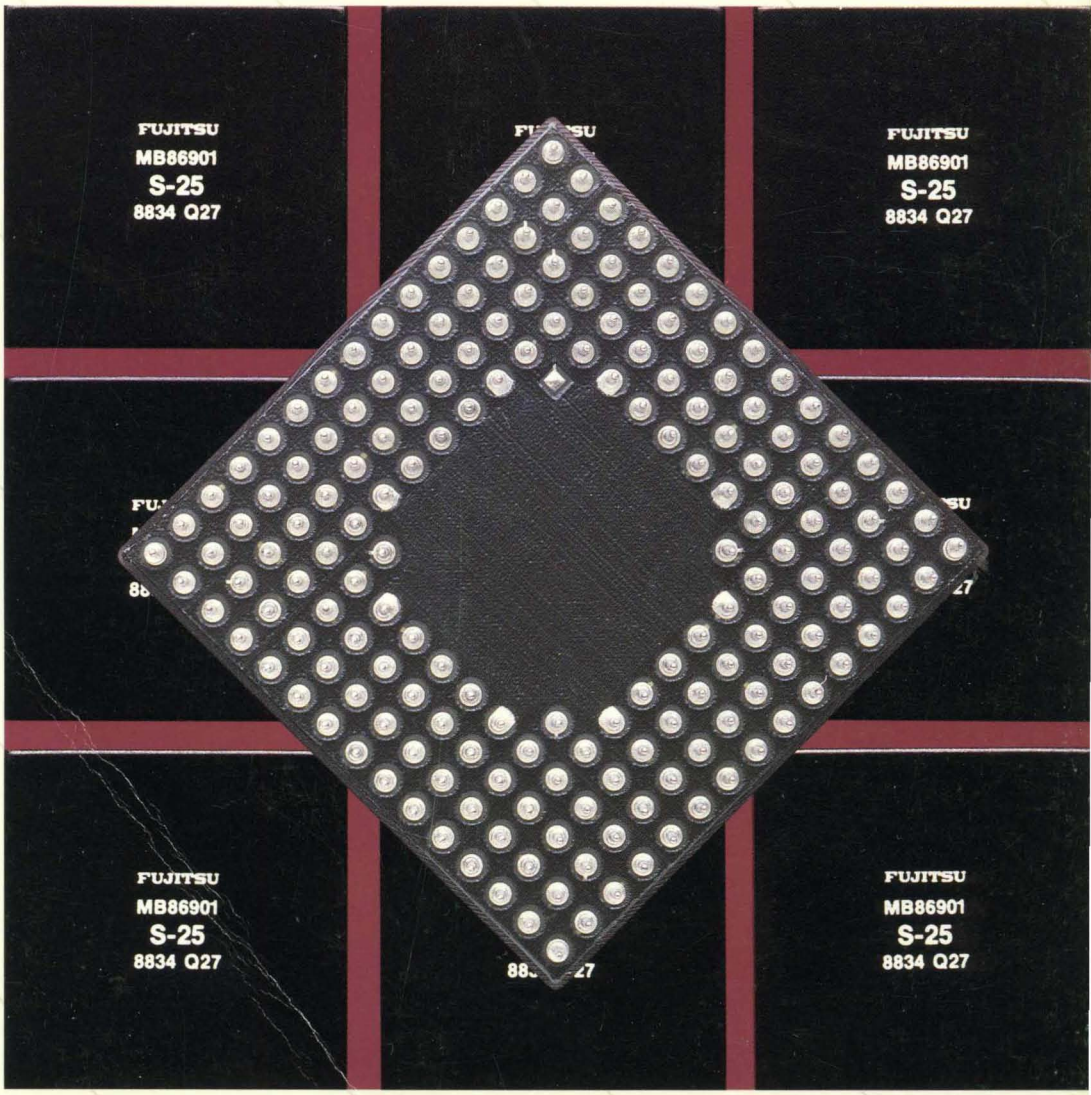
September 1988

Product Description

SPARC™ MB86901 (S-25)

High Performance 32-Bit RISC Processor



FUJITSU

SPARC™ MB86901 (S–25) High Performance 32-Bit RISC Processor

Product Description

Fujitsu Limited

Fujitsu Microelectronics, Inc.

Fujitsu Mikroelektronik GmbH

Fujitsu Microelectronics Pacific Asia Ltd.

Circuit diagrams utilizing Fujitsu products are included as a means of illustrating typical semiconductor applications. Consequently, complete information sufficient for construction purposes is not necessarily given.

The information contained in this document has been carefully checked and is believed to be entirely accurate. However, Fujitsu Limited and Fujitsu Microelectronics, Inc. assume no responsibility for inaccuracies.

The information contained in this document does not convey any license under the copyrights, patent rights or trademarks claimed and owned by Fujitsu Limited or its subsidiaries.

Fujitsu Limited and its subsidiaries reserve the right to change products or specifications without notice.

This document is published by the marketing department of Fujitsu Microelectronics, Inc.'s Advanced Products Division, 50 Rio Robles, Bldg. 3, San Jose, California, U.S.A. 95134-1804

Printed in the U.S.A.

SPARC is a trademark of Sun Microsystems, Inc.
VAX is a trademark of Digital Equipment Corp.

Table of Contents

List of Figures

List of Tables

FUJITSU

High Performance 32-Bit RISC Processor SPARC™

MB86901

FEATURES

- Architecture supports scalability towards faster technologies
- 15 times VAX™ 11/780 equivalent MIPS typical performance
- 25 MHz (40 ns/cycle) operating frequency
- Single-cycle execution for majority of instructions
- Separate 32-bit address and data buses
- 120 registers organized into seven register windows and eight global registers
- Load/store architecture
- 100 MBytes/sec. maximum memory bandwidth
- 256 alternate address spaces, each four gigabytes
- Address presentation which supports high-performance cache
- Three-operand instruction format
- Tagged operands which support AI
- Floating point interface
- Concurrent floating point operation with MB86911
- 4-stage instruction pipeline
- Delayed branch handling
- 179-pin plastic pin grid array packaging
- High-level language support

The Fujitsu MB86901 has been designed for speed. It is a high-performance, 32-bit RISC (Reduced Instruction Set Computer) processor which can be used in a wide variety of system applications including workstations, minicomputers, and real-time systems.

The MB86901 features a high-bandwidth, 32-bit bus interface with separate data and address buses, an optimized four-stage instruction pipeline, and an execution unit comprised of a CPU, a barrel shifter, two data aligners, and a three-port register file consisting of 120 registers. A floating point interface supports concurrent floating point instruction execution.

1. Introduction

The Fujitsu MB86901 SPARC is a high-performance, 32-bit, RISC-architecture processor which executes with 25 MIPS peak, and 15 MIPS sustained performance.

The MB86901 instruction set is streamlined and hardwired for fast execution, with the majority of instructions executing in a single cycle. The processor features a four-stage instruction pipeline which has been designed to handle data interlocks, an optimized branch handler for efficient control transfers, and a bus interface that supports single-cycle memory accesses.

The processor execution unit consists of a three-port ALU, a data aligner, and a barrel shifter. These units provide fast arithmetic, logical, and shift operations. A separate Address Generation Unit calculates load/store and branch addresses, enhancing processor performance by allowing the ALU to operate concurrently on data.

An internal register file comprised of 120 registers organized into seven windows minimizes accesses to memory during procedure linkages and facilitates passing of parameters and assignment of variables. By reducing external memory accesses with their inherent cycle overhead, the windowed register file significantly improves SPARC throughput.

Separate data and address buses allow fast memory and I/O accesses without the delay and design difficulties inherent in multiplexed buses. The 32-bit data bus supports single cycle data transfers. The 32-bit address bus allows direct access to 4 gigabytes of address space. Timing of the address presentation supports a high-performance cache. A separate floating point port directly interfaces to the MB86911 RISC Floating Point Controller which, together with the TI SN74ACT8847 floating point chip, allows concurrent floating point instruction execution.

These features combine to give the MB86901 superior speed, power, and flexibility which can be easily utilized by the designer of high performance systems.

2. CPU

The MB86901 processor is comprised of four primary functional units: the Control Unit, the Execution Unit, the Address Generation Unit, and the Register File. These units are supported by various registers and interface logic, as shown in Figure 2.1.

Three software-accessible registers configure and control operation of the processor. These are the Processor State Register (PSR), the Window Invalid Mask Register (WIM), and the Trap Base Register (TBR).

A fourth software-accessible register, the Y register, supports 32 × 32-bit multiplication with 64-bit result.

2.1. Instruction Pipeline

The MB86901 uses a four-stage instruction pipeline to achieve an instruction execution rate approaching one instruction per clock cycle. A "single cycle" instruction, i.e. one which **occupies each stage of the pipeline for only one clock cycle**, does not **complete** in one cycle. It enters the pipeline and completes four cycles later. During these four cycles, three more instructions may enter the pipeline. Once the pipeline is full, it is possible for one (single cycle) instruction to enter and one (single cycle) instruction to exit the pipeline every clock cycle. The stages of the MB86901 instruction pipeline are the Fetch Stage, the Decode Stage, the Execute Stage, and the Write Stage. Each stage utilizes one or more corresponding instruction registers, as shown in Figure 2.2. It should be pointed out that in the most general sense, instruction decoding is not confined exclusively to the Decode Stage—in each stage of the pipeline, the processor determines how the associated resources are to be utilized.

2.1.1. Fetch Stage

During the Fetch Stage, the processor outputs an instruction address, inputs the instruction via the data bus, and transfers it to the Instruction (Decode) Register to be decoded. In the event that this register cannot accept the instruction immediately because of a pipeline delay—possibly due to a preceding multi-cycle instruction being executed—the instruction is queued in the Instruction Buffer, which holds instructions until pipeline forwarding recommences. These buffer registers are used only as necessary to keep the pipeline full and fully utilize the external bus bandwidth.

2.1.2. Decode Stage

During the Decode Stage, the processor computes the register file addresses of required source oper-

Figure 2.1 Block Diagram

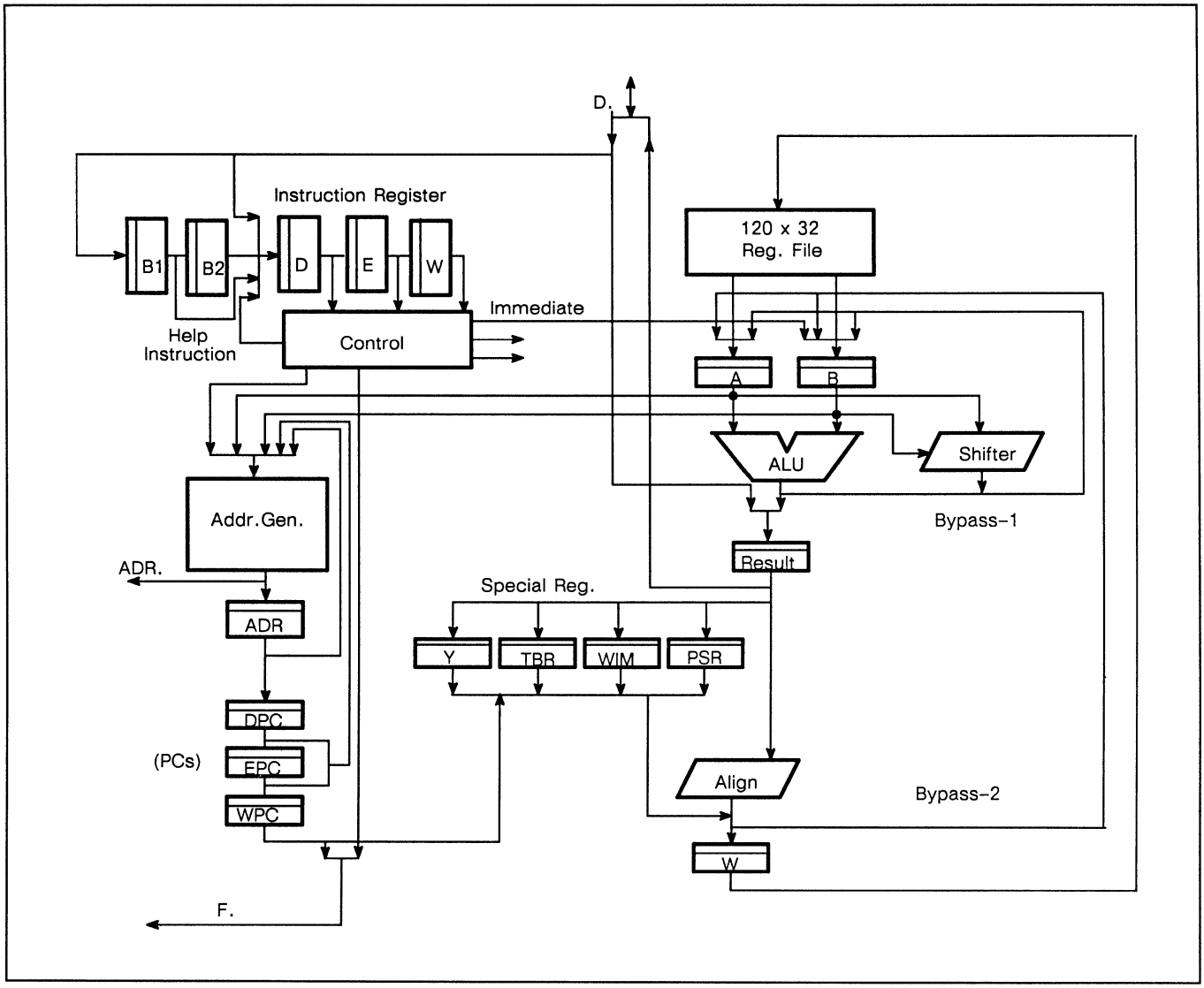
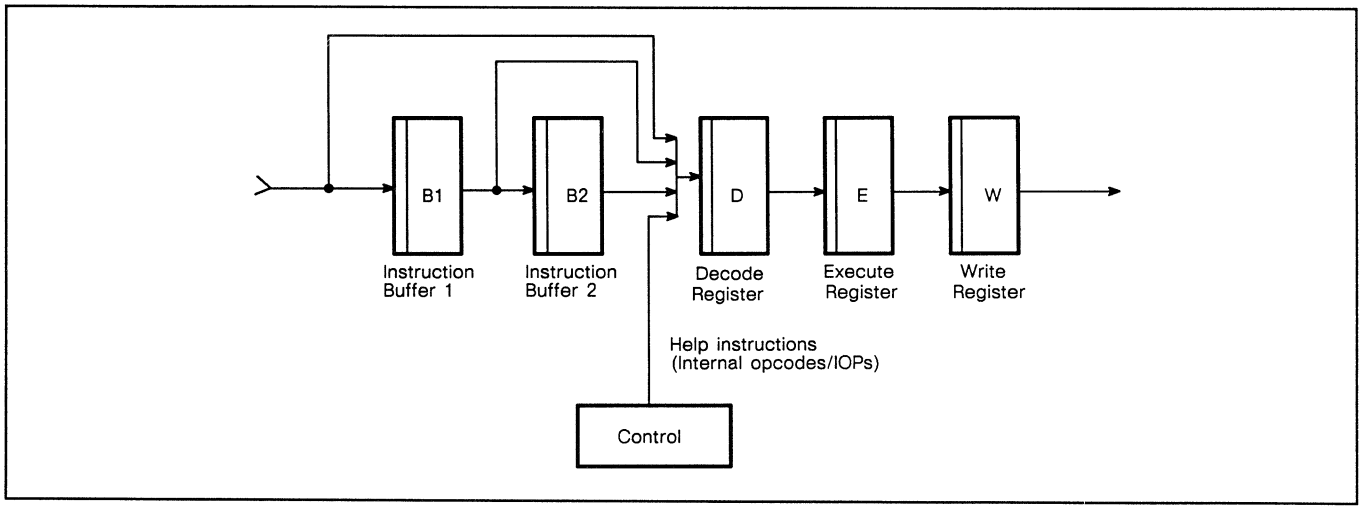


Figure 2.2 Pipeline Registers



ands, reads the source operands from the register file, and computes the next instruction address. Source operands are available to the processor for use during instruction execution.

2.1.3. Execute Stage

During the Execute Stage, arithmetic and logical operations are performed, and the results are saved in (the processor's) temporary registers. Integer arithmetic, logical, and shift operations are executed in the processor's integer ALU; floating point operations are executed in a separate floating point co-processor.

2.1.4. Write Stage

During the Write Stage, the processor stores the operation results in the destination register. This update of the register file can occur only if no traps or exceptions have occurred during execution. If instruction execution results in a trap, the write to the register file is aborted.

Note that all operands are read from the processor register file, and all execution results are written into the register file. This **Load/Store** architecture, which is characteristic of RISC processors in general, dictates that data transfers between the processor and memory occur only through the use of Load and Store instructions.

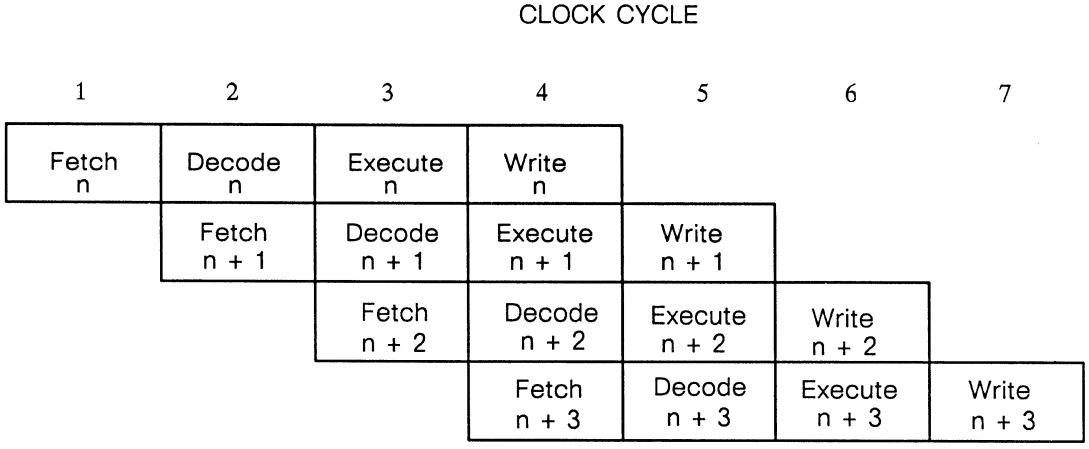
2.1.5. Pipeline Examples

A peak instruction execution rate of one instruction per cycle is achieved by the processor when a sequence of single-cycle instructions—those which occupy each pipeline stage for only one clock cycle (see Table 3.2)—passes through the pipeline.

Figure 2.3 shows single-cycle instructions, none of which require memory access, moving through the instruction pipeline. Each instruction passes through each of the four pipeline stages in one cycle. Note that the instructions are fetched during consecutive clock cycles.

Multiple-cycle instructions and instructions which access memory require more than four cycles to pass through the pipeline. Figure 2.4 shows a pipeline example in which the first instruction, a 2-cycle Load instruction fetched during clock cycle #1, requires one extra cycle to complete. Extra cycles are

Figure 2.3 Single-cycle Instruction Pipeline Progression



required when the processor needs to use the bus for something other than fetching the next instruction. These extra cycles of multi-cycle instructions are actually generated by internal opcodes (IOPs) which are inserted into the pipeline as needed (see Figure 2.4.1 in which a store instruction requires 2 extra cycles). A two-stage Instruction Buffer is used to hold instructions prior to decoding, so that the processor may continue to fetch additional instructions while internal opcodes are injected into the pipeline (see Figure 2.2). Only two stages are required because a maximum of two extra cycles will elapse between the issue of a multicycle instruction and the presentation of an address for bus access.

With pipelined execution, it is possible that an instruction may try to use the contents of a register that is in the process of being updated by a previous instruction. Special bypass paths in the MB86901 make the correct data available to following instructions for all internal register to register operations (see Figure 2.1), but do not solve the problem for loads to the registers from external memory. Interlock hardware prevents an instruction following a Load instruction from reading the register being loaded until the load is complete. This is accomplished by the processor generating a one-cycle delay as shown in Figure 2.4.2.

While this operation is transparent to software, compilers and assembly language programmers should avoid Loads followed immediately by instructions using the register contents.

2.1.6. Delayed Control Transfers

Instructions are fetched and executed from sequential memory locations until a control transfer occurs. A control transfer results from execution of a Branch, Jump, Call, or Return From Trap instruction or from a trap.

Figure 2.4 Two-cycle Instruction Pipeline Progression

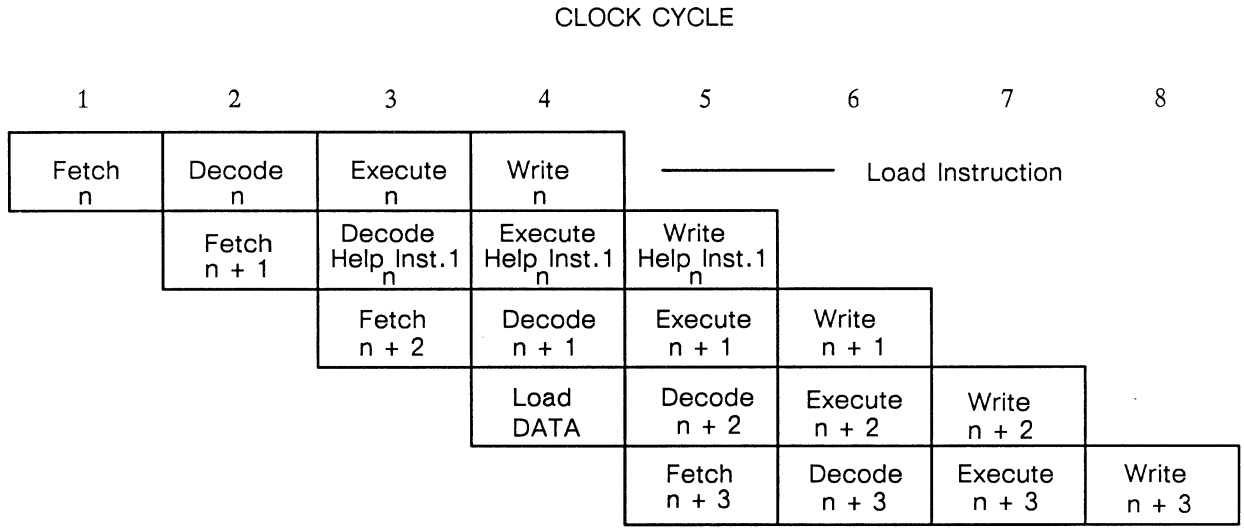


Figure 2.4.1 Three-cycle Instruction Pipeline Progression

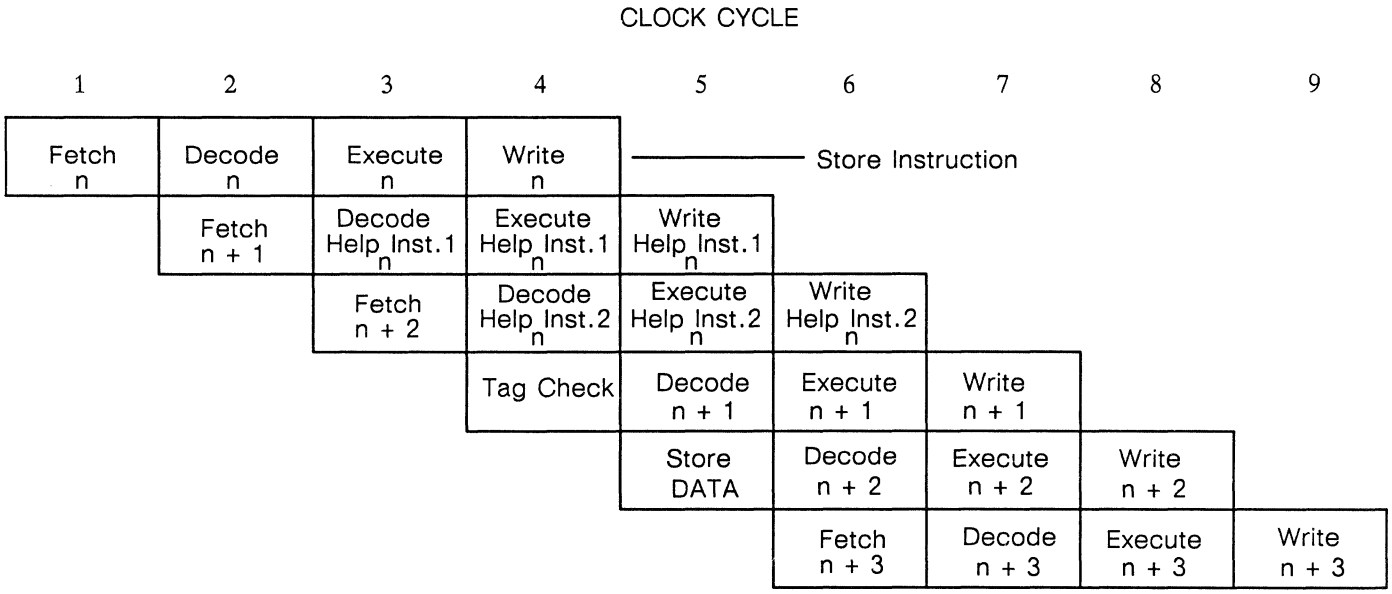
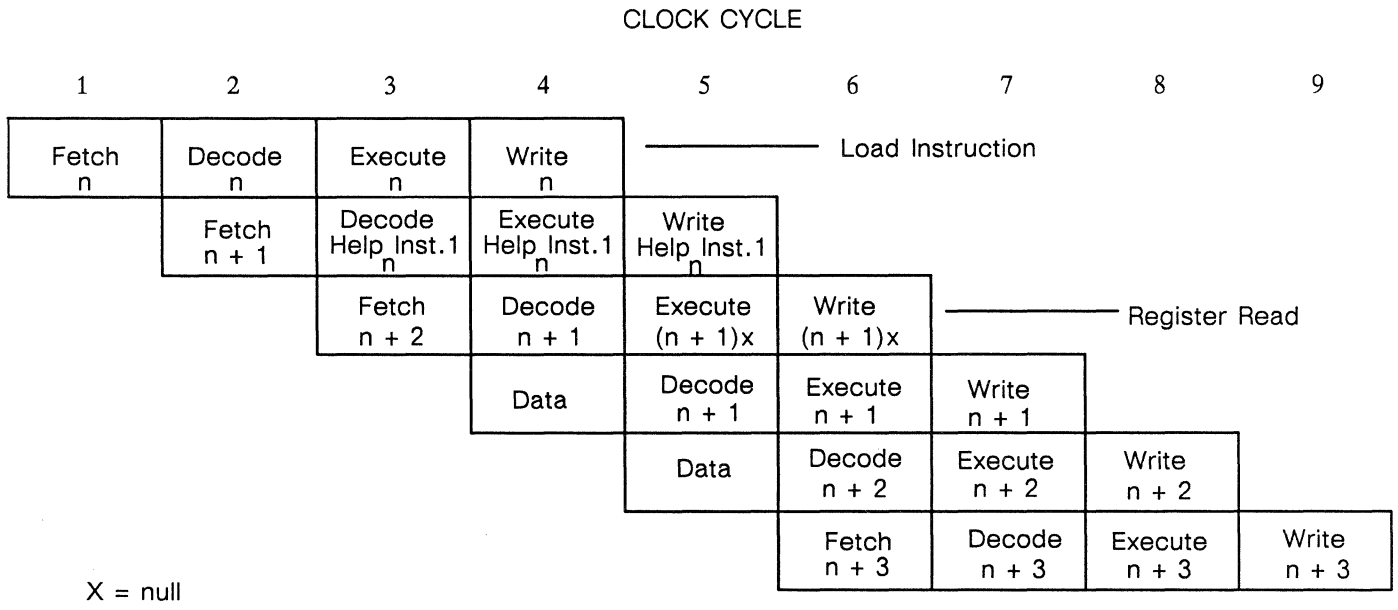


Figure 2.4.2 Illustration of Register Interlock



Traditional processors execute the first instruction of a branch target immediately after executing the instruction which causes the transfer. However, this method of control transfer is not very efficient in pipelined RISC architectures, since instructions which are in the pipeline when a control transfer occurs must be "flushed". Thus, instructions which have been fetched after a control transfer instruction would normally be discarded, and the pipeline "refilled" with instructions from the branch target—resulting in wasted cycles.

The MB86901, however, optimizes instruction pipeline utilization by providing delayed control transfer for the Branch, Call, Jump and Link, and Return From Trap instructions. The processor always fetches the instruction, called a **delay instruction**, which follows a delayed control transfer instruction. This delayed branch instruction feature of the MB86901 allows branches (whether taken or not) to occur without causing any flushes of the pipeline as shown in Figure 2.4.3a. If the compiler or programmer cannot place an appropriate instruction—i.e. one which is suitable whether the branch is taken or not—in the delay slot, the instruction fetched may be annulled as shown in Figure 2.4.3b. (See discussion of Annul Bit under Section 3.3.1.) Figure 2.8 in Section 2.5.3 shows an example of delay instruction execution.

Whenever the processor pipeline is frozen as the result of an externally generated hold input such as MHOLD or BHOLD, the pipeline stalls in the execute phase of the instruction which caused the hold as shown in Figure 2.4.4.

Pending traps, either synchronous or asynchronous, are prioritized and taken during the write phase of each instruction as shown in Figure 2.4.5. Instructions in the pipeline are annulled following the instruction which trapped.

2.2. Execution Unit

The MB86901 Execution Unit includes a fast, 32-bit carry-look-ahead integer Arithmetic Logic Unit (ALU), a 32-bit barrel shifter, and a Data Aligner as shown in Figure 2.1.

The ALU executes all processor arithmetic and logic operations in one cycle, and sets the processor Integer Condition Code flags (icc<23:20> in the Processor State Register, see Section 2.5.2.1) according to operation results. It has two input ports, and one output port.

The ALU receives inputs from the A and B Operand registers. By means of multiplexers, these registers may receive source operands from the register file or pipeline bypasses or instruction immediate operands from instruction pipeline registers. The ALU result is captured in the Result register, from which data is transferred to the register file via the Data Aligner.

Two data bypasses are used to handle data dependencies between consecutive instructions (see Figure 2.1).

One bypass transfers the output of the ALU directly to the Operand registers, bypassing the Result register. This bypass is used when a source operand of an instruction depends on the result of the previous instruction.

The second bypass transfers the output of the Result register directly to the Operand registers. This bypass is used when an instruction source operand depends on the result of the instruction two cycles previous.

The Data Aligner is used during Load instruction execution to align bytes and halfwords transferred to the register file.

Closely related to the ALU is the barrel shifter which shifts, in one cycle, 32-bit data by the count specified by instructions, and is used during Store instruction execution to align bytes and halfwords transferred to the system data bus.

2.3. Address Generation Unit

The Address Generation Unit contains logic which calculates addresses according to instruction format and address mode selection, relieving the integer ALU of the task. The unit calculates addresses utilizing instruction displacement data for Format 1 instructions, immediate data for Format 2a instructions, displacement data for Format 2b instructions, register file data or register file and immediate data, respectively, for Format 3a and Format 3b instructions, and Trap Base Register data for trap address calculation (see Section 3 for a detailed discussion of Instruction Formats).

Figure 2.4.3a Delay Instruction Executed During Branch Taken (a=0)

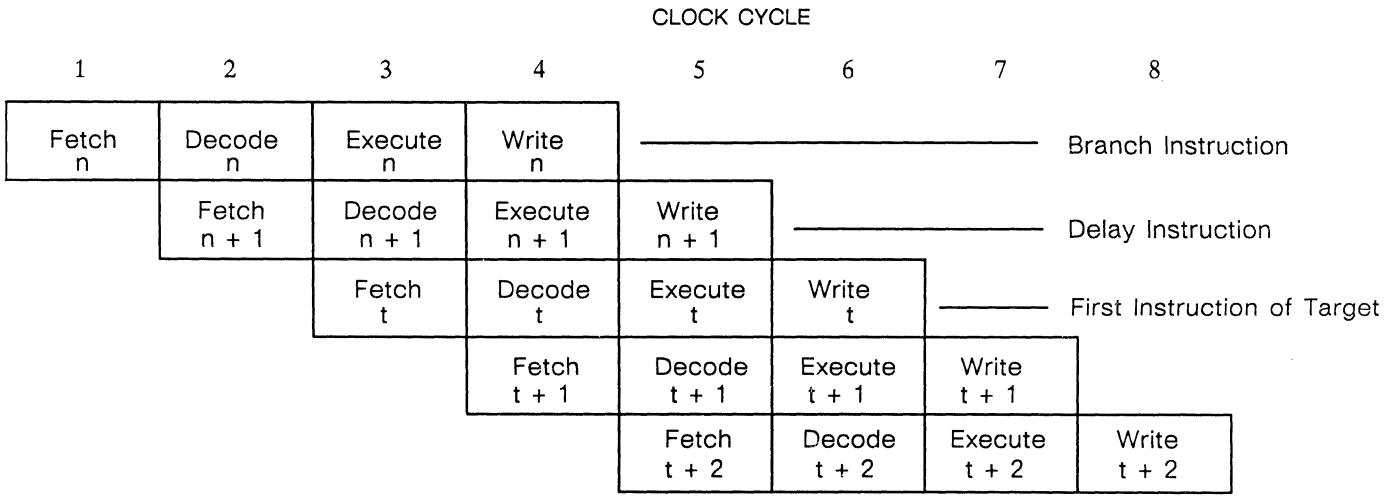


Figure 2.4.3b Delay Instruction Annulled (a=1) During Branch Not Taken

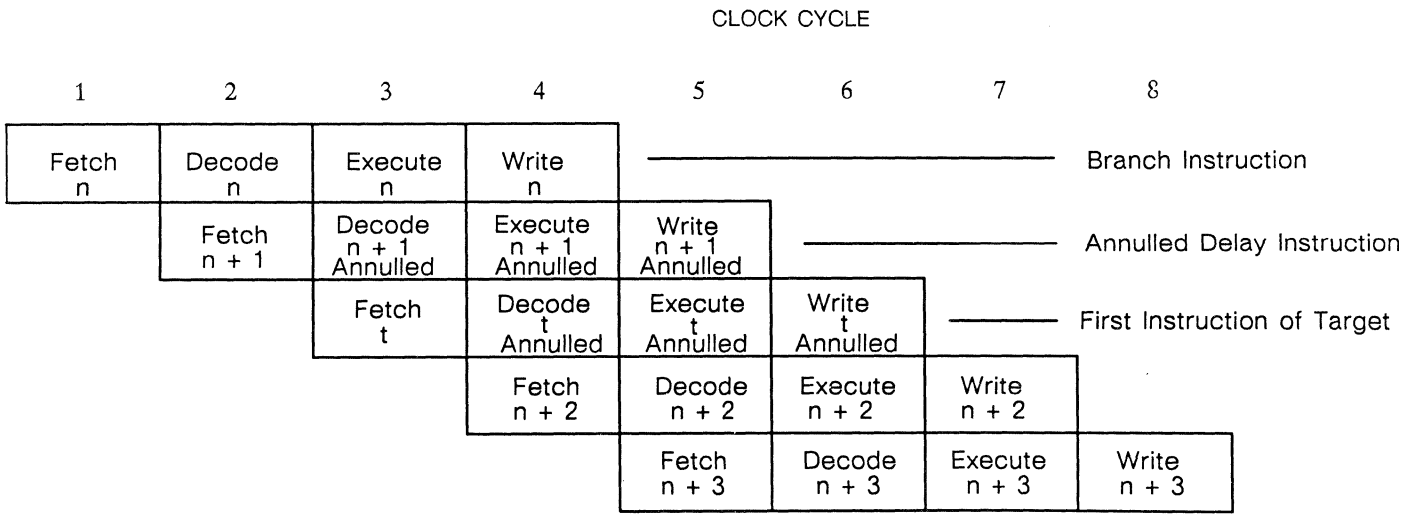
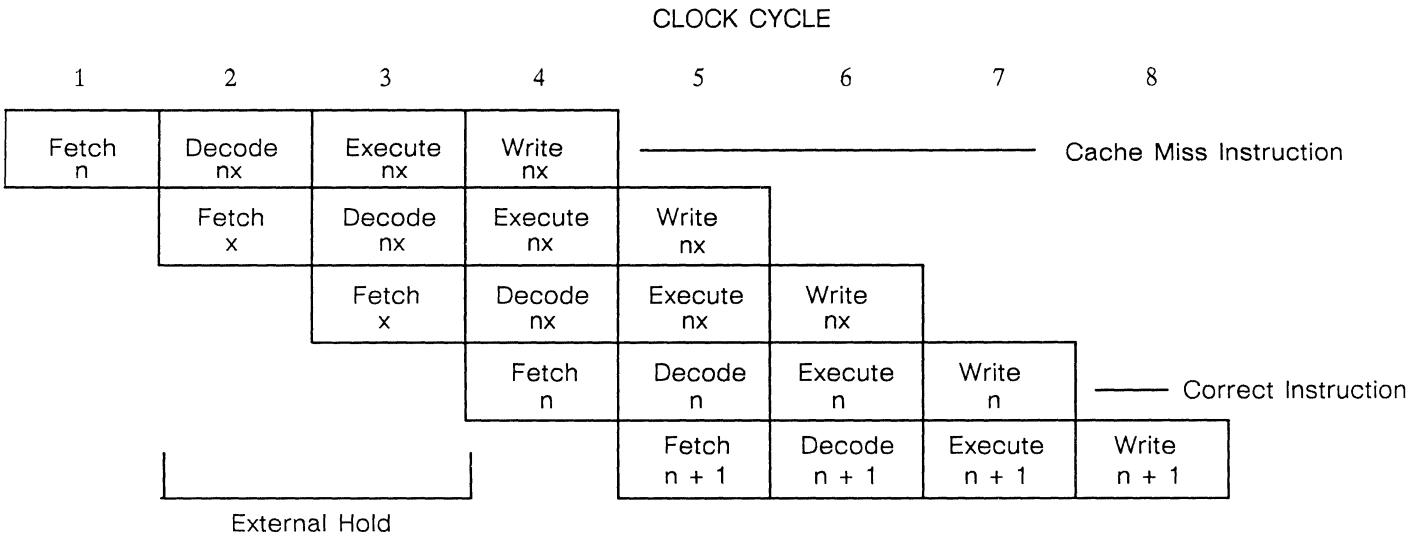
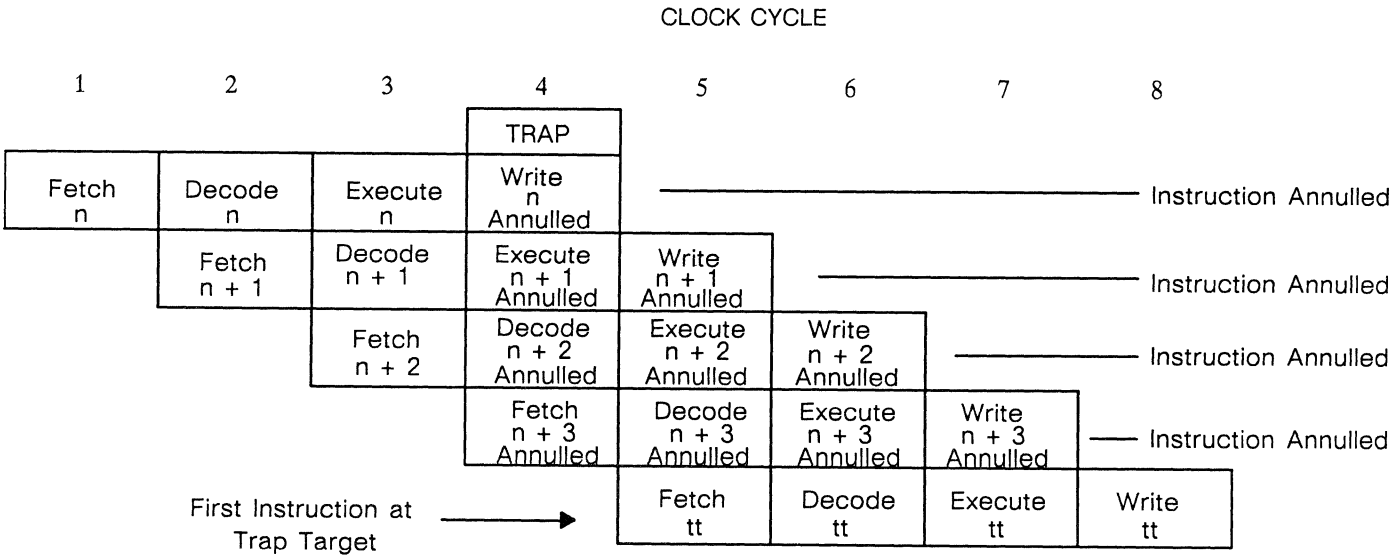


Figure 2.4.4 Instruction Fetch (Cache Miss)



x = null

Figure 2.4.5 Trap Handling in Pipeline



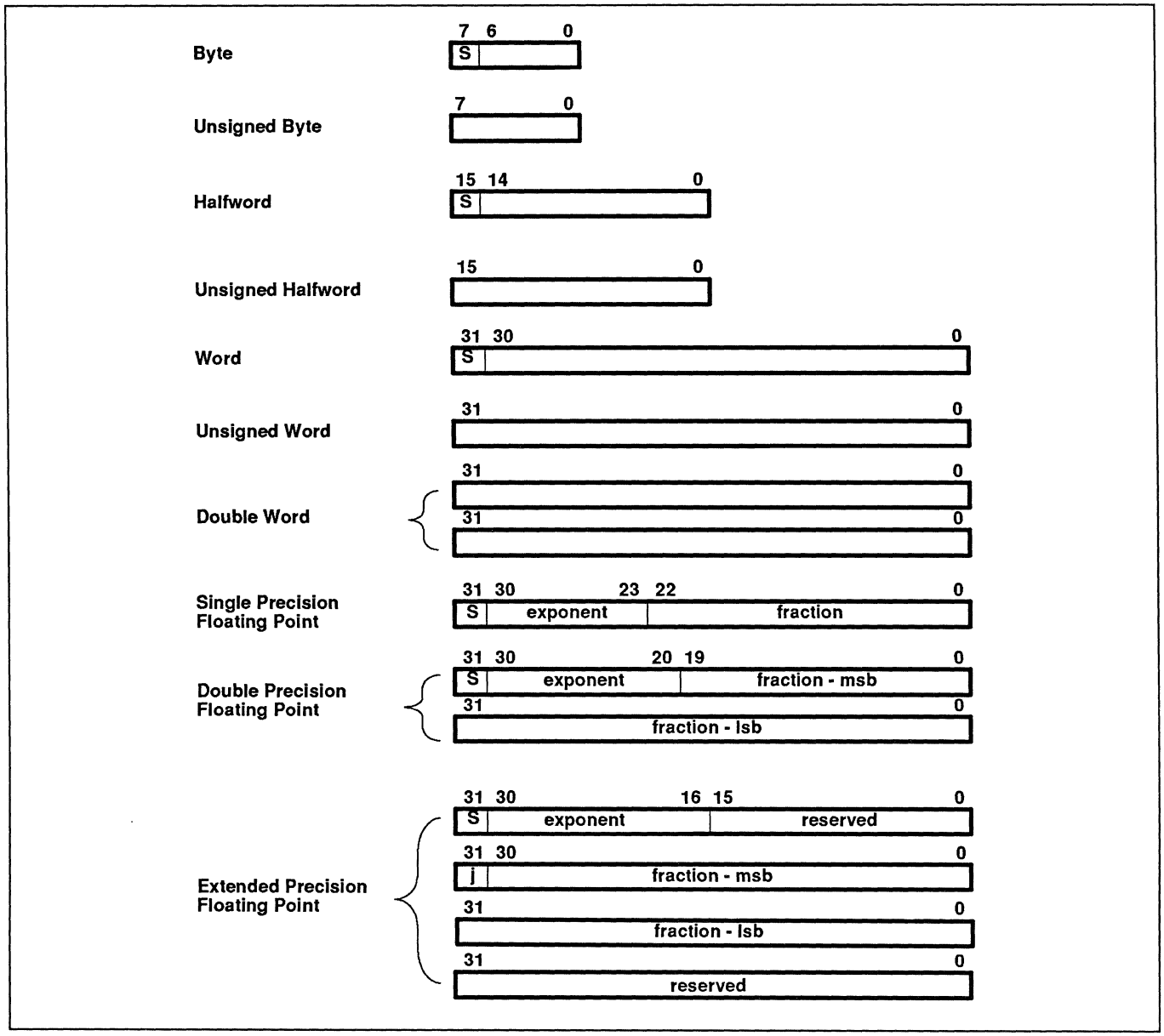
Included in the Address Generation Unit are four program counter registers. Two of the registers, The Program Counter and Next Program Counter registers, are saved during trap servicing to allow resumption of program execution after traps are serviced.

