intel

i860™ Microprocessor Family Programmer's Reference Manual

LITERATURE

To order Intel Literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your ***local*** sales office or distributor.

INTEL LITERATURE SALES
P.O. BOX 7641
Mt. Prospect, IL 60056-7641

In the U.S. and Canada
call toll free
(800) 548-4725

CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information. All handbooks can be ordered individually, and most are available in a pre-packaged set in the U.S. and Canada.

TITLE	INTEL ORDER NUMBER	ISBN
SET OF THIRTEEN HANDBOOKS (Available in U.S. and Canada)	**231003**	**N/A**
CONTENTS LISTED BELOW FOR INDIVIDUAL ORDERING:		
COMPONENTS QUALITY/RELIABILITY	210997	1-55512-132-2
EMBEDDED APPLICATIONS	270648	1-55512-123-3
8-BIT EMBEDDED CONTROLLERS	270645	1-55512-121-7
16-BIT EMBEDDED CONTROLLERS	270646	1-55512-120-9
16/32-BIT EMBEDDED PROCESSORS	270647	1-55512-122-5
MEMORY PRODUCTS	210830	1-55512-117-9
MICROCOMMUNICATIONS	231658	1-55512-119-5
MICROCOMPUTER PRODUCTS	280407	1-55512-118-7
MICROPROCESSORS	230843	1-55512-115-2
PACKAGING	240800	1-55512-128-4
PERIPHERAL COMPONENTS	296467	1-55512-127-6
PRODUCT GUIDE (Overview of Intel's complete product lines)	210846	1-55512-116-0
PROGRAMMABLE LOGIC	296083	1-55512-124-1
ADDITIONAL LITERATURE: (Not included in handbook set)		
AUTOMOTIVE HANDBOOK	231792	1-55512-125-x
INTERNATIONAL LITERATURE GUIDE (Available in Europe only)	E00029	N/A
CUSTOMER LITERATURE GUIDE	210620	N/A
MILITARY HANDBOOK (2 volume set)	210461	1-55512-126-8
SYSTEMS QUALITY/RELIABILITY	231762	1-55512-046-6

intel®

U.S. and CANADA LITERATURE ORDER FORM

NAME: ____________________

COMPANY: ____________________

ADDRESS: ____________________

CITY: ____________________ STATE: __________ ZIP: __________

COUNTRY: ____________________

PHONE NO.: (   ) ____________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal __________

Must Add Your Local Sales Tax __________

Include postage:
Must add 15% of Subtotal to cover U.S. and Canada postage. (20% all other.)

→ Postage __________

Total __________

Pay by check, money order, or include company purchase order with this form ($100 minimum). We also accept VISA, MasterCard or American Express. Make payment to Intel Literature Sales. Allow 2-4 weeks for delivery.

☐ VISA ☐ MasterCard ☐ American Express Expiration Date __________

Account No. ____________________

Signature ____________________

Mail To: Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

International Customers outside the U.S. and Canada should use the International order form on the next page or contact their local Sales Office or Distributor.

For phone orders in the U.S. and Canada
Call Toll Free: (800) 548-4725

Prices good until 12/31/91.
Source HB

intel®

INTERNATIONAL LITERATURE ORDER FORM

NAME: _____

COMPANY: _____

ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

COUNTRY: _____

PHONE NO.: (____) _____

ORDER NO.	TITLE	QTY.		PRICE	TOTAL
			×		=
			×		=
			×		=
			×		=
			×		=
			×		=
			×		=
			×		=
			×		=
			×		=

Subtotal _____

Must Add Your Local Sales Tax _____

Total _____

PAYMENT

Cheques should be made payable to your ***local*** Intel Sales Office (see inside back cover).

Other forms of payment may be available in your country. Please contact the Literature Coordinator at your ***local*** Intel Sales Office for details.

The completed form should be marked to the attention of the LITERATURE COORDINATOR and returned to your ***local*** Intel Sales Office.

intel®

i860™ MICROPROCESSOR FAMILY PROGRAMMER'S REFERENCE MANUAL

1991

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

376, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, ActionMedia, BITBUS, Code Builder, COMMputer, CREDIT, Data Pipeline, DeskWare, DVI, ETOX, FaxBACK, Genius, i, i287, i386, i387, i486, i750, i860, i960, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, I^2ICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, Intel287, Intel386, Intel387, Intel486, int$_e$lBOS, Intel Certified, Intelevision, int$_e$ligent Identifier, int$_e$ligent Programming, Intellec, Intellink, iOSP, iPAT, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, iWARP, Library Manager, MAPNET, Matched, Media Mail, MCS, Megachassis, MICROMAINFRAME, MULTI CHANNEL, MULTIMODULE, MultiSERVER, NetPort, ONCE, OpenNET, OTP, PRO750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, READY-LAN, RMX/80, RUPI, SatisFAXtion, Seamless, SLD, SnapIn 386, SugarCube, SUPERCHARGER, The Computer Inside, ToolTalk, UNIPATH, UPI, VAPI, Visual Edge, VLSiCEL, WYPIWYF, and ZapCode.

MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

OS/2 is a trademark of IBM Corp.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

CUSTOMER SUPPORT

INTEL'S COMPLETE SUPPORT SOLUTION WORLDWIDE

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, consulting services and network management services. For detailed information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is extensive. It can start with assistance during your development effort to network management. 100 Intel sales and service offices are located worldwide—in the U.S., Canada, Europe and the Far East. So wherever you're using Intel technology, our professional staff is within close reach.

HARDWARE SUPPORT SERVICES

Intel's hardware maintenance service, starting with complete on-site installation will boost your productivity from the start and keep you running at maximum efficiency. Support for system or board level products can be tailored to match your needs, from complete on-site repair and maintenance support to economical carry-in or mail-in factory service.

Intel can provide support service for not only Intel systems and emulators, but also support for equipment in your development lab or provide service on your product to your end-user/customer.

SOFTWARE SUPPORT SERVICES

Software products are supported by our Technical Information Service (TIPS) that has a special toll free number to provide you with direct, ready information on known, documented problems and deficiencies, as well as work-arounds, patches and other solutions.

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and *;COMMENTS Magazine*). Basic support consists of updates and the subscription service. Contracts are sold in environments which represent product groupings (e.g., iRMX® environment).

NETWORK SERVICE AND SUPPORT

Today's broad spectrum of powerful networking capabilities are only as good as the customer support provided by the vendor. Intel offers network services and support structured to meet a wide variety of end-user computing needs. From a ground up design of your network's physical and logical design to implementation, installation and network wide maintenance. From software products to turn-key system solutions; Intel offers the customer a complete networked solution. With over 10 years of network experience in both the commercial and Government arena; network products, services and support from Intel provide you the most optimized network offering in the industry.

CONSULTING SERVICES

Intel provides field system engineering consulting services for any phase of your development or application effort. You can use our system engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training and customizing an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, BITBUS™ and LAN applications.

PREFACE

The Intel i860™ microprocessor family delivers supercomputer performance in a single VLSI component. The 64-bit i860 architecture balances integer, floating point, and graphics performance for applications such as engineering workstations, scientific computing, 3-D graphics, and multiuser systems. The architecture achieves high throughput with RISC design techniques, parallel and pipelined processing units, wide data paths, large on-chip caches, and fast, submicron CHMOS silicon technology. The i860 microprocessor family includes:

- i860 XR Microprocessor (part number 80860XR)
- i860 XP Microprocessor (part number 80860XP)

This book is the basic source of the detailed information that enables software designers and programmers to use i860 microprocessors. This book explains all programmer-visible features of the architecture.

Even though the principal users of this Programmer's Reference Manual will be programmers, it contains information that is of value to systems designers and administrators of software projects, as well. Readers of these latter categories may choose to read only the higher-level sections of the manual, skipping over much of the programmer-oriented detail.

HOW TO USE THIS MANUAL

- Chapter 1, "Architectural Overview," describes the i860 microprocessors "in a nutshell" and presents for the first time the terms that will be used throughout the book.
- Chapter 2, "Data Types," defines the basic units operated on by the instructions of the i860 microprocessor.
- Chapter 3, "Registers," presents the processor's database. A detailed knowledge of the registers is important to programmers, but this chapter may be skimmed by administrators.
- Chapter 4, "Addressing," presents the details of operand alignment, virtual memory, and on-chip caches. Systems designers and administrators may choose to read the introductory sections of each topic.
- Chapter 5, "On-Chip Caches," explains cache operation in detail sufficient for applications programmers to optimize for the caches and for systems programmers to manage the caches correctly.
- Chapter 6, "Concurrency Control," shows how the detached CCU of the i860 XP microprocessor supports programs designed for concurrent operations, even in a uniprocessor system.
- Chapter 7, "Core Instructions," presents detailed information about those instructions that deal with memory addressing, integer arithmetic, and control flow.

- Chapter 8, "Floating-Point Instructions," presents detailed information about those instructions that deal with floating-point arithmetic, long-integer arithmetic, and 3-D graphics support. It explains how extremely high performance can be achieved by utilizing the parallelism and pipelining of the i860 architecture.
- Chapters 9 and 10, "Traps and Interrupts," deal with both systems- and applications-oriented exceptions, external interrupts, writing exception handlers, saving the state of the processor (information that is also useful for task switching), and initialization.
- Chapter 11, "Programming Model," defines standards for the use of many features of the i860 architecture. Software administrators should be aware of the need for standards and should ensure that they are implemented. Following the standards presented here guarantees that compilers, applications programs, and operating systems written by different people and organizations will all work together.
- Chapter 12, "Programming Examples," illustrates the use of the i860 architecture by presenting short code sequences in assembly language.
- The appendices present instruction formats and encodings, timing information, and summaries of instruction characteristics. These appendices are of most interest to assembly-language programmers and to writers of assemblers, compilers, and debuggers.

RELATED DOCUMENTATION

The following books contain additional material concerning the i860 microprocessor:

- *i860™ 64-Bit Microprocessor* (Data Sheet), order number 240296
- *i860™ XP Microprocessor* (Data Sheet), order number 240874
- *i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual*, order number 240436
- *i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual*, order number 240437
- *i860™ Microprocessor Math Library Reference Manual*, order number 464411

NOTATION AND CONVENTIONS

This manual uses special notation for symbolic representation of instructions and for hexadecimal numbers. A review of this notation makes the manual easier to read.

Instruction Descriptions

The instruction chapters contain an algorithmic description of each instruction that uses a notation similar to that of the Algol or Pascal languages. The metalanguage uses the following special symbols:

- **A ← B** indicates that the value of B is assigned to A.
- Compound statements are enclosed between the keywords of the "if" statement (IF ... , THEN ... , ELSE ... , FI) or of the "do" statement (DO ... , OD).

- The operator + + indicates autoincrement addressing.
- Register names and instruction mnemonics are printed in a contrasting typestyle to make them stand out from the text; for example, **dirbase**. Individual programming languages may require the use of lowercase letters.

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, fsr, bear, ccr, p0, p1, p2,** or **p3**
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 microprocessor sign-extends to 32 bits when computing the effective address.
const32	A 32-bit constant. Only 16 bits of the constant can be used at one time in any i860 microprocessor instruction. The operators *l%* and *h%* select the low-order and high-order half, respectively.

lbroff — A signed, 26-bit, immediate, relative branch offset. The offset has a resolution of four bytes; it does not address individual bytes.

sbroff — A signed, 16-bit, immediate, relative branch offset. The offset has a resolution of four bytes; it does not address individual bytes.

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, as shown in Table 0-1. In instruction descriptions, the following codes represent precision specifications:

.p — Precision specification **.ss**, **.sd**, or **.dd** (**.ds** not permitted). Refer to Table 0-1.

.r — Precision specification **.ss**, **.sd**, **.ds**, or **.dd**. Refer to Table 0-1.

.v — **.sd** or **.dd**. Refer to Table 0-1.

.w — **.ss** or **.dd**. Refer to Table 0-1.

Other abbreviations include:

.x — **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)

.y — **.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)

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.

Table 0-1. Precision Specification

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

Hexadecimal Numbers

Hexadecimal constants are written, according to the C language convention, with the prefix **0x**. For example, 0x0F is a hexadecimal number that is equivalent to decimal 15.

RESERVED BITS AND SOFTWARE COMPATIBILITY

In many register and memory layout descriptions, certain bits are marked as *reserved* or *undefined*. When bits are thus marked, it is essential for compatibility with future processors that software not utilize these bits. Software should follow these guidelines in dealing with reserved or undefined bits:

- Do not depend on the states of any reserved or undefined bits when testing the values of registers that contain such bits. Mask out the reserved and undefined bits before testing.
- Do not depend on the states of any reserved or undefined bits when storing them in memory or in another register.
- Do not depend on the ability to retain information written into any reserved or undefined bits.
- When updating a control register, always set the reserved and undefined bits to values previously retrieved from the same register.
- When initializing memory layouts, set reserved bits to zero.

NOTE

Depending upon the values of reserved or undefined bits makes software dependent upon the unspecified manner in which the i860 microprocessor handles these bits. Depending upon values of reserved or undefined bits risks making software incompatible with future processors that define usages for these bits. **AVOID ANY SOFTWARE DEPENDENCE UPON THE STATE OF RESERVED OR UNDEFINED BITS.**

TABLE OF CONTENTS

Figures

Figures

Tables

Tables

Examples

Examples

Architectural Overview 1

CHAPTER 1
ARCHITECTURAL OVERVIEW

The Intel i860 Microprocessor architecture balances integer, floating-point, and graphics performance. Target applications include engineering workstations, scientific computing, 3-D graphics workstations, numerics accelerators, multiuser and multiprocessor systems. The architecture achieves high throughput with RISC design techniques, pipelined and parallel processing units, wide data paths, and large on-chip caches.

The i860 architecture was implemented first in the i860 XR microprocessor. The second generation, the i860 XP microprocessor, is upward compatible for applications programs and enhances the i860 microprocessor family with higher clock speeds, greater bus bandwidth, multiprocessor capabilities, larger on-chip caches, four Mbyte pages, and second-level cache support.

1.1 OVERVIEW

The i860 microprocessor architecture supports more than just integer operations. The architecture includes on a single chip:

- Integer operations
- Floating-point operations
- Graphics operations
- Memory management
- Data and instruction caches

Having a data cache as an integral part of the architecture provides support for vector operations. The data cache supports applications programs in the conventional manner, without explicit programming. For vector operations, however, programmers can explicitly use the data cache as if it were a large block of vector registers.

To sustain high performance, i860 microprocessors incorporate wide information paths that include:

- 64-bit external data bus
- 128-bit on-chip data bus
- 64-bit on-chip instruction bus

Floating-point and graphics programs can simultaneously use all three buses.

The i860 microprocessors include a RISC integer core processing unit with one-clock instruction execution. The core unit processes integer instructions and provides complete support for operating systems, such as UNIX and OS/2. The core unit also drives the graphics and floating-point hardware.

The i860 microprocessors support vector floating-point operations without special vector instructions or vector registers. They accomplish this by using the on-chip data cache and a variety of parallel techniques that include:

- Pipelined instruction execution with delayed branch instructions to avoid breaks in the pipeline.
- Instructions that automatically increment index registers so as to reduce the number of instructions needed for vector processing.
- Simultaneous integer and floating-point processing.
- Parallel multiplier and adder units within the floating-point unit.
- Pipelined floating-point hardware, with both scalar (nonpipelined) and vector (pipelined) variants of floating-point instructions. Software can switch between scalar and pipelined modes.
- Large register set:
  - 32 general-purpose integer registers, each 32 bits wide.
  - 32 floating-point registers, each 32 bits wide, which can also be configured as 64- and 128-bit registers. The floating-point registers also serve as the staging area for data going into and out of the floating-point and graphics pipelines.

Figures 1-1 and 1-2 illustrate the registers and data paths of the i860 XR microprocessor and i860 XP microprocessor respectively.

1.2 INSTRUCTIONS

There are two classes of instruction:

- Core instructions (executed by the integer core unit).
- Floating-point and graphics instructions (executed by the floating-point unit and graphics unit).

The processors have a dual-instruction mode that can simultaneously execute one instruction from each class (core and floating-point). Software can switch between dual- and single-instruction modes without overhead. Within the floating-point unit, dual-operation instructions (add-and-multiply, subtract-and-multiply) use the adder and multiplier units in parallel. Using both dual-instruction mode and dual operation instructions, i860 microprocessors can execute three operations simultaneously.

The integer core unit manages data flow and loop control for the floating-point units. Together, they efficiently execute such common tasks as evaluating systems of linear equations and performing Fast Fourier Transforms (FFT) and graphics transformations.

