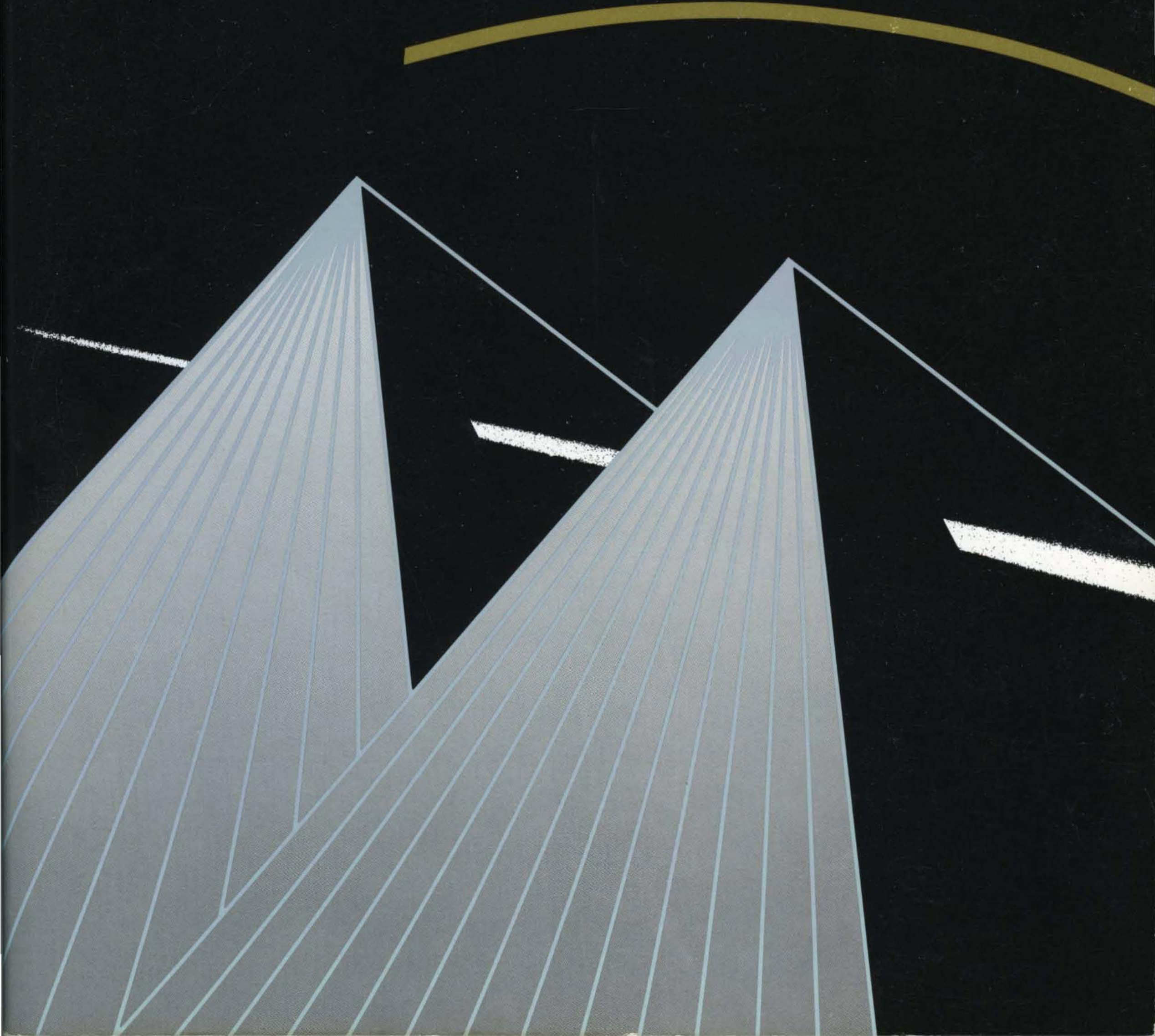
intel®

i860™ 64-BIT MICROPROCESSOR

i860™



i860™ 64-Bit Microprocessor

- **Parallel Architecture that Supports Up to Three Operations per Clock**
  - **— One Integer or Control Instruction per Clock**
  - **— Up to Two Floating-Point Results per Clock**
- **High Performance Design**
  - **— 33.3/40 MHz Clock Rates**
  - **— 80 Peak Single Precision MFLOPs**
  - **— 60 Peak Double Precision MFLOPs**
  - **— 64-Bit External Data Bus**
  - **— 64-Bit Internal Instruction Cache Bus**
  - **— 128-Bit Internal Data Cache Bus**
- **High Level of Integration on One Chip**
  - **— 32-Bit Integer and Control Unit**
  - **— 32/64-Bit Pipelined Floating-Point Adder and Multiplier Units**
  - **— 64-Bit 3-D Graphics Unit**
  - **— Paging Unit with Translation Lookaside Buffer**
  - **— 4 Kbyte Instruction Cache**
  - **— 8 Kbyte Data Cache**
- **Compatible with Industry Standards**
  - **— ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic**
  - **— 386™/486™ Microprocessor Data Formats and Page Table Entries**
  - **— JEDEC 168-pin Ceramic Pin Grid Array Package (see *Packaging Outlines and Dimensions*, order #231369)**
- **Easy to Use**
  - **— On-Chip Debug Register**
  - **— Assembler, Linker, Simulator, Debugger, C and FORTRAN Compilers, FORTRAN Vectorizer, Scalar and Vector Math Libraries for both OS/2* and UNIX* Environments**

The Intel i860™ Microprocessor (order codes A80860-33 and A80860-40) delivers supercomputing performance in a single VLSI component. The 64-bit design of the 860 microprocessor balances integer, floating point, and graphics performance for applications such as engineering workstations, scientific computing, 3-D graphics workstations, and multiuser systems. Its parallel architecture achieves high throughput with RISC design techniques, pipelined processing units, wide data paths, large on-chip caches, million-transistor design, and fast one-micron CHMOS IV silicon technology.

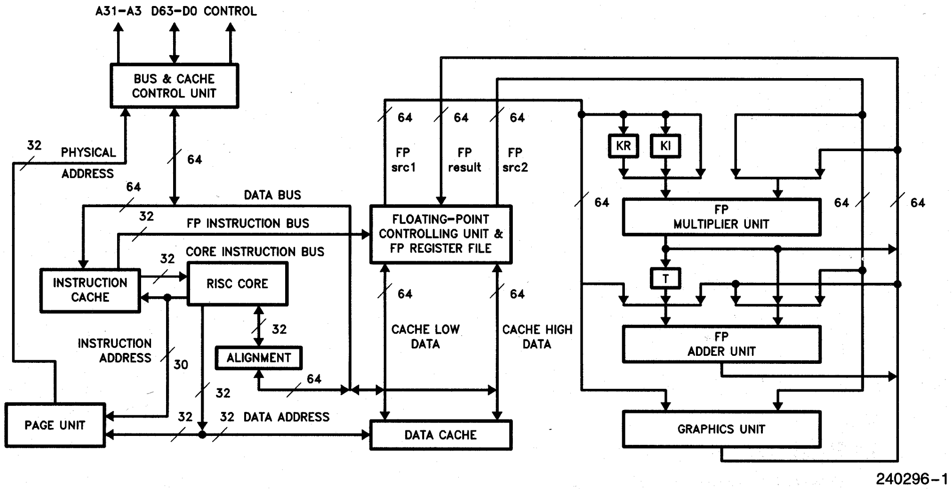


Figure 0.1. Block Diagram

TABLE OF CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

1.0 FUNCTIONAL DESCRIPTION

As shown by the block diagram on the front page, the 860 microprocessor consists of 9 units:

1. Core Execution Unit
2. Floating-Point Control Unit
3. Floating-Point Adder Unit
4. Floating-Point Multiplier Unit
5. Graphics Unit
6. Paging Unit
7. Instruction Cache
8. Data Cache
9. Bus and Cache Control Unit

The core execution unit controls overall operation of the 860 microprocessor. The core unit executes load, store, integer, bit, and control-transfer operations, and fetches instructions for the floating-point unit as well. A set of 32 x 32-bit general-purpose registers are provided for the manipulation of integer data. Load and store instructions move 8-, 16-, and 32-bit data to and from these registers. Its full set of integer, logical, and control-transfer instructions give the core unit the ability to execute complete systems software and applications programs. A trap mechanism provides rapid response to exceptions and external interrupts. Debugging is supported by the ability to trap on data or instruction reference.

The floating-point hardware is connected to a separate set of floating-point registers, which can be accessed as 16 x 64-bit registers, or 32 x 32-bit registers. Special load and store instructions can also access these same registers as 8 x 128-bit registers. All floating-point instructions use these registers as their source and destination operands.

The floating-point control unit controls both the floating-point adder and the floating-point multiplier, issuing instructions, handling all source and result exceptions, and updating status bits in the floating-point status register. The adder and multiplier can operate in parallel, producing up to two results per clock. The floating-point data types, floating-point instructions, and exception handling all support the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985).

The floating-point adder performs addition, subtraction, comparison, and conversions on 64- and 32-bit floating-point values. An adder instruction executes in three to four clocks; however, in pipelined mode, a new result is generated every clock.

The floating-point multiplier performs floating-point and integer multiply and floating-point reciprocal operations on 64- and 32-bit floating-point values. A multiplier instruction executes in three to four clocks; however, in pipelined mode, a new result can be generated every clock for single-precision and every other clock for double precision.

The graphics unit has special integer logic that supports three-dimensional drawing in a graphics frame buffer, with color intensity shading and hidden surface elimination via the Z-buffer algorithm. The graphics unit recognizes the pixel as an 8-, 16-, or 32-bit data type. It can compute individual red, blue, and green color intensity values within a pixel; but it does so with parallel operations that take advantage of the 64-bit internal word size and 64-bit external bus. The graphics features of the 860 microprocessor assume that the surface of a solid object is drawn with polygon patches whose shapes approximate the original object. The color intensities of the vertices of the polygon and their distances from the viewer are known, but the distances and intensities of the other points must be calculated by interpolation. The graphics instructions of the 860 microprocessor directly aid such interpolation.

The paging unit implements protected, paged, virtual memory via a 64-entry, four-way set-associative memory called the TLB (Translation Lookaside Buffer). The paging unit uses the TLB to perform the translation of logical address to physical address, and to check for access violations. The access protection scheme employs two levels of privilege: user and supervisor.

The instruction cache is a two-way set-associative memory of four Kbytes, with 32-byte blocks. It transfers up to 64 bits per clock (266 Mbyte/sec at 33.3 MHz).

The data cache is a two-way set-associative memory of eight Kbytes, with 32-byte blocks. It transfers up to 128 bits per clock (533 Mbyte/sec at 33.3 MHz). The 860 microprocessor normally uses write-back caching, i.e. memory writes update the cache (if applicable) without necessarily updating memory immediately; however, caching can be inhibited by software where necessary.

The bus and cache control unit performs data and instruction accesses for the core unit. It receives cycle requests and specifications from the core unit, performs the data-cache or instuction-cache miss processing, controls TLB translation, and provides the interface to the external bus. Its pipelined structure supports up to three outstanding bus cycles.

2.0 PROGRAMMING INTERFACE

The programmer-visible aspects of the architecture of the 860 microprocessor include data types, registers, instructions, and traps.

2.1 Data Types

The 860 microprocessor provides operations for integer and floating-point data. Integer operations are performed on 32-bit operands with some support also for 64-bit operands. Load and store instructions can reference 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit operands. Floating-point operations are performed on IEEE-standard 32- and 64-bit formats. Graphics oriented instructions operate on arrays of 8-, 16-, or 32-bit pixels.

2.1.1 INTEGER

An integer is a 32-bit signed value in standard two's complement form. A 32-bit integer can represent a value in the range $-2,147,483,648$ (-2^{31}) to $2,147,438,647$ ($+2^{31}-1$). Arithmetic operations on 8- and 16-bit integers can be performed by sign-extending the 8- or 16-bit values to 32 bits, then using the 32-bit operations.

There are also add and subtract instructions that operate on 64-bit long integers.

Load and store instructions may also reference (in addition to the 32- and 64-bit formats previously mentioned) 8- and 16-bit items in memory. When an 8- or 16-bit item is loaded into a register, it is converted to an integer by sign-extending the value to 32 bits. When an 8- or 16-bit item is stored from a register, the corresponding number of low-order bits of the register are used.

2.1.2 ORDINAL

Arithmetic operations are available for 32-bit ordinals. An ordinal is an unsigned integer. An ordinal can represent values in the range 0 to 4,294,967,295 ($+2^{32}-1$).

Also, there are add and subtract instructions that operate on 64-bit ordinals.

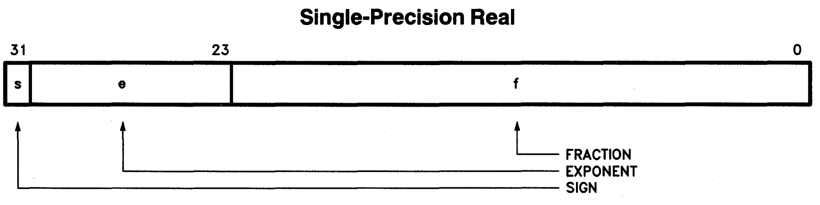
2.1.3 SINGLE- AND DOUBLE-PRECISION REAL

Figure 2.1 shows the real number formats. A single-precision real (also called "single real") data type is a 32-bit binary floating-point number. Bit 31 is the sign bit; bits 30..23 are the exponent; and bits 22..0 are the fraction. In accordance with ANSI/IEEE standard 754, the value of a single-precision real is defined as follows:

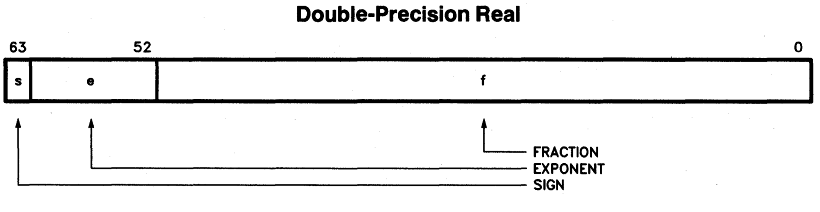
1. If $e = 0$ and $f \neq 0$ or $e = 255$ then generate a floating-point source-exception trap when encountered in a floating-point operation.
2. If $0 < e < 255$, then the value is $-1^{s} \times 1.f \times 2^{e-127}$.
3. If $e = 0$ and $f = 0$, then the value is signed zero.

A double-precision real (also called "double real") data type is a 64-bit binary floating-point number. Bit 63 is the sign bit; bits 62..52 are the exponent; and bits 51..0 are the fraction. In accordance with ANSI/IEEE standard 754, the value of a double-precision real is defined as follows:

1. If $e = 0$ and $f \neq 0$ or $e = 2047$, then generate a floating-point source-exception trap when encountered in a floating-point operation.
2. If $0 < e < 2047$, then the value is $-1^{s} \times 1.f \times 2^{e-1023}$.



240296-2



240296-3

Figure 2.1. Real Number Formats

3. If e = 0 and f = 0, then the value is signed zero.

The special values infinity, NaN ("Not a Number"), indefinite, and denormal generate a trap when encountered. The trap handler implements IEEE-standard results.

A double real value occupies an even/odd pair of floating-point registers. Bits 31..0 are stored in the even-numbered floating-point register; bits 63..32 are stored in the next higher odd-numbered floating-point register.

2.1.4 PIXEL

A pixel may be 8, 16, or 32 bits long depending on color and intensity resolution requirements. Regardless of the pixel size, the 860 microprocessor always operates on 64 bits worth of pixels at a time. The pixel data type is used by two kinds of instructions:

- The selective pixel-store instruction that helps implement hidden surface elimination.
- The pixel add instruction that helps implement 3-D color intensity shading.

To perform color intensity shading efficiently in a variety of applications, the 860 microprocessor defines three pixel formats according to Table 2.1.

Figure 2.2 illustrates one way of assigning meaning to the fields of pixels. These assignments are for illustration purposes only. The 860 microprocessor defines only the field sizes, not the specific use of each field. Other ways of using the fields of pixels are possible.

Table 2.1. Pixel Formats

Pixel Size (in bits)	Bits of Color 1 Intensity	Bits of Color 2 Intensity	Bits of Color 3 Intensity	Bits of Other Attribute (Texture)
8	N (≤ 8) bits of intensity*			8 − N
16	6	6	4	
32	8	8	8	8

The intensity attribute fields may be assigned to colors in any order convenient to the application.

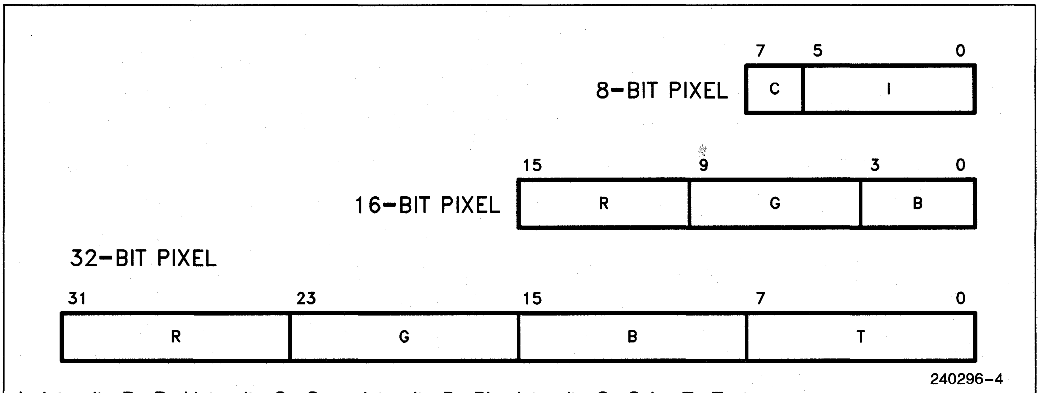
*With 8-bit pixels, up to 8 bits can be used for intensity; the remaining bits can be used for any other attribute, such as color. The intensity bits must be the low-order bits of the pixel.

2.2 Register Set

As Figure 2.3 shows, the 860 microprocessor has the following registers:

- An integer register file
- A floating-point register file
- Six control registers (**psr, epsr, db, dirbase, fir,** and **fsr)**
- Four special-purpose registers (KR, KI, T, and MERGE)

The control registers are accessible only by load and store control-register instructions; the integer and floating-point registers are accessed by arithmetic operations and load and store instructions. The special-purpose registers KR, KI, T, and MERGE are used by a few specific instructions.



I—Intensity, R—Red intensity, G—Green intensity, B—Blue intensity, C—Color, T—Texture
These assignments of specific meanings to the fields of pixels are for illustration purposes only. Only the field sizes are defined, not the specific use of each field.

Figure 2.2. Pixel Format Example

2.2.1 INTEGER REGISTER FILE

There are 32 integer registers, each 32-bits wide, referred to as **r0** through **r31**, which are used for address computation and scalar integer computations. Register **r0** always returns zero when read, independently of what is stored in it.

2.2.2 FLOATING-POINT REGISTER FILE

There are 32 floating-point registers, each 32-bits wide, referred to as **f0** through **f31**, which are used for floating-point computations. Registers **f0** and **f1** always return zero when read, independently of what is stored in them. The floating-point registers are also used by a set of integer operations, primarily for vector integer computations.

When accessing 64-bit floating-point or integer values, the 860 microprocessor uses an even/odd pair of registers. When accessing 128-bit values, it uses an aligned set of four registers (**f0**, **f4**, **f8**, ..., **f28**). The instruction must designate the lowest register number of the set of registers containing 64- or 128-bit values. Misaligned register numbers produce undefined results. The register with the lowest number contains the least significant part of the value. For 128-bit values, the register pair with the lower number contains the 64 bits at the lowest memory address; the register pair with the higher number contains the 64 bits at the highest address.

The 128-bit load and store instructions, along with the 128-bit data path between the floating-point registers and the data cache help to sustain an extraordinarily high rate of computation.

2.2.3 PROCESSOR STATUS REGISTER

The processor status register (**psr**) contains miscellaneous state information for the current process. Figure 2.4 shows the format of the **psr**.

- BR (Break Read) and BW (Break Write) enable a data access trap when the operand address matches the address in the **db** register and a read or write (respectively) occurs.
- Various instructions set CC (Condition Code) according to tests they perform. The branch-on-condition-code instructions test its value. The **bla** instruction sets and tests LCC (Loop Condition Code).
- IM (Interrupt Mode) enables external interrupts if set; disables interrupts if clear.
- U (User Mode) is set when the 860 microprocessor is executing in user mode; it is clear when the 860 microprocessor is executing in supervisor mode. In user mode, writes to some control registers are inhibited. This bit also controls the memory protection mechanism.

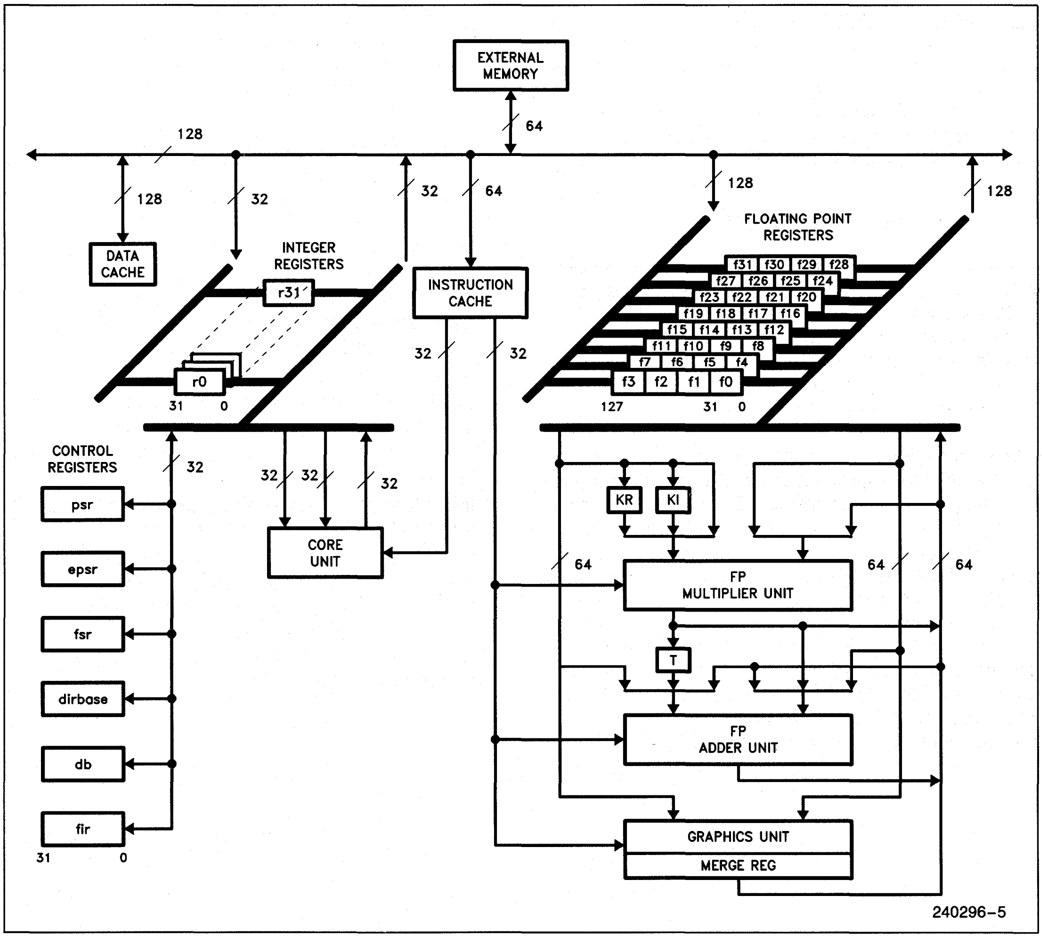


Figure 2.3. Registers and Data Paths

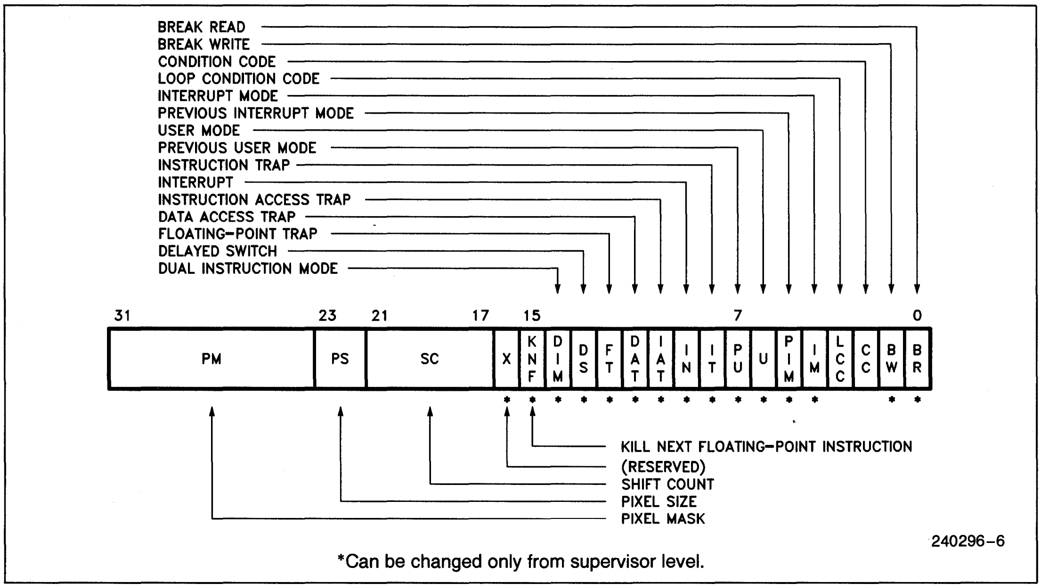


Figure 2.4 Processor Status Register

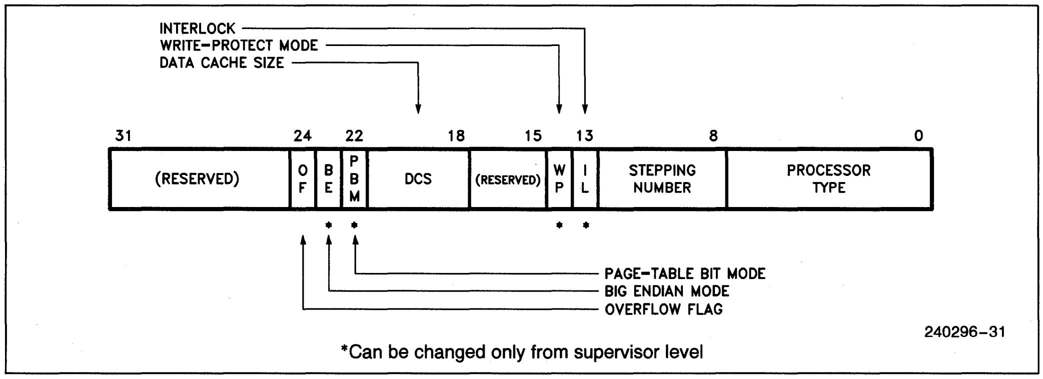


Figure 2.5 Extended Processor Status Register

- PIM (Previous Interrupt Mode) and PU (Previous User Mode) save the corresponding status bits (IM and U) on a trap, because those status bits are changed when a trap occurs. They are restored into their corresponding status bits when returning from a trap handler with a branch indirect instruction when a trap flag is set in the **psr**.
- FT (Floating-Point Trap), DAT (Data Access Trap), IAT (Instruction Access Trap), IN (Interrupt), and IT (Instruction Trap) are trap flags. They are set when the corresponding trap condition occurs. The trap handler examines these bits to determine which condition or conditions have caused the trap.
- DS (Delayed Switch) is set if a trap occurs during the instruction before dual-instruction mode is entered or exited. If DS is set and DIM (Dual Instruction Mode) is clear, the 860 microprocessor switches to dual-instruction mode one instruction after returning from the trap handler. If DS and DIM are both set, the 860 microprocessor switches to single-instruction mode one instruction after returning from the trap handler.
- When a trap occurs, the 860 microprocessor sets DIM if it is executing in dual-instruction mode; it clears DIM if it is executing in single-instruction mode. If DIM is set after returning from a trap handler, the 860 microprocessor resumes execution in dual-instruction mode.