2.4. Address Space Organization

The MB86901 processor is capable of supporting up to 256 address spaces of 4 gigabytes each. The external system distinguishes these spaces by means of an address space identifier (asi) generated by the processor. These spaces can include system control registers, main memory, etc. The SPARC architecture defines four address spaces and their asi values as follows:

Address Space Identifier (asi)	Address Space
00001000	User Instruction
00001001	Supervisor Instruction
00001010	User Data
00001011	Supervisor Data

Figure 2.5 Data Types



These four asi values indicate to the external system whether the processor is in user or supervisor mode, as indicated by the Processor State Register (see Section 2.5.2.1), and whether the access is an instruction or data reference. Load/store instructions normally generate an asi of either 00001010 (user mode) or 00001011 (supervisor mode) for data access. However, load from alternate space and store into alternate space instructions, which are privileged instructions executable only in supervisor mode, use the asi value supplied by the instruction itself (see Section 3.1). This asi value may be one of the remaining 252 which are implementation definable.

Address spaces are selected by the ASI<7:0> bus interface signals as follows:

ASI<7:0>	Address Space
0–7	Implementation Definable
8	User Instruction Space
9	Supervisor Instruction Space
10	User Data Space
11	Supervisor Data Space
12–255	Implementation Definable

2.4.1. Processor Data Types

The MB86901 processor supports seven integer and three ANSI/IEEE 754-1985 standard floating point data types, as shown in Figure 2.5. The integer data types include byte, unsigned byte, halfword, unsigned halfword, word, unsigned word, and double word. Floating point types include single, double, and extended precision.

The floating point double type includes two subfields: 1) the **double-e**, which contains the sign, exponent, and high-order fraction, and 2) the **double-f**, which includes the low order fraction. The floating point extended type includes four subfields: 1) the **extended-e**, which contains the sign and exponent, 2) the **extended-f**, which contains the integer part of the mantissa, and the high order part of the fraction, 3) the **extended-f-low**, which contains the low-order fraction, and 4) the **extended-u**, which is unused.

Tagged data for arithmetic operations on dynamically allocated variables is also supported. This type of data is useful in Lisp and Smalltalk programming environments.

2.4.2. Addressing Conventions

Load and store instructions follow the big endian addressing convention:

Bytes

For load and store byte instructions, increasing the address means decreasing the significance of the byte within the word. The most significant byte (MSB) of a word is accessed when address bits <1:0> = 0, and the least significant byte is accessed when <1:0> = 3.

Halfwords

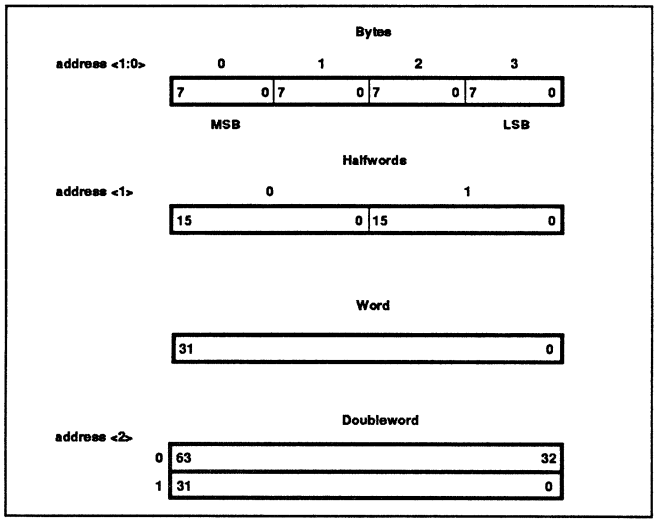
For load and store halfword instructions, the most significant halfword is accessed when address bit <1> = 0, and the least significant halfword is accessed when <1> = 1.

Doublewords

For load and store doubleword instructions, the most significant word is accessed when address bit <2> = 0, and the least significant word is accessed when <2> = 1.

In general, the address of a doubleword, word, or halfword is the address of its most significant byte. A doubleword datum is located at a doubleword address, which must be evenly divisible by eight. A word datum is located at a word address, which must be evenly divisible by four. A halfword datum is located at a halfword address, which must be evenly divisible by two. If a doubleword, word, or halfword load or store instruction generates an improperly aligned address, a "memory address not aligned" trap occurs.

Figure 2.5.1 Addressing Conventions



2.5. Registers

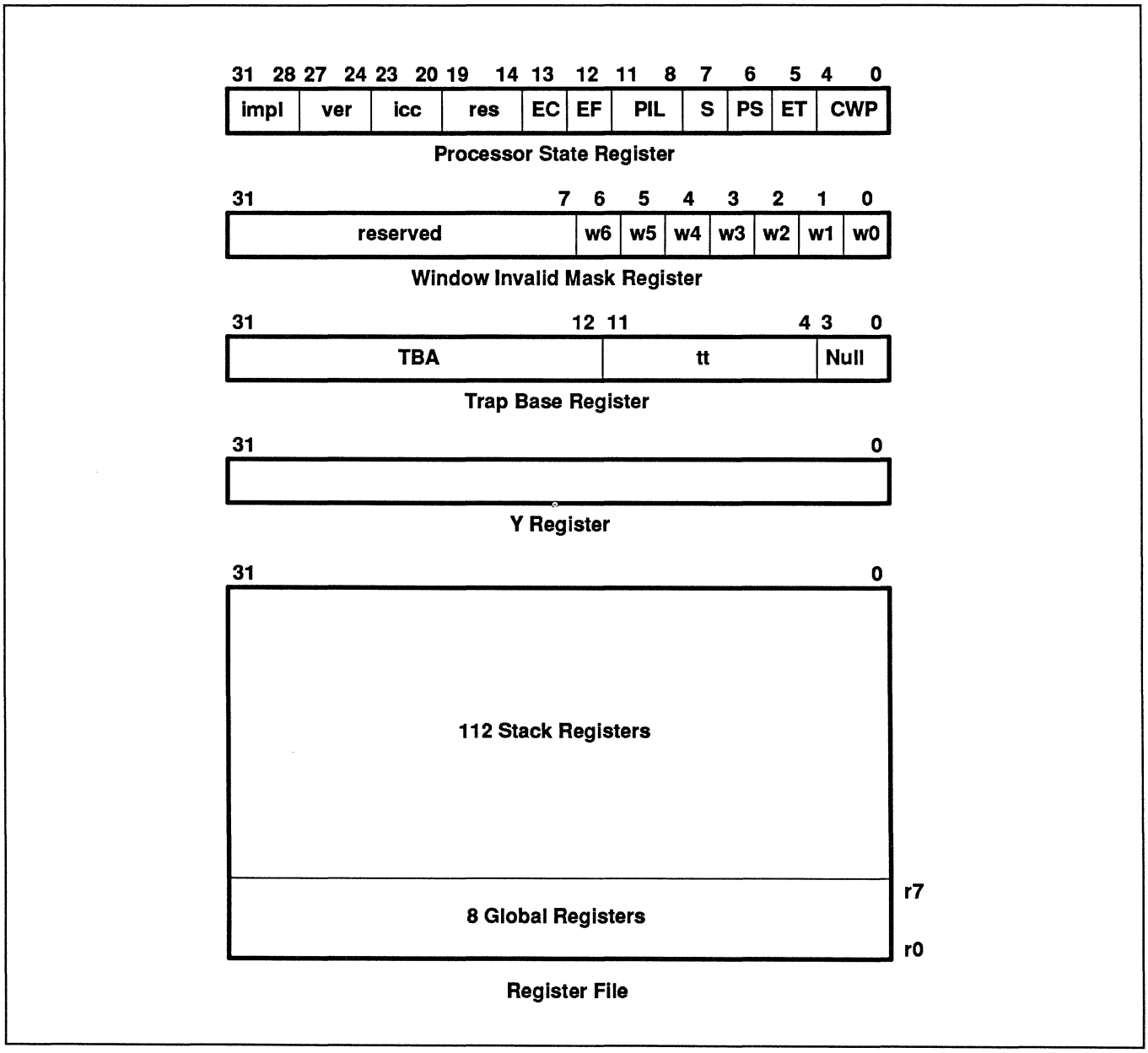
The MB86901 (IU) features two types of registers: working registers (called **r** registers), which are used for normal operations, and control/status registers, which keep track of and control the state of the IU. All r registers are 32 bits wide. They are divided into eight global registers and a number of overlapping blocks called windows. Each window contains 24 registers. Because of the overlap, 16 registers are counted for each window. Therefore, with 7 windows and eight global registers, the MB86901 has a total of 120 working registers.

The IU's control/status registers are all 32-bit read/write registers unless specified otherwise. They include the program counters (PC and nPC), the Processor State Register (PSR), the Window Invalid Mask Register (WIM), the Trap Base Register (TBR), and the multiply-step (Y) register. These registers are discussed in detail in Section 2.5.2, and are shown in Figure 2.6.

2.5.1. Register File

The MB86901 register file contains 120, 32-bit registers.

Figure 2.6 Programming Model

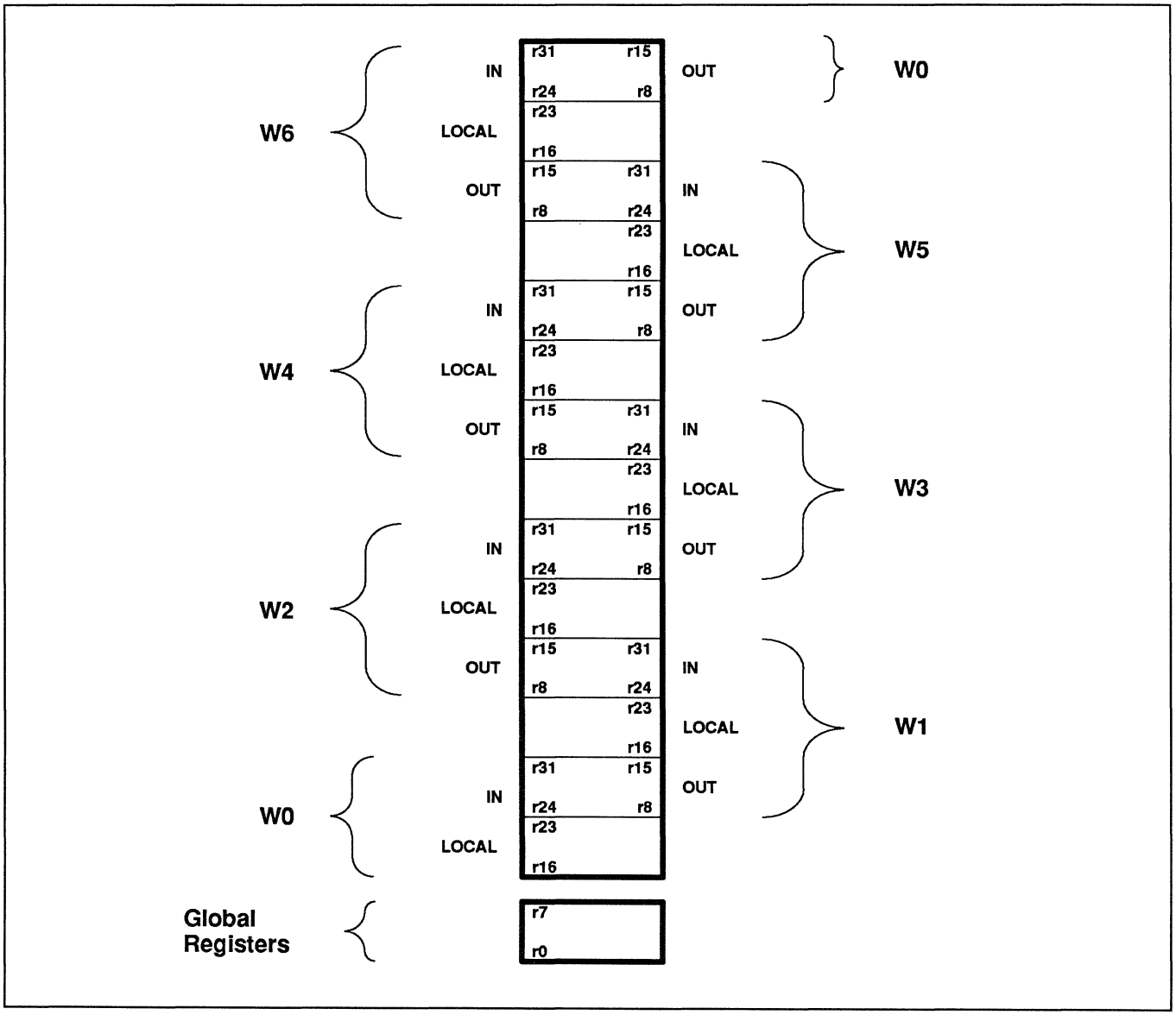


Eight of the registers, r0–r7, are global general-purpose registers which are accessible at all times. 112 of the registers are grouped into seven windows which overlap to form a circular stack, with only one window, the **active** window, accessible at any time.

Figures 2.7 and 2.7.1 show a representation of the register windows, which are numbered contiguously from 0 to 6. In general, the number of windows (NWINDOWS) could range from 2 to 32, depending on the implementation. Implemented windows must be numbered contiguously from 0 to NWINDOWS – 1. The windows are addressed by the CWP (Current Window Pointer), a field of the Processor State Register. The **active** window is defined as the one currently pointed to by the CWP. At most 6 (NWINDOWS – 1, in general) are available to user code since one window must be available for trap handlers.

At any given time, there are 32 working registers available to a given process. Registers r0–r7 are global registers which are equally available from all windows. The active window provides an additional 24 registers designated r8–r31. These registers are segmented into OUT registers (r8–r15), LOCAL registers (r16–r23), and IN registers (r24–r31).

Figure 2.7 Register File with Windows



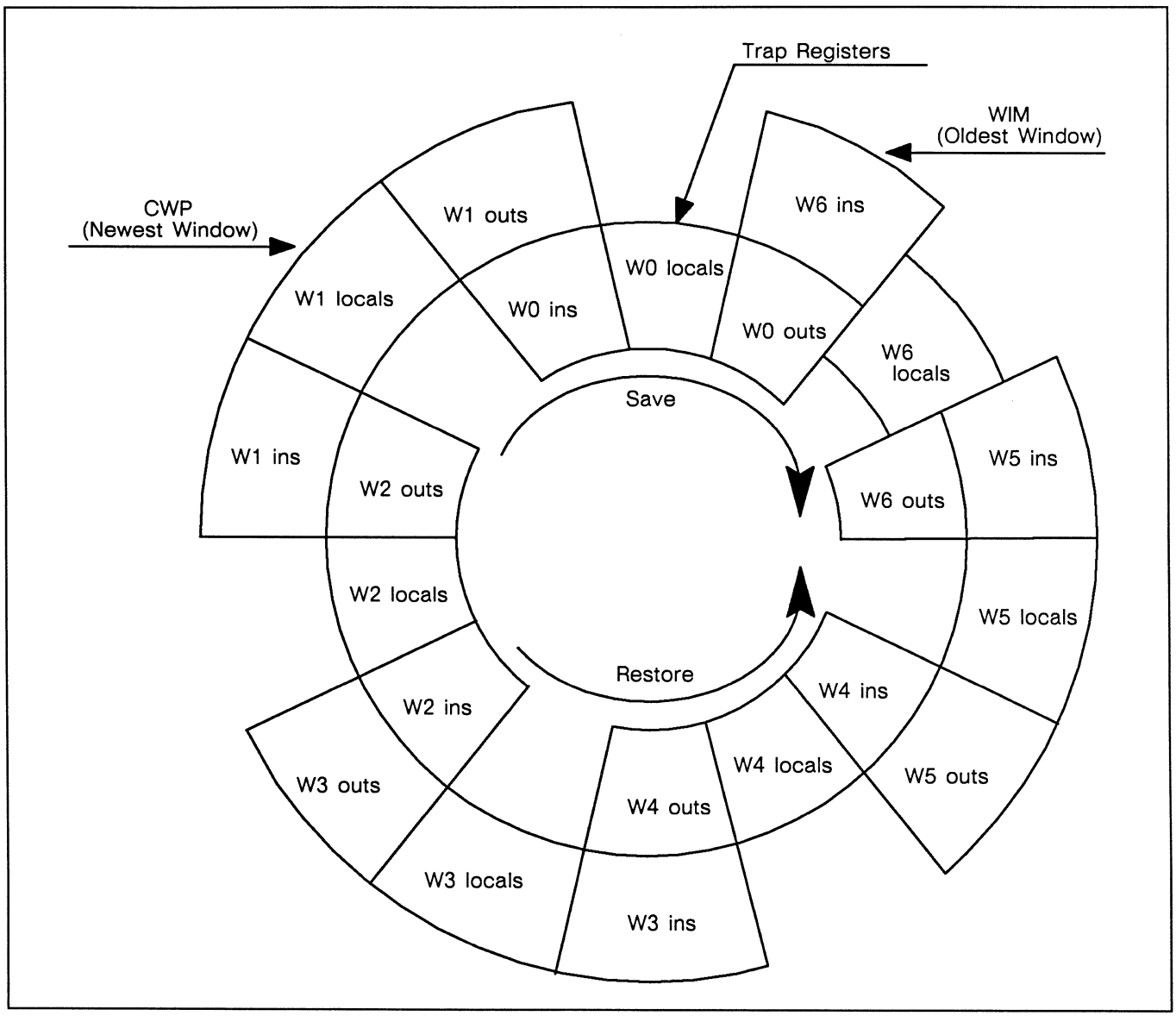
The LOCAL registers are unique to each window. The IN and OUT registers, however, are shared between adjacent windows. The OUTS from a previous window (CWP + 1) are the INS of the active (current) window, and the OUTS from the active window are the INS of the next window (CWP – 1). The windows are joined together in a circular stack, where the highest numbered window is adjacent to the lowest: the INS of window 6 are the OUTS of window 0.

The number of registers in the IN, LOCAL, and OUT window segments may be varied by software convention, but the total number of registers in the window may not exceed 24. The number of IN or OUT registers must be no greater than eight, and the number of IN and OUT registers must be equal. For instance, r8–r12 can be designated OUT registers, r13–r26 can be designated LOCAL registers, and r27–r31 can be designated IN registers for a total of 5 OUT registers, 14 LOCAL registers, and 5 IN registers.

2.5.1.1. Window Selection

The CWP field (bits <4:0>) of the Processor State Register (see Section 2.5.2.1) acts as a pointer to the active window. This field is written with the WRPSR (Write Processor State Register) instruc-

Figure 2.7.1 Register Windows



tion, is decremented by the SAVE instruction, and is incremented by the RESTORE and RETT (Return From Trap) instructions. The CWP cannot point to an unimplemented window; therefore arithmetic done on the CWP is done modulo the number of implemented windows (7 for the MB86901, NWINDOWS in general). The WRPSR instruction does not write the PSR and causes an "illegal instruction" trap if the result would cause the CWP field to point to an unimplemented window.

The Window Invalid Mask (WIM) Register is a register which, under software control, detects the occurrence of IU register file overflows and underflows. This register is further described in Section 2.5.2.2.

2.5.1.2. Register Window Use

In a conventional, non-windowed architecture, there is significant overhead associated with procedure call linkage. This overhead is the time and bus traffic required to save register contents and transfer parameters on entry to a procedure and to restore previous register contents and transfer results on return from the called procedure.

In the windowed architecture of the MB86901, the general registers visible to the user are a "window" into the larger register file. A procedure linkage mechanism can effectively "save" local variables in registers by simply moving the "window" to a fresh set of registers for the called procedure. This is accomplished by decrementing the CWP. Similarly, the previous register contents can be restored by changing the window pointer to indicate the original window. Since only a pointer is changed to bring about save and restore operations, procedure linkage overhead is greatly reduced.

Because the processor logically provides new locals and outs after every procedure call, register local values need not be saved and restored across calls. For instance, the local variables of a procedure in window "n" remain unaltered during trap servicing because the trap service routine uses window "n–1", with its own set of local registers. Once trap servicing is complete and the Return From Trap instruction is executed, the procedure continues with window "n" active again and containing the unaltered local values.

The overlapping registers also minimize the overhead of passing and returning parameters. In preparation for a call, a procedure can move parameters required by the called routine into the caller's OUT registers. After the CALL, the CWP is decremented with the SAVE instruction, and what was the "next" window becomes the active window with the parameters directly accessible to the called routine in its IN registers, since the OUT registers of the caller's register window are the IN registers of the called routine's register window.

Likewise, in preparing for a procedure return, the called routine can then move its results into its IN registers. After the CWP is incremented via the RESTORE instruction what was the "previous" window becomes the active window once again with the results available to the calling procedure, since the IN registers of the called routine's register window become the OUT registers of the calling procedure's register window. Note that the terms IN and OUT are defined relative to calling, not returning.

Window Overflow/Underflow

Since only a finite number of register windows are available, the register file can fill if the number of procedure calls exceeds the number of procedure returns by 6 (NWINDOWS – 1, in general). A subsequent call results in a Window Overflow trap (see Table 5.1). The overflow trap service routine must move one or more windows from the register file to memory (IN and LOCAL registers only) to release register window(s) for use.

Similarly, the register file can become empty if the number of procedure returns exceeds the number of calls by 6 (NWINDOWS – 1, in general). A subsequent return results in a Window Underflow trap. The underflow trap service routine can move previously saved windows from memory into the register file.

The SAVE instruction automatically checks for window overflow, and the RESTORE instruction automatically checks for window underflow according to the contents of the Window Invalid Mask Register. The Window Invalid Mask Register and its associated traps are further discussed in Section 2.5.2.2.

The representation of the register windows in Figure 2.7.1 illustrates the seven windows. Assume W6 is the oldest window, and W0 is the newest and corresponds to a procedure that attempted to execute a SAVE instruction and generated a window-overflow

trap. The trap handler cannot use W6's ins or W1's outs, but it is always guaranteed W0's locals.

Dedicated Register File Registers

Several registers have special uses which must be considered when using the register file.

If global register r0 is addressed as a source operand (rs1 or rs2 = 0), the operand value of 0 is returned. If r0 is addressed as a destination operand (rd = 0), **no** register is modified (see Instruction Formats, Section 3.1, for operand addressing descriptions). The r0 register is therefore a convenient source of 0 value for software, and a convenient destination register for instructions which are executed for condition code results only.

When a CALL instruction is executed, the current value of PC, which points to the CALL itself, is saved in Register 15 (r15) of the active window prior to the transfer of control to the CALL target. The called routine returns by a Jump and Link (JMPL) instruction (see Section 3.3.3) to the value in r15, plus an offset to account for the CALL and its delay instruction. In other words, the typical return address is the value in r15 plus 8. Note that the called routine must not use r15 without first saving the return address contained within.

Each trap saves the program counters (PC and nPC) into registers r17 and r18, respectively, of the "next" window for use in returning from the traps. These registers must therefore not be used by the trap service routines unless they are first saved in memory.

2.5.2. Special Purpose Registers

The MB86901 has four 32-bit, read/write registers which are used for special purposes. The Processor State Register (PSR), the Window Invalid Mask register (WIM), and the TRAP Base Register (TBR), are control/status registers. The fourth register, the Y register, is used during multiplication to create 64-bit products.

2.5.2.1. Processor State Register (PSR)

The Processor State Register is the primary processor control and status register. It contains 11 mode and status fields which configure the processor and report processor status and execution results. The mode fields, shown in upper case in the following PSR register diagram, are set by the operating system to configure the processor. The status fields, shown in lower case, are set by the processor to indicate effects of instruction execution.

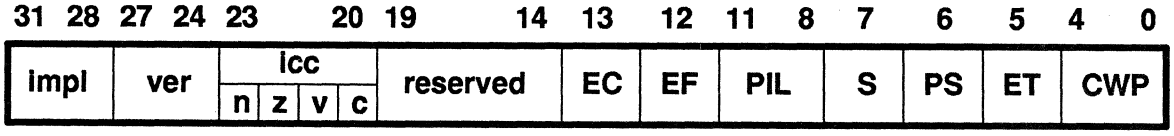
The PSR, with the exception of several fields described below, is written and read directly with the privileged WRPSR (Write PSR) and RDPSR (Read PSR) instructions. It is modified by the SAVE, RESTORE, Ticc,and RETT instructions, as well as instructions which affect the Integer Condition Code (icc) flags—bits <23:20>.

The PSR fields and their respective bits are:

impl<31:28>: Implementation
This field identifies the implementation number of the processor; it is hardwired to 0 in the MB86901 processor.

ver<27:24>: Version Number
This field identifies the processor version, and is intended for factory use. This field is hardwired to 0 in the MB86901 processor and can be read, but cannot be written.

icc<23:20>: Integer Condition Codes
This field contains the negative (n), zero (z), overflow (v), and carry (c) integer condition code flags. These bits are modified by the WRPSR instruction, and by arithmetic and logical instructions whose names end with the letters **cc** (for example, ANDcc). The Bicc and Ticc instructions base their control transfers on these bits, which are defined as follows:



Processor State Register (PSR)

n<23> —Set equal to 1 if the ALU result was negative for the last instruction that modified the icc field; equal to 0 otherwise.

z<22> —Set equal to 1 if the ALU result was zero for the last instruction that modified the icc field; equal to 0 otherwise.

v<21> —If this bit equals 1, an arithmetic overflow occurred during the last instruction that modified the icc field; equal to 0 otherwise. Logical instructions that modify the icc field always set the overflow bit equal to 0.

c<20> —If this bit equals 1, either an arithmetic carry out of bit 31 occurred as the result of the last addition that modified the icc, or a borrow into bit 31 occurred as the result of the last subtraction that modified the icc; equal to 0 otherwise. Logical operations that modify the icc field always set the carry bit to 0.

reserved<19:13>: Reserved Field
This field is reserved and should only be written to 0 with the WRPSR instruction.

EC: Enable Coprocessor
This bit determines whether the coprocessor is enabled or disabled. 1 = enabled, 0 = disabled.

EF: Enable Floating Point Unit
This bit determines whether the floating point unit is enabled, or disabled. 1 = enabled, 0 = disabled.

If the FPU is either disabled, or enabled and not present, an FPop, FPfcc, or floating point load/store instruction causes an fp disabled trap. Similarly, if the coprocessor (CP) is either disabled, or enabled and not present, a CPop, CPccc, or coprocessor load/store instruction causes a cp disabled trap.

When the FPU (or CP) is disabled, it retains its state until it is reenabled or reset. When disabled, the FPU can continue to execute instructions in its queue. The CP can also, if it has a queue.

When the FPU is present, software can use the EF bit to determine whether a particular process uses the FPU. If a process does not use the FPU, the FPU's registers need not be saved and restored across context switches. Also, if the FPU is not present, (as indicated by the bp_FPU_present signal), the fp disabled trap can be used to emulate the floating point instruction set. (This also applies to the coprocessor.)

PIL<11:8>: Processor Interrupt Level
This field identifies the processor interrupt level, with the processor accepting only interrupts with level 15 (non-maskable interrupts), or with levels higher than the value in the PIL field (maskable interrupts). Bit 11 is the most significant bit, and bit 8 is the least significant bit.

S<7>: Supervisor Mode
This mode bit determines whether the processor is in supervisor or user mode: when S = 1, the processor is in supervisor mode.

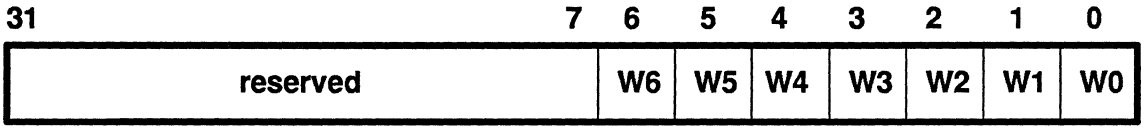
Note that because the instructions to write the PSR are available only in supervisor mode, the supervisor mode can be entered from only by hardware reset, or by a software or hardware trap.

PS<6>: Prior S State
The PS bit records the state of the S bit when each trap is entered so that the processor can return from the trap in the proper supervisor or user mode.

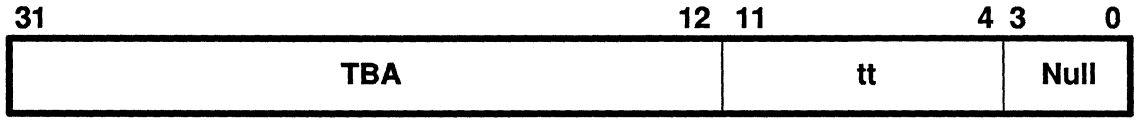
Processor hardware changes the PS bit to the state of the S bit when entering a trap, then changes the S bit to the state of the PS bit when returning from the trap.

ET<5>: Enable Traps
Traps are enabled when ET = 1. When ET = 0, traps are disabled and all asynchronous traps (interrupts) are ignored. Synchronous traps and floating point/coprocessor traps cause the IU to halt and enter error mode (see ERROR description, Table 7.1).

CWP<4:0>: Current Window Pointer
The CWP field points to the register window which is currently active. CWP is written and read with the WRPSR and RDPSR instructions, is decremented by traps and the SAVE instruction, and is incremented by the RESTORE and RETT instructions. The CWP cannot point to an unimplemented window. Attempting to write a value to the CWP field which points to an unimplemented window results in an "illegal instruction" error. Therefore, arithmetic done on the CWP is done modulo 7 to ensure that it always points to one of the seven implemented windows.



Window Invalid Mask Register (WIM)



Trap Base Register (TRB)

2.5.2.2. Window Invalid Mask Register (WIM)

The WIM register contains seven register window mask bits, each of which corresponds to an implemented register window. If a SAVE, RESTORE, or RETT would cause the CWP to point to a window whose corresponding WIM bit equals 1, it causes a Window Overflow (SAVE) or Window Underflow (RESTORE, RETT) trap.

The WIM register can be written with the WRWIM instruction, and the register can be read with the RDWIM instruction. Bits corresponding to unimplemented windows are read as zeroes, and values written to unimplemented bits are ignored.

The WIM provides the following fields:

reserved<31:7>: Reserved Field
This field is reserved for potential future expansion to additional windows.

W6-W0<6:0>: Window Masks
Each of these bits is a register window mask bit, with W0 the mask bit for Window 0, W1 the mask bit for Window 1, etc.

2.5.2.3. Trap Base Register (TBR)

The Trap Base Register contains three fields which generate the address of the trap handler when a trap occurs. These are:

TBA<31:12>: Trap Base Address
The TBA field contains the base address of the Trap Dispatch Table, which is controlled by software. The most significant 20 bits of the trap table base address are written into the TBA field with the WRTBR instruction. Note that the reset trap is an exception; it traps to address 0.

tt<11:4>: Trap Type
Each trap is identified by a unique 8-bit trap type number. The processor writes the appropriate trap type number into the tt field of the TBR when it recognizes a trap, then uses the number as an offset into the Trap Dispatch Table. The tt field remains unchanged until the next trap occurs. The WRTBR instruction does not affect the tt field.

null<3:0>: Null Field
This field is hardwired to 0 to force word access in the Trap Dispatch Table. The WRTBR instruction does not affect this field.

The entire TBR register can be read with the RDTBR instruction. Use of the TBR is further described in Section 5.4, **Trap Addressing**.

2.5.2.4. Y Register

The multiply step instruction (MULScc) uses the 32-bit Y register to generate 64-bit products. Use of the Y register is further described in Section 3.2.3.

The Y register can be written with the WRY instruction, and read with the RDY instruction.

2.5.3. Program Counters

Two registers, the Program Counter (PC) and the Next Program Counter (nPC), are used by the processor for instruction addressing.

Figure 2.8 Program Counter Sequencing

PC	NPC	Instruction
8	12	Load
12	16	Branch To 40
16	40	Store
40	44	Add

The PC register contains the word address of the instruction currently being executed by the IU, and the nPC register contains the word address of the next instruction to be executed, assuming a trap does not occur. In delayed control transfers (see Section 3.3), the instruction that immediately follows a control transfer (the delay instruction) may be executed before control is transferred to the target. The nPC is necessary to implement this feature. Most instructions complete by copying the contents of the nPC into the PC, then either increment nPC by 4, or, if the instruction implies a control transfer, write the computed target address into nPC. The PC now points to the instruction which will be executed next, and the nPC points to the instruction which will be executed **after** the next one; in other words, **two** instructions hence.

Figure 2.8 shows PC and NPC changes during instruction execution. Note that the delay instruction is executed before the control transfer occurs.

The PC and NPC registers cannot be directly written or read.

3. Instruction Set

The Fujitsu MB86901 instruction set is comprised of 66 basic instructions which are categorized into four main functional groups as shown in Table 3.1.

Mnemonics for the complete MB86901 instruction set may be obtained from the above table of basic instructions by forming all possible combinations of boldface letters. For example, the complete set of load instructions is:

LDSB	Load Signed Byte
LDSBA	Load Signed Byte from Alternate space
LDSH	Load Signed Halfword
LDSHA	Load Signed Halfword from Alternate space
LDUB	Load Unsigned Byte
LDUBA	Load Unsigned Byte from Alternate space
LDUH	Load Unsigned Halfword
LDUHA	Load Unsigned Halfword from Alternate space
LD	Load word
LDA	Load word from Alternate space
LDD	Load Doubleword
LDDA	Load Doubleword from Alternate space

Table 3.1 Instruction Functional Groups

DATA TRANSFER:				
LoaD	Signed	Byte	normal	
	Unsigned	Halfword	**A**lternate	8
LoaD		word	normal	
		Doubleword	**A**lternate	4
STore		Byte	normal	
		Halfword	**A**lternate	
		word		
		Doubleword		8
atomic **SWAP** word				
atomic Load-Store Unsigned Byte				2
ARITHMETIC/LOGICAL/SHIFT:				
ADD		normal	normal	
SUB		e**X**tended	set **CC**	8
AND		normal	normal	
OR		Not	set **CC**	
XOR				12
Shift **L**eft Logical				
Shift **R**ight Logical				
Shift **R**ight **A**rithmetic				3
Tagged	**ADD** set **CC**	normal		
	SUB set **CC**	Trap o**V**erflow		4
MULtiply Step set **CC**				
SET HIgh				
SAVE				
RESTORE				4
CONTROL TRANSFER:				
Branch on Integer **C**ondition **C**ode				
CALL				
JuMP and Link				
RETurn from Trap				
Trap on Integer Condition Code				5
READ/WRITE CONTROL REGISTER:				
ReaD	**PSR**			
WRite	**TBR**			
	WIM			
	Y			8
TOTAL NUMBER OF INSTRUCTIONS:				66

Table 3.2 Instruction Set

Instruction	Mnemonic	Format	OP	OP2	OP3	opf	Cycles
Absolute Value **	FABS	3c	2		34	009	
Add	ADD	3a/3b	2		00		1
Add and Set Condition Code	ADDCC	3a/3b	2		10		1
Add with Carry	ADDX	3a/3b	2		08		1
Add with Carry and Set Condition Code	ADDXCC	3a/3b	2		18		1
Add Double **	FADDD	3c	2		34	042	
Add Single **	FADDS	3c	2		34	041	
And	AND	3a/3b	3		01		1
And and Set Condition Code	ANDCC	3a/3b	3		11		1
And Not	ANDN	3a/3b	3		05		1
And Not and Set Condition Code	ANDNCC	3a/3b	3		15		1
Atomic Load-Store Unsigned Byte	LDSTUB	3a/3b	3		0D		4
Atomic Load-Store Unsigned Byte In Alternate Space *	LDSTUBA	3a/3b	3		1D		4
Branch on Integer Condition Code	BICC	2b	0	2			1-2(1)
Branch on Floating Point Condition Code	FBFCC	2b	0	6			1-2(1)
Call	CALL	1					1
Compare Double **	FCMPD	3c	2		35	052	
Compare Double and Exception if Unordered **	FCMPED	3c	2		35	056	
Compare Single **	FCMPS	3c	2		35	051	
Compare Single and Exception if Unordered **	FCMPES	3c	2		35	055	
Convert Double to Integer **	FDTOI	3c	2		34	0D2	
Convert Double to Single **	FDTOS	3c	2		34	0C6	
Convert Integer to Double **	FITOD	3c	2		34	0C8	
Convert Integer to Single **	FITOS	3c	2		34	0C4	
Convert Single to Double **	FSTOD	3c	2		34	0C5	
Convert Single to Integer **	FSTOI	3c	2		34	0D1	
Divide Double **	FDIVD	3c	2		34	04E	
Divide Single **	FDIVS	3c	2		34	04D	
Exclusive Nor	XNOR	3a/3b	3		07		1
Exclusive Nor and Set Condition Code	XNORCC	3a/3b	3		17		1
Exclusive Or	XOR	3a/3b	3		03		1
Exclusive Or and Set Condition Code	XORCC	3a/3b	3		13		1
Inclusive Or	OR	3a/3b	3		02		1
Inclusive Or and Set Condition Code	ORCC	3a/3b	3		12		1
Inclusive Or Not	ORN	3a/3b	3		06		1
Inclusive Or Not and Set Condition Code	ORNCC	3a/3b	3		16		1
Jump and Link	JMPL	3a/3b	2		38		2
Load Double Word	LDD	3a/3b	3		03		3
Load Double Word From Alternate space *	LDDA	3a/3b	3		13		3
Load Double Floating Point Register	LDDF	3a/3b	3		23		3
Load Floating Point Register	LDF	3a/3b	3		20		2
Load Floating Point Status Register	LDFSR	3a/3b	3		21		
Load Signed Byte	LDSB	3a/3b	3		09		2
Load Signed Byte From Alternate space *	LDSBA	3a/3b	3		19		2
Load Signed Halfword	LDSH	3a/3b	3		0A		2
Load Signed Halfword From Alternate space *	LDSHA	3a/3b	3		1A		2
Load Unsigned Byte	LDUB	3a/3b	3		01		2
Load Unsigned Byte From Alternate space *	LDUBA	3a/3b	3		11		2
Load Unsigned Halfword	LDUH	3a/3b	3		02		2
Load Unsigned Halfword From Alternate space *	LDUHA	3a/3b	3		12		2
Load Word	LD	3a/3b	3		00		2
Load Word From Alternate space *	LDA	3a/3b	3		10		2

Table 3.2 Instruction Set (cont.)

Instruction	Mnemonic	Format	OP	OP2	OP3	opf	Cycles
Move **	FMOVS	3c	2		34	001	
Multiply Double **	FMULD	3c	2		34	04A	
Multiply Single **	FMULS	3c	2		34	049	
Multiply Step and Set Condition Code	MULSCC	3a/3b	3		24		1
Negate **	FNEGS	3c	2		34	005	
Read Processor State Register *	RDPSR	3(2)	2		29		1
Read Trap Base Register	RDTBR	3(2)	2		2B		1
Read Window Invalid Mask Register *	RDWIM	3(2)	2		2A		1
Read Y Register	RDY	3(2)	2		28		1
Restore	RESTORE	3a/3b	2		3D		1
Return from Trap *	RETT	3a/3b	2		39		2
Save	SAVE	3a/3b	2		3C		1
Set r Register High 22 bits	SETHI	2a	0	4			1
Shift Left Logical	SLL	3a/3b	3		25		1
Shift Right Arithmetic	SRA	3a/3b	3		27		1
Shift Right Logical	SRL	3a/3b	3		26		1
Store Byte	STB	3a/3b	3		05		3
Store Byte to Alternate space *	STBA	3a/3b	3		15		3
Store Double Floating Point Queue	STDFQ	3a/3b	3		26		4
Store Double Floating Point	STDF	3a/3b	3		27		4
Store Double Word	STD	3a/3b	3		07		4
Store Double Word to Alternate space *	STDA	3a/3b	3		17		4
Store Floating Point Register	STF	3a/3b	3		24		3
Store Floating Point Status Register	STFSR	3a/3b	3		25		4
Store Halfword	STH	3a/3b	3		06		3
Store Halfword to Alternate space *	STHA	3a/3b	3		16		3
Store Word	ST	3a/3b	3		04		3
Store Word to Alternate space *	STA	3a/3b	3		14		3
Subtract	SUB	3a/3b	3		04		1
Subtract and Set Condition Code	SUBCC	3a/3b	3		14		1
Subtract with Carry	SUBX	3a/3b	3		0C		1
Subtract with Carry and Set Condition Code	SUBXCC	3a/3b	3		1C		1
Subtract Double **	FSUBD	3c	2		34	046	
Subtract Single **	FSUBS	3c	2		34	045	
Tagged Add and Set Condition Code	TADDCC	3a/3b	3		20		1
Tagged Add and Set Condition Code and Trap on Overflow	TADDCCTV	3a/3b	3		22		1
Tagged Subtract and Set Condition Code	TSUBCC	3a/3b	3		21		1
Tagged Subtract and Set Condition Code and Trap on Overflow	TSUBCCTV	3a/3b	3		23		1
Trap on Integer Condition Code	TICC	3a/3b	2		3A		1-4(3)
Write Processor State Register *	WRPSR	3a/3b	2		31		1
Write Trap Base Register *	WRTBR	3a/3b	2		33		1
Write Window Invalid Mask Register *	WRWIM	3a/3b	2		32		1
Write Y Register	WRY	3a/3b	2		30		1

* Privileged Instruction
** Floating Point Operate Instruction

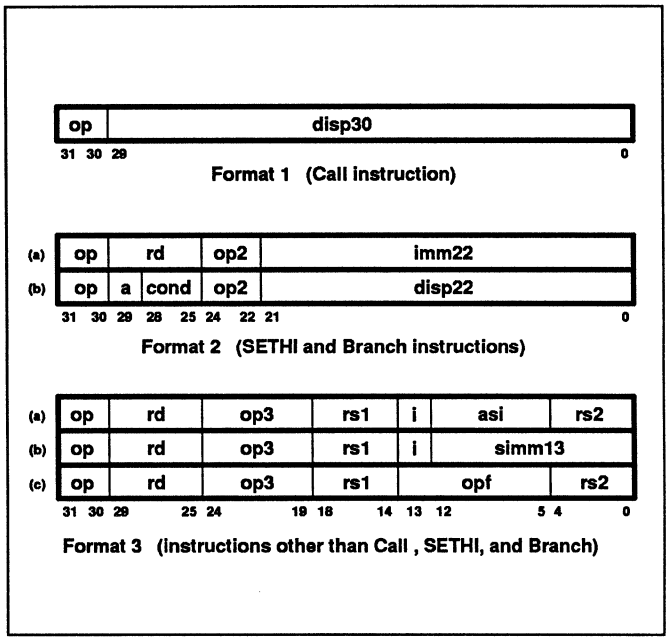
Notes:
(1) One cycle if branch taken; two cycles if branch not taken.
(2) Fields other than op, rd, and op3 are ignored.
(3) Four cycles if trap taken; one cycle if trap not taken.

