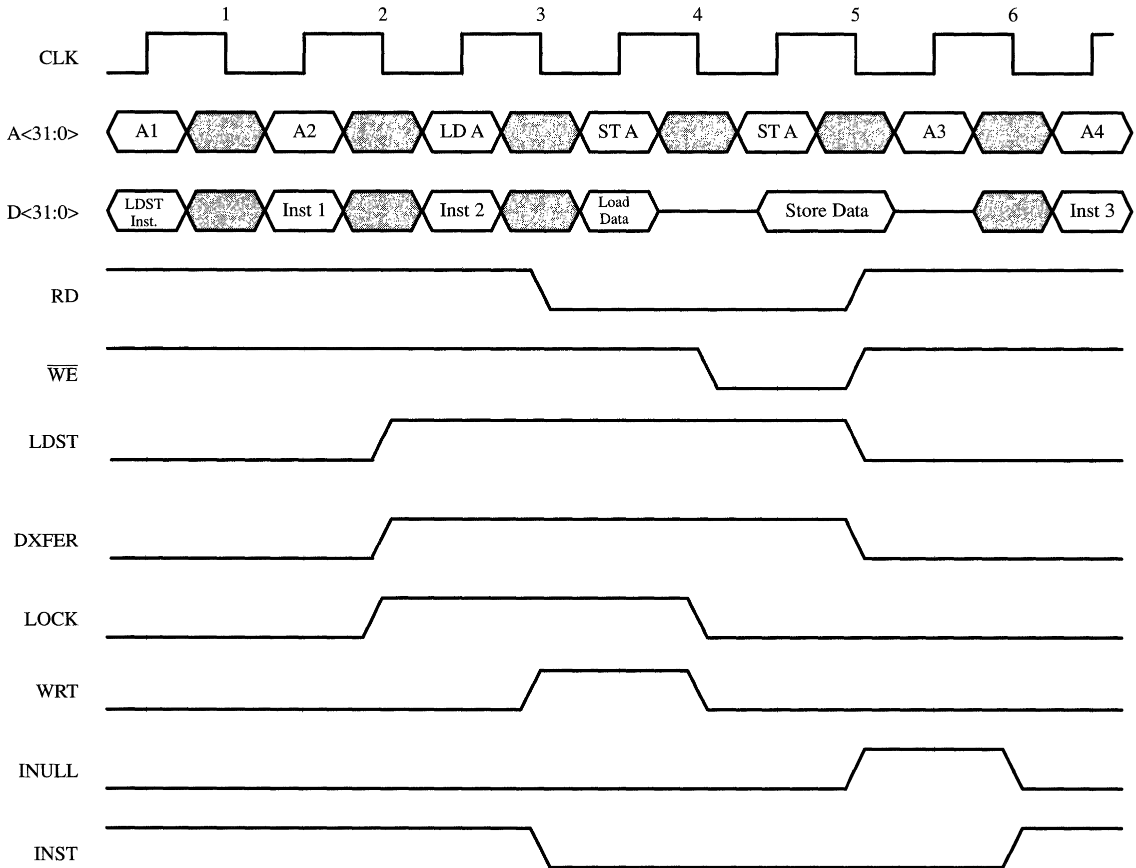


Figure 6–20. Atomic Load-Store Timing

6.4.8 Floating-Point Operations

The timing for floating-point operations and integer operations is the same except for the addition of the FINS1 and FINS2 signals in floating-point operations. In this example, Instruction 1 is a floating-point operation (see *Figure 6–21*). FINS1/2 tell the floating-point unit to move an instruction out of its decode buffer and begin execution. The FPU also makes use of the INST signal to latch instructions into its decode buffers.

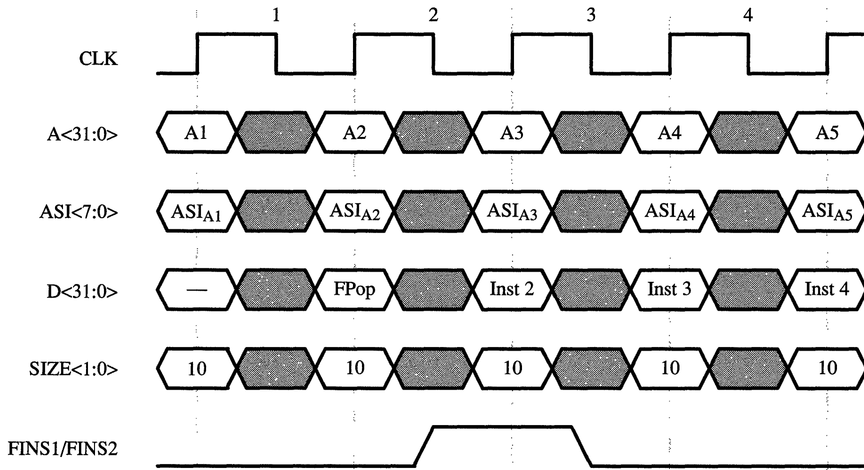


Figure 6–21. Floating-Point Operation Timing

6.4.9 Bus Arbitration

The CY7C601/CY7C611 does not have on-chip bus arbitration circuitry because it is designed to operate as a bus slave. Therefore, external circuitry must arbitrate between external bus requests and the CY7C601/CY7C611. When the CY7C601/CY7C611 needs to retain the buses it asserts the LOCK signal. The arbitration circuitry should assert $\overline{\text{BHOLD}}$ when it needs to keep the CY7C601/CY7C611 off the buses. When $\overline{\text{BHOLD}}$ is asserted, the processor's instruction pipeline is frozen until it is deasserted. The arbitration circuitry should also deassert the $\overline{\text{DOE}}$, $\overline{\text{AOE}}$, and $\overline{\text{COE}}$ signals to three-state the CY7C601/CY7C611's address bus, data bus and control signal output drivers so they may be driven by an external source (see *Figure 6–22*).

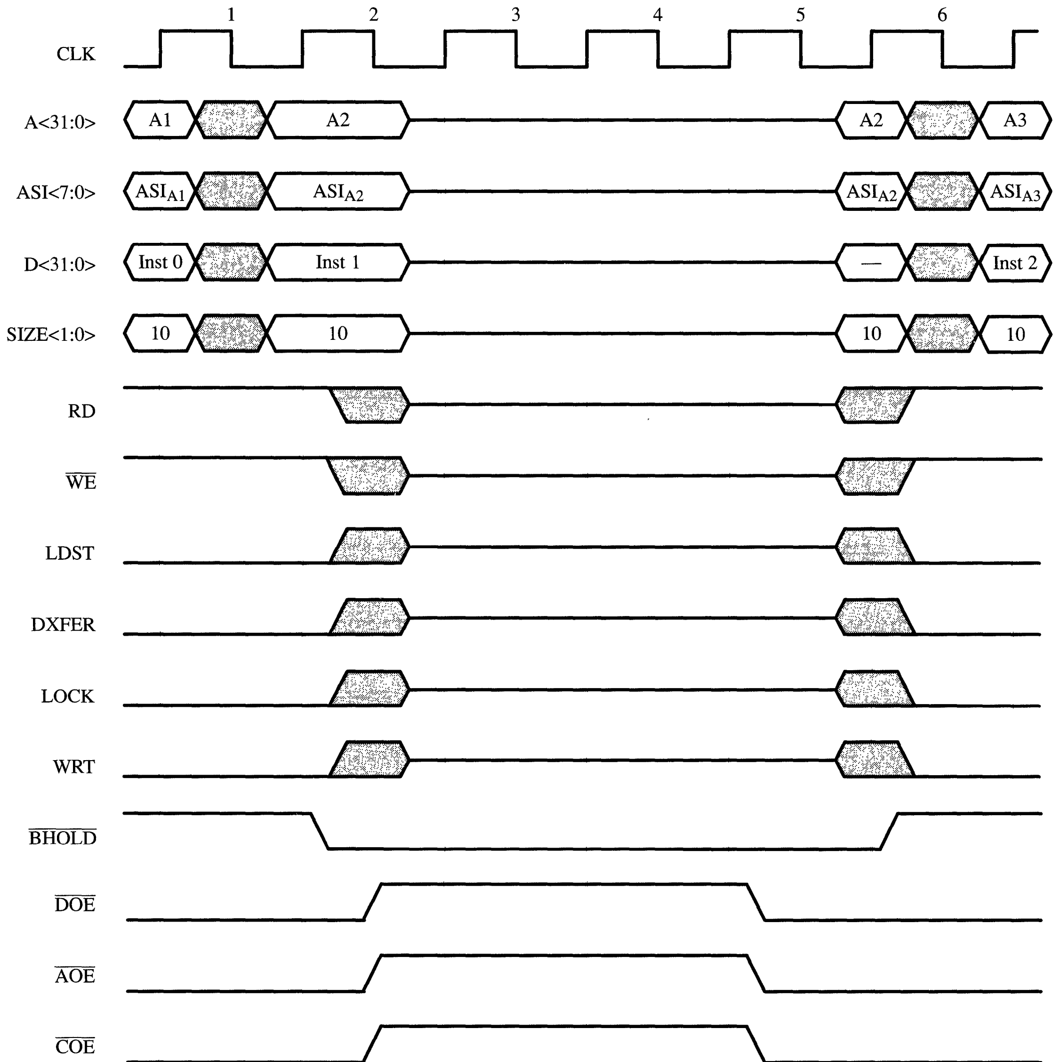


Figure 6–22. Bus Arbitration Timing

6.4.10 Load with Cache Miss

Figure 6–23 gives the timing for a load with cache miss. Cache logic must stop the processor by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ in the next cycle. However, the processor stops with the address of the next instruction on the address bus rather than the instruction that caused the miss. In order to retrieve the proper load data, the missed address (the address that was on the bus in the cycle before $\overline{\text{MHOLD}}$ was asserted) must be latched externally and placed back on the bus (this is done automatically by the cache controller in CY7C604/605–based systems). The $\overline{\text{MHOLD}}$ signal must be maintained while the missed data is strobed into the processor with the $\overline{\text{MDS}}$ signal (it must be strobed externally because the internal processor clock is frozen by the $\overline{\text{MHOLD}}$).

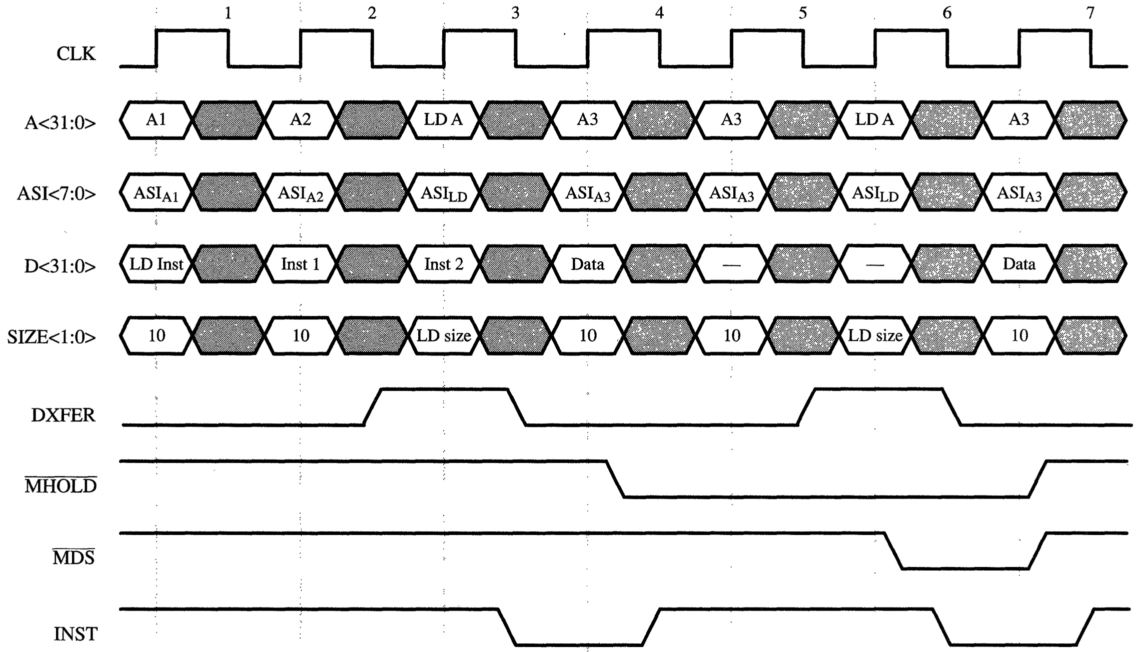


Figure 6–23. Load with Cache Miss Timing

6.4.11 Store with Cache Miss

The timing for a store with cache miss is similar to the load with cache miss situation, except $\overline{\text{MDS}}$ is not required (see *Figure 6–24*). Because the processor outputs the store address twice, it already has the proper address on the bus when it is stopped by $\overline{\text{MHOLD}}$. $\overline{\text{MDS}}$ is not required because nothing needs to be strobed into the processor.

INULL is asserted for the second occurrence of the store address so that it does not trigger the miss circuitry during the time the cache is processing the miss on the first occurrence of that address.

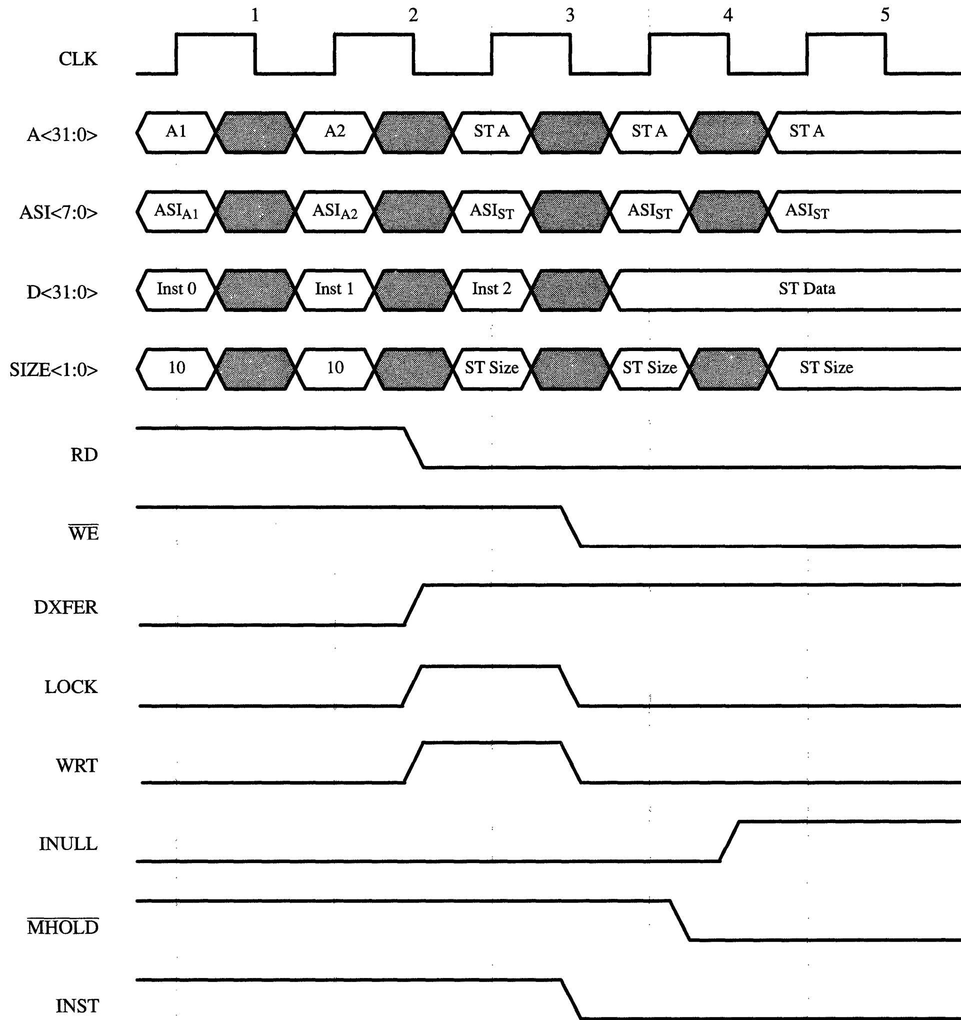


Figure 6–24. Store with Cache Miss Timing (1 of 2)

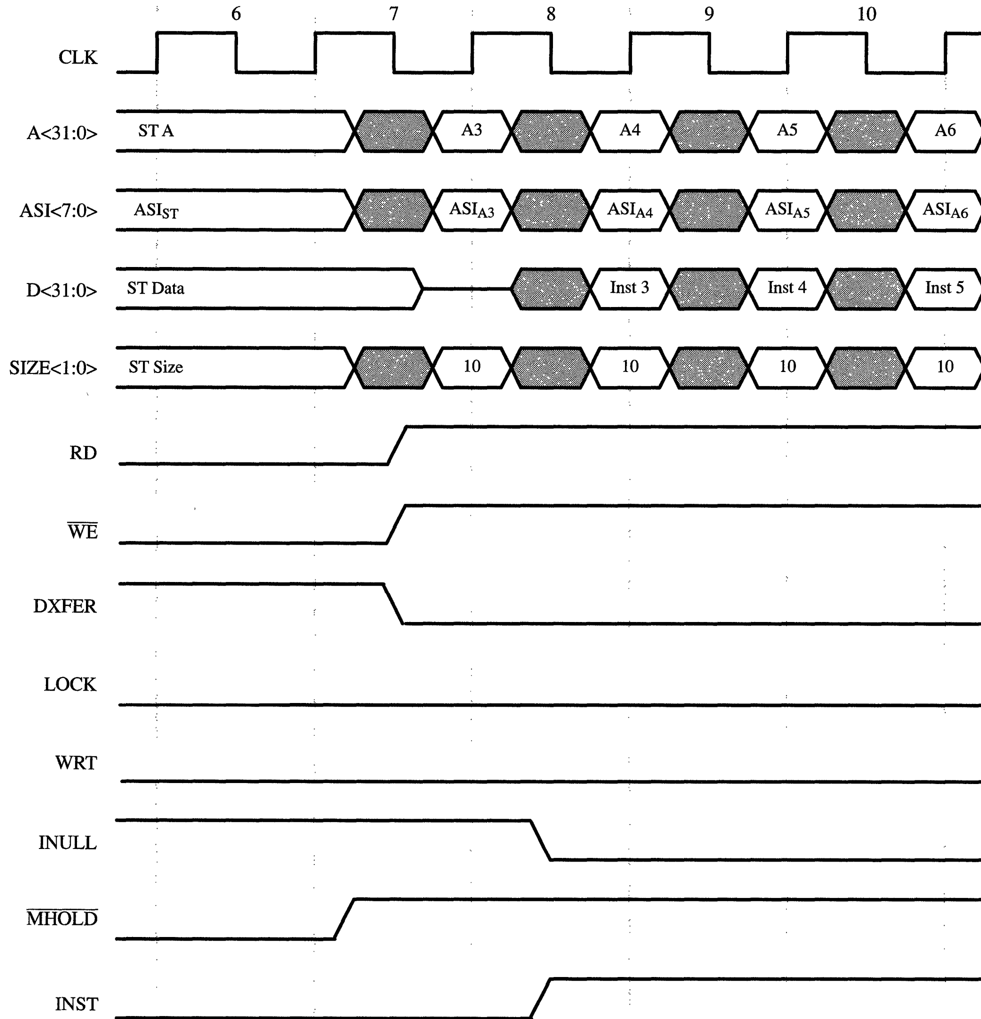


Figure 6–24. Store with Cache Miss Timing (2 of 2)

6.4.12 Memory Exceptions

Load with memory exception timing is shown in *Figure 6–25*. As with a cache miss, memory logic must stop the processor by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ in the next cycle. The $\overline{\text{MHOLD}}$ signal must be maintained while the memory exception ($\overline{\text{MEXC}}$) signal is strobed into the processor with the $\overline{\text{MDS}}$ signal (it must be strobed in externally because the internal processor clock is frozen by the $\overline{\text{MHOLD}}$). $\overline{\text{MEXC}}$ must be deasserted in the same clock cycle in which $\overline{\text{MHOLD}}$ is deasserted. Note that INULL is asserted in the cycle 8 instruction fetch to annul that fetch. This is the same action shown in cycle 2 of *Figure 6–12* for an internal trap. Store with memory exception has the same timing (see *Figure 6–26*) except INULL is asserted from the second store address through to the annulled cycle 8 instruction fetch.

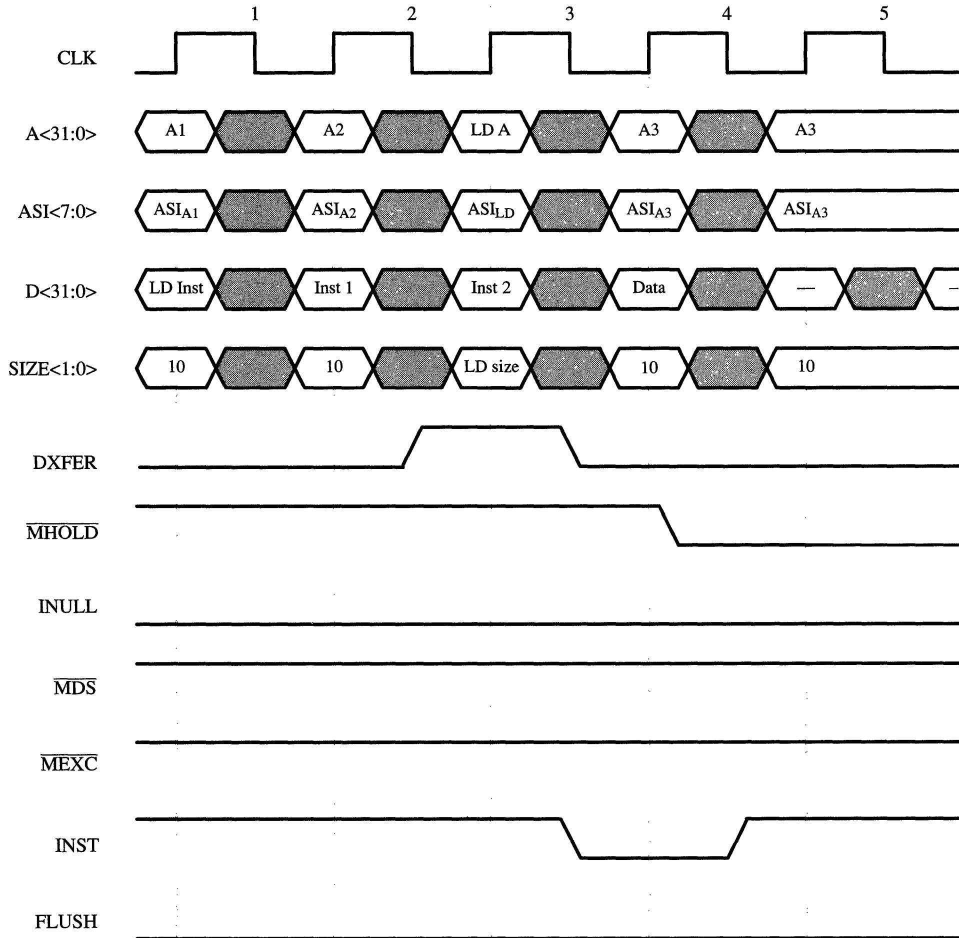


Figure 6–25. Load with Memory Exception Timing (1 of 2)

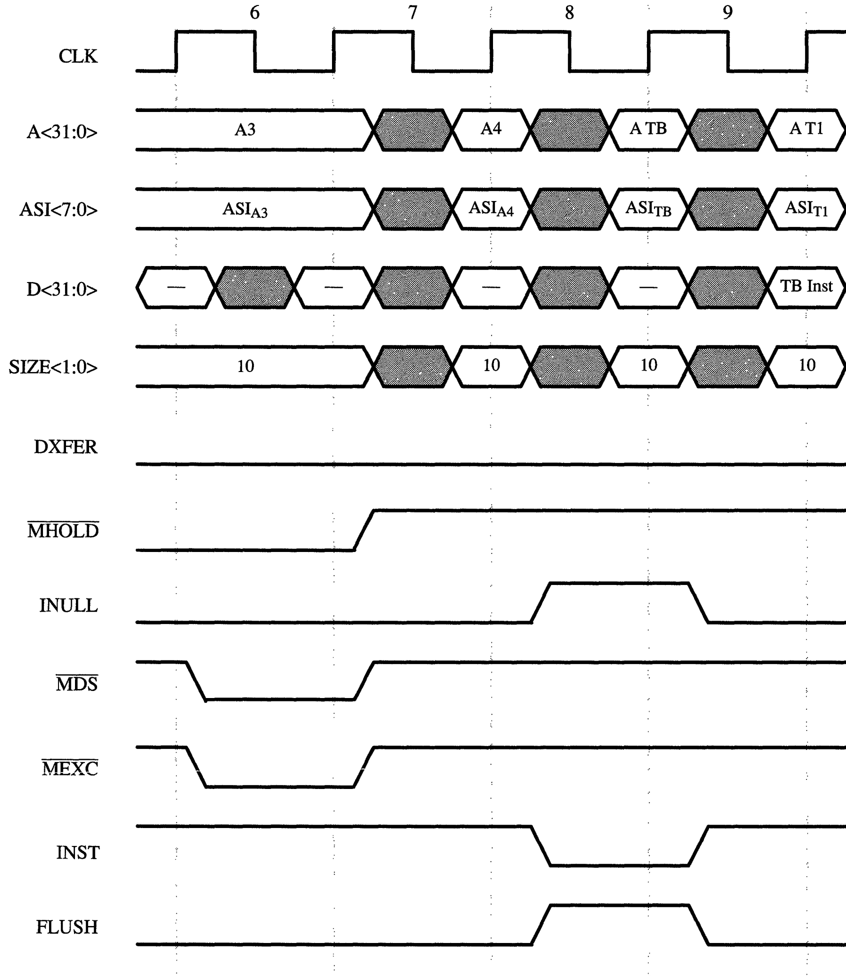


Figure 6–25. Load with Memory Exception Timing (2 of 2)

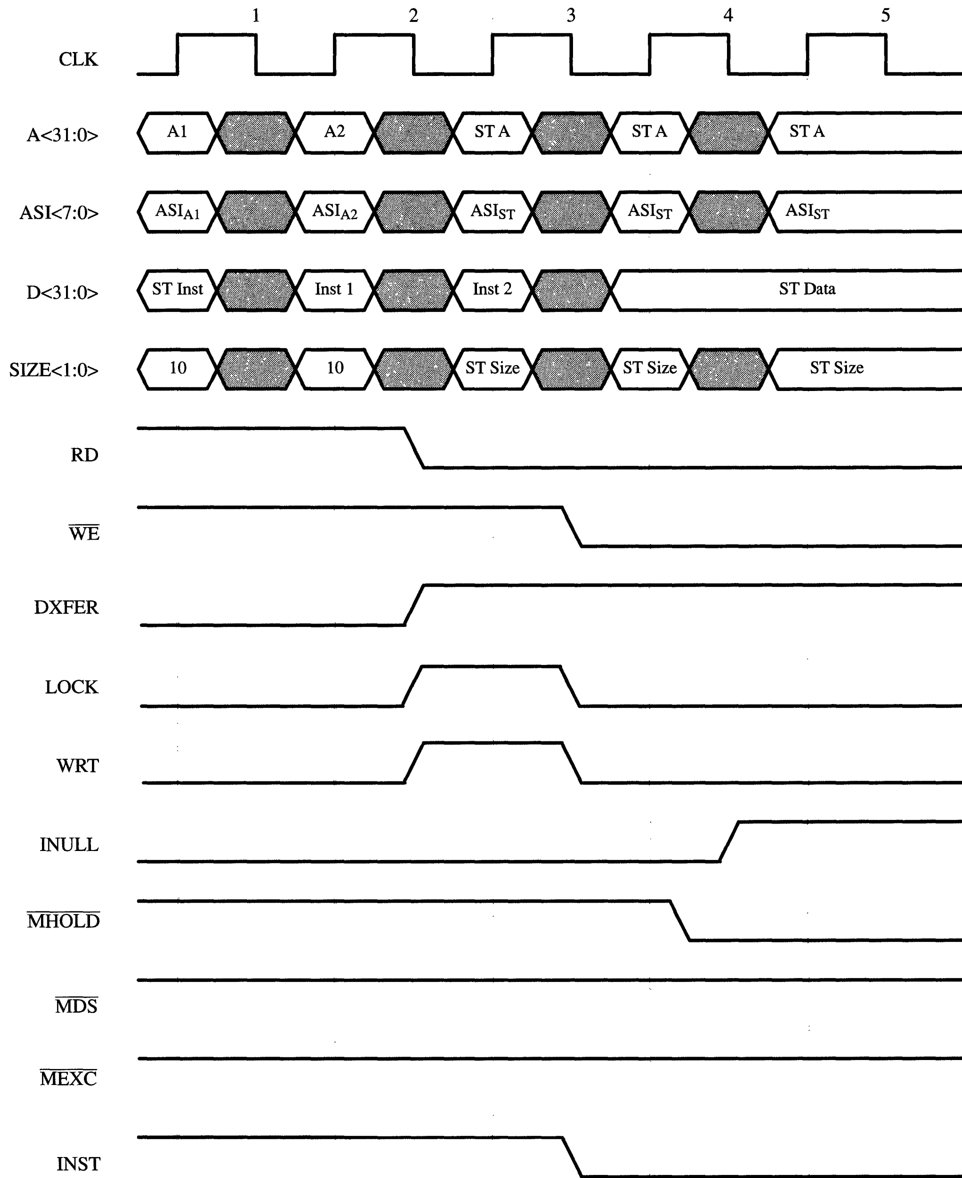


Figure 6–26. Store with Memory Exception Timing (page 1 of 2)

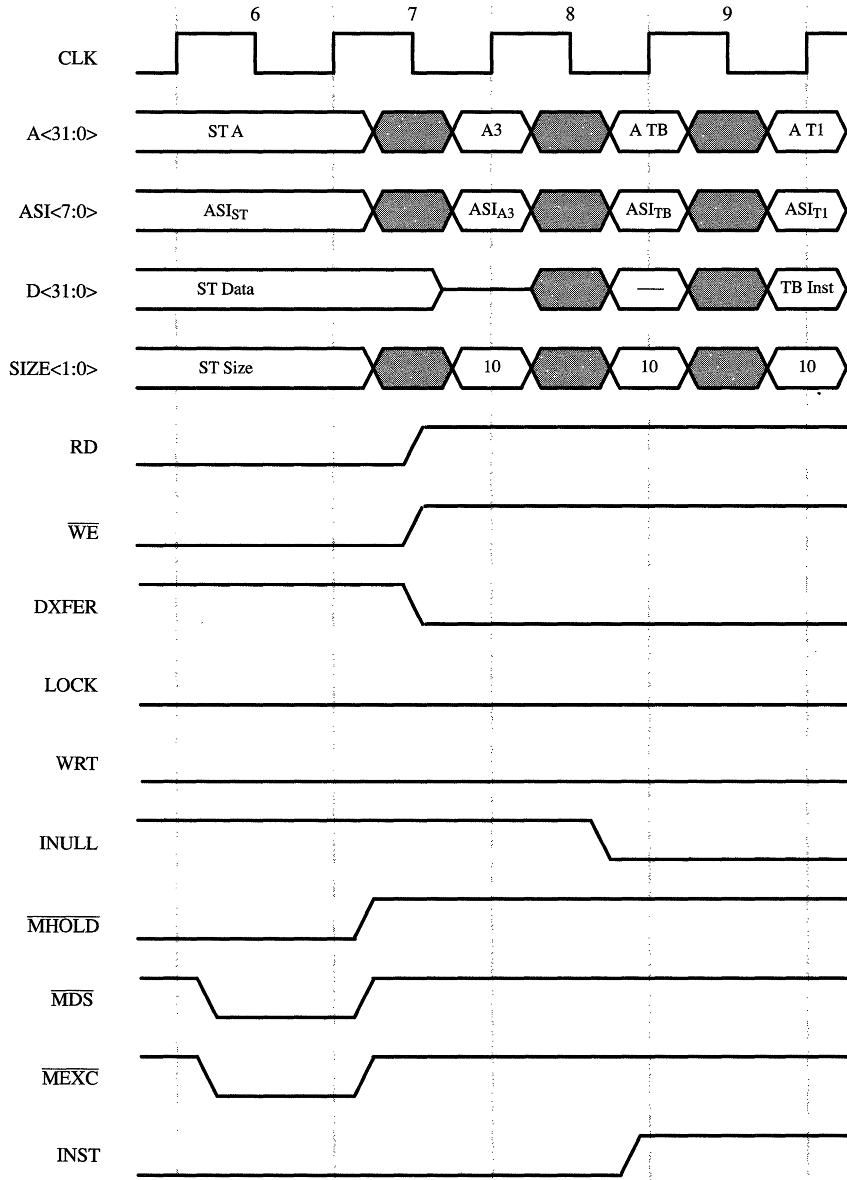


Figure 6–26. Store with Memory Exception Timing (page 2 of 2)

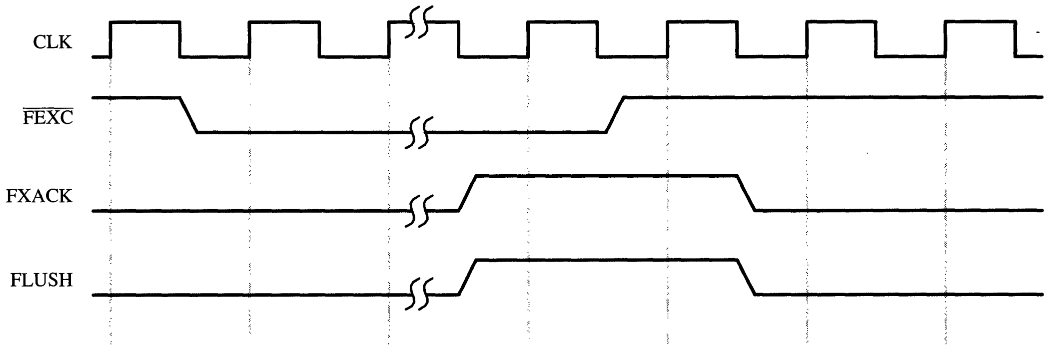


Figure 6–27. Floating-Point Exception Handshake Timing

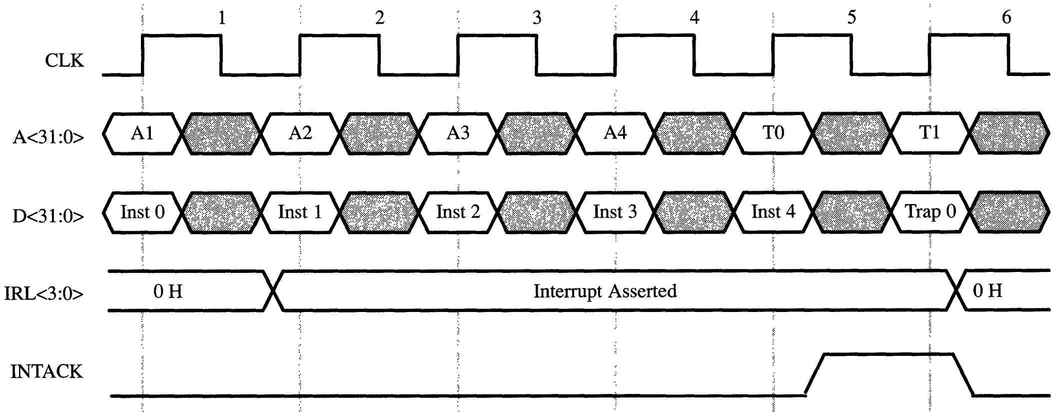


Figure 6–28. Asynchronous Interrupt Timing

6.4.13 Floating-Point Exceptions

The floating-point unit asserts $\overline{FEXC}$ to notify the CY7C601/CY7C611 that a floating-point exception has occurred and that it should take a trap on the next floating-point instruction that it encounters in the instruction stream (see *Figure 6–27*). The CY7C601/CY7C611 asserts FXACK to signal the FPU that the trap is being taken, and FLUSH to clean out the FPU's decode buffers. From this point on, the FPU will execute only floating-point store queue instructions until its queue is emptied by the trap handler.

$\overline{FEXC}$ is deasserted by the FPU after FXACK is asserted. FXACK is deasserted by the CY7C601/CY7C611 after $\overline{FEXC}$ is deasserted.

6.4.14 Interrupts

The asynchronous IRL<3:0> inputs are sampled on the rising edge of every clock. If the interrupt value represented by those inputs is greater than the masking value in the processor, and no higher priority trap supersedes it, the CY7C601/CY7C611 will take the interrupt. The IRL input level should be held stable until the processor asserts INTACK. *Figure 6–28* shows the timing for the best case response time where the IRL

input value is asserted one clock and a setup time before the Execute stage of a single-cycle instruction. Refer to *Section 2.4.5.3* for more information on interrupts.

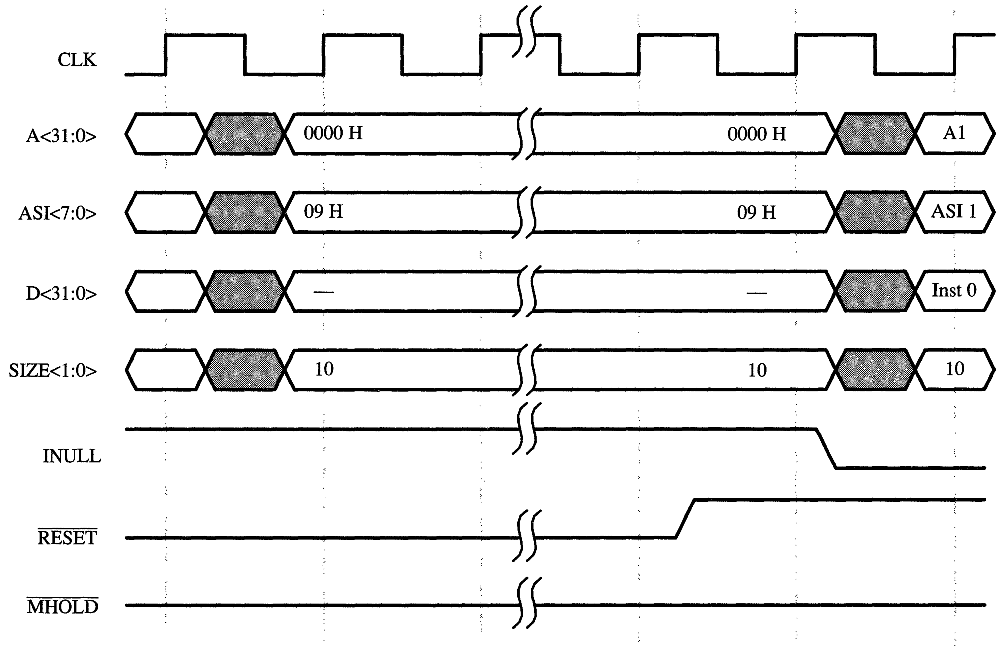


Figure 6–29. Power-On Reset Timing

6.4.15 Reset Condition

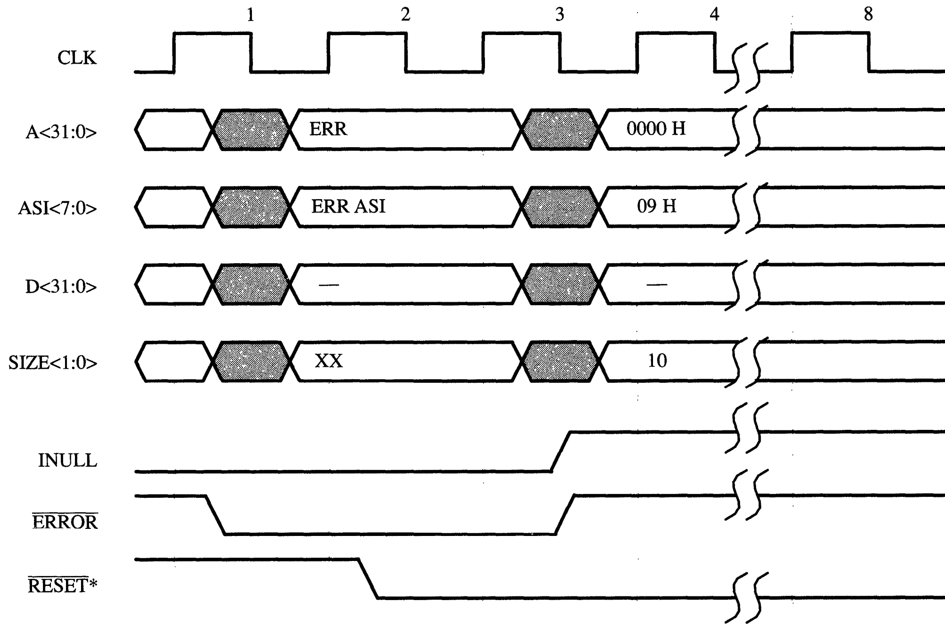
Figure 6–29 shows the timing for a power-on reset. RESET must be asserted for at least eight cycles so that the processor can synchronize the reset input and initialize its internal state. For RESET to be synchronized, the CLK signal must be active.

During the initialization, the processor disables traps (ET=0), sets the supervisor mode (S=1), and sets the program counter to location zero (PC=0, nPC=4).

6.4.16 Error Condition

Error mode is one of the three states in which the CY7C601/CY7C611 can exist. To get into the error mode, a synchronous trap must occur while traps are disabled (the processor state register's ET bit is set to zero). This essentially means that a trap which cannot be ignored occurs while another trap is being serviced. In order for that synchronous trap to be serviced, the processor goes through the normal operations of a trap (see *Section 2.4.5*), including setting the *tt* bits to identify the trap type. It then enters error mode, halts, and asserts the ERROR signal (see *Figure 6–30*).

The only way to leave error mode is to receive an external RESET signal, which forces the processor into reset mode. All information placed in the CY7C601/CY7C611's registers from the last Execute mode (the trap operation) remains unchanged and the processor resumes operation at address zero. The reset trap handler can examine the trap type of the synchronous trap and deal with it accordingly.



*RESET must be asserted for a minimum of 8 clocks

Figure 6–30. Error/Reset Timing (part 1 of 2)

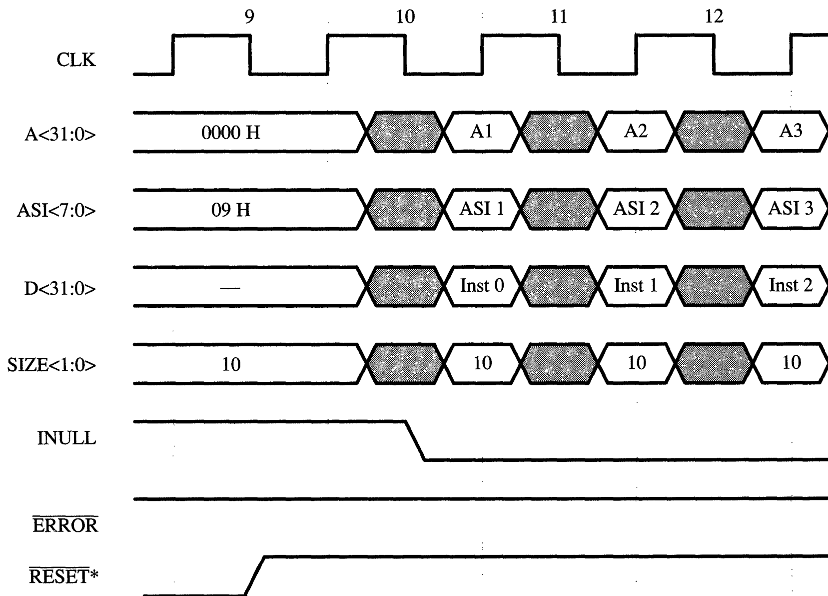


Figure 6–30. Error/Reset Timing (part 2 of 2)

*MHOLD must be driven to a deasserted state when RESET is asserted.

Table 6–6. Externally Generated Synchronous Exception Traps

Trap	Initiating Signal	Condition
Data Access Exception	$\overline{MEXC}$	Memory error during data access
Instruction Access Exception	$\overline{MEXC}$	Memory error during instruction access
Floating-Point Exception	$\overline{FEXC}$	Floating-point unit error
Coprocessor Exception	$\overline{CEXC}$	Coprocessor unit error

6.5 Exception Model

The CY7C601/CY7C611 supports three types of traps: synchronous, floating-point/coprocessor, and asynchronous (also called interrupts). Synchronous traps are caused by hardware responding to a particular instruction or by the trap on integer condition code (Ticc) instructions; they occur during the instruction that caused them.

Floating-point/coprocessor traps caused by a floating-point-operate (FPop) or coprocessor-operate (CPop) instruction occur before that instruction is complete. However, because floating-point (and coprocessor) exceptions are pended until the next floating-point (coprocessor) instruction is executed, other non-floating-point (coprocessor) instructions may have executed before the trap is taken. See *Figure 3–57.*

Asynchronous traps occur when an external event interrupts the processor. They are not related to any particular instruction and occur between the execution of instructions. See *Section 2.4.5.3.*

6.5.1 Reset

The reset trap is a special case of the external asynchronous trap type. It is asynchronous because it is triggered by asserting the $\overline{RESET}$ input signal. But from that point on, its behavior is entirely different from that of an asynchronous interrupt (see *Section 2.4.5.3*).

As soon as the CY7C601/CY7C611 recognizes the $\overline{RESET}$ signal, it enters reset mode and stays there until the $\overline{RESET}$ line is deasserted. The processor then enters Execute mode and then the Execute trap procedure. Here, it deviates from the normal action of a trap (*Section 2.4.5.4*) by modifying the enable traps bit (ET=0), and the supervisor bit (S=1). It then sets the PC to 0 (rather than changing the contents of the TBR), the nPC to 4, and transfers control to location 0. *All other PSR fields, and all other registers retain their values from the last Execute mode.*

Note: Upon power-up reset the state of all registers other than the PSR are undefined.

If the processor got to reset mode from error mode, then the normal actions of a trap have already been performed, including setting the *tt* field to reflect the cause of the error mode. Because this field is not changed by the reset trap, a post-mortem can be conducted on what caused the error mode. The processor enters error mode whenever a synchronous trap occurs while traps are disabled.

6.5.2 Synchronous Traps

Synchronous traps are caused by the actions of an instruction, with the trap stimulus occurring either internally to the CY7C601/CY7C611 or from an external signal which was provoked by the instruction. These traps are taken immediately and the instruction that caused the trap is aborted *before* it changes any state in the processor.

The external signals that can cause a synchronous trap are listed in *Table 6–6.*

6.5.2.1 External Signals

Synchronous traps generated by the input signal MEXC (Memory Exception) occur during the Execute phase of an instruction or occur immediately for data accesses. Traps generated by the FEXC and CEXC signals belong to the special floating-point/coprocessor category, and may not occur immediately.

6.5.2.1.1 Instruction Access Exception

An instruction access exception trap is generated if a memory exception occurs (the MEXC input signal is asserted) during an instruction fetch.

6.5.2.1.2 Data Access Exception

A data access exception trap is generated if a memory exception occurs (the MEXC input signal is asserted) during the data cycle of any instruction that moves data to or from memory.

6.5.2.2 Internal/Software

Synchronous traps generated by internal hardware are associated with an instruction. The trap condition is detected during the Execute stage of the instruction and the trap is taken immediately, before the instruction can complete.

6.5.2.2.1 Illegal Instruction

An illegal instruction trap occurs:

- When the UNIMP instruction is encountered,
- When an unimplemented instruction is encountered (excluding FPops and CPops),
- In any of the situations below where the continued execution of an instruction would result in an illegal processor state:
  1. Writing a value to the PSR's CWP field that is greater than the number of implemented windows (with a WRPSR)
  2. Executing an alternate space instruction with its *i* bit set to 1
  3. Executing a RETT instruction with traps enabled (ET=1)
  4. Executing an FLUSH instruction with IFT=0

Unimplemented floating-point and unimplemented coprocessor instructions do not generate an illegal instruction trap. They generate fp exception and cp exception traps, respectively.

6.5.2.2.2 Privileged Instruction

This trap occurs when a privileged instruction is encountered while the PSR's supervisor bit is reset (S=0).

6.5.2.2.3 fp Disabled

A fp disabled trap is generated when an FPop, FBfcc, or floating-point Load/Store instruction is encountered while the PSR's EF bit =0, or if no FPU is present (FP input signal =1).

6.5.2.2.4 CP Disabled

A cp disabled trap is generated when a CPop, CBccc, or coprocessor Load/Store instruction is encountered while the PSR's EC bit =0, or if no coprocessor is present ($\overline{CP}$ input signal =1).

6.5.2.2.5 Window Overflow

This trap occurs when the continued execution of a SAVE instruction would cause the CWP to point to a window marked invalid in the WIM register.

6.5.2.2.6 Window Underflow

This trap occurs when the continued execution of a RESTORE instruction would cause the CWP to point to a window marked invalid in the WIM register. The window underflow trap type can also be set in the PSR during a RETT instruction, but the trap taken is a reset. See *Section 2.4.5.3.2* on reset traps and *Chapter 12* for the instruction definition for RETT.

6.5.2.2.7 Memory Address Not Aligned

Memory address not aligned trap occurs when a load or store instruction generates a memory address that is not properly aligned for the data type or if a JMPL instruction generates a PC value that is not word aligned (low-order two bits nonzero).

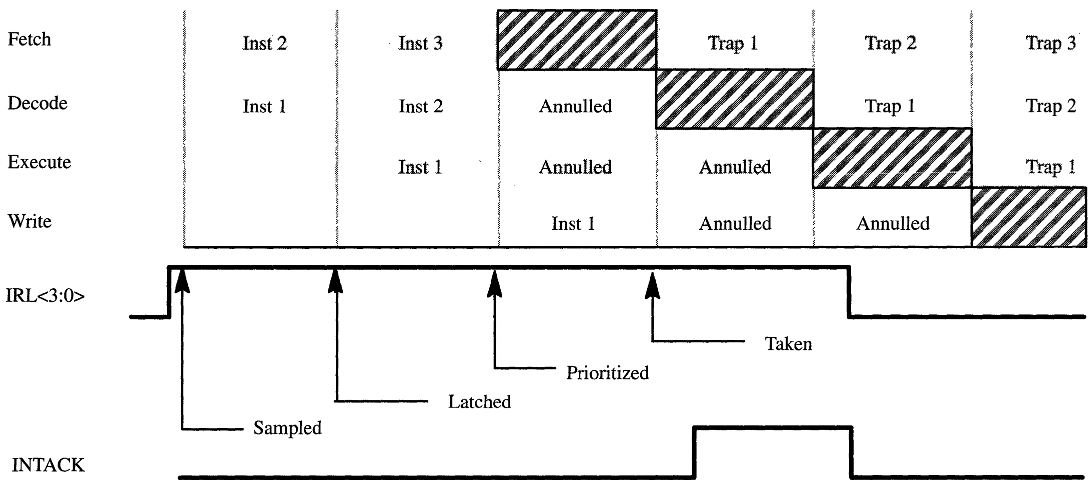


Figure 6–31. Best-Case Interrupt Response Timing

6.5.2.2.8 Tag Overflow

This trap occurs if execution of a TADDccTV or TSUBccTV instruction causes the overflow bit of the integer condition codes to be set. See the instruction definitions of TADDccTV and TSUBccTV and *Section 2.4.3.2.3* for details.

6.5.2.2.9 Trap Instruction

This trap occurs when a Ticc instruction is executed and the trap conditions are met. There are 128 programmable trap types available within the trap instruction trap (see *Chapter 6, Ticc Instruction*).

6.5.3 Interrupts (Asynchronous Traps)

Asynchronous traps occur in response to the Interrupt Request Level (IRL<3:0>) inputs. This type of trap is not associated with an instruction and is said to happen between instructions. This is because, unlike synchronous traps, an interrupt allows the instruction in whose Execute stage it is prioritized to complete execution (see *Figure 6–31*). Any instruction that has entered the pipeline behind the instruction which was allowed to complete is annulled, but can be restarted again after returning from the trap.

6.5.3.1 Priority

The level, or priority, of the interrupt is determined by the value on the IRL<3:0> pins. For the interrupt to be taken, the IRL value must be greater than the value in the processor interrupt level (PIL) field of the processor state register (PSR). A value of 0 indicates that no interrupt is requested. A value of 15 represents a non-maskable interrupt. All other IRL values between 0 and 15 represent interrupt requests which can be masked by the PIL field. The priority and trap type (*tt*) for each level is shown in *Table 6–7* in *Section 6.5.5.3*.

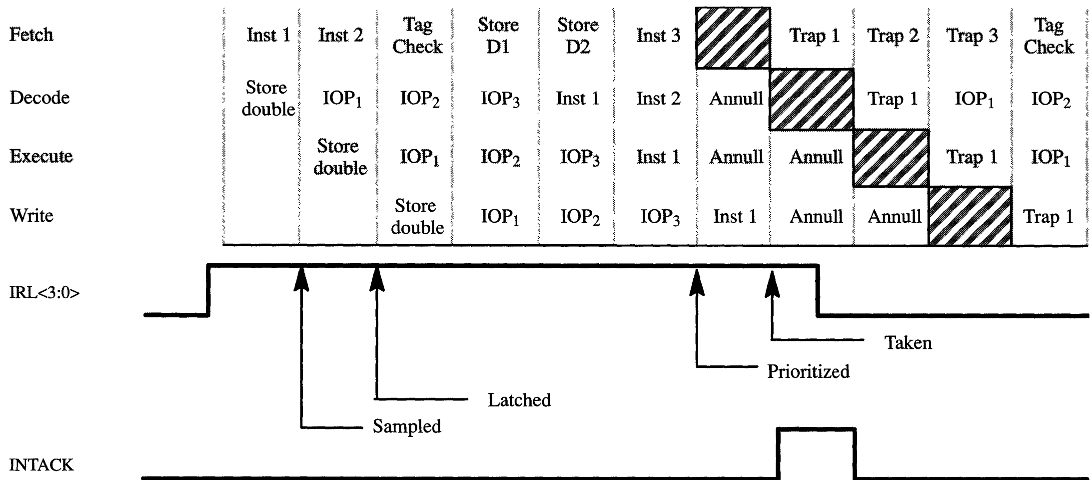


Figure 6–32. Worst-Case Interrupt Response Timing

6.5.3.2 Response Time

The CY7C601/CY7C611 samples the IRL inputs at the rising edge of every clock. In order to properly synchronize these asynchronous inputs, they are put through two synchronizing levels of D-type flip-flops. The outputs of the two levels must agree before the interrupt can be processed. If the outputs disagree, the interrupt request is ignored. This logic serves to filter transients on the IRL lines, but it means that the lines must be active for two consecutive clock edges to be accepted as valid.

Once the IRL input has been accepted, it is prioritized and the appropriate trap is taken during the next Execute stage of the instruction pipeline. Best case interrupt response occurs when the interrupt is applied one clock plus one setup time before the Execute phase of any instruction in the pipeline (see *Figure 6–31*). In this case, the first instruction of the interrupt service routine is fetched during the fourth clock following the application of an IRL value greater than the PIL field of the processor status register (PSR). This also holds for an IRL value of 0F H, which acts as a non-maskable interrupt.

The worst case interrupt response occurs when the detection of the IRL input just misses the cutoff point for the Execute stage of a four-cycle instruction, such as a store double or atomic load-store (see *Figure 6–32*). In this case, the interrupt input must wait an additional three cycles for the next pipeline Execute phase. In addition, if the IRL input just misses the sampling clock edge, an additional clock delay occurs. As a result, the first instruction of the service routine is fetched in the eighth clock following the application of IRL.

The best and worst case interrupt timing described above assumes that the processor is not stopped via the application of an external hold signal, and that the IRL input is not superceded by the occurrence of a synchronous (internal) trap.

6.5.3.3 Interrupt Acknowledge

As shown in *Figure 6–31*, and more clearly in *Figure 6–32*, the interrupt acknowledge (INTACK) output signal is asserted when the interrupt is *taken*, not when it is first detected and latched. Because of this delay, if the IRL<3:0> inputs are changed to reflect another interrupt condition before the corresponding INTACK for the latched condition is received, there could be some question as to which interrupt the INTACK is responding to. Therefore, external hardware should ensure that the IRL<3:0> inputs are held stable until an INTACK is received.

6.5.4 Floating-Point/Coprocessor Traps

Floating-point/coprocessor exception traps are considered a separate class of traps because they are both synchronous and asynchronous. They are asynchronous because they are triggered by an external signal ($\overline{\text{FEXC}}$ or $\overline{\text{CEXC}}$), and are taken sometime after the floating-point or coprocessor instruction that caused the exception. This can happen because the CY7C601/CY7C611 and the FPU (coprocessor) operate concurrently. However, they are also synchronous, because they are tied to an instruction—the next floating-point or coprocessor instruction encountered in the instruction stream after the signal is received.

When the FPU (coprocessor) recognizes an exception condition, it enters an "exception pending mode" state. It remains in this state until the CY7C601/CY7C611 signals that it has taken an fp exception (cp exception) trap by sending back an FXACK (CXACK) signal. The FPU (coprocessor) then enters the "exception mode" state, remaining there until the floating-point (coprocessor) queue has been emptied by execution of one or more STDFQ (STDCQ) instructions.

Although the PC will always point to a floating-point or coprocessor instruction after an exception trap is taken, it does not point to the instruction that caused the exception. However, the instruction that did cause the exception is always the front entry in the queue at the time the trap is taken, and the entry includes both the instruction and its address. The remaining entries in the queue point to FPops (CPops) that have been started but have not yet completed. Once the queue has been emptied, these can be re-executed or emulated.

6.5.4.1 Floating-Point Exception

This trap occurs when the FPU is in exception pending mode and an FPop, FBfcc, or floating-point Load/Store instruction is encountered. The type of exception is encoded in the *tt* field of the floating-point state register (FSR). Refer to *Section 7.3.1* for further details on the FSR.

6.5.4.2 Coprocessor Exception

This trap occurs when the coprocessor is in exception pending mode and a CPop, CBccc, or coprocessor Load/Store instruction is encountered. The type of exception should be encoded in the *tt* field of the coprocessor state register (CSR). The nature of the exception is implementation dependent.

6.5.5 Trap Operation

Once a trap is taken, the following operations take place:

- Further traps are disabled (asynchronous traps are ignored; synchronous traps force an error mode).
- The S bit of the PSR is copied into the PS bit; the S bit is then set to 1.
- The CWP is decremented by one (modulo the number of windows) to activate a trap window. This happens regardless of the contents of the WIM register, which is ignored upon entering a trap.
- The PC and nPC are saved into r[17] and r[18], respectively, of the trap window.
- The *tt* field of the TBR is set to the appropriate value.
- If the trap is not a reset, the PC is written with the contents of the TBR and the nPC is written with TBR + 4. If the trap is a reset, the PC is set to address zero and the nPC to address four.

Unlike many other processors, the SPARC architecture does not automatically save the PSR into memory during a trap. Instead, it saves the volatile S bit into the PSR itself and the remaining fields are either altered in a reversible manner (ET and CWP), or should not be altered in the trap handler until the PSR has been saved to memory.

6.5.5.1 Recognition

In most cases, traps are "recognized" in the pipeline's Execute stage. For a synchronous trap, the trap criteria are examined during the Execute stage of an instruction, and the trap is taken immediately, before the Write stage of that instruction takes place. This includes the fp disabled and cp disabled trap type. The special cases occur with those traps generated by external signals. A memory exception on an instruction fetch is detected at the beginning of the Execute stage of instruction execution. Memory exceptions occurring on data accesses are detected on the rising clock edge of the data cycle.

Because asynchronous traps happen "between" instructions, their timing is slightly different. As long as the ET bit is set to one, the CY7C601/CY7C611 checks for interrupts. The interrupt is sampled on a rising clock edge and latched on the next rising clock edge. The processor compares the IRL<3:0> input value against the PIL field of the PSR, and if IRL is greater than PIL, or IRL is 15 (unmaskable), then it is prioritized at the end of the next Execute stage of the pipeline. A trap keyed to the IRL level occurs after the Write stage completes.

Floating-point/coprocessor exception traps are not recognized when the $\overline{\text{FEXC}}$ or $\overline{\text{CEXC}}$ signal is first sampled. The processor waits until it encounters a floating-point or coprocessor instruction in the instruction stream and then handles it as if it were an internal synchronous trap.

6.5.5.2 Trap Addressing

The trap base register (TBR) is made up of two fields, the trap base address (TBA) and the trap type (*tt*). The TBA contains the most-significant 20 address bits of the trap table, which is in external memory. The trap type field, which was written by the trap, not only uniquely identifies the trap, it also serves as an offset into the trap table when the TBR is written to the PC. The TBR address is the first address of the trap handler. However, because the trap addresses are only separated by four words (the least-significant four bits of TBR are zero), the program must jump from the trap table to the actual address of the particular trap handler.

Of the 256 trap types allowed by the 8-bit *tt* field, half are dedicated to hardware traps (0-127), and half are dedicated to programmer-initiated traps (Ticc). For a Ticc instruction, the processor must calculate the *tt* value from the fields given in the instruction, while the hardware traps can be set from a table such as the one below. The *tt* field remains valid until another trap occurs. See the Ticc instruction definition for details.

6.5.5.3 Trap Types and Priority

Each type of trap is assigned a priority (see *Table 6–7*). When multiple traps occur, the highest priority trap is taken, and lower priority traps are ignored. In this situation, a lower priority trap must either persist or be repeated in order to be recognized and taken.

Table 6–7. Trap Type and Priority Assignments

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Reset	1	–	Async.
Instruction Access	2	1	Sync.
Illegal Instruction	3	2	Sync.
Privileged Instruction	4	3	Sync.
Floating-Point Disabled	5	4	Sync.
Coprocessor Disabled	5	36	Sync.
Window Overflow	6	5	Sync.
Window Underflow	7	6	Sync.
Memory Address not Aligned	8	7	Sync.
Floating-Point Exception	9	8	Sync.
Coprocessor Exception	9	40	Sync.
Data Access Exception	10	9	Sync.
Tag Overflow	11	10	Sync.
Trap Instructions (Ticc)	12	128 – 255	Sync.
Interrupt Level 15	15	31	Async.
Interrupt Level 14	16	30	Async.
Interrupt Level 13	17	29	Async.
Interrupt Level 12	18	28	Async.
Interrupt Level 11	19	27	Async.
Interrupt Level 10	20	26	Async.
Interrupt Level 9	21	25	Async.
Interrupt Level 8	22	24	Async.
Interrupt Level 7	23	23	Async.
Interrupt Level 6	24	22	Async.
Interrupt Level 5	25	21	Async.
Interrupt Level 4	26	20	Async.
Interrupt Level 3	27	19	Async.
Interrupt Level 2	28	18	Async.
Interrupt Level 1	29	17	Async.

6.5.5.4 Return From Trap

On returning from a trap with the RETT instruction, the following operations take place:

- The CWP is incremented by one (modulo the number of windows) to re-activate the previous window.
- The return address is calculated
- Trap conditions are checked. If traps have already been enabled (ET=1), an illegal instruction trap is taken. If traps are still disabled but S=0, or the new CWP points to an invalid window, or the return address is not properly aligned, then an error mode/reset trap is taken.
- If no traps are taken, then traps are re-enabled (ET=1).
- The PC is written with the contents of the nPC, and the nPC is written with the return address.
- The PS bit is copied back into the S bit.

The last two instructions of a trap handler should be a JMPL followed by a RETT. This instruction couple causes a non-delayed control transfer back to the trapped instruction or to the instruction following the trapped instruction, whichever is desired. See the RETT instruction definition for details.

6.6 Coprocessor Interface

In the SPARC architecture, the Integer Unit is the basic processing engine, but provision is made for two coprocessor extensions. The extensions are in the form of instruction set extensions and a pair of identical signal interfaces. In the CY7C601, one of these instruction and signal interface extensions is dedicated to floating-point operations and the other is designated for a second coprocessor that may be supplied by the user. Although signals and instructions have been named to reflect the assumption of how these two extensions will be used, either instruction set extension/signal interface may be used in any way desired. Note that the CY7C611 does not provide a separate coprocessor interface, but it does support the floating-point interface. Execution of coprocessor instructions by the CY7C611 will cause a *cp disabled* trap to occur.

In order for the CY7C601 to support a user-defined coprocessor, the coprocessor should contain certain elements defined by the SPARC architecture. These include an internal register set, a status register, a coprocessor queue, and a set of compatible interface pins. These elements are identical to the floating-point interface, and it is recommended that a user desiring to use the coprocessor interface thoroughly study the floating-point interface in *Section 7.2* as an example of a coprocessor interface application.

6.6.1 Protocol

The coprocessor extensions to the architecture are designed to allow the coprocessor to operate concurrently with the Integer Unit and the floating-point unit. To keep operations synchronized, address and data buses are shared. The initial CY7C601 instruction Decode determines which unit should execute the instruction. The CY7C601 executes its own instructions, but signals the coprocessor to continue the Decode and execution if it recognizes a coprocessor instruction. For coprocessor loads and stores, the CY7C601 supplies the memory address and the coprocessor receives or supplies the data. The coprocessor must deal with resource or data dependencies, signaling the problem to the CY7C601 by freezing the instruction pipeline with the $\overline{\text{CHOLD}}$ signal.

The signal interface between the CY7C601/CY7C611 and the coprocessor consists of shared address, data, clock, reset, and control signals, plus a special set of signals that provide synchronization and minimal status information between the coprocessor and the CY7C601.

6.6.1.1 Coprocessor Interface Signals

The SPARC architecture defines two sets of signals intended for interfacing with two coprocessors. The CY7C601 assigns one set of coprocessor signals for specific use by the floating-point unit, and the other

set of coprocessor signals for a user-defined coprocessor. All floating-point interface signal names begin with an *F*, and all coprocessor interface signal names begin with a *C*. Both sets of interface signals share the INST signal, which identifies a CY7C601 instruction fetch. The two groups of signals are symmetric, have identical timing requirements, and are listed in *Table 6–2*.

Instruction fetch is signaled by the CY7C601 using the INST signal. The coprocessor uses INST as an input to enable latching of an instruction on the data bus. The coprocessor latches all instructions fetched by the CY7C601, regardless of instruction type. The coprocessor is expected to use a two-stage instruction/address buffer as described in *Section 7.2* on the Floating-Point/Integer Unit interface. The CY7C601 asserts CINS1 or CINS2 at the beginning of the Decode stage of instruction execution of a coprocessor instruction. The CINS1 or CINS2 signals are used to start the execution of a coprocessor instruction and select which of the two most recently fetched instructions stored in the two-stage instruction cache is to be executed by the coprocessor.

The CY7C601 requires the $\overline{\text{CP}}$ signal to be driven low in order for the Integer Unit to recognize the presence of a coprocessor. Attempting to execute coprocessor instructions with $\overline{\text{CP}}$ high will cause the CY7C601 to execute a *cp disabled* trap.

Hardware interlocking for coprocessor instruction execution is provided with the $\overline{\text{CHOLD}}$ signal. This signal is asserted by the coprocessor to freeze the CY7C601. This signal is asserted in cases where the CY7C601 must be halted to prevent it from causing a condition from which the coprocessor cannot recover. An example of this would be fetching multiple coprocessor instructions that would otherwise overrun the coprocessor queue. The coprocessor would be expected to assert $\overline{\text{CHOLD}}$ until it could handle additional instructions.

Coprocessor interrupts are asserted with the $\overline{\text{CEXC}}$ signal. This signal is asserted by the coprocessor upon the detection of an exception case. The CY7C601 will continue normal execution until the execution stage of the next coprocessor instruction. At that time, the CY7C601 will acknowledge the interrupt with CXACK, and begin coprocessor trap execution.

Coprocessor branch on condition code (CBcc) instructions are executed by the CY7C601 Integer Unit based on the value of the CCC<1:0> signals supplied by the coprocessor. These signals are typically set by the execution of a coprocessor compare instruction (defined by the designer). The CCCV signal supplied by the coprocessor indicates whether the state of the CCC<1:0> signals is valid. CCCV is normally asserted, but is deasserted when a coprocessor compare instruction is executed and remains deasserted until that instruction is completed. The deassertion of this signal causes the CY7C601 to halt execution. This interlock prevents the CY7C601 from branching on invalid condition codes. The SPARC architecture requires at least one non-coprocessor instruction between a coprocessor compare and a coprocessor branch on condition code (CBcc) instruction.

6.6.2 Register Model

The coprocessor register model specified by the SPARC architecture is shown in *Figure 6–33*. The coprocessor has its own 32 x 32-bit working register file from which all operands for CPop instructions originate and to which all results return. The contents of these registers are transferred to and from memory under control of the CY7C601, using coprocessor Load/Store instructions.

The coprocessor state register (CSR) contains the current status of the coprocessor. The exact nature of the exception bits and trap types are implementation dependent. The CSR is read and written indirectly through memory using the LDCSR and STCSR instructions.

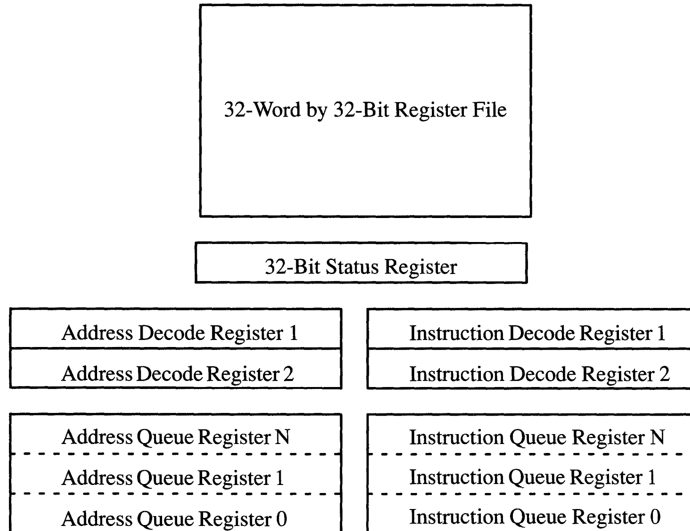


Figure 6–33. Coprocessor Register Model

The coprocessor queue is necessary to properly handle traps with concurrently operating units. The first-in, first-out queue records all pending coprocessor instructions and their addresses at the time of a coprocessor exception. The front entry of the queue contains the unfinished instruction that caused the exception. The rest of the queue contains unfinished CPops which would be restarted or emulated after the trap handler returns control to the main program.

The address and instruction decode buffers hold instructions and their addresses until the CY7C601 determines if they belong to the coprocessor. If one of the held instructions belongs to the coprocessor, the CY7C601 sends the appropriate CINS signal to move the instruction into the coprocessor Execute stage. The address and a copy of the instruction also move into the queue at this point and remain there until the instruction completes.

When a trap is taken, the CY7C601 asserts the FLUSH signal, causing the coprocessor to dump any instructions in the decode buffers. FLUSH does not affect instructions that are already in the queue.

6.6.3 Exceptions

Exactly what conditions will generate a cp exception trap are implementation dependent. However, most implementations would probably include Unfinished CPop as a condition that would cause an exception. An unfinished CPop trap is generated when the coprocessor cannot complete execution because the data has exceeded the capabilities of the coprocessor and/or has generated an inappropriate result.

6.7 CY7C611 Integer Unit for Embedded Control

The CY7C611 is a SPARC Integer Unit designed for embedded control applications. It is a functional equivalent of the CY7C601 with a reduced pin out for lower cost applications. The CY7C611 retains all internal features of the CY7C601, and maintains complete binary code compatibility with all other SPARC processors. The CY7C611 differs from the CY7C601 in that the address bus has been reduced to 24 bits, the ASI signals have been reduced to three bits, and several control signals not required for lower cost systems have been eliminated. The CY7C611 supports the floating-point interface, but does not include the coprocessor interface. The CY7C611 is packaged in a low-cost 160-pin plastic quad flat package (PQFP) and is available in speeds of 25 MHz.

CY7C601 signals not available on the CY7C611 are listed in *Table 6–8* below. The signal summary for the CY7C611 is listed in *Table 6–9*. All CY7C611 signals are identical to their CY7C601 counterparts, and the information regarding the CY7C601 in this chapter is also valid for the CY7C611.

Note that the EC (enable coprocessor) bit of the PSR register for the CY7C611 is permanently forced to zero.

A user-defined coprocessor can be connected to the CY7C611 instead of a floating-point unit, if desired. All floating-point interface signals are identical in function to their coprocessor counterparts. In order to use the floating-point interface to support a user-defined coprocessor, the floating-point instructions must be used to exercise the coprocessor. This will require software remapping of coprocessor instructions. The CY7C601 and CY7C611 do not decode the nine-bit opf field of a floating-point operate instruction. This can be used to map coprocessor instructions to valid and invalid FPop instructions (as specified by the op3 and opf fields of the op code) without causing an invalid FP instruction trap, since the invalid FP instruction must recognized by the floating-point unit.

Table 6–8. Signal Differences Between CY7C601 and CY7C611

CY7C601 Signals Not Available on CY7C611	
A<31:24>	Address bits 31 through 24
$\overline{\text{AOE}}$	Address Output Enable
ASI<7:3>	ASI bits 7 through 3
CCC<1:0>	Coprocessor Condition Codes <1:0>
CCCV	Coprocessor Condition Codes Valid
$\overline{\text{CEXC}}$	Coprocessor Exception
$\overline{\text{CHOLD}}$	Coprocessor Hold
CINS1	Coprocessor Instruction Stage 1
CINS2	Coprocessor Instruction Stage 2
$\overline{\text{COE}}$	Control Output Enable
$\overline{\text{CP}}$	Coprocessor Present
CXACK	Coprocessor Exception Acknowledge
$\overline{\text{DOE}}$	Data Output Enable
DXFER	Data Transfer
$\overline{\text{IFT}}$	Instruction Cache Flush Trap

Table 6–9. CY7C611 Signal Summary

CY7C611 Signal Summary			
Signal Name	**Signal Description**	**Input/Output**	**Active**
A<23:0>	Address Bus	Three-State Output	
ASI<2:0>	Address Space Identifier	Three-State Output	
$\overline{\text{BHOLD}}$	Bus Hold	Input	Low
CLK	Clock	Input	
D<31:0>	Data	Three-State Bidir.	
$\overline{\text{ERROR}}$	IU Error Mode	Three-State Output	Low
FCC<1:0>	Floating-Point Condition Codes	Input	
FCCV	Floating-Point Condition Codes Valid	Input	High
$\overline{\text{FEXC}}$	Floating-Point Exception	Input	Low
$\overline{\text{FHOLD}}$	Floating-Point Hold	Input	Low
FINS1	Floating-Point Instruction Stage 1	Three-State Output	High
FINS2	Floating-Point Instruction Stage 2	Three-State Output	High
FLUSH	Flush FP Instruction	Three-State Output	High
$\overline{\text{FP}}$	Floating-Point Present	Input	Low
FPSYN	FP Synonym Mode	Input	High
FXACK	FP Exception Acknowledge	Three-State Output	High
IRL<3:0>	Interrupt Level <3:0>	Input	
INST	Instruction Fetch Cycle	Three-State Output	High
INULL	Instruction Cycle Nullify	Three-State Output	High
INTACK	Interrupt Acknowledge	Three-State Output	High
LDST	Atomic Load-Store Operation	Three-State Output	High
LOCK	Multicycle Bus Lock	Three-State Output	High
$\overline{\text{MDS}}$	Memory Data Strobe	Input	Low
$\overline{\text{MEXC}}$	Memory Exception	Input	Low
$\overline{\text{MHOLDA}}$	Memory Hold A	Input	Low
$\overline{\text{MHOLDB}}$	Memory Hold B	Input	Low
RD	Read	Three-State Output	High
$\overline{\text{RESET}}$	Reset	Input	Low
SIZE<1:0>	Bus Transaction Size	Three-State Output	
$\overline{\text{TOE}}$	Test Output Enable	Input	Low
WRT	Advanced Write	Three-State Output	High
$\overline{\text{WE}}$	Write	Three-State Output	Low

CY7C602 Floating-Point Unit

The CY7C602 Floating-Point Unit (FPU) is a high-performance, single-chip implementation of the SPARC reference Floating-Point Unit. The CY7C602 FPU is designed to provide execution of single and double-precision floating-point instructions concurrently with execution of integer instructions by the CY7C601 Integer Unit (IU). The CY7C602 is compliant to the ANSI/IEEE–754 floating-point standard.

The CY7C602 provides a 64-bit internal datapath, a 64-bit ALU, and a 64-bit multiply/divide/square-root unit for efficient execution of double-precision floating-point instructions. For efficient data management, the CY7C602 provides thirty-two 32-bit floating-point registers. These 32-bit registers can be concatenated for use as 64-bit registers for double-precision operations. The internal 64-bit architecture of the CY7C602 allows high speed execution of both single- and double-precision operations.

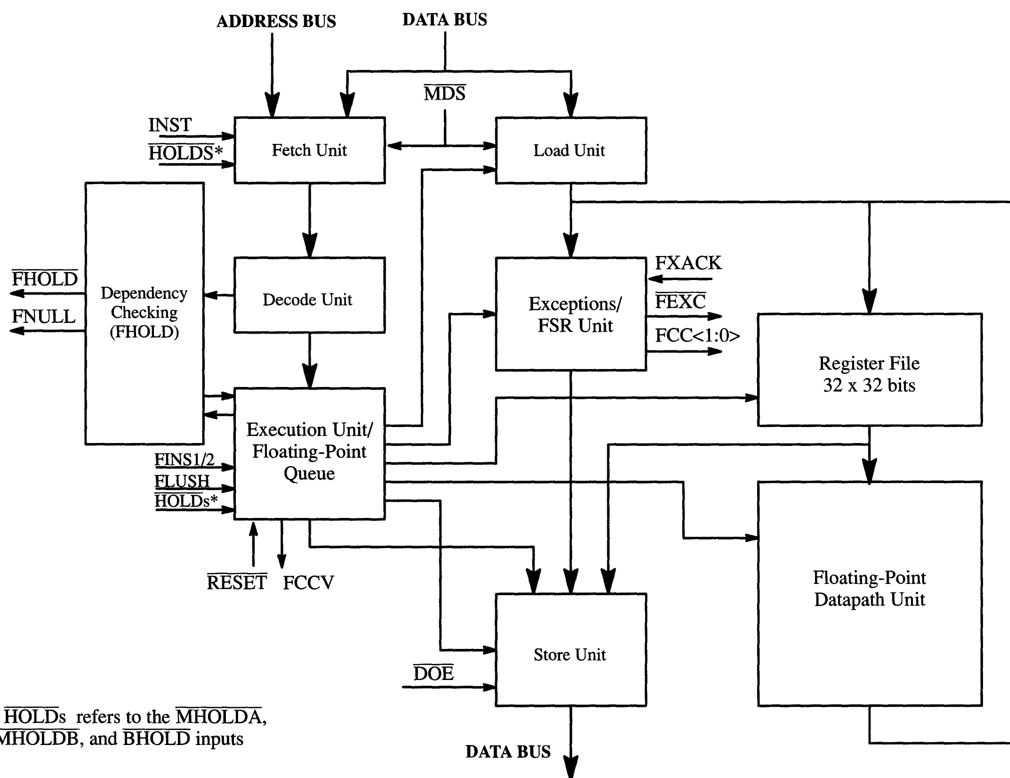


Figure 7–1. CY7C602 Functional Block Diagram

The SPARC floating-point/integer unit interface supports concurrent execution of integer and floating-point instructions. The tightly coupled floating-point/integer unit interface requires the integer unit to provide all addressing and control signals for memory access. All instructions are fetched by the integer unit, and these instructions are simultaneously latched and decoded by both the CY7C601 and CY7C602. Execution of a floating-point instruction is enabled by CY7C601, which signals the CY7C602 to begin execution of the floating-point instruction when that instruction reaches the Execute stage of the CY7C601 instruction pipeline. In the case of a floating-point load or store instruction, the CY7C601 executes the FP load or store in conjunction with the CY7C602 by asserting address and control signals for memory access while the CY7C602 loads or stores the data. All other floating-point instructions execute independently of the integer unit and in parallel with integer instruction execution.

The floating-point/integer unit interface provides hardware interlocking to ensure synchronization between the CY7C601 and CY7C602. Hardware interlocking ensures software compatibility among SPARC systems with different levels of floating-point performance.

7.1 CY7C602 Functional Description

Figure 7–1 illustrates the functional block diagram for the CY7C602. The fetch unit captures instructions and their addresses from the D(31:0) and A(31:0) buses. The decode unit contains logic to decode the floating-point instruction opcodes. The execution unit handles all instruction execution. The execution unit includes a floating-point queue (FP queue), which contains stored floating-point operate (FPop) instructions (see *Section 7.3.2*) under execution and their addresses. The execution unit controls the load unit, the store unit, and the datapath unit.

The load unit holds data that is fetched from memory via the data bus before it is written into the register file. The register file contains the 32 *f*-registers. The exceptions/floating-point status register (FSR) unit keeps the status of completing FPops, as well as the operating mode of the CY7C602. The store unit holds data that is supplied to the data bus during a store operation. The dependency checking unit checks for conditions where the FPU must freeze the CY7C601 Integer Unit pipeline so that an incoming instruction does not overflow the floating-point queue (described below). The datapath unit contains arithmetic logic used by FPops to operate on the data in the register file and is comprised of a 64-bit ALU and a 64-bit multiply/divide/square-root/compare unit. *Figure 7–2* gives a more detailed block diagram of the CY7C602.

The CY7C602 provides three types of registers: *f*-registers, FSR, and the FP queue. The *f*-registers are the 32 floating-point operand registers, each 32-bits in size. Adjacent even-odd *f*-register pairs (for instance, *freg0* and *freg1*) can be concatenated to support double-precision operands. The FSR is a 32-bit status and control register. It keeps track of rounding modes, floating-point trap types, queue status, condition codes, and various IEEE exception information. The floating-point queue contains the floating-point instructions currently under execution, along with their corresponding addresses. The floating-point queue provides an efficient method of handling floating-point exceptions. When an FPop instruction causes a floating-point exception, the queue contains the offending instruction/address pair along with any other instructions that have started execution. The CY7C601 Integer Unit acknowledges the floating-point exception, enters a floating-point trap routine, empties the queue, and corrects the exception case. After the exception case is corrected, unfinished floating-point instructions found in the floating-point queue are either executed or emulated in the trap handler before returning to normal execution.

The CY7C602 depends upon the CY7C601 to assert all addresses and control signals for memory access. Floating-point loads and stores are executed in conjunction with the CY7C601, which provides addresses and control signals while the CY7C602 supplies or stores the data. Instruction Fetch for integer and floating-point instructions is provided by the CY7C601. When the CY7C601 Integer Unit asserts an address for an instruction Fetch, it asserts the INST signal one clock later. The CY7C602 Floating-Point Unit uses INST to determine when a valid instruction is present on the D(31:0) bus. The instruction, which appears on the

data bus on the next clock cycle, is latched and paired with its corresponding address (refer to *Figure 7–4*). In any given cycle, the two previous instruction/address pairs are stored by the CY7C602, regardless of whether the instruction is an integer or floating-point instruction. Either of these two instruction/address pairs may be selected for execution by the CY7C601 upon asserting the FINS1 or FINS2 signal. The CY7C601/CY7C602 interface uses this two stage address/ instruction cache to accommodate delays in the instruction pipeline of the CY7C601 Integer Unit. The FINS1 or FINS2 signals select between the output of the two stages of the address/instruction cache, enabling a floating-point instruction to begin execution by the CY7C602.

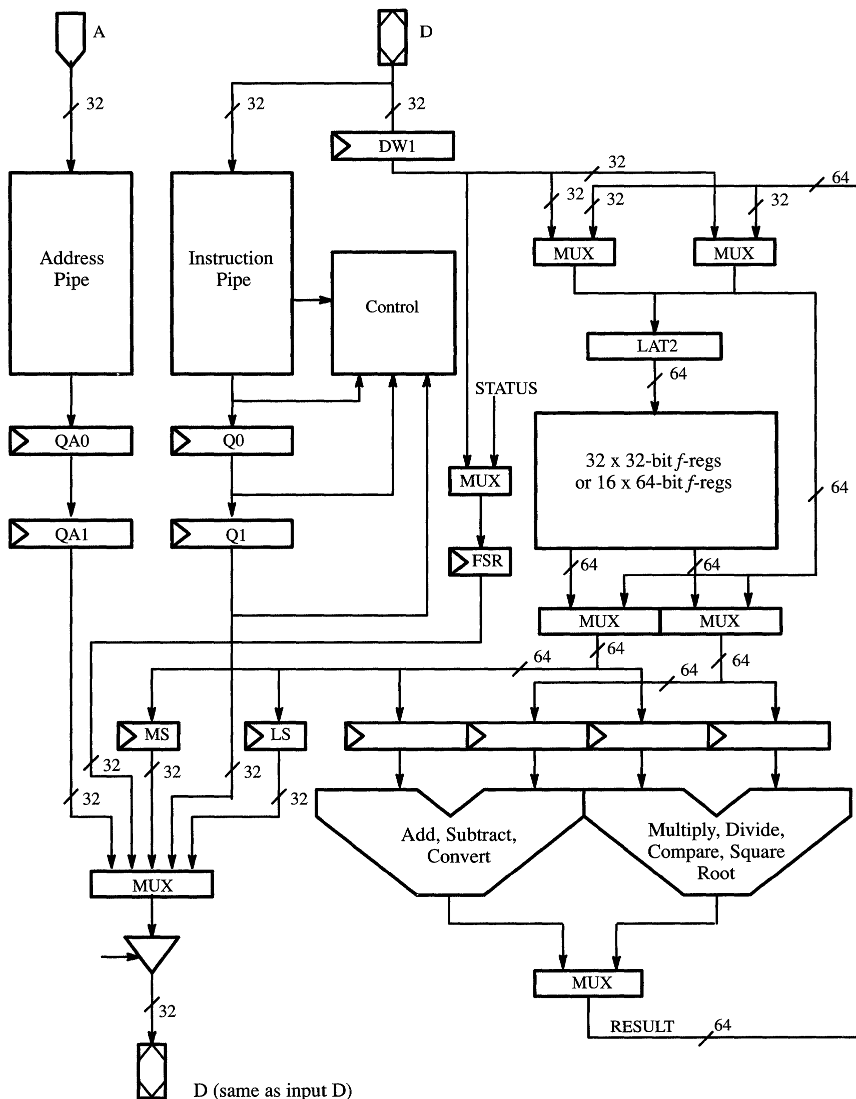


Figure 7–2. CY7C602 Block Diagram

Upon decoding a floating-point instruction, the CY7C601 will assert the FINS1 or the FINS2 signal to enable the CY7C602 to begin execution. The FINS1 or FINS2 signal is asserted during the Decode stage of the floating-point instruction and is recognized by the CY7C602 at the beginning of the Execute stage of the floating-point instruction. This ensures synchronization of the Decode and Execute stages of a floating-point instruction between instruction pipelines of the CY7C601 and the CY7C602.

7.2 Floating-Point/Integer Unit Interface

The CY7C602 is designed to directly interface with the CY7C601 without external glue logic. *Figure 7–3* illustrates the signals required to interconnect the CY7C601 and CY7C602. The control signals illustrated in *Figure 7–3* are used to interface with the remainder of the CPU system components. The FNULL, RESET, BHOLD, MHOLDA or MHOLDB, MDS, and DOE signals are used by the CY7C604 or CY7C605 for cache interface and virtual bus arbitration. The signal descriptions for the CY7C602 signals are described in *Section 7.4*.

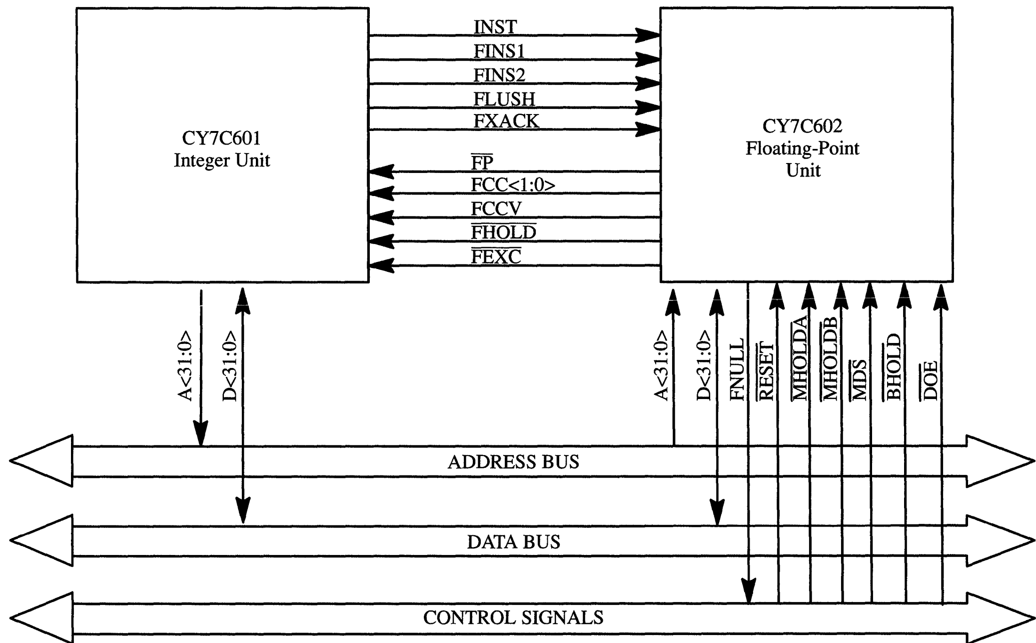


Figure 7–3. CY7C601 – CY7C602 Hardware Interface

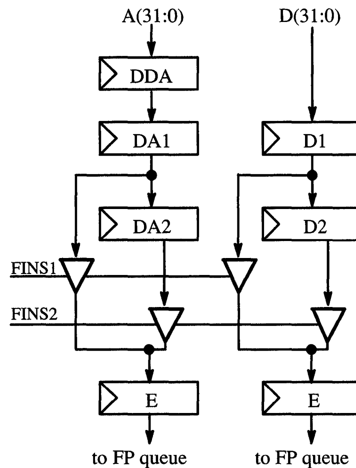


Figure 7–4. CY7C602 Address/Instruction Pipe

7.2.1 CY7C602 Instruction Fetch and Execution

The CY7C602 uses a four-stage instruction pipeline consisting of Fetch, Decode, Execute, and Write stages (F, D, E, and W). The instruction pipelines for the CY7C601 and the CY7C602 are concurrent and synchronized; a floating-point instruction will be in the same stage in both processors. Multiple cycle instructions such as floating-point operate instructions (FPops) leave the pipeline after the W stage and enter the FP queue until completion.

Addresses for both integer unit and Floating-Point Unit instructions are supplied by the CY7C601. The CY7C602 FPU latches all instructions and the corresponding addresses from the D(31:0) and A(31:0) buses. The CY7C602 uses the INST signal, supplied by the CY7C601, to identify an instruction Fetch by the integer unit.

Decode of the latched instruction occurs on the next clock cycle, with both the IU and the FPU decoding the instruction simultaneously. During the Decode stage of the floating-point instruction, the FPU checks for operand and resource dependencies. When the CY7C601 Integer Unit decodes a FPop, it asserts the FINS1 or FINS2 signal. This occurs before the end of the Decode stage, and is used by the CY7C602 to initiate the execution of a floating-point instruction. If the CY7C602 has detected an operand or resource dependency during the Decode stage, the FPU will assert $\overline{\text{FHOLD}}$ as the instruction begins the execution stage. This freezes the integer unit's pipeline until the FPU can resolve the dependency.

If no resource or operand dependencies exist, the decoded floating-point instruction begins execution. Instructions entering execution are stored in the FP queue, where they are held until execution is completed. Note that if the FP queue is full during an instruction's Decode stage, the CY7C602 asserts $\overline{\text{FHOLD}}$ as the instruction enters the execution stage in order to halt the CY7C601. $\overline{\text{FHOLD}}$ is released when space becomes available in the FP queue.

Table 7–1, Table 7–3, and *Table 7–2* describe the execution phases of CY7C602 instructions. Additional cycles beyond the F, D, E, and W stages are denoted as Wh (Write hold). Wh stages are equivalent to the additional cycles held by IOPs in the CY7C601.

Table 7–1. FPop execution

Cycle	Action
D stage	Decode FPop, check resource and operand dependencies
E stage	FHOLD if necessary, read operand(s) from register file
W stage	Read any additional operands from register file; start computing results
FP Queue • • •	Compute, FPop in queue • • •
FP Queue	Check exception status
FP Queue	Update FSR, write results or signal FP exception trap if necessary

Table 7–2. Load instruction execution

Cycle	Action
D stage	Decode instruction, check operand dependencies
E stage	FHOLD if necessary
W stage	Capture data from D(31:0) bus (LDF, LDFSR), capture MSW from D(31:0) bus (LDDF).
Wh1 stage	Write data into register FSR (LDF, LDFSR), capture LSW from D(31:0) bus (LDDF)
Wh2 stage	Write data into register (LDDF)

Table 7–3. Store instruction execution

Cycle	Action
D stage	Decode instruction, check operand dependencies
E stage	FHOLD if necessary, read data from FSR register or FP queue
W stage (mid-cycle)	Drive data onto D(31:0) bus (STF, STFSR), drive MSW or FP queue address onto D(31:0) bus (STDF, STDFQ)
Wh1 stage (mid-cycle)	Stop driving D(31:0) bus (STF, STFSR), drive LSW or FP queue opcode onto D(31:0) bus (STDF, STDFQ)
Wh2 stage (mid-cycle)	Stop driving D(31:0) bus

7.2.1.1 Instruction Fetch

As the CY7C601 fetches an instruction, the CY7C602 captures it at the same time from the D(31:0) bus. The address corresponding to this instruction is captured from the A(31:0) in the previous cycle. The INST signal is used to determine when a valid instruction is present on the D(31:0) bus, and when a valid address has been fetched from the A(31:0) bus in the previous cycle. *Figure 7–5* illustrates an example of an instruction Fetch with a cache hit. The transactions on the address and data buses show two instruction Fetches followed by a data Fetch.

In the case of an instruction Cache miss, a memory hold signal ($\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, or $\overline{\text{BHOLD}}$) is driven low by the cache system starting in the cycle following the instruction Fetch. The instruction which was

captured from the D(31:0) bus is invalid and is replaced when the system returns a valid instruction on the D(31:0) bus. The hold signal lasts for several cycles during which time the $\overline{\text{MDS}}$ signal is asserted by the cache system, notifying the CY7C602 that the valid instruction is available on the D(31:0) bus. $\overline{\text{MDS}}$ is also used when there is a cache miss on data (via load instructions) so the instruction is reloaded only if INST was asserted in the previous non-hold cycle. The same sequence of transactions in *Figure 7–5* are used in *Figure 7–6*, except that the second instruction Fetch (Inst 2) experiences a cache miss.

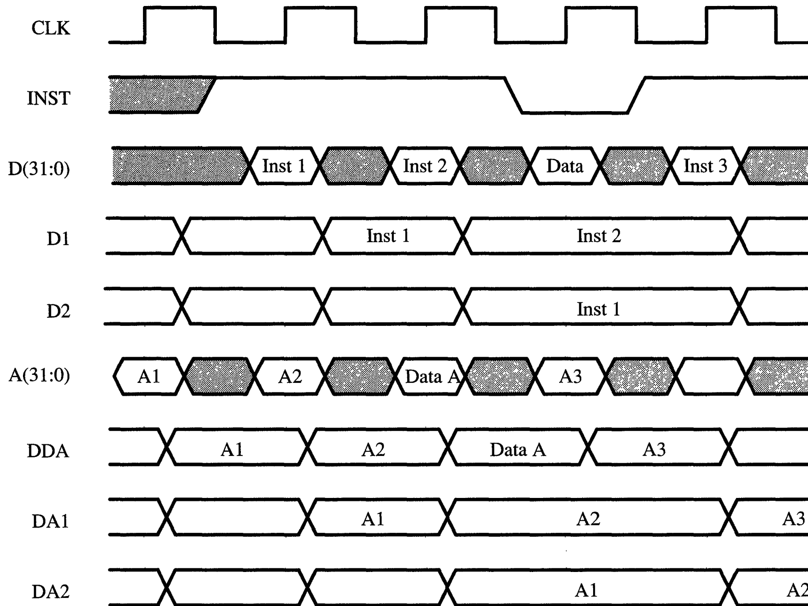


Figure 7–5. Instruction Fetch (Cache Hit)

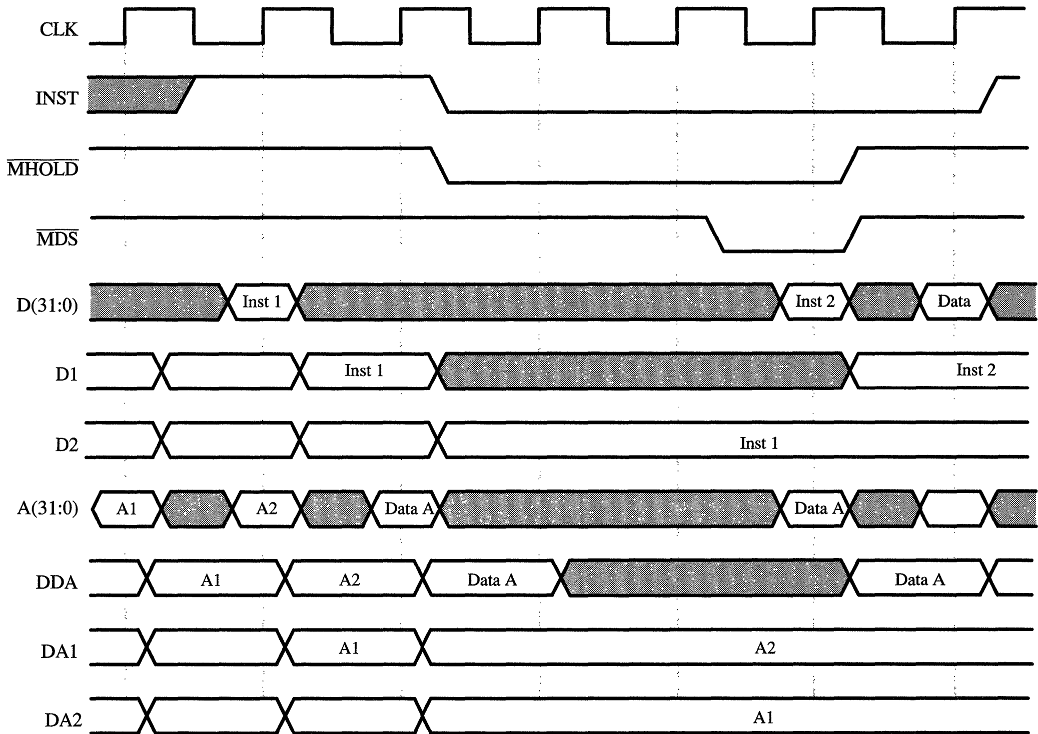


Figure 7–6. Instruction Fetch (Cache Miss on A2)

7.2.1.2 Instruction Execution

The FINS1 and FINS2 signals notify the CY7C602 when to launch a floating-point instruction. When FINS1/FINS2 is received, the floating-point instruction is in the D stage of the CY7C601 Integer Unit pipeline. The example in *Figure 7–7* shows a situation where both FINS1 and FINS2 are used. A load instruction is immediately followed by two FPops. The FPops are fetched while the load instruction is executing. Because the load takes more than one cycle to execute, the starting of the FPops are deferred, and thus two instructions are held in the instruction caches of the CY7C602. When the CY7C601 reaches the D stage of the first FPop (Inst 2), it issues FINS2 to start the FPop. When the D stage of the second FPop (Inst 3) is reached, FINS1 is issued to start the second FPop.

FINS1 and FINS2 are never asserted in the same cycle. Both FINS1 and FINS2 are ignored in the following conditions:

1. FLUSH is asserted.
2. $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, $\overline{\text{BHOLD}}$,$\overline{\text{CHOLD}}$, or $\overline{\text{FHOLD}}$ is asserted.
3. FCCV or CCCV is deasserted.

