i860™ XR 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**
  - **—25/33.3/40 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**
  - **—Intel386™/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™ XR Microprocessor (order codes A80860XR-25, A80860XR-33 and A80860XR-40) delivers supercomputing performance in a single VLSI component. The 64-bit design of the i860 XR 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.

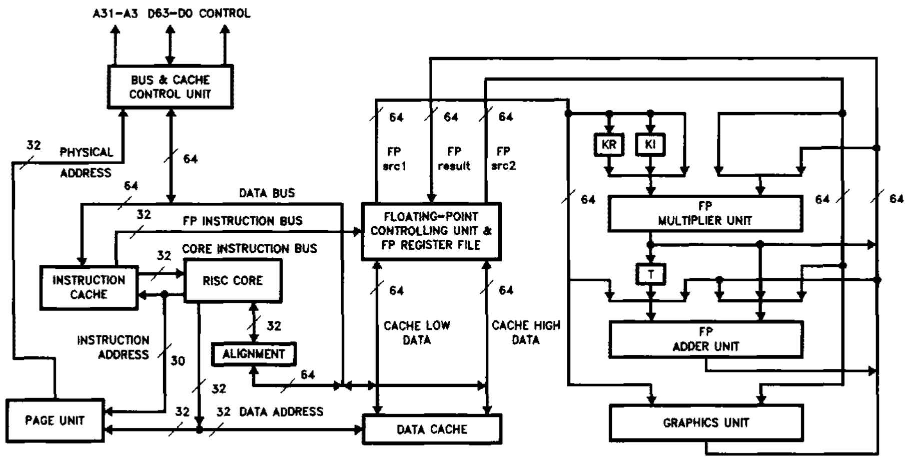


Figure 0.1. Block Diagram

October 1992
Order Number: 240296-006

i860™ XR 64-Bit Microprocessor

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

1.0 FUNCTIONAL DESCRIPTION

As shown by the block diagram on the front page, the i860 XR 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 i860 XR 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 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 i860 XR 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 i860 XR 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 (320 Mbyte/sec at 40 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 (640 Mbyte/sec at 40 MHz). The i860 XR 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 i860 XR microprocessor include data types, registers, instructions, and traps.

2.1 Data Types

The i860 XR 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 2,147,483,647 ($+2^{31}-1$). 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.

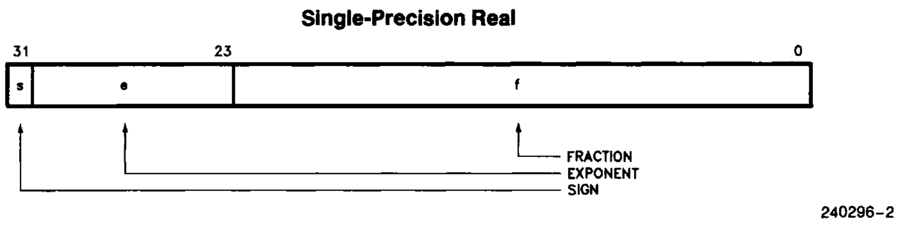
2.1.3 SINGLE- AND DOUBLE-PRECISION REAL

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 $(-1)^s \times 1.f \times 2^{e-1023}$.



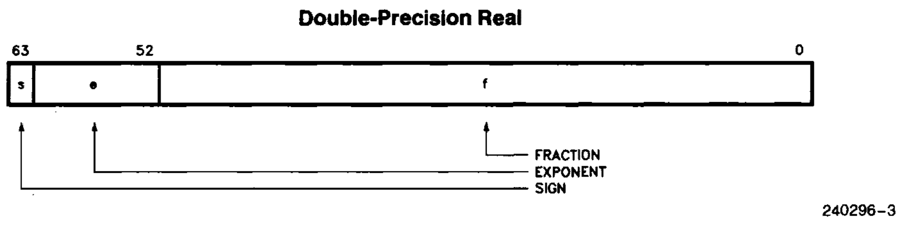


Figure 2.1. Real Number Formats

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 i860 XR 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 i860 XR 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 i860 XR 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.

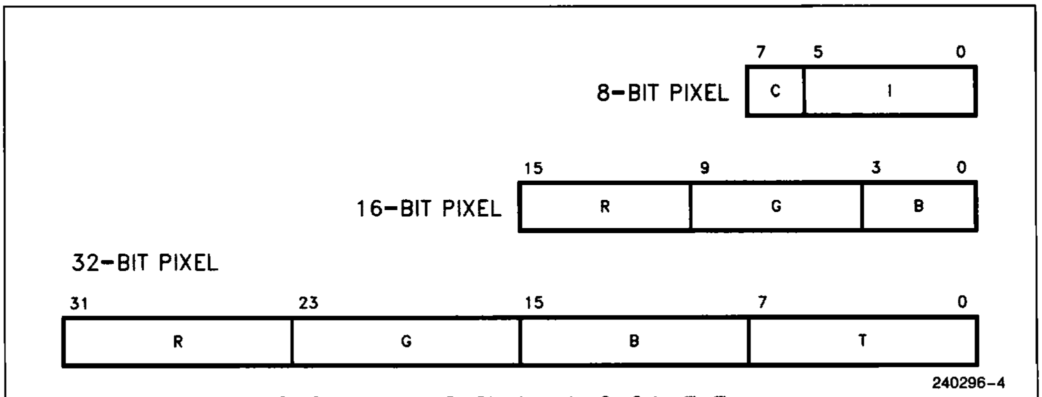
*With 8-bit pixels, up to 8 bits can be used for intensity; the remaining bits can be used for any other attribute, such as color. The intensity bits must be the low-order bits of the pixel.

2.2 Register Set

As Figure 2.3 shows, the i860 XR 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 graphics operations, primarily for 3D graphics computations.

When accessing 64-bit floating-point or integer values, the i860 XR 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 numbers contain the least significant 64 bits while the register pair with the higher numbers contain the most significant 64 bits.

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 the 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 i860 XR microprocessor is executing in user mode; it is clear when the i860 XR microprocessor is executing in supervisor mode. In user mode, writes to some control registers are inhibited. This bit also controls the memory protection mechanism. See section 2.4.4.3 for a description of memory protection in user and supervisor modes.

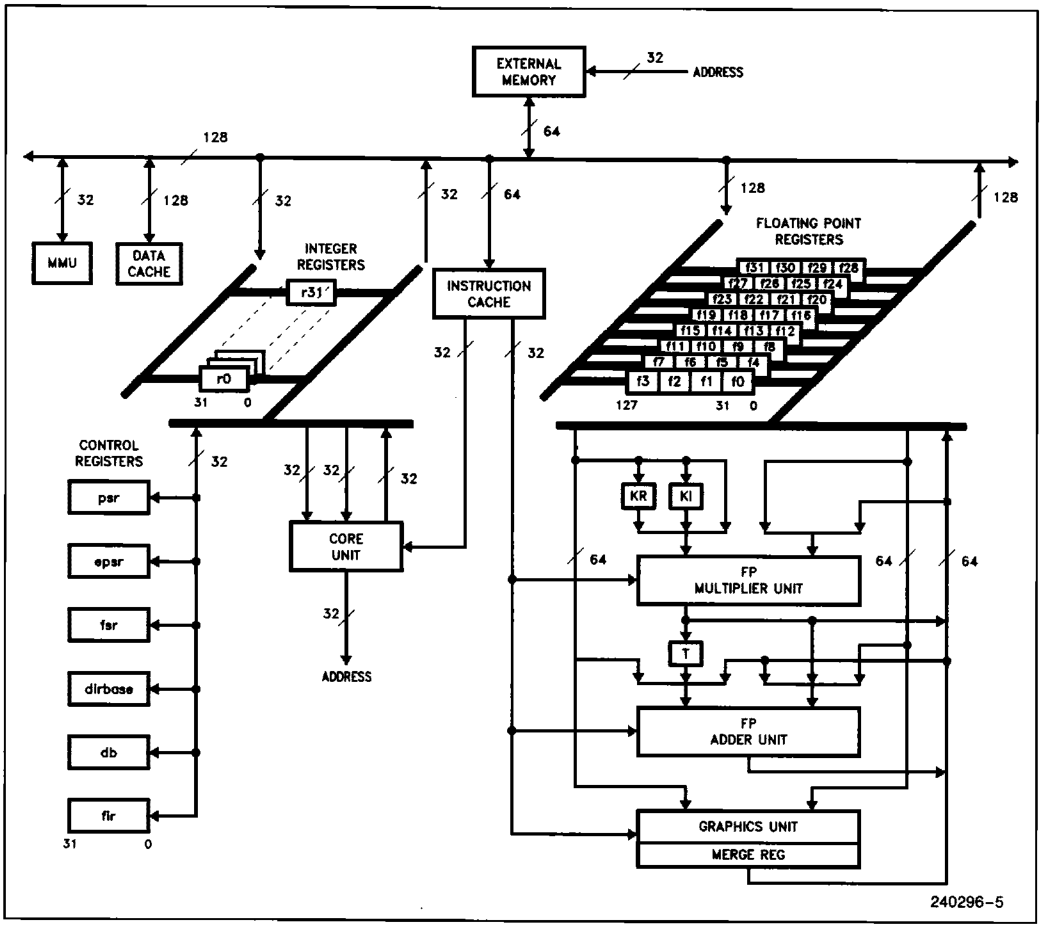


Figure 2.3. Registers and Data Paths

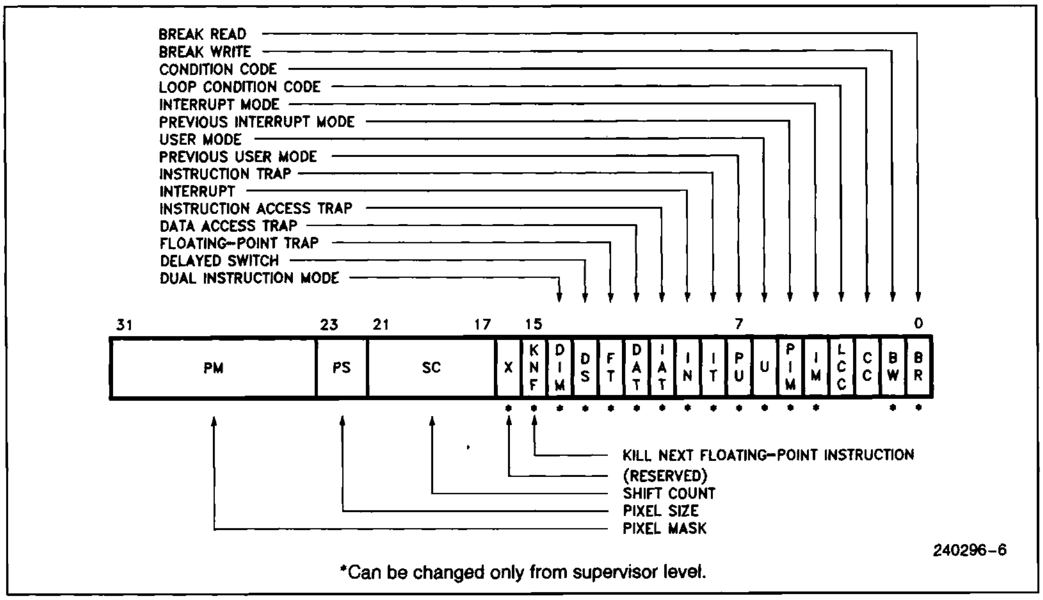


Figure 2.4 Processor Status Register

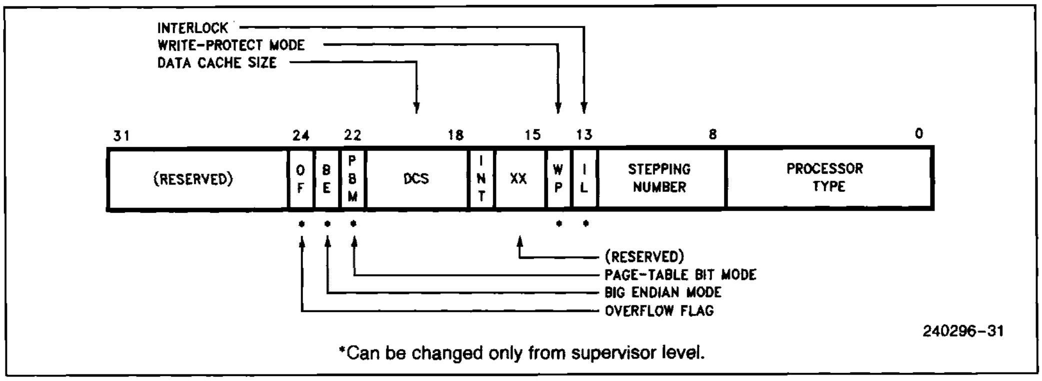


Figure 2.5 Extended Processor Status Register

- 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 i860 XR microprocessor switches to dual-instruction mode one instruction after returning from the trap handler. If DS and DIM are both set, the i860 XR microprocessor switches to single-instruction mode one instruction after returning from the trap handler.
- When a trap occurs, the i860 XR 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 i860 XR 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 graphics 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 i860 XR 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 the load or store following the subsequent **unlock** instruction. IL indicates to the trap handler that a locked sequence has been interrupted. 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 30–33 instructions of the trap indicates a programming error.
- 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. After the value of the WP bit is changed, the TLB must be invalidated by setting the ITI bit of the **dirbase** register, before any stores are performed.
- INT (Interrupt) is the value of the INT input pin.
- 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+DCS}; therefore, a value of zero indicates 4 Kbytes, one indicates 8 Kbytes, etc.

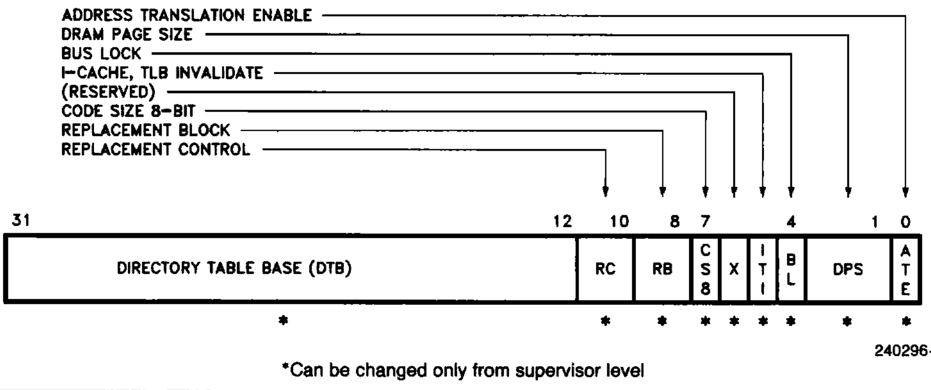


Figure 2.6. Directory Base Register

- 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 i860 XR microprocessor operates in little endian mode, in which the addressed byte is the low-order byte. When BE is set (big 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. Section 2.3 discusses little and big endian addressing.
- 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. OF cannot be written in user mode using ST.C.

2.2.5 DATA BREAKPOINT REGISTER

The data breakpoint register (**db**) is used to generate a trap when the i860 XR microprocessor makes a data-operand access to the address stored in this register. The trap is enabled by BR and BW in **psr**. The **db** register can only be changed from supervisor level. 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. The DAT occurs before the data is accessed and prevents the load or store from completing.

2.2.6 DIRECTORY BASE REGISTER

The directory base register **dirbase** (shown in Figure 2.6) controls address translation, caching, and bus options. The **dirbase** register can only be changed from supervisor level. The BL bit is changed from user level with the **lock** and **unlock** instructions.

- 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. For interleaved memory, increase DPS by one for each power of interleaving—add one for 2-way, and two for 4-way, 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 load or store instruction after BL is cleared. Traps immediately clear BL. The **lock** and **unlock** instructions control the BL bit. The result of modifying BL with the **st.c** instruction is not defined.
- ITI (I-Cache, TLB Invalidate), when set in the value that is loaded into **dirbase**, causes all entries in the instruction cache and address-translation cache (TLB) to be invalidated. The ITI bit does not remain set in **dirbase**. ITI always appears as zero when reading **dirbase**. Section 2.5 discusses flushing the data cache 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 instruction 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.

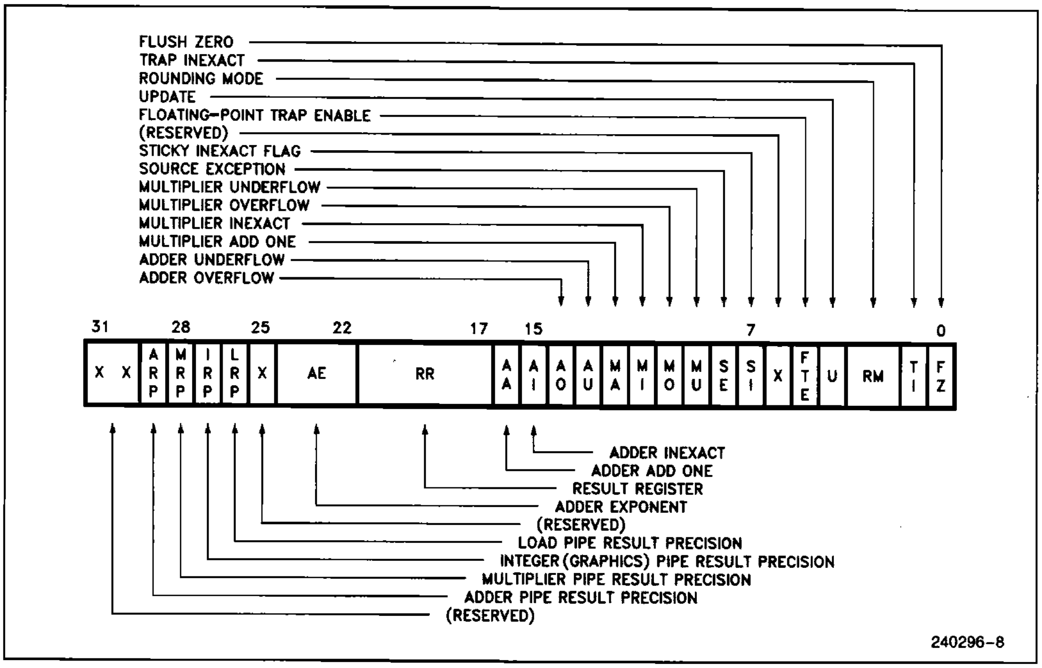


Figure 2.7. Floating-Point Status Register

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 algorithm 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. Instruction and TLB caches use random replacement.
11	Disables data cache replacement. Instruction and TLB caches use random 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). The **fir** is a read-only register. In single-instruction mode, using a **ld.c** instruction to read the **fir** anytime except the first time after a trap saves in *idest* the address of the **ld.c** instruction; in dual-instruction mode, the address of its floating-point companion (address of the **ld.c** − 4) is saved.

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 2.7 shows its format. The **fsr** is writable in user level.

- 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

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 − ∞)	*a*
10	Round up (toward + ∞	*c*
11	Chop (toward zero)	Smaller in magnitude of *a* or *c*.

by the target data type, the i860 XR microprocessor determines the two representable numbers *a* and *c* that most closely bracket *b* in value ($a < b < c$). The i860 XR 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. A **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.

  The Adder Add One and Multiplier Add One bits indicate that the absolute value of the result fraction grew by one least-significant bit due to rounding. AA and MA are not influenced by the sign of the result.

  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 written 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 (Integer (Graphics) 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, pfsm,** 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 Figure 2.14.)

The MERGE register is used only by the graphics 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 are 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, the ordering can be dynamically selected by software in supervisor mode. The i860 XR microprocessor also offers big endian mode, in which the most significant byte of a data item is at the lowest address. 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 low-order byte is at the lowest memory address.

Code accesses are always done with little endian addressing. This implies that code will appear differently than documented here when accessed as big endian data. Intel recommends that disassemblers running in a big endian system, convert instructions which have been read as data back to little endian form and present them in the format documented here.

Page directories and page tables are also accessed in little endian mode, regardless of the value of the BE bit.

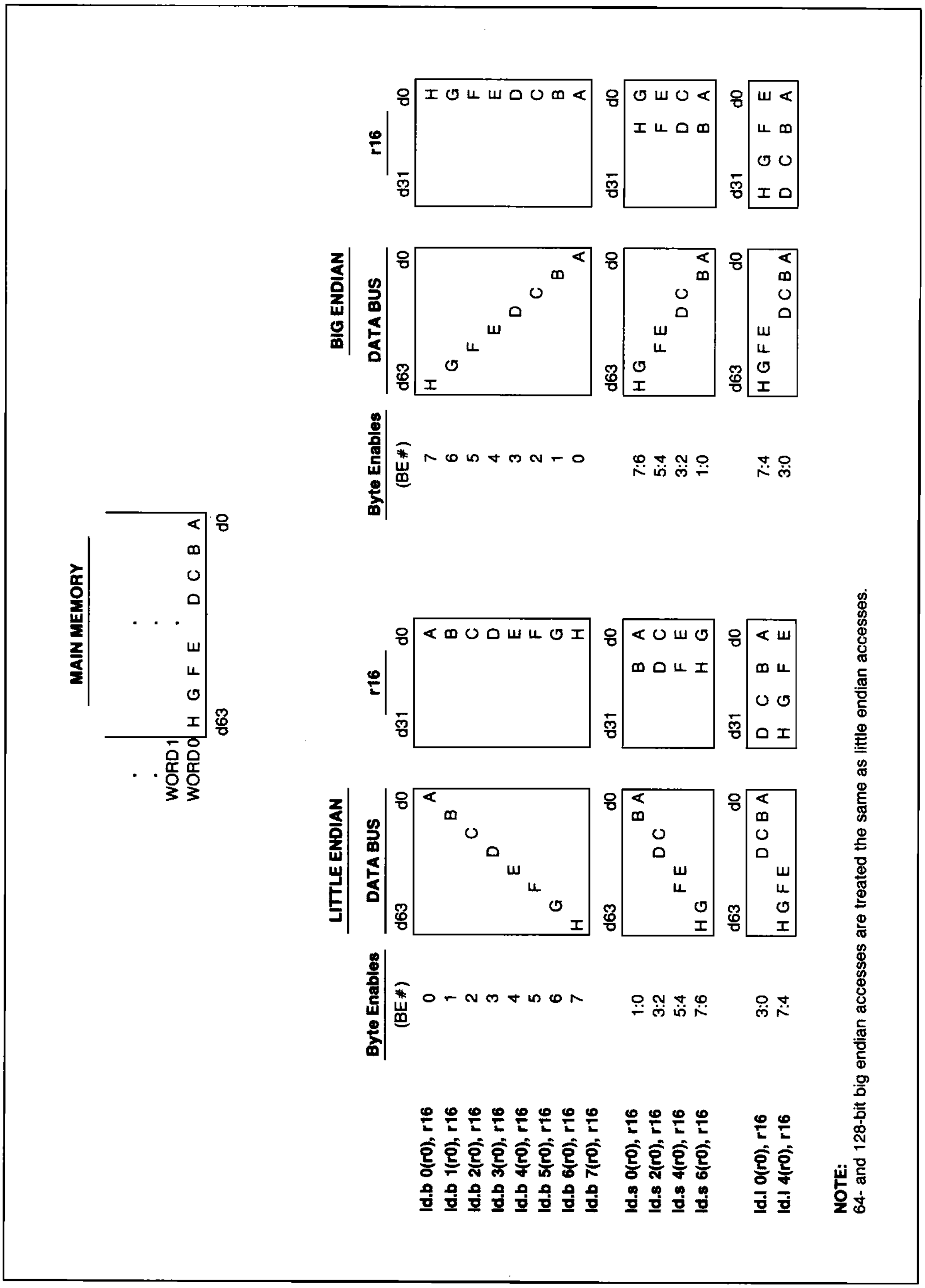
Alignment requirements are as follows (any violation results in a data-access trap):

- 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 i860 XR microprocessor maps instruction and data virtual addresses into physical addresses before referencing memory. This address transformation is compatible with that of the Intel386™ 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.