3.1. Data Transfer Instructions

The Load and Store instructions are the only instructions which access main memory, using two IU registers (Format 3a) or an IU register and a signed immediate value (Format 3b) to calculate the 32-bit byte address in memory. In addition to the address, the processor always generates an address space identifier, or asi (see Section 2.4). The instruction's destination field specifies either an IU register, an FPU register, or a coprocessor register to supply the data for a store or receive the data from a load. I/O device registers are accessed via load/store instructions.

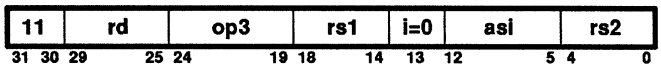
Mnemonic	op	op3	operation
LDSB	11	001001	Load Signed Byte
LDSBA	11	011001	Load Signed Byte from Alternate space
LDSH	11	001010	Load Signed Halfword
LDSHA	11	011010	Load Signed Halfword from Alternate space
LDUB	11	000001	Load Unsigned Byte
LDUBA	11	010001	Load Unsigned Byte from Alternate space
LDUH	11	000010	Load Unsigned Halfword
LDUHA	11	010010	Load Unsigned Halfword from Alternate space
LD	11	000000	Load word
LDA	11	010000	Load word from Alternate space
LDD	11	000011	Load Doubleword
LDDA	11	010011	Load Doubleword from Alternate space
STB	11	000101	Store Byte
STBA	11	010101	Store Byte into Alternate space
STH	11	000110	Store Halfword
STHA	11	010110	Store Halfword into Alternate space
ST	11	00100	Store Word
STA	11	010100	Store Word into Alternate space
STD	11	000111	Store Doubleword
STDA	11	010111	Store Doubleword into Alternate space

Figure 3.1 Instruction Formats

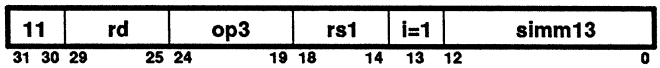


The effective address for a load or store instruction is either "r[**rs1**] = r[**rs2**]" if the **i** field is zero, or "r[**rs1**] + **simm13**" if the **i** field is one. Instructions which load from or store to an alternate address space must have zero in the **i** field and the address space identifier to be used in the **asi** field. Otherwise, the address space indicated is either a user or system data space, according to the S bit of the PSR (see Section 2.5.2.1).

Note that the load/store alternate instructions are privileged; they can only be executed in the supervisor mode.



Format 3a

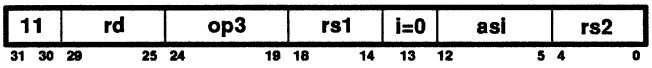


Format 3b

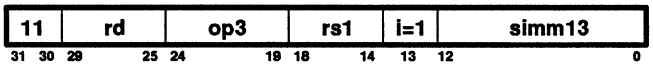
3.1.1. Multiprocessor Support Instructions

The special instruction "atomic load and store unsigned byte" (LDSTUB) supports tightly coupled multiprocessors. The LDSTUB instruction reads a byte from memory into an IU register and the rewrites the same byte in memory to all ones, while also precluding intervening accesses on the memory or I/O bus. The LDSTUB instruction may be used to construct semaphores. It has byte, rather than word, addressing because word-wide registers may not be aligned at word addresses in general purpose I/O buses.

Mnemonic	op	op3	operation
LDSTUB	11	001101	Atomic Load-Store Unsigned Byte
LDSTUBA	11	011101	Atomic Load-Store Unsigned Byte into Alternate space



Format 3a



Format 3b

These instructions move a byte from memory into the r register specified by the **rd** field, rewriting the same byte to memory to all ones without allowing intervening asynchronous traps. In a multiprocessor system, two or more processors executing atomic load-store instructions addressing the same byte simultaneously are guaranteed to execute them in some serial order.

The effective address for a load or store instruction is either "r[**rs1**] = r[**rs2**]" if the **i** field is zero, or "r[**rs1**] + **simm13**" if the **i** field is one. LDSTUBA must have zero in the **i** field, or an illegal instruction trap occurs. The address space identifier used for the memory accesses is taken from the **asi** field. For LDSTUB, the address space indicates either a user or system data space access, according to the S bit in the PSR.

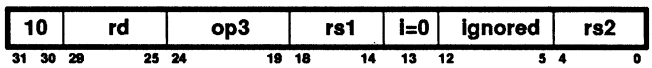
If an atomic load-store instruction traps, memory remains unchanged. However, an implementation may cause a "data access exception" trap during the store memory access, but not during the load access. In this case, the destination register can be changed.

3.2. Arithmetic/Logical/Shift Instructions

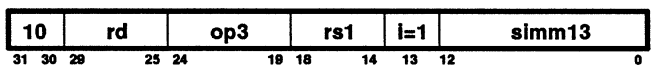
The Arithmetic/Logical/Shift instructions, with the exception of SETHI, compute a result that is a function of two source operands, and either write the result into a destination register, r[**rd**], or discard it. One of the operands is always register r[**rs1**]. The other operand depends on the **i** bit in the instruction format: if **i** = 0, the operand is register r[**rs2**], but if **i** = 1, the operand is a 13-bit sign-extended constant, **simm 13**.

3.2.1. Add and Subtract

Mnemonic	op	op3	operation
ADD	10	000000	Add
ADDcc	10	010000	Add and modify icc
ADDX	10	001000	Add with carry
ADDXcc	10	011000	Add with carry and modify icc
SUB	10	000100	Subtract
SUBcc	10	010100	Subtract and modify icc
SUBX	10	001100	Subtract with carry
SUBXcc	10	011100	Subtract with carry and modify icc



Format 3a



Format 3b

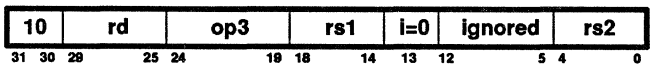
ADD and ADDcc (SUB and SUBcc) compute either "r[**rs1**] + (–) r[**rs2**]" if **i** = 0, or "r[**rs1**] + (–) **simm13**" if **i** = 1, and place the result in the r register specified in the **rd** field.

ADDX and ADDXcc (SUBX and SUBXcc) add (subtract) the PSR's carry bit **c** also; that is, they compute "r[**rs1**] + (–) r[**rs2**] + (–) **c**" or "r[**rs1**] + (–) **simm13** + (–) **c**", if **i** = 0 or **i** = 1, respectively, and place the result in the r register specified in the **rd** field.

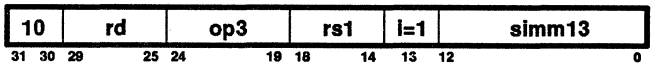
ADDcc and ADDXcc (SUBcc and SUBXcc) modify all the integer condition codes.

3.2.2. Tagged Add and Subtract

Mnemonic	op	op3	operation
TADDcc	10	100000	Tagged Add and modify icc
TADDccTV	10	100010	Tagged Add, modify icc, Trap on Overflow
TSUBcc	10	100001	Tagged Subtract, and modify icc
TSUBccTV	10	100011	Tagged Subtract, modify icc, Trap on Overflow



Format 3a



Format 3b

Tagged add and subtract instructions operate on tagged data where the tag is the least significant two bits of data. TADDcc and TADDccTV (TSUBcc and TSUBccTV) compute either "r[**rs1**] + (–) r[**rs2**]" if **i** = 0, or "r[**rs1**] + (–) **simm13**" if **i** = 1, and place the result in the r register specified in the **rd** field. If either of the source operands has a nonzero tag, the overflow bit, **v**, of the PSR is set. This bit is also set if the addition (subtraction) generates an arithmetic overflow.

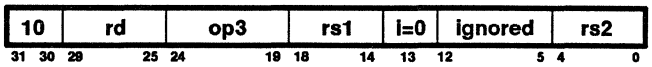
If a TADDccTV (TSUBccTV) results in an overflow condition, a "tag overflow" trap is generated and the destination register and condition codes remain unchanged. If a TADDccTV (TSUBccTV) does not result in an overflow condition, all the integer condition codes are updated (in particular, the overflow **v** bit is set to 0), and the result of the addition (subtraction) is written into the r register specified by the **rd** field.

If a TADDcc (TSUBcc) results in an overflow condition, the overflow bit **v** of the PSR is set; if the instruction does not result in an overflow, the bit is cleared. In either case, the remaining integer condition codes are also updated and the result of the addition (subtraction) is written into the r register specified by the **rd** field.

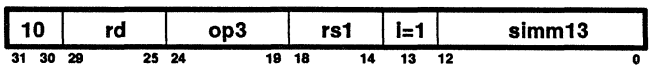
The tagged arithmetic instructions can be used by languages such as Lisp, Smalltalk, and Prolog which benefit from tags. Normally, a tagged add/subtract instruction is followed by a conditional branch, which, if the overflow bit has been set, transfers control to code which further deciphers the operand types. The two variants, TADDccTV and TSUBccTV, which trap when the overflow bit has been set, can be used for error checking when the compiler knows the operand types.

3.2.3. Multiply Step Instruction

Mnemonic	op	op3	operation
MULScc	10	100100	Multiply Step and modify icc



Format 3a



Format 3b

The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words. In the following description, the incoming partial product is located in register r[**rs1**], and the multiplicand is either the contents of register r[**rs2**] (if **i** = 0) or the sign extended **simm13** (if **i** = 1). The instruction operates as follows:

(1) The value obtained by shifting the incoming partial product right by one bit and replacing its high-order bit by **n** XOR **v** (the sign of the previous partial product—see Section 2.5.2.1 **Integer Condition Codes**) is computed.

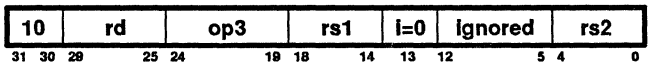
(2) If the least significant bit of the Y register (the multiplier) is set, the value from step (1) is added to the multiplicand. If the LSB of the Y register is not set, then zero is added to the value from step (1).

(3) The result from step (2), the outgoing partial product, is written into r[**rd**]. The integer condition codes in the Processor State Register are updated according to the addition performed in step (2).

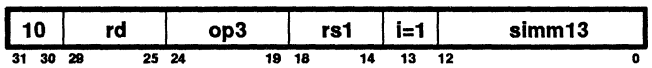
(4) The Y register (the multiplier) is shifted right by one bit, and its high-order bit is replaced by the least significant bit of the incoming partial product.

3.2.4. Logical Instructions

Mnemonic	op	op3	operation
AND	10	000001	And
ANDcc	10	010001	And and modify icc
ANDN	10	000101	And Not
ANDNcc	10	010101	And Not and modify icc
OR	10	000010	Inclusive Or
ORcc	10	010010	Inclusive Or and modify icc
ORN	10	000110	Inclusive Or Not
ORNcc	10	010110	Inclusive Or Not and modify icc
XOR	10	000011	Exclusive Or
XORcc	10	010011	Exclusive Or and modify icc
XNOR	10	000111	Exclusive Nor
XNORcc	10	010111	Exclusive Nor and modify icc



Format 3a



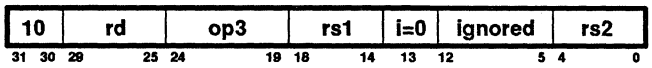
Format 3b

These instructions implement the bitwise logical operations. With op denoting the generic logical operation, these instructions compute either r[**rs1**] op r[**rs2**] (if **i** = 0) or r[**rs1**] op **simm13** (if **i** = 1).

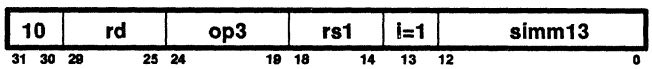
ANDcc, ANDNcc, ORcc, ORNcc, XORcc, and XNORcc modify all the integer condition codes as described in Section 2.5.2.1.

3.2.5. Shift Instructions

Mnemonic	op	op3	operation
SLL	10	100101	Shift Left Logical
SRL	10	100110	Shift Right Logical
SRA	10	100111	Shift Right Arithmetic



Format 3a



Format 3b

The shift count for these instructions is either the least significant five bits of r[**rs2**] (if **i** = 0) or the least significant five bits of **simm13** (if **i** = 1).

SLL shifts the value of r[**rs1**] left by the number of bits implied by the shift count.

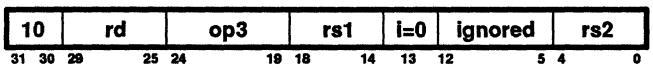
SRL and SRA shift the value of r[**rs1**] right by the number of bits implied by the shift count.

SLL and SRL replace vacated positions with zeroes, whereas SRA fills vacated positions with the most significant bit of r[**rs1**]. No shift occurs when the shift count is zero.

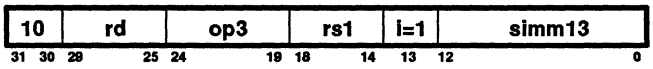
All of these instructions place the shifted result in r[**rd**]. These instructions do not modify the integer condition codes.

3.2.6. Save and Restore Instructions

Mnemonic	op	op3	operation
SAVE	10	111100	Save caller's window
RESTORE	10	111101	Restore caller's window



Format 3a



Format 3b

The SAVE instruction subtracts one from the CWP (modulo 7) and compares this value, the "new CWP", against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the "new CWP" is set, then a window overflow trap is generated. If the corresponding WIM bit is reset, then a window overflow trap is not generated and "new CWP" is written into CWP. This causes the **active** window to become the **previous** window, thereby saving the caller's window.

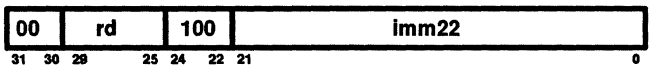
The RESTORE instruction adds one to the CWP (modulo 7) and compares this value, the "new CWP", against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the "new CWP" is set, then a window underflow trap is generated. If the corresponding WIM bit is reset, then a window underflow trap is not generated and "new CWP" is written into CWP. This causes the **previous** window to become the **active** window, thereby restoring the caller's window.

Furthermore, if an overflow or underflow trap is not generated, SAVE and RESTORE behave like normal ADD instructions, except that the operands r[**rs1**] or r[**rs2**] are read from the **old** window (i.e., the window addressed by the original CWP) and the result is written into r[**rd**] of the **new** window (i.e., the window addressed by the new CWP).

Note that CWP arithmetic is performed modulo 7.

3.2.7. SETHI Instruction

Mnemonic	op	op2	operation
SETHI	00	100	Set High



Format 2a

The SETHI instruction is a special instruction that can be used in combination with a standard arithmetic instruction to construct a 32-bit constant in two instructions. It loads a 22-bit immediate value, **imm22**, into the high order bits of the destination register r[**rd**] and zeroes the least significant 10 bits. The integer condition codes are not affected. Another instruction is used to load the low 10 bits. Because other arithmetic/logical instructions have a 13-bit signed immediate value, the 22-bit SETHI value implies an overlap of 2 bits in the result. In combination with a load or store instruction, SETHI can also be used to construct a 32-bit load/store address. The SETHI instruction uses Format 2a, shown.

It is suggested that sethi 0, %0, where %0 stands for global register 0, be used as the preferred NOP, since it will not cause an increase in execution time if it follows a load instruction.

3.3. Control Transfer Instructions

Control transfer instructions change the values of PC and nPC. There are five types of control transfer instructions:

(1) Conditional branch (Bicc)
(2) Call (CALL)
(3) Jump and Link (JMPL)
(4) Trap (Ticc)
(5) Return from trap (RETT)

Each of these instructions can be further categorized according to whether it is 1) PC (program counter)-relative or register-indirect, or 2) delayed or non-delayed. A PC-relative control transfer computes its target address by adding the (shifted) sign-extended immediate displacement to the program counter. A register-indirect instruction computes its target address as either r[**rs1**] + r[**rs2**] (if **i** = 0), or r[**rs1**] + **simm13** (if **i** = 1). A control transfer

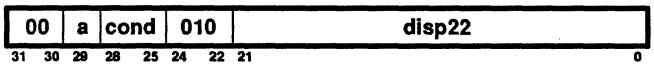
instruction is delayed if it transfers control to the target after a one-instruction delay. In summary:

Instruction	Characteristic	Delayed
Bicc, CALL	PC-relative	Yes
JMPL, RETT	register-indirect	Yes
Ticc	register-indirect	No

The rationale behind the use of delayed control transfers is explained in Section 2.1.5 under the discussion of the MB86901 pipeline. The mechanics of the operation is explained in Section 2.5.3 **Program Counters**.

3.3.1. Branch on Integer Condition Instructions

Mnemonic	op	op2	operation
Bicc	00	010	See Table 3.3



Format 2b

A Bicc instruction (except BA and BN) evaluates the integer condition codes (**icc**) of the Processor State Register (PSR) according to the **cond** field. If the condition codes evaluate to true, the branch is taken, and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4*sign_ext(**disp22**))". If the condition codes evaluate to false, the branch is not taken.

The annul bit **a** changes the behavior of the delay instruction. This bit is only available on conditional branch instructions (Bicc). The effect of the annul bit may be summarized as follows:

If **a** = 1, and the branch in a conditional branch is taken, then the delay instruction which follows is executed.

If **a** = 1, and the branch is not taken, then the delay instruction is annulled (not executed).

Branch Never (BN) acts like a NOP except that if **a** = 1, the delay instruction is annulled (not executed).

Branch Always (BA) causes transfer of control regardless of the value of the condition code bits. If **a** = 1, the delay instruction is annulled (not executed).

In every case, if **a** = 0, the delay instruction is executed.

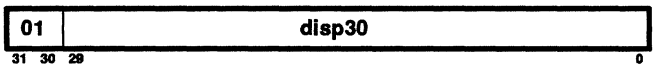
	Execution of Delay Instruction	
	a = 0	a = 1
Conditional/Taken	YES	YES
Conditional/Not Taken	YES	NO
BA	YES	NO
BN	YES	NO

Note that the result for BA with **a** = 1 is the reverse of what might be expected. The delay instruction is NOT executed even though the branch IS taken.

An annulled instruction has no effect on the state of the IU, nor can any trap occur during an annulled instruction.

3.3.2. CALL Instruction

Mnemonic	op	operation
CALL	01	Call



Format 1

A procedure that requires a register window is invoked by executing both a CALL and a SAVE (see Section 3.2.6) instruction. A procedure which does not need a register window, a so-called "leaf" routine, is invoked by executing only a CALL instruction. Leaf routines can use only the **out** registers.

The CALL instruction stores the PC, which points to the CALL itself, into **out** register r[**15**]. The CALL instruction then causes an unconditional, delayed, PC-relative control transfer to address "PC + (4 * **disp30**)". The PC-relative displacement is formed by appending two low order zeroes to the instruction's 30-bit word displacement field. Since the word displacement (**disp30**) field is 30 bits wide, the target address can be arbitrarily distant.

A JMPL instruction (see Section 3.3.3) with **rd** = 15 can be used as a register indirect CALL.

Table 3.3 Integer Branch Conditions

Mnemonic	cond	Operation	icc Test
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not z
BE	0001	Branch on Equal	z
BG	1010	Branch on Greater	not (z or (n xor V))
BLE	0010	Branch on Less or Equal	z or (n xor v)
BGE	1011	Branch on Greater or Equal	not (n xor v)
BL	0011	Branch on Less	n xor v
BGU	1100	Branch on Greater Unsigned	not (c or z)
BLEU	0100	Branch on Less or Equal Unsigned	(c or z)
BCC	1101	Branch on Carry Clear *	not c
BCS	0101	Branch on Carry Set **	c
BPOS	1110	Branch on Positive	not n
BNEG	0110	Branch on Negative	n
BVC	1111	Branch on Overflow Clear	not v
BVS	0111	Branch on Overflow Set	v

* Greater Than or Equal, Unsigned
** Less Than, Unsigned

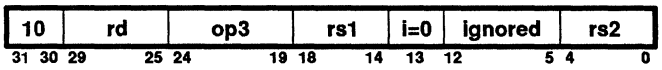
Table 3.4 Integer Trap Conditions

Mnemonic	cond	Operation	icc Test
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not z
TE	0001	Trap on Equal	z
TG	1010	Trap on Greater	not (z or (n xor v))
TLE	0010	Trap on Less or Equal	z or (n xor v)
TGE	1011	Trap on Greater or Equal	not (n xor v)
TL	0011	Trap on Less	n xor v
TGU	1100	Trap on Greater Unsigned	not (c or z)
TLEU	0100	Trap on Less or Equal Unsigned	(c or z)
TCC	1101	Trap on Carry Clear *	not c
TCS	0101	Trap on Carry Set **	c
TPOS	1110	Trap on Positive	not n
TNEG	0110	Trap on Negative	n
TVC	1111	Trap on Overflow Clear	not v
TVS	0111	Trap on Overflow Set	v

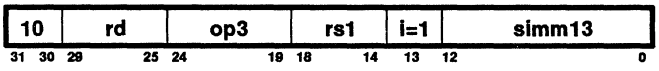
* Greater Than or Equal, Unsigned
** Less Than, Unsigned

3.3.3. Jump and Link Instruction

Mnemonic	op	op3	operation
JMPL	10	111000	Jump and Link



Format 3a



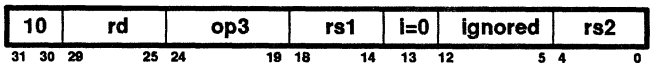
Format 3b

A procedure which uses a register window returns by executing both a RESTORE (see Section 3.2.6) and a JMPL instruction. A leaf procedure returns by executing a JMPL only. The JMPL instruction typically returns to the instruction following the CALL's delay instruction. In other words, the typical return address is 8 plus the address saved by the CALL.

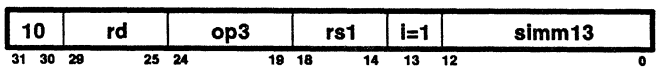
The JMPL instruction causes a register indirect control transfer to an address specified by either r[**rs1**] + r[**rs2**] (if **i** = 0), or r[**rs1**] + **simm13** (if **i** = 1). The JMPL instruction writes the PC, which contains the address of the JMPL instruction, into the destination register specified in the **rd** field. If either of the two low order bits of the jump address is nonzero, a "mem address not aligned" trap occurs.

3.3.4. Trap on Integer Condition Codes

Mnemonic	op	op3	operation
Ticc	10	111010	See Table 3.4



Format 3a



Format 3b

A Ticc instruction evaluates the integer condition codes (**icc**) of the Processor State Register according to the **cond** field. If the condition codes evaluate to true, and there are no higher priority traps pending, then a "trap instruction" trap is generated with no delay instruction. If the condition codes evaluate to false, it executes as a NOP.

If a trap is generated, the **tt** field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of either r[**rs1**] + r[**rs2**] (if **i** = 0), or r[**rs1**] + **simm13** (if **i** = 1). The processor enters the supervisor mode, disables traps, decrements the CWP, and saves PC and nPC into local registers r[17] and r[18] (respectively) of the new window. 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 indices, integer overflow, etc.

See Section 5 on **Traps, Exceptions, and Error Handling** for a more complete discussion of traps.

3.3.5. Return from Trap Instruction

Mnemonic	op	op3	operation
RETT	10	111001	Return from Trap

The RETT instruction uses either Format 3a or Format 3b, as described in Section 3.3.4. The RETT instruction adds one to the CWP (modulo 7) and compares this value, the "new CWP", against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the "new CWP" is set, then a window underflow trap is generated. If the corresponding WIM bit is reset, then a window underflow trap is not generated and "new CWP" is written into CWP. This causes the **previous** window to become the **active** window, thereby restoring the window that existed at the time of the trap.

If a window underflow trap is not generated, RETT causes a delayed control transfer to the target address. The target address is either r[**rs1**] + r[**rs2**] (if **i** = 0), or r[**rs1**] + **simm13** (if **i** = 1). Furthermore, RETT restores the S field of the PSR from the PS field, and sets the ET field to one (see Section 2.5.2.3).

If traps are enabled (ET = 1), an illegal instruction trap occurs. If traps are disabled (ET = 0), and the processor is not in supervisor mode (S = 0), or if a window underflow condition is detected, or if either of the two low order bits of the target address is nonzero, a reset trap occurs. If a reset trap occurs, the **tt** field of the TBR encodes the trap condition: privileged instruction, window overflow, or memory address not aligned.

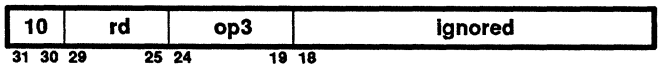
The instruction executed immediately before a RETT must be a JMPL instruction. If it is not, the location where execution continues is not necessarily within the address space implied by the PS bit of the PSR. Trap handlers complete execution by executing the "JMPL, RETT" couple.

3.4. Read/Write Control Register Instructions

These instructions read or write the contents of the programmer visible control registers. This category includes instructions to read and write the PSR, the WIM, the TBR, the Y register (see Section 2.5.2), as well as the Floating point State Register (FSR) and the Coprocessor State Register (CSR). These instructions are all privileged (available in supervisor mode only), except those that read and write the Y register, the FSR, and the CSR.

3.4.1. Read State Register Instructions

Mnemonic	op	op3	operation
RDY	10	101000	Read Y register
RDPSR	10	101001	Read Processor State Register
RDWIM	10	101010	Read Window Invalid Mask register
RDTBR	10	101011	Read Trap Base Register

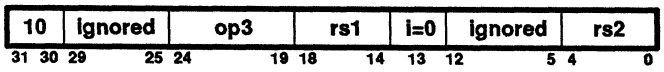


Format 3a

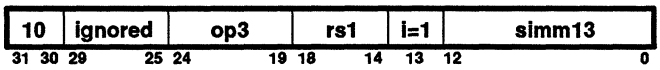
These instructions read the specified IU state registers into the r register specified in the **rd** field

3.4.2. Write State Register Instructions

Mnemonic	op	op3	operation
WRY	10	110000	Write Y register
WRPSR	10	110001	Write Processor State Register
WRWIM	10	110010	Write Window Invalid Mask register
WRTBR	10	110011	Write Trap Base Register



Format 3a



Format 3b

These instructions write either r[**rs1**] XOR r[**rs2**] (if **i** = 0), or r[**rs1**] XOR **simm13** (if **i** = 1), to those subfields of the specified IU state register which may be written. WRPSR does not write the PSR and causes an illegal instruction trap if the result would cause the CWP field of the PSR to point to an unimplemented window.

These instructions are delayed-write instructions:

(1) If any of the three instructions after a WRPSR uses any field of the PSR that is changed by the WRPSR, the value of that field is unpredictable. (Note that any instruction which references a non-global register implicitly uses the CWP.)

(2) If a WRPSR instruction is updating the PSR's PIL to a new value and is simultaneously setting ET to 1, this can result in an interrupt trap at a level equal to the old value of the PIL. Two WRPSR instructions should be used when enabling traps and changing the value of the PIL. The first WRPSR should specify ET = 0 with the new PIL value, and the second WRPSR should specify ET = 1 and the new PIL value.

(3) If any of the three instructions after a WRWIM is a SAVE, RESTORE, or RETT, the occurrence of window overflow and window underflow traps is unpredictable.

(4) If any of the three instructions which follow a WRY is a MULScc or RDY, the value of Y used is unpredictable.

(5) If any of the three instructions which follow a WRTBR causes a trap, the trap base address (TBA) used may be either the old or the new value.

(6) If any of the three instructions after a write state register instruction reads the modified state register, the value read is unpredictable.

(7) If any of the three instructions after a write state register instruction is trapped, a subsequent read state register instruction in the trap handler will get the register's new value.

4. Floating Point Operations

The MB86901 executes all SPARC instructions except for Floating Point Operate (FPop) instructions. All the floating point operations are actually dispatched through the F<31:0> bus to the MB86911 Floating Point Controller (FPC). Here, they enter a First In First Out (FIFO) floating point queue to await actual floating point processing done by the TI SN74ACT8847 Floating Point Processor (FPP). In addition to the floating point queue, the MB86911 also includes the data register file, status register, and control circuitry for mastering floating point exceptions. The FPC and the FPP are collectively referred to as the Floating Point Unit (FPU).

The IU and the FPU operate concurrently, assuming that the floating point queue remains in a non-overflow status. If the FPU encounters a floating point operate instruction that doesn't fit in the queue, the IU stalls untils the required FPU resource becomes available. The architecture hides floating point concurrency from the programmer by means of appropriate register interlocks. Hence, a program including floating point computations will generate the same results as if all instructions were executed sequentially.

The "F-bus" is used only to transfer floating point instructions and associated instruction Program Counter (PC) values to the FPC. Any data transfer to or from the FPC is done through the FPC memory data bus. Floating point load/store instructions are used to facilitate this scheme. The MB86901 IU generates a memory address and the FPU either sources or sinks the data.

Note that floating point loads and stores are **not** Floating Point operate (FPop) instructions.

The MB86901 Floating Point Interface includes floating point condition code, trap reporting, and handshake signals, to maintain coordination between the IU and the FPU. This interface is discussed further in Section 7.2. The MB86911 and TI SN74ACT8847 data sheets should be consulted for a complete discussion of these chips.

5. Trap and Exception Handling

The MB86901 provides three types of traps: synchronous, floating point, and asynchronous. A trap is like a random procedure call which, as a side effect, causes the processor to enter the supervisor state.

Synchronous traps are caused by an IU instruction or a floating point load/store instruction, and occur before the instruction is completed. Floating point traps are caused by a Floating Point Operate (FPop) instruction and occur before the instruction is completed. However, due to the concurrent operation of the IU and the FPU, other non-floating point instructions may have executed in the meantime. Asynchronous traps (also called interrupts) occur when an external event interrupts the processor. They are not related to any particular instruction, and occur between the execution of instructions.

5.1. Synchronous Traps

A Synchronous trap is defined to be a trap that occurs during the course of an instruction execution, prior to the alteration of any processor or system state visible to the programmer. Instructions which access memory twice (double loads and stores and atomic instructions) are the only exceptions. If multiple traps occur during one instruction, the highest priority trap is taken. Lower priority traps are ignored because the traps are arranged under the assumption that the lower priority traps persist, recur, or are meaningless due to the presence of the higher priority trap.

The ET bit in the PSR (see Section 2.5.2.1) must be set for synchronous traps to occur normally. Should a synchronous trap occur while traps are disabled, the processor halts and enters an error state. A synchronous trap may originate due to either internal or external events.

An example of an internal event is an attempt to execute an illegal instruction or to execute a privileged instruction while the processor is in the User mode. External events include asserting the $\overline{\text{MEXC}}$ (Memory EXCeption) line during a given active bus cycle, thus initiating an instruction access or data access.

5.2. Floating Point Traps

All floating point instructions are dispatched from the IU to a floating point instruction queue, in the FPU, where they execute concurrently with any non-floating point IU operations. It is therefore typical that a floating point exception occur sometime after the dispatch of the floating point instruction that caused the exception. However, a floating point exception trap is not taken until another floating point

instruction has been encountered in the IU instruction stream.

When the FPU recognizes an exception condition, it enters an "exception pending" state and signals the IU through assertion of the "FEXC" signal. It remains in this state until the IU takes the floating point exception trap, whereafter it enters the "exception mode" state. To exit this state, the floating point queue must be emptied by one or more "STDFQ" instructions.

The PC (Program Counter) that is saved during a floating point exception trap always points to a floating point instruction. However, the exception itself is always due to a previously executed floating point instruction. This instruction and the value of the PC from which it was fetched are in the floating point queue.

5.3. Asynchronous Traps

All asynchronous traps are signalled to the processor through the IRL<3:0> (Interrupt Request Level) bus. They are divided up into two categories: the maskable interrupts, and the Non-Maskable Interrupt (NMI).

The maskable interrupts must meet two critera for the processor to initiate an asynchronous trap procedure. First, the ET (enable trap) bit must be set in the PSR. Second, the IRL must be greater than the value in the Processor Interrupt Level (PIL) field of the PSR. An IRL<3:0> = 0000 value is the standby mode, and will not result in any action.

The Non-Maskable Interrupt is signalled by IRL<3:0> = 1111, and can only be disabled by turning off ET. Any asynchronous interrupts have a lower priority when arbitrated with pending synchronous traps.

5.4. Trap Addressing

The Trap Base Register (TBR) generates the exact address of a trap handling routine. When a trap occurs, the hardware writes a value into the Trap Type (**tt**) field of the TBR. This uniquely identifies the trap and serves as an offset into the table whose starting address is given by the TBA field of the TBR. The 8-bit wide **tt** field allows for 256 distinct types of traps. The lower half is used by hardware initiated traps while the upper half is dedicated to programmer initiated traps (see Ticc instruction, Section 3.3.4).

The MB86901 currently utilizes 26 out of the 128 hardware trap types.

5.5. Trap Priorities

Table 5.1 shows the trap types, priorities, and assignments. Note that "**tt**" is not affected by the reset trap.

5.6. Trap Processing

A trap causes the following activities:

- It disables traps (ET = 0)
- It copies the S field of the PSR into the PS field and then sets the S field to 1.
- It decrements the CWP by 1, modulo the number of implemented windows.
- It saves the PC and nPC into r[17] and r[18], respectively, of the new window.
- It sets the "**tt**" field of the TBR to the appropriate value

Table 5.1 Trap Priorities and tt Assignments

Trap	Priority	tt
Reset	1	—
Instruction Access Exception	2	1
Illegal Instruction	3	2
Privileged Instruction	4	3
Floating Point Disabled	5	4
Window Overflow	6	5
Window Underflow	7	6
Memory Address Not Aligned	8	7
Floating Point Exception	9	8
Data Access Exception	10	9
Tag Overflow	11	10
Trap Instruction	12	128-255
Interrupt Level 15	13	31
Interrupt Level 14	14	30
Interrupt Level 13	15	29
Interrupt Level 12	16	28
Interrupt Level 11	17	27
Interrupt Level 10	18	26
Interrupt Level 9	19	25
Interrupt Level 8	20	24
Interrupt Level 7	21	23
Interrupt Level 6	22	22
Interrupt Level 5	23	21
Interrupt Level 4	24	20
Interrupt Level 3	25	19
Interrupt Level 2	26	18
Interrupt Level 1	27	17

- If the trap is not a reset, it writes the PC with the contents of TBR, and the nPC with the contents of TBR+ 4. If the trap is a reset, it loads the PC with 0 and nPC with 4.

The MB86901, like all SPARC processors, saves the volatile S field into the PSR itself and the remaining fields are either altered in a reversible way (ET and CWP), or should not be altered in a trap handler until the PSR has been saved into memory. To restore the PSR S-bit and PC/nPC, a trap handler should include "JMPL" and "RETT" as the last two instructions.

5.7. Interrupt Detection

The MB86901 will latch the IRL <3:0> bus at the rising edge of every clock cycle. If the output of this latch equals the current value of the IRL bus by the following cycle, an asynchronous trap may be initiated if the "ET" and "IPL" criteria are met. The best case for an interrupt response time is 3 cycles. This assumes the request arriving within the IU trap arbitration logic at a time when an instruction has just completed. The worst case is asserting an interrupt during the execution of a 4 cycle double store. That would add 3 cycles to the overhead, thus yielding a response time of 6 cycles. Neither of these estimates include any additional delay caused by assertion of MHOLD/FHOLD or interrupts being disabled.

5.8. Trap Definitions

The following discussion describes the 86901 traps and the conditions which cause them.

instruction access exception
This trap occurs when the $\overline{\text{MEXC}}$ signal line has been strobed during an instruction fetch. It may be used to indicate error conditions such as page faults and access violation.

illegal instruction
This trap occurs when:

(1) the UNIMP instruction is encountered, or

(2) an unimplemented instruction which is not an FPop is encountered, or

(3) when an instruction is fetched which, if executed, would result in an illegal processor state (e.g. writing an illegal CWP into the PSR).

Unimplemented FPop instructions generate **fp exception** traps.

privileged instruction
This trap occurs when a privileged instruction is encountered while the processor is in User mode (S = 0 in PSR).

fp disabled
This trap occurs when a Floating Point Operate, Floating Point Branch, or Floating Point Load or Store instruction is encountered while the EF bit in the PSR is equal to 0, or no FPC is present ($\overline{\text{FP}}$ = 1).

window overflow
This trap occurs when a SAVE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

window underflow
This trap occurs when a RESTORE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

mem address not aligned
This trap occurs when a load, store or JMPL instruction would, if executed, generate a memory address or a new PC value that is not properly aligned.

fp exception
This trap occurs when the FPU is in exception pending state and a floating point instruction (FPop, floating point load/store, FBbcc) is encountered.

data access exception
This trap occurs when the $\overline{\text{MEXC}}$ signal line has been strobed during an instruction fetch. It may be used to indicate error conditions such as page faults and access violation.

tag overflow
This trap occurs when a TADDccTV or a TSUBccTV instruction is executed which causes the overflow bit of the integer condition codes to be set.

trap instruction
This trap occurs when a taken Ticc instruction is executed.

interrupt level<3:0>
A trap of this category occurs during assertion of a vector on the IRL<3:0> bus. The lower four bits of tt are equivalent to the vector asserted on the IRL bus.

6. Reset

A reset trap occurs when the IU leaves reset mode and enters execute mode. This is controlled by the RESET signal. The IU enters reset mode upon asserting RESET, and enters execute mode when negating RESET. Except in one situation, reset does not change the value of the tt field of the TBR; the exception is when a return from trap instruction is executed while traps are not enabled and the processor is not in supervisor mode (see description of return from trap instruction in 3.3.5). Also, a reset trap causes the IU to begin execution at location 0, regardless of the value of the TBR.

Reset traps set the PSR S bit to 1 and the ET bit to 0. All other PSR fields, and all other registers retain their values from the last execute mode, except that on power up they are undefined.

7. Bus Signals

The MB86901 features several buses and associated handshaking and control signals through which the processor communicates with system memory, peripheral devices, a floating point unit, and support logic such as bus arbiters and interrupt controllers. These buses and signals can be classified into two major groups: the System Interface, and the Floating Point Coprocessor Interface. A third group, the Test Interface, is used for factory testing of the processor.

Figure 7.1 Processor Signals

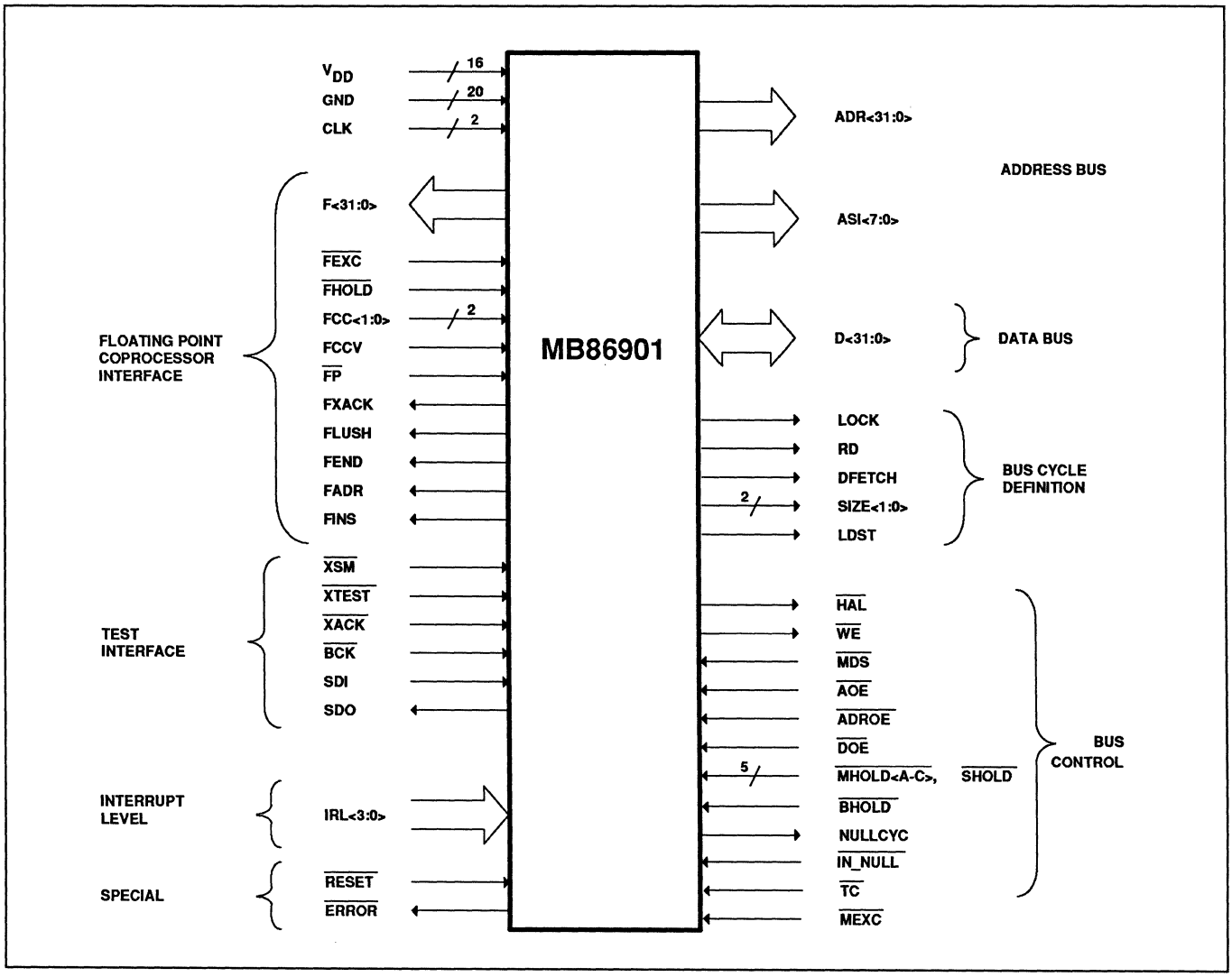


Table 7.1 Bus Signal Descriptions

Signal	Type	Description
CLK1	I	CLOCK 1. This is typically a 75/25% duty cycle clock which is high during the first three quarters of each processor cycle, and low during the last quarter.
CLK2	I	CLOCK 2. This is typically a 25/75% duty cycle clock which is low during the first quarter of each processor cycle, and high during the last three quarters.
ADR<31:0>	O	ADDRESS BUS. This is the address bus which is used during all memory transactions. It is unlatched and therefore only valid at the very beginning of a bus cycle. It is tristated while $\overline{\text{ADROE}}$ is high.
ASI<7:0>	O	ADDRESS SPACE IDENTIFIER. The Address Space Identifier identifies accessed address space as follows: **ASI<7:0> — Address Space** 0–7 — Implementation Definable 8 — User Instruction Space 9 — Supervisor Instruction Space 10 — User Data Space 11 — Supervisor Data Space 12–255 — Implementation Definable ASI<7:0> are forced to a high impedance while $\overline{\text{AOE}}$ is high.
D<31:0>	I/O	DATA BUS. This is a 32-bit data bus which is used for instruction and data transfer. D<31:0> are forced to a high impedance while $\overline{\text{DOE}}$ is high.
$\overline{\text{HAL}}$	O	HOLD ADDRESS LATCH. The $\overline{\text{HAL}}$ signal is used to inhibit latching by the ADR<31:0> external latches. $\overline{\text{HAL}}$ is asserted during instruction pipeline interlock activity, during execution of some multiple cycle instructions, and during assertion of an MHOLD signal.
$\overline{\text{WE}}$	O	WRITE ENABLE. The $\overline{\text{WE}}$ signal is asserted during Store operations and during the store phase of Atomic Load/Store operations to indicate that data to be stored is present on the Data Bus. Assertion of $\overline{\text{WE}}$ is inhibited while $\overline{\text{FHOLD}}$ is active, and is terminated when $\overline{\text{FEXC}}$ or $\overline{\text{MEXC}}$ becomes active.
RD	O	READ. The RD signal remains asserted during Load operations to indicate that the processor will read data, and is released throughout Store and Atomic Load/Store operations to indicate that the processor will write data. RD is forced to a high impedance while $\overline{\text{AOE}}$ is high.