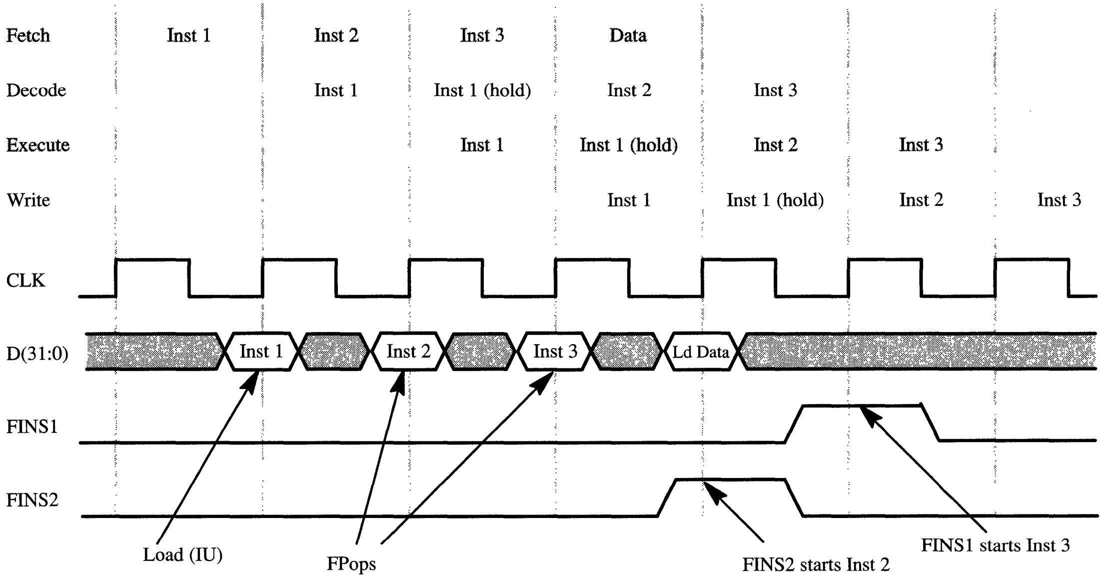


Figure 7–7. Floating-Point Instruction Dispatching

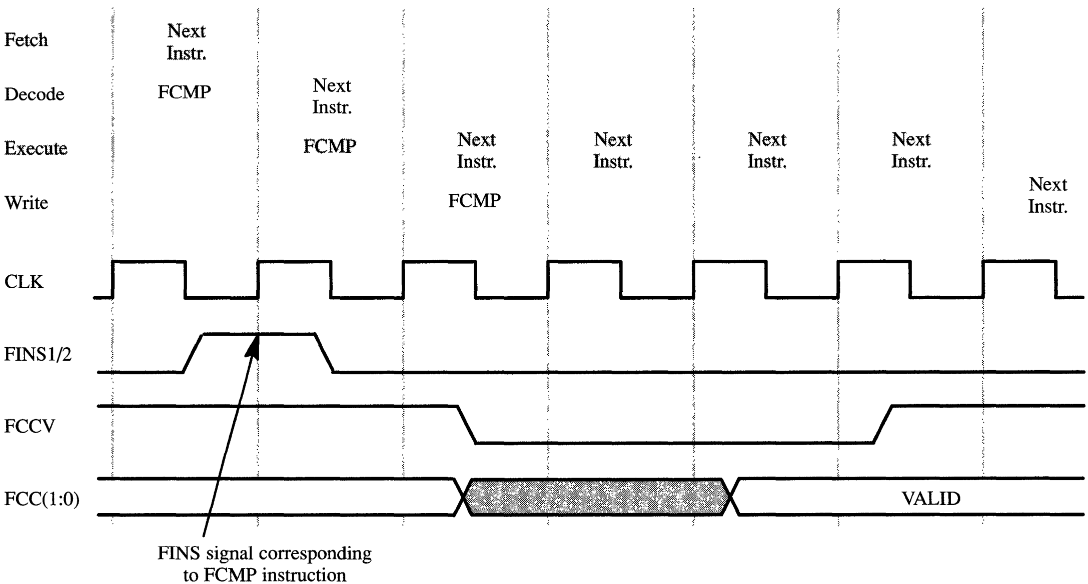


Figure 7–8. Floating-Point Compare (FCMP) Execution

7.2.1.2.1 Floating-Point Compare Execution

Floating-point compare instructions cause the instruction pipeline to be frozen by the use of FCCV, starting from the E stage of the instruction following the compare instruction until the FCC condition codes become valid. FCCV is deasserted, causing the CY7C601 to halt execution until FCCV is asserted. *Figure 7–8* illustrates the timing of FCCV relative to the FCMP instruction and the FCC condition codes.

FCCV is deasserted in the W stage of the FCMP instruction. The instruction that immediately follows the FCMP is held in its E stage until FCCV is reasserted. FCC(1:0) is valid one cycle before FCCV is reasserted. For unimplemented compare instructions, the CY7C602 freezes the instruction pipeline and causes an unimplemented FPop trap, which the CY7C601 takes immediately.

7.2.1.2.2 FPop Queuing

When a FPop has passed the first cycle of the W stage and FLUSH has not been asserted, the FPop enters the FP queue. Note that the W stage of an FPop may be extended to more than one cycle if a hold condition exists. As an FPop completes execution successfully and results are written to the register file, it is removed from the FP queue. The front entry of the FP queue contains the instruction/address pair of the oldest FPop which is still being executed by the CY7C602.

7.2.2 Instruction Pipeline Flush

When a trap or interrupt occurs in the integer unit, normal program execution is halted and control is transferred to the trap handler. The instruction in the E stage of the pipeline and any instructions fetched after it are aborted and must be restarted after the trap handler is done (or emulated in the trap handler). Instructions that have not yet been transferred to the FP queue are aborted by the CY7C602 when the trap occurs. The CY7C601 asserts the FLUSH signal in the W stage of the instruction to be aborted (refer to *Figure 7–9*). FPops which were issued before this instruction continue execution (and are in the queue) while instructions issued after it are aborted.

The following figures illustrate how each type of floating-point instruction is affected by the FLUSH signal. *Figure 7–10* illustrates the effect of the FLUSH signal during a load floating-point instruction (LDF). A FLUSH signal asserted any time on or before the last Wh stage of a load instruction causes the load to abort, leaving the contents of the floating-point register file unchanged.

Figure 7–11 illustrates the effect of FLUSH on a store floating-point instruction (STF). A FLUSH signal asserted on or before the last Wh stage of a store instruction causes the store to abort and the CY7C602 to stop driving the D(31:0) bus by the middle of the next clock cycle.

Figure 7–12 illustrates the effect of FLUSH on an FPop instruction. A FLUSH signal asserted anytime on or before the W stage of a FPop instruction causes the FPop to abort, leaving the contents of the register file and the FSR unchanged by that instruction. FPops that have passed the W stage but are still executing (stored in the FP queue) are not affected.

Figure 7–13 illustrates the effect of FLUSH on a floating-point compare. FLUSH asserted in the W stage of a FCMP instruction causes the FCMP to abort, leaving the FSR unchanged by that instruction. FCCV is reasserted in the next clock cycle.

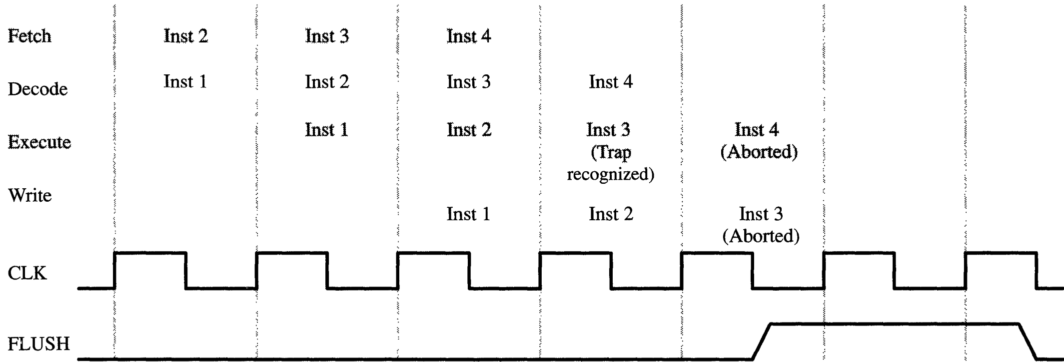


Figure 7–9. Floating-Point Instruction Pipeline During A Trap

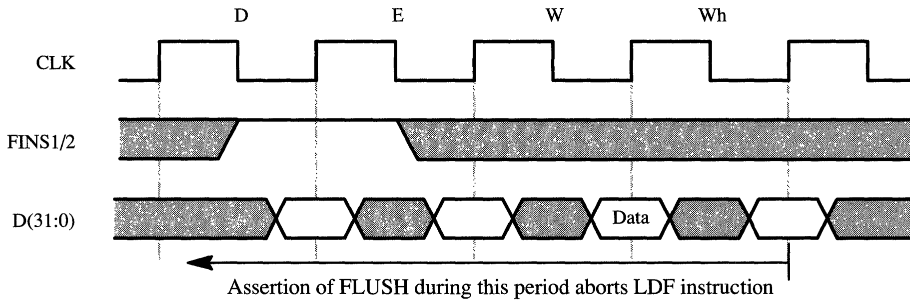


Figure 7–10. Effect of FLUSH on LDF Instruction

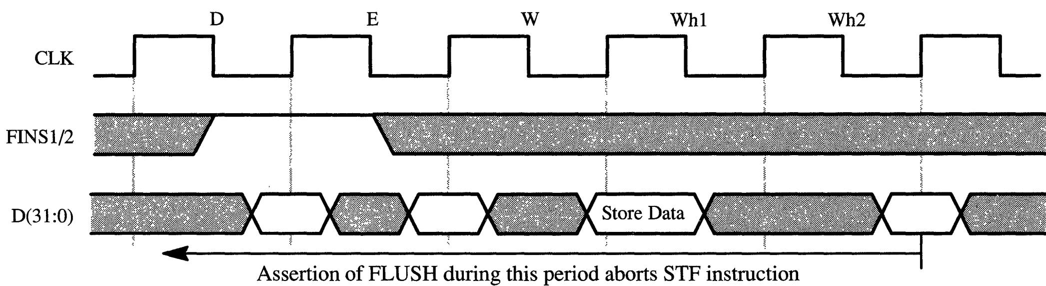


Figure 7–11. Effect of FLUSH on STF Instruction

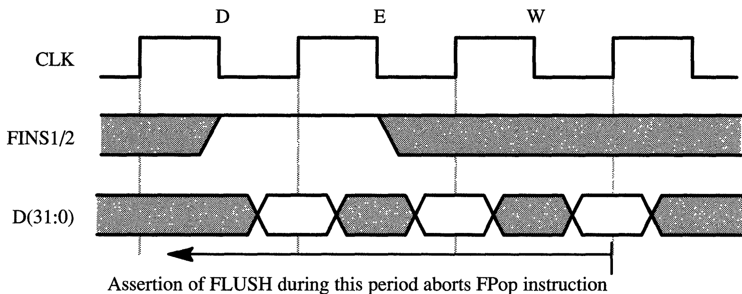


Figure 7–12. Effect of FLUSH on FPop Instruction

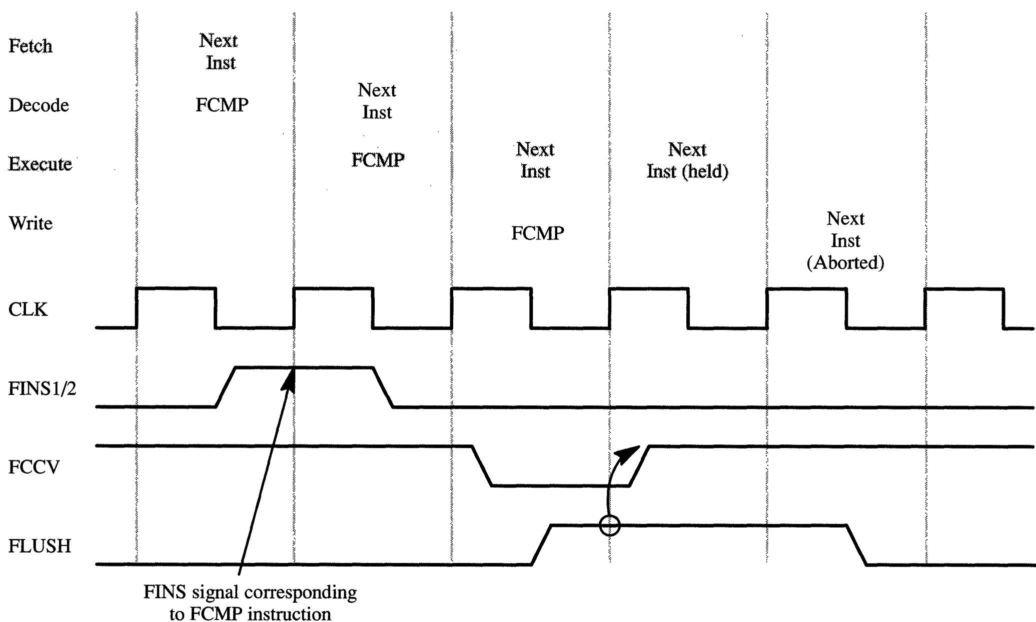


Figure 7–13. Effect of FLUSH on FCMP Instruction

7.2.2.1 Hold Signals

If MHOLDA, MHOLDB, BHOLD, CHOLD, or FHOLD is active, or FCCV or CCCV is inactive, the instruction pipelines of the CY7C601 and CY7C602 are frozen. FHOLD and FCCV are generated by the CY7C602, CHOLD and CCCV are generated by the coprocessor, and the others are generated by the system.

In the CY7C602, "freezing" or "holding" the instruction pipeline means that instructions that are still being tracked by the CY7C601 are not allowed to continue executing. The instructions are allowed to continue execution when all of the hold signals are inactive and all of the condition code valid signals are active.

Holds affect all Load/Store instructions, and only FPops which are in the F, D, and E stages of the instruction pipeline. Hold signals do not affect the execution of FPops in the FP queue.

7.2.2.2 Interlocking with FHOLD

In some situations it is necessary to stop the CY7C601 pipeline, either because a FP Load/Store instruction must be suspended due to an operand dependency, or because the CY7C602 cannot accept any more instructions due to a resource dependency. FHOLD is used to freeze the instruction pipeline in these cases. *Table 7–4* describes mandatory conditions under which FHOLD is asserted.

Operand dependencies listed in *Table 7–4* apply to all FPops that are defined in the architecture. For example, suppose an unimplemented FPop is in the FP queue, waiting to cause an exception. If a store instruction is issued to the CY7C602 to store the contents of the unimplemented FPop's destination register, the store instruction must cause a FHOLD so that the wrong data is not stored. The unimplemented FPop eventually causes a trap that is taken by the CY7C601 in the E stage of the store instruction.

The following simplification could be applied when handling all unimplemented FPops: when an unimplemented FPop has been issued to the CY7C602 but has not yet caused a trap, assert FHOLD on the next floating-point instruction issued until FEXC is asserted. There is no loss in performance because any FPops entering the FP queue after the unimplemented FPop would be re-executed after the unimplemented FPop has been taken care of in the trap handler.

Table 7–4. FHOLD Resource/Operand Dependency Cases

Resource Dependencies:		
If the CY7C602 will not have FP queue entries available to accommodate additional FPops, the CY7C602 asserts FHOLD to stop the CY7C601 from issuing any more instructions to the CY7C602.		
Operand Dependencies:		
LDF, LDDF	Load data from memory to f-register	Load instructions must not overwrite the source or destination registers of any FPop that has not completed execution. In other words, the rd field of the load instruction must not refer to the same *f*-register as any valid rs1, rs2 or rd field of an outstanding FPop. The source registers of FPops (rs1, rs2) may not be altered because an FP exception trap would require that the source registers be unaltered for the trap handler.
STF, STDF	Store data from f-register to memory	If a store instruction accesses an *f*-register that is the destination register of an FPop that has not yet finished execution, the store instruction waits until all outstanding FPops with that register as a destination are complete.
LDFSR, STFSR	Load/Store data between memory and floating-point status register	If any instructions are currently executing in the CY7C602 when a LDFSR/STFSR instruction is issued by the CY7C601, the CY7C602 holds until all instructions have completed execution and are no longer in the FP queue.

If the CY7C602 goes into exception mode, FHOLD is deasserted. If there is a floating-point sequence error (see *Section 7.3.3*), FHOLD is asserted for one cycle. This is the only case where FHOLD is asserted in the exception mode.

If a floating-point trap condition occurs while FHOLD is asserted, FHOLD is deasserted at least one cycle after FEXC is asserted. Similarly, if FCCV is deasserted, it is reasserted at least one cycle after FEXC is asserted. For the FHOLD case, the CY7C601 takes the FP trap on the FP instruction that triggered the FHOLD.

7.2.2.3 FNULL Signal

FNULL is used to signal a pipeline delay of the CY7C601 by the CY7C602. FNULL replaces FCCV and FHOLD for informing the system that the pipeline is being held. FNULL is asserted when either FHOLD is asserted or FCCV is deasserted. This signal is used as an input by the CY7C604/605 to monitor pipeline freezes initiated by the CY7C602.

7.3 CY7C602 Programming Model

7.3.1 CY7C602 Registers

The CY7C602 has three types of user accessible registers: the *f*-registers, the FP queue, and the floating-point status register (FSR). The *f*-registers are the CY7C602 data registers. The FSR is the CY7C602 status and operating mode register. The FP queue contains the CY7C602 instructions that have started execution and are awaiting completion. The following section describes these registers in detail.

7.3.1.1 f-Registers

The CY7C602 provides 32 registers for floating-point operations, referred to as *f*-registers. These registers are 32 bits in length, which can be concatenated to support 64-bit double words. Extended precision instructions are not supported in the CY7C602, but the extended precision data format and its position in the SPARC FPU is defined for the SPARC architecture. *Figure 7–14* illustrates the data organization for the *f*-registers.

Integer and single precision data requires a single 32-bit *f* register. Double precision data requires 64 bits of storage and occupies an even-odd pair of adjacent *f*-registers. Extended precision data requires 128 bits of storage and occupies a group of four consecutive *f*-registers, always starting with register f0, f4, f8, f12, f16, f20, f24, or f28.

The CY7C602 forces register addressing to match the data type specified by the floating-point instruction. This ensures data alignment in the *f*-register file for double and extended precision data. *Figure 7–15* illustrates how the CY7C602 uses the five register address bits in a floating-point instruction for the different types of data. Single data word transfers (integer, single-precision floating-point) can be stored in any register. Consequently, all five bits of the register address specified in the floating-point instruction are valid. Double precision data must reside in an even-odd pair of adjacent registers. By ignoring the LSB of the register address for a FPop requiring a register pair, the CY7C602 ensures data alignment. In a similar manner, the two LSBs of the register address are ignored in a SPARC FPU that supports extended precision data.

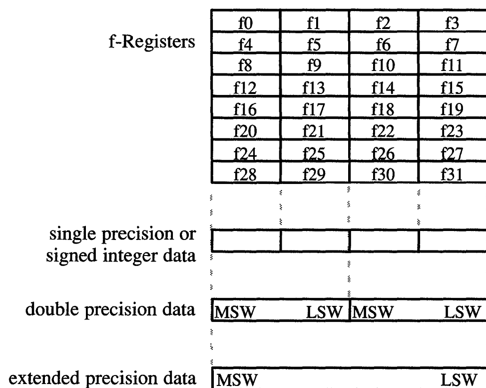


Figure 7–14. *f*-Register Organization

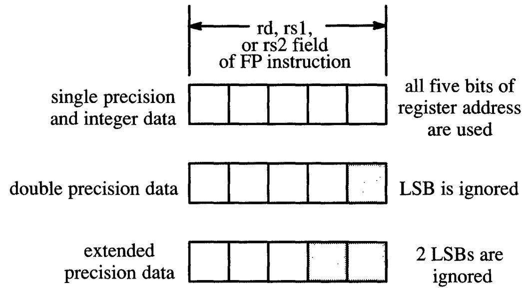


Figure 7–15. *f*-Register Addressing

7.3.1.2 FP Queue

The CY7C602 maintains a floating-point queue of instructions that have started execution, but have yet to complete execution. The FP queue is used to accommodate the multiple clock nature of floating-point instructions and to support the handling of FP exceptions.

When the CY7C602 encounters an exception case, it asserts $\overline{\text{FEXC}}$ and enters pending exception mode. The CY7C602 remains in pending exception mode until the CY7C601 encounters another floating-point instruction, at which time the CY7C601 asserts the FXACK signal to force the CY7C602 into exception mode. When the CY7C602 enters the exception mode, floating-point execution halts until the FP queue is emptied. This allows the CY7C601 to store the floating-point instructions under execution when the exception case occurred. Emptying the FP queue frees the CY7C602 for use by the trap handler without losing the pre-exception state of the CY7C602.

The FP queue contains the 32-bit address and 32-bit FPop instruction of up to two instructions under execution. Floating-point load and store instructions and FP branch instructions are not queued. The front entry of the FP queue is accessible by executing the store double floating-point queue (STDFQ) instruction. The FP queue acts as a FIFO stack, pushing later entries to the top of the stack as the top entry is removed (or executed). A load FP queue instruction does not exist, as the FP queue must be loaded by launching instructions.

7.3.1.3 Floating-Point Status Register (FSR)

The following paragraphs describe the bit fields of the floating-point status register (FSR). Refer to *Figure 7–16* for bit assignments for the FSR fields.

RD *FSR(31:30)* Rounding Direction: These two bits define the rounding direction used by the CY7C602 during an FP arithmetic operation.

RP *FSR(29:28)* Rounding Precision: These two bits define the rounding precision to which *extended-precision* results are rounded. This bit is included in accordance with the ANSI/IEEE STD-754-1985. The CY7C602 does not currently support rounding of extended-precision results and this bit does not affect CY7C602 operation.

TEM *FSR(27:23)* Trap Enable Mask: These five bits enable traps caused by FPops. These bits are ANDed (1= enable, 0= disable) with the bits of the CEXC (current exception field) to determine whether to force a floating-point exception to the CY7C601. All trap enable fields correspond to the similarly named bit in the CEXC field (see below). The TEM field only affects which bits in the CEXC field will cause the $\overline{\text{FEXC}}$ signal to be asserted.

NS *FSR(22)* Non-Standard floating-point: This bit enables non-standard floating-point operations in the CY7C602. When enabled, the CY7C602 inserts zeros for denormalized floating-point numbers before using them in a floating-point operation. The CY7C602 also writes back zero if a denormalized number results from an operation. This is not consistent with the IEEE-754-1985 specification, and is therefore, non-standard.

version *FSR(19:17)* The version number is used to identify the SPARC floating-point processor type. This field is set to 011 (3H) for the CY7C602, and is read-only.

FTT *FSR(16:14)* Floating-point Trap Type: This field identifies the floating-point trap type of the current FP exception. This field is read-only.

QNE *FSR(13)* Queue Not Empty: This bit signals whether the FP queue is empty. (0= empty, 1= not empty)

FCC *FSR(11:10)* Floating-point Condition Codes: These two bits report the FP condition codes (see *Figure 2–10*).

AEXC *FSR(9:5)* Accumulated Exceptions: This field reports the accumulated FP exceptions that are masked by the TEM field. All masked exception cases are ORed with the contents of the AEXC and accumulated as status. All accumulated fields have the same definition as the corresponding field for CEXC (see below). This field can be read and written, and must be cleared by software (see *Figure 2–10*).

CEXC *FSR(4:0)* Current EXCeptions: This field reports the current FP exceptions. This field is automatically cleared upon the execution of the next floating-point instruction. CEXC status is not lost upon assertion of a floating-point exception, because instructions following a valid exception are not executed by the CY7C602. The five CEXC bits are:

nvc = 1 Indicates invalid operation exception. This is defined as an operation using an improper operand value. An example of this is 0/0.

ofc = 1 Indicates overflow exception. The rounded result would be larger in magnitude than the largest normalized number in the specified format.

ufc = 1 Indicates underflow exception. The rounded result is inexact, and would be smaller in magnitude than the smallest normalized number in the indicated format.

dzc = 1 Indicates division-by-zero: X/0, where X is subnormal or normalized. Note that 0/0 does not set the dzc bit.

nxc = 1 Indicates inexact exception. The rounded result differs from the infinitely precise correct result.

R *FSR21, 20, and 12*. Reserved – always set to 0.

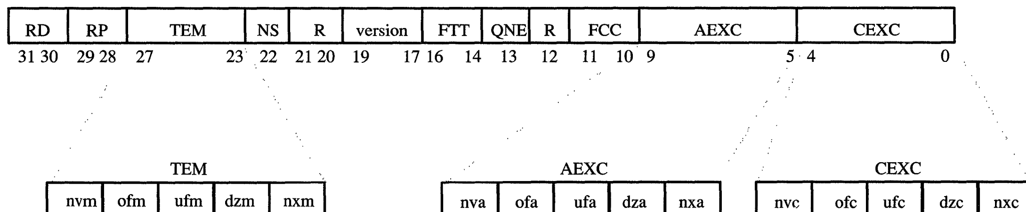


Figure 7–16. Floating-Point Status Register

Table 7–5. Floating-Point Status Register Summary

Field	Values	FSR bits	Description	Loadable by LDFSR
RD	0 – Round to nearest (tie–even) 1 – Round to 0 2 – Round to +1 3 – Round to – 1	31:30	Rounding Direction	yes
RP	0 – Extended precision 1 – Single precision 2 – Double precision 3 – Reserved	29:28	Extended Rounding Precision	yes
TEM	0 – Disable trap 1 – Enable trap NVM OFM UFM DZM NXM	27:23 27 26 25 24 23	Trap Enable Mask invalid operation trap mask overflow trap mask underflow trap mask divide by zero trap mask inexact trap mask	yes
NS	0 – Disable 1 – Enable	22	Non-standard Floating-point: 0 = IEEE mode; multiplier and ALU generate denormalized operand exceptions and produce unrounded normalized values on underflow exceptions. 1 = FAST mode; multiplier and ALU flush denormalized operands to zero and round underflow results to zero.	yes
version	0 – 7	19:17	FPU version number	no
FTT	0 – None 1 – IEEE Exception 2 – Unfinished FPop 3 – Unimplemented FPop 4 – Sequence Error 5 – 7 Reserved	16:14	Floating-point trap type	no
QNE	0 – queue empty	13	Queue Not Empty	no
FCC	0 – = 1 – < 2 – > 3 – Unordered	11:10	Floating-point Condition Codes	yes
AEXC	 NVA OFA UFA DXA NXA	9:5 9 8 7 6 5	Accrued Exception Bits accrued invalid exception accrued overflow exception accrued underflow exception accrued divide by zero exception accrued inexact exception	yes
CEXC	 NVC OFC UFC DZC NXC	4:0 4 3 2 1 0	Current Exception Bits current invalid exception current overflow exception current underflow exception current divide by zero exception current inexact exception	yes
r	Always set to 0	21, 20, 1 2	reserved bits	no

7.3.2 CY7C602 Floating-Point Instructions

SPARC floating-point instructions are separated into three groups: floating-point Load/Store, floating-point branch (FBfcc), and floating-point operate instructions (FPops). Floating-point Load/Store instructions are used to transfer data to and from the data registers (*f* registers). FP Load/Store instructions also allow the CY7C601 Integer Unit to read and write the floating-point status register (FSR) and to read the front entry of the floating-point queue. Floating-point load and store instructions are executed by both the CY7C601 and the CY7C602; the CY7C601 supplying all address and control signals for memory access and the CY7C602 loading or storing the data.

Floating-point branch (FBfcc) instructions (and coprocessor branch instructions (CBccc)) are executed by the CY7C601, since the CY7C601 is responsible for generating address and control signals for memory access. Conditional FBfcc branches are based upon the FCC(1:0) signals supplied by the CY7C602. FCC(1:0) is set by executing a FCMP instruction, which belongs to the FPop group of instructions. Floating-point branch instructions will cause the CY7C601 to recognize a pending floating-point exception in the same manner as other floating-point instructions (see *Section 7.3.3*).

FPops include all other floating-point instructions executed by the CY7C602. Floating-point operate instructions (FPops) include basic numeric operations (add, subtract, multiply, and divide), conversions between data types, register to register moves, and floating-point number comparison. FPops operate only on data in the floating-point registers.

The SPARC architecture supports four data types: 32-bit signed integer, single-precision FP, double-precision FP, and extended-precision FP. Extended precision instructions are defined in the SPARC architecture, but are not supported in the CY7C602. The CY7C602 supports execution of extended precision floating-point instructions by asserting an unimplemented instruction trap. This allows the CY7C601 to trap to a software emulation of extended precision floating-point.

Seven Load/Store instructions are executed by the CY7C602. The following describes the CY7C602 Load/Store instructions:

- LDF and LDDF transfer data from memory to *f*-registers 32 and 64 bits at a time, respectively.
- STF and STDF transfer data from the *f*-registers to memory in data widths of 32 and 64 bits.
- LSFSR and STFSR allow the FSR to be read and written to.
- STDFQ is a privileged instruction which allows the FP queue to be read.

All FPops operate only on data located in the *f*-registers. The FPops are divided into four groups: basic arithmetic operations, compares, format conversions, and register-to-register moves. Move operations do not cause exceptions. The converts, moves and the square root instruction use only a single source operand. FP compare instructions modify only the FCC(1:0) signals. FPops are dispatched in one cycle in the CY7C601, and require multiple cycles to execute in the CY7C602.

Floating-point performance can be improved in the CY7C602 by scheduling FPop instructions such that the floating-point ALU and the floating-point multiply/divide/compare/square-root units are concurrently operating. With the exception of data dependencies, the ALU and multiply/divide/compare/square-root units are independent and can execute separate instructions without requiring the other unit to complete execution. Therefore, an FPop using the ALU followed by a FPop using the multiply/divide/compare/square-root unit does not require the previous instruction to finish before starting (assuming there are no data dependencies).

Table 7–6 and *Table 7–7* illustrate the CY7C602 instructions and their execution cycle count. For further information on the SPARC floating-point instructions, please refer to *Chapter 6, SPARC Instruction Set.*

Table 7–6. Floating-Point Load and Store Instruction Cycle Count

Mnemonic	Operation	Cycles
LDF	load floating-point	2
LDDF	load double floating-point	3
LDFSR	load FSR	2
STF	store floating-point	3
STDF	store floating-point double	4
STFSR	store FSR	3
STDFQ	store double FP queue	4

Table 7–7. Floating-Point Operate (FPops) Instruction Cycle Count

Mnemonic	Operation	Cycles
FABSs	absolute value	4
FADDs	add single	4
FADDd	add double	4
FCMPs	compare single	4
FCMPd	compare double	4
FCMPEs	compare single and exception if unordered	4
FCMPEd	compare double and exception if unordered	4
FDIVs	divide single	14
FDIVd	divide double	21
FMOVs	move	4
FMULs	multiply single	4
FMULd	multiply double	5
FNEGs	negate	4
FSQRTs	square root single	19
FSQRTd	square root double	34
FSUBs	subtract single	4
FSUBd	subtract double	4
FdTOi	convert double to integer	4
FdTOs	convert double to single	4
FiTOs	convert integer to single	8
FiTOd	convert integer to double	4
FsTOi	convert single to integer	4
FsTOd	convert single to double	4

7.3.3 CY7C602 Internal Operation

The CY7C602 operates in one of three modes: execution mode, pending exception mode, and exception mode (see *Figure 7–17*). After reset, the CY7C602 enters execution mode, which is the normal mode of operation. When the CY7C602 encounters a floating-point exception condition, the CY7C602 asserts $\overline{\text{FEXC}}$ and enters the pending exception mode. All FPop instructions under execution at this point are suspended. The CY7C601 asserts FXACK and enters the floating-point trap when the next floating-point instruction is encountered. Upon receiving FXACK, the CY7C602 FPU enters exception mode. The

CY7C602 returns to execution mode as soon as the trap handler empties the FP queue using store double floating-point queue instructions (STDFQ).

7.3.3.1 Exception Handling

Upon encountering an exception condition, the CY7C602 asserts $\overline{\text{FEXC}}$ to notify the CY7C601 that a floating-point exception has occurred and enters the pending exception mode. The CY7C601 enters the trap handler on the next floating-point instruction it encounters in the instruction stream, asserting FXACK to signal to the CY7C602 that the trap is being taken. At this point, the CY7C602 enters exception mode (see *Figure 7–17*).

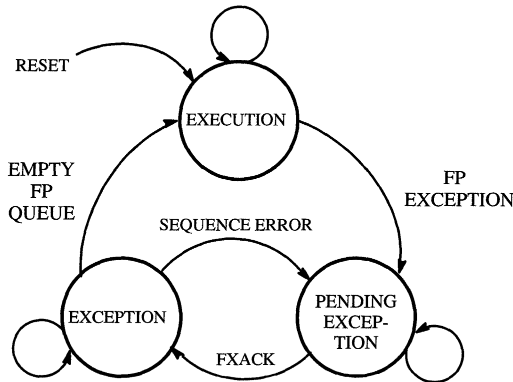


Figure 7–17. FPU Operation Modes

Upon receiving FXACK from the CY7C601, the mode of the CY7C602 changes from pending exception to exception mode. All FPops in the CY7C602 stop executing during pending exception and exception modes. While in exception mode, the CY7C602 will execute only store floating-point instructions until the FP queue is emptied. All floating-point store instructions are allowed while in this operating mode. Any load or FPop issued to the CY7C602 while in this mode causes a sequence error and returns the CY7C602 to exception pending mode. Once the queue is emptied by successive STDFQ instructions, the CY7C602 returns to execution mode.

Due to the latency of floating-point instruction execution, an exception caused by a FPop occasionally may not occur until one or more FP instructions have been fetched and executed (or entered into the FP queue for execution). This is a case where $\overline{\text{FEXC}}$ is not asserted before the next floating-point instruction is fetched and executed. In this case, $\overline{\text{FEXC}}$ is asserted as soon as the exception case is recognized, and the CY7C601 acknowledges the FP exception during the Execute stage of the next floating-point instruction fetched after $\overline{\text{FEXC}}$ is asserted.

Figure 7–18 illustrates the handshake of signals between the CY7C601 and the CY7C602 during a floating-point exception. The qne (queue not empty) bit of the FSR is shown in *Figure 7–18* to illustrate the dependency of clearing the FP queue to return to execution mode.

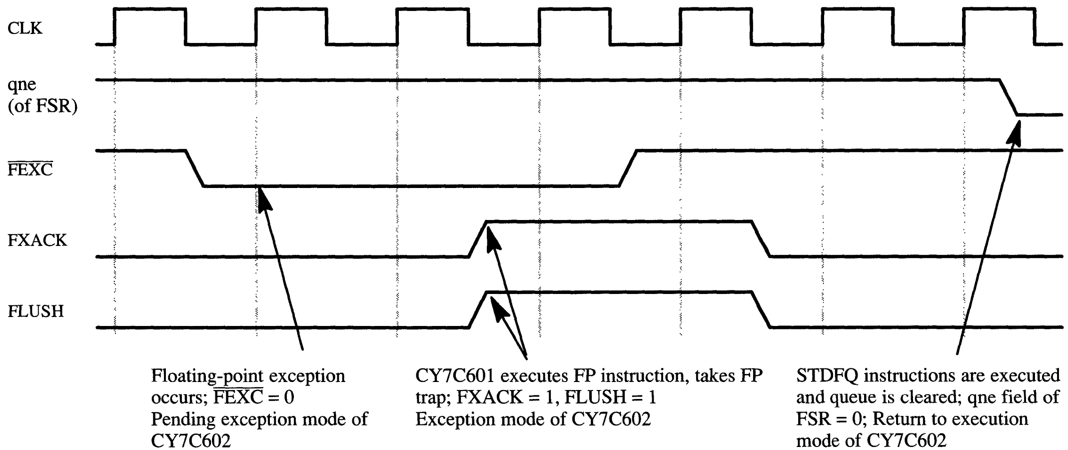


Figure 7–18. Floating-Point Exception Handshake

7.3.4 CY7C602 Exception Cases

The following section describes the CY7C602 exception cases, including exceptions specified by the IEEE-754 standard.

Unfinished FPop. This exception case can occur when operations on normalized floating-point numbers either encounter a denormalized operand or produce a denormalized result. This exception case is asserted upon executing any FPop encountering an NaN as one of the operands. The CY7C602 also asserts this trap when a floating-point to integer conversion overflow occurs.

Unimplemented FPop. This exception is asserted by the CY7C602 upon encountering a defined SPARC FPop instruction that is not supported by the CY7C602. This includes all operations using extended-precision format operands. The trap handler is expected to emulate the unimplemented instruction.

Sequence Error. This exception is asserted by the CY7C602 when a floating-point instruction (other than FP store) is attempted after the CY7C602 has entered either pending exception or exception mode. The CY7C602 suspends all instruction execution with the exception of FP stores until the FP exception has been acknowledged and the FP queue has been cleared.

IEEE Exceptions. This class of exceptions is defined as part of the IEEE-754 Standard. The five exceptions defined as IEEE Exceptions are reported in the CEXC and AEXC fields of the FSR. These exceptions are: invalid, overflow, underflow, division-by-zero, and inexact. The only exceptions that can coincide are inexact with overflow and inexact with underflow. The following paragraphs discuss these exception cases.

Invalid Operation. The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The result, when the exception occurs without a trap, shall be a quiet NaN provided the destination has a floating-point format. The invalid operations are:

1. Any operation on a signaling NaN
2. Addition or subtraction: Magnitude subtraction of infinities such as (+1) + (–1)
3. Multiplication: 0 x 1
4. Division: 0/0 or 1/1
5. Square root if the operand is less than zero

6. Conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled
7. Floating-point compare operations: when one or more of the operands are NaN

Division-by-zero. If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result, when no trap occurs, shall be a correctly signed 1.

Overflow. The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:

1. Round to nearest carries all overflows to 1 with the sign of the intermediate result.
2. Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result.
3. Round toward –1 carries positive overflows to the format's largest positive finite number, and carries negative overflows to –1.
4. Round toward +1 carries negative overflows to the format's most negative finite number, and carries positive overflows to +1.

Underflow. The CY7C602 does not assert an underflow exception. Underflow cases are covered in the unfinished FPop trap, which is asserted in any case where a denormalized number is used as an operand. The unfinished FPop trap handler must resolve the underflow condition and update this bit to reflect correct accumulated exception status (AEXC field of FSR).

Inexact. The inexact exception is generated whenever there is a loss of accuracy (or significance) in the result. The CY7C602 computes results to higher precision than the number of fraction bits in the format. If any of the fraction bits to the right of the LSB was one prior to rounding, the inexact exception is signaled.

7.3.4.1 CY7C602 IEEE-754 Compliance

The CY7C602 meets the requirements of the IEEE Std. 754-1985 for floating-point arithmetic. Accuracy of the results of its operations are within ± 1/2 LSB, as specified by the IEEE standard.

7.4 CY7C602 Signal Descriptions

The following sections describe the external signals of the CY7C602. Active low signals are marked with an overbar, active high signals are not.

7.4.1 Integer Unit Interface Signals

$\overline{\text{FP}}$ *active-low output* Floating-point Present

This signal indicates to the CY7C601 that a FPU is present in the system. In the absence of a FPU, this signal is pulled up to VCC by a resistor. This is a static signal; it always asserts a low output. The CY7C601 generates a floating-point disable trap if $\overline{\text{FP}}$ is not asserted during the execution of a floating-point instruction.

FCC(1:0) *output* Floating-point Condition Codes

The FCC(1:0) bits indicate the current condition code of the FPU, and are valid only if FCCV is asserted. FBfcc instructions use the value of these bits during the Execute cycle if they are valid. If the

FCC(1:0) bits are not valid, then FCCV is released, which halts the CY7C601 until the FCC bits become valid.

Table 7–8. FCC(1:0) Condition Codes

FCC1	FCC0	Condition
0	0	equal
0	1	Op1 < Op2
1	0	Op1 > Op2
1	1	Unordered

FCCV *output* Floating-point Condition Codes Valid

The CY7C602 asserts the FCCV signal when the FCC(1:0) represent a valid condition. The FCCV signal is deasserted when a pending floating-point compare instruction exists in the floating-point queue. FCCV is reasserted when the compare instruction is completed and FCC bits are valid.

$\overline{\text{FHOLD}}$ *output* Floating-point HOLD

The $\overline{\text{FHOLD}}$ signal is asserted by the CY7C602 if it cannot continue execution due to a resource or operand dependency. The CY7C602 checks for all dependencies in the Decode stage, and if necessary, asserts $\overline{\text{FHOLD}}$ in the next cycle. The $\overline{\text{FHOLD}}$ signal is used by the CY7C601 to freeze its pipeline in the same cycle. The CY7C602 must eventually deassert $\overline{\text{FHOLD}}$ to release the CY7C601 pipeline.

$\overline{\text{FEXC}}$ *output* Floating-point exception

The $\overline{\text{FEXC}}$ is asserted if a floating-point exception has occurred. It remains asserted until the CY7C601 acknowledges that it has taken a trap by asserting FXACK. Floating-point exceptions are taken only during the execution of a floating-point instruction. The CY7C602 releases $\overline{\text{FEXC}}$ when it receives FXACK.

FXACK *input* Floating-point exception acknowledge

The FXACK signal is asserted by the CY7C601 to acknowledge to the CY7C602 that the current FP trap is taken.

INST *input* Instruction Fetch

The INST signal is asserted by the CY7C601 whenever a new instruction is being fetched. It is used by the CY7C602 to latch the instruction on the D(31:0) bus into the FPU instruction cache. The CY7C602 has two instruction caches (D1 and D2) to save the last two fetched instructions (see *Figure 7–4*). When INST is asserted, the new instruction enters the D1 buffer and the old instruction is pushed into the D2 buffer.

FINS1 *input* Floating-point instruction in buffer 1

The FINS1 signal is asserted by the CY7C601 during the Decode stage of a FPU instruction if the instruction is stored in the D1 buffer of the CY7C602. The CY7C602 uses this signal to launch the instruction in the D1 buffer into its Execute stage instruction register.

FINS2 *input* Floating-point instruction in buffer 2

The FINS2 signal is asserted by the CY7C601 during the Decode stage of a FPU instruction if the instruction is stored in the D2 buffer of the CY7C602. The CY7C602 uses this signal to launch the instruction in the D2 buffer into its Execute stage instruction register.

FLUSH *input* Floating-point instruction flush

The FLUSH signal is asserted by the CY7C601 to signal to the CY7C602 to flush the instructions in its instruction registers. This may happen when a trap is taken by the CY7C601. The CY7C601 will restart the flushed instructions after returning from the trap. FLUSH has no effect on instructions in the floating-point queue. In addition to freezing the FPU pipeline, the CY7C602 uses FLUSH to shut off the D bus drivers during store operations. To ensure correct operation of the CY7C602, FLUSH must not change state more than once during a clock cycle.

7.4.2 Coprocessor Interface Signals

$\overline{\text{CHOLD}}$ *input* Coprocessor HOLD

The $\overline{\text{CHOLD}}$ signal is asserted by the coprocessor if it cannot continue execution. The coprocessor must check all dependencies in the Decode stage of the instruction and assert the $\overline{\text{CHOLD}}$ signal, if necessary, in the next cycle. The coprocessor must eventually deassert this signal to unfreeze the CY7C601 and CY7C602 pipelines. The $\overline{\text{CHOLD}}$ signal is latched with a transparent latch in the CY7C602 before it is used.

CCCV *input* Coprocessor Condition Codes Valid

The coprocessor asserts the CCCV signal when the CCC(1:0) represent a valid condition. The CCCV signal is deasserted when a pending coprocessor compare instruction exists in the coprocessor queue. CCCV is reasserted when the compare instruction is completed and the CCC(1:0) bits are valid. The CY7C602 will enter a wait state if CCCV is deasserted. The CCCV signal is latched with a transparent latch in the CY7C602 before it is used.

7.4.3 System/Memory Interface Signals

A(31:0) *input* Address bus (31:0)

The address bus for the CY7C602 is an input-only bus. The CY7C601 supplies all addresses for instruction and data Fetches for the CY7C602. The CY7C602 captures addresses of floating-point instructions from the A(31:0) bus into the DDA register. When INST is asserted by the CY7C601, the contents of the DDA is transferred to the DA1 register.

D(31:0) *input/output* Data bus (31:0)

The D(31:0) bus is driven by the FPU only during the execution of floating-point store instructions. The store data is sent out unlatched and must be latched externally before it is used. Once latched, store data is valid during the second data cycle of a store single access and on the second and third data cycle of a store double access. The data alignment for load and store instructions is done inside the FPU. A double word is aligned on an eight-byte boundary. A single word is aligned on a four-byte boundary.

DOE *input* Data Output Enable

The DOE signal is connected directly to the data output drivers and must be asserted during normal operation. Deassertion of this signal three-states all output drivers on the data bus. This signal should be deasserted only when the bus is granted to another bus master, i.e, when either BHOLD, CHOLD, MHOLDA, or MHOLDB is asserted.

MHOLDA, MHOLDB *input* Memory HOLD

Asserting MHOLDA or MHOLDB freezes the CY7C602 pipeline. Either MHOLDA or MHOLDB is used to freeze the FPU (and the IU) pipelines during a cache miss (for systems with cache) or when slow memory is accessed.

BHOLD *input* Bus HOLD

This signal is asserted by the system's I/O controller when an external bus master requests the data bus. Assertion of this signal will freeze the FPU pipeline. External logic should guarantee that after deassertion of BHOLD, the state of all inputs to the chip is the same as before BHOLD was asserted.

MDS *input* Memory Data Strobe

The MDS signal is used to load data into the FPU when the internal FPU pipeline is frozen by assertion of MHOLDA, MHOLDB, or BHOLD.

FNULL *output* Fpu Nullify cycle

This signal signals to the memory system when the CY7C602 is holding the instruction pipeline of the system. This hold would occur when FHOLD is asserted or FCCV is deasserted. This signal is used by the memory system in the same fashion as the integer unit's INULL signal. The system needs this signal because the IU's INULL does not take into account holds requested by the FPU.

RESET *input* RESET

Asserting the RESET signal resets the pipeline and sets the writable fields of the floating-point status register (FSR) to zero. The RESET signal must remain asserted for a minimum of eight cycles.

CLK *input* Clock

The CLK signal is used for clocking the FPU's pipeline registers. It is high during the first half of the processor cycle and low during the second half. The rising edge of CLK defines the beginning of each pipeline stage in the FPU.

CY7C604/CY7C605 Cache Controller and Memory Management Unit

The CY7C604 Cache Controller and Memory Management Unit (CMU) and CY7C605 Cache Controller and Memory Management Unit for Multiprocessing (CMU-MP) are combined memory management unit (MMU) and cache controllers with on-chip cache tag memory. The CY7C604 and CY7C605 are designed as an integral part of the CY7C600 family to provide a high-performance solution for cache and virtual memory support. The CY7C604 is designed for uniprocessor systems, providing control for a 64-Kbyte virtual cache. The CY7C604/605 cache is extendible to 256-Kbyte through the addition of cache RAMs and CY7C604/605s. Expansion of the CY7C604/605 cache increases the number of translation lookaside buffer (TLB) entries available to the system for MMU address translation, as well as increasing the number of cache tag entries available to the cache. Another feature of the CY7C604 is cache locking, which provides deterministic response time for real-time systems controlling time-critical processes. The CY7C604, as well as the CY7C605, provides the SPARC reference MMU and supports the SPARC MBus standard for interfacing to physical memory.

The CY7C605, a derivative of the CY7C604, is designed to support the requirements of multiprocessing systems. The CY7C605 provides two separate cache tag memories, as compared to the single cache tag memory used on the CY7C604. The second cache tag memory is physically addressed and allows concurrent bus snooping without stalling the CY7C601. This allows the CY7C605 to maintain cache coherency with other cache systems without degrading CPU performance. The CY7C605 supports the MBus Level 2 cache coherency protocol.

The MMU portion of the CY7C604 and CY7C605 provides translation from a 32-bit virtual address range (4 Gigabytes) to a 36-bit physical address (64 Gigabytes), as provided in the SPARC reference MMU specification. Virtual address translation is further extended with the use of a context register, which is used to identify up to 4096 contexts or tasks. The cache tag entries and TLB entries contain context numbers to identify tasks or processes. This minimizes unnecessary cache tag and TLB entry replacement during task switching.

The MMU features a 64-entry translation lookaside buffer. The TLB acts as a cache for address mapping entries used by the MMU to map a virtual address to a physical address. These mapping entries, referred to as page table entries or PTEs, allow one of four levels of address mapping. A PTE can be defined as the address mapping for a single 4-Kbyte page, a 256-Kbyte region, a 16-Mbyte region, or a 4-Gbyte region. The TLB entries are lockable, allowing important TLB entries to be excluded from replacement.

The MMU performs its address translation task by comparing a virtual address supplied by the CY7C601 Integer Unit to the address tags in the TLB entries. If the virtual address and the value of the context register match a TLB entry, a TLB "hit" occurs. When this occurs, the physical address stored in the TLB is used to translate the virtual address to a physical address. The access type (read/write of data or instruction) and privilege level (user/supervisor) are checked during translation. If a TLB hit occurs but access-level protection is violated, the MMU signals an exception and the operation ends.

If the virtual address or context does not match any valid TLB entry, a TLB "miss" occurs. This causes a table walk to be performed by the MMU. The table walk is a search performed by the MMU through the address translation tables stored in main memory. The MMU searches through several levels of tables for the PTE corresponding to the virtual address. Upon finding the PTE, the MMU translates the address and selects a TLB entry for replacement, where it then stores the PTE.

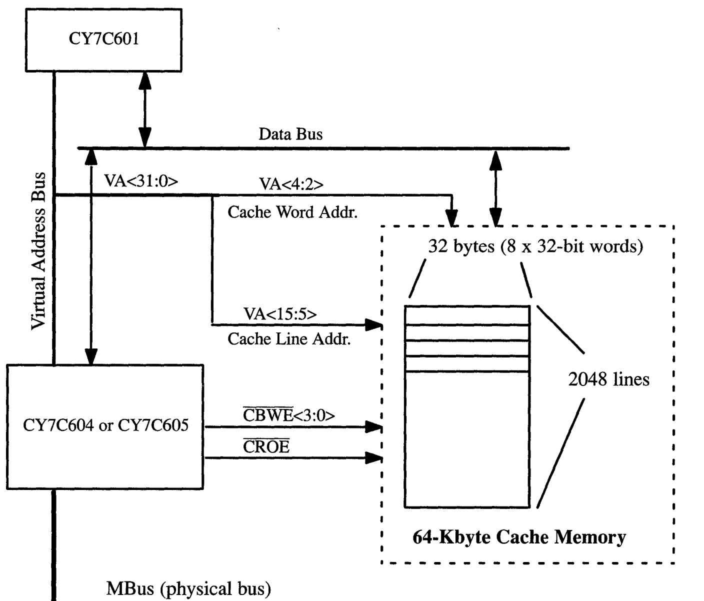


Figure 8–1. Virtual 64-Kbyte Cache

The 64-Kbyte virtual cache is organized into 2048 lines of 32 bytes each. The term "virtual cache" refers to the direct addressing of the cache by the integer unit (CY7C601) with the virtual address bus. Virtual address bits VA(15:5) select the cache line, and virtual address bits VA(4:2) select the 32-bit word of the cache line, as illustrated in *Figure 8–1*. The CY7C604/605 provides access control for the cache by checking the context and virtual address against the cache tags. If the virtual address, access-level, and context match the cache tag for the cache line addressed, a cache hit occurs and the access is enabled. If the virtual address or context do not match the cache tag for the cache line, a cache miss occurs and the cache controller accesses main memory for the required data.

The CY7C604/605 cache controller supports two modes of caching: write-through with no write allocate and copy-back with write allocate. Write-through mode is a simpler style of cache management that causes write accesses to the cache to be written through to main memory upon each write access. The advantage of this method is that the cache always remains coherent with main memory. Its disadvantage is that each write to the cache is echoed to main memory, which increases traffic on the system bus. Another disadvantage to write-through is that the processor is delayed by the time required to arbitrate the system bus and write the data to main memory. However, in the case of the CY7C604/605, this disadvantage is significantly offset by the inclusion of write buffers. The write buffers can store up to four doubleword accesses, allowing the CY7C601 to continue execution while data is written to main memory.

Copy-back cache mode causes write accesses to be written to the cache only. This causes the cache line to become modified. Modified cache lines are automatically written back to main memory only when the cache line is no longer needed. Copy-back mode is a more complex mode of cache management, but provides substantial system performance improvements over write-through due to decreased traffic on the system bus.

A 32-byte write buffer and a 32-byte read buffer are provided in the CY7C604/605 to fully buffer the transfer of a cache line. This feature allows the CY7C604/605 to simultaneously read a cache line from main memory as it is flushing a modified cache line from the cache. This feature is also used in write-through

cache mode for write accesses to main memory. The write buffer avoids stalling the CY7C601 on writes to main memory by storing the write data until the physical bus becomes available. The write buffer writes the data to memory as a background task.

The CY7C604 and CY7C605 support the SPARC MBus reference standard interface. The MBus is a peer-level, high-speed, 64-bit, multiplexed address and data bus that supports a full peer-level protocol (i.e., multiple bus masters). The CY7C604/605 MBus supports data transfers in transaction sizes of 1, 2, 4, 8, or 32 bytes. These data transfers are performed in either burst or non-burst mode, depending upon size. Data transactions larger than eight bytes (one doubleword) are transferred in burst mode, which consists of an address phase followed by four data phases. Non-burst transactions consist of an address phase followed by one data phase, and are used for data transactions of eight or less bytes. Bus mastership is granted and controlled by an external bus arbiter. The bus arbiter sets bus priorities, and grants access to a bus master. Additional information on the MBus can be found in *Chapter 11, MBus Operation.*

MBus is divided into two levels of implementation: Level 1 and Level 2. Level 1, implemented on the CY7C604, is the uniprocessor version of MBus. Level 1 is a subset of Level 2, which is the multiprocessor version of MBus. The CY7C605 supports Level 2 MBus. Level 2 MBus includes the IEEE Futurebus cache coherency protocol, which has been recognized in the industry as a superior method of supporting multiprocessing systems. Level 2 MBus defines five cache states for describing cache line status. Transactions on the MBus are monitored or "snooped" by the CY7C605 and other bus agents on the Level 2 MBus to maintain ownership status for each cache line. Transactions on the Level 2 MBus are made with respect to the cache line ownership status to ensure consistency for shared data images.

The Level 2 MBus supports direct data intervention, which allows a cache system with the up-to-date version of a cache line to directly supply the data to another cache system without having to first update main memory. Direct data intervention provides a significant performance improvement over systems which do not support this feature. In addition, the CY7C605 provides support for memory systems with reflective memory controllers. A memory system with reflective memory control can recognize a cache-to-cache data transaction and automatically update itself without delaying the system. Another system concept supported by the CY7C605 is secondary caching. Secondary caching provides a performance advantage over systems directly using main memory, and provides an economic advantage over systems using large caches for each processing node.

8.1 Memory Management Unit

This section describes the SPARC reference MMU implemented on the CY7C604 and CY7C605. This function is identical for both the CY7C604 and CY7C605, and all details of *Sections 8.1* and *8.2* apply to both.

The MMU provides virtual to physical address translation with the use of an on-chip translation lookaside buffer. The TLB is in reality a full address translation cache for address translation entries stored from tables in main memory. These entries, referred to as page table entries or PTEs, contain the mapping information used by the MMU to translate the virtual addresses. Addresses presented to the MMU for translation are compared against the set of PTEs stored in the TLB. All entries in the TLB are simultaneously accessed through the use of advanced content addressable memory (CAM) technology. If a match for the virtual address and context is found in a valid TLB entry and the access protection is not violated, a TLB hit occurs and the address is translated. A virtual address and context that matches a valid TLB entry but violates the memory access protections will cause the CY7C604/605 to generate a memory exception to the CY7C601. If the TLB entries do not match the address and context, or the TLB entry is invalid, then a TLB miss occurs. The MMU responds to the TLB miss by initiating a table walk to find the correct PTE stored in main memory for the virtual address.

The MMU uses a tree-structured table walk algorithm to find page table entries not found in the TLB. The table walk is a search through a series of four tables in main memory for the PTE corresponding to a virtual

address. These tables are: the context table, the level 1 table, the level 2 table, and the level 3 table. The table walk uses the context table pointer register as a base register and the context number as a offset to point to an entry in the context table. At any address, the MMU finds either a PTE, which terminates its search, or a page table pointer (PTP). A PTP is a pointer used in conjunction with a field in the virtual address to select an entry in the next level of tables. The table walk continues searching through levels of tables as long as PTPs are found pointing to the next table. The table walk terminates when a PTE is found, or an exception is generated if a PTE is not found after accessing the level 3 table. An exception is also generated if the table walk finds an invalid or reserved entry in the page tables. Upon finding the PTE, the CY7C604/605 stores it in an available TLB entry and translates the corresponding virtual address. The table walk processing is implemented in the CY7C604/605 hardware. It is self-initiated, and is transparent to the user.

Virtual Section (CAM Array)		Physical Section (RAM Array)						
VA(31:12)	CXN(11:0)	PPN(35:12)	C	M	ACC(2:0)	ST(1:0)	V	
								64 TLB entries
VA(31:12)	CXN(11:0)	PPN(35:12)	C	M	ACC(2:0)	ST(1:0)	V	

Figure 8–2. Translation Lookaside Buffer (TLB)

8.1.1 Translation Lookaside Buffer (TLB)

The CY7C604/605 uses a 64-entry fully associative TLB for address translation. The TLB consists of two sections: a virtual section and a physical section, as shown in *Figure 8–2*. The virtual section is compared against the virtual address and the contents of the context register. A content addressable memory (CAM) is used as the virtual section of the TLB. The CAM provides simultaneous comparison of all 64 TLB entries with the current virtual address and context. The physical section of the TLB is a RAM array, and its entries are addressed by a valid compare output from a CAM entry. If a CAM entry matches the virtual address and context, the corresponding RAM entry in the TLB provides the physical address for use by the CY7C604/605.

The virtual section of a TLB entry consists of 20 bits of virtual address (VA(31:12)) and a 12-bit context number (CXN(11:0)). The physical section of a TLB entry consists of a 24-bit physical page number (PPN(35:12)), a cacheable bit (C), a modified bit (M), a three-bit field for page access-level protection (ACC(2:0)), a two-bit short translation field (ST(1:0)), and one valid bit (V).

As described by the SPARC reference MMU specification, bits 31 through 12 of the virtual address are translated to an expanded physical address using bits 35 through 12. The translation of these bits depends upon the ST field of the TLB entry (or PTE) and the MMU operation mode (refer to page 8-15). Bits 11 through 0 of the virtual address are not translated, and are defined as the page offset for the 4-Kbyte memory page.

A TLB entry (PTE) can be defined to map a virtual address into one of four sizes of addressing regions using the ST field. The four sizes of addressing regions are: 4-Kbyte, 256-Kbyte, 16-Mbyte, or 4-Gbyte. *Table 8-1* illustrates the values assigned to the ST(1:0) field.

The value of the short translation bits affects both the addresses generated using the TLB entry and the virtual addresses allowed to match with the TLB entry. The virtual address supplied by the integer unit is divided

into four fields: index 1, index 2, index 3, and page offset, as illustrated in *Figure 8–3*. For ST = (1,1) (4-Gbyte addressing range), only the context register is used to match a TLB entry. Setting ST = (1,1) essentially causes the CAM array to ignore the index 1, 2, and 3 fields of the virtual address. Consequently, the address generated using the TLB entry only supplies the upper four bits of the 36-bit physical address. Index 1, 2, and 3 fields, along with the page offset, are passed along to the physical address unchanged.

The three remaining values of the ST field "turn on" comparison of the three index fields. The index fields that are required to match a TLB entry also become the fields that are replaced by the TLB entry during virtual to physical translation. Setting ST = (1,0), (16-Mbyte addressing region), requires the TLB to match the context and index 1 fields of the virtual address to the TLB entry. The TLB entry with ST = (1,0) will supply the upper four address bits and replace the index 1 field of the virtual address with a physical address field. The index 2, 3, and page offset fields are passed along to the physical address from the virtual address. Setting ST = (0,1) and (0,0) adds index 2 and index 3 fields to the comparison, respectively. Setting ST = (0,0) causes the TLB to require matching of the context, index 1, 2, and 3, and will replace all but the page offset when translating the virtual address.

Physical addresses are generated using the contents of the PPN field of the TLB entry. The portion of the PPN field used to map the virtual address to a physical address is dependent upon the ST(1:0) bit field, as described above. If a 4-Kbyte linear addressing range is specified by the ST(1:0) bits, then the entire 24 bit field is used as the upper 24 bits of the physical address. When a 256-Kbyte linear addressing range is specified, the upper 18 bits of the PPN(35:18) field are used in the physical address. The remaining bits of the physical address are supplied from the virtual address. The upper 12 bits of the PPN(35:24) field are used for a 16-Mbyte addressing region. If a 4-Gbyte region is selected, only the upper four bits of the PPN(35:32) field are used in the address translation. The page offset field of the virtual address is always used as the lower 12 bits of the physical address.

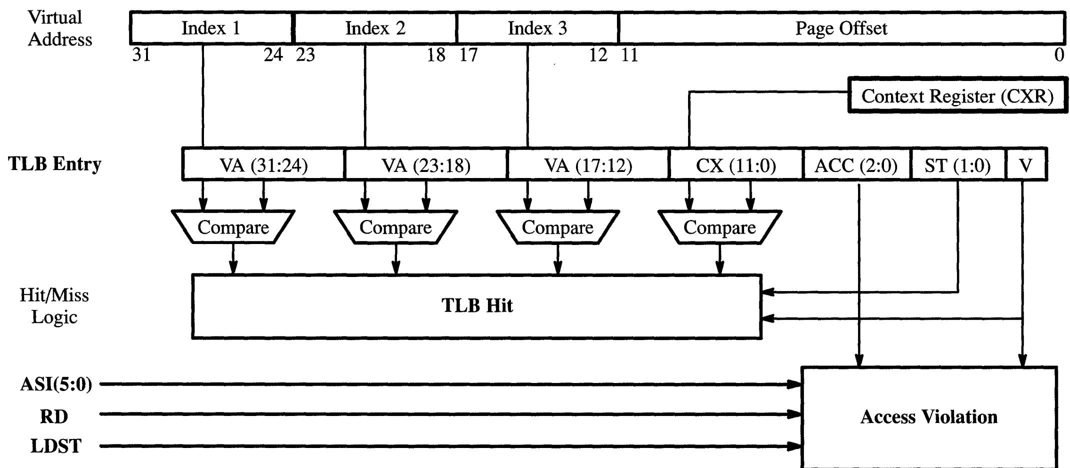


Figure 8–3. Address Comparison

Table 8-1. Short Translation Bits - ST(1:0)

ST1	ST0	Address Mapping
0	0	4-Kbyte (page size)
0	1	256-Kbyte
1	0	16-Mbyte
1	1	4-Gbyte

The cacheable bit (C) indicates whether the memory addressed by the TLB entry is cacheable or not. If the MMU is enabled, the value of the C bit is output on the MC pin (MAD(43)) of the MBus during the address phase of a transaction. The MBus is described in *Chapter 11*.

The modified bit (M) in the TLB is set when the CY7C601 modifies the memory page. This bit may be checked by an operating system to determine the modified status of a memory area.

The access-level protection (ACC) bits are described in *Table 8-2*. The ACC bits define the access-level protection for the addressing region controlled by the TLB entry. Access-level protection is checked during a TLB access. If a TLB hit occurs but access-level protection is violated, the MMU generates a synchronous fault and the operation terminates (see *Section 8.9, Synchronous Faults*).

The valid bit (V) reports the valid status of the TLB entry. These bits are cleared upon power-on reset ($\overline{POR}$) to invalidate the TLB entries. These bits are also cleared for a TLB entry flush.

Programmer's Note: When loading the TLB entries under software control (i.e., TLB entries loaded by the integer unit with ASI = 6), care must be taken to ensure that multiple TLB entries cannot map to the same virtual address. This may inadvertently occur when combining TLB entries that map different sizes of addressing regions. For example, a 4-Kbyte region described by a TLB entry could be included in a TLB entry for a 16-Mbyte region. Violation of this restriction will result in an invalid output from the TLB. Note that this case cannot happen when the TLB entries are automatically loaded by the CY7C604/605 during a table walk, as the TLB is checked for a "hit" first.

Table 8-2. Access-Level Protection Bits—ACC(2:0)

ACC	User Access	Supervisor Access
0	Read Only	Read Only
1	Read / Write	Read / Write
2	Read / Execute	Read / Execute
3	Read / Write / Execute	Read / Write / Execute
4	Execute Only	Execute Only
5	Read Only	Read / Write
6	No Access	Read / Execute
7	No Access	Read / Write / Execute

8.1.1.1 TLB Look-up

A virtual address to be translated by the CY7C604/605 is compared against each entry in the TLB as shown in *Figure 8–3*. If a TLB hit (match) occurs and access-level requirements are satisfied, then the TLB outputs the physical address and the cacheable bit. This physical address is output by the CY7C604/605 onto the MBus (refer to *Chapter 11, MBus Operation*) if the cache has been disabled or if the page is non-cacheable.

If the cache controller is enabled and a cache miss occurs, the physical address of the cache miss is used to access the new cache line in main memory for cache line replacement.

The short translation bits specify a linear address mapping range of 4-Kbytes, 256-Kbytes, 16-Mbytes, or 4-Gbytes for each TLB entry. The short translation bits also determine the index fields of the virtual address that are matched with the TLB entry to determine a TLB hit. For a TLB entry with a linear address range of 4 Kbytes, index fields 1, 2, and 3 of the virtual address and the context register are compared against the TLB entry. A TLB entry with a 256-Kbyte linear addressing range requires a match of the context and of the index 1 and index 2 fields. A 16-Mbyte linear addressing range requires a match of the index 1 field and the context. The 4-Gbyte linear address mapping requires only a context match to produce a TLB hit.

If the modified bit is not set in a TLB entry, write or Load/Store accesses that match the TLB entry and meet all access-level requirements will cause a table walk. (see *Section 8.1.2., Table Walk*) If the modified (M) bit is not set for a write access, then the table walk sets the modified bit in the page table pointer entry for the memory region. This information is used by an operating system to ensure that modified regions of memory are stored in alternate memory media (typically a disk drive) before they are overwritten during memory page swap operations.

If there is a matched entry, but the access-level requirements are not satisfied, then a synchronous address fault exception is asserted. Context number matching is not required if the access-level field (ACC) is either 6 or 7 and the memory access is a supervisor mode access (ASI = 9,B H). This produces a means of mapping the kernel of an operating system into the same virtual address locations of every context.

The TLB ignores access-level checking during MMU probe operations, copy-back flush cycles, and alias detection cycles.

8.1.1.2 TLB Entry Replacement and Locking

The CY7C604/605 supports a random replacement algorithm to replace a TLB entry during TLB miss processing. The random replacement is implemented by using a counter to point to one of the 64 TLB entries. A 6-bit replacement counter (RC) is incremented by one during each clock cycle to point to one of the TLB entries as shown in *Figure 8–4*. Upon encountering a TLB miss, the CY7C604/605 uses the counter value to address a TLB entry to be replaced. The hardware automatically replaces an entry pointed to by the replacement counter (RC) during TLB miss processing.

Locking of TLB entries is supported with a 6-bit initial replacement counter (IRC). The number of locked entries is specified by setting the value of the IRC. The value of the IRC is used as a counter preset for the replacement counter. Once the replacement counter (RC) reaches the maximum value, it wraps to the initial replacement counter (IRC) value. Upon power-on reset ($\overline{\text{POR}}$), both the IRC and RC are initialized to zero.

Locked TLB entries can be changed (read/write) only through the alternate space Load/Store instructions with ASI = 6 (see Diagnostics Support, page 8-52.) These locked entries will not participate in the random replacement algorithm during TLB miss processing. The IRC should be initialized to the number of lockable entries by writing to the TLB replacement control register (TRCR).

Programming Note: When changing the IRC, the RC should also be written with the same value. This ensures that the RC is always pointing to the replacement area of the TLB.

8.1.1.3 TLB Entries (TLBEs)

Both the virtual and physical sections of each TLB entry can be accessed (read/write) through single load or store instructions. Software has the option to write and to lock high-usage or high-priority TLB entries to optimize system response time (Refer to MMU TLB Entries, page 8-52, for more details.)

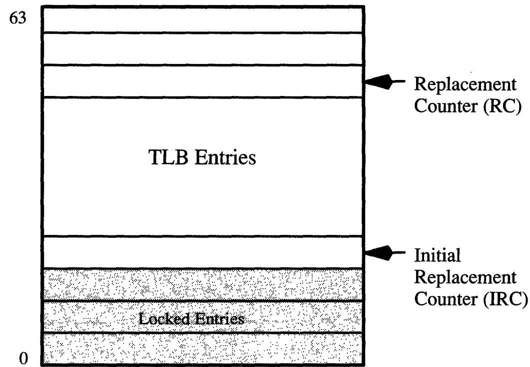


Figure 8–4. TLB Replacement and Locking

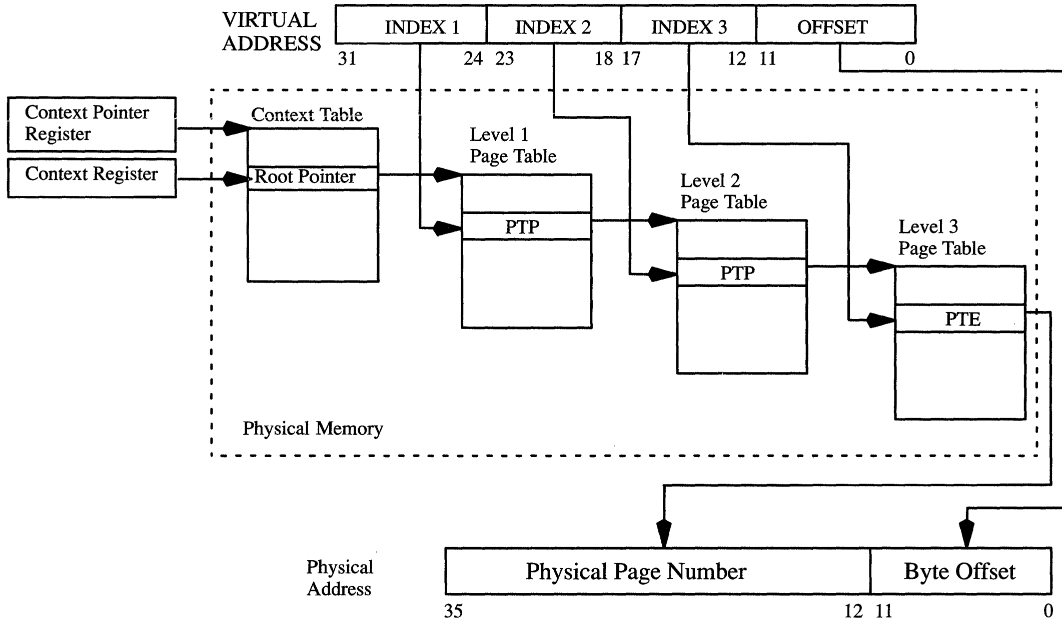


Figure 8–5. Four-Level Table Walk (4-Kbyte Addressing)

8.1.2 Table Walk

The CY7C604/605 supports tree-structured, 4-level table walk processing (including the context table level) as shown in *Figure 8–5*. All of the virtual to physical address mapping tables are located in physical memory. These tables are accessed in the case of a TLB miss or of a write or Load/Store operation with a cleared (Modified) bit in the TLB entry.

Upon starting a table walk, the CY7C604/605 walks through a series of tables to find a page table entry (PTE). The page table entry contains the physical page number, the access-level permission, cacheable, modified, and referenced bits for the address generating the table walk. (Refer to page 8-11 for information on PTEs.) A table walk caused by a TLB miss causes the CY7C604/605 to update an available TLB entry with the new PTE. A table walk forced by a write or Load/Store operation on an unmodified memory region causes the CY7C604/605 to set the modified bit in the page table entry and in the TLB entry.

The table walk begins with an access to the context table. The CY7C604/605 uses the context table pointer register (CTPR) as a base register to point to the beginning of the context table. The context register (CXR) is used as an index register to point to the table entry. The upper 22 bits of the CTPR are concatenated with the 12 bits of the CXR to provide a 36-bit address. The lowest two bits of all addresses pointing to a page table entry or pointer are always forced to zero.

If a page table entry (PTE) is found at the context table level, the table walk terminates. The PTE is stored in the TLB and, if necessary, the modified bits and/or the reference bits are updated. If a page table entry is not found, then a Page Table Pointer (PTP) must be located at the address pointed to in the context table. (See page 8-10 for more information on PTPs and PTEs.) The page table pointer is used as the base address for the next table.

If a PTE is not found, the table walk continues by accessing the level 1 table using the PTP as a base address and the index 1 field from the virtual address as an index pointer. It is possible to find a PTE instead of a page table pointer at any level during the table walk. The index 1 field (virtual address (31:24)) is used to select an entry in the level 1 table. If a page table entry is not found at this location, a page table pointer stored at this entry is used as the base address for the level 2 table. The index 2 field (virtual address (23:18)) is used to select an entry in the level 2 table. The entry in the level 2 table, if not a page table entry, is used as the base address for the level 3 table. The index 3 field (virtual address (17:12)) is used to select an entry in the level 3 table, which must be a page table entry.

If a page table entry is not found after the level 3 table access, a synchronous fault exception is asserted. A synchronous fault exception is also generated if an invalid entry is found at any level of the table walk. The table walk terminates immediately when an exception is generated.

The level at which the table walk terminates is related to the size of addressing region associated with the entry. A table walk that finds its page table entry in the context table corresponds to an addressing region of 4-Gbyte. Each level deeper into the table walk corresponds to a smaller size of address mapping. A PTE for a 16-Mbyte addressing region will be found in a level 1 table. A 256-Kbyte PTE will be found in a level 2 table. Only an addressing region of 4 Kbytes will require a table walk of four levels to find the correct page table entry.

An example of a table walk for a 256-Kbyte linear address space is shown in *Figure 8–6*. The value of the short translation bits are related to the level at which the table walk terminates. The short translation bits decrease from (1,1) for a table walk with a context table PTE to (0,0) for a table walk with a level 3 table PTE. (Refer to *Table 8-1.)*

Each table walk access is performed as a non-burst transaction on the MBus (physical bus). The MBus busy ($\overline{MBB}$) signal is asserted from the beginning of the table walk to the end of the table walk process. This locks the MBus and prevents another bus master from gaining the bus until the table walk is complete. The MLOCK bit in the address phase of the MBus transaction will be set (refer to *Chapter 11, MBus Operation*),

indicating a locked transaction. During these transactions, the C bit in the SCR register is output on the MC signal of the MBus. There will be Write transactions during the table walk only if the reference bit (R) and/or the modified bit (M) has to be set in the page tables.

If there is an invalid page table entry (ET = 0) at any level, an invalid address error exception occurs and the table walk terminates immediately. If an external bus error occurs, a reserved entry (ET = 3) is detected, or a PTP entry is detected in level 3, a translation error exception occurs, and the table walk terminates immediately. If an access-level protection occurs, the table walk is terminated and a protection/privilege violation exception is asserted.

The reference bit (R) and the modified bit (M) are set according to the access type. In order to record the exceptions in the synchronous fault status registers properly, the table walk hardware must indicate the fault type and the level at which the fault occurred (Refer to *Section 8.9* for more details). For access-level checking during the table walk, Load/Store cycles are treated as write cycles. The table walk state diagram is shown in *Figure 8–10.*

During MMU probe operations, copy-back flush cycles, and alias detection cycles, the table walk controller ignores access-level checking.

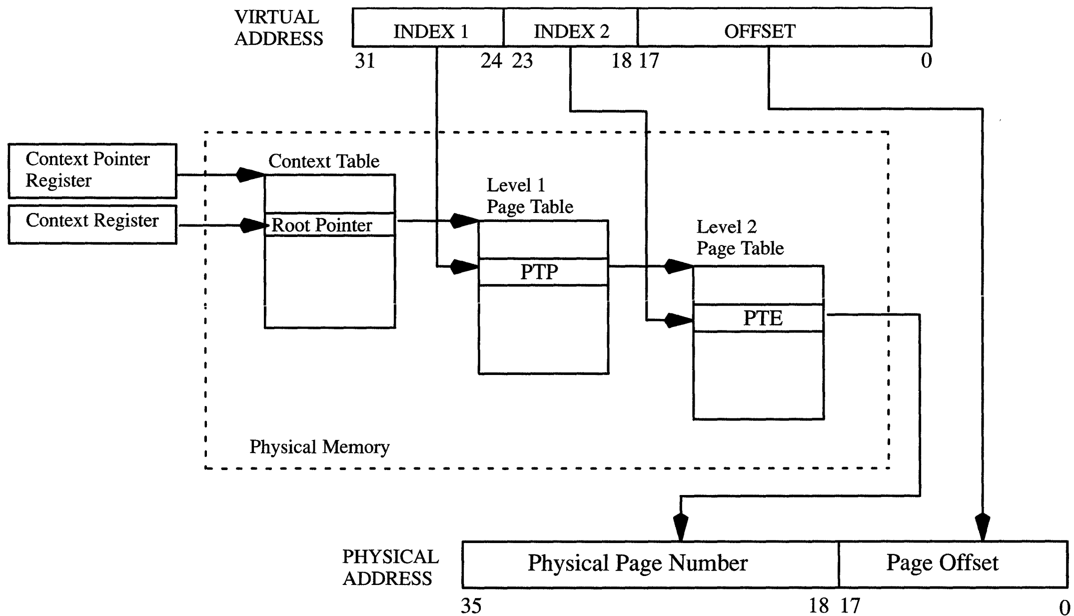


Figure 8–6. Three-Level Table Walk (256-Kbyte Addressing)

8.1.3 Page Table Pointer (PTP)

A page table pointer (PTP), as shown in *Figure 8–7*, may be found in the context, level 1, or level 2 tables. The PTP is used in conjunction with an index field of the virtual address to point to the next level of table in a table walk. The PTP found at the context level is called the root pointer. Bits 31 through 6 of the root pointer are output on bits 35 through 10 of the MBus (MAD<35:10>) and are concatenated with the eight bits of the index 1 field of the virtual address to access the entry in the first level page table. (Refer to *Figure 8–6.*) The lowest two bits of the address are equal to zero, as addressing is aligned on word boundaries.

Similarly, bits 31 through 4 of the PTP in level 1 or level 2 tables are output on bits 35 through 8 of the MBus (MAD<35:8>). The index 2 or index 3 fields are concatenated with the PTP to yield the address of the next table entry. The ET field (*see Table 8-3*) describes the entry type: invalid, page table pointer, or page table entry.

In order to reduce the penalty for a TLB miss, the root pointer from the context level table and two PTPs from the level 2 table are cached in the PTP cache. The PTPs from the most recent data and instruction misses using a four-level table walk are cached for later use. The TLB checks the PTP cache upon a TLB miss, and uses the cached PTP to access the level 3 table if an entry matches the access. The PTP cache is discussed in more detail in *Section 8.1.5*.

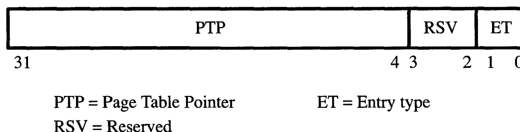


Figure 8–7. Page Table Pointer

Table 8-3. Page Table Entry Type

ET	Entry Type
0	Invalid
1	Page Table Pointer
2	Page Table Entry
3	Reserved

8.1.4 Page Table Entry (PTE)

The page table entry (PTE) is shown in *Figure 8–8* and may be found in the context, level 1, level 2 or level 3 tables. The page table entry contains the address mapping information used by the MMU to translate a range of virtual addresses to physical addresses.

The level of the table in which the PTE is found is related to the addressing range associated with the PTE. A PTE found in the context table will map a 4-Gbyte addressing region. A level 1 PTE will map a 16-Mbyte addressing region. A level 2 PTE corresponds to a mapping region of 256 Kbytes. A level 3 PTE maps a 4-Kbyte addressing region.

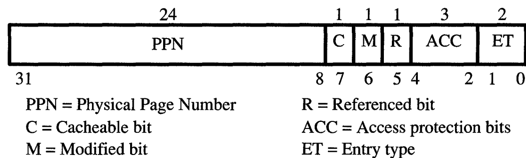
The addressing region mapped to the PTE determines how many bits in the PPN field of the PTE are used to form the physical address. PTE(31:28) from a context level table PTE are output on bits 35 through 32 of the physical address bus (MAD<35:32>) to offer 4-Gbytes of linear address mapping. Similarly, PTE(31:20) from a level 1 table PTE are asserted on bits 35 through 24, and provides 16 Mbytes of linear addressing. PTE(31:14) from a level 2 table PTE are asserted on bits 35 through 18, and PTE(31:8) from a level 3 table PTE are asserted on bits 35 through 12 to offer 256 Kbytes and 4 Kbytes of linear address mapping, respectively. The remainder of the PPN field not used for address translation is reserved. The remaining physical address bits not specified by the PPN field are supplied from the virtual address.

The ACC bits describe the access-level and privilege protection assigned to the PTE. These bits are described in *Table 8-2*. The referenced (R) bit is set in the PTE when the CY7C604/605 has read the value of the PTE in a table walk. The CY7C604/605 automatically sets this bit upon access of the PTE. The modi-

fied (M) bit is set upon a write or Load/Store access of a previously unmodified memory region. This information is commonly used by an operating system to flag regions of memory that must be written to mass storage before being replaced by another memory page.

The cacheable (C) bit indicates whether or not the memory region addressed by the PTE is allowed to be cached. This bit may be used to prevent shared memory pages from being cached, thereby avoiding potential aliasing problems. It also may be used to prevent caching of memory mapped input/output devices.

The ET field, illustrated in *Table 8-3*, is used by the CY7C604/605 to determine the type of table entry during a table walk. The ET field is set to 2 to indicate a PTE, and is set to 1 to indicate a PTP. If the CY7C604/605 encounters a table entry with ET = 0 during a table walk, the CY7C604/605 generates an invalid address error. The CY7C604/605 generates a translation error if ET = 3 (reserved) is encountered in a table entry during a table walk.



PPN = Physical Page Number
C = Cacheable bit
M = Modified bit
R = Referenced bit
ACC = Access protection bits
ET = Entry type

Figure 8–8. Page Table Entry Format

8.1.5 Page Table Pointer Cache (PTPC)

In order to reduce the penalty for a TLB miss, the CY7C604/605 supports a three-PTP entry page table pointer cache. The page table pointer cache (PTPC) caches the most recently used PTPs, as shown in *Figure 8–9*. The three entries are: the root pointer register (RPR), the instruction access level 2 PTP (IPTP), and the data access level 2 PTP (DPTP). The IPTP and DPTP registers are referenced by a fourth register, the index tag register (ITR). These entries are cached during table walk processing for a TLB miss.

The root pointer for a context is cached in the RPR. The RPR remains valid until the context register (CXR) or the context table pointer register (CTPR) value is changed. The instruction access PTP register contains the latest level 2 PTP for an instruction access. This PTP is cached from the last TLB miss requiring a four-level table walk for an instruction access. The data access PTP register contains the latest level 2 PTP for a data access. This PTP is also cached from the last four-level table walk for a data access. The IPTP and DPTP registers are invalidated when another table walk that accesses level 3 of the page tables is forced for an instruction or data access or a TLB flush. They also are invalidated when either the context register or context pointer register is changed. Refer to page 8-45 for more information on these registers.

Figure 8–9 illustrates the PTPC. The index tag register (ITR) is used to reference the IPTP and DPTP registers. The ITAG and DTAG fields of the index tag register are used by the CY7C604/605 to compare against an address generating a TLB miss. Once a level 2 page table pointer is cached for an instruction or a data access, the same PTP is used if the index 1 and index 2 fields of the virtual address match the index 1 and index 2 tag fields of the ITAG or DTAG. The IPTP and DPTP registers are updated only if a TLB miss occurs that does not match the ITAG or DTAG and also generates a table walk that accesses level 3 of the page tables.

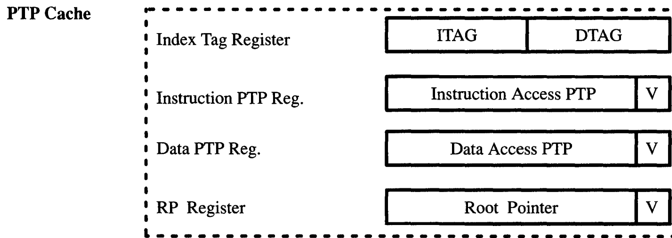


Figure 8–9. Page Table Pointer Cache

Once a root pointer is cached for a particular context, the same root pointer can be used as long as the context is not changed. If the table walk finds a context level or level 1 or level 2 entry PTE (i.e., is not a four-level table walk), then no caching of level 2 pointers is performed.

Whenever the context is changed, the entire PTPC (all three entries) is invalidated. Upon power-on reset, all the PTPC entries are invalidated. When the context pointer register (CTPR) is written, the page table pointer cache is invalidated by clearing the V bits in the IPTP, DPTP, and RPR registers. Any TLB flush invalidates the IPTP and DPTP registers of the PTP Cache.

The IPTP and DPTP registers are not updated during table walks caused by address alias detection and copy-back flush cycles.

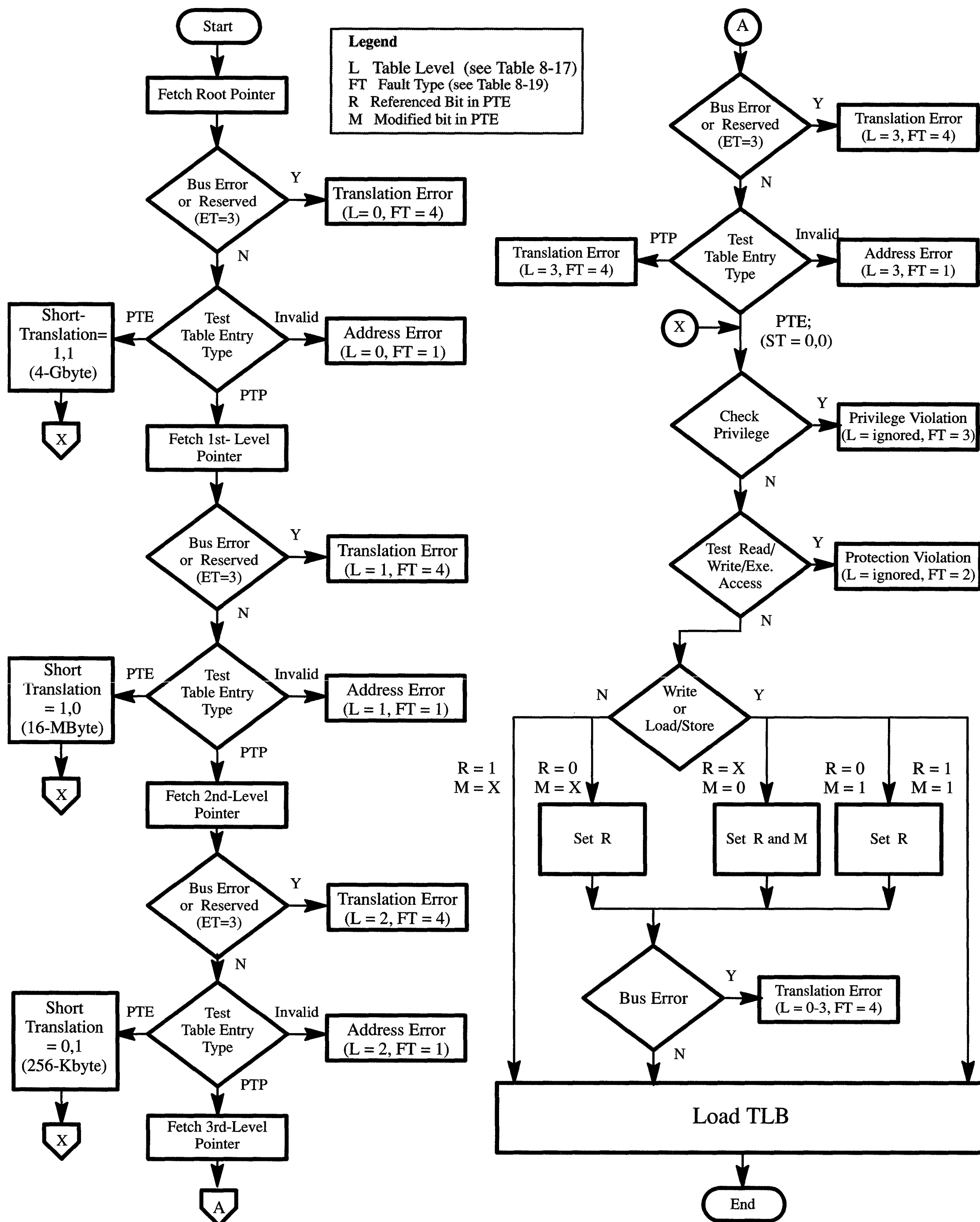


Figure 8–10. Table Walk Algorithm

8.2 MMU Operation Modes

This section describes the different modes of operation of the CY7C604/605, the conditions under which they occur, and what information is reflected on the pins. The operation mode for the MMU (and cache controller) is controlled by the system control register (SCR). Please refer to *Sections 8.4.1* and *8.4.2* for further information on the SCR.

The following symbols are used throughout the chart:

Symbol	Meaning
MC(MAD<43>)	MBus Cacheable indicator signal (Refer to Pin Definitions, *Section 8.10*)
MBL(MAD<45>)	MBus Boot/Local indicator signal (Refer to Pin Definitions, *Section 8.10*)
ASI	Address Space Identifier code for current access from CY7C601
SCR[C]	Cacheable bit of SCR
X	Not Defined or Don't Care
UN	Unassigned ASI
RES	Reserved ASI and ASI defined but not implemented (see *Table 8-15*)
PA	Physical Address
VA	Virtual Address
BM, ME, CE	Bits in System Control Register (SCR)
PTE[C]	Cacheable bit of page table entry

Table 8-4. MMU Operation Modes

MMU Operation Modes									
Mode	**Conditions**				**Results**				
	ASI	**BM**	**ME**	**CE**	**Physical Addressing**		**Caching**	**MC**	**MBL**
Local	1	X	X	X	PA<35:32> = 0	PA<31:0>= VA<31:0>	Not Cached	0	1
UN, RES	UN, RES	X	X	X	Ignore	Ignore	Ignore	N/A	N/A
By-pass	20-2F	X	X	X	PA<35:32> = ASI<3:0>	PA<31:0> = VA<31:0>	Not Cached	0	0
Pass-Through	8,9,A,B	0	0	X	PA<35:32> = 0	PA<31:0> = VA<31:0>	Not Cached	SCR [C]	0
Boot (Instr. access)	8,9	1	X	X	PA<35:28>= FF	PA<27:0> = VA<27:0>	Not Cached	SCR [C]	1
Boot (Data access)	A,B	1	0	X	PA<35:32> = 0	PA<31:0> = VA<31:0>	Not Cached	SCR [C]	1
Translation 1 (Data Access and Cache Disabled)	A,B	X	1	0	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Not Cached	PTE [C]	0
Translation 2 (Data Access and Cache Enabled)	A,B	X	1	1	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Cached if PTE[C] =1	PTE [C]	0
Translation 3 (Instruction Access and Cache Disabled)	8,9	0	1	0	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Not Cached	PTE [C]	0
Translation 4 (Instruction Access and Cache Enabled)	8,9	0	1	1	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Cached if PTE[C] =1	PTE [C]	0

* Concatenation field sizes vary depending upon the short translation (ST) bits to provide 4G, 16M, 256K, 4 Kbytes of linear address mapping. Refer to *Section 8.1.1* for further details.

The MMU provides three types of operating modes: boot modes, direct-access modes, and translation modes. Two boot modes are defined for the MMU, one for data accesses, and one for instruction accesses. The boot modes force the upper eight bits of the physical address to FF H for instruction accesses. The upper four bits are forced to zero for data accesses. These two modes also assert the MBus boot mode/local indicator (MBL) signal. This signal can be used in the system to enable a memory region used only for system boot and configuration. This allows the system a secure method of accessing bootstrap ROM and shadow RAM separate from the main memory space.

The direct access modes allow the integer unit to access the main memory without address translation by the MMU. These modes include: local, by-pass, and pass-through. Local mode enables the MBL signal and forces the upper four bits of the physical address to zero. The lower 32 bits of the physical address are supplied directly from the virtual address bus. This mode allows the integer unit to access the boot mode memory (if supported in the system) without changing the state of the system control register (SCR). Local mode is enabled by using a load or store alternate instruction with ASI = 1 H.†

Bypass mode allows complete access to the main memory space. MBL is not enabled, and the lower four bits of the ASI are used as the upper bits of the physical address. The remaining 32 bits are supplied directly from the virtual address bus. The state of the SCR does not have to be modified. This mode is mapped into the ASI space as ASI = 20 - 2F H.

Pass-through mode describes the CY7C604/605 operation with the MMU disabled. The upper four address bits of the physical address are forced to zero. The MBL signal is not asserted. This mode does not require non standard ASI assignments (i.e., ASI = 8,9,A,B H), but the boot mode (BM) and MMU enable (ME) bits of the SCR must be cleared.

The translation modes are considered to be the normal operating modes of the MMU. This group includes four modes of translation operations: Translation 1–4. Translation 1 and 2 are the non-cached and cached data access modes. Translation 3 and 4 are the non-cached and cached instruction access modes. The cached and non-cached modes are identical in results for both data and instruction accesses, with the exception that the data access modes ignore the boot mode (BM) bit of the SCR. This feature allows the system to enable the MMU for data accesses, yet still access instructions from the boot memory space without changing the BM bit.

8.2.1 MMU Flush and Probe Operations

8.2.1.1 Flush Operations

The flush operation allows software invalidation of selected entries in the TLB. TLB entries are flushed by executing a store alternate ASI instruction using ASI = 3 H and supplying a virtual address in the format shown in *Figure 8–11*. The context number is given by the context register (CXR). All TLB entries that match the virtual address, context, and TLB flush type will be flushed (invalidated) simultaneously. The flush type is specified in bits 11–8 of the virtual address for the flush operation.

The CY7C604/605 supports five different types of TLB flushing operations. These types are: page, segment, region, context, and entire flush. The five types of flushing are listed in *Table 8-5*, and define the address comparison required to match a TLB entry for flushing. The short translation (ST) bits in the TLB entries are ignored for TLB matching. All TLB entries matching the compare criterion of the flush type are invalidated, including those locked by the IRC.

† The SPARC architecture reference supports the concept of Address Space Identifiers (ASI), which provide an extension of the standard addressing space. These bits are used to enable special addressing modes, or to provide access to registers and other features of the CY7C604. Refer to *Section 8.8* for more information.

Virtual Address Format:

INDEX1	INDEX2	INDEX3	TYPE	RSV
31 24	23 18	17 12	11 8	7 0

Figure 8–11. MMU Flush Address Format

Table 8-5. TLB Entry Flushing

Type	Flush	Compare Criterion
0	Page	Context (or ACC = 6, 7), Index 1, Index 2, and Index 3
1	Segment	Context (or ACC = 6, 7), Index 1, and Index 2
2	Region	Context (or ACC = 6, 7), and Index 1
3	Context	Context (user pages with ACC = 0 to 5)
4	Entire	None
5 to F	Reserved	

8.2.1.2 Probe Operation

The probe operation allows testing the TLB and page tables for a PTE entry corresponding to a virtual address. The operation is initiated by executing a load alternate ASI instruction with ASI = 3 H, the appropriate virtual address, and the context number. The context is specified by the context register. Upon starting a probe operation, the TLB is probed first. If there is a TLB hit, it returns the 32-bit physical section of the matched entry. The returned entry fields are formatted such that it is identical to a PTE (see *Section 8.1.4* on page 8-11, for PTE format information). If a matching entry could not be found in the TLB, a table walk is started and an appropriate 32-bit value (PTE) is returned and loaded into the TLB.

A probe operation causes the reference bit (R) to be set in the PTE by means of a table walk. When a probe operation hits the TLB, the R bit is always returned as set.

The context register and access-level protection checking are ignored for TLB matching and during the probe operation table walk. The table walk hardware checks for invalid address error and translation error exceptions and records appropriate fields in the SFSR register as in the normal table walk process. If a bus error occurs or an invalid or reserved entry is detected during the table walk, a 32-bit zero value is returned as status. If a zero value is returned, the UC, TO, BE, L, and FT fields of the SFSR are updated accordingly, but the operation does not cause an exception to the CY7C601.

8.3 CY7C604 / CY7C605 Cache Controllers

The differences between the CY7C604 and CY7C605 become evident in the features of their respective cache controllers. The CY7C604 cache controller is designed for a uniprocessor system, and provides cache locking for real-time system support. The CY7C605 cache controller is enhanced to accommodate the requirements of a multiprocessing system. The CY7C605 provides bus snooping and a Futurebus style of cache coherency protocol. The CY7C605 is designed to provide high visibility into its cache operations from the perspective of the shared physical bus in order to simplify support by a secondary cache system. The following sections discuss the CY7C604 and CY7C605 cache controllers. Sections specific to the

CY7C604 or CY7C605 are marked with that part number only. Sections applying to both the CY7C604 and the CY7C605 are marked "CY7C604/605."

8.3.1 CY7C604/605 Cache Modes

The CY7C604/605 virtual cache can be programmed for either write-through with no write allocate or copy-back with write allocate. The two cache modes differ in how they treat cache write accesses. Write-through cache mode causes write hits to the cache to be written to both cache and main memory. Write-through write cache misses only update main memory; they do not modify the cache. For the CY7C604, write-through write cache misses also invalidate the cache tag, but the CY7C605 only invalidates the cache tag if an alias is detected.

A write access in copy-back mode only modifies the cache. The writing of the modified cache line to main memory is deferred until the cache line is no longer required. Copy-back cache mode has the advantage of reducing traffic on the system bus. Bus traffic is reduced since all updates to memory are deferred and are performed subsequently only as absolutely required. In addition, all such data transfers are made utilizing the more efficient burst mode. The following describes the two cache modes in detail.

8.3.1.1 CY7C604/605 Write-Through Mode with No Write Allocate

For write-through cache mode, write access cache hits cause both the cache and main memory to be updated simultaneously. A write access cache miss causes only main memory to be updated (no write allocate). The selected cache line is invalidated for a write access cache miss. Write-through caching mode normally requires a processor to delay during a write miss while the data is written to main memory. The CY7C604/605 provides write buffers to prevent this delay in most cases. The write buffers store the write access and write the data to main memory as a background task. (Refer to page 8-37 for further information on the write buffers.)

During read access cache hits, the cached data is read out and supplied to the CY7C601. In the case of a read access cache miss, a cache line is fetched from main memory to load into the cache and the required data is supplied to the CY7C601.

8.3.1.2 CY7C604/605 Copy-Back Mode with Write Allocate

When the cache is configured for copy-back mode, only the cache is updated on write access cache hits (i.e., main memory is not updated). The modified bit of the cache tag for the cache line is set on a copy-back write access (write hit or after a write miss is corrected). During write access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache and only the cache is updated. If the selected cache line is modified, the selected cache line is flushed out to update main memory. The CY7C604/605 simultaneously fetches the new cache line from main memory and stores it into the read buffer as it flushes the modified cache line from the cache and stores it into its write buffer. After the modified cache line has been flushed, the CY7C604/605 writes the modified cache line out of its write buffer into main memory while the new cache line is stored into the cache memory from the read buffer.

During read access cache hits, the cached data is read out and supplied to the CY7C601. During read access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache. If the selected cache line is modified, the selected cache line is flushed out to the CY7C604/605 write buffer, and a new cache line is fetched from main memory and stored into the read buffer. The new cache line is then stored in the cache from the read buffer, while the modified cache line stored in the write buffer is written out to main memory.

8.3.2 CY7C604 Cache Controller

The cache controller provides cache memory access control for a 64-Kbyte direct mapped virtual cache. The cache controller is designed to use two CY7C157 Cache Storage Units for the cache memory. These cache

RAMs are 16-Kbyte x 16 SRAMs with on-chip address and data latches and timing control. The CY7C601 cache can be expanded to a maximum of 256 Kbytes by adding additional groups of one CY7C604 and two CY7C157s. Using multiple CY7C604s to expand the cache is referred to as a multichip configuration for the CY7C604, and is described in *Section 8.5, Multichip Configuration.*

The cache is organized as 2048 cache lines of 32 bytes each. The CY7C604 has 2048 cache tag entries on-chip, one tag entry for each cache line. Addressing for the virtual cache is provided directly from the virtual address bus. The virtual address field VA<15:5> selects one of the 2048 lines of the cache. This address field also selects one of the corresponding cache tag entries in the CY7C604. A cache hit occurs when the upper sixteen bits of the virtual address and the context register match with the virtual address and context stored in the selected cache tag entry. The lowest five bits of the virtual address bus (VA<4:0>) select one or more of the 32 bytes in the cache line. Cache data replacement is always performed by replacing cache lines.

The cache is designed to provide data with every read access asserted on the virtual bus, regardless of the cache controller. The CY7C604 controls cache read access by holding the CY7C601 with $\overline{\text{MHOLD}}$ if a cache hit is not detected by the cache controller. The cache controller then reads the new cache line from main memory, and supplies the correct data to the CY7C601. After the correct data is latched into the CY7C601 by strobing the $\overline{\text{MDS}}$ signal, the CY7C601 is released and execution proceeds normally.

Writes to the cache are controlled by the CY7C604, which decodes the lowest two bits of the virtual address, the SIZE<1:0> signal, and checks for a cache hit to enable the correct cache byte write enable signals. If a cache write hit occurs, the CY7C604 decodes the correct $\overline{\text{CBWE}}$ signals for the write access, and outputs these to the CY7C157 Cache Storage Unit write enables. If the cache mode is set to write-through (see *Section 8.3.1, Cache Modes*), the write data is also written to main memory. If a write cache miss occurs for write-through cache mode, the data is written to main memory and the cache is not updated. If the write cache miss occurs during copy-back cache mode (see *Figure 8–14*) and the selected cache line is not modified, the missed cache line is fetched from main memory. If a write cache miss occurs during copy-back mode and the selected cache line is modified, the CY7C604 simultaneously flushes the modified cache line into the write buffers while it fetches the new cache line from main memory. After the cache line has been replaced, the write access is enabled by the CY7C604. The modified cache line is written to main memory from the write buffers as a background task.