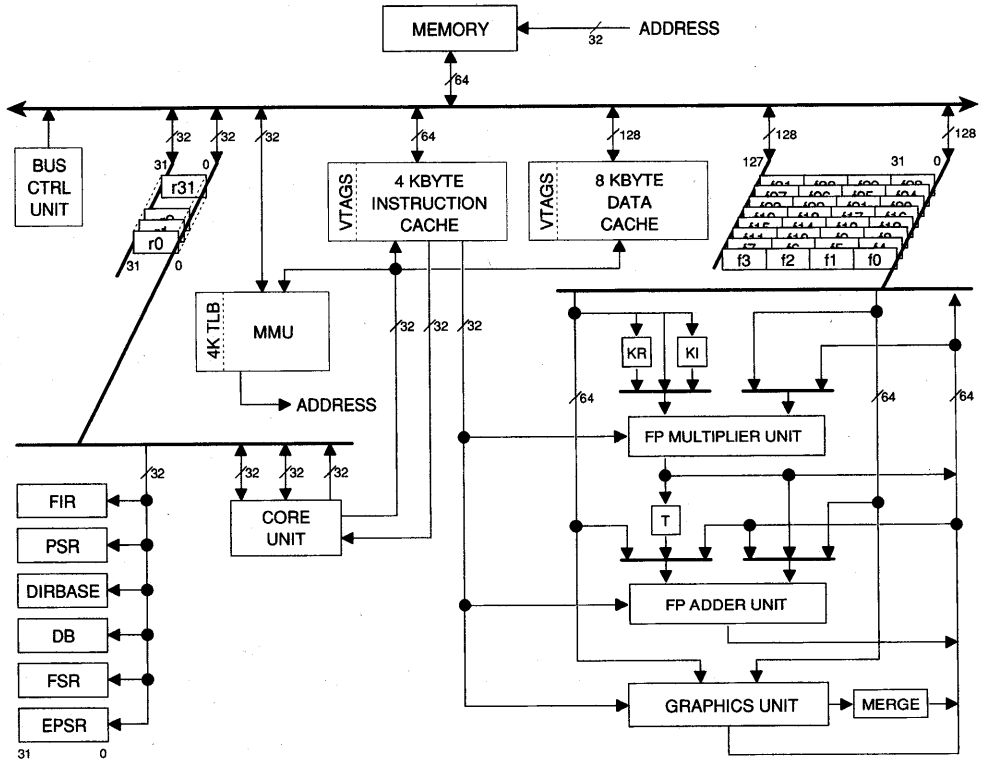


Figure 1-1. i860™ XR CPU Registers and Data Paths

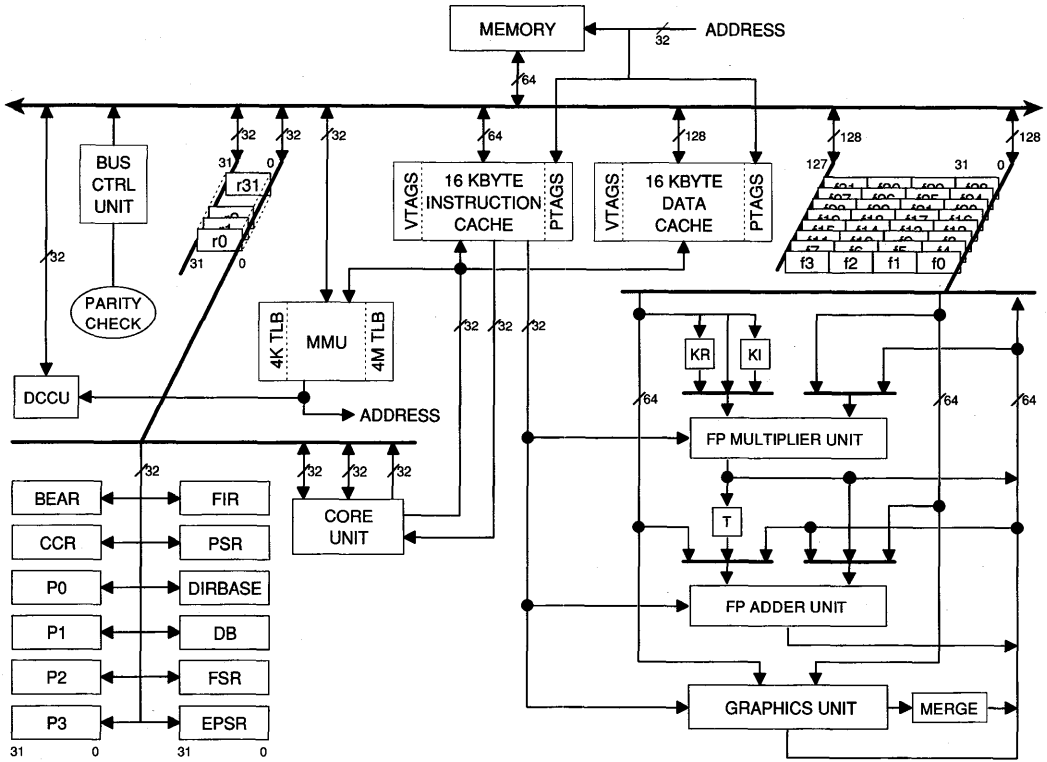


Figure 1-2. i860™ XP CPU Registers and Data Paths

1.3 INTEGER CORE UNIT

The core unit is the administrative center of the processor. The core unit fetches both integer and floating-point instructions. It contains the integer register file, and executes load, store, integer, bit, and control-transfer operations. Its pipelined organization with extensive bypassing and scoreboarding maximizes performance. Its instructions include:

- Loads and stores between memory and the integer and floating-point registers. Floating-point loads can be pipelined in three levels to tolerate external memory latency. A pixel store instruction contributes to efficient hidden-surface elimination.
- Transfers between the integer registers and the floating-point registers.
- Integer arithmetic for 32-bit signed and unsigned numbers. The 32-bit operations can also perform arithmetic on smaller (8- or 16-bit) integers. (The graphics unit provides arithmetic for 64-bit integers.)
- Shifts of the integer registers.
- Logical operations on the integer registers.
- Control transfers. The instruction set includes both direct and indirect branches and call instructions as well as a branch for highly efficient loops. Many of these are delayed transfers that avoid breaks in the instruction pipeline. One instruction provides efficient loop control by combining the testing and updating of the loop index with a delayed control transfer.
- System control functions, such as control register manipulation and cache configuration.
- I/O and interrupt acknowledgment.

1.4 FLOATING-POINT UNIT

The floating-point unit contains the floating-point register file. This file can be accessed as 8 × 128-bit registers, 16 × 64-bit registers, or 32 × 32-bit registers. Three additional registers (KR, KI, and T) hold intermediate floating-point results.

The floating-point unit contains both the floating-point adder and the floating-point multiplier. The adder performs floating-point addition, subtraction, comparison, and conversions. The multiplier performs floating-point and integer multiply as well as floating-point reciprocal operations. Both units support 64- and 32-bit floating-point values in IEEE Standard 754 format. Each of these units uses pipelining to deliver up to one result per clock. The adder and multiplier can operate in parallel, producing up to two results per clock. Furthermore, the floating-point unit can operate in parallel with the core unit, sustaining a rate of two floating-point results per clock rate by overlapping administrative functions with floating-point operations.

The RISC design philosophy minimizes circuit delays and enables using all the available chip area to achieve the greatest performance for floating-point operations. The RISC design philosophy, the use of pipelining and parallelism in the floating-point unit, and the wide on-chip caches – all these factors contribute to extremely high levels of floating-point performance.

Because i860 microprocessors employ RISC design principles, they do not have high-level math instructions. High-level math (and other) functions are implemented in software macros and libraries. For example, there is no **sin** instruction. The **sin** function is implemented in software on i860 microprocessors. The **sin** routine for an i860 microprocessor, however, is still fast due to the high speed of the basic floating-point operations. Commonly used math operations, such as the **sin** function, are offered by Intel as part of a software library.

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) with both single- and double-precision floating-point data types. Not all functions defined by the standard are implemented directly by the hardware. The i860 architecture supplies the underlying data types, instructions, exception checking, and traps to make it possible for software to implement the remaining functions of the standard efficiently. Intel offers a software library that provides full IEEE-compatible arithmetic.

1.5 GRAPHICS UNIT

The graphics unit has 64-bit integer logic that supports 3-D graphics drawing algorithms. This unit can operate in parallel with the core unit. It contains the special-purpose MERGE register, and performs additions on integers stored in the floating-point register file.

These special graphics features focus on applications that involve three-dimensional graphics with Gouraud or Phong color intensity shading and hidden surface elimination via the Z-buffer algorithm. The graphics features of the i860 architecture assume that:

- The surface of a solid object is drawn with polygon patches which, like the pieces of a puzzle, collectively approximate the shape of 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 microprocessor directly aid such interpolation. Furthermore, the i860 microprocessor 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 data bus. An eight-byte MERGE register assists in parallelizing graphics algorithms.

The graphics unit also provides addition and subtraction operations for 32- and 64-bit integers, which are especially useful for high-resolution distance interpolation.

In addition to the support provided by the graphics unit, many 3-D graphics applications directly benefit from the parallelism of the core and floating-point units. For example, the 3-D rotation represented in homogeneous vector notation by ...

$$[X\ Y\ Z\ W] = [x\ y\ z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos t & \sin t & 0 \\ 0 & -\sin t & \cos t & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

... is just one example of the kind of vector-oriented calculation that can be converted to a program that takes full advantage of the pipelining, dual-instruction mode, dual operations, and memory hierarchy of the i860 architecture.

1.6 MEMORY MANAGEMENT UNIT

The on-chip MMU of i860 microprocessors performs the translation of addresses from the linear logical address space to the linear physical address for both data and instruction access. Address translation is optional; when enabled, address translation uses page tables. Information from these tables is cached on-chip. The i860 architecture provides the basic features (bits and traps) to implement paged virtual memory and to implement user/supervisor protection at the page level—all compatible with the paged memory management of the Intel386™ and Intel486™ microprocessors.

The i860 XR microprocessor uses four-Kbyte pages with a two-level structure of page directories and page tables of 1K entries each. The TLB (translation look-aside buffer) is a 64-entry, four-way set-associative memory, which caches translation information for quick access.

The i860 XP microprocessor supports both four-Kbyte pages and four-Mbyte pages. The four-Kbyte pages are compatible with those of the i860 XR microprocessor. There is a two-level structure of page directories and page tables of 1K entries each. The TLB is a 64-entry, four-way set-associative memory. The four-Mbyte pages do not use second-level page tables. They are supported by a second TLB, which is a 16-entry, four-way set-associative memory. The four-Kbyte and four-Mbyte pages can be used together in any combination.

1.7 CACHES

In addition to the page translation caches (TLBs) mentioned previously, the i860 microprocessor contains separate on-chip caches for data and instructions. Caching is transparent, except to systems programmers who must maintain cache consistency when switching tasks, modifying instructions, or changing system memory parameters. The on-chip cache controller also provides the interface to the external bus with a pipelined structure that allows up to three outstanding bus cycles.

On the i860 XR microprocessor, the instruction cache is a two-way, set-associative memory of four Kbytes, with 32-byte blocks. The data cache is a write-back cache, composed of a two-way, set-associative memory of eight Kbytes, with 32-byte blocks.

On the i860 XP microprocessor, the instruction cache is a four-way, set-associative memory of 16 Kbytes, with 32-byte lines. The data cache is a write-back cache, composed of a four-way, set-associative memory of 16 Kbytes, with 32-byte lines. A MESI cache protocol, combined with support for inquiry cycles, ensures that cache consistency is maintained in multiprocessor and multimaster systems.

1.8 PARALLEL ARCHITECTURE

The i860 architecture offers a high level of parallelism in a form that is flexible enough to be applied to a wide variety of processing styles:

- Conventional programs and conventional compilers can use i860 microprocessors as scalar machines and still benefit from their high performance. Even when used as scalar machines, i860 microprocessors implement concurrency between integer and floating-point operations, as long as there are no conflicts for internal resources. An integer instruction that follows a floating-point instruction begins immediately, overlapping the floating-point instruction. A floating-point instruction that follows an integer instruction also begins immediately.
- Compilers designed for the vector model can treat i860 microprocessors as vector machines.
- Advanced instruction-scheduling technology for compilers can compare the processing requirements and data dependencies of programs with the available resources of the i860 microprocessor, and can take maximum advantage of its dual-instruction mode, pipelining, and caching.

An established compiler technology for the vector model of computation already exists. This technology can be applied directly to the i860 architecture. The key to treating the i860 microprocessors as vector machines is choosing the appropriate vector primitives that the compiler assumes are available on the target machine. (Intel has defined a standard library of vector primitives.) The vector primitives are implemented as hand-coded subroutines; the compiler generates calls to these subroutines. If a compiler depends on the traditional concept of vector registers, it can implement them by mapping these registers to specific memory addresses. By virtue of frequent access to these addresses, the simulated registers will reside permanently in the data cache.

Existing programs can be upgraded to take better advantage of the parallel i860 architecture using vector-oriented technology. Flow analysis or "vectorizing" tools can identify parallelism that is implicit in existing programs. When modified (either manually or automatically) and compiled by an appropriate compiler for the i860 architecture, these programs can achieve an even greater performance gain.

Designers of compilers will find that the i860 architecture offers more flexibility than traditional vector architectures. The instruction set of the i860 architecture separates addressing functions from arithmetic functions, which provides two benefits:

1. It is possible to address arbitrary data structures. Data structures are no longer limited to vectors, arrays, and matrices. Parallel algorithms can be applied to linked lists (for example) as easily as to matrices.

2. A richer set of operations is available at each node of a data structure. It becomes possible to perform different operations at each node, and there is no limit to the complexity of each operation. With the i860 architecture, it is no longer necessary to pass all elements of a vector several times to implement complex vector operations.

1.9 SOFTWARE DEVELOPMENT ENVIRONMENT

The software environment available from Intel for the i860 architecture includes:

- Assembler, linker, C and FORTRAN compilers, and FORTRAN vectorizer.
- Simulator and debugger.
- UNIX operating system.
- APX (Attached Processor Executive), an integrated operating environment for i860 microprocessors hosted on Intel386/Intel486 CPU platforms.
- Libraries of higher-level math functions and IEEE-standard exception support. Intel offers such libraries in a form that can be utilized by a variety of compilers.
- Libraries of vector arithmetic.
- Libraries of graphics functions.

Data Types 2

CHAPTER 2
DATA TYPES

i860 microprocessors provide 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 instructions operate on arrays of 8-, 16-, or 32-bit pixels.

Bits within data formats are numbered from zero starting with the least significant bit. Illustrations of data formats in this manual show the least significant bit (bit zero) at the right.

2.1 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 integers.

When an eight- 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 eight- or 16-bit item is stored from a register, the corresponding number of low-order bits of the register are used.

2.2 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.

2.3 SINGLE-PRECISION REAL

A single-precision real (also called "single real") data type is a 32-bit binary floating-point number. See Figure 2-1. 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 $\mathbf{e} = 0$ and $\mathbf{f} \neq 0$ or $\mathbf{e} = 255$ then generate a floating-point source-exception trap when encountered in a floating-point operation.

2. If $0 < \mathbf{e} < 255$, then the value is $-1^{s} \times 1.\mathbf{f} \times 2^{e-127}$. (The exponent adjustment 127 is called the *bias*.)

3. If $\mathbf{e} = 0$ and $\mathbf{f} = 0$, then the value is signed zero.

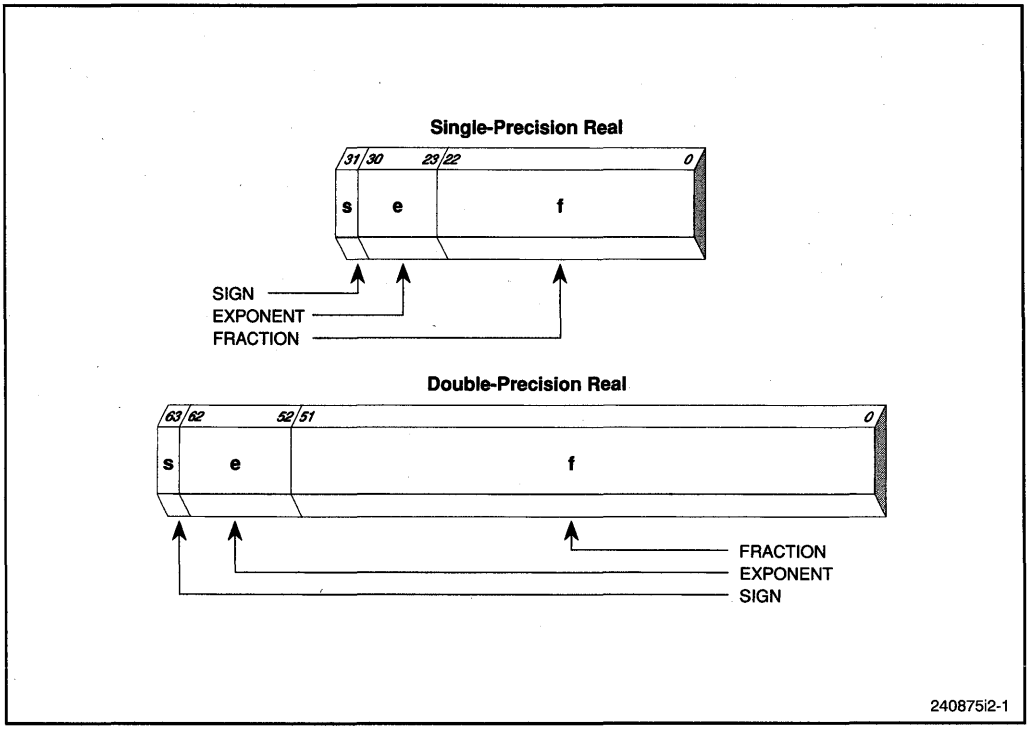


Figure 2-1. Real Number Formats

The special values infinity, NaN, indefinite, and denormal generate a trap when encountered. The trap handler implements IEEE-standard results. (Refer to Table 2-2 for encoding of these special values.)

2.4 DOUBLE-PRECISION REAL

A double-precision real (also called "double real") data type is a 64-bit binary floating-point number. See Figure 2-1. 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** ≠ 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 $-1^{s} \times 1.\mathbf{f} \times 2^{e-1023}$. (The exponent adjustment 1023 is called the *bias*.)

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

The special values infinity, NaN, indefinite, and denormal generate a trap when encountered. The trap handler implements IEEE-standard results. (Refer to Table 2-2 for encoding of these special values.)

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.

2.5 PIXEL

A pixel may be 8, 16, or 32 bits long, depending on color and intensity resolution requirements. Regardless of the pixel size, the processor always operates on 64 bits of pixel data 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 processor 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 processor defines only the field sizes, not the specific use of each field. Other ways of using the fields of pixels are possible.

2.6 REAL-NUMBER ENCODING

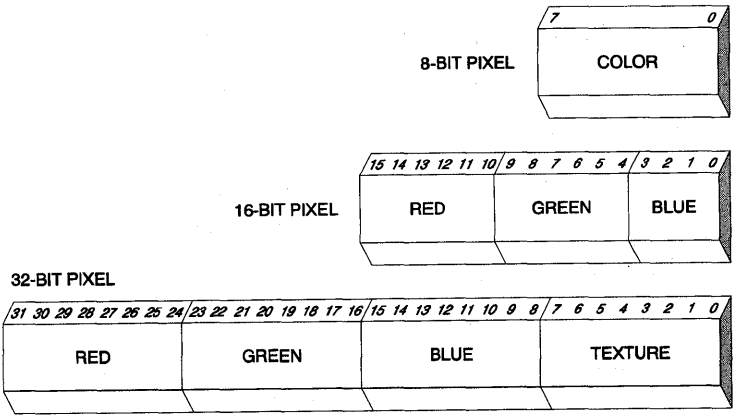
Table 2-2 presents the complete range of values that can be stored in the single and double real formats. Not all possible values are directly supported by the processor. The supported values are the normals and the zeros, both positive and negative. Other values are not generated by i860 microprocessors, and, if encountered as input to a floating-point instruction, they trigger the floating-point source exception. Exception-handling software can use the unsupported values to implement denormals, infinities, and NaNs.

Table 2-1. Pixel Formats

Pixel Size (in bits)	Bits of Color 1 Intensity[1]	Bits of Color 2 Intensity[1]	Bits of Color 3 Intensity[1]	Bits of Other Attribute (Texture, Color)
8	N (≤ 8) bits of intensity[2]			8 – N
16	6	6	4	0
32	8	8	8	8

NOTES: [1] The intensity attribute fields may be assigned to colors in any order convenient to the application.

[2] 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 or texture. Bits that require interpolation (shading), such as those for intensity, must be the low-order bits of the pixel.