Table 7.1 Bus Signal Descriptions (cont.)

Signal	Type	Description
DFETCH	O	DATA FETCH. The DFETCH signal is asserted for one cycle at the beginning of data transfer operations to indicate that data will be transferred. DFETCH remains inactive during instruction fetches and during Floating Point Operate instruction transfers. Assertion of NULL_CYC forces release of DFETCH.
SIZE<1:0>	O	DATA SIZE. These signals identify transferred data as follows: SIZE<1:0> — Data Size 0 — Byte 1 — Halfword 2 — Word 3 — Double Word* * (LDD, LDDF, STD, STDF double-word instructions) The Data Size signals are forced to a high impedance while $\overline{AOE}$ is high.
LOCK	O	BUS LOCK. The LOCK signal is asserted by the processor during Store, Load Double, Store Double, and Atomic Load/Store operations to indicate to other bus masters that the System Interface is busy with multicycle operations. Bus masters must not try to gain control of the system bus during the cycle following assertion of LOCK. LOCK is forced to high impedance while AOE is high.
$\overline{MDS}$	I	MEMORY DATA STROBE. The $\overline{MDS}$ strobe is asserted for one or more cycles while an MHOLD<A–C> signal is asserted during load and instruction fetch operations to indicate that valid data will be asserted on the Data Bus. The data must be present on the Data Bus one at least one cycle before $\overline{MDS}$ is released.
$\overline{ADROE}$	I	ADDRESS OUTPUT ENABLE. This active low signal controls the output drivers of the ADR<31:0> bus. The drivers are enabled when $\overline{ADROE}$ is low, and disabled to a high impedance when $\overline{ADROE}$ is high. $\overline{AOE}$ should be asserted while the BHOLD signal is active to allow another bus master to gain control of the affected bus signal lines.
$\overline{AOE}$	I	ALTERNATE OUTPUT ENABLE. This active low signal controls the output drivers of the ASI<7:0>, RD, SIZE<1:0>, and LDST signals.The drivers are enabled when $\overline{AOE}$ is low, and disabled to a high impedance when $\overline{AOE}$ is high. $\overline{AOE}$ should be asserted while the BHOLD signal is active to allow another bus master to gain control of the effected bus signal lines.
$\overline{DOE}$	I	DATA OUTPUT ENABLE. This active low signal controls the output drivers of the D<31:0> data bus. The drivers are enabled when $\overline{DOE}$ is low, and disabled to a high impedance when $\overline{DOE}$ is high. $\overline{DOE}$ should be asserted while a Memory Hold signal is active to allow another bus master to gain control of the Data Bus.

Table 7.1 Bus Signal Descriptions (cont.)

Signal	Type	Description
$\overline{\text{MHOLD<A–C>}}$ $\overline{\text{SHOLD}}$	I	MEMORY HOLD. Assertion of one or more of the four Memory Hold signals freezes the processor instruction pipeline, and holds the processor bus signals in a wait state. A Memory Hold signal can be asserted to introduce wait states in order to accommodate slow memory or I/O response, and cache misses. All of the Memory Hold signals are OR'd by processor logic, and function identically. Four Memory Hold inputs are provided to allow flexibility in system design. The Memory Hold signals should be latched externally, and should be valid prior to CLK1 falling edges (see timing diagrams).
$\overline{\text{BHOLD}}$	I	Bus Hold freezes the processor pipeline and allows another bus master to gain control of the data bus. $\overline{\text{BHOLD}}$ is active low. It should be latched externally and be valid prior to the CLK1 falling edge.
$\overline{\text{IRL<3:0>}}$	I	INTERRUPT REQUEST LEVEL. External system logic, typically an interrupt controller, reports and identifies system interrupts via the $\overline{\text{IRL<3:0>}}$ bus as follows: **IRL<3:0> — Interrupt Request** 0 — None 11–4 — Maskable 15 — Non Maskable Maskable interrupts are recognized and serviced only if the ET bit is set in the PSR, and the IRL<3:0> value is greater than the PSR PIL<11:8> value. The Nonmaskable Interrupt is always recognized while the ET bit is set. External logic must prioritize interrupts in cases of simultaneous interrupt requests, and must latch and assert the IRL value corresponding to the highest pending interrupt until the interrupt request is cleared by software. $\overline{\text{IRL<3:0>}}$ must be asserted synchronously with respect to CLK1.
$\overline{\text{RESET}}$	I	RESET. Assertion of $\overline{\text{RESET}}$ initializes the processor as follows: (1) Supervisor mode is selected (S = 1 in PSR). (2) The PC is set to 0. (3) The NPC is set to 4. Other register fields are either undefined (power on), or retain their state at the time $\overline{\text{RESET}}$ is asserted. The processor fetches the first instruction from address 0 at the first rising edge of CLK1 following release of $\overline{\text{RESET}}$.

Table 7.1 Bus Signal Descriptions (cont.)

Signal	Type	Description
$\overline{TC}$	I	$\overline{TC}$ determines the behavior of the IFLUSH instruction. When $\overline{TC}$ is high, IFLUSH executes like a NOP with no side effects. When $\overline{TC}$ is low, IFLUSH causes an unimplemented instruction trap.
$\overline{MEXC}$	I	MEMORY EXCEPTION. The memory or cache controller asserts $\overline{MEXC}$ to report a memory error. The processor recognizes $\overline{MEXC}$ during the cycle following its assertion.
$\overline{ERROR}$	O	ERROR. The processor asserts $\overline{ERROR}$ to report to the system that it has halted in the error state as a result of a synchronous trap it has encountered while ET = 0 in the PSR. $\overline{RESET}$ must be asserted for recovery from the error-state.
NULL_CYC	O	NULLIFY CURRENT CYCLE. The processor asserts this signal to indicate that the address currently latched in the external memory latches is not a valid address. This occurs when the processor fetches an instruction (such as the target of an untaken branch) which it doesn't use. The processor also asserts NULL_CYC when $\overline{FHOLD}$ or an $\overline{MHOLD}$<A–C> signal is asserted. NULL_CYC should be used by cache logic to eliminate false cache misses due to invalid addresses. Assertion of NULL_CYC forces release of DFETCH.
$\overline{IN\_NULL}$	I	INHIBIT NULL CYCLE ENABLE. $\overline{IN\_NULL}$ is an asynchronous input which system logic can use to control the assertion of NULL_CYC. When $\overline{IN\_NULL}$ = 1, the processor can assert NULL_CYC; when $\overline{IN\_NULL}$ = 0, NULL_CYC is forced low.
LDST	O	ATOMIC LOAD/STORE. This signal is asserted to indicate that an atomic Load or Store operation is in progress. LDST is forced to a high impedance while $\overline{AOE}$ is high.
F<31:0>	O	FLOATING POINT INSTRUCTION/ADDRESS BUS. The processor transfers Floating Point Operate instructions and their addresses to the floating point unit via this dedicated 32-bit floating point bus. The processor first transfers an instruction, then transfers the instruction address the following cycle (see MB86911 data sheet).
$\overline{FEXC}$	I	FLOATING POINT EXCEPTION. The floating point unit asserts $\overline{FEXC}$ to report a floating point error. $\overline{FEXC}$ must remain asserted until the processor recognizes the trap and asserts FXACK.

Table 7.1 Bus Signal Descriptions (cont.)

Signal	Type	Description
$\overline{\text{FHOLD}}$	I	FLOATING POINT HOLD. Assertion of $\overline{\text{FHOLD}}$ halts the processor. The floating point unit asserts $\overline{\text{FHOLD}}$ when it detects data dependencies which require that the processor be halted. The processor resumes execution when the floating point unit releases $\overline{\text{FHOLD}}$.
$\overline{\text{FCC<1:0>}}$	I	FLOATING POINT CONDITION CODE. These are floating point condition code signals which are asserted by the floating point unit to indicate results of Floating Point Operate instruction executions as follows: **FCC<1:0> — Result Description** 0 — operand 1 = operand 2 1 — operand 1 < operand 2 2 — operand 1 > operand 2 3 — unordered relationship
FCCV	I	FLOATING POINT CONDITION CODE VALID. The floating point unit asserts FCCV to indicate that the floating point condition code asserted on $\overline{\text{FCC<1:0>}}$ is valid.
$\overline{\text{FP}}$	I	FLOATING POINT UNIT PRESENT. This input is active low when a floating point coprocessor is present on the floating point interface bus. The processor checks this signal when it encounters a Floating Point Operate (FPop) or a floating point Load or Store instruction. The IU acts as follows when such an instruction is encountered. When $\overline{\text{FP}}$ is asserted, the IU will dispatch the instruction to the FPC, depending on the state of the EF bit in the PSR (see Section 2.5.2). When $\overline{\text{FP}}$ is negated, the IU will always initiate an FP disabled trap, after which, a floating point library routine can emulate the FPop.
FXACK	O	FLOATING POINT EXCEPTION ACKNOWLEDGE. The processor asserts FXACK in response to a Floating Point Exception trap ($\overline{\text{FEXC}}$) to acknowledge that it has recognized the trap. The floating point unit should release $\overline{\text{FEXC}}$ when FXACK is asserted.
FLUSH	O	FLUSH. The FLUSH signal, when asserted, forces the floating point unit to abort (flush) the instruction in the current floating point unit write cycle. FLUSH does not affect instructions in the floating point unit instruction queue.
FEND	O	FLOATING POINT END. The processor asserts this signal to indicate to the floating point unit that the last cycle of the current processor Floating Point Operate instruction transfer is in progress. This allows the floating point unit to synchronize its operation with the processor.

Table 7.1 Bus Signal Descriptions (cont.)

Signal	Type	Description
FADR	O	FLOATING POINT ADDRESS. The processor asserts this signal to indicate that a valid address is present on the F<31:0> bus. The floating point unit should use FADR to latch floating point addresses.
FINS	O	FLOATING POINT INSTRUCTION. The processor asserts FINS to indicate that a valid instruction is present on the F<31:0> bus.The floating point unit should use FINS to latch floating point instructions.
$\overline{\text{XACK}}$, BCK	I I	SCAN CLOCKS. The scan clocks are used only during processor scan testing, are intended for factory use, and must be high during normal operation.
$\overline{\text{XSM}}$, $\overline{\text{XTEST}}$	I I	SCAN CLOCKS ENABLE. The $\overline{\text{XSM}}$ and $\overline{\text{XTEST}}$ signals are asserted only during processor testing to disable the system clocks and enable the Scan Clocks, is intended for factory use, and must be high during normal operation.
SDI	I	SCAN DATA INPUT. This is the serial data pattern input signal which is used only during processor scan testing. It is intended for factory use, and must be high during normal operation.
SDO	O	SCAN DATA OUTPUT. This is the serial data pattern output signal which is used only during processor scan testing. It is intended for factory use, and may be left unconnected.

The $\overline{\text{AOE}}$, $\overline{\text{DOE}}$, and $\overline{\text{IN\_NULL}}$ input signals are asynchronous, disabling output drivers or inhibiting the NULL_CYC signal immediately when released, without regard to clocks. This allows fast transfer of essential bus signals to other bus masters ($\overline{\text{AOE}}$, $\overline{\text{DOE}}$), and fast nullification of system addresses ($\overline{\text{IN\_NULL}}$) without the "wait" for clock edges that would otherwise be required.

All other processor signals are synchronous, allowing the fast addressing and data transfer possible only in a synchronous system. $\overline{\text{MHOLD}}$<A-C> are referenced to the falling edge of CLK1; all other signals are referenced to the rising edge of CLK1.

The MB86901 interface signals are shown in Figure 7.1, Processor Signals, and are fully described in Table 7.1, Bus Signal Descriptions. The following sections contain brief descriptions of the signals.

7.1. System Interface

The System Interface is comprised of signals and buses which interface to system logic other than the floating point unit. These include the Address Bus, the Data Bus, the Bus Cycle Definition signals, the Bus Control signals, the Interrupt Request Level bus, and the $\overline{\text{RESET}}$ and $\overline{\text{ERROR}}$ Special signals (see Figure 7.1).

The System Interface protocol supports multiple bus masters, allows wait states to accommodate slow memory and I/O, and facilitates cache implementation.

The separate, non-multiplexed Address and Data buses are designed for single-cycle operation without the design complexities inherent in the multiplexed address/data bus used in other processors. The Interrupt Level Bus allows fast response to system interrupts, and increases the effective bandwidth of the Data Bus by allowing it to be used for

instruction and data transfer only. The extensive offering of Bus Cycle Definition and Bus Control signals in the System Interface allows a high degree of design flexibility without performance compromise.

These and other System Interface features combine with the MB86901 RISC architecture to give the processor its high throughput.

Address Bus

The Address Bus includes all signals necessary to locate instructions, data, and I/O. These include ASI<7:0> which select address spaces and are used by memory management protection logic which may be implemented, and the ADR<31:0> address bus signals which identify particular locations in selected address spaces.

The asi signals allow selection of up to 256 address spaces, with each address space containing as many as 4 gigabytes, for a maximum combined accessible space of 1024 gigabytes.

Four of the address spaces are dedicated according to mode (user or supervisor) and type of access (instruction or data). 252 of the address spaces are implementation definable and are selected by the "asi" field of the Load From Alternate Space, Store Into Alternate Space, and Atomic Load-Store Into Alternate Space instructions.

The address signals select individual words in accessed address spaces.

Data Bus

The Data Bus transfers all instructions and data between the processor, and memory and I/O. It is a non-multiplexed, 32-bit bus capable of transferring one word each cycle.

Bus Cycle Definition Signals

The Bus Cycle Definition signals are asserted during bus operations to identify to the system the types of operations in progress as follows:

- RD identifies operations as loads (reads) or stores (writes).
- LDST identifies Atomic Load-Store operations.
- LOCK identifies the multi-cycle Load Double and Store Double operations.
- DFETCH identifies data transfer operations.
- SIZE<1:0> identifies transferred data as bytes, halfwords, or words.

Bus Control Signals

The Bus Control signals directly control the state of the System Interface signals, directly control system logic associated with the System Interface, and indirectly control bus operations by supporting bus protocol as follows:

- $\overline{\text{WE}}$ indicates that data to be stored is present on the Data Bus.
- $\overline{\text{MDS}}$ indicates that data to be loaded is present on the Data Bus.
- $\overline{\text{HAL}}$ inhibits latching by the address latches.
- $\overline{\text{MHOLD<A-C>}}$ hold the processor in a wait state.
- NULL_CYC indicates that the address in the external address latches is not a new valid address, or that $\overline{\text{FHOLD}}$ or one of the $\overline{\text{MHOLD<A-C>}}$ signals is asserted.
- $\overline{\text{IN\_NULL}}$ gates assertion of NULL_CYC.
- $\overline{\text{TC}}$ indicates the presence of a cache.
- $\overline{\text{MEXC}}$ indicates a memory error.
- $\overline{\text{AOE}}$ enables the ASI<7:0>, RD, SIZE<1:0>, LOCK, and LDST output drivers.
- $\overline{\text{DOE}}$ enables the D<31:0> output drivers.

Interrupt Request Level Bus

The IRL<3:0> Interrupt Request Level bus is used by system logic to report both maskable and non-maskable interrupt service requests, and their level (see Section 5.1.1).

Special Signals

$\overline{\text{RESET}}$ and $\overline{\text{ERROR}}$ are special signals which are not normally asserted during operation.

$\overline{\text{RESET}}$ is asserted by system logic to initialize the processor, then is released to force boot program execution at location 0 (see Section 6). $\overline{\text{RESET}}$ is always asserted at power-on, but is usually asserted during operation only in response to a catastrophic system failure.

$\overline{\text{ERROR}}$ is asserted by the processor to indicate that it has halted in the error state in response to a trap occurrence while ET = 0 in the PSR. $\overline{\text{RESET}}$ must be asserted for recovery from the error state.

7.1.1. Basic Timing

Basic processor timing is shown in Figure 7.2. The figure shows fetching of instruction I1, followed by fetches of sequential instructions I2 and I3, followed by instruction I1 data transfer (such as a data read for a Load instruction), followed by instruction I4 fetch.

This timing is typical, showing instruction fetches or data transfer each cycle, with no delays (wait cycles) due to cache misses or memory exceptions.

The notation with format "S_I" above each clock cycle identifies the instruction pipeline stage in which a particular instruction is being processed (S), and the instruction or instruction operation (I) as follows:

Pipeline Stage	Instruction
F = Fetch Stage	IX = Instruction X
D = Decode Stage	LD = Load Operation
E = Execute Stage	ST = Store Operation
W = Write Stage	T = Trap Dispatch Table Instruction
WH = Write Stage Hold	

Note that instruction I1 is decoded and executed in the instruction pipeline while instructions which follow are fetched.

All instructions are pipelined in this manner.

7.1.2. Basic Data Transfer Timing

Figures 7.3–7.7 show timing for the Load, Store, Load Double Word, Store Double Word, and Atomic Load-Store operations. These operations are identified by the assertion of DFETCH and ASI<7:0> at the beginning of each operation to indicate that data, rather than an instruction, is to be transferred via the D<31:0> data bus.

The Load operation completes in one cycle as shown in Figure 7.3.

RD and $\overline{\text{WE}}$ remain released during the operation to indicate that the processor is to read data.

The Store Operation, shown in Figure 7.4, completes in two cycles. The processor releases RD during the first cycle to indicate that it will write data, then asserts $\overline{\text{WE}}$ during the second cycle to indicate that the data to be written is asserted on the data bus. The processor also asserts LOCK to indicate that a multi-cycle operation is in progress, and HAL and NULL_CYC to disable the External Instruction Pipeline and to indicate that the second cycle is "nulled," since it is not a new operation.

The Load Double Word operation (Figure 7.5) is essentially two back-to-back load operations. The processor asserts LOCK during the second cycle to indicate that the operation is multi-cycle.

The Store Double Word operation (Figure 7.6) is essentially two back-to-back store operations. The second store completes in one cycle rather than two, however, so that the entire operation completes in three cycles. Once again HAL, LOCK, and NULL_CYC are asserted as in the Store case, with LOCK held asserted for two cycles to indicate a three-cycle operation.

The Atomic Load-Store operation (Figure 7.7) is essentially nondivisible—a load followed by a store. The processor asserts LDST to identify the operation as an Atomic Load-Store.

Figure 7.2 Basic Timing (Cache Hit)

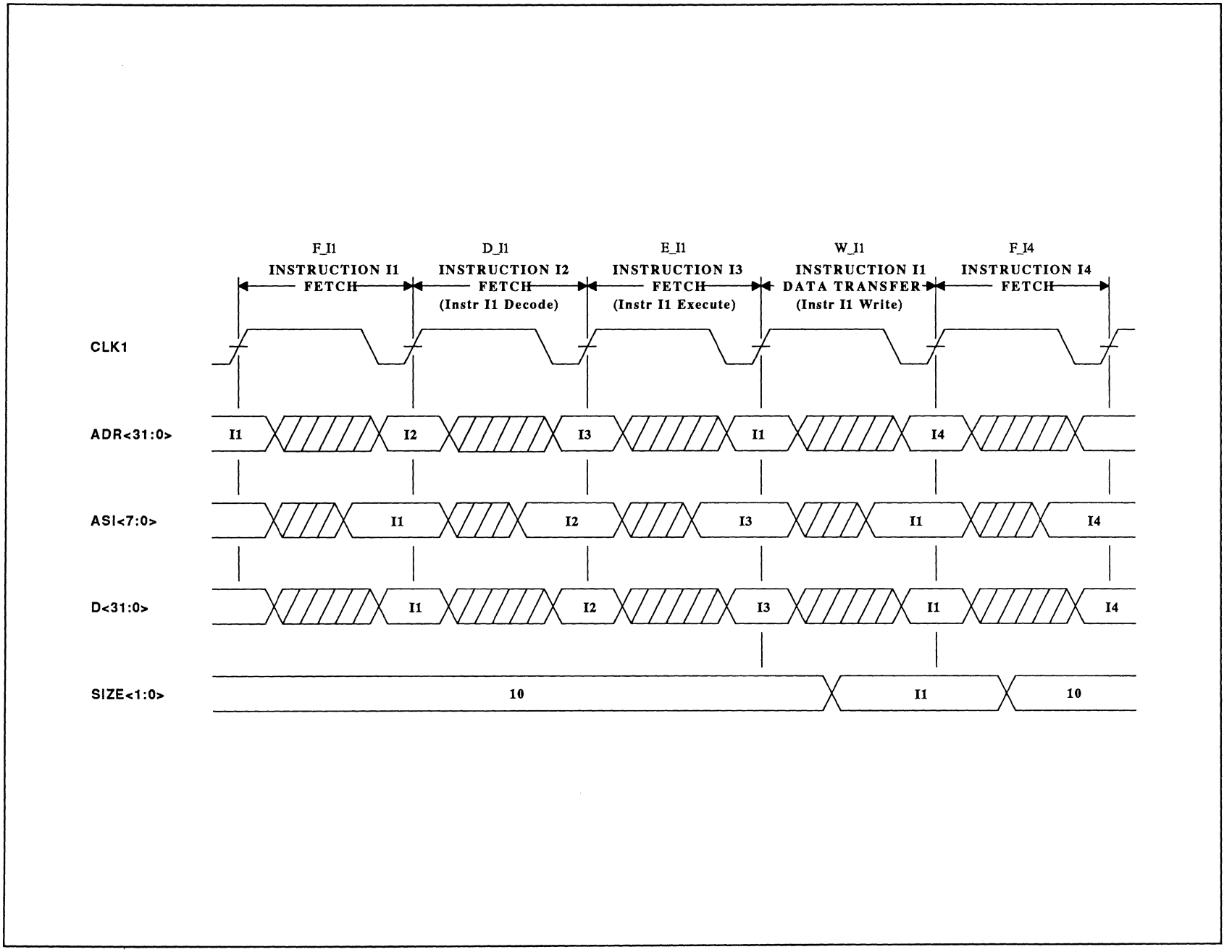


Figure 7.3 Load (Cache Hit)

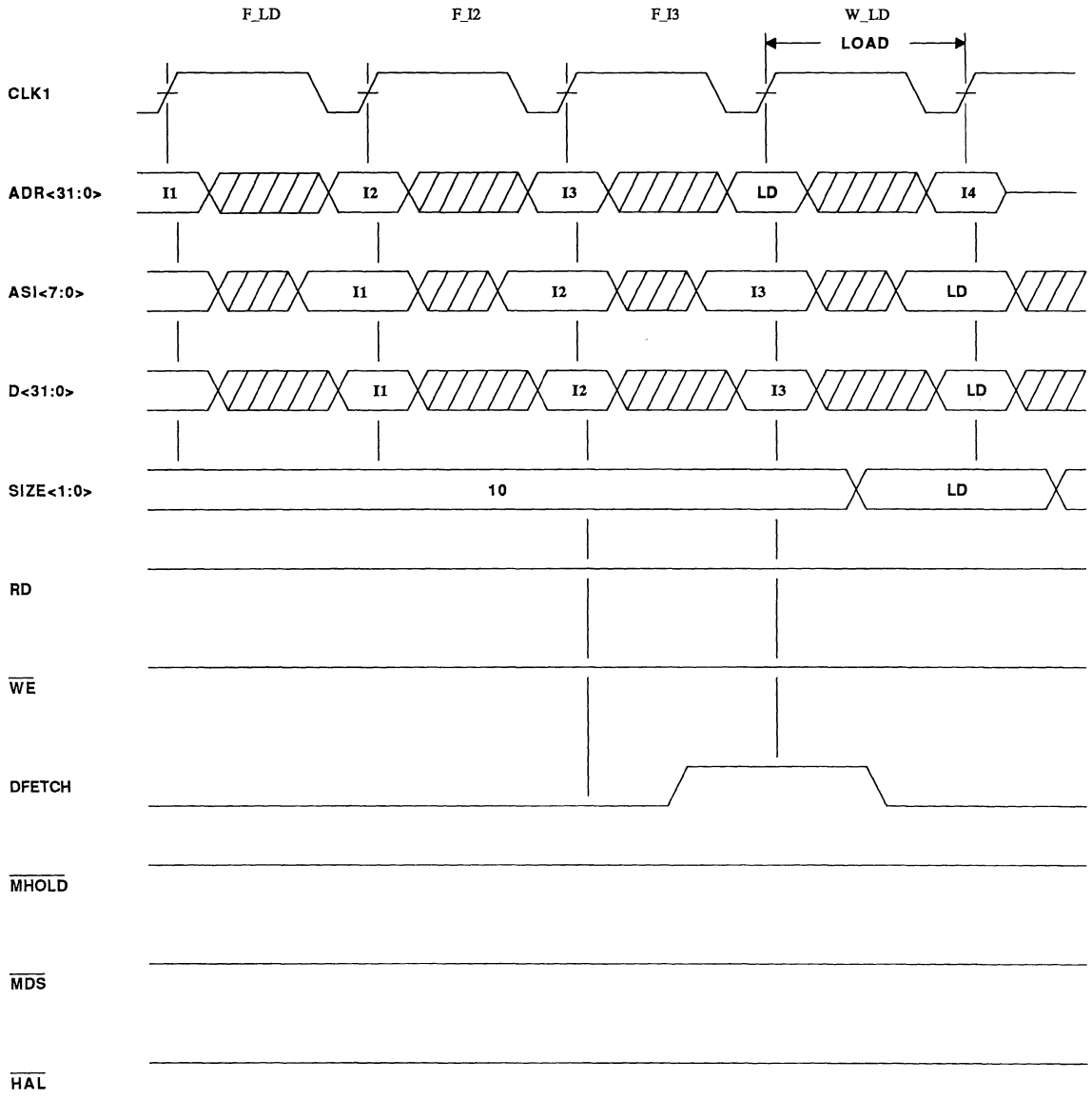


Figure 7.4 Store (Cache Hit)

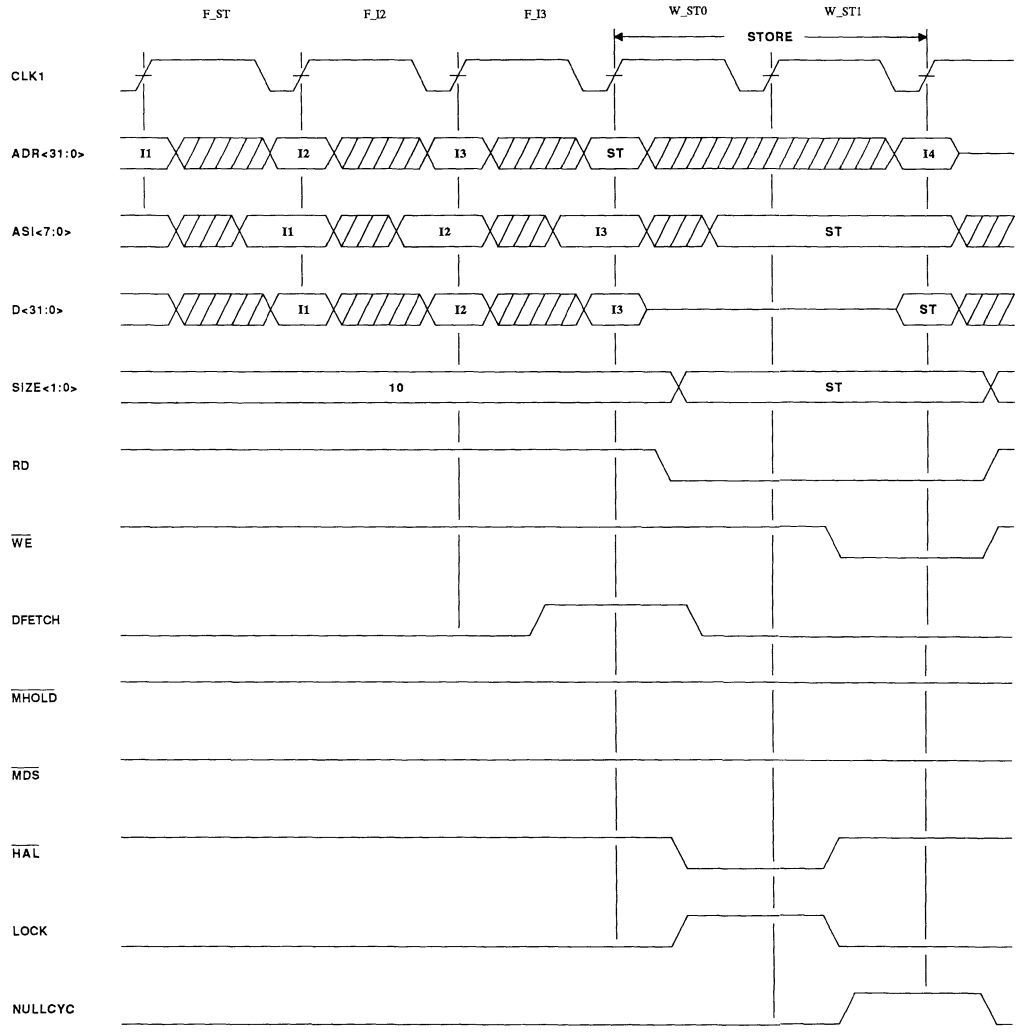


Figure 7.5 Load Double Word (Cache Hit)

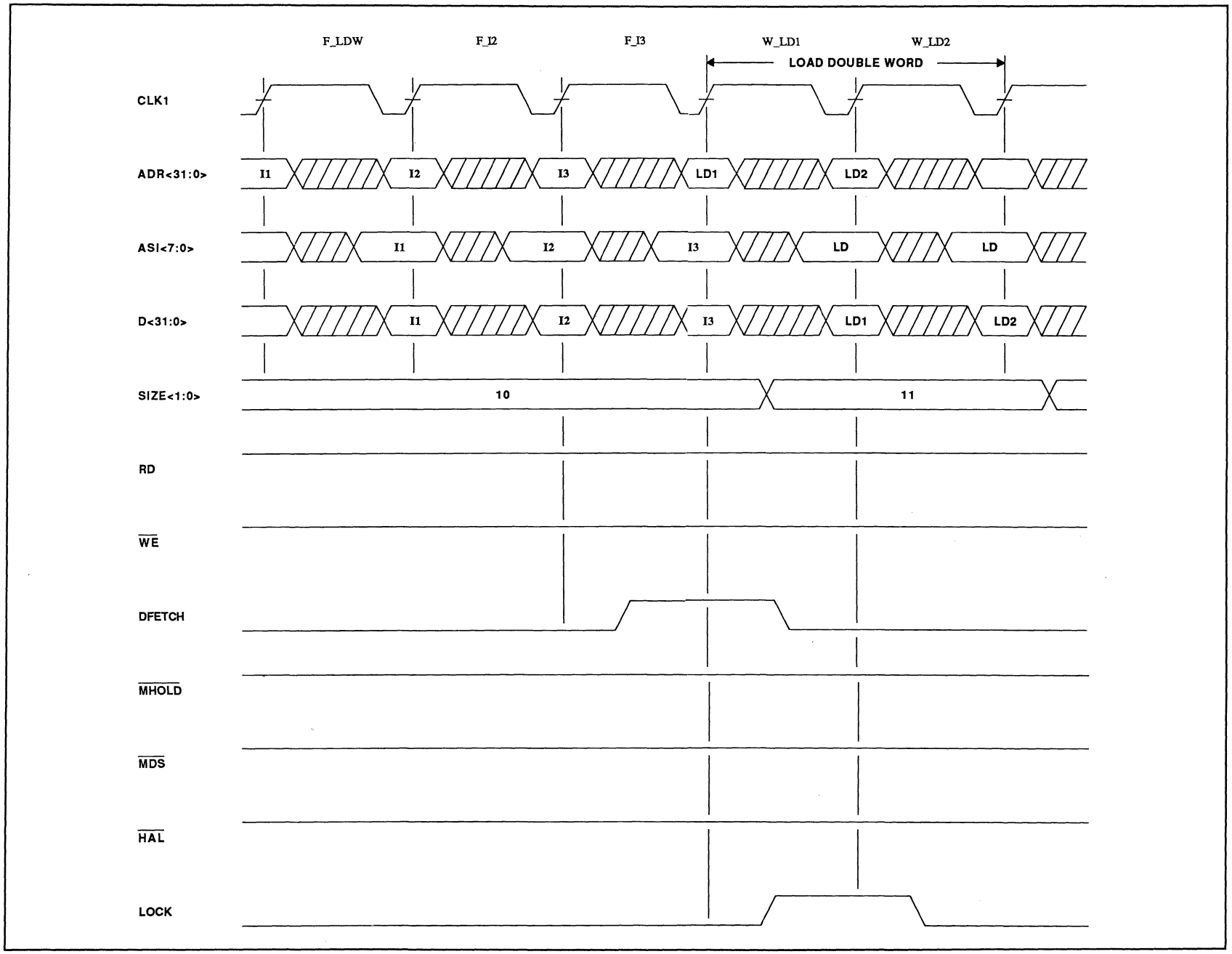


Figure 7.6 Store Double Word (Cache Hit)

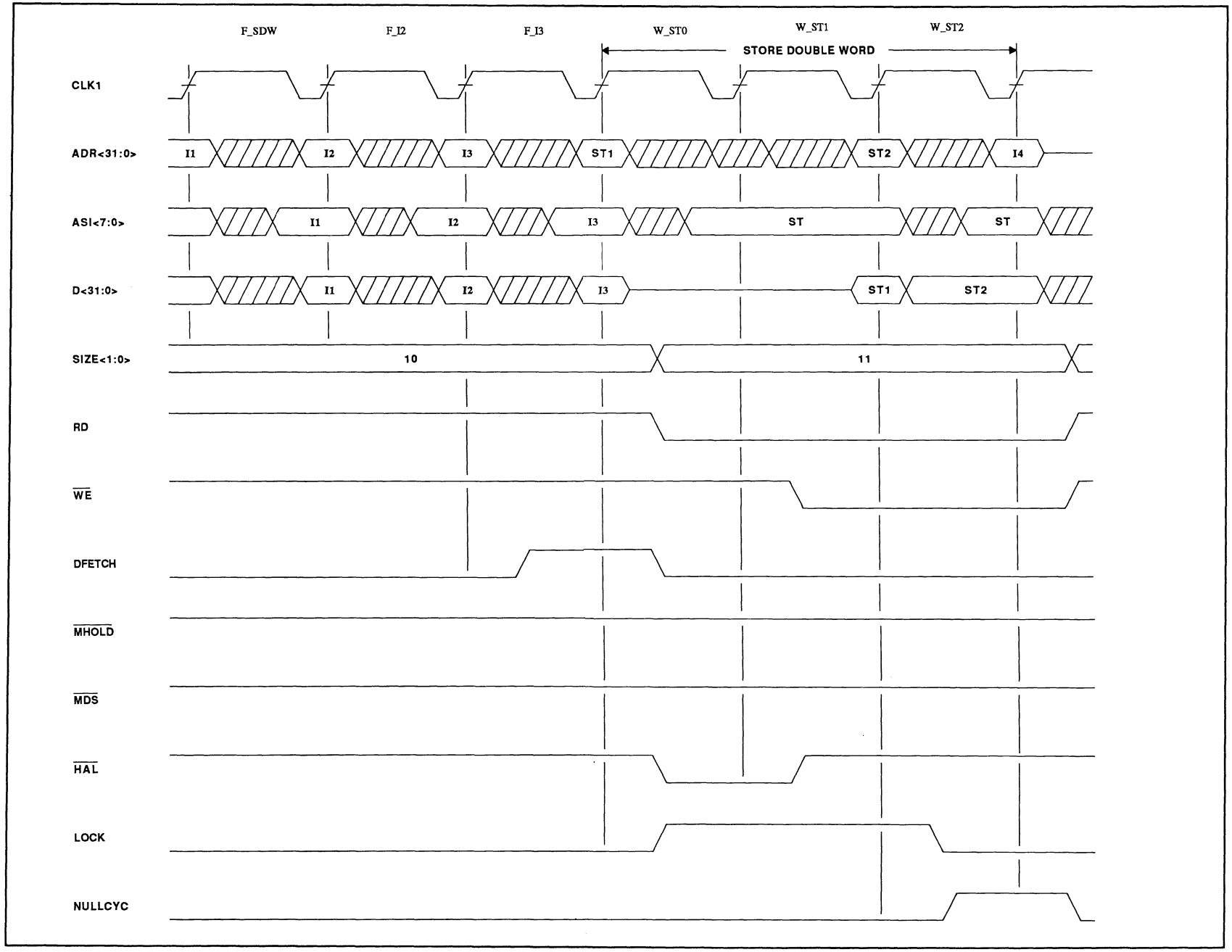
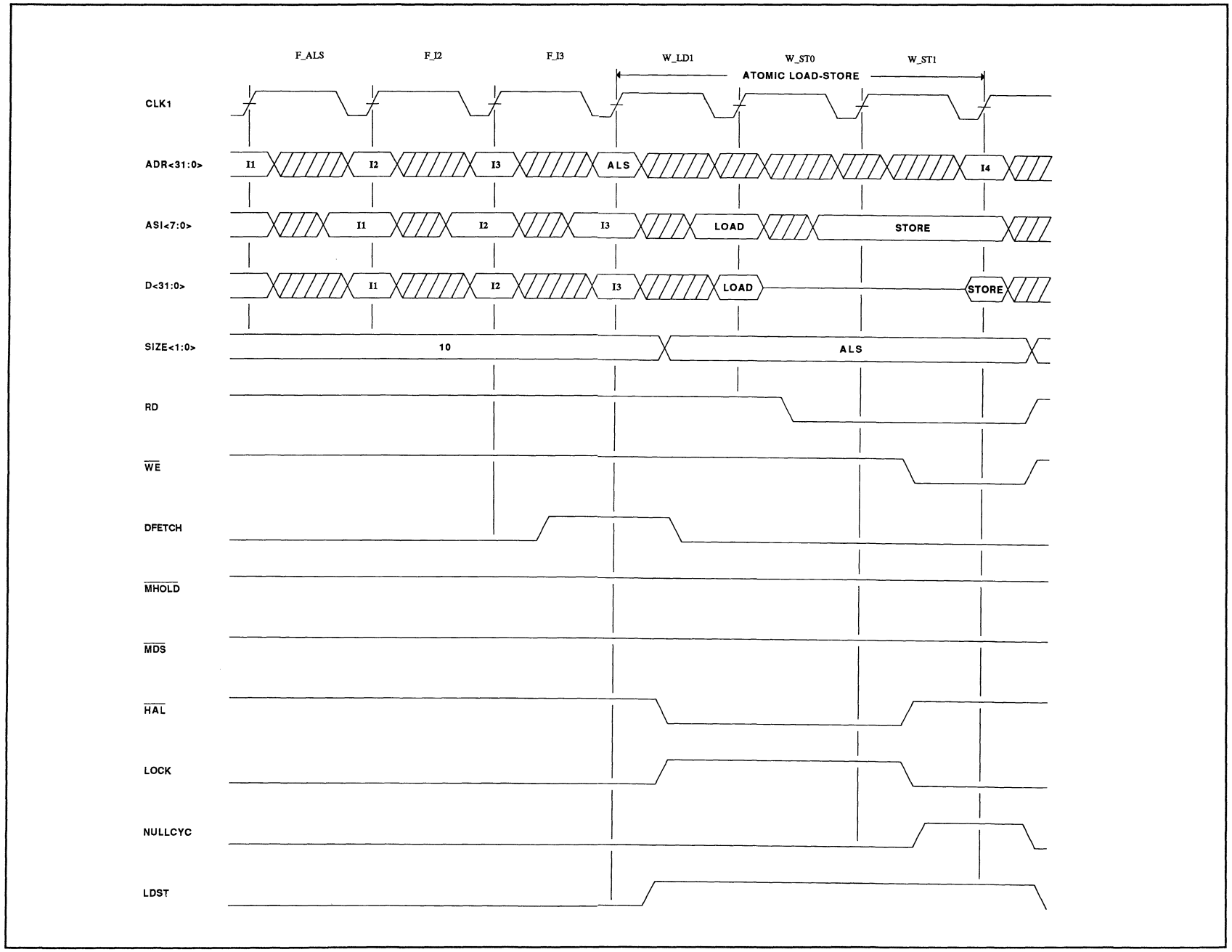


Figure 7.7 Atomic Load-Store (Cache Hit)



7.1.3. Cache Miss Timing

Figures 7.8–7.12 show timing which results from cache misses.

The processor first tries to access instructions/data in the system cache, if a cache is implemented. If an accessed instruction/data is in the cache, a cache hit occurs, and the operation proceeds with no delays as shown in Figures 7.2–7.7.

If an accessed instruction/data is not in the cache, a cache miss occurs, and the operation proceeds with delays as shown in Figures 7.8–7.12.

System logic introduces delay during an operation by asserting MHOLD as shown in the figures, which holds the processor in a pause state until released.

Note that when MHOLD is asserted, the processor has advanced to the next instruction. External latches must therefore capture the bus signal states which correspond to the "miss" instruction (see External Address Pipeline, Section 8.2) for the required memory access.

The system logic strobes MDS during a "held" instruction fetch or load operation to indicate to the processor that the instruction or data is present on the data bus, then releases MHOLD to complete the operation. During a store operation, the logic simply releases MHOLD to indicate that the destination device (memory or I/O) has latched the data.

Other signal timing is similar to the cache hit cases.

7.1.4. Memory Exception Timing

Memory exception timing for various bus operations is shown in Figures 7.13–7.16.

The operations begin normally. System logic then asserts MHOLD to hold the processor, asserts MEXC to report the memory exception, then releases both MHOLD and MEXC as shown in the figures to allow the processor to begin exception servicing. T1 and T2 are Trap Dispatch Table instructions which vector the processor to the appropriate trap handler.