NOTE:
64- and 128-bit big endian accesses are treated the same as little endian accesses.

Figure 2.8 Little and Big Endian Accesses

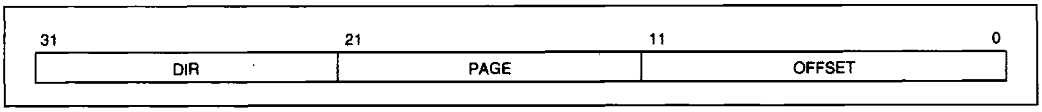


Figure 2.9. Format of a Virtual Address

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. The data may also occupy 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 2.10 shows how the i860 XR 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.

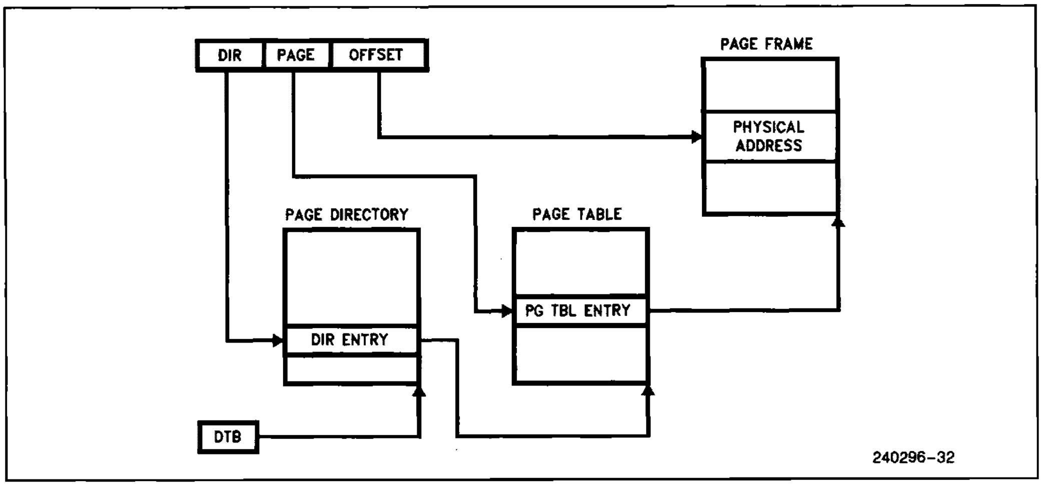


Figure 2.10. Address Translation

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 i860 XR 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.

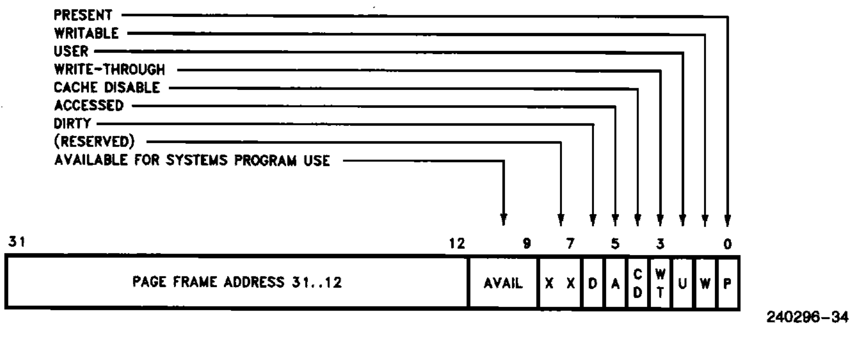
Note that there is no P bit for the page directory itself. The page directory may be not-present while the associated process is suspended, but the operating system must ensure that the page directory indicated by the **dirbase** image associated with the process is present in physical memory before 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 i860 XR 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.

The U bit of the **psr** indicates whether the i860 XR microprocessor is executing at user or supervisor level. The i860 XR microprocessor maintains the U bit of **psr** as follows:



NOTE:
X indicates Intel reserved. Do not use.

Figure 2.11. Format of a Page Table Entry

- The i860 XR 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 of U is copied into PU.
- The i860 XR 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 i860 XR microprocessor enters user level.

With the U bit of **psr** and the W and U bits of the page table entries, the i860 XR microprocessor implements the following protection rules:

- When at user level, a read or write of a supervisor-level page causes a trap.
- When at user level, a write to a page whose W bit is clear causes a trap.
- When at user level, **st.c** to certain control registers is ignored.

When the i860 XR 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 i860 XR 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 a page whose W bit is clear causes a trap.

When the i860 XR 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 i860 XR microprocessor does not implement a write-through caching policy for the on-chip data cache; 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. The i860 XR CPU may place pages having WT = 1 into the instruction cache. Future implementations of the i860 XR architecture may adhere to a write-through data caching policy. Therefore, they may cache pages having the WT bit of the PTE set. 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 instruction or data caches. Clearing CD permits the cache hardware to place data from the associated page into caches. The CD bit of page directory entries is not referenced by the processor, but is **reserved.**

To control external caches, the i860 XR microprocessor outputs on its PTB pin either the CD or WT bit. The PBM bit of **epsr** determines which bit is output.

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 i860 XR 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 then has the opportunity to maintain appropriate values in the dirty bits. The dirty bit in directory entries is not tested by the i860 XR microprocessor. The precise algorithm for using these bits is specified in Section 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 i860 XR microprocessor automatically asserts the LOCK# signal while setting the A bit. If an A-bit of a PTE is found not set during a locked sequence (created by the **lock** instruction), a trap will occur and the processor will not update 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 i860 XR 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 directory table base address stored in the **dirbase** register.

1. Read the PTE (page table entry) at the physical address formed by DTB:DIR:00.
2. If P in the PTE is zero, generate a data- or instruction-access fault.
3. 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.
4. 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.
5. If A in the PTE is zero, and if the TLB miss occurred while the bus was locked, generate a data or instruction access fault. (The trap allows software to set A to one and restart the sequence. This avoids ambiguity in determining what address corresponds to a locked semaphore for external bus hardware use.)
6. If A in the PTE is zero, and if the TLB miss occurred while the bus was not locked, assert LOCK#. Re-fetch and check the PTE, set A, and store the PTE. Deassert LOCK# during the store.
7. Locate the PTE at the physical address formed by PFA1:PAGE:00.
8. Perform the P, W, U, and A checks as in steps 2 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.

The i860 XR microprocessor looks only in external memory for Page Directories and Page Tables, in the translation process. The data cache is not searched. Therefore, any code which modifies Page Directories or Page Tables must keep them out of the cache. The tables should be kept in non-cacheable memory, or flushed from the cache.

Table 2.6. Combining Directory and Page Protections

Page Directory Entry		Page Table Entry		Combined Protection		
				User Access	Supervisor Access	
U-bit	W-bit	U-bit	W-bit	WP = X	WP = 0	WP = 1
0	0	0	0	N	R/W	R
0	0	0	1	N	R/W	R
0	0	1	0	N	R/W	R
0	0	1	1	N	R/W	R
0	1	0	0	N	R/W	R
0	1	0	1	N	R/W	R/W
0	1	1	0	N	R/W	R
0	1	1	1	N	R/W	R/W
1	0	0	0	N	R/W	R
1	0	0	1	N	R/W	R
1	0	1	0	R	R/W	R
1	0	1	1	R	R/W	R
1	1	0	0	N	R/W	R
1	1	0	1	N	R/W	R/W
1	1	1	0	R	R/W	R
1	1	1	1	R/W	R/W	R/W

NOTES:

N = No access allowed	R/W = Both reads and writes allowed
R = Read access only	X = Don't care

The i860 XR microprocessor expects Page Directories and Page Tables to be in little endian format. The operating system must maintain these tables in little endian format by either setting BE = 0 when manipulating the tables or by complementing bit 2 of the address when loading or storing entries.

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 i860 XR 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 i860 XR microprocessor has the ability to cache instruction, data, and address-translation information in on-chip caches. Caching uses virtual-address tags. The effects of mapping two different virtual addresses in the same address space to the same physical address are undefined.

Instruction, data, and address-translation caching on the i860 XR microprocessor are not transparent. Because the data cache uses a write-back protocol, writes do not immediately update memory, and writes to memory by other bus devices do not update the cache. Changes to page tables do not automatically update the TLB, and changes to instructions do not automatically update the instruction cache. 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. The i860 XR 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 of the associated second-level PTE is set, caching of data and instructions is disabled. The i860 XR CPU may place pages having WT = 1 into the instruction cache. Future implementations of the i860 XR architecture may adhere to a write-through data cache policy. Thus, they may cache pages having the WT bit of the PTE set. The value of the CD bit or the WT bit is output on the PTB pin for use by external caches.
- **Invalidating 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 modifying a page table, when modifying a page containing instructions, or when changing the DTB field of **dirbase** or the WP bit of the **epsr**. 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 containing the currently executing instruction and the next six instructions should not be different in the new page tables when **st.c dirbase** changes DTB or activates ITI. The six instructions following the **st.c** should be **nops** and should lie in the same page as the **st.c**.
- **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 invalidating the instruction or address-translation caches (as controlled by the ITI bit of **dirbase**) or enabling or disabling address translation (via the ATE bit). The data cache does not need flushing if the program is modifying only the P, U, W, A, or D bits of a PTE (as long as the Page Frame Address is not changed and the PTE itself was not in the data cache.) The i860 XR CPU does not check these protection bits on cache line write-back. Thus, a trap handler can service a DAT for D-bit-zero by setting D = 1 and then ITI = 1. In the case of setting the P or A bits active, there is no need to invalidate or flush any caches because the processor does not load entries into the TLB that have P = 0 or A = 0. The i860 XR microprocessor searches only external memory for Page Directories and Page Tables in the translation process. The data cache is not searched. Therefore, Page Tables and Directories should be kept in non-cacheable memory, or flushed from the cache by any code which accesses them.

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 i860 XR microprocessor uses parallelism to increase the rate at which operations may be introduced into the unit. Parallelism in the i860 XR 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 graphics units can be described similarly, except that the number of stages may be different.

Figure 2.12 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 *fdest*, the pipeline is advanced one stage, and the input operands *fsrc1* and *fsrc2* are transferred to the first stage of the pipeline.

In the i860 XR 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 graphics 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. Single precision has three stages and double precision has two stages. The graphics 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 i860 XR 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
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**
Register to Register Moves	
fxfr	Transfer F-P to integer register
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
famov.r	F-P adder move
pfamov.r	Pipelined F-P adder move
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.p	Pipelined F-P multiply with add
pfmsm.p	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.r	F-P reg-reg move
pfmov.r	Pipelined F-P reg-reg move
nop	Core no-operation
fnop	F-P no-operation
pfle.p	Pipelined F-P less-than or equal

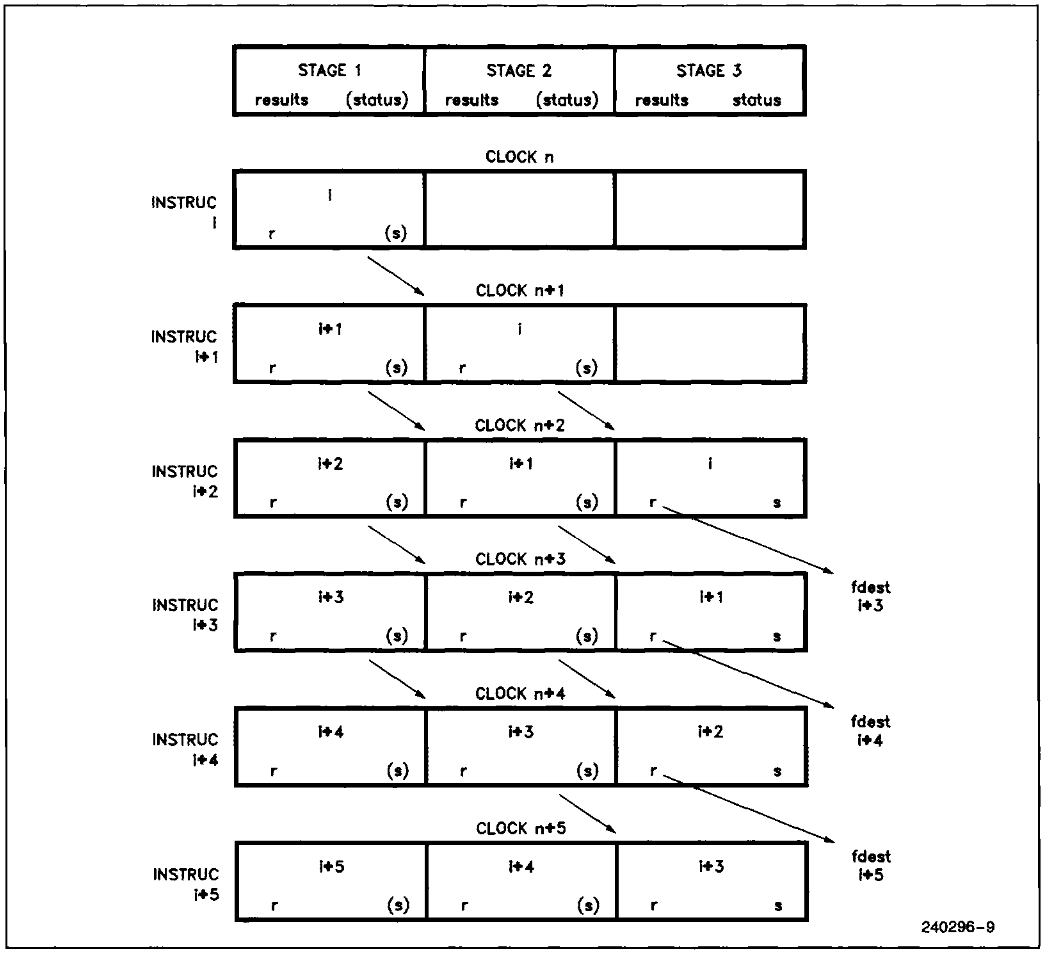


Figure 2.12. Pipelined Instruction Execution

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 i860 XR 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 *fdest* is that which was specified by the instruction that initiated that operation.

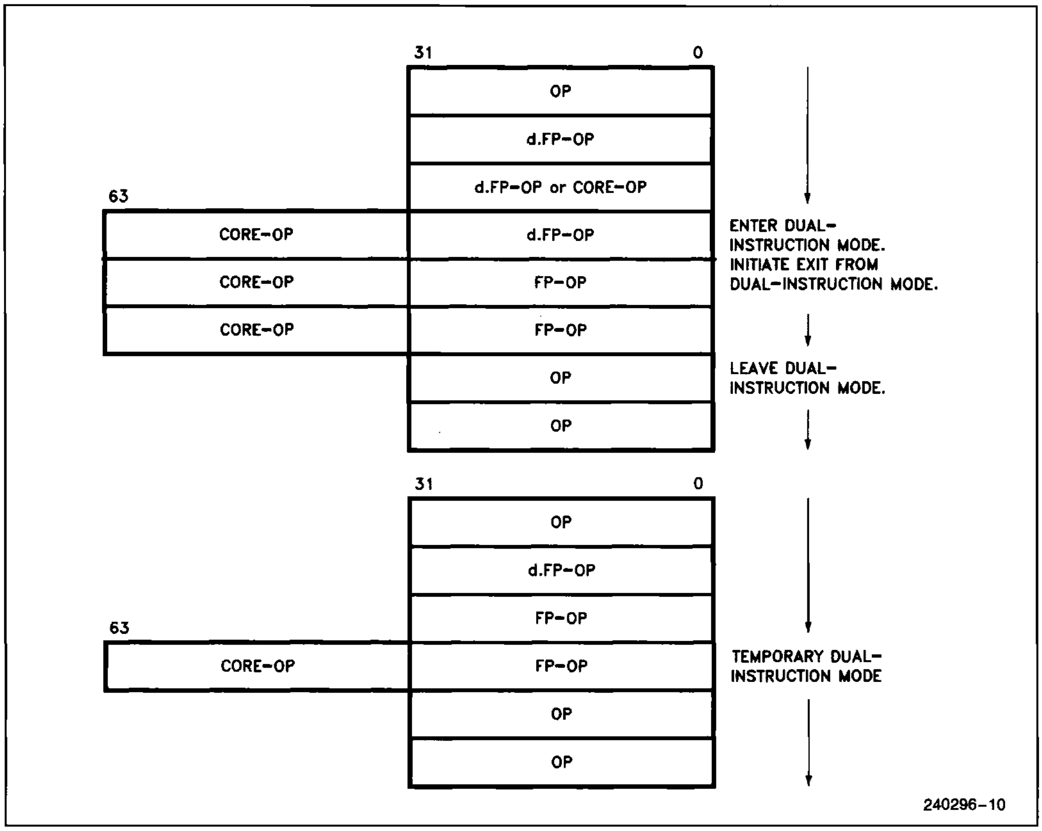

31
0
OP
d.FP-OP
d.FP-OP or CORE-OP
63
CORE-OP
d.FP-OP
ENTER DUAL-INSTRUCTION MODE.
CORE-OP
FP-OP
INITIATE EXIT FROM DUAL-INSTRUCTION MODE.
CORE-OP
FP-OP
OP
LEAVE DUAL-INSTRUCTION MODE.
OP
31
0
OP
d.FP-OP
FP-OP
63
CORE-OP
FP-OP
TEMPORARY DUAL-INSTRUCTION MODE
OP
OP
240296-10

Figure 2.13. Dual-Instruction Mode Transitions

If *fdest* is the same as *fsrc1* or *fsrc2*, the value being stored in *fdest* is used as the input operand. In this case, the precision of *fdest* 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 operands.

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

2.6.2 DUAL-INSTRUCTION MODE

Another form of parallelism results from the fact that the i860 XR 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 2.7 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 i860 XR 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 i860 XR 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** *fsrc1*, *fsrc2*, *fdest* (add and multiply), **pfsm** *fsrc1*, *fsrc2*, *fdest* (subtract and multiply), **pfmam** *fscr1*, *fsrc2*, *fdest* (multiply and add), and **pfmsm** *fsrc1*, *fsrc2*, *fdest* (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 *fsrc1* and subsequently supply that value to the multiplier pipeline in place of *fsrc1*.
    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 *fsrc1*.
- 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 *fsrc1*.
    2. Operand-2 of the multiplier can be *fsrc2* or the last stage result of the adder pipeline.
    3. Operand-1 of the adder can be *fsrc1*, the T-register, or the last stage result of the adder pipeline.
    4. Operand-2 of the adder can be *fsrc2*, 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. Table 8.8 shows the various encodings of the DPC field. Refer to Dual Operation Instructions section in the i860 Microprocessor Programmer's Reference Manual for pictorial description.

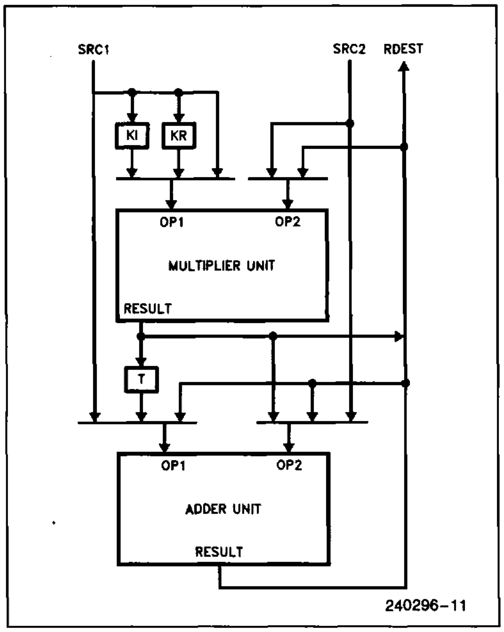


Figure 2.14. Dual-Operation Data Paths

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: *isrc1* and *isrc2*.

1. *isrc1* either contains the identifier of a 32-bit integer register or contains an immediate 16-bit address offset.
2. *isrc2* always specifies a register.

Table 2.8. Types of Traps

Type	Indication		Caused by	
	PSR,EPSR	FSR	Condition	Instruction
Instruction Fault	IT OF IL		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	

NOTES:
*These cases can be distinguished by examining the operand addresses.
The IL bit of the **epsr** must be checked by the trap handler to tell if the bus is currently in a locked sequence.

Because either *isrc1* or *isrc2* 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 or last 32K of the logical address space.

In addition, the floating-point load and store instructions may select autoincrement addressing. In this mode *isrc2* is replaced by the sum of *isrc1* and *isrc2* after performing the load or store. This mode makes stepping through arrays more efficient, because it eliminates one address-calculation instruction.

2.8 Traps and Interrupts

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. Interrupts and traps start execution in single instruction mode at virtual address 0xFFFFFF00 in supervisor level (U = 0).

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.

The processor begins executing the trap handler by transferring execution to virtual 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.

1. By the **trap** instruction. When **trap** is executed in dual-instruction mode, the floating-point companion of the **trap** instruction is not executed before the trap is taken.
2. 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. When **intovr** causes a trap in dual-instruction mode, the floating-point companion of the **intovr** instruction is completely executed before the trap is taken.
3. By violation of lock/unlock protocol, explained below. (Note that **trap** and **intovr** should not be used within a locked sequence; otherwise, it would be difficult to distinguish between this and the prior cases.)

   The lock protocol requires the following sequence of activities:

   1. **lock**
   2. Any load or store instruction that misses the cache
   3. **unlock**
   4. Any load or store instruction (regardless of whether it misses the cache)

There may be other instructions between any of these steps. The bus is locked after step 2, and remains locked until step 4. Step 4 must follow step 1 by 30 instructions or less, otherwise the instruction trap occurs. In case of a trap, IL is also set. If the load or store instruction in step 2 hits the cache, the sequence is legal, but the bus is not locked.

2.8.3 FLOATING-POINT FAULT

The floating-point fault is reported on floating-point instructions, **pst**, **fst**, and sometimes **fld**, **pfld**, **ixfr**. The floating-point faults of the i860 XR microprocessor support the floating-point exceptions defined by the IEEE standard as well as some other useful classes of exceptions. The i860 XR 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

When used as inputs to the multiplier or adder, 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 ⅓ 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, **fst.x** or **pst.x** (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 and the result is in the last stage of the pipeline of the appropriate unit. The trap handler must flush the pipeline, saving the results and the status bits.

In either pipelined or scalar mode, the trap handler must then compute the trapping result. 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 (if RR = 0) the trap handler must reload the pipeline with the saved results and status bits.

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 occurs during address translation for instruction fetches in any of these cases:

- The address fetched is in a page whose P (present) bit in the page table is clear (not present).
- The address fetched is in a supervisor mode page, but the processor is in user mode.
- The address fetched is in a page whose PTE has A = 0, and the access occurs during a locked sequence (i.e., between **lock** and **unlock**).

Note that several instructions are fetched at one time, either due to instruction prefetching or to instruction caching. Therefore, a trap handler can change from supervisor to user mode and continue to execute instructions fetched from a supervisor page. An instruction access trap occurs only when the next group of instructions is fetched from a supervisor page (up to eight instructions later). If, in the meantime, the handler branches to a user page, no instruction access trap occurs. No protection violation results, because the processor does not permit data accesses to supervisor pages while running in user mode.

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 (Dirty) 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 **db** 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.
- The operand was in a page whose PTE had A = 0, and the access occurred during a locked sequence. (i.e., between **lock** and **unlock**.)
- Write protection (determined by **epsr** bit WP = 1) is violated in supervisor mode.

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 i860 XR microprocessor is reset, execution begins in single-instruction mode at physical 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 read-only fields of the **espr** are set to identify the processor, while the IL, WP, and PBM bits are cleared. The bits U, IM, BR, and BW in **psr** are cleared, as are the trap bits FT, DAT, IAT, IN, and IT. All other bits of **psr** and all other register contents are **undefined.**

Refer to Table 2.9 for a summary of these initial settings.