- When KNF (Kill Next Floating-Point Instruction) is set, the next floating-point instruction is suppressed (except that its dual-instruction mode bit is interpreted). A trap handler sets KNF if the trapped floating-point instruction should not be reexecuted.
- SC (Shift Count) stores the shift count used by the last right-shift instruction. It controls the number of shifts executed by the double-shift instruction.
- PS (Pixel Size) and PM (Pixel Mask) are used by the pixel-store instruction and by the vector integer instructions. The values of PS control pixel size as defined by Table 2.2. The bits in PM correspond to pixels to be updated by the pixel-store instruction **pst.d**. The low-order bit of PM corresponds to the low-order pixel of the 64-bit source operand of **pst.d**. The number of low-order bits of PM that are actually used is the number of pixels that fit into 64-bits, which depends upon PS. If a bit of PM is set, then **pst.d** stores the corresponding pixel. Refer also to the **pst.d** instruction in section 8.

Table 2.2. Values of PS

Value	Pixel Size in bits	Pixel Size in bytes
00	8	1
01	16	2
10	32	4
11	(undefined)	(undefined)

2.2.4 EXTENDED PROCESSOR STATUS REGISTER

The extended processor status register (**epsr**) contains additional state information for the current process beyond that stored in the **psr**. Figure 2.5 shows the format of the **epsr**.

- The processor type is one for the 860 microprocessor.
- The stepping number has a unique value that distinguishes among different revisions of the processor.
- IL (Interlock) is set if a trap occurs after a **lock** instruction but before theload or store following the subsequent **unlock** instruction. IL indicates to the trap hadnler that a locked sequence has been interrupted.
- WP (write protect) controls the semantics of the W bit of page table entries. A clear W bit in either the directory or the page table entry causes writes to be trapped. When WP is clear, writes are trapped in user mode, but not in supervisor mode. When WP is set, writes are trapped in both user and supervisor modes.
- DCS (Data Cache Size) is a read-only field that tells the size of the on-chip data cache. The number of bytes actually available is $2^{12+\mathrm{DCS}}$; therefore, a value of zero indicates 4 Kbytes, one indicates 8 Kbytes, etc.

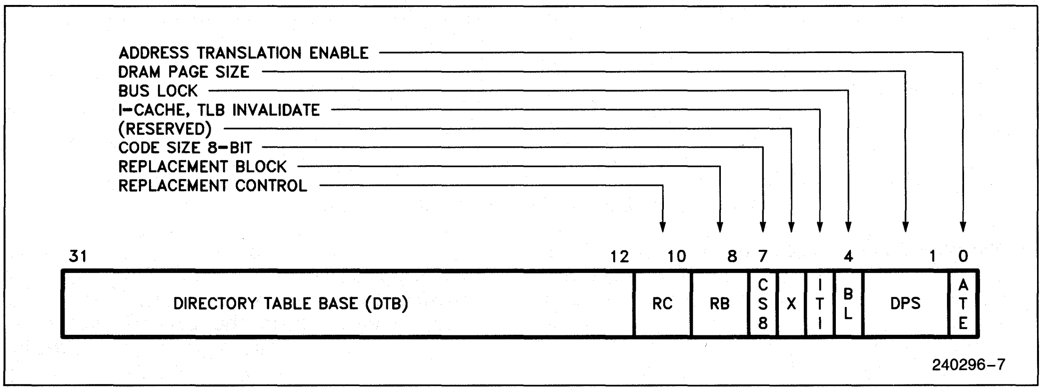


Figure 2.6. Directory Base Register

- PBM (Page-Table Bit Mode) determines which bit of page-table entries is output on the PTB pin. When PBM is clear, the PTB signal reflects bit CD of the page-table entry used for the current cycle. When PBM is set, the PTB signal reflects bit WT of the page-table entry used for the current cycle.
- BE (Big Endian) controls the ordering of bytes within a data item in memory. Normally (i.e. when BE is clear) the 860 microprocessor operates in little endian mode, in which the addressed byte is the low-order byte. When BE is set (bit endian mode), the low-order three bits of all load and store addresses are complemented, then masked to the appropriate boundary for alignment. This causes the addressed byte to be the most significant byte.
- OF (Overflow Flag) is set by **adds**, **addu**, **subs**, and **subu** when integer overflow occurs. For **adds** and **subs**, OF is set if the carry from bit 31 is different than the carry from bit 30. For **addu**, OF is set if there is a carry from bit 31. For **subu**, OF is set if there is no carry from bit 31. Under all other conditions, it is cleared by these instructions. OF controls the function of the **intovr** instruction.

2.2.5 DATA BREAKPOINT REGISTER

The data breakpoint register (**db**) is used to generate a trap when the 860 microprocessor makes a data-operand access to the address stored in this register. The trap is enabled by BR and BW in **psr**. When comparing, a number of low order bits of the address are ignored, depending on the size of the operand. For example, a 16-bit access ignores the low-order bit of the address when comparing to **db**; a 32-bit access ignores the low-order two bits. This ensures that any access that overlaps the address contained in the register will generate a trap.

2.2.6 DIRECTORY BASE REGISTER

The directory base register **dirbase** (shown in Figure 2.5) controls address translation, caching, and bus options.

- ATE (Address Translation Enable), when set, enables the virtual-address translation algorithm. The data cache must be flushed before changing the ATE bit.
- DPS (DRAM Page Size) controls how many bits to ignore when comparing the current bus-cycle address with the previous bus-cycle address to generate the NENE# signal. This feature allows for higher speeds when using static column or page-mode DRAMs and consecutive reads and writes access the row. The comparison ignores the low-order 12 + DPS bits. A value of zero is appropriate for one bank of 256K $\times$ *n* RAMs, 1 for 1M $\times$ *n* RAMS, etc.
- When BL (Bus Lock) is set, external bus accesses are locked. The LOCK# signal is asserted the next bus cycle whose internal bus request is generated after BL is set. It remains set on every subsequent bus cycle as long as BL remains set. The LOCK# signal is deasserted on the next bus cycle whose internal bus request is generated after BL is cleared. Traps immediately clear BL. The **lock** and **unlock** instructions control the BL bit.
- ITI (I-Cache, TLB Invalidate), when set in the value that is loaded into **dirbase**, causes the instruction cache and address-translation cache (TLB) to be flushed. The ITI bit does not remain set in **dirbase**. ITI always appears as zero when reading **dirbase**. The data cache must be flushed before invalidating the TLB.
- When CS8 (Code Size 8-Bit) is set, instruction cache misses are processed as 8-bit bus cycles. When this bit is clear, instruction cache misses are processed as 64-bit bus cycles. This bit can not be set by software; hardware sets this bit at initialization time. It can be cleared by software (one time only) to allow the system to execute out of 64-bit memory after bootstrapping from 8-bit EPROM. A nondelayed branch to code in 64-bit memory should directly follow the **st.c** (store control register) instruction that clears CS8, in order to make the transition from 8-bit to 64-bit memory occur at the correct time. The branch must be aligned on a 64-bit boundary.
- RB (Replacement Block) identifies the cache block to be replaced by cache replacement algorithms. The high-order bit of RB is ignored by the instruction and data caches. RB conditions the cache flush instruction **flush**, which is discussed in Section 8. Table 2.3 explains the values of RB.
- RC (Replacement Control) controls cache replacement algorithms. Table 2.4 explains the significance of the values of RC.
- DTB (Directory Table Base) contains the high-order 20 bits of the physical address of the page directory when address translation is enabled (i.e. ATE = 1). The low-order 12 bits of the address are zeros.

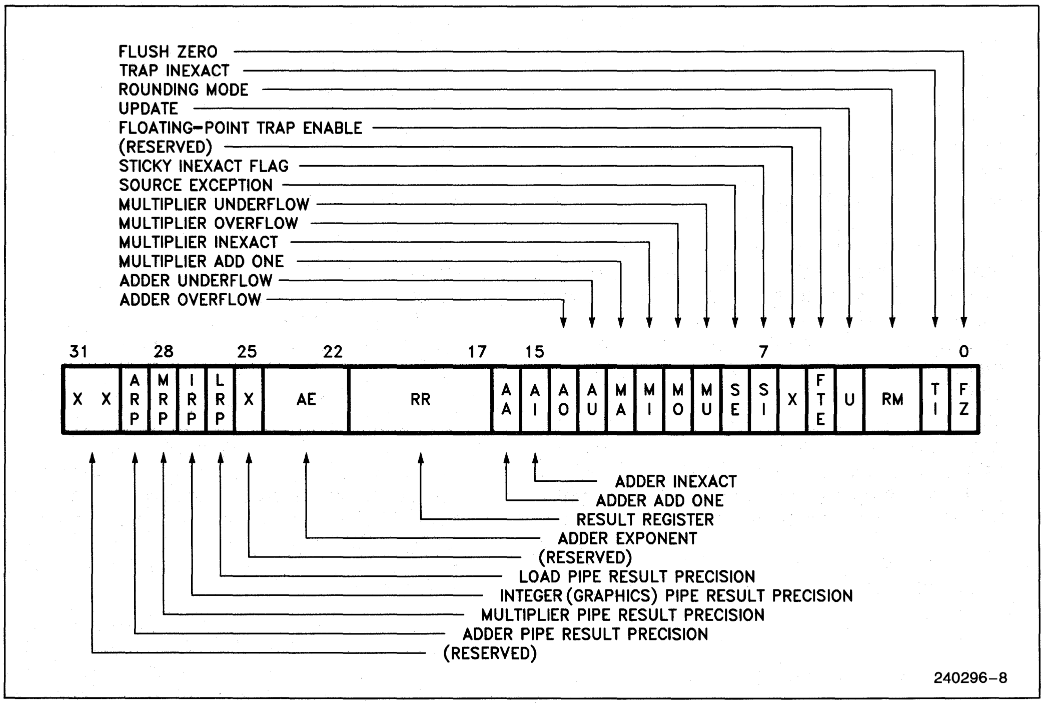


Figure 2.7. Floating-Point Status Register

Table 2.3. Values of RB

Value	Replace TLB Block	Replace Instruction and Data Cache Block
0 0	0	0
0 1	1	1
1 0	2	0
1 1	3	1

Table 2.4. Values of RC

Value	Meaning
00	Selects the normal replacement alogrithm where any block in the set may be replaced on cache misses in all caches.
01	Instruction, data, and TLB cache misses replace the block selected by RB. The instruction and data caches ignore the high-order bit of RB. This mode is used for instruction cache and TLB testing.
10	Data cache misses replace the block selected by the low-order bit of RB.
11	Disables data cache replacement.

2.2.7 FAULT INSTRUCTION REGISTER

When a trap occurs, this register contains the address of the trapping instruction (not necessarily the instruction that created the conditions that required the trap).

2.2.8 FLOATING-POINT STATUS REGISTER

The floating-point status register (**fsr**) contains the floating-point trap and rounding-mode status for the current process. Figure 2.6 shows its format.

- If FZ (Flush Zero) is clear and underflow occurs, a result-exception trap is generated. When FZ is set and underflow occurs, the result is set to zero, and no trap due to underflow occurs.
- If TI (Trap Inexact) is clear, inexact results do not cause a trap. If TI is set, inexact results cause a trap. The sticky inexact flag (SI) is set whenever an inexact result is produced, regardless of the setting of TI.
- RM (Rounding Mode) specifies one of the four rounding modes defined by the IEEE standard. Given a true result *b* that cannot be represented by the target data type, the 860 microprocessor determines the two representable numbers *a* and

Table 2.5. Values of RM

Value	Rounding Mode	Rounding Action
00	Round to nearest or even	Closer to *b* of *a* or *c*; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward $-\infty$)	*a*
10	Round up (toward $+\infty$	*c*
11	Chop (toward zero)	Smaller in magnitude of *a* or *c*.

c that most closely bracket *b* in value ($a < b < c$). The 860 microprocessor then rounds (changes) *b* to *a* or *c* according to the mode selected by RM as defined in Table 2.5. Rounding introduces an error in the result that is less than one least-significant bit.

- The U-bit (Update Bit), if set in the value that is loaded into **fsr** by a **st.c** instruction, enables updating of the result-status bits (AE, AA, AI, AO, AU, MA, MI, MO, and MU) in the first-stage of the floating-point adder and multiplier pipelines. If this bit is clear, the result-status bits are unaffected by a **st.c** instruction; **st.c** ignores the corresponding bits in the value that is being loaded. An **st.c** always updates **fsr** bits 21..17 and 8..0 directly. The U-bit does not remain set; it always appears as zero when read.
- The FTE (Floating-Point Trap Enable) bit, if clear, disables all floating-point traps (invalid input operand, overflow, underflow, and inexact result).
- SI (Sticky Inexact) is set when the last-stage result of either the multiplier or adder is inexact (i.e. when either AI or MI is set). SI is "sticky" in the sense that it remains set until reset by software. AI and MI, on the other hand, can by changed by the subsequent floating-point instruction.
- SE (Source Exception) is set when one of the source operands of a floating-point operation is invalid; it is cleared when all the input operands are valid. Invalid input operands include denormals, infinities, and all NaNs (both quiet and signaling).
- When read from the **fsr,** the result-status bits MA, MI, MO, and MU (Multiplier Add-One, Inexact, Overflow, and Underflow, respectively) describe the last-stage result of the multiplier.

  When read from the **fsr**, the result-status bits AA, AI, AO, AU, and AE (Adder Add-One, Inexact, Overflow, Underflow, and Exponent, respectively) describe the last-stage result of the adder. The high-order three bits of the 11-bit exponent of the adder result are stored in the AE field.

  After a floating-point operation in a given unit (adder or multiplier), the result-status bits of that unit are undefined until the point at which result exceptions are reported.

  When written to the **fsr** with the U-bit set, the result-status bits are placed into the first stage of the adder and multiplier pipelines. When the processor executes pipelined operations, it propagates the result-status bits of a particular unit (multiplier or adder) one stage for each pipelined floating-point operation for that unit. When they reach the last stage, they replace the normal result-status bits in the **fsr.** When the U-bit is not set, result-status bits in the word being writeen to the **fsr** are ignored.

  In a floating-point dual-operation instruction (e.g. add-and-multiply or subtract-and-multiply), both the multiplier and the adder may set exception bits. The result-status bits for a particular unit remain set until the next operation that uses that unit.
- RR (Result Register) specifies which floating-point register (**f0–f31**) was the destination register when a result-exception trap occurs due to a scalar operation.
- LRP (Load Pipe Result Precision), IRP (Vector-Integer Pipe Result Precision), MRP (Multiplier Pipe Result Precision), and ARP (Adder Pipe Result Precision) aid in restoring pipeline state after a trap or process switch. Each defines the precision of the last-stage result in the corresponding pipeline. One of these bits is set when the result in the last stage of the corresponding pipeline is double precision; it is cleared if the result is single precision. These bits cannot be changed by software.

2.2.9 KR, KI, T, AND MERGE REGISTERS

The KR, KI, and T registers are special-purpose registers used by the dual-operation floating-point instructions **pfam,/pfmam,** and **pfmsm**, which initiate both an adder (A-unit) operation and a multiplier (M-unit) operation. The KR, KI, and T registers can store values from one dual-operation instruction and supply them as inputs to subsequent dual-operation instructions. (Refer to Table 2.9.)

The MERGE register is used only by the vector-integer instructions. The purpose of the MERGE register is to accumulate (or merge) the results of multiple-addition operations that use as operands the color-intensity values from pixels or distance values from a Z-buffer. The accumulated results can then be stored in one 64-bit operation.

Two multiple-addition instructions and an OR instruction use the MERGE register. The addition instructions are designed to add interpolation values to each color-intensity field in an array of pixels or to each distance value in a Z-buffer.

Refer to the instruction descriptions in section 8 for more information about these registers.

2.3 Addressing

Memory is addressed in byte units with a paged virtual-address space of 2^{32} bytes. Data and instructions can be located anywhere in this address space. Address arithmetic is performed using 32-bit input values and produces 32-bit results. The low-order 32 bits of the result ae used in case of overflow.

Normally, multibyte data values are stored in memory in little endian format, i.e., with the least significant byte at the lowest memory address. As an option that may be dynamically selected by software in supervisor mode, the 860 microprocessor also offers big endian mode, in which the most significant byte of a data item is at the lowest address. Code accesses are always done with little endian addressing. Figure 2.8 shows the difference between the two storage modes. Big endian and little endian data areas should not be mixed within a 64-bit data word. Illustrations of data structures in this data sheet show data stored in little endian mode, i.e., the rightmost (low-order) byte is at the lowest memory address.

Alignment requirements are as follows (any violation results in a data-access trap):

- 128-bit values are aligned on 16-byte boundaries when referenced in memory (i.e. the four least significant address bits must be zero).
- 64-bit values are aligned on 8-byte boundaries when referenced in memory (i.e. the three least significant address bits must be zero).
- 32-bit values are aligned on 4-byte boundaries when referenced in memory (i.e. the two least significant address bits must be zero).
- 16-bit values are aligned on 2-byte boundaries when referenced in memory (i.e. the least significant address bit must be zero).

2.4 Virtual Addressing

When address translation is enabled, the 860 microprocessor maps instruction and data virtual addresses into physical addresses before referencing memory. This address transformation is compatible with that of the 386 microprocessor and implements the basic features needed for page-oriented virtual-memory systems and page-level protection.

The address translation is optional. Address translation is in effect only when the ATE bit of **dirbase** is set. This bit is typically set by the operating system during software initialization. The ATE bit must be set if the operating system is to implement page-oriented protection or page-oriented virtual memory.

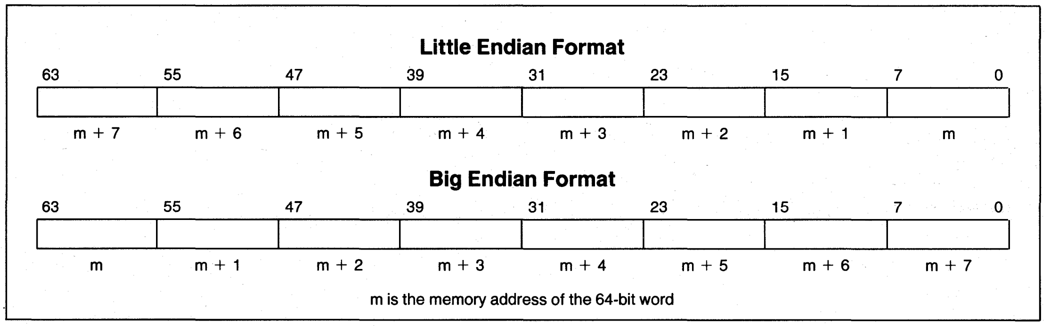

Little Endian Format
63 55 47 39 31 23 15 7 0
m + 7 m + 6 m + 5 m + 4 m + 3 m + 2 m + 1 m
Big Endian Format
63 55 47 39 31 23 15 7 0
m m + 1 m + 2 m + 3 m + 4 m + 5 m + 6 m + 7
m is the memory address of the 64-bit word

Figure 2.8. Little Big Endian Memory Format

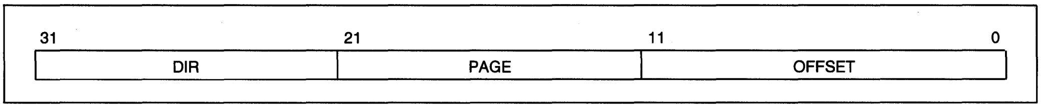


Figure 2.9. Format of a Virtual Address

Address translation is disabled when the processor is reset. It is enabled when a store to **dirbase** sets the ATE bit. It is disabled again when a store clears the ATE bit.

2.4.1 PAGE FRAME

A **page frame** is a 4-Kbyte unit of contiguous addresses of physical main memory. Page frames begin on 4-Kbyte boundaries and are fixed in size. A **page** is the collection of data that occupies a page frame when that data is present in main memory or occupies some location in secondary storage when there is not sufficient space in main memory.

2.4.2 VIRTUAL ADDRESS

A virtual address refers indirectly to a physical address by specifying a page table, a page within that table, and an offset within that page. Figure 2.9 shows the format of a virtual address.

Figure 2.9 shows how the 860 microprocessor converts the DIR, PAGE, and OFFSET fields of a virtual address into the physical address by consulting two levels of page tables. The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

2.4.3 PAGE TABLES

A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kbytes of memory or at most 1K 32-bit entries.

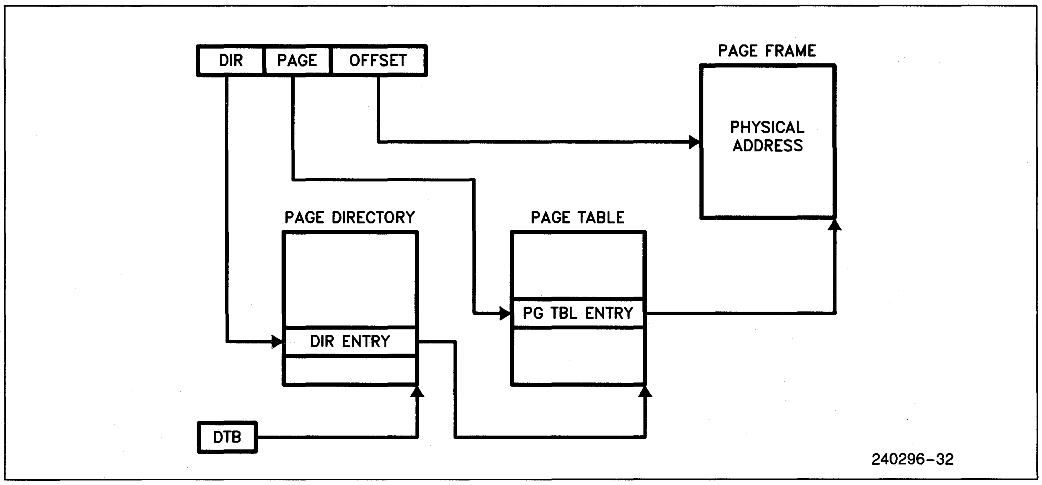


Figure 2.10. Address Translation

Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages (2^{20}). Because each page contains 4 Kbytes (2^{12} bytes), the tables of one page directory can span the entire physical address space of the 860 microprocessor ($2^{20} \times 2^{12} = 2^{32}$).

The physical address of the current page directory is stored in DTB field of the **dirbase** register. Memory management software has the option of using one page directory for all processes, one page directory for each process, or some combination of the two.

2.4.4 PAGE-TABLE ENTRIES

Page-table entries (PTEs) in either level of page tables have the same format. Figure 2.11 illustrates this format.

2.4.4.1 Page Frame Address

The page frame address specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand.

2.4.4.2 Present Bit

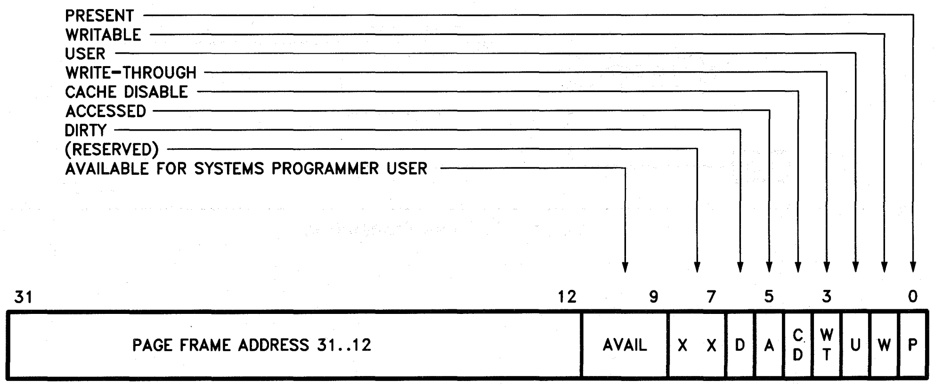
The P (present) bit indicates whether a page table entry can be used in address translation. P = 1 indicates that the entry can be used. When P = 0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware. If P = 0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals either a data-access fault or an instruction-access fault. In software systems that support paged virtual memory, the trap handler can bring the required page into physical memory.

Note that there is no P bit for the page directory itself. The page directory may be not-present while the associated process is suspended, but the operating system must ensure that the page directory indicated by the **dirbase** image associated with the process is present in physical memory before the process is dispatched.

2.4.4.3 Writable and User Bits

The W (writable) and U (user) bits are used for page-level protection, which the 860 microprocessor performs at the same time as address translation. The concept of privilege for pages is implemented by assigning each page to one of two levels:

1. Supervisor level (U = 0)—for the operating system and other systems software and related data.
2. User level (U = 1)—for applications procedures and data.



NOTE:
X indicates Intel reserved. Do not use.

Figure 2.11. Format of a Page Table Entry

The U bit of the **psr** indicates whether the 860 microprocessor is executing at user or supervisor level. The 860 microprocessor maintains the U bit of **psr** as follows:

- The 860 microprocessor clears the **psr** U bit to indicate supervisor level when a trap occurs (including when the **trap** instruction causes the trap). The prior value fo U is copied into PU.
- The 860 microprocessor copies the **psr** PU bit into the U bit when an indirect branch is executed and one of the trap bits is set. If PU was one, the 860 microprocessor enters user level.

With the U bit of **psr** and the W and U bits of the page table entries, the 860 microprocessor implements the following protection rules:

- When at user level, a read or write of a supervisor-level pages causes a trap.
- When at user level, a write to a page whose W bit is not set causes a trap.
- When at user level, **st.c** to certain control registers is ignored.

When the 860 microprocessor is executing at supervisor level, all pages are addressable, but, when it is executing at user level, only pages that belong to the user-level are addressable.

When the 860 microprocessor is executing at supervisor level, all pages are readable. Whether a page is writable depends upon the write-protection mode controlled by WP of **epsr**:

WP = 0	All pages are writable.
WP = 1	A write to page whose W bit is not set causes a trap.

When the 860 microprocessor is executing at user level, only pages that belong to user level and are marked writable are actually writable; pages that belong to supervisor level are neither readable nor writable from user level.

2.4.4.4 Write-Through Bit

The 860 microprocessor does not implement a write-through caching policy for the on-chip instruction and data caches; however, the WT (write-through) bit in the second-level page-table entry does determine internal caching policy. If WT is set in a PTE, on-chip caching of data from the corresponding page is inhibited. If WT is clear, the normal write-back policy is applied to data from the page in the on-chip caches. The WT bit of page directory entries is not referenced by the processor, but is **reserved**.

The WT bit is independent of the CD bit; therefore, data may be placed in a second-level coherent cache, but kept out of the on-chip caches.

2.4.4.5 Cache Disable Bit

If the CD (cache disable) bit in the second-level page-table entry is set, data from the associated page is not placed in external instruction or data caches. Clearing CD permits the external cache hardware to place data from the associated page into external caches. The CD bit of page directory entries is not referenced by the processor, but is **reserved.**

2.4.4.6 Accessed and Dirty Bits

The A (accessed) and D (dirty) bits provide data about page usage in both levels of the page tables.

The 860 microprocessor sets the corresponding accessed bits in both levels of page tables before a read or write operation to a page. The processor tests the dirty bit in the second-level page table before a write to an address covered by that page table entry, and, under certain conditions, causes traps. The trap handler than has the opportunity to maintain appropriate values in the dirty bits. The dirty bit in directory entries is not tested by the 860 microprocessor. The precise algorithm for using these bits is specified in Subsection 2.4.5.