7.1.5. Special Timing

Bus Request, Error/Reset, and asynchronous trap (interrupt) timing are shown in Figures 7.17–7.19, respectively.

A bus arbiter or bus master gains control of the system bus by asserting BHOLD to hold the processor in a wait state, and releasing AOE, ADROE, and DOE to force essential processor outputs to a high impedance as shown in Figure 7.17. The new bus master drives the three-stated lines while the processor output enable signals are released.

The bus arbiter releases BHOLD and the processor output enable signals once the bus master has finished operations, thereby allowing the processor to resume execution.

ERROR and RESET timing are shown in Figure 7.18. Note that RESET must be held asserted for a minimum of ten processor cycles to assure complete processor initialization.

Asynchronous interrupt timing is shown in Figure 7.19. IRL<3:0> = 0 indicates no interrupt. Any other IRL<3:0> state indicates an interrupt which may be recognized according to the value of the Processor Interrupt Level in the PSR (see Section 2.5.2).

IRL<3:0> must be held asserted until cleared by software.

Figure 7.8 Instruction Fetch (Cache Miss)

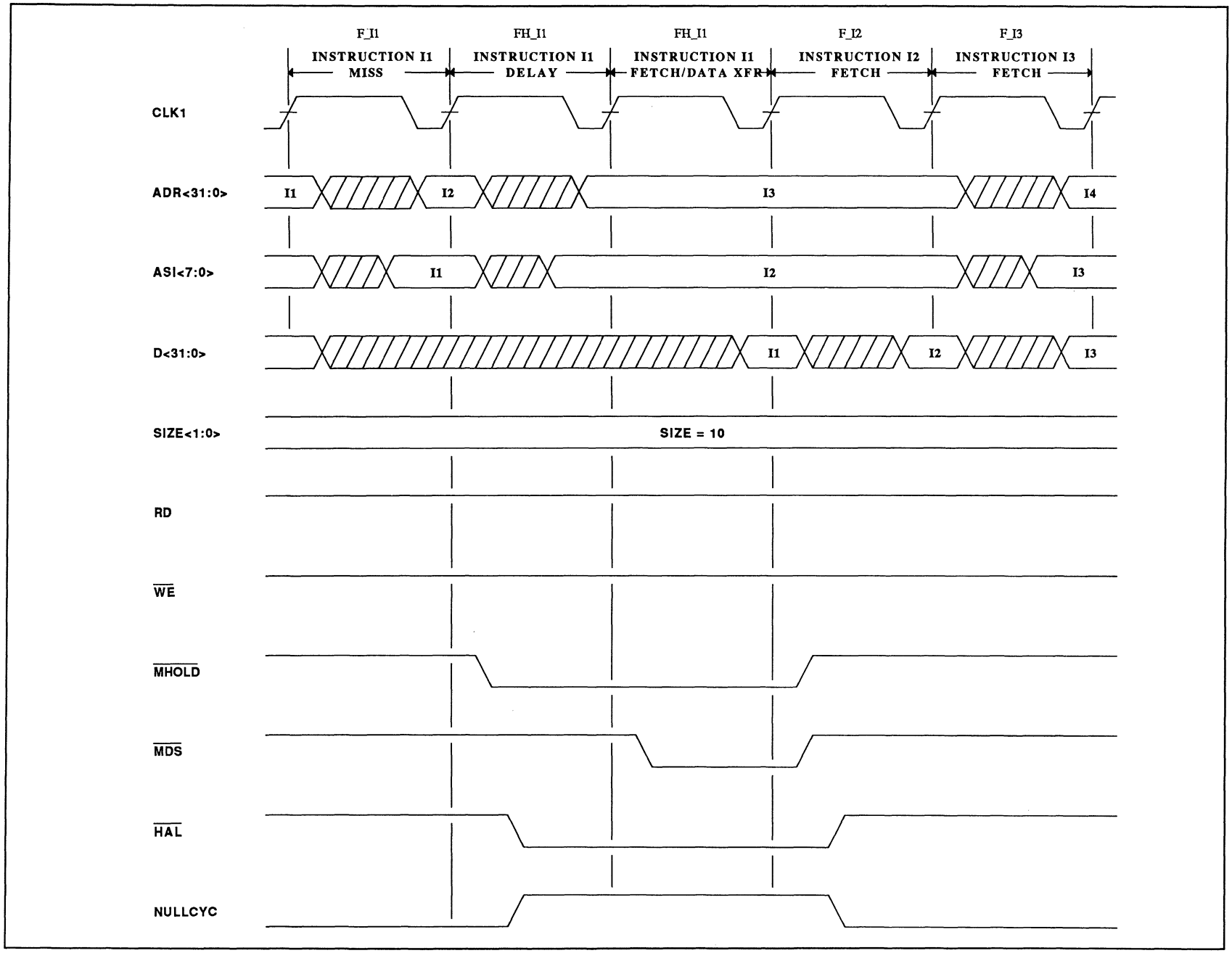


Figure 7.9 Load (Cache Miss)

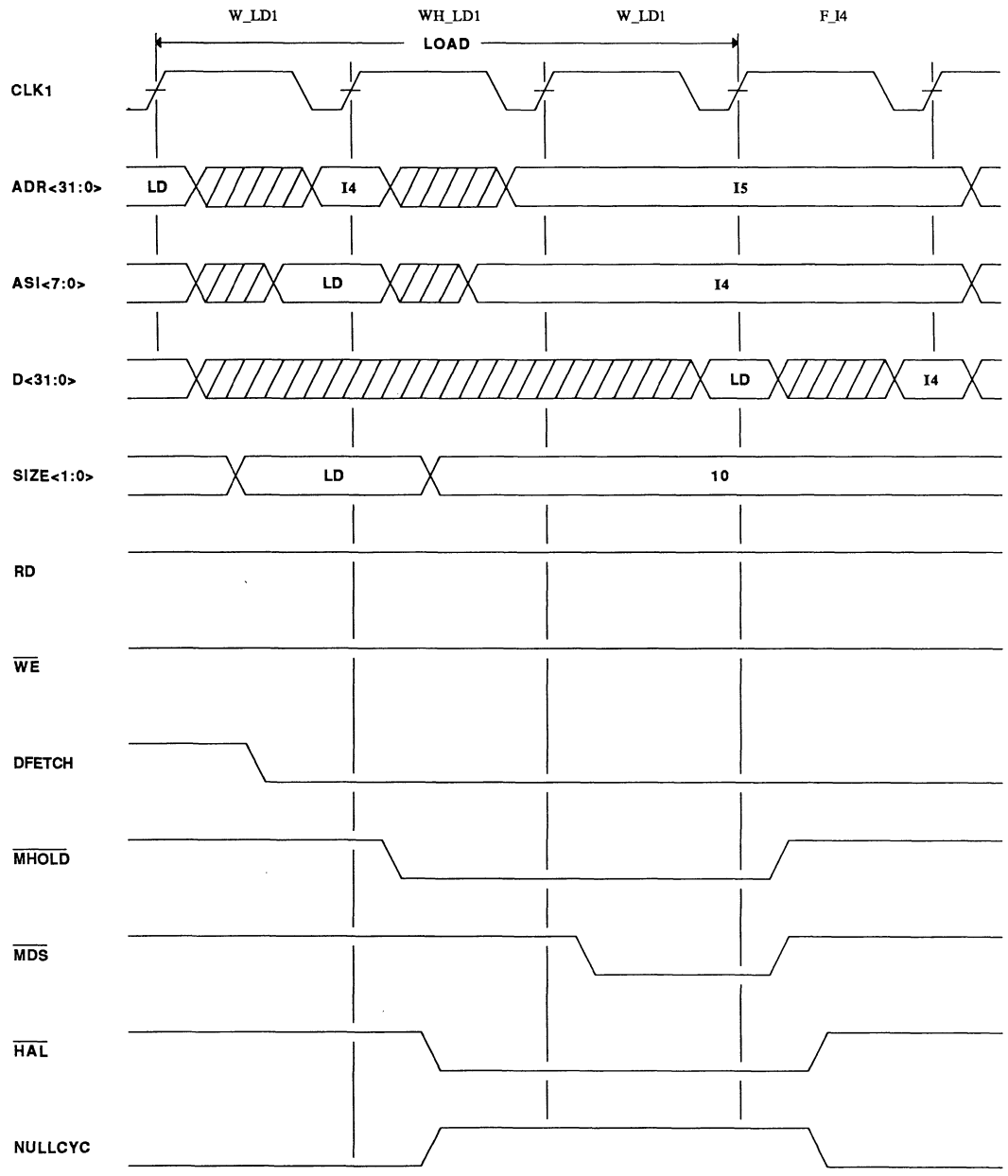


Figure 7.10 Store (Cache Miss)

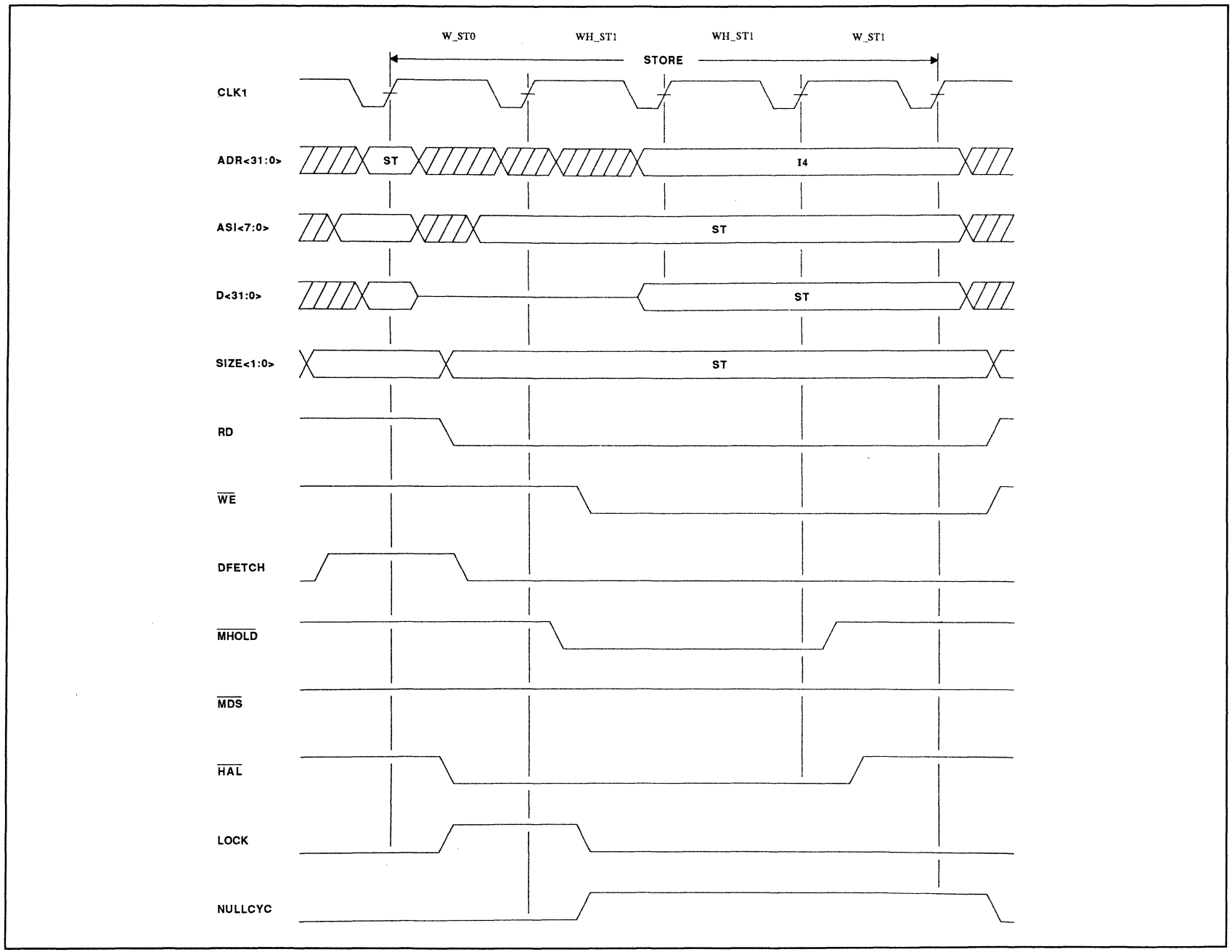


Figure 7.11 Store Double (Cache Miss)

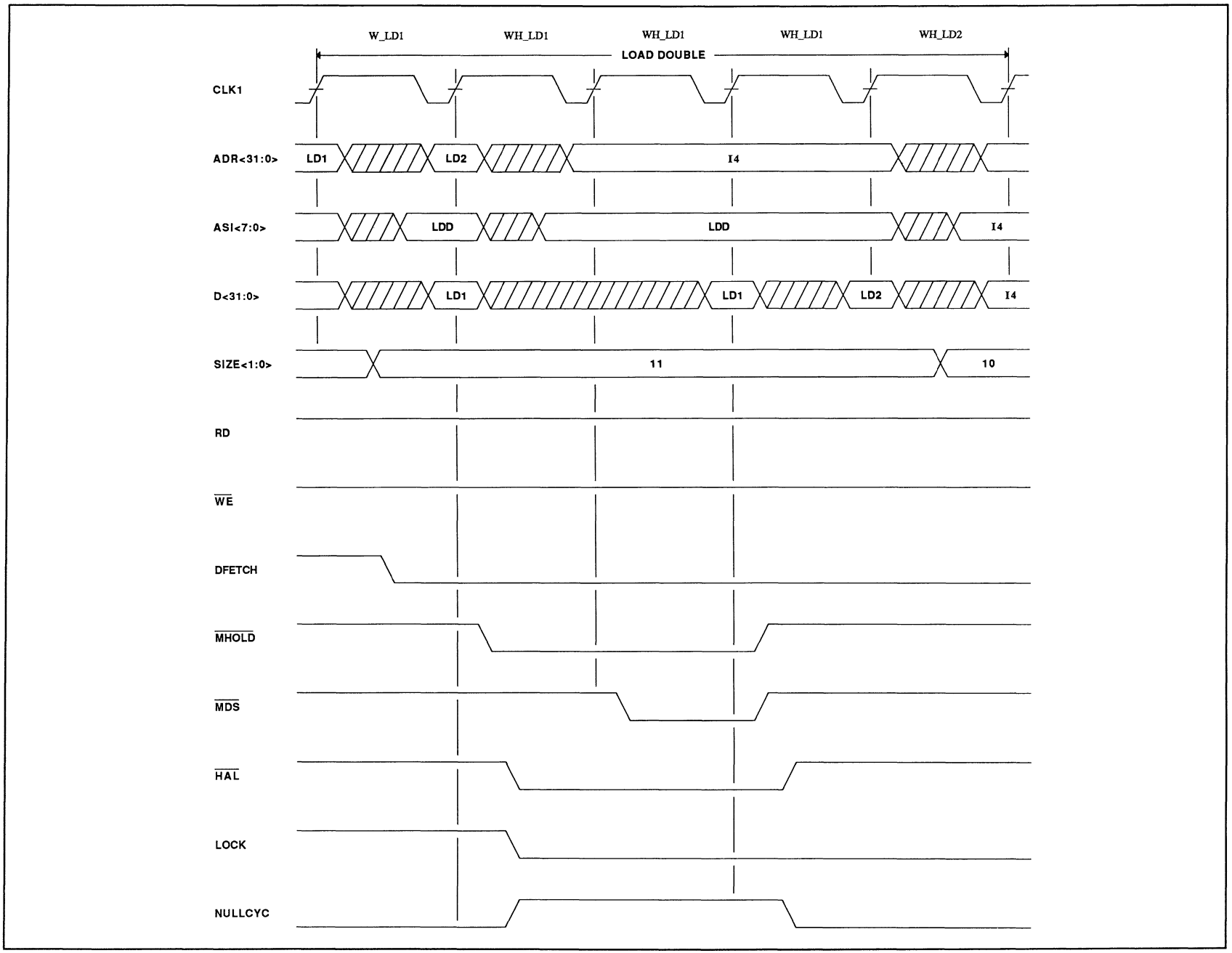


Figure 7.12 Atomic Load-Store (Cache Miss)

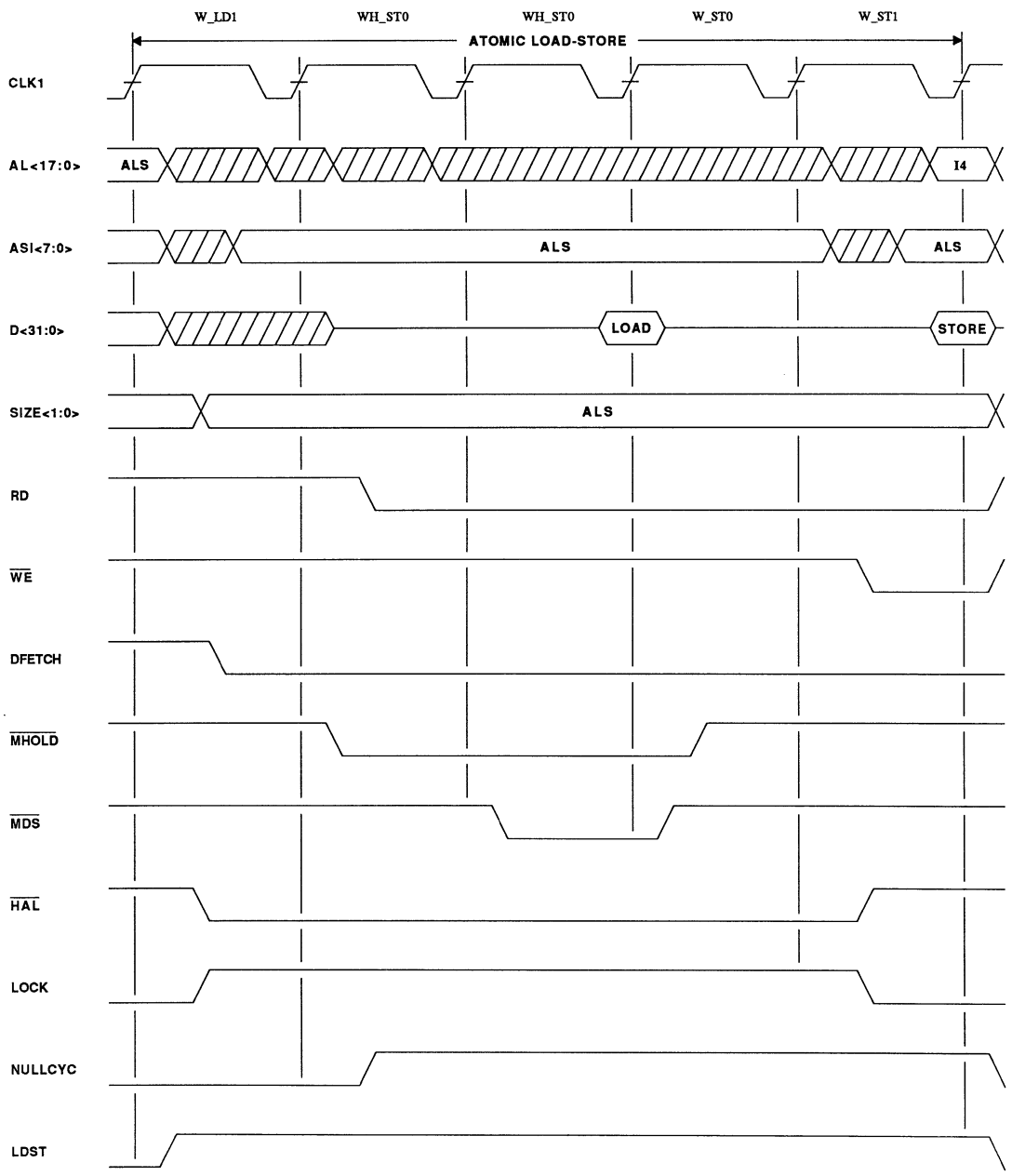


Figure 7.13 Instruction Fetch Memory Exception

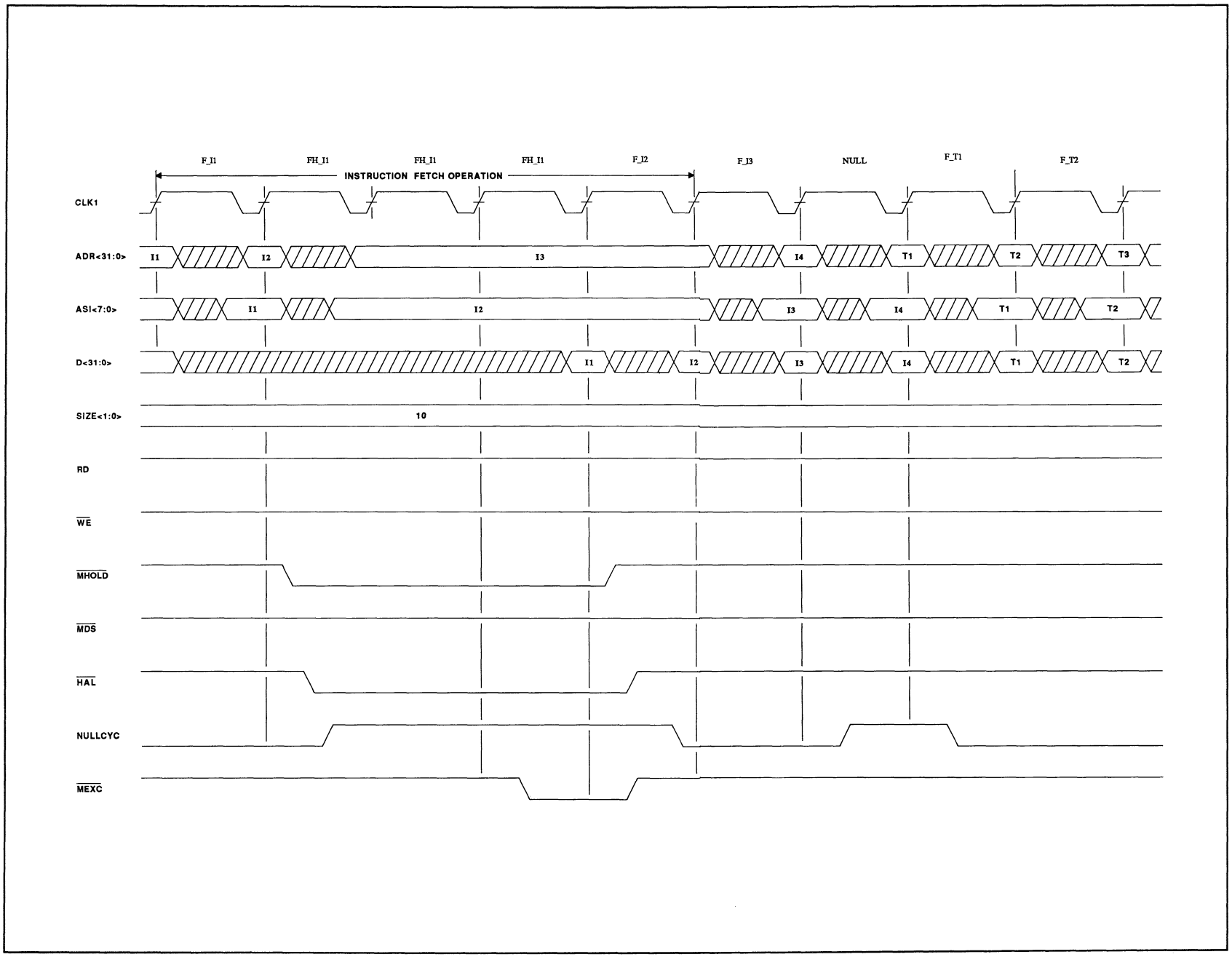


Figure 7.14 Load Memory Exception

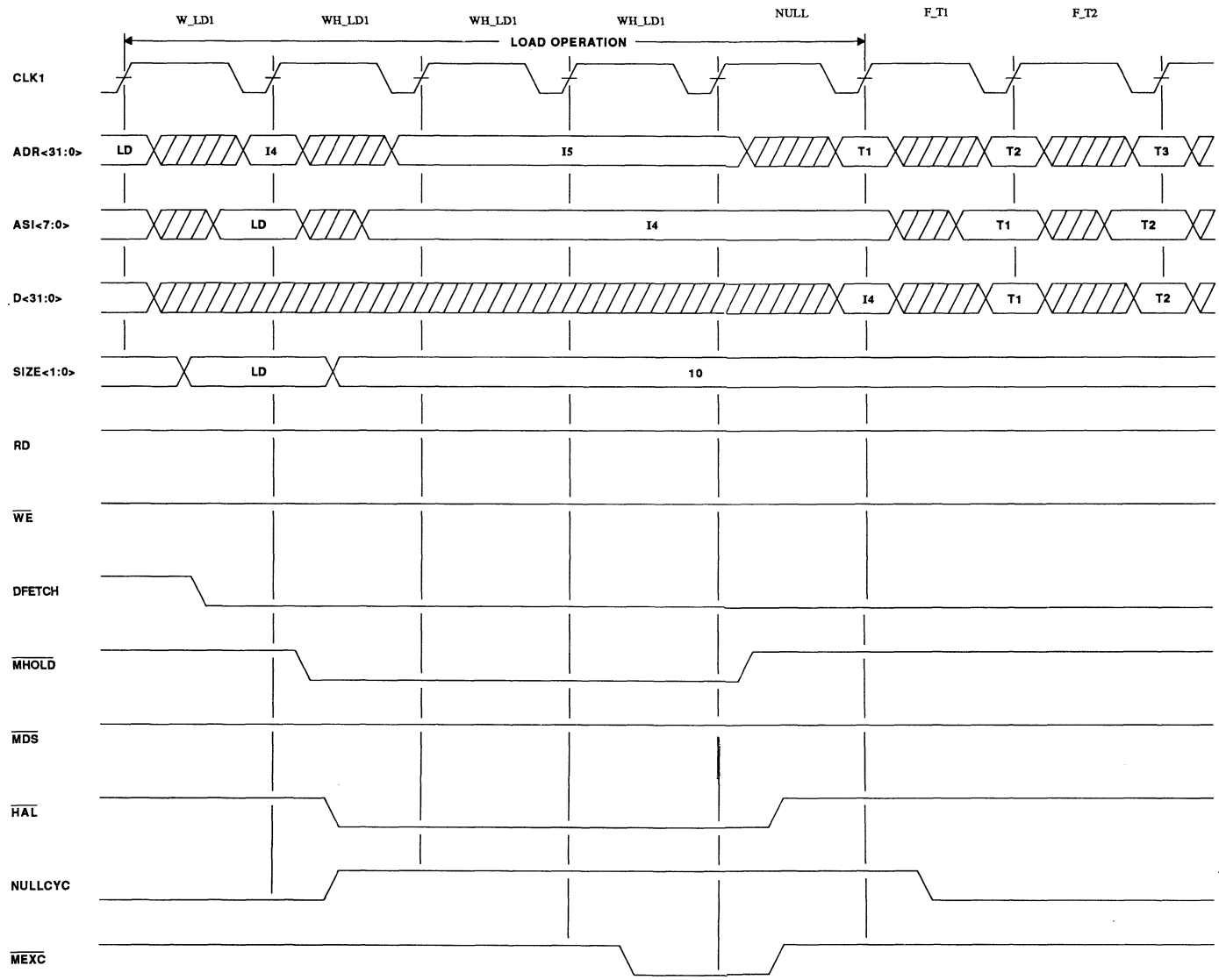


Figure 7.15 Store Memory Exception

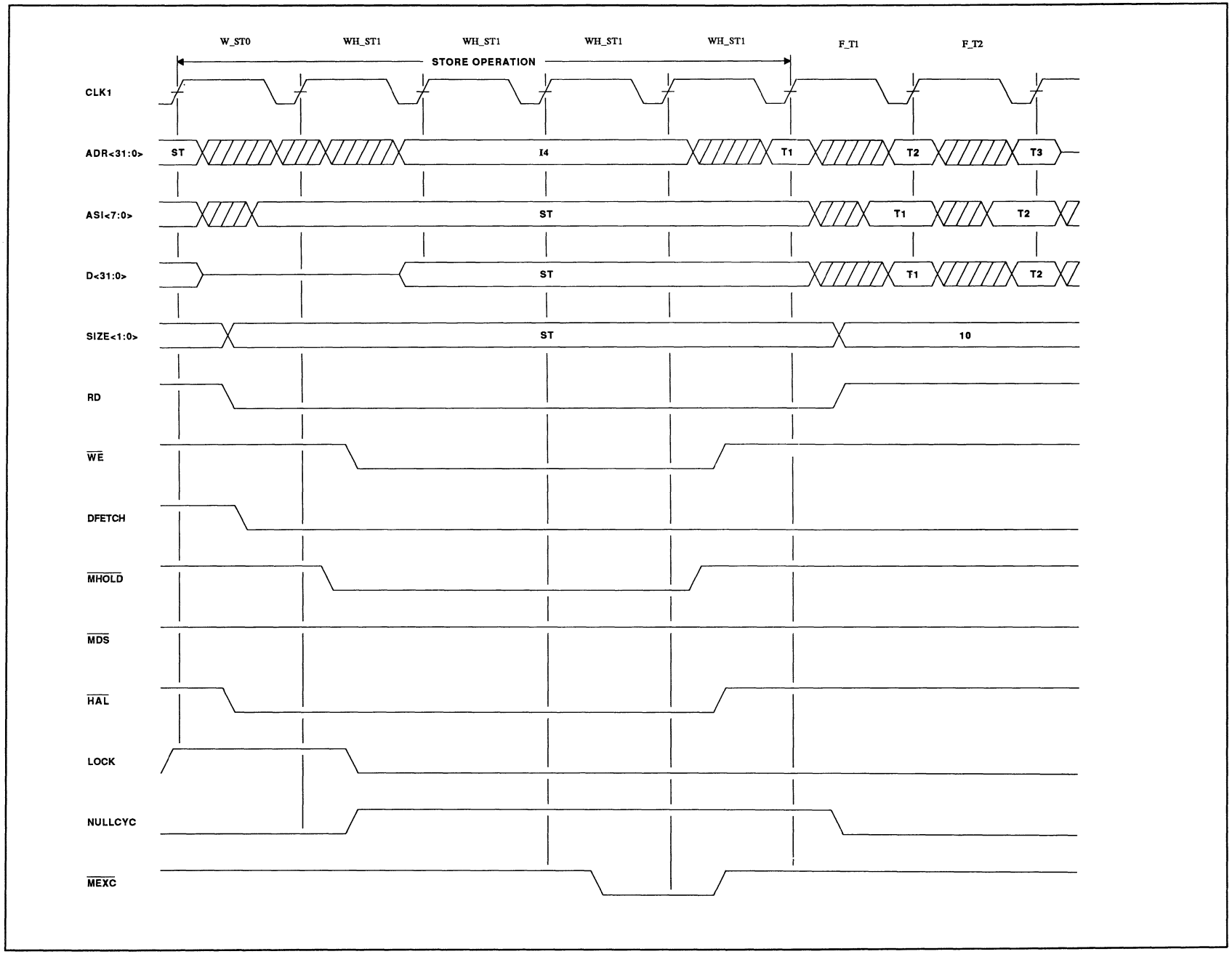


Figure 7.16 Atomic Load-Store Memory Exception

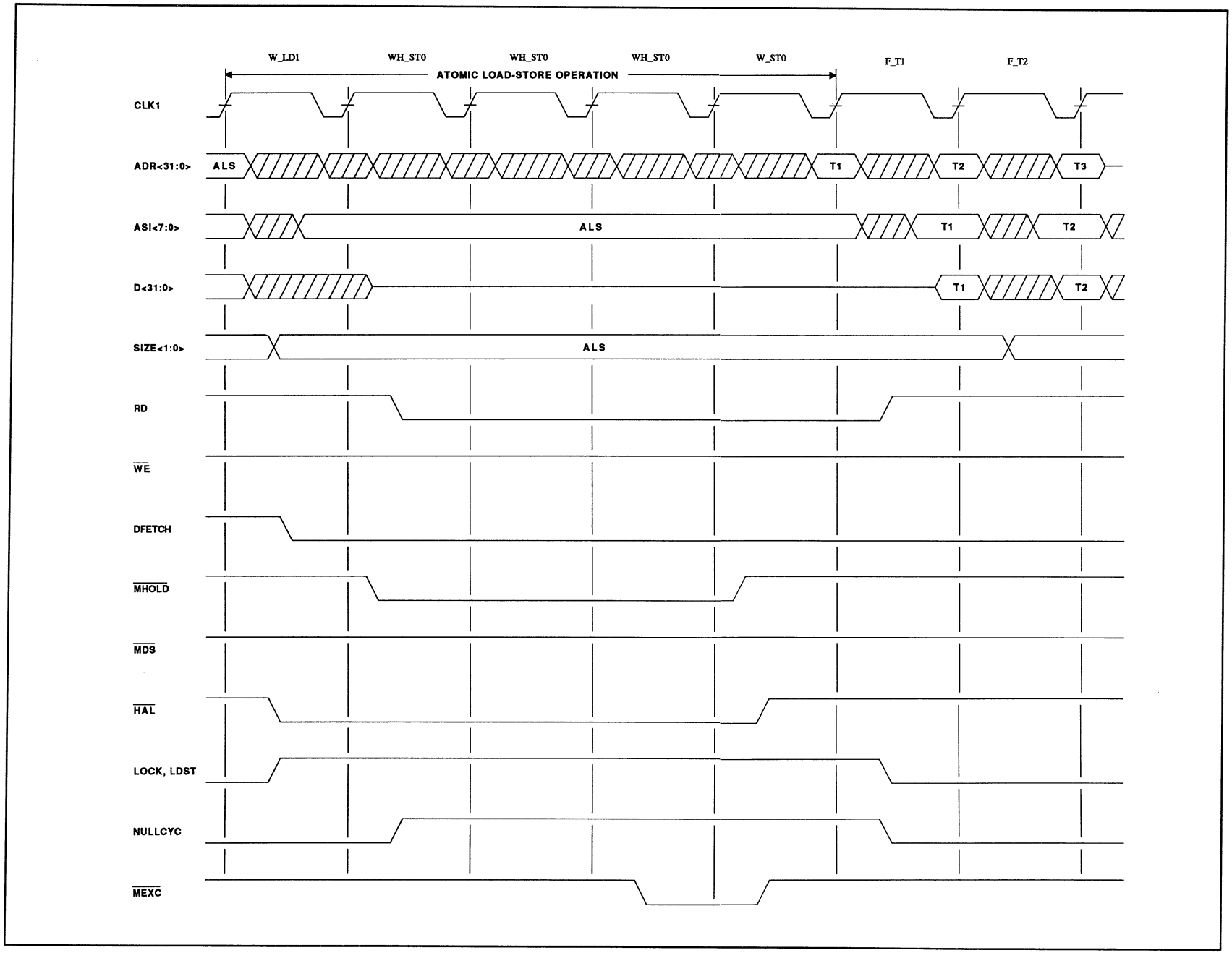


Figure 7.17 Bus Request

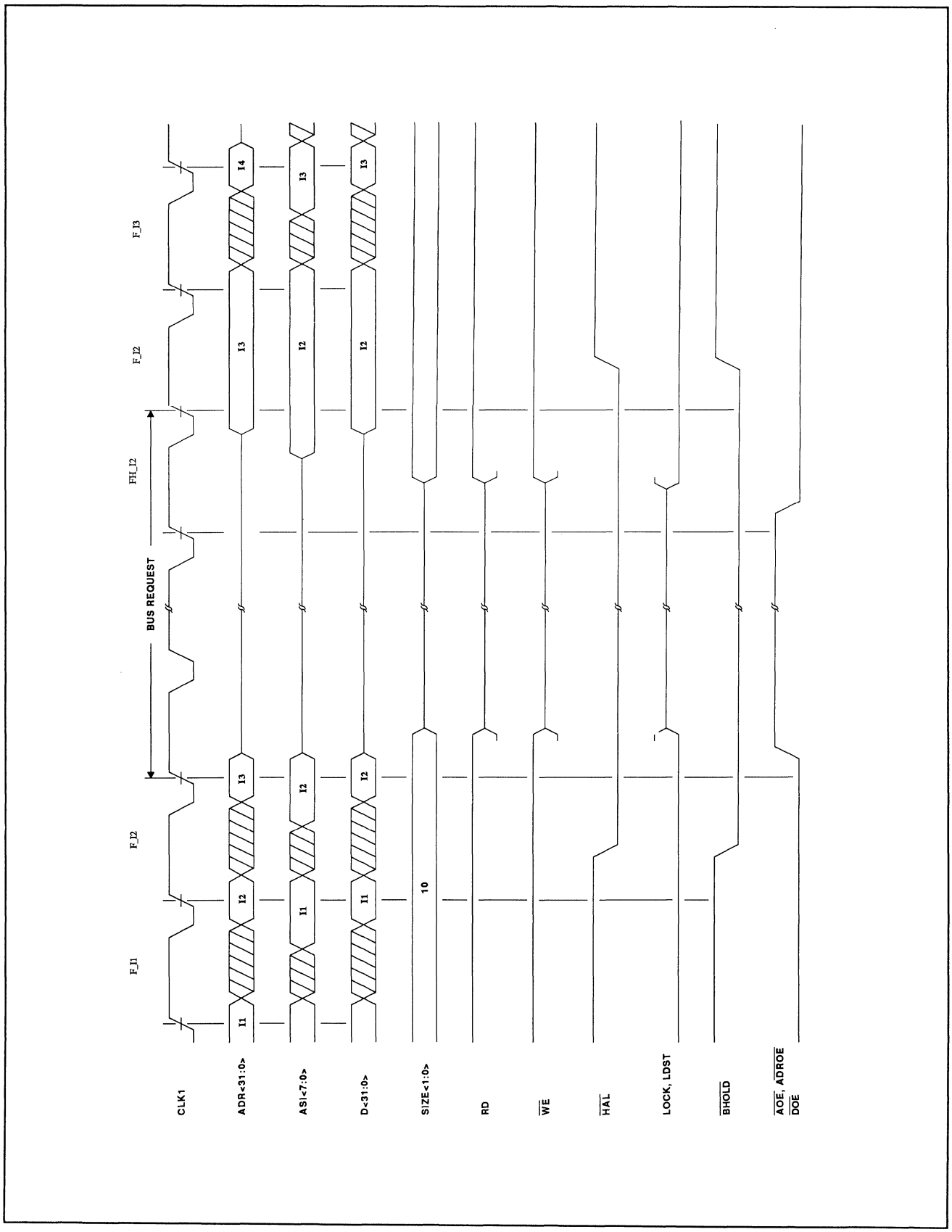


Figure 7.18 Error and Reset

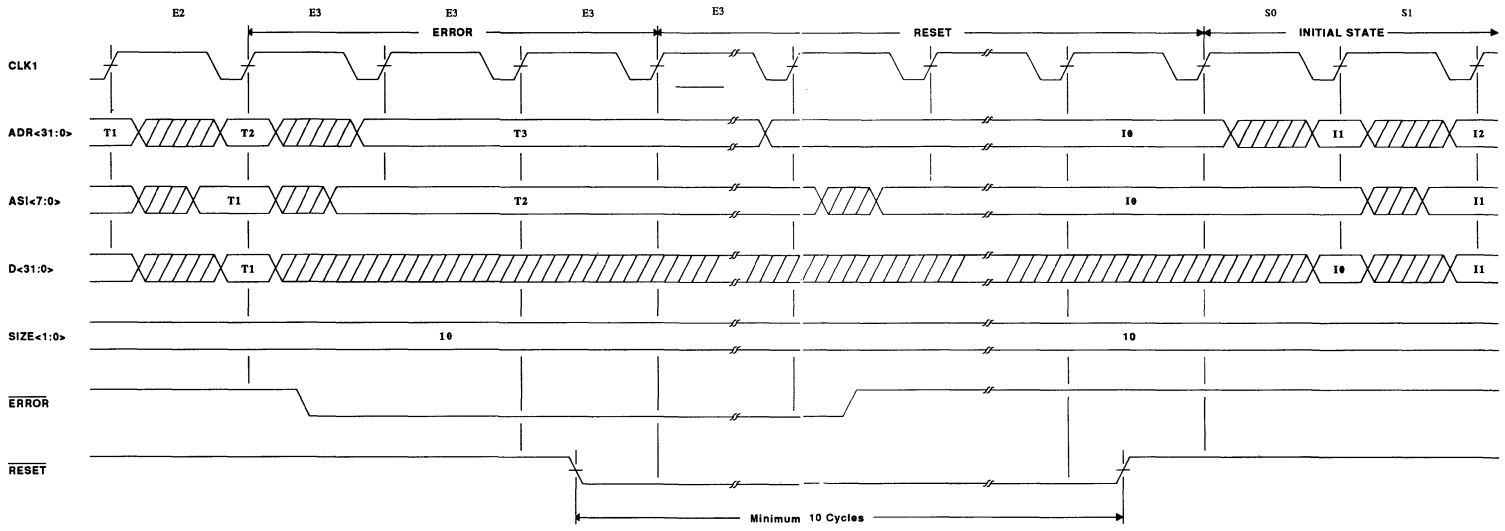
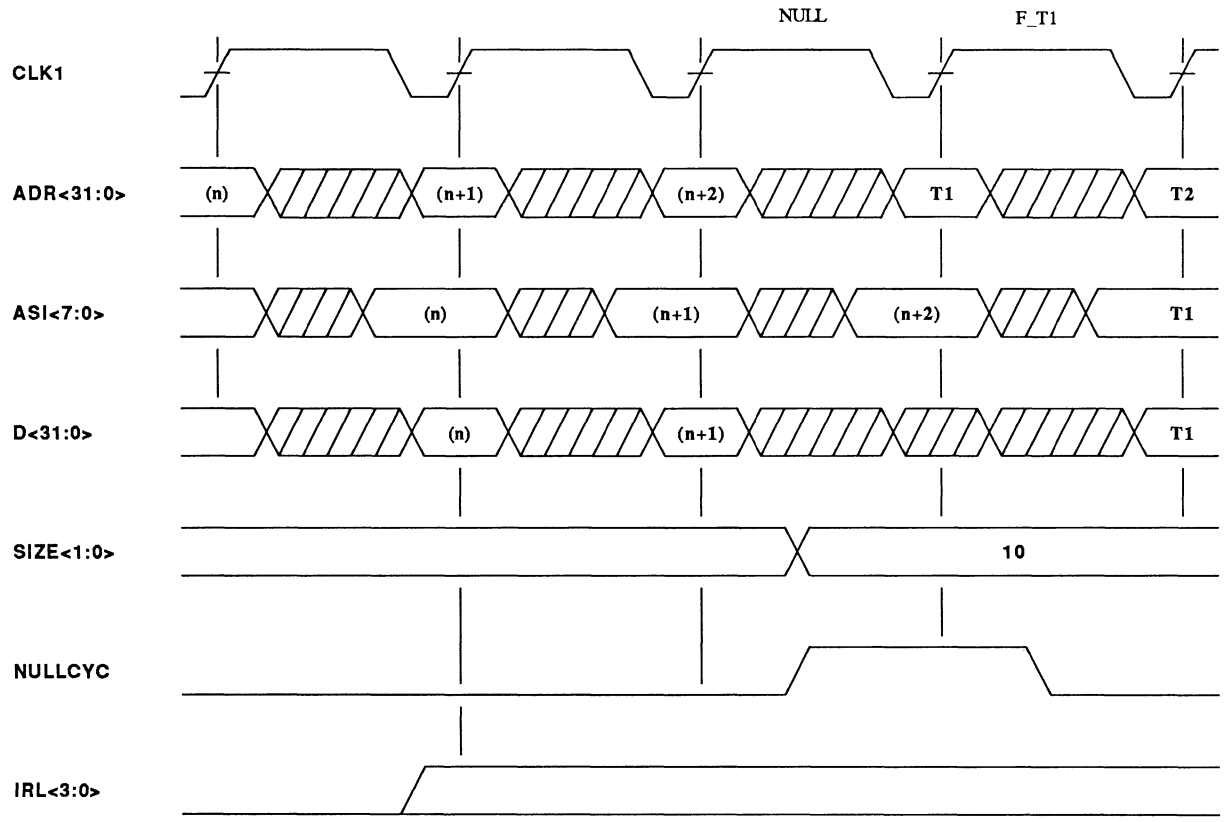


Figure 7.19 Asynchronous Trap (Interrupt)



Notes:

(1) IRL<3:0> is reset to 0 by an interrupt service routine.
(2) (n)-(n+1) are user instructions/data.
(3) T1-T2 are supervisor Trap Dispatch Table instructions.