Table 2.9. Register and Cache Values after Reset

Registers	Initial Value
Integer Registers	*Undefined*
Floating-Point Registers	*Undefined*
psr	U, IM, BR, BW, FT, DAT, IAT, IN, IT = 0; others are *undefined*
epsr	IL, WP, PBM, BE = 0; Processor Type, Stepping Number, DCS are read only; others are *undefined*
db	*Undefined*
dirbase	DPS, BL, ATE = 0; others are *undefined*
fir	*Undefined*
fsr	*Undefined*
KR, KI, T, MERGE	*Undefined*
Caches	**Initial Value**
Instruction Cache	Flushed
Data Cache	*Undefined*
TLB	Flushed

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. The data cache has no "validity" bits, so memory accesses before the flush may result in false data cache hits.

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 i860 XR 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 i860 XR microprocessor supports debugging with both data and instruction breakpoints. The features of the i860 XR architecture that support debugging include:

- **db** (data breakpoint register) which permits specification of a data addresses that the i860 XR 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 *isrc1* and *isrc2* 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 i860 XR microprocessor. Timing of other signals is specified relative to the rising edge of this signal. The i860 XR microprocessor can utilize a clock rate of 25 MHz, 33.3 MHz or 40 MHz. The internal operating frequency is the same as the external clock.

3.1.2 SYSTEM RESET (RESET)

Asserting RESET for at least 16 CLK periods causes initialization of the i860 XR 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 i860 XR microprocessor bus arbitration. At some clock after the HOLD signal is asserted, the i860 XR microprocessor releases con-

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	High/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.

trol of the local bus and puts all bus interface outputs (except BREQ and HLDA) into a 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 or LOCK# is asserted.

When leaving a bus hold, the i860 XR microprocessor deactivates HLDA and, in the same clock period, initiates a pending bus cycle, if any.

Hold is an asynchronous input.

3.1.4 BUS REQUEST (BREQ)

This signal is asserted when the i860 XR 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 i860 XR microprocessor. BREQ is asserted the clock after the i860 XR microprocessor realizes an internal request for the bus. In normal operation, BREQ goes low the clock after ADS# goes low for the final pending bus cycle. (Refer to Figure 4.10 for timing information.) During data or instuction cache fills, however, BREQ may be deasserted for one or more clocks, due to cache and TLB logic.

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 i860 XR 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). When the bus is not locked, the maximum time between the assertion of INT and the execution of the first instruction of the trap handler is ten clocks, plus the time for four 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), plus the time for twenty nonpipelined read cycles (six TLB misses, with eight refetches when the A-bit is zero), plus the time for eight nonpipelined writes (updates to the A-bit).

If the bus is locked from a **lock** instruction, the INT pin is ignored and the INT bit of **epsr** is always zero. The **lock** instruction can only assert LOCK# for 30-33 instructions before trapping.

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. In all noncacheable read cycles (KEN# deasserted), the byte enables match the length and address of the requested data. Cacheable read cycles (KEN# asserted), however, result in four 64-bit memory cycles to fill an entire 32-byte cache line. The BE*n*# pins activated are those that represent the operand of the load instruction that caused the line fill, and these same BE*n*# pins remain activated for all four cycles of the line fill. All 64 bits must be returned for each cycle without regard for the BE*n*# signals. In all write cycles (noncacheable writes as well as cache line write-backs) the BE*n*# signals indicate the bytes that must be written.

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. The A2 in the table does not represent a physical pin, just a conceptual internal address line value. The "x"under A2 for CS8 mode means "not applicable", or "don't care". All other combinations of byte enables indicate data accesses.

The address and byte-enable pins are driven until either NA# or READY# is asserted.

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 read bus cycles, all 64 bits of the data bus are latched, even in CS8-mode instruction fetches when only the low-order eight bits are used.

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. Data continues to be driven until READY# for the current cycle is returned.

3.1.8 BUS LOCK (LOCK#)

This signal is used to provide atomic (indivisible) read-modify-write sequences in multiprocessor systems. A multiprocessor bus arbiter must permit only one processor a locked access to the address which is on the bus when LOCK# first activates. The system must maintain the lock of that location until LOCK# deactivates.

The i860 XR 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 address operand of the first load or store instruction executed after the BL bit is set by the **lock** instruction. Pending bus cycles are locked according to the value of the BL bit when the instruction was executed. 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 i860 XR microprocessor still asserts LOCK# for that bus cycle.

If ADS# is active when LOCK# deactivates, then that request should complete before the hardware relinquishes the lock. If ADS# is not active, the locking of the location can immediately end when LOCK# deactivates. Of course the simplest arbitration hardware can just lock the entire bus against all other accesses during LOCK# assertion through RDY# of the cycle in which LOCK# goes inactive.

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	1	Low-order address bits		

When the BL bit is deasserted with the **unlock** instruction, LOCK# is deasserted with the next load or store but after any pending bus cycles. Between locked sequences, at least one cycle of no LOCK# is guaranteed by the behavior of the **unlock** instruction. LOCK# deassertion may occur independently of ADS# for the case of a trap or a cache hit after **unlock**.

The i860 XR 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 to perform a read-modify-write sequence. Instruction fetches do not alter the LOCK# pin.

Between **lock** and **unlock** instructions, the INT pin is ignored and the INT bit of **epsr** is zero when read by **ld.c epsr**. The time that interrupts are disabled is limited by the lock protocol outlined in Section 2.8.2.

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

This pin specifies whether a bus cycle is a read (LOW) or write (HIGH) cycle. It is driven until either NA# or READY# is asserted.

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 i860 XR microprocessor asserts NENE# when the current address is in the same DRAM page as the previous bus cycle. The i860 XR 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# for at least one clock to indicate that it is ready to accept the next address from the i860 XR 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 i860 XR microprocessor samples NA# every clock, starting one clock after the prior activation of ADS#. When NA# is active, the i860 XR microprocessor is free to drive address and bus-cycle definition for the next pending bus cycle. The i860 XR 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.

3.1.12 TRANSFER ACKNOWLEDGE (READY#)

The system must assert the READY# signal during read cycles when valid data is on the data pins and during write cycles when the system has accepted data from the data pins. READY# must be asserted for at least one clock. Sampling of READY# begins in the clock after an ADS# or in the second clock after a prior READY#.

3.1.13 ADDRESS STATUS (ADS#)

The i860 XR 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 held active for one clock.

3.1.14 CACHE ENABLE (KEN#)

The i860 XR 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 CD and WT bits to control cacheability on a page by page basis (refer to Table 3.3).

If the address is one that is permitted to be in the cache, KEN# must be continuously asserted during the sampling period starting from the second rising clock edge after ADS# is asserted, through the clock NA# or READY# is asserted. The entire 64 bits 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 be generated to fill the rest of the 32-byte cache block.

If KEN# is found deasserted at any clock from the clock after ADS# through the clock of the **first** NA# or READY#, the data being read will not be 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. To avoid caching a line, external hardware must deassert KEN# during or before the first NA# or READY#.

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. When paging is disabled, PTB remains inactive.

Table 3.3. Cacheability based on KEN# and CD OR WT

CD OR WT	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 3.3.

3.1.17 BOUNDARY SCAN ENABLE (BSCN)

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

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 i860 XR 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 i860 XR 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 i860 XR microprocessor begins executing at physical address 0xFFFFFF00. The program-visible state of the i860 XR microprocessor after reset is detailed in section 2.8.7.

Eight-bit code-size mode is selected when INT/CS8 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 i860 XR 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 3.1.14) 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 i860 XR microprocessor.

3.3 Testability

The i860 XR 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 i860 XR 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, BREQ, RESET, and HOLD.

The pins BSCN and SCAN control the boundary scan mode (refer to Table 3.5). When BSCN is asserted, the i860 XR 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 i860 XR 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 i860 XR 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 . . .

2

1. Enter shift mode to assign values to the latches that correspond with the pins.
2. Enter normal mode. In normal mode the i860 XR 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, the tester should assert both the RESET and HOLD pins. Then 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 i860 XR 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** $\geq$ **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.

$$\begin{tikzcd}
{} \arrow[r] & \overset{1}{\text{SHI}} \arrow[r] & \overset{2}{\text{BSCN}} \arrow[r] & \overset{3}{\text{SCAN}} \arrow[r] & \overset{4}{\text{RESET}} \arrow[r] & \overset{5}{\text{DATAt}} \arrow[r] & \overset{6}{\text{D0}} \arrow[r] & \dots \arrow[r] & \overset{69}{\text{D63}} \arrow[r] & {} \\
\overset{70}{\text{CC1}} \arrow[r] & \overset{71}{\text{CC0}} \arrow[r] & \overset{72}{\text{A31}} \arrow[r] & \dots \arrow[r] & \overset{100}{\text{A3}} \arrow[r] & \overset{101}{\text{ADDRt}} \arrow[r] & \overset{102}{\text{NENEt}} \arrow[r] & \overset{103}{\text{NENE\#}} \arrow[r] & \overset{104}{\text{PTBt}} \arrow[r] & {} \\
\overset{105}{\text{PTB\#}} \arrow[r] & \overset{106}{\text{W/Rt}} \arrow[r] & \overset{107}{\text{W/R\#}} \arrow[r] & \overset{108}{\text{ADSt}} \arrow[r] & \overset{109}{\text{ADS\#}} \arrow[r] & \overset{110}{\text{HLDA}} \arrow[r] & \overset{111}{\text{LOCKt}} \arrow[r] & \overset{112}{\text{LOCK\#}} \arrow[r] & \overset{113}{\text{READY\#}} \arrow[r] & {} \\
\overset{114}{\text{KEN\#}} \arrow[r] & \overset{115}{\text{NA\#}} \arrow[r] & \overset{116}{\text{INT/CS8}} \arrow[r] & \overset{117}{\text{HOLD}} \arrow[r] & \overset{118}{\text{BEt}} \arrow[r] & \overset{119}{\text{BE7\#}} \arrow[r] & \dots \arrow[r] & \overset{126}{\text{BE0\#}} \arrow[r] & \overset{127}{\text{BREQ}} \arrow[r] & {}
\end{tikzcd}$$

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**). If a bus request is pending, the next cycle will be initiated when NA# is sampled active, even if the current cycle has not terminated. In this case, pipelining occurs. NA# is not recognized unitl after 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 i860 XR microprocessor bus state machine. A bus cycle is composed of two or more states. Each bus state lasts for one CLK period.

The i860 XR 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 i860 XR 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 i860 XR microprocessor is not ready to start a new cycle.

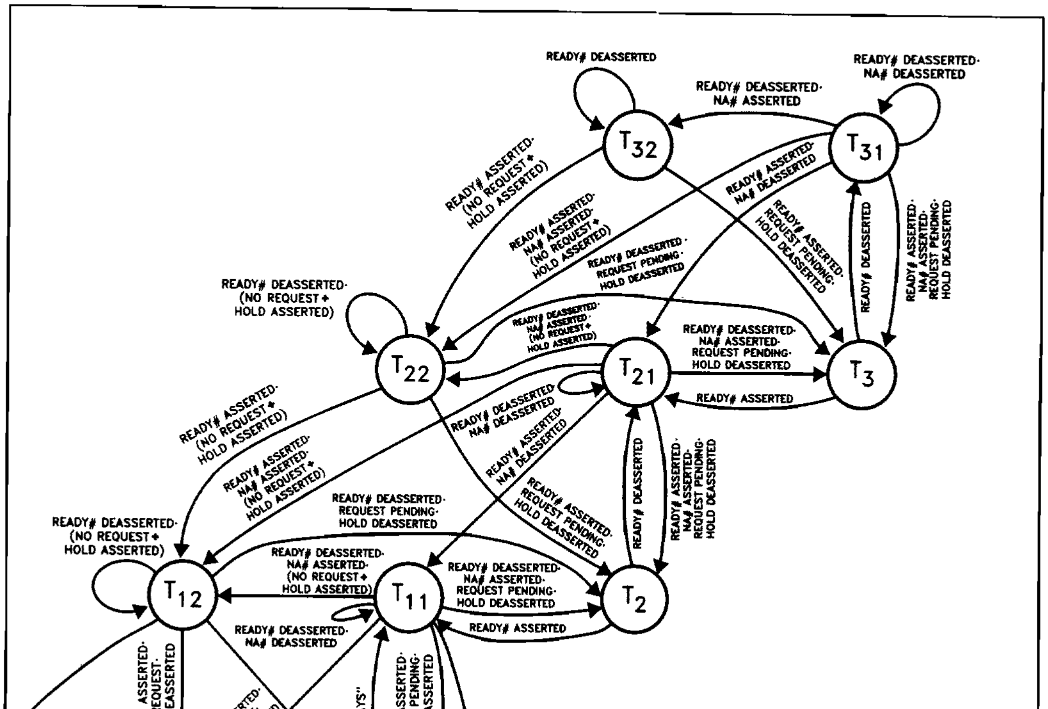
Five conditions have to be met to start a new cycle while one or more cycles are already pending:

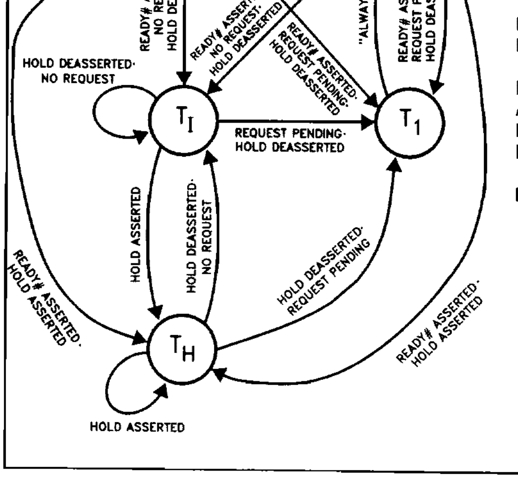
1. READY# inactive
2. NA# having been active
3. An internal request pending (BREQ active)
4. HOLD not active
5. Fewer than three cycles outstanding

Note that BREQ is asserted on the clock after the i860 XR microprocessor realizes an internal request for the bus.

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
T_{32}
T_{31}
READY# ASSERTED· (NO REQUEST + HOLD ASSERTED)
READY# ASSERTED· NA# ASSERTED· (NO REQUEST + HOLD ASSERTED)
READY# ASSERTED· NA# DEASSERTED
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)
T_{22}
T_{21}
T_3
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)
T_{12}
T_{11}
T_2
READY# DEASSERTED· NA# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
READY# ASSERTED
READY# DEASSERTED· NA# DEASSERTED
READY# ASSERTED· NO REQUEST· HOLD DEASSERTED
READY# ASSERTED· NO REQUEST· HOLD DEASSERTED
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
"ALWAYS"
READY# ASSERTED· REQUEST PENDING· HOLD DEASSERTED
HOLD DEASSERTED· NO REQUEST
T_I
T_1
REQUEST PENDING· HOLD DEASSERTED
HOLD ASSERTED
HOLD DEASSERTED· NO REQUEST
HOLD DEASSERTED· REQUEST PENDING
READY# ASSERTED· HOLD ASSERTED
READY# ASSERTED· HOLD ASSERTED
T_H
HOLD ASSERTED



NOTES:

READY#	Once READY# has been sampled active, it is not sampled again until two clocks later
NA#	Not sampled during ADS# active clock
ADS#	Active in T_1, T_2 and T_3
HLDA	Active in T_H
HOLD	HOLD in this figure is the internally synchronized version of the external signal HOLD
REQUEST	Internal Bus Request Pending (BREQ asserted)

240296–29

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).

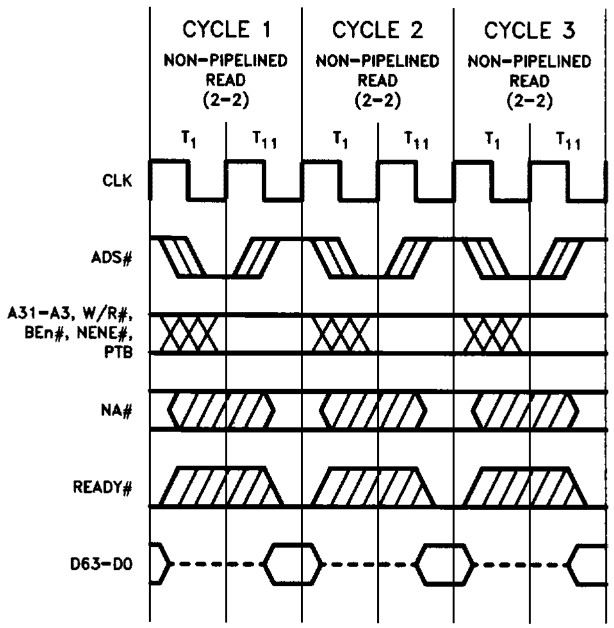
Internal pending bus requests start new bus cycles only if no HOLD request has been recognized. T_H is entered from the idle state T_I, T_{11}, and T_{12}. HLDA is active in this state. There is a one clock delay to synchronize the HOLD input when the signal meets the respective minimum setup and hold 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 i860 XR 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.



240296-13

Figure 4.2. Fastest Read Cycles

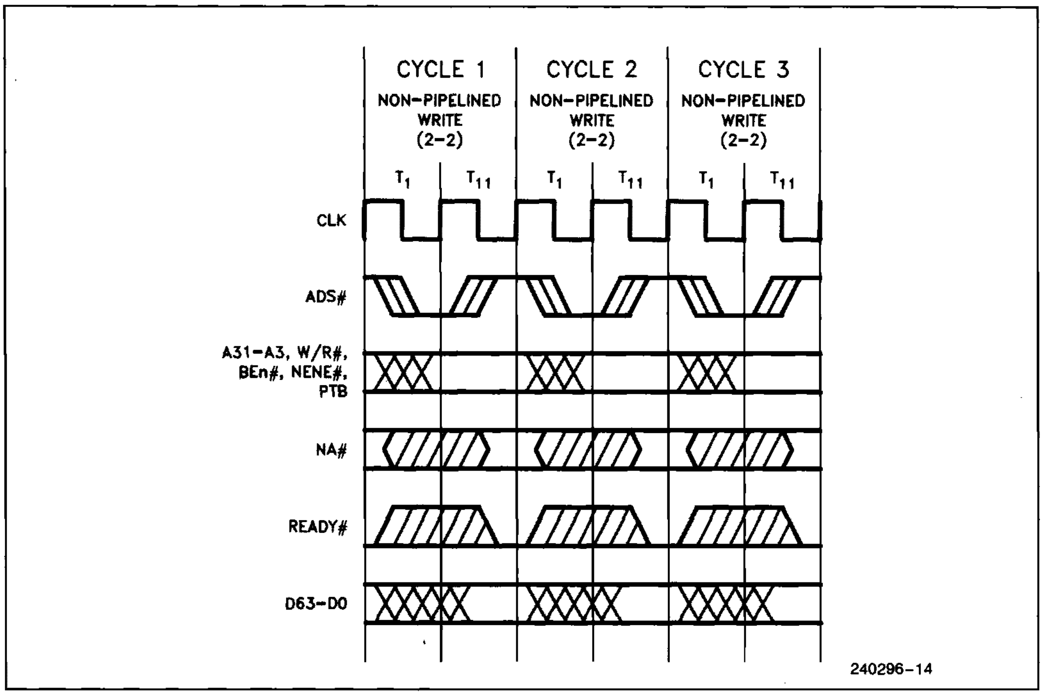

CYCLE 1
NON-PIPELINED WRITE (2-2)
CYCLE 2
NON-PIPELINED WRITE (2-2)
CYCLE 3
NON-PIPELINED WRITE (2-2)
T1
T11
CLK
ADS#
A31-A3, W/R#, BEn#, NENE#, PTB
NA#
READY#
D63-D0
240296-14

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 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 i860 XR 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 i860 XR microprocessor does not require a wait state.

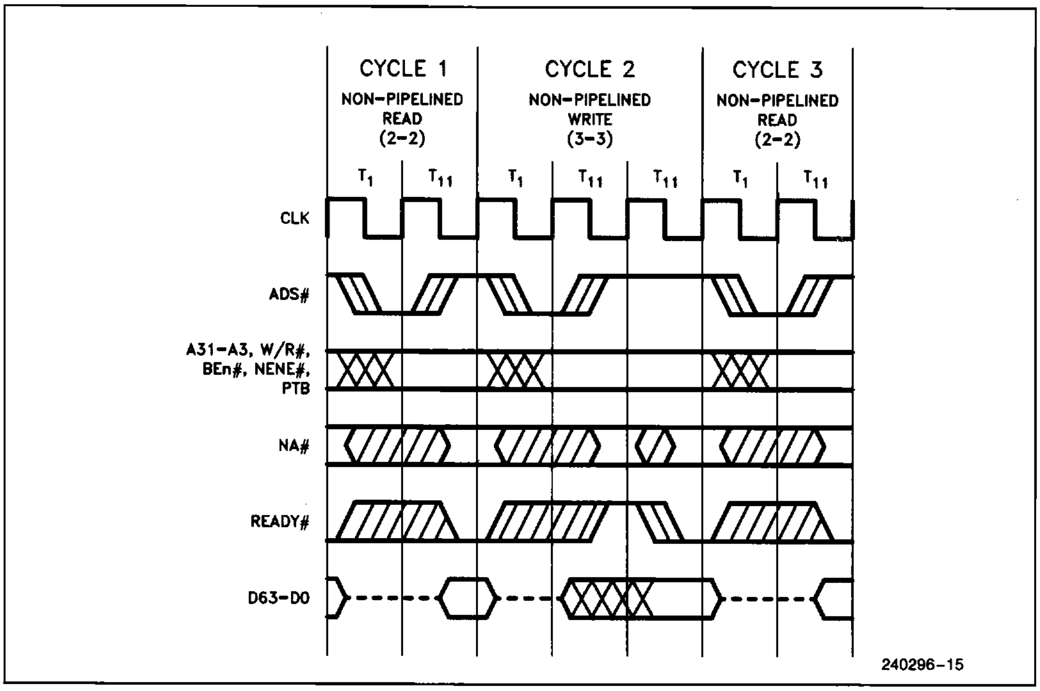


Figure 4.4. Fastest Read/Write Cycles

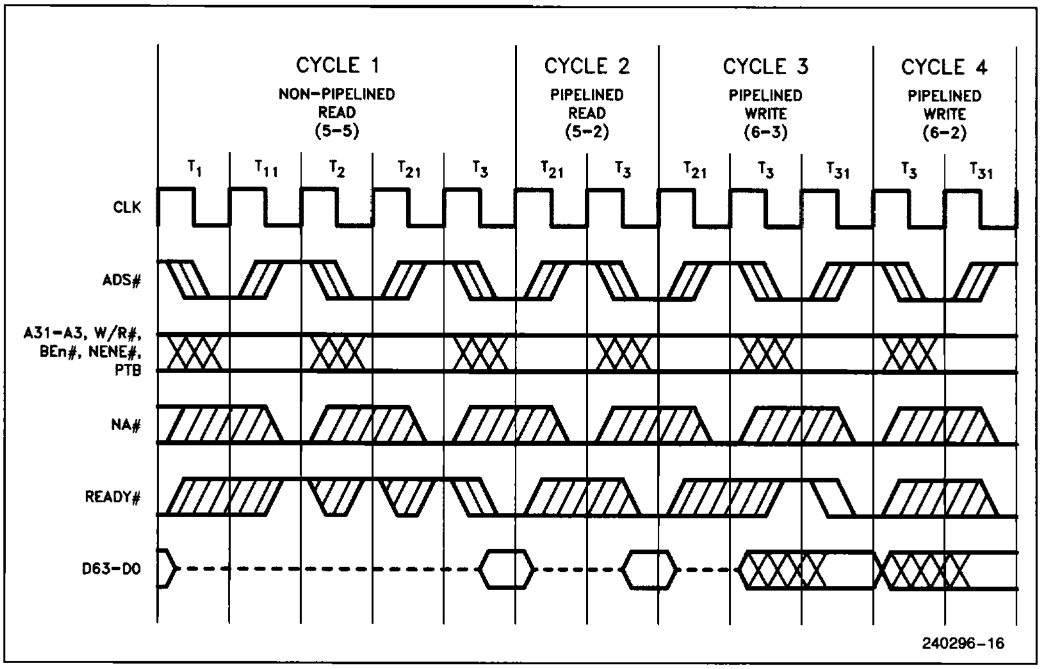


Figure 4.5. Pipelined Read Followed by Pipelined Write

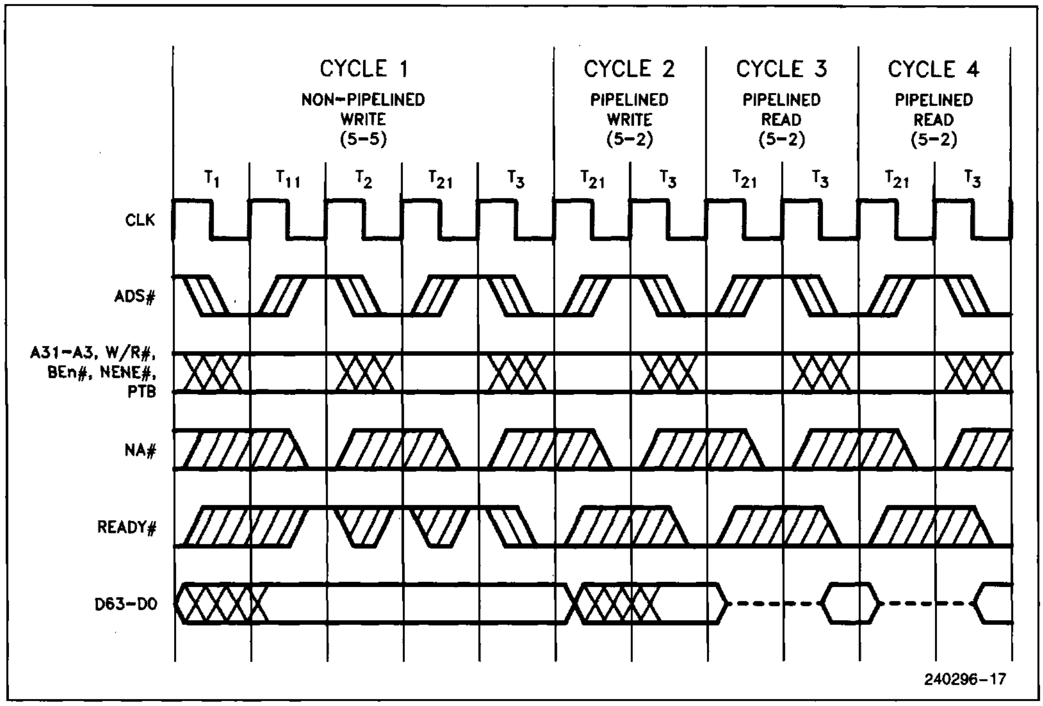


Figure 4.6. Pipelined Write Followed by Pipelined Read

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 the corresponding outstanding cycle, 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 i860 XR 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.