8.3.2.1 CY7C604 Cache Tag

The CY7C604 features 2048 direct-mapped cache tag entries, as shown in *Figure 8–12*. The on-chip cache tag and the TLB are accessed simultaneously. Each entry in the cache consists of 16 bits of virtual address (VA<31:16>), a 12-bit context number (CXN<11:0>), one valid bit (V) and one modified bit (M). The valid bit (V) is set or cleared to indicate the validity of the cache tag entry. The modified bit (M) of a cache tag entry is set during copy-back mode after a write access to the cache line. This indicates that the cache line has been modified. The modified bit has no meaning for write-through cache mode. The cache line select field (VA<15:5>) is used to select a cache line entry and its corresponding cache tag entry. The address field VA<31:16> and context register are compared against the virtual address and the context fields of the selected cache tag entry. If a match occurs, then a cache hit is generated. If a match is not found, then a cache miss is generated. To complete an access successfully, both the cache tag and the TLB must be hit with appropriate access-level permission. Upon power-on reset ($\overline{\text{POR}}$), all cache tag entries are invalidated (all V bits are cleared).

A supervisor bit (S) is included in the cache tag entry. For cache tag entries which are accessible by the supervisor only (access-level field 6 or 7), the S bit is set. During a cache tag look up, if the access is supervisor mode and the the S bit is set, the context number comparison is ignored and the context match is forced. This operation is similar to a TLB look up with access-level field set to either 6 or 7.

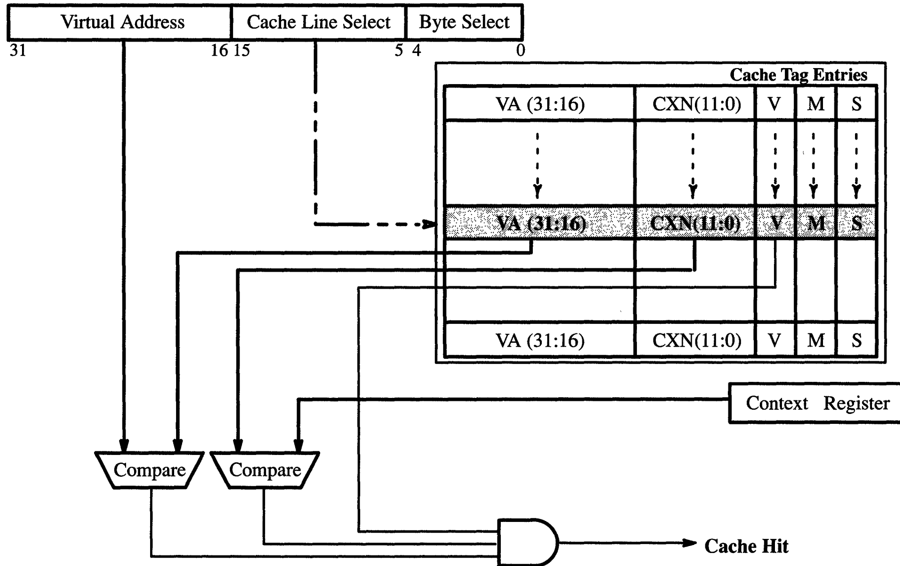


Figure 8–12. CY7C604 Cache Tag

8.3.2.2 CY7C604 Address Aliasing

Two or more virtual addresses mapped to the same physical address is known as *aliasing*. This must be detected to maintain data consistency in a virtual cache system. The SPARC reference system software convention permits the use of aliases in address spaces that are modulo with respect to the system's underlying cache size. In order to allow the efficient caching of physical memory pages where such aliases may occur, the CY7C604 supports automatic address aliasing protection.

The CY7C604 tests for address aliasing during copy-back read or copy-back write cache misses or during write-through read misses. The MMU must be enabled to allow the CY7C604 to test and correct address aliases.

To detect address aliasing, the virtual address of the selected cache tag entry is translated through the MMU. The translated physical address is compared with the physical address of the missed cache access. If the physical address of the selected cache tag entry and the physical address of the cache miss match, then address aliasing is detected.

The SPARC system software convention ensures that the aliasing maps to the same cache line address for a particular CY7C604. Coupled with this convention, the cache controller hardware automatically prevents any existence of address aliases in the virtual caches.

Aliasing is checked during a cache miss. If detected, an alias is corrected by updating the selected cache tag entry with the new virtual address. The CY7C604 then halts the cache miss processing and provides an access to the cache, as with a cache hit. If no alias is detected, the cache miss processing proceeds normally. The state diagrams for write-through and copy-back cache modes with alias detection and correction are illustrated in *Figure 8–13* and *Figure 8–14*.

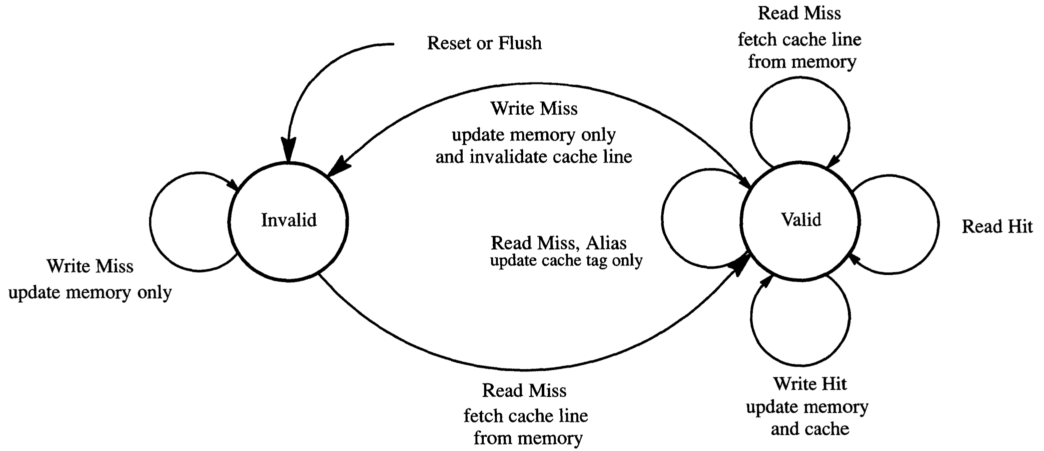


Figure 8–13. CY7C604 Write-Through with No Write Allocate

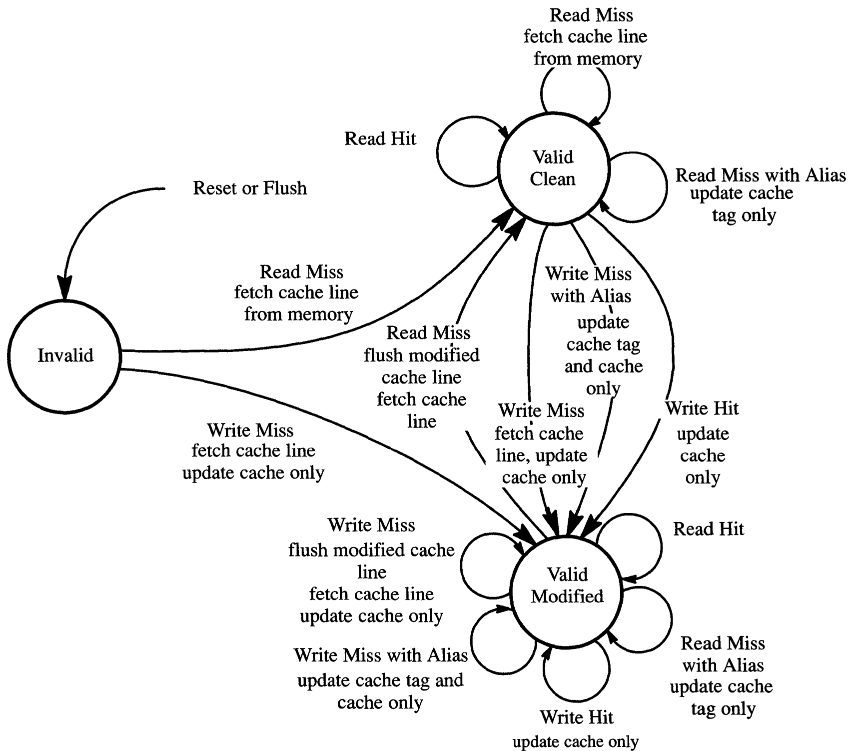


Figure 8–14. CY7C604 Copy-Back with Write Allocate

In copy-back mode, address aliasing is checked during a read- or a write-access cache miss. For an alias detected during a read-access cache miss, the selected cache tag entry is updated with the virtual address that caused the cache miss. The cache miss processing is halted, and the CY7C601 is supplied with data from the cache.

If an address alias is detected during a write access cache miss, the selected cache tag entry is updated with the new virtual address that caused the cache miss. The modified bit is set if it was not set previously. The cache miss processing is halted, and the cache write access is enabled.

In write-through mode, address aliasing is checked only on read-access cache misses. If an address alias is detected on a read-access cache miss, the old cache tag entry is replaced with the new virtual address. The cache miss is halted, and the cache supplies the data requested.

In write-through cache mode, address aliasing is not checked during write-access cache misses. In order to avoid potential address aliasing, the selected cache line is invalidated. Address aliasing is not checked in this case in order to avoid unnecessary performance degradation.

To detect address aliasing, the selected cache line address is translated through the TLB. Protection checking is ignored during this translation. The translation may occasionally cause a TLB miss. If this happens in a write-through read miss case, the alias checking and the TLB miss are ignored. In a copy-back read miss or a write miss when the selected cache line is clean, alias checking and TLB miss processing are ignored. To provide data consistency, the table walk is performed in order to detect address aliasing in a copy-back read miss or a write miss when the selected cache line is modified.

8.3.2.3 CY7C604 Cache Lock

The CY7C604 supports a cache lock mechanism that allows the system to lock all entries in the cache. This feature is provided to allow deterministic response times for real-time systems. The cache lock function affects only cache miss operations, since it locks out cache line replacement of valid entries. Since alias detection is not enabled, shared memory pages must be declared as non-cacheable when the cache is locked . The following description summarizes each case in detail:

a. *Write-through read miss and selected entry is invalid:* A new cache line is fetched from main memory to load into the cache and the requested data is supplied to CY7C601 as in normal operation mode.

b. *Write-through read miss and selected entry is valid:* The requested data is obtained from main memory as a non-burst transaction on the MBus and supplied to the CY7C601, but is not loaded into the cache.

c. *Write-through write miss:* The selected cache line is invalidated in order to prevent data inconsistency due to potential address aliasing.

d. *Copy-back read miss and selected entry is invalid:* A new cache line is fetched from main memory to load into the cache and the requested data is supplied to CY7C601 as in a normal operation.

e. *Copy-back read miss, selected entry is valid:* The requested data is obtained from main memory as a non-burst transaction on the MBus and supplied to the CY7C601, but is not loaded into the cache.

f. *Copy-back write miss and selected entry is invalid:* A new cache line is fetched from main memory to load into the cache and the CY7C601 data is stored in the cache as in a normal operation.

g. *Copy-back write miss and selected entry is valid:* The CY7C601 data is stored in the main memory as a non-burst transaction on the MBus, but the cache is not updated.

8.3.3 CY7C605 Cache Controller

The cache controller provides cache memory access control for a 64-Kbyte direct-mapped virtual cache. The cache controller performs this task by comparing memory accesses against the address and status entries in a cache tag memory. The CY7C605 provides two separate cache tag memories for access comparison. Cache memory accesses from the processor are compared against the processor virtual cache TAG (PVTAG) memory. Bus snooping operations are compared against the MBus physical cache TAG (MPTAG) memory. The use of two cache tag memories allows the cache controller to service processor cache accesses concurrently with bus snooping cache tag accesses. This feature of the CY7C605 provides significant performance improvements over cache systems sharing a single cache tag memory between the processor cache access and the bus snooping operations. Single cache tag systems typically must stall the processor when a bus snooping operation is required, causing serious performance degradation.

The cache controller is designed to use two CY7C157 Cache Storage Units for the cache memory. These cache RAMs are 16-Kbyte x 16 SRAMs with on-chip address and data latches and timing control. Two CY7C157s and one CY7C604/CY7C605 comprise an entire 64-Kbyte cache system with physical bus interface and read and write buffers.

The cache is organized as 2048 cache lines of 32 bytes each. The CY7C604/CY7C605 has 2048 cache tag entries in both the PVTAG and MPTAG, one entry in each cache tag memory per cache line. Addressing for the virtual cache is provided directly from the virtual address bus. The virtual address field VA<15:5> selects one of the 2048 lines of the cache (refer to *Figure 8–15*). This address field also selects the cache tag entry in the PVTAG dedicated to the selected cache line. A cache hit occurs when the upper sixteen bits of the virtual address and the context register match with the virtual address and context stored in the selected cache tag entry in PVTAG. The lowest five bits of the virtual address bus (VA<4:0>) select one or more of the 32 bytes in the cache line. Cache data replacement is always performed by replacing cache lines.

The cache is designed to provide data with every read access asserted on the virtual bus, regardless of the cache controller. The CY7C605 controls cache read access by holding the CY7C601 with $\overline{\text{MHOLD}}$ if a cache hit is not detected by the cache controller. The cache controller then reads the new cache line from main memory, and supplies the correct data to the CY7C601. After the correct data is latched into the CY7C601 by strobing the $\overline{\text{MDS}}$ signal, the CY7C601 is released and execution proceeds normally.

Writes to the cache are controlled by the CY7C604/CY7C605, which decodes the lowest two bits of the virtual address, the SIZE<1:0> signal, and checks for a cache hit to enable the correct cache byte write enable signals. If a cache write hit occurs, the CY7C604/CY7C605 the correct $\overline{\text{CBWE}}$ signals for the write access, and outputs these to the CY7C157 Cache Storage Unit write enables. If the cache mode is set to write-through (see *Section 8.3.1, Cache Modes*), the write data is also written to main memory. If a write cache miss occurs for write-through cache mode, the data is written to main memory and the cache is not updated. If the write cache miss occurs during copy-back cache mode, the cache line is fetched from main memory. If the cache line stored in the cache when the write cache miss occurred has been modified, the old cache line is written to main memory before the cache line is replaced by the new data. After the cache line has been replaced, the write access is enabled by the CY7C604/CY7C605.

8.3.3.1 CY7C605 Cache Tag

The CY7C605 features two separate cache tag arrays: the processor virtual cache tag memory (PVTAG) and the MBus physical cache tag memory (MPTAG). Cache controllers using only one cache tag array must delay the processor when bus snooping requires access to the cache tags. The inclusion of two independent cache tag memories allows the CY7C605 to support processor accesses to cache while simultaneously performing bus snooping on the MBus.

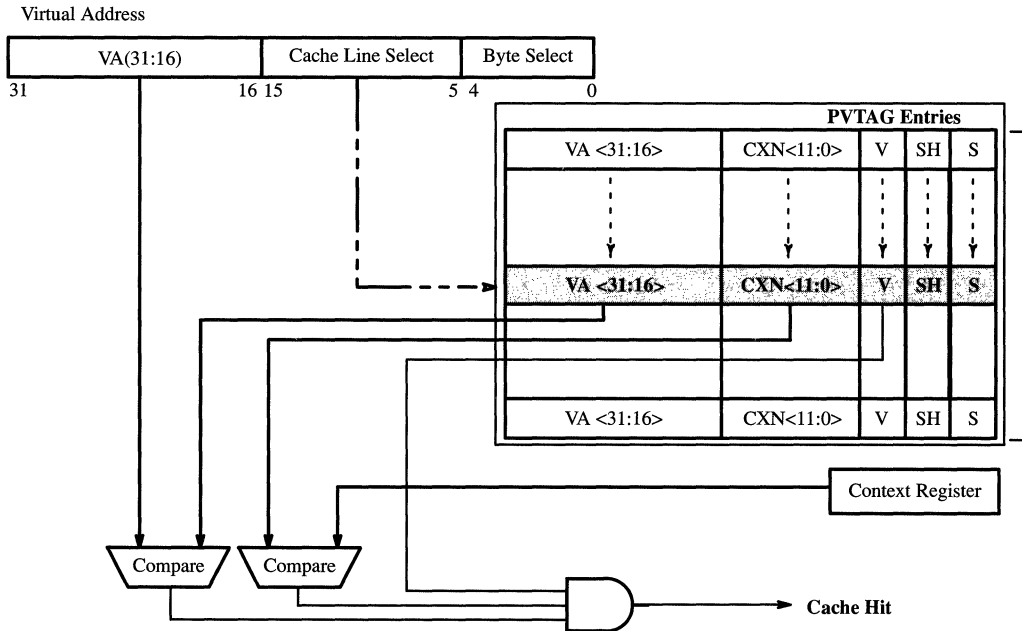


Figure 8–15. CY7C605 Processor Virtual Cache Tag (PVTAG) Comparison

8.3.3.1.1 CY7C605 Processor Virtual Cache Tag (PVTAG)

The PVTAG consists of 2048 direct-mapped cache tag entries, as shown in *Figure 8–16*. The PVTAG and the TLB are accessed simultaneously. Each entry in the cache consists of 16 bits of virtual address (VA<31:16>), a 12-bit context number (CXN<11:0>), one valid bit (V), and one shared bit (SH). The valid bit (V) is set or cleared to indicate the validity of the cache tag entry. The shared bit (SH) of a cache tag entry is set when bus snooping indicates that the cache line is shared. The cache line select field (VA<15:5>) is used to select a cache line entry and its corresponding cache tag entry. The address field VA<31:16> and context register are compared against the virtual address and the context fields of the selected cache tag entry. If a match occurs, then a cache hit is generated. If a match is not found, then a cache miss is generated. To complete an access successfully, both the cache tag and the TLB must be hit with appropriate access-level permission. On power-on reset ($\overline{POR}$), all cache tag entries are invalidated (all V bits are cleared).

A supervisor bit (S) is included in the cache tag entry. For cache tag entries which are accessible by the supervisor only (access-level field 6 or 7), the S bit is set. During a cache tag look up, if the access is supervisor mode and the the S bit is set, the context number comparison is ignored and the context match is forced. This operation is similar to a TLB look up with access-level field set to either 6 or 7.

8.3.3.1.2 CY7C605 MBus Physical Cache Tag (MPTAG)

The MPTAG consists of 2048 direct-mapped, physical address cache tag entries (refer to *Figure 8–16*). Each entry in the cache consists of 24 bits of physical address (PA(35:12)), a valid bit (V), a shared bit (SH), and a modified bit (M).

The 2048 MPTAG entries are virtual address indexed. The index field for MPTAG, as supplied by the MBus, is formed by concatenating the superset virtual address bits <15:12> (MAD<49:46>) with physical address bits <11:5> (MAD<11:5>) (refer to *Figure 8–17*). The format of the MBus address bus cycle is described in *Section 11.1.5* of *Chapter 11*.

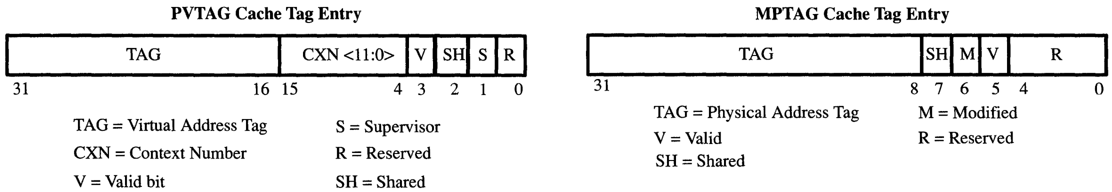


Figure 8–16. CY7C605 Cache Tag Entries

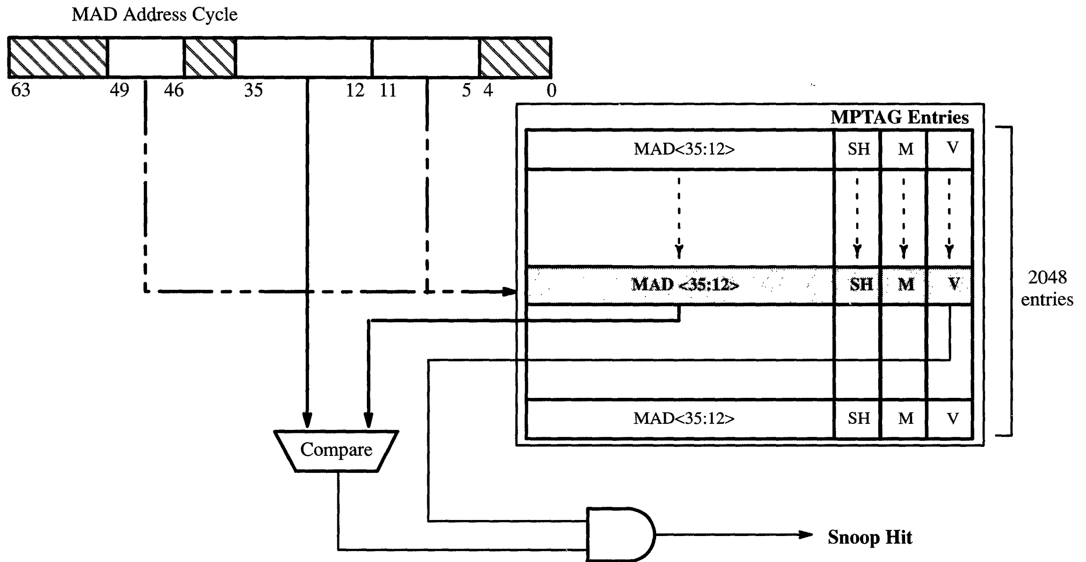


Figure 8–17. CY7C605 MBus Physical Cache Tag (MPTAG) Comparison

During a MPTAG compare operation, the physical address field <35:12> of the access is compared against the physical address field of the MPTAG entry selected by the virtual address index. If a match occurs and the valid bit is set, a cache hit is generated. If a match is not found, or the valid bit is not set, a cache miss is generated. On Power-On Reset (POR), all the MPTAG cache entries are invalidated (V bits are cleared).

8.3.3.2 CY7C605 Multiprocessing Support

The CY7C605 is specifically designed to support multiprocessing systems. The CY7C605 accomplishes this by providing features necessary to maintain cache coherency with a second-level memory system (typically main memory or a secondary cache) and other caching systems on the shared bus.

The CY7C605 supports two modes of caching: write-through and copy-back. Operation in write-through caching mode causes main memory to be modified with each write access to the cache. This avoids the issue of lack of coherency between the individual cache systems and main memory, but greatly increases memory bus traffic. The effect of this increased bus traffic is a degrading of the performance of a multiprocessor system as the processing nodes compete for memory bus bandwidth. This problem is greatly reduced when copy-back caching mode is used.

Operation in copy-back mode causes all changes to a cache line to be held until the line is flushed from the cache. This minimizes bus traffic to only those transactions necessary to maintain the cache. However, by

allowing the cache line to be modified without updating main memory, a problem arises when other processing nodes require an up-to-date copy of that memory location. The problem of modified cache lines is solved by the enforcement of a cache coherency protocol.

The CY7C605 implements a cache coherency protocol specified by the SPARC reference standard MBus level-2 interface. This protocol is modeled after that used by the IEEE Futurebus. In this protocol, each cache line is described by one of five states: invalid (I), exclusive clean (EC), exclusive modified (EM), shared clean (SC), and shared modified (SM). The following describes these five cache states:

Invalid (I): Cache line is not valid.

Exclusive Clean (EC): Only this cache module has a valid copy of this cache line, other than the next level of memory (main memory or secondary cache). No other cache module on the same level of memory has a valid copy of this cache line.

Exclusive Modified (EM): Only this cache module has a valid copy of this cache line. This cache module is the OWNER of the cache line, and has the responsibility to update the next level of memory (main memory or secondary cache) and also to supply data if any other cache references this memory location.

Shared Clean (SC): The same cache line may exist in more than one cache module. The next level of memory may or may not contain a valid copy of this cache line, depending upon whether this cache line has been modified in any other cache.

Shared Modified (SM): The same cache line may exist in more than one cache module, but this cache module is the OWNER of the cache line. The next level of memory does not have a valid copy of this cache line, and this cache module has the responsibility to update the next level of memory and to supply any other cache that may reference this same memory location.

These five states are described by three state bits (valid (V), shared (SH), and modified(M)) in each MPTAG cache tag entry (refer to *Figure 8–16*). The PVTAG cache tag entries are described by two state bits: valid (V), and shared (SH). The PVTAG cache tag entries corresponding to the same cache lines can be in one of three states: invalid, exclusive valid, and shared valid.

In write-through cache mode, only the valid and invalid states apply to either the MPTAG or PVTAG cache tag entries. The shared and modified bits in the MPTAG, and the shared bit in the PVTAG, are ignored by the CY7C605.

8.3.3.3 CY7C605 Cache State Transitions

The following sections describe the five cache line states (invalid, exclusive clean, exclusive modified, shared clean, and shared modified) and the transitions these states undergo due to transactions on the MBus. Each numbered transition in a section corresponds to a numbered transition on the state diagram for that section. Note that state transitions are dependent upon both the cache transaction and the state of the MBus signals: memory shared ($\overline{\text{MSH}}$), and memory inhibit ($\overline{\text{MIH}}$).

All processor transactions described in this section affect the processor serviced by the CY7C605. All Coherent transactions affect all bus agents on the MBus with a copy of the shared cache line. For further information on MBus transactions, please refer to *Section 11.1.7*.

8.3.3.3.1 Copy-Back Invalid

Processor Read Miss: CY7C605 issues a Coherent Read transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the data is supplied to the processor in the cycle following the last cache line entry.

1. If $\overline{MSH}$ = HIGH, then invalid changes to exclusive valid in PVTAG and invalid changes to exclusive clean in MPTAG.
2. If $\overline{MSH}$ = LOW, then invalid changes to shared valid in PVTAG and invalid changes to shared clean in MPTAG.

Processor Write Miss: CY7C605 issues a Coherent Read and Invalidate transaction on the MBus. The CY7C605 reads the cache line from the second-level memory and loads it into the cache RAM. Then the processor data is written into the cache RAM in the cycle following the last cache line entry.

3. Invalid changes to exclusive valid in PVTAG and invalid changes to exclusive modified in MPTAG.

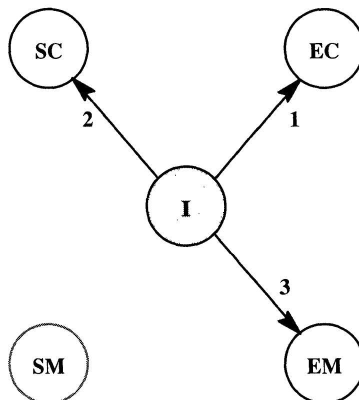


Figure 8–18. Copy-back Invalid

8.3.3.3.2 Copy-back Exclusive Clean

Processor Read Hit: The CY7C605 will supply data to the CY7C601 immediately.

1. PVTAG entry is exclusive valid; exclusive clean in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will issue a Coherent Read transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the data is supplied to the CY7C601 in the cycle following the last cache line entry.

2. If $\overline{MSH}$ = HIGH, then exclusive valid in PVTAG; exclusive clean in MPTAG.
3. If $\overline{MSH}$ = LOW, then shared valid in PVTAG; exclusive clean changes to shared clean in MPTAG.

Processor Write Hit: The CY7C605 will update the cache immediately with the CY7C601 data.

4. PVTAG entry is exclusive valid; exclusive clean changes to exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will issue a Coherent Read and Invalidate transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the processor data is written into the cache RAM in the cycle following the last cache line entry.

5. PVTAG entry is exclusive valid; exclusive clean changes to exclusive modified in MPTAG.

Software Flush (Store alternate instruction with ASI = 10H to 14H; see Section 8.3.7): The CY7C605 will invalidate both the PVTAG and MPTAG cache tag entries.

6. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the MBus Coherent Read transaction, the CY7C605 will assert $\overline{MSH}$ and change the state of the cache line from exclusive clean to shared clean.

7. Assert $\overline{MSH}$; exclusive clean is changed to shared clean in MPTAG and shared valid in PVTAG.

Coherent Read and Invalidate: Both the PVTAG and the MPTAG cache tag entries in the CY7C605 are invalidated.

8. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Invalidate: Both the PVTAG and the MPTAG entries in the CY7C605 are invalidated.

9. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Write and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

10. Exclusive valid is changed to invalid in PVTAG and exclusive clean is changed to invalid in MPTAG.

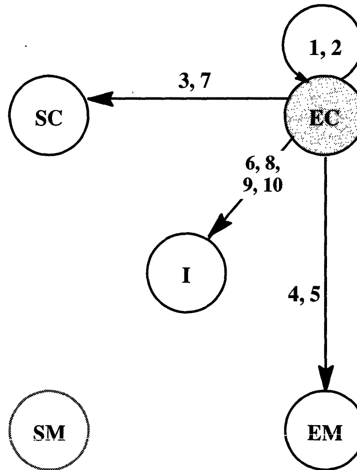


Figure 8–19. Copy-back Exclusive Clean

8.3.3.3.3 Copy-back Shared Clean

Processor Read Hit: The CY7C605 will supply data immediately to the CY7C601.

1. PVTAG entry is shared valid; shared clean in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will issue a Coherent Read transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. Then the data is supplied to the CY7C601 in the cycle following the last cache line entry.

2. If $\overline{MSH}$ = HIGH, then exclusive valid in PVTAG and shared clean is changed to exclusive clean in MPTAG.

3. If $\overline{MSH}$ = LOW, then shared valid in PVTAG and shared clean in MPTAG.

Processor Write Hit: The CY7C605 issues a Coherent Invalidate transaction on the MBus. The CY7C605 will wait for $\overline{MRDY}$ before updating the cache with the processor data in case a Relinquish and Retry is received.

4. PVTAG entry is exclusive valid; shared clean is changed to exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will issue a Coherent Read and Invalidate transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and then load the data into the cache

RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry.

5. PVTAG entry is changed to exclusive valid; shared clean is changed to exclusive modified in the MPTAG.

Software Flush: The CY7C605 will invalidate both the PVTAG and MPTAG cache tag entries.

6. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the MBus Coherent Read transaction, the CY7C605 will assert the $\overline{MSH}$.

7. Assert $\overline{MSH}$; shared clean in MPTAG and shared valid in PVTAG.

Coherent Read and Invalidate: Both the PVTAG and the MPTAG cache tag entries will be invalidated.

8. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Write and Invalidate: Both the PVTAG and MPTAG cache tag entries are invalidated.

10. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

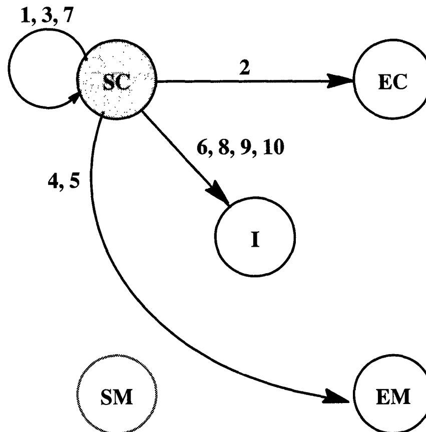


Figure 8–20. Copy-back Shared Clean

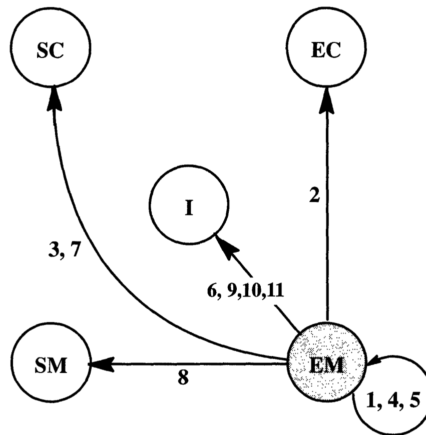


Figure 8–21. Copy-back Exclusive Modified

8.3.3.3.4 Copy-Back Exclusive Modified

Processor Read Hit: The CY7C605 will supply data to the processor immediately.

1. PVTAG entry is exclusive valid; exclusive modified in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will initiate a Coherent Read transaction followed by a block write transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. Then the data will be supplied to the processor in the cycle following the last cache line entry into the cache RAM. The modified cache line has to be written to update the second-level memory. The MBus busy ($\overline{MBB}$) signal is asserted from the beginning of the Coherent Read transaction to the end of the write transaction on the MBus, unless a Relinquish and Retry is received for either transaction. To insure coherency in this case, the system must insure that there is no coherent transaction on the MBus between the read and write that requires the invalidation or intervention of the data in the write buffer (there is no snooping on the write buffer).

2. If $\overline{MSH}$ = HIGH, then the PVTAG entry is exclusive valid, and the MPTAG entry is changed from exclusive modified to exclusive clean.
3. If $\overline{MSH}$ = LOW, then the PVTAG entry is changed to shared valid, and the MPTAG entry is changed from exclusive modified to shared clean.

Processor Write Hit: The CY7C605 will update the cache immediately with the processor data.

4. PVTAG entry is exclusive valid; exclusive modified remains as exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will initiate a Coherent Read and Invalidate transaction followed by a block write transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry into the cache RAM. The modified cache line must be written into the second-level memory in order to update the memory. The $\overline{MBB}$ signal is asserted from the beginning of the Coherent Read and Invalidate transaction to the end of the write transaction on the MBus, unless a Relinquish and Retry is received for either transaction. To insure coherency in this case, the system must insure that there is no coherent transaction on the MBus between the read and write that requires the invalidation or intervention of the data in the write buffer (there is no snooping on the write buffer).

5. PVTAG entry remains exclusive valid; the MPTAG entry remains exclusive modified.

Software Flush: The CY7C605 initiates a block write transaction on the MBus. The CY7C605 will write the modified cache line to update the second-level memory and then it invalidates both the PVTAG and MPTAG cache tag entries.

6. Exclusive valid is changed to invalid in PVTAG; exclusive modified is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the Coherent Read transaction on the MBus, the CY7C605 asserts both the $\overline{MSH}$ and $\overline{MIH}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the Coherent Read transaction on the MBus.

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the CY7C605 changes the state of the MPTAG cache tag entry from exclusive modified to shared clean, and the PVTAG entry from exclusive valid to shared valid.
8. If the memory reflection (MR) bit of the SCR is cleared, the CY7C605 changes the state of the MPTAG entry from exclusive modified to shared modified. The PVTAG entry is changed to shared valid.

Coherent Read and Invalidate: During the A+2 cycle of a Coherent Read and Invalidate transaction on the MBus, the CY7C605 asserts the $\overline{MIH}$ signal. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the Coherent Read transaction on the MBus. Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries in the CY7C605 are invalidated.

10. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

Coherent Write and Invalidate: Both the PVTAG and the MPTAG cache tag entries are invalidated.

11. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

8.3.3.3.5 Copy-back Shared Modified

Processor Read Hit: The CY7C605 will supply data immediately to the CY7C601.

1. PVTAG entry is shared valid; shared modified in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will initiate a block read transaction followed by a block write transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. Then the data will be supplied to the processor in the cycle following the last cache line entry into the cache RAM. The modified cache line has to be written to update the second-level memory. The $\overline{MBB}$ signal is asserted from the beginning of the Coherent Read transaction to the end of the write transaction on the MBus, unless a Relinquish and Retry is received for either transaction. To insure coherency in this case, the system must insure that there is no coherent transaction on the MBus between the read and write that requires the invalidation or intervention of the data in the write buffer (there is no snooping on the write buffer).

2. If $\overline{\text{MSH}}$ = HIGH, the PVTAG entry changes to exclusive valid. The MPTAG entry is changed from shared modified to exclusive clean.

3. If $\overline{\text{MSH}}$ = LOW, then the PVTAG entry changes to shared valid, and the MPTAG entry is changed from shared modified to shared clean.

Processor Write Hit: The CY7C605 issues a Coherent Invalidate transaction on the MBus. The CY7C605 will wait for $\overline{\text{MRDY}}$ before updating the cache with the processor data in case a Relinquish and Retry is received.

4. The PVTAG entry changes to exclusive valid; the entry in the MPTAG is changed from shared modified to exclusive modified.

Processor Write Miss: The CY7C605 will initiate a Coherent Read and Invalidate transaction followed by a block write transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry into the cache RAM. The modified cache line must be written into the second-level memory in order to update the memory. The $\overline{\text{MBB}}$ signal is asserted from the beginning of the Coherent Read and Invalidate transaction to the end of the write transaction on the MBus, unless a Relinquish and Retry is received for either transaction. To insure coherency in this case, the system must insure that there is no coherent transaction on the MBus between the read and write that requires the invalidation or intervention of the data in the write buffer (there is no snooping on the write buffer).

5. PVTAG entry is exclusive valid; the MPTAG entry is changed from shared modified to exclusive modified.

Software Flush: The CY7C605 initiates a block write transaction on the MBus. The CY7C605 will write the modified cache line to update the second-level memory and then it invalidates both the PVTAG and MPTAG cache tag entries.

6. Shared valid is changed to invalid in PVTAG; shared modified is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the Coherent Read transaction on the MBus, the CY7C605 asserts both the $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the Coherent Read transaction on the MBus.

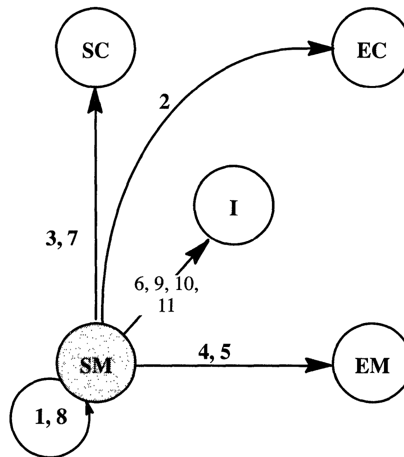


Figure 8–22. Copy-back Shared Modified

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the CY7C605 changes the state of the MPTAG from shared modified to shared clean, and the PVTAG entry is shared valid.
8. If the MR bit of the SCR is not set, then the PVTAG remains shared valid and the MPTAG remains shared modified.

Coherent Read and Invalidate: During the A+2 cycle of a Coherent Read and Invalidate transaction on the MBus, the CY7C605 asserts the $\overline{\text{MIH}}$ signal. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the Coherent Read transaction on the MBus. Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries in the CY7C605 are invalidated.

10. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

Coherent Write and Invalidate: Both the PVTAG and the MPTAG cache tag entries are invalidated.

11. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

8.3.3.3.6 Write-through Invalid

Processor Read Miss: The CY7C605 issues a block read transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. The data will be supplied to the processor in the cycle following the last cache line entry written to the cache RAM.

1. The PVTAG and MPTAG entries are changed from invalid to valid.

Processor Write Miss: The CY7C605 will issue a write-buffered Coherent Write and Invalidate transaction on the MBus.

2. The PVTAG and MPTAG entries remain invalid.

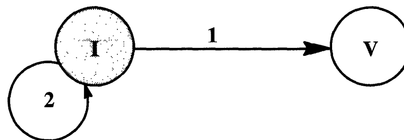


Figure 8–23. Write-through Invalid

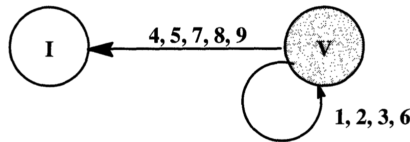


Figure 8–24. Write-through Valid

8.3.3.3.7 Write-through Valid

Processor Read Hit: The CY7C605 will supply data to the CY7C601 immediately.

1. The PVTAG and MPTAG entries remain valid: NO STATE CHANGE.

Processor Read Miss: The CY7C605 issues a block read transaction on the MBus. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. The data will be supplied to the processor in the cycle following the last cache line entry written to the cache RAM.

2. The PVTAG and MPTAG entries remain valid.

Processor Write Hit: The CY7C605 issues a write-buffered Coherent Write and invalidation transaction on the MBus. The CY7C605 will write data into the cache.

3. The PVTAG and MPTAG entries remain valid.

Processor Write Miss: The CY7C605 issues a write-buffered Coherent Write and Invalidate transaction on the MBus. The CY7C605 will write to main memory only and not to cache. If an alias is detected, the CY7C605 will also invalidate the cache line; if no alias is detected, the cache line is not invalidated.

4. The PVTAG and MPTAG entries change from valid to invalid.

Software Flush: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

5. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Read: During the A+2 cycle of the MBus Coherent Read transaction, the CY7C605 asserts $\overline{\text{MSH}}$.

6. Assert $\overline{\text{MSH}}$; the PVTAG and MPTAG entries remain valid.

Coherent Read and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

7. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Write and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

8. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

9. The PVTAG and MPTAG entries change from valid to invalid.

8.3.3.3.8 Bus Snooping

The CY7C605 bus snooper watches MBus transactions and snoops into the MPTAG array for certain transactions, as listed in *Table 8-6*.

8.3.3.4 CY7C605 Address Aliasing

Two or more virtual addresses mapped to the same physical address is known as *aliasing*. This must be detected to maintain data consistency in a virtual cache system. The SPARC reference system software convention permits the use of aliases in address spaces that are modulo with respect to the system's underlying cache size. In order to allow the efficient caching of physical memory pages where such aliases may occur, the CY7C605 supports automatic address aliasing protection.

Table 8-6. MBus Snooping Transactions

Cache Mode	Transaction Type	Snoop
Copy-Back	Coherent Read & Invalidate	yes
	Coherent Write & Invalidate	yes
	Coherent Read	yes
	Coherent Invalidate	yes
	Read	no
	Write	no
Write-Through	Coherent Read & Invalidate	yes*
	Coherent Write & Invalidate	yes
	Coherent Read	yes*
	Coherent Invalidate	yes*
	Read	no
	Write	no

* these transactions are not generated by the CY7C605, but the CY7C605 will snoop these transactions if generated by another bus master

The SPARC system software convention ensures that the aliased entry maps to the same cache line address for each CY7C605 in the multiprocessor system. Coupled with this convention, the cache controller hardware automatically prevents any existence of address aliases in the virtual caches.

The CY7C605 tests for address aliasing during all cache misses. Address aliasing cannot occur unless the MMU is enabled (ME bit of SCR). To detect address aliasing in the CY7C605, the physical address of the missed cache access is compared with the selected MPTAG entry.

If the physical address of the selected MPTAG entry and the physical address of the cache miss match, then address aliasing is detected. If detected, an alias is corrected by updating the selected cache tag entry with the new virtual address. The CY7C605 then halts the cache miss processing and provides an access to the cache, as with a cache hit. If no alias is detected, the cache miss processing proceeds normally.

For an alias detected during a read-access cache miss, the selected cache tag entry is updated with the virtual address that caused the cache miss. The cache miss processing is halted, and the CY7C601 is supplied with data from the cache.

If an address alias is detected during a copy-back mode write-access cache miss, the selected cache tag entry is updated with the new virtual address causing the cache miss. The modified bit is set if it was not set previously. The cache miss processing is halted, and the cache write access is enabled.

If an alias is detected on a write-through write-access cache miss, the cache line is written to main memory only (i.e., the cache is not updated) and the cache tag for that line is invalidated. If an alias is not detected, the cache line is still written to main memory, but the cache line is not invalidated.

8.3.4 CY7C604/CY7C605 Cache Control Signals

The CY7C604/605 controls the virtual cache through control signals supplied to the CY7C601 and to the cache RAMs. The signals used by the cache controller to control the CY7C601 consist of $\overline{\text{MHOLD}}$, $\overline{\text{MDS}}$,

and IOE. MHOLD is used to stall the CY7C601 until the CY7C604/605 can service the CY7C601 memory access request, such as during cache miss processing or during table walks. MDS is used by the CY7C604/605 to strobe data into the CY7C601 when MHOLD is asserted. This causes the CY7C601 to latch data on the data bus despite being stalled by the assertion of MHOLD. IOE is used as the enable signal for the AOE and DOE inputs of the CY7C601. When IOE is deasserted, the address and data bus output drivers of the CY7C601 are disabled. This feature is used to force the CY7C601 off of the virtual address and data buses.

The signals used to control the CY7C157 consist of the cache byte write enable (CBWE) and cache read output enable (CROE) signals. CROE is asserted low to enable the output of the cache RAMs during a cache read. CBWE <3:0> is asserted low to enable writing to the cache RAMs. The multiple CBWE signals allow the cache controller to enable byte, halfword, or word writes to the cache RAM. Single byte or halfword reads are handled by the CY7C601, which reads an entire 32-bit word and internally discards unwanted bytes.

During a cache read miss, the CY7C604/605 halts the CY7C601 by asserting MHOLD. The CY7C604/605 also deasserts IOE, which is used to disable the CY7C601 data bus and address bus output drivers. The cache controller fetches the new cache line from main memory, asserting CBWE<3:0> and the cache line addresses to write the data into the cache. Then the CY7C604/605 places the missed read data word on the data bus and toggles the memory data strobe signal (MDS) Toggling MDS forces the integer unit to latch the data on the data bus. The cache read miss terminates by reasserting the IOE signal and then releasing the MHOLD signal. IOE is typically reasserted one or more clocks before the MHOLD signal is deasserted, thus allowing the CY7C601 to output the next address onto the virtual address bus. This provides the address set-up time for the next memory access after MHOLD is released. Read misses are handled in the same manner for both copy-back and write-through modes of caching.

Cache write misses for write-through mode generally do not affect the operation of the CY7C601 due to the presence of write buffers in the CY7C604/605 (refer to the following section on the write buffer). In the case of a write miss, the write data is written to the write buffer instead of the cache memory and the cache tag for the cache line is invalidated. The write buffer writes the data to memory as a background task. The CY7C601 is stalled for a write-through write miss only if the write buffer is full. This occurs when the CY7C601 overruns the four doubleword buffers in the write buffer. In this case, MHOLD is asserted until space is made by the write buffer as it writes its contents into main memory.

On a write miss, if the cache mode is copy-back and the cache line is clean, the cache line is replaced in a similar manner as in the cache read miss described above. MHOLD is asserted to stall the CY7C601 and IOE is deasserted to force the CY7C601 off the data and address buses. A new cache line is read from main memory, and the cache is updated by writing the data into the cache. This is accomplished by supplying the cache addresses, cache line data from main memory, and asserting the CBWE signals to write the data. The write cache miss terminates by reasserting IOE, which causes the missed write data and address to reappear on their respective buses. The CY7C604/605 then strobes CBWE<3:0> according to the address and SIZE<1:0> signals to write the data into the cache. The copy-back write miss procedure terminates by deasserting MHOLD, which allows the processor to return to execution.

If the cache line is modified, the modified cache line is read out of the cache and stored into the write buffer during the same time the new cache line is fetched from main memory and stored in the read buffer (refer to the following sections on write and read buffers). MHOLD is asserted and IOE deasserted to force the CY7C601 into a halted and inactive state. The cache controller asserts CROE and the cache addresses to flush the modified cache line into the write buffer. The cache controller then writes the new cache line into the cache from the read buffer while simultaneously writing the modified cache line into main memory from the write buffer. This is accomplished by supplying the cache addresses for the cache line data, and asserting the CBWE<3:0> signals to write the data into the cache. The copy-back write miss for a modified cache line terminates by releasing IOE to allow the missed write data and address to reassert on the data and address

buses. The CY7C604/605 asserts the $\overline{\text{CBWE}}$<3:0> signals to write the data into the cache. The $\overline{\text{MHOLD}}$ signal is then deasserted to allow the CY7C601 to return to processing. See *Section 8.11* for virtual bus timing diagrams.

8.3.5 CY7C604/605 Write Buffer

The CY7C604/605 supports four write buffers on chip, as shown in *Figure 8–25*. In write-through mode, each buffer can store two 32-bit words, which efficiently supports store double operations. A physical address tag is associated with each of the four buffers in write-through mode. Upon a write access, the write buffers are loaded with the data to be written to main memory. This allows the CY7C601 to continue operation without stalling due to memory access delays on the physical bus.

In copy-back mode, the same buffers are configured to store a 32-byte cache line with a single physical address as shown in *Figure 8–26*. This allows for faster cache line flushes during modified cache line replacement. The modified cache line is flushed into the write buffer as the new cache line is simultaneously fetched from main memory. In either case, the contents of the buffers are transferred to main memory as a background task. On power-on reset ($\overline{\text{POR}}$), all of the write buffers are invalidated.

Non-cacheable writes use the four write buffers in the same manner as write-through cache transaction, even if copy-back mode is enabled. However, a copy-back cache line and non-cacheable data cannot simultaneously occupy the write buffer.

The CY7C604/605 requests MBus ownership as soon as one of the write buffers is valid. For each write buffer transfer, the CY7C604/605 re-arbitrates the MBus again. A modified cache-line flush is considered as one transaction. When the bus is still granted to the CY7C604/605 (i.e., bus parking), the CY7C604/605 can transfer the data immediately without any bus re-arbitration (so there are no dead clocks between transactions). Once all of the write buffers are full, further writes from the CY7C601 are held until a buffer is empty. If there is a read access cache miss, the CY7C601 is held until all of the write buffers are written back into main memory in order to maintain data consistency. After the write buffers are cleared, the CY7C604/605 resumes the task of fetching the cache line for the cache read miss.

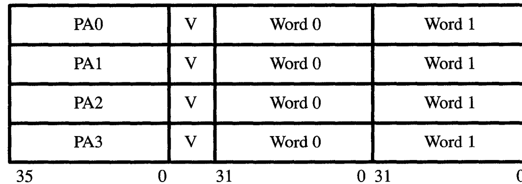


Figure 8–25. Write Buffers (Write-Through Mode or Non-Cacheable Write)

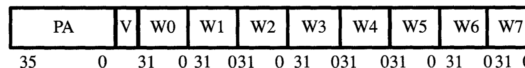


Figure 8–26. Write Buffer (Copy-Back Mode)

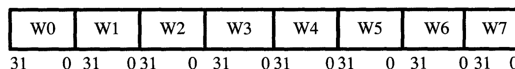


Figure 8–27. Read Buffer (Copy-Back Mode)

8.3.6 CY7C604/605 Read Buffer

The CY7C604/605 provides a read buffer of 32 bytes (one cache line) in order to support simultaneous writing of a modified cache line to main memory and reading of a new cache line from main memory into the cache under copy-back mode. The read buffer is shown in *Figure 8–27.* The read buffers are invalidated on power-on reset.

8.3.7 CY7C604/605 Cache Flushing Operations

The CY7C604/605 supports five different levels of cache flushing operations, as illustrated in *Table 8-7.* The cache flushing operations are dependent upon the cache mode and state. Flushing under copy-back cache mode for a modified cache line means flushing the cache line into main memory and invalidating the cache tag entry. If the cache line is clean (copy-back mode), or is in write-through cache mode, flushing only invalidates the cache tag entry.

Unlike a TLB flush operation, all cache flushing operations flush only one cache line at a time. Each cache line can be flushed on the basis of a page, segment, region, context, or user mode, as illustrated in *Table 8-7.* The levels of address matching for a cache line flush vary from a full 4-Kbyte page level match of address and context, to a match of user mode only.

The cache line selected for operation is indexed as in normal cache access operations (VA<15:5>). If the cache flush operation does not cause a match of the cache tag entry, no action occurs. The five types of cache flush operations are: page flush, segment flush, region flush, context flush, and user flush. These different levels of cache flush are mapped with the ASI bits. The store alternate space instructions for the CY7C601 must be used to assert the ASI value that corresponds with the level of cache flush operation desired. The combination of the ASI and a store operation using the virtual address specify the cache flush operation and the cache line to be matched for flushing. During flush operations, the context register provides the context number to be compared.

Table 8-7. Cache Flush Operations

Cache Flush	ASI	Compares:
PAGE	10 H	Context (or Supervisor S = 1,), Index 1, Index 2, and Index 3 (bits 17 and 16)
SEGMENT	11 H	Context (or Supervisor S = 1), Index 1, and Index 2
REGION	12 H	Context, (or Supervisor S = 1), and Index 1
CONTEXT	13 H	Context and User (S = 0)
USER	14 H	User (S = 0)

Table 8-8. Cacheable / Non-Cacheable accesses

Access	Condition
Not cached	ASI = 20-2F H (By-pass) or ASI = 1 (Local)
	ASI = UN, RES (unassigned/reserved)
	BM = 1 and ME = x and CE = x and ASI = 8,9 H
	BM = x and not (ME = 1 and CE = 1 and PTE[C] = 1)
	LDST cycles in write-through mode
	Table walk cycles
	Cache lock miss accesses which have valid entries, but no alias
Cached	BM = 0 and ME = 1 and CE = 1 and ASI = 8,9,A,B H and PTE[C] =1
	BM = 1 and ME = 1 and CE = 1 and ASI = A,B H and PTE[C] =1

8.3.8 CY7C604/605 Cacheable/Non-Cacheable Memory Accesses

Pages that are declared as non-cacheable (C = 0 in the page table entry (PTE)) are not cached in the cache RAM and, as such, there are no associated cache tag entries in the CY7C604/605. For data consistency and implementation reasons, the CY7C604/605 assumes the following cycles are also non-cacheable:

- LDST cycles in write-through mode
- Table walk accesses
- Cache-missed accesses during cache-lock mode (CY7C604 only)
- Boot mode accesses (except user/supervisor data accesses when the MMU is enabled and the cache is enabled)
- Pass-through mode accesses
- By-pass mode accesses
- Accesses while the cache is disabled
- Local-mode accesses
- When MMU is disabled (ME bit of SCR = 0)

Table 8-8 shows the CY7C604/605 operation conditions for cacheable and non-cacheable accesses. Refer to *Section 8.2* for additional information.

8.3.9 CY7C604/605 MBus Cacheable (MC) Bit

One of the CY7C604/605 output signals is a MBus cacheable bit, which is embedded in the MBus address phase as MAD<43> (Refer to *Chapter 11* for more information on MBus.) The MBus cacheable bit indicates the cacheable status of a memory access by the CY7C604/605. This information is consistent with the cache visibility philosophy of the CY7C604/605 and is made available for use by a secondary cache tag array.

When the MMU is enabled, the MC bit is set by the state of the C bit in the corresponding PTE entry. When the MMU function of the CY7C604/605 is disabled, the C bit of the SCR register sets the value of the MC bit. The C bit of the SCR register is loaded by the CY7C601, and it defines the cacheable status of memory accesses when the MMU is disabled. *Table 8-9* illustrates the state of the MC bit for various CY7C604/605 operation conditions.

Table 8-9. State Table for MC (Memory Cacheable) Bit

MC	Condition
0	ASI = 20-2F H or ASI = 1 H
not applicable	ASI = UN, RES
SCR[C]	Not one of the above and ME =0 or Not one of the above and (BM = 1 and ASI = 8,9 H) or Not one of the above and table walk
PTE[C]	Not one of the above

8.3.10 CY7C604/605 LDST (Atomic Load/Store Instruction) cycles

In order to maintain data consistency under write-through cache mode, LDST (atomic Load/Store) cycles are treated as non-cacheable transactions. All LDST accesses are forced into main memory in this case. The C bit in the TLB entry is output on the MBus as the MC (MAD<43>) bit. If a cache hit occurs on a LDST cycle with the cache in write-through mode, the cache line is invalidated. If the MMU is disabled, the C bit in the SCR is output on the MC signal of the MBus.

In copy-back mode, LDST cycles are treated as normal memory accesses and are cached according to the C bit of the PTE associated with the access.

LDST operations on the physical bus (MBus) are repeated if interrupted by a Relinquish and Retry before the load operation of the LDST has been completed. However, if the Relinquish and Retry occurs after the load operation has completed, only the store operation of the LDST is repeated.

8.3.11 CY7C604/605 Cache Byte Write Enables

The CY7C604/605 supports four separate byte write enables ($\overline{\text{CBWE}}$<3:0>) to control write accesses to the CY7C157 Cache Storage Units. These signals are generated using the lower two bits of the virtual address (VA<1:0>) and size (SIZE<1:0>) information during write accesses.

The decoding of the SIZE<1:0> and VA<1:0> bits is shown in *Table 8-10*. The $\overline{\text{CBWE0}}$ signal controls the most significant byte (MSB), which is located at a word-aligned address N. $\overline{\text{CBWE3}}$ controls the least-significant byte, located at address N+3. All of the byte write enables are asserted for a cache line load into the cache RAM during a cache miss.

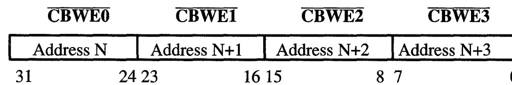


Figure 8–28. $\overline{\text{CBWE}}$ Byte Assignments

Table 8-10. Byte Write Enables

Size<1:0>	A<1:0>	$\overline{\text{CBWE3}}$	$\overline{\text{CBWE2}}$	$\overline{\text{CBWE1}}$	$\overline{\text{CBWE0}}$
00	00	1	1	1	0
00	01	1	1	0	1
00	10	1	0	1	1
00	11	0	1	1	1
01	00	1	1	0	0
01*	01*	1	1	1	1
01	10	0	0	1	1
01*	11*	1	1	1	1
10	00	0	0	0	0
10*	01*	1	1	1	1
10*	10*	1	1	1	1
10*	11*	1	1	1	1
11	00	0	0	0	0
11*	01*	1	1	1	1
11*	10*	1	1	1	1
11*	11*	1	1	1	1

* Denotes an illegal combination of Size<1:0> and A<1:0>.

8.4 CY7C604 / CY7C605 Registers

This section describes the control and data registers for the CY7C604/605. All registers for the CY7C604 and CY7C605 are identical with the exception of the system control register (SCR). Sections or diagrams specific to the CY7C604 or CY7C605 are named with that part name only, whereas sections or diagrams common to both will be named using CY7C604/605.

All values in all control registers are read/write (with the exception of the implementation and version fields of the SCR). Control registers are accessible by use of the alternate space load or store instructions with ASI = 4. Please refer to *Section 8.8, ASI and Register Mapping*, for more information on register addressing.

Programmer's Note: To ensure software compatibility with future versions of the CY7C604/605, reserved fields in a register should be written as zeros and masked out when read.

8.4.1 CY7C604 System Control Register (SCR)

The system control register, as shown in *Figure 8–29,* defines the operation modes for the cache controller and MMU. Refer to *Section 8.2, MMU Operational Modes*, for additional information on the operation modes of the MMU. The following describes the functions of the bit fields in the SCR.

IMPL, VER The Implementation number (SCR<31:28>) and the version number (SCR<27:24>) fields are hardwired; they are read only fields and writes to those fields are ignored. The assignments for the CY7C604 these fields are:

Implementation number field: 0001
Version number field: 0000

MCA<1:0> *Multichip address field* (SCR<23:22>) provides the address field in multichip configuration. Refer to *Section 8.5* on Multichip Configuration for more information.

MCM<1:0> *Multichip maskfield* (SCR<21:20>) provides a masking facility to mask certain multichip address (MCA) bits in order to provide a facility to build systems with a different number of CY7C604s (from 1 to 4).

MV *Multichip configuration valid bit* (SCR(19)) indicates that the MCA and MCM fields are valid (see *Section 8.5, Multichip Configuration*).

BM *Boot-mode bit* (SCR(14)) indicates the system is in boot mode. This bit is set to 1 to indicate boot mode. This bit is automatically set upon power-on reset.

C *Cacheable bit* (SCR(13)) indicates whether the access is cacheable or not when the MMU is disabled (this bit is independent of the CE bit, see *Section 8.3.8, Cacheable/Non-cacheable Memory Accesses* for more details.) This bit is set to 1 if accesses on the physical bus (with the MMU disabled) are to be considered cacheable.

CM *Cache-mode bit* (SCR(10)) indicates whether the cache is operating under write-through no write allocate policy or copy-back write allocate policy. This bit is set to 1 to enable copy-back cache mode. Setting this bit to 0 will enable write-through cache mode.

CL *Cache-lock bit* (SCR(9)) indicates whether the entire cache is locked or not (see *Section 8.3.2.3* on Cache Lock, page 8-22). This bit is set to 1 to lock the cache.

CE *Cache-enable bit* (SCR(8)) indicates whether the virtual cache is enabled or not. This bit is set to 1 to enable the cache controller.

IMPL	VER	MCA	MCM	MV	RSV	BM	C	RSV	CM	CL	CE	RSV	NF	ME
31 28	27 24	23 22	21 20	19	18 15	14	13	12 11	10	9	8	7 2	1	0

IMPL = Specific Implementation of the MMU
VER = Version of Specific Implementation (typically mask revision)
MCA <0:1> = Multichip Address
MCM <0:1> = Multichip Mask
MV = Multichip Valid
BM = Boot Mode
C = Cacheable (when MMU disabled)
CM = Cache Mode
CL = Cache Lock
CE = Cache Enable
NF = No Fault
ME = MMU Enable
RSV = Reserved

Figure 8–29. CY7C604 System Control Register (SCR)

NF *No-fault bit* (SCR(1)) prevents supervisor data accesses from signaling data faults to the CY7C601. When the NF bit is set, exception-generating logic (in both the TLB and the table walk) does not indicate supervisor data faults to the CY7C601 (via $\overline{\text{MEXC}}$), but status and address information is recorded in the SFSR and SFAR registers as in normal data access operations. When the NF bit is not set, the CY7C604 reports the supervisor data exceptions.

ME *MMU-enable bit* (SCR(0)) indicates whether the MMU is enabled or not. This bit is set to 1 to enable the MMU.

Upon power-on reset, all writable control bits except the BM bit are cleared. This sets the CY7C604 into the following state: cache disabled (CE = 0), cache unlocked (CL = 0), write-through mode (CM = 0), non-cacheable (C = 0), boot-mode enabled (BM = 1), multichip disabled (MV = 0), no fault disabled (NF = 0), and MMU disabled (ME = 0).

8.4.2 CY7C605 System Control Register (SCR)

The System Control Register, as shown in *Figure 8–30*, defines the operation modes for the cache controller and MMU. Refer to page 8-15 for additional information on the operation modes of the MMU. The following describes the functions of the bit fields in the SCR.

IMPL, VER The implementation number (SCR<31:28>) and the version number (SCR<27:24>) fields are hardwired; they are read only fields and writes to those fields are ignored. The assignments for the CY7C605 are:

Implementation number field: 0001
Version number field: 1111

MCA<1:0> *Multichip address field* (SCR<23:22>) provides the address field in multichip configuration. Refer to *Section 8.5 Multichip Configuration* for more information.

MCM<1:0> *Multichip mask field* (SCR<21:20>) provides a masking facility to mask certain multichip address (MCA) bits in order to provide a facility to build systems with a different number of CY7C605s (from 1 to 4).

MV *Multichip configuration valid bit* (SCR(19)) indicates that the MCA and MCM fields are valid (see *Section 8.5, Multichip Configuration*).

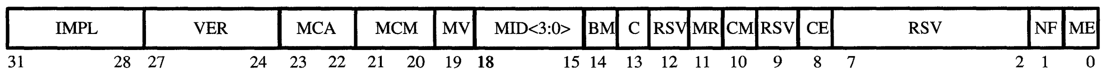
MID<3:0> *Module Identification number* (SCR<18:15>) identifies the processor module during transactions on the MBus (refer to *Chapter 11*). This four bit module identification number is embedded in the MBus address phase of all MBus transactions initiated by the CY7C605.

BM *Boot-mode bit* (SCR(14)) indicates the system is in boot mode. This bit is set to 1 to indicate boot mode. This bit is automatically set upon power-on reset.

C *Cacheable bit* (SCR(13)) indicates whether the access is cacheable or not when the MMU is disabled (this bit is independent of the CE bit, see *Section 8.3.8, Cacheable/Non-cacheable Memory Accesses* for more details.) This bit is set to 1 if accesses on the physical bus (with the MMU disabled) are to be considered cacheable.

MR *Memory Reflection* (SCR(11)) MR = 1 indicates that the main memory system on the MBus supports memory reflection. MR affects the status of the MPTAG cache tag bits as described in the cache state transitions section starting on page 8-26.

CM *Cache-mode bit* (SCR(10)) indicates whether the cache is operating under write-through no write allocate policy or copy-back write allocate policy. This bit is set to 1 to enable copy-back cache mode. Setting this bit to 0 will enable write-through cache mode.



IMPL = Specific Implementation of the MMU
VER = Version of Specific Implementation (typically mask revision)
MCA <1:0> = Multichip Address
MCM <1:0> = Multichip Mask
MV = Multichip Valid
MID <3:0> = Module Identifier <3:0>
BM = Boot Mode
C = Cacheable (when MMU disabled)
MR = Memory Reflection
CM = Cache Mode
CE = Cache Enable
NF = No Fault
ME = MMU Enable
RSV = Reserved

Figure 8–30. CY7C605 System Control Register (SCR)

CE *Cache-enable bit* (SCR(8)) indicates whether the virtual cache is enabled or not. This bit is set to 1 to enable the cache controller.

NF *No-fault bit* (SCR(1)) prevents supervisor data accesses from signaling data faults to the CY7C601. When the NF bit is set, exception-generating logic (in both the TLB and the table walk) does not indicate supervisor data faults to the CY7C601 (via $\overline{MEXC}$), but status and address information is recorded in the SFSR and SFAR registers as in normal data access operations. When the NF bit is not set, the CY7C604/CY7C605 reports the supervisor data exceptions.

ME *MMU-enable bit* (SCR(0)) indicates whether the MMU is enabled or not. This bit is set to 1 to enable the MMU.

Upon power-on reset, all writable control bits except the BM bit are cleared. This sets the CY7C604/CY7C605 into the following state: cache disabled (CE = 0), write-through mode (CM = 0), non-cacheable (C = 0), boot-mode enabled (BM = 1), memory reflection disabled (MR = 0), no fault disabled (NF = 0), and MMU disabled (ME = 0).

8.4.3 CY7C604/605 Context Table Pointer Register (CTPR)

The context table pointer points to the context table in physical memory. The table is indexed by the contents of the context register. The context table pointer appears on bits 35 through 14 of the MBus (MAD<35:14>) during the first fetch of TLB miss processing. Once the root pointer is cached in the page table pointer cache (PTPC), no fetching of the root pointer is required until the context is changed (see *Figure 8–31).*

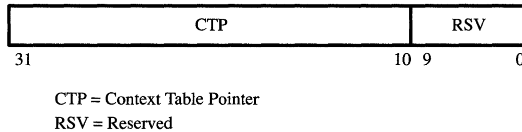


Figure 8–31. CY7C604/605 Context Table Pointer Register

8.4.4 CY7C604/605 Context Register (CXR)

The context register defines a virtual address space associated with the current process. The CXR is a twelve-bit register, which supports 4096 contexts. This register is used to define the current context for the CY7C604/605. Nearly all CY7C604/605 operations are dependent upon matching the value of this register to a cache tag entry or TLB entry.

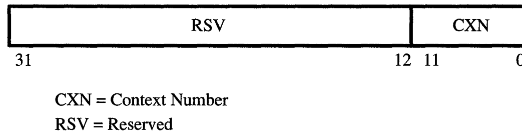


Figure 8–32. CY7C604/605 Context Register

8.4.5 CY7C604/605 Reset Register (RR)

The RR register contains information regarding whether watchdog reset (WDR) or software internal reset (SIR) occurred. This is a read/write register, and setting the software internal reset bit (SIR), or the software external reset bit (SER) in the case of the CY7C604, causes the corresponding reset. Upon power-on reset, the WDR and SIR bits in the RR will be cleared. Reading the RR will also clear these bits. Note that bit 0, the SER bit, can only be modified in the CY7C604; this bit is reserved in the CY7C605. For the CY7C605, this bit always reads "0," and writes to it are ignored. Refer to *Section 8.7, CY7C604/605 Reset* for more details on reset processing.

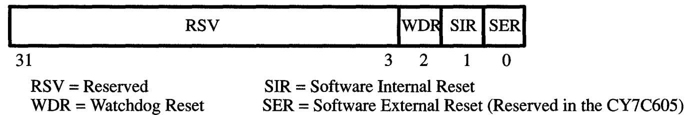


Figure 8–33. CY7C604/605 Reset Register

8.4.6 CY7C604/605 Root Pointer Register (RPR)

The RPR is the context level table page table pointer (PTP) and is cached in the page table pointer cache. Refer to *Section 8.1.5* on page 8-12 for information on the page table pointer cache.

On power-on reset, the V bit is cleared. When the current context is changed by writing to the context pointer register (CXR), the V bit of the RPR is cleared. The V bit is also cleared when the CTPR register is written.

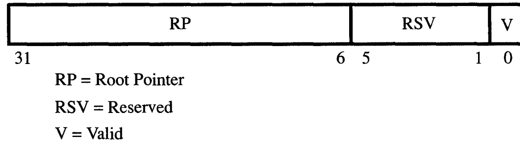


Figure 8–34. CY7C604/605 Root Pointer Register

8.4.7 CY7C604/605 Instruction access PTP (IPTP)

The IPTP is the instruction access level-2 table page table pointer (PTP) and is part of the page table pointer cache. On power-on reset, the V bit is cleared.

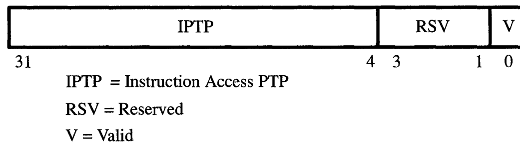


Figure 8–35. CY7C604/605 Instruction Access PTP Register

8.4.8 CY7C604/605 Data access PTP (DPTP)

The DPTP is the data access level 2 table page table pointer (PTP) and is a register in the page table pointer cache. On power-on reset, the V bit is cleared.

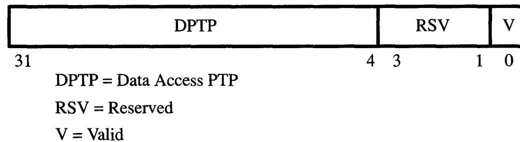


Figure 8–36. CY7C604/605 Data Access PTP Register

8.4.9 CY7C604/605 Index Tag Register (ITR)

The ITR contains the tag (index1 and index2) fields of the IPTP and DPTP entries. Refer to *Section 8.1.5* on page 8-12 for information on the PTP cache.

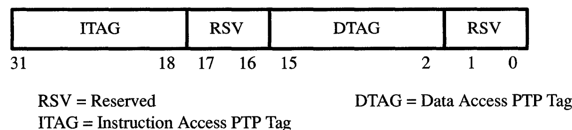


Figure 8–37. CY7C604/605 Index Tag Register

8.4.10 CY7C604/605 TLB Replacement Control Register (TRCR)

The TRCR contains the replacement counter (RC) and initial replacement counter (IRC) fields as shown in *Figure 8–38.* These fields are used in order to support random replacement and to support locking capabilities of the TLB. Refer to *Section 8.1.1.2* on page 8-7 for information on TLB entry locking. Upon power-on reset, both the RC and IRC fields are initialized to zero.

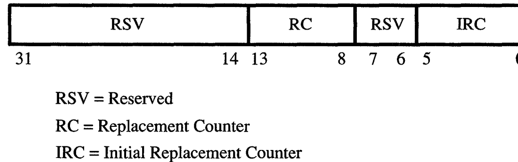


Figure 8–38. CY7C604/605 TLB Replacement Control Register

8.4.11 CY7C604/605 Synchronous Fault Status Register (SFSR)

The synchronous fault status register, illustrated in *Figure 8–39,* contains fault-associated information for synchronous faults. Synchronous faults are faults that occur during an integer unit access of memory. Synchronous faults include almost all possible faults for the CY7C604/605. This type of fault is synchronous to the operations of the CY7C601. For the CY7C604/605, this fault type covers all cases except those caused by delayed writes of data stored in the write buffers. These faults are asynchronous to the operation of the CY7C601, and are named asynchronous faults.

An example of a synchronous fault is a privilege violation fault caused by attempting an unauthorized memory access. These faults are discussed in detail in *Section 8.9.* Upon encountering a synchronous fault, the CY7C604/605 asserts the MEXC signal, along with MHOLD and MDS. Synchronous faults are the only exception type that assert the MEXC signal.

In the CY7C604, the copy-back translation error (CBT) bit indicates that a translation error occurred during a table walk for the flush of a modified cache line of a copy-back mode cache miss. The SFAR contains the address of the missed cache access, not the modified cache line address that caused the translation error. When this type of error occurs, the cache tag remains valid, and the cache line remains modified. Note that this bit is not used in the CY7C605, and is reserved. The physical address for a cache line is always available in the CY7C605, therefore making the CBT bit unnecessary in a CY7C605 based system.

The uncorrectable error (UE), timeout error (TO), and bus error bits (BE) report error status as encoded in the MERR, MRTY, and MRDY signals. (Refer to *Chapter 11* on MBus for further information.) The level bits (L) describe the level in a table walk process at which the fault occurred (if applicable). These bits are described in *Table 8-17* on page 8-58.

The access type bits (AT<2:0>) describes the access type that caused the fault. This field specifies user/supervisor access and whether the access is load or store of data or instruction. The AT bits are described in *Table 8-18* in the section on synchronous faults. The fault type bits (FT) describe the fault type, and are illustrated in *Table 8-19* on page 8-58. The fault address valid bit is set when the address in the synchronous fault address register (SFAR) is a valid fault address. The over-write bit (OW) is set in the case of a double fault where the fault status stored in the SFSR does not correspond with the fault first trapped on by the CY7C601. This is discussed in detail in the section on synchronous faults, page 8-56.

Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.

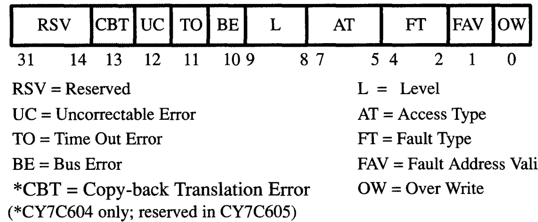


Figure 8–39. CY7C604/605 Synchronous Fault Status Register

8.4.12 CY7C604/605 Synchronous Fault Address Register (SFAR)

The synchronous fault address register contains the faulted virtual address.

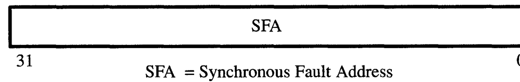


Figure 8–40. CY7C604/605 Synchronous Fault Address Register

8.4.13 CY7C604/605 Asynchronous Fault Status Register (AFSR)

Asynchronous faults are those faults caused by a delayed memory access initiated by the CY7C604/605. This type of error can only be caused by a delayed write to main memory initiated by the write buffer. Asynchronous faults cause the $\overline{\text{CMER}}$ signal to be asserted, which can be used as an interrupt to the CY7C601.

The UC, TO, and BE bits are identical to those in the SFSR. They are set by the information encoded into the $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MRDY}}$ signals of the MBus (see *Chapter 11*). The asynchronous fault address bits provide the upper four bits of the physical address not captured in the asynchronous fault address register (AFAR), which is a 32-bit register.

The asynchronous fault occurred (AFO) bit is set when an asynchronous fault is encountered. Once the asynchronous fault occurred bit is set, no further asynchronous faults are recorded until the AFO bit is cleared, which is accomplished by reading the asynchronous fault address register (see *Figure 8–41).* The UC, TO, BE, and AFO bits in the AFSR will be cleared upon power-on reset. Reading the AFAR will also clear these bits.

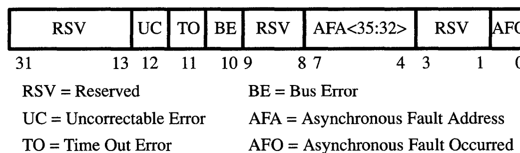


Figure 8–41. CY7C604/605 Asynchronous Fault Status Register

8.4.14 CY7C604/605 Asynchronous Fault Address Register (AFAR)

The AFAR contains bits 31 through 0 of the physical address for asynchronous faults (bus errors). Asynchronous faults can occur during delayed write accesses or during background cache line flush operations in copy-back mode (see *Figure 8–42).* If a bus error occurs during a write to memory, the $\overline{\text{CMER}}$ signal will

be asserted by the CY7C604/605 and will remain asserted until the AFAR register is read by software. The address in the AFAR is concatenated with the four AFA bits in the AFSR to define the entire 36-bit physical address.

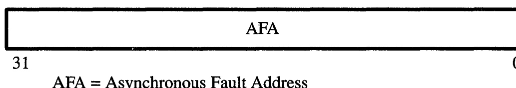


Figure 8–42. CY7C604/605 Asynchronous Fault Address Register

8.5 CY7C604 / CY7C605 Multichip Configuration

The CY7C604/605 is designed to allow expansion of the 64-Kbyte cache by adding additional CY7C604/605s, each controlling two CY7C157 Cache Storage Units. Up to four CY7C604/605s (for up to 256-Kbyte of cache) can be supported by a single CY7C601. A system using an expanded cache is required to configure the CY7C604/605s for multichip operation. Multichip operation is defined by the multichip address field (MCA<1:0>), multichip mask field (MCM<1:0>), and the multichip valid bit (MV) of the system control register (SCR). The two bit MCA and MCM fields control the addresses to which the CY7C604/605 is allowed to respond. The multichip valid bit enables the multichip mode for the CY7C604/605, and is to be set when the MCA and MCM fields are configured for the system.

System initialization under multichip operation mode is handled by designating one of the CY7C604/605s to respond to all addresses from the CY7C601 until the CY7C604/605s have been initialized. This CY7C604/605 is referred to as the boot mode CY7C604/605. The other CY7C604/605s remain inactive until multichip operation has been set.

8.5.1 System Initialization

The boot mode CY7C604/605 is responsible for accesses to memory during system initialization. The boot mode CY7C604/605 responds to all memory accesses until multichip operation is enabled by setting the multichip fields of the SCR. The other CY7C604/605s remain inactive for all memory accesses until their SCR has been enabled for multichip mode. The non-boot mode CY7C604/605s three-state $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ (in the CY7C605, $\overline{\text{IOE}}$ is also tri-stated).

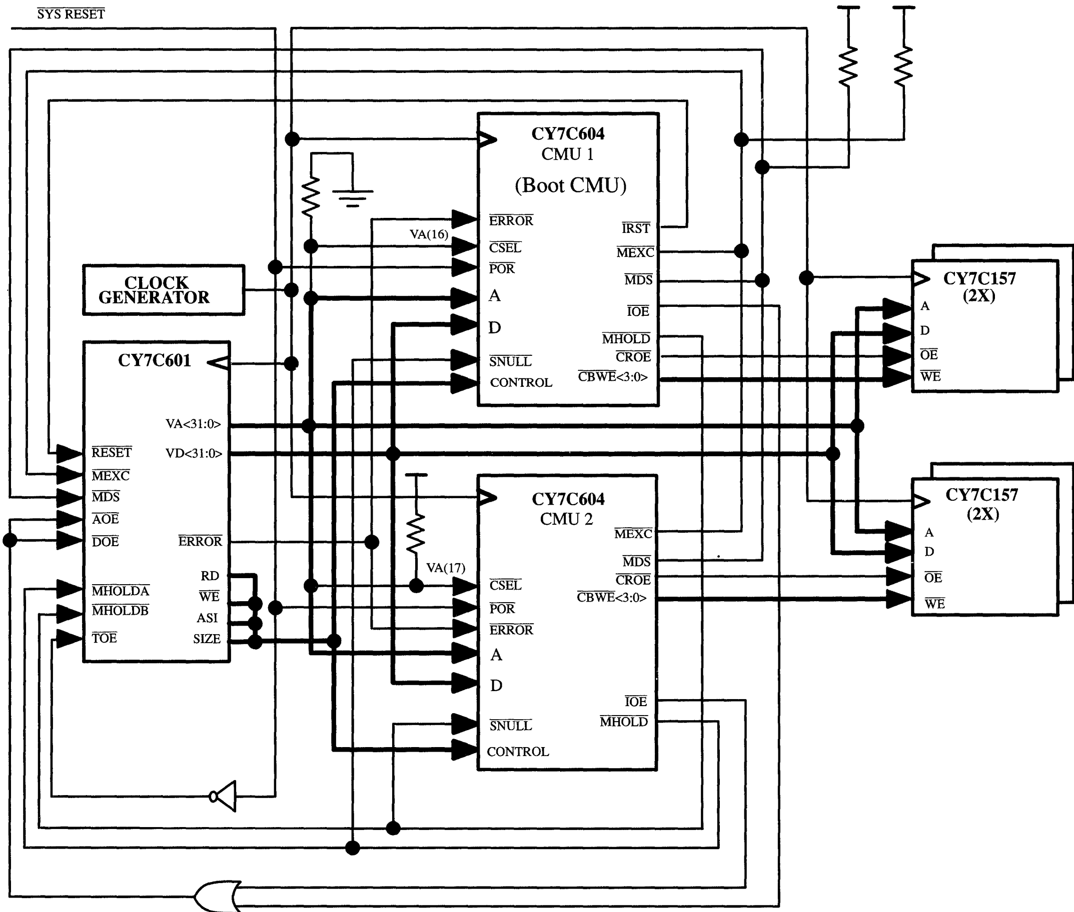


Figure 8–43. Dual–CY7C604 Multichip Configuration

The boot mode CY7C604/605 is selected by forcing LOW the $\overline{\text{CSEL}}$ signal as the power-on reset ($\overline{\text{POR}}$) signal is deasserted. The remaining CY7C604/605s are connected such that the $\overline{\text{CSEL}}$ signals are forced HIGH when the $\overline{\text{POR}}$ signal is deasserted. Each CY7C604/605 latches the state of its $\overline{\text{CSEL}}$ signal upon rising clock edge after $\overline{\text{POR}}$ is deasserted, and remains in either boot mode or becomes inactive until the multichip fields of its SCR have been set. (See $\overline{\text{CSEL}}$ power-on reset timing diagrams in the CY7C600 Electrical and Mechanical Specification document.) A single CY7C604/605 system should tie the $\overline{\text{CSEL}}$ signal to ground to ensure correct operation upon reset.

After reset, the $\overline{\text{CSEL}}$ signal of each CY7C604/605 is tied to one of the upper virtual address signals, thereby mapping each CY7C604/605 to a different virtual address. The CY7C604 and CY7C605 differ slightly in how $\overline{\text{CSEL}}$ and the multichip control bits in the SCR are used to initialize and access the registers.

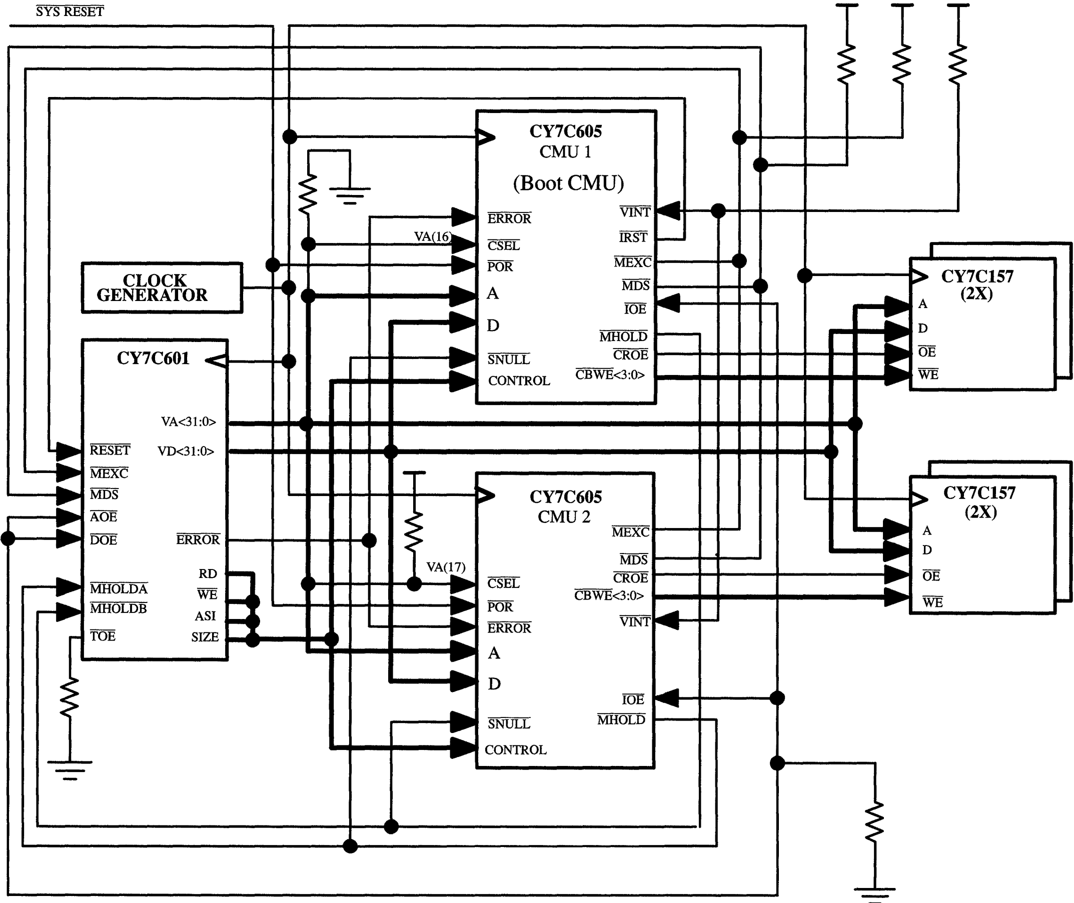


Figure 8–44. Dual–CY7C605 Multichip Configuration

The non-boot CY7C604s will ignore all register accesses except those to the SCR with $\overline{\text{CSEL}}$ asserted, until the multichip mode is enables for the CY7C604. Conversely, the boot CY7C604 will remain inactive only if $\overline{\text{CSEL}}$ is deasserted and the access is a register read/write to the SCR. The $\overline{\text{CSEL}}$ signal is ignored by the CY7C604 after the multichip fields in the SCR are initialized. Instead, after the MV bit has been set, the CY7C604 is selected by the MCA and MCM fields in the SCR as defined in Figure 8–45.

The CY7C605 will respond to register or diagnostic accesses only if the $\overline{\text{CSEL}}$ pin asserted during the access, whether it is the boot CY7C605, the non-boot CY7C605, and even after the SCR is written to enable multichip mode. This allows the same diagnostic programs to be used independently of whether multichip mode is enabled or not. The register and diagnostic accesses include ASIs 0x04, 0x06, 0x0E, and 0x0F. Only the boot CY7C605 will respond to other ASIs before the MV bit is set. After the MV bit is set, the CY7C605 is selected for other ASIs by the MCA and MCM fields in the SCR as defined in Figure 8–45.

The multichip fields of the SCR for the non-boot mode CY7C604/605s should be configured and enabled before the SCR for the boot mode CY7C604/605 is enabled. This prevents problems with the boot mode CY7C604/605 interfering during the configuration of the non-boot mode CY7C604/605s.

8.5.2 Cache Configurations

Figure 8–43 illustrates a 128-Kbyte cache using two CY7C604s in a multichip configuration. Note that VA<16> of the virtual address is connected to the $\overline{\text{CSEL}}$ input of CMU1 and is pulled to ground with a resistor. This signal is used to access the CMU1 registers before multichip operation has been enabled. Using a pull-down resistor also accomplishes the task of forcing the $\overline{\text{CSEL}}$ signal for CMU1 to low, which is latched on the rising clock edge after $\overline{\text{POR}}$ is deasserted to enable the CY7C604/605 as the boot mode CMU. VA<17> is connected to the $\overline{\text{CSEL}}$ input for CMU2. This signal is pulled up with a resistor to ensure that it is forced HIGH when the system reset signal is released. The virtual address bus VA<31:0> is three-stated by using the system reset signal to drive $\overline{\text{TOE}}$ HIGH, thereby forcing the CY7C601 off the address bus.

Figure 8–44 shows a dual CY7C605 configuration. The CY7C605 in multichip mode differs slightly from the CY7C604 in multichip due to the addition of the $\overline{\text{VINT}}$ signal and change to the $\overline{\text{IOE}}$ signal in the CY7C605. $\overline{\text{IOE}}$ changes from a bi-state on the CY7C604 to a tri-state signal on the CY7C605, and is driven on power-on reset to tri-state the drivers of the address bus (A<31:0>), data bus (D<31:0>), and address space identifiers (ASI<7:0>) on the CY7C601. Although all of the CY7C605s in the system share a common $\overline{\text{IOE}}$ signal, as well as a common $\overline{\text{MDS}}$, only one CY7C605 can drive these signals at a time. $\overline{\text{VINT}}$ is used by a CY7C605 to gain access to the virtual bus in response to a Coherent Read (or Coherent Read and Invalidate) transaction for a cache line it owns. See *Sections 8.10* and *8.11* for more information on $\overline{\text{VINT}}$.

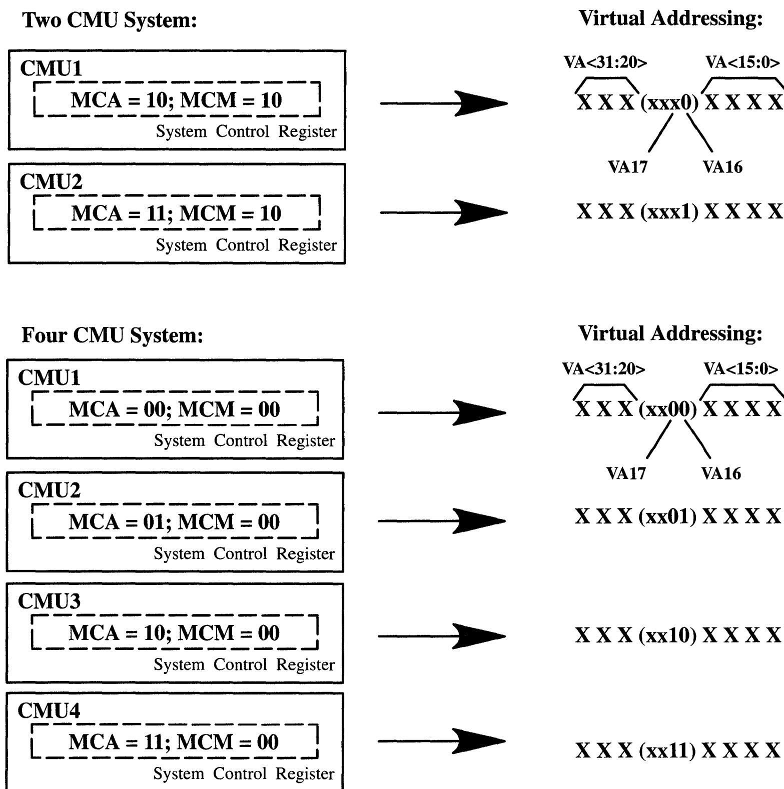


Figure 8–45. Examples of Multichip Addressing

The SNULL input signal causes the CY7C604/605 to ignore an address on the virtual address bus. This input is used in multichip operation to keep a CY7C604/605 from responding to addresses output on the virtual address bus by other CY7C604/605s. The MHOLD output signal from a CY7C604/605 is used as the SNULL input for the remaining CY7C604/605s. *Figure 8–43* and *Figure 8–44* illustrate the MHOLD to SNULL connections for a two-CY7C604/605 system.

The multichip address bits (MCA<1:0>) of the system control register (SCR) select the state of the VA<17:16> bits that must be matched for multichip addressing. The multichip mask bits (MCM<1:0>) select which of the VA<17:16> bits can be ignored. The combination of the two fields define the address mapping for the CY7C604/605. The multichip valid bit (MV) must be set when writing to the MCA and MCM fields in order to enable multichip mode. *Figure 8–45* illustrates two examples of how these fields are used to define the address mapping for multiple CY7C604/605 systems.

8.6 CY7C604/605 Diagnostic Support

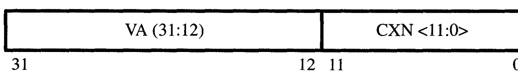
8.6.1 CY7C604/605 MMU TLB Entries

TLB entries can be accessed with a load or store alternate instruction with the TLB entry address and ASI = 6H. This feature is supported for diagnostic purposes and to provide CY7C601 access to locked TLB entries. The virtual and physical sections of each entry in the TLB can be accessed by the CY7C601 as a single-word read or write. The address mapping for the TLB entries is shown in *Table 8-11.* The format of CAM word and RAM word entries in the TLB are shown in *Figure 8–46.*

Table 8-11. TLB Entry Address Mapping

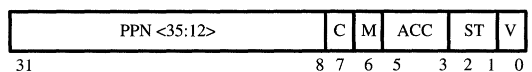
ADDRESS	TLB ENTRY REGISTER
0 H	Entry 0 RAM Word
4 H	Entry 0 CAM Word
8 H	Entry 1 RAM Word
C H	Entry 1 CAM Word
10 H	Entry 2 RAM Word
14 H	Entry 2 CAM Word
• • •	• • •
1F0 H	Entry 62 RAM Word
1F4 H	Entry 62 CAM Word
1F8 H	Entry 63 RAM Word
1FC H	Entry 63 CAM Word
200-FFFFFFF8 H	Reserved

TLB Entry CAM Word Format



VA = Virtual Address
CXN = Context Number

TLB Entry RAM Word Format



PPN = Physical Page Number
C = Cacheable bit
M = Modified bit
ACC = Access protection bits
ST = Short Translation Type
V = Valid

Figure 8–46. TLB Entry Format

Table 8-12. Cache Tag Entry Address Mapping

Address	Cache Tag Entry
000x H	0
002x H	1
004x H	2
006x H	3
• • •	• • •
FFEx H	2047

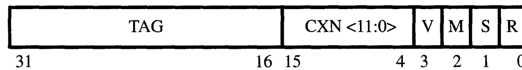
(x = don't care)

8.6.2 CY7C604/605 Cache Tag Entries

CY7C604 tag entries are accessed using a load or store alternate instruction with the cache tag entry address and ASI = 0E H. The CY7C605 PVTAG is accessed using a load or store alternate instruction specifing the entry address and ASI = 0E H. CY7C605 MPTAG entries are accessed in a similar manner using ASI = 30 H. Each tag entry can be read as a load single or can be written as a store single from the CY7C601. The address mapping for the cache tag entries is shown in *Table 8-12*. The format of a CY7C604 tag entry is shown in *Figure 8–47*. The CY7C605 PVTAG and MPTAG entry formats are illustrated in *Figure 8–48*.

8.6.3 CY7C604/605 Cache Data Entries

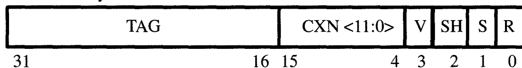
Cache data entries can be accessed from the cache RAM by using a load or store alternate instruction asserting the virtual address and ASI = 0F H. The CY7C604/605 cache controller causes a forced hit from the cache tag during these accesses. All data widths are supported for a read or write to the cache RAM.



TAG = Virtual Address Tag
CXN = Context Number
V = Valid bit
M = Modified bit
S = Supervisor
R = Reserved

Figure 8–47. CY7C604 Cache Tag Entry Format

PVTAG Entry



TAG = Virtual Address Tag
CXN = Context Number
V = Valid bit
SH = Shared
S = Supervisor
R = Reserved

MPTAG Entry

TAG = Physical Address Tag
V = Valid
SH = Shared
M = Modified
R = Reserved

Figure 8–48. CY7C605 Cache Tag Entry Format

8.7 CY7C604/605 Reset

8.7.1 Power-On Reset (POR)

Upon power-on reset, the entire system is forced into a defined state. The TLB and the cache tag(s) in the CY7C604/605 are invalidated, all valid bits in control registers are cleared, and certain bits in the AFSR and SFSR are cleared as described in the previous sections. The CY7C604 asserts IRST to the integer unit until POR is deasserted; the CY7C605 holds IRST low for an extra clock cycle, releasing it one cycle after IOE is driven low. Since INULL is asserted until after IRST is deasserted, MAS (or MBR in the case of arbitration for the bus) can be generated one cycle earlier in the CY7C604 than the CY7C605. MRST is not asserted in the case of the CY7C604. POR must be asserted for a minimum of 8 clocks. The bits in the reset register (RR) are cleared. Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. The SCR fields in the CY7C604/605 will have the following state after a power-on reset:

Table 8-13. CY7C604/605 Power-On Reset States

IMPL	Unchanged
VER	Unchanged
MCA<1:0>	Unchanged
MCM<1:0>	Unchanged
MV	0
BM	1
C	0
CM	0
CL	0
CE	0
NF	0
ME	0
MR	0
MID<3:0>	FH (605 only)

8.7.2 Watchdog Reset (WDR)

When the CY7C601 encounters a trap while traps are disabled, the CY7C601 enters into an error state, asserts the ERROR signal, and then halts. The only way to restart the CY7C601 in the error state is to assert its RESET signal. The CY7C604/605 does this by performing a watchdog reset, which asserts the IRST signal for 1024 clock cycles. MRST is not asserted. The TLB and the cache tag(s) in the CY7C604/605 are not invalidated. The WDR (RR[2]) bit in the RR register is set. All SCR fields except boot mode (BM) are unchanged. BM is set to 1 after a watchdog reset.

8.7.3 Software Internal Reset (SIR)

The operating system can reset the CY7C601 by setting the SIR bit in the reset register. The CY7C604/605 asserts IRST for 1024 clock cycles to reset the CY7C601. The TLB and the cache tag are not invalidated. All SCR fields except BM are unchanged, and BM is set to 1 after a software internal reset. The contents of the reset register are unchanged and the SIR bit will remain set. Refer to page 8-98 for timing diagrams for the SIR and SER resets.

8.7.4 Software External Reset (SER) (CY7C604 only)

The operating system can reset the system separately from the CY7C601 by writing 1 into the SER bit of the RR register of the CY7C604 only. (This bit is “reserved” in the CY7C605; writes to it are ignored.) Only the writing of a 1 into the SER bit will cause $\overline{\text{MRST}}$ to be asserted. The CY7C604 asserts $\overline{\text{MRST}}$ for 1024 MBus clock cycles to reset the system. The TLB and the cache tag are not invalidated. The SCR register remains unchanged. The CY7C604 will wait for its write buffers to empty before asserting $\overline{\text{MRST}}$ on a software external reset. The contents of the reset register are unchanged and the SER bit will remain set.

$\overline{\text{MRST}}$ will not be asserted on a software external reset (refer to page 8-54) until the write buffers have been flushed. Writing both the SIR and SER bits in the reset register will cause the assertion of both $\overline{\text{IRST}}$ and $\overline{\text{MRST}}$. A reset routine can poll the reset register to determine the source of any reset.

8.7.5 CY7C604/605 Reset in Multichip Configuration

In a multichip configuration, the CY7C604/605 that is responsible for handling boot mode can also assume the responsibility to handle the reset operations described above. The $\overline{\text{IRST}}$ to the CY7C601 and the $\overline{\text{MRST}}$ to the external system are connected only to this responsible CY7C604/605. The reset signals from the other CY7C604/605s are not connected. The $\overline{\text{ERROR}}$ pin of the CY7C601 should be connected to all CY7C604/605s thereby putting all CY7C604/605s in the same state during watchdog reset. Only the $\overline{\text{IRST}}$ of the boot-handling CY7C604/605 is connected to the $\overline{\text{RESET}}$ input of the CY7C601.

When performing a software internal reset in a multichip configuration, the reset register SIR bit should be set in all the non-boot-handling CY7C604/605s before SIR is set in the boot-handling CY7C604/605. This places all CY7C604/605s contained in the system in the same mode before the CY7C601 is reset. A software external reset in a uniprocessing multichip configuration can be performed in by writing the SER bit in the boot-handling CY7C604 only. It is not necessary to alter the non-boot-handling CY7C604s.

8.8 CY7C604/605 ASI and Register Mapping

The CY7C604/605 uses the address space identifier bus (ASI <5:0>) to provide access by the CY7C601 to internal registers and resources, such as the cache tag and the TLB. The CY7C604/605 also uses the ASI bus to map restricted memory access functions, such as local and pass-through memory addressing modes. Register access to the CY7C604/605 requires using a load or store alternate instruction with ASI = 04 H in addition to the register address, given in *Table 8-14*. *Table 8-15* illustrates the ASI mapping for the CY7C604/605.

Table 8-14. CY7C604/605 Register Address Mapping

VA<15:8>	CY7C604/605 Registers	VA<15:8>	CY7C604/605 Registers
0 H	System Control Register (SCR)	8 H - F H	Reserved
1 H	Context Table Pointer Register (CTPR)	10 H	Root Pointer Register (RPR)
2 H	Context Register (CXR)	11 H	Instruction Access PTP (IPTP)
3 H	Synchronous Fault Status Register (SFSR)	12 H	Data Access PTP (DPTP)
4 H	Synchronous Fault Address Register (SFAR)	13 H	Index Tag Register (ITR)
5 H	Asynchronous Fault Status Register (AFSR)	14 H	TLB Replacement Control Register (TRCR)
6 H	Asynchronous Fault Address Register (AFAR)	15 - FF H	Reserved
7 H	Reset Register (RR)		

Table 8-15. Standard ASI Assignments

ASI	Function	ASI	Function
0 H	Reserved	12 H	Flush combined cache line (region)*
1 H	MBus extended address space*	13 H	Flush combined cache line (context)*
2 H	Reserved	14 H	Flush combined cache line (user)*
3 H	MMU flush/probe*	15 H	Reserved
4 H	MMU registers*	16 H	Reserved
5 H	MMU diagnostics instruction only TLB	17 H	Block copy
6 H	MMU diagnostics instruction/data TLB*	18 H	Flush data cache line (page)
7 H	MMU diagnostics I/O TLB	19 H	Flush data cache line (segment)
8 H	User instruction*	1A H	Flush data cache line (region)
9 H	Supervisor instruction*	1B H	Flush data cache line (context)
A H	User data*	1C H	Flush data cache line (user)
B H	Supervisor data*	1D H	Reserved
C H	Cache tag for instruction cache	1E H	Reserved
D H	Cache data for instruction cache	1F H	Block zero
E H	Cache tag for combined (inst/data) cache*	20-2F H	MMU bypass physical address*
	(PVTAG if VA<18>=0, MPTAG if VA<18>=1)**	30-3F H	Unassigned
F H	Cache data for combined cache*	40–6F H	Reserved
10 H	Flush combined cache line (page)*	70-7F H	Unassigned
11 H	Flush combined cache line (segment)*	80-FF H	Reserved

* indicates functions supported by the CY7C604 and CY7C605

** indicates function is specific to the CY7C605

8.9 Synchronous Faults

Synchronous faults are grouped into three classes: instruction access faults, data access faults, and translation table access faults. The translation table access faults are further divided into translation instruction access faults and translation data access faults. The SPARC architecture causes the timing and priority of these fault classes to be handled differently. Due to delays caused by the instruction pipeline, the CY7C601 can possibly encounter a second fault before the CY7C601 enters a trap to correct the first. Depending upon the class of fault encountered, the status and address of a fault may be allowed to overwrite information for a previous fault that has not yet generated a trap. This potential condition requires a trap handler that can correct the various combinations of fault conditions. This section describes these potential fault conditions.