NOTE:
THESE ASSIGNMENTS OF SPECIFIC MEANINGS TO THE FIELDS OF PIXELS ARE FOR ILLUSTRATION ONLY.
ONLY THE FIELD SIZES ARE DEFINED, NOT THE SPECIFIC USE OF EACH FIELD.

240875i2-2

Figure 2-2. Pixel Format Examples

Table 2-2. Single and Double Real Encodings

Class			Sign	Biased Exponent	Fraction ff..ff*
Positives	NaNs	Quiet	0 . . 0	11..11 . . 11..11	11..11 . . 10..00
		Signaling	0 . . 0	11..11 . . 11..11	01..11 . . 00..01
	Infinity		0	11..11	00..00
	Reals	Normals	0 . . 0	11..10 . . 00..01	11..11 . . 00..00
		Denormals	0 . . 0	00..00 . . 00..00	11..11 . . 00..01
		Zero	0	00..00	00..00
Negatives	Reals	Zero	1	00..00	00..00
		Denormals	1 . . 1	00..00 . . 00..00	00..01 . . 11..11
		Normals	1 . . 1	00..01 . . 11..10	00..00 . . 11..11
	Infinity		1	11..11	00..00
	NANs	Signaling	1 . .	11..11 . . 11..11	00..01 . . 01..11
		Quiet	1 . . 1	11..11 . . 11..11	10..00 . . 11..11
			Single: Double:	← 8 bits→ ←11 bits→	←23 bits→ ←52 bits→

NOTE: *Integer bit is implied and not stored.

Registers 3

CHAPTER 3
REGISTERS

As Figure 3-1 shows, the i860 microprocessor has the following registers:

- An integer register file
- A floating-point register file
- Control registers
- Special-purpose registers

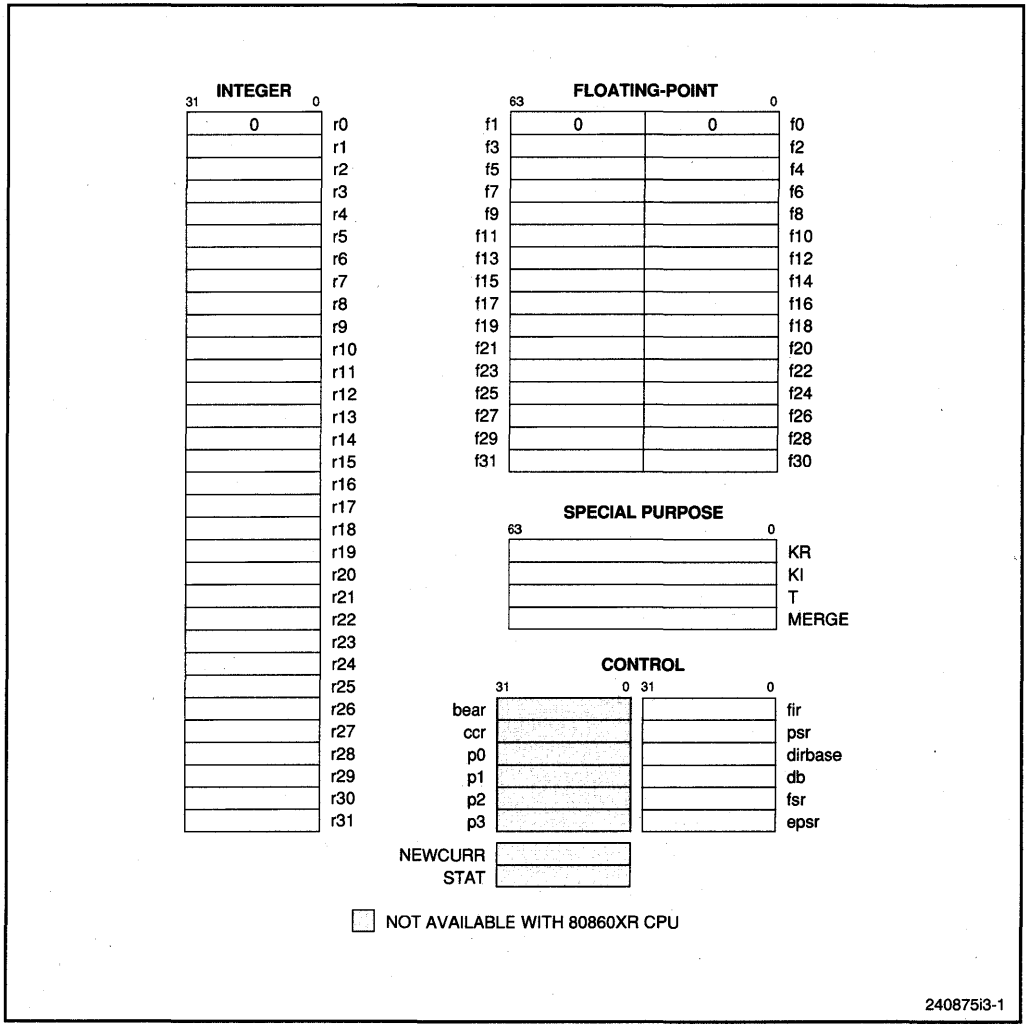


Figure 3-1. Register Set

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, and T are used by floating-point instructions; MERGE is used by graphics instructions. NEWCURR is a 32-bit counter used in the i860 XP microprocessor for concurrency control; it is accessed by memory load and store instructions. For information about initialization of registers, refer to the reset trap in Chapters 9 and 10. For information about protection as it applies to registers, refer to the **st.c** instruction in Chapter 7.

3.1 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. This special behavior of **r0** makes it useful for modifying the function of certain instructions. For example, specifying **r0** as the destination of a subtract (thereby effectively discarding the result) produces a compare instruction. Similarly, using **r0** as one source operand of an OR instruction produces a test-for-zero instruction.

3.2 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. The floating-point registers are also used by a set of integer operations, primarily for graphics computations.

The floating-point registers act as buffer registers in vector computations, while the data cache performs the role of the vector registers of a conventional vector processor.

When accessing 64-bit floating-point or integer values, the i860 microprocessor uses an even/odd pair of registers. When accessing 128-bit values, it uses an aligned set of four registers (**f0, f4, f8, f12, f16, f20, f24**, or **f28**). The instruction must designate the lowest register number of the set of registers containing 64- or 128-bit values. Misaligned register numbers produce undefined results. The register with the lowest number contains the least significant part of the value. For 128-bit values, the register pair with the lower number contains the value from the lower memory address; the register pair with the higher number contains the value from the higher address.

3.3 PROCESSOR STATUS REGISTER

The processor status register (**psr**) contains miscellaneous state information for the current process. Figure 3-2 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. (Refer to Section 3.5 for more about the **db** register.)

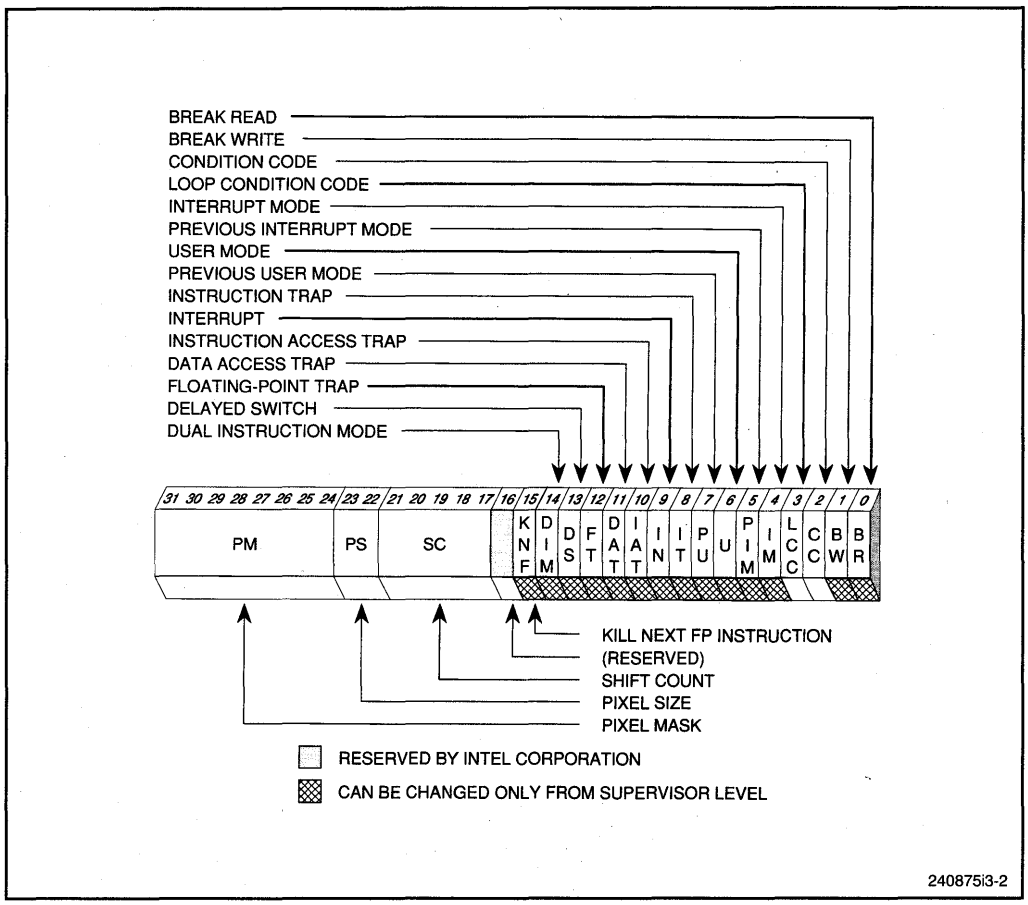


Figure 3-2. Processor Status Register

- Various instructions set CC (Condition Code) according to the value of the result, as explained in Chapter 7. The conditional branch instructions test CC. The **bla** instruction described in Chapter 7 sets and tests LCC (Loop Condition Code).
- IM (Interrupt Mode) enables external interrupts if set; disables interrupts if clear. IM does not affect parity error or bus error interrupts in the i860 XP microprocessor. (Chapters 9 and 10 cover interrupts.)
- U (User Mode) is set when the i860 microprocessor is executing in user mode; it is clear when the i860 microprocessor is executing in supervisor mode. In user mode, writes to some control registers are inhibited. This bit also controls the memory protection mechanism described in Chapter 4.
- 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**. (Chapters 9 and 10 provide the details about traps.)

- IT (Instruction Trap), IN (Interrupt), IAT (Instruction Access Trap), DAT (Data Access Trap), and FT (Floating-Point 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. Refer to Chapters 9 and 10 for a more detailed explanation.

- 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 microprocessor switches to dual-instruction mode one instruction after returning from the trap handler. If DS and DIM are both set, the i860 microprocessor switches to single-instruction mode one instruction after returning from the trap handler. Chapters 9 and 10 explain how trap handlers use these bits.

- When a trap occurs, the i860 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, the i860 microprocessor resumes execution in dual-instruction mode after returning from the trap handler.

- 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. KNF is especially useful for returning from a trap that occurred in dual-instruction mode, because it permits the core instruction to be executed while the floating-point instruction is suppressed. KNF is automatically reset by the i860 microprocessor when the instruction has been successfully bypassed. It is possible that the core instruction may cause a trap when the floating-point instruction is suppressed. In this case KNF remains set, permitting retry of the core instruction.

- 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, as described in Chapter 7.

- PS (Pixel Size) and PM (Pixel Mask) are used by the pixel-store instruction described in Chapter 7 and by the graphics instructions described in Chapter 8. The values of PS control pixel size as defined by Table 3-1. 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.

Table 3-1. Values of PS

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

3.4 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 3-3 shows the format of the **epsr**.

- The processor type is 1 for the i860 XR microprocessor; 2 for the i860 XP microprocessor.
- The stepping number has a unique value that distinguishes among different revisions of the processor.
- IL (Interlock) is set by the processor 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. The trap handler must clear IL.
- 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.

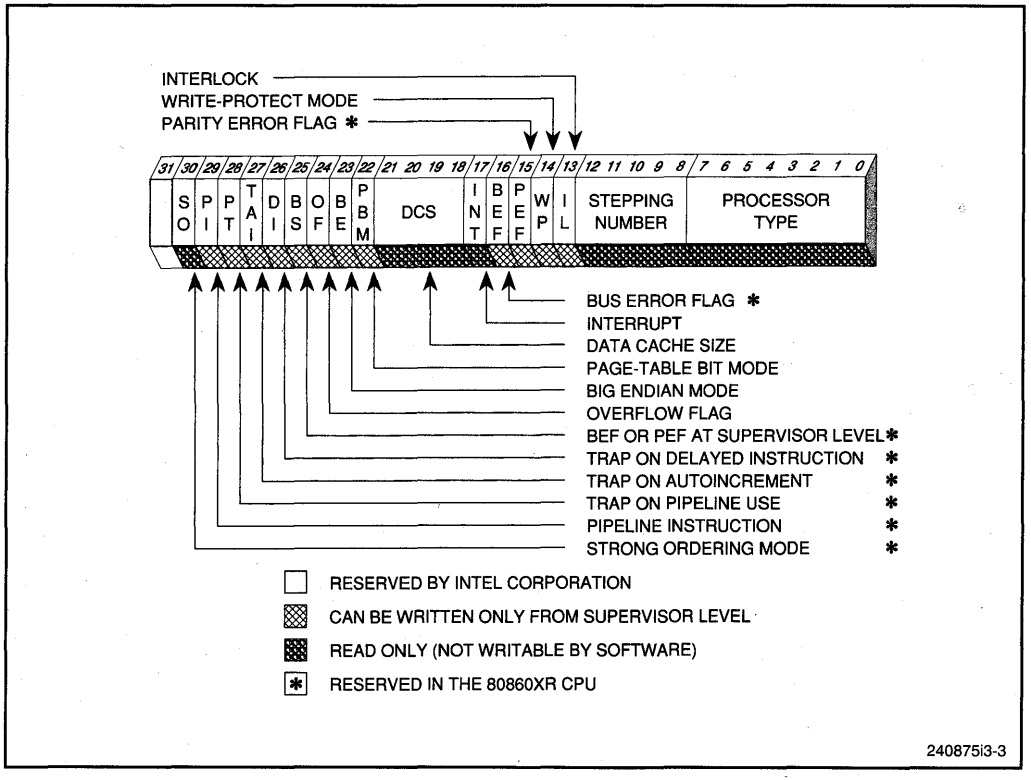


Figure 3-3. Extended Processor Status Register

- PEF (parity error flag) is set by the i860 XP microprocessor when a parity error trap occurs. As soon as PEF is set, further parity error and bus error traps are masked. Software must clear PEF to reenable such traps. PEF is set at RESET.
- BEF (bus error flag) is set by the i860 XP microprocessor when the BERR pin is asserted, indicating a bus error. As soon as BEF is set, further parity error and bus error traps are masked. Software must clear BEF to reenable such traps. BEF is set at RESET.
- INT (Interrupt) is the value of the INT input pin, except during a locked sequence when the INT flag is zero.
- 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. The value of DCS for the i860 XR microprocessor is one, which indicates eight Kbytes. The value of DCS for the i860 XP microprocessor is two, which indicates 16 Kbytes.
- PBM (Page-Table Bit Mode) determines which bit of page-table entries is output on the PTB pin of the i860 XR CPU. 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. PBM has no effect in the i860 XP microprocessor, it is used only by the i860 XR microprocessor.
- BE (Big Endian) controls the ordering of bytes within a data item in memory. Normally (i.e., when BE is clear) the i860 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. Big endian mode affects not only the memory load and store instructions but also the **ldio, stio, ldint,** and **scyc** instructions. Refer to Chapter 4 for more information on byte ordering.
- 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 can be changed by arithmetic instructions in either user or supervisor mode. It can be changed by the **st.c** instruction in supervisor mode only. OF controls the function of the **intovr** instruction (refer to Chapter 7).
- BS (bus or parity error trap in supervisor mode) is set by the i860 XP microprocessor when a bus or parity error occurs while the processor is in supervisor mode. The operating system can use this bit to decide, for example, whether to abort the currently running process (if BS = 0) or reboot the system (if BS = 1).
- DI (trap on delayed instruction) is set by the i860 XP microprocessor when a trap occurs on a delayed instruction (the instruction located after a delayed branch instruction). When DI is set, the trap handler must restart the interrupted procedure from the branch instruction rather than at the address in **fir**.
- TAI (trap on autoincrement instruction) is set by the i860 XP microprocessor when a trap occurs on an instruction with autoincrement (including the **bla** instruction). When TAI is set, the trap handler should undo the autoincrement (that is, restore *src2* to its original value).

- PT (trap on pipeline use) indicates to the i860 XP microprocessor that a trap should be generated and PI should be set when it executes an instruction that uses the floating-point or graphics unit. Such instructions include all the instructions of Chapter 8, plus the **pfld** instruction. PT is set and cleared only by software. It can be used by the trap handler to avoid unnecessary saving and restoring of the pipelines (refer to Chapters 9 and 10). When a trap due to PT occurs, the floating-point operation has not started, and the pipelines have not been advanced. Such a trap also sets the IT bit of **psr**.

- The behavior of PI (pipeline instruction) depends on the setting of PT. If PT = 0, the i860 XP microprocessor sets PI when any pipelined instruction or **pfld** is executed. If PT = 1, the processor sets PI when it decodes any instruction that uses the pipes, whether scalar or pipelined. Refer to Chapters 9 and 10.

- SO (strong ordering) indicates whether the processor is in strong ordering mode (SO=1) or weak ordering mode (SO=0). SO is set if the EWBE# pin is active (LOW) at RESET.

3.5 DATA BREAKPOINT REGISTER

The data breakpoint register (**db**) is used to generate a trap when the i860 microprocessor accesses an operand at the virtual address stored in this register. The trap is enabled by BR and BW in **psr**. When comparing, a number of low order bits of the address are ignored, depending on the size of the operand. For example, a 16-bit access ignores the low-order bit of the address when comparing to **db**; a 32-bit access ignores the low-order two bits. This ensures that any access that overlaps the address contained in the register will generate a trap. The trap occurs *before* the register or memory update by the load or store instruction.

3.6 DIRECTORY BASE REGISTER

The directory base register **dirbase** (shown in Figure 3-4) controls address translation, caching, and bus options.

- ATE (Address Translation Enable), when set, enables the virtual-address translation mechanism described in Chapter 4.

- 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 with static column or page-mode DRAMs when consecutive reads and writes access the same column or page. The comparison ignores the low-order 12 + DPS bits. A value of zero is appropriate for one bank of 256K×*n* RAMs, 1 for 1M×*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.