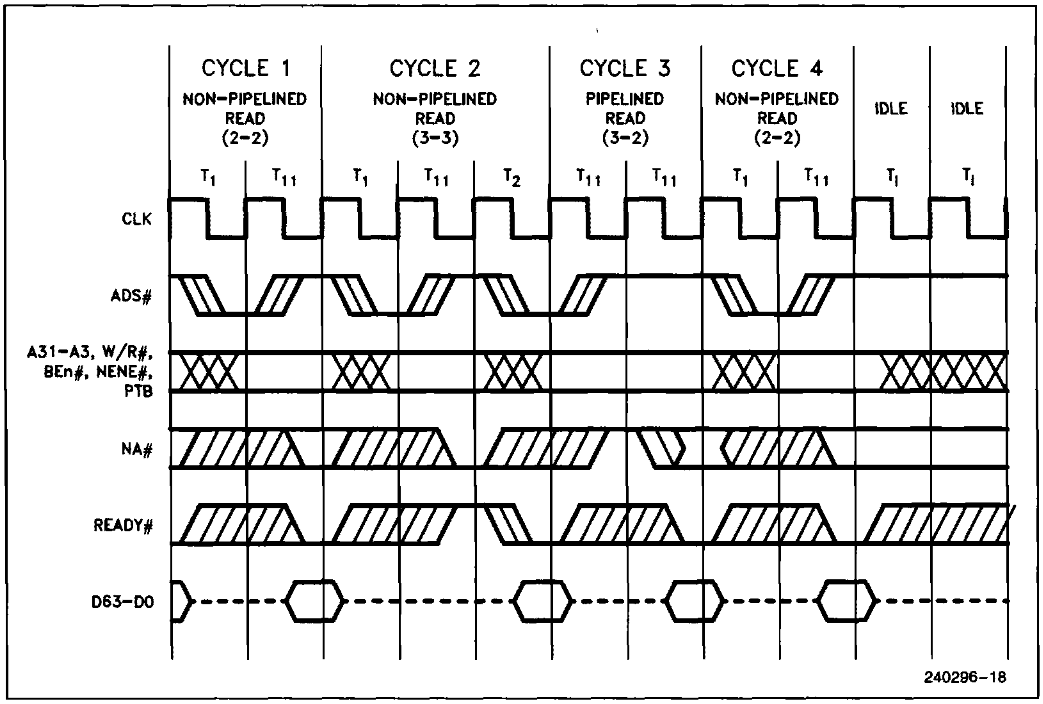


Figure 4.7. Pipelining Driven by NA#

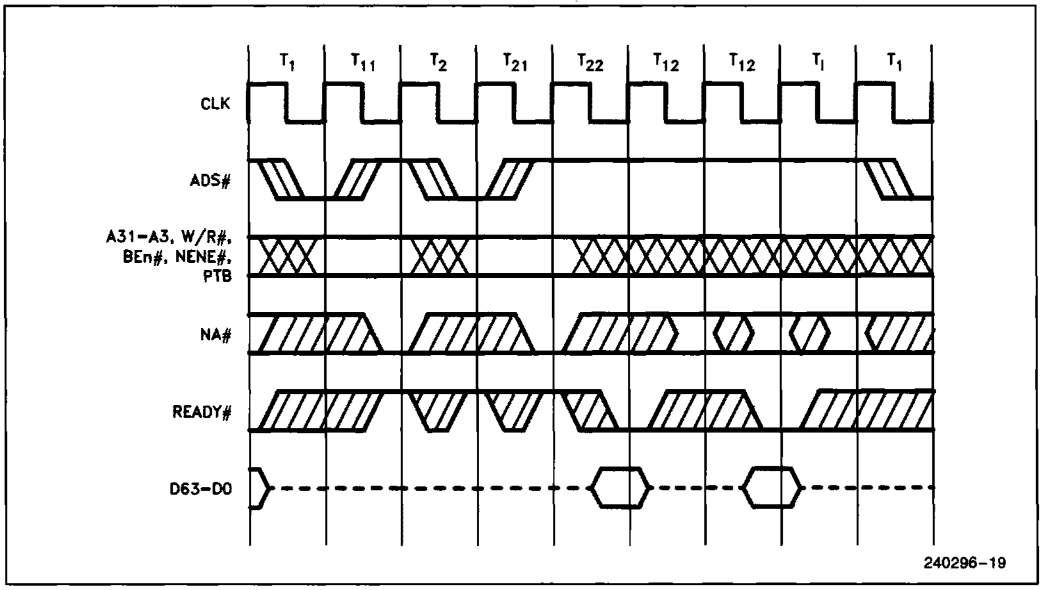


Figure 4.8. NA# Active with No Internal Bus Request

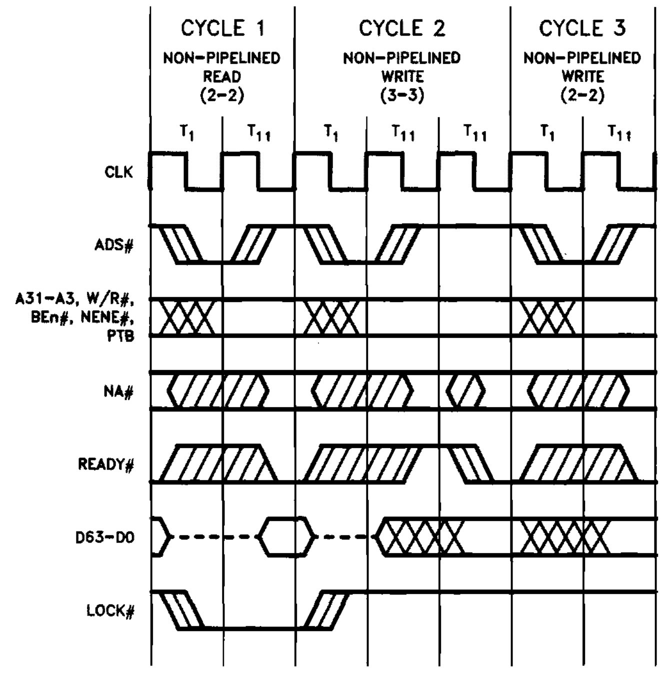


Figure 4.9. Locked Cycles

When there is no internal bus request, activating NA# does not start a new cycle; the i860 XR 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 using the **lock** instruction (refer to section 2) or by the i860 XR microprocessor 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. Cycle 3 is not locked because LOCK# is no longer asserted when Cycle 2 starts.

4.3.5 HOLD AND BREQ ARBITRATION CYCLES

The HOLD, HLDA, and BREQ signals permit bus arbitration between the i860 XR microprocessor and another bus master.

See Figure 4.10. When HOLD is asserted, the i860 XR microprocessor does not relinquish control of the bus until all outstanding cycles are completed. If HOLD were asserted one clock earlier, the last i860 XR microprocessor bus cycle before HLDA would not be started.

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. Likewise when HOLD is deasserted, there is a one-clock delay for internal synchronization before HLDA is deasserted. The outputs (except HLDA and BREQ) float when HLDA is asserted.

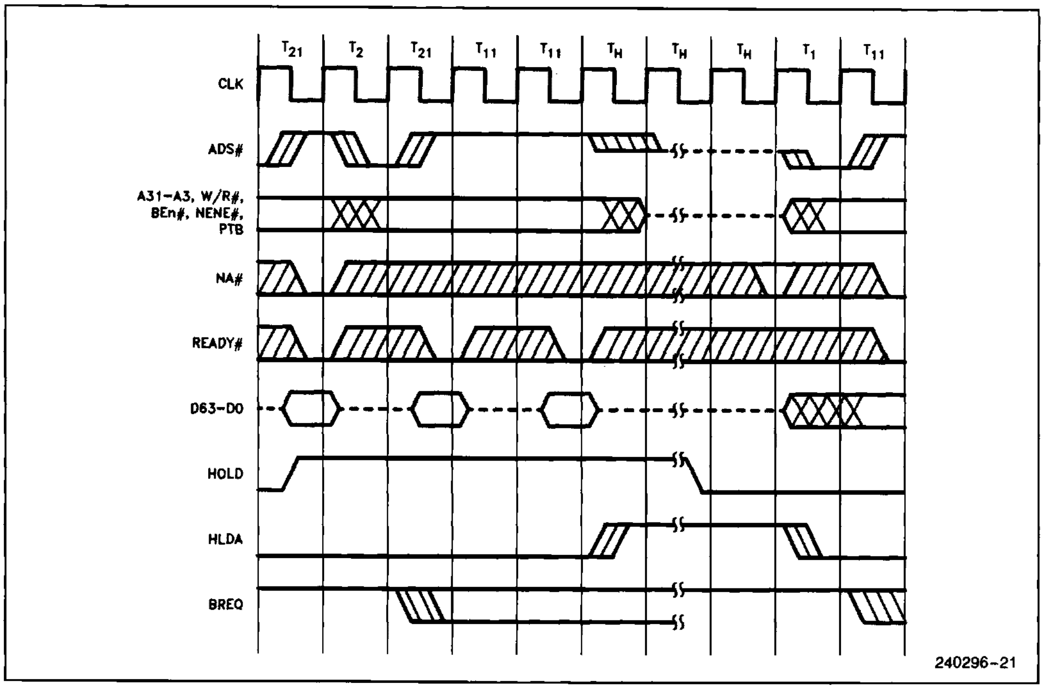


Figure 4.10. HOLD, HLDA, and BREQ

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 i860 XR microprocessor enters CS8 mode. No inputs (except for HOLD and INT/CS8) 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 Table 3.4 "Output Pin Status during Reset".

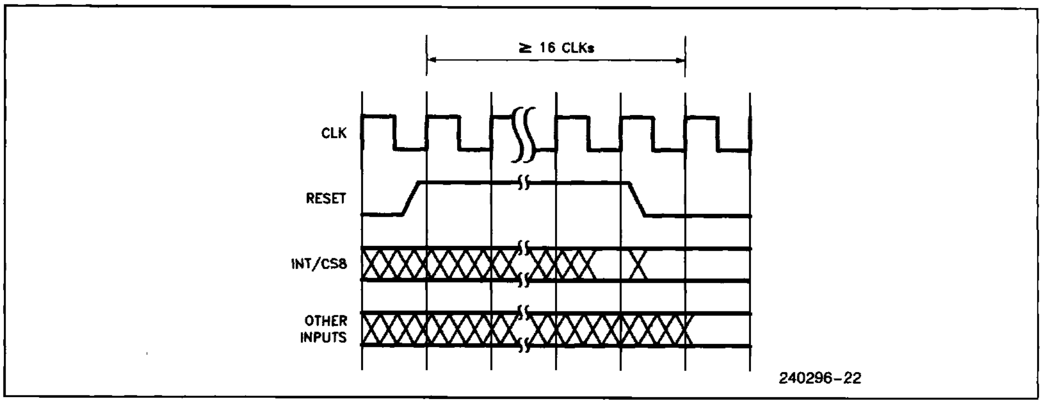


Figure 4.11. Reset Activities

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.

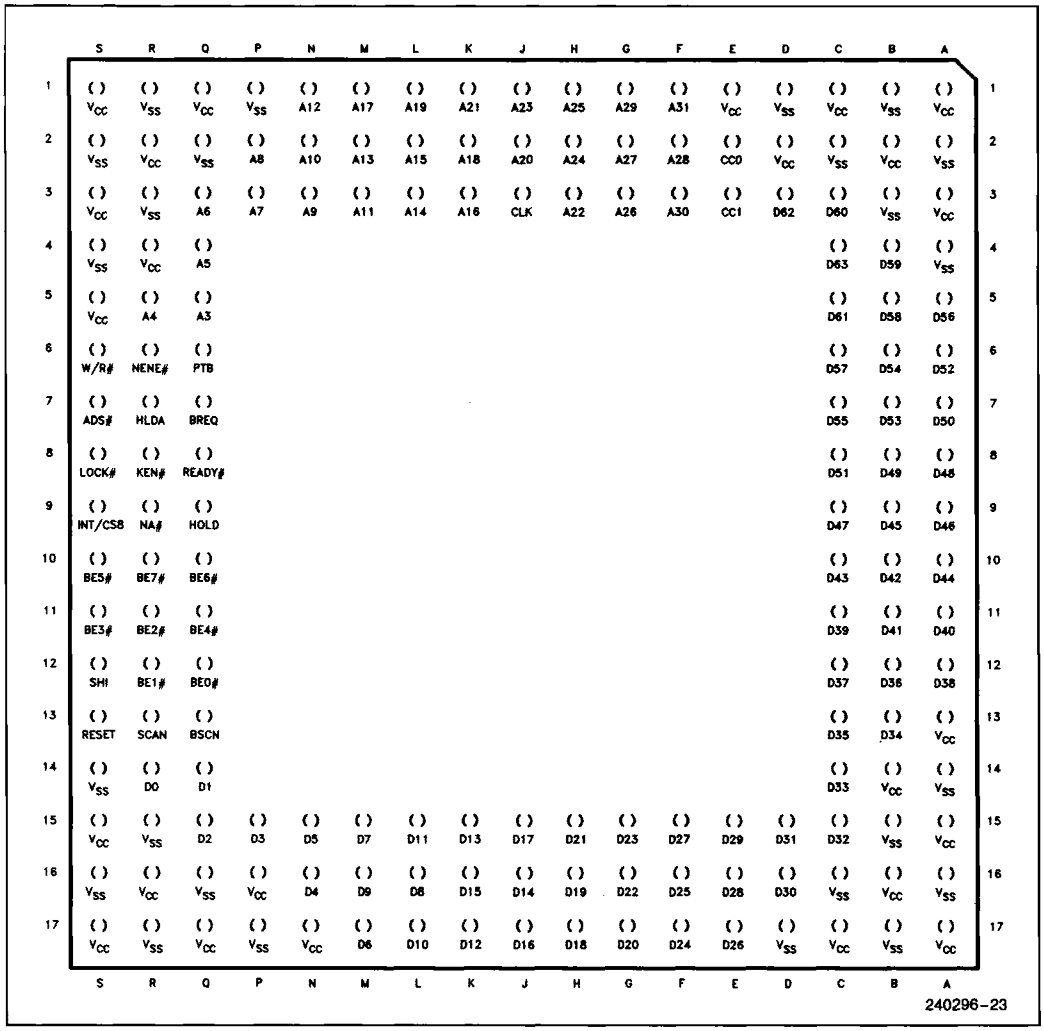

S R Q P N M L K J H G F E D C B A
1 Vcc Vss Vcc Vss A12 A17 A19 A21 A23 A25 A29 A31 Vcc Vss Vcc Vss Vcc 1
2 Vss Vcc Vss A8 A10 A13 A15 A18 A20 A24 A27 A28 CC0 Vcc Vss Vcc Vss 2
3 Vcc Vss A6 A7 A9 A11 A14 A16 CLK A22 A26 A30 CC1 D62 D60 Vss Vcc 3
4 Vss Vcc A5 D63 D59 Vss 4
5 Vcc A4 A3 D61 D58 D56 5
6 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 Vcc 13
14 Vss D0 D1 D33 Vcc Vss 14
15 Vcc Vss D2 D3 D5 D7 D11 D13 D17 D21 D23 D27 D29 D31 D32 Vss Vcc 15
16 Vss Vcc Vss Vcc D4 D9 D8 D15 D14 D19 D22 D25 D28 D30 Vss Vcc Vss 16
17 Vcc Vss Vcc Vss Vcc D6 D10 D12 D16 D18 D20 D24 D26 Vss Vcc Vss Vcc 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

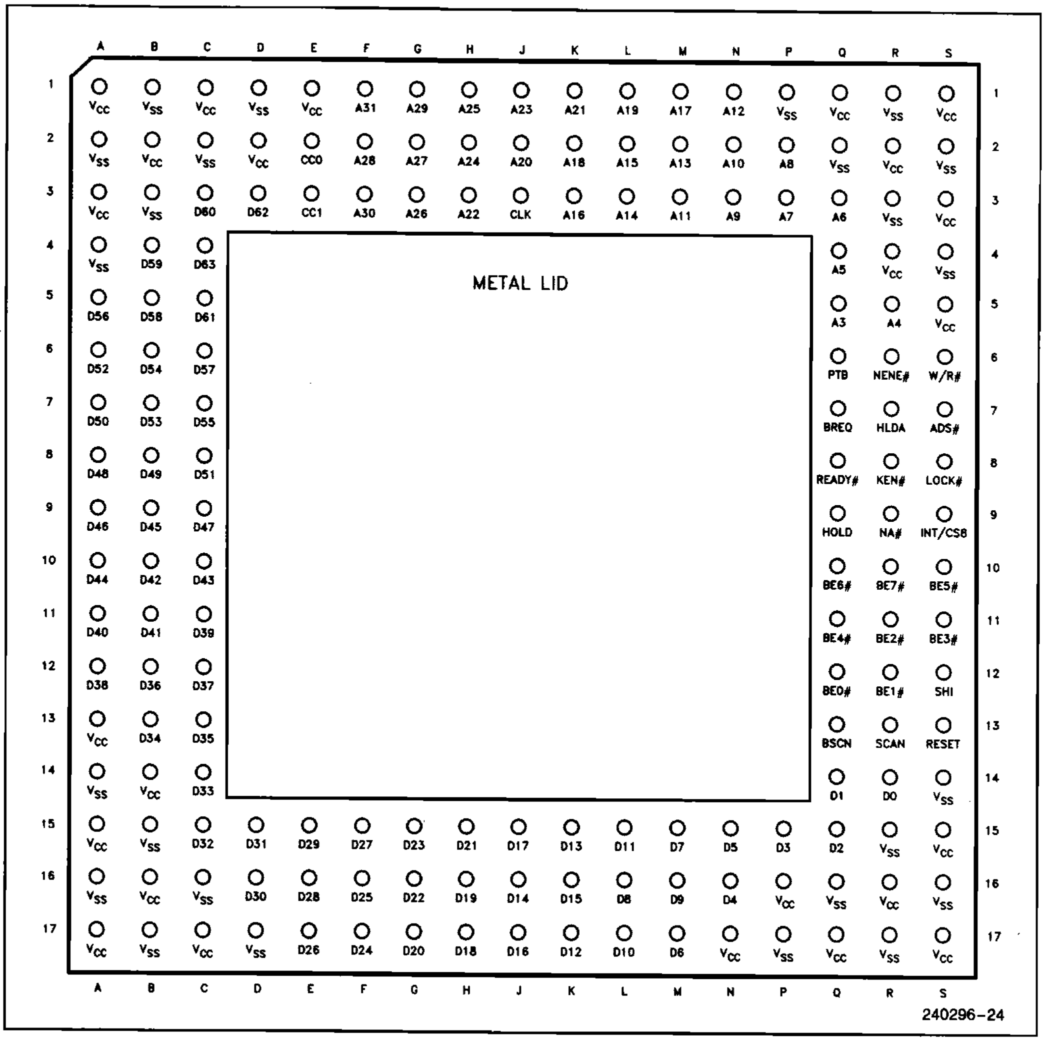

A B C D E F G H J K L M N P Q R S
1 V_{CC} V_{SS} V_{CC} V_{SS} V_{CC} A31 A29 A25 A23 A21 A19 A17 A12 V_{SS} V_{CC} V_{SS} V_{CC} 1
2 V_{SS} V_{CC} V_{SS} V_{CC} CC0 A28 A27 A24 A20 A18 A15 A13 A10 A8 V_{SS} V_{CC} V_{SS} 2
3 V_{CC} V_{SS} D60 D62 CC1 A30 A26 A22 CLK A16 A14 A11 A9 A7 A6 V_{SS} V_{CC}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

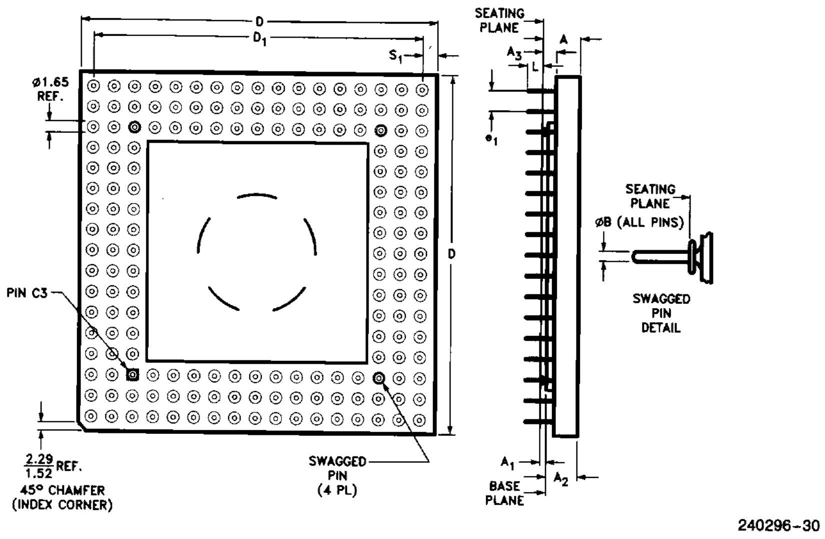
Table 5.2. Pin Cross Reference by Pin Name

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	2.79	3.56	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		# of Pins	168		# of Pins
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

For this section, let:

P	=	maximum power consumption
T_C	=	case temperature
T_A	=	ambient air temperature
θ_{CA}	=	thermal resistance from case to ambient air
θ_{JC}	=	thermal resistance from junction to case
θ_{JA}	=	thermal resistance from junction to ambient air

The i860 XR microprocessor is specified for operation when T_C is within the range of 0°C–85°C. T_C may be measured in any environment to determine whether the i860 XR microprocessor is within specified operating range. The case temperature should be measured at the center of the top surface opposite the pins.