An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available. The D and A bits in the PTE (page-table entry) are normally initialized to zero by the operating system. The processor sets the A bit when a page is accessed either by a read or write operation. When a data- or instruction-access fault occurs, the trap handler sets the D bit if an allowable write is being performed, then re-executes the instruction.

The operating system is responsible for coordinating its updates to the accessed and dirty bits with updates by the CPU and by other processors that may share the page tables. The 860 microprocessor automatically asserts the LOCK# signal while testing and setting the A bit.

2.4.4.7 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page directory entry may differ from those of its page table entry. The 860 microprocessor computes the effective protection attributes for a page by examining the protection attributes in both the directory and the page table. Table 2.6 shows the effective protection provided by the possible combinations of protection attributes.

2.4.5 ADDRESS TRANSLATION ALGORITHM

The algorithm below defines the translation of each virtual address to a physical address. Let DIR, PAGE, and OFFSET be the fields of the virtual address; let PFA1 and PFA2 be the page frame address fields of the first and second level page tables respectively; DTB is the page directly table base address stored in the **dirbase** register.

1. Assert LOCK#.
2. Read the PTE (page table entry) at the physical address formed by DTB:DIR:00.
3. If P in the PTE is zero, generate a data- or instruction-access fault.
4. If W in the PTE is zero, the operation is a write, and either the U-bit of the PSR is set or WP = 1, generate a data- or instruction-access fault.
5. If the U-bit in the PTE is zero and the U-bit in the **psr** is set, generate a data- or instruction-access fault.
6. If A in the PTE is zero, set A.
7. Locate the PTE at the physical address formed by PFA1:PAGE:00.
8. Perform the P, A, W, and U checks as in steps 3 through 6 with the second-level PTE.
9. If D in the PTE is clear and the operation is a write, generate a data- or instruction-access fault.
10. Form the physical address as PFA2:OFFSET.
11. Deassert LOCK#.

2.4.6 ADDRESS TRANSLATION FAULTS

The address translation fault is one instance of the data-access fault. The instruction causing the fault can be re-executed upon returning from the trap handler.

2.4.7 PAGE TRANSLATION CACHE

For greatest efficiency in address translation, the 860 microprocessor stores the most recently used page-table data in an on-chip cache called the TLB (translation lookaside buffer). Only if the necessary paging information is not in the cache must both levels of page tables be referenced.

2.5 Caching and Cache Flushing

The 860 microprocessor has the ability to cache instruction, data, and address-translation information in on-chip caches. Caching may use virtual-address tags. The effects of mapping two different virtual addresses in the same address space to the same physical address are undefined.

Table 2.6. Combining Directory and Page Protection

Page Directory Entry		Page Table Entry		Combined Protection			
				WP = 0		WP = 1	
U-bit	W-bit	U-bit	W-bit	U	W	U	W
0	0	0	0	0	x	0	0
0	0	0	1	0	x	0	0
0	0	1	0	0	x	0	0
0	0	1	1	0	x	0	0
0	1	0	0	0	x	0	0
0	1	0	1	0	x	0	1
0	1	1	0	0	x	0	0
0	1	1	1	0	x	0	1
1	0	0	0	0	x	0	0
1	0	0	1	0	x	0	0
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0
1	1	0	0	0	x	0	0
1	1	0	1	0	x	0	1
1	1	1	0	1	0	1	0
1	1	1	1	1	1	1	1

NOTES:

U = 0—Supervisor	W = 0—Read only
U = 1—User	W = 1—Read and write

x indicates that, when the combined U attribute is supervisor and WP = 0, the W attribute is not checked.

Instruction, data, and address-translation caching on the 860 microprocessor are not transparent. Writes do not immediately update memory, the TLB, nor the instruction cache. Writes to memory by other bus devices do not update the caches. Under certain circumstances, such as I/O references, self-modifying code, page-table updates, or shared data in a multiprocessing system, it is necessary to bypass or to flush the caches. 860 microprocessor provides the following methods for doing this:

- **Bypassing Instruction and Data Caches.** If deasserted during cache-miss processing, the KEN# pin disables instruction and data caching of the referenced data. If the CD bit from the associated second-level PTE is set, caching of data and instructions is disabled. The value of the CD bit is output on the PTB pin for use by external caches.
- **Flushing Instruction and Address-Translation Caches.** Storing to the **dirbase** register with the ITI bit set invalidates the contents of the instruction and address-translation caches. This bit should be set when a page table or a page containing code is modified or when changing the DTB field of **dirbase**. Note that in order to make the instruction or address-translation caches consistent with the data cache, the data cache must be flushed *before* invalidating the other caches.

  NOTE:

  The mapping of the page(s) containing the currently executing instruction, the next six instructions, and any data referenced by these instructions should not be different in the new page tables when the DTB is changed.
- **Flushing the Data Cache.** The data cache is flushed by a software routine using the **flush** instruction. The data cache must be flushed prior to flushing the instruction or address-translation cache (as controlled by the ITI bit of **dirbase**) or enabling or disabling address translation (via the ATE bit). While the cache is being flushed, no interrupt or trap routines should be executed that load sharable data into the cache.

2.6 Instruction Set

Table 2.7 shows the complete set of instructions grouped by function within processing unit. Refer to Section 8 for an algorithmic definition of each instruction.

The architecture of the 860 microprocessor uses parallelism to increase the rate at which operations may be introduced into the unit. Parallelism in the 860 microprocessor is **not** transparent; rather, programmers have complete control over parallelism and therefore can achieve maximum performance for a variety of computational problems.

2.6.1 PIPELINED AND SCALAR OPERATIONS

One type of parallelism used within the floating-point unit is "pipelining". The pipelined architecture treats each operation as a series of more primitive operations (called "stages") that can be executed in parallel. Consider just the floating-point adder unit as an example. Let **A** represent the operation of the adder. Let the stages be represented by $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$. The stages are designed such that $\mathbf{A}_{i+1}$ for one adder instruction can execute in parallel with $\mathbf{A}_i$ for the next adder instruction. Furthermore, each $\mathbf{A}_i$ can be executed in just one clock. The pipelining within the multiplier and vector-integer units can be described similarly, except that the number of stages may be different.

Figure 2.7 illustrates three-stage pipelining as found in the floating-point adder (also in the floating-point multiplier when single-precision input operands are employed). The columns of the figure represent the three stages of the pipeline. Each stage holds intermediate results and also (when introduced into first stage by software) holds status information pertaining to those results. The figure assumes that the instruction stream consists of a series of consecutive floating-point instructions, all of one type (i.e. all adder instructions or all single-precision multiplier instructions). The instructions are represented as **i**, **i+1**, etc. The rows of the figure represent the states of the unit at successive clock cycles. Each time a pipelined operation is performed, the result of the last stage of the pipeline is stored in the destination register *rdest*, the pipeline is advanced one stage, and the input operands *src1* and *src2* are transferred to the first stage of the pipeline.

In the 860 microprocessor, the number of pipeline stages ranges from one to three. A pipelined operation with a three-stage pipeline stores the result of the third prior operation. A pipelined operation with a two-stage pipeline stores the result of the second prior operation. A pipelined operation with a one-stage pipeline stores the result of the prior operation.

There are four floating-point pipelines: one for the multiplier, one for the adder, one for the vector-integer unit, and one for floating-point loads. The adder pipeline has three stages. The number of stages in the multiplier pipeline depends on the precision of the source operands in the pipeline; it may have two or three stages. The vector-integer unit has one stage for all precisions. The load pipeline has three stages for all precisions.

Changing the FZ (flush zero), RM (rounding mode), or RR (result register) bits of **fsr** while there are results in either the multiplier or adder pipeline produces effects that are not defined.

2.6.1.1 Scalar Mode

In addition to the pipelined execution mode, the 860 microprocessor also can execute floating-point instructions in "scalar" mode. Most floating-point instructions have both pipelined and scalar variants, distinguished by a bit in the instruction encoding. In scalar mode, the floating-point unit does not start a new operation until the previous floating-point operation is completed. The scalar operation passes through all stages of its pipeline before a new operation is introduced, and the result is stored automatically. Scalar mode is used when the next operation depends on results from the previous few floating-point operations (or when the compiler or programmer does not want to deal with pipelining).

2.6.1.2 Pipelining Status Information

Result status information in the **fsr** consists of the AA, AI, AO, AU, and AE bits, in the case of the adder, and the MA, MI, MO, and MU bits, in the case of the multiplier. This information arrives at the **fsr** via the pipeline in one of two ways:

Table 2.7. Instruction Set

Core Unit	
Mnemonic	**Description**
Load and Store Instructions	
ld.x	Load integer
st.x	Store integer
fld.y	F-P load
pfld.z	Pipelined F-P load
fst.y	F-P store
pst.d	Pixel store
Register to Register Moves	
ixfr	Transfer integer to F-P register
fxfr	Transfer F-P to integer register
Integer Arithmetic Instructions	
addu	Add unsigned
adds	Add signed
subu	Subtract unsigned
subs	Subtract signed
Shift Instructions	
shl	Shift left
shr	Shift right
shra	Shift right arithmetic
shrd	Shift right double
Logical Instructions	
and	Logical AND
andh	Logical AND high
andnot	Logical AND NOT
andnoth	Logical AND NOT high
or	Logical OR
orh	Logical OR high
xor	Logical exclusive OR
xorh	Logical exclusive OR high
Control-Transfer Instructions	
trap	Software trap
intovr	Software trap on integer overflow
br	Branch direct
bri	Branch indirect
bc	Branch on CC
bc.t	Branch on CC taken
bnc	Branch on not CC
bnc.t	Branch on not CC taken
bte	Branch if equal
btne	Branch if not equal
bla	Branch on LCC and add
call	Subroutine call
calli	Indirect subroutine call
System Control Instructions	
flush	Cache flush
ld.c	Load from control register
st.c	Store to control register
lock	Begin interlocked sequence
unlock	End interlocked sequence

Floating-Point Unit	
Mnemonic	**Description**
F-P Multiplier Instruction	
fmul.p	F-P multiply
pfmul.p	Pipelined F-P multiply
pfmul3.dd	3-Stage pipelined F-P multiply
fmlow.p	F-P multiply low
frcp.p	F-P reciprocal
frsqr.p	F-P reciprocal square root
F-P Adder Instructions	
fadd.p	F-P add
pfadd.p	Pipelined F-P add
fsub.p	F-P subtract
pfsub.p	Pipelined F-P subtract
pfgt.p	Pipelined F-P greater-than compare
pfeq.p	Pipelined F-P equal compare
fix.p	F-P to integer conversion
pfix.p	Pipelined F-P to integer conversion
ftrunc.p	F-P to integer truncation
pftrunc.p	Pipelined F-P to integer truncation
Dual-Operation Instructions	
pfam.p	Pipelined F-P add and multiply
pfsm.p	Pipelined F-P subtract and multiply
pfmam	Pipelined F-P multiply with add
pfmsm	Pipelined F-P multiply with subtract
Long Integer Instructions	
fisub.z	Long-integer subtract
pfisub.z	Pipelined long-integer subtract
fiadd.z	Long-integer add
pfiadd.z	Pipelined long-integer add
Graphics Instructions	
fzchks	16-bit Z-buffer check
pfzchks	Pipelined 16-bit Z-buffer check
fzchkl	32-bit Z-buffer check
pfzchkl	Pipelined 32-bit Z-buffer check
faddp	Add with pixel merge
pfaddp	Pipelined add with pixel merge
faddz	Add with Z merge
pfaddz	Pipelined add with Z merge
form	OR with MERGE register
pform	Pipelined OR with MERGE register

Assembler Pseudo-Operations	
Mnemonic	**Description**
mov	Integer register-register move
fmov.q	F-P reg-reg move
pfmov.q	Pipelined F-P reg-reg move
nop	Core no-operation
fnop	F-P no-operation
pfle.p	Pipelined F-P less-than or equal

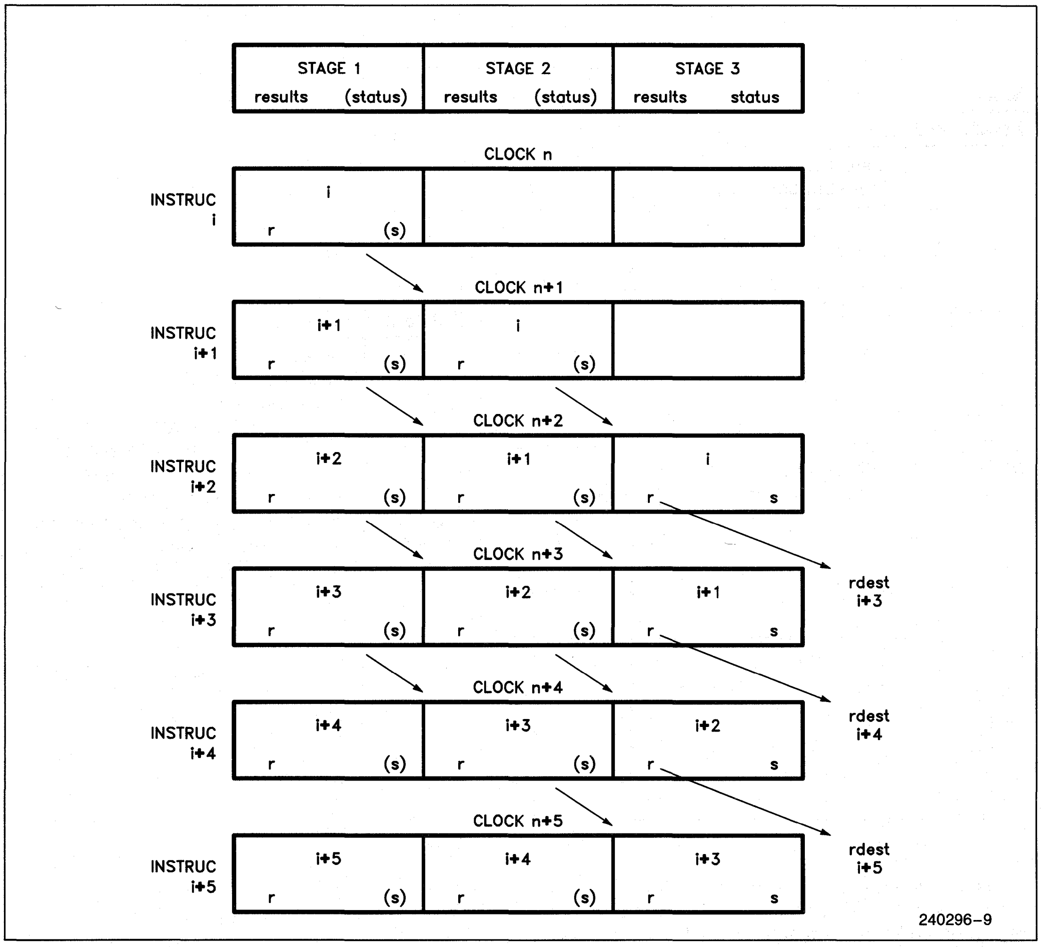


Figure 2.12. Pipelined Instruction Execution

1. It is calculated by the last stage of the pipeline. This is the normal case.
2. It is propagated from the first stage of the pipeline. This method is used when restoring the state of the pipeline after a preemption. When a store instruction updates the **fsr** and the value of the U bit in the word being written into the **fsr** is set, the store updates the result status bits in the first stage of both the adder and multiplier pipelines. When software changes the result-status bits of the first stage of a particular unit (multiplier or adder), the updated result-status bits are propagated one stage for each pipelined floating-point operation for that unit. In this case, each stage of the adder and multiplier pipelines holds its own copy of the relevant bits of the **fsr**. When they reach the last stage, they override the normal result-status bits computed from the last-stage result.

At the next floating-point instruction (or at certain core instructions), after the result reaches the last stage, the 860 microprocessor traps if any of the status bits of the **fsr** indicate exceptions. Note that the instruction that creates the exceptional condition is not the instruction at which the trap occurs.

2.6.1.3 Precision in the Pipelines

In pipelined mode, when a floating-point operation is initiated, the result of an earlier pipelined floating-point operation is returned. The result precision of the current instruction applies to the operation being initiated. The precision of the value stored in *rdest* is that which was specified by the instruction that initiated that operation.

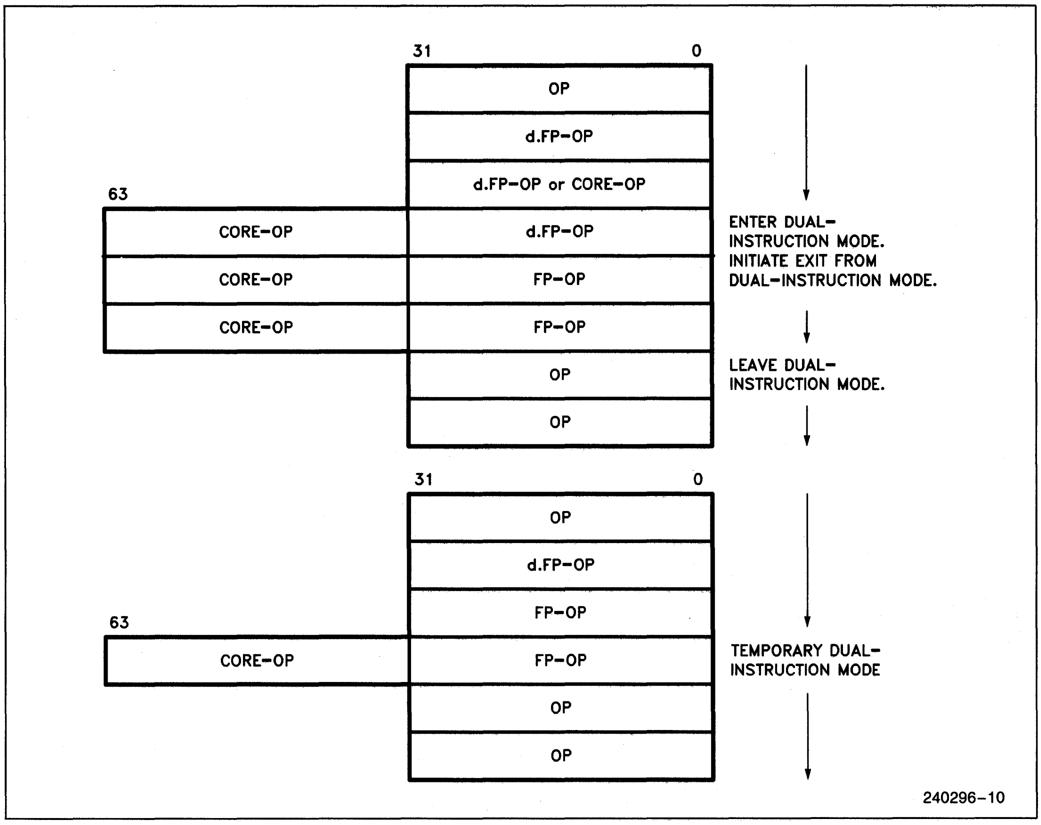

31
0
OP
d.FP-OP
d.FP-OP or CORE-OP
63
CORE-OP
d.FP-OP
CORE-OP
FP-OP
CORE-OP
FP-OP
OP
OP
ENTER DUAL-INSTRUCTION MODE.
INITIATE EXIT FROM DUAL-INSTRUCTION MODE.
LEAVE DUAL-INSTRUCTION MODE.
31
0
OP
d.FP-OP
FP-OP
63
CORE-OP
FP-OP
OP
OP
TEMPORARY DUAL-INSTRUCTION MODE
240296-10

Figure 2.13. Dual-Instruction Mode Transitions

If *rdest* is the same as *src1* or *src2*, the value being stored in *rdest* is used as the input operand. In this case, the precision of *rdest* must be the same as the source precision.

The multiplier pipeline has two stages when the source operand is double-precision and three stages when the precision of the source operand is single. This means that a pipelined multiplier operation stores the result of the second previous multiplier operation for double-precision inputs and third previous for single-precision inputs (except when changing precisions).

2.6.1.4 Transition between Scalar and Pipelined Operations

When a scalar operation is executed, it passes through all stages of the pipeline; therefore, any unstored results in the affected pipeline are lost. To avoid losing information, the last pipelined operations before a scalar operation should be dummy pipelined operations that unload unstored results from the affected pipeline.

After a scalar operation, the values of all pipeline stages of the affected unit (except the last) are undefined. No spurious result-exception traps result when the undefined values are subsequently stored by pipelined operations; however, the values should not be referenced as source oeprands.

For best performance a scalar operation should not immediately precede a pipelined operation whose *rdest* is nonzero.

2.6.2 DUAL-INSTRUCTION MODE

Another form of parallelism results from the fact that the 860 microprocessor can execute both a floating-point and a core instruction simultaneously. Such parallel execution is called [dual-instruction mode]. When executing in dual-instruction mode, the instruction sequence consists of 64-bit aligned instructions with a floating-point instruction in the lower 32 bits and a core instruction in the upper 32 bits. Table 2.6 identifies which instructions are executed by the core unit and which by the floating-point unit.

Programmers specify dual-instruction mode either by including in the mnemonic of a floating-point instruction a **d.** prefix or by using the Assembler directives **.dual** .... **.enddual**. Both of the specifications cause the D-bit of floating-point instructions to be set. If the 860 microprocessor is executing in single-instruction mode and encounters a floating-point instruction with the D-bit set, one more 32-bit instruction is executed before dual-mode execution begins. If the 860 microprocessor is executing in dual-instruction mode and a floating-point instruction is encountered with a clear D-bit, then one more pair of instructions is executed before resuming single-instruction mode. Figure 2.13 illustrates two variations of this sequence of events: one for extended sequences of dual-instructions and one for a single instruction pair.

When a 64-bit dual-instruction pair sequentially follows a delayed branch instruction in dual-instruction mode, both 32-bit instructions are executed.

2.6.3 DUAL-OPERATION INSTRUCTIONS

Special dual-operation floating-point instructions (add-and-multiply, subtract-and-multiply) use both the multiplier and adder units within the floating-point unit in parallel to efficiently execute such common tasks as evaluating systems of linear equations, performing the Fast Fourier Transform (FFT), and performing graphics transformations.

The instructions **pfam** *src1*, *src2*, *rdest* (add and multiply), **pfsm** *src1*, *src2*, *rdest* (subtract and multiply), **pfmam** *scr1*, *src2*, *rdest* (multiply and add), and **pfmsm** *src1*, *src2*, *rdest* (multiply and subtract) initiate both an adder operation and a multiplier operation. Six operands are required, but the instruction format specifies only three operands; therefore, there are special provisions for specifying the operands. These special provisions consist of:

- Three special registers (KR, KI, and T), that can store values from one dual-operation instruction and supply them as inputs to subsequent dual-operation instructions.
  1. The constant registers KR and KI can store the value of *src1* and subsequently supply that value to the multiplier pipeline in place of *src1*.
  2. The transfer register T can store the last-stage result of the multiplier pipeline and subsequently supply that value to the adder pipeline in place of *src1*.
- A four-bit data-path control field in the opcode (DPC) that specifies the operands and loading of the special registers.
  1. Operand-1 of the multiplier can be KR, KI, or *src1*.
  2. Operand-2 of the multiplier can be *src2* or the last-stage result of the adder pipeline.
  3. Operand-1 of the adder can be *src1*, the T-register, or the last-stage result of the adder pipeline.
  4. Operand-2 of the adder can be *src2*, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.

Figure 2.14 shows all the possible data paths surrounding the adder and multiplier. A DPC field in these instructions select different data paths. Section 8 shows the various encodings of the DPC field.

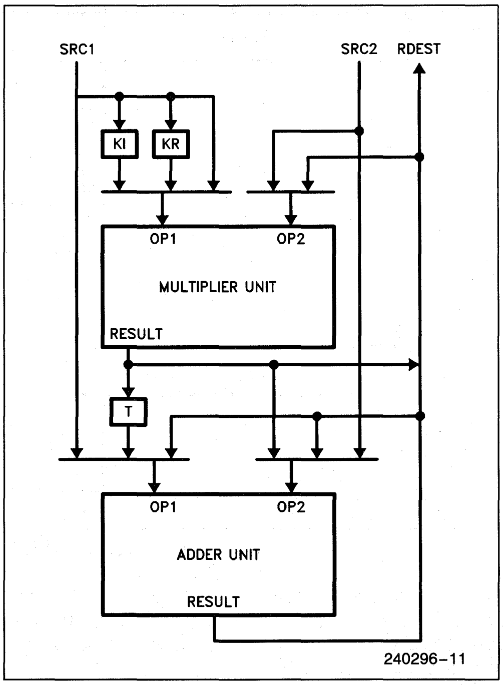


Figure 2.14. Dual-Operation Data Paths

Note that the mnemonics **pfam.p**, **pfsm.p**, **pfmam.p**, and **pfmsm.p** are never used as such in the assembly language; these mnemonics are used here to designate classes of related instructions. Each value of DPC has a unique mnemonic associated with it.

2.7 Addressing Modes

Data access is limited to load and store instructions. Memory addresses are computed from two fields of load and store instructions: *src1* and *src2*.

1. *src1* either contains the identifier of a 32-bit integer register or contains an immediate 16-bit address offset.
2. *src2* always specifies a register.

Table 2.8. Types of Traps

Type	Indication		Caused by	
	PSR	FSR	Condition	Instruction
Instruction Fault	IT		Software traps Missing **unlock**	**trap, intovr** Any
Floating Point Fault	FT	SE AO, MO AU, MU AI, MI	Floating-point source exception Floating-point result exception overflow underflow inexact result	Any M- or A-unit except **fmlow** Any M- or A-unit except **fmlow, pfgt,** and **pfeq**. Reported on any F-P instruction plus **pst, fst,** and sometimes **fld, pfld, ixfr**
Instruction Access Fault	IAT		Address translation exception during instruction fetch	Any
Data Access Fault	DAT*		Load/store address translation exception Misaligned operand address Operand address matches **db** register	Any load/store Any load/store Any load/store
Interrupt	IN		External interrupt	
Reset	No trap bits set		Hardware RESET signal	

*These cases can be distinguished by examining the operand addresses.

Because either *src1* or *src2* may be null (zero), a variety of useful addressing modes result:

offset + *register*	Useful for accessing fields within a record, where *register* points to the beginning of the record. Useful for accessing items in a stack frame, where *register* is **r3**, the register used for pointing to the beginning of the stack frame.
register + *register*	Useful for two-dimensional arrays or for array access within the stack frame.
register	Useful as the end result of any arbitrary address calculation.
offset	Absolute address into the first 64K of the logical address space.

In addition, the floating-point load and store instructions may select autoincrement addressing. In this mode *src2* is replaced by the sum of *src1* and *src2* after performing the load or store. This mode makes stepping through arrays more efficient, because it eliminates one address-calculation instruction.

2.8 Interrupts and Traps

Traps are caused by exceptional conditions detected in programs or by external interrupts. Traps cause interruption of normal program flow to execute a special program known as a trap handler. Traps are divided into the types shown in Table 2.8.

2.8.1 TRAP HANDLER INVOCATION

This section applies to traps other than reset. When a trap occurs, execution of the current instruction is aborted. The instruction is restartable. The processor takes the following steps while transferring control to the trap handler:

1. Copies U (user mode) of the **psr** into PU (previous U).
2. Copies IM (interrupt mode) into PIM (previous IM).
3. Sets U to zero (supervisor mode).
4. Sets IM to zero (interrupts disabled).
5. If the processor is in dual instruction mode, it sets DIM; otherwise it clears DIM.
6. If the processor is in single-instruction mode and the next instruction will be executed in dual-instruction mode or if the processor is in dual-instruction mode and the next instruction will be executed in single-instruction mode, DS is set; otherwise, it is cleared.
7. The appropriate trap type bits in **psr** are set (IT, IN, IAT, DAT, FT). Several bits may be set if the corresponding trap conditions occur simultaneously.
8. An address is placed in the fault instruction register (**fir**) to help locate the trapped instruction. In

single-instruction mode, the address in **fir** is the address of the trapped instruction itself. In dual-instruction mode, the address in **fir** is that of the floating-point half of the dual instruction. If an instruction or data access fault occurred, the associated core instruction is the high-order half of the dual instruction (**fir** + 4). In dual-instruction mode, when a data access fault occurs in the absence of other trap conditions, the floating-point half of the dual instruction will already have been executed (except in the case of the **fxfr** instruction).