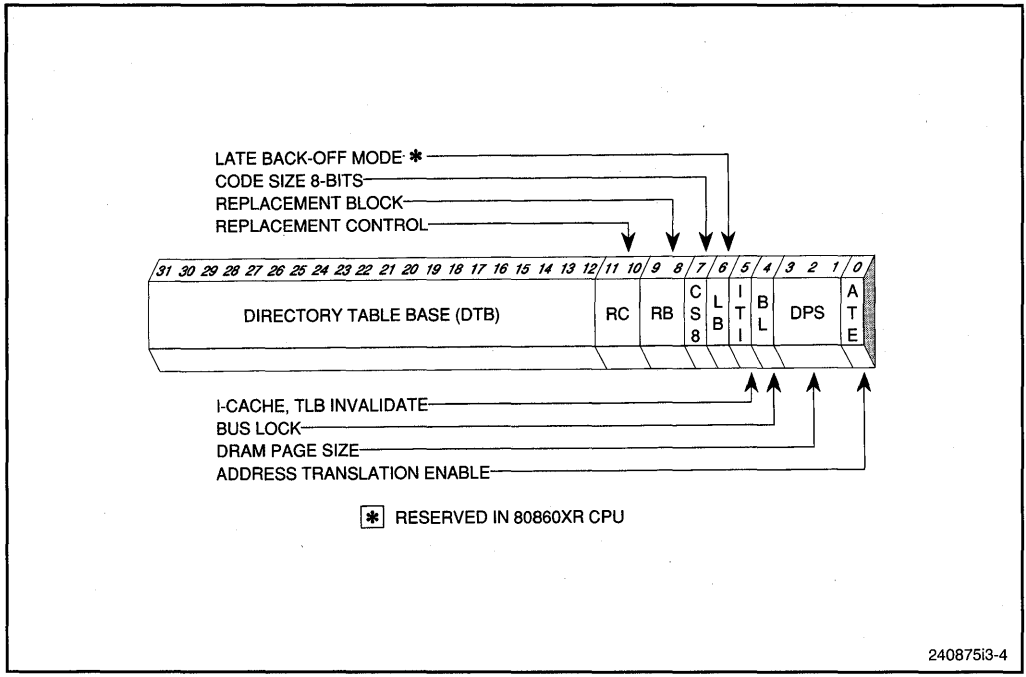


Figure 3-4. Directory Base Register

- When BL (Bus Lock) is set, external bus accesses are locked. The LOCK# signal is asserted with the next bus cycle (excluding instruction fetch and write-back cycles) 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. A trap that occurs during a locked sequence immediately clears BL and sets IL in **epsr**. In this case the trap handler should resume execution at the beginning of the locked sequence. The **lock** and **unlock** instructions control the BL bit (refer to Chapter 7). The result of modifying BL with the **st.c** instruction is not defined.

- ITI (Instruction-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 marked invalid. With the i860 XP microprocessor, ITI also invalidates all virtual tags in the data cache. The ITI bit does not remain set in **dirbase**. ITI always appears as zero when read from **dirbase**.

- When software sets the LB bit, the i860 XP microprocessor enters two-clock late back-off mode. This mode gives two additional clock periods of decision time to the external logic that may need to use the BOFF# signal to cancel a bus cycle or data transfer. If the processor enters one-clock late back-off mode during RESET via configuration pin strapping, the LB bit has no effect, and it is impossible to enter two-clock late back-off mode. Furthermore, software cannot exit two-clock late back-off mode once it is activated; the LB bit cannot be cleared except by resetting the processor.

- 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 cannot be set by software; hardware sets this bit at initialization time with the INT/CS8 pin. 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 PROM. A nondelayed branch to code in 64-bit memory should directly follow the **st.c** 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. Refer to the CS8 mode in the *i860™ 64-Bit Microprocessor Hardware Design Guide* for more information.
- RB (Replacement Block) identifies the cache block (line or way) to be replaced by cache replacement algorithms. RB conditions the cache flush instruction **flush**, which is discussed in Chapter 7. Table 3-2 explains the values of RB.
- RC (Replacement Control) controls cache replacement algorithms. Table 3-3 explains the significance of the values of RC. The use of the RC and RB to implement data cache flushing is described in Chapter 4.
- DTB (Directory Table Base) contains the high-order 20 bits of the physical addess of the page directory when address translation is enabled (i.e., ATE = 1). The low-order 12 bits of the address are zeros (therefore the directory must be located on a 4K boundary).

3.7 FAULT INSTRUCTION REGISTER

When a trap occurs, this register (the **fir**) contains the address of the instruction that caused the trap, as described in Chapters 9 and 10. The value of the **fir** can be accessed by an **ld.c** instruction. The trap address can be read from the **fir** only once; reading the **fir** anytime except the first time after a trap saves in *idest* one of the following values:

- In single-instruction mode, the address of the **ld.c** instruction
- In dual-instruction mode, the address of its floating-point companion of the **ld.c** instruction (address of the **ld.c** − 4).

The **fir** cannot be modified by the **st.c** instruction.

Table 3-2. Values of RB

Value	Replace TLB Way	Replace Instruction and Data Cache Way	
		i860™ XR CPU	i860™ XP CPU
0 0	0	0	0
0 1	1	1	1
1 0	2	0	2
1 1	3	1	3

Table 3-3. Values of RC

Value	Meaning
00	Selects the normal (random) 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. This mode is used for cache and TLB testing.
10	Data cache misses replace the block selected by RB. Instruction and TLB caches use random replacement. This mode is used when flushing the data cache with the **flush** instruction.
11	Disables data cache replacement. Instruction and TLB caches use random replacement.

3.8 FLOATING-POINT STATUS REGISTER

The floating-point status register (**fsr**) contains the floating-point trap and rounding-mode status for the current process. Figure 3-5 shows its format.

- If FZ (Flush Zero) is clear and underflow occurs, a result-exception trap is generated. When FZ is set and underflow occurs, the result is set to zero, and no trap due to underflow occurs.
- If TI (Trap Inexact) is clear, inexact results do not cause a trap. If TI is set, inexact results cause a trap. The sticky inexact flag (SI) is set whenever an inexact result is produced, regardless of the setting of TI.
- RM (Rounding Mode) specifies one of the four rounding modes defined by the IEEE standard. Given a true result b that cannot be represented by the target data type, the i860 microprocessor determines the two representable numbers a and c that most closely bracket b in value ($a < b < c$). The i860 microprocessor then rounds (changes) b to a or c according to the mode selected by RM as defined in Table 3-4. 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 a zero when read. A trap handler that has interrupted a pipelined operation sets the U-bit to enable restoration of the result-status bits in the pipeline. Refer to Chapters 9 and 10 for details.
- The FTE (Floating-Point Trap Enable) bit, if clear, disables all floating-point traps (invalid input operand, overflow, underflow, and inexact result). Trap handlers clear it while saving and restoring the floating-point pipeline state (refer to Chapters 9 and 10) and to produce NaN, infinite, or denormal results without generating traps.
- 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.

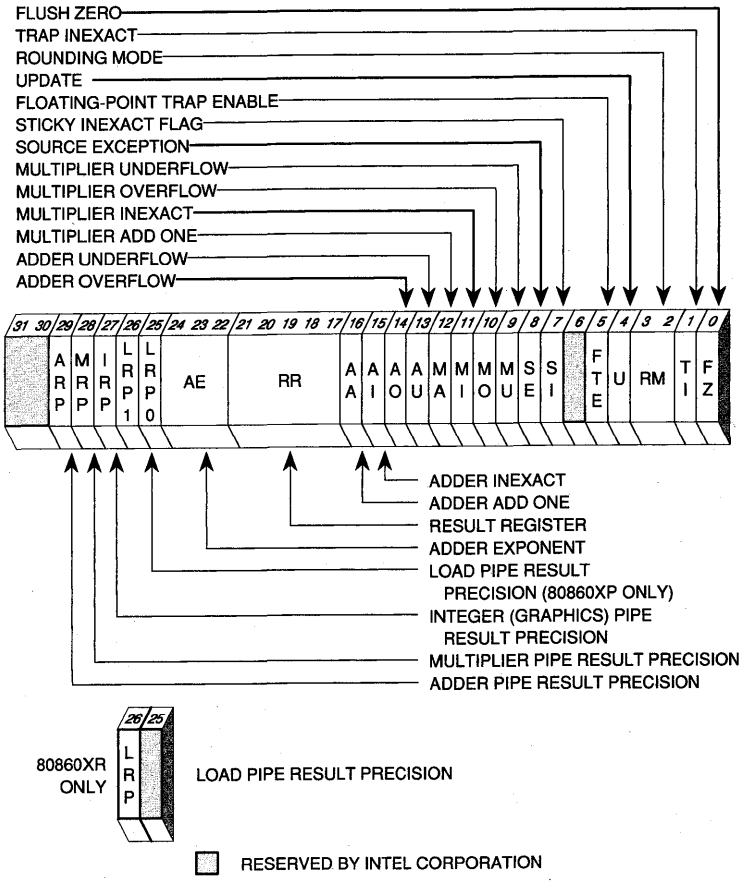


Figure 3-5. Floating-Point Status Register

Table 3-4. Values of RM

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

- 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). Trap handler software can implement IEEE-standard results for operations on these values.

- 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 trap handler needs the AE bits when overflow or underflow occurs with double-precision inputs and single-precision outputs.

  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**.

  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.

- AA (Adder Add One), when set, indicates that the absolute value of the fraction of the result of an adder operation was increased by one due to rounding. AA is not influenced by the sign of the result.

- MA (Multiplier Add One), when set, indicates that the absolute value of the fraction of the result of a multiplier operation was increased by one due to rounding. MA is not influenced by the sign of the result.

- 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.

- LRP applies only to the i860 XR microprocessor. The i860 XP microprocessor uses LRP1 and LRP0 (Load Pipe Result Precision), which together define the size of the last-stage result of the load pipeline. They are encoded as Table 3-5 shows.

Table 3-5. Values of LRP1 and LRP0

LRP1	LRP0	pfld Length
0	0	*(reserved)*
0	1	4 Bytes
1	0	8 Bytes
1	1	16 Bytes

3.9 KR, KI, T, AND MERGE REGISTERS

The KR and KI ("Konstant") registers and the T (Temporary) register are 64-bit, special-purpose registers used by the dual-operation floating-point instructions described in Chapter 8. The 64-bit MERGE register is used only by the graphics instructions also presented in Chapter 8. Refer to that chapter for details of their use.

3.10 BUS ERROR ADDRESS REGISTER

In i860 XP microprocessor systems, the **bear** helps the trap handler determine faulty memory locations. The i860 XP microprocessor loads a valid address into **bear** under these conditions:

- For bus errors, the **bear** receives the address of the cycle for which the BERR signal is asserted, if external hardware synchronizes assertion of BERR with BRDY# for that cycle.
- For parity errors on a read, the **bear** receives the address of the cycle during which the processor detects the error, if external hardware asserts PEN# with BRDY# for that cycle.

If external hardware does not meet these conditions, the contents of the **bear** are undefined.

A valid address in **bear** is accurate to 29 bits; that is, address signals A31–A3 are latched in the high-order 29 bits of **bear**. At RESET and after every parity and bus error trap, software must read the **bear** before further parity and bus error traps can occur. The **bear** is a read-only register.

3.11 PRIVILEGED REGISTERS (80860XP ONLY)

The registers **p0, p1 p2,** and **p3** are provided for the operating system to use. They do not affect processor operation. They can be accessed by the **ld.c** and **st.c** instructions, but they can be written only in supervisor mode. They may be used to store information such as the interrupt stack pointer, current user stack pointer at the beginning of the trap handler, register values during trap handling, processor ID in a multiprocessor system, or for any other purpose.

3.12 CONCURRENCY CONTROL REGISTER (80860XP ONLY)

The concurrency control register (**ccr**) controls the operation of the internal Concurrency Control Unit (CCU), which is described in Chapter 6. The **ccr** can be written in supervisor mode only, but can be read in user or supervisor mode. Figure 3-6 shows the format of the **ccr**.

The DO (Detached Only) bit and the CO (CCU On) bit together specify the CCU configuration. DO, when set, indicates that there is no external CCU. CO (CCU On) bit, when set, indicates that the Concurrency Control Architecture is enabled. Table 3-6 summarizes the modes defined by CO and DO bits. The reserved combinations should not be used by software.

If the DCCU is on (CO = DO = 1), the processor intercepts and interprets all memory loads and stores which are to the CCU address space, which is the two pages defined by CCUBASE. Loads and stores to that address range do not go to memory, but to the DCCU.

CCUBASE is the virtual address of the memory area into which the CCU registers are mapped. Software must set bit 12 to zero, because the CCUBASE must be aligned on a two-page (8-Kbyte) boundary. This is because an external CCU contains supervisor registers mapped to the second page.

3.13 NEWCURR REGISTER (80860XP ONLY)

The NEWCURR register is part of the detached CCU (concurrency control unit). It is a 32-bit counter that supplies an iteration count for loop execution. (Refer to Chapter 6.)

NEWCURR is architecturally a 64-bit register, but only the low-order 32 bits are provided in the i860 XP microprocessor. Compiler and operating-system data structures should provide for a 64-bit size for future implementation.

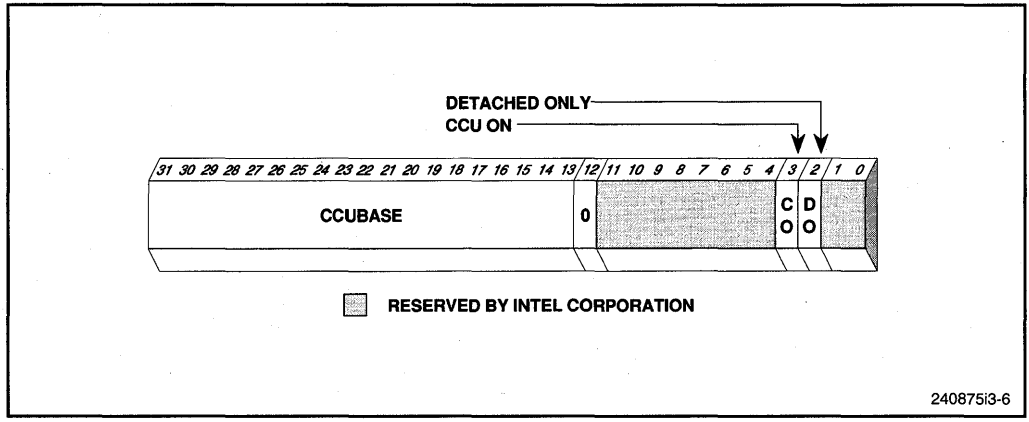


Figure 3-6. Concurrency Control Register (80860XP Only)

Table 3-6. Values of CO and DO

CO	DO	Mode
0	0	External CCU, or no CCU
0	1	*reserved*
1	0	*reserved*
1	1	Internal CCU (DCCU) only

3.14 STAT REGISTER (80860XP ONLY)

The STAT register is part of the detached CCU (concurrency control unit). As Figure 3-7 shows, it contains the following bits:

InLoop Indicates that the processor is currently executing a concurrent loop. This bit is set when a processor starts a concurrent, non-nested loop, and it is cleared when the processor enters serial code when not nested or idle. It can also be read or written directly.

Nested Indicates whether the processor is in the nested state. InLoop is copied into this bit when starting a nested loop. Otherwise, it can be read or written directly.

Detached Always contains the value of **ccr** bit DO.

STAT is architecturally a 64-bit register. Compiler and operating-system data structures should provide for a 64-bit size for future implementation.

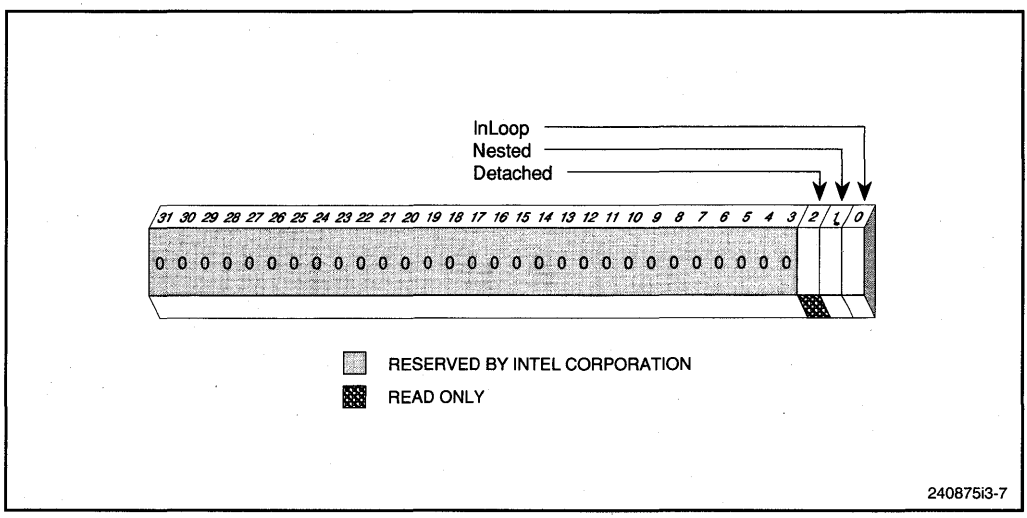


Figure 3-7. Concurrency Status Register

Addressing 4

CHAPTER 4
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 that may be dynamically selected by software in supervisor mode, i860 microprocessors also offer big endian mode, in which the most significant byte of a data item is at the lowest address. The BE bit of **epsr** selects the mode, as Chapter 3 describes. Figure 4-1 shows the difference between the two storage modes. Figure 4-2 defines by example how data is transferred from memory over the bus into a register in both modes. Big endian and little endian data areas should not be mixed within a 64-bit data word. Illustrations of data structures in this manual show data stored in little endian mode, i.e., the rightmost (low-order) byte is at the lowest memory address.

i860 microprocessors always fetch instructions with little endian addressing. This implies that instruction codes appear differently than documented here when accessed as big endian data. Intel Corporation recommends that disassemblers running in a big endian system convert instructions that have been read as data back to little endian form and present them in the format documented here.