The case of a pair of faults occurring presents a problem in reporting the correct fault status. This problem is solved by use of an overwrite (OW) bit in the SFSR and by prioritizing which types of faults may overwrite a previous fault. The OW bit signals the trap handler that the status and address stored in the fault registers are not valid for the trap that the CY7C601 has entered. The SFSR logic sets the OW bit according to a state sequence based on the fault handling of the CY7C601 and the type of faults encountered.

Since the CY7C601 delays entering a trap handler for an instruction fault, a trap caused by another fault will overwrite the trap information for the initial instruction fault. If the second fault causes a trap in the CY7C601 before the initial instruction fault trap is entered, the OW bit is not set. This is because the information in the fault registers will be correct for the first trap reading the registers. However, if the initial instruction trap is entered before the second fault trap is entered, the OW bit will be set. This is because the first trap reading the fault status registers will have the fault data for the second trap. The OW bit is set only

if the trap that will be executed first by the CY7C601 does not match the status information stored in the SFSR. The setting of the OW bit is entirely based upon the types of faults and their order of occurrence. *Table 8-16* illustrates the possible fault cases and their effect on OW.

Table 8-16. OW Bit States

First Fault	Second Fault	Update SFSR	OW
single fault		yes	0
instruction	instruction	yes	1
instruction	data	yes	0
instruction	translate instr.	yes	1
instruction	translate data	yes	0
data	instruction	no	0
data	data	yes	1*
data	translate	yes	1
translate	instruction, data	no	0
translate	translate	no	0

* not possible with CY7C601 (and related processors)

The CY7C601 delays a trap caused by an instruction access fault until that instruction reaches the Execute stage. However, since data accesses are not pipelined, the CY7C601 jumps to a trap immediately upon encountering a data access fault.

Faults are allowed to overwrite another fault status dependent upon priority. An instruction fault is allowed to overwrite only another instruction fault. It is not allowed to overwrite either a data fault or a translation fault. Data faults may overwrite an instruction fault, but not a translation fault. Data faults cannot overwrite another data fault, since the CY7C601 traps immediately upon encountering a data fault. Translation faults may overwrite any type of fault, but cannot be overwritten. Translation faults may not overwrite another translation fault.

All double fault cases are recoverable by re-executing the instruction or access that caused the fault whose status has been overwritten. If an instruction access fault occurs and the OW bit is set, the system software must determine the cause by probing the MMU and/or memory.

Upon encountering a synchronous fault, the SFSR records the bus error status (bus error, timeout, and uncorrectable error) when a bus error occurs during memory accesses. The level field (L), as shown in *Table 8-17*, is set to the page table level of the entry that caused the fault, if the fault is associated with a table walk. The access type (AT) field, illustrated in *Table 8-18*, defines the type of access that caused the fault. The fault type field FT (see *Table 8-19*) defines the type of the current fault.

A translation table access fault (FT = 4) occurs if an MMU page table access causes an external system error. This also occurs if a reserved entry type (ET = 3 in the PTE) is found in any level of the table walk. A translation table access fault (FT = 4) also can occur if a PTP (page table pointer) is found in level 3, instead of a PTE. If the page table entry is invalid (ET = 0 in the PTE), the fault type is an invalid address error (FT = 1). *Table 8-20* illustrates the fault type (FT) assigned for valid TLB entries or PTE entries (ET = 2) that cause a fault condition. These fault conditions are always either a protection error (read/write of data or instruction) or a privilege violation (user/supervisor access) error.

The copy-back translation fault bit (CBT) is set if there is an error occurring during a table walk for a modified cache line replacement or during a modified cache line flush operation. The fault address valid bit (FAV) is set to one if the content of the synchronous fault address register is valid. The SFAR may not be valid for instruction faults. The SFAR is always valid for data faults and translation errors.

If multiple fault types apply to the same fault occurrence, the highest priority fault is recorded. The highest fault priority is a translation fault (priority 2), as shown in *Table 8-21*. Priority 1 is reserved for an internal fault.

Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.

Table 8-17. Fault Register Level Field

L	Level
0	Entry in Context Field
1	Entry in Level 1 Table
2	Entry in Level 2 Table
3	Entry in Level 3 Table

Table 8-18. Fault Register Access Type Field

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

Table 8-19. Fault Register Fault Type Field

FT	Fault Type
0	None
1	Invalid Address Error
2	Protection Error
3	Privilege Violation Error
4	Translation Error
5	Bus Access Error
6	Not Generated
7	Reserved

Table 8-20. Fault Type (FT) for PTE[ET] = 2

AT	ACC							
	0	1	2	3	4	5	6	7
0	0	0	0	0	2	0	3	3
1	0	0	0	0	2	0	0	0
2	2	2	0	0	0	2	3	3
3	2	2	0	0	0	2	0	0
4	2	0	2	0	2	2	3	3
5	2	0	2	0	2	0	2	0
6	2	2	2	0	2	2	3	3
7	2	2	2	0	2	2	2	0

Table 8-21. Fault Register Error Priorities

Priority	Error
1	Internal Error
2	Translation Error
3	Invalid Address Error
4	Privilege Violation Error
5	Protection Error
6	Bus Access Error

8.9.1 Synchronous Fault Cases

The following seventeen cases describe the combinations of fault cases that can occur:

Case 1: *Instruction fault with no further faults.* The CY7C601 trap is delayed until the CY7C601 tries to execute the instruction.

The trap is taken immediately if the instruction access is actually a data access that is interpreted by the CY7C604/605 as an instruction access due to asserting ASI = 8 or 9 with a load alternate instruction. In this case, the trap handlers cannot probe main memory using the PC of the instruction. If the instruction is a load alternate instruction, the trap handler has to calculate the effective address to probe. The SFAR has the valid address if the OW bit is not set.

Case 1: Single-Instruction Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus access error occurred (external bus error: UC or TO or BE is set).
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 2: *Multiple instructionfault.* Instruction fault (1) followed by another instruction fault (2) will always cause at least one more instruction fault. The CY7C601 traps on instruction fault (1).

If the latest instruction fault is due to a load alternate access with ASI = 8 or 9), it overwrites the fault associated information of any previous fault. In this case, the setting of the FAV bit in the SFAR indicates a valid address for the latest fault instruction.

The fault address of fault (1) can be obtained from the PC in the CY7C601 for the trap handler with the exception of the following case.

It is possible that a data access may be interpreted by the CY7C604/605 as an instruction access because of the use of a load or store alternate instruction with ASI = 8, 9. Before the CY7C601 takes the trap on the data access fault (which is recorded as an instruction fault in the CY7C604/605), another instruction fault may occur. The second instruction fault will overwrite the data access fault information because it is recorded as an instruction fault in the CY7C604/605. In this case, the trap handler cannot just probe the PC of the instruction. If the instruction is a load alternate instruction, the trap handler must calculate the effective address to probe, and the SFAR will not contain the fault address of the data access fault.

Case 2: Multiple-Instruction Fault		
OW	1	
FAV	1	SFAR has valid address for latest fault
FT	1,2,3,5	Fault type of latest fault
AT	2,3	Access type of latest fault
L	0,1,2,3	Level of table walk at which latest fault (2) occurred (only valid with FT = 1)

Case 3: *Single Data fault.* CY7C601 trap (taken immediately).

Case 3: Single Data Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus error occurred (external bus error, UC or TO or BE is set)
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 4: *Instruction fault followed by data fault.* CY7C601 traps on the data fault

The history of the instruction fault is lost, but the same fault can be obtained again, once the return from the trap handler of the data fault is completed.

Case 4: Instruction Fault then Data Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1,2,3,5	Fault type of data fault
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 5: *Data fault followed by instruction fault.* The instruction fault cannot overwrite the data fault. The instruction fault will occur again, once the return from the data fault trap handler is completed. CY7C601 will trap on data fault.

Case 5: Data Fault then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1,2,3,5	Fault type of data fault
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 6: *Data fault followed by data fault.* (not possible with CY7C601.)

Case 7: *Translation fault (instruction access); no further faults.* The CY7C601 trap is delayed until the CY7C601 tries to execute the instruction or is taken immediately if the access is data due to a load alternate instruction.

Case 7: Translation Fault on Instruction Access		
OW	0	
FAV	1	SFAR has valid address for translation fault.
FT	4	Translation error occurred (bus error or ET = 3 or PTP in level 3 during table walk)
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 8: *Translation fault (data access).* The CY7C601 trap is taken immediately.

Case 8: Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	Translation error occurred (bus error or ET = 3 or PTP in level 3 during table walk)
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 9: *Instruction fault followed by translation fault (instruction.)* The CY7C601 traps on the instruction fault.

The fault address of the instruction fault can be obtained from the PC in the CY7C601 for the trap handler with the exception of the following case.

A data access fault can be recorded as an instruction fault if a load alternate instruction with ASI = 8, 9 causes a fault. Before the CY7C601 takes the trap on the data access fault (which is recorded as an instruction fault in the CY7C604/605), a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 9: Instruction Fault then Translation Fault (I)		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 10: *Translation fault (instruction access) followed by instruction fault.* The CY7C601 traps on the translation fault. The instruction fault cannot overwrite the translation fault.

Case 10: Translation Fault (I) then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 11: *Translation fault1 (instruction access) followed by translation fault2 (instruction).* The CY7C601 traps on translation fault1.

Case 11: Translation Fault (I) then Translation Fault (I)		
OW	0	
FAV	1	SFAR has valid address for first translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which first translation fault occurred during table walk

The second translation fault cannot overwrite the first translation fault.

Case 12: *Translation fault1 (instruction access) followed by translation fault2 (data access).* The CY7C601 traps on translation fault2. The translation fault2 cannot overwrite translation fault1.

Case 12: Translation Fault (I) then Translation Fault (D)		
OW	0	
FAV	1	SFAR has valid address for translation fault1
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault1 occurred during table walk

Case 13: *Translation fault (instruction access) followed by data fault.* The CY7C601 traps on the data fault. The data fault cannot overwrite the translation fault.

Case 13: Translation Fault (I) then Data Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 14: *Data fault followed by translation fault (instruction access).* The CY7C601 traps on the data fault.

Before the CY7C601 takes the trap on the data access fault, a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 14: Data Fault then Translation Fault (I)		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 15: *Instruction fault followed by translation fault (data).* The CY7C601 will trap on the data fault.

Case 15: Instruction Fault then Translation Fault (D)		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

Case 16: *Translation fault (data) followed by instruction fault.* The CY7C601 will trap on the data fault.

Case 16: Translation Fault (D) then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

Case 17: *Translation fault (data) followed by translation fault (instruction).* The CY7C601 will trap on the data fault.

Case 17: Translation Fault (D) then Translation Fault (I)		
OW	0	
FAV	1	SFAR has valid address for data translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

8.10 CY7C604/605 Pin Definitions

The functional pinouts for the CY7C604 and CY7C605 are shown in *Figure 8–49.* Note that all three-state output signals are driven to their inactive state before they are released to three-state. All signals described are common to both the CY7C604 and CY7C605 unless otherwise stated.

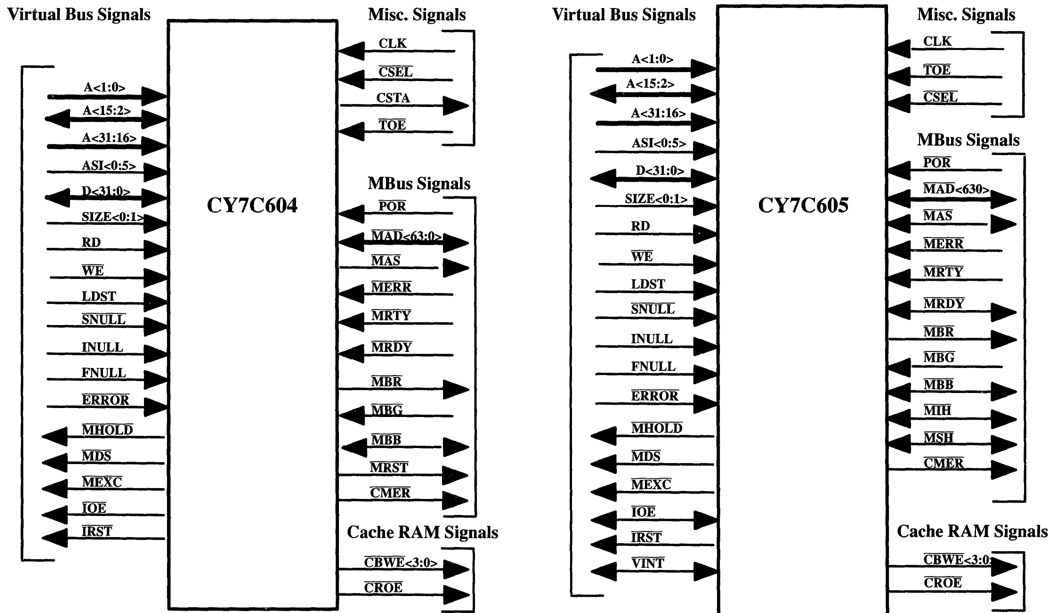


Figure 8–49. CY7C604 and CY7C605 I/O Signals

CY7C604/605 Virtual Bus Signals

Signal Name	I/O	Description
A<31:16>	I	Virtual Address bus. A<31:16> are input signals during normal read/write accesses and are latched into the CY7C604/605 on the rising edge of clock.
A<15:2>	I/O	Virtual Address bus. Three-state input/output signals. A<15:2> are input signals during normal read/write accesses and are latched into the CY7C604/605 on the rising edge of the clock. They are output signals during cache line loads into the CY7C157 and modified cache-line reads from the CY7C157.
A<1:0>	I	Virtual Address bus. A<1:0> are input signals during normal read/write accesses and are latched on the rising edge of clock.
ASI<5:0>	I	Address Space Identifiers. The ASI bits are used to: 1. Identify various types of accesses (user/supervisor, instruction/data) 2. Access CY7C604/605 registers 3. Initiate MMU flush/probe operation 4. Identify CACHE flush operations 5. Recognize diagnostic operations 6. Recognize pass physical address space
D<31:0>	I/O	Virtual Data bus. Three-state input/output signals. D<31:0> are input signals during CY7C601 normal write accesses, modified cache-line reads from the CACHE RAM, CY7C604/605 register writes or CY7C604/605 diagnostic accesses. They are output signals during cache line loads into CACHE RAM, CY7C604/605 register reads, non-cacheable loads, or CY7C604/605 diagnostic accesses.
$\overline{\text{ERROR}}$	I	Error (active LOW) signal from the CY7C601. When this signal asserted, it indicates the CY7C601 has halted due to entering the error state. The CY7C604/605 reads this signal and initiates a watchdog reset. (Refer to *Section 8.7.2* for more details.)
FNULL	I	Floating-point unit nullification cycle (active HIGH). When FNULL is active, the current access is ignored.
INULL	I	Integer unit nullification cycle (active HIGH). When INULL is active, the current access is ignored.
$\overline{\text{IOE}}$	O (604) I/O (605)	Integer unit output enable (active LOW). This signal is connected to the $\overline{\text{AOE}}$ and $\overline{\text{DOE}}$ inputs of the CY7C601. When deasserted (HIGH), the $\overline{\text{IOE}}$ will place the address (A<31:0>), address space identifiers (ASI<7:0>), and data (D<31:0>) drivers of the CY7C601 in a three-state condition. On the CY7C604, this signal is continually driven high or low, and is driven LOW during power-on reset.
(605 ONLY)		In the CY7C605, this signal is a three-state signal. During power-on reset, $\overline{\text{IOE}}$ is driven HIGH. This signal must be tied to ground through a resistor.

Signal Name	I/O	Description
IRST	O	Integer unit reset (active LOW) is asserted to reset the integer unit. (Refer to *Section 8.7.2* for more details.) This signal is continually driven HIGH or LOW.
LDST	I	Atomic Load-Store operation indicator (active HIGH). Asserted by the CY7C601 during atomic load store cycles and is sampled by the CY7C604/605 on the rising edge of the clock.
$\overline{MDS}$	O	Memory data strobe (active LOW) is asserted for one clock to strobe data into the CY7C601 during a cache miss. $\overline{MHOLD}$ must be low when $\overline{MDS}$ is asserted. It is driven off of the falling edge of the clock. This is a three-state output. This signal must be tied to V_{CC} through a resistor.
$\overline{MEXC}$	O	Memory exception (active LOW) is asserted for one clock whenever a privilege or protection violation is detected. $\overline{MHOLD}$ and $\overline{MDS}$ must be low when $\overline{MEXC}$ is asserted. This is a three-state output. This signal must be tied to V_{CC} through a resistor.
$\overline{MHOLD}$	O	Memory hold (active LOW) is asserted by the CY7C604/605 whenever it requires additional time to complete the current access such as during cache miss etc. It is driven off of the falling edge of the clock.
RD	I	Read cycle indicator (active HIGH). Asserted by the CY7C601 during read cycles and is sampled by the CY7C604/605 on the rising edge of the clock. This signal is also used to generate cache read output enable ($\overline{CROE}$).
SIZE(1:0)	I	SIZE of access indicator. Specifies the data width of the CY7C601 access and is sampled by the CY7C604/605 at the rising edge of the clock.
$\overline{SNULL}$	I	System nullification cycle (active LOW). When $\overline{SNULL}$ is active, the current access is ignored.
$\overline{WE}$	I	Write enable to indicate write cycle (active LOW). Asserted by the CY7C601 during write cycles and is sampled by the CY7C604/605 on the rising edge of the clock. This signal is also used to generate cache byte-write enables ($\overline{CBWE}$<3:0>).
$\overline{VINT}$ (605 ONLY)	I/O	Virtual interrupt (active LOW). This signal is a three-state signal asserted by a CY7C605 during a Coherent Read (or Coherent Read and Invalidate) transaction to respond to a request for a cache line it owns. When operating in a multiprocessor systems, assertion of $\overline{VINT}$ causes the other CY7C605(s) in the system to stop driving the address and data buses. $\overline{VINT}$ is asserted during the second cycle after the address is strobed on the MBus (with $\overline{MAS}$), or later if the CY7C605 is busy and cannot immediately supply the data. This signal must be tied to V_{CC} through a resistor.

Signal Name	I/O	Description
		When this pin is asserted by a CY7C605, the other CY7C605(s) in the system will tri-state their virtual bus signals (A<31:0>, D<31:0>, and ASI<5:0>) and assert $\overline{\text{MHOLD}}$ on the next rising clock. $\overline{\text{VINT}}$ remains asserted until the owned data is read from the cache RAMs. $\overline{\text{MHOLD}}$ remains asserted for one additional cycle after $\overline{\text{VINT}}$ is deasserted. In a multichip configuration, all of the CY7C605s connect to a common $\overline{\text{VINT}}$ pin.

MBus Signals

Signal Name	I/O	Description
$\overline{\text{CMER}}$	O	CMU Error (active LOW). This signal is open drain and is asserted if any bus error has occurred during writes to main memory. A system can use this signal to cause an interrupt. This signal will remain asserted until the asynchronous fault address register is read, at which time it will be tri-stated. This signal must be tied to V_{CC} through a resistor.
(605 ONLY)		In the CY7C605, $\overline{\text{CMER}}$ is also asserted if $\overline{\text{ERROR}}$ is asserted (watchdog reset). In this case, $\overline{\text{CMER}}$ will remain asserted until the reset register is read.
MAD<63:0>	I/O	MBus address and data (three-state bus). During the address phase of a transaction MAD<35:0> contains the physical address PA(35:0). The remaining signals MAD<63:36> contain the transaction-associated information as shown below during the address phase of the transaction:

MAD<39:36>	Transaction Type
0 H	MBus write
1 H	MBus read
2 H*	Coherent Invalidate
3 H*	Coherent Read
4 H*	Coherent Write and Invalidate
5 H*	Coherent Read and Invalidate
6–F H	Reserved

* CY7C605 ONLY

MAD<42:40>	Transaction Size
0	Byte (8 bits)
1	Halfword (16 bits)
2	Word (32 bits)
3	Doubleword (64bits)
4	16 Bytes**
5	32 Bytes
6	64 Bytes**
7	128 Bytes**

** Not supported by CY7C604/605.

MAD(43) (MC) MBus cacheable (active HIGH). Indicates the current MBus transaction is cacheable.

MAD(44) (MLOCK) MBus LOCK (active HIGH). Indicates the currrent MBus transaction is a locked transaction.

Signal Name	I/O	Description
		MAD(45) (MBL) MBus boot mode/local indicator. MBL is high during the address phase of boot mode transactions. The instruction fetch and data accesses to the MBus while the MMU is disabled in boot mode are considered BOOT MODE transactions. The data transactions on the MBus required for Load/Store alternate instructions with ASI = 01 are considered LOCAL transactions.
		MAD<53:46>† **(VA)** Virtual address bits VA<19:12>. The CY7C605 uses VA<15:12> for the virtually indexed cache.
		MAD<59:54> (Reserved) Driven HIGH during the address phase.
		MAD<63:60>† **(MID)** module identifier. This field is defined by the module ID number field in the SCR of a CY7C605. It is used by an MBus agent to identify the master who should be re-granted the bus on a Relinquish and Retry acknowledgement.
		During the data phase of the transaction, the MAD<63:0> lines contain the 64 bits of data being transferred.
$\overline{\text{MAS}}$	O (604) I/O(605)	MBus address strobe (active LOW). Asserted by the bus master during the first cycle of every bus transaction to indicate the address phase of that transaction. This is a three-state output. This signal must be tied to V_{CC} through a resistor.
$\overline{\text{MBB}}$	I/O	MBus bus busy (active LOW) asserted by the current MBus master during an entire transaction and, if required, during both the read and write transactions of indivisible accesses. The potential bus master devices sample $\overline{\text{MBB}}$ in order to obtain bus mastership as soon as the current master releases the bus. This is a three-state output. This signal must be tied to V_{CC} through a resistor.
$\overline{\text{MBG}}$	I	MBus Bus grant (active LOW). Asserted by external arbiter when the MBus is granted to a master. This signal is continually driven.
$\overline{\text{MBR}}$	O	MBus bus request (active LOW). Asserted by potential MBus master devices to acquire bus mastership. This signal is continually driven.
$\overline{\text{MERR}}$	I	MBus error (active LOW). Asserted or deasserted by an MBus slave during every data phase of a transaction. This signal is three-stated when released. This signal must be tied to V_{CC} through a resistor.
$\overline{\text{MIH}}$ (605 ONLY)	I/O	Memory inhibit (active LOW). Asserted by the CY7C605 for MBus transactions where the cache owns the data that has been requested on the MBus. This signal is monitored during bus snooping by the CY7C605. Refer to *Chapter 11* for further details. This signal must be tied to V_{CC} through a resistor.

† Applies to CY7C605 (Level 2 MBus) systems only.

Signal Name	I/O	Description
MRDY	I (604) I/O (605)	MBus ready (active LOW). Asserted or deasserted by an MBus slave during every data phase of a transaction. This signal is to be three-stated when released. This signal must be tied to V_{CC} through a resistor.
MRST (604 ONLY)	O	MBus reset (active LOW). Asserted for 1024 clock cycles by only one source on the MBus to initialize all devices on the MBus. This signal is continually driven.
MRTY	I	MBus retry (active LOW). Asserted or deasserted by an MBus slave during every data phase of a transaction. This signal is three-stated when released. This signal must be tied to V_{CC} through a resistor.

MERR	MRDY	MRTY	Action
H	H	H	Nothing
H	H	L	Relinquish and Retry*
H	L	H	Data Strobe
H	L	L	Reserved
L	H	H	Bus Error
L	H	L	Time Out
L	L	H	Uncorrectable Error
L	L	L	Retry*

* See Chapter 11 on MBus.

Signal Name	I/O	Description
MSH (605 ONLY)	I/O	Memory shared (active LOW). Asserted by the CY7C605 after detecting a data request on the MBus for which the CY7C605 has a copy. This signal is monitored by the CY7C605 during bus snooping. Refer to *Chapter 11* for further information. This signal must be tied to V_{CC} through a resistor.
POR	I	Power-on reset (active LOW). The POR initializes all on-chip logic to a known state, invalidates all the TLB entries, and all cache tag entries. It must be asserted for a minimum of 8 clocks. It also causes the CY7C604/605 to assert IRST to reset the CY7C601.

Cache RAM Signals

Signal Name	I/O	Description
CBWE<3:0>	O	Cache byte write enables (active LOW). During normal write operations, certain byte enable signals are asserted depending upon the size and A(1:0) inputs. During a cache line load all four byte enable signals are asserted. These signals can also be driven by using a store alternate instruction with ASI = 0F H. This feature is supported for diagnostic purposes. This output is continually driven (not three-stated). CBWE0 controls the most significant byte (MSB) and CBWE3 controls the least significant byte (LSB). Refer to page 8-40 for further information on this signal.
CROE	O	Cache RAM output enable (active LOW). Asserted during normal read operations with ASI = 8, 9, A, B and during modified cache line read operations. This signal is also asserted during cache data read operations with ASI = 0F H for diagnostic purposes. This signal is continually driven.

Miscellaneous Signals

Signal Name	I/O	Description
CLK	I	System clock. This is the same clock used by the 7C601 Integer Unit.
$\overline{\text{CSEL}}$	I	Chip select (active LOW). In multi-CMU systems, $\overline{\text{CSEL}}$ on each CY7C604/605 is connected to different address lines (any one from A<31:16>) to initialize the multichip configuration. In single-CMU systems, $\overline{\text{CSEL}}$ should be connected to ground in order to permanently enable the CY7C604/605. In multi-CMU systems, $\overline{\text{CSEL}}$ should be connected to ground or VCC through a resistor during power-on reset. This is required in order to enable only one boot mode CMU. (Refer to *Section 8.5, Multichip Configuration* for more details.)
CSTA (604 only)	O	Cache status. This pin provides the status of cache. In write-through, the CSTA indicates whether the cache line associated with a write transaction on the MBus is valid or not. For MBus read transactions, in both write-through and copy-back mode, the CSTA indicates whether the CY7C604 is replacing a valid cache line entry or not. This signal has the same timing specifications as the MBus signals such as MC and has meaning only in the address phase of MBus transactions. This signal is continually driven HIGH or LOW.

Cache Mode	CSTA	Condition
Write-through	1	read and valid cache line replacement
	0	read and invalid cache line replacement
	1	write cache hit
	1	write cache miss and cache line valid
	0	write and cache line invalid
Copy-back	1	read and valid cache line replacement
	0	read and invalid cache line replacement
	undef.	write

Signal Name	I/O	Description
TOE	I	Test/output enable (active LOW). When HIGH, this signal is used to three-state all output drivers of the CY7C604/605. $\overline{\text{TOE}}$ SHOULD BE TIED LOW DURING NORMAL OPERATION. It is used to isolate the CY7C604/605 from the rest of the system for debugging purposes.

8.11 Virtual Bus Operation

The following timing diagrams illustrate CY7C604/605 virtual bus operations:

Page

Figure 8–50. Write-Through (Copy-Back) Read Cache Hit 8-72
Figure 8–51. Write-Through (Copy-Back, Clean Cache Line) Read Cache Miss 8-73
Figure 8–52. Write-Through, Read Cache Miss (Alias Detected) 8-76
Figure 8–53. Write-Through Write Cache Hit 8-77
Figure 8–54. Write-Through Write Cache Miss 8-78
Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line 8-79
Figure 8–56. Copy-Back, Write Cache Miss, Modified or Non-Modified (Alias Detected) 8-84
Figure 8–57. Copy-Back Read Cache Miss, Modified or Non-Modified (Alias Detected) 8-85
Figure 8–58. Copy-Back, Write Cache Hit 8-86
Figure 8–59. Write-Through Load-Double Cache Hit 8-86
Figure 8–60. Write-Through, Store-Double Cache Hit 8-87
Figure 8–61. Table Walk (with Modified Bit Update) 8-88
Figure 8–62. Read Access with Protection or Privilege Violation 8-92
Figure 8–63. CY7C604/605 Diagnostic Cache Tag Write Access 8-92
Figure 8–64. CY7C604/605 Register Read 8-93
Figure 8–65. CY7C604/605 Register Write 8-93
Figure 8–66. Power-On Reset Timing (CY7C604 only) 8-94
Figure 8–67. Power-On Reset Timing (CY7C605 only) 8-96
Figure 8–68. Software External Reset 8-98
Figure 8–69. Software Internal Reset 8-98

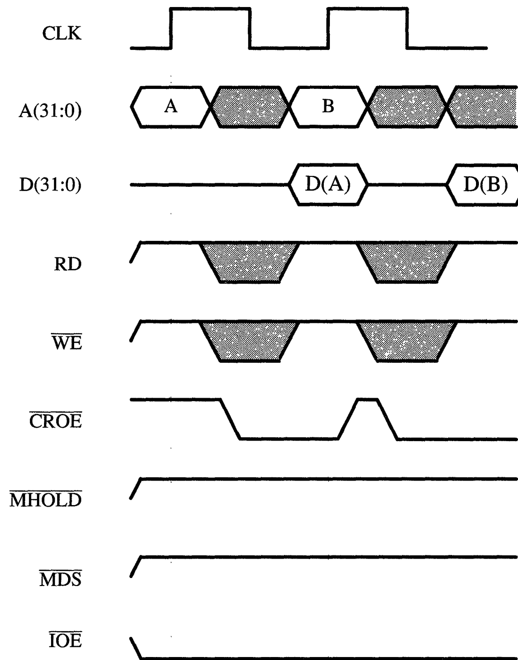


Figure 8–50. Write-Through (Copy-Back) Read Cache Hit

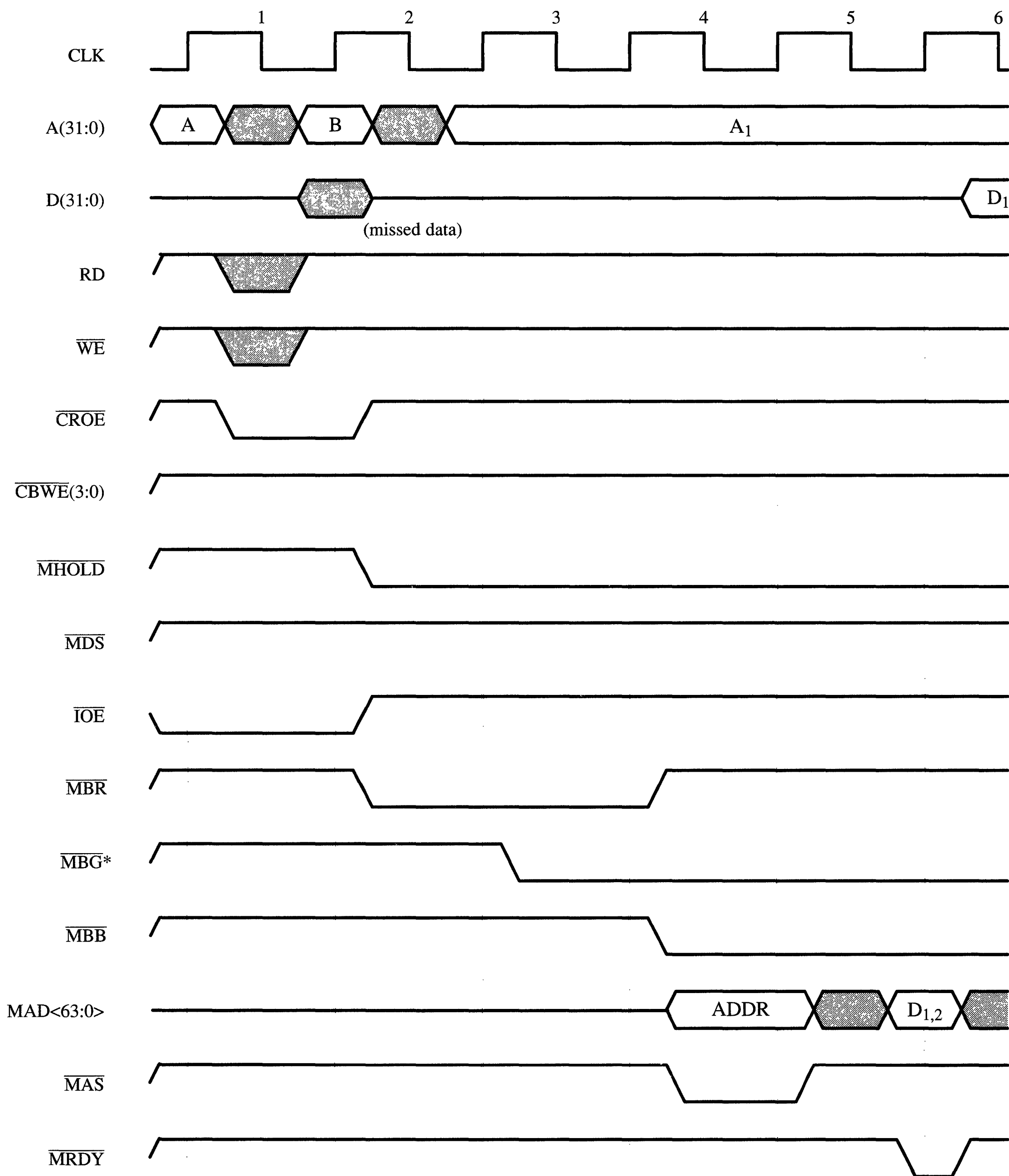


Figure 8–51. Write-Through (Copy-Back, Clean Cache Line) Read Cache Miss (page 1 of 3)*

* Two clocks can be deleted from the cache miss timing if $\overline{\text{MBG}}$ is already granted.

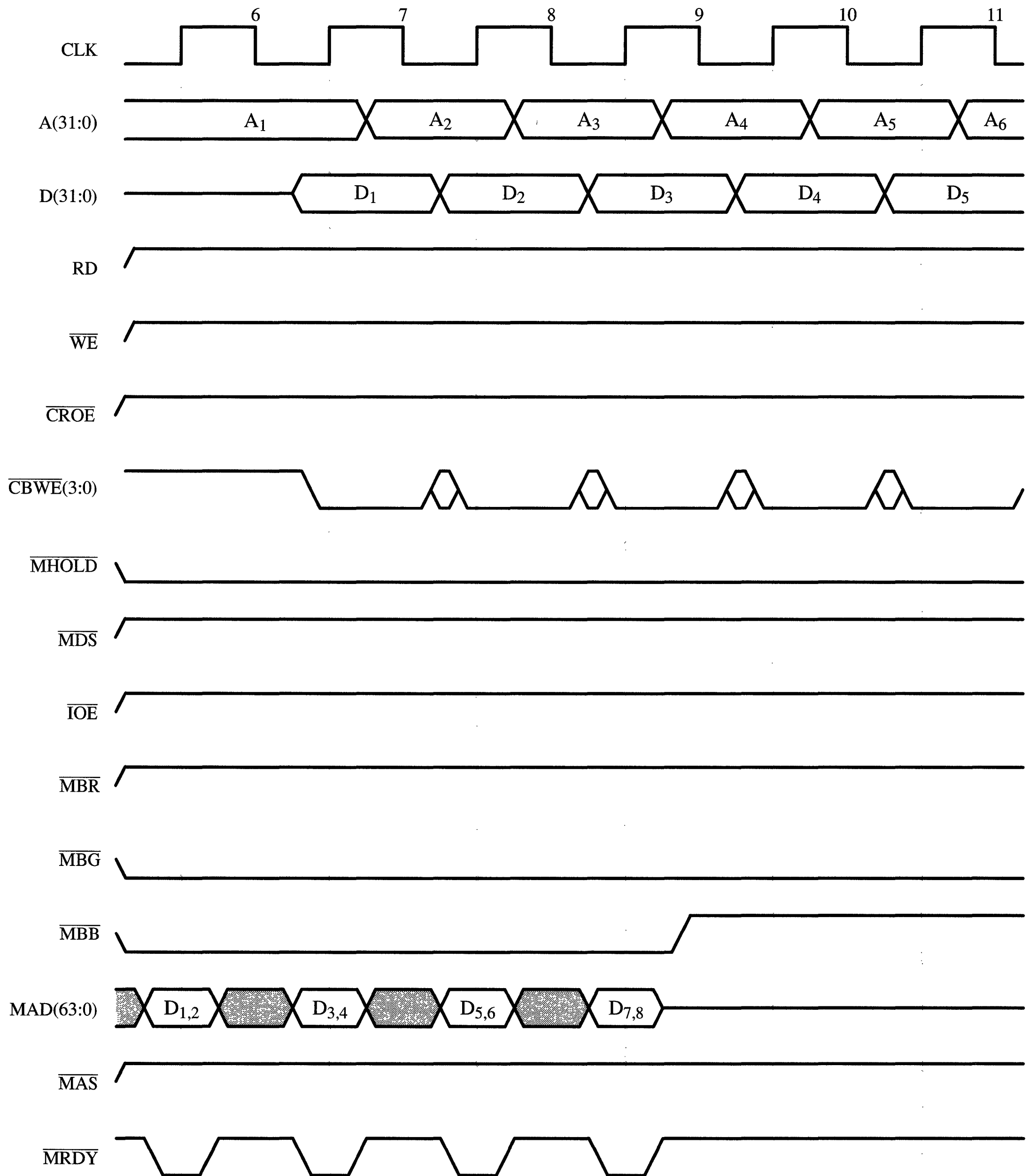


Figure 8–51. Write-Through (Copy-Back) Read Cache Miss (page 2 of 3)

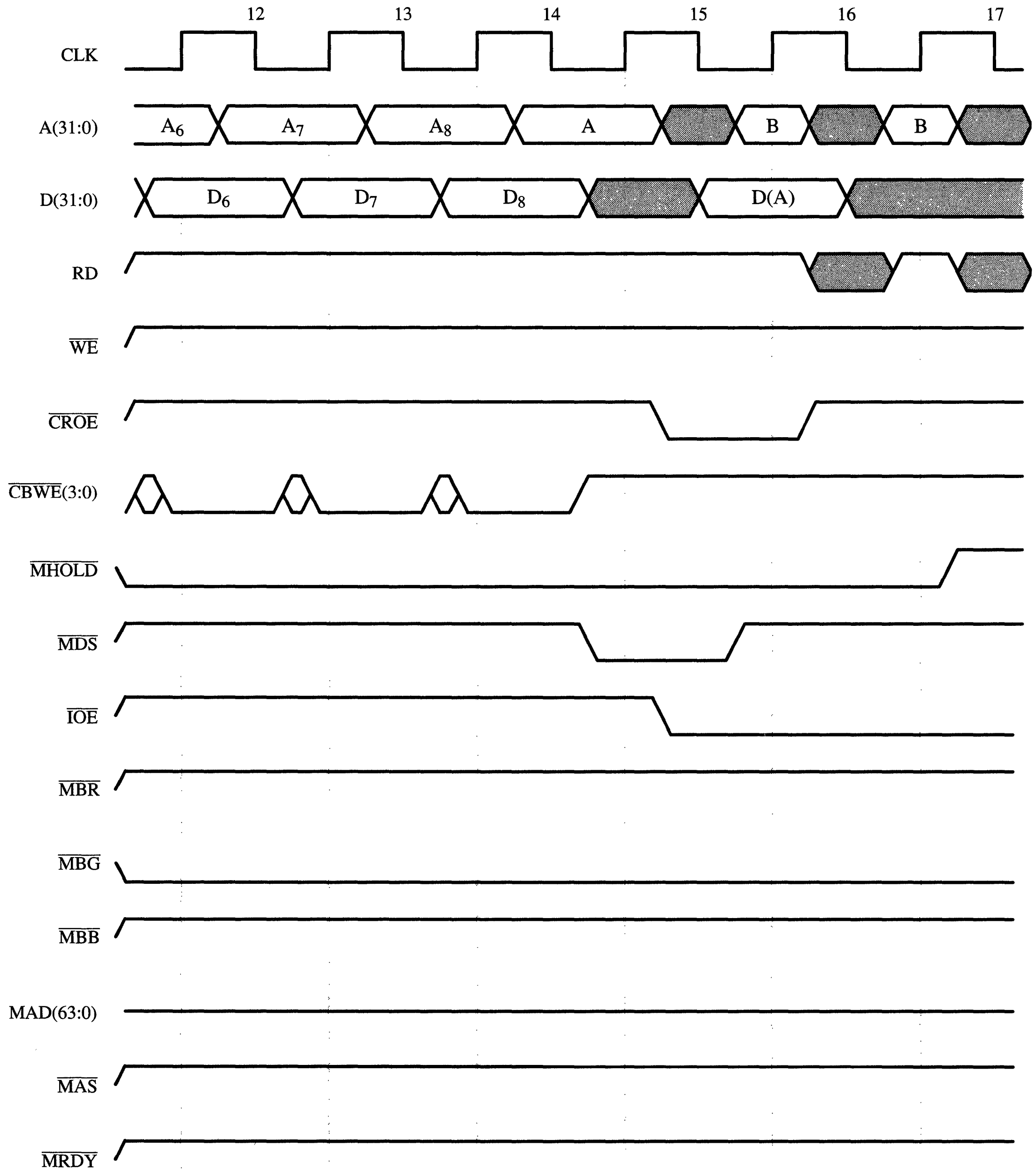


Figure 8–51. Write-Through (Copy-Back) Read Cache Miss (page 3 of 3)

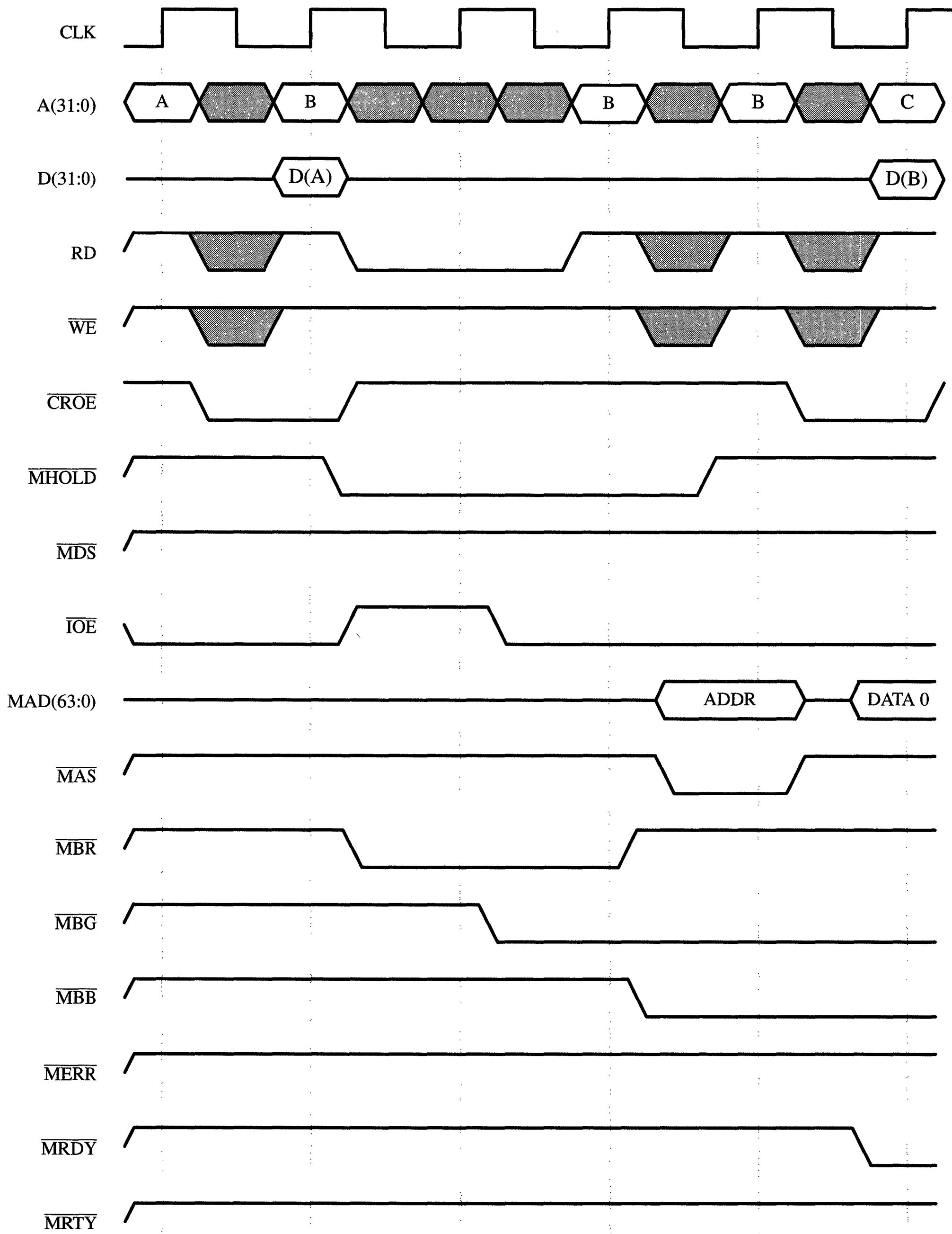


Figure 8–52. Write-Through, Read Cache Miss (Alias Detected)*

* Although aliasing is detected, the MBus access is not aborted (the CY7C604/605 ignores the access). The MBus transaction terminates normally.

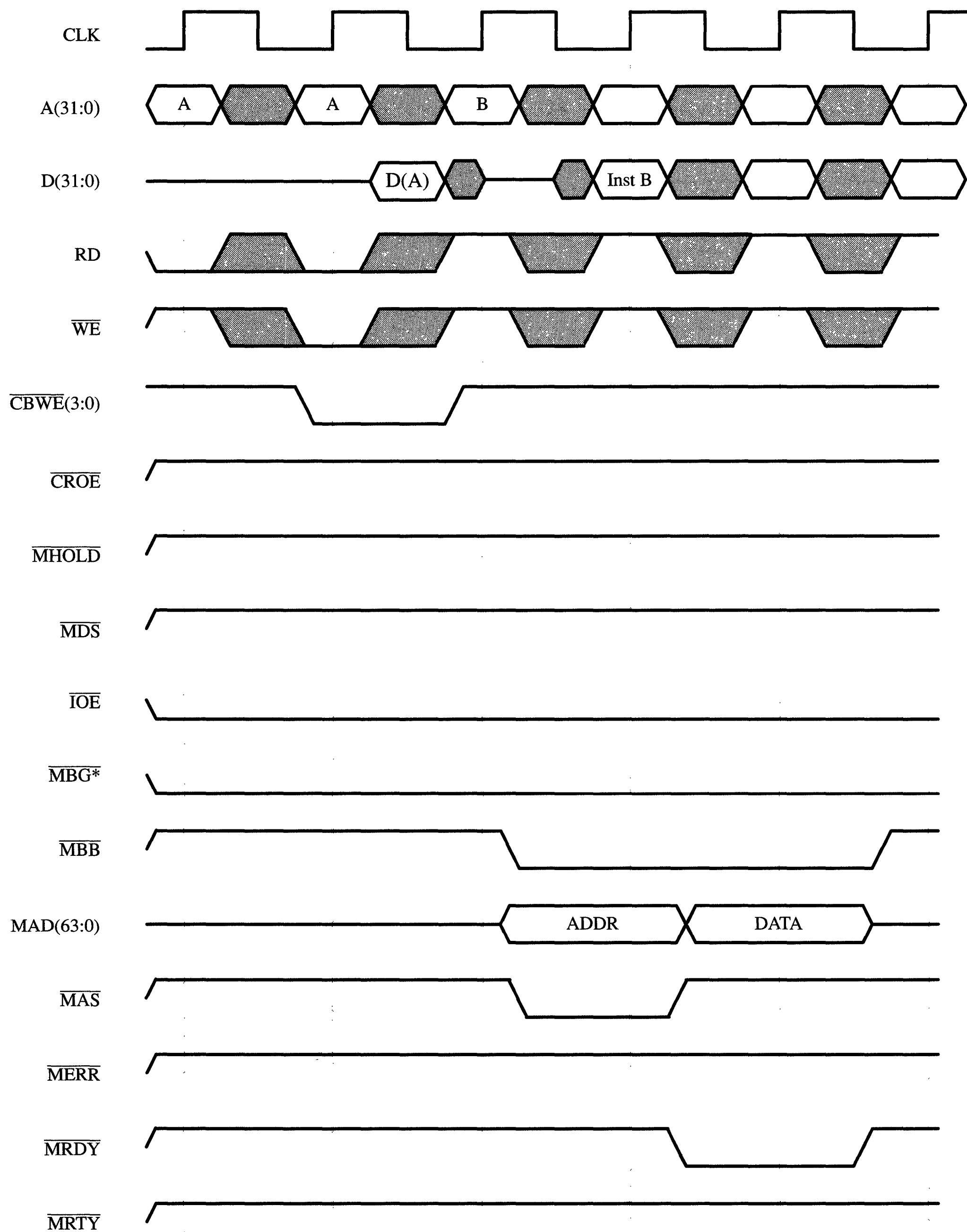


Figure 8–53. Write-through, Write Cache Hit

* This timing diagram is an example of bus parking (i.e. MBG granted by default to the CY7C604/605).

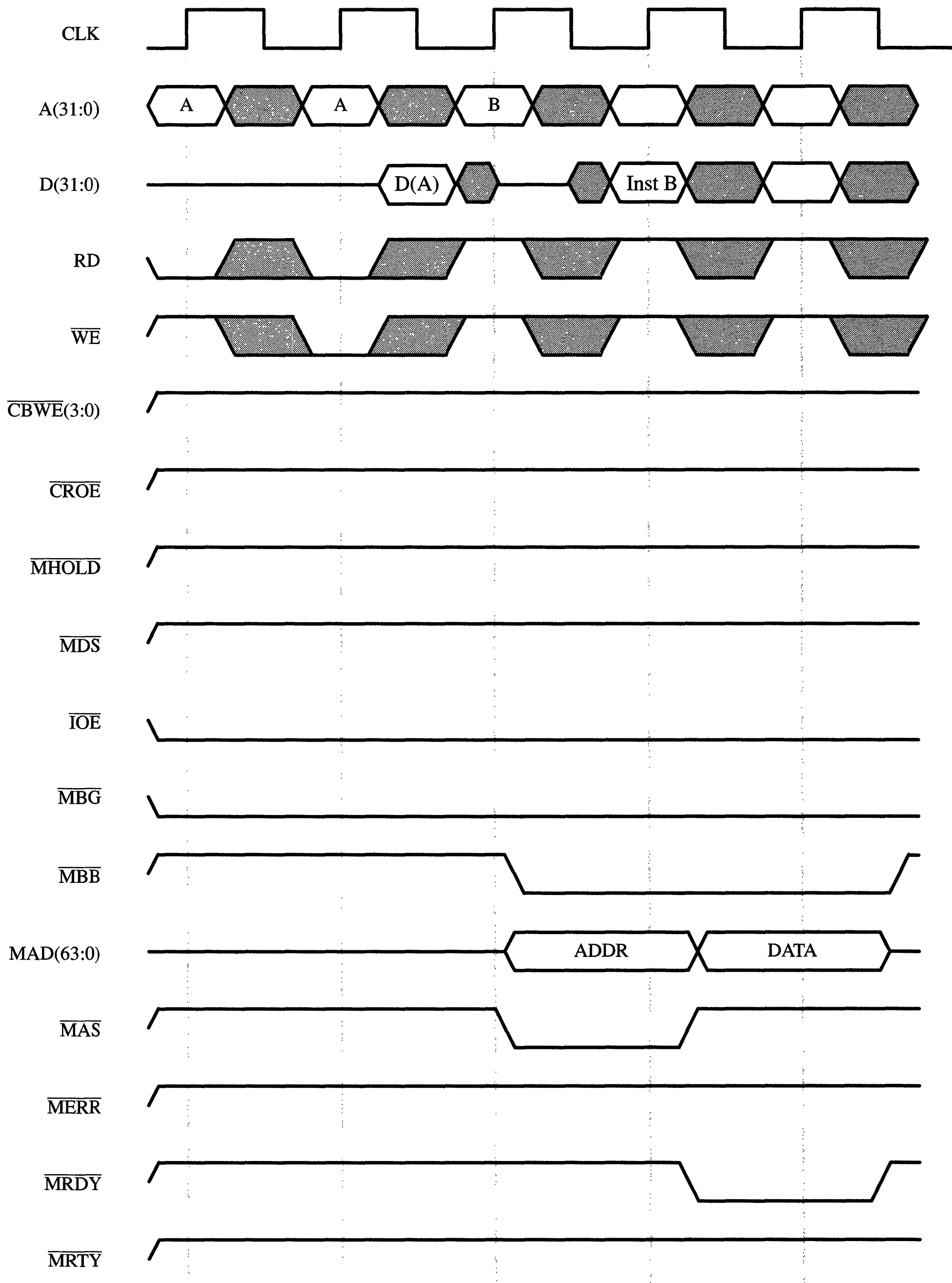


Figure 8–54. Write-Through, Write Cache Miss

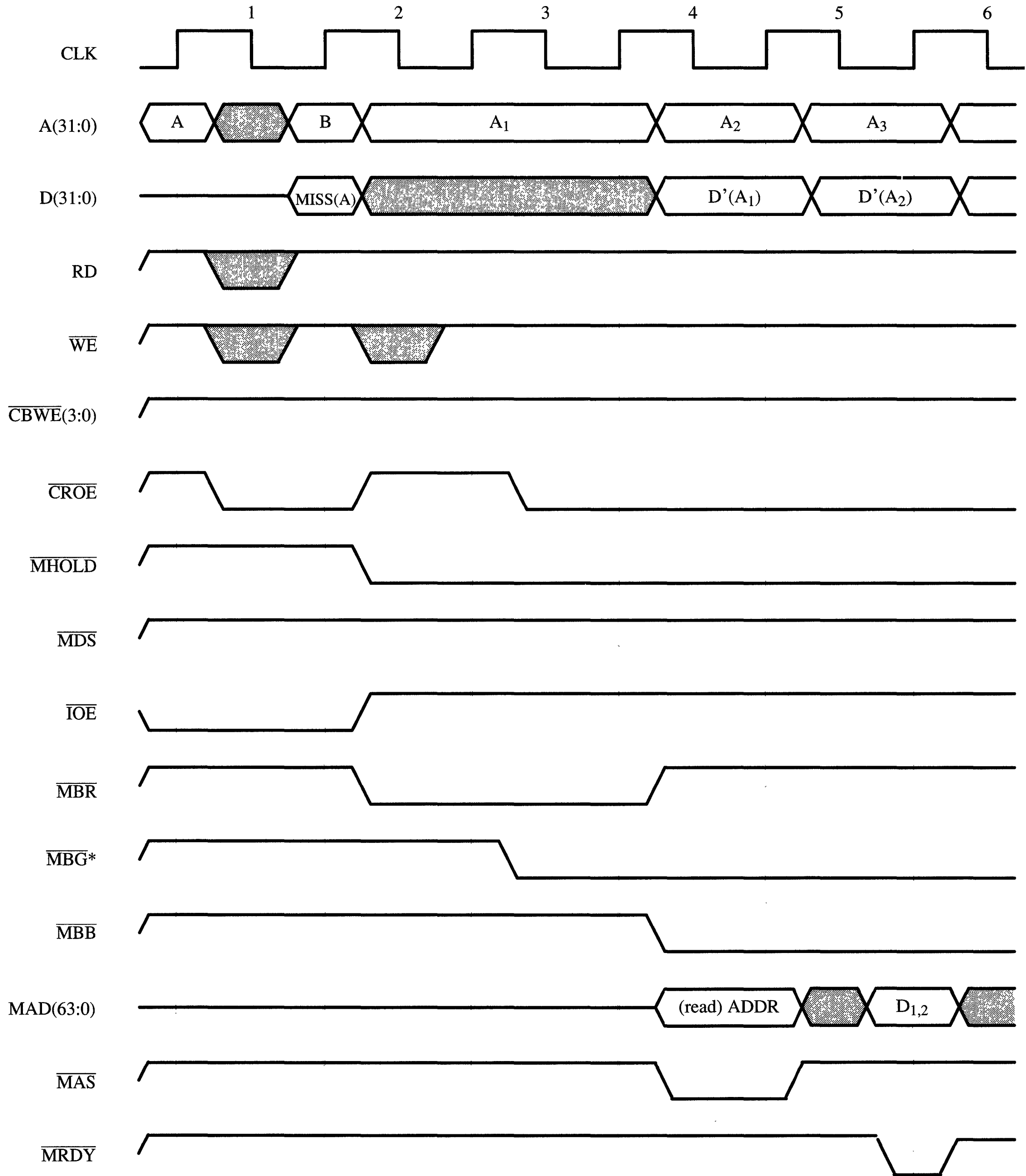


Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line (page 1 of 5)*

* Two clock cycles can be deleted from this timing diagram if the $\overline{\text{MBG}}$ signal is already asserted.

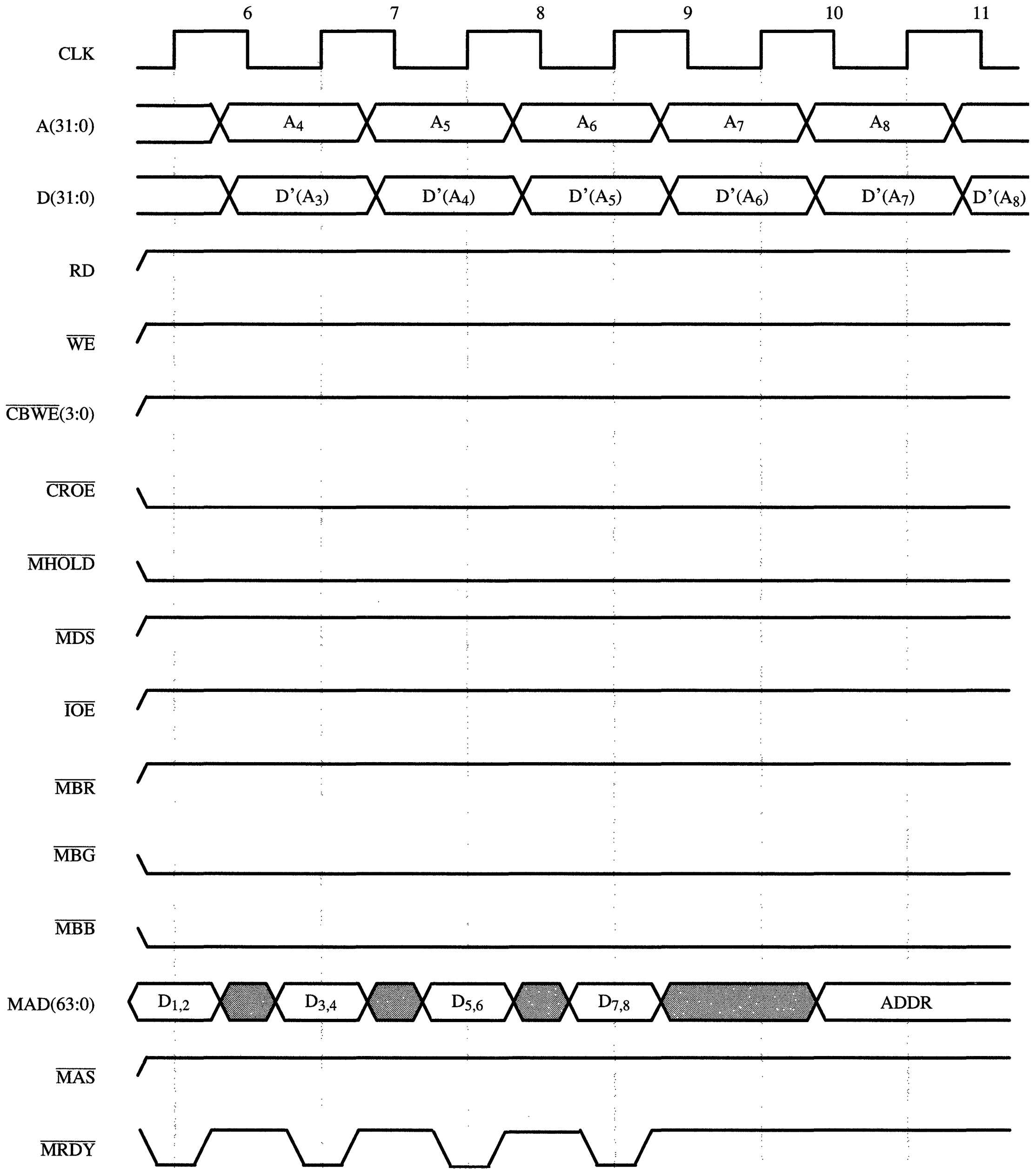


Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line (page 2 of 5)

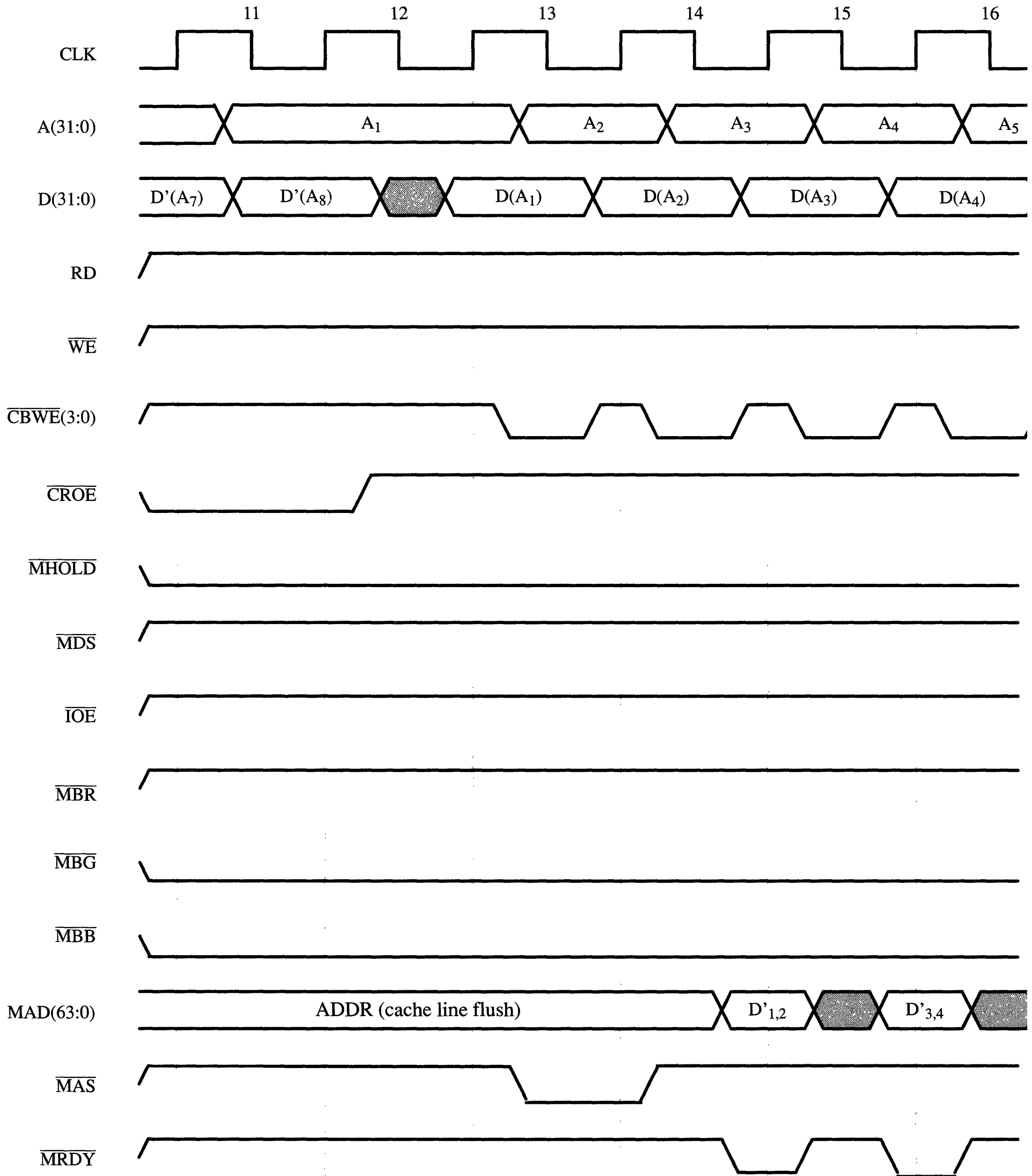


Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line (page 3 of 5)

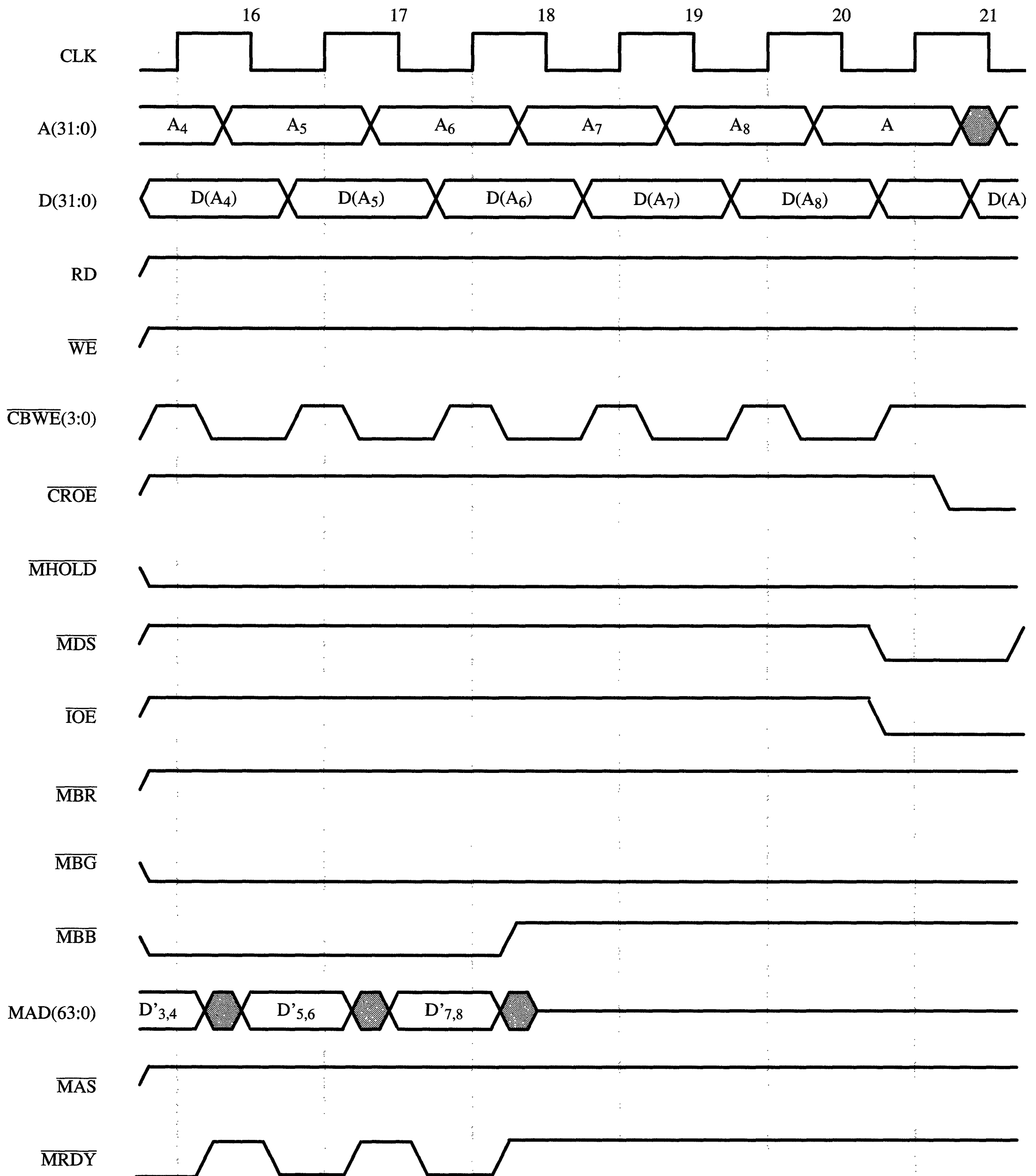


Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line (page 4 of 5)

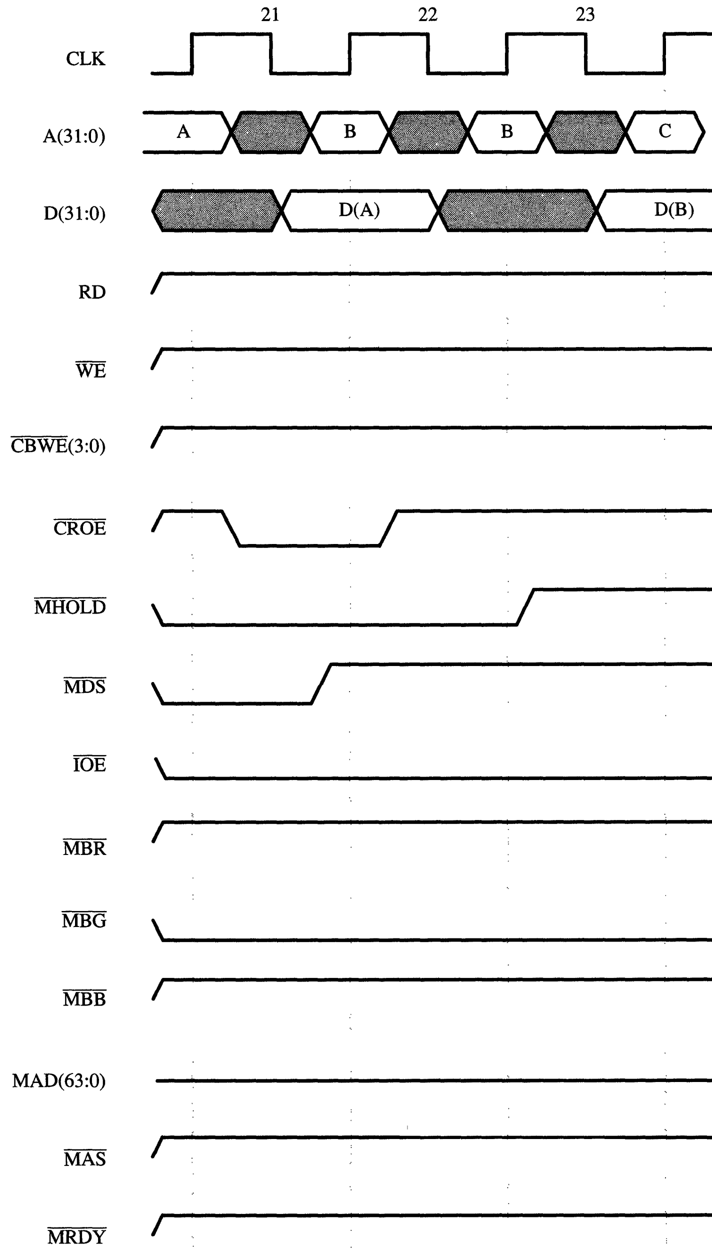


Figure 8–55. Copy-Back Cache Read Cache Miss, Modified Cache Line (page 5 of 5)

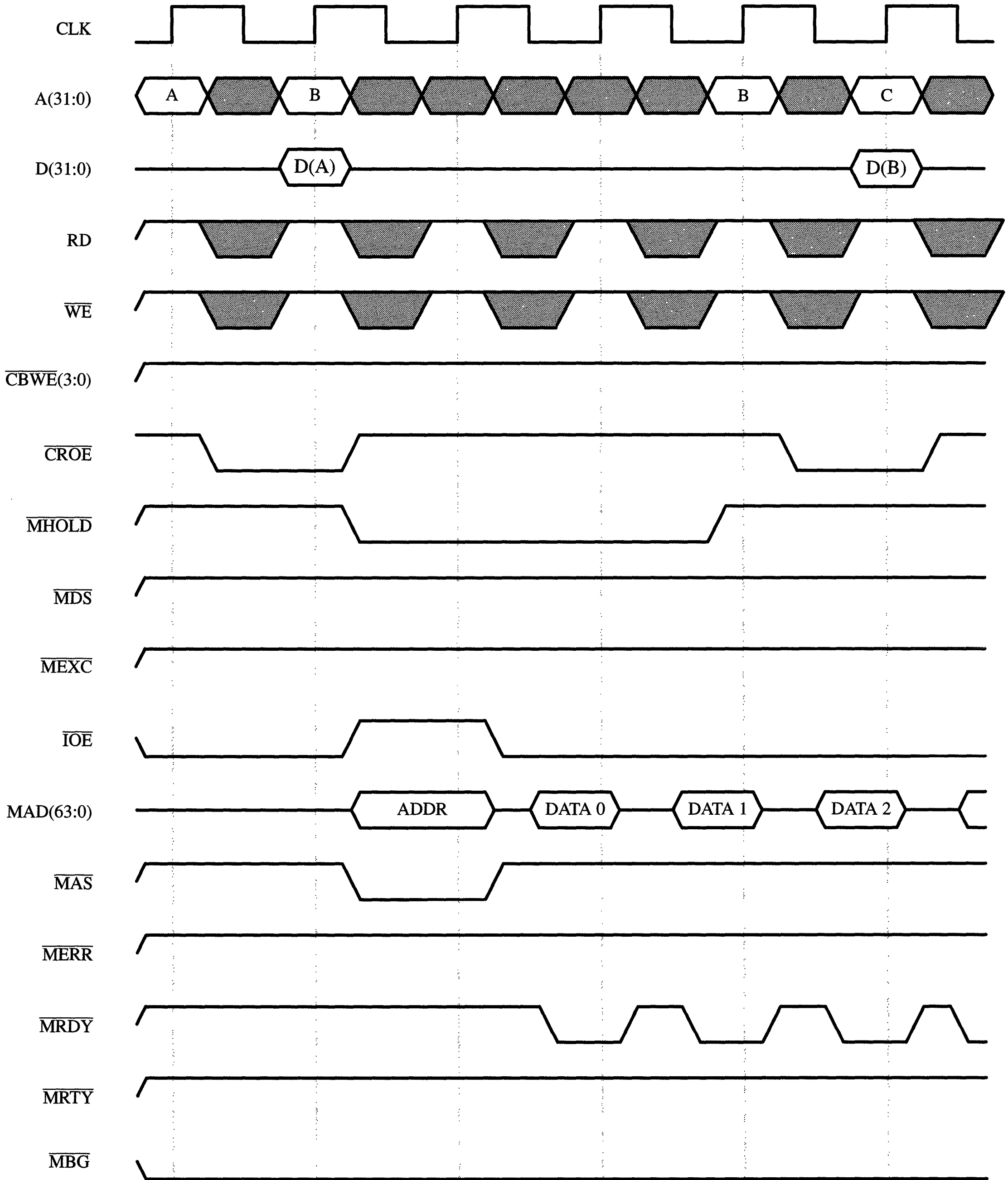


Figure 8–56. Copy-Back, Write Cache Miss, Modified or Non-Modified (Alias Detected)*

* Even though aliasing is detected, the MBus is not aborted (the CY7C604/605 ignores the access). The MBus transaction terminates normally. Timing assumes MBus is parked (already granted).

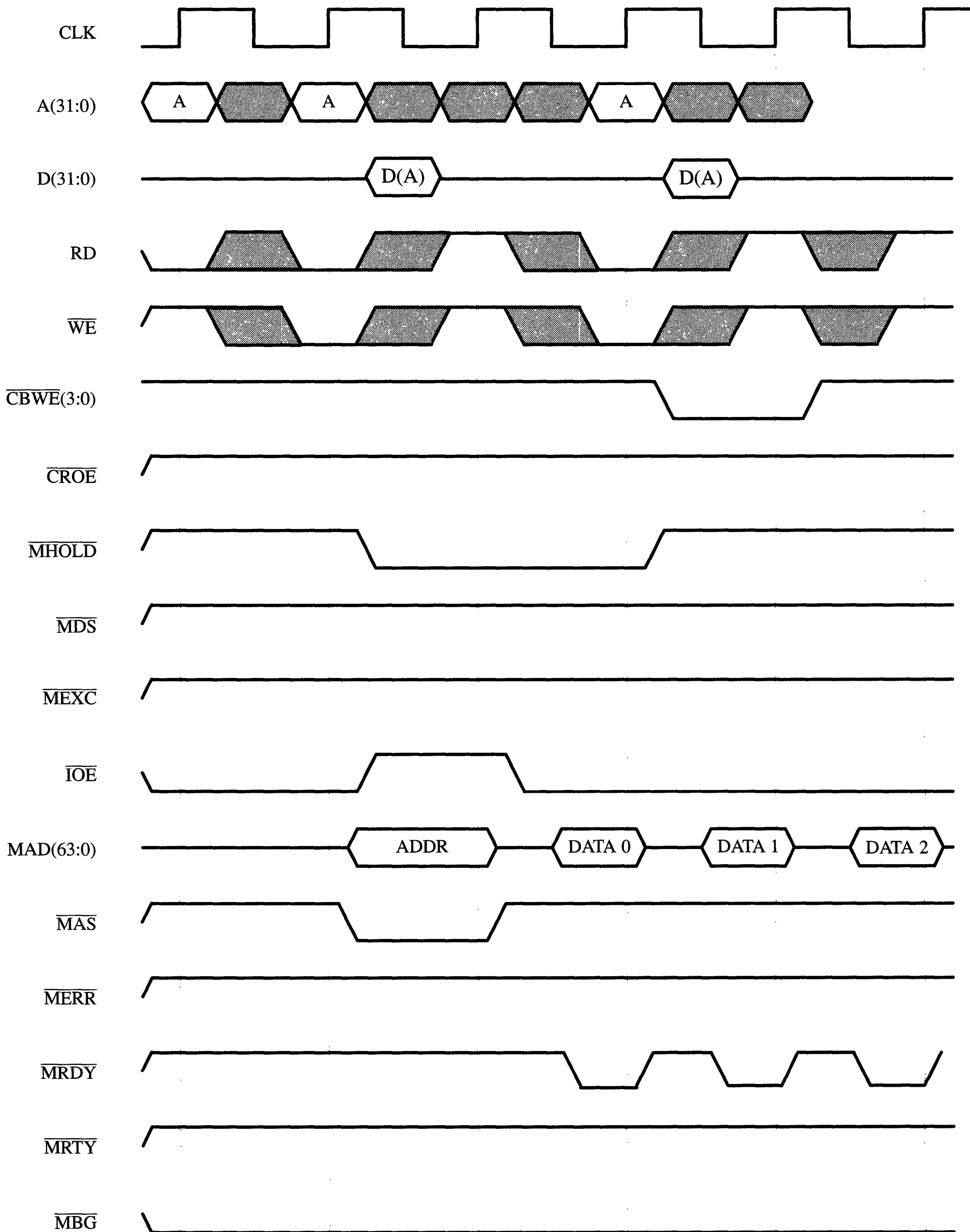


Figure 8–57. Copy-Back Read Cache Miss, Modified or Non-Modified (Alias Detected)

* Even though aliasing is detected, the MBus is not aborted (the MBus controller ignores the access). Timing assumes MBus is parked (already granted).

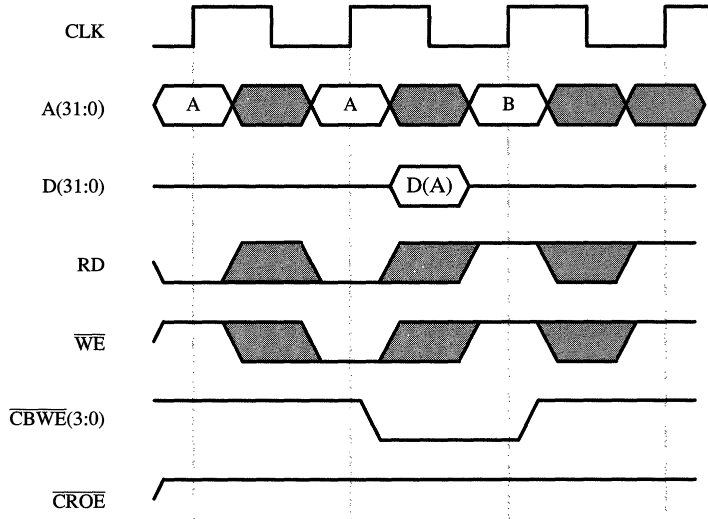


Figure 8–58. Copy-Back, Write Cache Hit

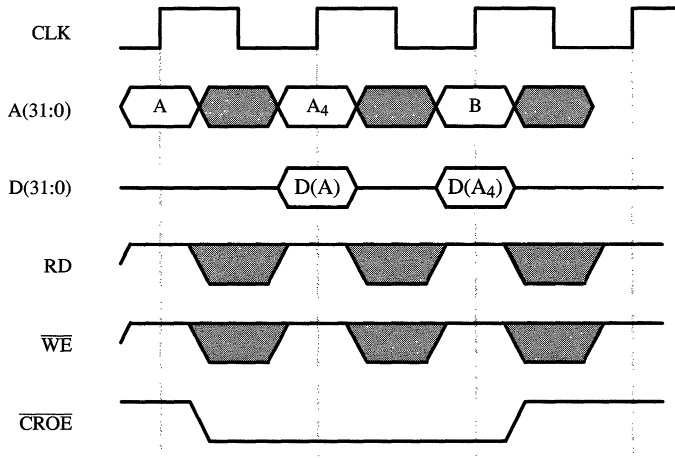


Figure 8–59. Write-Through Load-Double Cache Hit

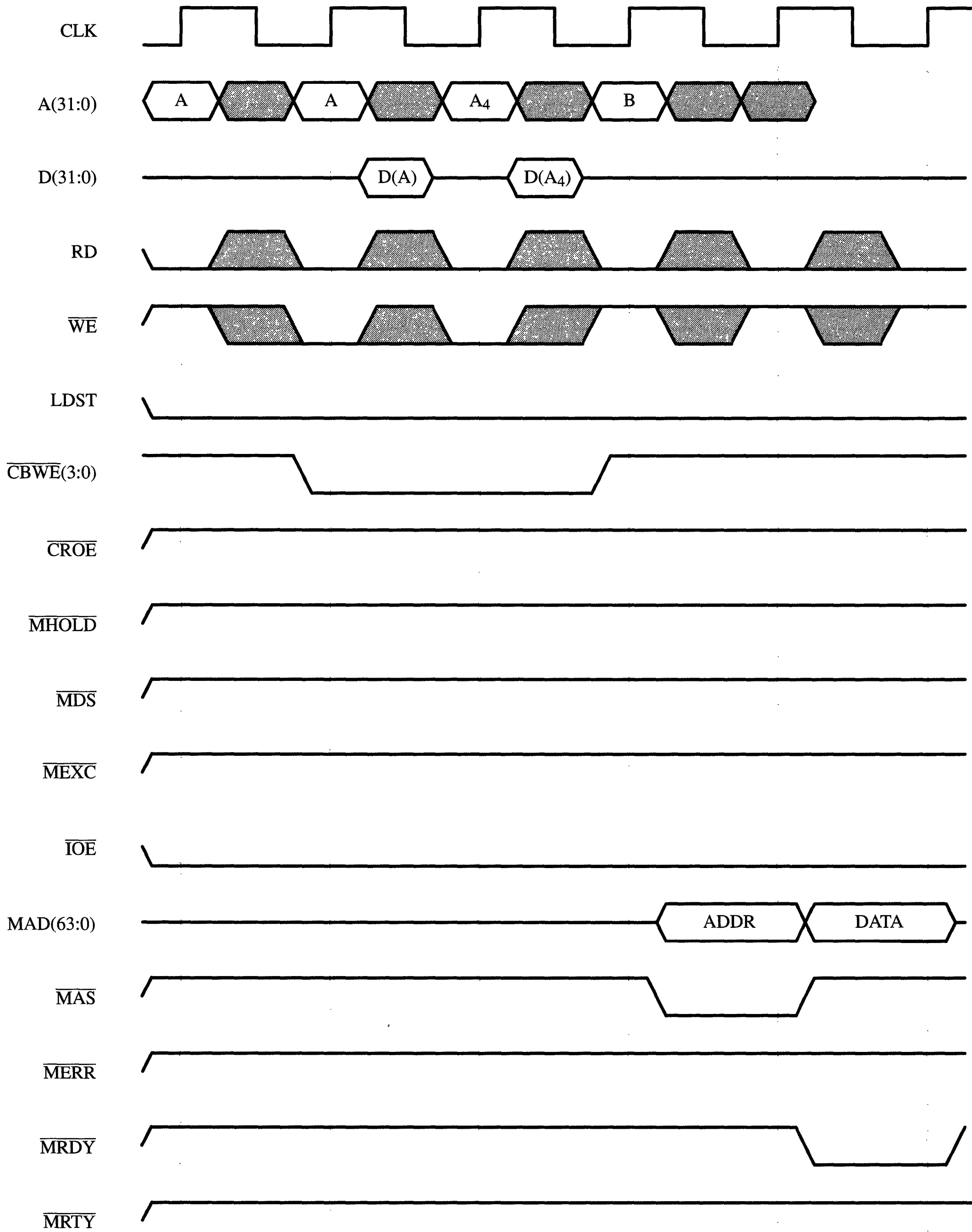


Figure 8–60. Write-Through, Store-Double Cache Hit*

* The MBus cycle is not initiated until both 32-bit transfers of the double store are received.

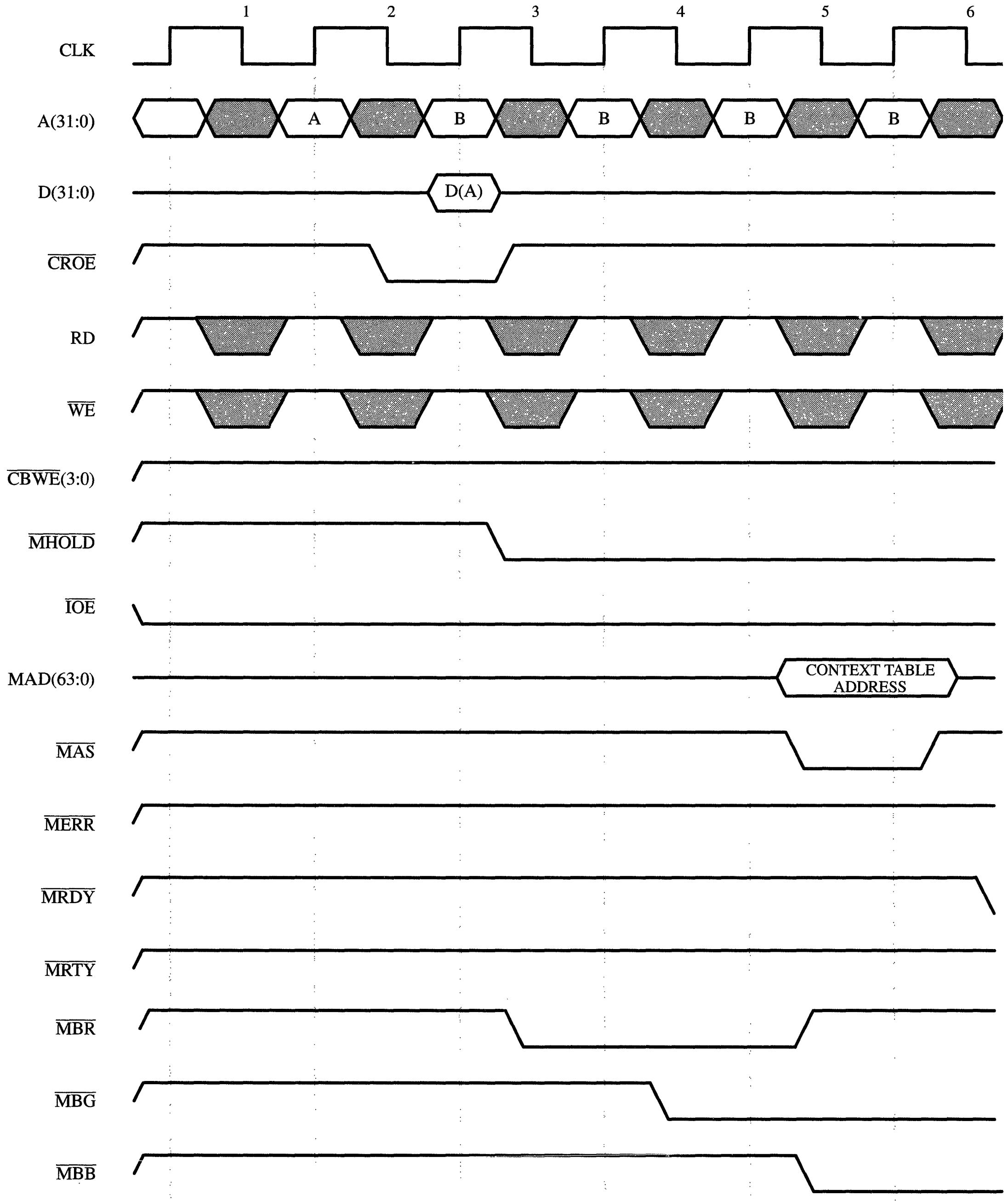


Figure 8–61. Table Walk Timing Diagram (with Modified Bit Update)* (page 1 of 4)

* This table walk illustrates a cache read hit with TLB miss. This table walk updates the TLB and performs access protection checking.