T_A 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} and θ_{JC} at various airflows are given in Table 6.1 for the 1.75 sq. in., 168 pin, ceramic PGA. θ_{JC} is also shown so that θ_{JA} can be calculated by:

$$\theta_{CA} = \theta_{JA} - \theta_{JC}$$

Note that θ_{JC} with a heatsink differs from θ_{JC} without a heatsink because case temperature is measured differently. Case temperature for θ_{JC} with heatsink is measured at the center of the heat fin base. Case temperature for θ_{JC} without heatsink is measured at the center of package top surface.

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.

Figure 6.1 gives typical I_{CC} derating with case temperature. For more information on heat sinks, measurement techniques, or package characteristics, refer to *Intel Packaging Handbook*, order number 240800.

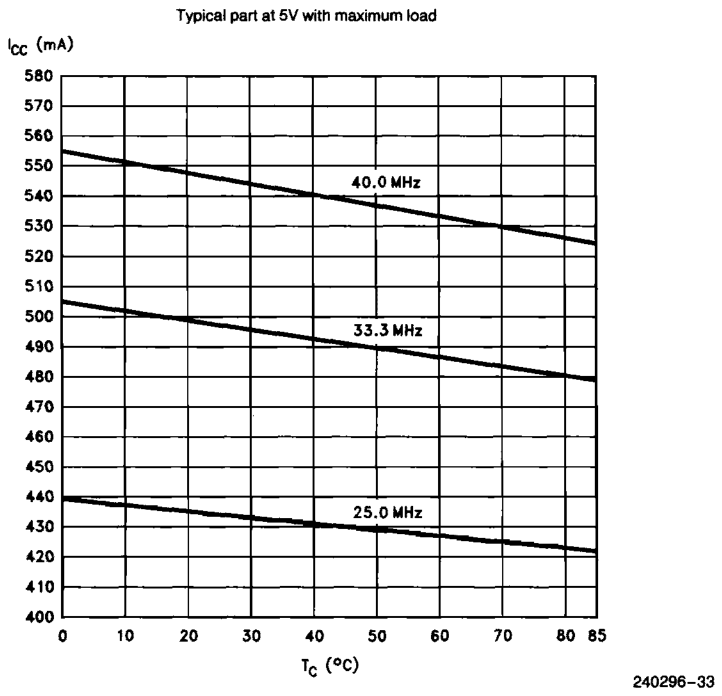


Figure 6.1. I_{CC} vs Case Temperature

Table 6.1. Thermal Resistance (°C/W) θ_{JC} and θ_{CA}

	θ_{JC}	θ_{CA} at Airflow-ft/min (m/sec)					
		0 (0)	200 (1.01)	400 (2.03)	600 (3.04)	800 (4.06)	1000 (5.07)
With Heat Sink*	2	11	6	4	3.2	2.5	2.2
Without Heat Sink	1.5	17.5	13	11	9.5	8.5	8

*Nine-fin, unidirectional heat sink (fin dimensions: 0.350" height, 0.040 width, 0.115" center-to-center spacing, 1.530" length).

Table 6.2. Maximum Allowable T_A at Various Airflows

In °C

	f_{CLK} (MHz)	Airflow-ft/min (m/sec)					
		0 (0)	200 (1.01)	400 (2.03)	600 (3.04)	800 (4.06)	1000 (5.07)
T_A with Heat Sink*	25.0	57.5	70	75	77	78.8	79.5
	33.3	52	67	73	75.5	77.4	78.5
	40.0	49.3	65.5	72	74.6	76.9	77.9
T_A without Heat Sink	25.0	41.3	52.5	57.5	61.3	63.8	65
	33.3	32.5	46	52	56.5	59.5	61
	40.0	28.1	42.8	49.3	54.1	57.4	59

*Nine-fin unidirectional heat sink (fin dimensions: 0.350″ height, 0.040 width, 0.115″ center-to-center spacing, 1.530″ length).

7.0 ELECTRICAL DATA

Inputs and outputs are TTL compatible, except for CLK. 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 signals reach 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 6.5V

NOTICE: This data sheet contains preliminary information on new products in production. The specifications are subject to change without notice. Verify with your local Intel Sales office that you have the latest data sheet before finalizing a design.

**WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

7.2 D.C. Characteristics

Table 7.1. DC Characteristics

T_C = 0°C to 85°C, V_{CC} = 5V ±5%

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_{ILC}	CLK Input LOW Voltage	−0.3	+0.8	V	
V_{IHC}	CLK Input HIGH Voltage	3.0	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 = 25.0 MHz		500	mA	V_{CC} @5V
	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	(Note 3)
C_O	I/O or Output Capacitance		15	pF	(Note 3)
C_{CLK}	Clock Capacitance		20	pF	(Note 3)

NOTES:
1. This parameter is measured at 4.0 mA for A31–A3, D63–D0, BE7#–BE0#; at 5.0 mA for all other outputs.
2. This parameter is measured at 1.0 mA for A31–A3, D63–D0, BE7#–BE0#; at 0.9 mA all other outputs.
3. These are not tested. They are guaranteed by design characterization.

7.3 A.C. Characteristics

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

Symbol	Parameter	25 MHz		33 MHz		40 MHz		Notes
		Min (ns)	Max (ns)	Min (ns)	Max (ns)	Min (ns)	Max (ns)	
t1	CLK Period	40	125	30	125	25	125	
t2	CLK High Time	6		5		3		at 3V
t3	CLK Low Time	8		7		5		at 0.8V
t4	CLK Fall Time		7		7		7	3V-0.8V
t5	CLK Rise Time		7		7		7	0.8V-3V
t6a	A31-A3, PTB, W/R#, NENE# Valid Delay	3.5	25	3.5	23	3.5	19	50 pF Load
t6b	BEn#* Valid Delay	3.5	27	3.5	25	3.5	21	50 pF Load
t7	Float Time, All	3.5	40	3.5	30	3.5	25	(Note 1)
t8	ADS#, BREQ, LOCK#, HLDA Valid Delay	3.5	22	3.5	20	3.5	15	50 pF Load
t9	D63-D0 Valid Delay	3.5	38	3.5	35	3.5	31	50 pF Load
t10	Setup Time, All Inputs	13		11		8		(Note 2)
t11a	Hold Time, All Inputs except DATA	4		4		3		(Note 2)
t11b	DATA Hold Time	5		4		3		

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.
* n = 0, 1, ..., 7

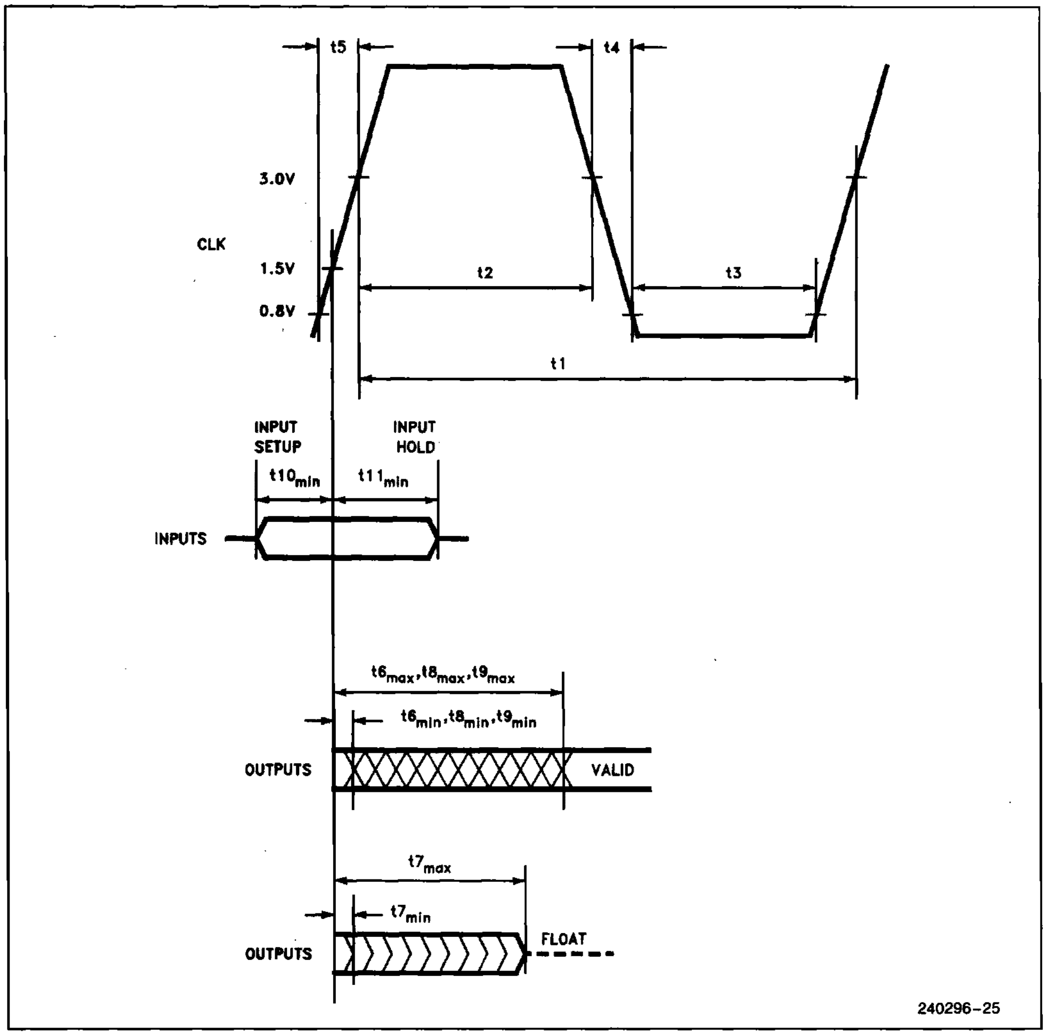
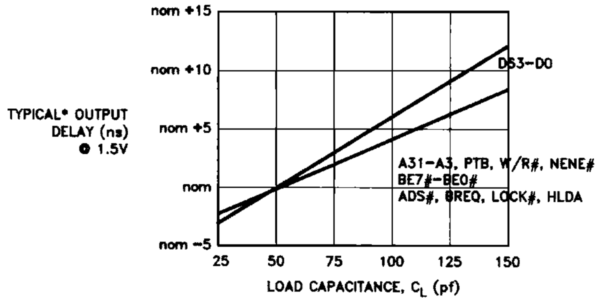


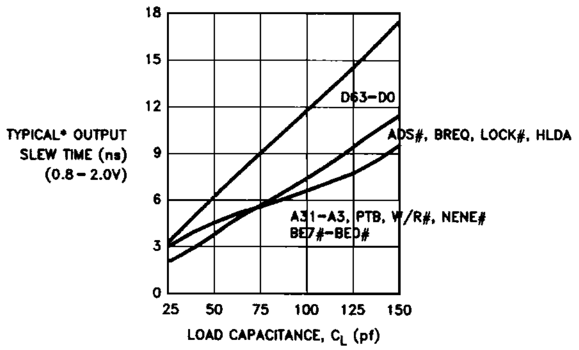
Figure 7.1. CLK, Input, and Output Timings



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.

240296–26

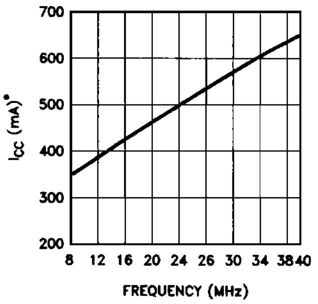
Figure 7.2. Typical Output Delay vs Load Capacitance under Worst-Case Conditions



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

240296–27

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.

240296–28

Figure 7.4. Typical I_{CC} vs Frequency

8.0 INSTRUCTION SET

Key to abbreviations:

For register operands, the abbreviations that describe the operands are composed of two parts. The first part describes the type of register:

c	One of the control registers **fir, psr, epsr, dirbase, db,** or **fsr**
f	One of the floating-point registers: **f0** through **f31**
i	One of the integer registers: **r0** through **r31**

The second part identifies the field of the machine instruction into which the operand is to be placed:

src1	The first of the two source-register designators, which may be either a register or a 16-bit immediate constant or address offset. The immediate value is zero-extended for logical operations and is sign-extended for add and subtract operations (including **addu** and **subu**) and for all addressing calculations.
src1ni	Same as *src1* except that no immediate constant or address offset value is permitted.
src1s	Same as *src1* except that the immediate constant is a 5-bit value that is zero-extended to 32 bits.
src2	The second of the two source-register designators.
dest	The destination register designator.

Thus, the operand specifier *isrc2*, for example, means that an integer register is used and that the encoding of that register must be placed in the *src2* field of the machine instruction.

Other (nonregister) operands are specified by a one-part abbreviation that represents both the type of operand required and the instruction field into which the value of the operand is placed:

#const	A 16-bit immediate constant or address offset that the i860 XR microprocessor sign-extends to 32 bits when computing the effective address.
lbroff	A signed, 26-bit, immediate, relative branch offset.
sbroff	A signed, 16-bit, immediate, relative branch offset.
brx	A function that computes the target address by shifting the offset (either *lbroff* or *sbroff*) left by two bits, sign-extending it to 32 bits, and adding the result to the current instruction pointer plus four. The resulting target address may lie anywhere within the address space.

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.

Other abbreviations include:

.p	Precision specification **.ss, .sd,** or **.dd** (**.ds** not permitted). Refer to Table 8.1.
.r	Precision specification **.ss, .sd, .ds,** or **.dd**. Refer to Table 8.1.
.v	**.sd** or **.dd**. Refer to Table 8.1.
.w	**.ss** or **.dd.** Refer to Table 8.1.
.x	**.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)
.y	**.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)
.z	**.l** (32 bits), or **.d** (64 bits)

Table 8.1. Precision Specification

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

mem.x(address)	The contents of the memory location indicated by *address* with a size of *x*.
PM	The pixel mask, which is considered as an array of eight bits PM[7]..PM[0], where PM[0] is the least significant bit.

8.1 Instruction Definitions in Alphabetical Order

adds *isrc1, isrc2, idest* .... **Add Signed**

```
    idest ← isrc1 + isrc2
    OF ← (bit 31 carry ≠ bit 30 carry)
    CC set if isrc2 < −isrc1 (signed)
    CC clear if isrc2 ≥ −isrc1 (signed)
```

addu *isrc1, isrc2, idest* .... **Add Unsigned**

```
    idest ← isrc1 + isrc2
    OF ← bit 31 carry
    CC ← bit 31 carry
```

and *isrc1, isrc2, idest* .... **Logical AND**

```
    idest ← isrc1 and isrc2
    CC set if result is zero, cleared otherwise
```

andh *#const, isrc2, idest* .... **Logical AND High**

```
    idest ← (#const shifted left 16 bits) and isrc2
    CC set if result is zero, cleared otherwise
```

andnot *isrc1, isrc2, idest* .... **Logical AND NOT**

```
    idest ← not isrc1 and isrc2
    CC set if result is zero, cleared otherwise
```

andnoth *#const, isrc2, idest* .... **Logical AND NOT High**

```
    idest ← not (#const shifted left 16 bits) and isrc2
    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  execute one more sequential instruction
          continue execution at brx(lbroff)
    ELSE  skip next sequential instruction
    FI
```

bla *isrc1ni, isrc2, sbroff* .... **Branch on LCC and Add**

```
          LCC-temp clear if isrc2 < −isrc1ni (signed)
          LCC-temp set if isrc2 ≥ −isrc1ni (signed)
    isrc2 ← isrc1ni + isrc2
    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 [*isrc1ni*] .... **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 isrc1ni
  (The original contents of isrc1ni is used even if the next instruction
  modifies isrc1ni. Does not trap if isrc1ni is misaligned.)
```

bte *isrc1s, isrc2, sbroff* .... **Branch If Equal**

```
IF      isrc1s = isrc2
THEN    continue execution at brx(sbroff)
FI
```

btne *isrc1s, isrc2, sbroff* .... **Branch If Not Equal**

```
IF      isrc1s ≠ isrc2
THEN    continue execution at brx(sbroff)
FI
```

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

```
r1 ← address of next sequential instruction + 4 (+8 in dual mode)
Execute one more sequential instruction
Continue execution at brx(lbroff)
```

calli [*isrc1ni*] .... **Indirect Subroutine Call**

```
r1 ← address of next sequential instruction + 4 (+8 in dual mode)
Execute one more sequential instruction
Continue execution at address in isrc1ni
    (The original contents of isrc1ni is used even if the next instruction
    modifies isrc1ni. Does not trap if isrc1ni is misaligned.
    The register isrc1ni must not be r1.)
```

fadd.p *fsrc1, fsrc2, fdest* .... **Floating-Point Add**

```
fdest ← fsrc1 + fsrc2
```

faddp *fsrc1, fsrc2, fdest* .... **Add with Pixel Merge**

```
fdest ← fsrc1 + fsrc2
Shift and load MERGE register as defined in Table 8.2
```

faddz *fsrc1, fsrc2, fdest* .... **Add with Z Merge**

```
fdest ← fsrc1 + fsrc2
Shift MERGE right 16 and load fields 31..16 and 63..48
```

famov.r *fsrc1, fdest* .... **Floating-Point Adder Move**

```
fdest ← fsrc1
Send fsrc1 through the floating-point adder. (Preserves −0 (minus zero) when fsrc1 is −0. fsrc2
must be coded as f0 by the assembler.)
```

fiadd.w *fsrc1, fsrc2, fdest* .... **Long-Integer Add**
fdest ← *fsrc1* + *fsrc2*

fisub.w *fsrc1, fsrc2, fdest* .... **Long-Integer Subtract**
fdest ← *fsrc1* − *fsrc2*

fix.v *fsrc1, fdest* .... **Floating-Point to Integer Conversion**
fdest ← 64- bit value with low-order 32 bits equal to integer part of *fsrc1* rounded

Floating-Point Load
fld.y *isrc1(isrc2), fdest* .... **(Normal)**
fld.y *isrc1(isrc2)++, fdest* .... **(Autoincrement)**

```
fdest ← mem.y (isrc1 + isrc2)
IF autoincrement
THEN isrc2 ← isrc1 + isrc2
FI
```

Cache Flush
flush *#const(isrc2)* .... **(Normal)**
flush *#const(isrc2)++* .... **(Autoincrement)**

```
Replace block in data cache with address (#const + isrc2).
Contents of block undefined.
IF autoincrement
THEN isrc2 ← #const + isrc2
FI
```

fmlow.dd *fsrc1, fsrc2, fdest* .... **Floating-Point Multiply Low**
fdest ← low-order 53 bits of *fsrc1* mantissa × *fsrc2* mantissa
fdest bit 53 ← most significant bit of mantissa

fmov.r *fsrc1, fdest* .... **Floating-Point Reg-Reg Move**
Assembler pseudo-operation
- **fmov.ss** *fsrc1, fdest* = **fiadd.ss** *fsrc1*, **f0,** *fdest*
- **fmov.dd** *fsrc1, fdest* = **fiadd.dd** *fsrc1*, **f0,** *fdest*
- **fmov.sd** *fsrc1, fdest* = **famov.sd** *fsrc1*, *fdest*
- **fmov.ds** *fsrc1, fdest* = **famov.ds** *fsrc1*, *fdest*

fmul.p *fsrc1, fsrc2, fdest* .... **Floating-Point Multiply**
fdest ← *fsrc1* × *fsrc2*

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

form *fsrc1, fdest* .... **OR with MERGE Register**
fdest ← *fsrc1* OR MERGE
MERGE ← 0

frcp.p *fsrc2, fdest* .... **Floating-Point Reciprocal**
fdest ← 1/*fsrc2* with maximum mantissa error $< 2^{-7}$

frsqr.p *fsrc2, fdest* .... **Floating-Point Reciprocal Square Root**
fdest ← 1/SQRT (*fsrc2*) with maximum mantissa error $< 2^{-7}$

Floating-Point Store
fst.y *fdest, isrc1(isrc2)* .... **(Normal)**
fst.y *fdest, isrc1(isrc2)++* .... **(Autoincrement)**

```
mem.y (isrc2 + isrc1) ← fdest
IF autoincrement
THEN isrc2 ← isrc1 + isrc2
FI
```

fsub.p *fsrc1, fsrc2, fdest* .... **Floating-Point Subtract**
fdest ← *fsrc1* − *fsrc2*

ftrunc.v *fsrc1, fdest* .... **Floating-Point to Integer Conversion**
fdest ← 64-bit value with low-order 32 bits equal to integer part of *fsrc1*

fxfr *fsrc1, idest* .... **Transfer F-P to Integer Register**
idest ← *fsrc1*

fzchkl *fsrc1, fsrc2, fdest* .... **32-Bit Z-Buffer Check**

```
Consider fsrc1, fsrc2, and fdest as arrays of two 32-bit
    fields fsrc1(0)..fsrc1(1), fsrc2(0)..fsrc2(1), and fdest(0)..fdest(1)
    where zero denotes the least-significant field.
PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
    PM [i + 6] ← fsrc2(i) ≤ fsrc1(i) (unsigned)
    fdest(i) ← smaller of fsrc2(i) and fsrc1(i)
OD
MERGE ← 0
```

fzchks *fsrc1, fsrc2, fdest* .... **16-Bit Z-Buffer Check**

```
Consider fsrc1, fsrc2, and fdest as arrays of four 16-bit
    fields fsrc1(0)..fsrc1(3), fsrc2(0)..fsrc2(3), and fdest(0)..fdest(3)
    where zero denotes the least-significant field.
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← fsrc2(i) ≤ fsrc1(i) (unsigned)
    fdest(i) ← smaller of fsrc2(i) and fsrc1(i)
OD
MERGE ← 0
```

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

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

ixfr *isrc1ni, fdest* .... **Transfer Integer to F-P Register**

fdest ← *isrc1ni*

ld.c *csrc2, idest* .... **Load from Control Register**

idest ← *csrc2*

ld.x *isrc1(isrc2), idest* .... **Load Integer**

idest ← *mem.x* (*isrc1* + *isrc2*)

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

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

mov *isrc2, idest* .... **Register-Register Move**

Assembler pseudo-operation

mov *isrc2, idest* = **shl r0,** *isrc2, idest*

mov *const32, idest* .... **Constant-to-Register Move**

Assembler pseudo-operation

adds *l%const32*, **r0**, *idest*
... when *const32* < 0x8000

orh *h%const32*, **r0**, *idest*
or *l%const32, idest, idest*
... when *const32* ≥ 0x8000

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

Assembler pseudo-operation

nop = **shl r0, r0, r0**

or *isrc1, isrc2, idest* .... **Logical OR**

idest ← *isrc1* OR *isrc2*
CC set if result is zero, cleared otherwise

orh *#const, isrc2, idest* .... **Logical OR High**

idest ← (*#const* shifted left 16 bits) OR *isrc2*
CC set if result is zero, cleared otherwise

pfadd.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Add**

fdest ⟵ last stage Adder result
Advance A pipeline one stage
A pipeline first stage ⟵ *fsrc1* + *fsrc2*

pfaddp *fsrc1, fsrc2, fdest* .... **Pipelined Add with Pixel Merge**

fdest ⟵ last stage Graphics result
last stage Graphics result ⟵ *fsrc1* + *fsrc2*
Shift and load MERGE register from last stage Graphics result as defined in Table 8.2

pfaddz *fsrc1, fsrc2, fdest* .... **Pipelined Add with Z Merge**

fdest ⟵ last stage Graphics result
last stage Graphics result ⟵ *fsrc1* + *fsrc2*
Shift MERGE right 16 and load fields 31..16 and 63..48 from last stage Graphics result

pfam.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Add and Multiply**

fdest ⟵ last stage Adder 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

pfamov.r *fsrc1, fdest* .... **Pipelined Floating-Point Adder Move**

fdest ⟵ last stage Adder result
Advance A pipeline one stage
A pipeline first stage ⟵ *fsrc1*

pfeq.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Equal Compare**

fdest ⟵ last stage Adder result
CC set if *fsrc1* = *fsrc2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfgt.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Greather-Than Compare**