The processor begins executing the trap handler by transferring execution to address 0xFFFFFF00. The trap handler begins execution in single-instruction mode. The trap handler must examine the trap-type bits in **psr** (IT, IN, IAT, DAT, FT) to determine the cause or causes of the trap.

2.8.2 INSTRUCTION FAULT

This fault is caused by any of the following conditions. In all cases the processor sets the IT bit before entering the trap handler.

- By the **trap** instruction.
- By the **intovr** instruction. The trap occurs only if OF in **epsr** is set when **intovr** is executed. The trap handler should clear OF before returning.
- By the lack of an **unlock** instruction within 32 instructions of a **lock**. In this case IL is also set. When the trap handler finds IL set, it should scan backwards for the **lock** instruction and restart at that point. The absence of a **lock** instruction within 32 instructions of the trap indicates a programming error.

2.8.3 FLOATING-POINT FAULT

The floating-point fault occurs on floating-point instructions **pst**, **fst**, and sometimes **fld**, **pfld**, **ixfr**. The floating-point faults of the 860 microprocessor support the floating-point exceptions defined by the IEEE standard as well as some other useful classes of exceptions. The 860 microprocessor divides these into two classes: source exceptions and result exceptions. The numerics library supplied by Intel provides the IEEE standard default handling for all these exceptions.

2.8.3.1 Source Exception Faults

All exceptional operands, including infinities, denormalized numbers and NaNs, cause a floating-point fault and set SE in the **fsr**. Source exceptions are reported on the instruction that initiates the operation. For pipelined operations, the pipeline is not advanced.

The SE value is undefined for faults on **fld**, **pfld**, **fst**, **pst**, and **ixfr** instructions when in single-instruction mode or when in dual-instruction mode and the companion instruction is not a multiplier or adder operation.

2.8.3.2 Result Exception Faults

The class of result exceptions includes any of the following conditions:

- **Overflow.** The absolute value of the rounded true result would exceed the largest positive finite number in the destination format.
- **Underflow** (when FZ is clear). The absolute value of the rounded true result would be smaller than the smallest positive finite number in the destination format.
- **Inexact result** (when TI is set). The result is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (generally acceptable) accuracy has been lost.

The point at which a result exception is reported depends upon whether pipelined operations are being used:

- **Scalar (nonpipelined) operations.** Result exceptions are reported on the next floating-point, **fst.x**, or **pst.x** (and sometimes **fld**, **pfld**, **ixfr**) instruction after the scalar operation. When a trap occurs, the last-stage of the affected unit contains the result of the scalar operation.
- **Pipelined operations.** Result exceptions are reported when the result is in the last stage and the next floating-point (and sometimes **fld**, **pfld**, **ixfr**) instruction is executed. When a trap occurs, the pipeline is not advanced, and the last-stage results (that caused the trap) remain unchanged.

When no trap occurs (either because FTE is clear or because no exception occurred), the pipeline is advanced normally by the new floating-point operation. The result-status bits of the affected unit are undefined until the point that result exceptions are reported. At this point, the last-stage result-status bits (bits 29..22 and 16..9 of the **fsr**) reflect the values in the last stages of both the adder and multiplier. For example, if the last-stage result in the multiplier has overflowed and a pipelined floating-point **pfadd** is started, a trap occurs and MO is set.

For scalar operations, the RR bits of **fsr** specify the register in which the result was stored. RR is updated when the scalar instruction is initiated. The trap, however, occurs on a subsequent instruction. Programmers must prevent intervening stores to **fsr** from modifying the RR bits. Prevention may take one of the following forms:

- Before any store to **fsr** when a result exception may be pending, execute a dummy floating-point operation to trigger the result-exception trap.
- Always read from **fsr** before storing to it, and mask updates so that the RR bits are not changed.

For pipelined operations, RR is cleared; the result is in the pipeline of the appropriate unit.

In either case, the result has the same fraction as the true result and has an exponent which is the low-order bits of the true result. The trap handler can inspect the result, compute the result appropriate for that instruction (a NaN or an infinity, for example), and store the correct result. The result is either stored in the register specified by RR (if nonzero) or in the last stage of the pipeline (if RR = 0). The trap handler must clear the result status for the last stage, then reexecute the trapping instruction.

Result exceptions may be reported for both the adder and multiplier units at the same time. In this case, the trap handler should fix up the last stage of both pipelines.

2.8.4 INSTRUCTION ACCESS FAULT

This trap results from a page-not-present exception during instruction fetch. If a supervisor-level page is fetched in user mode, an exception may or may not occur.

2.8.5 DATA ACCESS FAULT

This trap results from an abnormal condition detected during data operand fetch or store. Such an exception can be due only to one of the following causes:

- An attempt is being made to write to a page whose D-bit is clear.
- A memory operand is misaligned (is not located at an address that is a multiple of the length of the data).
- The address stored in the debug register is equal to one of the addresses spanned by the operand.
- The operand is in a not-present page.
- An attempt is being made from user level to write to a read-only page or to access a supervisor-level page.

2.8.6 INTERRUPT TRAP

An interrupt is an event that is signaled from an external source. If the processor is executing with interrupts enabled (IM set in the **psr**), the processor sets the interrupt bit IN in the **psr**, and generates an interrupt trap. Vectored interrupts are implemented by interrupt controllers and software.

2.8.7 RESET TRAP

When the 860 microprocessor is reset, execution begins in single-instruction mode at address 0xFFFFFF00. This is the same address as for other traps. The reset trap can be distinguished from other traps by the fact that no trap bits are set. The instruction cache is flushed. The bits DPS, BL, and ATE in **dirbase** are cleared. CS8 is initialized by the value at the INT pin at the end of reset. The bits U, IM, BR, and BW in **psr** are cleared. All other bits of **psr** and all other register contents are **undefined.**

The software must ensure that the data cache is flushed and control registers are properly initialized before performing operations that depend on the values of the cache or registers.

Reset code must initialize the floating-point pipeline state to zero with floating-point traps disabled to ensure that no spurious floating-point traps are generated.

After a RESET the 860 microprocessor starts execution at supervisor level (U=0). Before branching to the first user-level instruction, the RESET trap handler or subsequent initialization code has to set PU and a trap bit so that an indirect branch instruction will copy PU to U, thereby changing to user level.

2.9 Debugging

The 860 microprocessor supports debugging with both data and instruction breakpoints. The features of the 860 architecture that support debugging include:

- **db** (data breakpoint register) which permits specification of a data addresses that the 860 microprocessor will monitor.
- BR (break read) and BW (break write) bits of the **psr**, which enable trapping of either reads or writes (respectively) to the address in **db**.
- DAT (data access trap) bit of the **psr**, which allows the trap handler to determine when a data breakpoint was the cause of the trap.
- **trap** instruction that can be used to set breakpoints in code. Any number of code breakpoints can be set. The values of the *src1* and *src2* fields help identify which breakpoint has occurred.
- IT (instruction trap) bit of the **psr**, which allows the trap handler to determine when a **trap** instruction was the cause of the trap.

3.0 HARDWARE INTERFACE

In the following description of hardware interface, the # symbol at the end of a signal name indicates that the active or asserted state occurs when the signal is at a low voltage. When no # is present after the signal name, the signal is asserted when at the high voltage level.

3.1 Signal Description

Table 3.1 identifies functional groupings of the pins, lists every pin by its identifier, gives a brief description of its function, and lists some of its characteristics. All output pins are tristate, except HLDA and BREQ. All inputs are synchronous, except HOLD and INT.

3.1.1 CLOCK (CLK)

The CLK input determines execution rate and timing of the 860 microprocessor. Timing of other signals is specified relative to the rising edge of this signal. The 860 microprocessor can utilize a clock rate of 33.3 MHz. The internal operating frequency is the same as the external clock. This signal is TTL compatible.

3.1.2 SYSTEM RESET (RESET)

Asserting RESET for at least 16 CLK periods causes initialization of the 860 microprocessor. Refer to section 3.2 "Initialization" for more details related to RESET.

3.1.3 BUS HOLD (HOLD) AND BUS HOLD ACKNOWLEDGE (HLDA)

These pins are used for 860 microprocessor bus arbitration. At some time after the HOLD signal is asserted, the 860 microprocessor releases control of the local bus and puts all bus interface outputs (except BREQ and HLDA) in floating state, then asserts HLDA—all during the same clock period. It maintains this state until HOLD is deasserted. Instruction execution stops only if required instructions or data cannot be read from the on-chip instruction and data caches.

The time required to acknowledge a hold request is one clock plus the number of clocks needed to finish any outstanding bus cycles. HOLD is recognized even while RESET is asserted.

When leaving a bus hold, the 860 microprocessor deactivates HLDA and, in the same clock period, initiates a pending bus cycle, if any.

Hold is an asynchronous input.

Table 3.1. Pin Summary

Pin Name	Function	Active State	Input/ Output
Execution Control Pins			
CLK	CLocK		I
RESET	System reset	High	I
HOLD	Bus hold	High	I
HLDA	Bus hold acknowledge	High	O
BREQ	Bus request	High	O
INT/CS8	Interrupt, code-size	High	I
Bus Interface Pins			
A31–A3	Address bus	High	O
BE7#–BE0#	Byte Enables	Low	O
D63–D0	Data bus	High	I/O
LOCK#	Bus lock	Low	O
W/R#	Write/Read bus cycle	Hi/Low	O
NENE#	NExt NEar	Low	O
NA#	Next Address request	Low	I
READY#	Transfer Acknowledge	Low	I
ADS#	ADdress Status	Low	O
Cache Interface Pins			
KEN#	Cache ENable	Low	I
PTB	Page Table Bit	High	O
Testability Pins			
SHI	Boundary Scan Shift Input	High	I
BSCN	Boundary Scan Enable	High	I
SCAN	Shift Scan Path	High	I
Intel-Reserved Configuration Pins			
CC1–CC0	Configuration	High	I
Power and Ground Pins			
V_{CC}	System power		
V_{SS}	System ground		

A # after a pin name indicates that the signal is active when at the low voltage level.

3.1.4 BUS REQUEST (BREQ)

This signal is asserted when the 860 microprocessor has a pending memory request, even when HLDA is asserted. This allows an external bus arbiter to implement an "on demand only" policy for granting the bus to the 860 microprocessor.

3.1.5 INTERRUPT/CODE-SIZE (INT/CS8)

This input allows interruption of the current instruction stream. If interrupts are enabled (IM set in **psr**) when INT is asserted, the 860 microprocessor fetches the next instruction from address 0xFFFFFF00. To assure that an interrupt is recognized, INT should remain asserted until the software acknowledges the interrupt (by writing, for example, to a memory-mapped port of an interrupt controller). The maximum time between the assertion of INT and execution of the first instruction of the trap handler is 10 clocks, plus the time for eight nonpipelined read cycles (four TLB misses), plus the time for eight nonpipelined writes (updates to the A bit), plus the time for three sets of four pipelined read cycles and two sets of four pipelined writes (instruction and data cache misses and write-back cycles to update memory).

If INT is asserted during the clock before the falling edge of RESET, the eight-bit code-size mode is selected. For more about this mode, refer to section 3.2 "Initialization".

INT is an asynchronous input.

3.1.6 ADDRESS PINS (A31–A3) AND BYTE ENABLES (BE7#–BE0#)

The 29-bit address bus (A31–A3) identifies addresses to a 64-bit location. Separate byte-enable signals (BE7#–BE0#) identify which bytes should be accessed within the 64-bit location. Cache reads should return 64 bits without regard for the byte-enable signals.

Instruction fetches (W/R# is low) are distinguished from data accesses by the unique combinations of BE7#–BE0# defined in Table 3.2. For an eight-bit code fetch in eight-bit code-size (CS8) mode, BE2#–BE0# are redefined to be A2–A0 of the address. In this case BE7#–BE3# form the code shown in Table 3.2 that identifies an instruction fetch.

3.1.7 DATA PINS (D63–D0)

The bus interface has 64 bidirectional data pins (D63–D0) to transfer data in eight- to 64-bit quantities. Pins D7–D0 transfer the least significant byte; pins D63–D56 transfer the most significant byte.

In write bus cycles, the point at which data is driven onto the bus depends on the type of the preceding cycle. If there was no preceding cycle (i.e. the bus was idle), data is driven with the address. If the preceding cycle was a write, data is driven as soon as READY# is returned from the previous cycle. If the preceding cycle was a read, data is driven one clock after READY# is returned from the previous cycle, thereby allowing time for the bus to be turned around.

3.1.8 BUS LOCK (LOCK#)

This signal is used to provide atomic (indivisible) read-modify-write sequences in multiprocessor systems. Once the external bus arbiter has accepted a memory access for a locked bus cycle from the 860 microprocessor, it should not accept locked cycles (or *any* cycles, depending on software convention) from other bus masters until LOCK# is deasserted.

The 860 microprocessor coordinates the external LOCK# signal with the software-controlled BL bit of the **dirbase** register. Programmers do not have to be concerned about the fact that bus activity is not always synchronous with instruction execution. LOCK# is asserted with ADS# for the first bus cycle that results from an instruction executed after the BL bit is set. Even if the BL bit is changed between the time that an instruction generates an internal bus request and the time that the cycle appears on the bus, the 860 microprocessor still asserts LOCK# for that bus cycle. LOCK# is deasserted with ADS# for the next bus cycle that results from an instruction executed after the BL bit is cleared.

The 860 microprocessor also asserts LOCK# during TLB miss processing for updates of the accessed bit in page-table entries. The maximum time that LOCK# can be asserted in this case is five clocks plus the time required by software to perform a read-modify-write sequence.

The 860 microprocessor does not acknowledge bus hold requests while LOCK# is asserted.

3.1.9 WRITE/READ BUS CYCLE (W/R#)

This pin specifies whether a bus cycle is a read (LOW) or write (HIGH) cycle.

3.1.10 NEXT NEAR (NENE#)

This signal allows higher-speed reads and writes in the case of consecutive reads and writes that access static column or page-mode DRAMs. The 860 microprocessor asserts NENE# when the current address is in the same DRAM page as the previous bus cycle. The 860 microprocessor determines the DRAM page size by inspecting the DPS field in the **dirbase** register. The page size can range from 2^9 to 2^{16} 64-bit words, supporting DRAM sizes from 256K × 1, 256K × 4, and up. NENE# is never asserted on the next bus cycle after HLDA is deasserted.

3.1.11 NEXT ADDRESS REQUEST (NA#)

NA# makes address pipelining possible. The system asserts NA# to indicate that it is ready to ac-

Table 3.2. Identifying Instruction Fetches

Code Fetch	A2	BE7#	BE6#	BE5#	BE4#	BE3#	BE2#	BE1#	BE0#
Normal (Non-CS8)	0	1	1	1	1	1	0	1	0
Normal (Non-CS8)	1	1	0	1	0	1	1	1	1
CS8 Mode	x	1	0	1	0	x	Low-order address bits		

cept the next address from the 860 microprocessor. NA# may be asserted before the current cycle ends. (If the system does not implement pipelining, NA# does not have to be activated.) The 860 microprocessor samples NA# every clock, starting one clock after the prior activation of ADS#. When NA# is active, the 860 microprocessor is free to drive address and bus-cycle definition for the next pending bus cycle. The 860 microprocessor remembers that NA# was asserted when no internal request is pending; therefore, NA# can be deactivated after the next rising edge of the CLK signal. Up to three bus cycles can be outstanding simultaneously on the processor's bus.

3.1.12 TRANSFER ACKNOWLEDGE (READY#)

The system asserts the READY# signal during read cycles when valid data is on the data pins and during a write cycles when the system has accepted data from the data pins. READY# is sampled one clock after prior ADS# or prior READY# in case of pipelining.

3.1.13 ADDRESS STATUS (ADS#)

The 860 microprocessor asserts ADS# during the first clock of each bus cycle to identify the clock period during which it begins to assert outputs on the address bus. This signal is not held active during a pipelined bus cycle. This allows two-level pipelining, for a maximum of three outstanding cycles.

3.1.14 CACHE ENABLE (KEN#)

The 860 microprocessor samples KEN# to determine whether the data being read for the current cache-miss cycle is to be cached. This pin is internally NORed with the PTB pin to control cache ability on a page by page basis (refer to Table 3.3).

If the address in one that is permitted to be in the cache, KEN# must be continuously asserted during the sampling period starting from the clock after ADS# is asserted, through the clock NA# or READY# is asserted. The entire 64-bit of the data bus will be used for the read, regardless of the state of the byte-enable pins. Three additional 64-bit bus cycles will generate to fill the rest of the 32-byte cache block. KEN# must continue to be asserted for each of these cycles as well.

If KEN# is found deasserted at any time during the above-described sampling period, the data being read will not cached and two scenarios can occur: 1) if the cycle is due to data-cache miss, no subsequent cache-fill cycles will be generated; 2) if the cycle is due to an instruction-cache miss, additional cycle(s) will be generated until the address reaches a 32-byte boundary.

3.1.15 PAGE TABLE BIT (PTB)

Depending on the setting of the PBM (page-table bit mode) bit of the **epsr**, the PTB reflects the value of either the CD (cache disable) bit or the WT (write through) bit of the page-table entry used for the current cycle. This pin is internally NORed with the KEN# pin to control cacheability on a page by page basis. Table 3.3 shows the relationship between PTB and KEN#. When paging is disabled, PTB remains inactive.

Table 3.3. Cacheability based on KEN# and PTB

PTB	KEN#	Meaning
0	0	Cacheable access
0	1	Noncacheable access
1	0	Noncacheable page
1	1	Noncacheable page

3.1.16 BOUNDARY SCAN SHIFT INPUT (SHI)

This pin is used with the testability features. Refer to section 3.4.

3.1.17 BOUNDARY SCAN ENABLE (BSCN)

This pin is used with the testability features. Refer to section 3.4.

3.1.18 SHIFT SCAN PATH (SCAN)

This pin is used with the testability features. Refer to section 3.3.

3.1.19 CONFIGURATION (CC1–CC0)

These two pins are reserved by Intel. Strap both pins LOW.

3.1.20 SYSTEM POWER (V_{CC}) AND GROUND (V_{SS})

The 860 microprocessor has 48 pins for power and ground. All pins must be connected to the appropriate low-inductance power and ground signals in the system.

3.2 Initialization

Initialization of the 860 microprocessor is caused by assertion of the RESET signal for at least 16 clocks. Table 3.4 shows the status of output pins during the time that RESET is asserted. Note that HOLD requests are honored during RESET and that the status of output pins depends on whether a HOLD request is being acknowledged.

Table 3.4. Output Pin Status during Reset

Pin Name	Pin Value	
	HOLD Not Acknowledged	HOLD Acknowledged
ADS#, LOCK#	HIGH	Tri-State OFF
W/R#, PTB	LOW	Tri-State OFF
BREQ	LOW	LOW
HLDA	LOW	HIGH
D63-D0	Tri-State OFF	Tri-State OFF
A31-A3, BE7#-BE0#, NENE#	Undefined	Tri-State OFF

After a reset, the 860 microprocessor begins executing at address 0xFFFFFF00. The program-visible state of the 860 microprocessor after reset is detailed in section 2.

Eight-bit code-size mode is selected when INT is asserted during the clock before the falling edge of RESET. While in eight-bit code-size mode, instruction cache misses are byte reads (transferred on D7-D0 of the data bus) instead of eight-byte reads. This allows the 860 microprocessor to be bootstrapped from an eight-bit EPROM. For these code reads, byte enables BE2#-BE0# are redefined to be the low order three bits of the address, so that a complete byte address is available. These reads update the instruction cache if KEN# is asserted (refer to section 3.1.1.4) and are not pipelined even if NA# is asserted. While in this mode, instructions must reside in an eight-bit wide memory, while data must reside in a separate 64-bit wide memory. After the code has been loaded into 64-bit memory, initialization code can initiate 64-bit code fetches by clearing the CS8 bit of the dirbase register (refer to section 2). Once eight-bit code-size mode is disabled by software, it cannot be reenabled except by resetting the 860 microprocessor.

3.3 Testability

The 860 microprocessor has a *boundary scan mode* that may be used in component- or board-level testing to test the signal traces leading to and from the 860 microprocessor. Boundary scan mode provides a simple serial interface that makes it possible to test all signal traces with only a few probes. Probes need be connected only to CLK, BSCN, SCAN, SHI, and BREQ.

The pins BSCN and SCAN control the boundary scan mode (refer to Table 3.5). When BSCN is asserted, the 860 microprocessor enters boundary scan mode on the next rising clock edge. Boundary scan mode can be activated even while RESET is active. When BSCN is deasserted while in boundary scan mode, the 860 microprocessor leaves boundary scan mode on the next rising clock edge. After leaving boundary scan mode, the internal state is undefined; therefore, RESET should be asserted.

Table 3.5. Test Mode Selection

BSCN	SCAN	Testability Mode
LO	LO	No testability mode selected
LO	HI	(Reserved for Intel)
HI	LO	Boundary scan mode, normal
HI	HI	Boundary scan mode, shift SHI as input; BREQ as output

For testing purposes, each signal pin has associated with it an internal latch. Table 3.6 indentifies these latches by name and classifies them as input, output, or control. The input and output latches carry the name of the corresponding pins.

Table 3.6. Test Mode Latches

Input Latch	Output Latch	Associated Control Latch
SHI		
BSCN		
SCAN		
RESET		
D0-D63	D0-D63	DATAt
CC1-CC0		
	A31-A3	ADDRt
	NENE#	NENEt
	PTB#	PTBt
	W/R#	W/Rt
	ADS#	ADSt
	HLDA	
	LOCK#	LOCKt
READY#		
KEN#		
NA#		
INT/CS8		
HOLD		
	BE7#-BE0#	BEt
	BREQ	

Within boundary scan mode the 860 microprocessor operates in one of two submodes: normal mode or shift mode, depending on the value of the SCAN input. A typical test sequence is . . .

1. Enter shift mode to assign values to the latches that correspond with the pins.
2. Enter normal mode. In normal mode the 860 microprocessor transfers the latched values to the output pins and latches the values that are being driven onto the input pins.
3. Reenter shift mode to read the new values of the input pins.

3.3.1 NORMAL MODE

When SCAN is deasserted, the normal mode is selected. For each input pin (RESET, HOLD, INT/CS8, NA#, READY#, KEN#, SHI, BSCN, SCAN, CC1, and CC0), the corresponding latch is loaded with the value that is being driven onto the pin.

The tristate output pins (A31–A3, BE7#–BE0#, W/R#, NENE#, ADS#, LOCK#, and PTB) are enabled by the control latches ADDRt (for A31–A3), BEt, W/Rt, NENEt, ADSt, LOCKt, and PTBt. If a control latch is set, the corresponding output latches drive their output pins; otherwise the pins are not driven.

The I/O pins (D63–D0) are enabled by the control latch DATAt, which is similar to the other control latches. In addition, when DATAt is not set, the data pins are treated as input pins and their values are latched.

3.3.2 SHIFT MODE

When SCAN is asserted, the shift mode is selected. In shift mode, the pins are organized into a *boundary scan chain*. The scan chain is configured as a shift register that is shifted on the rising edge of CLK. The SHI pin is connected to the input of one end of the boundary scan chain. The value of the most significant bit of the scan chain is output on the BREQ pin. To avoid glitches while the values are being shifted along the chain, all tristate outputs are disabled. The order of the pins within the chain is shown in Figure 3.1.

A tester causes entry into this mode for one of two purposes:

1. To assign values to output latches to be driven onto output pins upon subsequent entry into normal mode.
2. To read the values of input pins previously latched in normal mode.

4.0 BUS OPERATION

A bus cycle begins when ADS# is activated and ends when READY# is sampled active. READY# is sampled one clock after assertion of ADS# and thereafter until it becomes active. New cycles can start as often as every other clock until three cycles are outstanding. A bus cycle is considered outstanding as long as READY# has not been asserted to terminate that cycle. After READY# becomes active, it is not sampled again for the following (outstanding) cycle until the second clock after the one during which it became active. READY# is assumed to be inactive when it is not sampled.

With regard to how a bus cycle is generated by the 860 microprocessor, there are two types of cycles: pipelined and nonpipelined. Both types of cycles can be either read or write cycles. A pipelined cycle is one that starts while one or two other bus cycles are outstanding. A nonpipelined cycle is one that starts when no other bus cycles are outstanding.

4.1 Pipelining

A **m-n** read or write cycle is a cycle with a total cycle time of **m** clocks and a cycle-to-cycle time of **n** clocks (**m** ≥ **n**). Total cycle time extends from the clock in which ADS# is activated to the clock in which READY# becomes active; whereas, cycle-to-cycle time extends from the time that READY# is sampled active for the previous cycle to the time that it is sampled active again for the current cycle. When **m** = **n**, a nonpipelined cycle is implied; **m** > **n** implies a pipelined cycle.

→ 1 SHI → 2 BSCN → 3 SCAN → 4 RESET → 5 DATAt → 6 D0 → ... → 69 D63 →

70 CC1 → 71 CC0 → 72 A31 → ... → 100 A3 → 101 ADDRt → 102 NENEt → 103 NENE# → 104 PTBt →

105 PTB# → 106 W/Rt → 107 W/R# → 108 ADSt → 109 ADS# → 110 HLDA → 111 LOCKt → 112 LOCK# → 113 READY# →

114 KEN# → 115 NA# → 116 INT/CS8 → 117 HOLD → 118 BEt → 119 BE7# → ... → 126 BE0# → 127 BREQ →

Figure 3.1. Order of Boundary Scan Chain

Pipelining may occur for the next bus cycle any time the current bus cycle requires more than two clock periods to finish (**m** $>$ **2**). The next cycle can be initiated when NA# is sampled active, even if the current cycle has not terminated. In this case, pipelining occurs. NA# is recognized only in the clock when ADS# has become inactive.

To allow high transfer rates in large memory systems, two-level pipelining is supported (i.e there may be up to three cycles in progress at one time). Pipelining enables a new word of data to be transferred every two clocks, even though the total cycle time may be up to six clocks.

4.2 Bus State Machine

The operation of the bus is described in terms of a bus state machine using a state transition diagram. Figure 4.1 illustrates the 860 microprocessor bus state machine. A bus cycle is composed of two or more states. Each bus state lasts for one CLK period.

The 860 microprocessor supports up to two levels of address pipelining. Once it has started the first bus cycle, it can generate up to two more cycles as long as READY# remains inactive. To start a new bus cycle while other cycles are still outstanding, NA# must be active for at least one clock cycle starting with the clock after the previous ADS#. NA# is latched internally.