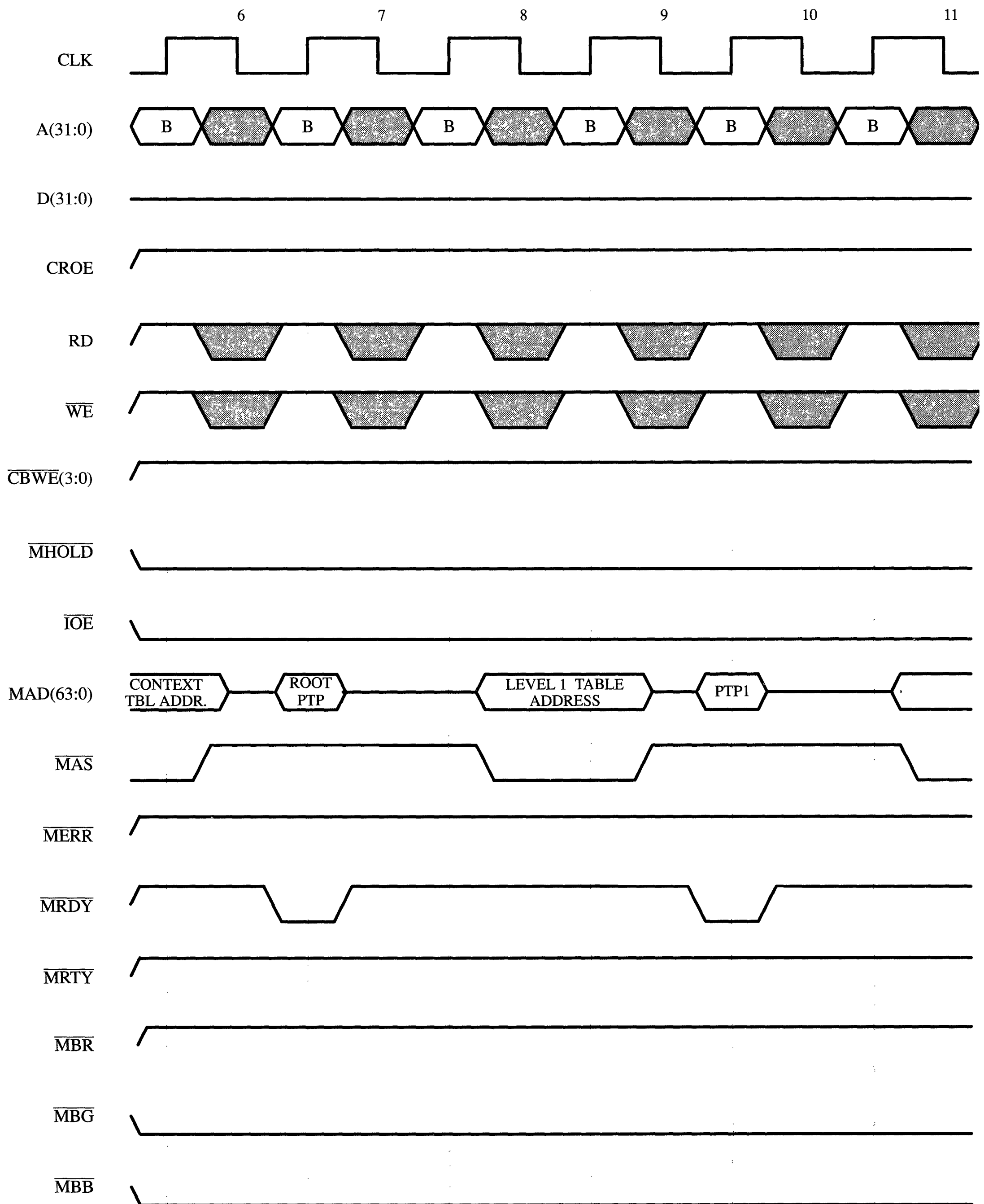


Figure 8–61. Table Walk Timing Diagram (with Modified Bit Update) (page 2 of 4)

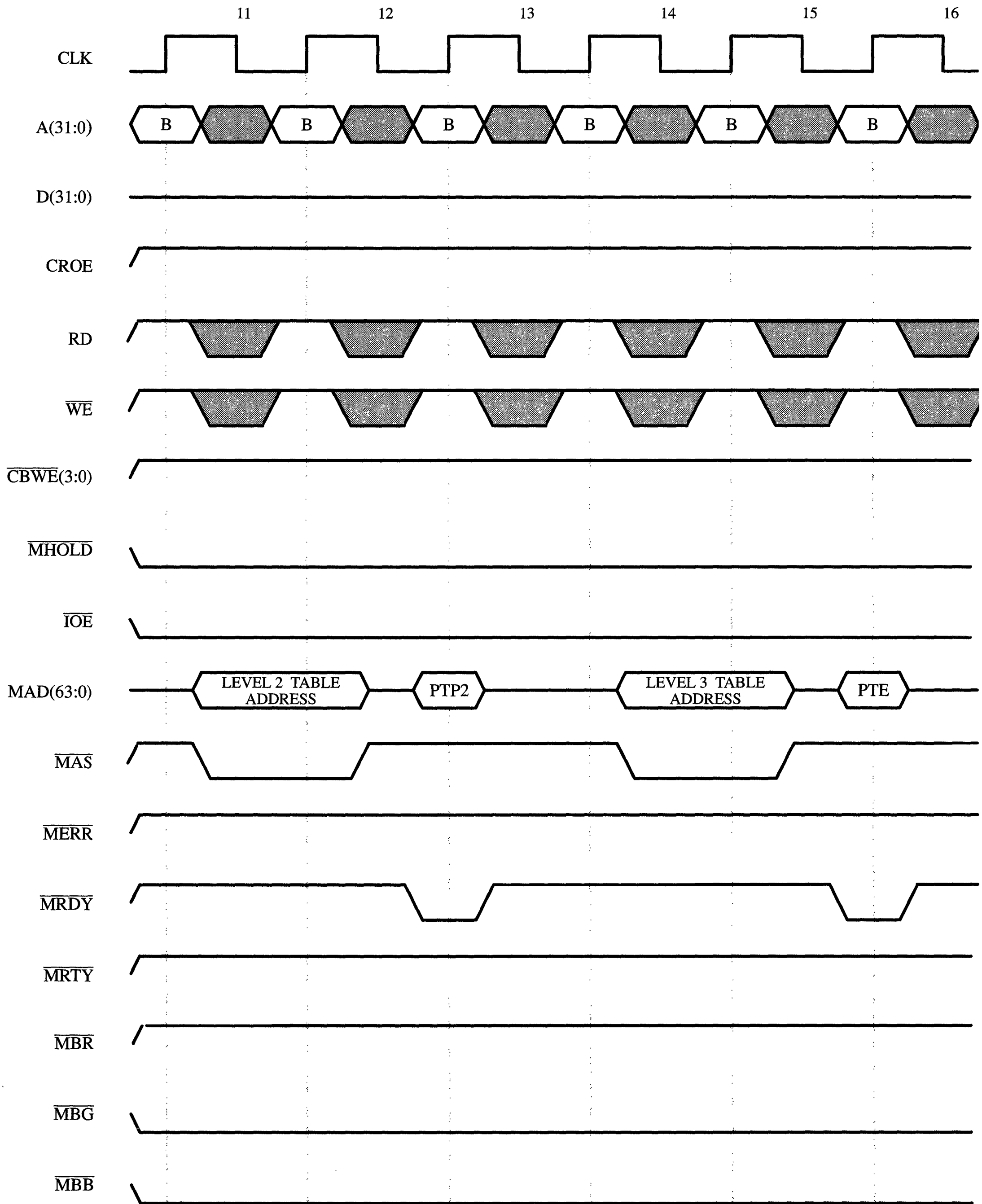


Figure 8–61. Table Walk Timing Diagram (with Modified Bit Update) (page 3 of 4)

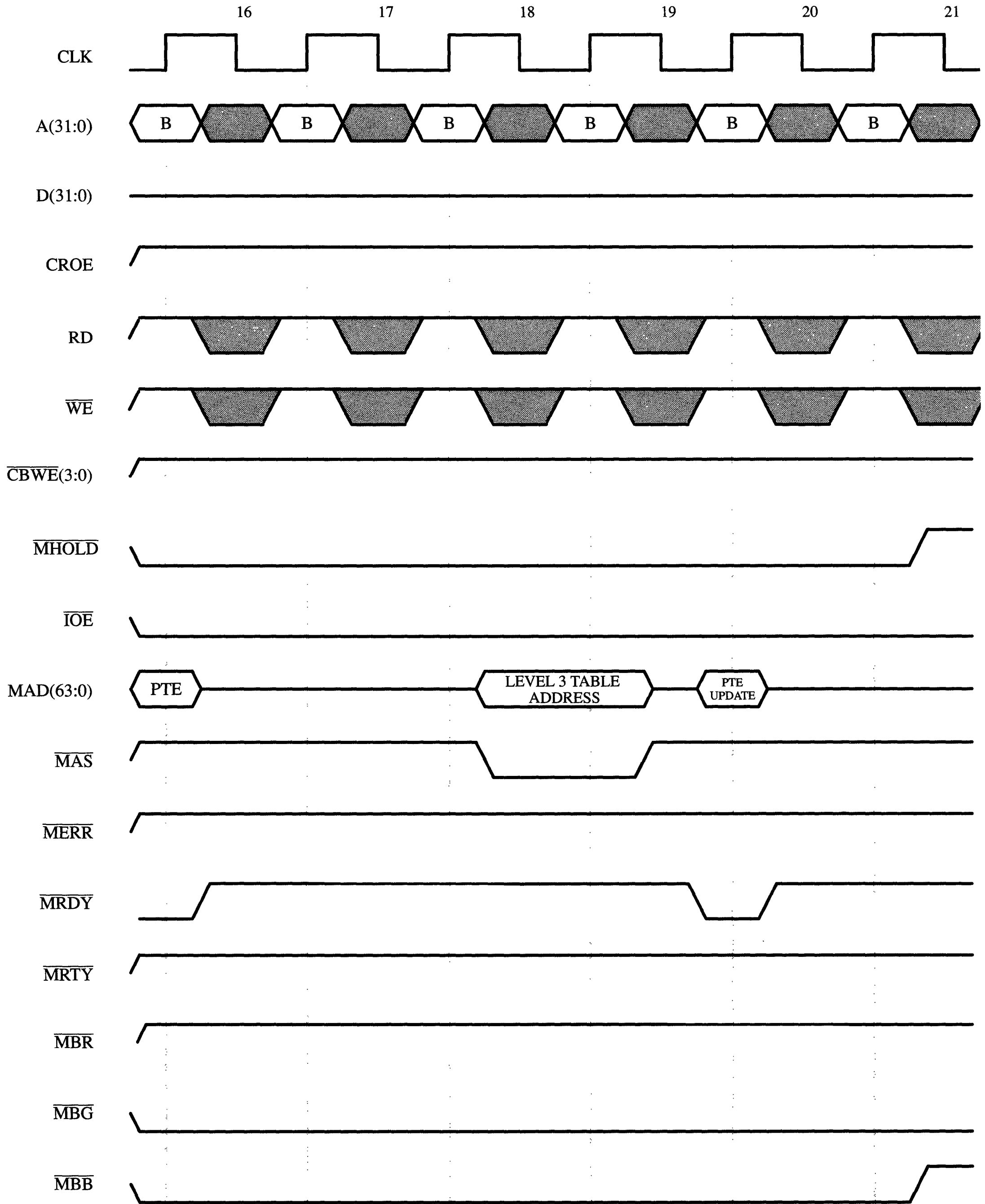


Figure 8–61. Table Walk Timing Diagram (with Modified Bit Update) (page 4 of 4)

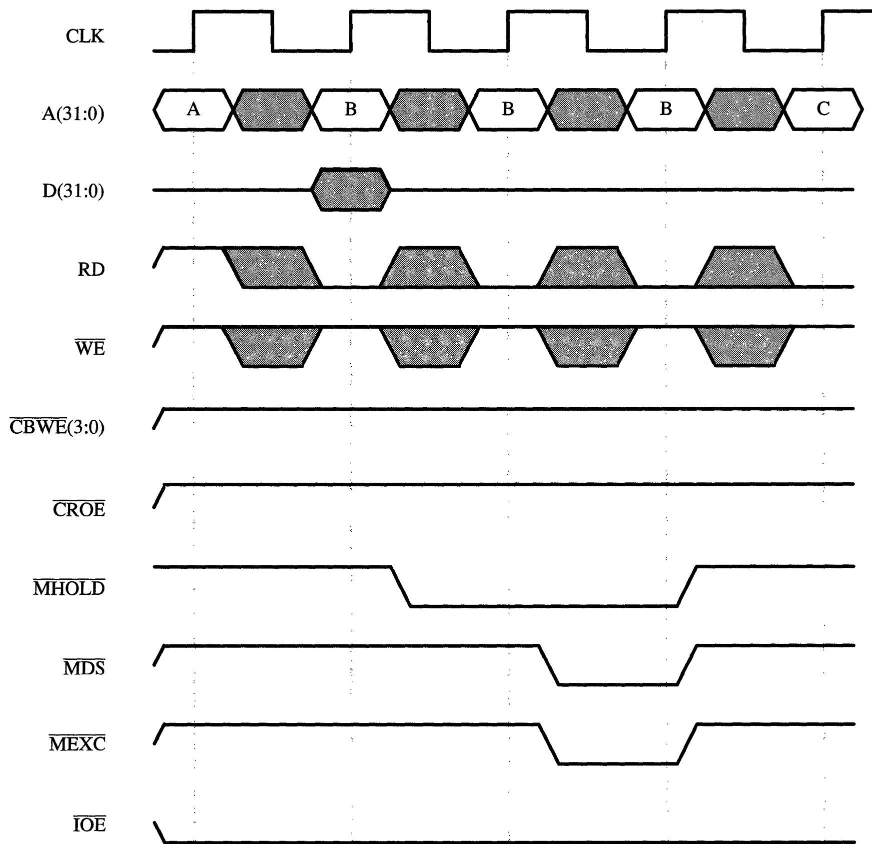


Figure 8–62. Read Access with Protection or Privilege Violation

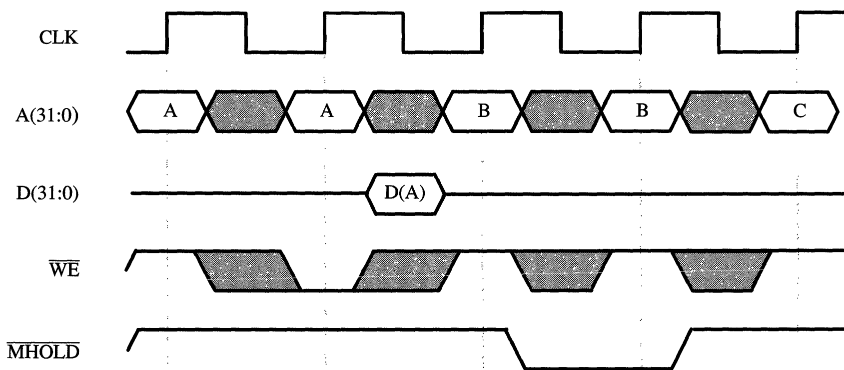


Figure 8–63. CY7C604/605 Diagnostic Cache Tag Write Access

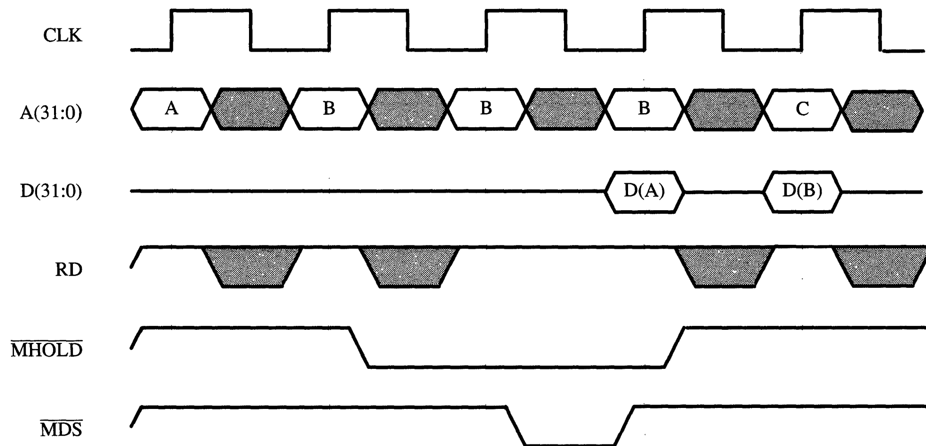


Figure 8–64. CY7C604/605 Register Read

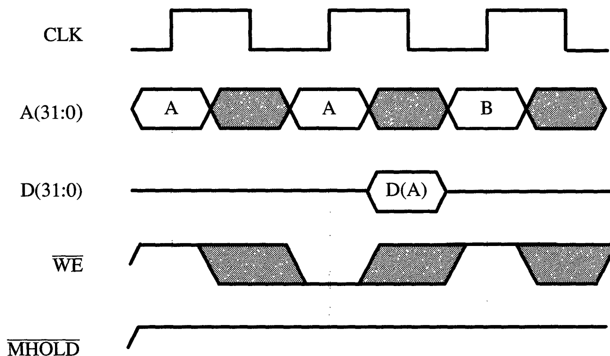


Figure 8–65. CY7C604/605 Register Write

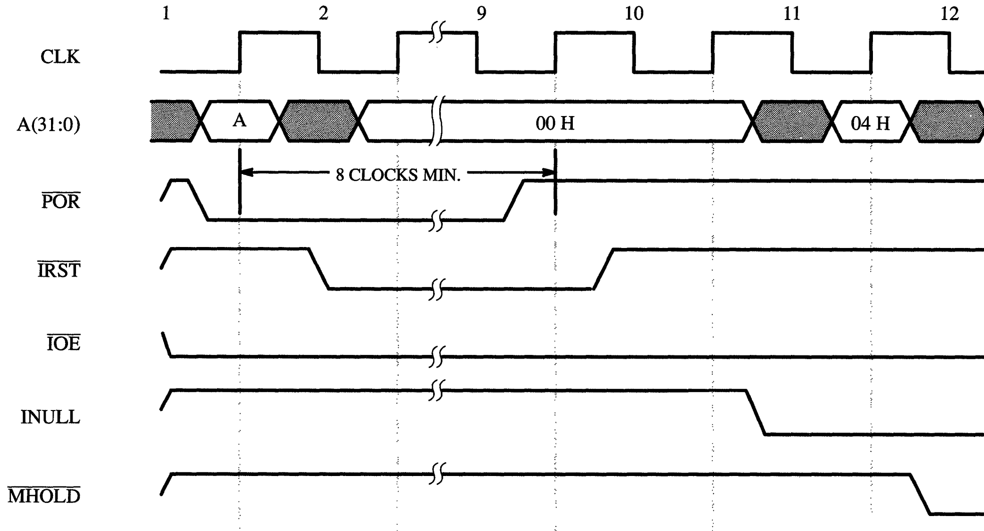


Figure 8–66. Power-On Reset Timing (CY7C604 only) (page 1 of 2)

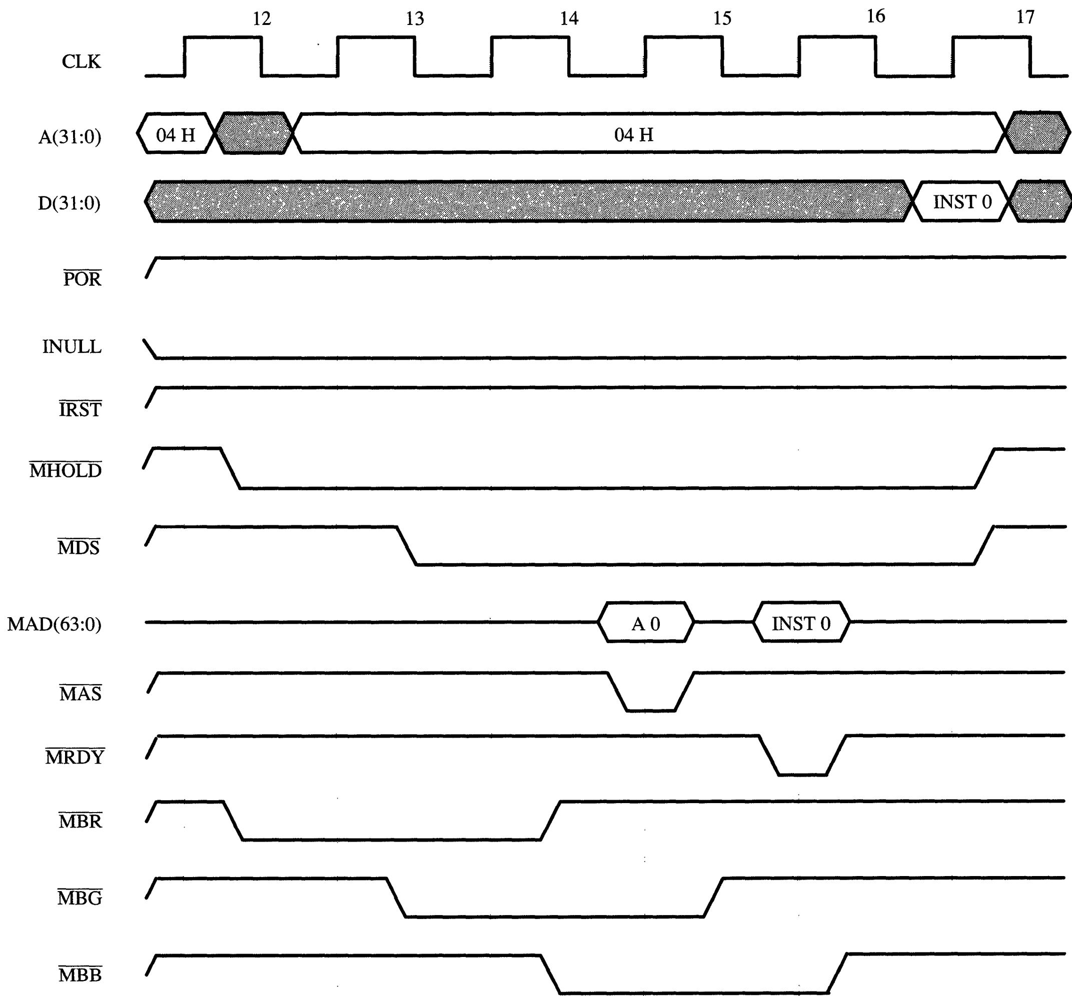


Figure 8–66. Power-On Reset Timing (CY7C604 only) (page 2 of 2)

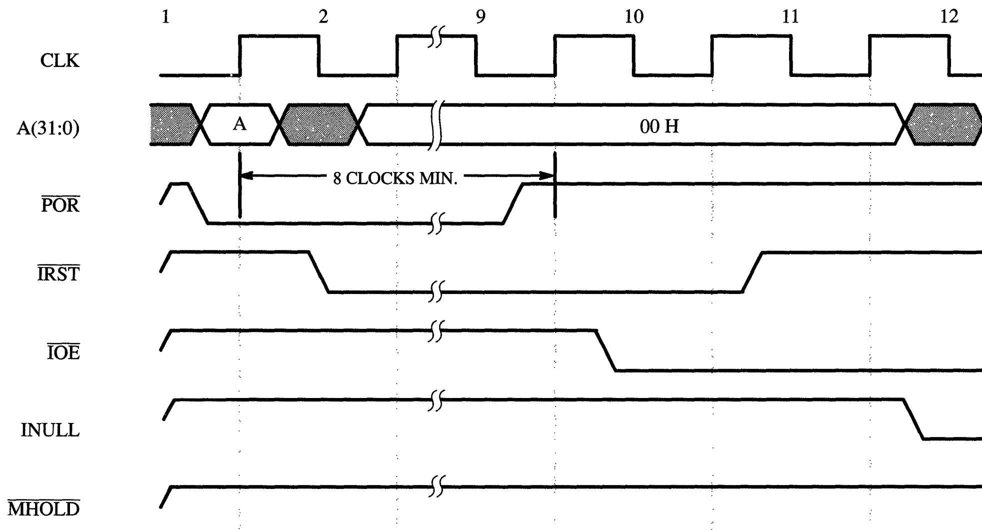


Figure 8–67. Power-On Reset Timing (CY7C605 only) (page 1 of 2)

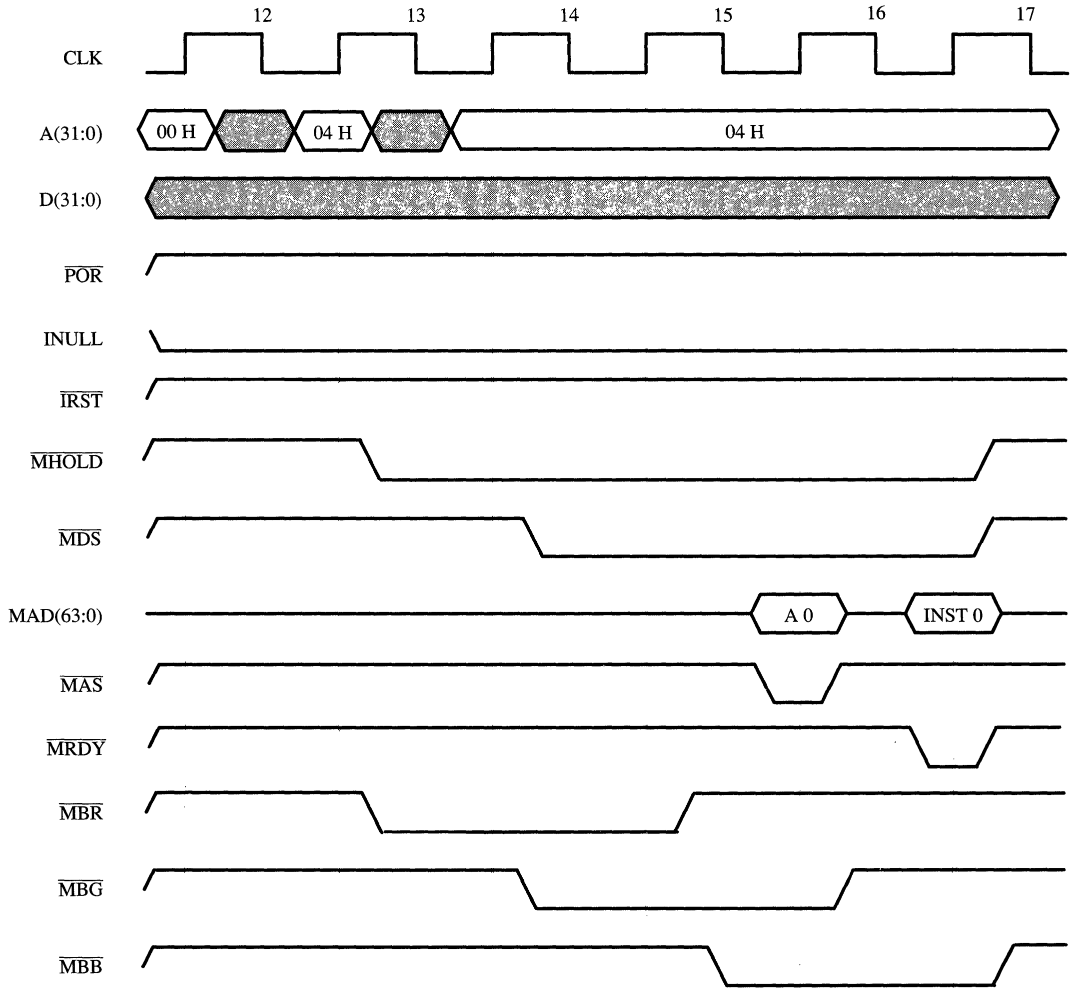


Figure 8–67. Power-On Reset Timing (CY7C605 only) (page 2 of 2)

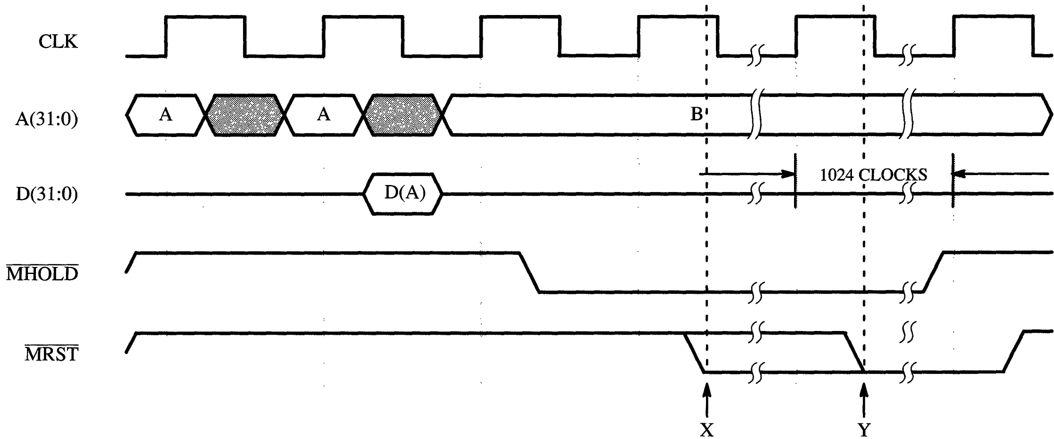


Figure 8–68. Software External Reset

Notes:

1. Address A will be 00000700 H and ASI will be 04 H
2. Data A will be 00000001 H
3. MRST will not be asserted until the write buffers are empty. If empty, MRST will be asserted at point X. If not empty, MRST will be asserted at point Y (the rising clock following the final data phase of emptying the write buffer.) In either case, MRST will be asserted for 1024 clock cycles.

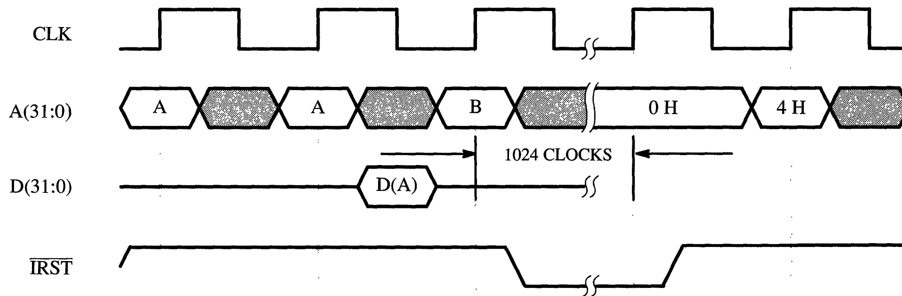


Figure 8–69. Software Internal Reset

Notes:

1. Address A will be 00000700 H and ASI will be 4 H.
2. Data A will be 00000002 H
3. IRST causes CY7C601 to place address 0 on address bus while asserted. CY7C601 continues with reset address sequence after IRST is deasserted.

CY7C157 Cache Storage Unit

The CY7C157 Cach Storage Unit (CSU) is a high-performance CMOS static RAM organized as 16 Kbytes x 16 bits. It is intended specifically for use as a high-speed cache memory for the CY7C600 family of SPARC devices. The CY7C157's 20-ns access time allows operation at processor clock speeds to 40 MHz.

The CY7C157 includes registered inputs as well as data-in and data-out latches. Because it was designed specifically for 7C600 family devices, the CY7C157 CSU requires no glue logic to interface with the CY7C601, CY7C602, CY7C604, or CY7C605. All relevant pins on each device connect directly to one another. The combination of direct connection and on-chip latches and registers yields system designs requiring less board space at a lower cost and with increased reliability. In addition, the CY7C157's self-timed byte-write mechanism relieves the system of any write timing chores.

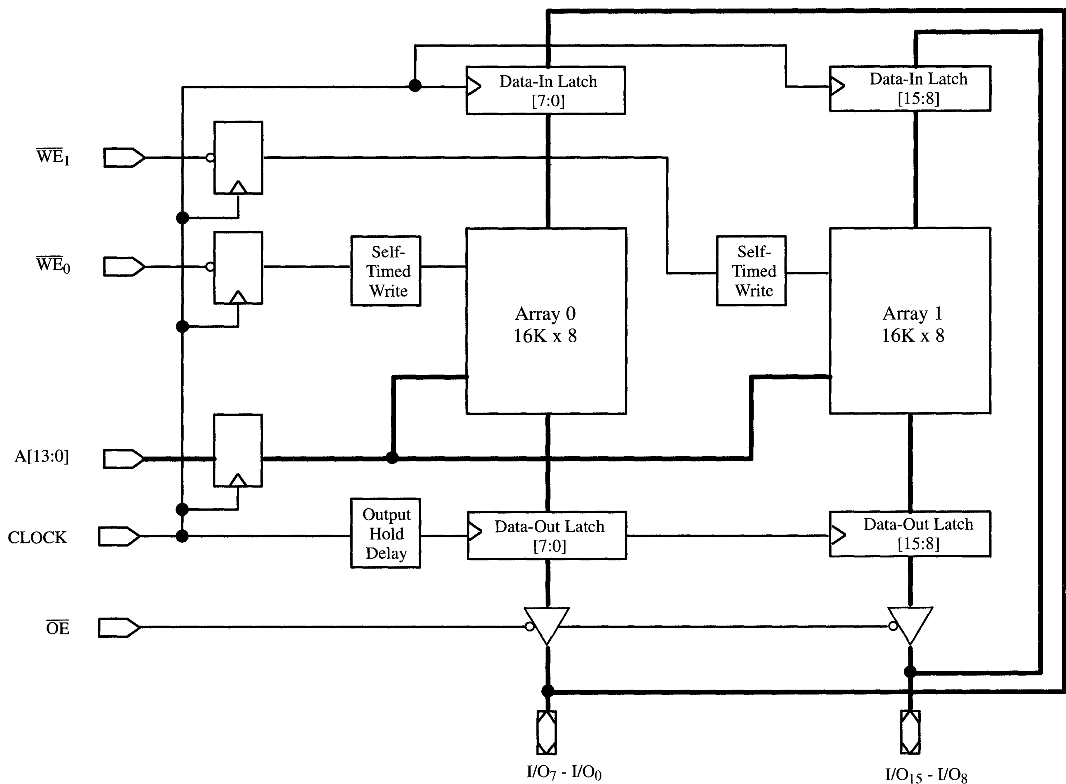


Figure 9–1. CY7C157 Block Diagram

9.1 Description Of Part

The CY7C157 is organized as two arrays of 16-Kbyte static memory. In order to minimize external timing and interface logic, the CY7C157 contains self-timed byte write logic, registered inputs, data in and data out latches, and output hold delay logic to control the data out latches.

Reading the device is accomplished by deasserting $\overline{WE}$ HIGH, and $\overline{OE}$ LOW. On the rising edge of CLOCK, addresses A(13:0) are loaded into the input registers. A memory access occurs, and data is held until the next rising edge of CLOCK in order to meet the hold time requirements of the CY7C601.

To write to the CY7C157, $\overline{OE}$ must be taken HIGH. If the falling edge of CLOCK samples either or both $\overline{WE}_0$ or $\overline{WE}_1$ LOW, a self timed byte write mechanism is triggered. Data is written from the data-in latch into the memory array at the corresponding address.

Note that the $\overline{OE}$ signal must be HIGH for a proper write as the $\overline{WE}_0$ and $\overline{WE}_1$ signals do not three-state the outputs. A die coat insures alpha immunity.

9.2 Operation

Reading the device is accomplished by taking the appropriate WE HIGH and OE LOW. On the rising edge of CLOCK, addresses A0 through A13 are loaded into the input registers. A memory access occurs, and data is held after a read cycle beyond the next rising edge of CLOCK to meet the hold time requirement of the microprocessor.

To write the device, OE must be taken HIGH. If the falling edge of CLOCK samples one or both of WE_0 or WE_1 LOW, a self-timed byte-write mechanism is triggered. Data is written from the data-in latch into the memory array at the corresponding address.

OE must be taken HIGH for a proper write because the write enables do not three-state the outputs.

9.3 Signal Descriptions

A<13:0> *Address Inputs:* Addresses on inputs A<13:0> are loaded into the address registers on the rising edge of CLOCK.

I/O<15:0> *Data Inputs/Outputs:* The 16 bidirectional data I/O pins are input signals during write accesses and output signals during read accesses. Data direction is controlled by the output enable pin, $\overline{OE}$.

$\overline{WE}$<1:0> *Write Enables:* The write enables initiate the self-timed write mechanism when they are sampled LOW on the falling edge of CLOCK. $\overline{WE}_0$ controls byte writing on data lines I/O<7:0> and $\overline{WE}_1$ controls data lines I/O<15:8>.

$\overline{OE}$ *Output Enable:* The output enable pin controls the output drivers of the bidirectional data lines. To begin a read access, $\overline{OE}$ is taken LOW to enable the output drivers. To begin a write access, $\overline{OE}$ is taken HIGH to three-state the output drivers.

CLOCK *Clock input:* CLOCK is the system clock input and is the same signal used by the microprocessor.

SPARC MBus CPU Modules

ROSS Technology offers the hyperSPARC RT600 and the original CY7C600 families of processors in several varieties of SPARC MBus module products. These central processing unit (CPU) modules provide many advantages, such as high reliability and ease of system upgrade. Design issues such as clock skew, chip to chip timing, and signal termination are solved, allowing engineering effort to be invested in system-specific design issues.

ROSS SPARC modules allow ease of system upgrade due to the use of SPARC-standard MBus connectors as well as specialty connectors for height-sensitive applications. This allows interchangeability between MBus modules, and enables system designs to be upgraded to new processor technology with little or no hardware modifications. The hyperSPARC-based CPU modules have the added advantage of hyperSPARC's dual clock architecture, which allows the CPU to run at a clock speed higher than that of the system. This decoupling of processor clock from the system clock allows a single system design to support a wide range of performance levels without modification.

This chapter provides a general description of the ROSS Technology SPARC CPU modules using the RT600 and CY7C600 processor families. For further information, please refer to the data sheet for the specific module of interest.

10.1 hyperSPARC Modules

hyperSPARC-based modules offer the advantages of this second-generation SPARC processor family. These advantages include:

- High-performance, superscalar hyperSPARC processor(s) with integrated FPU and 8-Kbyte instruction cache
- Full multiprocessing capability
  - Hardware support for symmetric, shared-memory multiprocessing
  - Level 2 MBus support for cache consistency
  - Direct data intervention and reflective memory support
- Compliant to SPARC Instruction Set Architecture Version 8, SPARC Reference MMU, and Level 2 MBus Module Specification
- Dual-clock architecture
  - CPU scalable from 55 to 80 MHz
- Module design
  - 3.3V logic level for IMB to reduce power and increase speed
  - Two power and two ground planes
  - Minimum-skew clock distribution
  - MBus-standard form factor: 3.30″ (8.34 cm.) x 5.78″ (14.67 cm.)
  - TAB packaging technology for a more compact design
- MBus connector

  - SPARC standard
  - Separate power and ground blades (100 active pins)
  - Designed for high frequency (low capacitance, low inductance)
- Specialty connectors
  - Low profile MBus connector
  - PGA-pin MBus connector

10.1.1 hyperSPARC Module Description

The hyperSPARC module is a complete SPARC CPU, including on-board primary and secondary cache memories. It is packaged as a compact PCB and interfaces to the remainder of the system via a SPARC-standard MBus connector (or special low profile MBus connector). Each RT62XX module consists of one or two high-speed RT620 CPUs supported with a corresponding RT625 Cache Controller, Memory Management, and Tag Unit (CMTU) and two or four 16-Kbyte x 32-bit RT627 Cache Data Units (CDUs) for either 128- or 256-Kbyte of cache per processor. IC components are packaged using tape automated bonding (TAB) technology for a compact footprint and higher frequency of operation.

RT62XX modules interface to the rest of the system via the SPARC MBus and conforms to the SPARC Reference MMU. This standardization allows a RT62XX module to be replaced by other ROSS SPARC MBus-based CPU modules without having to modify any portion of the memory system or I/O. This CPU "building block" strategy not only decreases the user's time-to-market, but also provides a mechanism for upgrading in the field.

Table 10–22. hyperSPARC Modules

Module Number	Number of processors	Cache (Per Processor)	Connector Type
RT6221K	1	128-Kbyte	MBus
RT6224K	1	256-Kbyte	MBus
RT6226K	2	256-Kbyte	MBus
RT6236K	2	256-Kbyte	PGA pin
RT6246K	2	256-Kbyte	low-profile MBus

10.1.2 hyperSPARC Module Design

10.1.2.1 Advanced Packaging Technology

The hyperSPARC Module utilizes an advanced packaging technology known as tape automated bonding (TAB). Copper leads supported by polyimide carrier connect the chip directly to a printed circuit board, eliminating the traditional IC package. In addition to eliminating one layer of connections, TAB provides higher density on the PCB and improved electrical performance for higher clock speeds.

10.1.2.2 Clock Distribution

RT62XX modules use two module clock signals (MCLK0 and MCLK1) as defined in the MBus Specification. In order to minimize clock skew, traces have been carefully routed. All clock lines are routed on inner layers of the module PCB, and their impedances have been matched. The MBus clock lines have diode termination to reduce signal undershoot and overshoot and the IMB clock lines use a parallel resistive termination of 60Ω.

10.1.2.3 MBus Connector

With the exception of the RT6236K and the RT6246K, the RT62XX interface utilizes the 100-pin SPARC MBus connector, which is a two-row male connector with 0.050″ spacing (AMP part number 121354–4 or

Fujitsu part number FCN–264P100–G/C). The connector is a controlled impedance-type, based on a microstrip configuration that provides a controlled characteristic impedance plus very low inductance and capacitance. Separate power and ground blades are provided for isolation to prevent noise.

10.1.2.4 PGA Pin Connector (RT6236K Only)

The RT6236K interface is via a 120-pin SPARC MBus PGA footprint. These pins allow thru-hole attachment to the motherboard which, in combination with a lower profile heatsink, provides a low-height solution for space-sensitive applications. This MBus connection supports Level 2 MBus.

10.1.2.5 Low Profile MBus Connector (RT6246K Only)

The RT6246K interface is via the 100-pin SPARC low-profile MBus connector, which is a two-row male connector with 0.050" spacing (AMP part number 121344–4). The connector is a controlled impedance-type (50Ω ±10%) based on a microstrip configuration that provides a controlled characteristic impedance plus very low inductance and capacitance. Separate power and ground blades are provided for isolation to prevent noise. This MBus connector supports Level 2 MBus.

10.1.2.6 Mating MBus Connector (System Interface Board)

With the exception of the RT6236K, RT62XX modules connect to the system interface through the standard MBus female connector (vertical receptacle assembly, AMP part number 121340–4 or Fujitsu FCN–264J100–G/0).

10.1.2.7 Reset and Interrupt Signals

A power-on reset signal is generated to the module from the MBus via the $\overline{\text{RSTIN}}$ signal. Level-sensitive interrupts (15 max) are generated to the RT620 via the IRL[3:0] lines from the MBus. A value of 0000b means that there is no interrupt, while a value of 1111b means a non-maskable interrupt (NMI) is being asserted. IRL values between 0 and 15 represent interrupt requests that can be masked by the processor.

10.1.2.8 MBus Request and Grant Signals

One (or two sets) of request and grant signals are generated to/from the RT62XX module to arbitration logic on the motherboard.

10.1.2.9 MBus SCAN Test Feature

RT62XX modules support the SCAN test feature of the MBus through the hyperSPARC Test Access Port (TAP) interface. For more details, refer to the TAP Interface Specification available from ROSS Technology.

10.1.3 hyperSPARC System Design Considerations

RT62XX modules implement a subset of all possible MBus signals; signals that are optional and/or specifically for multiple processor modules may not supported. Systems designers should be aware of these assignments in order to more easily upgrade to other and future MBus modules. RT62XX MBus pinout assignments and a list of reserved MBus pins are detailed in the RT62XX data sheets.

All MAD, bused control, and point-to-point control signals use 8-mA drivers except for $\overline{\text{MAS}}$, which uses 16-mA. The $\overline{\text{MSH}}$, $\overline{\text{MIH}}$ and $\overline{\text{AERR}}$ signals use an open-drain driver.

The following pull-up resistors are recommended for the MBus signals: $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ are pulled up to 5V with a 620Ω resistor; $\overline{\text{AERR}}$ is pulled up to 5V with a 1.5-kΩ resistor; all other MBus signals are pulled up to 5V with 10-kΩ resistors.

As the frequency of operation increases, transmission line effects play a bigger role. Care must be taken to keep skew between any two clock signals at the MBus connector within the specifications given in the Synchronous Signals table in the AC Characteristics section. MBus signal lines must be routed carefully to minimize crosstalk and interference. A thorough SPICE or other transmission-line analysis of the motherboard design is recommended.

Use of HH Smith #4387 (3/4" length by 1/4" OD) stand-offs on the motherboard or equivalent is recommended to support the module and prevent damage to the connector.

10.2 CYM600X Modules

CYM600X modules provide one or two complete SPARC MBus-based CPUs utilizing the original CY7C600 chipset. It is packaged as a compact PCB and interfaces to the remainder of the system via a SPARC-standard MBus connector. Each CPU on a CYM600X module consists of a CY7C601 high-speed Integer Unit, a CY7C602 Floating-Point Unit, a CY7C604 or CY7C605 Cache Controller and Memory Management Unit, and two 16-Kbyte x 16 CY7C157 Cache Storage Units (providing a 64-Kbyte cache for each CPU).

CYM600X modules interface to the rest of the system via the SPARC MBus and conforms to the SPARC Reference MMU. This standardization allows a CYM600X module to be replaced by other SPARC MBus-based CPU modules without having to modify any portion of the memory system or I/O. This CPU "building block" strategy not only decreases the user's time to market, but also provides a mechanism for upgrading in the field. Some of the features of the CYM600X module family are:

- High performance SPARC RISC CPU CY7C600 family
  - CY7C601 Integer Unit (IU)
  - CY7C602 Floating-Point Unit (FPU)
  - Two Cache Controller and Memory Management Unit (CMU) options:
    CY7C604 CMU for uniprocessor systems
    CY7C605 CMU-MP for multiprocessor systems
  - Two CY7C157 Cache Storage Units (CSUs) provide 64-Kbyte cache for each CPU
- SPARC standard
  - SPARC Instruction Set Architecture (ISA) compliant
  - Conforms to SPARC Reference MMU architecture
  - Conforms to SPARC MBus Module Specification (Level 1 and Level 2)
- Each module features:
  - SPARC integer and floating-point processing
  - Zero-wait-state, 64-Kbyte cache per processor
  - 4-Kbyte page virtual memory management
  - Surface-mount packaging for more compact design
  - Provides simple CPU upgrade path at module level
- MBus Module designs feature:
  - Two power and two ground planes
  - Minimum-skew clock distribution
  - MBus-standard form factor: 3.30" (8.34 cm) x 5.78" (14.67 cm)
- SPARC-standard MBus connector
  - Separate power and ground blades (100 active pins)
  - Designed for high frequency (low capacitance, low inductance)

Table 10–23. CYM600X Modules

Module Number	Number of processors	CMU	Module Type
CYM6001K	1	CY7C604	Level 1 MBus Module
CYM6002K	2	CY7C605	Level 2 MBus Module
CYM6003K	1	CY7C605	Level 2 MBus Module
CYM6111	1	CY7C604	Multi-Die Package

10.2.1 CYM600X Module Design

10.2.1.1 Clock Distribution

CYM600X modules use four module clock signals (MCLK0, MCLK1, MCLK2, and MCLK3) as defined in the MBus Specification. In order to minimize clock skew, traces have been carefully routed. All clock lines are routed on inner layers of the module PCB, and their impedances have been matched. All clock lines have diode termination to reduce signal undershoot and overshoot.

10.2.1.2 MBus Connector

The CYM600X interface is via the 100-pin SPARC MBus connector, which is a two-row male connector with 0.050" spacing (AMP "microstrip" part number 121354–4 or Fujitsu part number FCN–264P100–G/C). The connector is a controlled impedance-type based on a microstrip configuration which provides a controlled characteristic impedance plus very low inductance and capacitance. Separate power and ground blades are provided for isolation to prevent noise.

10.2.1.3 Mating MBus Connector (System Interface Board)

The module connects to the system interface by means of a standard MBus female connector (AMP vertical receptacle assembly, part number 121340–4).

10.2.1.4 Reset and Interrupt Signals

A power-on reset signal is generated to the module from the MBus via the $\overline{\text{RSTIN}}$ signal. Level-sensitive interrupts (15 max) are generated to the CY7C601 via the IRL[3:0] lines from the MBus. A value of 0000b means that there is no interrupt while a value of 1111b means a non-maskable interrupt (NMI) is being asserted. IRL values between 0 and 15 represent interrupt requests that can be masked by the processor.

10.2.2 System Design Considerations

CYM600X modules implement a subset of all possible MBus signals; signals that are optional and/or specifically for multiprocessing, dual CPUs, and SCAN test capabilities may not be supported. Systems designers should be aware of these assignments in order to more easily upgrade to other and future MBus modules. MBus pinout assignments and a list of reserved MBus pins are detailed in the data sheet for the specific CYM600X module.

All MAD, bused control, and point-to-point control signals use 8-mA drivers (with the exception of $\overline{\text{MAS}}$, which uses a 16-mA driver). The $\overline{\text{AERR}}$ signal uses an open-drain driver.

The following pull-up resistors are recommended for the MBus signals: $\overline{\text{MSH}}$ is pulled up to 5V with a 620-Ω resistor, and $\overline{\text{AERR}}$ is pulled up to 5V with a 1.5-kΩ resistor; all other MBus signals are pulled up to 5V with 10-kΩ resistors.

As the frequency of operation increases, transmission line effects play a bigger role. Care must be taken to keep skew between any two clock signals at the MBus connector within the specifications given in the Synchronous signals table. MBus signal lines must be routed carefully to minimize crosstalk and interference. A thorough SPICE or other transmission-line analysis of the motherboard design is recommended.

Use of HH Smith #4387 (3/4" length by 1/4" OD) stand-offs on the motherboard or equivalent is recommended to support the module and prevent damage to the connector.

10.3 CYM6111 Multi-Die Package CPU

The CYM6111 is a complete SPARC CPU mounted in a single surface mount package. It utilizes an advanced multi-die packaging (MDP) technology that provides single-chip integration with multiple die. The CYM6111 interfaces to the remainder of the system utilizing the SPARC-standard MBus. The CPU in the CYM6111 consists of a high-speed Integer Unit (CY7C601), a Floating-Point Unit (CY7C602), a Cache Controller and Memory Management Unit (CY7C604), and two 16-Kbyte x 16-bit CY7C157 CSUs (providing a 64-Kbyte cache for the CPU). These die are packaged in a single 256-pin CQFP package for a compact footprint and lower power consumption.

10.3.1 Multi-Die Packaging Technology

Multi-die packaging (MDP) technology provides low inductance, low capacitance interconnect over waferscale distances. Die are mounted directly on a silicon circuit board, eliminating much of the package related capacitances. The dense interconnect and ability of route interconnect under the die without interference from thermal vias enable extremely close placement of ICs. This results in net trace lengths typically 4 to 8 times shorter than PCB implementations, and therefore, lower load capacitance and inductance to be driven by chip outputs.

In the CYM6111, virtually all of the interconnections between the individual die are done through the package's silicon substrate. The only connections to the package are for the signals that must interface the CPU to the rest of the system (i.e., MBus). In addition to the shorter trace lengths, routing interconnections within the MDP package also relaxes I/O limitations that are typically imposed on high bandwidth VLSI devices by the physical layout of the die pads around the perimeter of the package

Intra-package trace connections also improve the integrity of signal transmissions. Noise on power and ground signals, as well as skew on clock lines are greatly reduced due to the reduction in trace lengths and controlled electrical environment.

MBus Operation

The SPARC MBus is a high-speed interface designed to connect SPARC processor modules to physical memory modules and I/O modules. The MBus is an integrated circuit interface, and is not intended to operate as a general expansion bus across a system backplane. It is intended to operate as an interface between modules and interface circuitry located on a single printed circuit board within a carefully controlled geographical area. Modules consist of one or more integrated circuits which contain the MBus interface. A CPU based upon the hyperSPARC is an example of such a module.

In addition to the electrical and architectural characteristics specified by MBus, the MBus also identifies mechanical guidelines for MBus processor modules. This includes length, width, and height dimensions as well as the MBus-standard module connector. Details on ROSS' MBus CPU modules are given elsewhere in this User's Guide.

MBus is divided into two levels of implementation: Level 1 and Level 2. Level 1 includes the basic MBus signals and transactions needed to support a uniprocessor system. Level 2 introduces additional signals and transactions needed to design a symmetric, cache-coherent, shared-memory multiprocessor system. The CY7C604 supports Level 1, while the CY7C605 and RT625 support both Level 1 and Level 2.

The *SPARC MBus Interface Specification* (available from SPARC International) provides further information on MBus. This section describes the MBus as it pertains to the operation of the CY7C604, CY7C605 and RT625 and may not explicitly represent all MBus signals and functionality

11.1 MBus Principles

- Fully synchronous bus.
- Multiplexed 64-bit address/data bus.
- 64 Gigabytes of physical memory address space.
- All signals changed and sampled on the rising edge of clock.
- Bus arbiter is a separate bus unit.
- Centralized arbitration, reset, interrupt distribution, and clock distribution.
- Peer level (multi-master) bus protocol.
- Overlapped arbitration with bus "parking."
- Shared memory multiprocessor signals and transactions (Level 2).
- Write-invalidate type of cache-consistency protocol (Level 2).

MBus assumes that there are central functional elements to perform reset, arbitration, interrupt distribution, timeout, and MBus clock generation. Refer to the *SPARC MBus Interface Specification* for a detailed description of MBus as defined for system implementation.

11.1.1 MBus Level 1 Overview

Level 1 MBus supports two transactions: read and write. These transactions simply read or write a specified SIZE of bytes from a specified physical address. These transactions are supported using a subset of the

MBus signals, namely a 64-bit multiplexed address/data bus (MAD<63:0>, an address strobe signal (MAS), and an encoded acknowledge on the three signals MRDY, MRTY, and MERR). Additional Level 1 signals support arbitration for modules (MBR, MBG, and MBB), reset (POR on CY7C604/605, RSTIN in the SPARC MBus Specification and RT625), and memory error (CMER on CY7C604/605, AERR in the SPARC MBus Specification and RT625). The MBus reference clock (CLK) completes the signal requirements for a Level 1 system.

11.1.2 MBus Level 2 Overview

The Level 2 MBus includes all Level 1 transactions and signals and adds four transactions and two signals to support cache coherency. This is to facilitate the design of symmetric, shared memory, multiprocessor systems. In Level 1, details of the cache operations inside modules are not visible to the MBus transactions. This changes with Level 2, where many aspects of the cache operation are assumed as part of the new MBus transactions. To participate in cache consistent sharing using Level 2 transactions, a cache must have a copy-back with write-allocate policy and have a block or sub-block of size 32 bytes. An ownership-based protocol is employed in which cache lines are assumed to be described as being in one of five states: invalid (I), exclusive clean (EC), exclusive modified (EM), shared clean (SC), and shared modified (SM). A simplified state transition diagram is shown below in *Figure 11–1.*

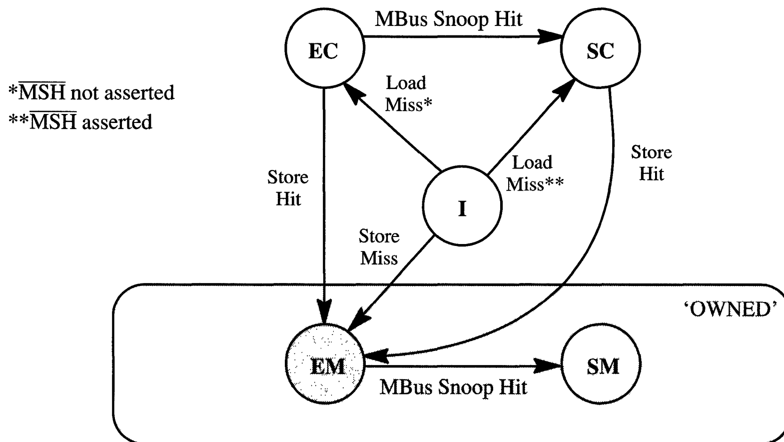


Figure 11–1. Level 2 MBus Cache State Diagram

The additional transactions present in Level 2 systems are Coherent Read, Coherent Invalidate, Coherent Read and Invalidate, and Coherent Write and Invalidate. The two additional signals are MBus shared (MSH) and MBus inhibit (MIH). All coherent transactions have SIZE = 32 bytes, except Coherent Write and Invalidate which will support any valid MBus size. The cache coherency protocol is a "write-invalidate" protocol, where the writing cache issues a Coherent Invalidate transaction if the cache line is not exclusive. This indicates to all caches that they should invalidate the cache line since it contains "stale data" after the write completes. All caches "snoop" Coherent Read transactions and assert MSH if the address of the transaction is present in their cache. By observing the MSH signal, other caches can update the state of the cache lines they hold. If a cache is the "owner," it asserts the signal MIH to prevent memory from sending data. The cache then supplies the data to the requesting cache (this is referred to as direct data intervention). Coherent Read and Invalidate and Coherent Write and Invalidate are simply the combination of a Coherent Invalidate and either a Coherent Read or a Write. Their purpose is to reduce the quantity of MBus transactions needed and thus conserve bandwidth.

11.1.3 MBus Physical Signal Summary

Table 11–1 summarizes the physical signals that comprise the MBus interface. Bus agents (master, slave, arbiter, etc.) are listed in the output or input column of *Table 11–1* to denote whether the signal is an input or output for that bus agent. The “line type” column of *Table 11–1* lists signals as bussed or dedicated. Bussed signals are those driven or received by multiple bus agents, whereas dedicated signals are driven by one agent and received by only one other.

Table 11–1. MBus Signal Summary

Symbol	Description	Output	Input	Line Type	Signal Type
MCLK	MBus Clock	Clock buffer	Master/Slave/ Arbiter	dedicated	BS
MAD<63:0>	MBus Address/Data	Master/Slave	Master/Slave	bussed	TS
$\overline{\text{MAS}}$	MBus Address Strobe	Master	Slave	bussed	TS
$\overline{\text{MERR}}$	MBus Error	Slave	Master	bussed	TS
$\overline{\text{MRDY}}$	MBus Ready	Slave	Master	bussed	TS
$\overline{\text{MRTY}}$	MBus *Retry*	Slave	Master	bussed	TS
$\overline{\text{MBR}}$	MBus Bus Request	Master	Arbiter	dedicated	BS
$\overline{\text{MBG}}$	MBus Bus Grant	Arbiter	Master	dedicated	BS
$\overline{\text{MBB}}$	MBus Bus Busy	Master	Arbiter/Master	bussed	TS
$\overline{\text{MSH}}$*	MBus Shared	Bus Watcher	Master	bussed	OD
$\overline{\text{MIH}}$*	Memory Inhibit	Bus Watcher	Master/Memory	bussed	TS
IRL[3:0]	Interrupt Level	Interrupt Logic	CPU Module	dedicated	BS
ID[3:0]	Module Identifier	System	MBus Module	dedicated	BS
$\overline{\text{AERR}}$	Asynchronous Error	Module	Interrupt Logic	dedicated	OD
$\overline{\text{RSTIN}}$	Module Reset In	Reset Logic	Master/Slave	impl dependent	BS
TDI	Scan Data In	System	Module	dedicated	BS
TDO	Scan Data Out	Module	System	dedicated	BS
TCK	Scan Clock	System	Module	dedicated	BS
$\overline{\text{TMS}}$	Scan TAP Control	System	Module	dedicated	BS
$\overline{\text{TRST}}$	Scan TAP Reset	System	Module	dedicated	BS

TS: Three-state BS: Bi-state OD: Open Drain
* Level 2 ONLY

MCLK MBus master clock. The distribution of the MCLK signal in a system is implementation dependent. For example, depending on the connector, each module on the MBus may be given one or more identical MCLK lines which could originate from a single clock generator.

MAD<63:0> Memory address and data. During the address phase, MAD<35:0> contains the physical address (PA[35:0]). The remaining signals (MAD<63:36>) on the bus contain the transaction-specific information which will be described in *Section 11.1.4.* During the data phase, MAD<63:0> contains the data of the transfer. The bytes are organized as shown in *Figure 11–2.* For transactions involving less than a double word (8-bytes), the data must be aligned. For example, all even-addressed words will be aligned on MAD<63:32> whereas all odd-addressed words will be aligned on MAD<31:0>. As another example, byte address 2 of an odd-addressed word will be carried on MAD<15:8> i.e. byte 6 on the MBus. Unused data lines during the data phase are undefined.

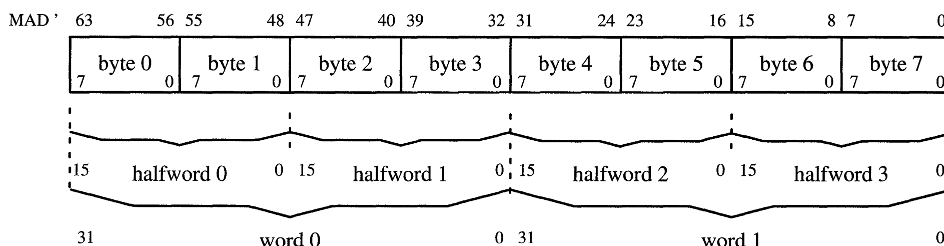


Figure 11–2. Byte Organization

$\overline{\text{MAS}}$ Memory address strobe. This signal is asserted by the bus master during the very first cycle of a bus transaction. The cycle in which it is asserted is referred to as the "address cycle" or "address phase" of the transaction. For transactions that receive a *Relinquish and Retry* (*R&R*) or a *Retry* acknowledgment, $\overline{\text{MAS}}$ will be asserted again until the transaction gets a normal or error acknowledgment. Other cycle timing is discussed with respect to $\overline{\text{MAS}}$. e.g., A+2 indicates the second cycle after $\overline{\text{MAS}}$ assertion. i.e., $\overline{\text{MAS}}$ assertion is A+0.

$\overline{\text{MRDY}}$ MBus ready transaction status bit. This bit is one of the three bits used to encode the transaction status as shown in *Table 11–2.* The encoding with $\overline{\text{MRDY}}$ asserted alone indicates that valid data has been transferred. The three status bits ($\overline{\text{MRDY}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MERR}}$) are normally asserted by the addressed slave. The ERROR2 (timeout) acknowledgement will be asserted by the bus monitor.

$\overline{\text{MRTY}}$ MBus *Retry* transaction status bit. This bit is one of the three bits used to encode the transaction status as shown in *Table 11–2.* The encoding with $\overline{\text{MRTY}}$ asserted alone indicates that the slave wants the master to abort the current transaction immediately and start over. The master will relinquish bus ownership upon this type of a *Retry* acknowledgment. Note that if any type of acknowledgement other than "valid data transfer" is issued, the cycle it is issued is the last cycle, regardless of how many further acknowledgement cycles would normally occur. The three status bits ($\overline{\text{MRDY}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MERR}}$) are normally asserted by the addressed slave.

Table 11–2. Transaction Status Bit Encoding

$\overline{\text{MERR}}$	$\overline{\text{MRDY}}$	$\overline{\text{MRTY}}$	Meaning
H	H	H	idle cycle
H	H	L	*Relinquish and Retry* (*R&R*)
H	L	H	Valid Data Transfer
H	L	L	reserved
L	H	H	ERROR1 => Bus Error
L	H	L	ERROR2 => Timeout
L	L	H	ERROR3 => Uncorrectable
L	L	L	*Retry*

$\overline{\text{MERR}}$ MBus error transaction status bit. This bit is one of the three bits used to encode the transaction status as shown in *Table 11–2*. The encoding with $\overline{\text{MERR}}$ asserted alone indicates that a bus error (or other system implementation specific error) has occurred. Note that if any type of acknowledgement other than "valid data transfer" is issued, the cycle it is issued is the last cycle, regardless of how many further acknowledgement cycles would normally occur. The three status bits ($\overline{\text{MRDY}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MERR}}$) are normally asserted by the addressed slave.

$\overline{\text{MBR}}$ MBus request signal. This signal is asserted by an MBus master to acquire bus ownership. There is one unique $\overline{\text{MBR}}$ signal per master.

$\overline{\text{MBG}}$ MBus grant signal. This signal is asserted by the external arbiter when the particular MBus master is granted the bus. There is one unique $\overline{\text{MBG}}$ signal per master.

$\overline{\text{MBB}}$ MBus busy signal. This signal is asserted as an output during the entire transaction, from and including the assertion of $\overline{\text{MAS}}$ to the assertion of the last $\overline{\text{MRDY}}$ or first other acknowledgment which terminates the transaction (such as an error acknowledgment). If a master wishes to keep the bus and perform several transactions without releasing the bus between them, it keeps $\overline{\text{MBB}}$ asserted until the last $\overline{\text{MRDY}}$ of the last transaction of the group. The potential master device samples this signal in order to obtain the bus ownership as soon as the current master releases the bus. $\overline{\text{MBB}}$ locks arbitration on a particular MBus. A master is allowed to assert $\overline{\text{MBB}}$ prior to the assertion of $\overline{\text{MAS}}$ (to hold the bus). It is also allowed to keep $\overline{\text{MBB}}$ asserted after the assertion of the last acknowledgment in a few special cases for performance reasons. This continued assertion of $\overline{\text{MBB}}$ should only occur while $\overline{\text{MBG}}$ is still parked on the current master. The $\overline{\text{MAS}}$ of the transaction prompting the continued assertion of $\overline{\text{MBB}}$ should be generated quickly (2 cycles is the recommended maximum delay). For more details on arbitration, see *Section 11.1.9*.

$\overline{\text{MIH}}$ Memory inhibit signal. This signal is only present in Level 2 MBus modules. It is asserted by the owner of a cache block at the beginning of the second cycle after it receives the address (its A+2 cycle)† to inform the main memory that the current Coherent Read or Coherent Read and Invalidate request should be ignored. This is because the owner, not the memory, will be responsible for delivering the cache data block. If no device asserts $\overline{\text{MIH}}$ during its A+2 cycle, main memory will be responsible for delivering the data. If main memory starts delivering data and $\overline{\text{MIH}}$ is asserted, the memory delivery shall be aborted immediately. Any data which was received from main memory in this case should be ignored. It should be noted that

† See SPARC International *SPARC MBus Specification* for notes to designers who wish to avoid the A+2 requirement.

because of the restriction on MIH either occurring simultaneously with or before MRDY, there will be at most two cycles worth of data to ignore. While MIH is sourced by a module (for one cycle) two cycles after receiving MAS it may be observed by a cache in the interval from its A+2 until it observes an acknowledgement. This variation in where MIH can be observed is due to the possibility for MBus repeaters and modules that do not meet the A+2 timing.

MSH Cache block shared signal (wired-or, open-drain). This signal is only present in Level 2 MBus modules. Whenever a Coherent Read transaction appears on the bus, the bus monitor of each processor module should immediately search its cache directory. If a valid copy is found, the MSH signal should be asserted in the second cycle after the address is received (its A+2 cycle). The MSH signal is also sampled (observed) by external caches. It is asserted as an output (for a single cycle) if there is a cache hit in the snooping directory. While MSH is sourced by a module two cycles † after receiving MAS it may be observed by a cache from its A+2 until it observes an acknowledgement. This variation in where MSH may be observed is due to the possibility for MBus repeaters and modules that do not meet the A+2 timing. Signals are sourced at the beginning and observed or sampled at the end of a cycle. Due to the open drain nature of MSH and the associated slow rise time, while the signal is driven active low for one cycle, it can be observed active low for up to two cycles and the trailing or rising edge of the signal is considered asynchronous.

RSTIN Module reset input signal. This signal should reset all logic on a module to its initial state, and ensure that all MBus signals are inactive or tri-state as appropriate. The minimum assertion time of RSTIN will be system implementation dependent, although the default assertion time will be 1024 MBus clock (MCLK) periods (25.6 microseconds). RSTIN should be treated as asynchronous.

AERR Module asynchronous error detect out signal. This signal is asserted by the module as a level to indicate that an asynchronous error was detected by the module. It remains asserted until a software initiated action resets a bit that is maintaining the signal assertion. AERR is open drain because several modules could assert it simultaneously. AERR may be asynchronous.

INTOUT Module interrupt out signal. This signal is asserted by an I/O module as a level to indicate an interrupt request to the system. It remains asserted until a software initiated action resets a bit that is maintaining the signal assertion. INTOUT may be asynchronous. This signal is **optional**.

IRL[3:0] These pins carry the interrupt request level inputs to a SPARC integer unit. They are only used by processor modules. IRL[3:0] may be asynchronous. Each processor module receives a dedicated set of IRL[3:0].

ID[3:0] These pins carry the module identifier. These signals are not needed by Level 1 processor modules, which have a default ID of 0xF, and are **optional** for other modules who may obtain this information by other means. ID[3:0] is reflected as MID[3:0] during the address phase of every transaction, and is also used to identify a unique address range for module identification, initialization and configuration.

If modules do not have ID[3:0] input pins, the system must provide a function that allows each module to obtain a unique ID[3:0] value in an internal ID[3:0] register. One way this can be accomplished is to have a logic function with a known MBus

address attached to the MBus arbiter. There are unique $\overline{\text{MBR}}$ and $\overline{\text{MBG}}$ lines per module and so this function when addressed, would return a unique ID[3:0] value on MAD, based on which module's $\overline{\text{MBG}}$ was asserted just prior to the beginning of the ID read transaction when $\overline{\text{MBB}}$ was deasserted.

TDI This signal is an input to the module and is used for receiving scan data from the system. This is the input of the scan ring and should not be inverted or gated. This signal changes on the falling edge of TCK and should be sampled on the rising edge of TCK. Scan is **optional**.

TDO This signal is an output of the module and is used for sending the scan data to the system. This is the output of the scan ring and should not be inverted or gated. **TDO** should be driven on the falling edge of TCK and will be sampled on the rising edge of TCK. Scan is **optional**.

TCK This signal is used to supply the clock to the scan ring on the module. (Typically 5 MHz). Scan is **optional**.

$\overline{\text{TMS}}$ This signal is an input to the module. It is used to control the TAP controller state machine. Scan is **optional**.

$\overline{\text{TRST}}$ This signal is an input to the module. It is used to reset the TAP controller state machine. Scan is **optional**.

11.1.4 MBus Multiplexed Signal Summary

The MBus is a 64-bit multiplexed address/data bus. *Table 11–3* summarizes the multiplexed MBus signals. All multiplexed signals are active HIGH (true when "1").

Table 11–3. Multiplexed Signal Summary

Multiplexed Signals (valid during address phase)		
Signal Name	Physical Signal	Signal Description
PA[35:0]	MAD[35:0]	Physical address for current transaction
TYPE[3:0]	MAD[39:36]	Transaction type
SIZE[2:0]	MAD[42:40]	Transaction data size
MC	MAD[43]	Data cacheable (advisory)
MLOCK	MAD[44]	Bus lock indicator (advisory)
MBL	MAD[45]	Boot mode / local bus (advisory)(optional)
VA[19:12]	MAD[53:46]	Virtual address (optional)(Level 2)
reserved	MAD[59:54]	reserved for future expansion
MID[3:0]	MAD[63:60]	Module Identifier

11.1.5 MBus Address Cycle

The address cycle of an MBus transaction consists of a 36-bit physical address and 28 bits of control and transaction information. *Figure 11–3* illustrates the MBus address cycle. The address fields of the MBus address cycle follow.

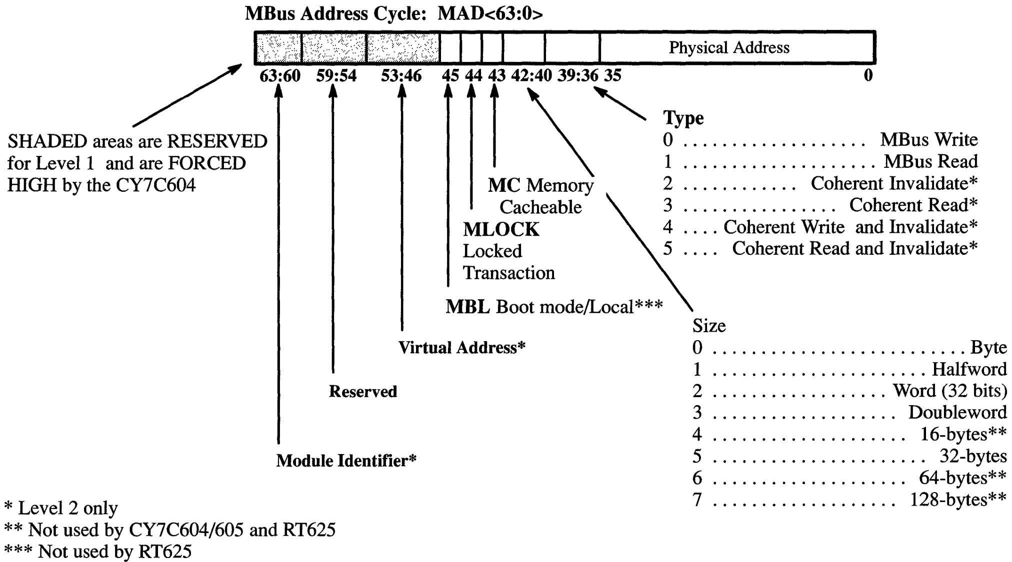


Figure 11–3. MBus Address Cycle

PA[35:0] Physical address of current transaction which is multiplexed on MAD<35:0>.

TYPE[3:0] The transaction types are encoded in bits MAD<39:36> as shown below in *Table 11–4*. Most of the transaction types are reserved.

Table 11–4. TYPE Encodings

TYPE[3]	TYPE[2]	TYPE[1]	TYPE[0]	Data Size	Transaction Type
H	H	H	H	–	reserved
H	H	H	L	–	reserved
H	H	L	H	–	reserved
H	H	L	L	–	reserved
H	L	H	H	–	reserved
H	L	H	L	–	reserved
H	L	L	H	–	reserved
H	L	L	L	–	reserved
L	H	H	H	–	reserved
L	H	H	L	–	reserved
L	H	L	H	32B	Coherent Read & Invalidate(CRI)
L	H	L	L	any*	Coherent Write & Invalidate(CWI)
L	L	H	H	32B	Coherent Read(CR)
L	L	H	L	32B	Coherent Invalidate(CI)
L	L	L	H	any*	Read (RD)
L	L	L	L	any*	Write (WR)

*as defined by the SIZE signals in *Table 11–5*.

SIZE[2:0] The transaction data SIZE information is encoded in bits MAD<42:40>. The size field is encoded as $\log_2$ [number of data bytes transferred]. The encoding of the SIZE bits is shown in *Table 11–5.*

For transactions with SIZE greater than 8 bytes, more than one $\overline{\text{MRDY}}$ will be needed. For transactions with SIZE less than or equal to 8 bytes, unneeded address lines are undefined (e.g. for SIZE = 8 bytes, MAD<2:0> are undefined). The CY7C604/605 and RT625 support all non-burst transaction sizes, and a 32-byte burst transfer.

Table 11–5. SIZE Encodings

SIZE[2]	SIZE[1]	SIZE[0]	Transaction Size
L	L	L	Byte
L	L	H	Halfword (2 bytes)
L	H	L	Word (4 bytes)
L	H	H	Doubleword (8 bytes)
H	L	L	16-byte Burst
H	L	H	32-byte Burst
H	H	L	64-byte Burst
H	H	H	128-byte Burst

MC Cacheable indicator (multiplexed on MAD[43]). When this signal is asserted, it indicates the state of the cacheable bit for the address of the transaction in the module MMU. This is an **advisory** bit, not used by MBus transactions, but possibly of use to the slave device.

An example use of C would be to inform a second level cache of the cacheability state of the address of a transaction with SIZE less than 32 bytes.

MLOCK Lock indicator signal (multiplexed on MAD<44>). This bit indicates that the MBus transaction is a "locked" transaction. If the MBus master intends to lock access to a device residing on MBus (main memory is one MBus device) or some other bus connected to MBus, and perform N indivisible MBus transactions to the device, this bit needs to be asserted during the address cycles of all N MBus transactions. The locking master must keep $\overline{\text{MBB}}$ asserted during each locked cycle and not deassert it until the end of the final locked transaction (however $\overline{\text{MBB}}$ may be suspended for a time by an *R&R* acknowledgment). The deassertion of $\overline{\text{MBB}}$ signals the MBus arbiter to release the MBus. It is the final deassertion of $\overline{\text{MBB}}$ after possible intervening *R&R* acknowledgments which tells the device to release its lock. LOCK is an **advisory** bit, not used by MBus transactions directly, but possibly of use to the slave device or bus interface.

An example use of LOCK would be to "lock" an MBus master to a particular slave. If an MBus processor performed an atomic operation to a resource arbitrated externally to MBus, such as a dual-ported memory device or another bus, then the external arbiter could prevent any other (non MBus) device from accessing that resource by locking arbitration. The referenced slave device in a LOCKed transaction could be, in essence, dedicated to the requesting master. The MBus slave port interface interprets an assertion of the MBus LOCK bit as saying "become locked" and a final

deassertion of $\overline{\text{MBB}}$ at the end of the locked sequence as saying "become unlocked," and reports this information to the arbiter for the "locked" device (or bus). If the slave port supports *R&R* acknowledgments, it must know not to clear the locked state when $\overline{\text{MBB}}$ is removed due to an *R&R*.

MBL MBus boot mode/local bus indicator (multiplexed on MAD<45>). This signal is asserted by CY7C604/605 and RT625 processor modules during the address phase of boot mode transactions, or during local bus transactions (SPARC processor accesses with ASI = 0x1). It is system implementation dependent whether or not local bus transactions are employed in a system. This is an **advisory** bit, not used by MBus transactions, but possibly of use to the slave device. This bit is **optional**. If unused by an implementation it should remain deasserted. The RT625 always drives this bit LOW.

VA[19:12] Virtual Address 19 through 12 (multiplexed on MAD<53:46>). This field only applies to MBus Level 2 coherent transactions; for non-coherent transactions, these bits are undefined. This field is used to carry the virtual address bits 19 through 12 associated with the physical address of a Coherent Read transaction (bits 15:12 are used by the CY7C605 only, and bits 17:12 are used by the RT62). These bits are used by virtually indexed caches that desire to index into the dual directories via the virtual "superset" bits to avoid synonym problems. This assumes a minimum page size of 4 Kbytes in the system and maximum cache size of 1 Mbyte. Modules that choose not to provide this function or to support coherent transactions (such as a Level 1 device) should drive these lines HIGH.

reserved This 5-bit field (multiplexed on MAD<58:54>), is reserved for future MBus expansion. The lines should be driven HIGH if not used.

SUP Supervisor access indicator (multiplexed on MAD<59>). This signal is asserted by processor modules and indicates that the MBus transaction is a processor supervisor access. This is an **advisory** bit, not used by MBus transactions, but possibly of use to the slave device. This bit is **optional**. If unused by an implementation it should remain asserted. This bit is not used by the CY7C604/605 and RT625; it is always driven HIGH.

An example use of SUP would be to enable more state to be captured on processor asynchronous write errors.

MID[3:0] Module identifier signals (multiplexed on MAD<63:60>). This field is sourced by all MBus modules and reflects the value input into the module on the ID[3:0] input signals. For Level 1 processor modules this field is driven *HIGH (0xF)*. This field is observed by slave ports that wish to issue an *R&R* acknowledgment (see *Section 11.1.8.2*), so that they can identify the master with which to reconnect in a multi-master system configuration. For the CY7C605 these signals are specified by the MID field in the SCR register. For the RT625 these signals are specified by the ID[3:0] input signals.

11.1.6 MBus Data Cycle

MBus transactions consist of an address cycle followed by one or more data cycles. A single data cycle transaction is referred to as a non-burst transaction. Note that all noncacheable transactions made by the CY7C604/605 and RT625 are transferred as non-burst transactions. During non-burst read or write transac-

tions, data appears in the byte locations of the MBus as determined by the size (MAD<42:40>) and address bits MAD<2:0> (see *Figure 11–4*). The data on any unused MBus lines is undefined.

Burst transactions are used by the CY7C604/605 and RT625 for cache line transfers. Burst transactions made by the CY7C604/605 will always be on cache line boundaries (i.e., MAD<4:0> = 0 for the address cycle of a burst transaction). All burst transactions made by the CY7C604/605 and RT625 are 32 bytes (one cache line) in length.

Note: The CY7C604/605 and RT625 are designed to ensure one "implicit clock" after a MBus read transaction before it will assert an address for the next MBus transaction. This allows time for slow memory data buffers to release the MBus. Also, the RT625 will add one "implicit clock" after an MBus write transaction that ends in an *Error*, *Retry*, or *R&R*.

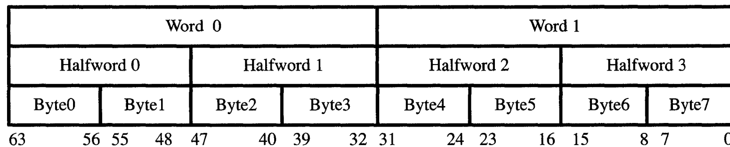


Figure 11–4. MBus Data Ordering

11.1.7 MBus Transactions

The MBus has three separate bus agents: master, slave, and arbiter. The bus arbiter is essentially a "traffic cop" for the MBus. It is external to all bus masters or slaves, and is responsible for granting bus ownership to one of the various bus masters. The algorithm by which the arbiter assigns priority to the various bus masters is left to the system designer. More information on MBus arbitration is available in *Section 11.1.9*.

The MBus bus cycle consists of an address cycle followed by one or more data cycle(s). Transaction sizes supported by MBus are: 1, 2, 4, 8, 16, 32, 64, and 128 bytes. A data transaction requiring more than one data cycle is referred to as a burst transaction.

Since the 64-bit MBus can transfer eight bytes in a single data cycle, transactions greater than eight bytes are performed as burst transactions, in which a single address phase is followed by multiple data phases. Transactions less than or equal to eight bytes are performed as non-burst transactions. Non-burst transactions consist of a single address phase and a single data phase. *Figure 11–5* illustrates an example of a burst transaction. The CY7C604/605 and RT625 support 1, 2, 4, 8, and 32-byte transactions on the MBus. The 32-byte burst transaction corresponds to the 32-byte cache line size.

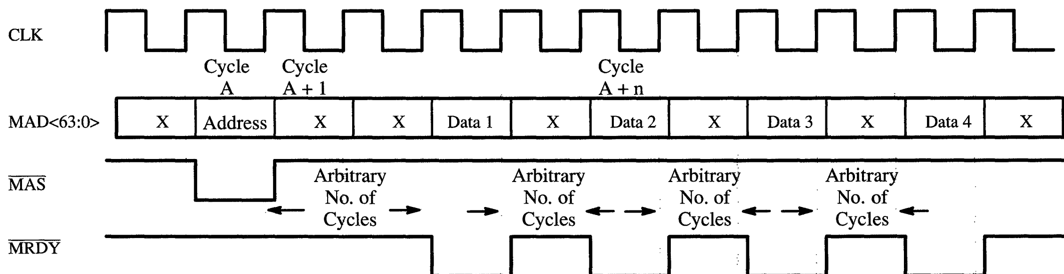


Figure 11–5. MBus Burst Transaction Example

An MBus cycle begins after the bus master has acquired the MBus and asserted $\overline{MBB}$. The bus master supplies the address and strobes the MBus address strobe ($\overline{MAS}$) for one clock period. The bus slave (usually

the memory system) acknowledges the data transfer by strobing the MRDY , MERR, and MRTY signals. MRDY is strobed for each successful data cycle. Unsuccessful data cycles are acknowledged with other combinations of the MRDY, MERR, and MRTY signals. *Table 11–6* describes the decoding of the MRDY, MERR, and MRTY signals.

All MBus transactions can be terminated by an *Error*, which is reported by the state of the MRDY, MERR, and MRTY signals. These signals can be asserted during any data phase. All MBus transactions can be suspended immediately by a *Retry* or by an *R&R*, also signaled by the MRDY, MERR, and MRTY signals. If *Retry* is signaled by the bus slave, the suspended transaction then restarts from the beginning with a new address phase. If *R&R* is signaled by the bus slave, the bus master must deassert MBB and re-arbitrate for MBus ownership.

A special case occurs for the CY7C604/605 and RT625 if an *R&R* is returned for an atomic Load/Store transaction. If the *R&R* occurs for the read section of the Load/Store transaction, the transaction is halted and MBB is deasserted. The entire transaction is repeated after re-arbitration (the normal case). If the read section has completed and the write section encounters an *R&R*, the transaction is halted and MBB is deasserted. However, in this case the transaction will retry with the write section and will not repeat the read section of the load/store transaction.

Table 11–6. Bus Status Encoding

MERR	MRDY	MRTY	Action
H	H	H	Nothing
H	H	L	*Relinquish and Retry*
H	L	H	Data Strobe
H	L	L	Reserved
L	H	H	Bus Error
L	H	L	Time Out
L	L	H	Uncorrectable Error
L	L	L	*Retry*

The data transfer rate on the MBus is controlled by the MBus slave. All MBus masters must be capable of accepting a burst transfer of the requested size at the maximum transfer rate supported by the bus. Bus slaves that cannot support the maximum transfer rate of the MBus must insert wait states by delaying the MRDY, MERR, and MRTY signals until the data cycle is completed. After the MBus transaction has finished, the bus master terminates the bus cycle by deasserting MBB.

Two transactions are defined for Level 1 MBus: read and write. Level 2 defines four additional transactions: Coherent Read, Coherent Invalidate, Coherent Read and Invalidate, and Coherent Write and Invalidate. The following section describes these transaction types.

11.1.7.1 Level 1 Transaction Types

11.1.7.1.1 Read (CY7C604/605 and RT625)

A read operation can be performed on any size of data transfer which is specified by the SIZE bits in the address cycle. Read transactions involving less than eight bytes will have undefined data on the unused bytes. The minimum MBus read transaction takes two cycles. The minimum time is for the cases when no data is returned on MAD, such as during *R&R* or *Error* acknowledgements. If data is being returned, an extra cycle is required to avoid bus contention. The arbitration protocol creates a dead cycle between trans-

actions which ensures there will be no bus contention between back-to-back reads from different masters. If a module locks the bus and performs back-to-back reads, it is its responsibility to ensure a dead cycle to avoid contention Note that the protocol means that a master *must* be able to receive data at the maximum rate of the MBus for the duration of the transaction, i.e., eight bytes on every consecutive clock. *Figure 11–6* illustrates a read transaction on MBus.

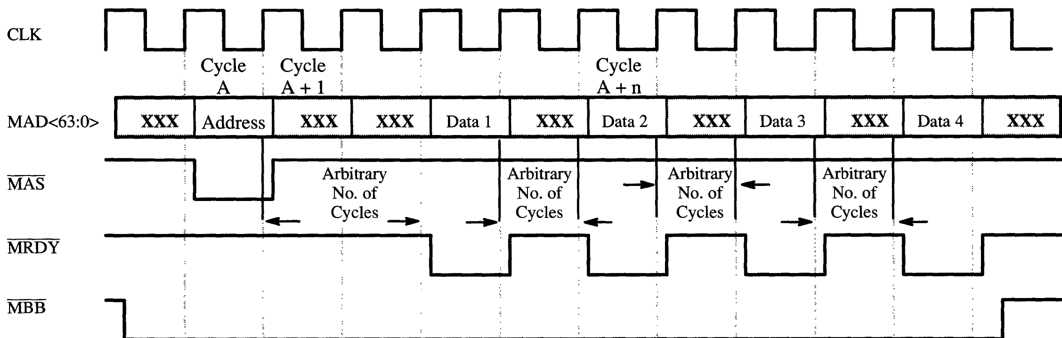


Figure 11–6. MBus Read Transaction

11.1.7.1.2 Write (CY7C604/605 and RT625)

A write operation can be performed on any size of data transfer specified by the SIZE bits in the MBus address cycle. Write transactions involving less than eight bytes will have undefined data on the unused bytes. The bus master performing the write immediately drives the data in the period after the address phase of the transaction, and immediately after receipt of each $\overline{\text{MRDY}}$ in transactions with SIZE greater than 8 bytes. Note that the protocol means that a master must be able to supply data at the maximum rate of the MBus for the entire transaction (i.e., 8 bytes on every consecutive transaction). The minimum MBus write operation takes two cycles (the minimum is three cycles if different masters are performing back-to-back writes). *Figure 11–7* shows an MBus write transaction.

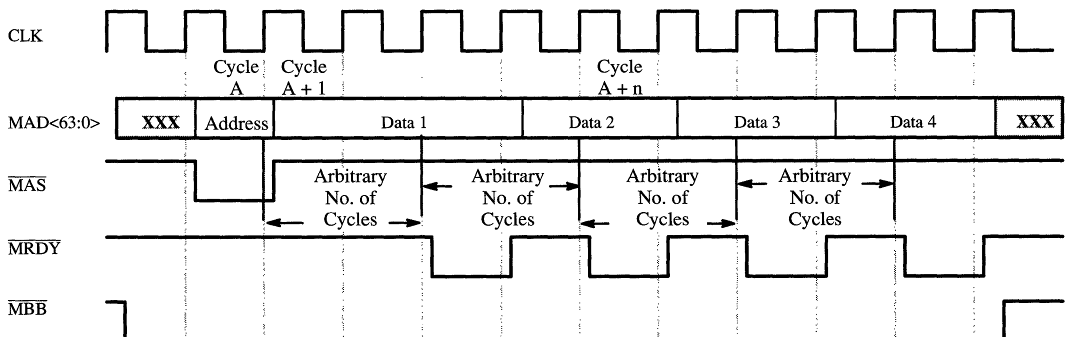


Figure 11–7. MBus Write Transaction

Due to the nature of the cache-consistency protocol, the write transaction works equally well in Level 1 and Level 2 MBus implementations. Writes can be used for non-cacheable accesses as well as for write-backs of modified cache lines. Write transactions do not need to be snooped and the $\overline{\text{MIH}}$ and $\overline{\text{MSH}}$ signals must not be asserted during the operation.

R&R acknowledgements issued to Block write transactions to cacheable locations introduce a detailed design problem, in that the write back buffer in this case may be the only source of the most up to date data. This introduces the prospect of having to snoop the write back buffer. To simplify the design of processor modules, the MBus specification eliminates the need for processor modules to snoop write back buffers and places the burden of handling this case of *R&R* acknowledgement on the module that issues the *R&R*. Modules that issue *R&R* acknowledgements to cacheable block write transactions must capture the address(es) of the cache line(s) until they complete the transaction to which they issued *R&R*. Should other modules attempt to read the line(s) during this interval the *R&R* issuing module must detect this and issue *R&R* to the intervening Coherent Read (CR) or Coherent Read and Invalidate (CRI) transaction(s). In general it should be possible for most modules to avoid the need to issue *R&R* to cacheable block write operations and hence avoid this complexity (the only likely exception is a coherent bus adaptor).

11.1.7.2 Additional Transaction Types for Level 2

Level 2 requires two additional signals over Level 1 in order to support cache coherency operations. $\overline{MSH}$ (Memory Shared) and $\overline{MIH}$ (Memory Inhibit) are asserted during MBus coherent transactions to describe the shared and ownership status of a cache line whose address has been asserted on the MBus. $\overline{MSH}$ is asserted by a CY7C605 and RT625 in response to a bus snooping operation for which it discovers it has a copy of the cache line involved in the current coherent MBus transaction. $\overline{MIH}$ is asserted by a CY7C605 and RT625 in response to a coherent transaction on a cache line which the CY7C605 and RT625 own (i.e., have the most up-to-date copy). The $\overline{MIH}$ signal is used to inhibit the output of the memory system, and is asserted to indicate that the CY7C605 and RT625 will respond to the memory request by supplying the data directly to the requesting cache.

11.1.7.2.1 Coherent Read (CY7C605 and RT625 only)

A Coherent Read operation is a block read transaction that maintains cache consistency. The participants in the transaction are the requesting cache, the other caches performing bus snooping, and memory (or a second-level cache). There are three possible read scenarios for a multiprocessing system with snooping caches:

1. For a snooping cache that does not have a copy of the requested block, the cache simply ignores this transaction.
2. For a snooping cache that has a copy of the requested block but does not own it, the cache must assert $\overline{MSH}$ for one cycle during the cycle A+2 or A+3†. It will mark its copy as shared (if not already marked as such).
3. For a snooping cache which owns the requested block, the cache must assert both $\overline{MSH}$ and $\overline{MIH}$ signals for one cycle during the A+2 or A+3 cycle. The cache supplies the requested data no sooner than cycle A+6 (four cycles after it issued $\overline{MIH}$). If the cache's own copy of the block was labeled exclusive, it will be changed to shared. Otherwise, no status change will take place for the cache's own copy.

Upon receiving the data block, the requesting master shall label the block exclusive if no one asserts $\overline{MSH}$ during the A+2 or A+3 cycle. The requesting master shall label the block as shared if the $\overline{MSH}$ signal is asserted during the A+2 or A+3 cycle.

Case (3) is the only case where $\overline{MIH}$ is asserted. This signal affects three parties. It is sourced by the snooping (intervening) cache and observed by both memory (or a second level cache) and the requesting cache.

† See SPARC International *SPARC MBus Specification* for notes to designers who wish to avoid the A+2 requirement.
The CY7C605 asserts $\overline{MSH}$ and $\overline{MIH}$ during cycle A+2.
The RT625 asserts $\overline{MSH}$ and $\overline{MIH}$ during cycle A+3.

It tells the requesting cache that it may have received stale data from memory, and to ignore that data and data it may receive on the next clock and wait until the fourth or later clock for the correct data. It tells memory to stop sending data *immediately*, which means memory may send one more MRDY before it can stop. The delay of 4 clocks at the requesting cache and the snooping (intervening) cache serve two related purposes. The first is to allow time for MRDY and MAD from the memory to be turned off before the snooping cache asserts MRDY and MAD, and so avoid bus contention. The second is to allow for implementations that buffer the MBus.

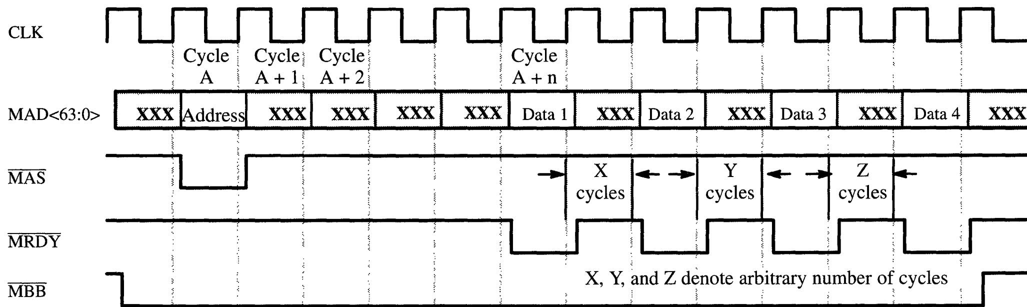


Figure 11–8. MBus Coherent Read Transaction - MIH not asserted

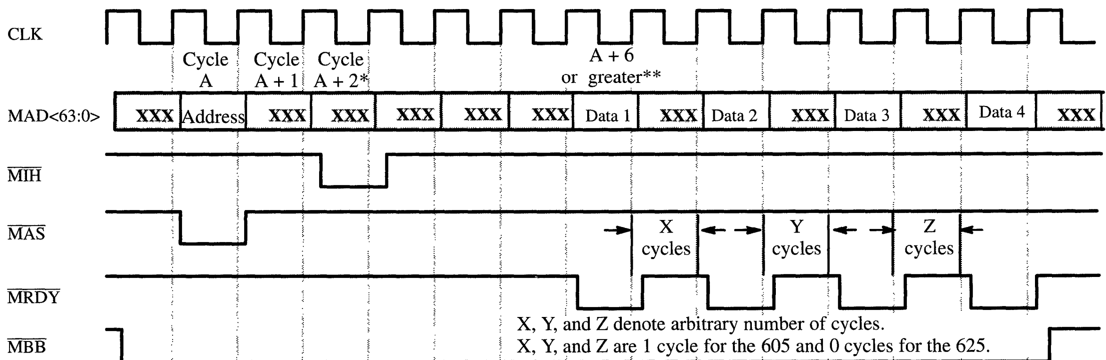


Figure 11–9. MBus Coherent Read Transaction - MIH asserted

11.1.7.2.2 Coherent Invalidate (CY7C605 and RT625)

An invalidate operation can only be performed on a cache-line basis. All invalidate operations are snooped. In an invalidate operation that hits in a cache, the cache line copy is invalidated immediately regardless of its state. One MBus module (normally a memory or second-level cache controller) is responsible for the acknowledgment of a Coherent Invalidate transaction on the A+2† cycle or later. All acknowledgment types are possible. Memory will only issue normal acknowledgments (i.e., MRDY) to Coherent Invalidate transactions, but other bus adaptors, such as a second-level cache, may issue the full range of acknowledgments, especially *R&R*. It should be noted that a Coherent Invalidate transaction has SIZE = 32 bytes during

† See SPARC International *SPARC MBus Specification* for notes to designers who wish to avoid the A+2 requirement. For the RT625 the Coherent Invalidate must be acknowledged during cycle A+3 or later.

the address phase, but will only be expecting one MRDY as the acknowledgment. Also, the address may not be 32-byte aligned. For a cache system that cannot guarantee to complete the invalidation before the A+2 cycle, the memory controller for that system should delay the acknowledgment as required. (This implies that memory controllers should have a feature that allows the time to acknowledge invalidates to be varied to some extent, either hard wired or through a programmable register. A recommended range for the programmable delay is A+2 to A+10. This programmable delay is the MBus flow control technique to guarantee that invalidates can be completed at any rate they are issued.)

The Coherent Invalidate transaction is issued when a write is being performed on a shared cache line. Before the write can be performed, all other caches in the system must invalidate their copies (write-invalidate cache consistency protocol). Snooping caches need not assert MSH during the A+2 cycle†. The MAD<63:0> bus is undefined during the data cycles. If a Coherent Invalidate transaction should receive an *R&R* acknowledgement, there is a possibility that the line which is about to be written becomes invalidated by an intervening invalidation transaction on the bus. This means that when the cache regains the bus it should issue a Coherent Read and Invalidate transaction, not a Coherent Invalidate transaction, to once again allocate the cache line. *Figure 11–10* shows the basic Coherent Invalidate operation.

For any particular system, selecting which module will be responsible for acknowledging Coherent Invalidates introduces some issues for memory controller designers. In most systems a single memory controller will be responsible. In systems with a coherent bus adaptor, the adaptor will be responsible. If it is desired to use a memory controller in a system that also has a coherent bus adaptor, it is then required to be able to tell the memory controller not to respond to invalidates. This should be accomplished during system initialization prior to enabling any caches, preferably by writing a bit in a register in a memory controller.

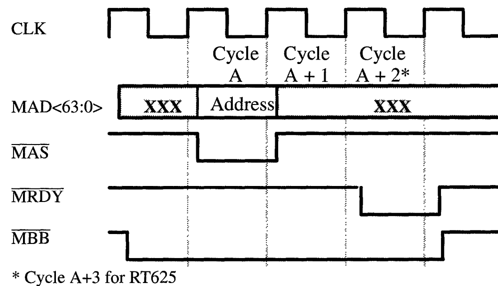
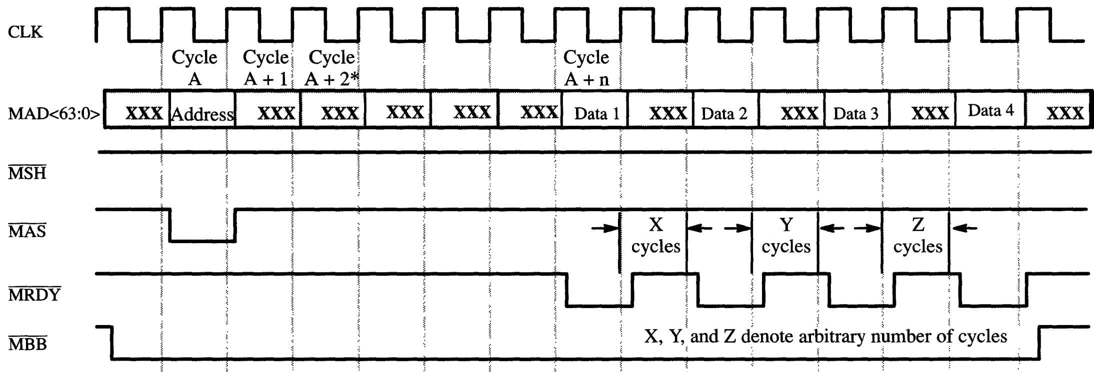


Figure 11–10. MBus Coherent Invalidate Transaction

11.1.7.2.3 Coherent Read and Invalidate (CRI) (CY7C605 and RT625)

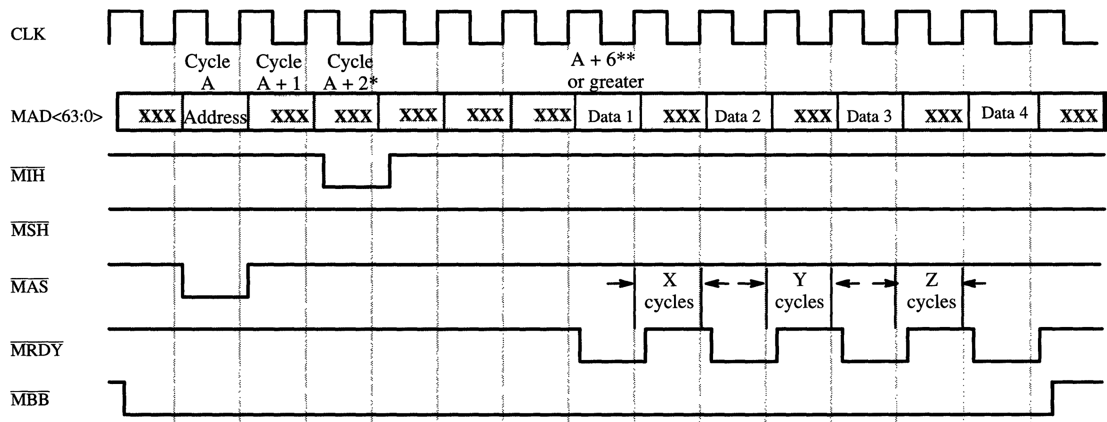
The Coherent Read and Invalidate transaction combines a Coherent Read transaction with a Coherent Invalidate transaction. This transaction is included to reduce the number of MBus Coherent Invalidate transactions. Caches performing Coherent Reads that intend to immediately modify the data can issue this transaction.

Each CRI transaction is snooped by all system caches. If the address hits in a cache but the cache does not own the block, then the cache invalidates its copy of this block regardless of the state of the data. If the address hits in a cache and the cache owns the block, then it asserts MIH and supplies the data. When the data has been successfully supplied, the cache then invalidates its copy of the block. *Figure 11–11* and *Figure 11–12* show the CRI operation. Note that it is identical to the Coherent Read operation, except that the snooping caches invalidate their copy of the cache line upon a cache hit. All of the comments concerning MIH for the Coherent Read transaction apply to the CRI transaction. MSH is not driven during the CRI transaction.



* Cycle A+3 for the RT625

Figure 11–11. MBus Coherent Read and Invalidate Transaction - MIH not asserted



* A+3 for the RT625
** A+7 for the RT625

X, Y, and Z denote arbitrary number of cycles.
X, Y, and Z are 1 cycle for the 605 and 0 cycles for the 625.

Figure 11–12. MBus Coherent Read and Invalidate Transaction - MIH asserted

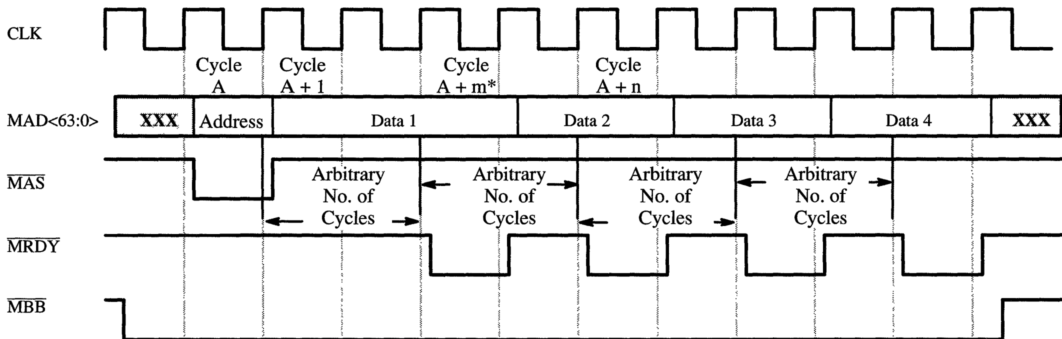
11.1.7.2.4 Coherent Write and Invalidate (CWI) (CY7C605 and RT625)

The Coherent Write and Invalidate transaction combines a write transaction with a Coherent Invalidate transaction. This transaction is included to reduce the number of MBus Coherent Invalidate transactions. This transaction can be used by modules that can operate with a write-through cache, useful for second-level caches that support inclusion.

Each CRI transaction is snooped by all system caches. If the address hits in a cache, then that cache invalidates its copy of the cache line regardless of the state of the data. *Figure 11–13* illustrates the basic CWI operation, which is a block transaction in copy–back mode. Note that this transaction is identical to the write operation, except that the snooping caches invalidate their block upon a cache hit. The SIZE for this transac-

tion can be any valid MBus transaction size (though the CY7C605 and RT625 only support up to 32 bytes), and a single, 32-byte cache line is invalidated regardless of the value of SIZE. Due to the nature of the cache coherency protocol, neither MIH nor MSH is asserted.

Figure 11–14 shows a typical CRI transaction in write-through mode (in which non–block transfers can occur) in which different bus masters are asserting MAS. This diagram represents the timing for the occurrence of MRDY in A+3 or later for the RT625 (or in A+2 or later for the CY7C605). MRDY may be asserted in A+2 (this is the earliest cycle allowed by MBus), but only if MBG is deasserted during MRDY (see Figure 11–15). This applies to byte, halfword, word, or doubleword writes in write-through mode. For Block CWI transactions in write-through mode (generated by the RT625 only for Block Copy and Block Fill operations). MRDY may be asserted earlier than A+3, but bus acknowledgements (*Error*, *Retry*, or *R&R*) must not be asserted before A+3.



* *m* must be 2 or greater for the CY7C605 and 3 or greater for the RT625.

Figure 11–13. MBus Block Coherent Write and Invalidate Transaction

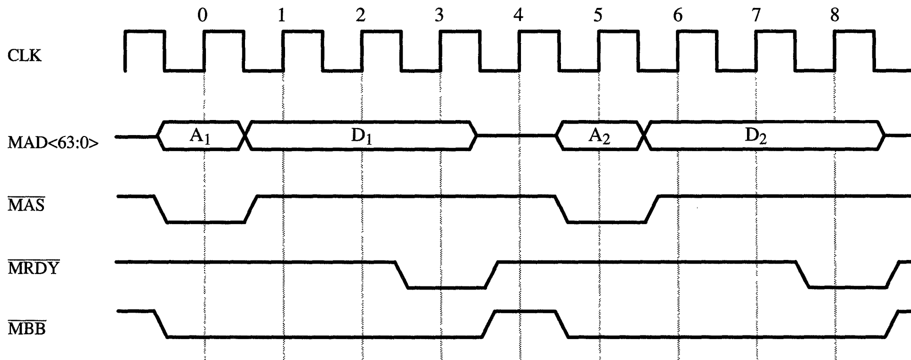


Figure 11–14. MBus Write-through Coherent Write and Invalidate Transaction

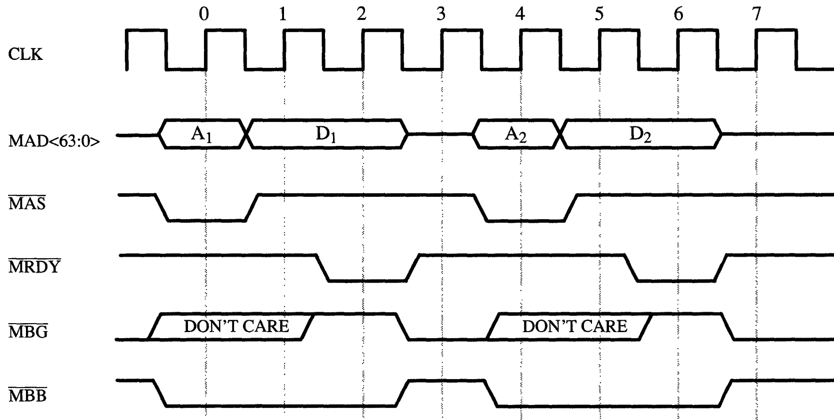


Figure 11–15. MBus Write-through Coherent Write and Invalidate Transaction ($\overline{\text{MRDY}}$ in A+2)

11.1.8 MBus Acknowledgement Cycles

It is a requirement that any transaction once issued *must* correctly accept *any* acknowledgment type. This applies to all Level 1 and Level 2 transactions. The earliest that an acknowledgment can be issued is A+1 for read and write and A+2 (CY7C605) or A+3 (RT625) for all coherent transactions. Processor caches that are supplying data as part of a Coherent Read transaction may only issue either normal or *Erro*r acknowledgements. They may not issue *R&R* or *Retry* acknowledgements. The CY7C605 and RT625 only issue normal acknowledgements while supplying data for a CR or CRI.

11.1.8.1 Idle Cycles

When there is no bus activity or when it is necessary to insert wait states in between the address cycle and the data cycle or between consecutive data cycles, an addressed slave can simply refrain from asserting any transaction status bits ($\overline{\text{MERR}}$, $\overline{\text{MRDY}}$, and $\overline{\text{MRTY}}$). The number of wait cycles which can be inserted is arbitrary, as long as it does not exceed the system timeout interval (see *Section 11.1.8.5* for timeout details).

11.1.8.2 Relinquish and Retry (R&R)

When a slave device cannot accept or supply data immediately, it can perform an *R&R* acknowledgment cycle by asserting $\overline{\text{MRTY}}$ for only one bus cycle. This will indicate to the requesting master that it should release the bus immediately so that the bus can be rearbitrated and possibly used by another MBus master. This involves at least one dead cycle until the suspended transaction can be performed in the case when the bus is still granted to the retrying master. When the bus is no longer granted to the master in question, then the suspended transaction must wait until bus ownership is once again attained. When a transaction that receives an *R&R* acknowledgment regains bus mastership it must issue the same transaction over from the beginning. An exception to this is when a Coherent Invalidate is a Coherent Read and Invalidate. For Level 1 modules, for all transactions with SIZE greater than 8 bytes, an *R&R* acknowledgment can be asserted on any data transfer. For Level 2 modules, for all transactions with SIZE greater than 8 bytes, (including the Level 1 READ and WRITE transactions) *R&R* can only be issued on the first acknowledgement. It is the responsibility of the slave port to time the duration of the transaction that is causing it to issue *R&R*, and return an ERROR2 acknowledgment to the correct master when its device specific timeout interval has passed and the master has reconnected to the slave.

There are two different cases that cause slaves to issue *R&R* acknowledgments. The first is slow devices. If a device is slow to respond, the slave interface should wait a short interval (around one microsecond is recommended), and then issue an *R&R* acknowledgment. It should also capture the ID of the master from the MAD lines during the address phase (MID[3:0] field) and enter a "port busy" state while waiting for the device attached to the slave to respond. The master will eventually reconnect and the *R&R* process will be repeated until either the device responds or the slave timeout interval is exceeded. The slave will then issue the normal or error acknowledgment respectively and exit the "port busy" state.

In systems with multiple masters, the slave that issues *R&R* must capture the ID of the master whose transaction is being postponed in order to know which master should receive the normal or error acknowledgment when the slave can complete the transaction. If a master with an ID other than that captured by the slave port should access the slave port while it is in the "port busy" state, it should simply be given an *R&R* acknowledgment.

The second cause of *R&R* acknowledgments is the resolution of deadlock situations where there is a master and a slave port sharing an MBus interface and simultaneous transactions on both ports requires one transaction to back off. *R&R* requires the current owner of MBus to relinquish ownership in order to resolve the deadlock. *R&R*'s used to resolve deadlocks are inherently stateless and do not require a "port busy" state.

A detail of significance is that *R&R* can be issued to a transaction that is part of a locked sequence of transactions. By definition, all transactions in a locked sequence are addressed to the same device, e.g. main memory (or second level cache) or an I/O adapter. There is only one "port busy" state per device, so there is only one source of *R&R* for a locked sequence.

Normally, main memory will not issue *R&R*. Multiple *R&R* sources from main memory would restrict locked sequences to addresses within one memory bank, Also, some aspects of coherent cache design are simplified by locking some MBus sequences such as fills and their associated write-back (if any). These sequences rely on either a memory system that does not issue *R&R* or an appropriately designed second level cache or coherent bus adaptor that does.

It should be noted that processor caches which assert $\overline{\text{MIH}}$ and then supply data cannot issue *R&R* acknowledgements.