7.2. Floating Point Interface

The Floating Point Interface directly interfaces the processor to the Fujitsu MB86911 Floating Point Controller, which in turn directly interfaces to the TI SN74ACT8847 Floating Point Processor (FPP). This allows use of the MB86911 and the FPP as a Floating Point Unit (FPU) for the MB86901 which executes floating point operations concurrently with processor integer operations, as explained in Section 4.

Data transfer between the processor and FPU is via the D<31:0> System Interface data bus. All other information transfer between the processor and FPU is via Floating Point Interface buses and signals as follows:

- F<31:0> transfers Floating Point Operate instructions and their addresses to the FPU.
- FADR indicates a valid address on the F<31:0> bus.
- FINS indicates a valid instruction on the F<31:0> bus.
- FEND indicates the last processor cycle of the current instruction transfer.
- $\overline{\text{FCC<1:0>}}$ transfers floating point operation condition codes to the processor.
- FCCV indicates a valid condition code on the $\overline{\text{FCC<1:0>}}$ bus.
- $\overline{\text{FEXC}}$ indicates a floating point exception.
- FXACK indicates processor recognition of a floating point exception.
- $\overline{\text{FHOLD}}$ halts the processor.
- FLUSH forces the unit to abort the current instruction.
- $\overline{\text{FP}}$ indicates the presence of an FPU.

7.2.1. FPOP Instruction Transfer Timing

Timing for Floating Point Operate (FPOP) instruction transfers are shown in Figures 7.20 and 7.21.

Figure 7.20 shows FPOP instruction transfer without delays. The FPOP instruction is transferred during the first cycle, followed by transfer of the FPOP instruction address the second cycle. The instruction address is latched by the FPU to support recovery from faults which may occur (see FPU data sheets).

The processor asserts FINS during the first cycle to indicate to the FPU that it is an instruction cycle, asserts FADR during the second cycle to indicate that it is an address cycle, and asserts FEND after the second cycle to indicate that the operation is completed.

Figure 7.21 shows FPOP instruction transfer timing with delay cycles. The FPU has asserted $\overline{\text{FHOLD}}$ during the operation as shown in the figure, holding the processor in a wait state until $\overline{\text{FHOLD}}$ is released.

7.2.2. FP Load And Store Timing

Timing for the floating point Load, Store, Load Double Word, and Store Double Word instructions are shown in Figures 7.22–7.25. These instructions, containing memory source or destination operand addresses, are transferred to the FPU for execution.

The processor asserts DFETCH at the beginning of each operation to indicate that data will be transferred, transfers the instruction during the first cycle of the operation, then transfers the instruction address during the second cycle.

The FPU transfers data to/from memory via the D<31:0> bus as shown in the figures.

7.2.3. Cache Miss and Exception Timing

Cache miss timing, memory exception timing, floating point exception timing, and various combinations of each are shown in Figures 7.26–7.34. This timing is similar to that of non-floating point operations (see Sections 7.1.3 and 7.1.4), with floating point exceptions resulting in the same timing as memory exceptions.

Figure 7.20 FPOP Instruction Transfer

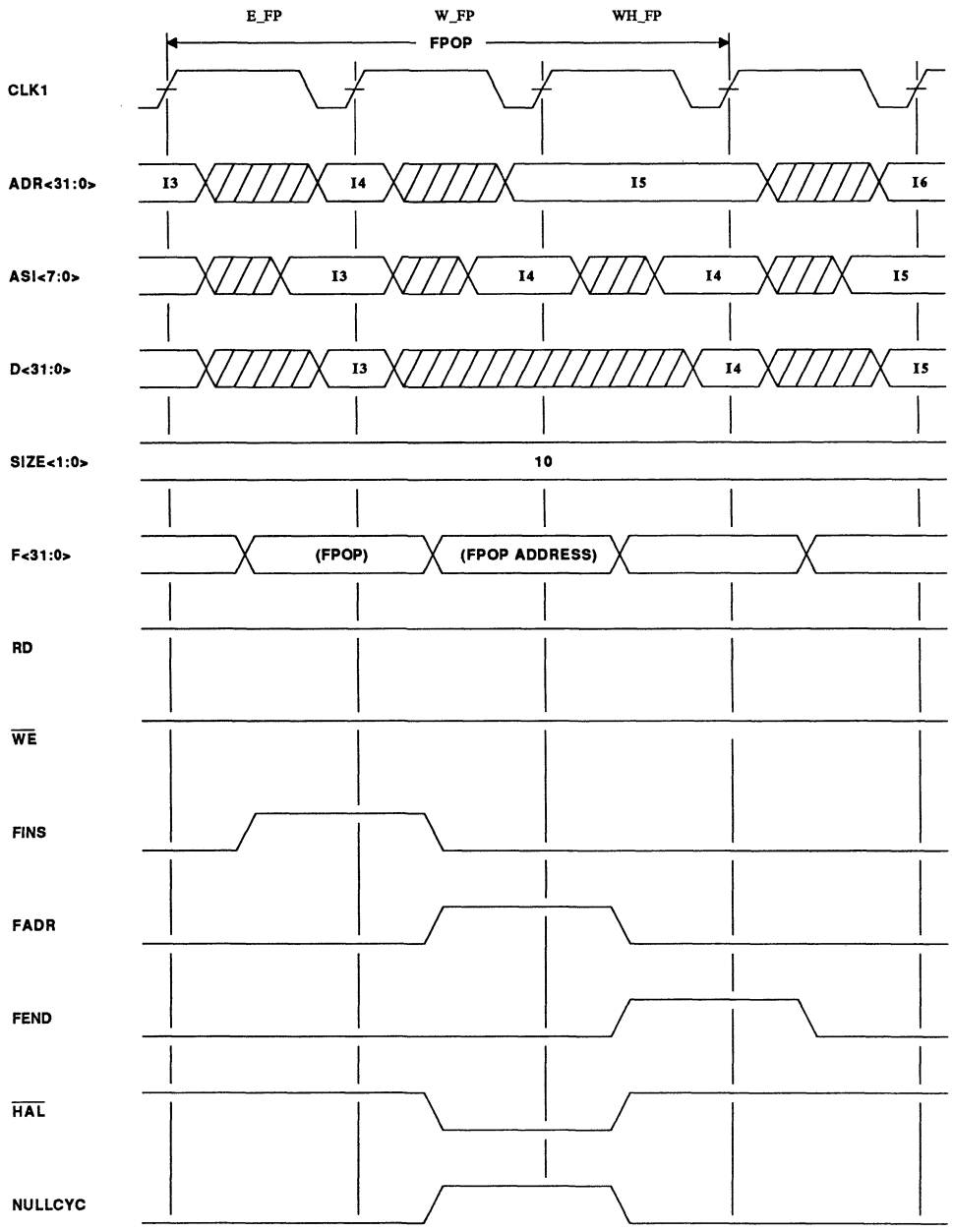


Figure 7.21 FPOP Instruction Hold

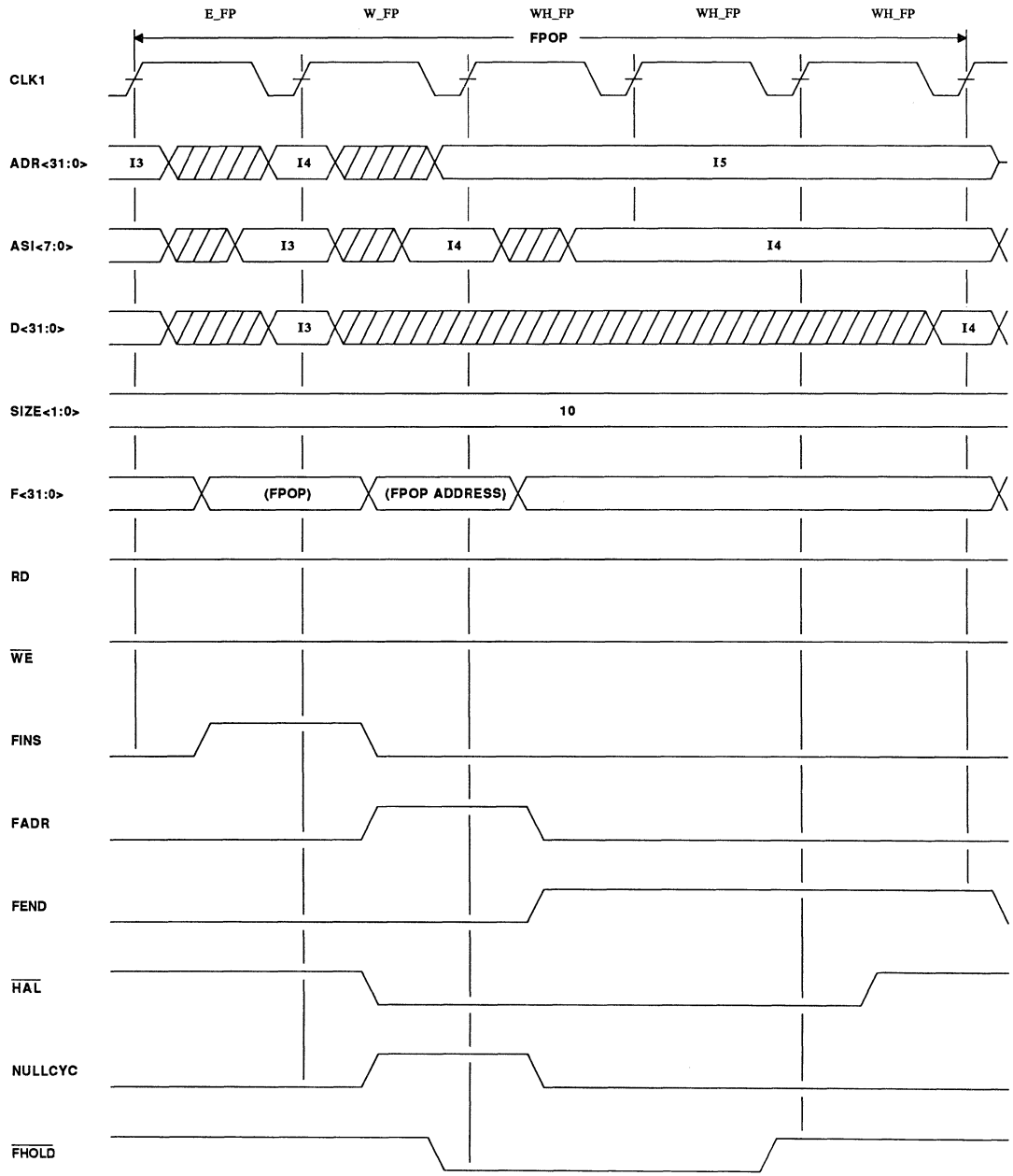


Figure 7.22 FP Load (Cache Hit)

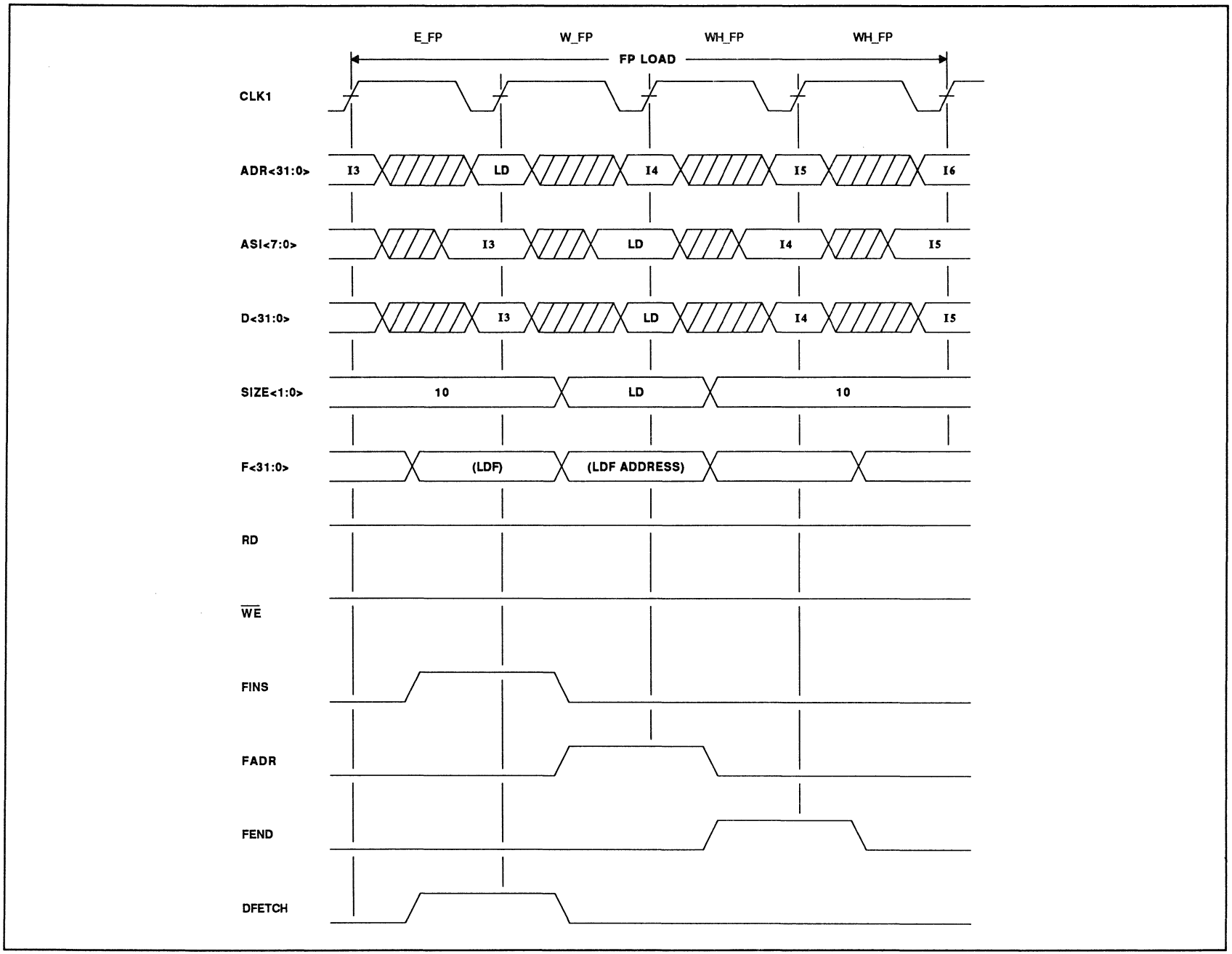


Figure 7.23 FP Store (Cache Hit)

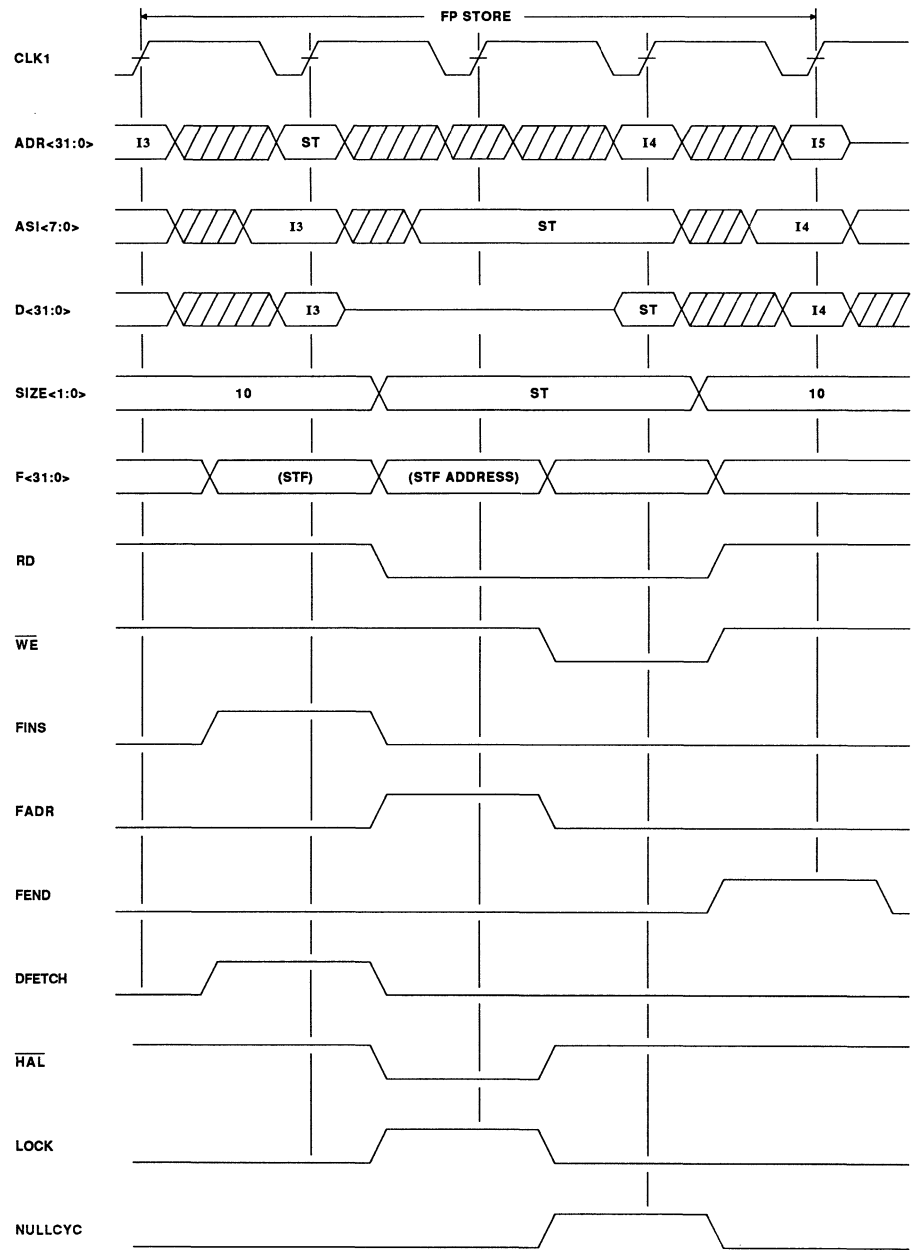


Figure 7.24 FP Load Double (Cache Hit)

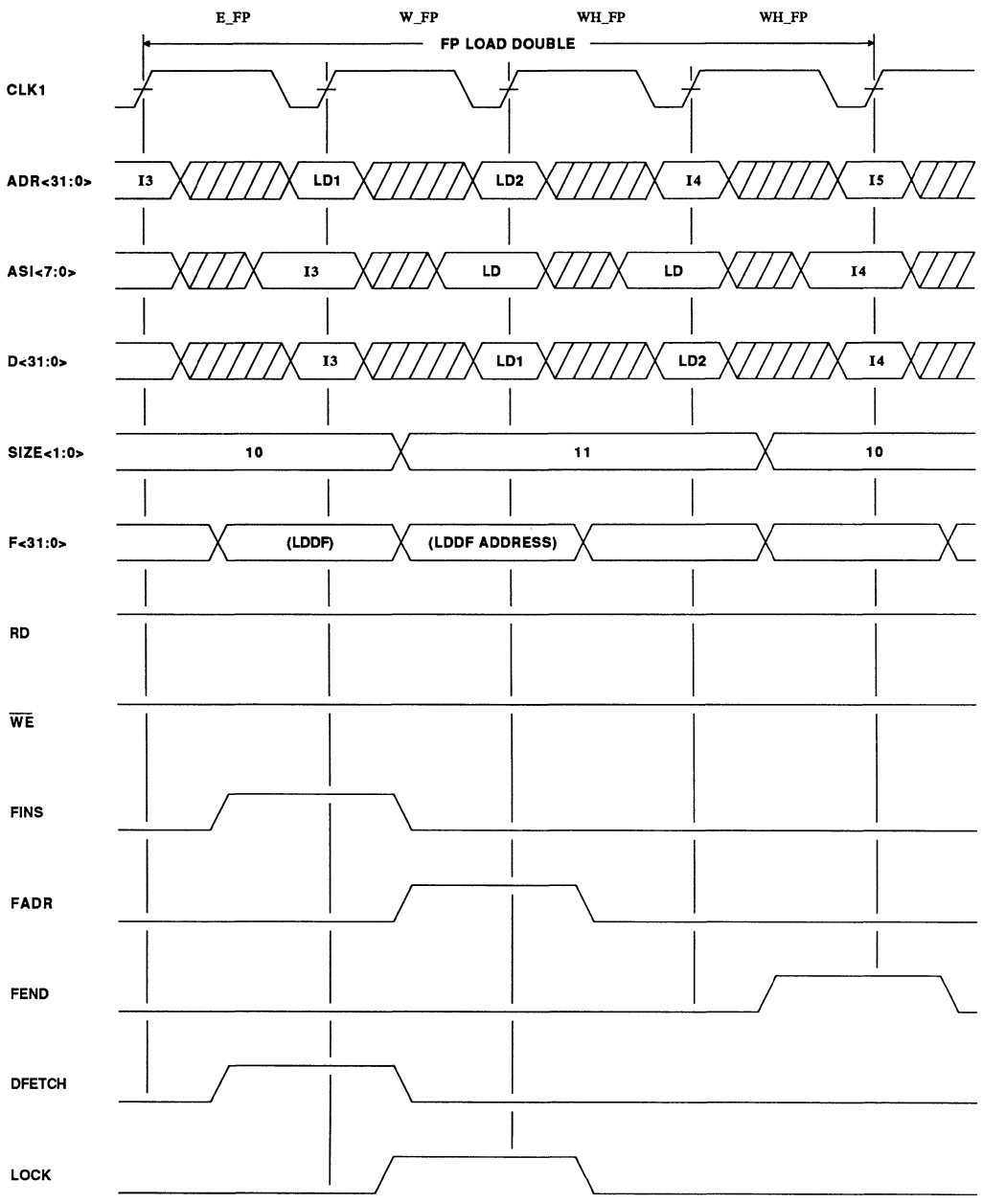


Figure 7.25 FP Store Double (Cache Hit)

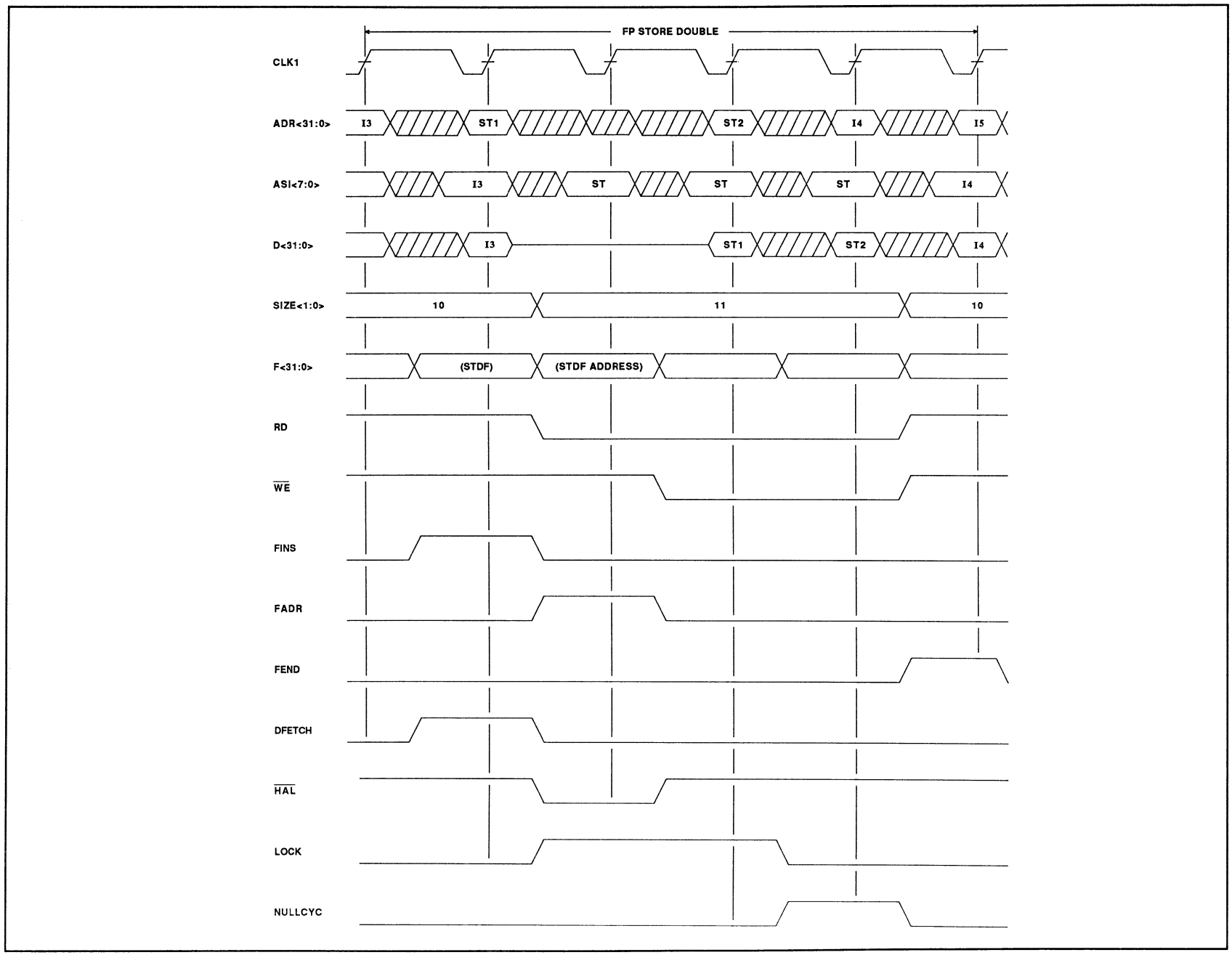


Figure 7.26 FP Store (Hold and Cache Miss)

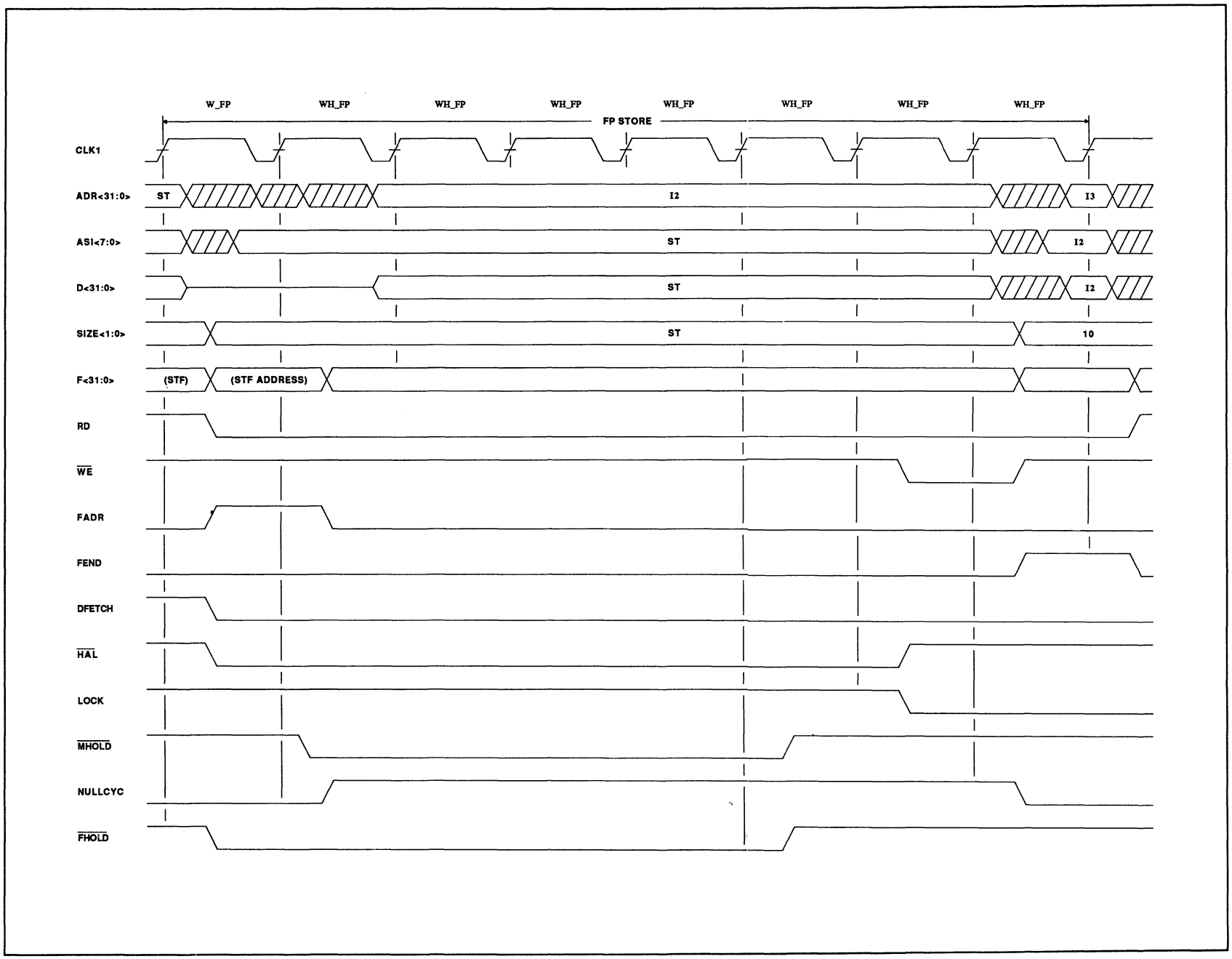


Figure 7.27 FP Load Double (Cache Miss)

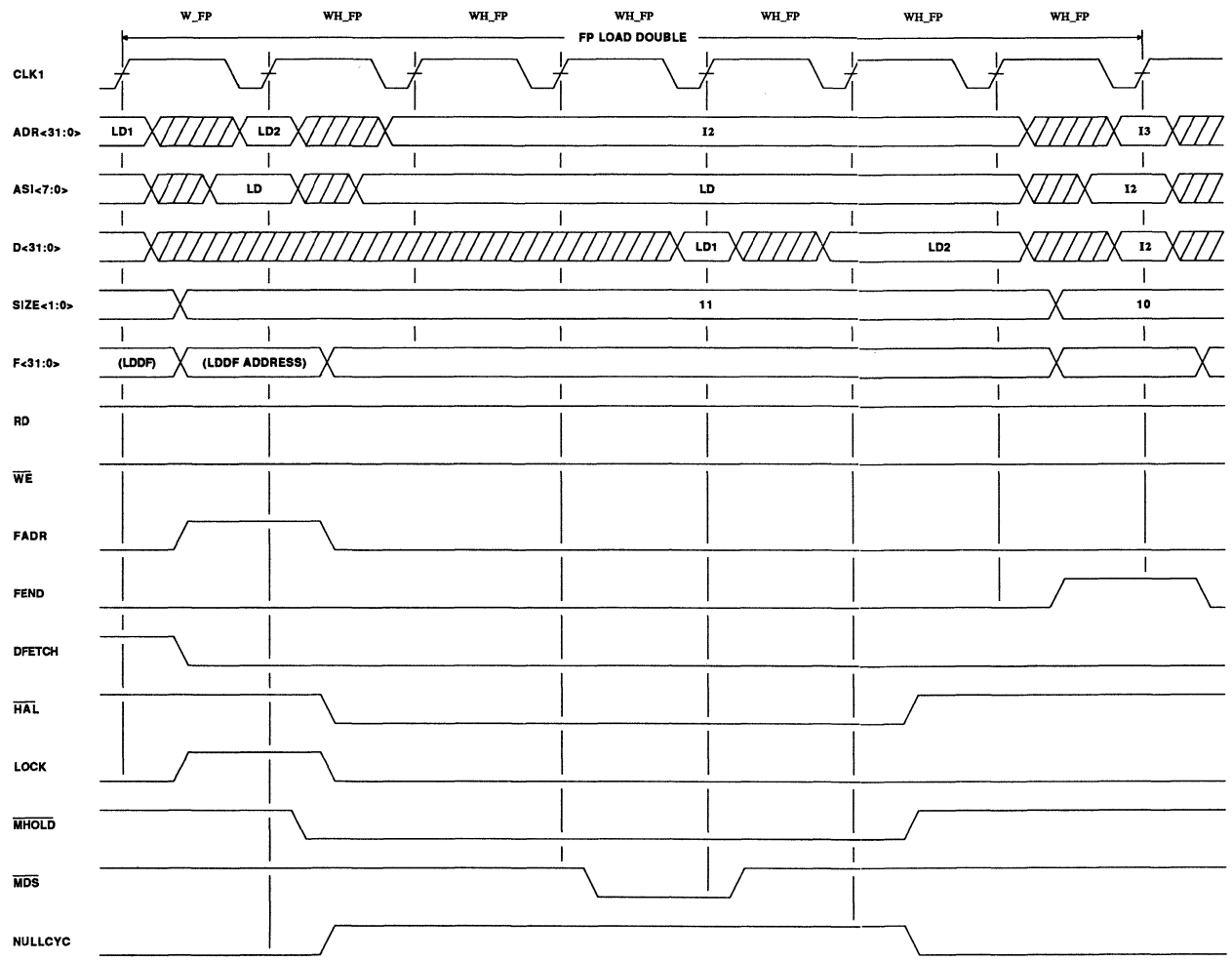


Figure 7.28 FP Store Double (Cache Miss)

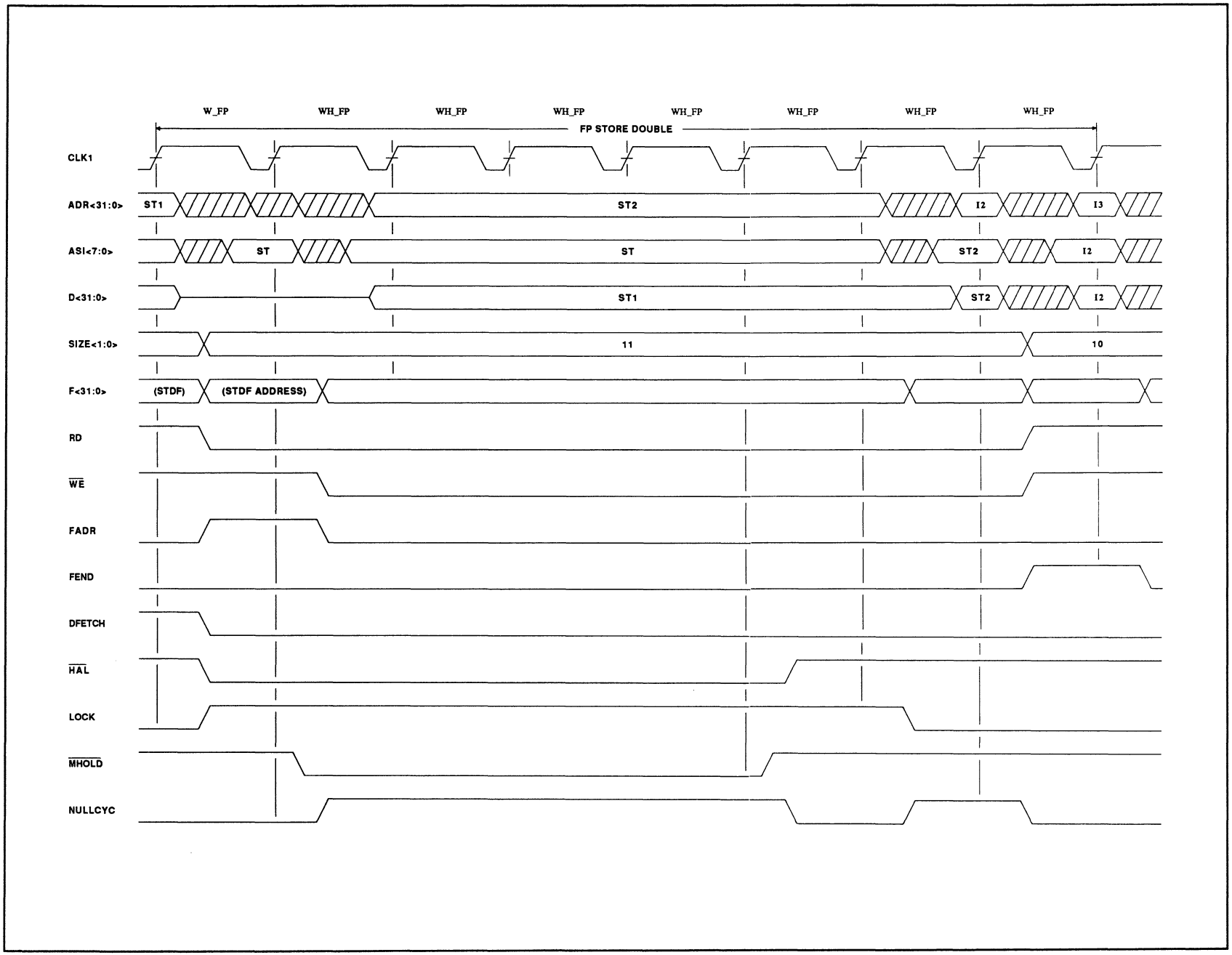


Figure 7.29 FP Store Double (Hold and Cache Miss)