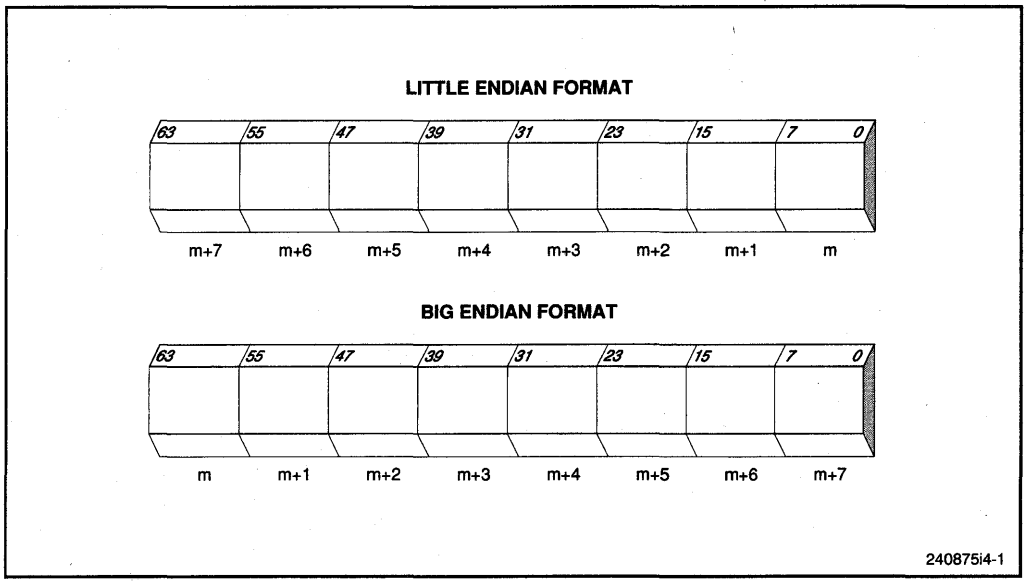


Figure 4-1. Memory Formats

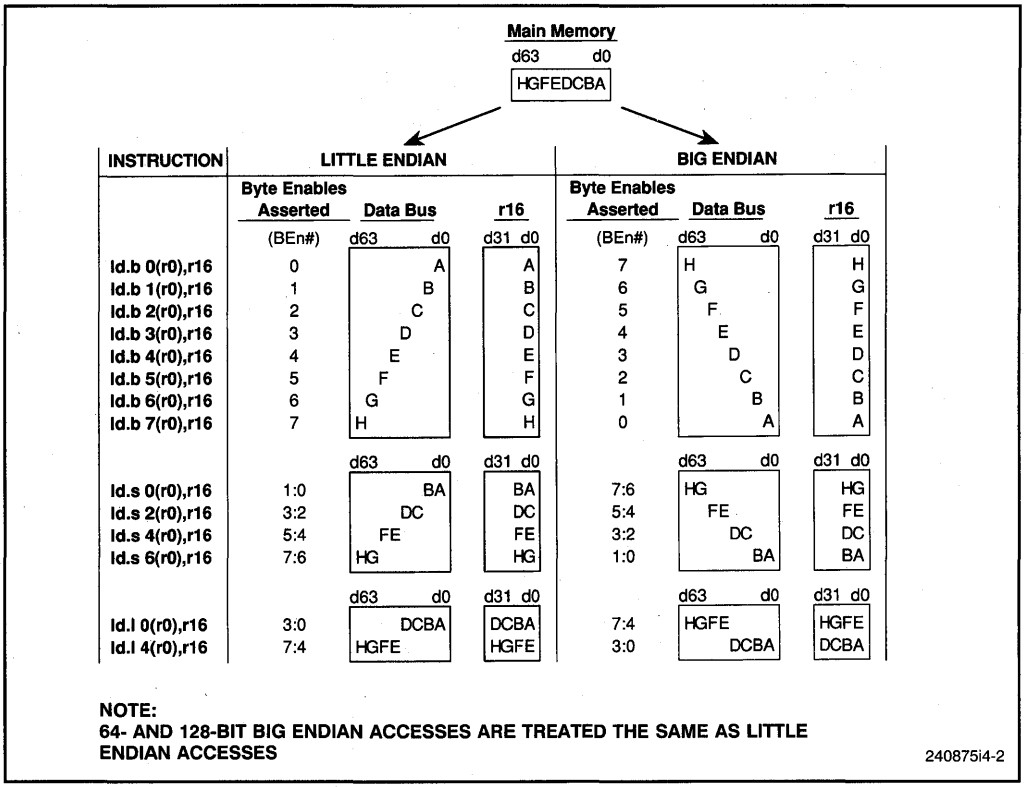


Figure 4-2. Big and Little Endian Memory Transfers

Page directories and page tables are also accessed in little endian mode, regardless of the value of the BE bit. Operating systems, therefore, must maintain these tables in little endian mode, either by accessing the tables only while BE = 0 or by using properly offset addresses to load and store PTEs. Because all PTEs are 32 bits long, software running in big endian mode must complement bit 2 of the 32-bit target address to produce the offset-by-4 address that will be transformed to the desired memory location by big endian processing.

Big endian mode affects not only the memory load and store instructions but also the **ldio, stio, ldint,** and **scyc** instructions.

4.1 ALIGNMENT

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

- A 128-bit value is aligned to an address divisible by 16 when referenced in memory (i.e., the four least significant address bits must be zero).
- A 64-bit value is aligned to an address divisible by eight when referenced in memory (i.e., the three least significant address bits must be zero).

- A 32-bit value is aligned to an address divisible by four when referenced in memory (i.e., the two least significant address bits must be zero).
- A 16-bit value is aligned to an address divisible by two when referenced in memory (i.e., the least significant address bit must be zero).

4.2 VIRTUAL ADDRESSING

When address translation is enabled, the processor maps instruction and data virtual addresses into physical addresses before referencing memory. This address transformation is compatible with that of the Intel386 and Intel486 microprocessors and implements the basic features needed for page-oriented virtual-memory systems and page-level protection.

The address translation is optional. Address translation is disabled when the processor is reset. It is enabled when a store (**st.c**) to **dirbase** sets the ATE bit. The operating system typically does this during software initialization. Address translation is disabled again when **st.c** clears the ATE bit. The ATE bit must be set if the operating system is to implement page-oriented protection or page-oriented virtual memory.

4.2.1 Page Frame

A *page frame* is a unit of contiguous addresses of physical main memory. A *page* is the collection of data that occupies a page frame when that data is present in main memory or occupies some location in secondary storage when there is not sufficient space in main memory.

– XR –

Page frames begin on four-Kbyte boundaries and are fixed in size.

– XP –

The i860 XP microprocessor supports two sizes of pages and page frames: four Mbytes and four Kbytes. Four-Kbyte page frames begin on four-Kbyte boundaries and are fixed in size. Four-Mbyte page frames begin on four-Mbyte boundaries and are fixed in size.

The four-Kbyte address transformation is compatible with that of the Intel386 and Intel486 microprocessors.

4.2.2 Virtual Address

A virtual address refers indirectly to a physical address by specifying a page and an offset within that page. Figure 4-3 shows the formats of virtual addresses.

– XR –

There is a single format for all pages.

– XP –

The format for virtual addresses that refer to four-Mbyte pages is different from that of four-Kbyte pages.

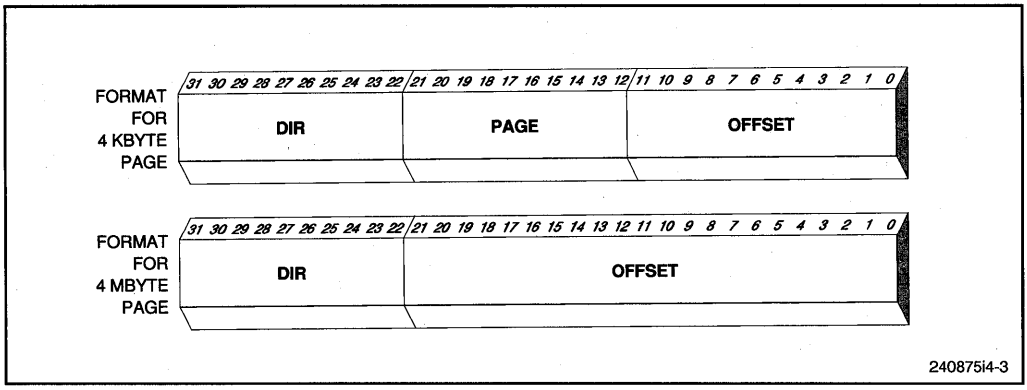


Figure 4-3. Formats of Virtual Addresses

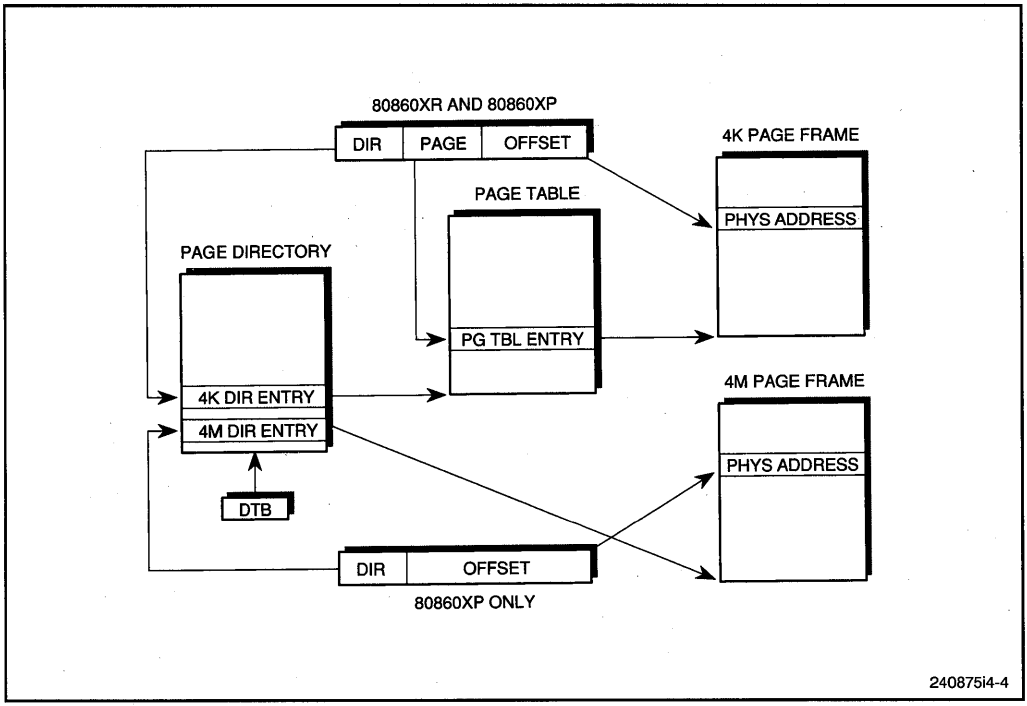


Figure 4-4. Address Translation

Figure 4-4 shows how i860 microprocessors convert the DIR, PAGE, and OFFSET fields of a virtual address into the physical address by consulting page tables. The addressing mechanism uses the DIR field as an index into a page directory. For 4K pages, it 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.

For 4M pages (80860XP only), the page directory entry determines the page address, and the OFFSET field addresses a byte within that page table.

4.2.3 Page Tables

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

At the highest level is a page directory.

– XR –

The page directory addresses up to 1K page tables of the second level.

– XP –

The page directory holds up to 1K entries that address either page tables of the second level or four-Mbyte pages.

A page table of the second level addresses up to 1K four-Kbyte pages. All the tables addressed by one page directory, therefore, can address 1M four-Kbyte pages.

Whether four Mbyte pages, four Kbyte pages, or some combination of the two are used, one page directory can cover the processor's entire four gigabyte physical address space (1K page directory entries × 4M page or 1K page directory entries × 1K page table entries × 4K page).

The physical address of the current page directory is stored in the 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.

4.2.4 Page-Table Entries

Page-table entries (PTEs) have one of the formats shown by Figure 4-5.

4.2.4.1 PAGE FRAME ADDRESS

The page frame address specifies the physical starting address of a page.

– XR –

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.

– XP –

In a page directory, the page frame address is either the address of a page table or the address of the four Mbyte page frame that contains the desired memory operand. In a second-level page table, the page frame address is the address of the 4Kbyte page frame that contains the desired memory operand.

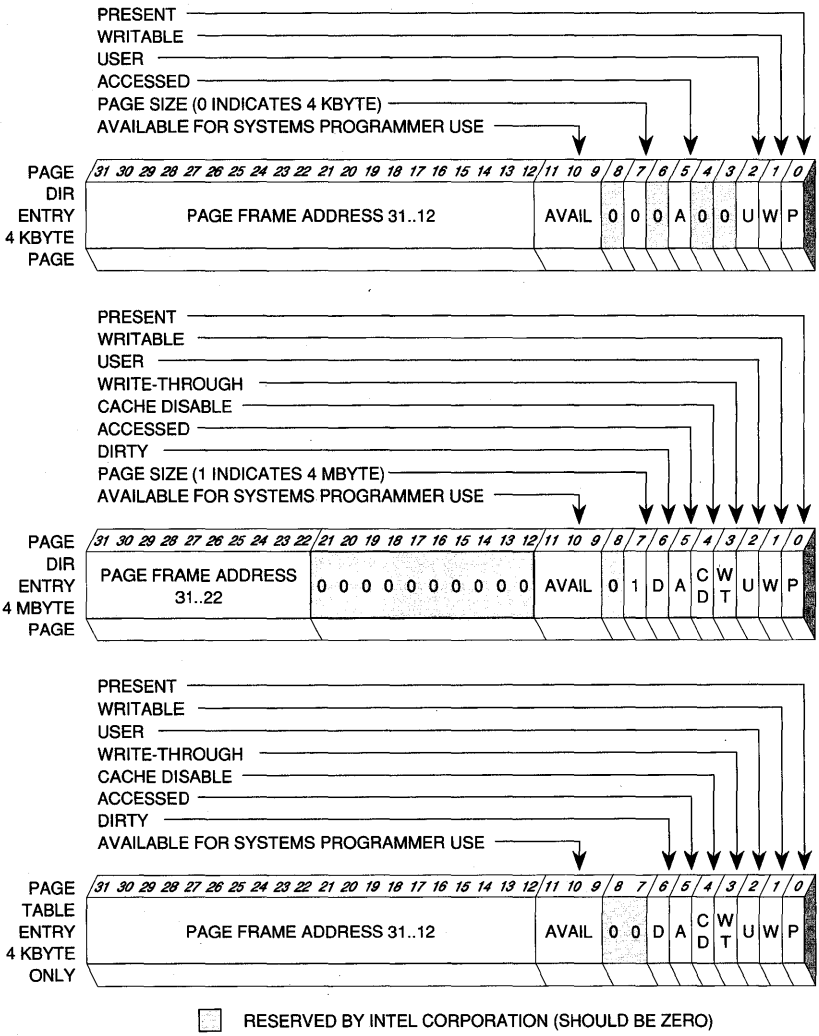


Figure 4-5. Formats of Page Table Entries

4.2.4.2 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. Figure 4-6 illustrates the format of a page-table entry when P = 0.

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. Refer to Chapters 9 and 10 for more information on trap handlers.

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.

4.2.4.3 WRITABLE AND USER BITS

The W (writable) and U (user) bits are used for page-level protection, which the processor 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:

Supervisor level (U = 0) — For the operating system and other systems software and related data.

User level (U = 1) — For applications procedures and data.

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

- The processor 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 trap mechanism is described in Chapters 9 and 10; the **trap** instruction is described in Chapter 7.)
- The processor 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 processor enters user level.

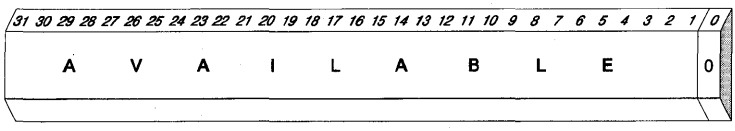


Figure 4-6. Invalid Page Table Entry

With the U bit of **psr** and the W and U bits of the page table entries, i860 microprocessors implement 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 not set causes a trap.
- When at user level, a store (**st.c**) to certain control registers is ignored.
- When at user level, privileged instructions (**ldio, stio, scyc, ldint**) have no effect.

When the processor 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 processor 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 not set causes a trap.

When the processor 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.

4.2.4.4 WRITE-THROUGH BIT

The write-through bit controls caching policy. Refer to Chapter 5 for more information.

– XR –

The i860 XR microprocessor does not implement a write-through caching policy for the on-chip instruction and data caches; however, the WT (write-through) bit in the second-level page-table entry does determine internal caching policy. If WT is set in a PTE, on-chip data caching from the corresponding page is inhibited. This is logically consistent with write-through, because all writes update main memory. (Note, however, that instruction caching is not inhibited.) If WT is clear, the normal write-back policy is applied to data from the page in the data cache. The WT bit of page directory entries is not referenced by the processor, but is *reserved*.

– XP –

The i860 XP microprocessor implements both write-back and write-through caching policies for the on-chip data cache. If WT is set, the write-through policy is applied to data from the corresponding page. If WT is clear, the normal write-back policy is applied to data from the page.

For four-Mbyte pages, the WT bit of the page directory entry is used. For four-Kbyte pages, only the WT bit of the second-level page table entry is used; the WT bit of the page directory entry is not referenced by the processor, but is *reserved*.

– XR –

To control external caches, the PTB output pin reflects either CD or WT depending on the PBM bit of **epsr**.

– XP –

The value of the WT bit is driven externally on the PWT pin, so that external caches can employ the same policy used internally.

4.2.4.5 CACHE DISABLE BIT

If a page's CD (cache disable) bit is set, data from the page is not placed in the instruction or data caches. Clearing CD permits the processor to place data from the associated page into internal caches.

– XR –

Only the CD bit of the second-level page-table entry is used. The CD bit of page directory entries is not referenced by the processor, but is *reserved*.

– XP –

For four-Mbyte pages, the CD bit of the page directory entry is used. For four-Kbyte pages, only the CD bit of the second-level page table entry is used; the CD bit of the page directory entry is not referenced by the processor, but is *reserved*.

– XR –

To control external caches, the PTB output pin reflects either CD or WT depending on the PBM bit of **epsr**.

– XP –

The value of the CD bit is driven externally on the PCD pin, so that cacheability can be the same in both internal and external caches.

4.2.4.6 ACCESSED AND DIRTY BITS

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

The processor sets the A-bit before a read or write operation to a page. For four-Kbyte pages, it sets the A-bit of both levels of page tables.

The processor tests the dirty bit before a write, and, under certain conditions, causes traps. The trap handler then has the opportunity to maintain appropriate values in the dirty bits. For four-Mbyte pages, the D bit of the page directory entry is used. For four-Kbyte pages, only the D bit of the second-level page table entry is used; the D bit of the page directory entry is not referenced by the processor, but is *reserved*. The precise algorithm for using these bits is specified in Section 4.2.5.

An operating system that supports paged virtual memory can use the D and A 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 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 (except during a locked sequence, when a trap occurs instead). When a data-access fault occurs, the trap handler sets the D bit if an allowable write is being performed, then reexecutes 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 processor automatically asserts the LOCK# signal while testing and setting the A bit.

4.2.4.7 PAGE TABLES FOR TRAP HANDLERS

– XR –

It is not strictly necessary to ensure that A=1 in trap handler pages, because the i860 XR microprocessor automatically deasserts the LOCK# signal before entering the trap handler. However, for compatibility with the i860 XP microprocessor, it is recommended that A be preset as required by the i860 XP microprocessor.