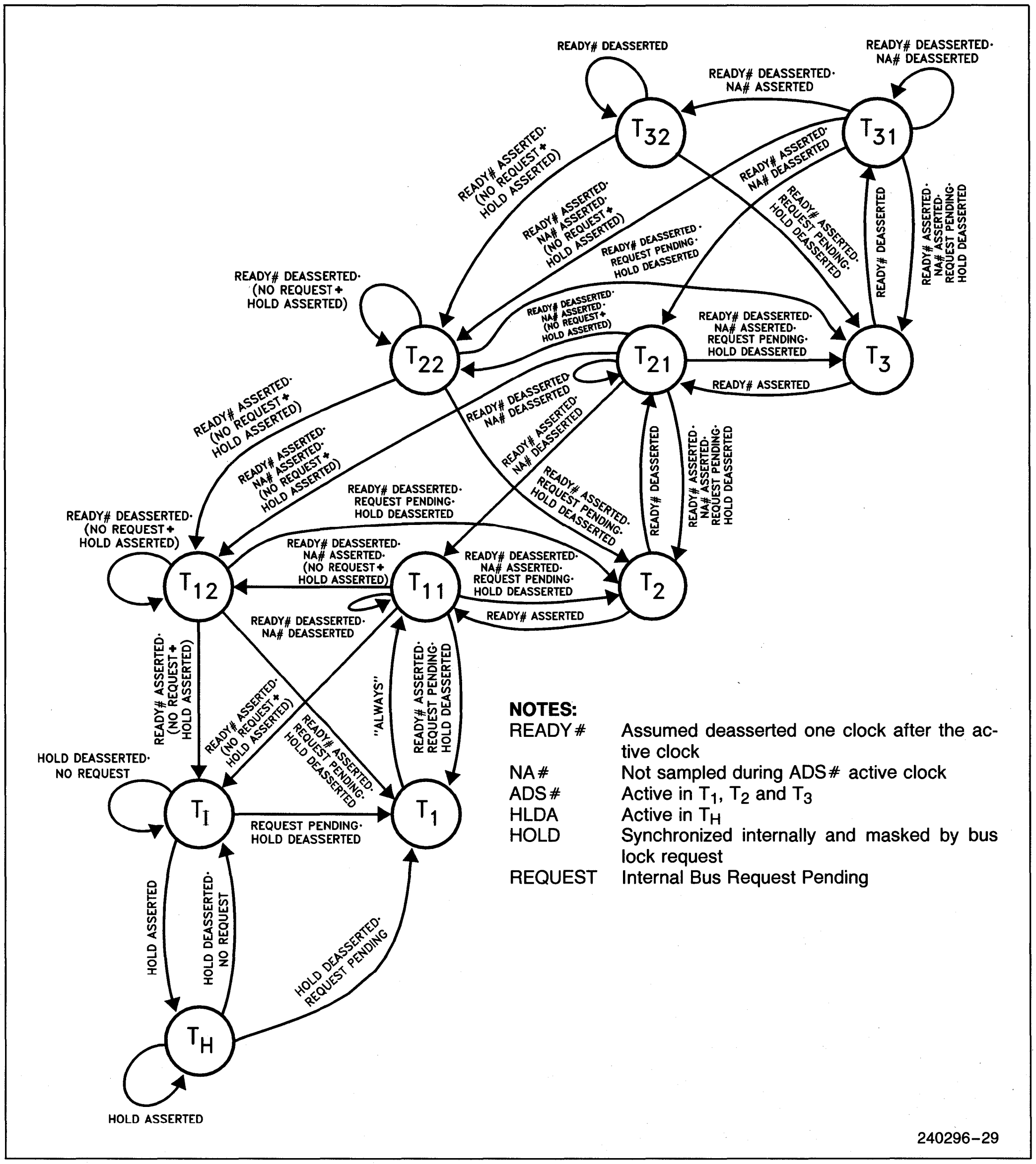
States T_j and T_{jk}, for **j** $= \{1,2,3\}$ and **k** $= \{1,2\}$, are used to describe the state of the 860 microprocessor Bus State Machine. Index **j** indicates the number of outstanding bus cycles while index **k** distinguishes the intermediate states for the **j**-th outstanding cycle. Therefore there can be up to three outstanding cycles, and there are two possible intermediate states for each level of pipelining. T_{j1} is the next state after T_j, as long as **j** cycles are outstanding. T_{j2} is entered when NA# is active but the 860 microprocessor is not ready to start a new cycle.

Five conditions have to be met to start a new cycle while one or more cycles are already pending:

1. READY# inactive
2. NA# having been active
3. An internal request pending
4. HOLD not active (or HOLD active, but not being serviced because LOCK# is active)
5. Fewer than three cycles outstanding

Upon hardware RESET, the bus control logic enters the idle state T_I and awaits an internal request for a bus cycle. If a bus cycle is requested while there is no hold request from the system, a bus cycle begins, advancing to state T_1. On the next cycle, the state machine automatically advances to state T_{11}. If READY# is active in state T_{11}, the bus control logic returns either to T_I, if no new cycle is started, or to T_1, if a new cycle request is pending internally. In fact, if an internal bus request is pending each time READY# is active, the state machine continues to cycle between T_{11} and T_1.

However, if READY# is not active but the next address request is pending (as indicated by an active NA#), the state machine advances either to state T_2 (if an internal bus request is pending, signifying that two bus cycles are now outstanding), or to state T_{12} (if no bus internal request is pending, signifying NA# has been found active). Transitions from state T_{12} are similar to those from T_{11}.


READY# DEASSERTED
READY# DEASSERTED· NA# DEASSERTED
READY# DEASSERTED· NA# ASSERTED
T32
T31
READY# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# ASSERTED· NA# DEASSERTED
READY# ASSERTED· NA# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# DEASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# DEASSERTED
READY# ASSERTED· NA# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# DEASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# DEASSERTED· NA# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
T22
T21
T3
READY# DEASSERTED· NA# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# ASSERTED
READY# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# ASSERTED· NA# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# DEASSERTED· NA# DEASSERTED
READY# ASSERTED· NA# DEASSERTED
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# DEASSERTED
READY# ASSERTED· NA# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# DEASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# DEASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# DEASSERTED· NA# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
T12
T11
T2
READY# DEASSERTED· NA# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# ASSERTED
READY# DEASSERTED· NA# DEASSERTED
READY# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# ASSERTED· (NO REQUEST+ HOLD ASSERTED)
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
"ALWAYS"
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
HOLD DEASSERTED· NO REQUEST
TI
T1
REQUEST PENDING· HOLD DEASSERTED
HOLD ASSERTED
HOLD DEASSERTED· NO REQUEST
HOLD DEASSERTED· REQUEST PENDING
TH
HOLD ASSERTED
240296-29

NOTES:

READY#	Assumed deasserted one clock after the active clock
NA#	Not sampled during ADS# active clock
ADS#	Active in T_1, T_2 and T_3
HLDA	Active in T_H
HOLD	Synchronized internally and masked by bus lock request
REQUEST	Internal Bus Request Pending

Figure 4.1. Bus State Machine

If two bus cycles are already outstanding (as indicated by T_{2k} for **k** = {1,2}) and NA# is latched active but READY# is not active, one more bus request causes entry into state T_3. Transitions from this state are similar to those from T_2.

In general, if there is an internal bus request each time both READY# and NA# are active, the state machine continues to oscillate between T_{j1} and T_j, for **j** = {2,3}.

When NA# is sampled active while there is a pending bus request, ADS# is activated in the next clock period (provided no more than two cycles are already outstanding).

Internal pending bus requests start new bus cycles only if no HOLD request has been recognized. T_H is entered from the idle state T_I only. HLDA is active in this state. There is a one clock delay to synchronize the HOLD input when the signal meets the respective minimum setup and hold time requirements. The state machine uses the synchronized HOLD to move from state to state.

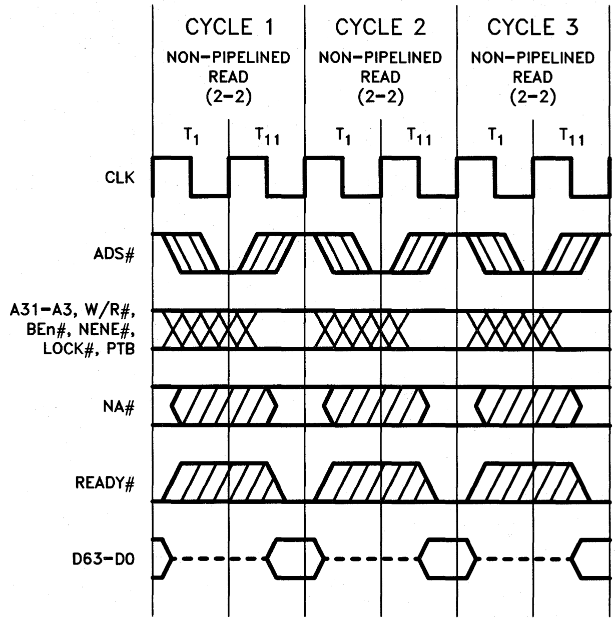
4.3 Bus Cycles

Figures 4.2 through 4.10 illustrate combinations of bus cycles.

4.3.1 NONPIPELINED READ CYCLES

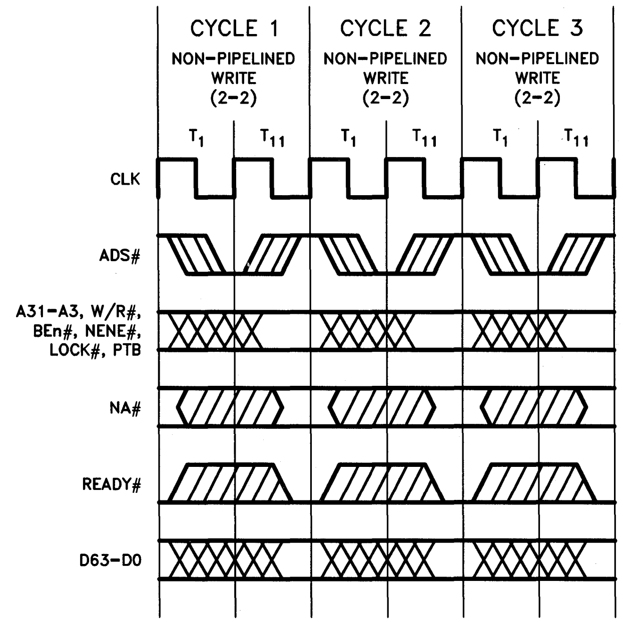
A read cycle begins with the clock in which ADS# is asserted. The 860 microprocessor begins driving the address during this clock. It samples READY# for active state every clock after the first clock. A minimum of two clocks is required per cycle. Data is latched when READY# is found active when sampled at the end of a clock period. Figure 4.2 illustrates nonpipelined read cycles with zero wait states.

Normally, all 64 bits of the data bus are latched; however, in the case of noncacheable bus cycles, the byte enables BE7#–BE0# determine which bytes are used. In CS8 mode, only the low-order eight bits are latched.



240296-13

Figure 4.2. Fastest Read Cycles


CYCLE 1
NON-PIPELINED WRITE (2-2)
CYCLE 2
NON-PIPELINED WRITE (2-2)
CYCLE 3
NON-PIPELINED WRITE (2-2)
T1
T11
CLK
ADS#
A31-A3, W/R#, BEn#, NENE#, LOCK#, PTB
NA#
READY#
D63-D0

240296-14

Figure 4.3. Fastest Write Cycles

4.3.2 NONPIPELINED WRITE CYCLES

The ADS# and READY# activity for write cycles follows the same logic as that for read cycles, as Figure 4.3 illustrates for back-to-back, nonpipelined write cycles with zero wait-states. The byte enables BE7#-BE0# indicate which bytes on the data bus are valid.

The fastest write cycle takes only two clocks to complete. However, when a read cycle immediately precedes a write cycle, the write cycle must contain a wait state, as illustrated in Figure 4.4. Because the device being read might still be driving the data bus during the first clock of the write cycle, there is a potential for bus contention. To help avoid such contention, the 860 microprocessor does not drive the data bus until the second clock of the write cycle. The wait state is required to provide the additional time necessary to terminate the write cycle. In other read-write combinations, the 860 microprocessor does not require a wait state.

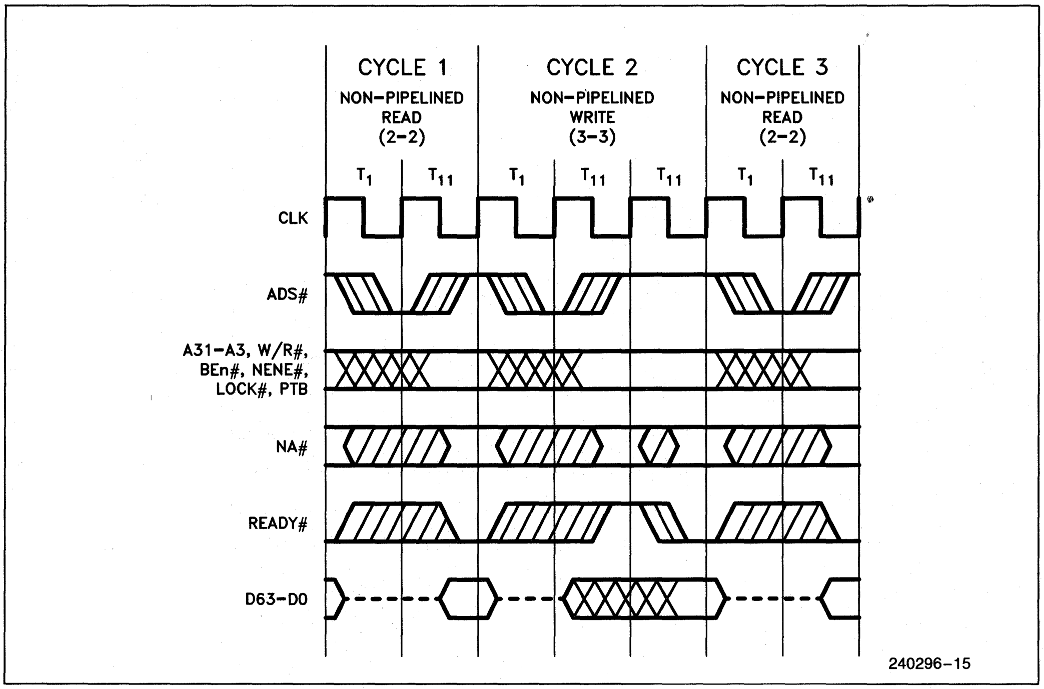


Figure 4.4. Fastest Read/Write Cycles

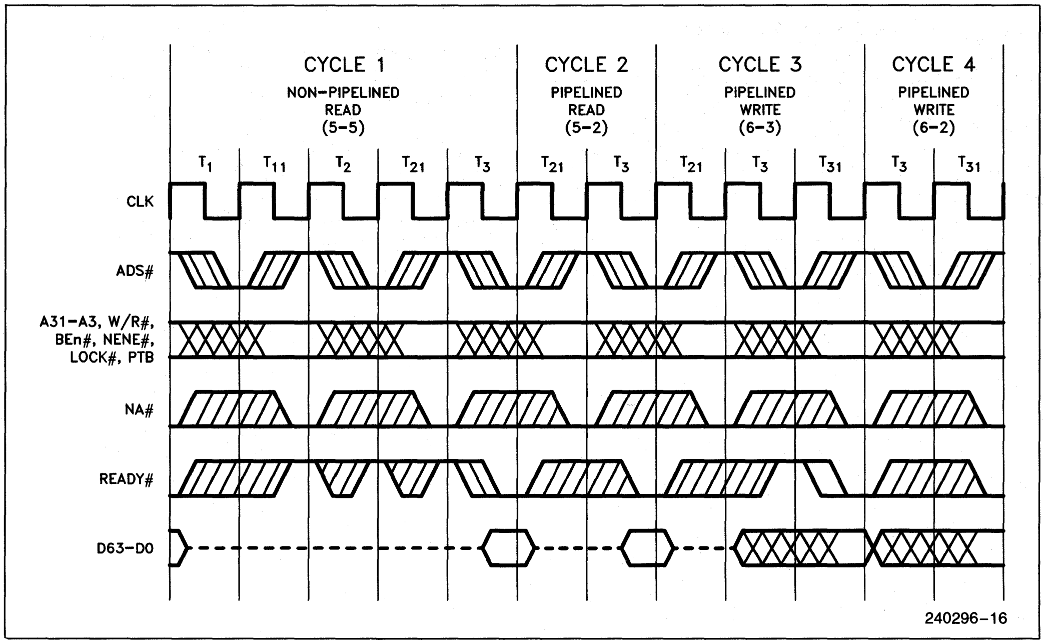


Figure 4.5. Pipelined Read Followed by Pipelined Write

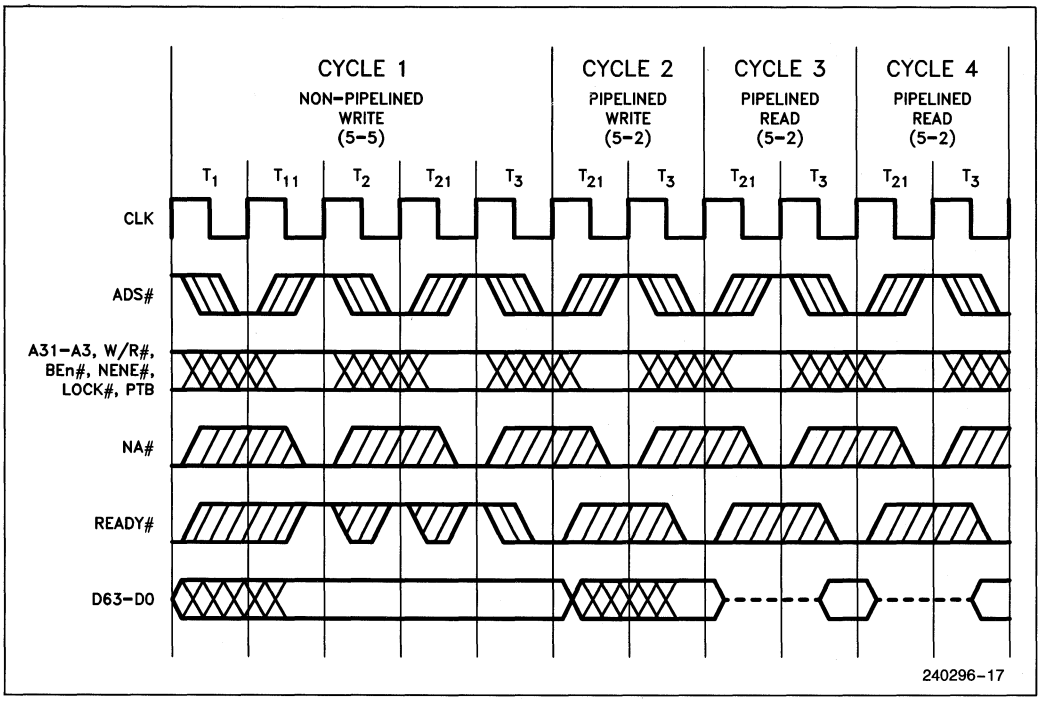


Figure 4.6. Pipelined Write Followed by Pipelined Read

4.3.3 PIPELINED READ AND WRITE CYCLES

Figures 4.5 and 4.6 illustrate combinations of non-pipelined and pipelined read and write cycles. The following description applies to both diagrams. While Cycle 1 is still in progress, two new cycles are initiated. By the time READY# first becomes active, the state machine has moved through states T_1, T_{11}, T_2, T_{21}, and T_3. Cycles 3 and 4 show how activating READY# terminates an outstanding cycle (Cycle 3 in this case), and yet activating NA# while there is an internal request pending adds a new outstanding cycle.

In Figure 4.5, Cycle 3 is a write cycle following a read cycle; therefore, one wait state must be inserted. The 860 microprocessor does not drive the data bus until one clock after the read data is returned from the preceding read cycle. During Cycles 3 and 4, the state machine oscillates between states T_3 and T_{31} maintaining full bus capacity (two levels of pipelining; three outstanding cycles). Cycles 2, 3, and 4 in Figure 4.6 are 5-2 cycles; i.e. each requires a total cycle time of five clocks while the throughput rate is one cycle every two clocks.

Figure 4.7 illustrates in a more general manner how the NA# signal controls pipelining. Cycle 1 is a 2-2 cycle, the fastest possible. The next cycle cannot be started any earlier; therefore, there is no need to activate NA# to start the next cycle early. Cycle 2, a 3-3 read, is different. Cycle 3 can be started during the third state (a wait state) of Cycle 2, and NA# is asserted to accomplish this.

NA# is not activated following the ADS# clock of Cycle 3, thereby allowing Cycle 3 to terminate before the start of Cycle 4. As a result, Cycle 4 is a nonpipelined cycle.

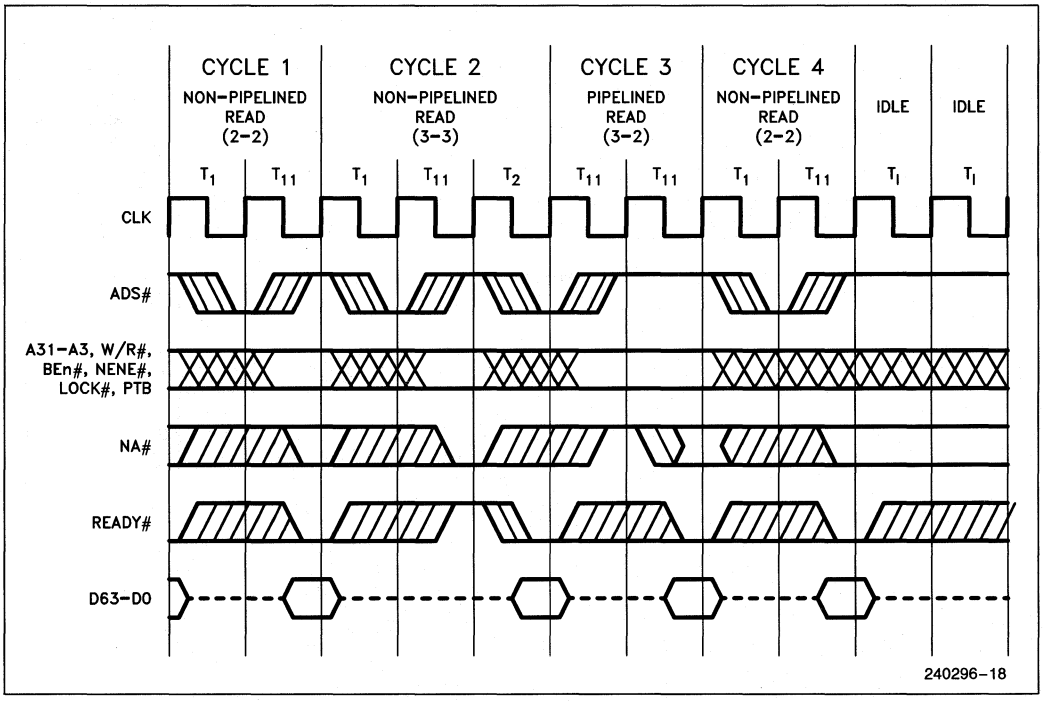


Figure 4.7. Pipelining Driven by NA#

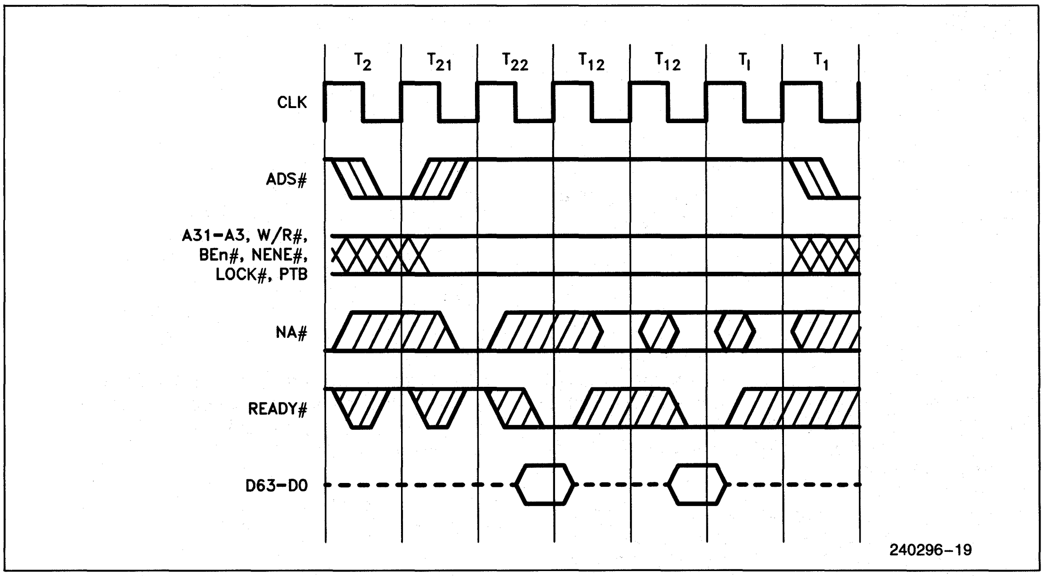

T_2 T_{21} T_{22} T_{12} T_{12} T_I T_1
CLK
ADS#
A31-A3, W/R#, BEn#, NENE#, LOCK#, PTB
NA#
READY#
D63-D0
240296-19

Figure 4.8. NA# Active with No Internal Bus Request

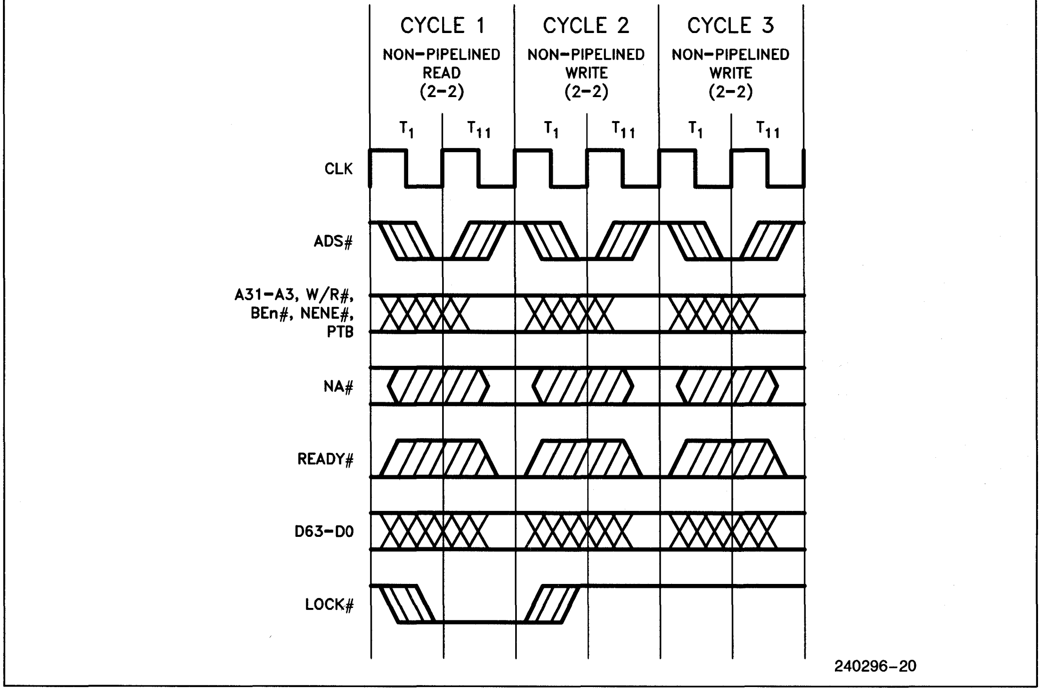


Figure 4.9. Locked Cycles

When there is no internal bus request, activating NA# does not start a new cycle; the 860 microprocessor, however, remembers that NA# has been activated. Figure 4.8 illustrates the situation where NA# is active but no internal bus request is pending. NA# is activated when two cycles are outstanding. Because there is no internal request pending until after one idle state, no new bus cycle is started during that period.

4.3.4 LOCKED CYCLES

The LOCK# signal is asserted when the current bus cycle is to be locked with the next bus cycle. Assertion of LOCK# may be initiated by a program's setting the BL bit of the **dirbase** register (refer to section 2) or by the 860 micrcprocessor itself during page table updates.

In Figure 4.9, the first read cycle is to be locked with the following write cycle. If there were idle states between the cycles, the LOCK# signal would remain asserted. This is the case for a read/modify/write operation. HOLD is not acknowledged until all locked cycles are finished. The second write cycle is not locked because LOCK# is no longer asserted when the first write cycle starts.

4.3.5 HOLD AND BREQ ARBITRATION CYCLES

The HOLD, HLDA, and BREQ signals permit bus arbitration between the 860 microprocessor and another bus master.