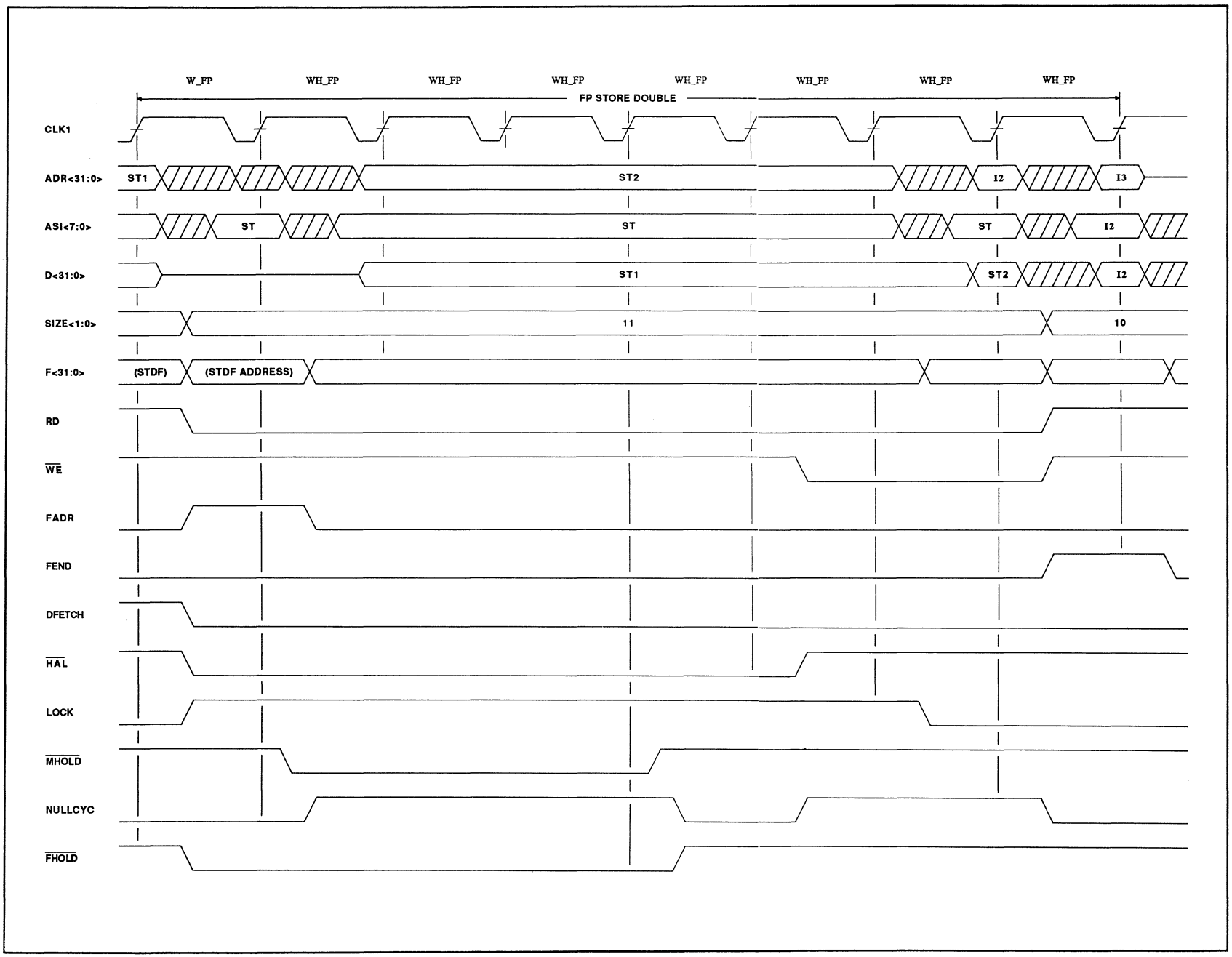


Figure 7.30 FP Load Exceptions

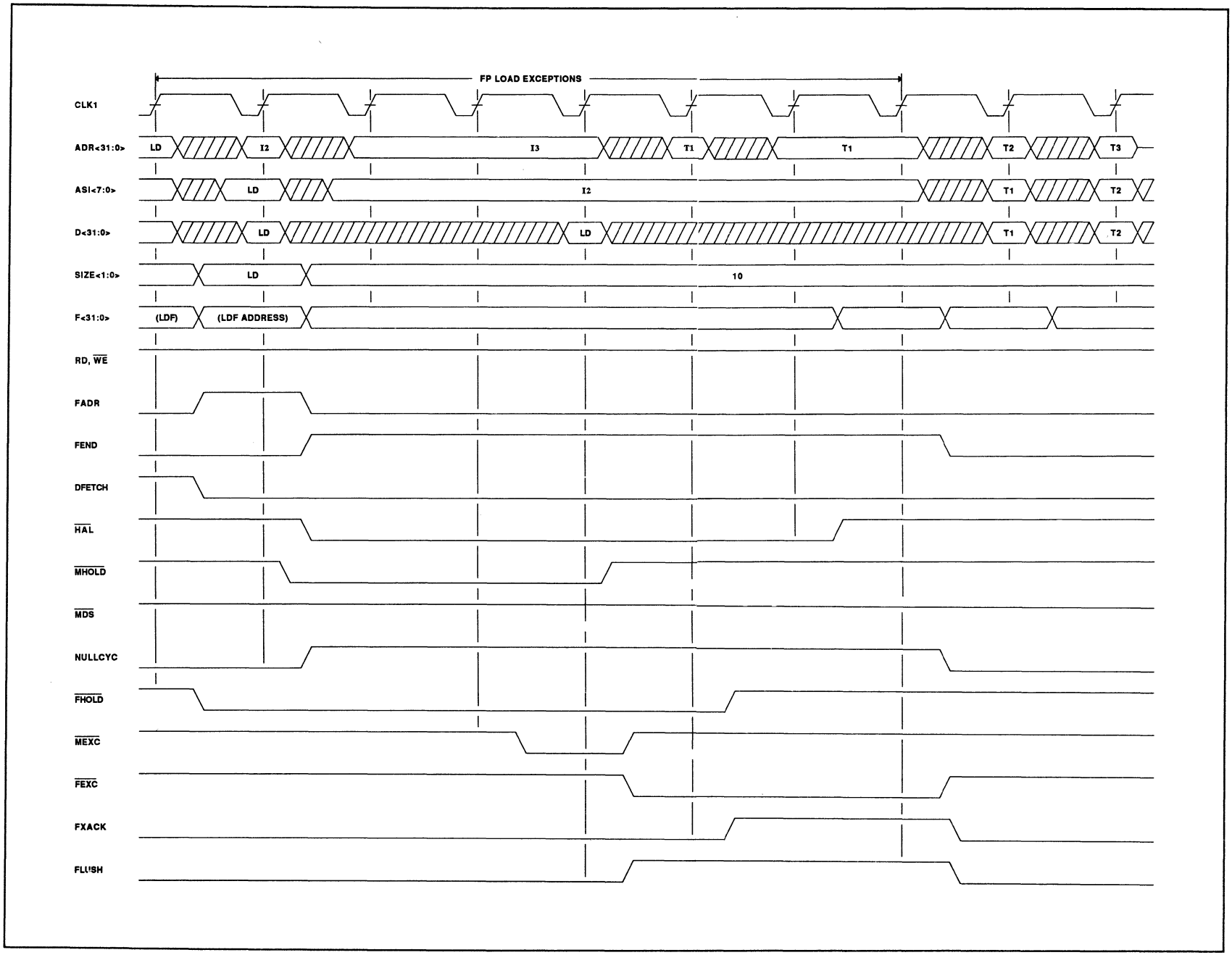


Figure 7.31 FP Store Exceptions

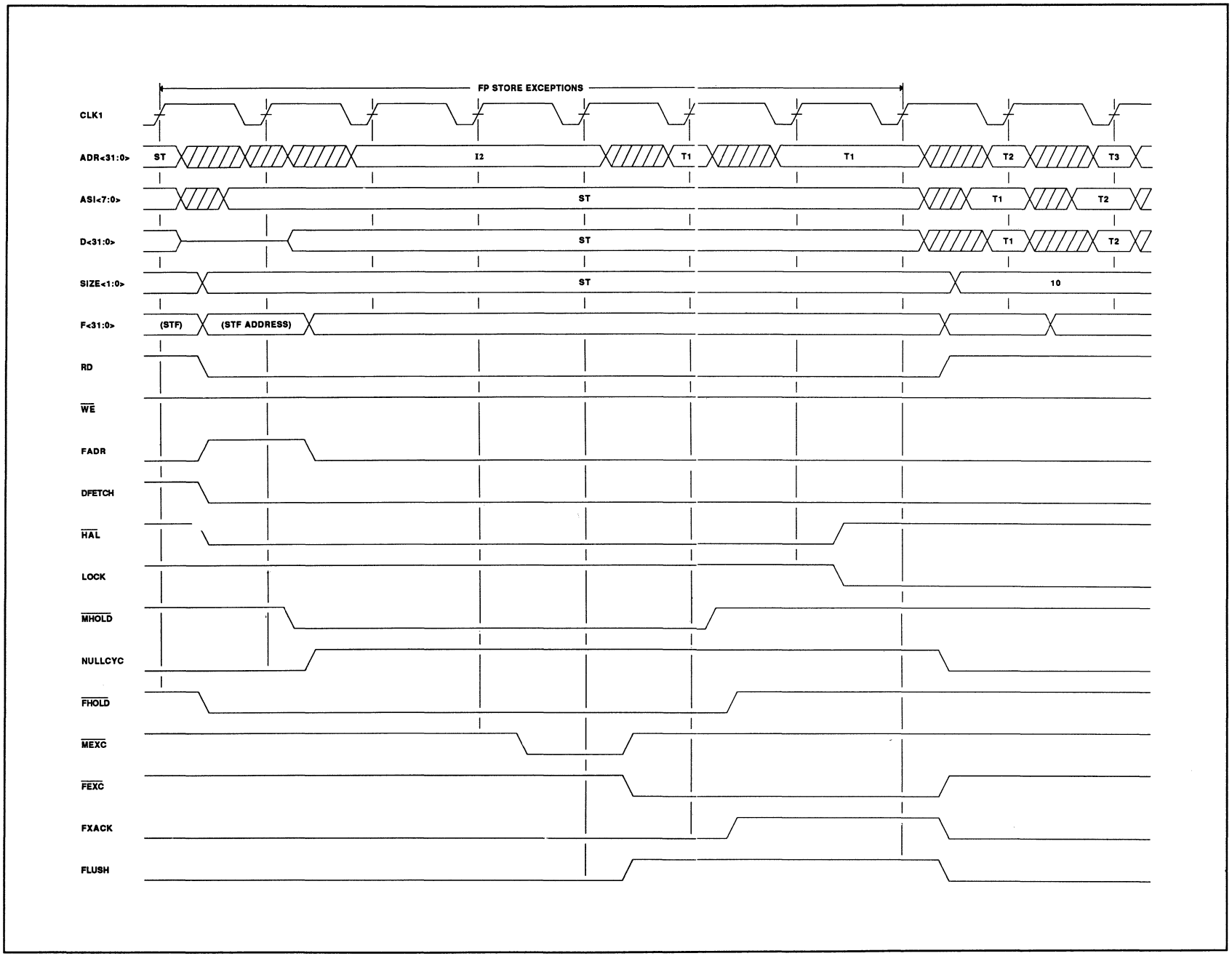


Figure 7.32 FP Load Double Memory Exception

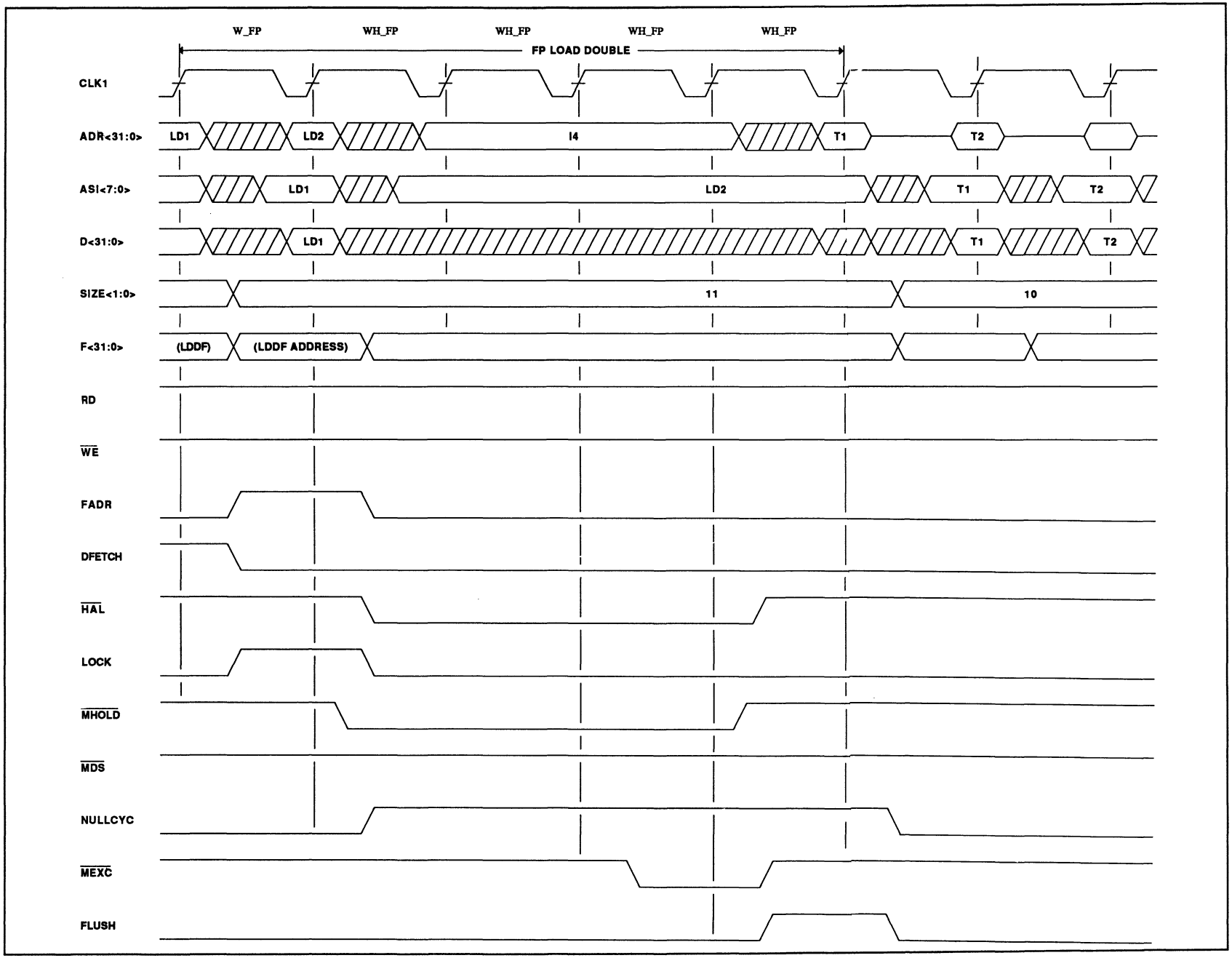


Figure 7.33 FP Store Double Memory Exception

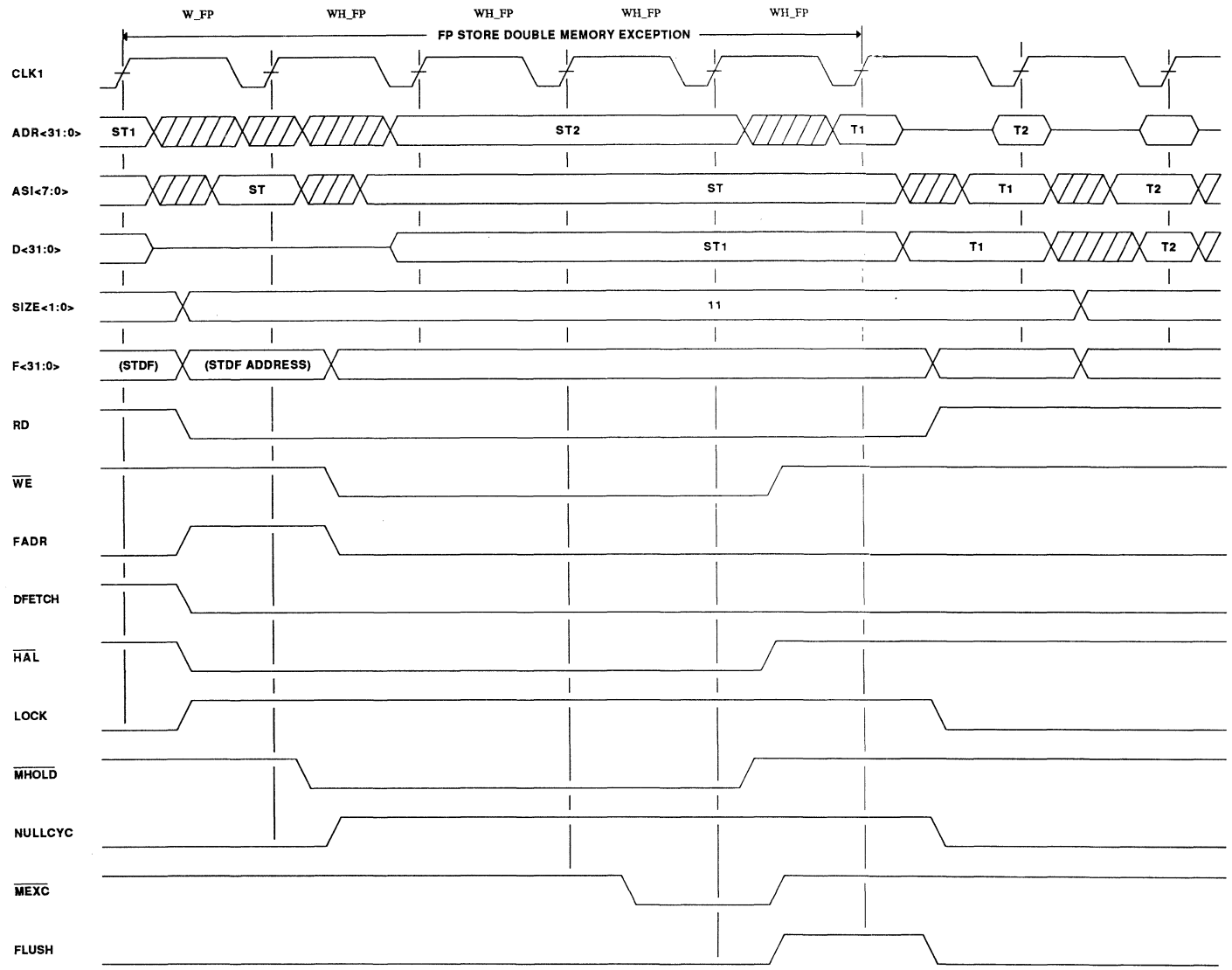
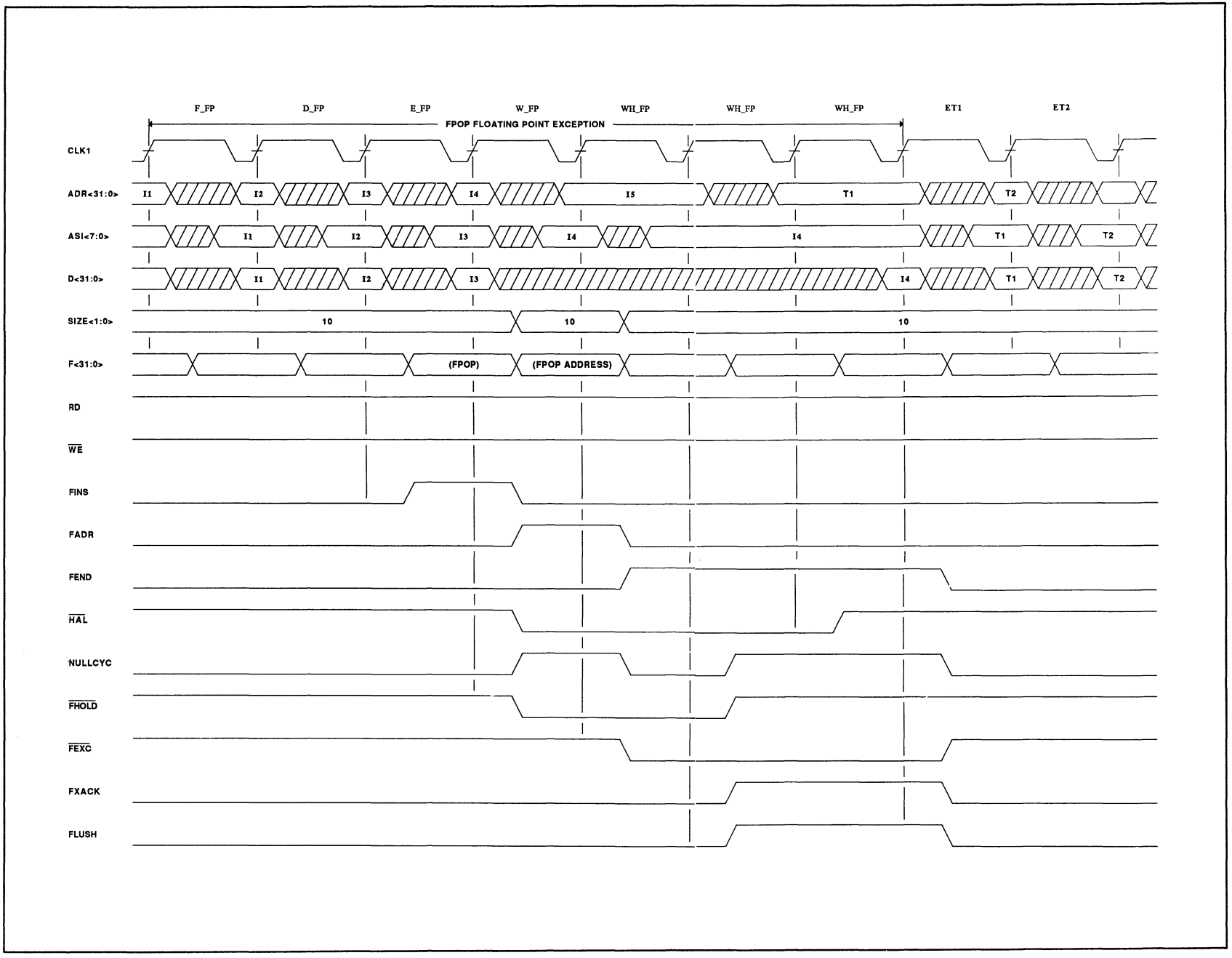


Figure 7.34 FPOP Floating Point Exception



7.3. System Configuration

Figure 7.35 shows a basic MB86901 system configuration utilizing an FPU, an interrupt controller, and a bus arbiter. The diagram is for illustration purposes only, and may require modifications such as a separate cache bus, for instance, to minimize system bus loading.

The FPU consists of the Fujitsu MB68911 RISC Floating Point Controller and the TI SN74ACT8847 floating point chip, interfaced to the processor via the processor Floating Point Interface and the System Interface D<31:0> bus.

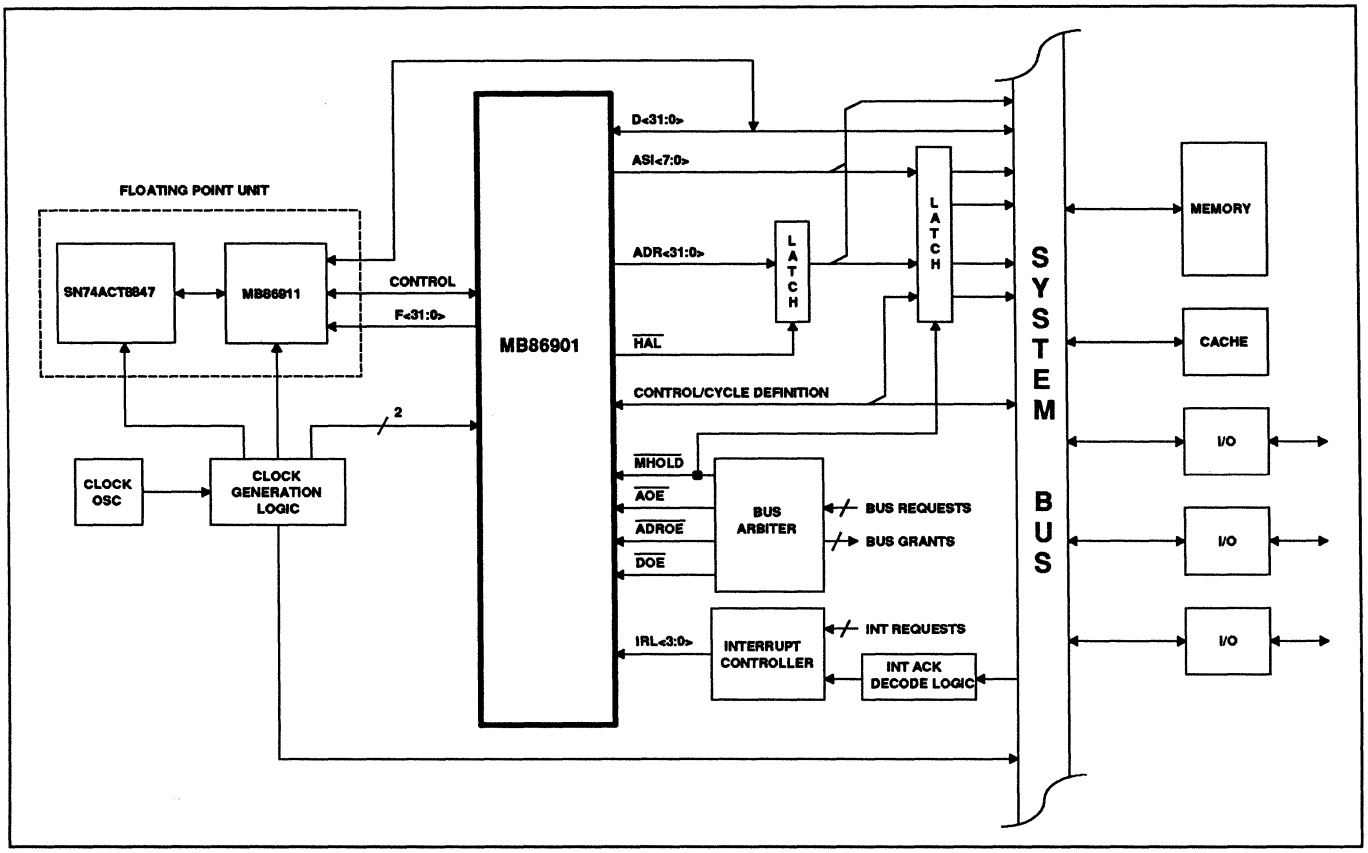
Data is transferred between the processor and the FPU via the processor System Interface D<31:0> bus. All other signals are transferred via the Floating Point Interface as shown.

The Interrupt Controller receives interrupt requests from system peripheral devices, prioritizes or otherwise arbitrates multiple interrupt requests, then asserts the selected interrupt request level number on the IRL<3:0> bus to alert the processor of the pending interrupt. The controller holds the level number asserted until the processor acknowledges recognition of the request via software, typically by accessing a location that the Interrupt Acknowledge Decode Logic recognizes as an interrupt acknowledge.

The Bus Arbiter receives bus requests from bus masters for control of the system bus, halts the processor and three-states its bus drivers in response to a bus request, then grants the system bus to the requesting bus master. The requesting bus master holds its bus request asserted until finished with the bus.

The Bus Arbiter halts the processor by asserting MHOLD. It three-states all of the processor System Interface output drivers except the DFETCH and HAL drivers by releasing the AOE, ADROE, and DOE driver enable signals. The arbiter asserts the driver enable signals once again when the current bus master releases its bus request, allowing the processor to continue execution.

Figure 7.35 Basic System Configuration



Note that ASI<7:0>, ADR<31:0>, and several System Interface control and cycle-definition signals are externally latched under control of the HAL processor signal. These latches, part of the External Address Pipeline, are explained in Section 8.2.

8. System Design Considerations

System design with the MB86901 is similar to design with other processors. Two areas of system design, however, require special consideration: the clock generation circuit, and the External Address Pipeline.

Examples of each are described in the following sections, although other logic which adheres to the processor timing specifications may be used.

8.1. Clock Generator

To generate CLK1 and CLK2 accurately, a 4x base clock frequency, operating at 100 MHz (G25), is required. With today's high frequency crystal oscillators, and "AS" type TTL flip flops going beyond 100 MHz, this poses no problem.

Figure 8.1 illustrates an example of such a clock generator based on the frequency division principle. The 2 units, U2a and U2b are initially set to opposite states. They will maintain this interrelationship indefinitely. The two gates U3a and U3b are used to derive the 75/25% and 25/75% cycle for CLK1 and CLK2 respectively.

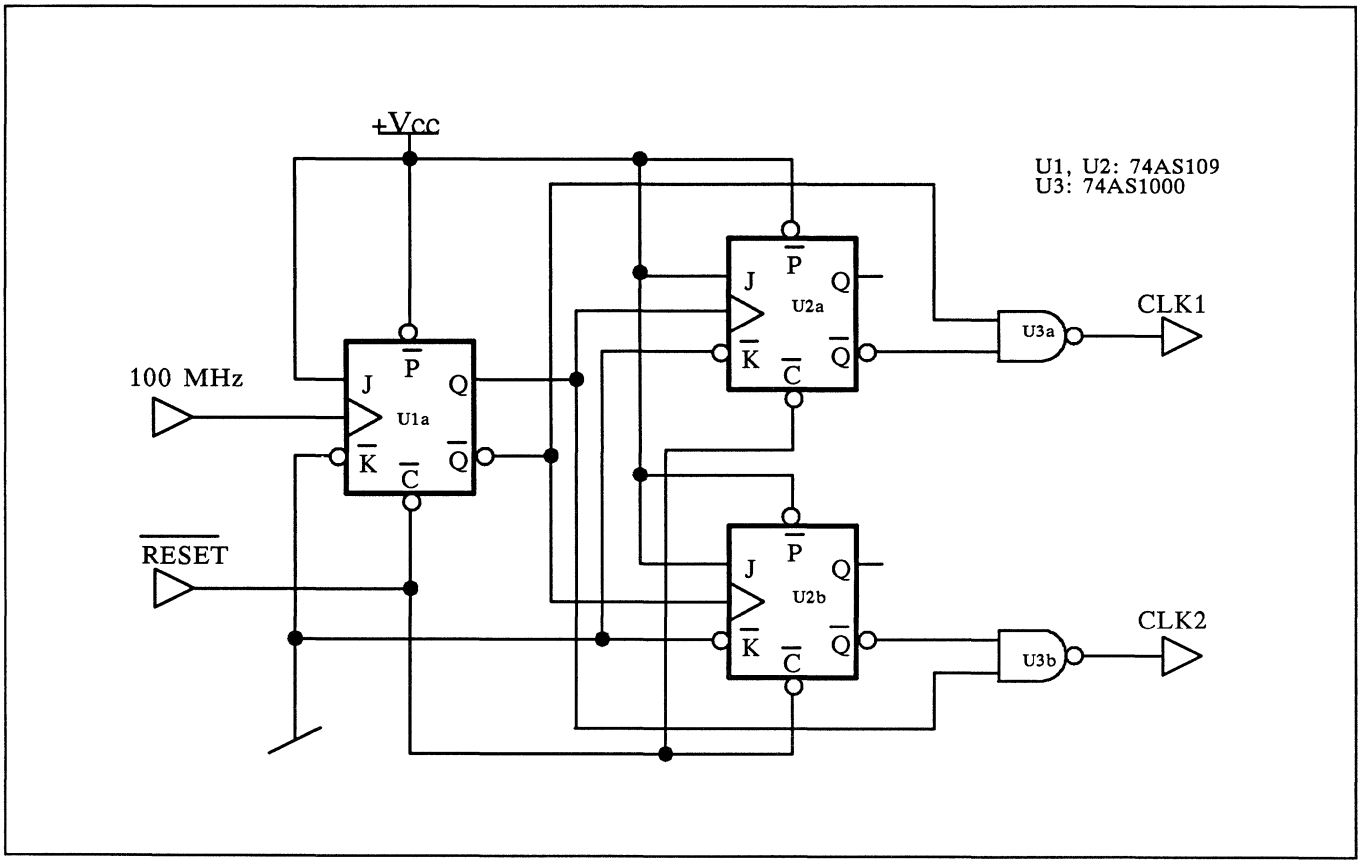
To minimize any skew between CLK1 & CLK2, U3a and U3b must be part of a 74AS1000 unit, while U2a and U2b must occupy one 74AS109 unit.

8.2. External Address Pipeline

Figure 8.2 shows an External Address Pipeline and associated logic.

The ADR<31:0> Address Bus is unlatched, and therefore not valid for the duration of a bus cycle. An external set of registered latches (U2) is therefore required to maintain the address bus state throughout a bus cycle, and drive any cache or memory subsystem.

Figure 8.1 Clock Circuit



During a cache miss, the processor is typically held in the cycle following the one that actually caused the miss. A second set of registered latches (U3) is therefore required to generate the previous address and associated cycle parameters.

Clocking of the first level registers is done by gating HAL with CLK1. The processor controls HAL to maintain address integrity on the first set of latches. To utilize the address setup time, and not exceed the hold time, CLK1 should also be delayed by one gate level before entering the processor. Ideally the gates used to invert CLK1, and gate HAL, would reside in one package to minimize skew. The first level registers are only tri-stated during miss processing.

The second level registers must be clocked by gating CLK1 with MHOLD to guarantee a stable address during miss processing. They are normally tri-stated as they only are used during miss processing.

9. Processor Specifications

Figures 9.1–9.10 show MB86901 timing specifications, test load, and output delay. AC timing parameters are listed in Table 9.4.

All timing parameters are referenced to the 1.5V midpoint of CLK1 rising edges.

Figure 8.2 External Address Pipeline

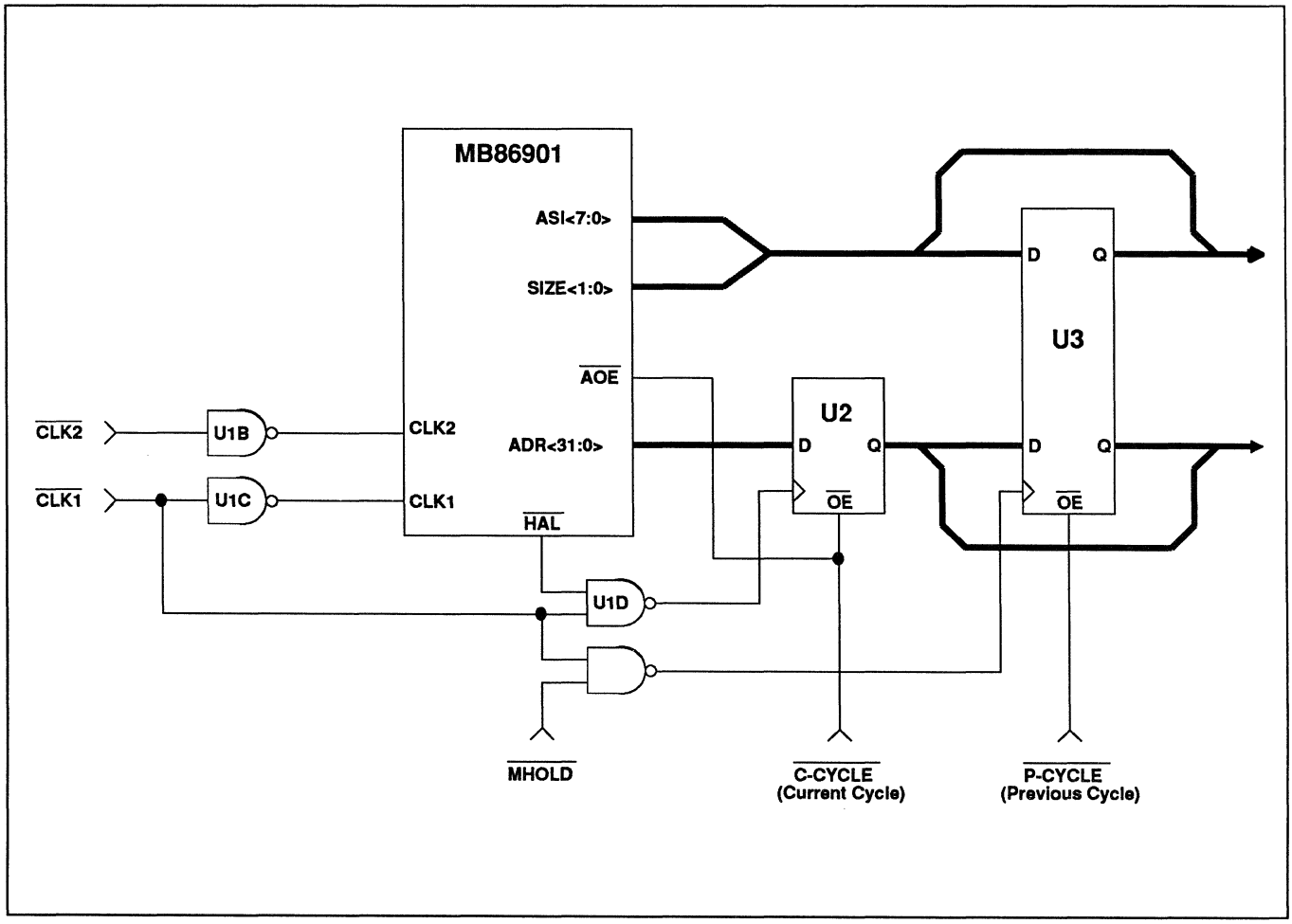


Table 9.1 Absolute Maximum Ratings

Rating		Symbol	Min.	Max.	Unit
Supply Voltage		V_{DD}	V_{SS} - 0.5	6.0	V
Input Voltage		V_I	V_{SS} - 0.5	V_{DD} + 0.5	V
Output Voltage		V_O	V_{SS} - 0.5	V_{DD} + 0.5	V
Temperature Under Bias	Ceramic Plastic	T_{bias}	-40 -25	125 85	• C
Storage Temperature	Ceramic Plastic	T_{stg}	-65 -40	150 125	• C

Note: (*) Permanent device damage may occur if Absolute Maximum Ratings are exceeded. Functional operation should be restricted to the conditions as detailed in the operational sections of the data sheet. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Table 9.2 Recommended Operating Conditions

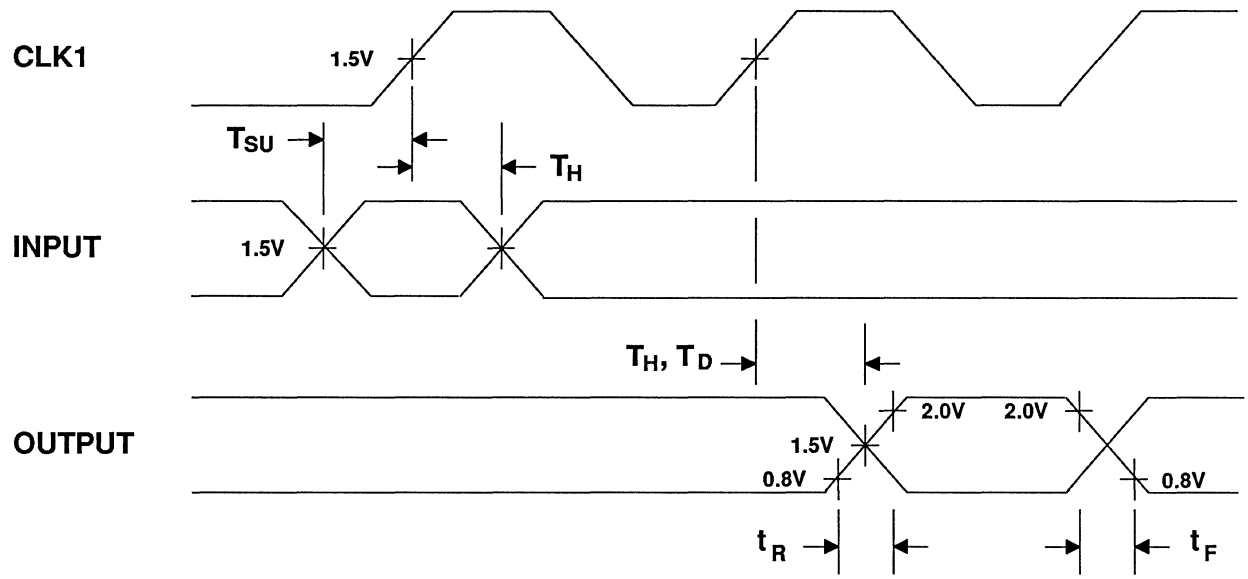
Parameter	Symbol	Condition	Min	Typ	Max	Units
Supply Voltage	V_{DD}		4.75	5.0	5.25	V
Power Supply Current	I_{DDS}	$V_{IH} = V_{DD}$, $V_{IL} = V_{SS}$	0		0.2	mA
Output High Voltage	V_{OH}	I_{OH} = -2 mA	4.0		V_{DD}	V
Output Low Voltage	V_{OL}	I = 3.2 mA	V_{SS}		0.4	V
Input High Voltage	V_{IH}		2.2		--	V
Input Low Voltage	V_{IL}		--		0.8	V
Input Leakage Current	I_{IN}	V_{IN} = 0 to V_{DD}	-10		10	uA
Tri-state Leakage Current	I_{IN}	V_{IN} = 0 to V_{DD}	-40		40	uA
Operating Temperature	T_A		0		70	•C

Table 9.3 Capacitance & Termination

(T = 25 C , V_{DD}= V_I = 0 Volt , Frequency = 1 MHz)

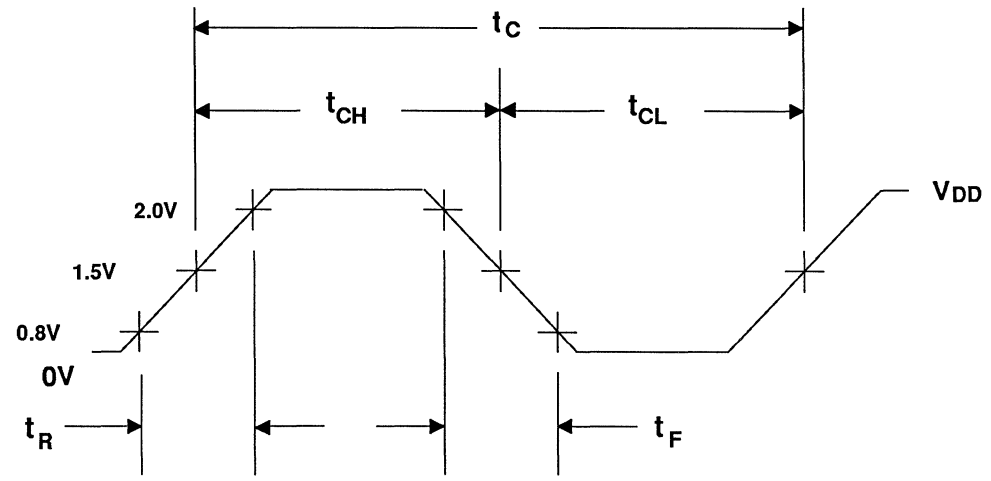
Parameter		Symbol	Min.	Typ.	Max.	Unit
Input Pin Capacitance		C_{IN}			16	pF
Output Pin Capacitance		C_{OUT}			16	pF
I/O Pin Capacitance		$C_{I/O}$			16	pF
Input Pull-Up Resistor	$V_{IH} = V_{DD}$ $V_{IL} = V_{SS}$	RP	25		100	KΩ

Figure 9.1 Signal AC Measurement Points



Note:
T_{SU} = Setup Time, T_H = Hold Time, T_D = Delay Time, t_R = Rise Time, t_F = Fall Time

Figure 9.2 Clock AC Measurement Points



Note:
t_C = Clock Period, t_{CH} = Clock High Time, t_{CL} = Clock Low Time,
t_R = Rise Time, t_F = Fall Time

Table 9.4 AC Characteristics

V_{CC}= 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 Clock requirement

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Min	Max	Min	Max		
1	System Clock Cycle Time (T)		50		40	ns	
2	System Clock Rise Time		3		2	ns	
3	System Clock Fall Time		3		2	ns	
4	System Clocks (Clk1, Clk2) Skew Time		+/- 4		+/- 3	ns	Both CLK1 &CLK2 High Time = 37.5nS Low Time = 12.5 nS
5	System Clock (Clk1) High Time	35		28		ns	
6	System Clock (Clk1) Low Time	10		8		ns	
7	System Clock (Clk2) High Time	20		16		ns	
8	System Clock (Clk2) Low Time	11		9		ns	

Figure 9.3 Clocks Timing Diagram

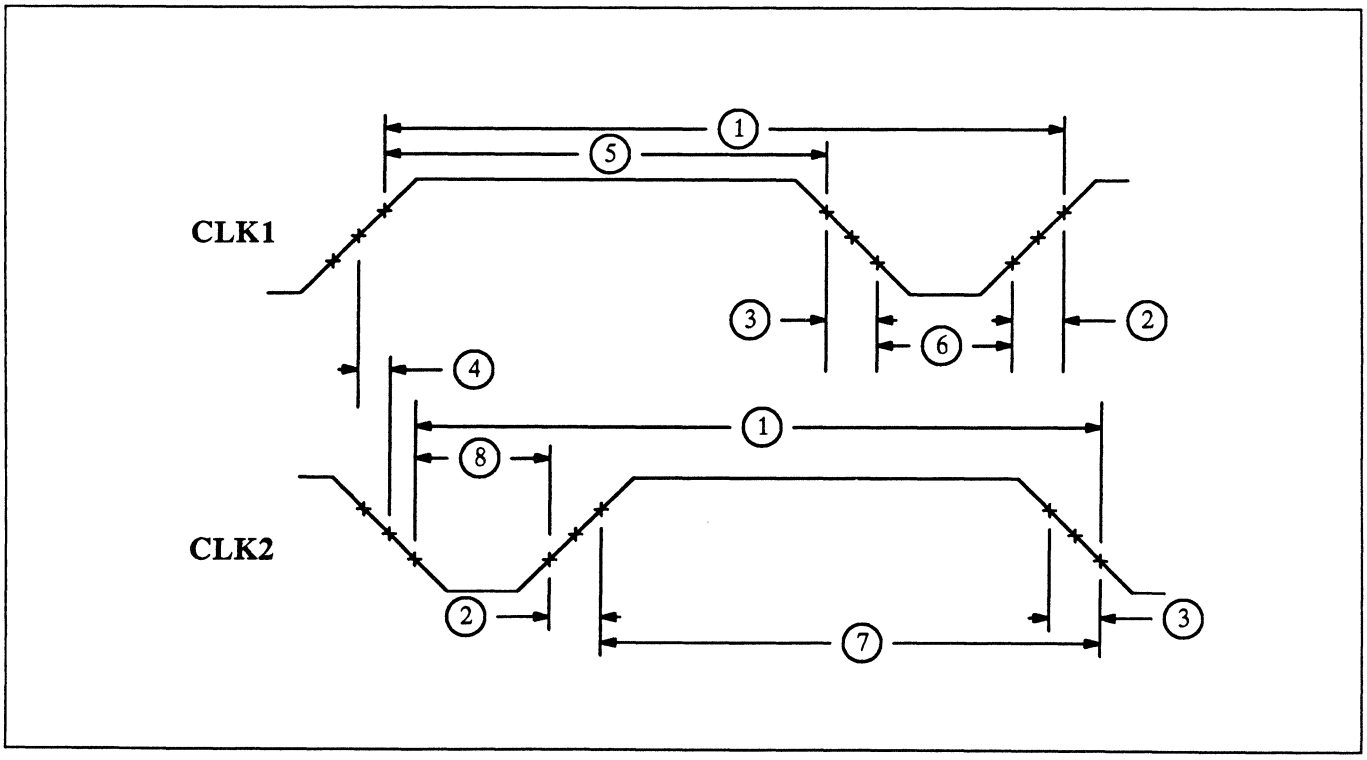


Table 9.4 AC Characteristics (cont.)