(Assembler clears R-bit of instruction)
fdest ⟵ last stage Adder result
CC set if *fsrc1* > *fsrc2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfiadd.w *fsrc1, fsrc2, fdest* .... **Pipelined Long-Integer Add**

fdest ⟵ last stage Graphics result
last stage Graphics result ⟵ *fsrc1* + *fsrc2*

pfisub.w *fsrc1, fsrc2, fdest* .... **Pipelined Long-Integer Subtract**

fdest ⟵ last stage Graphics result
last stage Graphics result ⟵ *fsrc1* − *fsrc2*

pfix.v *fsrc1, fdest* .... **Pipelined Floating-Point to Integer Conversion**

fdest ⟵ last stage Adder result
Advance A pipeline one stage
A pipeline first stage ⟵ 64-bit value with low-order 32 bits
  equal to integer part of *fsrc1* rounded

Pipelined Floating-Point Load

pfld.z *isrc1(isrc2), fdest* .... **(Normal)**
pfld.z *isrc1(isrc2)++, fdest* .... **(Autoincrement)**

fdest ⟵ mem.z (third previous **pfld's** (*isrc1* + *isrc2*))
  (where .z is precision of third previous **pfld.z)**
If autoincrement
THEN *isrc2* ⟵ *isrc1* + *isrc2*
FI

pfle.p *fsrc1, fsrc2, fdest* .... **Pipelined F-P Less-Than or Equal Compare**

Assembler pseudo-operation, identical to **pfgt.p** except that
  assembler sets R-bit of instruction.
fdest ⟵ last stage Adder result
CC clear if *fsrc1* ≤ *fsrc2*, else set
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfmam.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Add and Multiply**
```
    fdest ← last stage Multiplier 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 *fsrc1, fdest* .... **Pipelined Floating-Point Reg-Reg Move**
```
    Assembler pseudo-operation
        pfmov.ss fsrc1, fdest = pfiadd.ss fsrc1, f0, fdest
        pfmov.dd fsrc1, fdest = pfiadd.dd fsrc1, f0, fdest
        pfmov.sd fsrc1, fdest = pfamov.sd fsrc1, fdest
        pfmov.ds fsrc1, fdest = pfamov.ds fsrc1, fdest
```

pfmsm.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Subtract and Multiply**
```
    fdest ← last stage Multiplier 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 *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Multiply**
```
    fdest ← last stage Multiplier result
    Advance M pipeline one stage
    M pipeline first stage ← fsrc1 × fsrc2
```

pfmul3.dd *fsrc1, fsrc2, fdest* .... **Three-Stage Pipelined Multiply**
```
    fdest ← last stage Multiplier result
    Advance 3-Stage M pipeline one stage
    M pipeline first stage ← fsrc1 × fsrc2
```

pform *fsrc1, fdest* .... **Pipelined OR to MERGE Register**
```
    fdest ← last stage Graphics result
    last stage Graphics result ← fsrc1 OR MERGE
    MERGE ← 0
```

pfsm.p *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Subtract and Multiply**
```
    fdest ← last stage Adder 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 *fsrc1, fsrc2, fdest* .... **Pipelined Floating-Point Subtract**
```
    fdest ← last stage Adder result
    Advance A pipeline one stage
    A pipeline first stage ← fsrc1 + fsrc2
```

pftrunc.v *fsrc1, fdest* .... **Pipelined Floating-Point to Integer Conversion**
```
    fdest ← last stage Adder result
    Advance A pipeline one stage
    A pipeline first stage ← 64-bit value with low-order 32 bits
        equal to integer part of fsrc1
```

pfzchkl *fsrc1, fsrc2, fdest* .... **Pipelined 32-Bit Z-Buffer Check**
```
    Consider fsrc1, fsrc2, and fdest, as arrays of two 32-bit
        fields fsrc1(0)..fsrc1(1), fsrc2(0)..fsrc2(1), and fdest(0)..fdest(1)
        where zero denotes the least significant field.
    PM ← PM shifted right by 2 bits
    FOR i = 0 to 1
    DO
        PM [i + 6] ← fsrc2(i) ≤ fsrc1(i) (unsigned)
        fdest(i) ← last stage Graphics result
        last stage Graphics result ← smaller of fsrc2(i) and fsrc1(i)
    OD
    MERGE ← 0
```

pfzchks *fsrc1, fsrc2, fdest* .... **Pipelined 16-Bit Z-Buffer Check**

```
Consider fsrc1, fsrc2, and fdest, as arrays of four 16-bit
    fields fsrc1(0)..fsrc1(3), fsrc2(0)..fsrc2(3), and fdest(0)..fdest(3)
    where zero denotes the least significant field.
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← fsrc2(i) ≤ fsrc1(i) (unsigned)
    fdest(i) ← last stage Graphics result
    last stage Graphics result ← smaller of fsrc2(i) and fsrc1(i)
OD
MERGE ← 0
```

pst.d *fdest, #const(isrc2)* .... **Pixel Store**
pst.d *fdest, #const(isrc2)++* .... **Pixel Store Autoincrement**

```
Pixels enabled by PM in mem.d (isrc2 + #const) ← fdest
Shift PM right by 8/pixel size (in bytes) bits
IF autoincrement
THEN isrc2 ← #const + isrc2
FI
```

shl *isrc1, isrc2, idest* .... **Shift Left**

```
idest ← isrc2 shifted left by isrc1 bits
```

shr *isrc1, isrc2, idest* .... **Shift Right**

```
SC (in psr) ← isrc1
idest ← isrc2 shifted right by isrc1 bits
```

shra *isrc1, isrc2, idest* .... **Shift Right Arithmetic**

```
idest ← isrc2 arithmetically shifted right by isrc1 bits
```

shrd *isrc1, isrc2, idest* .... **Shift Right Double**

```
idest ← low-order 32 bits of isrc1:isrc2 shifted right by SC bits
```

st.c *isrc1ni, csrc2* .... **Store to Control Register**

```
csrc2 ← isrc1ni
```

st.x *isrc1ni, #const(isrc2)* .... **Store Integer**

```
mem.x (isrc2 + #const) ← isrc1ni
```

subs *isrc1, isrc2, idest* .... **Subtract Signed**

```
idest ← isrc1 − isrc2
OF ← (bit 31 carry ≠ bit 30 carry)
CC set if isrc2 > isrc1 (signed)
CC clear if isrc2 ≤ isrc1 (signed)
```

subu *isrc1, isrc2, idest* .... **Subtract Unsigned**

```
idest ← isrc1 − isrc2
OF ← NOT (bit 31 carry)
CC ← bit 31 carry
(i.e.   CC set if isrc2 ≤ isrc1 (unsigned)
        CC clear if isrc2 > isrc1 (unsigned)
```

trap *isrc1ni, isrc2, idest* .... **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.
Enable interrupts after bus is unlocked.

xor *isrc1, isrc2, idest* .... **Logical Exclusive OR**

```
idest ← isrc1 XOR isrc2
CC set if result is zero, cleared otherwise
```

xorh *#const, isrc2, idest* .... **Logical Exclusive OR High**

```
idest ← (#const shifted left 16 bit) XOR isrc2
CC set if result is zero, cleared otherwise
```

Table 8.2. 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 32 bits long and begin on a four-byte boundary. When operands are registers, the register encodings shown in Table 8.3 are used. There are two general core-instruction formats, REG-format and CTRL-format, as well as a separate format for floating-point instructions.

8.2.1 REG-FORMAT INSTRUCTIONS

Within the REG-format are several variations as shown in Figure 8.1. Table 8.4 gives the encodings for these instructions. One encoding is an escape code that defines yet another variation: the core escape instructions. Figure 8.2 shows the format of this group, and Table 8.5 shows the encodings.

In these 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.

Table 8.3. Register Encoding

Register	Encoding
r0	0
.	.
.	.
.	.
r31	31
f0	0
.	.
.	.
.	.
f31	31
Fault Instruction	0
Processor Status	1
Directory Base	2
Data Breakpoint	3
Floating-Point Status	4
Extended Process Status	5

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. For **fld**, **fst**, **pfld**, and **pst**, 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.

For **flush**, bits one and two must be zero.

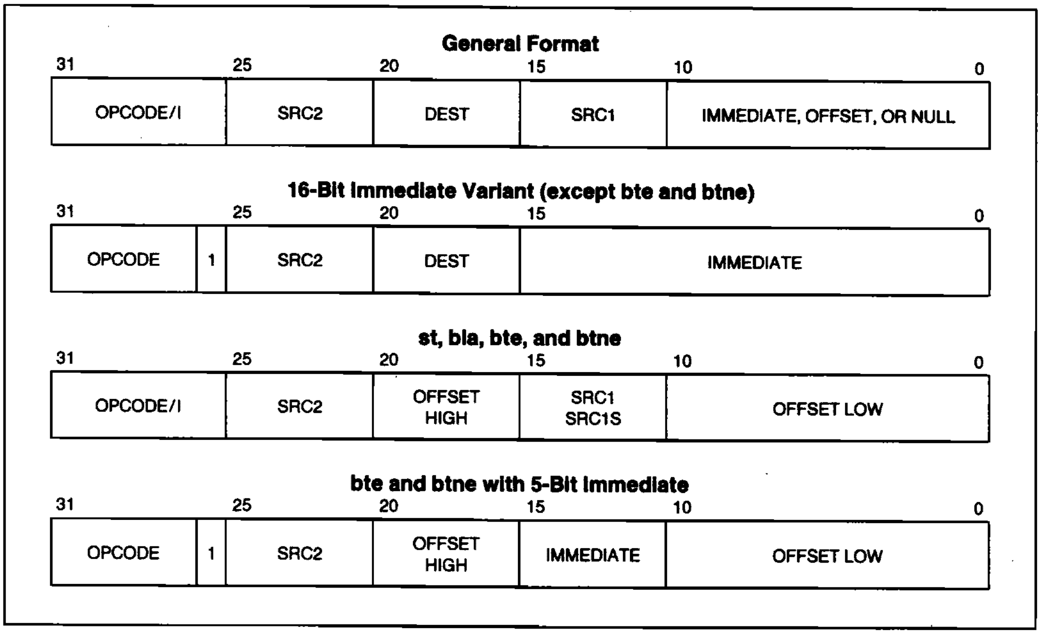


Figure 8.1. REG-Format Variations

Table 8.4. 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

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

*reserved (must be set to zero by assemblers)

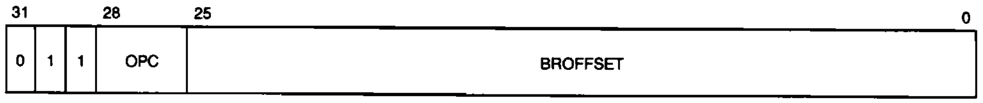
Figure 8.2. Core Escape Instruction Format

Table 8.5. 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

8.2.2 CTRL-FORMAT INSTRUCTIONS

The CTRL instructions do not refer to registers, so instead of the register fields, they have a 26-bit relative branch offset. Figure 8.3 shows the format of these instructions and Table 8.6 defines the encodings.



BROFFSET is a signed 26-bit relative branch offset.

Figure 8.3. CTRL Instruction Format

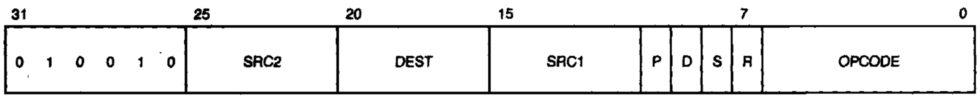
Table 8.6. CTRL-Format Opcodes

		28		26
	(reserved)	0	0	0
	(reserved)	0	0	1
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**

8.2.3 FLOATING-POINT INSTRUCTIONS

The floating-point instructions also constitute an escape series. All these instructions begin with the bit sequence 010010. Figure 8.4 shows the format of the floating point instructions, and Table 8.7 gives the encodings. Within the dual-operation instructions is a subcode DPC whose values are given in Table 8.8 along with the mnemonic that corresponds to each.



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

Figure 8.4. Floating-Point Instruction Encoding

Table 8.7. Floating-Point Opcodes

		6						0
pfam	Add and Multiply*							
pfmam	Multiply with Add*	0	0	0	DPC			
pfsm	Subtract and Multiply*							
pfmsm	Multiply with Subtract*	0	0	1	DPC			
(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
pfmul3.dd	3-Stage Pipelined Multiply	0	1	0	0	1	0	0
(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
(p)famov	Adder Move	0	1	1	0	0	1	1
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.

NOTE:
All opcodes not shown are **reserved**.

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

Table 8.8. 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	Intel-Reserved							

*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

i860 XR 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 **ld** 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 until first READY# returned (for 32- or 64-bit read cycles) or until second READY# returned (for 128-bit **fld.q** read cycles)
call, calli, ixfr, fxfr, ld.c, or **st.c** and data cache load miss processing in progress	One plus number of clocks until first READY# returned (for 64-bit read cycles) or until second READY# returned (for 128-bit **fld.q** read cycles)
ld/st/pfld/fld/fst and data cache load 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. (*Dest* of **call** and **calli** is **r1**.)	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 **addu/adds/subu/subs/pfeq/pfle/pfgt**	One clock
Fsrc1 of multiplier operation refers to result of previous operation	One clock
Floating-point operation or graphics-unit instruction 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,** two minus the number of instructions (or dual-mode pairs) already executed after the scalar operation. If the scalar operation is **fmul.dd,** three minus the number of instructions (or dual-mode pairs) executed after it. Add one if either or both of these two situations occur: 1. There is an overlap between the result register of the previous scalar operation and the source of the floating-point operation, and the destination precision of the scalar operation is different than the source precision of the floating-point operation. 2. The floating-point operation is pipelined and its destination is not **f0.** There is no delay if the result is negative.
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
st, pst or **fst** miss, **ld** miss, or **flush** with modified block when store path full (two stores or one 256-bit write-back internally waiting for bus plus external bus pipeline full)	One plus the number of clocks until READY# active on next 64-bit write cycle or second READY# of next 128-bit write cycle.
ld, fld, pfld, st, pst, or **fst** when address path full (one address internally waiting for bus plus external bus pipeline full)	Number of clocks until next nonrepeated address can be issued (i.e., an address that is not the 2nd–4th cycle of a cache fill, the 2nd–8th cycle of a CS8 mode instruction fetch, nor the 2nd cycle of a 128-bit write)
ld/fld following **st/fst** hit	One clock

Freeze Condition	Delay
Delayed branch not taken	One clock
Nondelayed branch taken:	
bc, bnc	One clock
bte, btne	Two clocks
Indirect branch **bri** or call **calli**	One clock
st.c	Two clocks
Result of graphics-unit instruction (other than **fmov.dd**) 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**	Three clocks minus the number of instructions between the two **flush** instructions. There is no delay if the result is negative.
fst or **pst** 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 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

  Note that this is not the same as specifying at which instructions faults may be reported. A result exception is reported on the subsequent floating-point instruction, **pst, fst,** or sometimes **fld, pfld**, and **ixfr**.

  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 i860 XR 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, *fsrc1* should not be the same as the *fdest* of the prior operation.
  5. The *fdest* 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. (For **call** and **calli**, the destination is **r1**.)

7. When the prior operation is scalar and multiplier *op1* is *fsrc1*, *fsrc2* should not be the same as the *fdest* of the prior operation.
8. When the prior operation is scalar, *fsrc1* and *fsrc2* of the current operation should not be the same as *fdest* of the prior operation.
9. A **pfld** should not immediately follow a **pfld**.

- 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 **brl** 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 **brl** is executed.
  c. If *fdest* is not zero, *fsrc1* must not be the same as *fdest*.
  d. When *fsrc1* goes to the multiplier *op1*, KR, or KI, *fsrc1* must not be the same as *fdest*.
  e. If *fdest* is not zero, *fsrc1* and *fsrc2* must not be the same as *fdest*.
  f. *isrc1* must not be the same as *isrc2* for the autoincrementing form of this instruction.
  g. *isrc1* must not be the same as *isrc2*.
- Core and Floating-Point Instruction Interaction in Dual-Instruction Mode
  1. If one of the branch-on-condition instructions **bc** or **bnc** is paired with a floating-point compare, the branch tests the value of the condition code prior to the compare.
  2. If an **ixfr, fld,** or **pfld** loads the same register as a source operand in the floating point instruction, the floating-point instruction references the register value before the load updates it.
  3. An **fst** or **pst** that stores a register that is the destination register of the companion pipelined floating-point operation will store the result of the companion operation.
  4. When the core instruction sets CC and the floating-point instruction is **pfgt, pfle,** or **pfeq,** CC is set according to the result of **pfgt, pfle,** or **pfeq.**
  5. When a **trap** instruction causes a trap in dual-instruction mode, the floating-point instruction has neither completed execution nor has updated the FT bit or any result status bits. This is not a problem when the **trap** is inserted by a debugger, because the **trap** is replaced by the original instruction, and the dual-mode pair is reexecuted. However, when the **trap** is programmed, the trap handler must avoid reexecuting the **trap** by returning to user code at the address in **fir** + 8. In this case, the trap handler must emulate the floating-point instruction before returning to the user code. Emulation of the instruction must include all side-effects (for example, the effect of its D-bit, effect on the pipelines, and effect on FT and result-status bits), just as if the instruction had been executed by the processor in the original context.
  6. In dual-instruction mode, when the **intovr** instruction causes a trap, the floating-point companion instruction has completely finished execution before the trap is taken.

- Programming Restrictions for Dual-Instruction Mode

1. The result of placing a core instruction in the low-order 32 bits or a floating-point instruction in the high-order 32 bits is not defined (except for **shrd r0, r0, r0** which is interpreted as **fnop**).
2. A floating-point instruction that has the D-bit set must be aligned on a 64-bit boundary (i.e., the three least-significant bits of its address must be zero). This applies as well to the initial 32-bit floating-point instruction that triggers the transition into dual-instruction mode, but does not apply to the following instruction.
3. When the floating-point operation is scalar and the core operation is **fst** or **pst,** the store should not reference the result register of the floating-point operation. When the core operation is **pst,** the floating-point instruction cannot be **(p)fzchks** or **(p)fczhkl.**
4. When the core instruction of a dual-mode pair is a control-transfer operation and the previous instruction had the D-bit set, the floating-point instruction must also have the D-bit set. In other words, an exit from dual-instruction mode cannot be initiated (first instruction pair without D-bit set) when the core instruction is a control-transfer instruction.
5. When the core operation is a **ld.c** or **st.c,** the floating-point operation must be **d.fnop.**
6. When the floating-point operation is **fxfr,** the core instruction cannot be **ld, ld.c, st, st.c, call ixfr,** or any instruction that updates an integer register (including autoincrement indexing). Furthermore, the core instruction cannot be a **fld, fst, pst,** or **pfld** that uses as *isrc1* or *isrc2* the same register as the *idest* of the **fxfr.** Additionally, in dual instruction mode, **fxfr** may not be used in a branch delay slot if its destination register is referenced by the preceding branch instruction.
7. A **bri** must not be executed in dual-instruction mode if any trap bits are set.
8. When the core operation is **bc.t** or **bnc.t,** the floating point operation cannot be **pfeq** or **pfgt.** The floating-point operation in the sequentially following instruction pair cannot be **pfeq** or **pfgt,** either.
9. A transition to or from dual-instruction mode cannot be initiated on the instruction following a **bri.**
10. An **ixfr, fld, or pfld** cannot update the destination of the companion floating-point instruction (unless the destination is **f0** or **f1**) or of the following pipelined floating-point instruction (regardless of its destination register). No overlap of register destinations is permitted; for example, the following instructions must not be paired:

```
// Illegal case 1
    d.fmul.ss  f9, f10, f5
      fld.d     address, f4
        ; Overlaps f5

// Illegal case 2
    d.fmul.ss  f0, f0, f3
      fld.q     address, f0
        ; Overlaps f3