11.1.8.3 Valid Data Transfer

A valid or ready data transfer is indicated by a responding slave with the assertion of the $\overline{\text{MRDY}}$ transaction status bit for only one cycle. This signal needs to be asserted on reads to indicate to the requesting master that valid data has just arrived. On writes, $\overline{\text{MRDY}}$ indicates to the writing master that the data has been accepted and that the writing master shall stop driving the accepted data. The next doubleword, if a write burst were being performed, will be driven onto the bus in the cycle immediately following the assertion of $\overline{\text{MRDY}}$.

11.1.8.4 ERROR1 => Bus Error

When the responding device asserts only the $\overline{\text{MERR}}$ transaction status bit, the requesting master will interpret this as an external bus error just having taken place. The meaning of "Bus Error" is implementation dependant.

11.1.8.5 ERROR2 => Timeout

This acknowledgment is expected to be generated by some sort of watchdog timer logic in the system that primarily detects transactions that are not acknowledged. This is accomplished by timing the shorter of, either the assertion of $\overline{\text{MBB}}$, or the time since the last $\overline{\text{MAS}}$ assertion, as follows.

A timeout counter should start on the assertion of $\overline{MBB}$ and count until the deassertion of $\overline{MBB}$, or until the timer has counted the timeout interval. If the counter counts to the timeout limit, a timeout error acknowledgement should be generated by the timeout monitor circuitry. When counting ceases, the counter should be reinitialized to its initial condition. If $\overline{MAS}$ is asserted during the time that counting is enabled ($\overline{MBB}$ assertion) the counter should be reinitialized, but continue counting. The number of cycles for the timeout interval is system implementation dependent. This error code can also be used to indicate a system implementation dependent error. Timeout is the suggested interpretation of an ERROR2 acknowledgment.

11.1.8.6 ERROR3 => Uncorrectable

This acknowledgment is mainly used by the addressed memory controller to inform the requester that in the process of accessing the data some sort of uncorrectable error has been encountered (like parity, uncorrectable ECC, etc). This error code can also be used to indicate a system implementation dependent error. Uncorrectable error is the suggested interpretation of an ERROR3 acknowledgment.

11.1.8.7 Retry

This acknowledgment differs from the *R&R* acknowledgment in that the master will not, in this case, release bus ownership if it is no longer granted the bus, but rather the transaction will immediately begin again with an address phase ($\overline{MAS}$, etc.) as soon as the retried master is ready to do so. *Retry* errors can occur on any acknowledgment of a transaction. This type of acknowledgment can be useful when a correctable ECC error has occurred in the main memory subsystem.

Should a *Retry* acknowledgement occur on other than the first acknowledgement cycle, the issue of "Data Correctness" arises. Modules that use delivered data prior to completion of the transaction may not be able to tolerate delivery of bad data. They may choose to treat *Retry* acknowledgements as equivalent to ERROR3 acknowledgements. This assumes that the *Retry* is "stateless" and the slave device issuing it will not hang or otherwise malfunction if the transaction is not retried. This is a detail at the discretion of system implementors. In the CY7C605 *Retry* can be issued on any acknowledgement cycle. In the RT625 a *Retry* after the first acknowledgement will be treated as an error. In this case the data strobed prior to the *Retry* must be valid.

11.1.8.8 Reserved

This acknowledgement is reserved for future use. Should a master receive a reserved acknowledgement its behavior is undefined.

11.1.9 MBus Arbitration

11.1.9.1 Arbitration Principles

- The Arbiter is a separate unit from both the slave(s) and master(s).
- Arbitration is overlapped with current bus cycle.
- Back-to-back transactions by different masters are not allowed. There must be at least one dead cycle in between each transaction during which $\overline{MBB}$ is deasserted.
- Arbitration algorithm is implementation dependent (fair bandwidth allocation should be maintained).
- Bus parking is employed (i.e., current master keeps the bus until it is taken away by another request).
- Locked cycles are accommodated to handle indivisible operations.

11.1.9.2 Arbitration Protocol

The MBus arbitration scheme assumes a central arbiter. The exact algorithm used by the arbiter (e.g., round robin, etc) is implementation dependent. The arbiter uses only the MBRn and MBGn signals from each master and the common bussed MBB signal. The arbiter receives the requests (MBRn) and resolves which grant (MBGn) to assert.

A bus master requests bus ownership by asserting its dedicated MBR signal. The arbiter grants bus ownership by asserting the dedicated MBG signal for that bus master. If the MBB (MBus bus busy) signal is not asserted, the bus master asserts MBB and starts the bus transaction. If the MBB signal is asserted, the bus master must wait until is has been released. The bus master does not own the bus until it has asserted MBB, and MBB cannot be asserted until it has been released by the previous bus master. This protocol allows the MBus to support overlapped bus arbitration. Note that MBG should stay asserted until MBB has been released by the current bus master.

Upon receiving its dedicated MBG signal, the requesting master can start using the bus by asserting MAS and MBB as soon as MBB is released (deasserted) by the previous master, and should remove its dedicated request (MBR) on the next clock edge. (It is allowed to assert MBR in anticipation of needing the bus, and then deassert it prior to receiving MBG. However, this may waste bus cycles and should be avoided.) It is not necessary for the requesting master to assert MAS immediately, but it is necessary to assert MBB to acquire and hold the bus. A requesting master is not guaranteed to gain bus ownership if it does not immediately assert MBB upon detecting the condition of its MBG asserted and MBB deasserted.

After MBB has been released by the current bus master, MBG may be deasserted at any time in response to other bus requests. If no further requests are made, the MBG should stay asserted. This is referred to as "bus parking," and it allows subsequent requests from the same bus master to be serviced without the delay of arbitrating the MBus. If MBG for a particular bus master has already been asserted (i.e., the bus has been parked on that bus master), the bus master may assert MBB and claim the MBus without first asserting MBR. Only one grant (MBG) is asserted at any time. A dead cycle between successive transactions of different masters will always occur with the MBus arbitration scheme.

Timing diagrams for example transactions are shown in *Section 11.1.11.*

A grant remains asserted until at least one cycle after the current master has deasserted MBB when it becomes "parked," and may be removed at any time after this in response to assertion of further requests.

11.1.10 MBus Configuration Address Map

A small portion of the MBus memory space has been preallocated to each potential MBus module, to allow for a uniform method of system configuration. There is an individual space per MBus ID, 16 spaces in total. The ID of a module is determined by the value on the ID[3:0] pins. Level 1 processor modules do not have slave interfaces and so will not respond at the configuration address map locations. The CY7C605 and RT625, though Level 2 devices, do not implement the configuration address map. *Figure 11–16* shows the configuration address spaces of MBus.

Configuration Spaces	Mbus Identifier
0xFF0000000 to 0xFF0FFFFFF	Range for ID=0x0*
0xFF1000000 to 0xFF1FFFFFF	Range for ID=0x1
0xFF2000000 to 0xFF2FFFFFF	Range for ID=0x2
. .	.
. .	.
. .	.
. .	.
0xFFF000000 to 0xFFFFFFFFF	Range for ID=0xF

*reserved for “boot PROM”

Figure 11–16. MBus Configuration Address Map

One 32-bit location in each space (0xFFnFFFFFC where n=ID) is fixed and should contain the Implementation Number and Version Number of the MBus module in an **MBus Port Register (MPR)** which is shown in *Figure 11–17.* A processor accessing the 16 possible MPRs can determine what ID slots are present(through timeout) and what devices they contain from the contents of the MPR. Other than this one address, the use of the address range is implementation specific, and specified by module vendors. Examples of items located in these configuration address ranges are registers that determine the address ranges which memory and I/O modules respond to. Similarly, implementation specific registers and memories necessary to configure and test modules would reside at locations within the device specific configuration address space.

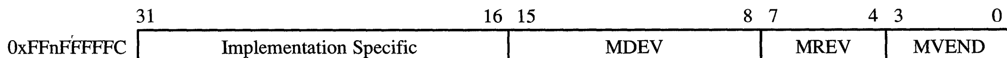


Figure 11–17. MBus Port Register Format

MDEV MBus device number. This field contains a unique number which indicates the vendor specific MBus device that is present at the referenced MBus port. Refer to Vendors for their MDEV assignments.

MREV Device revision number. This field contains a number that can be interpreted as a revision number or some other variable of a device. Refer to Vendors for their MREV assignments, if any.

MVEND MBus vendor number. This field contains a unique number which indicates the vendor of the device present at the referenced MBus port. Refer to Appendix A for current MVEND assignments.

On coming out of reset, processor modules will fetch instructions from MBus address 0xFF000000 and subsequent memory locations. This means that the configuration address space for ID=0x0 is special and always needs to be present. MBus ID=0x0, then, can be considered as the “boot PROM” MBus module.

11.1.11 MBus Transaction Timing†

Page

Figure 11–18. Initial MBus Arbitration 11-25
Figure 11–19. MBus Mastership Transfer 11-25
Figure 11–20. MBus Arbitration with Multiple Requests 11-26
Figure 11–21. MBus Single-Cycle – Read Transaction 11-27
Figure 11–22. MBus Single-Cycle – Write Transaction 11-27
Figure 11–23. MBus Burst-Cycle – Read Transaction 11-28
Figure 11–24. MBus Burst-Cycle – Read Transaction (Slow memory) 11-28
Figure 11–25. MBus Burst-Cycle – Write Transaction 11-29
Figure 11–26. MBus Burst-Cycle – Write Transaction (Slow memory) 11-30
Figure 11–27. MBus Locked Transaction 11-31
Figure 11–28. MBus Relinquish and Retry 11-32
Figure 11–29. MBus Retry 11-32
Figure 11–30. MBus Error (Bus Error) 11-33
Figure 11–31. MBus Error (Timeout) 11-33
Figure 11–32. MBus Error (Uncorrectable) 11-34
Figure 11–33. MBus Coherent Read – Shared Data†† 11-35
Figure 11–34. MBus Coherent Read – Owned Data (CY7C605) (Slow Memory)†† 11-37
Figure 11–35. MBus Coherent Read – Owned Data (CY7C605) (Fast Memory)†† 11-39
Figure 11–36. MBus Coherent Read – Owned Data (RT625) (Slow Memory)†† 11-41
Figure 11–37. MBus Coherent Read – Owned Data (RT625) (Fast Memory)†† 11-43
Figure 11–38. MBus Coherent Write and Invalidate†† 11-45
Figure 11–39. MBus Coherent Write and Invalidate (RT625) (Block Copy/Fill)†† 11-47
Figure 11–40. MBus Coherent Invalidate†† 11-49
Figure 11–41. MBus Coherent Read and Invalidate – Shared Data†† 11-50
Figure 11–42. MBus Coherent Read and Invalidate – Owned Data (CY7C605) (Slow Memory)†† 11-52
Figure 11–43. MBus Coherent Read and Invalidate – Owned Data (CY7C605) (Fast Memory)†† . 11-54
Figure 11–44. MBus Coherent Read and Invalidate – Owned Data (RT625) (Slow Memory)†† . . . 11-56
Figure 11–45. MBus Coherent Read and Invalidate – Owned Data (RT625) (Fast Memory)†† 11-58

† All transactions apply to the CY7C604, CY7C605, and RT625 unless otherwise noted.
†† MBus Level 2 transaction only.

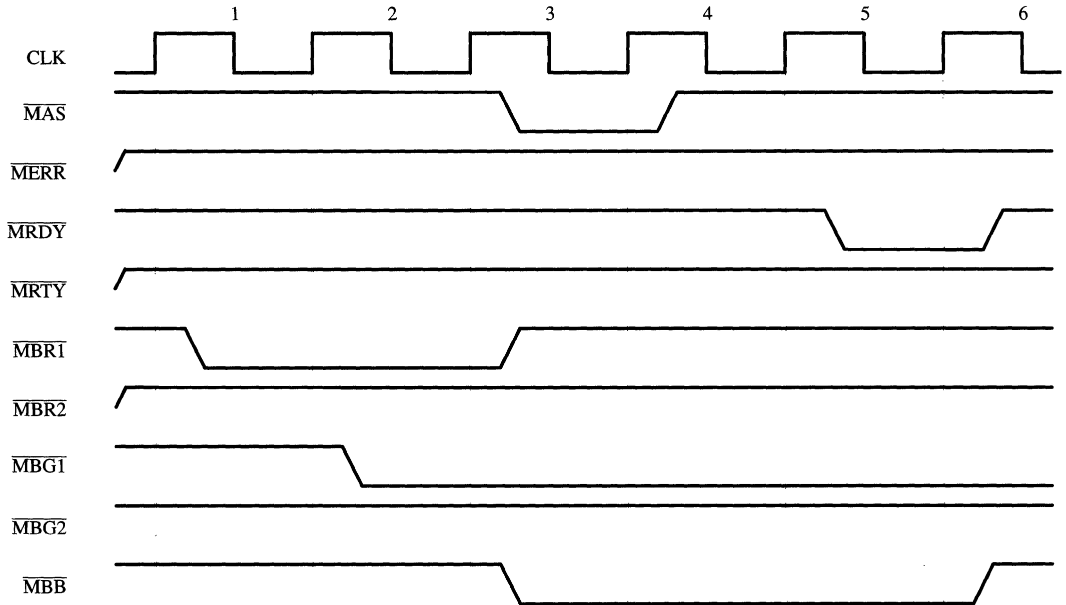


Figure 11–18. Initial MBus Arbitration

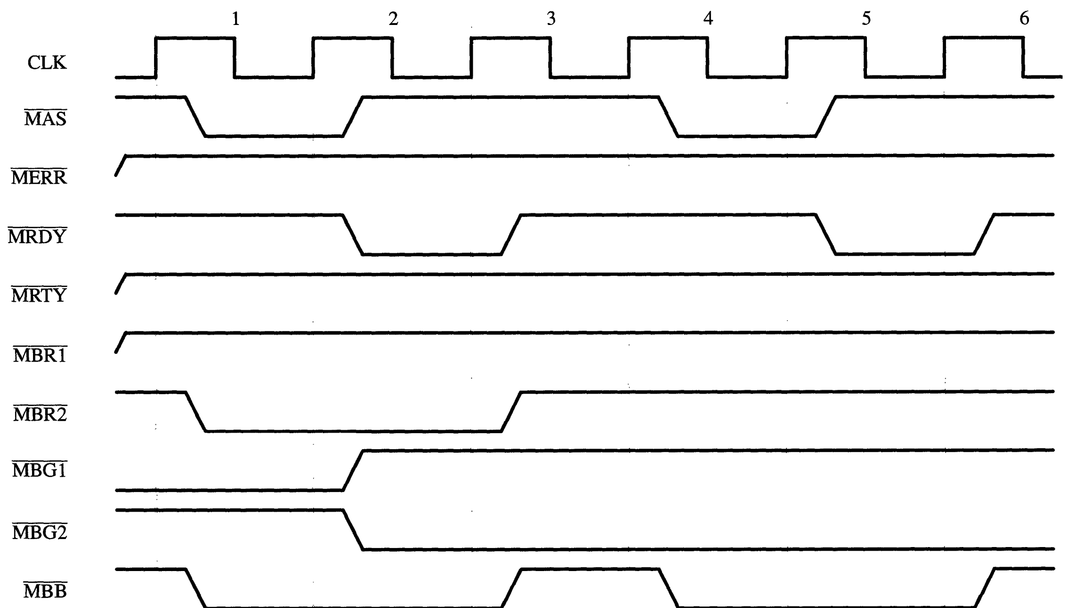


Figure 11–19. MBus Mastership Transfer

Note on arbitration: $\overline{MBR2}$ can appear at any time and does not have to be granted immediately as shown.

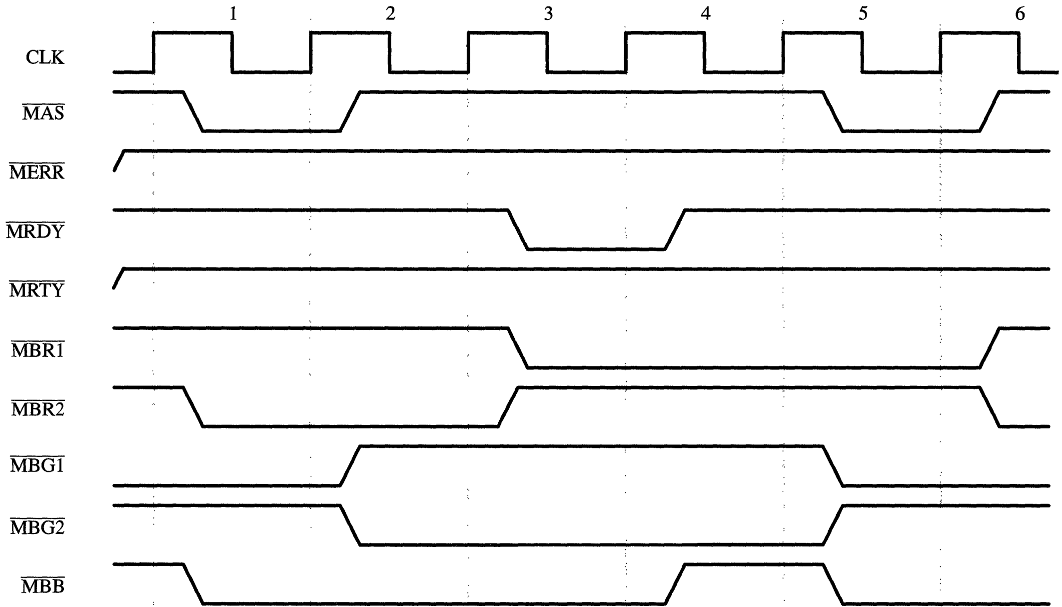


Figure 11–20. MBus Arbitration with Multiple Requests (part 1 of 2)

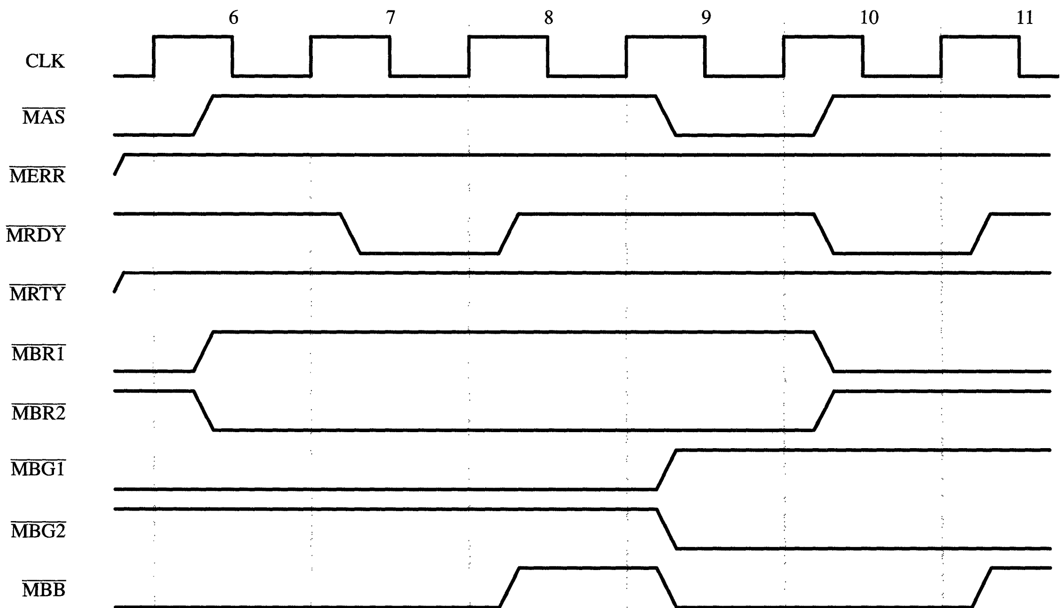


Figure 11–20. MBus Arbitration with Multiple Requests (part 2 of 2)

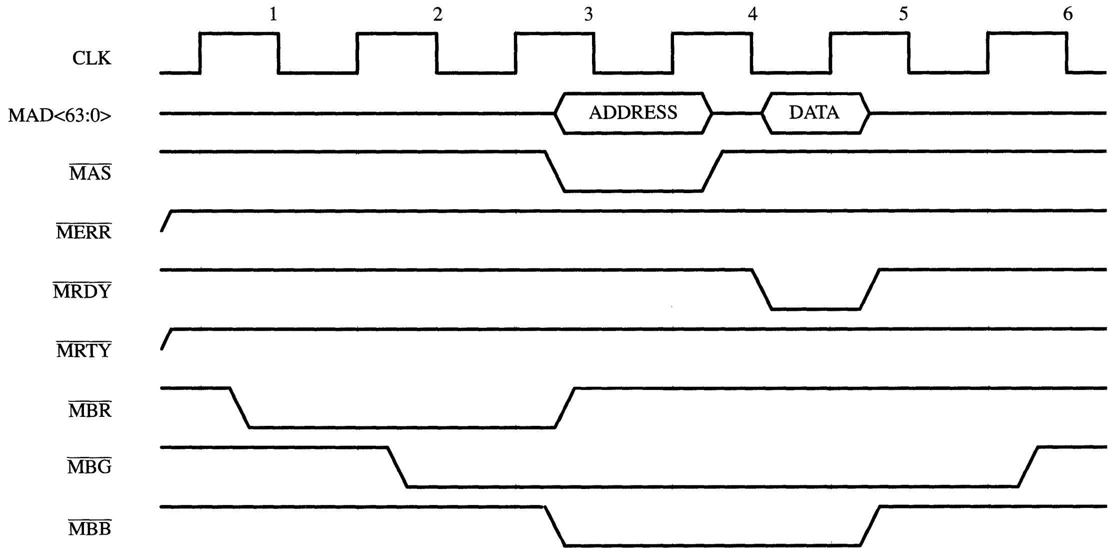


Figure 11–21. MBus Single-Cycle Read Transaction

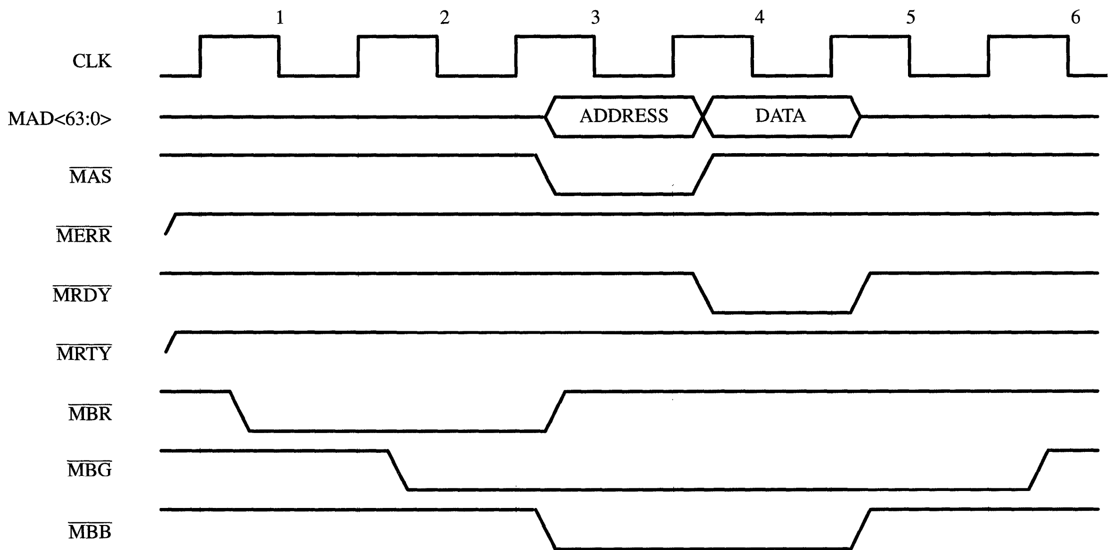


Figure 11–22. MBus Single-Cycle Write Transaction

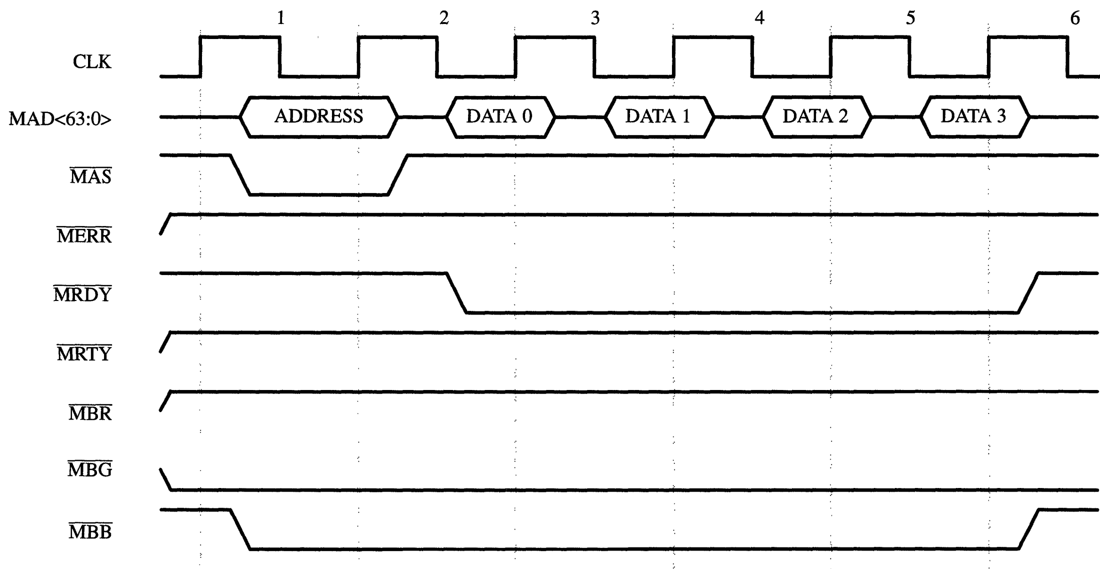


Figure 11–23. MBus Burst-Cycle Read Transaction*

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

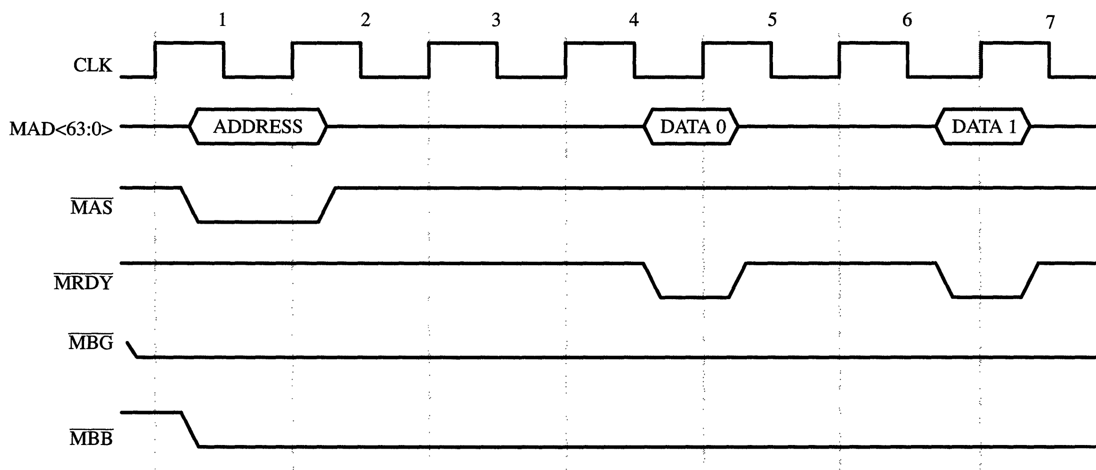


Figure 11–24. MBus Burst-Cycle Read Transaction (Slow memory)* (part 1 of 2)

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

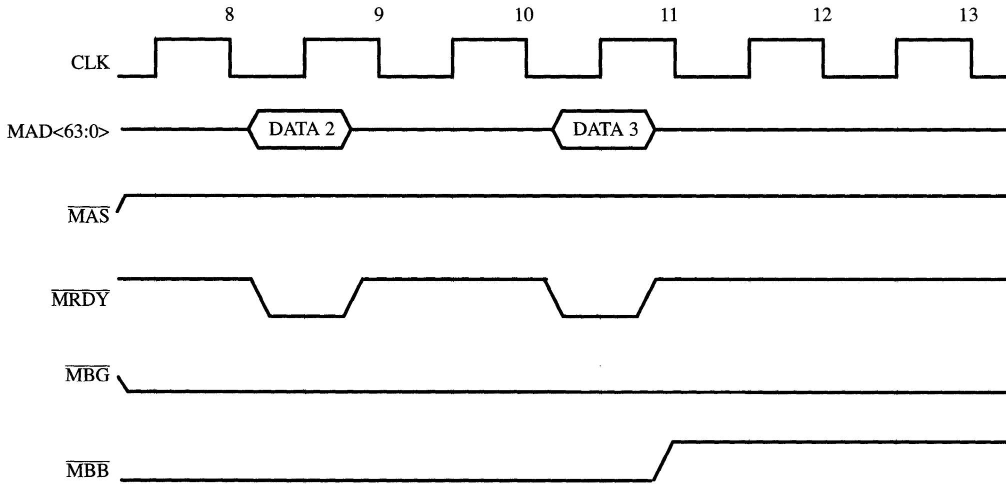


Figure 11–24. MBus Burst-Cycle Read Transaction (Slow memory)* (part 2 of 2)

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

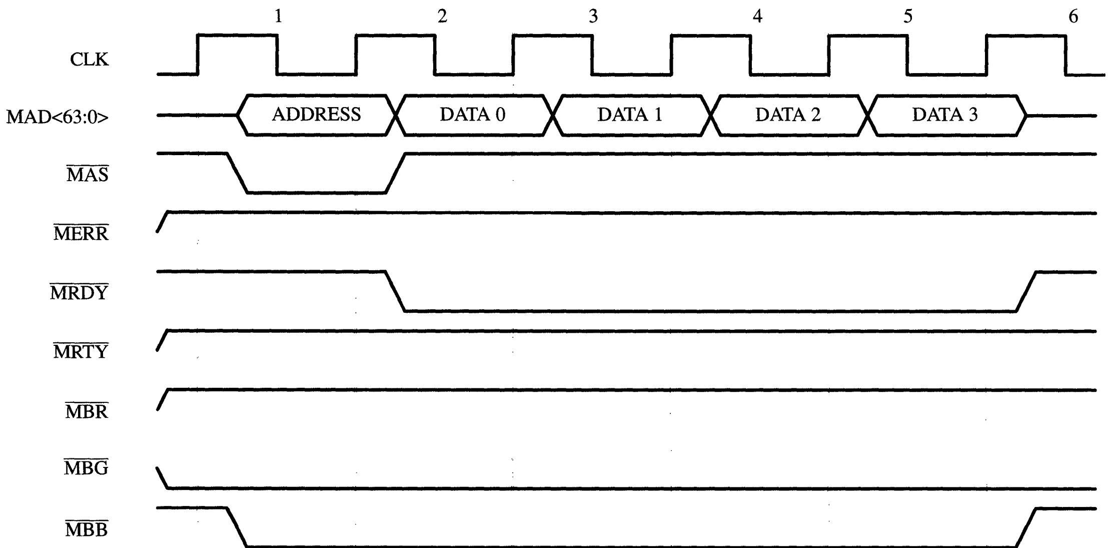


Figure 11–25. MBus Burst-Cycle Write Transaction*

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

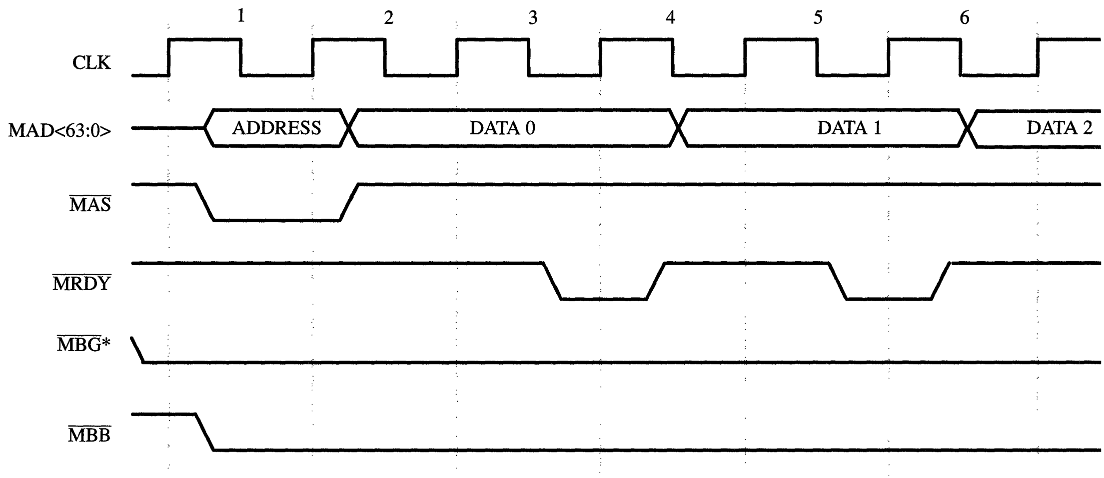


Figure 11–26. MBus Burst-Cycle Write Transaction (Slow memory)* (part 1 of 2)

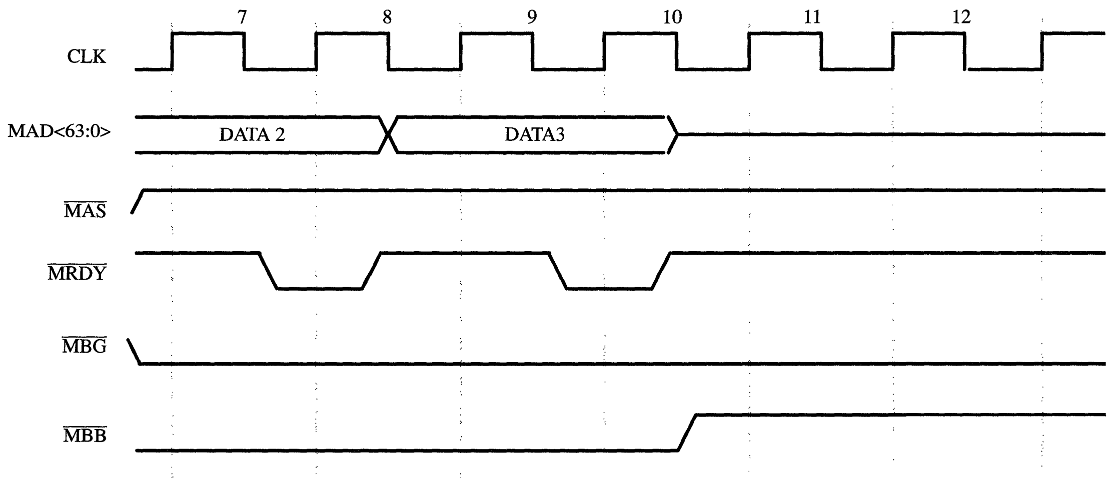


Figure 11–26. MBus Burst-Cycle Write Transaction (Slow memory)* (part 2 of 2)

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

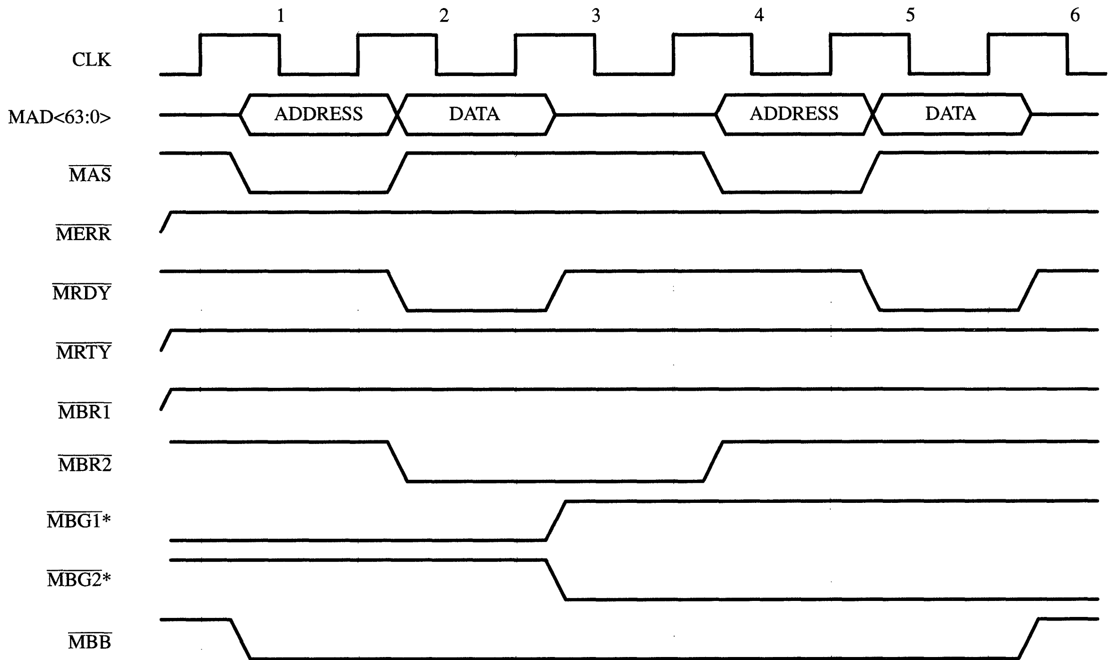


Figure 11–27. MBus Locked Transaction*

* Note that the bus has been granted to the the CY7C604/605 or RT625 prior to the beginning of the transaction.

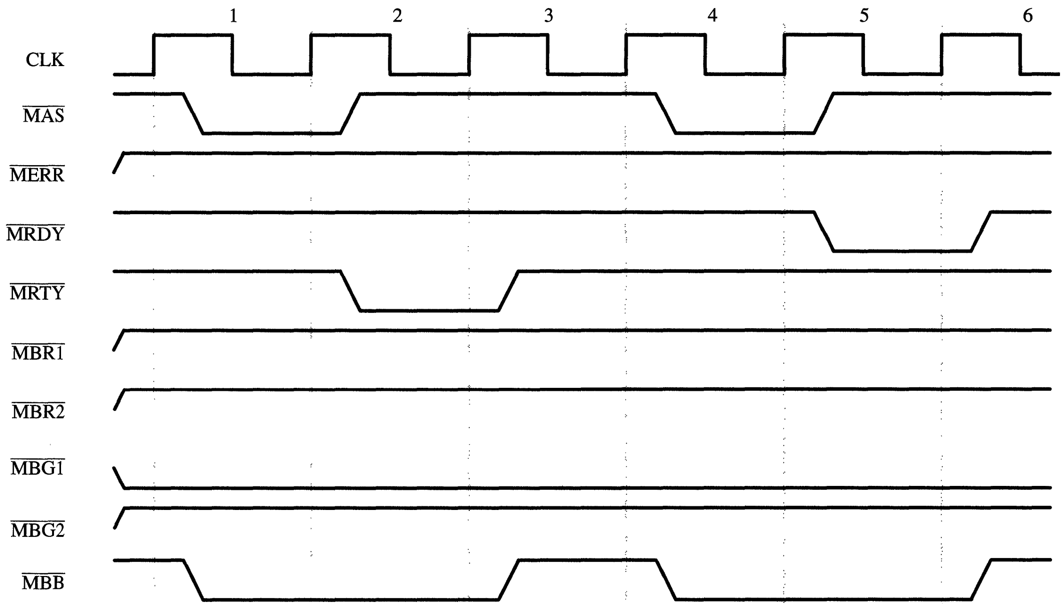


Figure 11–28. MBus Relinquish and Retry

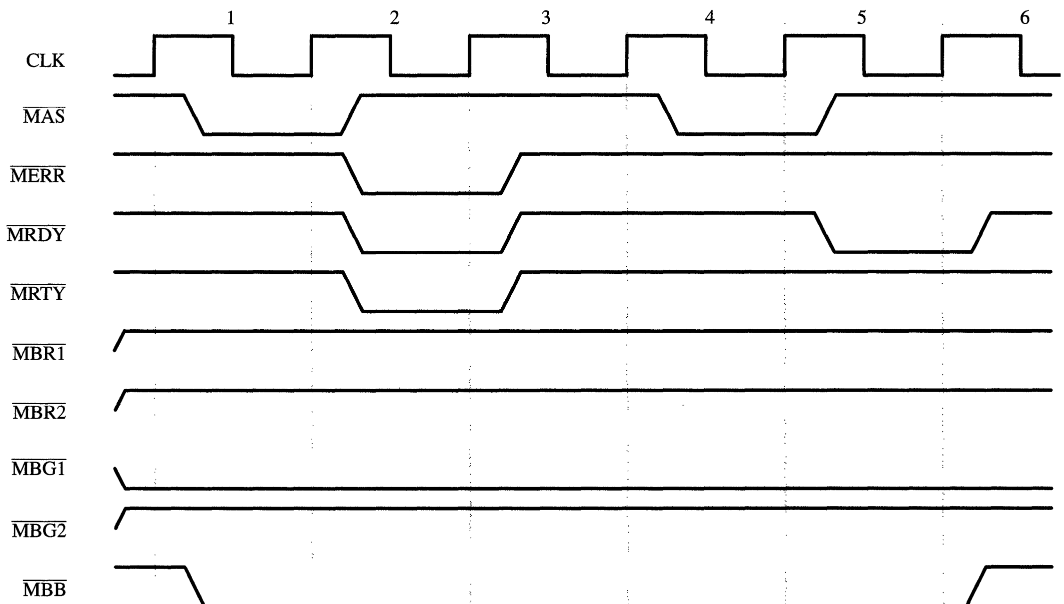


Figure 11–29. MBus Retry

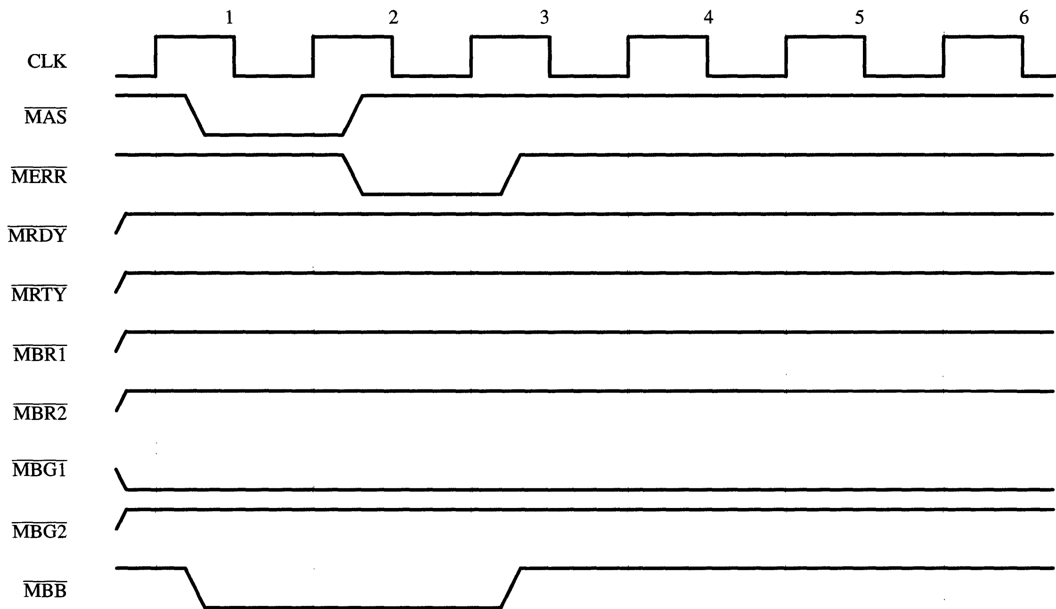


Figure 11–30. MBus Error (Bus Error)

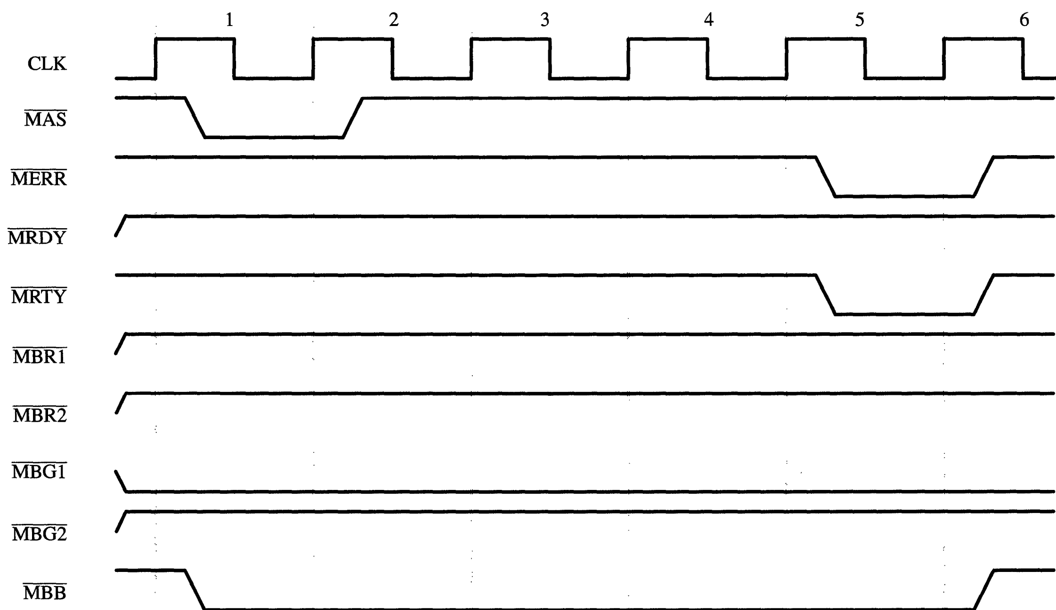


Figure 11–31. MBus Error (Timeout)

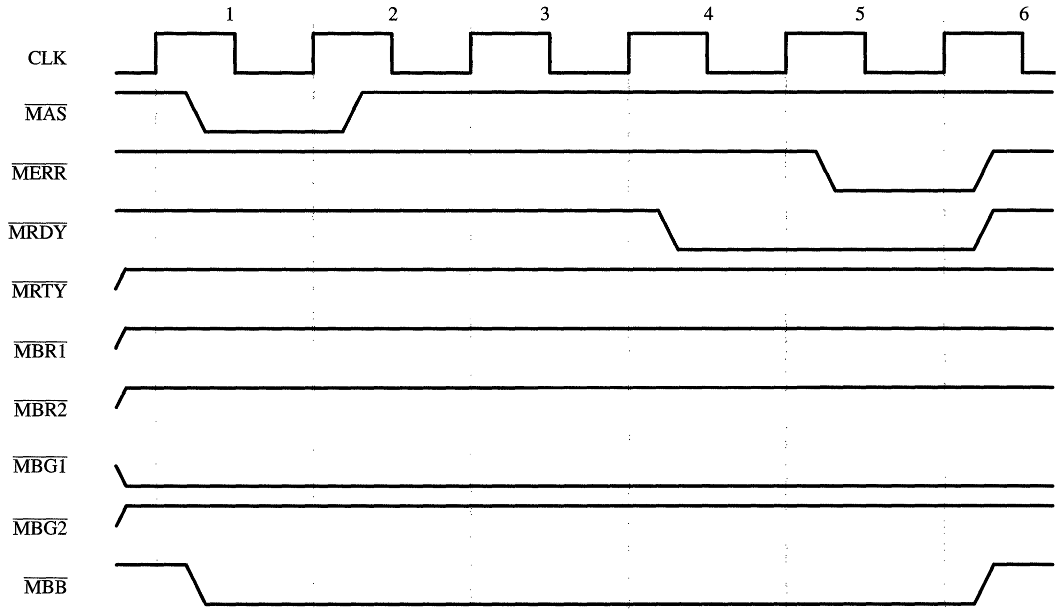


Figure 11–32. MBus Error (Uncorrectable)

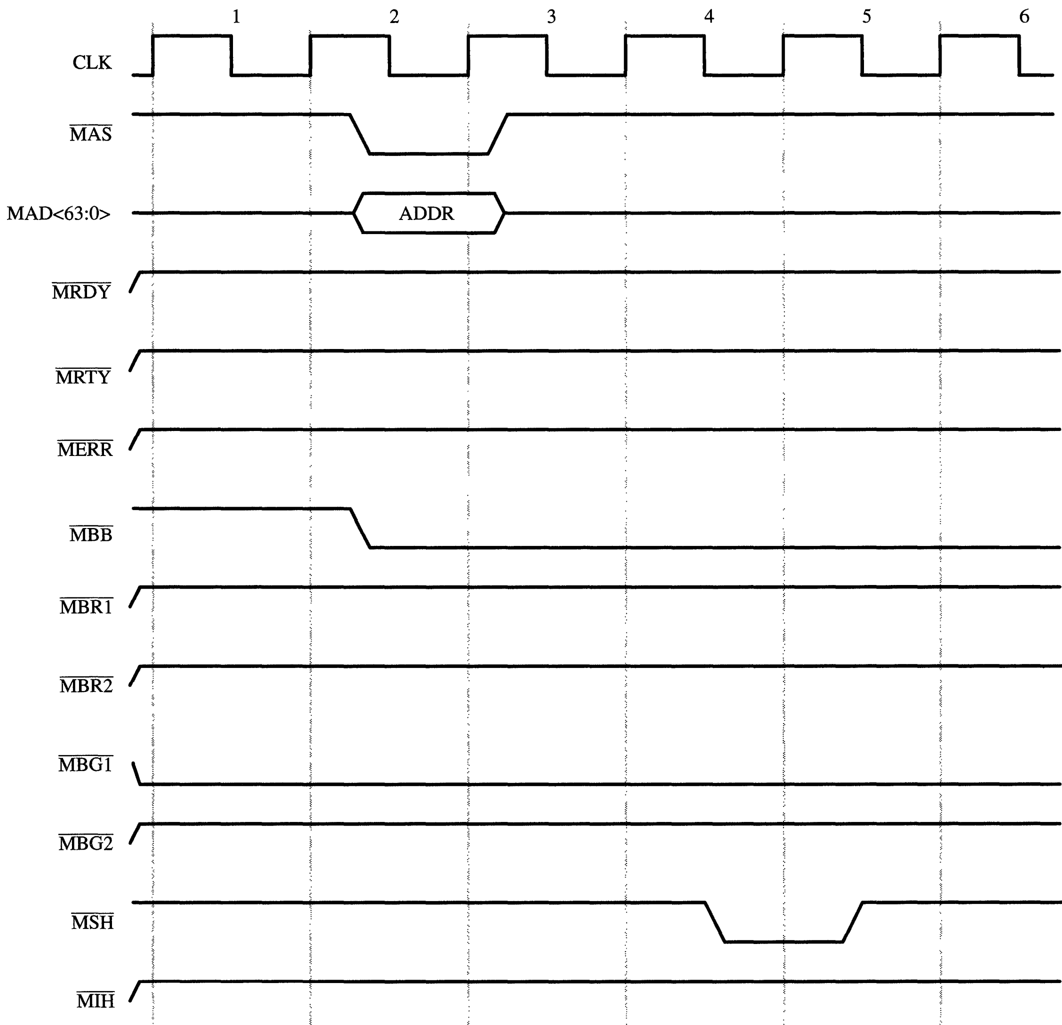


Figure 11–33. MBus Coherent Read—Shared* Data (page 1 of 2)

* This timing diagram illustrates a Coherent Read in which the requested data exists in one or more caches in the system, but is not owned by any cache. These caches must assert $\overline{\text{MSH}}$ on cycle A+2 as shown for the CY7C605 or during cycle A+3 for the RT625.

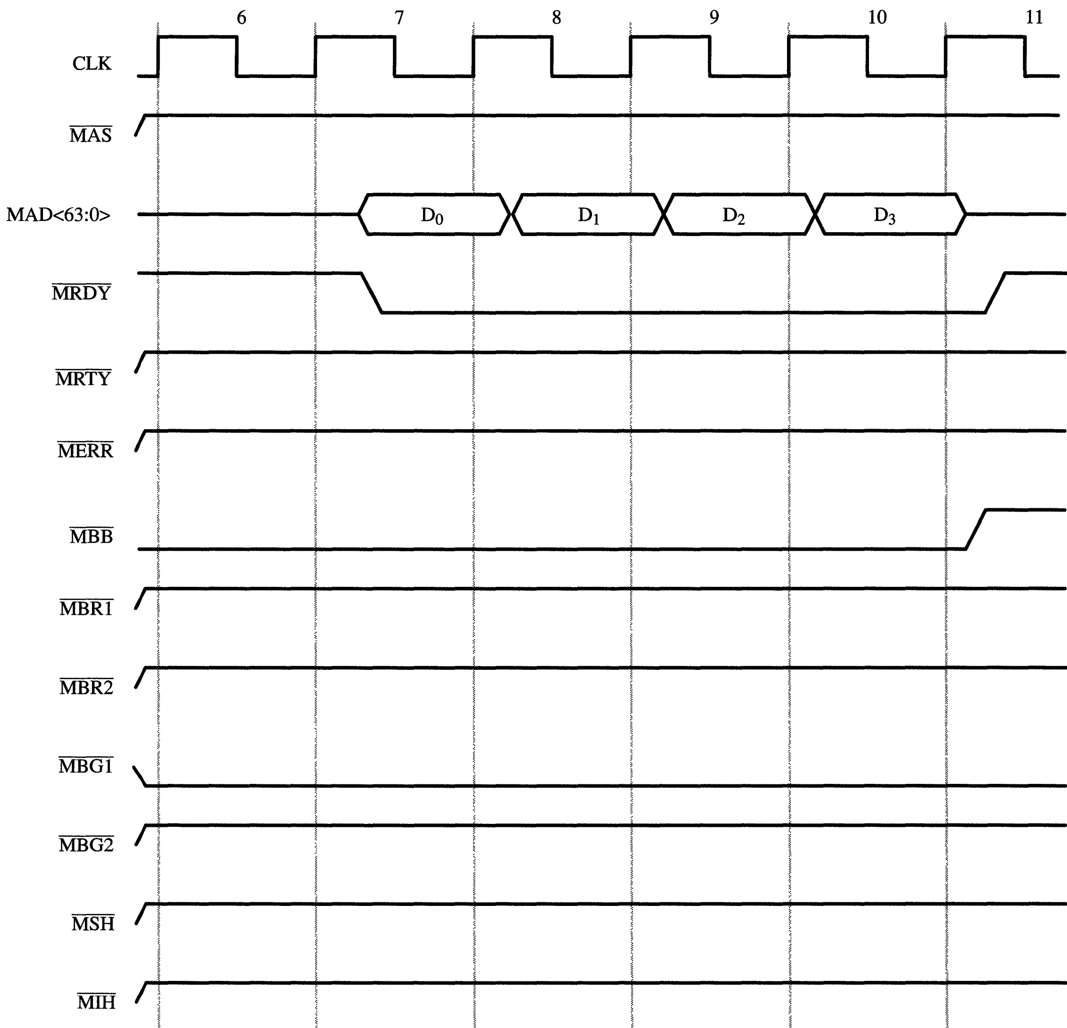


Figure 11–33. MBus Coherent Read—Shared Data (page 2 of 2)

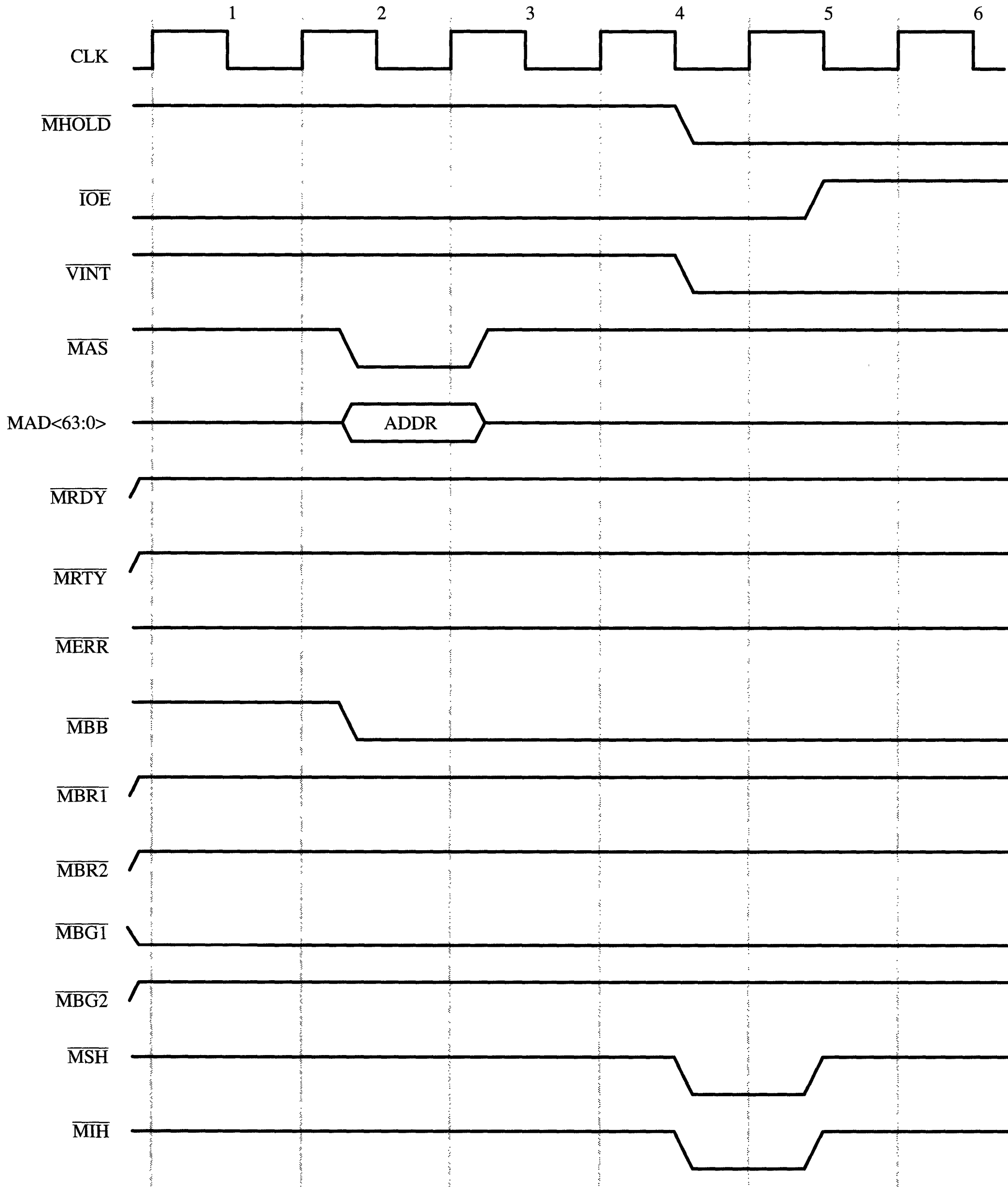


Figure 11–34. MBus Coherent Read—Owned Data (CY7C605) (Slow Memory)* (page 1 of 2)

* This timing diagram illustrates a Coherent Read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) must assert $\overline{\text{MSH}}$. Only the owning cache will assert $\overline{\text{MIH}}$ on cycle A+2 and supply the data.

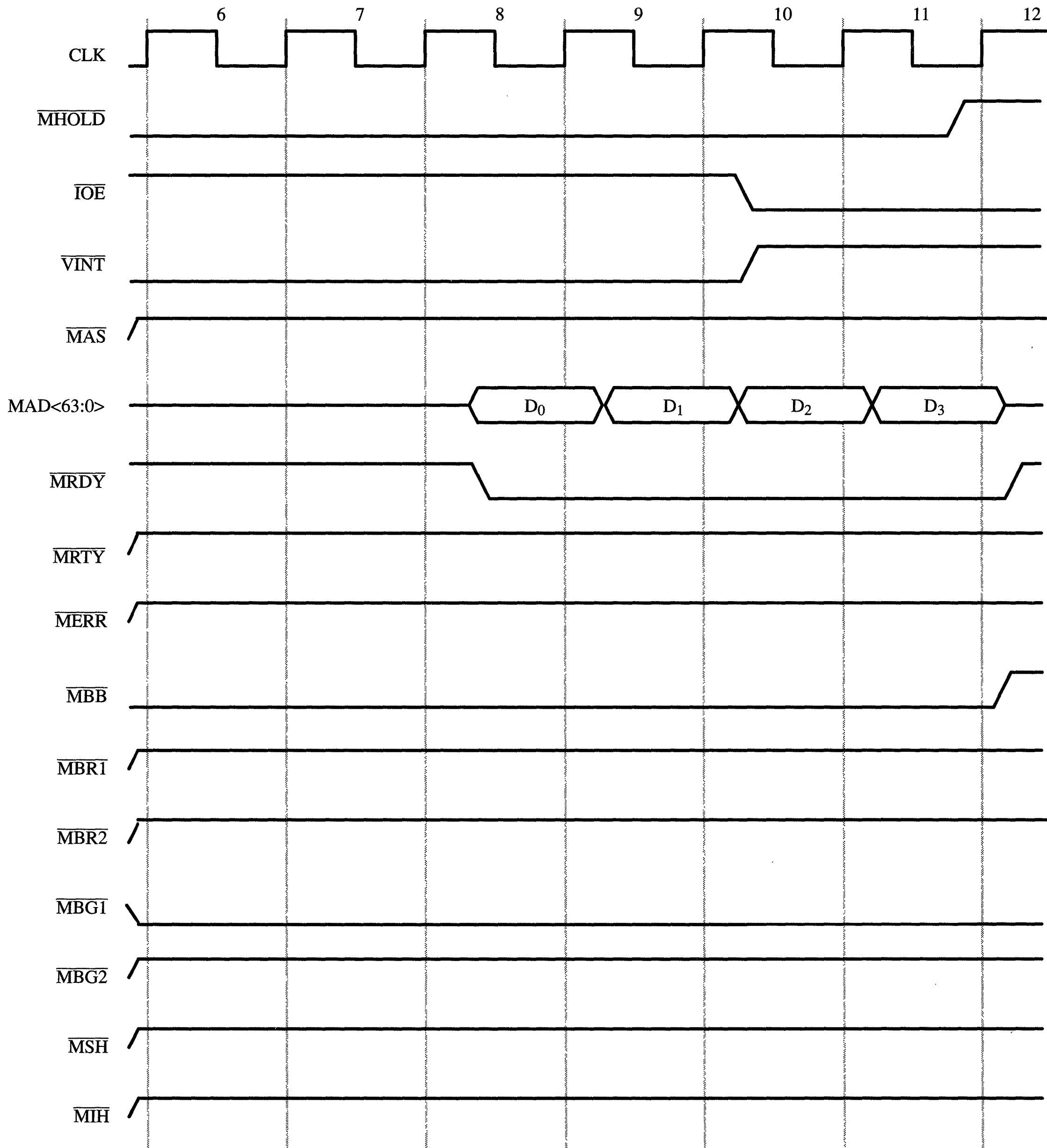


Figure 11–34. MBus Coherent Read—Owned Data (CY7C605) (Slow Memory) (page 2 of 2)

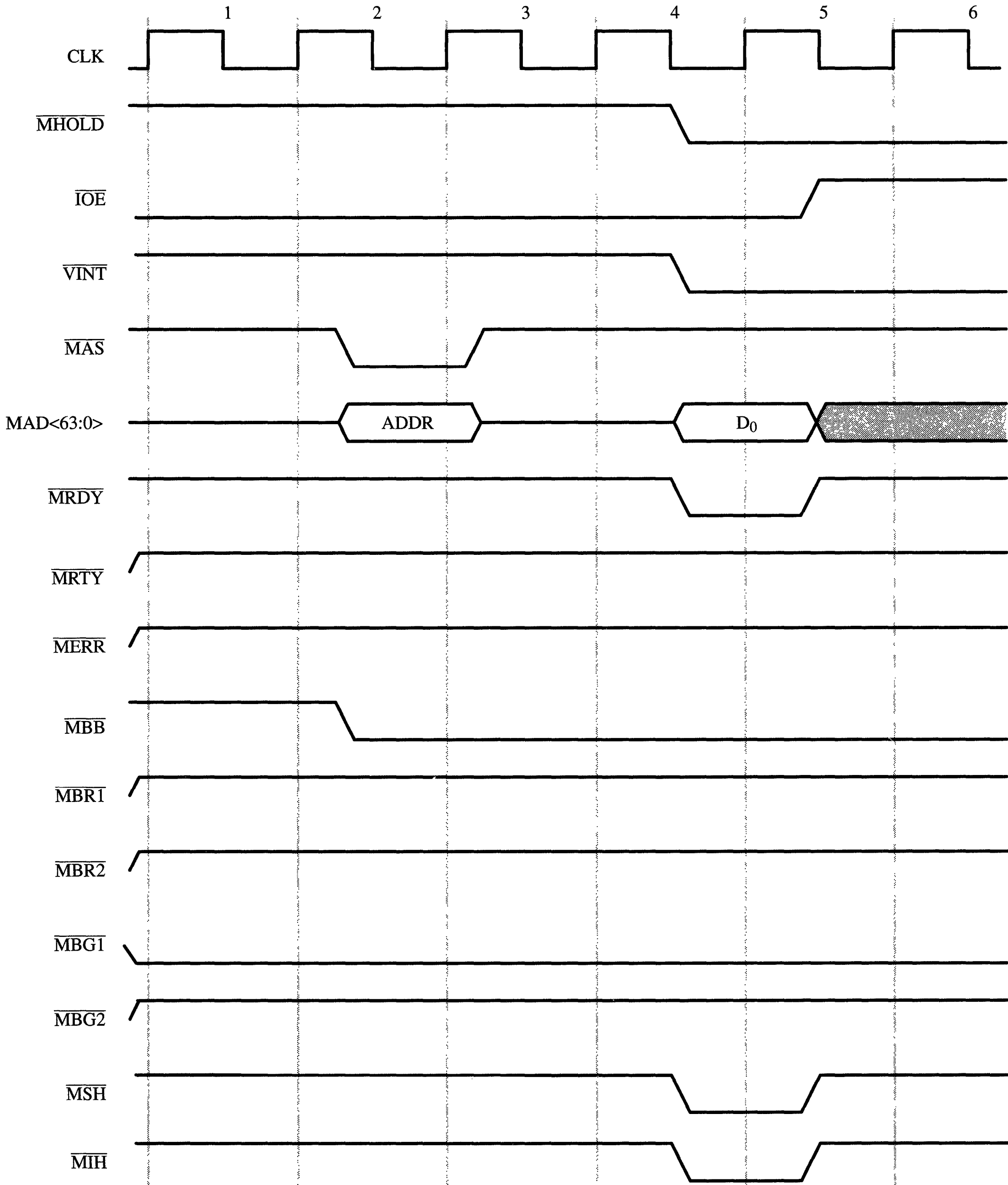


Figure 11–35. MBus Coherent Read—Owned Data (CY7C605) (Fast Memory)* (page 1 of 2)

* This timing diagram illustrates a Coherent Read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) will assert $\overline{MSH}$. Only the owning cache asserts $\overline{MIH}$ on cycle A+2 and supplies the data. In this case, memory has already started to respond (with MRDY in the A+2 cycle) and thus must get off the bus immediately to allow the cache which owns the data to drive the bus.

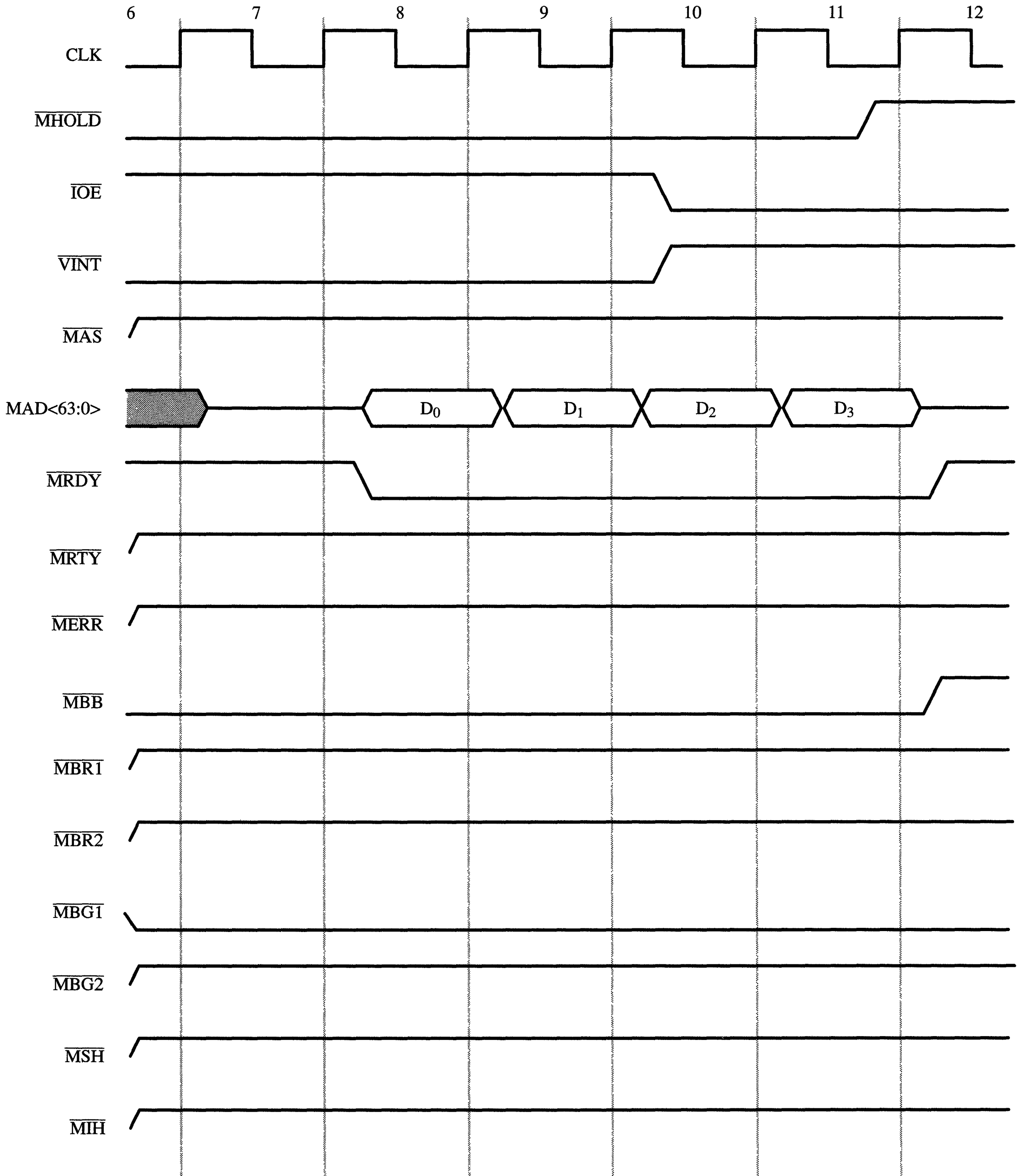


Figure 11–35. MBus Coherent Read—Owned Data (Fast Memory) (page 2 of 2)

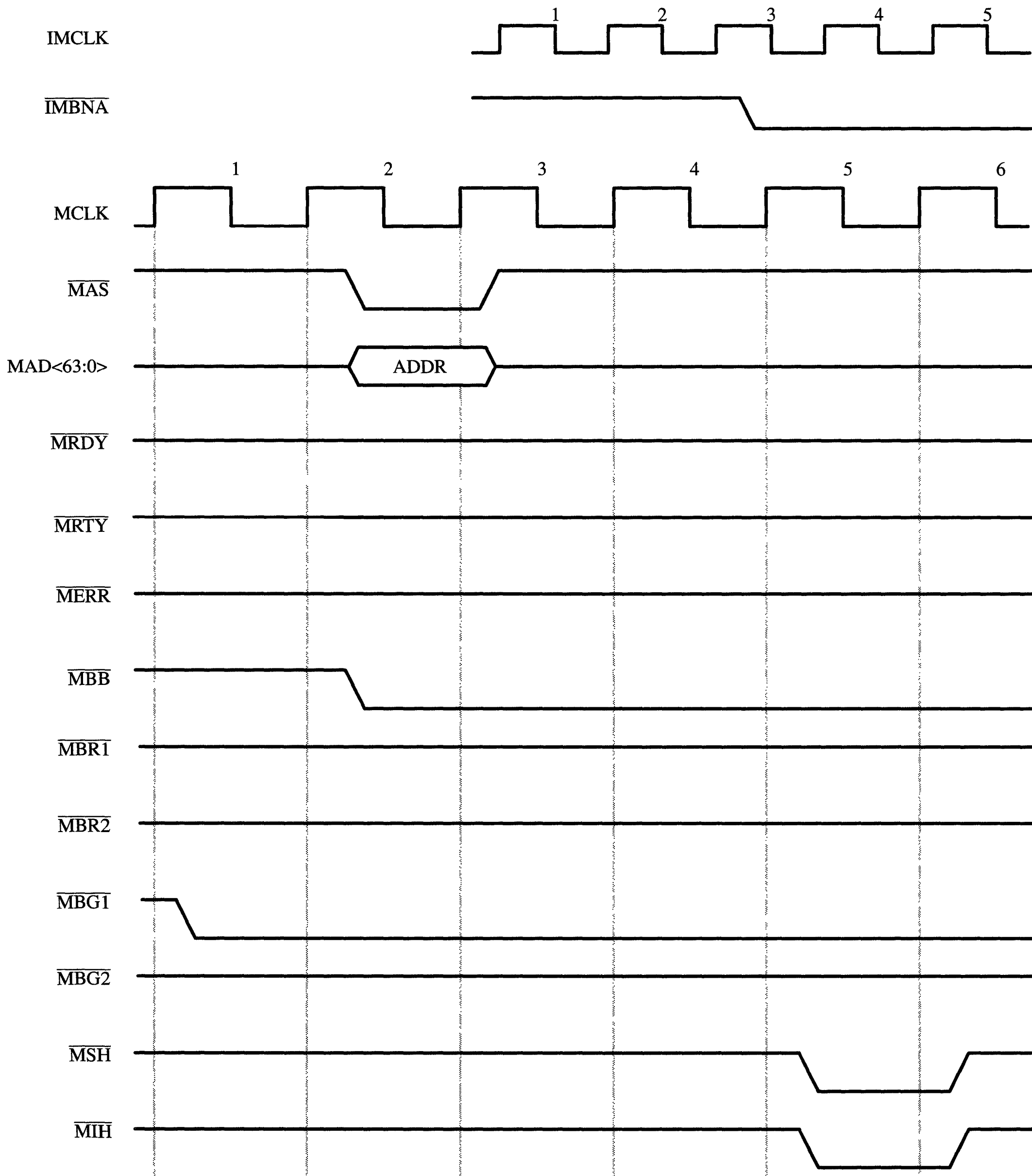


Figure 11–36. MBus Coherent Read—Owned Data (RT625) (Slow Memory)* (page 1 of 2)

* This timing diagram illustrates a Coherent Read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) must assert $\overline{MSH}$. Only the owning cache will assert $\overline{MIH}$ on cycle A+3 and supply the data.

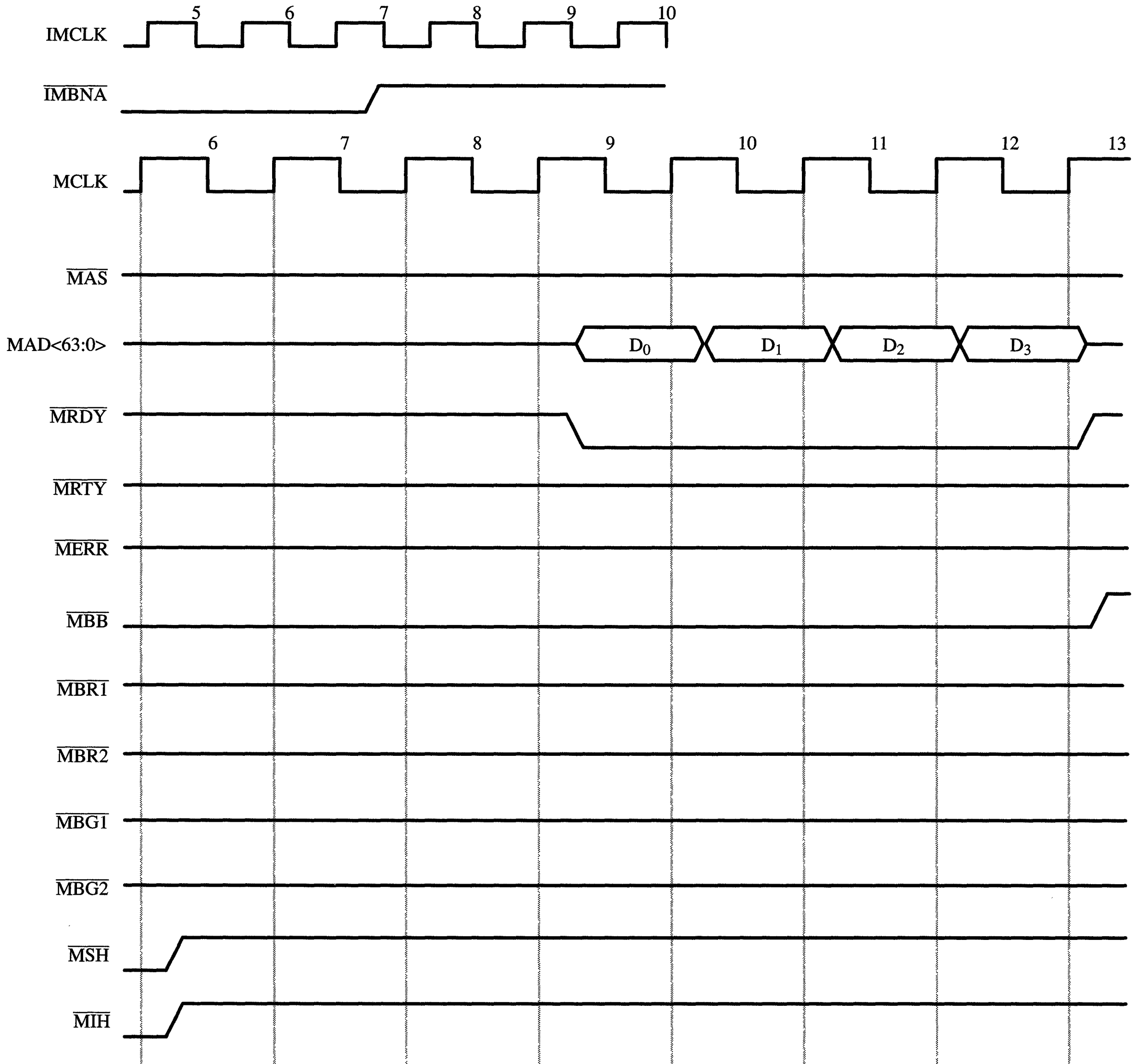


Figure 11–36. MBus Coherent Read—Owned Data (RT625) (Slow Memory) (page 2 of 2)

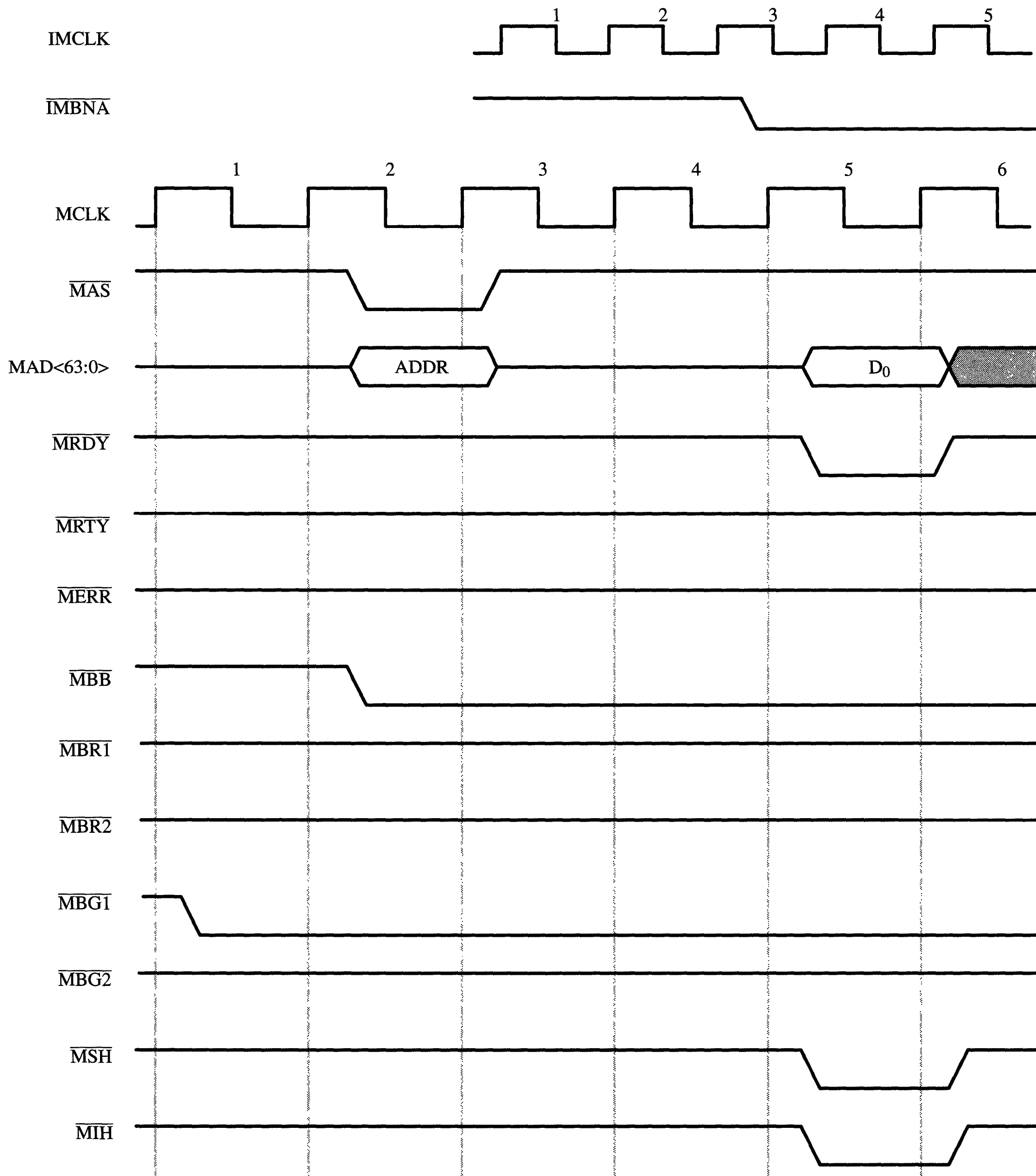


Figure 11–37. MBus Coherent Read—Owned Data (RT625) (Fast Memory)* (page 1 of 2)

* This timing diagram illustrates a Coherent Read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) will assert $\overline{\text{MSH}}$. Only the owning cache asserts $\overline{\text{MIH}}$ on cycle A+3 and supplies the data. In this case, memory has already started to respond (with MRDY in the A+3 cycle) and thus must get off the bus immediately to allow the cache which owns the data to drive the bus.

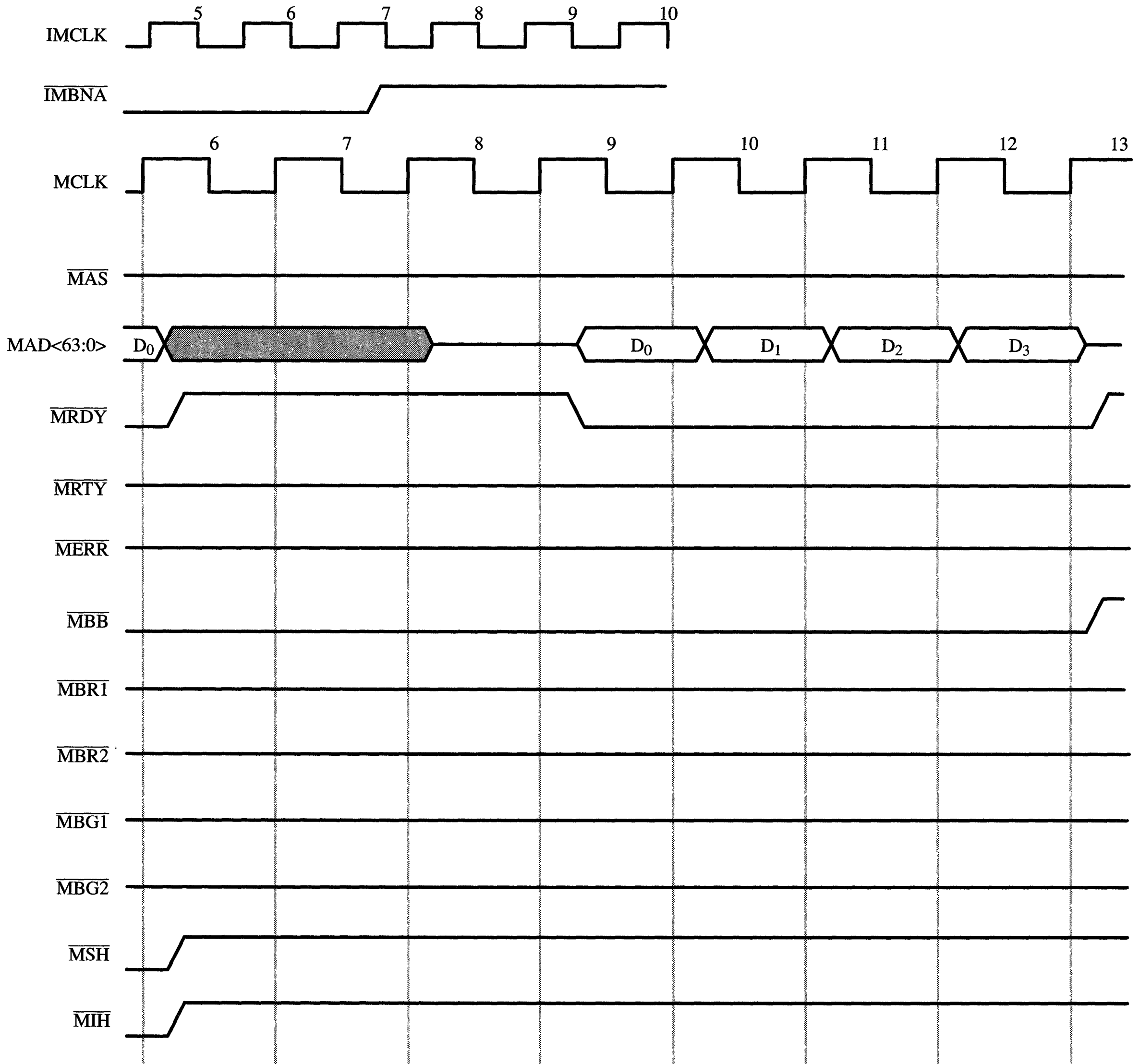


Figure 11–37. MBus Coherent Read—Owned Data (RT625) (Fast Memory) (page 2 of 2)

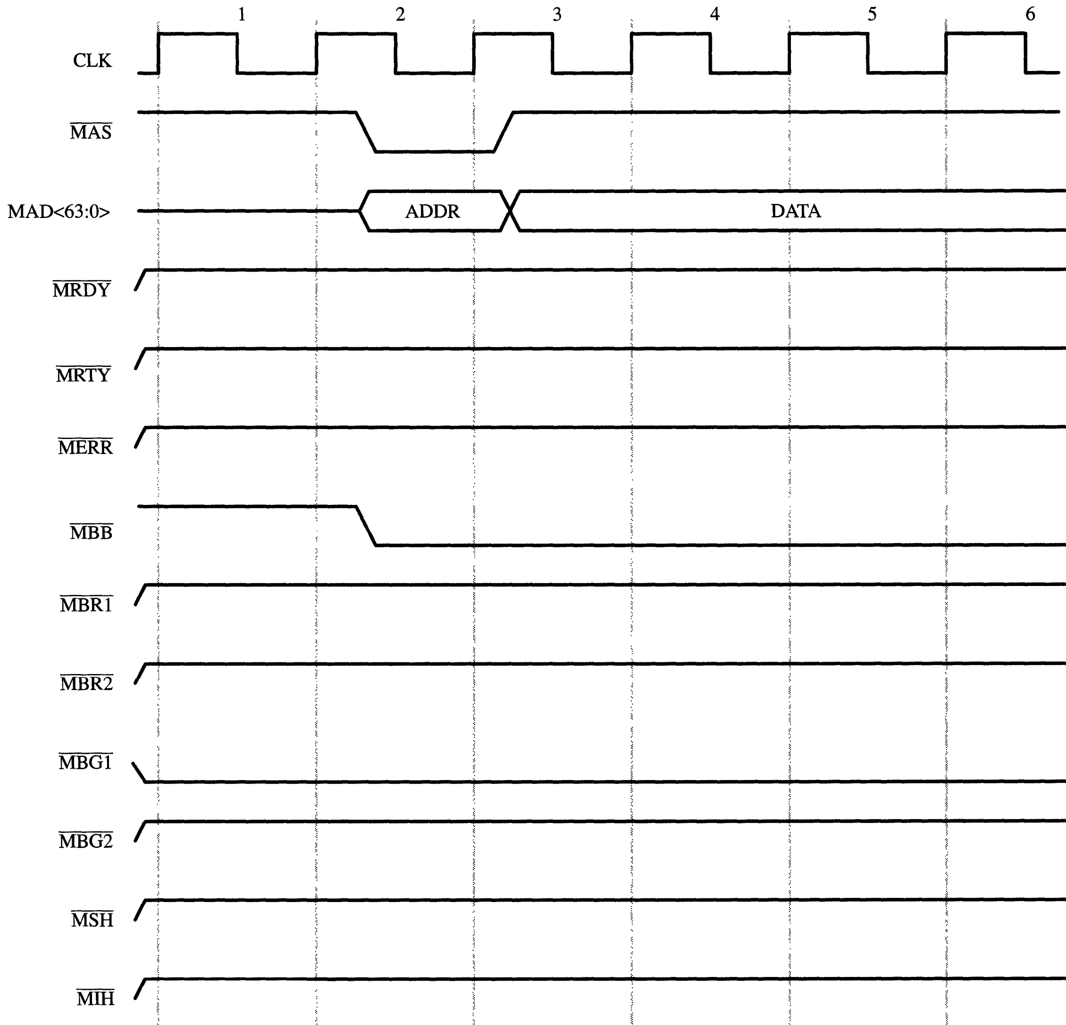


Figure 11–38. MBus Coherent Write and Invalidate* **(page 1 of 2)**

* This timing diagram illustrates a Coherent Write and Invalidate operation in which one or more other caches have a copy of the cache line. The other caches invalidate their copy of the cache line but do not assert $\overline{\text{MSH}}$. Memory (or second-level cache) asserts $\overline{\text{MRDY}}$ during A+2 for the CY7C605 or A+3 or later for the RT625. System caches which contain the data being invalidated do not assert $\overline{\text{MSH}}$ during this transaction. The CY7C605 and RT625 only issue this transaction in write-through mode.

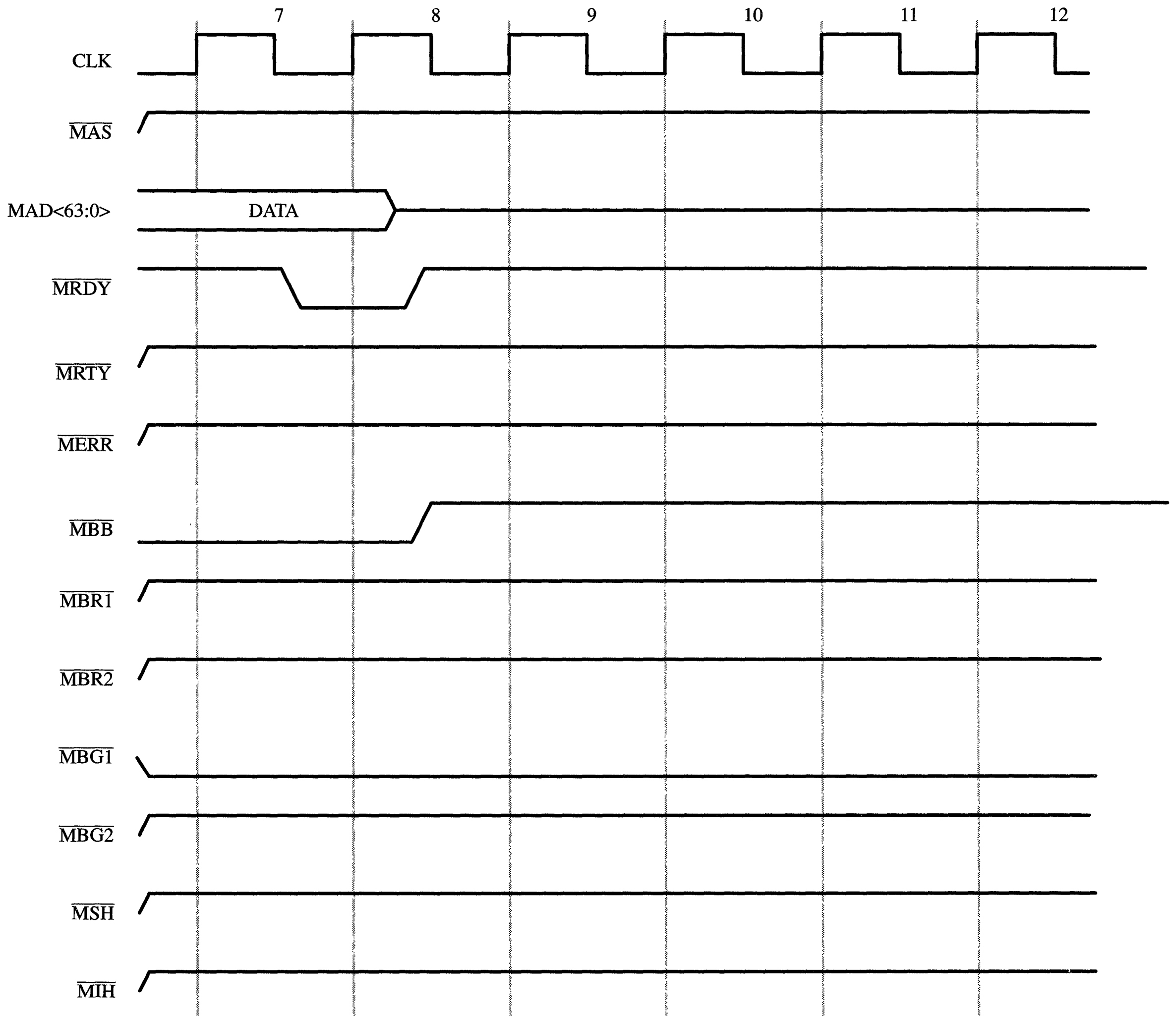


Figure 11–38. MBus Coherent Write and Invalidate (page 2 of 2)

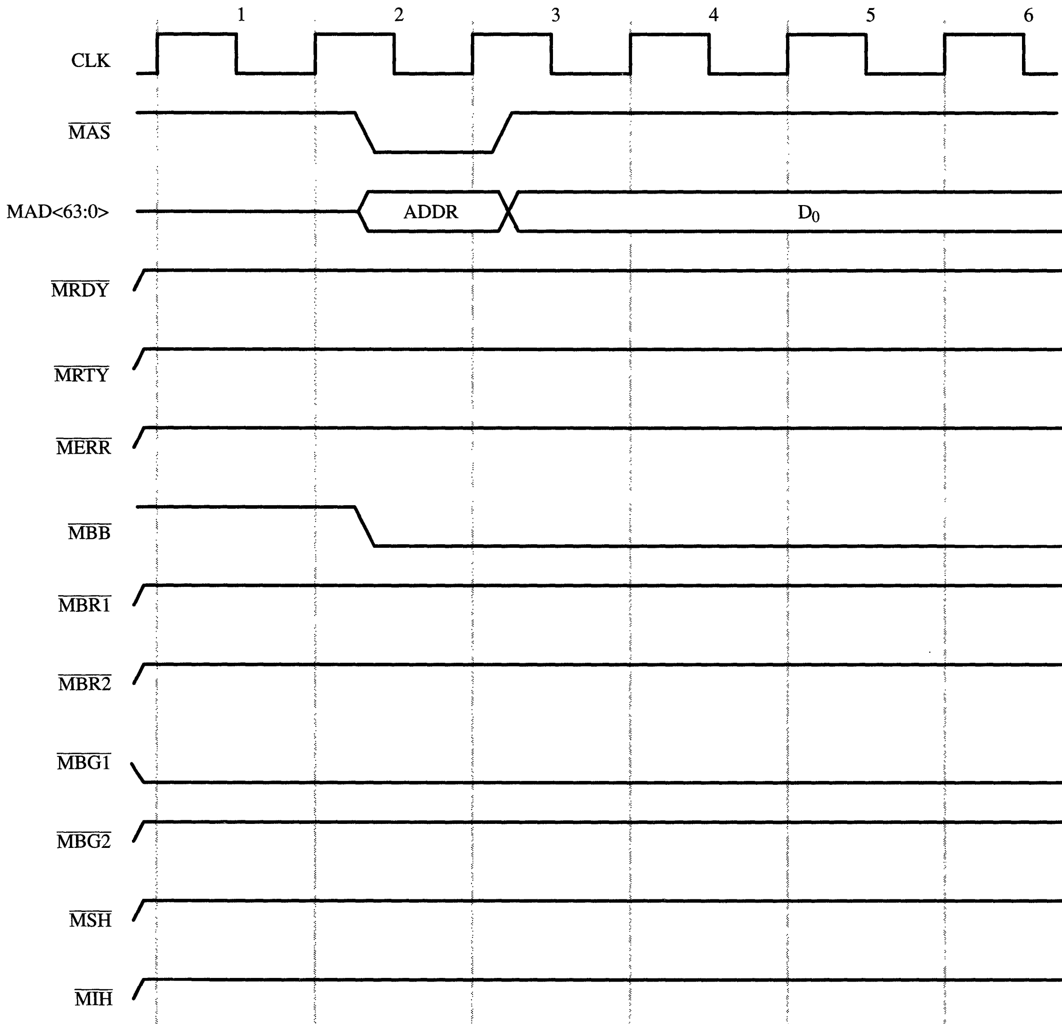


Figure 11–39. MBus Coherent Write and Invalidate (RT625) (Block Copy/Fill)* (page 1 of 2)

* This timing diagram illustrates a Coherent Write and Invalidate operation for Block Copy/Fill transactions (RT625 only) in which one or more other caches have a copy of the cache line. The other caches invalidate their copy of the cache line but do not assert $\overline{MSH}$. Memory (or second-level cache) asserts $\overline{MRDY}$ during A+3 or later for the RT625. System caches which contain the data being invalidated do not assert $\overline{MSH}$ during this transaction.