– XP –

When paging is enabled (ATE=1), software that creates page tables and directories must assure that A=1 always in the PTEs and PDEs for the code pages of the trap handler and the first data page accessed by the handler. Preallocation of these pages is required in case a trap occurs during a lock sequence. Otherwise, recursive traps would be generated, as the A-bit would need to be set by the translation hardware while the lock pin is active, which is a trapping situation in itself.

4.2.4.8 COMBINING PROTECTION OF BOTH LEVELS OF PAGE TABLES

For any four-Kbyte page, the protection attributes of its page directory entry may differ from those of its page table entry. The processor computes the effective protection attributes for a page by examining the protection attributes in both the directory and the page table and choosing the more restrictive of the two.

4.2.5 Address Translation Algorithm

The algorithms below define 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.

Algorithm for i860 XR microprocessor:

1. Read the PTE (page table entry) at the physical address formed by DTB:DIR:00. Note that the data cache is *not* accessed during PTE fetches; therefore, the operating system must ensure that the page table is not in the cache.

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-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#, refetch and check the PTE, set A, and store the PTE, deasserting LOCK# during the store.

7. Locate the PTE at the physical address formed by PFA1:PAGE:00.

8. Perform the P, A, W, and U checks as in steps 3 through 6 with the second-level PTE.

9. If D in the PTE is clear and the operation is a write, generate a data-access fault.

10. Form the physical address as PFA2:OFFSET.

Algorithm for i860 XP microprocessor:

1. Read the PDE (Page Directory Entry) at the physical address formed by DTB:DIR:00.

2. If P in the PDE is zero, generate a data- or instruction-access fault.

3. If W in the PDE is zero, the operation is write, and either the U bit of the PSR is set or WP = 1, generate a data-access fault.

4. If the U bit in the PDE is zero and U bit in the **psr** is set, generate a data- or instruction-access fault.

5. If A in the PDE is zero and the TLB miss occurred inside a locked sequence, generate a data or instruction access fault. (The trap allows software to set A to one and restart the sequence. This helps external bus hardware determine unambiguously what address corresponds to a locked semaphore.)

6. If bit 7 of the PDE is one (four-Mbyte page), and the operation is write, and D = 0 in the PDE, generate a data-access fault.

7. If A = 1 in the PDE, continue at step 11. Otherwise, assert LOCK#.

8. Perform the PDE read as in step 1 and the P, W and U bit checks as in steps 2 through 4.

9. Write the PDE with A bit set.

10. Deassert LOCK#.

11. If bit 7 of the PDE is one (four-Mbyte page), form the physical address as PFA1:OFFSET, and exit address translation. In this case, PFA1 is 10 bits and OFFSET is 22 bits.

12. The remaining steps are for four-Kbyte pages. If the A-bit in the PDE was zero before translation began, assert LOCK#.

13. Fetch the PTE at the physical address formed by PFA1:PAGE:00.

14. Perform the P-, W-, U-, and A-bit checks as in steps 2 through 5 with the second-level PTE. If A = zero in the PTE, and the TLB miss occurred inside a locked sequence, generate a data or instruction access fault. LOCK# remains active.

15. If the operation is write, and D in the PTE is zero, generate a data access fault.

16. If the A-bit in the PDE was already active before translation began, and the A-bit in the PTE is already active, go to step 20.

17. If LOCK# is not already active, assert it and refetch the PTE.

18. Perform the U-, W-, and P-bit checks and A-bit setting in the PTE as in steps 8 through 9. Do the locked write update of the PTE to unlock the bus, even if the A-bit in the PTE is already one.

19. Deassert LOCK#.

20. Form the physical address as PFA2:OFFSET. In this case, PFA2 is 20 bits and OFFSET is 12 bits.

During translation, the processor looks only in external memory for page directories and page tables. The data cache is not searched. Therefore, any code that modifies page directories or page tables must keep them out of the cache. The tables should be kept in noncacheable memory or in write-through pages or should be flushed from the cache.

The processor expects page directories and page tables to be in little endian format. The operating system must maintain these tables in little endian format either by setting BE to zero when manipulating the tables or by complementing bit two of the 32-bit address when loading or storing entries.

4.2.6 Address Translation Faults

An address translation fault can be signalled as either an instruction-access fault or a data-access fault. (Refer to Chapters 9 and 10 for more information on this and other faults.) The instruction that causes the fault can be reexecuted by the return-from-trap sequence defined in Chapters 9 and 10.

On-Chip Caches 5

CHAPTER 5
ON-CHIP CACHES

By holding data, instructions, and address translation on-chip, the caches of the i860 XP microprocessor provide the following advantages:

1. Low chip count for the CPU subsystem.
2. Wide processor-to-cache path: 16 bytes for data, 8 bytes for instructions.
3. Fast access without requiring much additional high-speed design in the system. The fast cache-access circuitry is hidden on chip; the external bus can respond more slowly without significantly degrading performance.

The caches of the i860 XP microprocessor differ from those of the i860 XR microprocessor in size, multiprocessor orientation, and other details. A single version of an operating system can execute interchangeably on either processor by examining the processor type field of **psr** and acting accordingly.

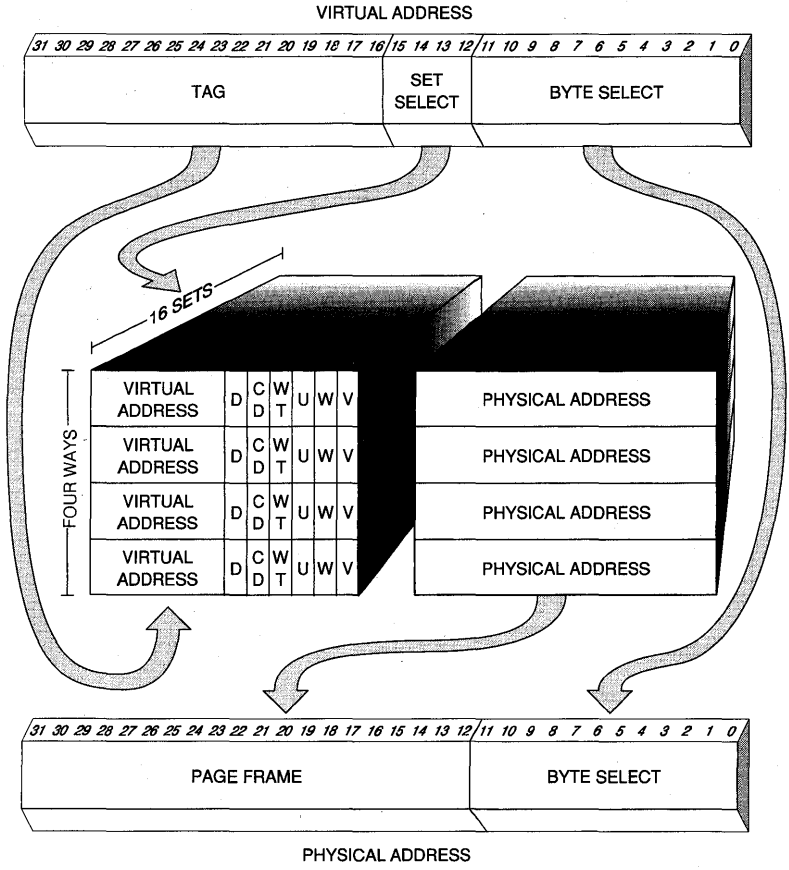
5.1 ADDRESS TRANSLATION CACHES

– XR –

The i860 XR microprocessor has four-Kbyte pages. A translation look-aside buffer (TLB) caches address translation information from the page tables. The TLB (see Figure 5-1) is a 64-entry, four-way, set associative cache. The TLB functions when paging is enabled. When a page is first accessed, its translation information is saved in the TLB along with other page attributes, such as access rights and cacheability. Every address translation operation looks up the virtual address in the TLB. Only if the necessary paging information is not in the cache must the paging tables in memory be referenced. The TLB employs a random replacement algorithm to choose which of the four ways to replace.

– XP –

The i860 XP microprocessor allows both four-Kbyte and four-Mbyte page sizes, and a separate translation look-aside buffer (TLB) is used to cache address translation information for each page size. The TLB for four-Kbyte pages (Figure 5-1) has 64 entries, and the TLB for four-Mbyte pages (Figure 5-2) has 16 entries. Both are four-way set associative. The TLBs function when paging is enabled. When a page is first accessed, its translation information is saved in the appropriate TLB along with other page attributes, such as access rights and cacheability. Every address translation operation looks up the virtual address simultaneously in both TLBs. Only if the necessary paging information is not in either of the caches must the paging tables in memory be referenced. Both TLBs employ a random replacement algorithm to choose which of the four ways to replace; however, invalid (empty) ways are replaced before valid ways are overwritten.

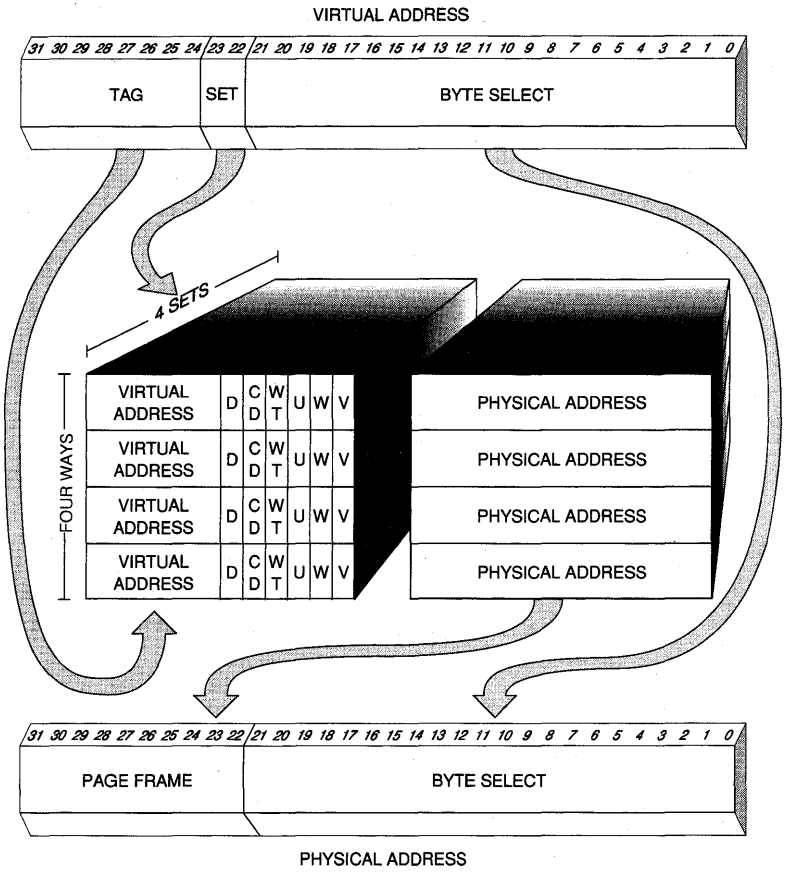


NOTES:

D	**DIRTY**
CD	**CACHE DISABLE**
WT	**WRITE-THROUGH**
U	**USER MODE**
W	**WRITABLE**
V	**VALIDITY**

240875i5-1

Figure 5-1. 4K TLB Organization



NOTES:

D	**DIRTY**
CD	**CACHE DISABLE**
WT	**WRITE-THROUGH**
U	**USER MODE**
W	**WRITABLE**
V	**VALIDITY**

240875i5-2

Figure 5-2. 4M TLB Organization

If an instruction's virtual address is found in the instruction cache, the virtual address is not translated, and code access rights are not verified. However, when an instruction's virtual address is not found in the cache, address translation does occur, and all access rights are verified. The virtual addresses of data are always translated, and access rights are always verified.

i860 microprocessors require simultaneous access to data and instruction caches, but the paging unit can service only one address translation at a time. Data address translation has higher priority in the paging unit than instruction address translation, if both are required at the same time.

Any data or instruction access fault halts address translation at once, and the TLB is not updated. If a directory read causes an access fault, the page table is not read at all.

– XR –

If the paging unit generates a fault (in setting the D bit for the first write to a nondirty page, for example), the corresponding entry remains in the TLB. Therefore, software needs to invalidate the TLB in response to paging-related data access trap or instruction access trap faults (but not for misalignment or **db** debugging traps).

– XP –

If the paging unit generates a fault (in setting the D bit for the first write to a nondirty page, for example), the corresponding entry is deleted from the TLB. Therefore, software does not need to invalidate the TLB entry in response to data access trap or instruction access trap faults.

If TLB replacement is initiated during a locked sequence generated by the **lock** instruction and if another locked sequence has to be executed to set the A-bit in the page directory or page table entry, the paging unit generates an access fault. This helps external hardware implement "locking by address" by preventing generation of nested lock sequences. (Refer to the **lock** instruction in Chapter 7.)

5.2 INTERNAL INSTRUCTION AND DATA CACHES

The i860 microprocessors have separate data and instruction caches on-chip. Having separate caches for instructions and data allows simultaneous cache look-up. Up to two instructions and 128 bits of data can be accessed simultaneously from these caches. The caches are fully transparent to applications software.

– XR –

The data cache holds eight Kbytes; the instruction cache holds four Kbytes. A line can be filled from memory with four 8-byte bus cycles. The four cycles are never interrupted by other loads or stores.

– XP –

The data and instruction caches hold 16 Kbytes each. A line can be filled from memory with a four-transfer burst. Snooping (address monitoring) is designed into both instruction and data caches, to maintain cache consistency in multiprocessor systems.

– XR –

Each cache uses *virtual* tags for internal access; there is no provision for snooping. Figure 5-3 shows how the bits of virtual addresses are mapped for caching. Because only virtual tags are used, software that uses *aliasing* (a situation in which the TLBs associate a single physical address with two or more virtual addresses) must take care not to violate intertask protection.

– XP –

Each cache has two sets of tags: *virtual* tags used for internal access, and *physical* tags used for snooping. Figure 5-3 shows how the bits of both virtual and physical addresses are mapped for caching. The presence of both virtual and physical tags supports *aliasing*, a situation in which the TLBs associate a single physical address with two or more virtual addresses.

Any area of memory can be cached, although both software and hardware can disallow certain areas from being cached – software by setting the CD bit in their page table entries; hardware by deasserting the KEN# signal for bus cycles with addresses that fall

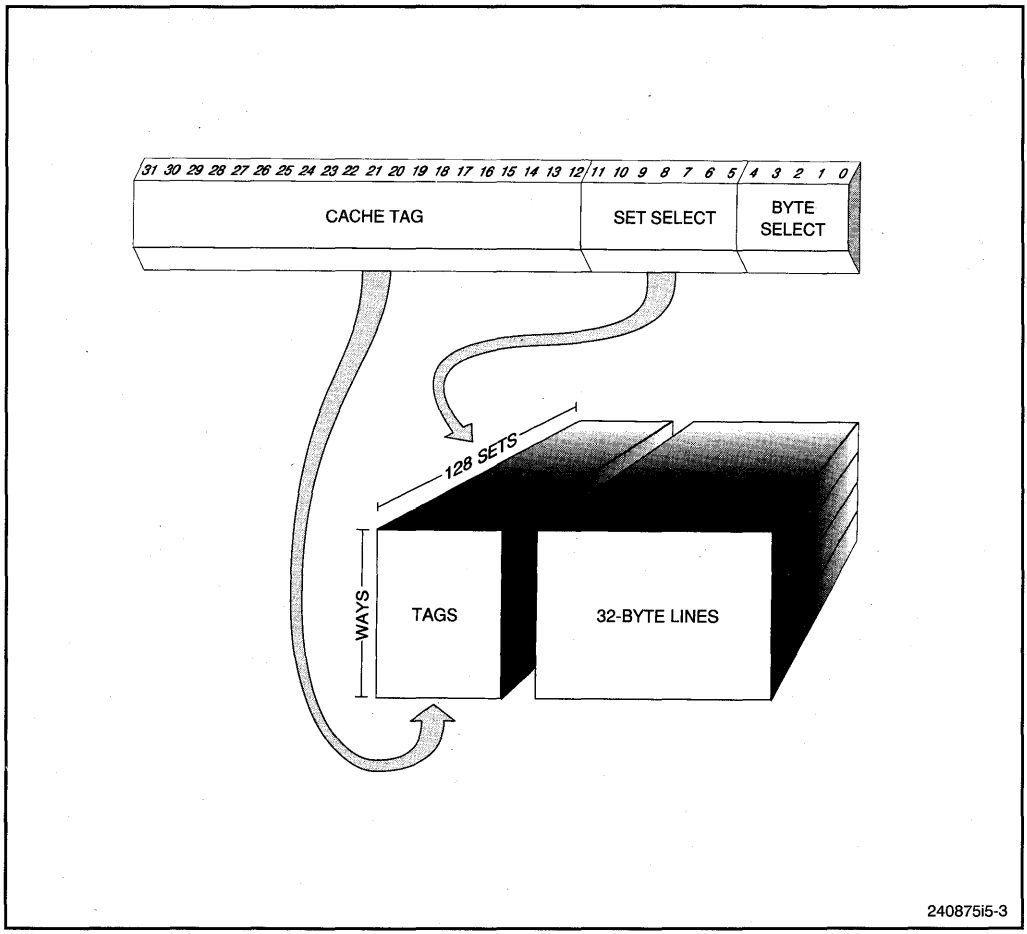


Figure 5-3. Cache Address Usage

in those areas. (In the i860 XP microprocessor, data reads from the two four-Kbyte pages pointed to by the CCUBASE field of **ccr** are not cached, either, if the CCU is activated by setting CO of the **ccr** register. This is independent of the value of KEN#.) When both software and hardware agree that a requested datum is cacheable, the processor reads an entire 32-byte line into the appropriate cache. Cache line fills are generated only for read misses, not for write misses. A store that misses the cache does not copy the missed line into cache from memory, but rather posts the datum in a write buffer, then sends it to the external bus when the bus is available.

Stores that hit the cache utilize it for two cycles (one to check the virtual tags for hit, another to update the cache line). However, the cache pipeline allows successive store hits to operate at one per cycle.

– XR –

The processor's internal write buffers can hold two successive stores, preventing a freeze upon store miss.

– XP –

The processor's internal write buffers can hold three successive stores, preventing a freeze upon store miss.

5.2.1 Data Cache

– XR –

Figure 5-4 shows the organization of the data cache. The data cache has two M (modified) bits per virtual tag, one for each half-line. If only one half-line is modified, only that half-line is written back to memory. There is no validity bit. To invalidate a virtual tag, it must be filled with a virtual address reserved for that purpose.

– XP –

Figure 5-5 shows the organization of the data cache. The data cache has two status bits per physical tag and one validity status bit for the virtual tag. A virtual tag hit is possible only when the validity bit of the virtual tag is set.