// Illegal case 3
    d.fmul.ss  f9, f10, f11
      fld.l     address, f5
    d.pfadd.ss fx, fx, f4
        ; Overlaps f5, if last
          stage result is double-
          precision
```

11. During a locked sequence, a transition to or from dual-instruction mode is not permitted.

2

Table 8.9 Instruction Characteristics

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, g
bnc	E					
bnc.t	E	D				a
br	E	D				a
bri	E	D				a, b
bte	E					
btne	E					
call	E	D			6	a
calli	E	D			6	a
fadd.p	A			SE, RE		
faddp	G				8	
faddz	G				8	
famov.r	A			SE, RE		
fiadd.z	G				8	
fisub.z	G				8	
fix.p	A			SE, RE		
fld.y	E			DAT	2, 3	f
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	f
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			
pfadd.p	A	P		SE, RE		
pfaddp	G	P			8	e

Table 8.9 Instruction Characteristics (Continued)

Instruction	Execution Unit	Pipelined? Delayed?	Sets CC?	Faults	Performance Notes	Programming Restrictions
pfaddz	G	P			8	e
pfam.p	A&M	P		SE, RE	7	d
pfamov.r	A	P		SE, RE		
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		DAT	2, 9	f
pfmam.p	A & M	P		SE, RE	7	d
pfmsm.p	A & M	P		SE, RE	7	d
pfmul.p	M	P		SE, RE	4	c
pfmul3.dd	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		f
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			

9.0 FUNCTIONAL CHARACTERISTICS

The following characteristics of the 80860XR Microprocessor are additions to revision 2 of the i860™ Microprocessor Family Programmers Reference Manual, Intel order number 240875-002. This document will also appear in the 1993 revision of the Multimedia and Supercomputing Processors data book, Intel order number 272084-002.

Four steppings of the 80860XR Microprocessor are covered in this document; B2, B3, C1 and D0. Each of the characteristics listed pertains to one or more of the steppings. Which steppings are affected appears in the left hand columns before the number and title of each functional characteristic and are defined as follows:

X indicates that the characteristic is affected the given stepping.

F indicates that the characteristic was fixed in the given stepping and no workaround is required.

— indicates that the erratum was fixed in a previous stepping and no workaround is required.

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	1. *Trap Handler User/Supervisor Fault* *Problem:* When returning from the trap handler, a false data access trap may occur on the **ld** following the **brl**. This can happen in either of the two following situations: 1. The target of the **brl** is an I-cache hit; the instruction after the target is an I-cache and TLB miss; and the page from which **ld** is loading is a supervisor-level page. 2. HOLD is asserted after the instruction fetch for the **bri** target is begun and before the external bus cycle for the **ld** begins. *Workaround:* The page on which the trap handler saves the registers must be made readable in User mode, i.e. both levels of page tables entries for that page must have U = 1.
X	X	F	—	2. *HOLD/HLDA in Multi-Transfer Stores* *Problem:* If HOLD is asserted between the first and second transfers of a 128-bit store or between any transfers of a cache writeback to memory, then upon leaving the bus hold, the data for the next write cycle is issued on the bus one clock *after* ADS# is driven low for that write cycle. *Workaround:* Since the write data lags ADS# by one clock cycle, systems should not use HOLD with zero-wait-state write cycles.

X	X	F	—	3. ***Flush** with Paging after (f) ld*
				Problem: In systems using paging, if an **(f) ld** causes a data cache writeback with a TLB miss and the next data access instruction is a **flush** instruction, then the data in the cache block being flushed will be corrupted and the processor may hang.
				NOTE:
				For more errata relating to the **flush** instruction see errata #23 and #43.
				*Workaround: If the workaround for Erratum #43 (**flush** with HOLD) is used, no workaround for this erratum is necessary, even if **flush** with paging is used.*
				If the workaround for Erratum #43 is *not* used, *both* of the two following workarounds are required:
				1. In order to prevent an **(f) ld**-induced cache writeback from immediately preceding the **flush** instruction, systems using paging must execute the last **(f) ld** before the **flush** instruction twice. Thus, if that **(f) ld** were to cause a writeback, the writeback will be executed at the first of the two **(f) lds** and not at the **(f) ld** immediately preceding the **flush**.
				2. Also, in order to prevent this problem from occurring when returning from the trap handler to the flush routine, the **(f) ld** in the delay slot of the **brl** at the end of the trap handler must be executed twice. However, since the **brl** only has one delay slot, the first of the two **(f) lds** from that address must precede the **brl** and must be discarded to r0 so as not to prematurely restore r1, which would corrupt the destination of the **brl**.
				The following is an example of the last few instructions of an appropriately modified trap handler:
				•
				•
				•
				`ld.l stack_area, r0`
				`bri      r1`
				`ld.l stack_area, r1`
				•
				•
				•

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	4. *NENE# Incorrectly Asserted After Incomplete HOLD* *Problem:* If HOLD is asserted and then deasserted before HLDA is asserted, then HLDA can be asserted for one clock cycle; on the following bus cycle, NENE# will be incorrectly asserted if that cycle would have been next-near given no hold state interruption. Although NENE# will usually still be valid at this point (depending on memory system design), its assertion does not follow the correct protocol, which specifies that NENE# should not be asserted on the next bus cycle after HLDA is deasserted. *Workaround:* Do not assert and then deassert HOLD before HLDA is asserted, and do not reassert HOLD until HLDA is deasserted for the previous HOLD.
X	X	F	—	5. *Multiprocessor A-bit Settings Cause Address Corruption* *Problem:* In systems using paging, the correct protocol for setting the A-bit (Accessed bit) during TLB miss processing is as follows: 1. Fetch PTE from memory with LOCK# deasserted and check the A-bit status. 2. If the A-bit is clear, then refetch PTE with LOCK# asserted. 3. Write PTE back to memory with the A-bit set and LOCK# deasserted. However, if between steps 1 and 2 the A-bit is set by another processor, then the TLB address transformation will be corrupted and the processor may hang. *Workaround:* In multiprocessor systems, the A-bits in shared page table entries must be set to 1 when the pages are allocated in order to avoid locked read/write A-bit set cycles for these pages.
X	X	F	—	6. *Incorrect Floating Point Result Trap in Multiplier Unit* *Problem:* When a sequence of pipelined single-precision multiplier operations is followed by a pipelined double-precision multiplier operation, the next-to-last single-precision operation may cause a result exception trap even though the result will be correctly discarded. This erroneous trap will only occur if the instruction executed immediately after the first double-precision multiply operation is 1) **fst**; 2) **pst**; 3) **fld**, **pfld**, or **ixfr** into a register or register set overlapping the **rdest** of the first double-precision multiplier instruction; or 4) any floating point instruction other than a multiplier operation. *Workaround:* In the situation described above, the instruction immediately following the first double-precision multiply operation must not be 1) **fst**; 2) **pst**; 3) **fld**, **pfld**, or **ixfr** into a register or register set overlapping the **rdest** of the first double-precision multiplier instruction; or 4) any floating point instruction other than a multiplier operation. In addition, if the instruction cache is enabled, then programs must not use a delayed branch whose delay slot instruction is the first pipelined double-precision multiplier operation when the first instruction at the target address is an **fst**, **pst**, **fld**, **pfld**, **ixfr**, or floating point instruction which could cause the spurious floating point trap.

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	7. ***fxfr** Result Exception Trap Corrupts Destination Register* *Problem:* An **fxfr** that reports a result exception may corrupt its destination register before control is transferred to the trap handler. In DIM, if either source of the core operation paired with the **fxfr** is the same as the **fxfr**'s destination register, then when the program returns from the trap handler, the core operation paired with the **fxfr** will use the incorrect value in that register. Also, in both SIM and DIM, if the **fxfr** is used in the delay slot of a **bri**, **calli**, or **bla** which references the same register as the **fxfr**'s destination, then upon returning from the trap handler, the branch will use the corrupted value in that register. Note that when this erratum is fixed, it will be permissible to use **fxfr** in the delay slot of a branch which references the **fxfr**'s destination in SIM, but it will still be illegal to use it in such a delay slot in DIM. *Workaround:* In DIM neither of the sources of a core operation paired with an **fxfr** may be the same as the **fxfr**'s destination register. In both SIM and DIM the **fxfr** instruction must not be used in the delay slot of a **bri**, **calli**, or **bla** which references the **fxfr**'s destination register. For example, **bri r5** **fxfr f2,r5** must not be used.
X	X	F	—	8. ***pfladd.ss/pflsub.ss/pfmov.ss** Results Corrupted by other Pipelined Graphics Instructions* *Problem:* When the next graphics unit instruction after a **pfladd.ss**, **pflsub.ss**, or **pfmov.ss** is a **pfzchkl**, **pfzchks**, **pfaddp**, **pfaddz**, or **pform**, the result of the **pfladd.ss**, **pflsub.ss**, or **pfmov.ss** may be incorrect. Note that this problem only occurs with *single* precision **pfladd**, **pflsub**, and **pfmov** instructions. *Workaround:* Flush the graphics unit pipeline between a **pfladd.ss**, **pflsub.ss**, or **pfmov.ss** and a following **pfzchkl**, **pfzchks**, **pfaddp**, **pfaddz**, or **pform** with a **pfladd.ss f0, f0, fn**.
X	X	F	—	9. *Multiplier Pipeline Result Not Discarded at Precision Transition* *Problem:* In correct execution, when a sequence of pipelined single-precision multiplier operations is followed by a pipelined double-precision multiplier operation, the result of the next-to-last single-precision multiply is discarded. However, if a floating point trap is reported during the two clocks after the first pipelined double-precision multiply, then the result of the next-to-last pipelined single-precision multiply may not be discarded, and the multiplier pipeline may not advance correctly. As a result, the next two double-precision pipelined multiplies or three single-precision pipelined multiplies after the first pipelined double-precision multiply may receive incorrect data. *Workaround:* Any one of the following workarounds is sufficient: 1. Flush the pipeline with two **pfmul.ss f0,f0,fn** instructions between the single- and double-precision multiplier operations. 2. Disable floating point traps after the last pipelined single-precision multiply; enable them again after the first pipelined double-precision multiply. 3. Ensure that the two instructions (or two pairs of instructions, in DIM) following the first pipelined double-precision multiply are instructions which cannot report a floating point trap. Those which cannot report floating point traps are all core instructions except for **fst**, **pst**, and an **fld**, **pfld**, or **ixfr** whose destination register (or register set) overlaps the destination of the first pipelined double-precision multiply.

Stepping				Description
B2	B3	C1	D0	

X	X	F	—	10. ***flush** with Paging May Corrupt Memory Data*
				Problem: In systems using paging, if a flush routine writeback or instruction fetch causes a TLB miss, then modified data at the target address of one **flush** instruction may also be written to the target address of the next **flush** instruction, and the chip may hang.
				*Workaround: If the workaround for Erratum #43 (**flush** with HOLD) is used, no workaround for this erratum is necessary, even if **flush** with paging is used.* If the workaround for Erratum #43 is *not* used, all three of the following workarounds must be implemented:
				1. The old **flush** inner loop:
				`D_FLUSH_LOOP:`
				`        bla   Rx,Ry,D_FLUSH_LOOP`
				`        flush   32 (Rw) ++`
				must be replaced with a new **flush** inner loop:
				`D_FLUSH_LOOP:`
				`        ixfr r0,f0`
				`        bla   Rx,Ry,D_FLUSH_LOOP`
				`        flush   32(Rw)++`
				`        ixfr r0,f0`
				*In addition, floating point traps must be disabled during execution of the **ixfrs**.*
				2. The D_FLUSH_LOOP label must be aligned on a 32-byte boundary by preceding the label with the **.align 32** assembler directive.
				3. External interrupts must be disabled before entering the flush routine and reenabled after exiting the routine by clearing and setting the IM bit of the PSR.

X	X	F	—	11. *KNF, DIM, DS, BW, and BR Bits Not Write Protected* *Problem:* The KNF, DIM, DS, BW, and BR bits of the PSR are not write protected in user mode. *Workaround:* Software should not assume write protection of these bits in user mode.
X	X	F	—	12. *TLB Miss Processing Address Corrupted After **ld.c** from **epsr*** *Problem:* If a TLB miss resulting from an instruction fetch or cache writeback cycle follows a **ld.c** from the **epsr**, the address for the TLB miss processing may be corrupted. *Workaround:* Every **ld.c epsr, rn** should be immediately preceded by an **ixfr r0, f0**, *and floating point traps must be disabled during execution of the **ixfr**.* In addition, the **ixfr** must be aligned on a 32 byte boundary by preceding it with the assembler directive **.align 32**.

Stepping				Description
B2	B3	C1	D0	

X	X	F	—	13. ***fmlow.dd** Incorrectly Causes Floating Point Traps*
				Problem: Although **fmlow.dd** is not supposed to cause floating point traps or update the result-status bits of the FSR, it can trigger a floating point trap in the following case:
				`fmlow.dd  fx,fy,fz  //Causes result underflow or overflow`
				`•`
				`•    //Any sequence of non-pfmul instructions`
				`•`
				`pfmul.xx  fa,fb,fc  //Any precision pfmul`
				`•`
				`•    //Any sequence of non-pfmul instructions`
				`•`
				`pfmul.dd  fl,fm,fn  //Double precision pfmul only`
				`pfadd.ss  fd,fe,ff  //Reports erroneous floating point trap`
				`                    //(Any FP trap-reporting instruction)`
				The instruction immediately following the **pfmul.dd f1, fm, fn** which reports the trap can be any floating point trap-reporting instruction other than another multiplier unit instruction.
				Although a trap will be reported and the FT bit will be set (FT = 1) in the PSR, no floating point result status bits will be set in the FSR.
				Workaround: If the trap handler finds FT set in the PSR but no result-status bits set in the FSR, then it should just return back to user code.

X	X	F	—	14. *Byte Enable Code Wrong on First Fetch in CS8 Mode* *Problem:* During the first CS8 mode instruction byte fetch of a sequence of fetches within a 32 byte block boundary, the byte enable pattern BE7#:BE0# is hex FF rather than hex AF as it should be. *Workaround:* Interpret the byte enable code hex FF to indicate the first instruction byte fetch in a CS8 mode instruction fetch sequence.
X	X	F	—	15. ***frcp, frsqr** After **pfmul** May Corrupt Data* *Problem:* If a **pfmul.ss** or **pfmul.sd** immediately precedes an **frcp** or **frsqr** which in turn is immediately followed by a scalar floating point instruction other than a graphics unit instruction, then the result of the scalar floating point instruction may be corrupted. Note that the **pfmul.ss/pfmul.sd** preceding the **frcp/frsqr** could reside in the delay slot of a branch. *Workaround:* Either one of the following two workarounds will suffice: 1. Separate the **frcp/frsqr** from the preceding **pfmul.ss/pfmul.sd** with a **nop** or other instruction. 2. Replace the **pfmul.ss/pfmul.sd** immediately preceding the **frcp/frsqr** with a **pfmul.dd**. The replacement will not affect any result, since the **pfmul** data entering the multiplier pipeline should be discarded because the **frcp/frsqr** is scalar.

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	16. *KNF (Kill Next Floating-point) Ineffective With **brl/calli** in DIM* *Problem:* When an instruction pair containing a **bri** or **calli** is executed in dual mode with the KNF bit set in the PSR, the floating point instruction paired with the **bri** or **calli** is executed, although it should not be. *Workaround:* Either one of the following two workarounds will suffice. The first is a trap-handler workaround, whereas the second is implemented in the compiler. 1. The trap handler should not return to a dual pair containing a **bri** or **calli** with KNF set in the PSR. Instead, if the floating point operation paired with the **bri/calli** should not be reexecuted, then the trap handler should execute the **bri/calli** in single instruction mode before transitioning to dual mode on the next pair if appropriate. The following two rules implement this workaround: a. If the trap handler finds the DIM bit set and the DS bit clear, then it should clear DIM, set DS, and return to the instruction at the address in **fir** + 4 with KNF clear. b. If the trap handler finds both the DIM and DS bits set, then it should clear both bits and return to the instruction at the address in **fir** + 4 with KNF clear. 2. In dual instruction mode, pair every **bri** or **calli** with an **fnop** or a graphics unit instruction. Since the **bri** or **calli** cannot cause a data-access fault and the **fnop** or graphics unit instructions cannot cause a source exception, these dual pairs will never require the use of KNF.

X	X	F	—	17. ***fld/pfld/ixfr** May Not Report Floating Point Result Exception*
				Problem: In correct operation an **fld**, **pfld**, or **ixfr** whose destination register (or register set) overlaps the destination of a preceding scalar instruction which caused a floating point result exception (RE) should always report that RE, unless a floating-point operation, **pst**, or **fst** has already done so. However, if a trap other than a floating-point trap occurs between the RE-causing instruction and the **fld/pfld/ixfr**, the **fld/pfld/ixfr** may not report the RE. Instead, the trap will be reported by the next floating point operation, **fst**, or **pst** after the **fld/pfld/ixfr**. The problem only occurs if the trap handler uses any floating point or graphics operations to handle the trap that occurred between the RE-causing instruction and the **fld/pfld/ixfr**. If the trap handler does not use floating point or graphics operations to handle the non-floating point trap, then execution will be correct. For example:
				`fmul.dd fx,fy,fs // Causes an FP result exception`
				●
				●
				●
				`<any non-floating point trap to trap handler requiring floating point pipeline manipulations>`
				●
				●
				`fld.l   0 (r0),fz // Should report RE but may not`
				●
				●
				`fmul.ss f0,f0,f0 // Will report the result exception if`
				`                 // fld.l does not, even though fz has`
				`                 // been overwritten by the fld.l`

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	17. (Continued) *Workaround:* If floating point traps are enabled (FTE = 1) and the trap handler uses any floating point or graphics operations to handle traps, then when handling any trap, the trap handler should check the result status bits of the FSR register. (The **ld.c fsr** which saves the result status bits of the FSR register must occur on the third or later instruction of the trap handler.) If any of the FSR result status bits is set, indicating a floating point RE, then the trap handler should handle the floating point RE as well as the other trap that caused the branch to the trap handler routine.
X	X	F	—	18. ***pfld** With HOLD May Corrupt Data* *Problem:* In systems using HOLD, if a **pfld** miss is followed by a **pfld** hit and the bus cycle for the **pfld** miss is delayed due to a HOLD request and acknowledge, then the **pfld** hit data fetch is completed before the **pfld** miss data fetch. As a result the data FIFO receives data out of order, and the data is corrupted. Note that this problem can only occur if **pfld** data has already been loaded into the data cache with **fld**. *Workaround:* Either one of the following two workarounds will suffice: 1. Do not use **pfld** with systems which use HOLD. 2. Only use **pfld** on data which cannot reside in the data cache. This workaround may be implemented by making pages containing **pfld** data noncacheable.
X	X	F	—	19. *Core Operation May Overwrite CC Bit Set by **pfgt/pfle/pfeq** in DIM* *Problem:* In correct operation, if a **pfgt**, **pfle**, or **pfeq** is paired with an ALU or logical core instruction in dual instruction mode, then the CC bit should be set according to the result of the **pfgt/pfle/pfeq**, and not according to the result of the ALU or logical core instruction. However, if a floating point source exception is reported on the **pfgt/pfle/pfeq**, then the trap handler will update the CC bit for the **pfgt/pfle/pfeq**, return to the user code, and reexecute the dual pair with KNF set. The ALU/logical core instruction may then modify the CC bit from its correct value. *Workaround:* Either one of the following two workarounds will suffice. The first is a trap-handler workaround, whereas the second is implemented in the compiler. 1. After handling a **pfgt/pfle/pfeq** floating point source exception, the trap handler should not return with KNF set to a dual pair containing an ALU or logical core operation. Instead, it should emulate the core operation, except for the update of the CC bit, and then return to the **next** instruction pair with KNF clear and the DIM and DS bits modified according to the rules below. If the dual pair containing the **pfgt/pfle/pfeq** is in the delay slot of a delayed branch, then the trap handler must resume at the branch target. *Rules for modifying DIM and DS upon returning to the subsequent instruction pair:* a. If the trap handler finds the DIM bit set, the DS bit clear, and the D-bit of the floating point instruction set, then it should leave both the DIM and DS bits as they are and return to the next instruction pair. b. If the trap handler finds the DIM bit set, the DS bit clear, and the D-bit of the floating point instruction clear, then it should leave the DIM bit set, set the DS bit, and return to the next instruction pair. c. If the trap handler finds both the DIM and DS bits set, then it should clear both bits and return to the next instruction pair. 2. In dual instruction mode do not pair a **pfgt/pfle/pfeq** with a core instruction that can modify the CC bit, i.e. an ALU or logical operation.

Stepping				Description
B2	B3	C1	D0	

X	X	F	—	20. *Register Bypass of f1 Does Not Work*
				Problem: If a single precision floating point instruction discards data into destination register **f1** and the same instruction (in pipelined mode) or next instruction (in scalar mode) references **f1** as a source, then the data being discarded into **f1** will be mistakenly bypassed to the instruction using **f1** as a source. Since **f1** is supposed to be a read-only register whose value is zero, incorrect program execution may result.
				Example 1
				`fmul.ss  fa,fb,f1`
				`fadd.ss  f1,fc,fd // op_1 will be fa*fb, rather than 0`
				`                  // fd = (fa*fb)+fc, rather than fc`
				Example 2
				`pfmul.ss fa,fb,fx`
				`pfmul.ss fe,ff,fx`
				`pfmul.ss fe,ff,fx`
				`pfmul.ss fc,f1,f1 // op_2 will be fa*fb, rather than 0`
				`pfmul.ss fe,ff,fx`
				`pfmul.ss fe,ff,fx`
				`pfmul.ss fe,ff,fk // fk = fa*fb*fc, rather than 0`
				Workaround: Do not use **f1** as the destination register when discarding a result. Discard the result to register **f0** instead.

X	X	X	F	21. *Multiplier Pipeline Not Cleared of Result Exception by **frcp/frsqr***
				Problem: If a result exception is in the first stage of the multiplier pipeline and a **frsqr/frcp** is executed, the result exception should be cleared from the pipe but is not. As a consequence, a result exception with no result-status bits set will occur.
				Example:
				`pfmul.dd  f2,f2,f8  // Causes result exception`
				`frsqr.dd  f4,f6`
				`pfmul.dd  fx,fx,fy`
				`pfmul.dd  fa,fb,fc`
				`pfiadd.dd fq,fr,fs  // Triggers RE reporting`
				Workaround: If the trap handler finds FT set in the PSR but no result-status bits set in the FSR, then it should just return back to user code.

X	X	F	—	22. *Short HOLD/HLDA Sequences With Paging Can Cause Data Corruption* *Problem:* In systems using HOLD and paging, internal data corruption may occur if HLDA is asserted for 3 clocks or fewer. This problem can only occur if all of the following conditions are met: 1. HOLD is asserted during the 1st, 2nd, or 3rd fetch of an instruction fetch sequence. 2. The last bus cycle before HLDA is asserted is an instruction fetch of a store instruction which causes a TLB hit and a data-access fault. 3. HOLD is deasserted on the 1st, 2nd, or 3rd clock during which HLDA is asserted. *Workaround:* Either HOLD must never be asserted during an instruction fetch sequence, or HOLD must remain asserted for at least four clocks after HLDA is asserted.

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	23. *Sticky Inexact Bit of FSR May Be Incorrectly Set* *Problem:* Scalar multiply operations may incorrectly set the SI bit of the FSR when the multiply unit has a data-dependent rounding freeze. If the SI bit is set, then the processor may or may not have encountered an inexact result. *Workaround:* Do not use the SI bit.
X	X	F	—	24. *Floating Point Underflow Trap May Occur When FZ Set* *Problem:* In correct operation, when the FZ bit in the FSR is set, no floating point traps are reported on underflow results, although the MU (or AU) bit is set. However, if during floating point pipeline resumption MU (or AU) is restored to the multiplier (or adder) pipeline by setting the U (Update) bit of the FSR, then a false underflow trap may occur even though FZ is set. This false underflow trap may occur upon execution of the first floating point trap-reporting instruction after the MU (or AU) bit has propagated to the third stage of the pipeline. *Workaround:* When restoring a floating point pipeline by setting the U bit in the FSR, the trap handler must clear the MU (or AU) bit whenever the FZ bit is set.
X	X	X	F	25. *AA Bit Not Set Correctly* *Problem:* The **(p) fix** and **(p) ftrunc** instructions may not set the AA bit of the FSR correctly. Specifically, the AA bit may report that the conversion of an exact non-positive number to an integer value involved an upward rounding, when in fact it did not. Note that the AA bit is not IEEE defined. *Workaround:* Disregard the AA bit when using **(p) fix** or **(p) ftrunc**. The conversion result will still be correct.
X	X	F	—	26. ***famov** and **pfamov** Erroneously Normalize Negative Denormals* *Problem:* **famov** and **pfamov** (which is used in the trap handler to restore the adder pipeline) erroneously normalize negative denormals. *Workaround:* If *fsrc1* of a **(p) famov** is a negative denormal as defined in PRM Table 2-2, then the **(p) famov** must be replaced by a **(p) fadd** whose *fsrc2* is negative zero or by a **(p) fsub** whose *fsrc2* is positive zero.
x	X	X	F	27. *Pipeline Precision Transition May Cause False Result Exception* *Problem:* If both of the following conditions are met, a false result exception can be reported. However, the result exception bits in the FSR due to this result exception will be cleared by the time the processor branches to the trap handler. 1. When the multiplier pipeline transitions from single- to double-precision pipelined mode, the second-to-last single precision pipelined instruction causes a result exception trap and a multiplier rounding freeze. 2. A floating point trap-reporting instruction is decoded during the multiplier freeze. *Workaround:* Either one of the following two workarounds will suffice: 1. Flush the multiplier pipeline when switching source precision. (This workaround is preferred.) 2. If the trap handler finds FT set in the PSR but no result-status bits set in the FSR, then it should just return back to user code.

Stepping				Description
B2	B3	C1	D0	
X	X	X	F	28. *Flush Instruction on 16-Byte Boundary Corrupts Data* *Problem:* If the address referenced by a **flush** instruction is aligned on a 16-byte boundary rather than a 32-byte boundary, data corruption may result. The data in the upper half of the cache line will be written back both to its corresponding address in memory and also to the memory address corresponding to the lower 16 bytes of that cache line. *Workaround:* Either one of the following two workarounds will suffice: 1. Align all addresses referenced by the flush instruction on 32-byte boundaries. Flushing on 32-byte boundaries causes both cache line halves to be written back to the correct memory location as needed, and clears the modified bit of the lower half of the cache line. 2. If flushing on a 16-byte boundary is necessary (for example, to clear the modified bit of the upper half of the cache line), a **flush** to the 32-byte boundary immediately below that 16-byte boundary must immediately precede the **flush** to the upper half of the cache line. This first **flush** changes the tag address for that line to indicate the reserved flush area; the subsequent flush on the 16-byte boundary will then cause an erroneous writeback to the flush area. However, the erroneous writeback will not corrupt system data because it does not affect the valid copy of the cache line which has already been written back to the correct memory address for that data.
X	X	F	—	29. *External Interrupts Ignored While Data = Lock Opcode During Read* *Problem:* During a read cycle, external interrupts are ignored while the opcode for the **lock** instruction appears on the data bus before READY# is asserted. External interrupt acknowledgment resumes normally when this data pattern is changed. This situation is only a problem for sites using multiple processors in lock-step; even then, there is no functional difference between the two processors as far as code execution is concerned. *Workaround:* When running multiple processors in lock-step, ensure that both processors always see the same data patterns.
X	X	F	—	30. ***flush** with HOLD May Corrupt Data or Hang Processor* *Problem:* In systems using HOLD/HLDA with **flush**, the bus cycle for a writeback caused by a **flush** instruction can be "lost", causing the writeback data to be stuck in the internal write buffers. This condition will cause data corruption and may cause the processor to hang. The problem occurs only when both of the following conditions are met: 1. The data cache has at least two cache line entries which have been half modified (EITHER the upper or lower half is modified, but not both). 2. HOLD is asserted during the **flush** routine. *Workaround:* For systems using HOLD/HLDA with **flush**, the following procedures are recommended for flushing the data cache and should be used in place of the one specified in the PRM. Upon RESET the data cache must be initialized using the ***flush*** instruction. To flush the data cache subsequently, the ***fld*** instruction must be used in the flush loop. The data cache initialization and flush routines require an 8 KB reserved *cacheable* flush memory area. The reserved area must be hardware (KEN#) and page table (CD/WT) cacheable. **NOTE:** If this workaround is used, no workarounds for Errata #10 and #23 are necessary, even if **flush** with paging is used.

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	30. (Continued) **Data cache initialization at RESET** The following procedure should be used to initialize the data cache at RESET:

```
   .data
flush_area::
   .byte [8192] 0   // Using method of your choice, reserve
                    // 8 KB of writable, cacheable memory.
   .text

reset initialization::
   ld.c   dirbase, Rv             // Save dirbase
   adds   -1, r0, Rx              // Loop decrement
   or     127, r0, Ry             // Loop counter
   or     l%flush_area-32, r0, Rt // Beginning virtual addr
   orh    h%flush_area-32, Rt, Rt // minus 32B (to allow for
                                  // autoincrement)

   // Initialize the first half of the cache
   andnot 0xF00,   Rv, Rw   // Clear RC, RB; put result in Rw
   or     x0800,   Rw, Rw   // Set MSB of RC
   bla    Rx, Ry,  init1    // One time to initialize LCC
    st.c   Rw, dirbase      // Store dirbase; RC = 2, RB = 0

init1:
   bla    Rx, Ry,  init1    // Loop for 128 iterations to
    flush  32(Rt)++         // initialize 1st 4 KB block

   // Now initialize the other half by changing RB
   or     0x900, Rw, Rw     // Set RC = 2 and RB = 1
   or     127, r0, Ry       // Reset loop counter
   bla    Rx, Ry, init2     // One time to initialize LCC
    st.c  Rw, dirbase       // RC = 2, RB = 1

init2:
   bla    Rx, Ry, init2     // Loop for 128 iterations to
    flush  32(Rt)++         // initialize 2nd 4 KB block
   bri    r1                // Return to calling procedure
    st.c   Rv, dirbase      // Restore original dirbase
```

Stepping				Description
B2	B3	C1	D0	

X	X	F	—	30. (Continued)
				Data cache flush other than at RESET
				The following procedure should be used to flush the D-cache at any time other than at RESET:
				```
				flush::
				   ld.c    dirbase, Rv              // Save dirbase
				   adds    -1, r0, Rx               // Loop decrement
				   or      127, r0, Ry              // Loop counter
				   or      l%flush_area-32, r0, Rt  // Beginning virtual addr
				   orh     h%flush_area-32, Rt, Rt  // minus 32B (to allow for
				                                    // autoincrement)

				   // Flush 1st half of cache (with writeback if modified)
				   andnot 0xF00, Rv, Rw     // Clear RC, RB; put result in Rw
				   or     0x800, Rw, Rw     // Set MSB of RC
				   bla    Rx, Ry, flush1    // One time to initialize LCC
				    st.c   Rw, dirbase      // Store dirbase; RC = 2, RB = 0

				flush1:
				   bla    Rx, Ry, flush1    // Loop for 128 iterations to
				    fld.d  32(Rt)++, f0     // load from each addr, causing
				                            // writebacks from modified lines
				   // Now flush second half of the cache
				   or     0x900, Rw, Rw     // Set RC = 2 and RB = 1
				   or     127, r0, Ry       // Reset loop counter
				   bla    Rx, Ry, flush2    // One time to initialize LCC
				    st.c   Rw, dirbase      // RC = 2, RB = 1

				flush2:
				   bla    Rx, Ry, flush2    // Loop for 128 iterations to
				    fld.d  32(Rt)++, f0     // load from each addr, causing
				                            // writebacks from modified lines

				   bri    r1                // Return to calling procedure
				    st.c   Rv, dirbase      // Restore original dirbase
				```

X	X	X	F	31. ***pfmul3.dd Can Modify CC Bit*** *Problem:* The **pfmul3.dd** instruction, which is intended primarily for use in the trap handler, can erroneously modify the CC bit in the PSR. This modification may result in code execution errors if an instruction which follows the **pfmul3.dd** tests the CC bit. *Workaround:* If an instruction following a **pfmul3.dd** tests the CC bit, save the value of the CC bit before the **pfmul3.dd** and restore it after that instruction.

Stepping				Description
B2	B3	C1	D0	
X	X	X	F	32. *IL Bit Is Not Set on DAT after UNLOCK* *Problem:* When a data access trap occurs on the first load or store after an **unlock** instruction, the IL bit of the EPSR is not set. Thus, the trap handler returns to that load or store instruction instead of returning to the **lock** instruction as it should. This problem also occurs if the address of the first load or store after the **unlock** instruction matches the data breakpoint (DB) register. *Workaround:* Both of the following workarounds must be implemented: 1. Move the load or store that follows the **unlock** instruction prior to the **unlock** instruction, and replicate that load or store as a dummy load or store to the same address after the **unlock** instruction. This guarantees that the unlocking dummy load or store will not trap. 2. Do not permit a data breakpoint register trap to occur on the unlocking load or store.
X	X	X	F	33. ***pfld** after Multicycle Write Cycle with HOLD May Corrupt Data* *Problem:* If all the following conditions are met, **pfld** data may be corrupted. 1. A **ld**- or **fld**-induced cache line writeback or a **fst.q** generates data in the write-back buffers to be written out to memory in a multicycle write cycle. A multicycle write cycle is any write cycle which requires more than one address issued to complete the bus cycle. (The **flush** instruction may also cause the problem, but only if it is used as a general purpose instruction, rather than in the flush loop. As stated in the Programmer's Reference Manual, such a usage has undefined results.) 2. The multicycle write cycle is interrupted by HOLD and HLDA, allowing another bus master on the bus between the writes of the multicycle write cycle. 3. When HOLD is asserted, there are no internal requests for bus cycles pending (other than for the cycles of the multicycle write described in section 1 above). 4. A **pfld** hits the data cache before all cycles of the multicycle write cycle have completed. The data at the address of this **pfld** may be corrupted. This data will be written to the *fdest* of the third **pfld** later. *Workaround:* Any one of the following three workarounds will suffice: 1. Only use **pfld** on data which cannot reside in the data cache. This workaround may be implemented by making pages containing **pfld** data noncacheable. 2. Do not use **pfld** with systems that use HOLD. 3. For systems that use HOLD and allow **pfld** to access cacheable data, insert a dummy store that is guaranteed to be a data cache miss between each **ld** or **fld** and a following **pfld**.
X	X	X	F	34. *IL Bit Mistakenly Set on Trap Before Lock* *Problem:* If the instruction executed immediately prior to a **lock** instruction traps on a DAT, IT, or FT, the trap handler will find the IL (interlock) bit of the EPSR set, mistakenly indicating that a locked sequence is in progress. Note that this problem can occur either in sequential execution or when the trapping instruction is in the delay slot of a branch or call whose target is a **lock** instruction. *Workaround:* Either one of the following two workarounds will suffice: 1. Put a **nop** before every **lock** instruction. In addition, if the **lock** is a branch target, change the branch target address to the **nop** before the **lock**. 2. Ensure that the instruction preceding a **lock** instruction can never trap.

Stepping				Description
B2	B3	C1	D0	
X	X	X	F	35. *Incorrect Address Translation on* ***st.c*** *Setting ITI after D-cache Miss/TLB Hit* *Problem:* If a data access instruction **[(f) ld, (f) st, *pst*,** or **pfld]**which is a data cache miss/TLB hit is followed by a **st.c** which sets the ITI bit of **dirbase**, then an incorrect or extra address translation may occur. The **st.c** does *not* need to immediately follow the data access instruction in order to cause the problem; any number of instructions may intervene. *Workaround:* Execute an **ixfr r0, f0** immediately before a **st.c** which sets the ITI bit, and follow the **st.c** immediately with six **nops**, as specified in the Programmer's Reference Manual. Floating point traps must be disabled during execution of the **ixfr**.
X	X	X	F	36. DAT in DIM with Paging May Cause FP Pipeline Corruption *Problem:* If all of the following conditions are met, a floating point pipeline may be corrupted as described. IF: 1. Paging is enabled. 2. Instruction caching is enabled both in hardware and in software. 3. A **ld** instruction that does not reside in the delay slot of a branch or call causes a data access trap (DAT). 4. The DAT-causing **ld**, which may occur either in single or dual instruction mode (DIM), is followed by a DIM pair. 5. The DIM pair which follows the **ld** instruction has: a) a scalar floating point half (i.e. not **fnop**, **fxfr**, or a pipelined instruction). b) an integer half which causes a freeze because one of its source operands overlaps with the **ld**'s destination. THEN: After trapping on the DAT, the first pipelined FP operation in the trap handler may corrupt its corresponding pipeline. For example, if the first pipelined FP operation in the trap handler is a **pfmul**, the multiplier pipeline may be corrupted. *Workaround:* Either one of the following two workarounds will suffice: 1. In DIM or a DIM transition, if the destination register of a **ld** instruction is referenced as a source in the integer half of the next dual pair, then the floating point instruction of that pair must be non-scalar (i.e. **fnop**, **fxfr**, or a pipelined instruction). 2. On any trap, the trap handler should first save the register state (but not the pipeline state) of the CPU and then determine whether a **ld** has caused a DAT while DIM and/or DS is set. If not, the trap should be handled normally, but if so, it should check whether the **ld** is in the delay slot of a branch or call. If it is, the trap should be handled normally, but if not, the trap handler should examine the next instruction or instruction pair in user code after the trapping instruction. If the **ld** is followed by a DIM pair in which the FP half is a scalar adder, multiplier, or graphics operation, then the first FP operation performed in the trap handler after saving register state should be a pipelined adder, multiplier, or graphics unit instruction respectively. Otherwise, the trap should be handled normally.
X	X	X	F	37. *BL Bit Not Set Immediately After* ***lock*** *Instruction* *Problem:* A **ld.c dirbase, rx** immediately after a **lock** instruction may show the BL bit not set, although it should be. However, if any other instruction intervenes between the **lock** and the **ld.c dirbase, rx**, the BL bit will appear correctly set. *Workaround:* Ensure that there is at least one instruction between a **lock** and a following **ld.c dirbase, rx**.

<table>
<tr><th colspan="4">Stepping</th><th rowspan="2">Description</th></tr>
<tr><th>B2</th><th>B3</th><th>C1</th><th>D0</th></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>38. <i>First 32-byte Block of Trap Handler Code May Execute Incorrectly</i>
<i>Problem:</i> Under certain rare conditions, requiring paging, caching and a trap, the CPU may skip or execute twice any of the instructions on the first 32-byte block of the trap handler code.
<i>Workaround:</i> For systems using paging and caching, place eight <b>nops</b> as the first eight instructions of the trap handler code (at address 0xFFFFFF00).</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>39. <i><b>INTOVR</b> May Trap Incorrectly</i>
<i>Problem:</i> The <b>OF</b> bit is unreliable in the trap handler during a <b>DAT</b>, <b>SE</b> or <b>RE</b> that is followed by an <b>adds/addu/subs/subu</b> instruction. In dual instruction mode, if the <b>intovr</b> instruction is followed by an <b>adds/addu/subs/subu</b> instruction, and the floating point instruction paired with the <b>intovr</b> traps, <b>intovr</b> may trap incorrectly upon returning from the trap handler. The <b>adds/addu/subs/subu</b> instruction can set the <b>OF</b> bit by the time the floating point trap is taken. When the trap handler returns to the <b>intovr</b> pair, the <b>OF</b> bit is set in the EPSR, and the <b>intovr</b> instruction causes an instruction trap (<b>IT</b>). Consider the following example:
<pre>
                        // OF=0
A: d.fop   // FP operation reports an SE or RE
          intovr   // Should not report an overflow trap
B: d.fop   // Another FP operation
          addu   // Sets OF
</pre>The <b>intovr</b> instruction should not trap since <b>OF</b> is clear. However, the <b>OF</b> bit is set by the <b>addu</b> instruction before an <b>SE</b> or <b>RE</b> is taken. When the trap handler returns to pair A, the <b>intovr</b> instruction now traps because <b>OF</b> is set, even though it should not trap.
<i>Workaround:</i> In single instruction mode, the trap handler should ignore the <b>OF</b> bit.
In dual instruction mode, if a floating point trap occurs, and the instruction at FIR + 4 (core half of the pair) is the <b>intovr</b> instruction, the trap handler must set the <b>OF</b> bit (EPSR) to the value of the <b>IT</b> bit (PSR). This will guarantee that the <b>OF</b> bit is set correctly upon return to the dual pair. If the <b>intovr</b> instruction trapped (<b>IT</b> = 1 and <b>FIR</b> contains <b>intovr</b>), then <b>OF</b> should be set to 1. If the <b>intovr</b> instruction didn't trap (<b>IT</b> = 0 and <b>FIR</b> contains <b>intovr</b>), then <b>OF</b> should be set to 0.
<b>NOTE:</b>
This workaround must be placed in the section before the normal <b>intovr</b> handling in the trap handler.</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>40. <i><b>ld.c fsr</b> in DIM May Return the Wrong Value</i>
<i>Problem:</i> In dual instruction mode, if <b>ld.c fsr</b> is followed by a pipelined floating point instruction, then this pipelined floating point instruction may update the <b>fsr</b> early. The <b>ld.c</b> may return the updated fsr value from the execution of the pipelined floating point instruction.
<b>NOTE:</b>
This problem does not occur in single instruction mode.
<i>Workaround:</i> In dual instruction mode, do not follow the <b>ld.c fsr</b> instruction with a pipelined floating point instruction.</td></tr>
</table>

Stepping				Description
B2	B3	C1	D0	
X	X	X	X	41. *PFLD Pipeline May Return Corrupted Data*

Problem: Under the following conditions the PFLD pipeline can lose synchronization, resulting in corrupted data. Once the pipeline is out of synchronization, it will remain out of synchronization until the chip is reset. All of the conditions listed below must be present in order for the error to occur.

1. A **pfld** instruction is near the end of an instruction cache line.
   and;
2. The next instruction to be fetched is both an instruction cache miss and a TLB miss.
   and;
3. Data for the **pfld** resides on the same page as the instruction cache miss.

Workaround: Ensure that **pfld** data resides on non-instruction pages.

NOTE:

The following procedure may be used during a context switch to test for pfld pipeline corruption. This allows the problem to be detected and localized to a single user. If the test fails then the pipeline is corrupted and the 80860XR must be RESET in order to use **pfld** instructions again.

```
  // pfld PIPELINE TEST ROUTINE.
  // Mem_addr is the address of three consecutive double word
  // memory locations that have been initialized with some
  // pfld load test data.
  // Save_Stage1, Save_Stage2 and Save_Stage3 are memory
  // locations used as temporary storage for pipeline data.
  // Rx is an integer register and Fx, Fy, Fz are floating
  // point registers used in the test.
  // The test data is '1', '2', and '3' in this example.

test_pipes:
  or    l%Mem_addr, r0, Rx    // Set Rx pointer to memory
  orh   h%Mem_addr, Rx, Rx    // addr containing test data.

  pfld.d 0(Rx), Fx            // Load values from memory into
  pfld.d 4(Rx), Fy            // pipeline and save current
  pfld.d 8(Rx), Fz            // pipeline data to Fx, Fy, Fz.

  fst.d Fx, Save_Stage1(r0)   // Save old Stage1 data
  fst.d Fy, Save_Stage2(r0)   // Save old Stage2 data
  fst.d Fz, Save_Stage3(r0)   // Save old Stage3 data

  pfld.d  Save_Stage1(r0), Fx// Fx <- '1' and restore Stage1
  pfld.d  Save_Stage2(r0), Fy// Fy <- '2' and restore Stage2
  pfld.d  Save_Stage3(r0), Fz// Fz <- '3' and restore Stage3

  fst.d  Fx, Save_Stage1(r0) // Memory (Save_Stage1) <- Fx
  fst.d  Fy, Save_Stage2(r0) // Memory (Save_Stage2) <- Fy
  fst.d  Fz, Save_Stage3(r0) // Memory (Save_Stage3) <- Fz

  ld.l  Save_Stage1(r0), Ry   // Ry <- Memory(Save_Stage1)
  btne  1, Ry, test_fail      // Test for (Ry = '1')
  ld.l  Save_Stage2(r0), Ry   // Ry <- Memory(Save_Stage2)
  btne  2, Ry, test fail      // Test for (Ry = '2')
  ld.l  Save_Stage3(r0), Ry   // Ry <- Memory(Save_Stage3)
  bte   3, Ry, test_pass      // Test for (Ry = '3')
test_fail:
  // Insert a branch to HALT the system or RESET the 80860XR.
test_pass:
  // pfld pipeline is not corrupted - Continue
```

Stepping				Description
B2	B3	C1	D0	
X	X	F	—	42. *Multiple Sequential Transfers between the Integer and Floating Point Units can Result in Corrupted Data* *Problem:* When executing a very tight loop resulting in large numbers of transfers between the integer and floating point units, data can become corrupted. All of the following conditions listed below must be present in order for the error to occur. 1. High Temperature ($>70°C\ T_{case}$) and; 2. Low Voltage (<50V) and; 3. A very long ($>10^9$), tight loop involving transfers between the integer and floating point units. The loop length required to cause failure differs with temperature and voltage. *Workaround:* No feasible workaround in B2/B3 steppings.
X	X	X	X	43. ***LOCK** Protocol Failure at Page Boundary* *Problem:* An Instruction Access Trap (IAT) may be incorrectly reported, and the access (A) bit of a Page Table or Page Directory Entry may be updated without assertion of the LOCK# pin, when all of the following conditions occur. 1. A **LOCK** sequence finishes near the end of a page (the **st** instruction following the **unlock** instruction is one of the last four instructions on the page) and; 2. Paging is enabled and; 3. The access (A) bit is not set in the following page. *Workaround:* Any one of the following four workarounds will suffice: 1. Disable Paging 2. Set the A bit in all Page Table (and Page Directory) Entries to 1. 3. Finish **lock** sequences before the last four instructions of the current page. 4. Ignore IATs signaled by the processor when none of the following valid IAT conditions is present: a) Present bit not set in PTE or PDE (PTE.P = 0) b) Supervisor Page Protection Violation (PSR.PU = 1 AND PTE.U = 0) c) Access Bit not set in PTE during a **lock** sequence (PTE.A = 0 AND PSR.IL = 1) **NOTE:** If workaround #4 is used, the A bit of the following page may be set by the processor without the LOCK# pin asserted.

DATA SHEET REVISION REVIEW

The following list represents the key differences between version 002 and version 001 of the i860 XR Microprocessor Data Sheet.

1. Big-endian description in section 2.3 has been expanded.
2. Bit 17 of the Extended Processor Status Register (EPSR) is the INT bit which reflects the value on the interrupt pin (INT), as described in section 2.2.4 entitled "EXTENDED PROCESSOR STATUS REGISTER". This is a documentation update only.
3. The cacheability of a page is controlled by NOR'ing the value of the CD, WT bits and the KEN# input pin, as described in section 2.5 entitled "Caching and Cache Flushing" and section 3.1.14 entitled "Cache Enable (KEN#)". This is a documentation update only.
4. The NOTE section in section 2.5 entitled "Caching and Cache Flushing" has been updated to clarify the paging requirement on changing the DTB field in the **dirbase** register.
5. Information on register encoding is added in section 8.2 entitled "Instruction Format and Encoding". This is a documentation update only.

The following list represents the key differences between version 003 and version 002 of the i860 XR Microprocessor Data Sheet.

Specification Changes:

1. Specification changes for improved AC performance are in section 7.3.
2. HOLD is acknowledged during locked bus cycles. See section 3.1.8.
3. Additional paths have been added to the bus state diagram to allow direct transitions from states T12 and T11 to state TH. See Figures 4.1 and 4.10.
4. Two new instructions, **(p)famov.r,** have been added. These replace **(p)fadd.ds** and **(p)fadd.sd** in the assembler pseudo-ops **(p)fmov.r**. These changes are in section 8.1 and tables 2.7, 8.7, and 8.9.

Documentation Changes:

1. Big and little endian description has been expanded in sections 2.2.2, 2.3, and Figure 2.8.
2. The actions and explanations of the **lock, unlock,** and **st.c dirbase** changing the BL bit have been updated in sections 2.2.4, 3.1.5, 3.1.8, 4.3.4, 4.3.5, and 8.1.
3. The explanation of the AA and MA bits of the **fpsr** have been expanded in section 2.2.8.
4. The explanation of the WT bit of the Page Table Entries has been expanded in sections 2.4.4.4 and 2.5.
5. A change concerning the locking of the bus during address translation is explained in sections 2.4.5 and 2.8.5.
6. A further explanation on when to flush the data cache is given in section 2.5.
7. The explanation of the floating point multiplier pipeline has been expanded in section 2.6.1.
8. The explanation of BREQ has been expanded in section 3.1.4 and Figure 4.1.
9. The explanation of result exceptions has been expanded in sections 2.8 and 3.2.
10. Instruction fetch identification has been clarified in section 3.1.6 and table 3.2.
11. Bus cycle diagrams in Figures 4.7, 4.8, and 4.10 have been clarified/corrected.
12. Precision specification **.r** has been added to section 8.0 and table 8.1.
13. In section 8.4, performance note 9 has been added, programming restriction d has been changed, and programming restriction f has been added. Table 8.9 has been updated to reflect these changes.
14. The description of testability has changed in sections 3.3. and 3.3.2. RESET and HOLD must be asserted by the tester to force the chip outputs to float (tri-state).

The following list represents the major differences between version 004 and version 003 of the i860 XR Microprocessor Data Sheet:

Section 2.2.4	The explanation of the WP bit of the **espr** has been expanded.
Section 2.8.2	More information on the instruction trap has been added.
Section 2.8.4	The instruction access trap has been clarified.
Section 2.8.7	The values of registers after a reset trap have been specified.
Section 3.1.4	BREQ timing has been clarified.
Section 3.1.5	The calculation of interrupt latency has bee corrected.
Section 3.1.6	The description of the byte-enable signals has been expanded.
Section 3.1.8	The relation between the **lock** instruction and the LOCK# signal has been clarified. The BL bit should no longer be changed by writing to the **dirbase** register.
Section 6.0	The thermal specifications have been updated.

Section 7.3	The A.C. Characteristics for CLK have changed.
Section 7.3	Advance timing information for the 50 MHz clock rate has been added. These timings are subject to change without notice.
Section 8.0	The operand naming conventions have improved.
Section 8.2.1	The encoding of the **flush** instruction has been corrected.
Section 8.3	The data-dependent multiplier freeze has been eliminated. Other freeze conditions have been corrected or clarified.

The following list represents the major differences between version 005 and version 004 of the i860 XR Microprocessor Data Sheet.

Section 2.2.4	OF bit is writable only in supervisor mode using ST.C.
Section 3.1.1	CLK rate has been updated.
Section 5.0	Figure 5.3 has been corrected.
Section 6.0	More information on measuring case temperature has been added.
Section 6.0	Figure 6.1 has been updated to include 25 MHz.
Section 6.0	Table 6.1 has been corrected.
Section 6.0	Table 6.2. has been updated to include 25 MHz.
Section 7.2	The D.C. Characteristics have been updated to include 25 MHz power supply current.
Section 7.3	The A.C. Characteristics for CLK have been changed.
Section 7.3	50 MHz clock rate has been deleted.
Section 7.3	25 MHz A.C. Specifications have been added.
Section 7.3	Figure 7.1 has been corrected.
Section 8.3	The data-dependent multiplier rounding freeze has been eliminated.
Section 8.4	Programming restrictions for dual-instruction mode are added.

The following list represents the differences between version 005 and version 006 of the 80860XR Microprocessor data sheet.

Section 9.0 Functional Characteristics section added.