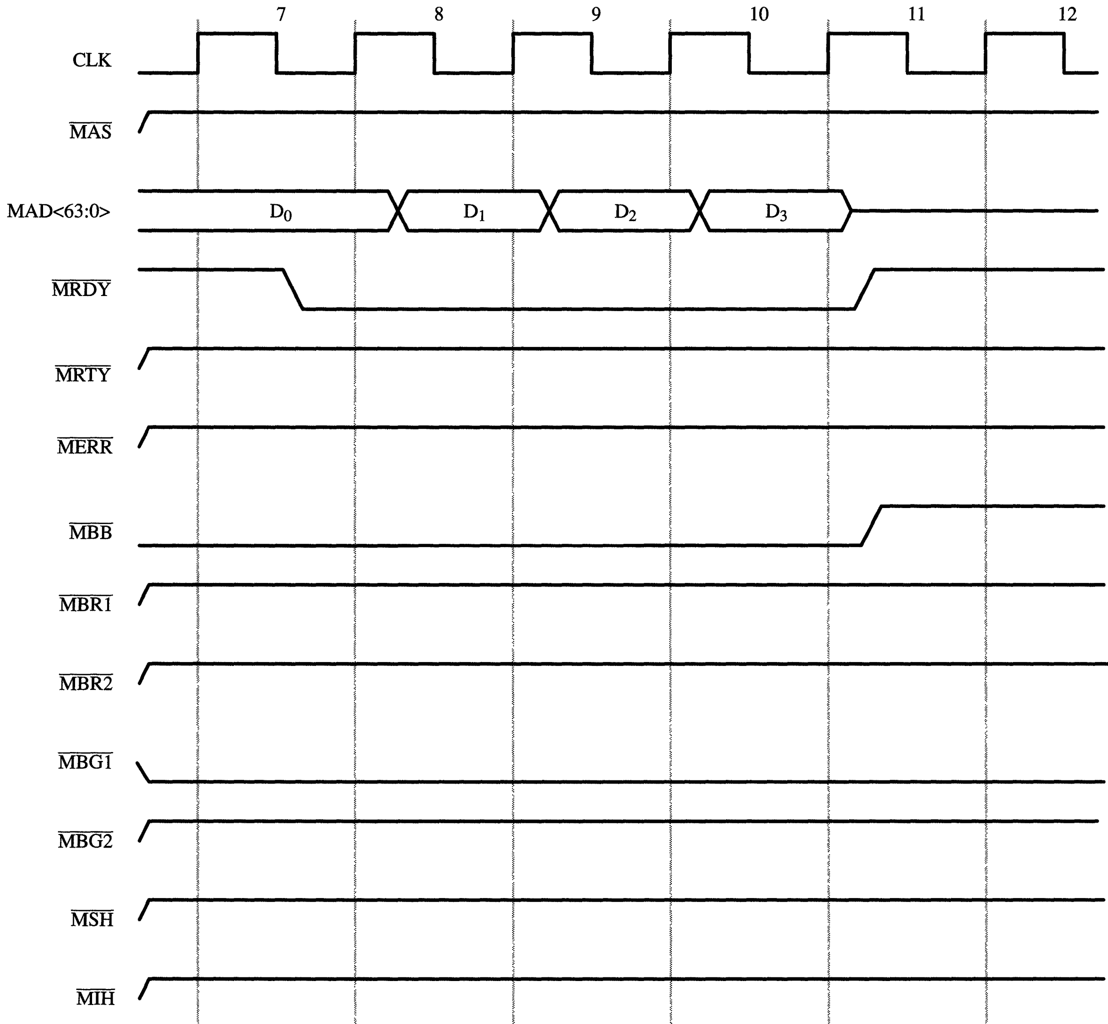


Figure 11–39. MBus Coherent Write and Invalidate (RT625) (Block Copy/Fill) (page 2 of 2)

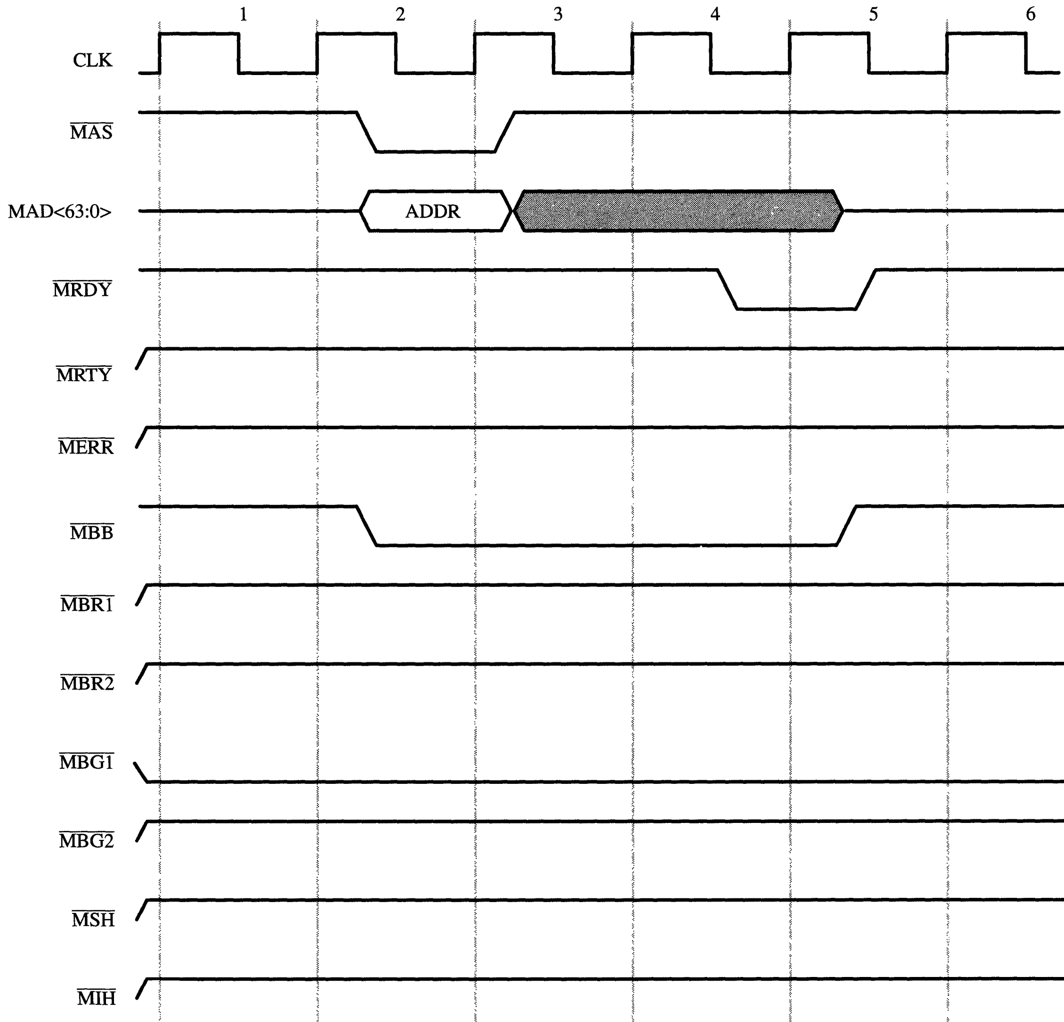


Figure 11–40. MBus Coherent Invalidate*

* This timing diagram illustrates a Coherent Invalidate operation. Memory (or second-level cache) asserts $\overline{\text{MRDY}}$ during A+2 for the CY7C605 or A+3 or later for the RT625. System caches which contain the data being invalidated do not assert $\overline{\text{MSH}}$ during this transaction.

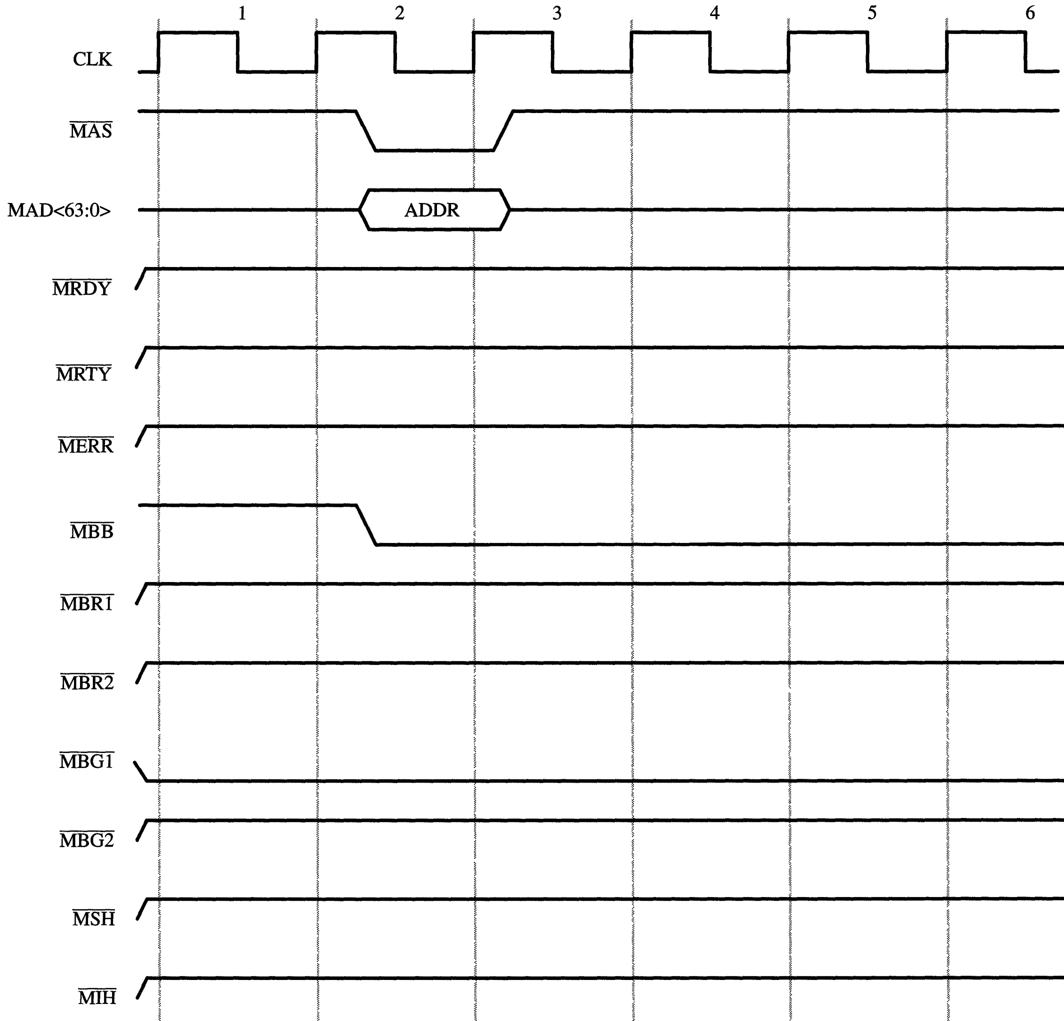


Figure 11–41. MBus Coherent Read and Invalidate—Shared Data* (page 1 of 2)

* This timing diagram illustrates a Coherent Read and Invalidate in which the requested data may exist in one or more caches in the system. $\overline{\text{MSH}}$ is not asserted for the Coherent Read and Invalidate transaction.

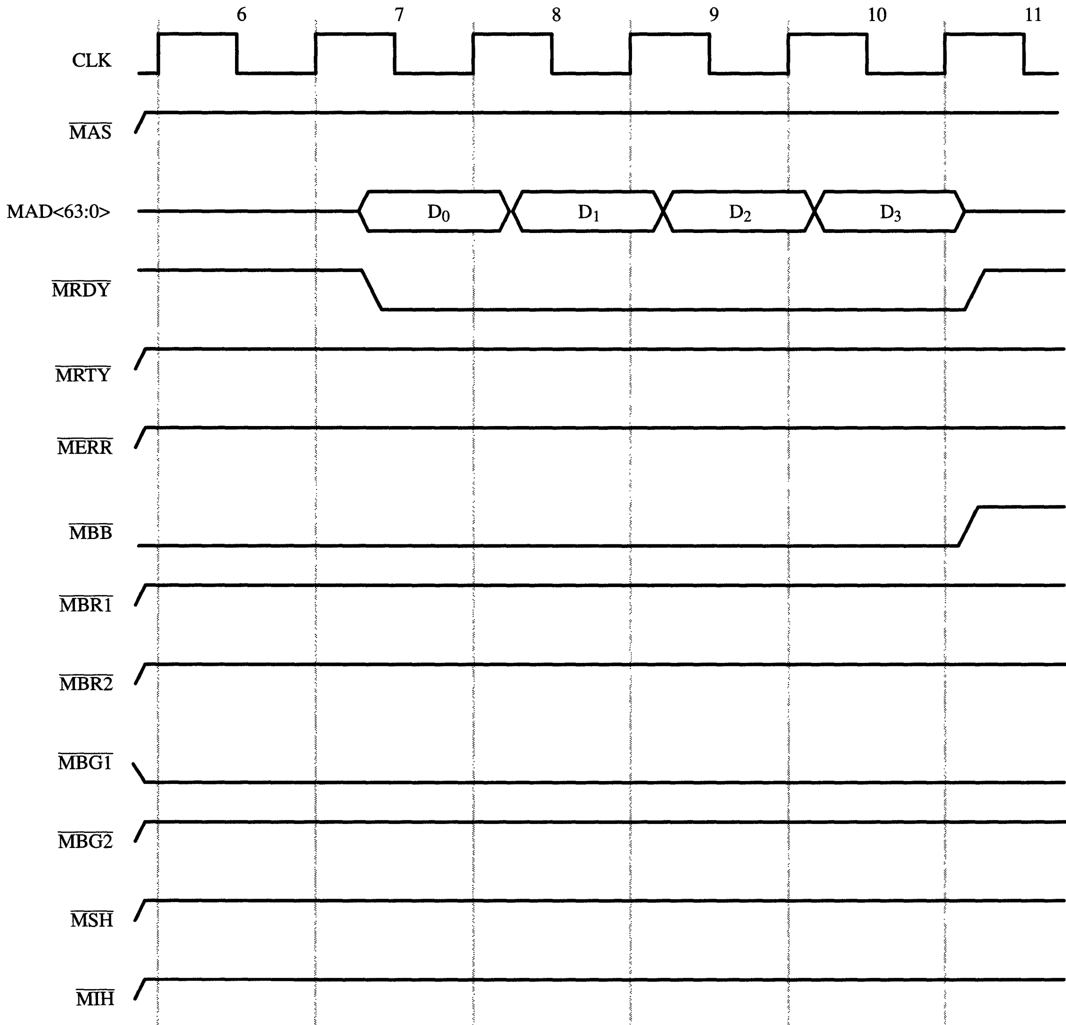


Figure 11–42. MBus Coherent Read and Invalidate—Shared Data (page 2 of 2)

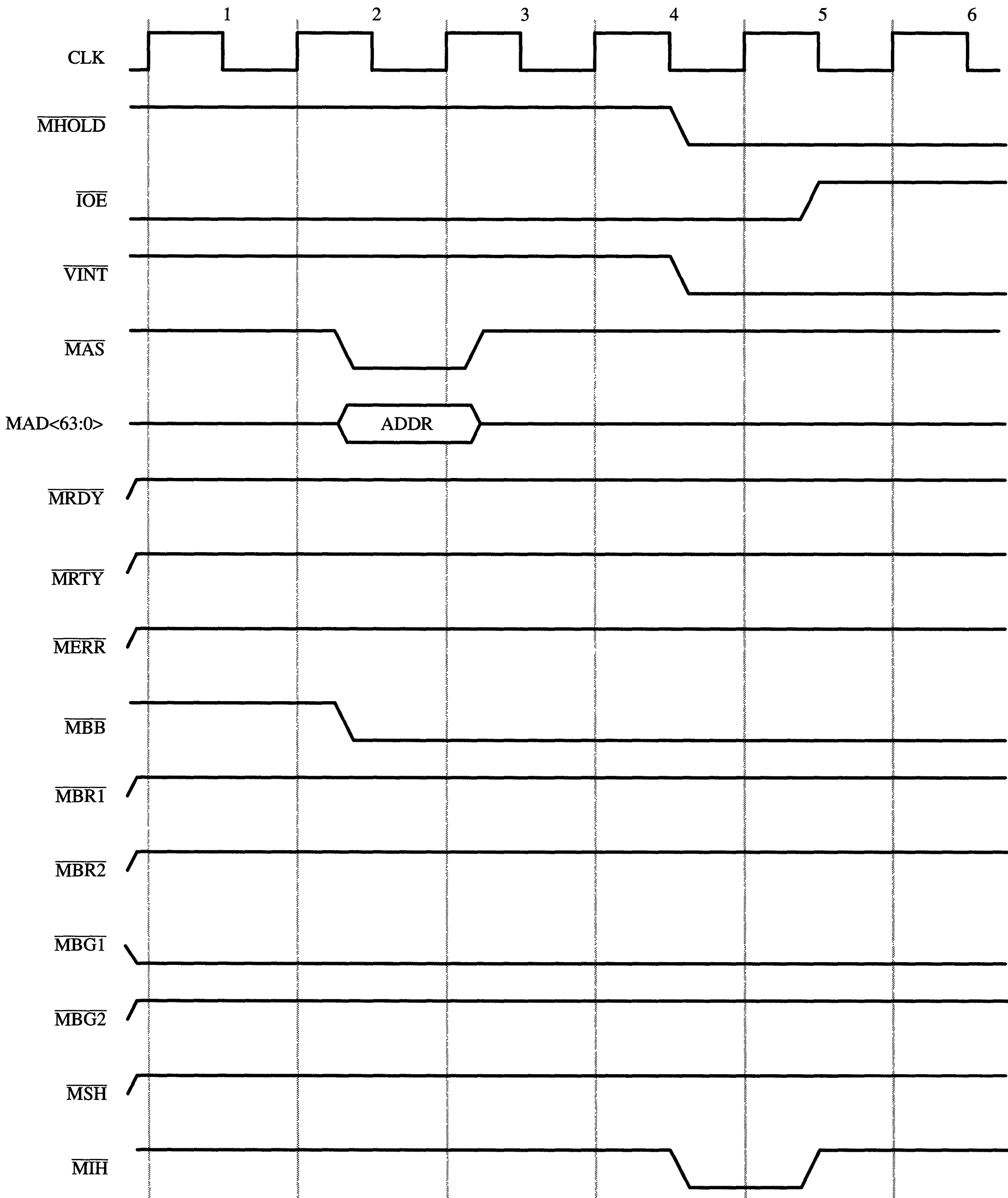


Figure 11–42. MBus Coherent Read and Invalidate—Owned Data (CY7C605) (Slow Memory)* (page 1 of 2)

* This timing diagram illustrates a Coherent Read and Invalidate in which the requested data exists in one or more caches in the system and is owned by a cache. Only the owning cache asserts MIH on cycle A+2 and supplies the data. After supplying the data, the owning cache invalidates its copy; all other caches with copies of the data also invalidate their copies.

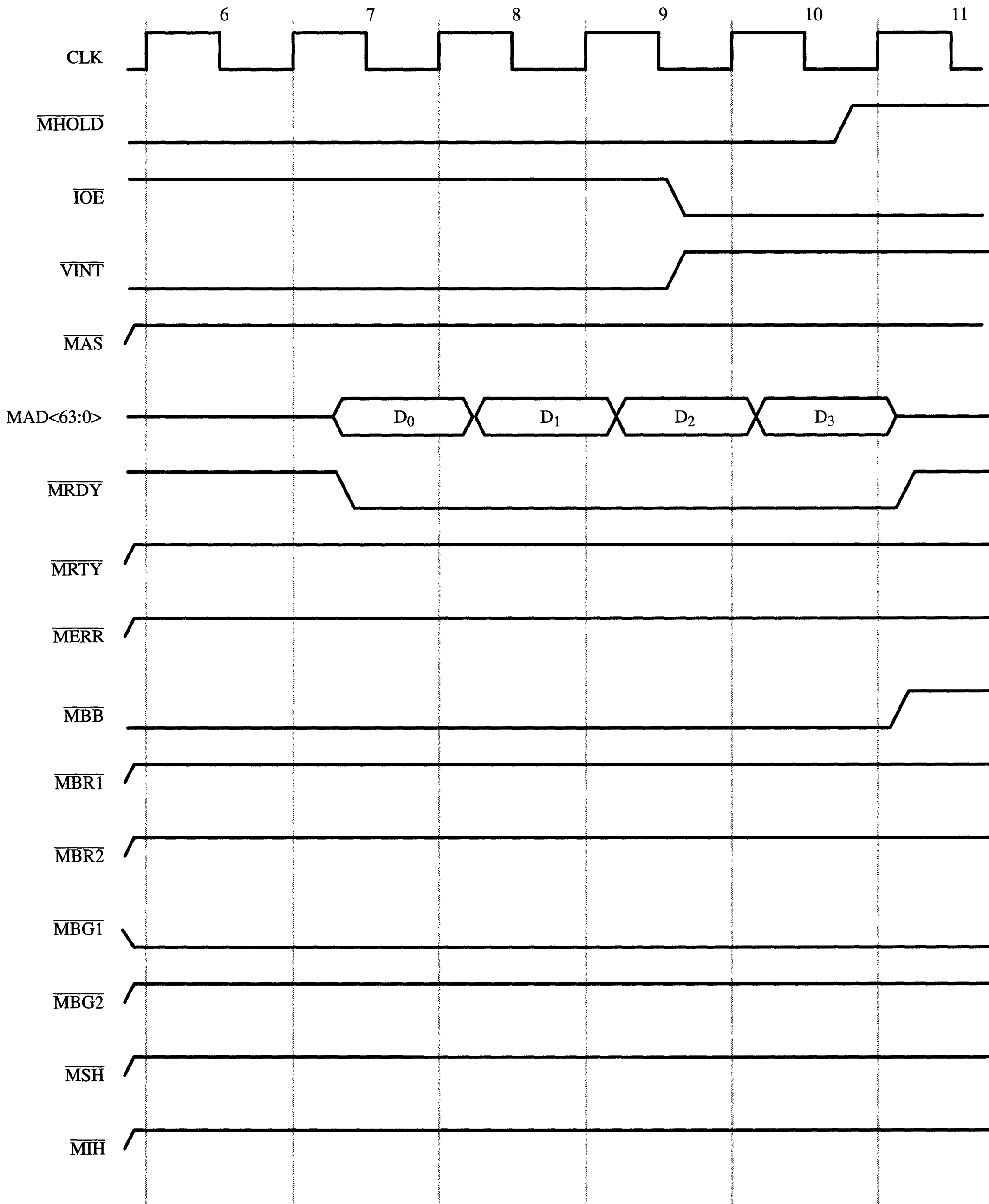


Figure 11–43. MBus Coherent Read and Invalidate—Owned Data (CY7C605) (Slow Memory) (page 2 of 2)

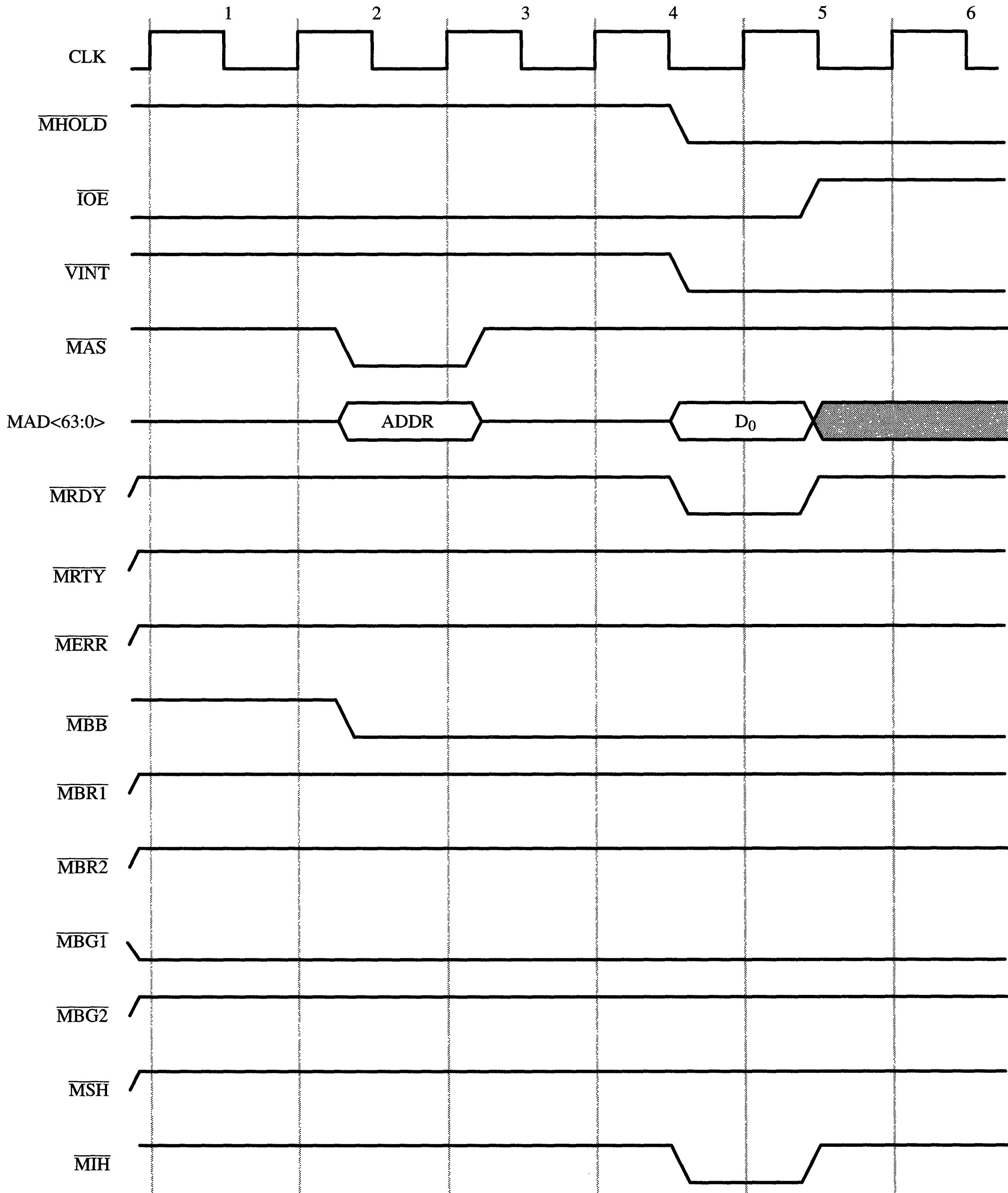


**Figure 11–43. MBus Coherent Read and Invalidate—Owned Data (CY7C605) (Fast Memory)*
(page 1 of 2)**

* This timing diagram illustrates a Coherent Read and Invalidate in which the requested data exists in one or more caches in the system and is owned by a cache. Only the owning cache asserts $\overline{\text{MIH}}$ on cycle A+2 and supplies the data. After supplying the data, the owning cache invalidates its copy; all other caches with copies of the data also invalidate their copies.

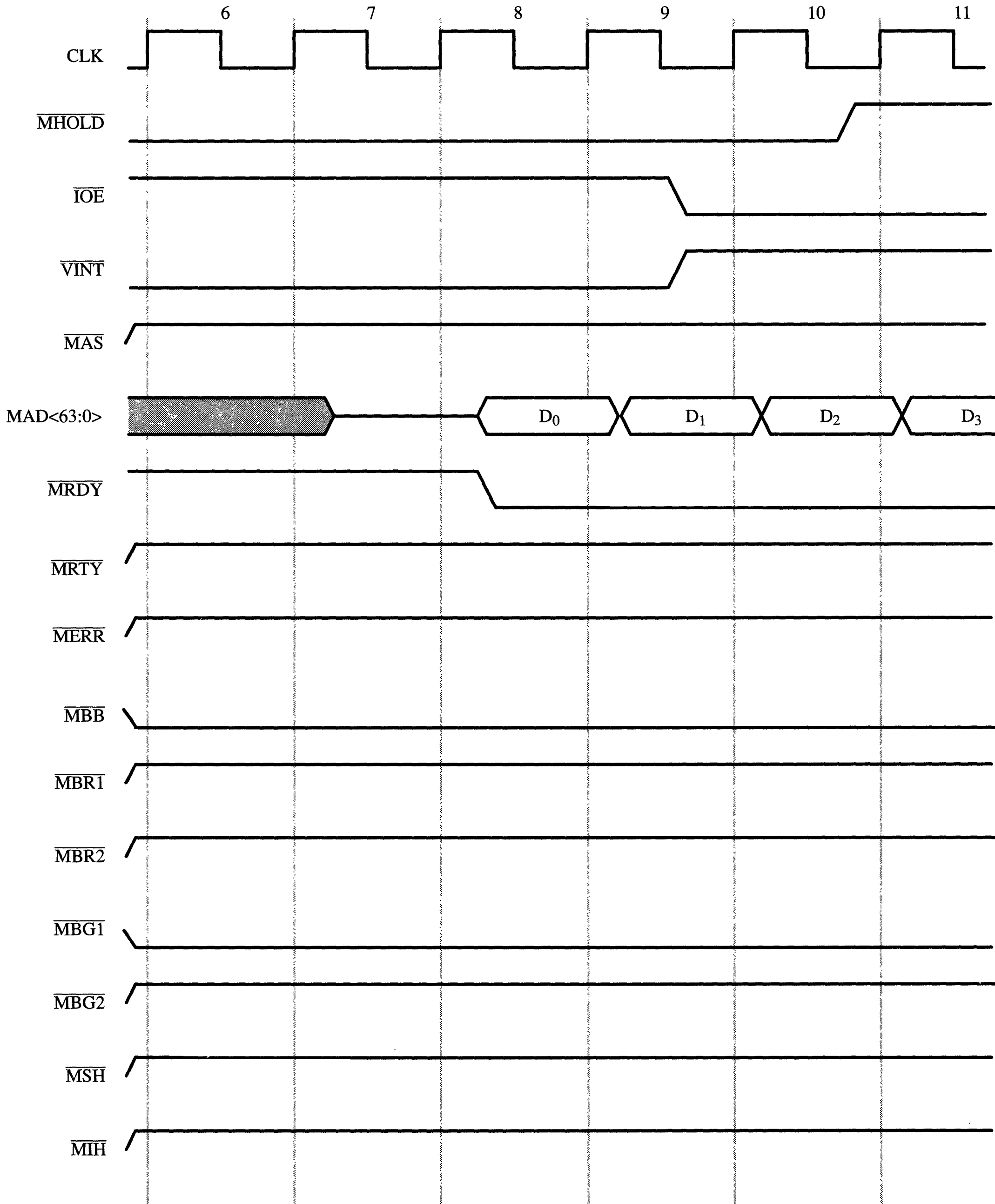


Figure 11–43. MBus Coherent Read and Invalidate—Owned Data (CY7C605) (Fast Memory) (page 2 of 2)

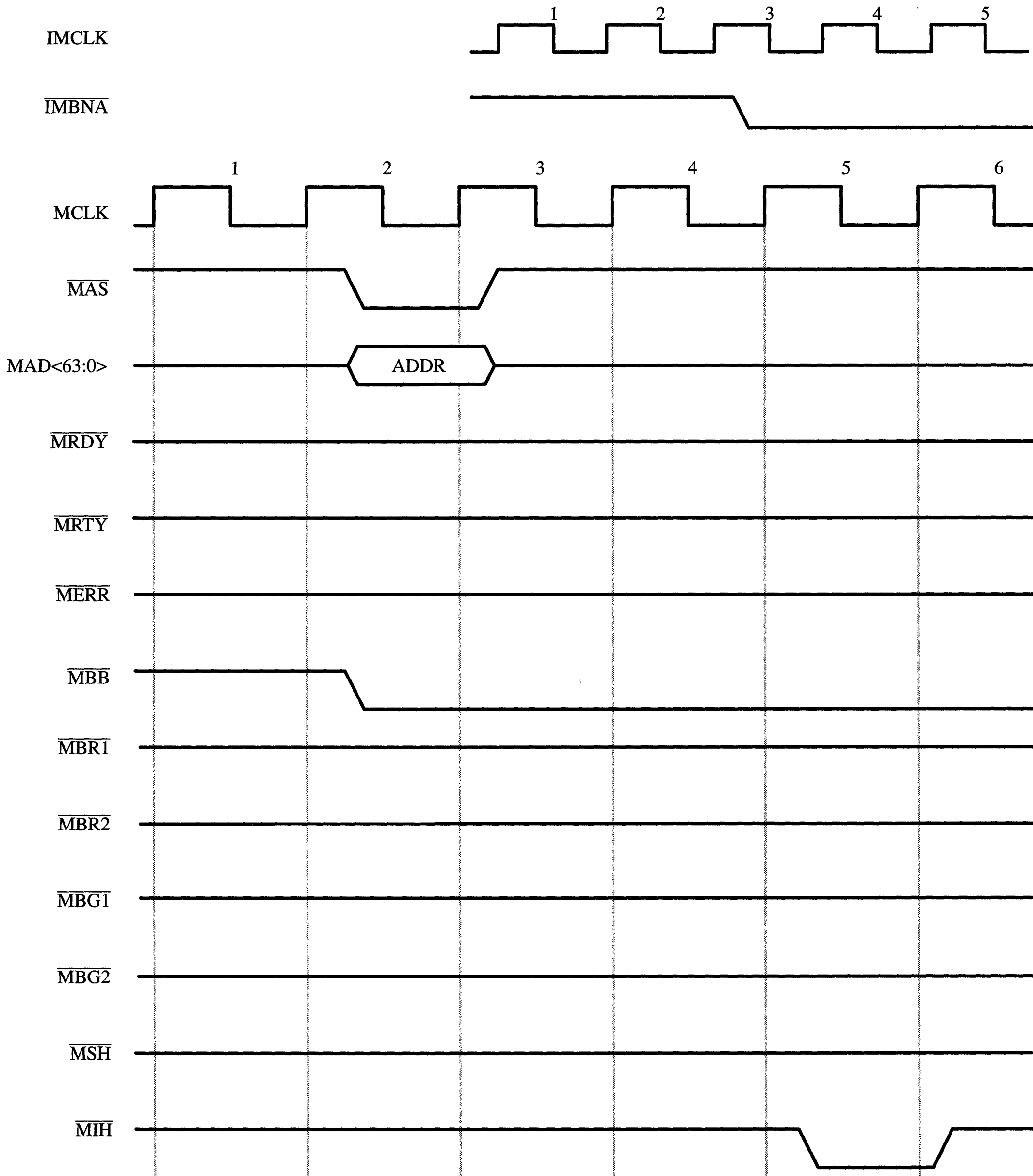


Figure 11–44. MBus Coherent Read and Invalidate—Owned Data (RT625) (Slow Memory)*
(page 1 of 2)

* This timing diagram illustrates a Coherent Read and Invalidate in which the requested data exists in one or more caches in the system and is owned by a cache. Only the owning cache asserts MIH on cycle A+3 and supplies the data. After supplying the data, the owning cache invalidates its copy; all other caches with copies of the data also invalidate their copies.

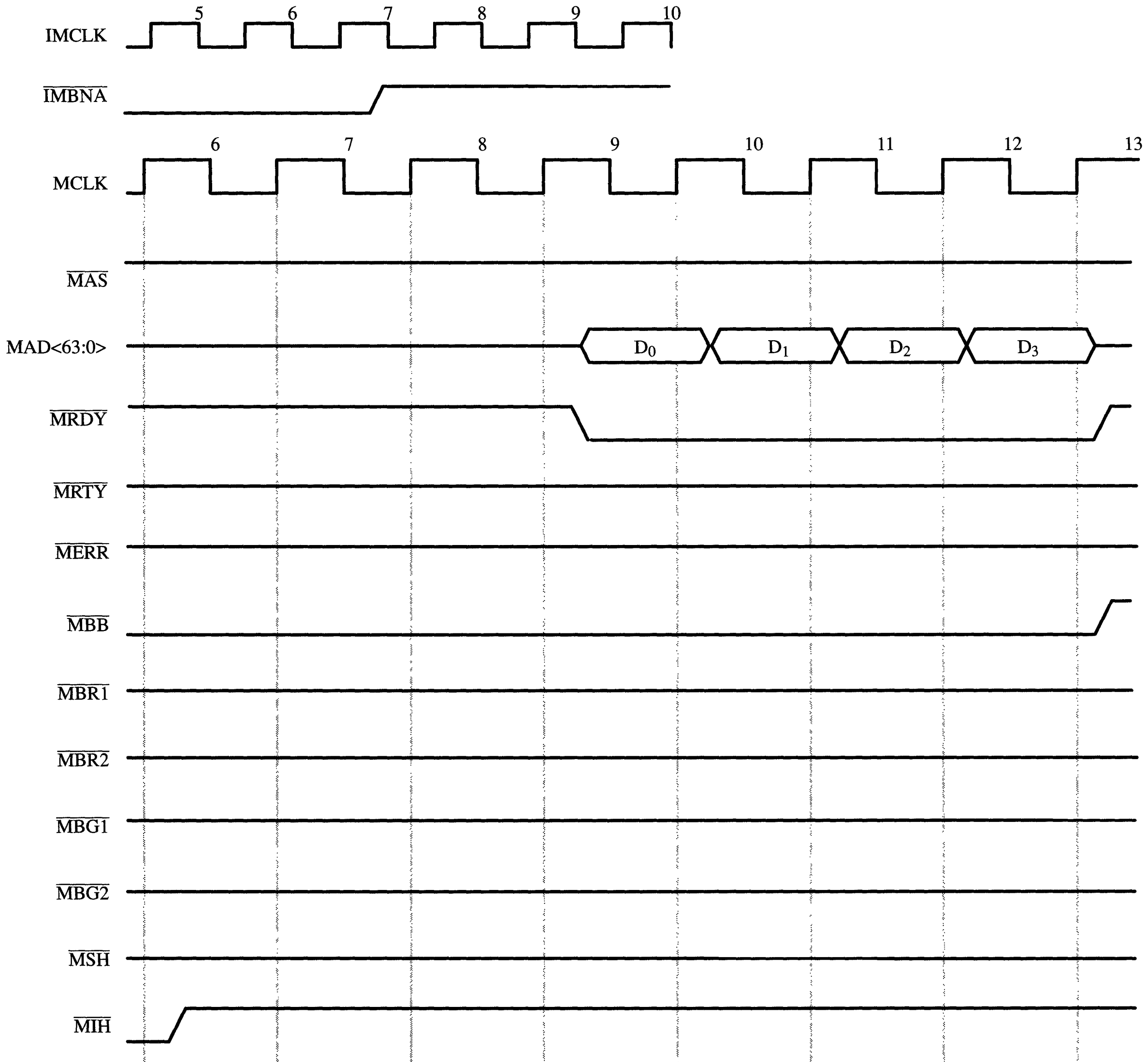


Figure 11–44. MBus Coherent Read and Invalidate—Owned Data (RT625) (Slow Memory) (page 2 of 2)

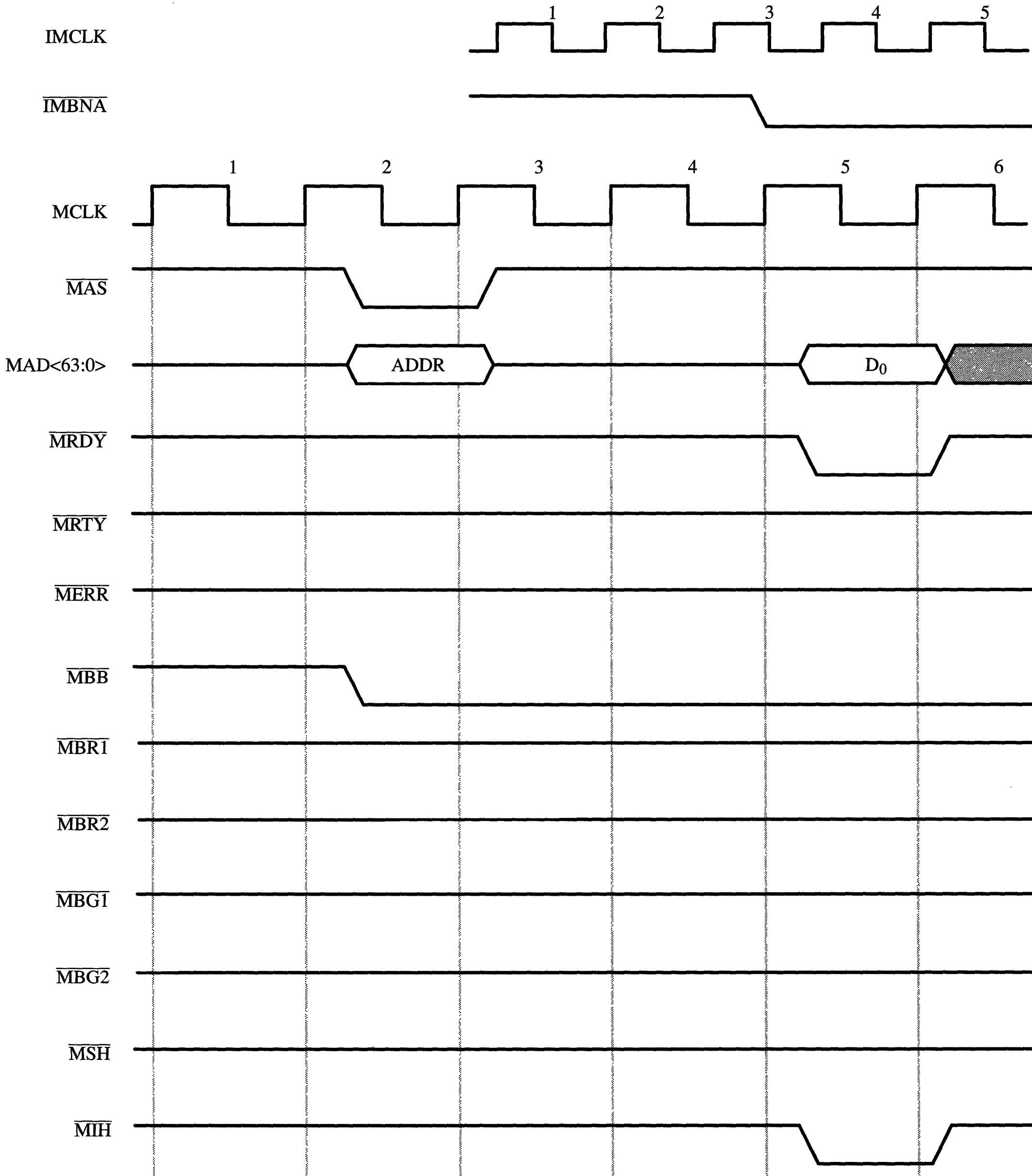


Figure 11–45. MBus Coherent Read and Invalidate—Owned Data (RT625) (Fast Memory)*
(page 1 of 2)

* This timing diagram illustrates a Coherent Read and Invalidate in which the requested data exists in one or more caches in the system and is owned by a cache. Only the owning cache asserts $\overline{\text{MIH}}$ on cycle A+3 and supplies the data. After supplying the data, the owning cache invalidates its copy; all other caches with copies of the data also invalidate their copies.

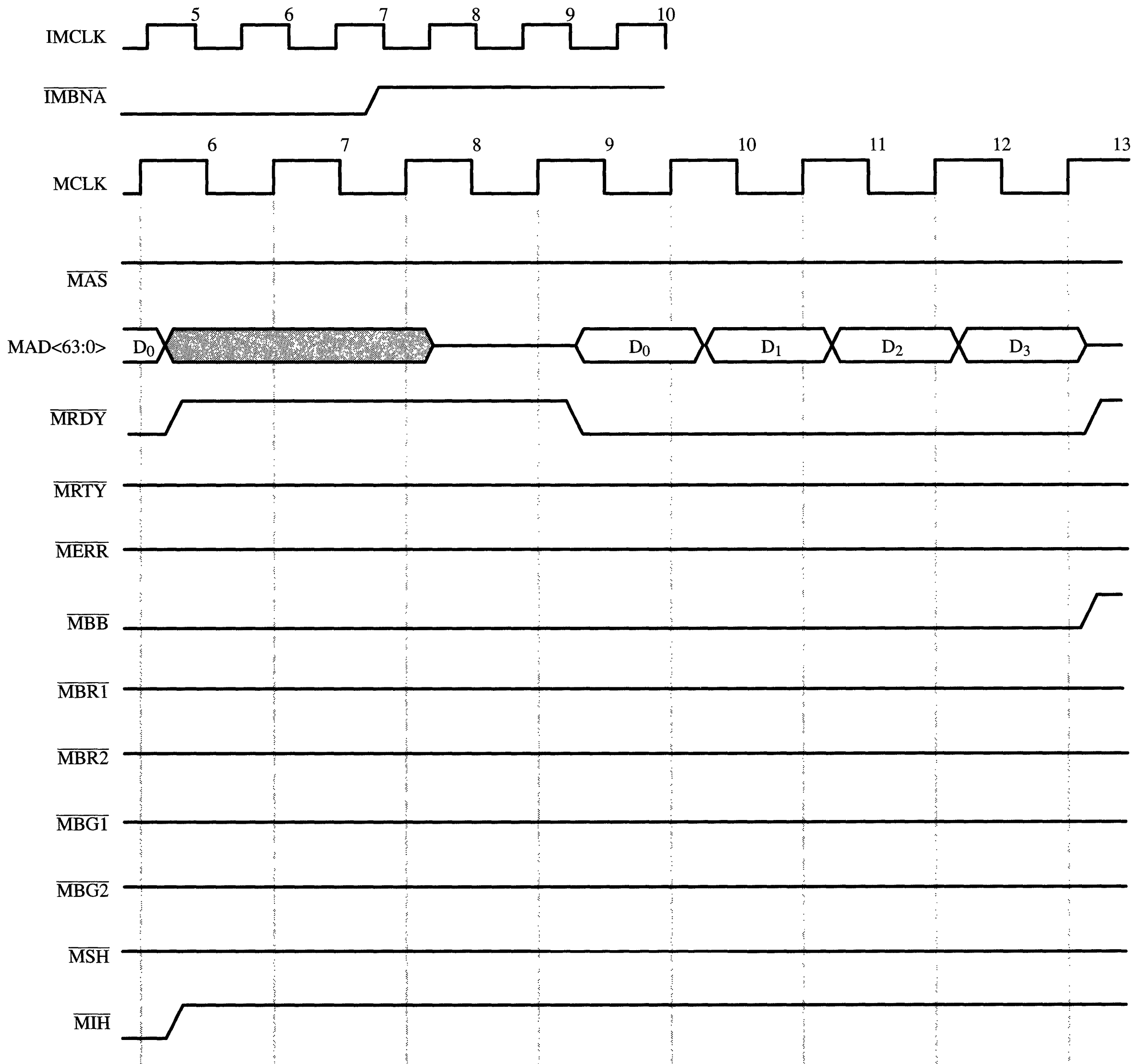


Figure 11–45. MBus Coherent Read and Invalidate—Owned Data (RT625) (Fast Memory) (page 2 of 2)

SPARC Instruction Set

12.1 Assembly Language Syntax

The notations given in this section are taken from Sun's SPARC Assembler and are used to describe the suggested assembly language syntax for the instruction definitions given in *Section 12.2*.

Understanding the use of type fonts is crucial to understanding the assembly language syntax in the instruction definitions. Items in typewriter font are literals, to be entered exactly as they appear. Items in *italic font* are metasymbols which are to be replaced by numeric or symbolic values when actual assembly language code is written. For example, *asi* would be replaced by a number in the range of 0 to 255 (the value of the bits in the binary instruction), or by a symbol which has been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, *regrs2* is a *reg* (i.e., register name) whose binary value will end up in the *rs2* field of the resulting instruction.

12.1.1 Register Names

reg

A *reg* is an integer unit register. It can have a value of:

%0	through	%31	all integer registers
%g0	through	%g7	global registers—same as %0 through %7
%o0	through	%o7	out registers—same as %8 through %15
%l0	through	%l7	local registers—same as %16 through %23
%i0	through	%i7	in registers—same as %24 through %31

Subscripts further identify the placement of the operand in the binary instruction as one of:

reg_{rs1} —*rs1* field
reg_{rs2} —*rs2* field
reg_{rd} —*rd* field

freg

A *freg* is a floating-point register. It can have a value from %f0 through %f31. Subscripts further identify the placement of the operand in the binary instruction as one of:

$freg_{rs1}$ —*rs*1 field
$freg_{rs2}$ —*rs*2 field
$freg_{rd}$ —*rd* field

creg

A *creg* is a coprocessor register. It can have a value from %c0 through %c31. Subscripts further identify the placement of the operand in the binary instruction as one of:

$creg_{rs1}$ —*rs*1 field
$creg_{rs2}$ —*rs*2 field
$creg_{rd}$ —*rd* field

12.1.2 Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in typewriter font, and are preceded by a percent sign (%). The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

%psr	Processor State Register
%wim	Window Invalid Mask register
%tbr	Trap Base Register
%y	Y register
%fsr	Floating-point State Register
%csr	Coprocessor State Register
%fq	Floating-point Queue
%cq	Coprocessor Queue
%hi	Unary operator that extracts high 22 bits of its operand
%lo	Unary operator that extracts low 10 bits of its operand

12.1.3 Values

Some instructions use operands comprising values as follows:

simm13—A signed immediate constant that fits in 13 bits
const22—A constant that fits in 22 bits
asi—An alternate address space identifier (0 to 255)

12.1.4 Label

A label is a sequence of characters comprised of alphabetic letters (a-z, A-Z (upper and lower case distinct)), underscore (_), dollar sign ($), period (.), and decimal digits (0-9), but which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

regaddr:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$

address:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$
$reg_{rs1} + simm13$
$reg_{rs1} - simm13$
$simm13$
$simm13 + reg_{rs1}$

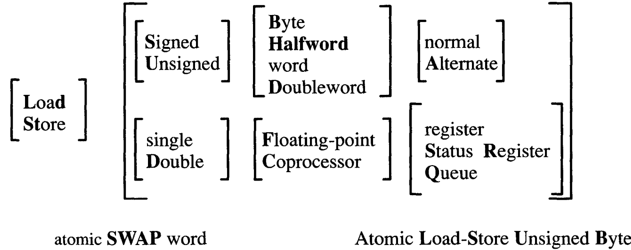
reg_or_imm:

reg_{rs2}
$simm13$

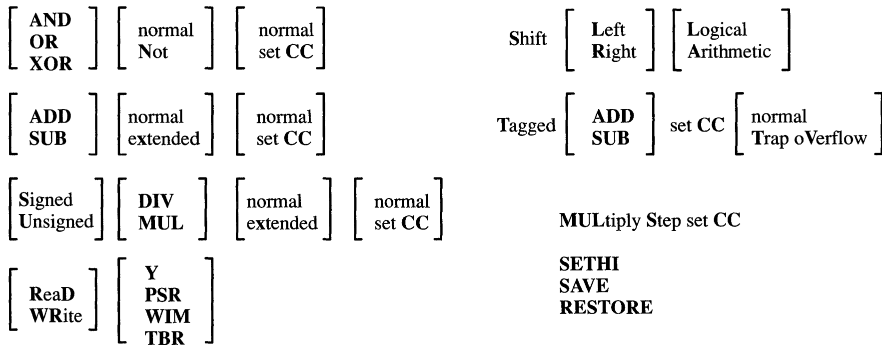
12.1.5 Instruction Mnemonics

Figure 12–1 illustrates the mnemonics used to describe the SPARC instruction set. Note that some combinations possible in *Figure 12–1* do not correspond to valid instructions (such as store signed or floating-point convert extended to extended). Refer to the instruction set summary, *Table 12–2,* for a list of valid SPARC instructions.

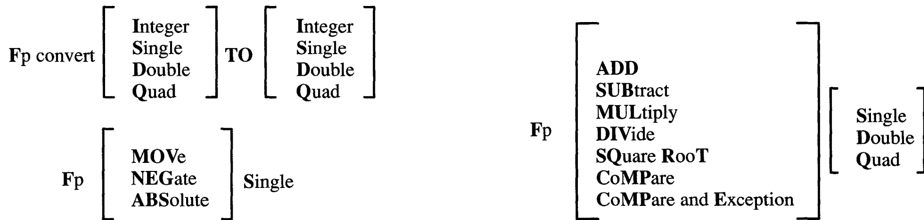
Data Transfer



Integer Operations



Floating-Point Operations



Control Transfer

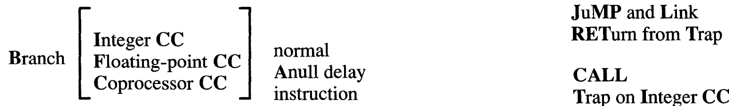


Figure 12–1. SPARC Instruction Mnemonic Summary

12.2 Definitions

This section provides a detailed definition for each CY7C601 instruction. Each definition includes: the instruction operation; suggested assembly language syntax; a description of the salient features, restrictions and trap conditions; a list of synchronous or floating-point/coprocessor traps which can occur as a consequence of executing the instruction; and the instruction format and op codes. Instructions are defined in alphabetical order with the instruction mnemonic shown in large bold type at the top of the page for easy reference. The instruction set summary that precedes the definitions, *Table 12–2,* groups the instructions by type.

Table 12–1 identifies the abbreviations and symbols used in the instruction definitions. An example of how some of the description notations are used is given below in *Figure 12–2.* Register names, labels and other aspects of the syntax used in these instructions are described in the previous section.

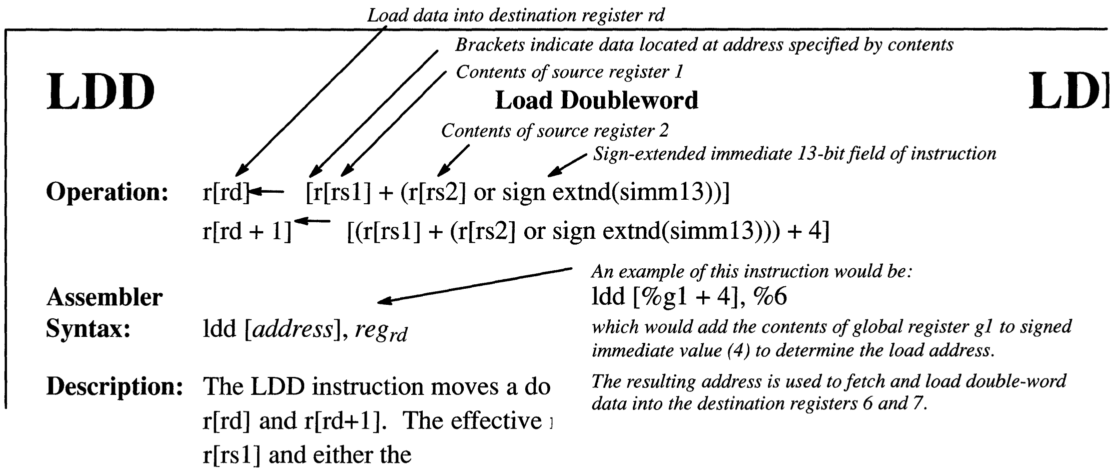


Figure 12–2. Instruction Description

Table 12–1. Instruction Description Notations (part 1 of 2)

Symbol	Description
a	Instruction field that controls instruction annulling during control transfers
AND, OR XOR, etc.	AND, OR, XOR, etc operators
asi	Instruction field that identifies the Load/Store alternate address space
c	The icc carry bit
ccc	The coprocessor condition code field of the CSR
CONCAT	Concatenate
cond	Instruction field that selects the condition code test for branches
CQ.ADDR	The address portion of the Coprocessor Queue
CQ.INSTR	The instruction portion of the Coprocessor Queue
c[rd]	Depending on context, the coprocessor register (or its contents) specified by the instruction field, e.g., rd, rs1, rs2
CSR	Coprocessor State Register
CWP	PSR's Current Window Pointer field
disp22	Instruction field that contains the 22-bit sign-extended displacement for branches
disp30	Instruction field that contains the 30-bit word displacement for calls
dz	Floating-point exception: division by zero

Table 12–1. Instruction Description Notations (part 2 of 2)

Symbol	Description
EC	PSR's Enable Coprocessor bit
EF	PSR's Enable FPU bit
ET	PSR's Enable Traps bit
fcc	The floating-point condition code field of the FSR
FQ.ADDR	The address portion of the Floating-point queue
FQ.INSTR	The instruction portion of the Floating-point queue
f[rd]s	The suffix (s, d, q) after the operand indicates the precision of the operand
f[rs1]	Depending on context, the floating-point register (or its contents) specified by the instruction field, e.g., rd, rs1, rs2
FSR	Floating-point State Register
i	Instruction field that selects rs2 or sign extnd(simm13) as the second operand
icc	The integer condition code field of the PSR
imm22	Instruction field that contains the 22-bit constant used by SETHI
n	The icc negative bit
not	Logical complement operator
nPC	next Program Counter
nv	Floating-point exception:invalid
nx	Floating-point exception:inexact result
of	Floating-point exception:overflow
opc	Instruction field that specifies the count for Coprocessor-operate instructions
operand2	Either r[rs2] or sign extnd(simm13)
PC	Program Counter
pS	PSR's previous Supervisor bit
PSR	Processor State Register
r[15]	A directly addressed register (could be floating-point or coprocessor)
rd	Instruction field that specifies the destination register (except for store)
r[rd]	Depending on context, the integer register (or its contents) specified by the instruction field, e.g. , rd, rs1, rs2
r[rd]<31>	<> are used to specify bit fields of a particular register or I/O signal
[r[rs1] + r[rs2]]	The contents of the address specified by r[rs1] + r[rs2]
rs1	Instruction field that specifies the source 1 register
rs2	Instruction field that specifies the source 2 register
S	PSR's Supervisor bit
shcnt	Instruction field that specifies the count for shift instructions
sign extn(simm13)	Instruction field that contains the 13-bit, sign-extended immediate value
Symbol	Description
TBR	Trap Base Register
tt	TBR's trap type field
uf	Floating-point exception:underflow
v	The icc overflow bit
WIM	Window Invalid Mask register
Y	Y Register
z	The icc zero bit
-	Subtract
x	Multiply
/	Divide
<--	Replaced by
7FFFFFF H	Hexadecimal number representation
+	Add

Table 12–2. Instruction Set Summary (part 1 of 2)

	Name		Operation		CY7C601	RT620 CPI[1]
Load and Store Instructions	LDSB	(LDSBA[4])	Load Signed Byte	(from Alternate Space)	2	1[5]
	LDSH	(LDSHA[4])	Load Signed Halfword	(from Alternate Space)	2	1[5]
	LDUB	(LDUBA[4])	Load Unsigned Byte	(from Alternate Space)	2	1[5]
	LDUH	(LDUHA[4])	Load Unsigned Halfword	(from Alternate Space)	2	1[5]
	LD	(LDA[4])	Load Word	(from Alternate Space)	2	1[5]
	LDD	(LDDA[4])	Load Doubleword	(from Alternate Space)	3	1[5]
	LDF		Load Floating Point		2	1
	LDFF		Load Double Floating-Point		3	1
	LDFSR		Load Floating-Point State Register		2	1
	LDC		Load Coprocessor		2	not supported
	LDDC		Load Double Coprocessor		3	not supported
	LDCSR		Load Coprocessor State Register		2	not supported
	STB	(STBA[4])	Store Byte	(into Alternate Space)	3	2[5]
	STH	(STHA[4])	Store Halfword	(into Alternate Space)	3	2[5]
	ST	(STA[4])	Store Word	(into Alternate Space)	3	2[5]
	STD	(STDA[4])	Store Doubleword	(into Alternate Space)	4	2[5]
	STF		Store Floating-Point		3	2
	STDF		Store Double Floating-Point		4	2
	STFSR		Store Floating-Point State Register		3	2
	STDFQ		Store Double Floating-Point Queue		4	2
	STC		Store Coprocessor		3	not supported
	STDC		Store Double Coprocessor		4	not supported
	STCSR		Store Coprocessor State Register		3	not supported
	STDCQ		Store Double Coprocessor Queue		4	not supported
	LDSTUB	(LDSTUBA[4])	Atomic Load-Store Unsigned Byte	(in Alternate Space)	4	3
	SWAP	(SWAPA[4])	Swap r-register with Memory	(in Alternate Space)	4	3
Arithmetic/Logical/Shift	ADD	(ADDcc)	Add	(and modify icc)	1	1
	ADDX	(ADDXcc)	Add with Carry	(and modify icc)	1	1
	SUB	(SUBcc)	Subtract	(and modify icc)	1	1
	SUBX	(SUBXcc)	Subtract with Carry	(and modify icc)	1	1
	TADDcc	(TADDccTV)	Tagged Add and Modify icc	(and Trap on overflow)	1	1
	TSUBcc	(TSUBccTV)	Tagged Subtract and Modify icc	(and Trap on overflow)	1	1
	SDIV	(SDIV)	Integer Divide	(and modify icc)	not supported	37
	SMUL	(SMULcc)	Integer Multiply	(and modify icc)	not supported	17
	MULScc		Multiply Step and Modify icc		1	1
	UDIV	(UDIVcc)	Unsigned Divide (integer)		not supported	37
	UMUL	(UMULcc)	Unsigned Multiply (integer)		not supported	17
	AND	(ANDcc)	And	(and modify icc)	1	1
	ANDN	(ANDNcc)	And Not	(and modify icc)	1	1
	OR	(ORcc)	Inclusive Or	(and modify icc)	1	1
	ORN	(ORNcc)	Inclusive Or Not	(and modify icc)	1	1

Table 12–2. Instruction Set Summary (part 2 of 2)

	Name		Operation		CY7C601	RT620 CPI[1]
Arithmetic/Logical/Shift	XOR	(XORcc)	Exclusive Or	(and modify icc)	1	1
	XNOR	(XNORcc)	Exclusive Nor	(and modify icc)	1	1
	SLL		Shift Left Logical		1	1
	SRL		Shift Right Logical		1	1
	SRA		Shift Right Arithmetic		1	1
	SETHI		Set High 22 Bits of r-register		1	1
	SAVE		Save Caller's Window		1	1
	RESTORE		Restore Caller's Window		1	1
Control Transfer	Bicc		Branch on Integer Condition Codes		1[6]	1[6]
	FBicc		Branch on Floating-Point Condition Codes		1[6]	1[6]
	CBccc		Branch on Coprocessor Condition Codes		1[6]	1[6]
	CALL		Call		1[6]	1[6]
	JMPL		Jump and Link		2[6]	1[6]
	RETT		Return from Trap		2[6]	1[6]
	Ticc		Trap on Integer Condition Codes		1 (4 if taken)	1
Read/Write Control Registers	RDASR[2,4]		Read Ancillary State Register		1	1
	RDY		Read Y Register		1	1
	RDPSR[4]		Read Processor State Register		1	1
	RDWIM[4]		Read Window Invalid Mask		1	1
	RDTBR[4]		Read Trap Base Register		1	1
	WRASR[2,4]		Write Ancillary State Register		1	1
	WRY		Write Y Register		1	1
	WRPSR[4]		Write Processor State Register		1	1
	WRWIM[4]		Write Window Invalid Mask		1	1
	WRTBR[4]		Write Trap Base Register		1	1
	UNIMP		Unimplemented Instruction		1	1
	FLUSH[7]		Instruction Cache Flush		1	4
FP (CP) Ops	FPop		Floating-Point Unit Operations		1 to launch	see note 3
	CPop		Coprocessor Operations		1 to launch	not supported

Notes:
1. All RT620 CPI assume worst case of single instruction launch.
2. Not a privileged instruction when accessing the Y register.
3. Refer to Table 3–6 on page 3-77.
4. Privileged instruction
5. Alternate ASI = 0xc, 0xd, 0x10 – 0x18, or 0x31 have varying CPI (all are true ICACHE instructions)
6. Assuming delay slot is filled with a useful instruction
7. This instruction is referred to as FLUSH in the SPARC Version 8 ISA and IFLUSH in the SPARC Version 7 ISA.

ADD — Add — ADD

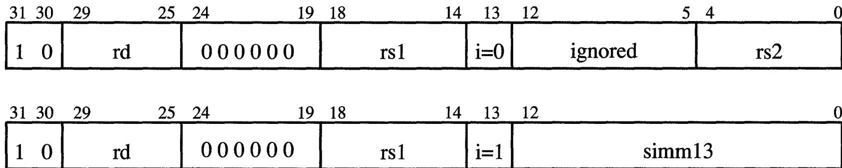
Operation: r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: add reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: The ADD instruction adds the contents of the register named in the *rs1* field ,r[rs1], to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The result is placed in the register specified in the *rd* field.

Traps: none

Format:



ADDcc — Add and modify icc — ADDcc

Operation: r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
  OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)

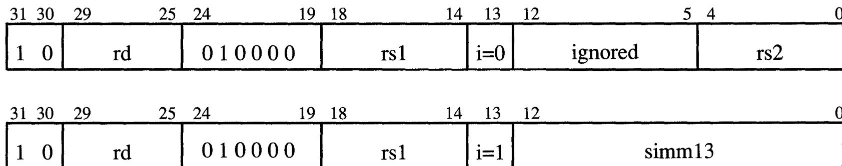
c ← (r[rs1]<31> AND operand2<31>)
  OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax: addcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: ADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition, ADDcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:



ADDX Add with Carry ADDX

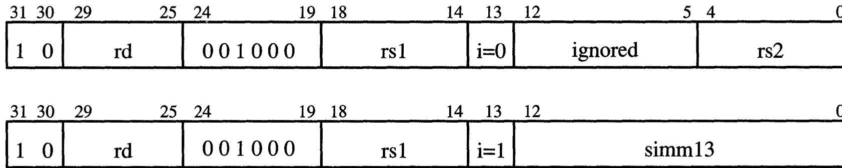
Operation: r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13)) + c

Assembler Syntax: addx reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: ADDX adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:



ADDXcc — Add with Carry and modify icc — ADDXcc

Operation: r[rd] ← r[rs1] + operand2 + c, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)

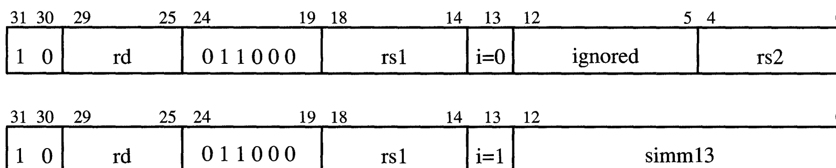
c ← (r[rs1]<31> AND operand2<31>)
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax: addxcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: ADDXcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field. ADDXcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



AND And AND

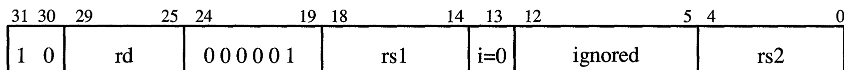
Operation: r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))

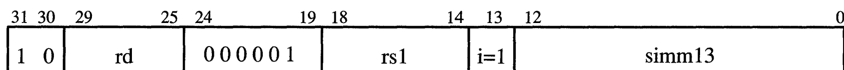
Assembler Syntax: and reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ANDcc — And and modify icc — ANDcc

Operation: r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← 0

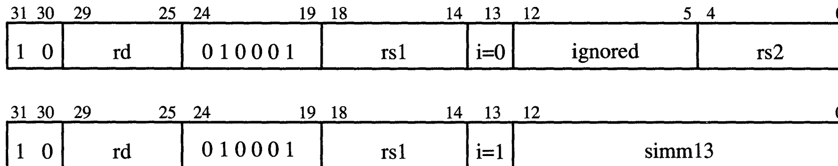
c ← 0

Assembler Syntax: andcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ANDcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



ANDN — And Not — ANDN

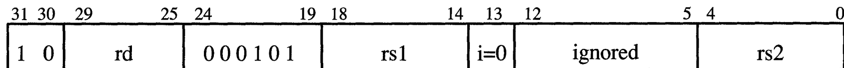
Operation: r[rd] ← r[rs1] AND $\overline{\text{(r[rs2] or sign extnd(simm13))}}$

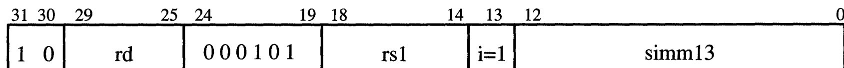
Assembler Syntax: andn reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: ANDN does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ANDNcc — And Not and modify icc — ANDNcc

Operation: r[rd] ← r[rs1] AND $\overline{\text{(r[rs2] or sign extnd(simm13))}}$

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← 0

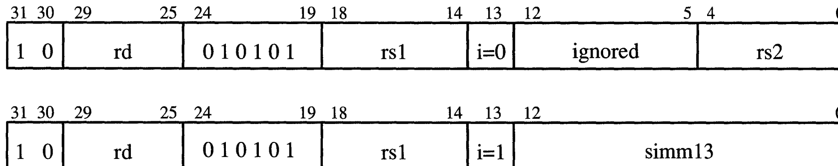
c ← 0

Assembler Syntax: andncc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: ANDNcc does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ANDNcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



Bicc — Integer Conditional Branch — Bicc

Operation: PC ← nPC

If condition true then nPC ← PC + (sign extnd(disp22) x 4)

else nPC ← nPC + 4

Assembler Syntax:

ba{,a}	*label*	
bn{,a}	*label*	
bne{,a}	*label*	synonym: bnz
be{,a}	*label*	synonym: bz
bg{,a}	*label*	
ble{,a}	*label*	
bge{,a}	*label*	
bl{,a}	*label*	
bgu{,a}	*label*	
bleu{,a}	*label*	
bcc{,a}	*label*	synonym: bgeu
bcs{,a}	*label*	synonym: blu
bpos{,a}	*label*	
bneg{,a}	*label*	
bvc{,a}	*label*	
bvs{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The Bicc instructions (except for BA and BN) evaluate specific integer condition code combinations (from the PSR's *icc* field) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address PC + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. Refer to *Section 2.4.3.3* for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See *Section 2.4.3.4* regarding delay-branch instructions.

Branch never (BN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch always (BA), because it always branches regardless of the condition codes, would

normally ignore the annul field. Instead, it follows the same annul field rules: if *a*=1, the delay instruction is annulled; if *a*=0, the delay instruction is executed.

The delay instruction following a Bicc (other than BA) should not be a delayed-control-transfer instruction. The results of following a Bicc with another delayed control transfer instruction are implementation-dependent and therefore unpredictable.

Traps: none

Mnemonic	Cond.	Operation	icc Test
BN	0000	Branch Never	No test
BE	0001	Branch on Equal	z
BLE	0010	Branch on Less or Equal	z OR (n XOR v)
BL	0011	Branch on Less	n XOR v
BLEU	0100	Branch on Less or Equal, Unsigned	c OR z
BCS	0101	Branch on Carry Set (Less than, Unsigned)	c
BNEG	0110	Branch on Negative	n
BVS	0111	Branch on oVerflow Set	v
BA	1000	Branch Always	No test
BNE	1001	Branch on Not Equal	not z
BG	1010	Branch on Greater	not(z OR (n XOR v))
BGE	1011	Branch on Greater or Equal	not(n XOR v)
BGU	1100	Branch on Greater, Unsigned	not(c OR z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not c
BPOS	1110	Branch on Positive	not n
BVC	1111	Branch on oVerflow Clear	not v

Format:

31 30	29	28 25	24 22	21 0
0 0	a	cond.	0 1 0	disp22

CALL   Call   CALL

Operation: r[15] ← PC
PC ← nPC
nPC ← PC + (disp30 x 4)

Assembler Syntax: call *label*

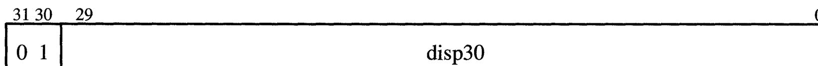
Description: The CALL instruction causes a delayed, unconditional, PC-relative control transfer to the address PC + (disp30 x 4). The CALL instruction does not have an annul bit, therefore the delay slot instruction following the CALL instruction is always executed (See *Section 2.4.3.4*). CALL first writes its return address (PC) into the *outs* register, r[15], and then adds 4 to the PC. The 32-bit displacement which is added to the new PC is formed by appending two low-order zeros to the 30-bit word displacement contained in the instruction. Consequently, the target address can be anywhere in the CY7C601's user or supervisor address space.

If the instruction following a CALL uses register r[15] as a source operand, hardware interlocks add a one cycle delay.

Programming note: a register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15.

Traps: none

Format:



CBccc — Coprocessor Conditional Branch — CBccc

Operation: PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

cba{,a} *label*
cbn{,a} *label*
cb3{,a} *label*
cb2{,a} *label*
cb23{,a} *label*
cb1{,a} *label*
cb13{,a} *label*
cb12{,a} *label*
cb123{,a} *label*
cb0{,a} *label*
cb03{,a} *label*
cb02{,a} *label*
cb023{,a} *label*
cb01{,a} *label*
cb013{,a} *label*
cb012{,a} *label*

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The CBccc instructions (except for CBA and CBN) evaluate specific coprocessor condition code combinations (from the CCC<1:0> inputs) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See *Section 2.4.3.3* regarding control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See *Section 2.4.3.4* regarding delayed branching.

Branch never (CBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch always (CBA), because it always branches regardless of the condition codes, would

normally ignore the annul field. Instead, it follows the same annul field rules: if *a*=1, the delay instruction is annulled; if *a*=0, the delay instruction is executed.

To prevent misapplication of the condition codes, a non-coprocessor instruction must immediately precede a CBccc instruction.

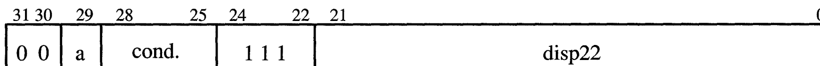
A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Traps: cp_disabled
cp_exception

Mnemonic	cond.	CCC<1:0> test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Format:



CPop — Coprocessor Operate — CPop

Operation: Dependent on coprocessor implementation

Assembler Syntax: Unspecified

Description: CPop1 and CPop2 are the instruction formats for coprocessor operate instructions. The *op3* field for CPop1 is 110110; for CPop2 it is 110111. The coprocessor operations themselves are encoded in the *opc* field and are dependent on the coprocessor implementation. Note that this does not include Load/Store coprocessor instructions, which fall into the integer unit's Load/Store instruction category.

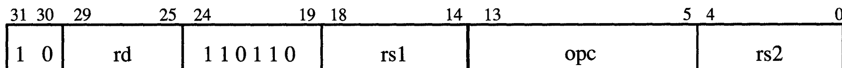
All CPop instructions take all operands from, and return all results to, the coprocessor's registers. The data types supported, how the operands are aligned, and whether a CPop generates a cp_exception trap are coprocessor dependent.

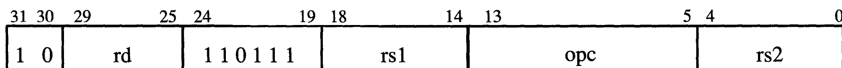
A CPop instruction causes a cp_disabled trap if the PSR's EC bit is reset or if no coprocessor is present.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Traps: cp_disabled
cp_exception

Format:





FABSs — Absolute Value Single — FABSs

Operation: f[rd]s ← f[rs2]s AND 7FFFFFFF H

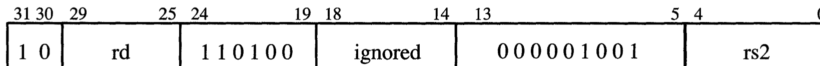
Assembler Syntax: fabss $freg_{rs2}$, $freg_{rd}$

Description: The FABSs instruction clears the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since rs2 can be either an even or odd register, FABSs can also operate on the high-order words of double and extended operands, which accomplishes sign bit clear for these data types.

Traps: fp_disabled
fp_exception*

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 1 0 0 1	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FADDd — Add Double — FADDd

Operation: f[rd]d ← f[rs1]d + f[rs2]d

Assembler Syntax: faddd $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FADDd instruction adds the contents of f[rs1] CONCAT f[rs1+1] to the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 0	rs2

FADDq Add Quad* FADDq

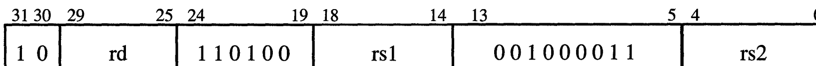
Operation: f[rd]q ← f[rs1]q + f[rs2]q

Assembler Syntax: faddq $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FADDq instruction adds the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] CONCAT f[rs1+3] to the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FADDs — Add Single — FADDs

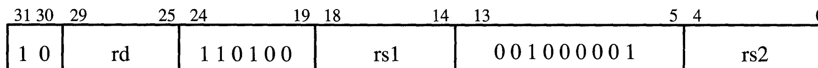
Operation: f[rd]s ← f[rs1]s + f[rs2]s

Assembler Syntax: fadds $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FADDs instruction adds the contents of f[rs1] to the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	110100	rs1	001000001	rs2

FBfcc — Floating-Point Conditional Branch — FBfcc

Operation: PC ← nPC

If condition true then nPC ← PC + (sign extnd(disp22) x 4)

else nPC ← nPC + 4

Assembler Syntax:

fba{,a}	*label*	
fbn{,a}	*label*	
fbu{,a}	*label*	
fbg{,a}	*label*	
fbug{,a}	*label*	
fbl{,a}	*label*	
fbul{,a}	*label*	
fblg{,a}	*label*	
fbne{,a}	*label*	synonym: fbnz
fbe{,a}	*label*	synonym: fbz
fbue{,a}	*label*	
fbge{,a}	*label*	
fbuge{,a}	*label*	
fble{,a}	*label*	
fbule{,a}	*label*	
fbo{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The FBfcc instructions (except for FBA and FBN) evaluate specific floating-point condition code combinations (from the FCC<1:0> inputs) based on the branch type, as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See *Section 2.4.3.3* for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See *Section 2.4.3.4* regarding delayed branch instructions.

Branch never (FBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch always (FBA), because it always branches regardless of the condition codes, would

normally ignore the annul field. Instead, it follows the same annul field rules: if *a*=1, the delay instruction is annulled; if *a*=0, the delay instruction is executed.

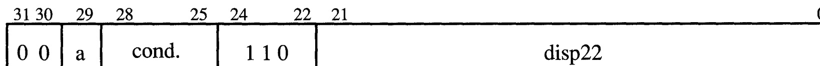
To prevent misapplication of the condition codes, a non-floating-point instruction must immediately precede an FBfcc instruction.

An FBfcc instruction generates an fp_disabled trap (and does not branch or annul) if the PSR's EF bit is reset or if no floating-point unit is present.

Traps: fp_disabled
fp_exception*

Mnemonic	Cond.	Operation	fcc Test
FBN	0000	Branch Never	no test
FBNE	0001	Branch on Not Equal	U or L or G
FBLG	0010	Branch on Less or Greater	L or G
FBUL	0011	Branch on Unordered or Less	U or L
FBL	0100	Branch on Less	L
FBUG	0101	Branch on Unordered or Greater	U or G
FBG	0110	Branch on Greater	G
FBU	0111	Branch on Unordered	U
FBA	1000	Branch Always	no test
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	U or E
FBGE	1011	Branch on Greater or Equal	G or E
FBUGE	1100	Branch on Unordered or Greater or Equal	U or G or E
FBLE	1101	Branch on Less or Equal	L or E
FBULE	1110	Branch on Unordered or Less or Equal	U or L or E
FBO	1111	Branch on Ordered	L or G or E

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FCMPd

Compare Double

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmpd $freg_{rs1}$, $freg_{rs2}$

Description: FCMPd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

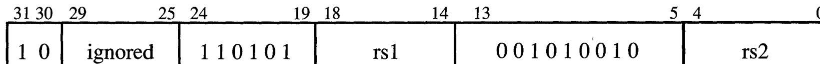
In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPd causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPEd Compare Double and Exception if Unordered FCMPEd

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmped $freg_{rs1}$, $freg_{rs2}$

Description: FCMPEd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

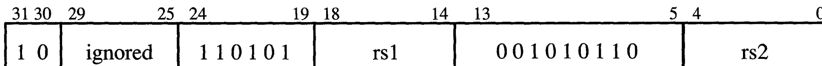
In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEd causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPEq — Compare Quad and Exception if Unordered* — FCMPEq

Operation: fcc ← f[rs1]q COMPARE f[rs2]q

Assembler Syntax: fcmpeq $freg_{rs1}$, $freg_{rs2}$

Description: FCMPEq subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] CONCAT f[rs1+3] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

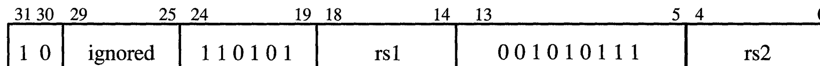
In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2], f[rs1+3] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2], f[rs2+3] .

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEq causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FCMPEs — Compare Single and Exception if Unordered — FCMPEs

Operation: fcc ← f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmpes $freg_{rs1}$, $freg_{rs2}$

Description: FCMPEs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

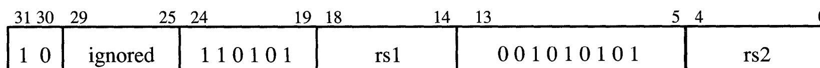
In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEs causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPq — Compare Quad* — FCMPq

Operation: fcc ← f[rs1]q COMPARE f[rs2]q

Assembler Syntax: fcmpq $freg_{rs1}$, $freg_{rs2}$

Description: FCMPq subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

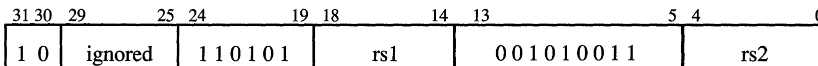
In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2], f[rs1+3] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2], f[rs2+3].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPq causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FCMPs — Compare Single — FCMPs

Operation: fcc ← f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmps $freg_{rs1}$, $freg_{rs2}$

Description: FCMPs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

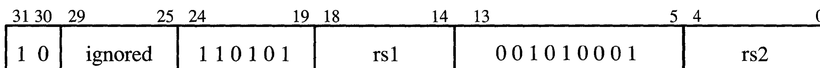
In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPs causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FDIVd — Divide Double — FDIVd

Operation: f[rd]d ← f[rs1]d / f[rs2]d

Assembler Syntax: fdivd $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FDIVd instruction divides the contents of f[rs1] CONCAT f[rs1+1] by the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:

31 30	29 – 25	24 – 19	18 – 14	13 – 5	4 – 0
1 0	rd	110100	rs1	001001110	rs2

FDIVq — Divide Quad* — FDIVq

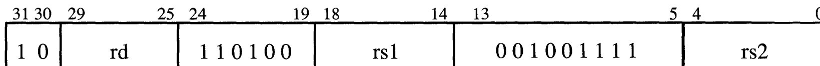
Operation: f[rd]q ← f[rs1]q / f[rs2]q

Assembler Syntax: fdivq $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FDIVq instruction divides the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] CONCAT f[rs1+3] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 1	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FDIVs Divide Single FDIVs

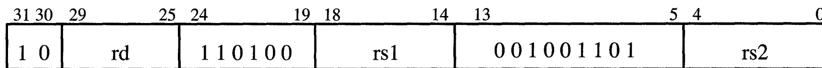
Operation: f[rd]s ← f[rs1]s / f[rs2]s

Assembler Syntax: fdivs $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FDIVs instruction divides the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



FdMULq Multiply Double to Quad* FdMULq

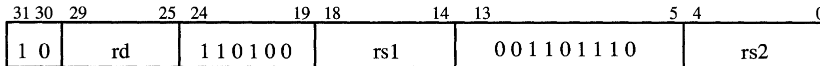
Operation: f[rd]q ← f[rs1]d x f[rs2]d

Assembler Syntax: fdmulq $freg_{rs1}, freg_{rs2}, freg_{rd}$

Description: The FdMULq instruction multiplies the double precision contents of f[rs1] CONCAT f[rs1+1] by the double precision contents of f[rs2] CONCAT f[rs2+1]. The result is of quad precision, and is placed in f[rd], f[rd+1] f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception (nv)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 1 0 1 1 1 0	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FdTOi — Convert Double to Integer — FdTOi

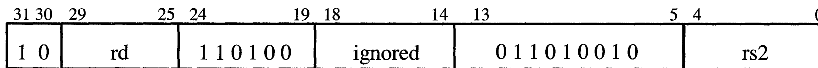
Operation: f[rd]i ← f[rs2]d

Assembler Syntax: fdtoi $freg_{rs2}$, $freg_{rd}$

Description: FdTOi converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding direction field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 0	rs2

FdTOq — Convert Double to Quad* — FdTOq

Operation: f[rd]q ← f[rs2]d

Assembler Syntax: fdtoq $freg_{rs2}$, $freg_{rd}$

Description: FdTOq converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a quad-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], f[rd+2], and f[rd+3]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 1 0	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FdTOs — Convert Double to Single — FdTOs

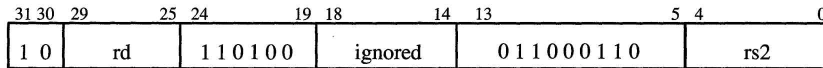
Operation: f[rd]s ← f[rs2]d

Assembler Syntax: fdtos $freg_{rs2}$, $freg_{rd}$

Description: FdTOs converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FiTOd — Convert Integer to Double — FiTOd

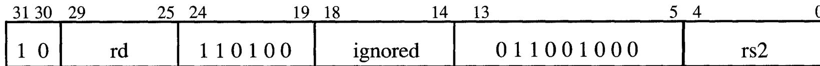
Operation: f[rd]d ← f[rs2]i

Assembler Syntax: fitod $freg_{rs2}$, $freg_{rd}$

Description: FiTOd converts the 32-bit, signed integer contents of f[rs2] to a floating-point, double-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception*

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 0	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FiTOq **Convert Integer to Quad*** FiTOq

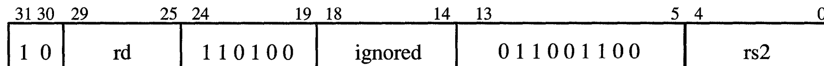
Operation: f[rd]q ← f[rs2]i

Assembler Syntax: fitoq $freg_{rs2}$, $freg_{rd}$

Description: FiTOq converts the 32-bit, signed integer contents of f[rs2] to a quad-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FiTOs Convert Integer to Single FiTOs

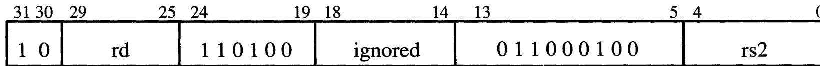
Operation: f[rd]s ← f[rs2]i

Assembler Syntax: fitos $freg_{rs2}$, $freg_{rd}$

Description: FiTOs converts the 32-bit, signed integer contents of f[rs2] to a floating-point, single-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field, *RD*.

Traps: fp_disabled
fp_exception (nx)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 0 0	rs2

FLUSH — Flush Instruction Memory — FLUSH

Operation: FLUSH ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: flush *address*

Description: The FLUSH instruction causes a word to be flushed from an instruction cache which may be internal to the processor. The word to be flushed is at the address specified by the contents of r[rs1] plus either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

Effect on hyperSPARC

FLUSH instructions may be used to flush individual instruction packets within the instruction cache (ICACHE). In order to flush an ICACHE line, all valid bits in the line must be cleared (refer to *Section 3.6*). Flushing an ICACHE line by flushing individual instruction packets requires a sequence of four flush instructions, each of which corresponds to an instruction packet in a particular ICACHE line. Refer to *Section 3.6.4.3* for additional information on FLUSH instruction actions.

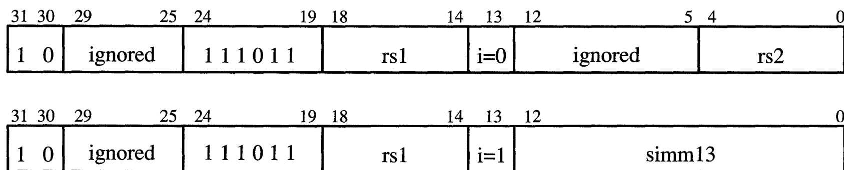
Effect on CY7C601

Since there is no internal instruction cache in the CY7C600 family, the result of executing an FLUSH instruction is dependent on the state of the input signal, instruction cache flush trap ($\overline{IFT}$). If $\overline{IFT}$ = 1, FLUSH executes as a NOP, with no side effects. If $\overline{IFT}$ = 0, execution of FLUSH causes an illegal_instruction trap.

Note that although the opcode is the same, the mnemonic for this instruction is FLUSH in the SPARC Version 8 ISA, and FLUSH in the SPARC Version 7 ISA.

Traps: illegal_instruction

Format:



FMOVs — Move — FMOVs

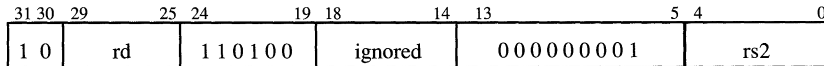
Operation: f[rd]s ← f[rs2]s

Assembler Syntax: fmovs $freg_{rs2}$, $freg_{rd}$

Description: The FMOVs instruction moves the word content of register f[rs2] to the register f[rd]. Multiple FMOVs's are required to transfer multiple-precision numbers between *f-registers*.

Traps: fp_disabled
fp_exception*

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	110100	ignored	000000001	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FMULd — Multiply Double — FMULd

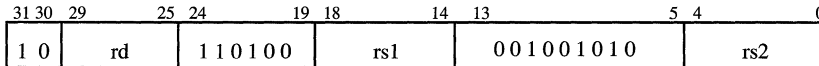
Operation: f[rd]d ← f[rs1]d x f[rs2]d

Assembler Syntax: fmuld $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FMULd instruction multiplies the contents of f[rs1] CONCAT f[rs1+1] by the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FMULq Multiply Quad* FMULq

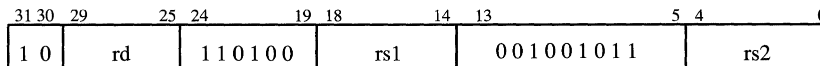
Operation: f[rd]q ← f[rs1]q x f[rs2]q

Assembler Syntax: fmulq *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FMULq instruction multiplies the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] CONCAT f[rs1+3] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FMULs Multiply Single FMULs

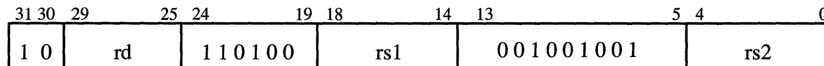
Operation: f[rd]s ← f[rs1]s x f[rs2]s

Assembler Syntax: fmuls $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FMULs instruction multiplies the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FNEGs — Negate — FNEGs

Operation: f[rd]s ← f[rs2]s XOR 80000000 H

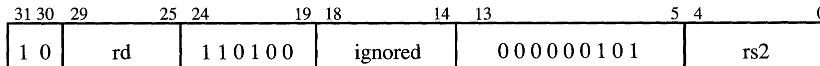
Assembler Syntax: fnegs $freg_{rs2}$, $freg_{rd}$

Description: The FNEGs instruction complements the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since this FPop can address both even and odd *f-register*s, FNEGs can also operate on the high-order words of double and extended operands, which accomplishes sign bit negation for these data types.

Traps: fp_disabled
fp_exception*

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FqTOd — Convert Quad to Double* — FqTOd

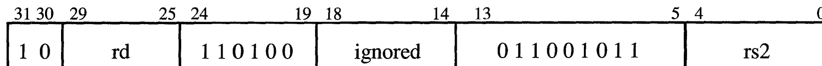
Operation: f[rd]d ← f[rs2]q

Assembler Syntax: fqtod *freg*$_{rs2}$, *freg*$_{rd}$

Description: FqTOd converts the floating-point quad contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FqTOi — Convert Quad to Integer* — FqTOi

Operation: f[rd]i ← f[rs2]q

Assembler Syntax: fqtoi $freg_{rs2}$, $freg_{rd}$

Description: FqTOi converts the floating-point quad contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 – 25	24 – 19	18 – 14	13 – 5	4 – 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 1	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FqTOs — Convert Quad to Single* — FqTOs

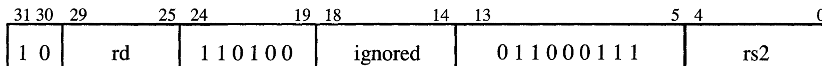
Operation: f[rd]s ← f[rs2]q

Assembler Syntax: fqtos $freg_{rs2}$, $freg_{rd}$

Description: FqTOs converts the floating-point quad contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FsMULd Multiply Single to Double FsMULd

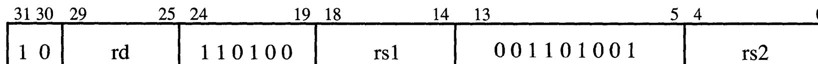
Operation: f[rd]d ← f[rs1]s x f[rs2]s

Assembler Syntax: fsmuld $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FsMULd instruction multiplies the single precision contents of f[rs1] by the single precision contents of f[rs2]. The result is of double precision, and is placed in f[rd] CONCAT f[rd+1].

Traps: fp_disabled
fp_exception (nv)

Format:



31 30	29–25	24–19	18–14	13–5	4–0
1 0	rd	110100	rs1	001101001	rs2

FSQRTd — Square Root Double — FSQRTd

Operation: f[rd]d ← SQRT f[rs2]d

Assembler Syntax: fsqrtd $freg_{rs2}$, $freg_{rd}$

Description: FSQRTd generates the square root of the floating-point double contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	110100	ignored	000101010	rs2

FSQRTq

Square Root Quad*

FSQRTq

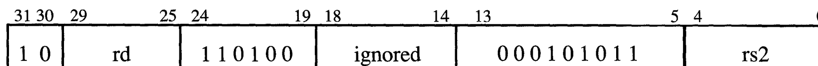
Operation: f[rd]q ← SQRT f[rs2]q

Assembler Syntax: fsqrtq $freg_{rs2}$, $freg_{rd}$

Description: FSQRTq generates the square root of the floating-point quad contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] f[rs2+3] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], f[rd+2], and f[rd+3]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FSQRTs — Square Root Single — FSQRTs

Operation: f[rd]s ← SQRT f[rs2]s

Assembler Syntax: fsqrts $freg_{rs2}$, $freg_{rd}$

Description: FSQRTs generates the square root of the floating-point single contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 … 25	24 … 19	18 … 14	13 … 5	4 … 0
1 0	rd	110100	ignored	000101001	rs2

FsTOd — Convert Single to Double — FsTOd

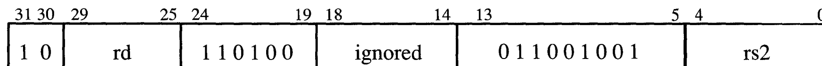
Operation: f[rd]d ← f[rs2]s

Assembler Syntax: fstod $freg_{rs2}$, $freg_{rd}$

Description: FsTOd converts the floating-point single contents of f[rs2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:



31 30	29 – 25	24 – 19	18 – 14	13 – 5	4 – 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 1	rs2

FsTOi — Convert Single to Integer — FsTOi

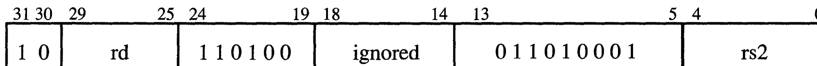
Operation: f[rd]i ← f[rs2]s

Assembler Syntax: fstoi $freg_{rs2}$, $freg_{rd}$

Description: FsTOi converts the floating-point single contents of f[rs2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



FsTOq Convert Single to Quad* FsTOq

Operation: f[rd]q ← f[rs2]q

Assembler Syntax: fstoq *freg*$_{rs2}$, *freg*$_{rd}$

Description: FsTOq converts the floating-point single contents of f[rs2] to a quad-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], f[rd+2], and f[rd+3]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 0 1	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FSUBd — Subtract Double — FSUBd

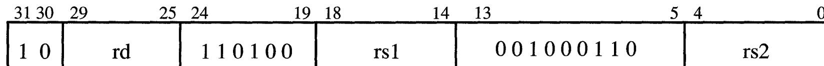
Operation: f[rd]d ← f[rs1]d – f[rs2]d

Assembler Syntax: fsubd $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FSUBd instruction subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:



FSUBq

Subtract Quad*

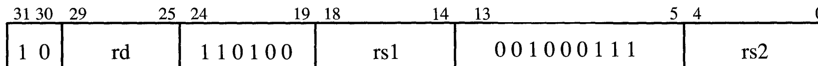
Operation: f[rd]q ← f[rs1]q – f[rs2]q

Assembler Syntax: fsubq $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FSUBq instruction subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] CONCAT f[rs2+3] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] CONCAT f[rs1+3] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], f[rd+2], and f[rd+3].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	110100	rs1	001000111	rs2

* NOTE: Quad-precision operations are not directly supported; an "Unimplmented FPop" trap is generated if a quad-precision operation is attempted. This operation must be emulated in software.

FSUBs — Subtract Single — FSUBs

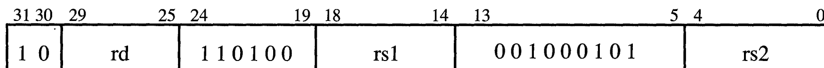
Operation: f[rd]s ← f[rs1]s − f[rs2]s

Assembler Syntax: fsubs $freg_{rs1}$, $freg_{rs2}$, $freg_{rd}$

Description: The FSUBs instruction subtracts the contents of f[rs2] from the contents of f[rs1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:



JMPL — Jump and Link — JMPL

Operation: r[rd] ← PC
PC ← nPC
nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: jmpl *address*, reg_{rd}

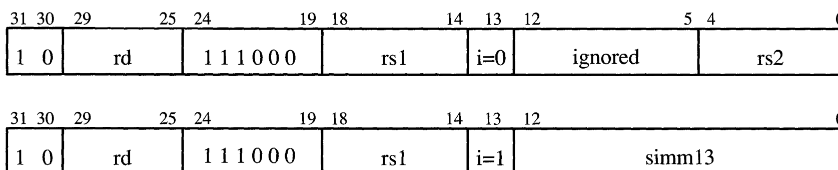
Description: JMPL first provides linkage by saving its return address into the register specified in the *rd* field. It then causes a register-indirect, delayed control transfer to an address specified by the sum of the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If either of the low-order two bits of the jump address is nonzero, a memory_address_not_aligned trap is generated.

Programming note: A register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15. JMPL can also be used to return from a CALL. In this case, *rd* is set to 0 and the return (jump) address would be equal to r[31] + 8.

Traps: memory_address_not_aligned

Format:



LD Load Word LD

Operation: r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], reg_{rd}

Description: The LD instruction moves a word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

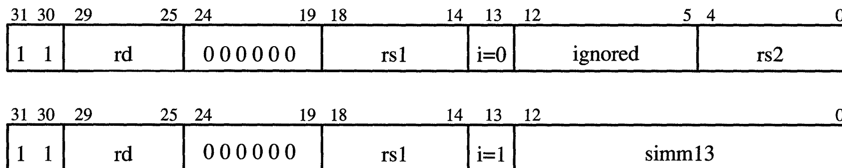
If LD takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



LDA — Load Word from Alternate space — LDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]

Assembler Syntax: lda [*regaddr*] *asi*, reg_{rd}

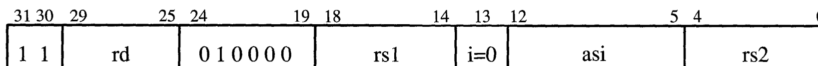
Description: The LDA instruction moves a word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If LDA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 0 0 0	rs1	i=0	asi	rs2

LDC Load Coprocessor register LDC

Operation: c[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], $creg_{rd}$

Description: The LDC instruction moves a word from memory into a coprocessor register, c[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDC takes a trap, the state of the coprocessor depends on the particular implementation.

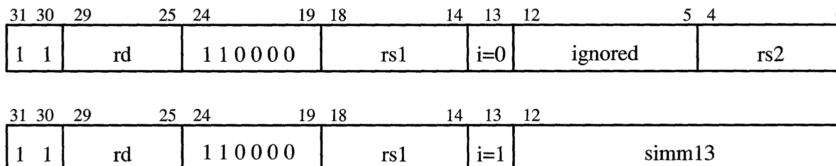
If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



LDCSR Load Coprocessor State Register LDCSR

Operation: CSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %csr

Description: The LDCSR instruction moves a word from memory into the coprocessor state register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDCSR takes a trap, the state of the coprocessor depends on the particular implementation.

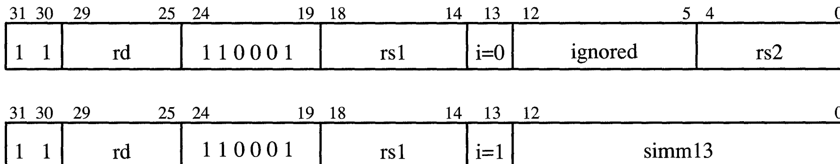
If the instruction following a LDCSR uses the CSR as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon implementation of the coprocessor.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



LDD — Load Double-word — LDD

Operation: r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
r[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], reg_{rd}

Description: The LDD instruction moves a double-word from memory into a destination register pair, r[rd] and r[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in *Section 2.3.1*).

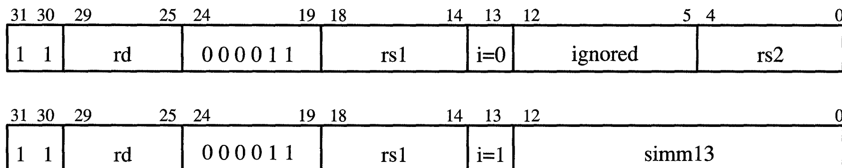
If a data_access_exception trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDD, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



LDDA Load Double-word from Alternate space LDDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]
r[rd +1] ← [r[rs1] + r[rs2] + 4]

Assembler Syntax: ldda [*regaddr*] *asi*, reg_{rd}

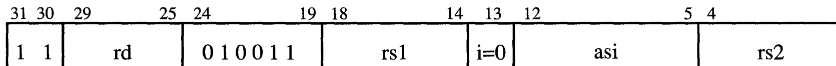
Description: The LDDA instruction moves a double-word from memory into the destination registers, r[rd] and r[rd+1]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in *Section 2.3.1*).

If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDA, this applies to both destination registers.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



LDDC Load Double-word Coprocessor LDDC

Operation: c[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
c[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], $creg_{rd}$

Description: The LDDC instruction moves a double-word from memory into the coprocessor registers, c[rd] and c[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in *Section 2.3.1*).

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDDC takes a trap, the state of the coprocessor depends on the particular implementation.

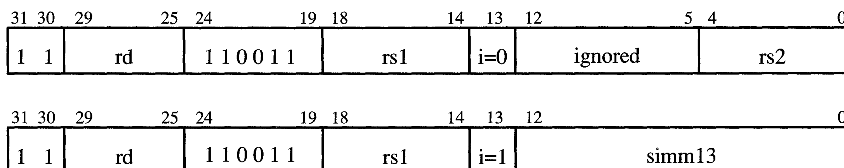
If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem and coprocessor implementation. For an LDDC, this applies to both destination registers.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



LDDF — Load Double-word Floating-Point — LDDF

Operation: f[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
f[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], $freg_{rd}$

Description: The LDDF instruction moves a double-word from memory into the floating-point registers, f[rd] and f[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in *Section 2.3.1*).

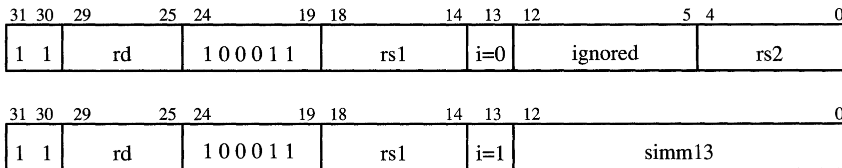
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDF, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDF Load Floating-Point register LDF

Operation: f[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *freg*$_{rd}$

Description: The LDF instruction moves a word from memory into a floating-point register, f[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

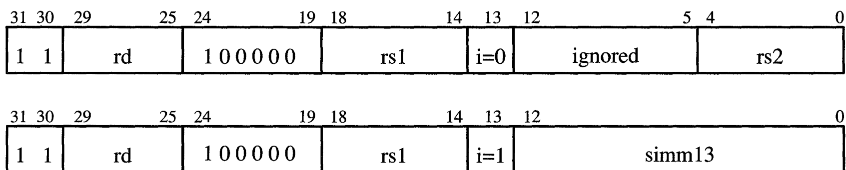
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If LDF takes a trap, the contents of the destination register remain unchanged.