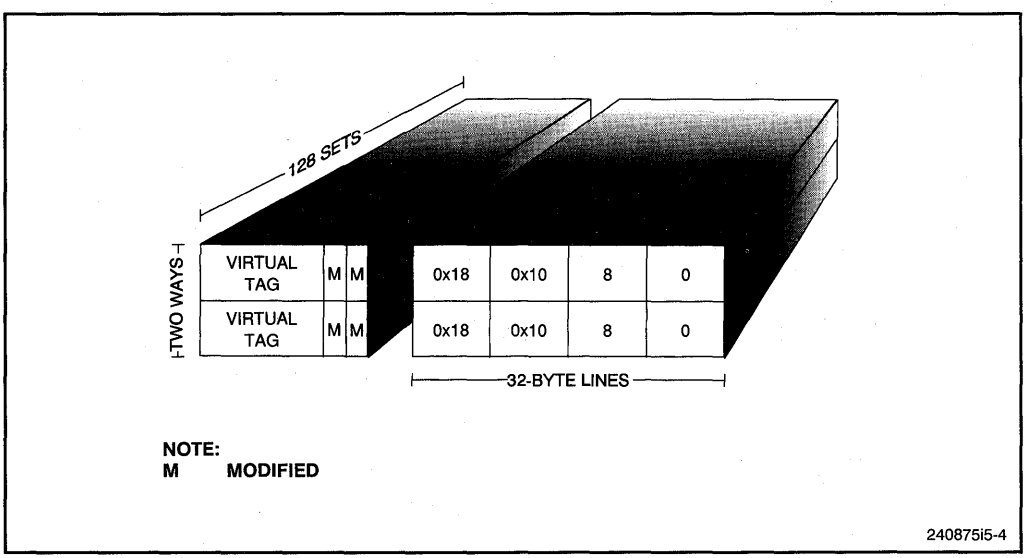


Figure 5-4. Data Cache Organization (80860XR)

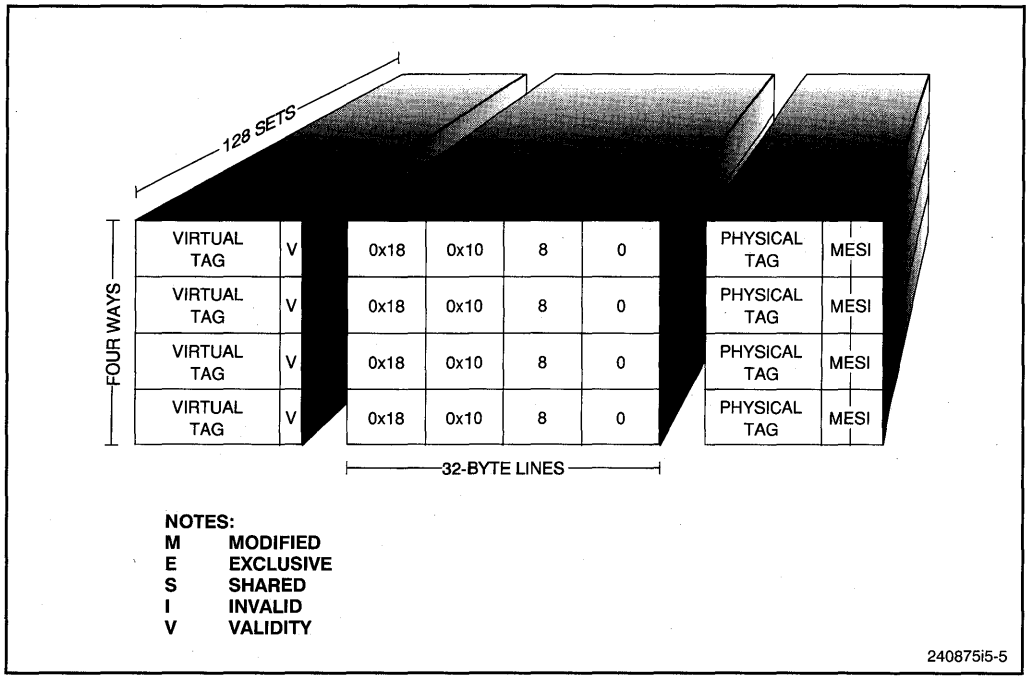


Figure 5-5. Data Cache Organization (80860XP)

5

– XR –

If an operating system allows aliases, it is possible for a physical line to appear in the cache twice, under different virtual addresses. Such a replication is undesirable, because a write to one cache line would not update the other line in the cache. Aliasing of instructions and read-only data presents no problems.

If necessary, software can implement reliable aliasing for writable, cacheable data by using only one way of the data cache (by flushing the data cache, the setting RC=10 and RB=00, for example). Because only one way is available and because the aliases necessarily have the same set selection bits, all aliases for a given physical address map to a single location in the cache.

– XP –

Alias support is built into the cache look-up algorithm. Even though a physical line may be aliased, the processor never enters the line twice in the data cache. If a virtual address is not found among the virtual tags in the data cache, a bus cycle is initiated and, at the same time, the physical tags are searched for the physical address (which by this time has been retrieved from the paging unit). For reads, if the physical address is found, the data returned from the bus is ignored, on-chip data is used, and the virtual tag is replaced with the new one. For writes, if a virtual address is not found, the write is issued on the bus, and memory is updated. If the physical address is found, the line in cache is updated, and the virtual tag is replaced with the new one. However, the cache state (M, E, or S) of the physical-address tag does not change when the virtual tag is overwritten.

Note that the BE (big endian) bit of **epsr** has no influence on data cache behavior. Data items are kept in cache in exactly the same ordering as in external memory. Byte-shifting operations invoked by the BE bit upon loads and stores occur at the input to the register files only.

5.2.1.1 DATA CACHE UPDATE POLICIES

To minimize bus traffic, a *write-back* policy is normally used. The write-back policy (also called *copy-back* and *deferred-write*) reduces bus traffic by eliminating many unnecessary writes. Writes to a line in the cache are not immediately forwarded to main memory; instead, they are accumulated in the cache. The modified cache line is written to main memory only when its cache space is needed for other data (or when a flush procedure is executed).

Under a *write-through* policy, a write request to a line in the cache triggers updates to both cache and main memory.

– XR –

The i860 XR microprocessor does not implement the write-through policy. Setting the WT bit in a page table, disables caching for the page. This is logically consistent with write-through, because all writes update main memory.

– XP –

An address decoder, for example, can select the write-through policy for writes to video RAM, where it is necessary that writes be seen on the video display. The decoder can dynamically change the update policy of the i860 XP microprocessor with each cache line by manipulating the WB/WT# input pin. Software, by setting the WT page-table bit, can select the write-through policy for specific areas of memory—those that are used for interprocessor message queues, for example. Setting WT overrides the WB/WT# pin.

A *write-once* policy combines write-through with write-back. Write-through is employed for the first write to a cache line, while subsequent writes to the same line follow the write-back policy.

– XR –

The i860 XR microprocessor does not implement the write-once policy.

– XP –

Write-once is valuable in multiprocessor systems to maintain cache consistency with the least possible bus traffic. The first write broadcasts to other processor nodes the fact that a line has been modified. Write-once is also used if a second-level cache is attached to the i860 XP microprocessor to maintain consistency between the first- and second-level caches.

5.2.2 Instruction Cache

– XR –

Figure 5-6 shows the organization of the instruction cache. The instruction cache has one validity bit for each virtual tag. As in the data cache, if an operating system allows aliases, it is possible for a physical line to appear in the instruction cache twice, under different virtual addresses. Such replication in the instruction cache is not problematic, however, because the instruction cache is read-only.

– XP –

Figure 5-7 shows the organization of the instruction cache. The instruction cache has one validity bit that is common to both virtual and physical tags. Aliasing support for instructions consists not simply of changing the virtual tag, but rather fetching a line whenever a virtual tag miss occurs. If the physical address already exists in the instruction cache, its line and its tags are overwritten. So, even though a physical line may be aliased, the processor never enters the line twice in the instruction cache.

5.2.3 Cache Replacement Algorithm

The data, instruction, and address-translation caches all use similar algorithms to choose which line of a set will be overwritten when a miss causes a line fetch.

– XR –

A pseudorandom replacement algorithm chooses which line to replace.

– XP –

The first invalid line in a set of four is replaced (in the order 0, 1, 2, 3). When there are no more invalid lines in a set, a pseudorandom replacement algorithm chooses which valid lines to replace.

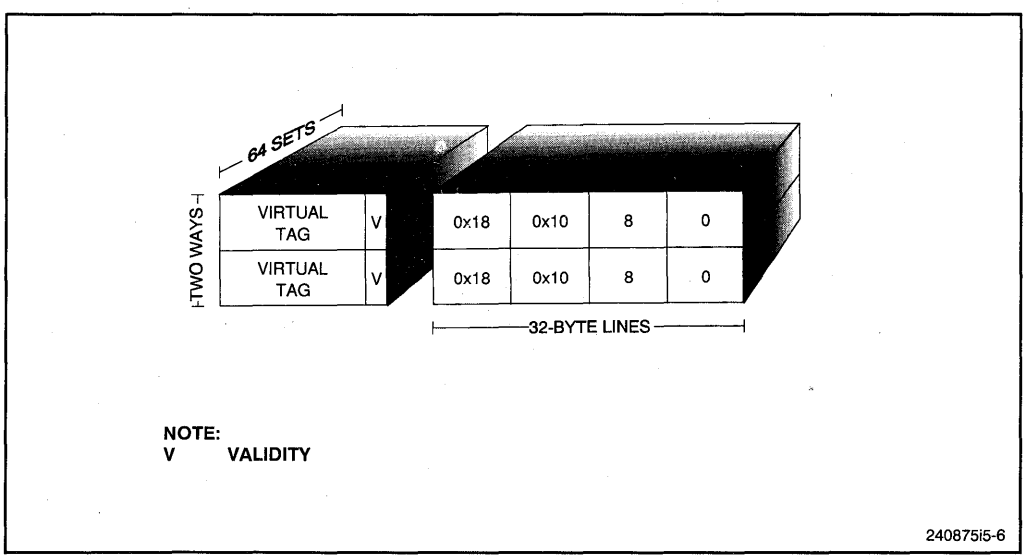


Figure 5-6. Instruction Cache Organization (80860XR)

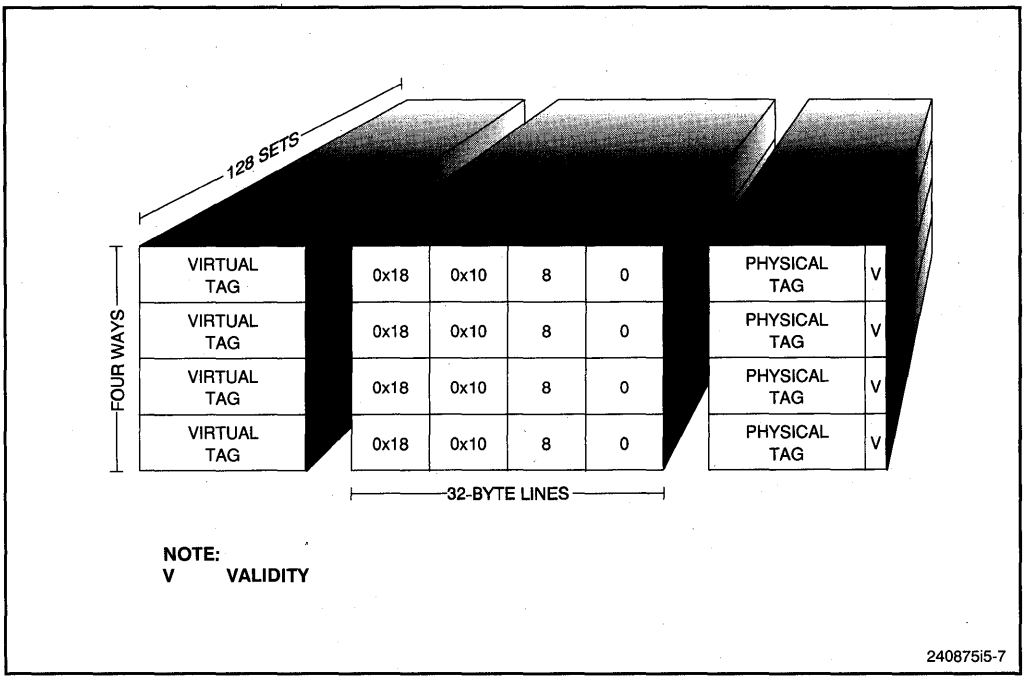


Figure 5-7. Instruction Cache Organization (80860XP)

The algorithm is controlled by counters inside the chip. RESET initializes these counters to zero, so that the "randomness" is deterministic, and two i860 XR CPUs or two i860 XP CPUs executing the same code on identical boards have exactly the same series of cache hits, misses, and replacements. Setting ITI to invalidate the caches and TLBs also resets the internal counters used for random replacement. This brings the cache-replacement mechanism to a known state without resetting the whole chip.

When the **flush** instruction is used to write back modified lines in the data cache, the flush routine must alter the RC (replacement control) field of **dirbase**. Therefore, the LFBSR is not used and replacement is not random. Instead, the block (or "way") replaced is the one selected by the RB (replacement block) field of **dirbase**.

5.2.4 Cache Consistency Protocol (80860XP Only)

The i860 XP microprocessor implements cache consistency via its use of a MESI (Modified, Exclusive, Shared, Invalid) protocol.

5.2.4.1 DATA CACHE STATES (80860XP ONLY)

Each line of the data cache of the i860 XP microprocessor can be in one of the states defined in Table 5-1. Note that the instruction cache of the i860 XP only implements the "SI" part of the MESI protocol, because the instruction cache is not writable.

Table 5-1. MESI Cache Line States (80860XP)

Cache Line State:	M Modified	E Exclusive	S Shared	I Invalid
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	...out of date	...valid	...valid	–
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line...	...does not go to bus	...does not go to bus	...goes to bus and updates the cache	...goes directly to bus

The state of a cache line can change as the result of either internal or external activity related to that line. Table 5-2 presents the line state transitions that result from internal activity of the i860 XP microprocessor in the data cache.

External cache-consistency support is provided through *inquiry cycles*. Inquiry cycles are initiated by other processors in a multiprocessor system to check whether an address is cached in the internal cache of the i860 XP microprocessor. Table 5-3 shows the line state transitions initiated by inquiry cycles.

5.2.4.2 WRITE-ONCE POLICY (80860XP ONLY)

A write-once cache policy can be implemented on the i860 XP microprocessor through use of the WB/WT# input pin. The signal on this pin is sampled in both read and write cycles. A read miss causes a line to enter either S or E after the line fill. If WB/WT# is sampled LOW at the time of NA# or the first BRDY# activation, the line enters S state, forcing the next write hit to this line to show up on the bus. If WB/WT# is sampled HIGH, the line enters E state. In write-through cycles, the state of a line is changed from S to E when WB/WT# is sampled HIGH, so that subsequent writes are not written through to the bus. Thus, if this signal is driven LOW on read cycles and

Table 5-2. Internally Initiated Cache State Transitions (80860XP)

State	Next State after Read	Next State after Write*
I	If WB/WT# = 1, E; else S Line fill	Write-through I
S	S	Write-through If WB/WT# = 1, E; else S
E	E	M
M	M	M

NOTE:
* "Write" does not include write-backs due to replacement. Those can only cause an M to I transition.

Table 5-3. Inquiry-Initiated Cache State Transitions (80860XP)

State	INV=0	INV=1
I	I	I
S	S	I
E	S	I
M	S; write back the line	I; write back the line

HIGH on write cycles, a write-once cache policy is implemented. The easiest way to implement write-once (in systems not using the 82495XP cache controller) is to tie this pin to the W/R# output of the processor.

If the WT bit in the page table entry is set, the i860 XP microprocessor ignores the WB/WT# signal for the cycles that hit that page and always performs a write-through. In other words, hardware cannot override software's selection of the write-through policy.

5.2.4.3 LOCKED ACCESSES (80860XP ONLY)

Locked accesses are those data loads and stores that occur after a **lock** instruction up to and including the first load or store after the corresponding **unlock** instruction.

On the i860 XP microprocessor, state transitions for locked accesses differ from those in Table 5-2 in ways that guarantee that locked accesses are seen by all processors in the system. Any locked load or store generates both a cache look-up and an external bus cycle, regardless of cache hit or miss.

1. In a locked read:
   a. If the required data is found in the cache in a modified (M) state, further accesses to the cache from subsequent loads or stores are delayed until data is returned from the bus. (The cache may, however, be accessed by an inquiry cycle.) If in the clock period before BRDY# the data is still found modified in the cache, the cached data is used, and the bus data is ignored. If, however, an intervening inquiry write-back changes the line to S or I state, the bus data is used.
   b. If the data is found in an unmodified (E or S) state, the data returned from the bus is used.
   c. If the data is not found in the cache, the data from the bus is used. The data is placed in the cache if it is cacheable and KEN# is also asserted.

2. A locked store is forced through the cache and issued on the bus. No more data accesses occur until the BRDY# for such a store. If the store hits the internal cache, the cache update is done after BRDY# from the bus. Note that the line written by a locked store remains in M state in spite of the write-through to the bus, because the length of the write-through is less than the line size of 32 bytes.

Locked accesses are totally *serializing* in the sense that:

1. All loads and stores that precede the **lock** instruction are issued on the bus (if they miss the cache) before the first locked access is issued. The locked access can be issued before the last BRDY# of the prior cycle if NA# is activated in response to the prior cycle.

2. No load or store after the last locked access is issued internally or on the bus until the final BRDY# for all locked accesses.

To maximize performance, instruction fetches during the locked sequence are *not* serializing. When NA# invokes pipelining, instruction fetches may be issued while locked data fetches or stores remain on the bus.

5.3 INTERNAL CACHE CONSISTENCY

Software must take care not to create inconsistencies such as the following among the internal caches (including the TLBs):

1. Changing the address space while leaving virtual-address tags from the prior space in the instruction or data cache.

2. Changing instructions in memory (or in the data cache) without changing them in the instruction cache.

3. Changing page table information in memory (or in the data cache) without changing the same information in the TLBs.

Under certain circumstances, such as I/O references, instruction modification, page-table updates, or access to shared data in a multiprocessing system, it is necessary to bypass, to invalidate, or to flush the caches. The i860 microprocessors provide the following methods for doing this.

5.3.1 Bypassing Instruction and Data Caches

There are a variety of methods to bypass the caches:

1. If deasserted during cache-miss processing, the KEN# pin disables instruction and data caching of the referenced data.