V_{CC} = 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 Address & Data Bus

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Min	Max	Min	Max		
9	ADR Valid Delay, From CLK1 Rising Edge		46		37	ns	
10	ADR Hold, From CLK1 Rising Edge	7		6		ns	
11	ASI Valid Delay, From CLK1 Rising Edge		29		23	ns	
12	ASI Hold, From CLK1 Rising Edge	7		5		ns	
13	Read Data Setup, Before CLK1 Rising Edge	4		4		ns	
14	Read Data Hold, After CLK1 Rising Edge	5		4		ns	
15	Write Data Valid Delay From CLK1 Rising Edge		33		26	ns	
16	Write Data Hold After CLK1 Rising Edge	6		5		ns	
17	Write Data Turn Off From CLK1		31		26	ns	
17a	Write Data Turn On From CLK1	11	27	9	23		

Figure 9.4 Address and Data Bus

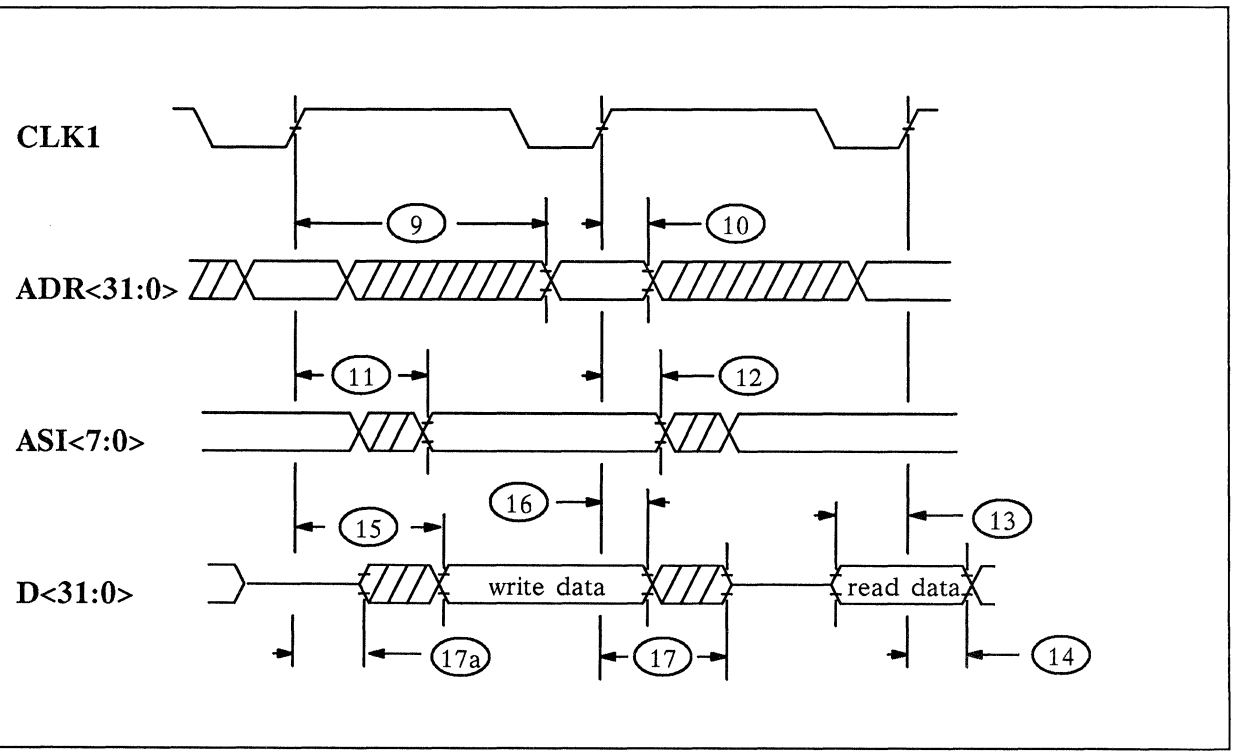


Table 9.4 AC Characteristics (cont.)

V_{cc} = 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 Address, Data & Control Tri-State

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Min	Max	Min	Max		
18	/AOE Turn Off Time	6	29	5	23	ns	
19	/AOE Turn On Time	6	19	5	15	ns	
20	/ADROE Turn Off Time	6	29	5	23	ns	
21	/ADROE Turn On Time	6	19	5	15	ns	
22	/DOE Turn Off Time	8	33	5	26	ns	
23	/DOE Turn On Time	8	23	5	18	ns	

Figure 9.5 Address, Data and Control Tri-state

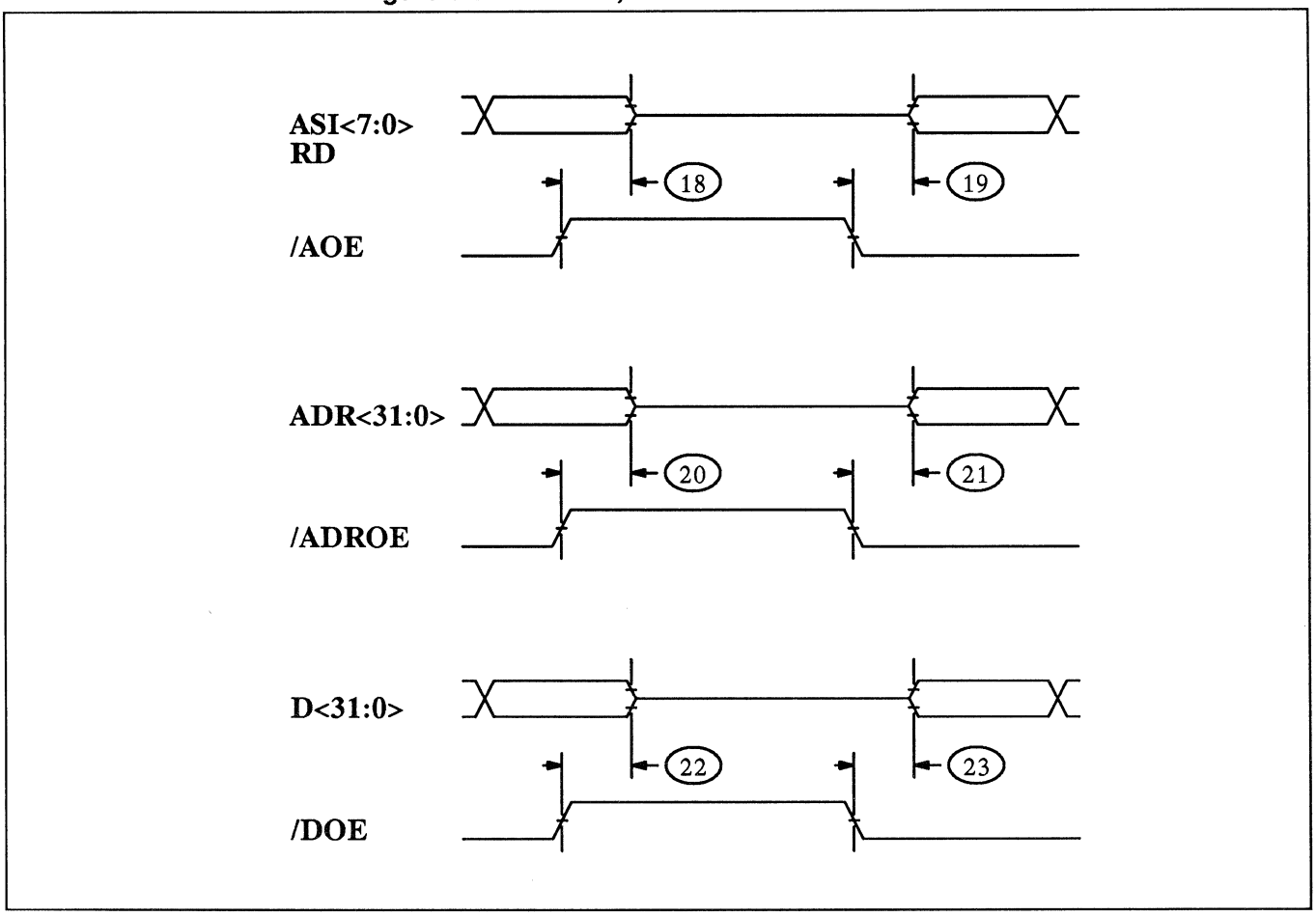


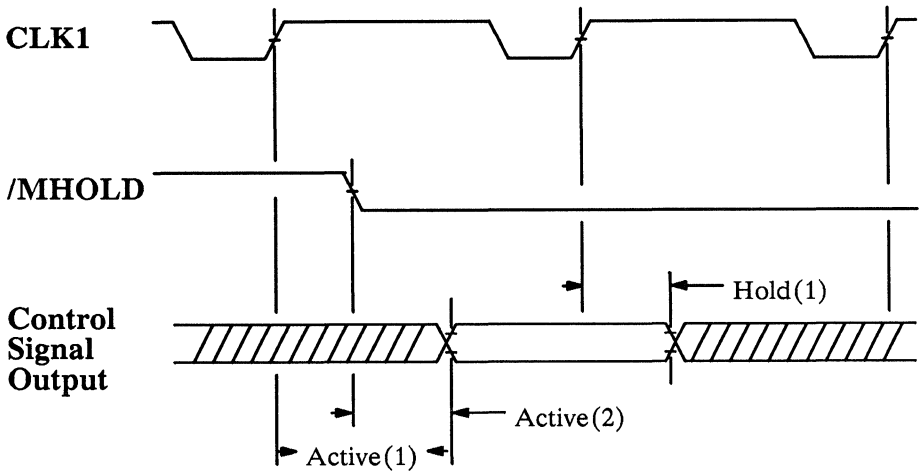
Table 9.4 AC Characteristics (cont.)

V_{CC} = 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 Control Signals, Output

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Active	Hold	Active	Hold		
24	SIZE, From CLK1 Rising Edge	22	6	18	5	ns	Max Rating for Active Min Rating for Hold
25	RD, From CLK! Rising Edge	22	6	18	5	ns	
26	/WE, From CLK1 Rising Edge	23	6	18	5	ns	
27	LDST, From CLK1 Rising Edge	22	6	18	5	ns	
28 29	NULL_CYC, From CLK1 Rising Edge From /MHOLD Falling Edge	40 24	6 –	32 19	5 –	ns	
30 31	/HAL From CLK1 Rising Edge From /MHOLD Falling Edge	37 23	6 –	30 18	5 –	ns	
32	LOCK, From CLK1 Rising Edge	25	6	20	5	ns	
33	DFETCH, From CLK1 Rising Edge	32	6	26	5	ns	
34	/ERROR, From CLK1 Rising Edge	32	6	26	5	ns	

Figure 9.6 Control Signals, Output



Timing Relation	Symbol No.
Active(1)	24, 25, 26, 27, 28 30, 32, 33, 34
Active(2)	29, 31
Hold(1)	24, 25, 26, 27, 28 30, 32, 33, 34

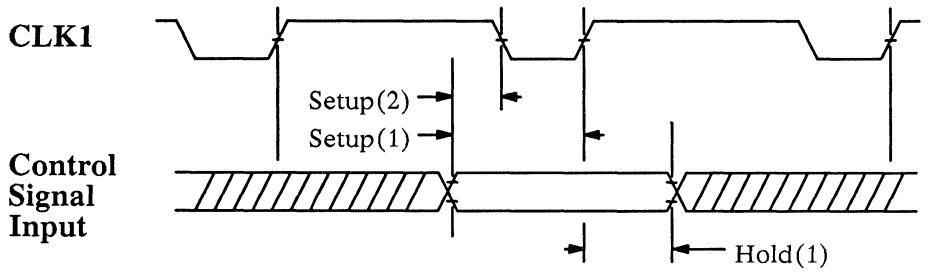
Table 9.4 AC Characteristics (cont.)

V_{CC} = 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 Control Signals, Input

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Setup	Hold	Setup	Hold		
35	/MDS, Before CLK1 Falling Edge	8	1	7	1	ns	Min Rating for Setup and Hold
36	Before CLK1 Rising Edge	25	–	20	–		
37	/MEXC, Before CLK1 Falling Edge	8	1	7	1	ns	
38	Before CLK1 Rising Edge	25	–	20	–	ns	
39	/MHOLDA, Before CLK1 Falling Edge	16	0	13	0	ns	
40	/MHOLDB, Before CLK1 Falling Edge	16	0	13	0	ns	
41	/MHOLDC, Before CLK1 Falling Edge	16	0	13	0	ns	
42	/SHOLD, Before CLK1 Falling Edge	16	0	13	0	ns	
43	/BHOLD, Before CLK1 Falling Edge	16	0	13	0	ns	
44	IRL, Before CLK1 Rising Edge	11	4	9	3	ns	
45	/RESET, Before CLK1 Rising Edge	4	5	3	4	ns	
	Active Time (cycles)	min 10		min 10		T	

Figure 9.7 Control Signals, Input



Timing Relation	Symbol No.
Setup(1)	36, 38, 44, 45
Setup(2)	35, 37, 39, 40, 41 42, 43
Hold(1)	35, 37, 39, 40, 41 42, 43, 44, 45

Table 9.4 AC Characteristics (cont.)

V_{cc}= 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 FP Bus, Output

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Active	Hold	Active	Hold		
46	F<31:0>, From CLK1 Rising Edge	41	6	33	5	ns	Max Rating for Active Min Rating for Hold
47	FINS, From CLK1 Rising Edge	32	6	26	5	ns	
48	FADR, From CLK1 Rising Edge	32	6	26	5	ns	
49	FEND, From CLK1 Rising Edge	32	6	26	5	ns	
50	FLUSH, From CLK1 Rising Edge	26	6	22	5	ns	
51	FXACK, From CLK1 Rising Edge	32	6	26	5	ns	

Figure 9.8 FP Bus, Output

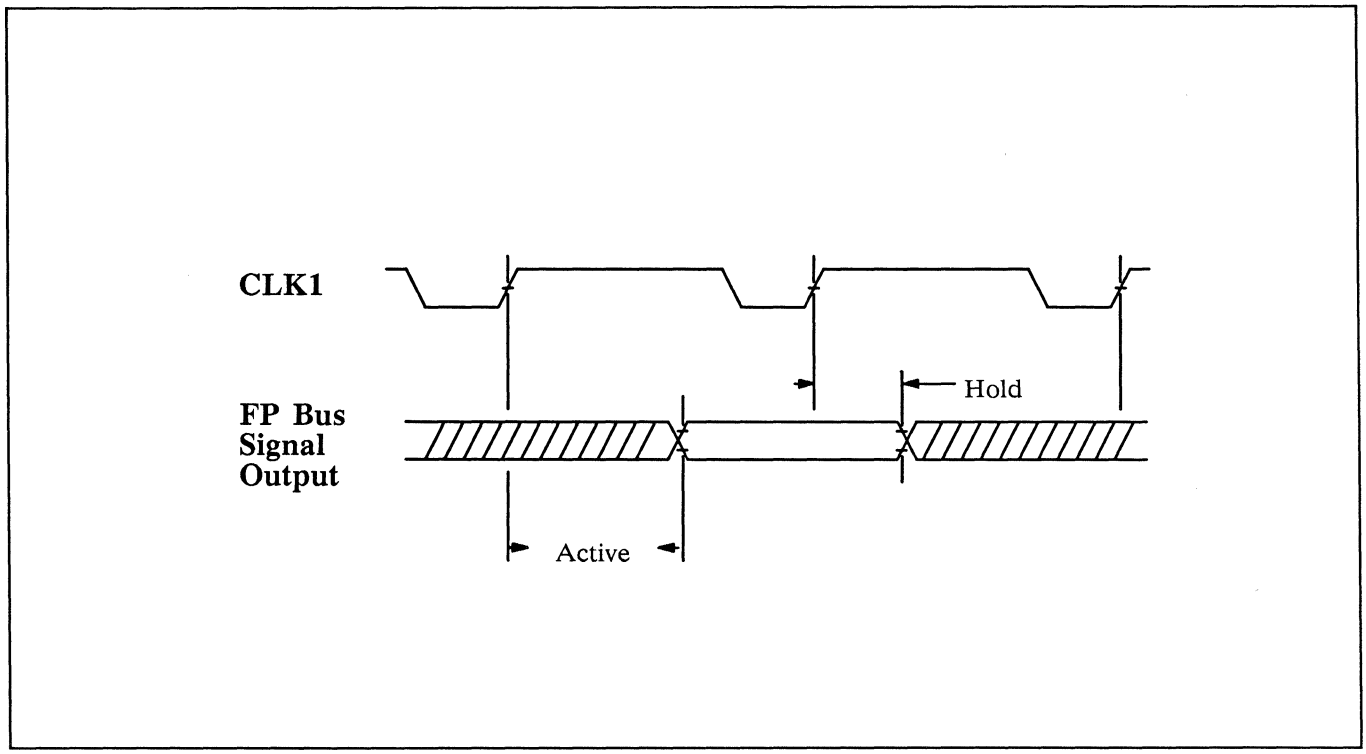


Table 9.4 AC Characteristics (cont.)

V_{CC} = 5V ± 5% T_A = 0 to +70 °C Capacitance = 50 pF

MB86901 FP Bus, Input

Symbol	Parameter	20 MHz		25 MHz		Units	Test Conditions
		Setup	Hold	Setup	Hold		
52	/FHOLD, Before CLK1 Rising Edge	4	8	4	7	ns	Min Rating for Setup and Hold
53	/FEXC, Before CLK1 Rising Edge	4	6	4	5	ns	
54	/FCC, Before CLK1 Rising Edge	4	6	4	5	ns	
55	FCCV, Before CLK1 Rising Edge	4	6	4	5	ns	

Figure 9.9 FP Bus, Input

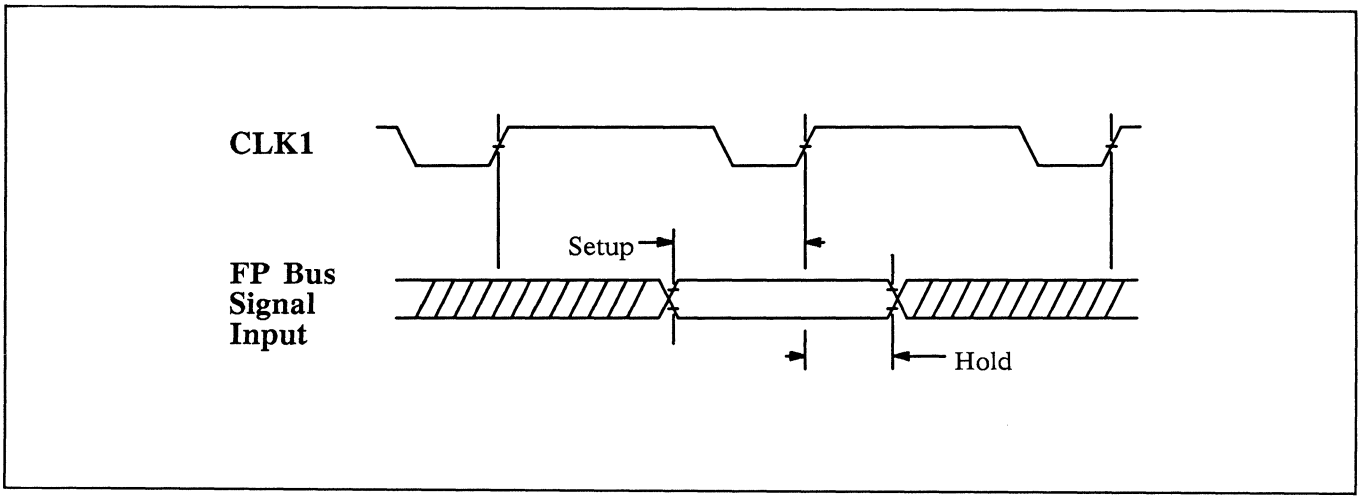


Figure 9.10 Signal Output Test Load

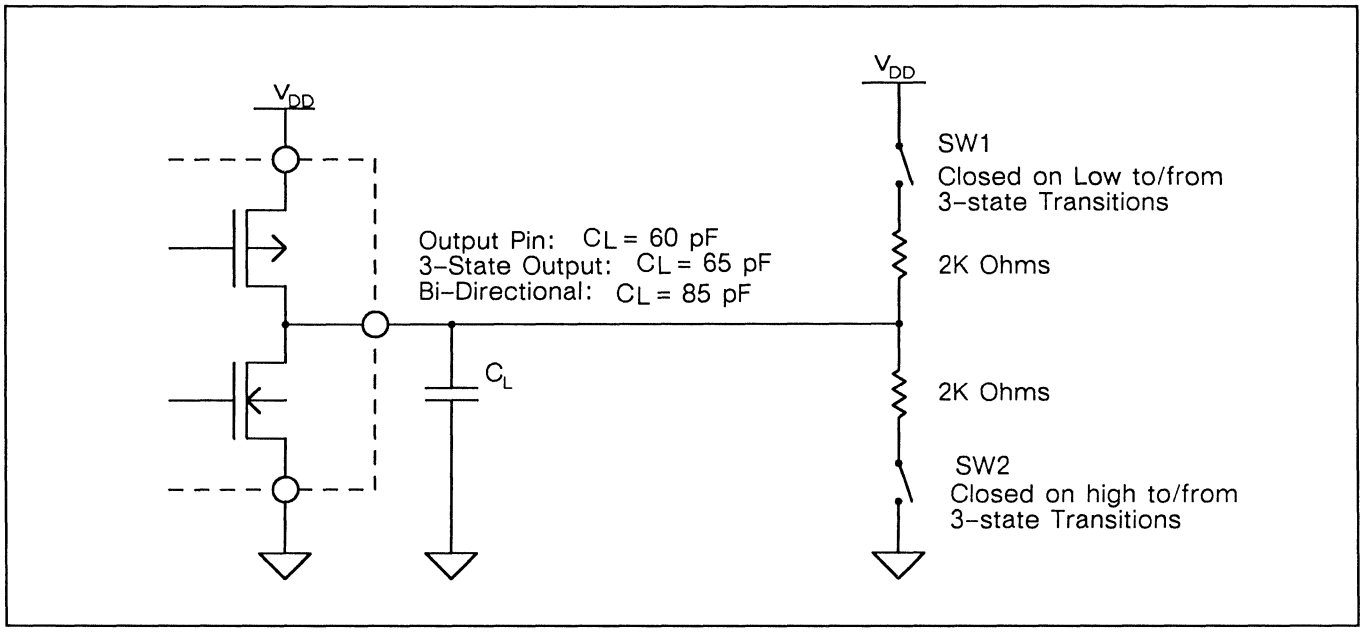


Figure 9.11 Maximum Output Delay vs. Capacitance Loading

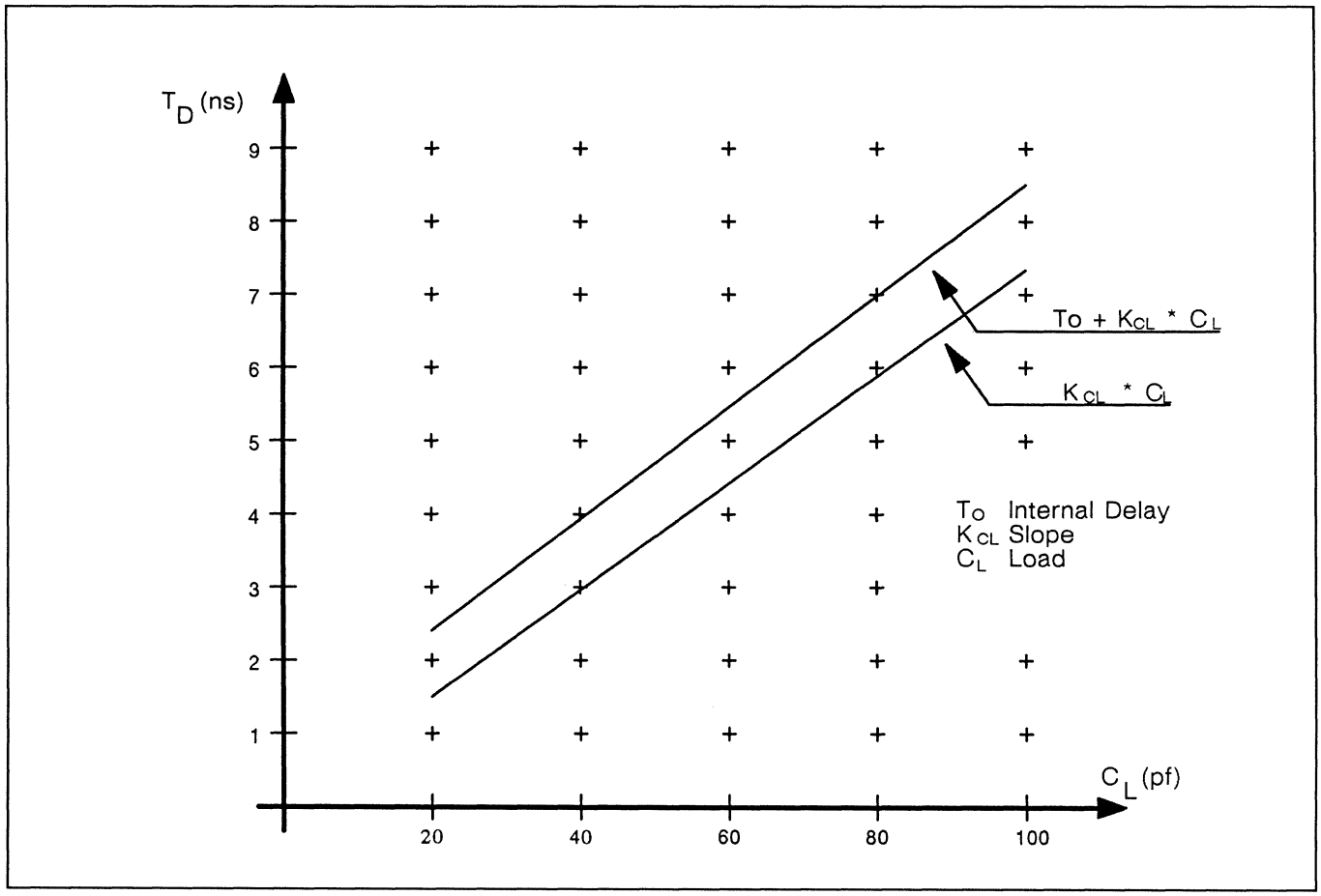


Figure 9.12 179-Lead Plastic Pin Grid Array Package

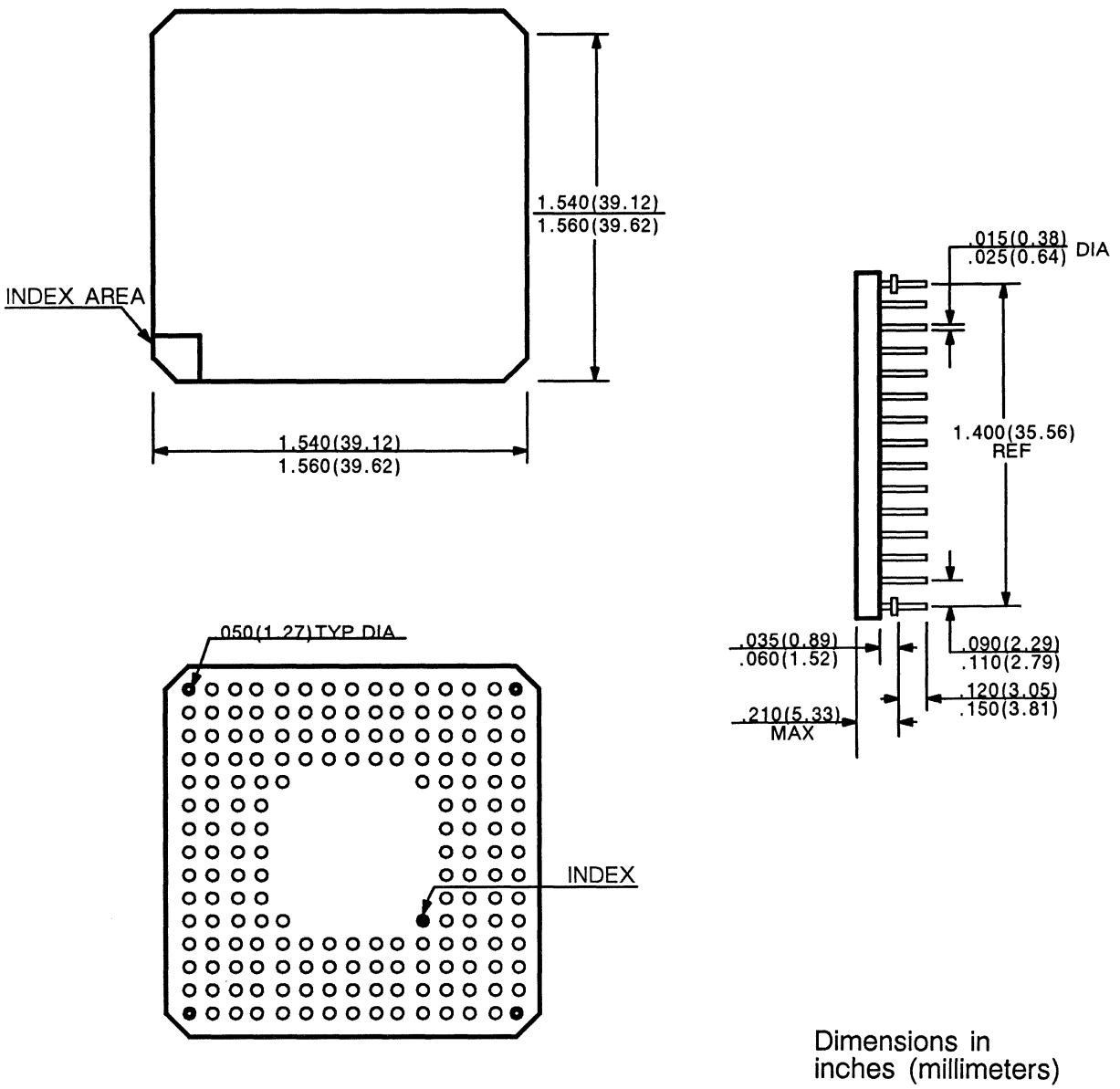
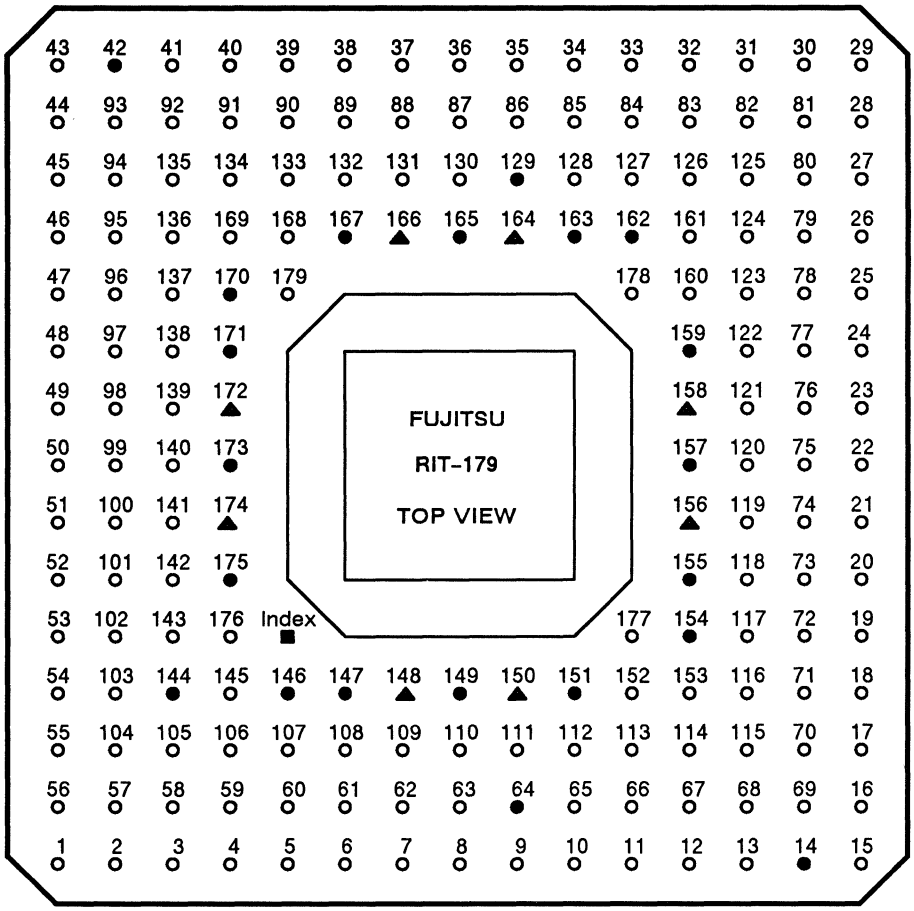


Figure 9.13 Pin Assignment



Pin Group	Symbol	Pinout
Vcc	▲	148, 150, 156, 158, 164, 166, 172, 174
GND	●	14, 42, 64, 129, 144, 146, 147, 149, 151, 154, 155, 157, 159, 162, 163, 167, 170, 171,173, 175
Signal	O	———
Index	■	180

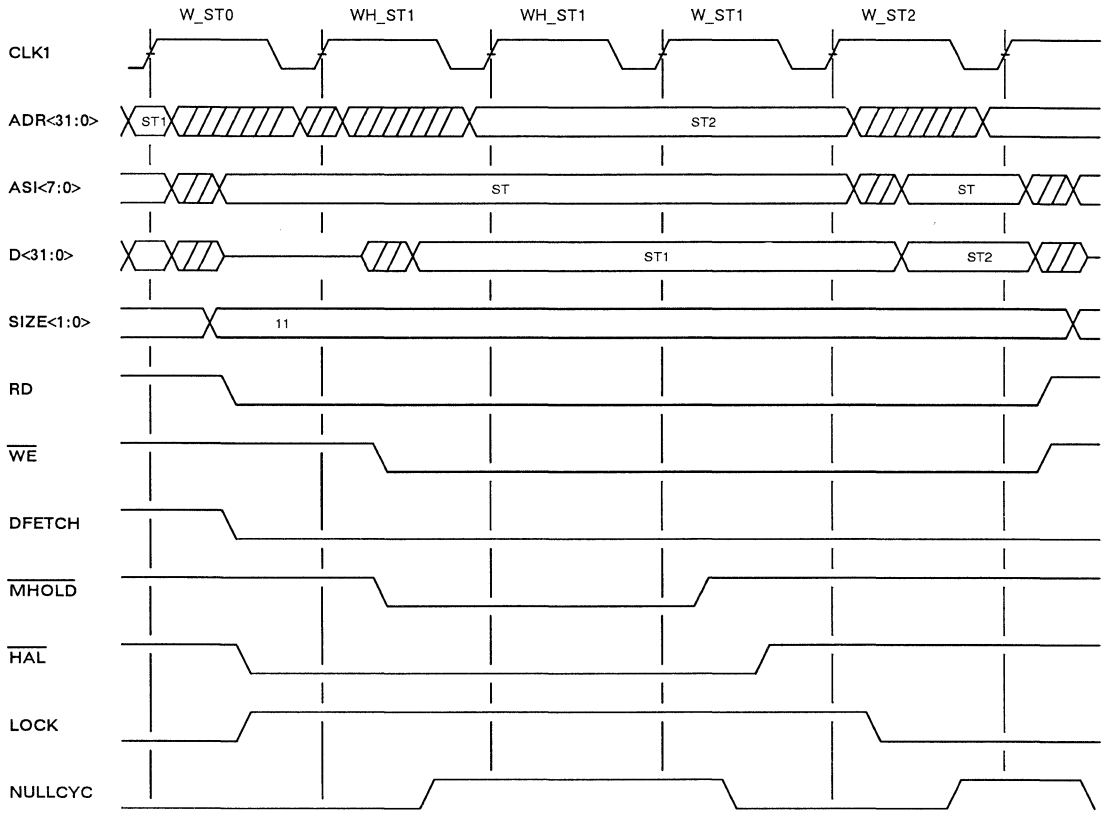
Figure 9.14 MB86901 Pin Out by Pin Number

PIN	I/O	NAME	PIN	I/O	NAME	PIN	I/O	NAME
1	O	ADR10	61	O	ADR27	121	O	ASI3
2	O	ADR02	62	O	ADR25	122	O	ASI6
3	O	ADR13	63	O	ADR28	123	I	RESET_
4	O	ADR24	64	-	VSS	124	O	DFETCH
5	O	ADR30	65	O	ADR23	125	I	FEXC_
6	I	FCCV	66	I	XTST	126	I	AOE_
7	O	ADR20	67	O	ADR11	127	I/O	D03
8	O	ADR21	68	O	ADR16	128	I/O	D30
9	O	ADR19	69	O	ADR08	129	-	VSS
10	O	ADR22	70	I	TC_	130	I/O	D21
11	O	ADR06	71	O	RD	131	I/O	D13
12	O	FXACK	72	I	MHOLDB_	132	I/O	D05
13	O	ADR03	73	O	SIZE0	133	I/O	D24
14	-	VSS	74	O	ASI2	134	I/O	D04
15	I	IRL2	75	I/O	D02	135	O	F18
16	O	ASI1	76	I	MEXC_	136	O	F26
17	O	FLUSH	77	O	ASI5	137	O	F02
18	O	WE_	78	I	SHOLD_	138	O	F27
19	I	IRL3	79	O	FINS	139	O	F07
20	O	HAL_	80	I	DOE_	140	O	F20
21	O	LOCK	81	O	NULL_CYC	141	O	F12
22	O	LDST	82	I/O	D27	142	O	F29
23	I	SDI	83	O	SDO	143	I	XACK
24	O	ASI4	84	I/O	D26	144	-	VSS
25	I	MDS_	85	O	ERROR_	145	O	ADR01
26	I	FHOLD	86	I/O	D06	146	-	VSS
27	I	ADROE_	87	I/O	D25	147	-	VSS
28	I	IRL0	88	I	FP_	148	-	VDD
29	I/O	D15	89	I	BCK	149	-	VSS
30	I/O	D07	90	I/O	D28	150	-	VDD
31	I/O	D11	91	I/O	D16	151	-	VSS
32	I/O	D23	92	I	IH_NULL_	152	I	BHOLD_
33	I/O	D01	93	I/O	D12	153	O	ADR12
34	I/O	D22	94	I/O	D14	154	-	VSS
35	I/O	D29	95	O	F06	155	-	VSS
36	I/O	D17	96	O	F23	156	-	VDD
37	I/O	D19	97	O	F19	157	-	VSS
38	I/O	D09	98	O	F03	158	-	VDD
39	I/O	D18	99	I	CLK1	159	-	VSS
40	I/O	D20	100	I	FCC0	160	I/O	D00
41	I/O	D31	101	O	F04	161	O	FEND
42	-	VSS	102	I	CLK2	162	-	VSS
43	I/O	D10	103	O	F17	163	-	VSS
44	O	F10	104	O	F09	164	-	VDD
45	O	F11	105	O	ADR04	165	-	VSS
46	O	F15	106	O	F14	166	-	VDD
47	I	FCC1	107	O	ADR09	167	-	VSS
48	O	F16	108	O	ADR26	168	I/O	D08
49	O	F28	109	O	ADR29	169	O	F30
50	O	F00	110	O	ADR18	170	-	VSS
51	O	F24	111	O	ADR31	171	-	VSS
52	O	F08	112	O	ADR17	172	-	VDD
53	O	F21	113	O	ADR07	173	-	VSS
54	O	F31	114	O	ADR14	174	-	VDD
55	O	F25	115	I	IRL1	175	-	VSS
56	O	F05	116	O	FADR	176	O	F13
57	O	F01	117	O	ASI0	177	O	ADR00
58	O	ADR15	118	I	MHOLDA_	178	O	ASI7
59	O	ADR05	119	I	MHOLDC_	179	O	F22
60	I	XSM	120	O	SIZE1			

ERRATA

Page	Correction
22,23	On page 22, paragraph 2, line 2, and page 23, paragraph 3, line 2, replace "r[rs1] = r[rs2]" with "r[rs1] + r[rs2]".
34	In Figure 7.1, Processor Signals, on the Bus Control input line for $\overline{\text{MHOLD}}$ <A–C>, $\overline{\text{SHOLD}}$, replace the value "5" with a "4".
53	The correct title of Figure 7.11 is: **Load Double (Cache Miss)**. The correct **Store Double (Cache Miss)** figure is shown below:
54	In Figure 7.12, **Atomic Load-Store (Cache Miss)**, replace "AL<17:0>" with "ADR<31:0>".
95	Under **Ordering Information** (opposite errata page), replace "MB86901-APR-G" with "MB86901ACR-G" and replace "MB86901-APR-G25" with "MB86901ACR-G25".

STORE DOUBLE WORD (CACHE MISS)



FUJITSU MICROELECTRONICS, INC. SALES OFFICES

CALIFORNIA
10600 N. DeAnza Blvd., # 225
Cupertino, CA 95014
(408) 996-1600

840 Newport Center Dr., # 460
Newport Beach, CA 92660-6323
(714) 720-9688

GEORGIA
3500 Parkway Lane, # 210
Norcross, GA 30092
(404) 449-8539

ILLINOIS
One Pierce Place, # 910
Itasca, IL 60143-2681
(312) 250-8580

MASSACHUSETTS
75 Wells Ave., # 5
Newton Center, MA 02159-3251
(617) 964-7080

MINNESOTA
3460 Washington Dr., #209
Eagan, MN 55122-1303
(612) 454-0323

NEW YORK
601 Veterans Memorial Highway, #P
Hauppauge, NY 11788-1054
(516) 361-6565

OREGON
5285 S.W. MEADOWS RD., #222
LAKE OSWEGO, OR 97035-9998
(503) 684-4545

TEXAS
14785 Preston Rd., #670
Dallas, TX 25240
(214) 233-9394

Ordering Information

Part Number	Clock Speed
MB86901-APR-G	20 MHz
MB86901-APR-G25	25 MHz

Advanced Products Division

For further information outside the U.S., please contact:

FUJITSU LIMITED
Semiconductor Marketing
Furukawa Sogo Bldg, 6-1 Marunouchi, 2-chome
Chiyoda-ku, Tokyo 100, Japan
TEL 011-81-3-216-3211 • FAX 011-81-3-216-9771

FUJITSU MIKROELEKTRONIK GmbH
Arabella Center 9.0G./A
Lyoner Strasse 44-48, D-6000, Frankfurt am Main 71, F.R. Germany
TEL 011-49-69-66320 • TELEX 0411 963

FUJITSU MICROELECTRONICS PACIFIC ASIA LIMITED
805 Tsimshatsui Centre, 66 Mody Road
Tsimshatsui East, Kowloon, Hong Kong
TEL 3-732-0100 • FAX 3-722-7984 • TELEX 31959 FUJIS HK

FUJITSU MICROELECTRONICS, INC.
Advanced Products Division
50 Rio Robles, San Jose, CA 95134-1804 • TEL (408) 922-9000 • FAX (408) 432-9070

880925BBGG Printed in U.S.A

*SPARC is a trademark of Sun Microsystems, Inc.