If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDFSR — Load Floating-Point State Register — LDFSR

Operation: FSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %fsr

Description: The LDFSR instruction moves a word from memory into the floating-point state register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it loads the memory word into the FSR.

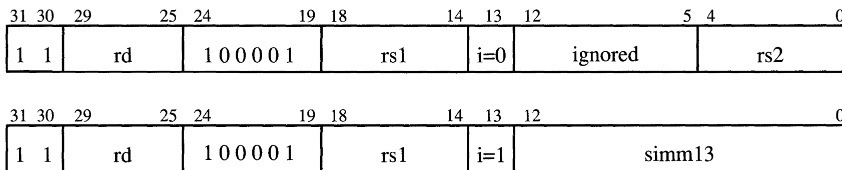
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If LDFSR takes a trap, the contents of the FSR remain unchanged.

If the instruction following a LDFSR uses the FSR as a source operand, hardware interlocks add one or more cycle delay to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDSB — Load Signed Byte — LDSB

Operation: r[rd] ← sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsb [*address*], reg_{rd}

Description: The LDSB instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and sign-extended in r[rd].

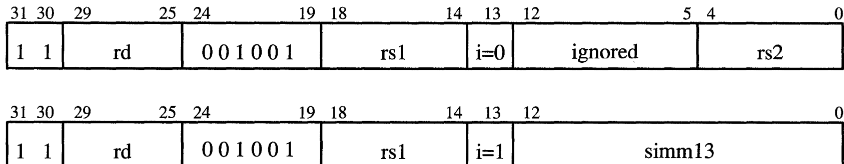
If LDSB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:



LDSBA — Load Signed Byte from Alternate space — LDSBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsba [*regaddr*] *asi*, reg_{rd}

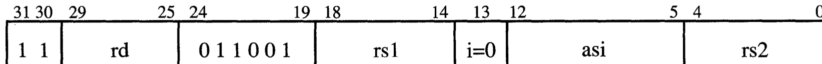
Description: The LDSBA instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and sign-extended in r[rd].

If LDSBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:



LDSH Load Signed Half-word LDSH

Operation: r[rd] ← sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsh [*address*], reg_{rd}

Description: The LDSH instruction moves a signed half-word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched half-word is right-justified and sign-extended in r[rd].

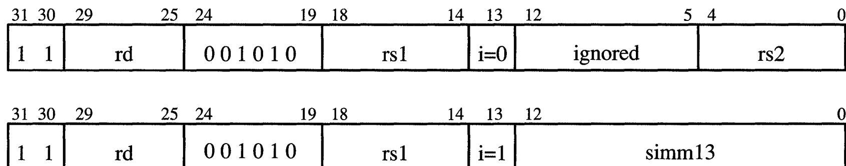
If LDSH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
memory_address_not_aligned
data_access_exception

Format:



LDSHA — Load Signed Half-word from Alternate space — LDSHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsha [reg_{addr}] *asi*, reg_{rd}

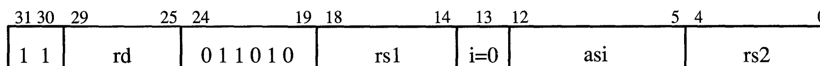
Description: The LDSHA instruction moves a signed half-word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched half-word is right-justified and sign-extended in r[rd].

If LDSHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 1 0 1 0	rs1	i=0	asi	rs2

LDSTUB — Atomic Load-Store Unsigned Byte — LDSTUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← FFFFFFFF H

Assembler Syntax: ldstub [*address*], reg_{rd}

Description: The LDSTUB instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic Load-Store instructions which address the same byte simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

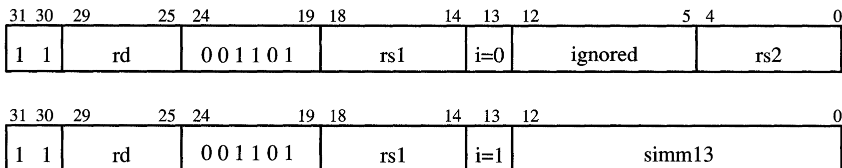
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:



LDSTUBA Atomic Load-Store Unsigned Byte in Alternate space (Privileged Instruction) LDSTUBA

Operation:

address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]
[r[rs1] + r[rs2]] ← FFFFFFFF H

Assembler Syntax:

ldstuba [reg_{addr}] *asi*, reg_{rd}

Description: The LDSTUBA instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic Load-Store instructions which address the same byte simultaneously are guaranteed to execute them in some serial order.

The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

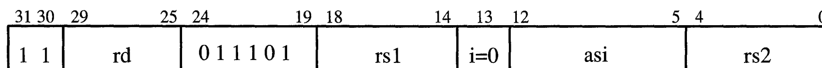
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUBA takes a trap, the contents of the memory address remain unchanged.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:



LDUB — Load Unsigned Byte — LDUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldub [*address*], reg_{rd}

Description: The LDUB instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

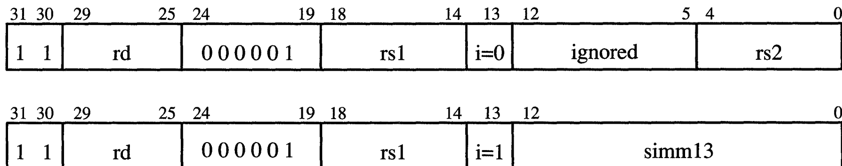
If LDUB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:



LDUBA Load Unsigned Byte from Alternate space LDUBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduba [reg_{addr}] *asi*, reg_{rd}

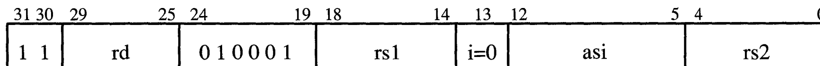
Description: The LDUBA instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If LDUBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:



LDUH Load Unsigned Half-word LDUH

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: lduh [*address*], reg_{rd}

Description: The LDUH instruction moves an unsigned half-word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched half-word is right-justified and zero-extended in r[rd].

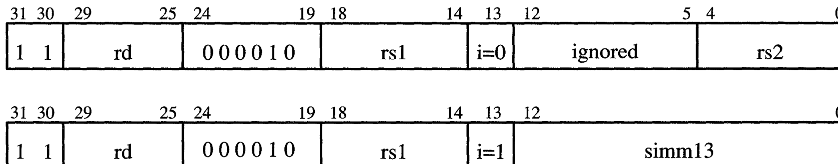
If LDUH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



LDUHA — Load Unsigned Half-word from Alternate space — LDUHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduha [*regaddr*] *asi*, *reg*$_{rd}$

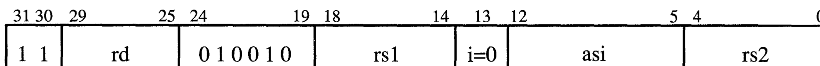
Description: The LDUHA instruction moves an unsigned half-word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched half-word is right-justified and zero-extended in r[rd].

If LDUHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 0 1 0	rs1	i=0	asi	rs2

MULScc — Multiply Step and modify icc — MULScc

Operation:

```
op1 = (n XOR v) CONCAT r[rs1]<31:1>
if (Y<0> = 0) op2 = 0, else op2 = r[rs2] or sign extnd(simm13)
Y ← r[rs1]<0> CONCAT Y<31:1>
r[rd] ← op1 + op2
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← ((op1<31> AND op2<31> AND not r[rd]<31>)
        OR (not op1<31> AND not op2<31> AND r[rd]<31>))
c ← ((op1<31> AND op2<31>)
        OR (not r[rd] AND (op1<31> OR op2<31>))
```

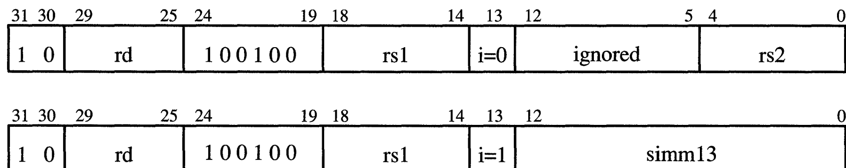
Assembler Syntax: mulscc *reg_{rs1}, reg_or_imm, reg_{rd}*

Description: The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words. MULScc works as follows:

1. The "incoming partial product" in r[rs1] is shifted right by one bit and the high-order bit is replaced by the sign of the previous partial product (n XOR v). This is operand1.
2. If the least significant bit of the multiplier in the Y register equals zero, then operand2 is set to zero. If the LSB of the Y register equal one, then operand2 becomes the multiplicand, which is either the contents of r[rs2] if the instruction *i* field is zero, or sign extnd(simm13) if the *i* field is one. Operand2 is then added to operand1 and stored in r[rd] (the outgoing partial product).
3. The multiplier in the Y register is then shifted right by one bit and its high-order bit is replaced by the least significant bit of the incoming partial product in r[rs1].
4. The PSR's integer condition codes are updated according to the addition performed in step 2.

Traps: none

Format:



OR Inclusive-Or OR

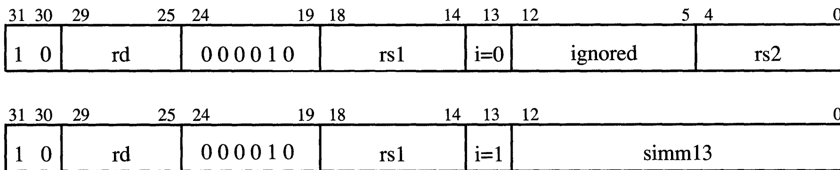
Operation: r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))

Assembler Syntax: or *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:



ORcc

Inclusive-Or and modify icc

ORcc

Operation: r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if [r[rd]]=0 then 1, else 0

v ← 0

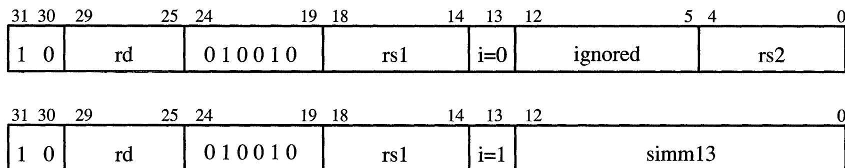
c ← 0

Assembler Syntax: orcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



ORN — Inclusive-Or Not — ORN

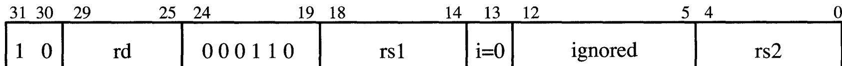
Operation: r[rd] ← r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))

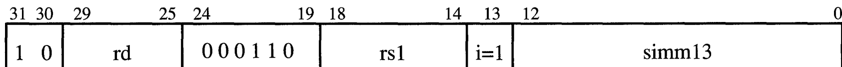
Assembler Syntax: orn reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ORNcc — Inclusive-Or Not and modify icc — ORNcc

Operation: r[rd] ← r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if [r[rd]]=0 then 1, else 0

v ← 0

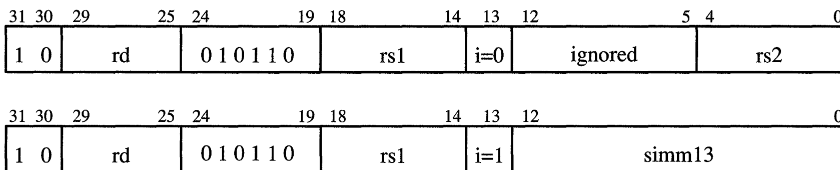
c ← 0

Assembler Syntax: orncc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORNcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



RDASR Read Ancillary State Register RDASR

Operation: r[rd] ← ASR (ancillary state register)

Assembler Syntax: rd %y, reg_{rd} (rdy special case)
rd asr_rs1, %rx (rdasr general case)

Description: RDASR copies the contents of the ancillary state register specified by rs1 into the register specified by the *rd* field.

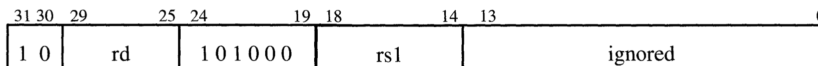
Version 8 of the SPARC Instruction Set Architecture defines instructions which access implementation dependent control registers called *ancillary state registers* (ASR's). As defined by the SPARC ISA, the existing **rdy** and **wry** instructions become special cases of the **rdasr** and **wrasr** instructions. When the rs1 field equals 0 – 15, a read Y register into the specified rd register is performed. When rs1 is $\leq$ 15, the function is affected by the PSR supervisor bit and is performed according to the table below.

For hyperSPARC, asr_rs1 can use the value %iccr = 0x1f (31 decimal) to read the instruction cache control register (ICCR), or 0x1e (30 decimal) to read the diagnostics register (DIAG). Any other asr_rs1 value is interpreted according to the following table:

rs1	read source	operation results
0	Y register	treat as rdy
1 to 15	Y register	treat as rdy
16 to 23	User unimplemented	treat as illegal trap
24 to 29	Privileged unimplemented	if S = 1 → illegal trap if S = 0 → priv viol trap
30	DIAG register	if S = 1 → rd DIAG if S = 0 → priv viol trap
31	ICCR register	if S = 1 → rd iccr if S = 0 → priv viol trap

Traps: As described in the table above.

Format:



RDPSR — Read Processor State Register — RDPSR

(Privileged Instruction)

Operation: r[rd] ← PSR

Assembler Syntax: rd %psr, reg_{rd}

Description: RDPSR copies the contents of the PSR into the register specified by the *rd* field.

Traps: privileged-instruction (if S=0)

Format:

RDTBR — Read Trap Base Register — RDTBR

(Privileged Instruction)

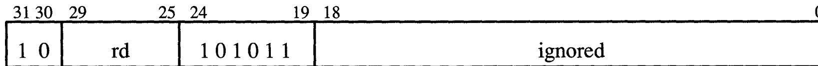
Operation: r[rd] ← TBR

Assembler Syntax: rd %tbr, reg_{rd}

Description: RDTBR copies the contents of the TBR into the register specified by the *rd* field.

Traps: privileged_instruction (if S=0)

Format:



RDWIM Read Window Invalid Mask register RDWIM

(Privileged Instruction)

Operation: r[rd] ← WIM

Assembler Syntax: rd %wim, reg_{rd}

Description: RDWIM copies the contents of the WIM register into the register specified by the *rd* field.

Traps: privileged_instruction (if S=0)

Format:

RDY Read Y register RDY

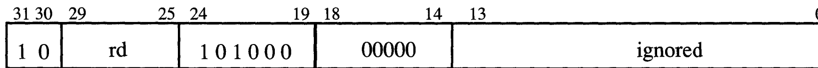
Operation: r[rd] ← Y

Assembler Syntax: rd %y, reg_{rd}

Description: RDY copies the contents of the Y register into the register specified by the *rd* field. Note that this is a special case of the RDASR instruction.

Traps: none

Format:



RESTORE — Restore caller's window — RESTORE

Operation:
ncwp ← CWP + 1
result ← r[rs1] + (r[rs2] or sign extnd(simm13))
CWP ← ncwp
r[rd] ← result
RESTORE does not affect condition codes

Assembler Syntax: restore reg_{rs1}, *reg_or_imm*, reg_{rd}

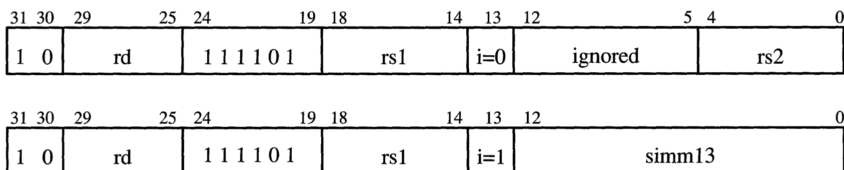
Description: RESTORE adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the window invalid mask (WIM) register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit $i = 1$) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit $i = 0$). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the called window).

The new CWP value is written into the PSR, causing the previous window (the caller's window) to become the active window. The result of the addition is now written into the r[rd] register of the restored window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps: window_underflow

Format:



RETT Return from Trap (Privileged Instruction) RETT

Operation:

ncwp ← CWP + 1

ET ← 1

PC ← nPC

nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))

CWP ← ncwp

S ← pS

Assembler Syntax: rett *address*

Description: RETT adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the window invalid mask (WIM) register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then RETT causes a delayed control transfer to the address derived by adding the contents of r[rs1] to either the contents of r[rs2] (field bit $i = 1$) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit $i = 0$).

Before the control transfer takes place, the new CWP value is written into the PSR, causing the previous window (the one in which the trap was taken) to become the active window. In addition, the PSR's ET bit is set to one (traps enabled) and the previous Supervisor bit (pS) is restored to the S field.

Although in theory RETT is a delayed control transfer instruction, in practice, RETT must always be immediately preceded by a JMPL instruction, creating a delayed control transfer couple (see *Section 2.4.3.4.4*). This has the effect of annulling the delay instruction.

If traps were already enabled before encountering the RETT instruction, an illegal_instruction trap is generated. If traps are not enabled (ET=0) when the RETT is encountered, but (1) the processor is not in supervisor mode (S=0), or (2) the window underflow condition described above occurs, or (3) if either of the two low-order bits of the target address are nonzero, then a reset trap occurs. If a reset trap does occur, the *tt* field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or memory_address_not_aligned.

RETT — Return from Trap — RETT

(Privileged Instruction)

Programming note: To re-execute the trapping instruction when returning from a trap handler, use the following sequence:

```
jmpl    %17, %0        ! old PC
rett    %18            ! old nPC
```

Note that the CY7C601 saves the PC in r[17] (local 1) and the nPC in r[18] (local2) of the trap window upon entering a trap.

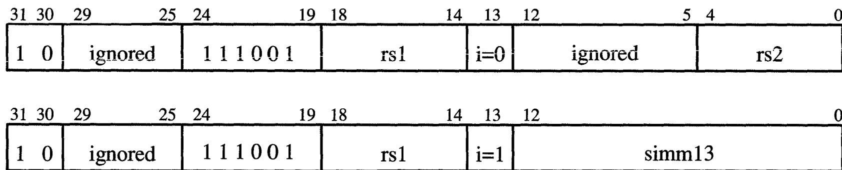
To return to the instruction after the trapping instruction (e.g., when the trapping instruction is emulated), use the sequence:

```
jmpl    %18,   %0      ! old nPC
rett    %18 + 4        ! old nPC + 4
```

Traps:

illegal_instruction
reset (privileged_instruction)
reset (memory_address_not_aligned)
reset (window_underflow)

Format:



SAVE — Save caller's window — SAVE

Operation:

ncwp ← CWP - 1

result ← r[rs1] + (r[rs2] or sign extnd(simm13))

CWP ← ncwp

r[rd] ← result

SAVE does not affect condition codes

Assembler Syntax: save reg_{rs1}, *reg_or_imm*, reg_{rd}

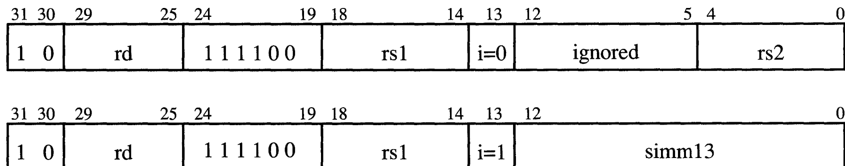
Description: SAVE subtracts one from the Current Window Pointer (modulo the number of implemented windows) and compares this value against the window invalid mask (WIM) register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_overflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit *i* = 1) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit *i* = 0). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the calling window).

The new CWP value is written into the PSR, causing the active window to become the previous window, and the called window to become the active window. The result of the addition is now written into the r[rd] register of the new window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps: window_overflow

Format:



SDIV Signed Divide SDIV

Operation: r[rd] ← Y r[rs1] ÷ (r[rs2] or sign extnd(simm13))

Assembler Syntax: sdiv reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. They either compute "((Y·*r*[*rs*1] ÷ *r*[*rs*2]))" (when the immediate field is zero) or "((Y·*r*[*rs*1] ÷ *sign_ext* (*simm13*))" (when the immediate field is one). The most signifacant 32 bits of the divisor are in the Y register, and the least significant 32 bits are in r[rs1]. The least significant 32 bits of the integer quotient are written into the destination register. The entire remainder and the most significant 32 bits of the quotient (if generated) are discarded. SDIV does not affect the condition code bits.

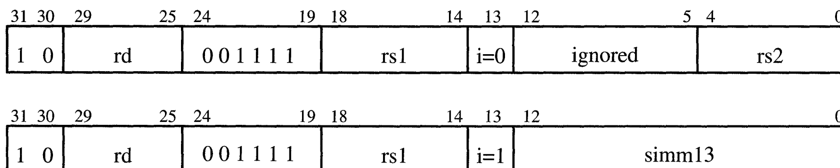
A signed divide (SDIV, SDIVcc) assumes a signed integer double-word dividend and a unsigned integer word divisor and computes a signed integer word quotient. Signed division rounds an inexact quotient towards zero if there is a non-zero remainder.

The result of a divide instruction can overflow the 32-bit destination register under certain conditions. When overflow occurs (whether or not the instruction sets the condition codes in the PSR), the largest appropriate integer is returned as the quotient in r[rd]. Overflow occurs when the result is greater than $2^{32}-1$ if the result is positive, or less than -2^{31} if the result is negative, and if the result has a remainder of divisor – 1. The value returned in r[rd] will be $2^{32}-1$ if the result is positive, and -2^{31} if the result is negative.

Note: For future compatibility, software should assume that the contents of the Y register are not preserved by the divide instructions.

Traps: If the divisor is zero, the instruction takes a divide-by-zero trap.

Format:



SDIVcc — Signed Divide (modify icc) — SDIVcc

Operation:

$r[rd] \leftarrow Y\ r[rs1] \div (r[rs2] \text{ or sign extnd}(simm13))$

$n \leftarrow r[rd]\langle 31 \rangle$

$z \leftarrow$ if r[rd]=0 then 1, else 0

$v \leftarrow$ 1 if overflow, else 0

$c \leftarrow 0$

Assembler Syntax: sdivcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. They either compute "((Y·*r*[*rs*1] ÷ *r*[*rs*2]))" (when the immediate field is zero) or "((Y·*r*[*rs*1] ÷ *sign_ext* (*simm13*))" (when the immediate field is one). The most signifacant 32 bits of the divisor are in the Y register, and the least significant 32 bits are in r[rs1]. The least significant 32 bits of the integer quotient are written into the destination register. The entire remainder and the most significant 32 bits of the quotient (if generated) are discarded.

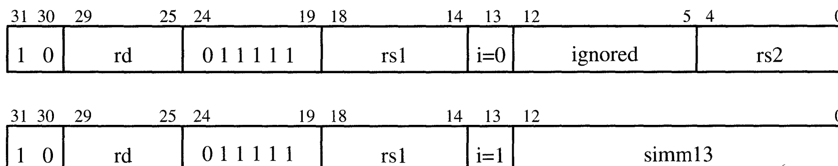
A signed divide (SDIV, SDIVcc) assumes a signed integer double-word dividend and a unsigned integer word divisor and computes a signed integer word quotient. Signed division rounds an inexact quotient towards zero if there is a non-zero remainder.

The result of a divide instruction can overflow the 32-bit destination register under certain conditions. When overflow occurs (whether or not the instruction sets the condition codes in the PSR), the largest appropriate integer is returned as the quotient in r[rd]. Overflow occurs when the result is greater than $2^{32}-1$ if the result is positive, or less than -2^{31} if the result is negative, and if the result has a remainder of divisor – 1. The value returned in r[rd] will be $2^{32}-1$ if the result is positive, and -2^{31} if the result is negative.

Note: For future compatibility, software should assume that the contents of the Y register are not preserved by the divide instructions.

Traps: If the divisor is zero, the instruction takes a divide-by-zero trap.

Format:



SETHI — Set High 22 bits of *r-register* — SETHI

Operation: r[rd]<31:10> ← imm22
r[rd]<9:0> ← 0

Assembler Syntax: sethi *const22*, reg_{rd}
sethi %hi *value*, reg_{rd}

Description: SETHI zeros the ten least significant bits of the contents of r[rd] and replaces its high-order 22 bits with *imm22*. The condition codes are not affected.

Programming note: SETHI 0, %0 is the preferred instruction to use as a NOP, because it will not increase execution time if it follows a load instruction.

Traps: none

Format:

SLL Shift Left Logical SLL

Operation: r[rd] ← r[rs1] SLL by (r[rs2] or shcnt)

Assembler Syntax: sll reg_{rs1}, *reg_or_imm*, reg_{rd}

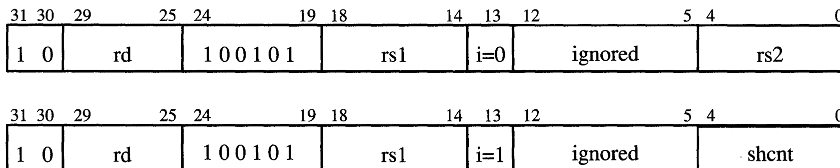
Description: SLL shifts the contents of r[rs1] left by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SLL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SLL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:



SMUL — Signed Multiply — SMUL

Operation: r[Y], r[rd] ← r[rs1] x (r[rs2] or sign extnd(simm13)), where the upper 32-bit result is placed in r[Y] (the Y register)

Assembler Syntax: smul reg_{rs1}, *reg_or_imm*, reg_{rd}

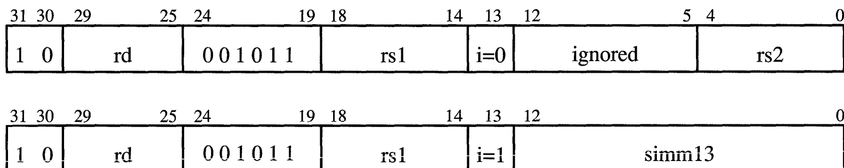
Description: The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They either multiply the contents of two registers (when the immediate field is zero) or the content of a register and the sign extended immediate operand (when the immediate field is one). They write the 32 most significant bits of the product into the Y register and the 32 least significant bits into the destination register.

A signed multiply (SMUL, SMULcc) assumes signed integer word operands and computes a signed integer double-word product. SMUL does not affect the condition code bits.

Note: 32-bit overflow after SMUL is indicated by $Y \neq 0$.

Traps: None

Format:



SMULcc — Signed Multiply (modify icc) — SMULcc

Operation: r[Y], r[rd] ← r[rs1] x (r[rs2] or sign extnd(simm13)),
where the upper 32-bit result is placed in r[Y] (the Y register)
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: smulcc reg_{rs1}, *reg_or_imm*, reg_{rd}

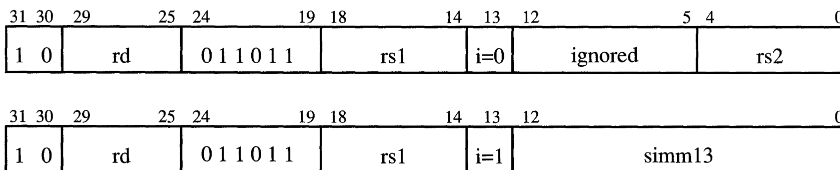
Description: The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They either multiply the contents of two registers (when the immediate field is zero) or the content of a register and the sign extended immediate operand (when the immediate field is one). They write the 32 most significant bits of the product into the Y register and the 32 least significant bits into the destination register.

A signed multiply (SMUL, SMULcc) assumes signed integer word operands and computes a signed integer double-word product. SMUL does not affect the condition code bits.

Note: 32-bit overflow after SMUL is indicated by $Y \neq 0$.
The negative (N) and zero (Z) condition code bits are set according to the least significant word of the product.

Traps: None

Format:



SRA — Shift Right Arithmetic — SRA

Operation: r[rd] ← r[rs1] SRA by (r[rs2] or shcnt)

Assembler Syntax: sra reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: SRA shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with the MSB of r[rs1]. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

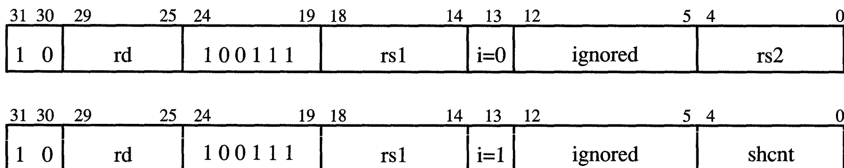
If the i bit field equals zero, the shift count for SRA is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRA is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt.*

This instruction does *not* modify the condition codes.

Programming note: A "Shift Left Arithmetic by 1 (and calculate overflow)" can be implemented with an ADDcc instruction.

Traps: none

Format:



SRL — Shift Right Logical — SRL

Operation: r[rd] ← r[rs1] SRL by (r[rs2] or shcnt)

Assembler Syntax: srl *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

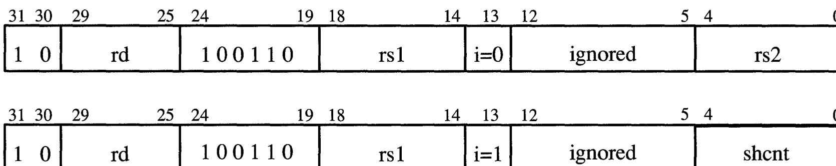
Description: SRL shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SRL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:



ST Store Word ST

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: st *reg*$_{rd}$, [*address*]

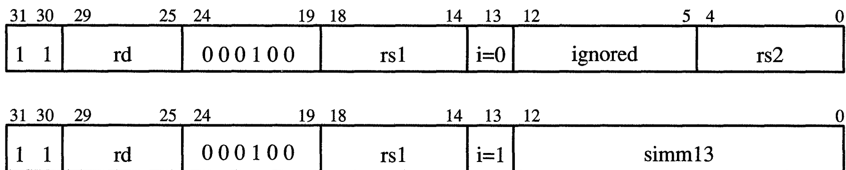
Description: The ST instruction moves a word from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If ST takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STA Store Word into Alternate space STA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

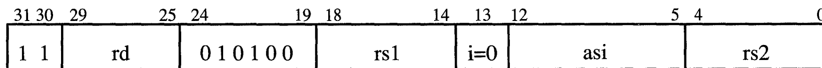
Assembler Syntax: sta reg_{rd}, [*regaddr*] *asi*

Description: The STA instruction moves a word from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 1 0 0	rs1	i=0	asi	rs2

STB — Store Byte — STB

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: stb reg_{rd}, [*address*]

synonyms: stub, stsb

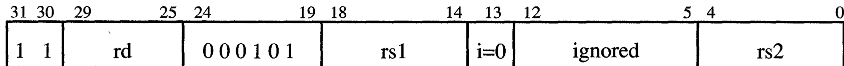
Description: The STB instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

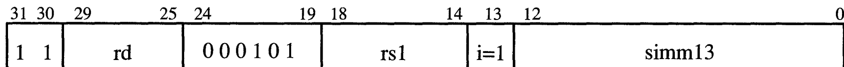
If STB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps: data_access_exception

Format:





STBA — Store Byte into Alternate space — STBA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

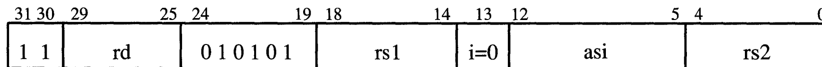
Assembler Syntax: stba reg_{rd}, [*regaddr*] *asi*
synonyms: stuba, stsba

Description: The STBA instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STBA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:



STC Store Coprocessor register STC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← c[rd]

Assembler Syntax: st $creg_{rd}$, [*address*]

Description: The STC instruction moves a word from a coprocessor register, c[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

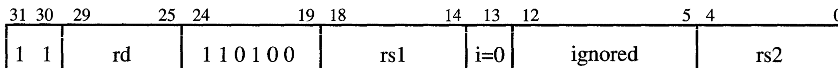
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STC takes a trap, memory remains unchanged.

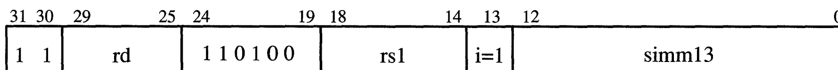
Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





STCSR — Store Coprocessor State Register — STCSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← CSR

Assembler Syntax: st %csr, [*address*]

Description: The STCSR instruction moves the contents of the coprocessor state register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

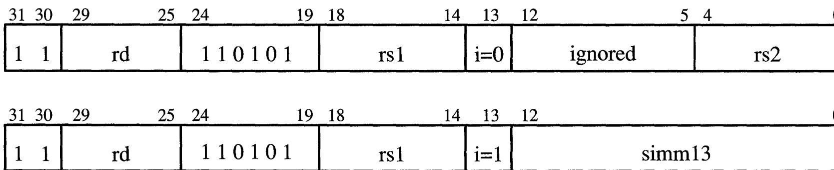
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STCSR takes a trap, the contents of the memory address remain unchanged.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



STD — Store Double-word — STD

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← r[rd + 1]

Assembler Syntax: std reg_{rd}, [*address*]

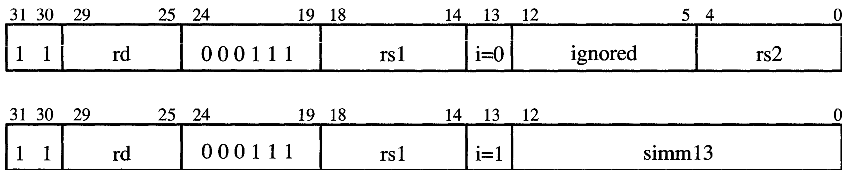
Description: The STD instruction moves a double-word from the destination register pair, r[rd] and r[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STDA — Store double-word into Alternate space — STDA

(Privileged Instruction)

Operation:
address space ← asi
[r[rs1] + (r[rs2]] ← r[rd]
[r[rs1] + (r[rs2] + 4] ← r[rd + 1]

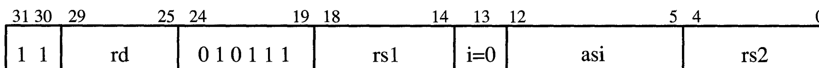
Assembler Syntax: stda *reg*$_{rd}$, [*regaddr*] *asi*

Description: The STDA instruction moves a double-word from the destination register pair, r[rd] and r[rd+1], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Traps:
illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



STDC — Store double-word Coprocessor — STDC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← c[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← c[rd + 1]

Assembler Syntax: std *$creg_{rd}$*, [*address*]

Description: The STDC instruction moves a double-word from the coprocessor register pair, c[rd] and c[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

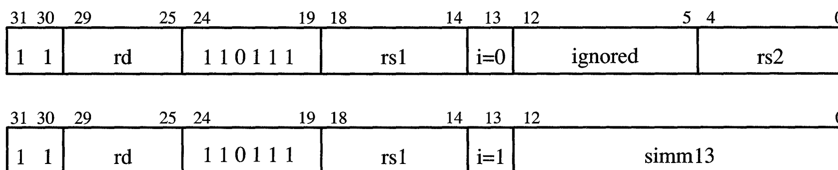
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



STDCQ — Store double-word Coprocessor Queue (Privileged Instruction) — STDCQ

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← CQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← CQ.INSTR

Assembler Syntax: std %cq, [*address*]

Description: The STDCQ instruction moves the front entry of the coprocessor queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4.

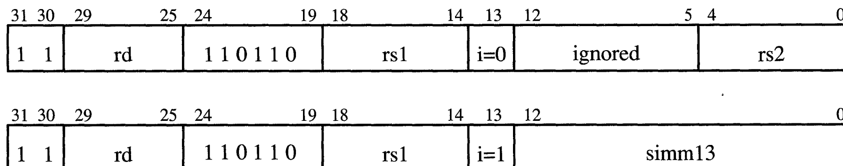
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Note that hyperSPARC processors do not support the coprocessor interface. The execution of a coprocessor instruction by a hyperSPARC instruction results in an cp_disabled trap.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



STDF — Store Double-word Floating-Point — STDF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← f[rd + 1]

Assembler Syntax: std *freg*$_{rd}$, [*address*]

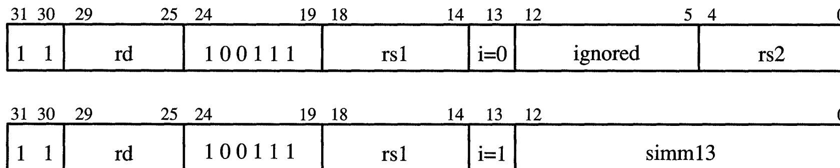
Description: The STDF instruction moves a double-word from the floating-point register pair, f[rd] and f[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STDFQ — Store Double-word Floating-Point Queue — STDFQ

(Privileged Instruction)

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← FQ.INSTR

Assembler Syntax: std %fq, [*address*]

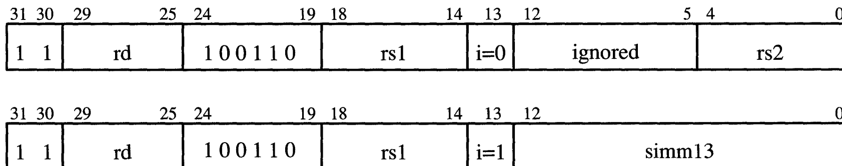
Description: The STDFQ instruction moves the front entry of the floating-point queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4. If the FPU is in exception mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the *qne* bit in the FSR).

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STF — Store Floating-Point register — STF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]

Assembler Syntax: st $freg_{rd}$, [*address*]

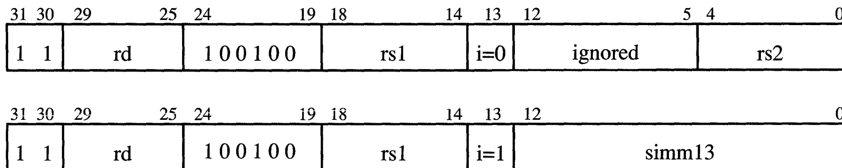
Description: The STF instruction moves a word from a floating-point register, f[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STF takes a trap, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STFSR — Store Floating-Point State Register — STFSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FSR

Assembler Syntax: st %fsr, [*address*]

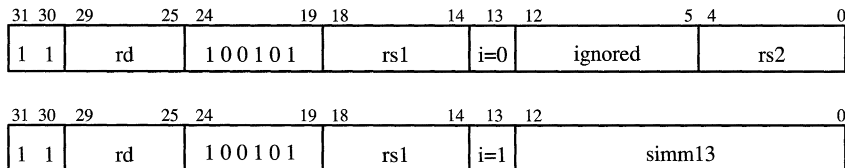
Description: The STFSR instruction moves the contents of the floating-point state register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it writes the FSR into memory.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STFSR takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STH — Store Half-word — STH

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: sth *reg*$_{rd}$, [*address*] synonyms: stuh, stsh

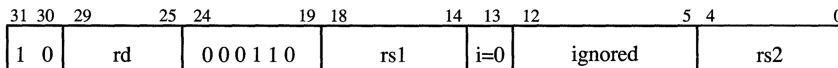
Description: The STH instruction moves the least significant half-word from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

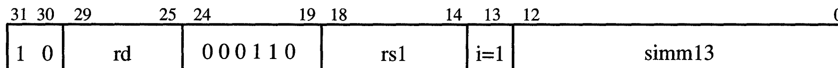
If STH takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be written to without setting up a register.

Traps:
memory_address_not_aligned
data_access_exception

Format:





STHA — Store Half-word into Alternate space — STHA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + (r[rs2]] ← r[rd]

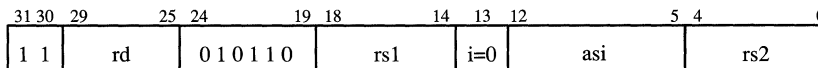
Assembler Syntax: stha *reg*$_{rd}$, [*address*]
synonyms: stuha, stsha

Description: The STHA instruction moves the least significant half-word from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STHA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



SUB Subtract SUB

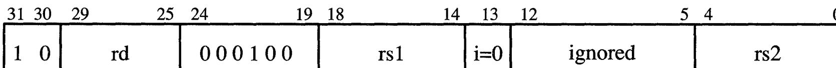
Operation: r[rd] ← r[rs1] - (r[rs2] or sign extnd(simm13))

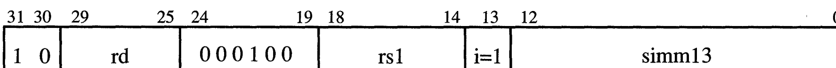
Assembler Syntax: sub *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: The SUB instruction subtracts either the contents of the register named in the *rs2* field, r[rs2], if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one, from register r[rs1]. The result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBcc **Subtract and modify icc** SUBcc

Operation: r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
  OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)

c ← (not r[rs1]<31> AND operand2<31>)
  OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

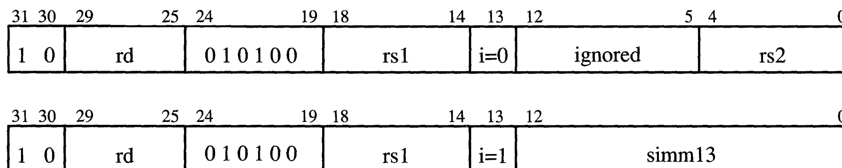
Assembler Syntax: subcc *regrs1, reg_or_imm, regrd*

Description: The SUBcc instruction subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in register r[rd]. In addition, SUBcc modifies all the integer condition codes in the manner described above.

Programming note: A SUBcc instruction with *rd* = 0 can be used for signed and unsigned integer comparison.

Traps: none

Format:



SUBX **Subtract with Carry** SUBX

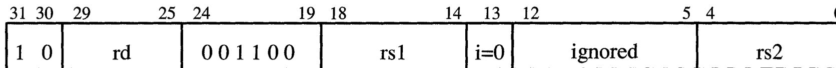
Operation: r[rd] ← r[rs1] - (r[rs2] or sign extnd(simm13)) - c

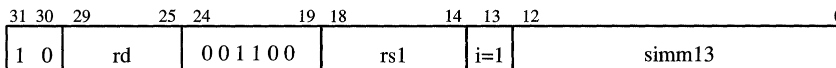
Assembler Syntax: subx *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SUBX subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBXcc Subtract with Carry and modify icc SUBXcc

Operation: r[rd] ← r[rs1] - operand2 - c, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
  OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)

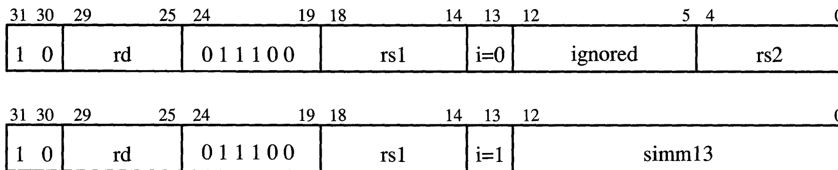
c ← (not r[rs1]<31> AND operand2<31>)
  OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

Assembler Syntax: subxcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SUBXcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field. In addition, SUBXcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:



SWAP — Swap *r-register* with memory — SWAP

Operation:
word ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
temp ← r[rd]
r[rd] ← word
r[rs1] + (r[rs2] or sign extnd(simm13)) ← temp

Assembler Syntax: swap [*source*], reg_{rd}

Description: SWAP atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAP instructions simultaneously are guaranteed to execute them serially, in some order.

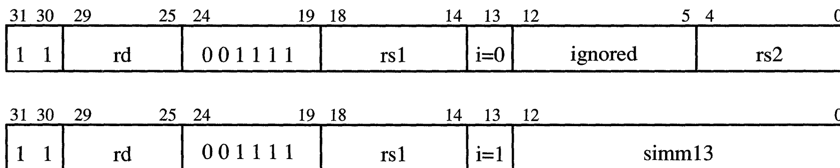
The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If SWAP takes a trap, the contents of the memory address and the destination register remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 Kbytes of an address space can be accessed without setting up a register.

Traps:
memory_address_not_aligned
data_access_exception

Format:



SWAPA — Swap *r-register* with memory in Alternate space — SWAPA

(Privileged Instruction)

Operation:

address space ← asi
word ← [r[rs1] + r[rs2]]
temp ← r[rd]
r[rd] ← word
[r[rs1] + r[rs2]] ← temp

Assembler Syntax: swapa [*regsource*] *asi*, reg_{rd}

Description: SWAPA atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAPA instructions simultaneously are guaranteed to execute them serially, in some order.

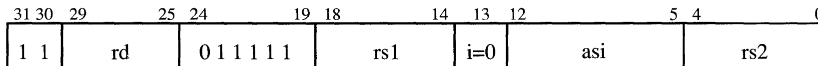
The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If SWAPA takes a trap, the contents of the memory address and the destination register remain unchanged.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



TADDcc — Tagged Add and modify icc — TADDcc

Operation: r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd]=0 then 1, else 0

v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
OR (r[rs1]<1:0> · 0 OR operand2<1:0> · 0)

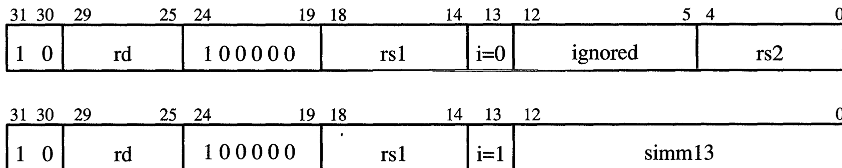
c ← (r[rs1]<31> AND operand2<31>
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax: taddcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: TADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TADDcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:



TADDccTV — Tagged Add (modify icc) Trap on Overflow — TADDccTV

Operation:

```
result ← r[rs1] + operand2, where operand 2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
        OR (r[rs1]<1:0> · 0 OR operand2<1:0> · 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (r[rs1]<31> AND operand2<31>
        OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
r[rd] ← result
```

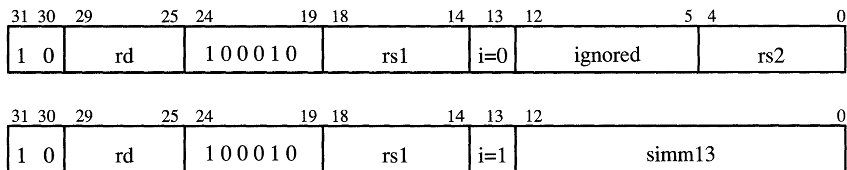
Assembler Syntax: taddcctv *reg_{rs1}, reg_or_imm, reg_{rd}*

Description: TADDccTV adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TADDccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TADDccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps: tag_overflow

Format:



Ticc — Trap on integer condition codes — Ticc

Operation: If condition true, then trap_instruction;

tt ← 128 + [r[rs1] + (r[rs2] or sign extnd(simm13))]<6:0>

else PC ← nPC

nPC ← nPC + 4

Assembler Syntax:

ta	*software_trap_number*	
tn	*software_trap_number*	
tne	*software_trap_number*	synonym: tnz
te	*software_trap_number*	synonym: tz
tg	*software_trap_number*	
tle	*software_trap_number*	
tge	*software_trap_number*	
tl	*software_trap_number*	
tgu	*software_trap_number*	
tleu	*software_trap_number*	
tcc	*software_trap_number*	synonym: tgeu
tcs	*software_trap_number*	synonym: tlu
tpos	*software_trap_number*	
tneg	*software_trap_number*	
tvc	*software_trap_number*	
tvs	*software_trap_number*	

Description: A Ticc instruction evaluates specific integer condition code combinations (from the PSR's *icc* field) based on the trap type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, and there are no higher-priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate as false, the trap is not generated.

If a trap_instruction trap is generated, the *tt* field of the trap base register (TBR) is written with 128 plus the least significant seven bits of r[rs1] plus either r[rs2] (bit field i =0) or the 13-bit sign-extended immediate value contained in the instruction (bit field i =1). See *Section 2.4.5* for the complete definition of a trap.

Traps: trap_instruction

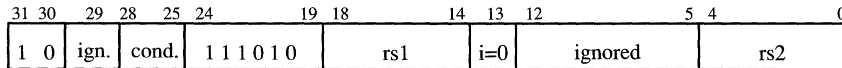
Ticc

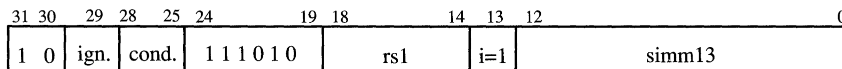
Trap on integer condition codes

Ticc

Mnemonic	Cond.	Operation	icc Test
TN	0000	Trap Never	No test
TE	0001	Trap on Equal	z
TLE	0010	Trap on Less or Equal	z OR (n XOR v)
TL	0011	Trap on Less	n XOR v
TLEU	0100	Trap on Less or Equal, Unsigned	c OR z
TCS	0101	Trap on Carry Set (Less then, Unsigned)	c
TNEG	0110	Trap on Negative	n
TVS	0111	Trap on oVerflow Set	v
TA	1000	Trap Always	No test
TNE	1001	Trap on Not Equal	not z
TG	1010	Trap on Greater	not(z OR (n XOR v))
TGE	1011	Trap on Greater or Equal	not(n XOR v)
TGU	1100	Trap on Greater, Unsigned	not(c OR z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not c
TPOS	1110	Trap on Positive	not n
TVC	1111	Trap on oVerflow Clear	not v

Format:





ign. = ignored
cond. = condition

TSUBcc — Tagged Subtract and modify icc — TSUBcc

Operation: r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd]=0 then 1, else 0

v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>) OR (r[rs1]<1:0> · 0 OR operand2<1:0> · 0)

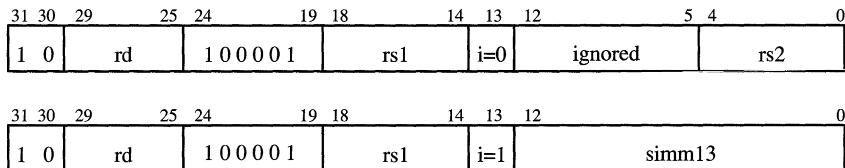
c ← (not r[rs1]<31> AND operand2<31> OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

Assembler Syntax: tsubcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: TSUBcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TSUBcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:



TSUBccTV — Tagged Subtract (modify icc) Trap on Overflow — TSUBccTV

Operation:

result ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))

tv ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31>
AND operand2<31> AND r[rd]<31>)
OR (r[rs1]<1:0> · 0 OR operand2<1:0> · 0)

if tv = 1, then tag overflow trap; else

n ← r[rd]<31>

z ← if r[rd]=0 then 1, else 0

v ← tv

c ← (not(r[rs1]<31>) AND operand2<31> OR
(r[rd]<31> AND (not(r[rs1]<31>) OR operand2<31>))

r[rd] ← result

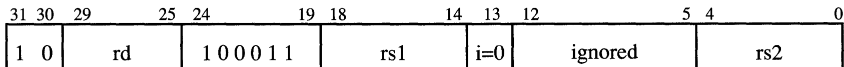
Assembler Syntax: tsubcctv reg_{rs1}, *reg_or_imm*, reg_{rd}

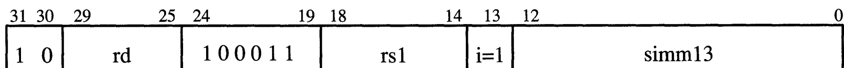
Description: TSUBccTV subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TSUBccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TSUBccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps: tag_overflow

Format:





UDIV — Unsigned Divide — UDIV

Operation: r[rd] ← Y r[rs1] ÷ (r[rs2] or sign extnd(simm13))

Assembler Syntax: udiv reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. They either compute "((Y *r*[*rs*1] ÷ *r*[*rs*2]))" (when the immediate field is zero) or "((Y *r*[*rs*1] ÷ *sign_ext* (*simm13*))" (when the immediate field is one). The most significant 32 bits of the dividend are in the Y register, and the least significant 32 bits are in r[rs1]. The least significant 32 bits of the integer quotient are written into the destination register. The entire remainder and the most significant 32 bits of the quotient (if generated) are discarded. UDIV does not affect the condition code bits.

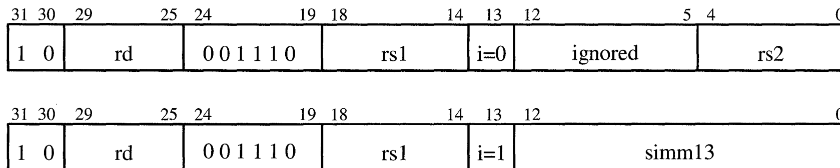
An unsigned divide (UDIV, UDIVcc) assumes a unsigned integer double-word dividend and an unsigned integer word divisor and computes an unsigned integer word quotient. However, if the second source operand is an immediate operand, it is sign-extended (to 32 bits) as usual.

The result of a divide instruction can overflow the 32-bit destination register under certain conditions. When overflow occurs (whether or not the instruction sets the condition codes in the PSR), the largest appropriate integer is returned as the quotient in r[rd]. Overflow occurs when the result is greater than $2^{32} - 1$ with a remainder of divisor – 1. The value returned in r[rd] will be $2^{32} - 1$.

Note: For future compatibility, software should assume that the contents of the Y register are not preserved by the divide instructions.

Traps: If the divisor is zero, the instruction takes a divide-by-zero trap.

Format:



UDIVcc **Unsigned Divide (modify icc)** UDIVcc

Operation:
r[rd] ← Y r[rs1] ÷ (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← 1 if overflow, else 0
c ← 0

Assembler Syntax: udivcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: The divide instructions perform 64-bit by 32-bit division, producing a 32-bit result. They either compute "((Y *r*[*rs*1] ÷ *r*[*rs*2]))" (when the immediate field is zero) or "((Y *r*[*rs*1] ÷ *sign_ext* (*simm13*))" (when the immediate field is one). The most significant 32 bits of the dividend are in the Y register, and the least significant 32 bits are in r[rs1]. The least significant 32 bits of the integer quotient are written into the destination register. The entire remainder and the most significant 32 bits of the quotient (if generated) are discarded.

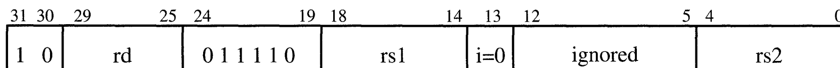
An unsigned divide (UDIV, UDIVcc) assumes a unsigned integer double-word dividend and an unsigned integer word divisor and computes an unsigned integer word quotient. However, if the second source operand is an immediate operand, it is sign-extended (to 32 bits) as usual.

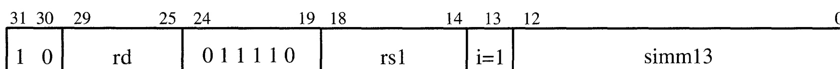
The result of a divide instruction can overflow the 32-bit destination register under certain conditions. When overflow occurs (whether or not the instruction sets the condition codes in the PSR), the largest appropriate integer is returned as the quotient in r[rd]. Overflow occurs when the result is greater than $2^{32}-1$ with a remainder of divisor – 1. The value returned in r[rd] will be $2^{32}-1$.

Note: For future compatibility, software should assume that the contents of the Y register are not preserved by the divide instructions.

Traps: If the divisor is zero, the instruction takes a divide-by-zero trap.

Format:





UMUL Unsigned Multiply UMUL

Operation: r[Y], r[rd] ← r[rs1] x (r[rs2] or sign extnd(simm13)),
where the upper 32-bit result is placed in r[Y] (the Y register)

Assembler Syntax: umul reg_{rs1}, *reg_or_imm*, reg_{rd}

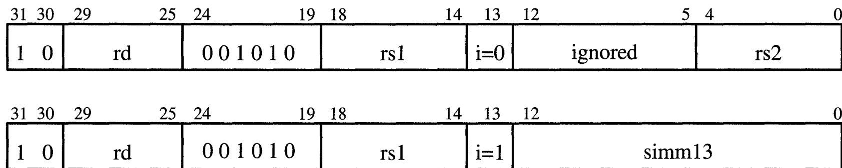
Description: The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They either multiply the contents of two registers (when the immediate field is zero) or the content of a register and the sign extended immediate operand (when the immediate field is one). They write the 32 most significant bits of the product into the Y register and the 32 least significant bits into the destination register.

An unsigned multiply, UMUL, assumes unsigned integer word operands and computes an unsigned integer double-word product. However, if the second source operand is an immediate operand, it is sign-extended (to 32 bits) as usual. UMUL does not affect the condition code bits.

Note: 32-bit overflow after UMUL is indicated by Y $\neq$ 0

Traps: None

Format:



UMULcc — Unsigned Multiply (modify icc) — UMULcc

Operation: r[Y], r[rd] ← r[rs1] x (r[rs2] or sign extnd(simm13)),
where the upper 32-bit result is placed in r[Y] (the Y register)
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: umulcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

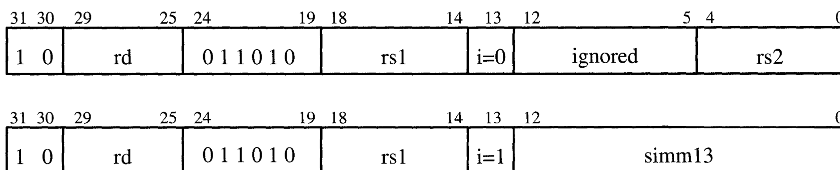
Description: The multiply instructions perform 32-bit by 32-bit multiplications, producing 64-bit results. They either multiply the contents of two registers (when the immediate field is zero) or the content of a register and the sign extended immediate operand (when the immediate field is one). They write the 32 most significant bits of the product into the Y register and the 32 least significant bits into the destination register.

An unsigned multiply, UMUL, assumes unsigned integer word operands and computes an unsigned integer double-word product. However, if the second source operand is an immediate operand, it is sign-extended (to 32 bits) as usual. UMUL does not affect the condition code bits.

Note: 32-bit overflow after UMUL is indicated by Y $\neq$ 0.
The negative (N) and zero (Z) condition code bits are set according to the least significant word of the product.

Traps: None

Format:



UNIMP — Unimplemented instruction — UNIMP

Operation: illegal instruction trap

Assembler Syntax: unimp *const22*

Description: Executing the UNIMP instruction causes an immediate illegal_instruction trap. The value in the const22 field is ignored.

Programming note: UNIMP can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure.

1. An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.
2. If the called function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the const22 operand of the UNIMP instruction. The called function can check the opcode to make sure it is indeed UNIMP.
3. If the function is not going to return a structure, upon returning, it attempts to execute UNIMP rather than skipping over it as it should. This causes the program to terminate. The behavior adds some run-time checking to an interface that cannot be checked properly at compile time.

Traps: illegal_instruction

Format:

31 30	29 … 25	24 … 22	21 … 0
0 0	ignored	0 0 0	const22

WRASR Write Ancillary State Register WRASR

Operation: r[rd] ← ASR (ancillary state register)

Assembler Syntax:

wr %rx, %ry, %y (**wry special case**)
wr %rx, simm13, %y
wr %rx, %ry, asr_rd (**wrasr general case**)
wr %rx, simm13, asr_rd

Description: WRASR computes the bitwise exclusive-or (XOR) of operand 1 (rs1) and operand 2 (either rs2 or simm13) and write the result into the specified state register. WRY is now a subcase of WRASR. When the rd field equals 0 – 15, a write into the Y register is performed. When rd is $\leq$ 15, the write function is affected by the PSR supervisor bit, and is performed as follows:

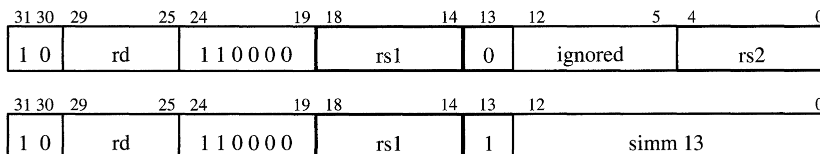
rd	write destination	operation results
0	Y register	treat as wry
1 to 15	Y register	treat as wry
16 to 23	User unimplemented	treat as illegal trap
24 to 29	Privileged unimplemented	If S= 1 → illegal trap If S = 0 → priv viol trap
30	DIAG register	If S = 1 → wr DIAG If S = 0 → priv viol trap
31	IBC r-register	If S = 1 → wr iccr If S = 0 → priv viol trap

For hyperSPARC, asr_rd can use the value %iccr = 0x1f (31 decimal) to write to the instruction cache control register, or 0x1e (30 decimal) to write to the diagnostic register (DIAG). Any other asr_rd value is interpreted according to the table above.

Traps: As described in the table above.

Note: For hyperSPARC, any write special register instruction causes packet splitting to be enforced for the next three instructions.

Format:



WRPSR — Write Processor State Register (Privileged Instruction) — WRPSR

Operation: PSR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr reg_{rs1}, *reg_or_imm*, %psr

Description: WRPSR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the writable subfields of the PSR. However, if the result's CWP field would point to an unimplemented window, an illegal_instruction trap is generated and the PSR remains unchanged.

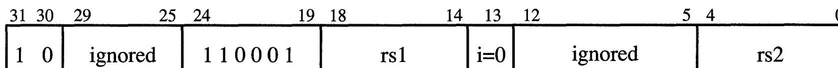
WRPSR is a delayed-write instruction:

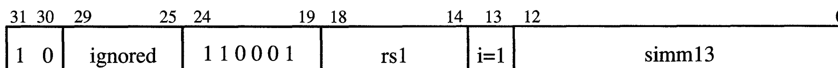
1. If any of the three instructions following a WRPSR uses any PSR field that WRPSR modified, the value of that field is unpredictable. Note that any instruction which references a non-global register makes use of the CWP, so following WRPSR with three NOPs would be the safest course.
2. If a WRPSR instruction is updating the PSR's processor interrupt level (PIL) to a new value and is simultaneously setting enable traps (ET) to one, this could result in an interrupt trap at a level equal to the old PIL value.
3. If any of the three instructions after a WRPSR instruction reads the modified PSR, the value read is unpredictable.
4. If any of the three instructions after a WRPSR is trapped, a subsequent RDPSR in the trap handler will get the register's new value.

Programming note: Two WRPSR instructions should be used when enabling traps and changing the PIL value. The first WRPSR should specify ET=0 with the new PIL value, and the second should specify ET=1 with the new PIL value.

Traps: illegal_instruction
privileged_instruction (if S=0)

Format:





WRTBR — Write Trap Base Register — WRTBR

(Privileged Instruction)

Operation: TBR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg*$_{rs1}$, *reg_or_imm*, %tbr

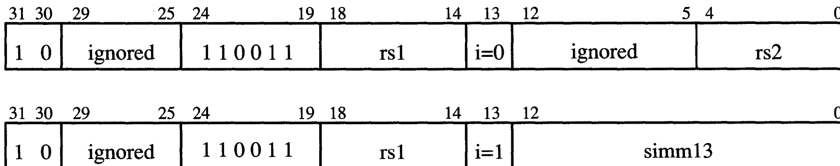
Description: WRTBR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the trap base address field of the TBR.

WRTBR is a delayed-write instruction:

1. If any of the three instructions following a WRTBR causes a trap, the TBA used may be either the old or the new value.
2. If any of the three instructions after a WRTBR is trapped, a subsequent RDTBR in the trap handler will get the register's new TBA value.

Traps: privileged_instruction (if S=0)

Format:



WRWIM — Write Window Invalid Mask register — WRWIM

(Privileged Instruction)

Operation: WIM ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr reg_{rs1}, *reg_or_imm*, %wim

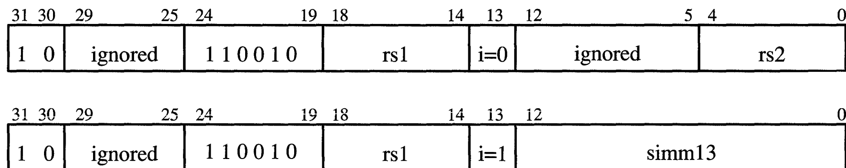
Description: WRWIM does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the writable bits of the WIM register.

WRWIM is a delayed-write instruction:

1. If any of the three instructions following a WRWIM is a SAVE, RESTORE, or RETT, the occurrence of window_overflow and window_underflow is unpredictable.
2. If any of the three instructions after a WRWIM instruction reads the modified WIM, the value read is unpredictable.
3. If any of the three instructions after a WRWIM is trapped, a subsequent RDWIM in the trap handler will get the register's new value.

Traps: privileged_instruction (if S=0)

Format:



WRY — Write Y register — WRY

Operation: Y ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg*$_{rs1}$, *reg_or_imm*, %y

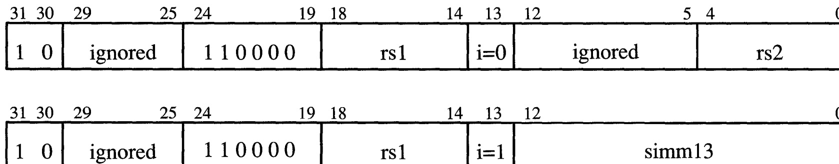
Description: WRY does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the Y register.

WRY is a delayed-write instruction:

1. If any of the three instructions following a WRY is a MULScc or a RDY, the value of Y used is unpredictable.
2. If any of the three instructions after a WRY instruction reads the modified Y register, the value read is unpredictable.
3. If any of the three instructions after a WRY is trapped, a subsequent RDY in the trap handler will get the register's new value.

Traps: none

Format:



XNOR Exclusive-Nor XNOR

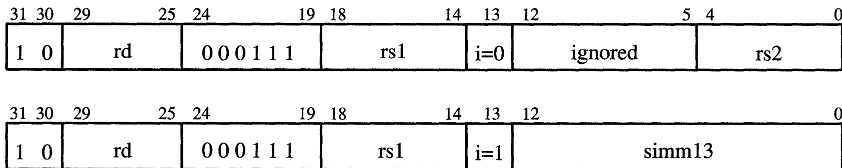
Operation: r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))

Assembler Syntax: xnor reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:



XNORcc Exclusive-Nor and modify icc XNORcc

Operation: r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← 0

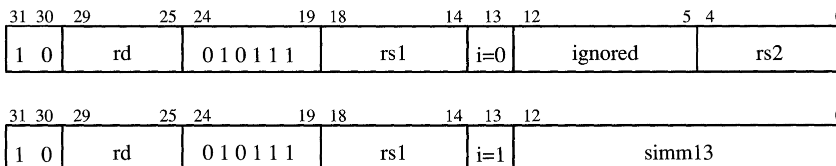
c ← 0

Assembler Syntax: xnorcc *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$*

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. XNORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



XOR Exclusive-Or XOR

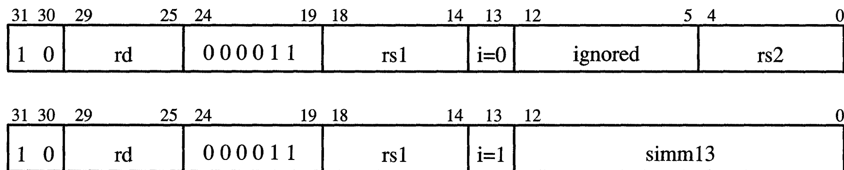
Operation: r[rd] ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: xor reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:



XORcc — Exclusive-Or and modify icc — XORcc

Operation: r[rd] ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

n ← r[rd]<31>

z ← if r[rd] =0 then 1, else 0

v ← 0

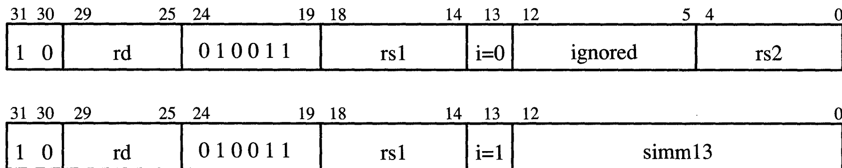
c ← 0

Assembler Syntax: xorcc reg_{rs1}, *reg_or_imm*, reg_{rd}

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. XORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



hyperSPARC Software Notes

The hyperSPARC architecture incorporates a number of features which compiler writers and/or assembly language programmers concerned with time critical systems such as operating systems or real-time applications can use to improve the instruction throughput. In this section, we discuss these features and also point out pitfalls and code sequences which should be avoided if feasible.

A.1 Reset Considerations

A reset trap can be caused by an initial system power-on. Also, it could be caused by a watchdog reset or software reset. A reset causes various status registers to be initialized to known values. This is done to support post-mortem analysis when system problems arise.

During a power-on reset when boot software begins execution through the reset trap handler, there are certain status register fields which should be initialized by software in order to support initial debugging and eliminate potential problems.

When reset occurs, the S bit is set and the enable trap (ET) bit is cleared automatically by the hardware. However, the enable floating-point (EF), processor interrupt level (PIL), previous supervisor (PS), and CWP (Current Window Pointer) fields are not initialized. Furthermore, the writeable window invalid mask (WIM) bits are not initialized and trap base register (TBR) (which includes the trap base address or TBA field) is not initialized. At power-on, these register fields take on unpredictable values. *Therefore, the boot program should initialize these critical fields after a power-on reset.*

EF – Initial debugging or diagnostic analysis is best accomplished by focusing on one section of hardware at a time. It is therefore recommended that the PSR's EF bit be cleared initially until the appropriate time to begin testing and/or executing fp instructions.

CWP – The CWP should be set to a known value (e.g., 0) so that the WIM can be correctly initialized and unanticipated window overflows and underflows can be avoided.

PIL – If interrupts are to be masked, the PIL should be set to the highest value (15 decimal) before traps are enabled (ET = 1). Likewise, a suggested method for modifying interrupt levels is to do so within a trap handler while traps are disabled.

WIM – The WIM is initialized through software, not hardware. If the WIM bits are not cleared initially, random bits could be set which can cause an unexpected window overflow (or underflow) trap at the execution of the first save (or restore) instruction. Typically, if the CWP is set to "n," then bit ((n+1) mod 8) in the WIM should be set. This reserves a register window for trap handling when window overflows or underflows occur.

TBR – The TBR is initially undefined. Reset causes instructions to be fetched beginning at virtual address 0. It is assumed that the trap table is initially loaded at virtual location 0. Once traps have been enabled, if a trap occurs and the TBA field has not either been written to zero, or written to the address page value at which a new trap table has been installed, it is likely that the processor will enter error mode since it will not vector to the actual trap table, but rather it will vector to some random location in memory.

Y – The Y register is initially undefined. It must be initialized before it is used.

ICCR – The instruction cache control register (ICCR) is initialized by hardware.

FSR – The only field initialized during reset is the qne bit which is cleared as a result of clearing the entries in the FPQ. The floating-point unit status register (FSR) can only be initialized if the EF bit of the PSR is set. The programmer should at least initialize the rounding direction (RD), trap enable mask (TEM), and non-standard floating-point (NS) fields to appropriate values before attempting to execute fp instructions.

A.1.1 Miscellaneous Notes

If there are fpops executing in the floating-point unit (FPU) and the FPU is disabled (via a WRPSR instruction which sets the EF bit in the PSR to zero) the behavior is undefined. Software should ensure that the queue is empty and all fp exceptions have been serviced before disabling the FPU.

When writing to a special register (e.g., PSR, WIM, TBR, Y,...), for the three subsequent instructions, DO NOT use any fields which are being modified.

Here is a sample of a bad code sequence...

```
wr CONST, %psr
add %i0, %l0, %o0 /* if CWP modified - problems... */
bg label /* if cc modified - problems... */
ta %g0 + %g0 /* if traps disabled - problems... */
wr %l0, % Y udiv reads incorrect value of Y
```

Here is a sample of an acceptable code sequence...

```
wr CONST, %psr
add %g2, %g3, %g1 /* if CWP modified - no problems... */
ba label /* if cc modified - no problems... */
nop
```

As mentioned in the previous section regarding power-on reset, it is best not to change interrupt levels except while traps are disabled. If traps are enabled and the PIL is modified, an interrupt can occur before the PIL modifications take effect and an interrupt intended to be masked can occur anyway.

When using Flush in a hyperSPARC uniprocessor system, there are no problems disabling the flush traps. Since there is only one instruction cache, flushing self-modifying code sequences from the on-chip instruction cache can be accomplished directly through the Flush instruction.

In a symmetric multiprocessing system running shared self-modifying code between processors, it can be difficult to distinguish which processors' on-chip instruction cache(s) needs to have its sub-block flushed. It is recommended that the flush trap disabled (FTD) bit be cleared in these environments. When the Flush instruction is attempted an unimplemented flush trap will occur. The unimplemented flush trap handler can be used to determine which processors' instruction cache(s) need flushing. These appropriate flushes will be performed through software.

A.2 Compiler Optimization Notes

A.2.1 Software pipelining:

Software pipelining means ordering instructions to increase performance, which is measured in terms of instructions executed in a fixed number of cycles. By observing some guidelines for instruction ordering,

execution speed can be greatly improved. These guidelines make use of special characteristics of the hyperSPARC architectural features as well as specific features tailored to the SPARC instruction set. These guidelines are:

Sequences to promote:

- Attempt to reorder instruction sequences so as to interleave ALU and Load/Store instructions (i.e., reorder resource dependencies). However, constant generation, array indexing, and ALUcc/Bicc sequences do not need interleaving since they are handled through the Fast Constant/Fast Index/Fast Branch features
- Try to interleave unrelated integer and fp instructions.

Sequences to avoid:

- any data dependency where a Load or atomic Load/Store, or an fp instruction is followed within three instructions by a dependent operation.
- a store instruction followed immediately by a Load, Store, or JMPL instruction.
- fp code sequences that involve an FDIV or FSQRT which is followed by more than 4 additional fp instructions. This does not include independent fp Loads, fp Stores, and fp Branches. Avoid a double precision multiplication followed by another multiplication.
- group F instructions except when necessary (Refer to *Section 2.4.1.4*).

A.2.2 Loop Unrolling:

Certain loops execute a fixed number of iterations and loop unrolling is used to reduce the number of times the conditional test and Branch instructions (present at the beginning or end of each iteration) are executed. It also increases the pipeline efficiency since control transfer now takes place after a larger number of instructions. However, the unrolled loop occupies a larger code space increasing the potential for cache misses and consequently, idle cycles. However, since most of the loops are quite small in size, if the cache is large enough, loop unrolling can be profitably employed to increase the execution speed. The hyperSPARC CPU has a large on-chip instruction cache of 8-Kbytes which facilitates loop unrolling. The loop executes from the instruction cache and, for most loops, will have a considerable speed-up in execution time.

A.2.3 Inter-procedure Optimizations:

Since the SPARC Architecture specifies a large number of overlapping register windows to support parameter passing for procedure calls, today's SPARC compilers do not usually make an effort to optimize register usage between procedure calls. Instead, a CALL or JMPL followed by a SAVE is used to enter the procedure. There are no problems as long as the nested procedure calls are not very deep. If the nested calls extend deeper, a window overflow occurs. Furthermore, additional nesting suffers window overflow handling as lower level procedures are called. The penalty doubles as the lower level procedures are exited and window underflow exceptions occur on the path upward through the calling procedure.

Today's structured design and programming styles advocate modularization. Software modules are expected to implement minimal data exchange and functional strength (i.e., each procedure or module is dedicated to computing a single specific result). This is usually achievable by passing a small number of parameters between procedures. Because of the generic approach used to make procedure calls in SPARC systems, many registers in the register file are unused, and therefore wasted.

However, many RISC microprocessors do not employ overlapping register windows. These processors suffer greatly when register states must be saved through expensive memory accesses. Therefore, compilers

which generate code for these machines optimize register usage between procedure calls, to minimize unnecessary register saves between procedure calls.

Therefore, the underflow/overflow trap overhead can be reduced by using smart register allocation algorithms for procedures which pass and use only small numbers of variables and eliminating unnecessary save and restore invocations.

A.2.3.1 Profile Tracing for Branch Prediction:

When the hyperSPARC CPU encounters a Branch instruction, it assumes that the branch is taken and starts fetching along the new instruction stream. Compilers for the hyperSPARC CPU should attempt to take advantage of this feature by providing trace scheduling facilities.

These trace scheduling facilities should allow execution of profile traces for programs and representative data such that reordering and modification of instruction branches are biased towards branch taken.

A.2.4 Operating Systems Notes

A.2.4.1 Context Switching and Machine State Saves:

Context switches and machine state saves should be conducted using store double and load double instructions to as great an extent as possible. hyperSPARC allows the programmer to move twice the data in the same amount of time using doubleword instructions rather than word size instructions.

Address Translation Cache (ATC): The ATC is a cache of address translation entries used by a Memory Management Unit (MMU) to translate virtual addresses to physical addresses. The CY7C604/605 uses an ATC for address translation, but the more familiar term translation lookaside buffer (TLB) is used throughout the text.

Aliasing: Mapping two or more virtual addresses to the same physical address. SPARC software conventions permit the use of aliases in address spaces that are modulo with respect to the system's underlying cache size.

Annul bit: This bit is used in the SPARC architecture to allow the designer or compiler to decide whether or not the delay slot instruction of a delay control transfer instruction will be executed if the conditional branch is taken. See *Section 2.3.3.4* for further information.

Cache controller: Provides cache memory access control for a 64-Kbyte direct-mapped virtual cache.

Cache lock: A mechanism that allows the system to lock all entries in the cache, supported by the CY7C604. This feature allows deterministic response times for real-time systems.

Content addressable memory (CAM): A memory that is accessed by supplying the value to be compared to the memory contents. When accessed, the CAM returns the location of the memory where the value is stored, or returns a no-match signal if the memory does not contain the value. In the case of the CY7C604/605 MMU, the value returned by the CAM array is used to address a value in the TLB RAM array, which in turn provides the physical translation value to be used by the MMU.

Copy-back mode: A style of cache management in which write accesses are written to the cache only, not to main memory.

Current window: The block of 24 *r*-registers pointed to by the current window pointer.

Current window pointer (CWP): Selects the current register window.

Delay instruction: The instruction immediately following a control transfer instruction. This instruction is always fetched, and is either executed or annulled before the control transfer takes place.

Double-precision floating point: A data type consisting of 64 bits.

Doubleword: A data type consisting of two 32-bit words used as a single 64-bit operand. A doubleword is always aligned with the most significant word at an even word boundary (bits 2–0 equal to zero). The subsequent least significant word is on an odd word boundary (bit 2 equal to one, bits 1–0 equal to zero).

Extended-precision floating point: A data type consisting of 128 bits.

***f*-register**: One of the FPU's 32 working registers.

Floating-point unit (FPU): The coprocessor that performs floating-point calculations.

Floating-point operate (FPop) instruction: Instructions that perform floating-point calculations. This category does not include loads and stores between the memory and the FPU.

Floating-point queue (FQ): A three-deep storage area for FPop instructions and their addresses while they are being executed in the FPU. Floating-point exception traps occur sometime after the floating-point instruction is issued, asynchronously to the integer unit (IU) and its pipeline. The queue supplies instruction/address pair information to the IU for the FPop that caused the exception.

Frame pointer: The pointer to the beginning of a memory stack. The frame pointer is often specific to a window, and is set from the stack pointer of the previous window.

Global registers: A block of eight registers within the register file that are always available to the IU regardless of the value of the CWP.

Halfword: A data type consisting of 16 bits.

Integer unit (IU): The main computing engine. It fetches all instructions and executes all but the FPop and CPop instructions.

MBus: The interface between a SPARC processing module and the memory subsystem.

Load/Store: The class of instructions that are either Load or Store instructions.

Load-Store: The class of instructions that are atomic (indivisible or locked) Load THEN Store. These instructions are typically used for the manipulation of multiprocessor semaphores or any other process where interruption during the process of loading a variable and storing a new value for that variable could be disastrous. The SPARC Load-Store instructions are: SWAP, SWAPA, LDSTUB, and LDSTUBA.

Next program counter (nPC): Contains the address of the next instruction to be executed, assuming no trap occurs.

Processor state register (PSR): The IU's status register.

Program counter: Contains the address of the current instruction being executed by the IU.

***r*-register**: A global register or a register in the current window of the register file.

Register window: A group of 24 working registers from the set of window r-registers (128 window registers or eight windows are available on the CY7C601/611). Register windows overlap by eight registers, causing three types of window registers: *ins, outs, and locals.* Ins are the window registers that were the outs for the previous window. Locals are specific to the register window, and are not shared. See *Section 2.2* for further information.

rd, rs1, and rs2: Instruction format fields which specify the register operands of an instruction. rd is the destination register and rs1 and rs2 are the source registers.

RISC: An acronym that stands for Reduced Instruction Set Computer.

r[rd], r[rs1], and r[rs2]: The actual r-registers specified by rd, rs1, and rs2.

Page table entry (PTE): An address mapping for a single 4-Kbyte page, a 256-Kbyte region, a 16-Mbyte region, or a 4-Gbyte region.

Page table pointer (PTP): The address pointer used to identify the beginning of a page table in memory.

Page table pointer cache (PTPC): The cache of page table pointers stored by the CY7C604/605 in order to minimize the levels of table walks required for a TLB miss. See *Section 4.1* for further details.

SPARC: An acronym that stands for Scalable Processor ARChitecture.

Stack pointer: The pointer to the next address in memory that registers are temporarily stored, typically in response to a procedure call or trap routine.

Table walk: The process of accessing levels of tables in memory to find a page table entry for a particular virtual address. Each level of the table either has a pointer to the next level of table, or has the page table entry. Upon finding a page table entry, the table walk is terminated by the MMU.

Test Access Port (TAP): The boundary scan test feature implemented on hyperSPARC.

Translation lookaside buffer (TLB): Acts as a cache for address mapping entries used by the MMU to map a virtual address to a physical address.

Virtual cache: Refers to the direct addressing of the cache by the integer unit using the virtual address bus.

Word: A data type consisting of 32 bits.

Write-through mode: A style of cache management that causes write accesses to the cache to be written through to main memory upon each write access.

A

ADD, 12-8

ADDcc, 12-9

address space identifier (ASI)
  CY7C601/611, 6-7
  CY7C604/605 signal, 8-65
  CY7C611, 6-52
  register mapping
    CY7C604/605, 8-55–8-56
    RT625, 4-46–4-47
  RT620, IMASI, 3-31
  SPARC architecture values, 2-25

ADDX, 12-10

ADDXcc, 12-11

AND, 12-12

ANDcc, 12-13

ANDN, 12-14

ANDNcc, 12-15

annul bit, 2-21, 2-28, 2-33

assembly language, description, 12-1–12-8

asynchronous fault address register (SFAR)
  CY7C604/605, 8-47
  RT625, 4-42

asynchronous fault status register (AFSR)
  CY7C604/605, 8-47
  RT625, 4-42

Atomic Load-Store. *See* Load-Store

B

BHOLD, 6-8, 6-30, 7-25

Bicc, 2-28–2-30, 2-33, 12-16

big-endian, 2-14, 2-16

Block Copy, RT625, 4-43

Block Fill, RT625, 4-43

C

cache byte write enables, RT625, 4-36

cache controller
  cache flushing, 4-34, 8-38
  cacheable/non-cacheable, 4-34, 8-39
  control signals, 4-31–4-34, 8-35–8-37
  CY7C604
    aliasing, 8-20–8-22
    cache locking, 8-22
    cache tag, 8-19
  CY7C605
    aliasing, 8-35
    bus snooping, 8-34
    cache state transitions, 8-26–8-34
    cache tag
      MPTAG, 8-24–8-25
      PVTAG, 8-24
    multiprocessing support, 8-25–8-26
  LDST cycles, 4-35, 8-40
  MC (MBus cacheable bit), 4-35, 8-39
  modes
    copy-back with write allocate, 4-19, 8-18
    RT625 cache modes, 4-18
    write-through with no write allocate, 4-19, 8-18
  read buffer, 4-33, 8-38
  RT625, 4-19–4-34
    aliasing, 4-31–4-32
    bus snooping, 4-31
    cache state transitions, 4-23–4-31
    cache tag, 4-2–4-7, 4-20
    multiprocessing support, 4-22–4-23
    signals, 4-31–4-32
  write buffer, 4-32, 8-37

cache data forwarding, RT625, 4-37

CALL, 2-7, 2-20, 2-24, 2-28, 2-32, 12-18

CBccc, 2-28, 2-29, 2-33, 12-19

CEXC, 6-12

CHOLD, 6-12, 6-49

context register (CXR)
  CY7C604/605, 8-44
  RT625, 4-39

context table pointer register (CTPR)
  CY7C604/605, 8-44
  RT625, 4-39

coprocessor interface, 6-49

CPop, 2-37–2-38, 6-29, 12-21

current window pointer (CWP), 2-4–2-5, 2-10, 2-30

D

data access PTP (DPTP)
  CY7C604/605, 8-45
  RT625, 4-40

delayed control transfer, 2-32
  instruction couples, 2-34

diagnostics
  CY7C604/605
    cache data entries, 8-53
    cache tag entries, 8-53
    TLB entries, 8-52
  RT625
    cache tag entries, 4-45
    CDU cache data entries, 4-45
    TLB entries, 4-44

E

ERROR
  CY7C601/611
    ERROR signal, 6-14
    timing example, 6-40–6-41
  RT620
    mode, 3-79, 3-100–3-102
    PERROR, 3-32
    state, 3-6
  signal, 8-65

F

f-registers, 2-7, 7-14

floating-point
  condition codes, 6-13
  coprocessor instruction flush, 6-14
  CY7C602
    BHOLD signal, 7-12
    CHOLD signal, 7-12
    denormalized numbers, 7-21
    exception cases, 7-21
    exception handling, 7-20
    FHOLD signal, 7-13
    FLUSH, 7-10
    FNULL, 7-13
    FP queue, 7-10
    FSR, 7-15
    IU-FPU interface, 7-4
    MHOLD signal, 7-12
    queue, 7-15
  double-precision, 2-18
  exception, 6-13
  exception acknowledge, 6-14
  extended-precision, 2-19
  hold, 6-13
  instruction in buffer, 6-13
  quad-precision, 2-19
  RT620, 3-15–3-24
    denormalized numbers, 3-23, 3-84
    double-precision, 3-23
    execution units, 3-23
    non-standard mode, 3-23
  RT620 FPU
    error mode, 3-101
    fp instr. pipeline, 3-75
    non-standard mode, 3-24
    queue, 3-17
    reset, 3-101
    traps, 3-82–3-101
  single-precision, 2-18
  status register (FSR), 2-12–2-14
  unit present, 6-14

flush
  instruction, 2-38, 6-14, 12-61
  RT620, 3-29–3-31

FLUSH, 6-14

FMOVs, 12-44

FMULd, 12-46

FMULs, 12-48, 12-53

FNEGs, 12-49

FNULL, 8-65

FP Queue, 7-15

FPop, 2-37, 6-29

FqTOd, 12-50

FqTOi, 12-51
FqTOs, 12-52
frame pointer, 2-4, G-2
FsMULd, 12-53
FSQRTd, 12-54
FSQRTs, 12-56
FSR, 2-12–2-14
FsTOd, 12-57
FsTOi, 12-58
FsTOq, 12-59
FSUBd, 12-60
FSUBq, 12-61
FSUBs, 12-62
FXACK, 6-14, 6-39

H

hardware interlocks
  CY7C601/611, 6-20
  RT620, 3-11–3-14

I

index tag register (ITR)
  CY7C604/605, 8-45
  RT625, 4-40

instructions
  ADD, 12-8
  ADDcc, 12-9
  ADDX, 12-10
  ADDXcc, 12-11
  AND, 12-12
  ANDcc, 12-13
  ANDN, 12-14
  ANDNcc, 12-15
  arithmetic/logical/shift, 2-27
  atomic load-store, 2-26
  Bicc, 2-28–2-30, 2-33, 12-16
  CALL, 2-7, 2-20, 2-24, 2-28, 2-32, 12-18
  CBccc, 2-28, 2-29, 2-33, 12-19
  control transfer, 2-28
  delay, 2-33
  delayed control transfer, 2-32
  formats, 2-20
  FABSs, 12-22
  FADDd, 12-23
  FADDq, 12-24
  FADDs, 12-25
  FBfcc, 2-28, 2-29, 2-33, 12-26
  FCMP, 2-29, 7-10
  FCMPd, 12-28
  FCMPEd, 12-29
  FCMPEq, 12-30
  FCMPEs, 12-31
  FCMPq, 12-32
  FCMPs, 12-33
  FDIVd, 12-34
  FDIVq, 12-35
  FDIVs, 12-36
  FdTOi, 12-38
  FdTOq, 12-39
  FdTOs, 12-40
  FEXC, 6-13, 6-39
  FHOLD, 6-13
  FINS1, FINS 2 signal, 7-8
  FiTOd, 12-41
  FiTOq, 12-42
  FiTOs, 12-43
  FsMULd, 12-53
  JMPL, 2-4, 2-7, 2-28, 2-30, 2-34, 12-63
  LD, 12-64
  LDA, 12-65
  LDC, 12-66
  LDCSR, 12-67
  LDD, 12-68
  LDDA, 12-69
  LDDC, 12-70
  LDDF, 12-71
  LDF, 12-72
  LDFSR, 12-73
  LDSB, 12-74
  LDSBA, 12-75
  LDSH, 12-76
  LDSHA, 12-77
  LDST
    CY7C604/605 operation, 8-40
    CY7C601/611 signal, 6-9
    CY7C601/611 timing example, 6-28

instructions *(continued)*
CY7C604/605 signal, 8-66
RT620, timing example, 3-41, 3-48, 3-67
LDSTUB, 2-26, 12-78
LDSTUBA, 12-79
LDUB, 12-80
LDUBA, 12-81
LDUH, 12-82
LDUHA, 12-83
load, 6-26
load and store, 2-25
mnemonics, 12-3
multiprocessing, 2-26
pipeline
CY7C601/611, 6-16–6-23
RT620, 3-58–3-78
MEXC, 8-66
MHOLD, 7-25, 8-66
MULScc, 2-11, 12-84
OR, 12-85
ORcc, 12-86
ORN, 12-87
ORNcc, 12-88
RDASR, 12-89
RDPSR, 2-10, 2-37, 12-90
RDTBR, 2-11, 2-37, 12-91
RDWIM, 2-10, 2-37, 12-92
RDY, 2-37, 12-93
register differences, RT620/CY7C601, 2-2
RESTORE, 2-4, 2-5, 2-10, 2-28, 2-32, 12-94
RETT, 2-2, 2-5, 2-10, 2-28, 2-30, 2-34, 2-55, 6-49, 12-95
SAVE, 2-4, 2-5, 2-10, 2-28, 2-32, 12-97
SDIV, 12-98
SDIVcc, 12-99
SETHI, 2-20, 2-28, 12-100
SLL, 12-101
SMUL, 12-102
SMULcc, 12-103
SNULL, 8-66
SRA, 12-104
SRL, 12-105
ST, 12-106
STA, 12-107
STB, 12-108
STBA, 12-109
STC, 12-110
STCSR, 12-111
STD, 12-112
STDA, 12-113
STDC, 12-114
STDCQ, 12-115
STDF, 12-116
STDFQ, 12-117
STF, 12-118
STFSR, 2-29, 12-119
STH, 12-120
STHA, 12-121
store, 6-27
SUB, 12-122
SUBcc, 12-123
SUBX, 12-124
SUBXcc, 12-125
SWAP, 2-26, 12-126
SWAPA, 12-127
TADDcc, 2-28, 12-128
TADDccTV, 2-28, 12-129
Ticc, 2-11, 2-28, 2-30, 12-130
TSUBcc, 2-28, 12-132
TSUBccTV, 2-28, 12-133
types, 2-25–2-38
UDIV, 12-135
UDIVcc, 12-135
UMUL, 12-136
UMULcc, 12-137
UNIMP, 2-38, 12-138
WRASR, 12-139
WRPSR, 2-10, 2-29, 2-37, 12-140
WRTBR, 2-11, 2-37, 12-141
WRWIM, 2-10, 2-37, 12-142
WRY, 2-37, 12-143
XNOR, 12-144
XNORcc, 12-145
XOR, 12-146
XORcc, 12-147

instruction access PTP register (IPTP)
CY7C604/605, 8-45
RT625, 4-40

instruction cache (RT620), 3-25–3-30

integer condition codes (icc), 2-8

integer unit control/status registers, 2-7–2-13
ICACHE control register (ICCR), 2-11
PSR, 2-8–2-10
TBR, 2-10
WIM, 2-10

Y register, 2-11
INULL, 8-65

J

JMPL, 2-4, 2-7, 2-28, 12-63
JMPL, RETT, 2-30, 2-34, 12-95,
JMPL-RETT instruction pair, 2-2

L

LD, 12-64
LDA, 12-65
LDC, 12-66
LDCSR, 12-67
LDD, 12-68
LDDA, 12-69
LDDC, 12-70
LDDF, 12-71
LDF, 12-72
LDFSR, 12-73
LDSB, 12-74
LDSBA, 12-75
LDSH, 12-76
LDSHA, 12-77
LDSTUB, 2-26, 12-78
LDSTUBA, 12-79
LDUB, 12-80
LDUBA, 12-81
LDUH, 12-82
LDUHA, 12-83
Load-Store (LDST)
  CY7C604/605 operation, 8-40
  CY7C601/611 signal, 6-9
  CY7C601/611 timing example, 6-28
  CY7C604/605 signal, 8-66

M

MBus
  burst transactions, 11-11
  data cycle(s), 11-10–11-11
  Level 1, 11-1–11-2
  Level 2, 11-2
  MAD bus, 8-67
  $\overline{\text{MAS}}$ signal, 8-68
  $\overline{\text{MBB}}$ signal, 8-68
  $\overline{\text{MBG}}$ signal, 8-68
  $\overline{\text{MBR}}$ signal, 8-68
  $\overline{\text{MERR}}$ signal, 8-68
  $\overline{\text{MIH}}$ signal, 8-68
  $\overline{\text{MRDY}}$ signal, 8-69
  $\overline{\text{MRST}}$ signal, 8-69
  $\overline{\text{MRTY}}$ signal, 8-69
  $\overline{\text{MSH}}$ signal, 8-69
  non-burst transactions, 11-11
  Relinquish and Retry, 11-12
  Retry, 11-12
  signal summary, 11-3–11-10, 11-11–11-23
  transactions
    Coherent Invalidate, 11-15–11-17
    Coherent Read, 11-14–11-17
    Coherent Read and Invalidate, 11-16–11-17
    Coherent Write and Invalidate, 11-17–11-19
    write, 11-13–11-14
memory data strobe (MDS), 8-66
memory management unit (MMU)
  CY7C604/605, 8-3
  flush, 8-16–8-17
  invalidate, 4-17–4-18
  operation modes, 4-16–4-18, 8-15–8-17
  probe, 4-18, 8-17
  RT625, 4-4
memory stack, 2-4
MEXC, 8-66
MHOLD, 7-25, 8-66
MULScc, 2-11, 12-84
multichip configuration, CY7C604/605, 8-48–8-52

O

OR, 12-85

ORcc, 12-86

ORN, 12-87

ORNcc, 12-88

P

page table entry (PTE), 4-4, 4-5, 4-12–4-13, 8-3, 8-4, 8-11–8-12

page table pointer (PTP), 4-11–4-12, 8-10–8-11

page table pointer cache (PTPC), 4-13, 8-12

pinouts, RT625, 4-56

power-on reset (POR), 8-69

processor interrupt level (PIL), 2-9

processor state register (PSR), 2-8–2-10

program counter(s), 2-8
  delayed control transfer, 2-33

R

r-register
  ins, 2-32–2-33
  locals, 2-32–2-33
  outs, 2-32–2-33
  r[0], 2-27
  RT620, 2-3–2-7
  special r-registers, 2-6–2-7

RDASR, 12-89

RDPSR, 2-10, 2-37, 12-90

RDTBR, 2-11, 2-37, 12-91

RDWIM, 2-10, 2-37, 12-92

RDY, 2-37, 12-93

register differences, RT620/CY7C601, 2-2

registers
  CY7C604/605
    CY7C604 system control register, 8-41–8-42
    CY7C605 system control register, 8-42
    asynchronous fault address (AFAR), 8-47
    asynchronous fault status (AFSR), 8-47
    context register (CXR), 8-44
    context table pointer (CTPR), 8-44
    data access PTP (DPTP), 8-45
    index tag register (ITR), 8-45
    instruction access PTP (IPTP), 8-45
    reset register (RR), 8-44
    root pointer register (RPR), 8-45
    synchronous fault address (SFAR), 8-47
    synchronous fault status (SFSR), 8-46
    TLB replacement control (TRCR), 8-46
  RT625
    asynchronous fault address (AFAR), 4-42
    asynchronous fault status (AFSR), 4-42
    context register (CXR), 4-39
    context table pointer (CTPR), 4-39
    data access PTP (DPTP), 4-40
    index tag register (ITR), 4-40
    instruction access PTP (IPTP), 4-40
    reset register (RR), 4-39
    root pointer register (RPR), 4-39
    synchronous fault address (SFAR), 4-42
    synchronous fault status (SFSR), 4-41
    system control register, 4-37–4-38
    TLB replacement control (TRCR), 4-40

reset
  CY7C601/611
    $\overline{\text{RESET}}$ signal, 6-15
    state, 6-42
  CY7C604/605, 8-54–8-98
    in multichip configuration, 8-55–8-98
    power-on (POR), 8-54–8-98
    power-on (RSTIN), 4-45–4-84
    software external (SER), 8-55–8-98
    software internal (SIR), 8-54–8-98
    watchdog (WDR), 8-54–8-98
  RT620, 3-78, 3-100–3-102
    $\overline{\text{PRST}}$ signal, 3-32
    state, 3-6–3-7
    timing example, 3-53
  RT625, 4-45–4-84
    software internal (SIR), 4-46–4-84
    watchdog (WDR), 4-46–4-84

reset register (RR)
  CY7C604/605, 8-44
  RT625, 4-39

RESTORE, 2-4, 2-5, 2-10, 2-28, 2-32, 12-94

RETT, 2-2, 2-5, 2-10, 2-28, 2-55, 6-49, 12-95

root pointer register (RPR)
 CY7C604/605, 8-45
 RT625, 4-39

S

SAVE, 2-4, 2-5, 2-10, 2-28, 2-32, 12-97

SDIV, 12-98

SDIVcc, 12-99

SETHI, 2-20, 2-28, 12-100

SLL, 12-101

SMUL, 12-102

SMULcc, 12-103

SNULL, 8-66

SPARC registers
 features, 2-2–2-15
 r-registers, 2-3–2-7
 special registers, 2-6

SRA, 12-104

SRL, 12-105

ST, 12-106

STA, 12-107

stack pointer, 2-4, G-3

STBA, 12-109

STC, 12-110

STCSR, 12-111

STD, 12-112

STDA, 12-113

STDC, 12-114

STDCQ, 12-115

STDF, 12-116

STDFQ, 12-117

STF, 12-118

STFSR, 2-29, 12-119

STH, 12-120

STHA, 12-121

SUB, 12-122

SUBcc, 12-123

SUBX, 12-124

SUBXcc, 12-125

supervisor
 CY7C601/611, reset, 6-40
 RT620
  ICACHE privilege match, 3-25–3-27
  reset, 3-78
  trap operation, 3-79
 mode, 2-2

SWAP, 2-26, 12-126

SWAPA, 12-127

synchronous fault address register (SFAR)
 CY7C604/605, 8-47
 RT625, 4-42

synchronous fault status register (SFSR)
 CY7C604/605, 8-46
 RT625, 4-41

synchronous faults
 CY7C604/605, 8-56–8-64
 RT625, 4-48–4-56
  cases, 4-51–4-56

system control register (SCR)
 CY7C605, 8-42–8-44
 RT625, 4-37–4-38, 8-41–8-42

T

table walk, 4-8–4-11, 8-9–8-10

TADDcc, 2-28, 12-128

TADDccTV, 2-28, 12-129

tagged arithmetic, 2-28

tagged data, 2-15, 2-28

TBR, 2-7, 2-10, 2-30, 2-54, 6-47

Ticc, 2-11, 2-28, 2-30, 12-130

Test Access Port (TAP)
 supporting signals, 4-61

Test Access Port (TAP) *(continued)*
MBus SCAN Test Feature, 10-3

TLB replacement control register (TRCR)
CY7C604/605, 8-46
RT625, 4-39

translation lookaside buffer (TLB)
CY7C604/605, 8-4
entries, 4-8, 4-44, 8-7, 8-52
locking, 4-8, 8-7
look-up, 4-7, 8-6
sections, 4-4
table walk, 4-8–4-11, 8-9–8-10

trap
addressing, 2-54, 6-47
CY7C601/611, 6-42–6-49
asynchronous, 6-42
interrupts, 6-45–6-46
priority, 6-48
reset, 6-42
synchronous, 6-42, 6-43–6-44
timing example, 6-22
CY7C602, floating-point, 7-20, 7-21
floating-point, 3-83–3-102
FP/CP, 2-53, 6-46
IEEE exceptions, 7-21
interrupts, 2-53, 6-39
operation, 2-54, 6-47
RT620, 3-77–3-102
types, 2-54

trap base register (TBR), 2-7, 2-10, 2-30, 2-54, 6-47

TSUBcc, 2-28, 12-132

TSUBccTV, 2-28, 12-132

U

UDIV, 12-134

UDIVcc, 12-135

UMUL, 12-136

UMULcc, 12-137

UNIMP, 2-38, 12-138

user, mode, 2-2

W

window invalid mask (WIM), 2-4, 2-5, 2-7, 2-10, 2-32

window overflow and underflow, 2-4
see also window invalid mask (WIM)

WRASR, 12-139

WRPSR, 2-10, 2-29, 2-37, 12-140

WRTBR, 2-11, 2-37, 12-141

WRWIM, 2-10, 2-37, 12-142

WRY, 2-37, 12-143

X

XNOR, 12-144

XNORcc, 12-145

XOR, 12-146

XORcc, 12-147

Y

Y register, 2-11

5316 Highway 290 West, Suite 500, Austin, Texas 78735
TEL (512) 892-7802 FAX (512) 892-3036