As Figure 4.10 illustrates, the T_H (hold) state can be entered only from the idle state T_I. When HOLD is asserted, the 860 microprocessor keeps control of the bus until all outstanding cycles, including locked cycles, are completed.

If HOLD were asserted one clock earlier, the last 860 microprocessor bus cycle before HLDA would not be started. LOCK# is assumed to be inactive in the last bus cycle. If LOCK# were active the 860 microprocessor would not give up the bus.

The outputs (except HLDA and BREQ) float when HLDA is asserted. HOLD is sampled at the end of the clock in which it is activated. Recommended setup and hold times must be met to guarantee sampling one clock after external HOLD activation. When HOLD is sampled active, a one clock delay for internal synchronization follows. HLDA may be deasserted as early as the clock following deassertion of HOLD.

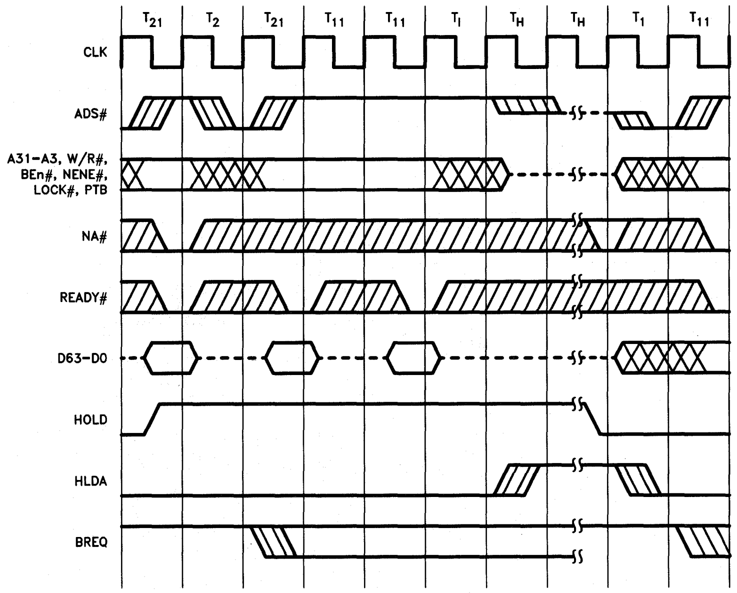


Figure 4.10. HOLD, HLDA, and BREQ

If, during a HOLD cycle, an internal bus request is generated, BREQ is activated even though HLDA is asserted. It remains active at least until the clock after ADS# is activated for the requested cycle.

4.4 Bus States During RESET

Figure 4.11 shows how INT/CS8 is sampled during the clock period just before the falling edge of RESET. If INT/CS8 is sampled active, the 860 microprocessor enters CS8 mode. No inputs (except for HOLD, INT/CS8, and CC1–CC0) are sampled during RESET.

Note that, because HOLD is recognized even while RESET is active, the HLDA output signal may also become active during RESET. Refer to Figure "Output Pin Status during Reset" in section 3.

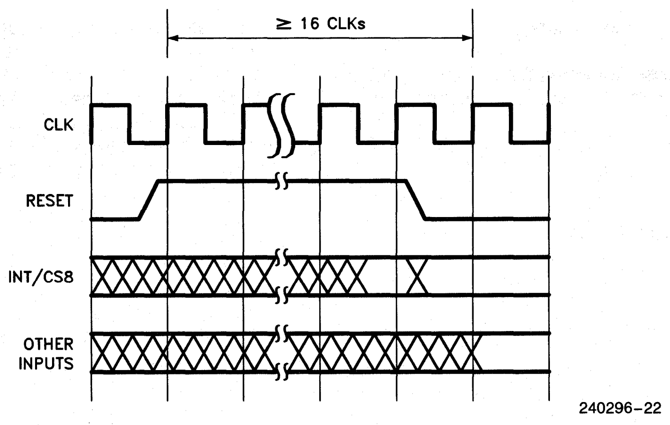
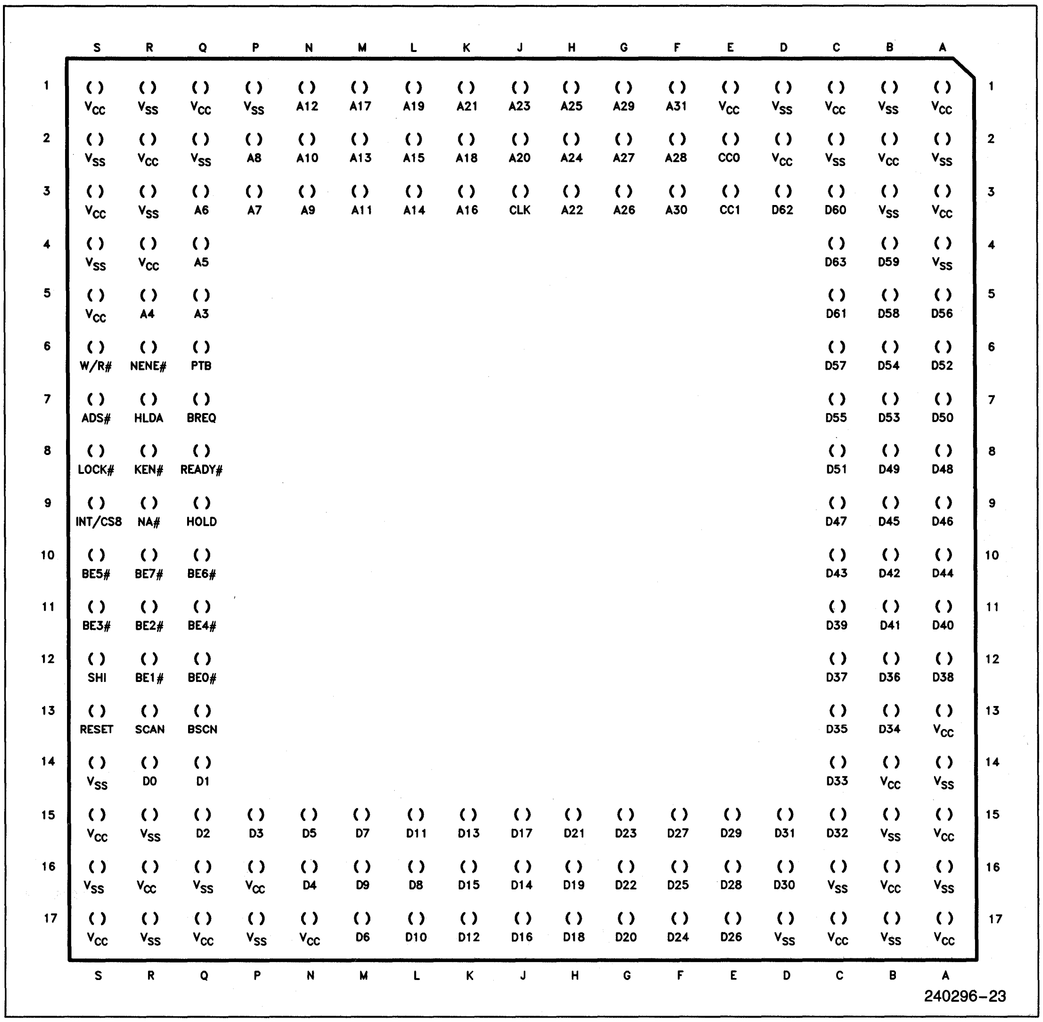


Figure 4.11. Reset Activities

5.0 MECHANICAL DATA

Figures 5.1 and 5.2 show the locations of pins; Tables 5.1 and 5.2 help to locate pin identifiers.



	S	R	Q	P	N	M	L	K	J	H	G	F	E	D	C	B	A	
1	V_{CC}	V_{SS}	V_{CC}	V_{SS}	A12	A17	A19	A21	A23	A25	A29	A31	V_{CC}	V_{SS}	V_{CC}	V_{SS}	V_{CC}	1
2	V_{SS}	V_{CC}	V_{SS}	A8	A10	A13	A15	A18	A20	A24	A27	A28	CC0	V_{CC}	V_{SS}	V_{CC}	V_{SS}	2
3	V_{CC}	V_{SS}	A6	A7	A9	A11	A14	A16	CLK	A22	A26	A30	CC1	D62	D60	V_{SS}	V_{CC}	3
4	V_{SS}	V_{CC}	A5												D63	D59	V_{SS}	4
5	V_{CC}	A4	A3												D61	D58	D56	5
6	W/R#	NENE#	PTB												D57	D54	D52	6
7	ADS#	HLDA	BREQ												D55	D53	D50	7
8	LOCK#	KEN#	READY#												D51	D49	D48	8
9	INT/CS8	NA#	HOLD												D47	D45	D46	9
10	BE5#	BE7#	BE6#												D43	D42	D44	10
11	BE3#	BE2#	BE4#												D39	D41	D40	11
12	SHI	BE1#	BE0#												D37	D36	D38	12
13	RESET	SCAN	BSCN												D35	D34	V_{CC}	13
14	V_{SS}	D0	D1												D33	V_{CC}	V_{SS}	14
15	V_{CC}	V_{SS}	D2	D3	D5	D7	D11	D13	D17	D21	D23	D27	D29	D31	D32	V_{SS}	V_{CC}	15
16	V_{SS}	V_{CC}	V_{SS}	V_{CC}	D4	D9	D8	D15	D14	D19	D22	D25	D28	D30	V_{SS}	V_{CC}	V_{SS}	16
17	V_{CC}	V_{SS}	V_{CC}	V_{SS}	V_{CC}	D6	D10	D12	D16	D18	D20	D24	D26	V_{SS}	V_{CC}	V_{SS}	V_{CC}	17
	S	R	Q	P	N	M	L	K	J	H	G	F	E	D	C	B	A	

240296-23

Figure 5.1. Pin Configuration—View from Top Side

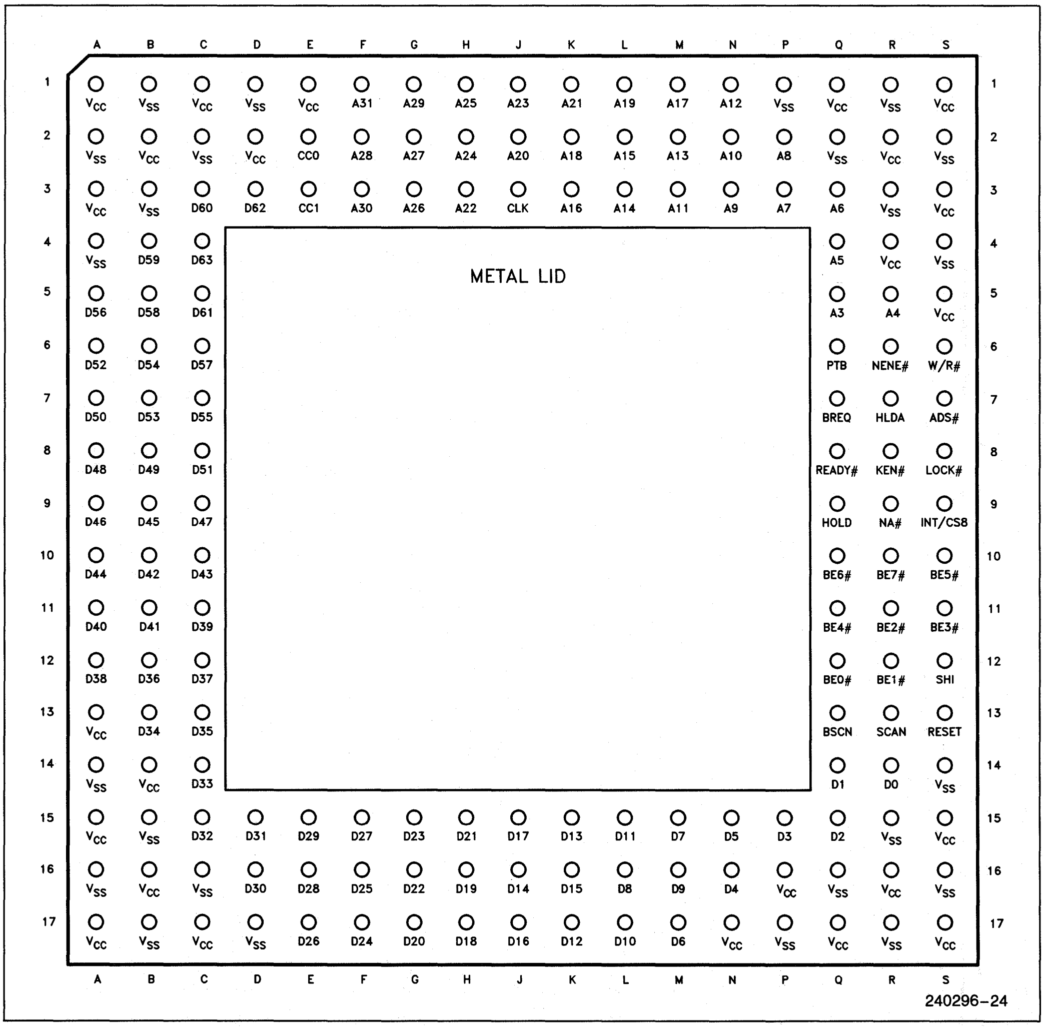

A B C D E F G H J K L M N P Q R S
1: Vcc Vss Vcc Vss Vcc A31 A29 A25 A23 A21 A19 A17 A12 Vss Vcc Vss Vcc
2: Vss Vcc Vss Vcc CC0 A28 A27 A24 A20 A18 A15 A13 A10 A8 Vss Vcc Vss
3: Vcc Vss D60 D62 CC1 A30 A26 A22 CLK A16 A14 A11 A9 A7 A6 Vss Vcc
4: Vss D59 D63 A5 Vcc Vss
5: D56 D58 D61 A3 A4 Vcc
6: D52 D54 D57 PTB NENE# W/R#
7: D50 D53 D55 BREQ HLDA ADS#
8: D48 D49 D51 READY# KEN# LOCK#
9: D46 D45 D47 HOLD NA# INT/CS8
10: D44 D42 D43 BE6# BE7# BE5#
11: D40 D41 D39 BE4# BE2# BE3#
12: D38 D36 D37 BE0# BE1# SHI
13: Vcc D34 D35 BSCN SCAN RESET
14: Vss Vcc D33 D1 D0 Vss
15: Vcc Vss D32 D31 D29 D27 D23 D21 D17 D13 D11 D7 D5 D3 D2 Vss Vcc
16: Vss Vcc Vss D30 D28 D25 D22 D19 D14 D15 D8 D9 D4 Vcc Vss Vcc Vss
17: Vcc Vss Vcc Vss D26 D24 D20 D18 D16 D12 D10 D6 Vcc Vss Vcc Vss Vcc
METAL LID
240296-24

Figure 5.2. Pin Configuration—View from Pin Side

Table 5.1. Pin Cross Reference by Location

Location	Signal	Location	Signal	Location	Signal	Location	Signal
A1	V_{CC}	C9	D47	J15	D17	Q10	BE6#
A2	V_{SS}	C10	D43	J16	D14	Q11	BE4#
A3	V_{CC}	C11	D39	J17	D16	Q12	BE0#
A4	V_{SS}	C12	D37	K1	A21	Q13	BSCN
A5	D56	C13	D35	K2	A18	Q14	D1
A6	D52	C14	D33	K3	A16	Q15	D2
A7	D50	C15	D32	K15	D13	Q16	V_{SS}
A8	D48	C16	V_{SS}	K16	D15	Q17	V_{CC}
A9	D46	C17	V_{CC}	K17	D12	R1	V_{SS}
A10	D44	D1	V_{SS}	L1	A19	R2	V_{CC}
A11	D40	D2	V_{CC}	L2	A15	R3	V_{SS}
A12	D38	D3	D62	L3	A14	R4	V_{CC}
A13	V_{CC}	D15	D31	L15	D11	R5	A4
A14	V_{SS}	D16	D30	L16	D8	R6	NENE#
A15	V_{CC}	D17	V_{SS}	L17	D10	R7	HLDA
A16	V_{SS}	E1	V_{CC}	M1	A17	R8	KEN#
A17	V_{CC}	E2	CC0	M2	A13	R9	NA#
B1	V_{SS}	E3	CC1	M3	A11	R10	BE7#
B2	V_{CC}	E15	D29	M15	D7	R11	BE2#
B3	V_{SS}	E16	D28	M16	D9	R12	BE1#
B4	D59	E17	D26	M17	D6	R13	SCAN
B5	D58	F1	A31	N1	A12	R14	D0
B6	D54	F2	A28	N2	A10	R15	V_{SS}
B7	D53	F3	A30	N3	A9	R16	V_{CC}
B8	D49	F15	D27	N15	D5	R17	V_{SS}
B9	D45	F16	D25	N16	D4	S1	V_{CC}
B10	D42	F17	D24	N17	V_{CC}	S2	V_{SS}
B11	D41	G1	A29	P1	V_{SS}	S3	V_{CC}
B12	D36	G2	A27	P2	A8	S4	V_{SS}
B13	D34	G3	A26	P3	A7	S5	V_{CC}
B14	V_{CC}	G15	D23	P15	D3	S6	W/R#
B15	V_{SS}	G16	D22	P16	V_{CC}	S7	ADS#
B16	V_{CC}	G17	D20	P17	V_{SS}	S8	LOCK#
B17	V_{SS}	H1	A25	Q1	V_{CC}	S9	INT/CS8
C1	V_{CC}	H2	A24	Q2	V_{SS}	S10	BE5#
C2	V_{SS}	H3	A22	Q3	A6	S11	BE3#
C3	D60	H15	D21	Q4	A5	S12	SHI
C4	D63	H16	D19	Q5	A3	S13	RESET
C5	D61	H17	D18	Q6	PTB	S14	V_{SS}
C6	D57	J1	A23	Q7	BREQ	S15	V_{CC}
C7	D55	J2	A20	Q8	READY#	S16	V_{SS}
C8	D51	J3	CLK	Q9	HOLD	S17	V_{CC}

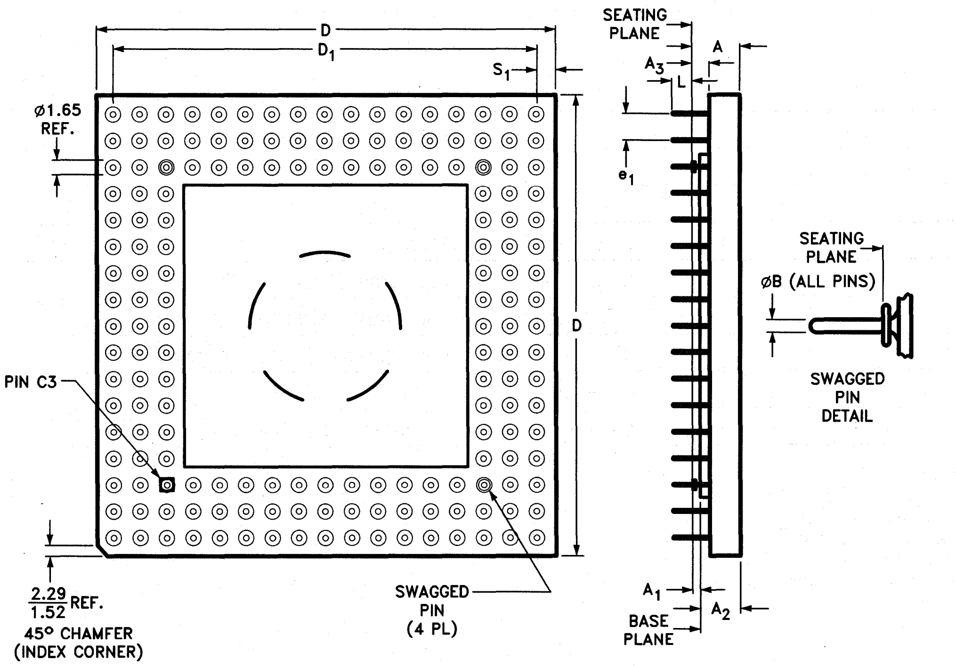
Table 5.2. Pin Cross Reference by Pin Name

Signal	Location	Signal	Location	Signal	Location	Signal	Location
A3	Q5	CLK	J3	D41	B11	V_{CC}	B16
A4	R5	D0	R14	D42	B10	V_{CC}	C1
A5	Q4	D1	Q14	D43	C10	V_{CC}	C17
A6	Q3	D2	Q15	D44	A10	V_{CC}	D2
A7	P3	D3	P15	D45	B9	V_{CC}	E1
A8	P2	D4	N16	D46	A9	V_{CC}	N17
A9	N3	D5	N15	D47	C9	V_{CC}	P16
A10	N2	D6	M17	D48	A8	V_{CC}	Q1
A11	M3	D7	M15	D49	B8	V_{CC}	Q17
A12	N1	D8	L16	D50	A7	V_{CC}	R2
A13	M2	D9	M16	D51	C8	V_{CC}	R4
A14	L3	D10	L17	D52	A6	V_{CC}	R16
A15	L2	D11	L15	D53	B7	V_{CC}	S1
A16	K3	D12	K17	D54	B6	V_{CC}	S3
A17	M1	D13	K15	D55	C7	V_{CC}	S5
A18	K2	D14	J16	D56	A5	V_{CC}	S15
A19	L1	D15	K16	D57	C6	V_{CC}	S17
A20	J2	D16	J17	D58	B5	V_{SS}	A2
A21	K1	D17	J15	D59	B4	V_{SS}	A4
A22	H3	D18	H17	D60	C3	V_{SS}	A14
A23	J1	D19	H16	D61	C5	V_{SS}	A16
A24	H2	D20	G17	D62	D3	V_{SS}	B1
A25	H1	D21	H15	D63	C4	V_{SS}	B3
A26	G3	D22	G16	HLDA	R7	V_{SS}	B15
A27	G2	D23	G15	HOLD	Q9	V_{SS}	B17
A28	F2	D24	F17	INT/CS8	S9	V_{SS}	C2
A29	G1	D25	F16	KEN#	R8	V_{SS}	C16
A30	F3	D26	E17	LOCK#	S8	V_{SS}	D1
A31	F1	D27	F15	NA#	R9	V_{SS}	D17
ADS#	S7	D28	E16	NENE#	R6	V_{SS}	P1
BE0#	Q12	D29	E15	PTB	Q6	V_{SS}	P17
BE1#	R12	D30	D16	READY#	Q8	V_{SS}	Q2
BE2#	R11	D31	D15	RESET	S13	V_{SS}	Q16
BE3#	S11	D32	C15	SCAN	R13	V_{SS}	R1
BE4#	Q11	D33	C14	SHI	S12	V_{SS}	R3
BE5#	S10	D34	B13	V_{CC}	A1	V_{SS}	R15
BE6#	Q10	D35	C13	V_{CC}	A3	V_{SS}	R17
BE7#	R10	D36	B12	V_{CC}	A13	V_{SS}	S2
BREQ	Q7	D37	C12	V_{CC}	A15	V_{SS}	S4
BSCN	Q13	D38	A12	V_{CC}	A17	V_{SS}	S14
CC0	E2	D39	C11	V_{CC}	B2	V_{SS}	S16
CC1	E3	D40	A11	V_{CC}	B14	W/R#	S6

Table 5.3. Ceramic PGA Package Dimension Symbols

Letter or Symbol	Description of Dimensions
A	Distance from seating plane to highest point of body
A_1	Distance between seating plane and base plane (lid)
A_2	Distance from base plane to highest point of body
A_3	Distance from seating plane to bottom of body
B	Diameter of terminal lead pin
D	Largest overall package dimension of length
D_1	A body length dimension, outer lead center to outer lead center
e_1	Linear spacing between true lead position centerlines
L	Distance from seating plane to end of lead
S_1	Other body dimension, outer lead center to edge of body

NOTES:
1. Controlling dimension: millimeter.
2. Dimension "e_1" ("e") is non-cumulative.
3. Seating plane (standoff) is defined by P.C. board hole size: 0.0415–0.0430 inch.
4. Dimensions "B", "B_1" and "C" are nominal.
5. Details of Pin 1 identifier are optional.



Family: Ceramic Pin Grid Array Package						
Symbol	Millimeters			Inches		
	Min	Max	Notes	Min	Max	Notes
A	3.56	4.57		0.140	0.180	
A_1	0.64	1.14	SOLID LID	0.025	0.045	SOLID LID
A_2	23	0.30	SOLID LID	0.110	0.140	SOLID LID
A_3	1.14	1.40		0.045	0.055	
B	0.43	0.51		0.017	0.020	
D	44.07	44.83		1.735	1.765	
D_1	40.51	40.77		1.595	1.605	
e_1	2.29	2.79		0.090	0.110	
L	2.54	3.30		0.100	0.130	
N	168			168		
S_1	1.52	2.54		0.060	0.100	
ISSUE	IWS REV X 7/15/88					

Figure 5.3. 168 Lead Ceramic PGA Package Dimensions

6.0 PACKAGE THERMAL SPECIFICATIONS

The 860 microprocessor is specified for operation when T_C (the case temperature) is within the range of 0°C–85°C. T_C may be measured in any environment to determine whether the 860 microprocessor is within specified operating range. The case temperature should be measured at the center of the top surface opposite the pins.

T_A (the ambient temperature) can be calculated from θ_{CA} (thermal resistance from case to ambient) with the following equation:

$$T_A = T_C - P^* \theta_{CA}$$

Typical values for θ_{CA} at various airflows are given in Table 6.1 for the 1.75 sq. in., 168 pin, ceramic PGA.

Table 6.2 shows the maximum T_A allowable (without exceeding T_C) at various airflows and operating frequencies (f_{CLK}).

Note that T_A is greatly improved by attaching "fins" or a "heat sink" to the package. P (the maximum power consumption) is calculated by using the maximum I_{CC} at 5V as tabulated in the *DC Characteristics* of section 7.

Table 6.1. Thermal Resistance (θ_{CA}) at Various Airflows

In °C/Watt

	Airflow-ft/min (m/sec)					
	0 (0)	**200 (1.01)**	**400 (2.03)**	**600 (3.04)**	**800 (4.06)**	**1000 (5.07)**
θ_{CA} with Heat Sink*	13	9	5.5	5.0	3.9	3.4
θ_{CA} without Heat Sink	17	14	11	9	7.1	6.6

*0.285" high unidirectional heat sink (Al alloy 6061, 50 mil fin width, 150 mil center-to-center fin spacing).

Table 6.2. Maximum T_A at Various Airflows

In °C

		Airflow-ft/min (m/sec)					
	f_{CLK} (MHz)	**0 (0)**	**200 (1.01)**	**400 (2.03)**	**600 (3.04)**	**800 (4.06)**	**1000 (5.07)**
T_A with Heat Sink*	33.3	46	58	69	70	73	75
	40.0	43	56	67	69	72	74
T_A without Heat Sink	33.3	34	43	52	58	64	65
	40.0	30	40	49	56	62	64

*0.285" high unidirectional heat sink (Al alloy 6061, 50 mil fin width, 150 mil center-to-center fin spacing).

7.0 ELECTRICAL DATA

Inputs and outputs are TTL compatible. All input and output timings are specified relative to the 1.5 volt level of the rising edge of CLK and refer to the point that the signal reaches 1.5V.

7.1 Absolute Maximum Ratings

Case Temperature T_C under Bias ...... 0°C to 85°C

Storage Temperature .... −65°C to +150°C

Voltage on Any Pin
with Respect to Ground .... −0.5 to V_{CC} + 0.5V

**Notice: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

NOTICE: Specifications contained within the following tables are subject to change.

7.2 D.C. Characteristics

Table 7.1. DC Characteristics

Symbol	Parameter	Min	Max	Units	Notes
V_{IL}	Input LOW Voltage	−0.3	+0.8	V	
V_{IH}	Input HIGH Voltage	2.0	V_{CC} + 0.3	V	
V_{OL}	Output LOW Voltage		0.45	V	(Note 1)
V_{OH}	Output HIGH Voltage	2.4		V	(Note 2)
I_{CC}	Power Supply Current				
	CLK = 33.3 MHz		600	mA	V_{CC} @5V
	CLK = 40.0 MHz		650	mA	V_{CC} @5V
I_{LI}	Input Leakage Current		±15	μA	No pullup or pulldown
I_{LO}	Output Leakage Current		±15	μA	
C_{IN}	Input Capacitance		15	pF	
C_O	I/O or Output Capacitance		15	pF	
C_{CLK}	Clock Capacitance		20	pF	

NOTES:
1. This parameter is measured at 4.0 mA for address, data, and byte enables; at 5.0 mA for definition and control.
2. This parameter is measured at 1.0 mA for address, data, and byte enables; at 0.9 mA for definition and control.

7.3 A.C. Characteristics

Table 7.2. A.C. Characteristics $T_C = 0$ to 85°C, $V_{CC} = 5V \pm 5\%$
All timings measured at 1.5V unless otherwise specified.

Symbol	Parameter	33.3 MHz		40.0 MHz		Test Conditions
		Min (ns)	Max (ns)	Min (ns)	Max (ns)	
t1	CLK period	30	125	25	125	
t2	CLK high time	7		5		at 2V
t3	CLK low time	7		5		at .8V
t4	CLK fall time		4		4	
t5	CLK rise time		4		4	
t6a	A31–A3, PTB, W/R#, NENE# valid	3.5	38	3.5	29	50 pF load
t6b	BEn# valid	3.5	41	3.5	32	50 pF load
t7	Float time, all outputs	3.5	30	3.5	25	(Note 1)
t8	ADS#, BREQ, LOCK#, HLDA valid delay	3.5	26	3.5	21	50 pF load
t9	D63–D0 valid delay	3.5	47	3.5	43	50 pF load
t10	Setup time, all inputs except INT, HOLD	13		10		
t11	Hold time, all inputs except INT, HOLD	4		4		
t12	INT, HOLD setup time	22		15		(Note 2)
t13	INT, HOLD hold time	5		5		(Note 2)

NOTES:
1. Float condition occurs when maximum output current becomes less than I_{LO} in magnitude. Float delay is not tested.
2. INT and HOLD are asynchronous inputs. The setup and hold specifications are given for test purposes or to assure recognition on a specific rising edge of CLK.

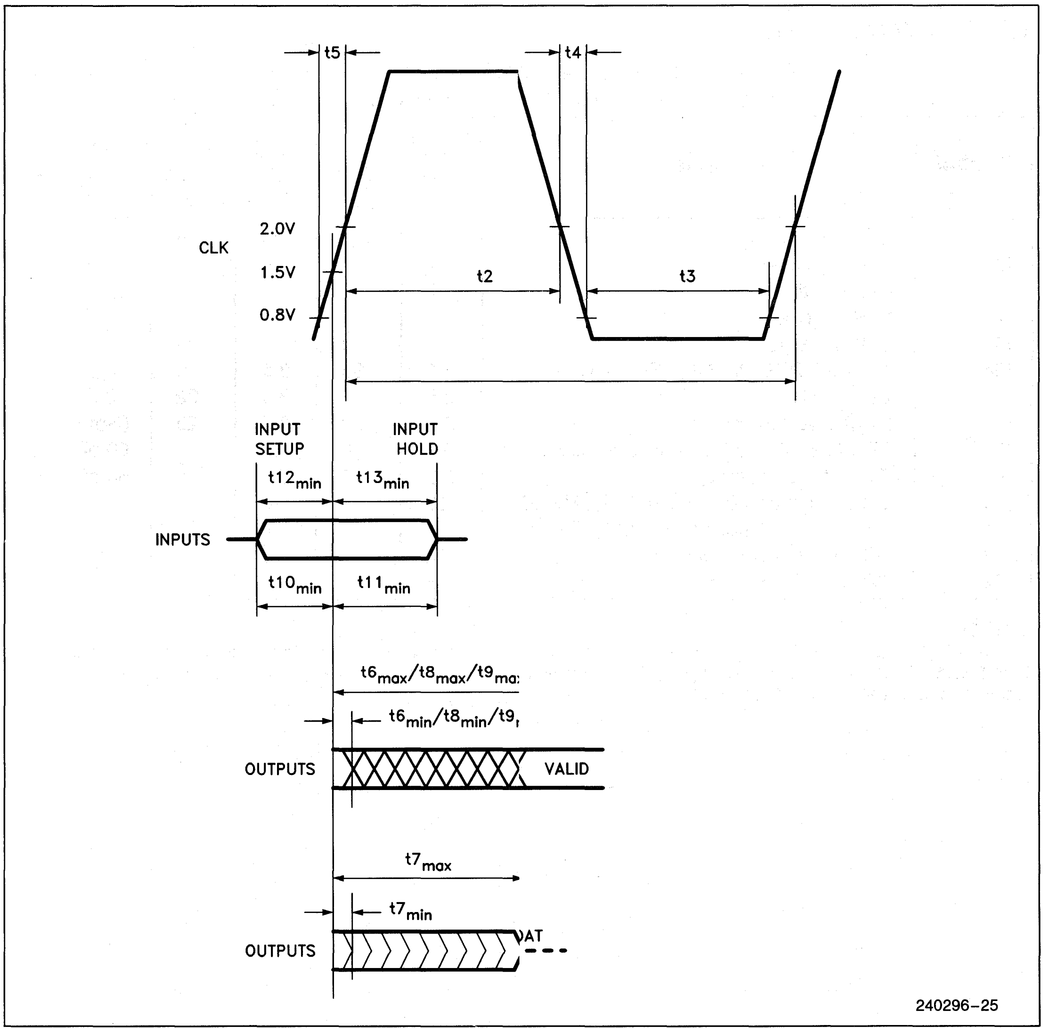
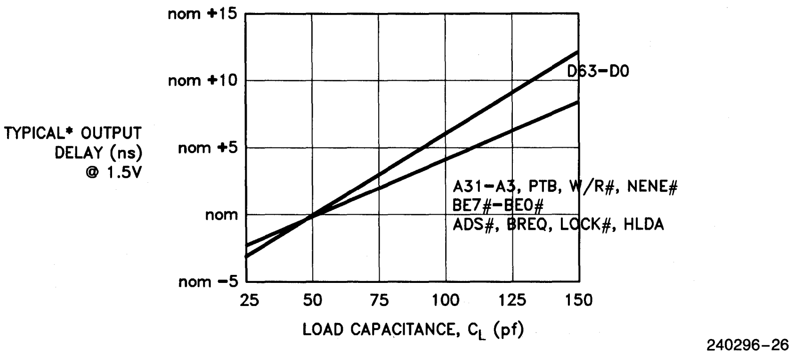
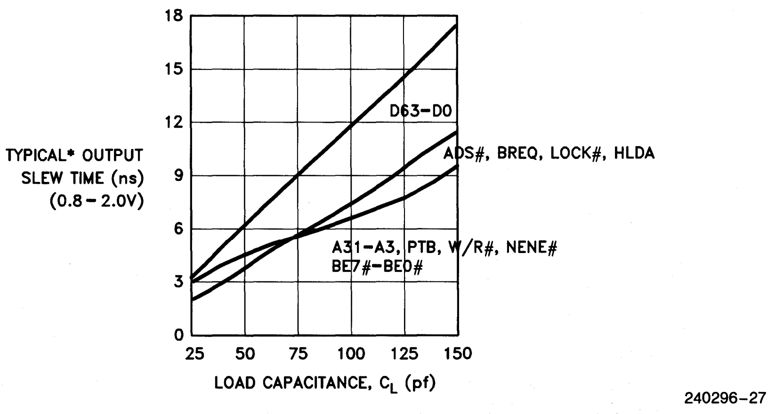


Figure 7.1. CLK, Input, and Output Timings



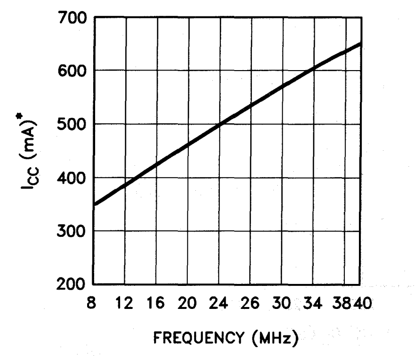
NOTES:
Graphs are not linear outside the C_L range shown.
nom = nominal value given in the AC timing table.
*Typical part under worst-case conditions.

Figure 7.2. Typical Output Delay vs Load Capacitance under Worst-Case Conditions



NOTES:
Graphs are not linear outside the C_L range shown.
*Typical part under worst-case conditions.

Figure 7.3. Typical Slew Time vs Load Capacitance under Worst-Case Conditions



NOTES:
Graphs are not linear outside the frequency range shown.
*Worst-case supply current at 5V.

Figure 7.4. Typical I_{CC} vs Frequency

8.0 INSTRUCTION SET

Key to abbreviations:

src1	A register (integer or floating-point depending on class of instruction) or a 16-bit immediate value. The immediate value is sign-extended for add and subtract operations and zero-extended for logical operations.
src1ni	Same as *src1* except that no immediate value is permitted.
src2	A register (integer or floating-point depending on class of instruction).
rdest	A register (integer or floating-point depending on class of instruction).
freg	A floating-point register.
ireg	An integer register.
ctrlreg	One of the control registers **fir, psr, dirbase, db**, or **fsr**.
#const	A 16-bit immediate address offset that the 860 microprocessor sign-extends to 32 bits when computing the effective address.
mem.x(address)	The contents of the memory location indicated by *address* with a size of *x*.
.p	Precision specification. Unless otherwise specified, floating-point operations accept single- or double-precision source operands and produce a result of equal or greater precision. Both input operands must have the same precision. The source and result precision are specified by a two-letter suffix to the mnemonic of the operation, as shown in the table below.

Suffix	Source Precision	Result Precision
.ss	single	single
.sd	single	double
.dd	double	double

.w	**.ss** (32 bits), or **.dd** (64 bits)
.x	**.b** (8 bits), **.s** (16 bits), or **.l1** (32 bits)
.y	**.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)
.z	**.l** (32 bits, or **.d** (64 bits)
lbroff	A signed, 26-bit, immediate, relative branch offset
sbroff	A signed, 16-bit, immediate, relatvie branch offset
brx	A function that computes the target address of a branch by shifting the offset (either *lbroff* or *sbroff*) left by two bits, sign-extending it to 32 bits, and adding the result to the address of the current control-transfer instruction plus four.
src1s	An integer register or a 5-bit immediate that is zero-extended to 32 bits.
PM	The pixel mask, which is considered as an array of eight bits PM[0]..PM[7], where PM[0] is the least significant bit.

8.1 Instruction Definitions in Alphabetical Order

adds *src1, src2, rdest* .... **Add Signed**

```
rdest ← src1 + src2
OF ← (bit 31 carry ≠ bit 30 carry)
CC set if src2 < comp2(src1) (signed)
CC clear if src2 ≥ comp2(src1) (signed)
```

addu *src1, src2, rdest* .... **Add Unsigned**

```
rdest ← src1 + src2
OF ← bit 31 carry
CC ← bit 31 carry
```

and *src1, src2, rdest* .... **Logical AND**

```
rdest ← src1 and src2
> CC set if result is zero, cleared otherwise
```

andh *#const, src2, rdest* .... **Logical AND High**

```
rdest ← (#const shifted left 16 bits) and src2
> CC set if result is zero, cleared otherwise
```

andnot *src1, src2, rdest* .... **Logical AND NOT**

```
rdest ← not src1 and src2
> CC set if result is zero, cleared otherwise
```

andnoth *#const, src2, rdest* .... **Logical AND NOT High**

```
rdest ← not (#const shifted left 16 bits) and src2
> CC set if result is zero, cleared otherwise
```

bc *lbroff* .... **Branch on CC**

```
IF    CC = 1
THEN  continue execution at brx(lbroff)
FI
```

bc.t *lbroff* .... **Branch on CC, Taken**

```
IF    CC = 1
THEN  execut one more sequential instruction
      continue execution at brx(lbroff)
ELSE  skip next sequential instruction
FI
```

bla *src1ni, src2, sbroff* .... **Branch on LCC and Add**

```
      LCC-temp clear if src2 < comp2(src1ni) (signed)
      LCC-temp set if src2 ≥ comp2(src1ni) (signed)
src2 ← src1ni + src2
Execute one more sequential instruction
IF    LCC
THEN  LCC ← LCC-temp
      continue execution at brx(sbroff)
ELSE  LCC ← LCC-temp
FI
```

bnc *lbroff* .... **Branch on Not CC**

```
IF    CC = 0
THEN  continue execution at brx(lbroff)
FI
```

bnc.t *lbroff* .... **Branch on Not CC, Taken**

```
IF    CC = 0
THEN  execute one more sequential instruction
      continue execution at brx(lbroff)
ELSE  skip next sequential instruction
FI
```

br *lbroff* .... **Branch Direct Unconditionally**

```
Execute one more sequential instruction.
Continue execution at brx(lbroff).
```

bri *[src1ni]* .... **Branch indirect unconditionally**

```
Execute one more sequential instruction
IF      any trap bit in psr is set
THEN    copy PU to U, PIM to IM in psr
        clear trap bits
        IF      DS is set and DIM is reset
        THEN    enter dual-instruction mode after executing one
                  instruction in single-instruction mode
        ELSE    IF      DS is set and DIM is set
                THEN    enter single-instruction mode after executing one
                          instruction in dual-instruction mode
                ELSE    IF      DIM is set
                        THEN    enter dual-instruction mode
                                  for next two instructions
                        ELSE    enter single-instruction mode
                                  for next two instructions
                        FI
                FI
        FI
FI
Continue execution at address in src1ni
  (The original contents of src1ni is used even if the next instruction
  modifies src1ni. Does not trap if src1ni is misaligned.)
```

bte *src1s, src2, sbroff* .... **Branch If Equal**

```
IF      src1s = src2
THEN    continue execution at brx(sbroff)
FI
```

btne *src1s, src2, sbroff* .... **Branch If Not Equal**

```
IF      src1s ≠ src2
THEN    continue execution at brx(sbroff)
FI
```

call *lbroff* .... **Subroutine Call**

```
r1 ← address of next sequential instruction + 4
Execute one more sequential instruction
Continue execution at brx(lbroff)
```

calli *[src1ni]* .... **Indirect Subroutine Call**

```
r1 ← address of next sequential instruction + 4
Execute one more sequential instruction
Continue execution at address in src1ni
    (The original contents of src1ni is used even if the next instruction
    modifies src1ni. Does not trap if src1ni is misaligned.)
```

fadd.p *src1, src2, rdest* .... **Floating-Point Add**

rdest ← *src1* + *src2*

faddp *src1, src2, rdest* .... **Add with Pixel Merge**

rdest ← *src1* + *src2*
Shift and load MERGE register as defined in Table 8.1

faddz *src1, src2, rdest* .... **Add with Z Merge**

rdest ← *src1* + *src2*
Shift MERGE right 16 and load fields 31..16 and 63..48

fiadd.w *src1, src2, rdest* .... **Long-Integer Add**

rdest ← *src1* + *src2*

fisub.w *src1, src2, rdest* .... **Long-Integer Subtract**

rdest ← *src1* − *src2*

fix.p *src1, rdest* .... **Floating-Point to Integer Conversion**

rdest ← 64- bit value with low-order 32 bits equal to integer part of *src1* rounded

fld.y *src1(src2), freg* .... **Floating-Point Load (Normal)**

fld.y *src1(src2)++, freg* . . . . **(Autoincrement)**
```
freg ← mem.y (src1 + src2)
IF autoincrement
THEN src2 ← src1 + src2
FI
```

Cache Flush

flush *#const(src2)* . . . . **(Normal)**

flush *#const(src2)++* . . . . **(Autoincrement)**
```
Replace block in data cache with address (#const + src2).
Contents of block undefined.
IF autoincrement
THEN src2 ← #const + src2
FI
```

fmlow.p *src1, src2, rdest* . . . . **Floating-Point Multiply Low**
```
rdest ← low-order 53 bits of src1 mantissa × src2 mantissa
rdest bit 53 ← most significant bit of mantissa
```

fmov.r *src1, rdest* . . . . **Floating-Point Reg-Reg Move**
```
Assembler pseudo-operation
    fmov.ss src1, rdest = fiadd.ss src1, f0, rdest
    fmov.dd src1, rdest = fiadd.dd src1, f0, rdest
    fmov.sd src1, rdest = fadd. sd src1, f0, rdest
    fmov.ds src1, rdest = fadd.ds src1, f0, rdest
```

fmul.p *src1, src2, rdest* . . . . **Floating-Point Multiply**
```
rdest ← src1 × src2
```

fnop . . . . **Floating-Point No Operation**
```
Assembler pseudo-operation
    fnop = shrd r0, r0, r0
```

form *src1, rdest* . . . . **OR with MERGE Register**
```
rdest ← src1 OR MERGE
MERGE ← 0
```

frcp.p *src2, rdest* . . . . **Floating-Point Reciprocal**
```
rdest ← 1/src2 with maximum mantissa error < $2^{-7}$
```

frsqr.p *src2, rdest* . . . . **Floating-Point Reciprocal Square Root**
```
rdest ← 1/SQRT (src2) with maximum mantissa error < $2^{-7}$
```

Floating-Point Store

fst.y *freg, src1(src2)* . . . . **(Normal)**

fst.y *freg, src1(src2)++* . . . . **(Autoincrement)**
```
mem.y (src2 + src1) ← freg
IF autoincrement
THEN src2 ← src1 + src2
FI
```

fsub.p *src1, src2, rdest* . . . . **Floating-Point Subtract**
```
rdest ← src1 − src2
```

ftrunc.p *src1, rdest* . . . . **Floating-Point to Integer Conversion**
```
rdest ← 64-bit value with low-order 32 bits equal to integer part of src1
```

fxfr *src1, ireg* . . . . **Transfer F-P to Integer Register**
```
ireg ← src1
```

fzchkl *src1, src2, rdest* . . . . **32-Bit Z-Buffer Check**
```
Consider src1, src2, and rdest as arrays of two 32-bit
    fields src1(0)..src1(1), src2(0)..src2(1), and rdest(0)..rdest(1)
    where zero denotes the least-significant field.
PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
    PM [i + 6] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

fzchks *src1, src2, rdest* .... **16-Bit Z-Buffer Check**

```
Consider src1, src2, and rdest as arrays of four 16-bit
    fields src1(0)..src1(3), src2(0)..src2(3), and rdest(0)..rdest(3)
    where zero denotes the least-significant field.
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

intovr .... **Software Trap on Integer Overflow**

If OF in **epsr** = 1, generate trap with IT set in **psr**.

ixfr *src1ni, freg* .... **Transfer Integer to F-P Register**

freg ← *src1ni*

ld.c *ctrlreg, rdest* .... **Load from Control Register**

rdest ← *ctrlreg*

ld.x *src1(src2), rdest* .... **Load Integer**

rdest ← *mem.x* (*src1* + *src2*)

lock .... **Begin Interlocked Sequence**

Set BL in **dirbase**. The next load or store locks the bus.
Disable interrupts until the bus is unlocked.

mov *src2, rdest* .... **Register-Register Move**

Assembler pseudo-operation
    mov *src2, rdest* = **shl r0,** *src2, rdest*

nop .... **Core-Unit No Operation**

Assembler pseudo-operation
    nop = **sh1 r0, r0, r0**

or *src1, src2, rdest* .... **Logical OR**

rdest ← *src1* OR *src2*
> CC set if result is zero, cleared otherwise

orh *#const, src2, rdest* .... **Logical OR High**

rdest ← (*#const* shifted left 16 bits) OR *src2*
> CC set if result is zero, cleared otherwise

pfadd.p *src1, src2, rdest* .... **Pipelined Floating-Point Add**

rdest ← last A-stage result
Advance a pipeline one stage
A pipeline first stage ← *src1* + *src2*

pfaddp *src1, src2, rdest* .... **Pipelined Add with Pixel Merge**

rdest ← last-stage I-result
last stage I-result ← *src1* + *src2*
Shift and load MERGE register from last-stage I-result as defined in Table 8.1

pfaddz *src1, src2, rdest* .... **Pipelined Add with Z Merge**

rdest ← last-stage I-result
last stage I-result ← *src1* + *src2*
Shift MERGE right 16 and load fields 31..16 and 63..48 from last-stage I-result

pfam.p *src1, src2, rdest* .... **Pipelined Floating-Point Add and Multiply**

rdest ← last A-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 + A-op2
M pipeline first stage ← M-op1 × M-op2

pfeq.p *src1, src2, rdest* .... **Pipelined Floating-Point Equal Compare**

rdest ← last A-stage result
CC set if *src1* = *src2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfgt.p *src1, src2, rdest* .... **Pipelined Floating-Point Greather-Than Compare**

(Assembler clears R-bit of instruction)
rdest ← last A-stage result
CC set if *src1* > *src2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfiadd.w *src1, src2, rdest* .... **Pipelined Long-Integer Add**

rdest ← last-stage I-result
last-stage I-result ← *src1* + *src2*

pfisub.w *src1, src2, rdest* .... **Pipelined Long-Integer Subtract**

rdest ← last-stage I-result
last-stage I-result ← *src1* − *src2*

pfix.p *src1, rdest* .... **Pipelined Floating-Point to Integer Conversion**

rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← 64-bit value with low-order 32 bits
    equal to integer part of *src1* rounded

Pipelined Floating-Point Load

pfld.z *src1(src2), freg* .... **(Normal)**
pfld.z *src1(src2)++, freg* .... **(Autoincrement)**

```
freg ← mem.z (third previous pfld's (src1 + src2))
    (where .z is precision of third previous pfld.z)
If autoincrement
THEN src2 ← src1 + src2
FI
```

pfle.p *src1, src2, rdest* .... **Pipelined F-P Less-Than or Equal Compare**

Assembler pseudo-operation, identical to **pfgt.p** except that
    assembler sets R-bit of instruction.
rdest ← last A-stage result
CC clear if *src1* ≤ *src2*, else set
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfmam.p *src1, src2, rdest* .... **Pipelined Floating-Point Add and Multiply**

rdest ← last M-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 − A-op2
M pipeline first stage ← M-op1 × M-op2

pfmov.r *src1, rdest* .... **Pipelined Floating-Point Reg-Reg Move**

Assembler pseudo-operation
    pfmov.ss *src1, rdest* = **pfiadd.ss** *src1*, **f0,** *rdest*
    pfmov.dd *src1, rdest* = **pfiadd.dd** *src1*, **f0,** *rdest*
    pfmov.sd *src1, rdest* = **pfadd.sd** *src1*, **f0,** *rdest*
    pfmov.ds *src1, rdest* = **pfadd.ds** *src1*, **f0,** *rdest*

pfmsm.p *src1, src2, rdest* .... **Pipelined Floating-Point Subtract and Multiply**

rdest ← last M-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 − A-op2
M pipeline first stage ← M-op1 × M-op2

pfmul.p *src1, src2, rdest* .... **Pipelined Floating-Point Multiply**

rdest ← last M-stage result
Advance M pipeline one stage
M pipeline first stage ← *src1* × *src2*

pfmul3.p *src1, src2, rdest*..**Three-Stage Pipelined Multiply**

```
rdest ← last M-stage result
Advance 3-Stage M pipeline one stage
M pipeline first stage ← src1 × src2
```

pform *src1, rdest*..**Pipelined OR to MERGE Register**

```
rdest ← last-stage I-result
last stage I-result ← src1 OR MERGE
MERGE ← 0
```

pfsm.p *src1, src2, rdest* ....**Pipelined Floating-Point Subtract and Multiply**

```
rdest ← last A-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 − A-op2
M pipeline first stage ← M-op1 × M-op2
```

pfsub.p *src1, src2, rdest* ....**Pipelined Floating-Point Subtract**

```
rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← src1 + src2
```

pftrunc.p *src1, rdest* ....**Pipelined Floating-Point to Integer Conversion**

```
rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← 64-bit value with low-order 32 bits
    equal to integer part of src1
```

pfzchkl *src1, src2, rdest*..**Pipelined 32-Bit Z-Buffer Check**

```
Consider src1, src2, and rdest, as arrays of two 32-bit
    fields src1(0)..src1(1), src2(0)..src2(1), and rdest(0)..rdest(1)
    where zero denotes the least significant field.
PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
    PM [i + 6] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← last-stage I-result
    last-stage I-result ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

pfzchks *src1, src2, rdest*..**Pipelined 16-Bit Z-Buffer Check**

```
Consider src1, src2, and rdest, as arrays of four 16-bit
    fields src1(0)..src1(3), src2(0)..src2(3), and rdest(0)..rdest(3)
    where zero denotes the least significant field.
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← last-stage I-result
    last-stage I-result ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

pst.d *freg, #const(src2)* ....**Pixel Store**
pst.d *freg, #const(src2)++* ....**Pixel Store Autoincrement**

```
Pixels enabled by PM in mem.D (src2 + #const) ← freg
Shift PM right by 8/pixel size (in bytes) bits
IF autoincrement THEN src2 ← #const + src2 FI
```

shl *srdc1, src2, rdest*..**Shift Left**

```
rdest ← src2 shifted left by src1 bits
```

shr *src1, src2, rdest*..**Shift Right**

```
SC (in psr) ← src1
rdest ← src2 shifted right by src1 bits
```

shra *src1, src2, rdest* .... **Shift Right Arithmetic**
rdest ← *src2* arithmetically shifted right by *src1* bits

shrd *src1, src2, rdest* .... **Shift Right Double**
rdest ← low-order 32 bits of *src1:src2* shifted right by SC bits

st.c *src1ni, ctrlreg* .... **Store to Control Register**
ctrlreg ← *src1ni*

st.x *src1ni, #const(src2)* .... **Store Integer**
mem.x (src2 + #const) ← *src1ni*

subs *src1, src2, rdest* .... **Subtract Signed**
rdest ← *src1* − *src2*
OF ← (bit 31 carry ≠ bit 30 carry)
CC set if *src2* > *src1* (signed)
CC clear if *src2* ≤ *src1* (signed)

subu *src1, src2, rdest* .... **Subtract Unsigned**
rdest ← *src1* − *src2*
OF ← NOT (bit 31 carry)
CC ← bit 31 carry
(i.e. CC set if *src2* ≤ *src1* (unsigned)
CC clear if *src2* > *src1* (unsigned)

trap *src1, src2, rdest* .... **Software Trap**
Generate trap with IT set in **psr**

unlock .... **End Interlocked Sequence**
Clear BL in **dirbase**. The next load or store unlocks the bus.

xor *src1, src2, rdest* .... **Logical Exclusive OR**
rdest ← *src1* XOR *src2*
CC set if result is zero, cleared otherwise

xorh *#const, src2, rdest* .... **Logical Exclusive OR High**
rdest ← (*#const* shifted left 16 bit) XOR *src2*
CC set if result is zero, cleared otherwise

Table 8.1. FADDP MERGE Update

Pixel Size (from PS)	Fields Loaded From Result into MERGE	Right Shift Amount (Field Size)
8	63..56, 47..40, 31..24, 15..8	8
16	63..58, 47..42, 31..26, 15..10	6
32	63..56, 31..24	8

8.2 Instruction Format and Encoding

All instructions are one word long and begin on a word boundary. There are two general core-instruction formats: REG-format and CTRL-format. Within the REG-format are several variations.

8.2.1 REG-FORMAT INSTRUCTIONS

The *src2* field selects one of the 32 integer registers (most instructions) or five control registers (**st.c** and **ld.c**). *Dest* selects one of the 32 integer registers (most instructions) or floating-point registers (**fld**, **fst**, **pfld**, **pst**, **ixfr**). For instructions where *src1* is optionally an immediate value, bit 26 of the opcode (I-bit) indicates whether *src1* is an immediate. If bit 26 is clear, an integer register is used; if bit 26 is set, *src1* is contained in the low-order 16 bits, except for **bte** and **btne** instructions. For **bte** and **btne**, the five-bit immediate value is contained in the *src1* field. For **st**, **bte**, **btne**, and **bla**, the upper five bits of the *offset* or *broffset* are contained in the *dest* field instead of *src1*, and the lower 11 bits of *offset* are the lower 11 bits of the instruction.

For **ld** and **st**, bits 28 and zero determine operand size as follows:

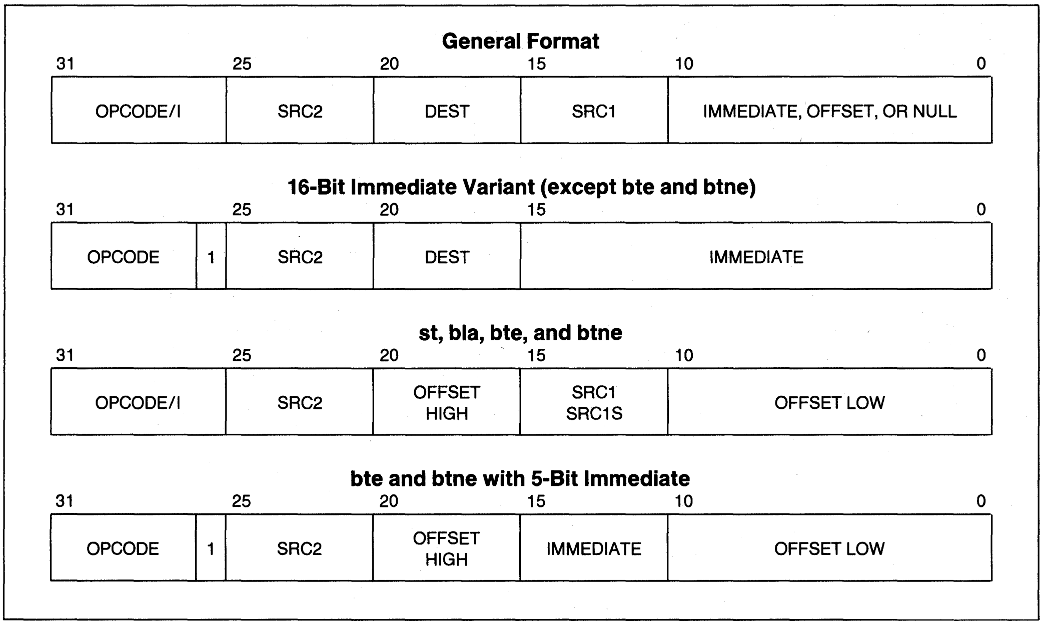
Bit 28	Bit 0	Operand Size
0	0	8-bits
0	1	8-bits
1	0	16-bits
1	1	32-bits

When *src1* is an immediate and bit 28 is set, bit zero of the immediate value is forced to zero.

For **fld**, **fst**, **pfld**, **pst**, and **flush**, bit 0 selects autoincrement addressing if set. Bits one and two select the operand size as follows:

Bit 1	Bit 2	Operand Size
0	0	64-bits
0	1	128-bits
1	0	32-bits
1	1	32-bits

When *src1* is an immediate value, bits zero and one of the immediate value are forced to zero to maintain alignment. When bit one of the immediate value is clear, bit two is also forced to zero.



8.2.2 REG-FORMAT OPCODES

		31					26
ld.x	Load Integer	0	0	0	L	0	I
st.x	Store Integer	0	0	0	L	1	1
ixfr	Integer to F-P Reg Transfer	0	0	0	0	1	0
	(reserved)	0	0	0	1	1	0
fld.x, fst.x	Load/Store F-P	0	0	1	0	LS	I
flush	Flush	0	0	1	1	0	1
pst.d	Pixel Store	0	0	1	1	1	1
ld.c, st.c	Load/Store Control Register	0	0	1	1	LS	0
bri	Branch Indirect	0	1	0	0	0	0
trap	Trap	0	1	0	0	0	1
	(Escape for F-P Unit)	0	1	0	0	1	0
	(Escape for Core Unit)	0	1	0	0	1	1
bte, btne	Branch Equal or Not Equal	0	1	0	1	E	I
pfld.y	Pipelined F-P Load	0	1	1	0	0	I
	(CTRL-Format Instructions)	0	1	1	x	x	x
addu, -s, subu, -s,	Add/Subtract	1	0	0	SO	AS	I
shl, shr	Logical Shift	1	0	1	0	LR	I
shrd	Double Shift	1	0	1	1	0	0
bla	Branch LCC Set and Add	1	0	1	1	0	1
shra	Arithmetic Shift	1	0	1	1	1	I
and(h)	AND	1	1	0	0	H	I
andnot(h)	ANDNOT	1	1	0	1	H	I
or(h)	OR	1	1	1	0	H	I
xor(h)	XOR	1	1	1	1	H	I
	(reserved)	1	1	x	x	1	0

L Integer Length
0 —8 bits
1 —16 or 32 bits (selected by bit 0)

LS Load/Store
0 —Load
1 —Store

SO Signed/Ordinal
0 —Ordinal
1 —Signed

H High
0 —**and, or, andnot, xor**
1 —**andh, orh, andnoth, xorh**

AS Add/Subtract
0 —Add
1 —Subtract

LR Left/Right
0 —Left Shift
1 —Right Shift

E Equal
0 —Branch on Not Equal
1 —Branch on Equal

I Immediate
0 —*src1* is register
1 —*src1* is immediate

8.2.3 CORE ESCAPE INSTRUCTIONS

31 26		15	10	5 0
0 1 0 0 1 1	reserved	SRC1	reserved	OPCODE

8.2.4 CORE ESCAPE OPCODES

		4				0
	(reserved)	0	0	0	0	0
lock	Begin Interlocked Sequence	0	0	0	0	1
calli	Indirect Subroutine Call	0	0	0	1	0
	(reserved)	0	0	0	1	1
intovr	Trap on Integer Overflow	0	0	1	0	0
	(reserved)	0	0	1	0	1
	(reserved)	0	0	1	1	0
unlock	End Interlocked Sequence	0	0	1	1	1
	(reserved)	0	1	x	x	x
	(reserved)	1	0	x	x	x
	(reserved)	1	1	x	x	x

8.2.5 CTRL-FORMAT INSTRUCTIONS

31			28	25 0
0	1	1	OPC	BROFFSET

BROFFSET is a signed 26-bit relative branch offset.

8.2.6 CTRL-FORMAT OPCODES

		28		26
br	Branch Direct	0	1	0
call	Call	0	1	1
bc(.t)	Branch on CC Set	1	0	T
bnc(.t)	Branch on CC Clear	1	1	T

T Taken
- 0 —**bc** or **bnc**
- 1 —**bc.t** or **bnc.t**

8.2.7 FLOATING-POINT INSTRUCTION ENCODING

31	25	20	15				7	0
0 1 0 0 1 0	SRC2	DEST	SRC1	P	D	S	R	OPCODE

SRC1, SRC2 —Source; one of 32 floating-point registers
DEST —Destination register
(instructions other than **fxfr**) one of 32 floating-point registers
(**fxfr**) one of 32 integer registers

P Pipelining
- 1 —Pipelined instruction mode
- 0 —Scalar instruction mode

D Dual-Instruction Mode
- 1 —Dual-instruction mode
- 0 —Single-instruction mode

S Source Precision
- 1 —Double-precision source operands
- 0 —Single-precision source operands

R Result Precision
- 1 —Double-precision result
- 0 —Single-precision result

8.2.8 FLOATING-POINT OPCODES

		6						0
pfam	Add and Multiply*							
pfmam	Multiply with Add*	0	0	0	DPC			
pfsm	Subtract and Multiply*	0	0	1	DPC			
pfmsm	Multiply with Subtract*							
(p)fmul	Multiply	0	1	0	0	0	0	0
fmlow	Multiply Low	0	1	0	0	0	0	1
frcp	Reciprocal	0	1	0	0	0	1	0
frsqr	Reciprocal Square Root	0	1	0	0	0	1	1
(p)fadd	Add	0	1	1	0	0	0	0
(p)fsub	Subtract	0	1	1	0	0	0	1
(p)fix	Fix	0	1	1	0	0	1	0
pfgt/pfle**	Greater Than	0	1	1	0	1	0	0
pfeq	Equal	0	1	1	0	1	0	1
(p)ftrunc	Truncate	0	1	1	1	0	1	0
fxfr	Transfer to Integer Register	1	0	0	0	0	0	0
(p)fiadd	Long-Integer Add	1	0	0	1	0	0	1
(p)fisub	Long-Integer Subtract	1	0	0	1	1	0	1
(p)fzchkl	Z-Check Long	1	0	1	0	1	1	1
(p)fzchks	Z-Check Short	1	0	1	1	1	1	1
(p)faddp	Add with Pixel Merge	1	0	1	0	0	0	0
(p)faddz	Add with Z Merge	1	0	1	0	0	0	1
(p)form	OR with MERGE Register	1	0	1	1	0	1	0

*pfam and pfsm have P-bit set; pfmam and pfmsm have P-bit clear.
__pfgt__ has R bit cleared; **pfle** has R bit set.

The following table shows the opcode mnemonics that generate the various encodings of DPC and explains each encoding.

Table 8.2. DPC Encoding

DPC	PFAM Mnemonic	PFSM Mnemonic	M-Unit op1	M-Unit op2	A-Unit op1	A-Unit op2	T Load	K Load*
0000	**r2p1**	**r2s1**	KR	src2	src1	M result	No	No
0001	**r2pt**	**r2st**	KR	src2	T	M result	No	Yes
0010	**r2ap1**	**r2as1**	KR	src2	src1	A result	Yes	No
0011	**r2apt**	**r2ast**	KR	src2	T	A result	Yes	Yes
0100	**i2p1**	**i2s1**	KI	src2	src1	M result	No	No
0101	**i2pt**	**i2st**	KI	src2	T	M result	No	Yes
0110	**i2ap1**	**i2as1**	KI	src2	src1	A result	Yes	No
0111	**i2apt**	**i2ast**	KI	src2	T	A result	Yes	Yes
1000	**rat1p2**	**rat1s2**	KR	A result	src1	src2	Yes	No
1001	**m12apm**	**m12asm**	src1	src2	A result	M result	No	No
1010	**ra1p2**	**ra1s2**	KR	A result	src1	src2	No	No
1011	**m12ttpa**	**m12ttsa**	src1	src2	T	A result	Yes	No
1100	**iat1p2**	**iat1s2**	KI	A result	src1	src2	Yes	No
1101	**m12tpm**	**m12tsm**	src1	src2	T	M result	No	No
1110	**ia1p2**	**ia1s2**	KI	A result	src1	src2	No	No
1111	**m12tpa**	**m12tsa**	src1	src2	T	A result	No	No

DPC	PFMAM Mnemonic	PFMSM Mnemonic	M-Unit op1	M-Unit op2	A-Unit op1	A-Unit op2	T Load	K Load*
0000	**mr2p1**	**mr2s1**	KR	src2	src1	M result	No	No
0001	**mr2pt**	**mr2st**	KR	src2	T	M result	No	Yes
0010	**mr2mp1**	**mr2ms1**	KR	src2	src1	M result	Yes	No
0011	**mr2mpt**	**mr2mst**	KR	src2	T	M result	Yes	Yes
0100	**mi2p1**	**mi2s1**	KI	src2	src1	M result	No	No
0101	**mi2pt**	**mi2st**	KI	src2	T	M result	No	Yes
0110	**mi2mp1**	**mi2ms1**	KI	src2	src1	M result	Yes	No
0111	**mi2mpt**	**mi2mst**	KI	src2	T	M result	Yes	Yes
1000	**mrmt1p2**	**mrmt1s2**	KR	M result	src1	src2	Yes	No
1001	**mm12mpm**	**mm12msm**	src1	src2	M result	M result	No	No
1010	**mrm1p2**	**mrm1s2**	KR	M result	src1	src2	No	No
1011	**mm12ttpm**	**mm12ttsm**	src1	src2	T	A result	Yes	No
1100	**mimt1p2**	**mimt1s2**	KI	M result	src1	src2	Yes	No
1101	**mm12tpm**	**mm12tsm**	src1	src2	T	M result	No	No
1110	**mim1p2**	**mim1s2**	KI	M result	src1	src2	No	No
1111	**mm12tpm**	**mm12tsm**	src1	src2	T	M result	No	No

*If K-load is set, KR is loaded when operand-1 of the multiplier is KR; KI is loaded when operand-1 of the multiplier is KI.

8.3 Instruction Timings

860 microprocessor instructions take one clock to execute unless a freeze condition is invoked. Freeze conditions and their associated delays are shown in the table below. Freezes due to multiple simultaneous cache misses result in a delay that is the sum of the delays for processing each miss by itself. Other multiple freeze conditions usually add only the delay of the longest individual freeze.

Freeze Condition	Delay
Instruction-cache miss	Number of clocks to read instruction (from ADS clock to first READY# clock) plus time to last READY# of block when jump or freeze occurs during miss processing plus two clocks if data-cache being accessed when instruction-cache miss occurs.
Reference to destination of load instruction that misses	One plus number of clocks to read data (from ADS# clock to first READY# clock) minus number of instructions executed since load (not counting instruction that references load destination)
fld miss	One plus number of clocks from ADS# to first READY#
call/calli/ixfr/fxfr/ld.c/st.c and data cache miss processing in progress	One plus number of clocks until first READY# returned
ld/st/pfld/fld/fst and data cache miss processing in progress	One plus number of clocks until last READY# returned
Reference to *dest* of **ld, call, calli, fxfr,** or **ld.c** in the next instruction	One clock

Freeze Condition	Delay
Reference to *dest* of **fld/pfld/ixfr** in the next two instructions	Two clocks in the first instruction; one in the second instruction
bc/bnc/bc.t/bnc.t following **fadd/fsub/pfeg/pfgt**	One clock
Src1 of multiplier operation refers to result of previous operation	One clock
Floating-point operation or **fst** and scalar operation in progress other than **frcp** or **frsqr**	If the scalar operation is **fadd, fix, fmlow, fmul.ss, fmul.sd, ftrunc,** or **fsub,** three minus the number of instructions executed after the scalar operation. If the scalar operation is **fmul.dd,** four minus the number of instructions executed after it. Add one if the precision of the result of the previous scalar operation is different than that of the source. Add one if the floating-point operation is pipelined and its destination is not **f0.** If the sum of the above terms is negative, there is no delay.
Multiplier operation preceded by a double-precision multiply	One clock
TLB miss	Five plus the number of clocks to finish two reads plus the number of clocks to set A-bits (if necessary)
pfld when three **pfld**'s are outstanding	One plus the number of clocks to return data from first **pfld**
pfld hits in the data cache	Two plus the number of clocks to finish all outstanding accesses
Store pipe full (two internal plus outstanding bus cycles) and **st/fst** miss, **ld** miss, or **flush** with modified block	One plus the number of clocks until READY# active on next write data
Address pipe full (one internal plus outstanding bus cycles) and **ld/fld/plfd/st/fst**	Number of clocks until next address can be issued
ld/fld following **st/fst** hit	One clock
Delayed branch not taken	One clock
Nondelayed branch taken	One clock
Branch indirect **br**	One clock
st.c	Two clocks
Result of graphics-unit instruction (other than **fmov**) used in next instruction when the next instruction is an adder- or multiplier-unit instruction	One clock
Result of graphics-unit instruction used in next instruction when the next instruction is a graphics-unit instruction	One clock
flush followed by **flush**	Two clocks
fst followed by pipelined floating-point operation that overwrites the register being stored	One clock

8.4 Instruction Characteristics

The following table lists some of the characteristics of each instruction. The characteristics are:

- What processing unit executes the instruction. The codes for processing units are:

A	Floating-point adder unit
E	Core execution unit
G	Graphics (vector-integer) unit
M	Floating-point multiplier unit

- Whether the instruction is pipelined or not. A *P* indicates that the instruction is pipelined.
- Whether the instruction is a delayed branch instruction. A *D* marks the delayed branches.
- Whether the instruction changes the condition code CC. A *CC* marks those instructions that change CC.
- Which faults can be caused by the instruction. The codes used for exceptions are:

IT	Instruction Fault
SE	Floating-Point Source Exception
RE	Floating-Point Result Exception, including overflow, underflow, inexact result
DAT	Data Access Fault

  The instruction access fault IAT and the interrupt trap IN are not shown in the table because they can occur for any instruction.

- Performance notes. These comments regarding optimum performance are recommendations only. If these recommendations are not followed, the 860 microprocessor automatically waits the necessary number of clocks to satisfy internal hardware requirements. The following notes define the numeric codes that appear in the instruction table:

  1. The following instruction should not be a conditional branch (**bc**, **bnc**, **bc.t**, or **bnc.t**).
  2. The destination should not be a source operand of the next two instructions.
  3. A load should not directly follow a store that is expected to hit in the data cache.
  4. When the prior instruction is scalar, *src1* should not be the same as the *rdest* of the prior operation.
  5. The *freg* should not reference the destination of the next instruction if that instruction is a pipelined floating-point operation.
  6. The destination should not be a source operand of the next instruction.
  7. When the prior operation is scalar and multiplier *op1* is *src1*, *src2* should not be the same as the *rdest* of the prior operation.
  8. When the prior operation is scalar, *src1* and *src2* of the current operation should not be the same as *rdest* of the prior operation.

- Programming restrictions. These indicate combinations of conditions that must be avoided by programmers, assemblers, and compilers. The following notes define the alphabetic codes that appear in the instruction table:

  a. The sequential instruction following a delayed control-transfer instruction may not be another control-transfer instruction (except in the case of external interrupts), nor a trap instruction, nor the target of a control-transfer instruction.

  b. When using a **bri** to return from a trap handler, programmers should take care to prevent traps from occurring on that or on the next sequential instruction. IM should be zero (interrupts disabled) when the **bri** is executed.

  c. If *rdest is not zero, src1* must not be the same as *rdest.*

  d. When the multiplier *op1* is *src1, src1* must not be the same as *rdest*.

  e. If *rdest is not zero, src1* and *src2* must not be the same as *rdest.*

Instruction	Execution Unit	Pipelined? Delayed?	Sets CC?	Faults	Performance Notes	Programming Restrictions
adds	E		CC		1	
addu	E		CC		1	
and	E		CC			
andh	E		CC			
andnot	E		CC			
andnoth	E		CC			
bc	E					
bc.t	E	D				a
bla	E	D				a
bnc	E					
bnc.t	E	D				a
br	E	D				a
bri	E	D				a, b
bte	E					
btne	E					
call	E	D			2	a
calli	E	D			2	a
fadd.p	A			SE, RE		
faddp	G				8	
faddz	G				8	
fiadd.z	G				8	
fisub.z	G				8	
fix.p	A			SE, RE		
fld.y	E			DAT	2, 3	
flush	E					
fmlow.p	M				4	
fmul.p	M			SE, RE	4	
form	G				8	
frep.p	M			SE, RE		
frsqr.p	M			SE, RE		
fst.y	E			DAT	5	
fsub.p	A			SE, RE		
ftrunc.p	A			SE, RE		
fxfr	G				6, 8	
fzchkl	G				8	
fzchks	G				8	
intovr	E			IT		
ixfr	E				2	
ld.c	E					
ld.x	E			DAT	6	
or	E		CC			
orh	E		CC			

Instruction	Execution Unit	Pipelined? Delayed?	Sets CC?	Faults	Performance Notes	Programming Restrictions
pfadd.p	A	P		SE, RE		
pfaddp	G	P			8	e
pfaddz	G	P			8	e
pfam.p	A&M	P		SE, RE	7	d
pfeq.p	A	P	CC	SE	1	
pfgt.p	A	P	CC	SE	1	
pfiadd.z	G	P			8	e
pfisub.z	G	P			8	e
pfix.p	A	P		SE, RE		
pfld.z	E	P			2	
pfmul.p	M	P		SE, RE	4	c
pform	G	P			8	e
pfsm.p	A&M	P		SE, RE	7	d
pfsub.p	A	P		SE, RE		
pftrunc.p	A	P		SE, RE		
pfzchkl	G	P			8	
pfzchks	G	P			8	
pst.d	E			DAT		
shl	E					
shr	E					
shra	E					
shrd	E					
st.c	E					
st.x	E			DAT		
subs	E		CC		1	
subu	E		CC		1	
trap	E			IT		
xor	E		CC			
xorh	E		CC			

DOMESTIC SALES OFFICES

ALABAMA

†Intel Corp.
5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

†Intel Corp.
11225 N. 28th Dr.
Suite D-214
Phoenix 85029
Tel: (602) 869-4980

†Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

†Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

†Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

†Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

†Intel Corp.
9665 Chesapeake Dr.
Suite 325
San Diego 95123
Tel: (619) 292-8086

†Intel Corp.*
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

†Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255
FAX: 408-727-2620

COLORADO

†Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (719) 594-6622

†Intel Corp.*
650 S. Cherry St., Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

†Intel Corp.
26 Mill Plain Road
2nd Floor
Danbury 06811
Tel: (203) 748-3130
TWX: 710-456-1199

FLORIDA

†Intel Corp.
6363 N.W. 6th Way, Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407
FAX: 305-772-8193

†Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (407) 240-8000
FAX: 407-240-8097

†Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33716
Tel: (813) 577-2413
FAX: 813-578-1607

GEORGIA

†Intel Corp.
20 Technology Parkway, N.W.
Suite 150
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

†Intel Corp.*
300 N. Martingale Road, Suite 400
Schaumburg 60173
Tel: (312) 605-8031
FAX: 312-605-9762

INDIANA

†Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
1930 St. Andrews Drive N.E.
2nd Floor
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

†Intel Corp.
10985 Cody St.
Suite 140, Bldg. D
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

†Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

†Intel Corp.
7833 Walker Drive
Suite 550
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

†Intel Corp.*
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (508) 692-3222
TWX: 710-343-6333

MICHIGAN

†Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8096

MINNESOTA

†Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

†Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

†Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

†Intel Corp.
280 Corporate Center
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111
FAX: 201-740-0626

NEW MEXICO

†Intel Corp.
8500 Menaul Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

†Intel Corp.*
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

†Intel Corp.*
2950 Expressway Dr., South
Suite 130
Islandia 11722
Tel: (516) 231-3300
TWX: 510-227-6236

†Intel Corp.
Westage Business Center
Bldg. 300, Route 9
Fishkill 12524
Tel: (914) 897-3860
FAX: 914-897-3125

NORTH CAROLINA

†Intel Corp.
5800 Executive Center Dr.
Suite 105
Charlotte 28212
Tel: (704) 568-8966
FAX: 704-535-2236

†Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

†Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

†Intel Corp.*
25700 Science Park Dr., Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

†Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086

OREGON

†Intel Corp.
15254 N.W. Greenbrier Parkway
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

PENNSYLVANIA

†Intel Corp.*
455 Pennsylvania Avenue
Suite 230
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Blvd., Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

†Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

†Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

†Intel Corp.*
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: 214-484-1180

†Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

UTAH

†Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051

VIRGINIA

†Intel Corp.
1504 Santa Rosa Road
Suite 108
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

†Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

†Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

†Intel Corp.
330 S. Executive Dr.
Suite 102
Brookfield 53005
Tel: (414) 784-8087
FAX: (414) 796-2115

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
Tel: (604) 298-0387
FAX: (604) 298-8234

ONTARIO

†Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TLX: 053-4115

†Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TLX: 06983574
FAX: (416) 675-2438

QUEBEC

†Intel Semiconductor of Canada, Ltd.
620 St. John Boulevard
Pointe Claire H9R 3K2
Tel: (514) 694-9130
TWX: 514-694-9134

†Sales and Service Office
*Field Application Location

intel

DOMESTIC SERVICE OFFICES

ALABAMA

Intel Corp.
5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Dr.
Suite D-214
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
500 E. Fry Blvd., Suite M-15
Sierra Vista 85635
Tel: (602) 459-5010

Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

Intel Corp.
1900 Prairie City Rd.
Folsom 95630-9597
Tel: (916) 351-6143

Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

Intel Corp.
4350 Executive Drive
Suite 105
San Diego 92121
Tel: (619) 452-5880

Intel Corp.*
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255

COLORADO

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (303) 594-6622

Intel Corp.*
650 S. Cherry St., Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
2nd Floor
Danbury 06811
Tel: (203) 748-3130
TWX: 710-456-1199

FLORIDA

Intel Corp.
6363 N.W. 6th Way
Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407
FAX: 305-772-8193

Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (305) 240-8000
FAX: 305-240-8097

Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33716
Tel: (813) 577-2413
FAX: 813-578-1607

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

Intel Corp.*
300 N. Martingale Road
Suite 400
Schaumburg 60173
Tel: (312) 310-8031

INDIANA

Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
1930 St. Andrews Drive N.E.
2nd Floor
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Intel Corp.
7833 Walker Drive
Suite 550
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

Intel Corp.*
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (508) 692-3222
TWX: 710-343-6333

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8096

MINNESOTA

Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

Intel Corp.
Raritan Plaza III
Raritan Center
Edison 08817
Tel: (201) 225-3000

Intel Corp.
385 Sylvan Avenue
Englewood Cliffs 07632
Tel: (201) 567-0821
TWX: 710-991-8593

Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

†Intel Corp.
280 Corporate Center
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111
FAX: 201-740-0626

NEW MEXICO

Intel Corp.
8500 Menaul Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.
127 Main Street
Binghamton 13905
Tel: (607) 773-0337
FAX: 607-723-2677

Intel Corp.*
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

Intel Corp.*
300 Motor Parkway
Hauppauge 11787
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
Westage Business Center
Bldg. 300, Route 9
Fishkill 12524
Tel: (914) 897-3860
FAX: 914-897-3125

NORTH CAROLINA

Intel Corp.
5700 Executive Drive
Suite 213
Charlotte 28212
Tel: (704) 568-8966

Intel Corp.
2306 W. Meadowview Road
Suite 206
Greensboro 27407
Tel: (919) 294-1541

Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
25700 Science Park Dr.
Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086

OREGON

Intel Corp.
15254 N.W. Greenbrier Parkway
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

Intel Corp.
5200 N.E. Elam Young Parkway
Hillsboro 97123
Tel: (503) 681-8080

PENNSYLVANIA

Intel Corp.*
455 Pennsylvania Avenue
Suite 230
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Blvd.
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

TEXAS (Cont'd.)

Intel Corp.*
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: 214-484-1180

Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

UTAH

Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051

VIRGINIA

Intel Corp.
1504 Santa Rosa Road
Suite 108
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

Intel Corp.
330 S. Executive Dr.
Suite 102
Brookfield 53005
Tel: (414) 784-8087
FAX: (414) 796-2115

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
Tel: (604) 298-0387
FAX: (604) 298-8234

ONTARIO

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TLX: 053-4115

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TLX: 06983574
FAX: (416) 675-2438

QUEBEC

Intel Semiconductor of Canada, Ltd.
620 St. John Boulevard
Pointe Claire H9R 3K2
Tel: (514) 694-9130
TWX: 514-694-9134

CUSTOMER TRAINING CENTERS

CALIFORNIA

2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 970-1700

ILLINOIS

300 N. Martingale, #300
Schaumburg 60173
Tel: (312) 310-5700

MASSACHUSETTS

3 Carlisle Road
Westford 01886
Tel: (508) 692-1000

MARYLAND

7833 Walker Dr., 4th Floor
Greenbelt 20770
Tel: (301) 220-3380

SYSTEMS ENGINEERING OFFICES

CALIFORNIA

2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 986-8086

ILLINOIS

300 N. Martingale, #300
Schaumburg 60173
Tel: (312) 310-8031

NEW YORK

300 Motor Parkway
Hauppauge 11788
Tel: (516) 231-3300

UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison, BP 303
78054 Saint-Quentin-en-Yvelines Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8

Order Number: 240296-001

Printed in U.S.A./SMD405/5K/0289/ML RJ
Microprocessors