2. If the CD bit of the associated page table is set, caching of a page is disabled.

3. With the i860 XP microprocessor, if the WT bit of the associated page table is set, caching is not disabled, but writes pass through the cache. (Note that WT does not affect policy for the instruction cache, because the instruction cache is not writable. However, when an instruction from a page having the WT bit of the PTE set is placed in the data cache, the write-through policy applies just as for a data page.)

With the i860 XR microprocessor, if the WT bit of the associated page table is set, data caching is disabled, just as if CD were set. However, WT does not affect instruction caching; the i860 XR microprocessor caches instructions from WT pages.

– XR –

The value of the WT bit or the CD bit is output on the PTB pin for use by external caches.

– XP –

The value of the WT bit is output on the PWT pin and the value of the CD bit is output on the PCD pin for use by external caches.

5.3.2 Invalidating Cache Entries

Storing to the **dirbase** register with the ITI bit set invalidates each line of the instruction and address-translation caches.

– XR –

Setting ITI does not invalidate the data cache.

– XP –

In the data cache, setting ITI invalidates the virtual tags, but not the physical tags.

5.3.3 Flushing the Data Cache

The data cache is flushed by a software routine that uses the **flush** instruction. The **flush** instruction speeds up write-backs. The same effect (writing back modified lines) can be achieved with the load instruction **ld.l**, but this would be more than twice as slow – the load must first do four bus transfers to get new data, then write back the modified line. The **flush** instruction causes the write-backs without requiring a read from external memory to replace the modified line.

– XR –

The flush procedure replaces the virtual tags with addresses reserved for that purpose, effectively invalidating each cache line.

– XP –

The flush instruction invalidates virtual tags, but not physical tags.

5.3.4 Address Space Consistency

In a multitasking virtual-address system, the operating system may intentionally employ aliasing, where several processes use the same physical memory while accessing it with different virtual addresses. When the operating system switches control from one process to the next, it changes the DTB field of the **dirbase** to point to a different page directory that defines the new address space. When this happens, all caches must be invalidated: the TLBs, so that the new page directory is read into the TLBs; the data and instruction caches, so that virtual addresses from the new space don't accidently match cached virtual addresses from the old space.

– XR –

The data cache is invalidated by executing the flush procedure. The TLB and instruction cache are invalidated by setting the ITI bit when writing to **dirbase**.

– XP –

The caches are invalidated by setting the ITI bit when writing to **dirbase**. Invalidating the instruction cache invalidates both the physical and the virtual tags, because the instruction cache has one status (valid) bit, when is common to both physical and virtual tags. In the data cache, setting ITI does not invalidate physical tags. However, any modified lines will eventually be written back when their space is required for lines from the new address space or when external agents on the bus express a need for the modified data via inquiry cycles.

Note that the operating system code that flushes the caches must be present during the flushing. Typically this code has the same virtual address for all processes.

NOTE

The mapping of the page(s) containing the currently executing instruction, the next six instructions, and any data referenced by these instructions should not be different in the new page tables when the DTB is changed.

Enabling or disabling address translation (via the ATE bit) is similar to changing the DTB, in that the address mapping is changed.

– XR –

The instruction cache must be invalidated and the data cache must be flushed prior to changing ATE.

– XP –

The virtual tags in the data and instruction cache must be invalidated prior to changing ATE.

5.3.5 Instruction Cache Consistency

When software modifies a page containing instructions (as when a debugger replaces an instruction with the **trap** instruction to set a breakpoint), the instruction cache can become inconsistent for any of the following reasons:

- Because the data cache uses a write-back policy, changes to cached instruction pages do not immediately update memory.
- Changes by software to instructions do not automatically update the instruction cache.
- Instruction cache misses are not checked in the data cache.

Software must ensure that modified lines containing instructions are written to main memory before the instruction cache tries to read them. There are two methods for this:

1. Flush the data cache using the **flush** instruction. Note that to make the instruction cache consistent with the data cache, the data cache must be flushed *before* invalidating the instruction cache.

2. Mark all instruction pages as WT (write through) so that modifications to instructions are immediately written to memory. This is the better alternative.

In either case, the instruction cache must be invalidated (by a store to **dirbase** with ITI set) after a code page has been modified, so that the updated instructions will be read from memory.

5.3.6 Page Table Consistency

When the operating system modifies page tables or directories, a TLB can become inconsistent with the modifications for any of the following reasons:

- Because the data cache uses a write-back policy, updates to cached page tables do not immediately update memory.
- Changes by software to page tables do not automatically update the TLB.
- The i860 microprocessors search only external memory for page directories and page tables in the translation process. The data cache is not searched. (Data is not transferred from the data cache to the TLBs during TLB replacement cycles.)

Software must ensure that modified lines containing page table entries are written to main memory before the paging unit tries to read them. There are two methods for this:

1. Keep page tables and directories in noncacheable memory or write-through pages.

2. Flush the data cache using the **flush** instruction. Note that to make the TLBs consistent with the data cache, the data cache must be flushed *before* invalidating the TLBs.

In any case, the TLBs must be invalidated (by a store to **dirbase** with ITI set) after a page table or directory has been modified, so that the updated entries will be read from memory.

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 is not in the data cache). The i860 microprocessors do not check these protection bits on cache line write-back. Thus, a trap handler can service a data access trap for D-bit zero by setting D = 1.

– XR –

Software must then invalidate the TLB.

– XP –

The processor itself invalidates the TLB entry that causes a data access trap.

When setting the P or A bits, 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.

Two potential TLB inconsistencies are avoided automatically by the i860 XP microprocessor.

1. If the paging unit issues a write cycle (to set the A bit, for example), this cycle is snooped by the data cache for invalidation.

2. Any TLB entry that causes a DAT or IAT is automatically invalidated.

5.3.7 Consistency of Cacheability

Normally, an operating system ensures that the page attributes (CD and WT) of a memory access are consistent with the cache contents. If, however, the operating system fails to maintain consistency and changes the CD or WT bits while related lines are in the cache, the processor gives priority to cache state. For example:

1. If a read or write request is to a noncacheable page (CD=1), but the data (or code) is found in cache, the request is satisfied by the cache, and no external cycle is issued.

2. On the i860 XP microprocessor, if a store to a write-through page (WT=1) hits a cache line in E or M state, no write-through cycle is issued; only the cache is updated.

5.3.8 Protection Consistency

– XR –

The ITI bit of **dirbase** should be set when changing the WP bit of the **epsr** so that the protection bits of page tables can be reinterpreted as they are reloaded into the TLBs.

– XP –

The TLBs do not cache the WP bit of the **epsr**. Therefore, there is no need to invalidate the TLBs when changing the WP bit.

5.3.9 Load Pipe Consistency

The **pfld** (pipelined floating-point load) instruction facilitates transfer of data from memory to registers, and avoids placing data in the data cache. When large amounts of data are used, **pfld** allows the programmer to keep rarely-used data out of the cache. The i860 microprocessors ensure consistency between cached data and **pfld** references. They check the data cache and, upon a data cache hit to a modified line, forward data from cache into the three-stage **pfld** pipeline.

5.3.10 Summary

Table 5-4 summarizes flush and invalidation requirements, assuming that WT is set in the PTEs of instruction and page-table pages.

Table 5-4. Summary of Cache Flushing and Invalidation

Action	i860™ XR CPU		i860 XP CPU	
	Flush Data Cache	Invalidate Caches (ITI)	Flush Data Cache	Invalidate Caches (ITI)
Setting A	No	No	No	No
Setting P	No	No	No	No
Clearing P	Yes[1]	Yes	No	Yes
Setting D	No	Yes	No	No
Changing protection (U,W)	No	Yes	No	Yes
Setting CD or WT	Yes	Yes	Yes	Yes
Changing PFA in a used[2] PTE	Yes	Yes	No	Yes
Changing **dirbase** DTB	Yes	Yes	No	Yes
Changing **dirbase** ATE	Yes	Yes	No	Yes
Changing **epsr** WP	No	Yes	No	No
Setting **ccr** DO and CO	–	–	Yes[3]	Yes[3]
Modifying code	Yes	Yes	No[4]	Yes

NOTES:
1. When marking a page "no longer present," a flush must be done before the page frame address in the PTE is changed; but the flush can be done after clearing P.
2. "Used" means a PTE that at some past time had P set. (Flush the data cache before making the PTE change; ITI after the change.)
3. Only if data from either of the CCU pages could have been cached.
4. Assuming all instructions and their page directories and page tables are in write-through or noncacheable pages.

Concurrency Control 6

CHAPTER 6
CONCURRENCY CONTROL

6.1 DETACHED CCU

The i860 XP supports parallel processing, where multiple processors work simultaneously on different parts of the same problem. The Concurrency Control Unit (CCU) controls work sharing among CPUs in multiprocessor systems. The CCU is a VLSI chip that allows multiple processors to work together to execute portions of a single program in parallel. The CCU performs the iteration assignment and synchronization for loop parallelization. Accesses to the CCU for synchronization are much faster than accesses to shared memory semaphores.

To take advantage of the parallel architecture, software must be compiled by parallelizing compilers that generate instructions to access the CCU. The CCU is memory mapped, and its internal registers are accessed via integer memory load and store instructions. However, such instructions cannot run on a system that does not include a CCU. To allow an application compiled for parallel execution to run on any system based on the i860 XP microprocessor, a "Detached Only" CCU (DCCU, also referred to as "internal CCU") is implemented in the i860 XP microprocessor. The DCCU is a compatible subset of the external CCU, consisting of the minimal set of features required for a single CPU. The DCCU alone increases neither performance nor concurrency, but does allow software designed for parallel processing to run unmodified on a single CPU.

6.2 DCCU INITIALIZATION

After reset, the i860 XP microprocessor DCCU is disabled (the CO and DO bits in the **ccr** are clear). To enable the DCCU, software must set the CO and DO bits in **ccr** after initializing CCUBASE to point to the CCU address space.

Before enabling the CCU, the operating system must invalidate the TLB and flush the data cache to make sure that they do not contain data from the pages of the CCU address space. The TLB is invalidated by setting ITI = 1 in the **dirbase** register. If the two pages at the CCUBASE address may have been cached, the **flush** instruction must be used once per each line of the data cache to invalidate the physical address of the cache entry. The flush is not needed if page tables or external hardware have prohibited caching of data from the CCUBASE pages.

Neither the external CCU nor the DCCU can be accessed within four instructions after **ccr** is modified.

6.3 DCCU ADDRESSING

CCU facilities are memory-mapped, manipulated by integer load and store instructions. The DCCU is memory-mapped to a single four-Kbyte user page. When the DCCU is active, all accesses to this page are satisfied by the DCCU, and no external bus cycle is generated. The address space of two adjacent pages beginning on an eight-Kbyte boundary is reserved for the CCU. The first (lower address) page contains locations accessible in user mode (which includes the DCCU registers), and the second page contains locations accessible in supervisor mode (used for external CCU only). The base address of these pages is specified by the CCUBASE field in **ccr**. Accesses to the second page in DCCU-only mode have no effect on the DCCU, and are treated as normal memory accesses.

When the DCCU is active, accesses to its address page use only the virtual address; no address translation is done. However, the accesses to an external CCU go through normal address translation. The OS should make sure that the page table entries for the CCU pages are set so that no fault occurs during address translation. If an external CCU is used, the two PTEs for the CCU should have CD = 1 (caching disabled) and should have page frame addresses that match the external hardware addresses of the CCU. Accesses to the DCCU that cause a TLB miss do not cause the PTE to be loaded into the TLB.

6.4 DCCU INTERNALS

The DCCU consists of an address decoder, a 32-bit counter (NEWCURR), and a status register (STAT), which has three bits of state information (InLoop, Nested, and Detached). InLoop, Nested and Detached correspond to the same bits of the external CCU STAT register. The Detached bit always reflects the value of the DO bit in **ccr**.

Several addresses within the DCCU memory page are decoded to cause actions to the NEWCURR, InLoop, and Nested state bits. The CCU address to be accessed is specified by address bits 11–3. The valid CCU addresses are shown in Table 6-1 with their mnemonics. Loads from any other addresses within the DCCU memory page return zero; stores to any other addresses have no effect. Access to the DCCU by any load or store instructions other than **ld.x** and **st.x** produce undefined results.

Assemblers should encode address bits 2–0 as zero for accesses in little-endian mode. However, in big-endian mode (**epsr** BE bit = 1), DCCU accesses should have address bit 2 active. Thus, software for big-endian access to the DCCU must differ from little-endian software. This allows an external CCU to be accessed in both big and little endian modes.

When reading from the DCCU, the access latency is the same as reading data from the data cache — the data is ready for use as a source by the second instruction after the load. The first instruction after the load may use the data, but that instruction will experience a one-clock freeze before the data becomes available.

Table 6-1. CCU Addresses

Mnemonic	A11–A8	A7–A4	Little Endian A3–A0	Big Endian A3–A0
cbr_*i*	0000	0**abc**	**d**000	**d**100
cget	1111	0110	0000	0100
cnewcurr	1111	1100	0000	0100
cstat	1111	1100	1000	1100
cstatci	1111	1101	0000	0100
cstatn	1111	1101	1000	1100
cclm	1111	1110	1000	1100
cver	1111	1111	1000	1100

NOTE: Variable *i* is a four-bit index formed by A6–A3. Let its binary form be represented by the symbols **abcd**.

6.5 DCCU PROGRAMMING

Compilers employ the CCU by emitting code sequences that access the register locations via the load and store instructions of the CPU. For example, the code sequences **cstart** and **crepeat** tell the CCU to perform scheduling functions. One processor, called the *lead processor*, begins executing the serial code leading to a parallel loop. The lead processor reaches the **cstart** code sequence at the beginning of the parallel section. This code sequence causes the lead processor to broadcast the program counter, the frame pointer, and the number of iterations in the loop to the other processors. All processors that are ready to start processing are also assigned an iteration number. In this way, each processor starts off knowing the total number of iterations to be executed, the starting program state, and the iteration to be processed. Other important information that does not change dynamically (for example, the number of processors that are working on the loop) is stored in shared memory pages.

When a processor finishes an iteration, it executes the **crepeat** code sequence that the compiler generates at the end of the loop code. If another iteration remains to be performed, the processor branches back to the top of the loop, receives a new iteration number, and begins processing.

Eventually, one of the processors will execute the last iteration of the loop. A processor can recognize this condition, because it was notified in the **cstart** sequence of the total number of iterations to be performed. That processor begins executing the serial code beyond the end of the loop, and becomes the new lead processor for the next loop.

Other processors will find that there are no more iterations to be performed. Each of these processors waits until the new lead processor signals it with the next **cstart** code sequence to begin parallel execution again.

Another pair of code sequences, **cadvance** and **cawait**, control task synchronization. Together, they implement event-count synchronization using the broadcast registers contained in the CCU. They allow parallel execution of code that otherwise could not be executed in parallel because of data dependencies. The compiler assigns a broadcast register to use as an event count for each dependency in a loop. The **cadvance** sequence increments the event count. The **cawait** sequence causes the process to wait until the event count reaches a given count, usually the current loop count. With these controls, a given loop iteration can wait until lower-numbered iterations on which it depends have passed the point of dependency.

When the **ccr** is configured for "internal CCU only," load and store instructions within these high-level code sequences access the locations of the DCCU instead of the locations of an external CCU. The action that results from loading and storing each of the DCCU locations is described below. It is assumed that the register *isrc2* contains the value of CCUBASE in the upper 20 bits, and zeros in the lower 12 bits. The effects listed for these accesses assume that the **ccr** is configured with CO = DO = 1. When the **ccr** is configured for the external CCU (CO = DO = 0), the effects are determined by the external CCU hardware.

ld.l l%cbr_*i* (*isrc2*), *idest* — **Load from Broadcast Register**

idest ← 0xffffffff

The four-bit index *i* in **cbr_*i***, formed by A6–A3, ranges between 0 and 15 (**cbr_0, cbr_1, ... , cbr_15**).

Loads from any of these 16 DCCU registers place all ones into the specified register *idest*. Thus, a **cawait** operation is always satisfied, because it makes the processor wait until the number of the current iteration is less than or equal to **cbr_*i***.

st.l *isrc1ni*, **l%cbr_*i*** (*isrc2*) — **Store to Broadcast Register**

```
IF     (i = 0)
THEN NEWCURR ← 0
       InLoop ← 1
FI
```

A store to register **l%cbr_0** clears the 32 bit counter and sets the InLoop bit. This action begins a loop.

Stores to the other fifteen broadcast register locations of the DCCU have no effect. Such a store causes a **cadvance** operation to become a no-op.

ld.l l%cget (*isrc2*), *idest* — **Load New Iteration Count**

NEWCURR ← NEWCURR + 1
idest ← NEWCURR

A load from **cget** causes the counter to increment and the results to be placed in the specified register. This is used to start the next iteration of a loop.

ld.l l%cnewcurr (*isrc2*), *idest* — **Load from Iteration Counter**

idest ← NEWCURR

A load from **cnewcurr** loads the *idest* with the contents of the counter.

st.l *isrc1ni*, **l%cnewcurr** (*isrc2*) — **Store to Iteration Counter**

NEWCURR ← *isrc1ni*

A store to **cnewcurr** causes the contents of the register *isrc1* to be loaded into the 32 bit counter.

ld.l l%cstat (*isrc2*), *idest* — **Load Status**

idest ← STAT

A load from **cstat** causes the contents of the STAT register to be loaded into the specified register *idest*. Refer to Chapter 3 for the register format.

st.l *isrc1ni*, **l%cstat**(*isrc2*)
st.l *isrc1ni*, **l%cstatci**(*isrc2*)
st.l *isrc1ni*, **l%cstatn**(*isrc2*) — **Store Status**

InLoop ← bit 0 of *isrc1ni*
Nested ← bit 1 of *isrc1ni*

A store to **cstat, cstatn,** or **cstatci** puts bits 0 and 1 of the register designated by *isrc1* into the two state bits, InLoop and Nested. Stores to **cstat, cstatn,** and **cstatci** all have the same effect.

ld.l l%cstatci (*isrc2*), *idest* — **Load Status Clearing Inloop**

```
idest ← STAT
IF      (Nested = 0)
THEN  InLoop ← 0
FI
```

This is similar to the previous **ld.l l%cstat** except that the InLoop bit is cleared if Nested is cleared. This is done when all iterations of a concurrent loop are completed.

ld.l l%cstatn (*isrc2*), *idest* — **Load Status Setting Nested**

idest ← STAT
Nested ← InLoop

This is similar to the load from **cstat** except that the contents of InLoop is loaded into Nested. This is used before starting a new loop, to cause the CCU to switch to nested mode if the processor is already in a concurrent loop.

st.l *isrc1ni*, **l%cclm** (*isrc2*)	**Clear Broadcast Registers**
(no effect)	

A store to **l%cclm** has no effect on the DCCU. This operation is normally used before entering a loop to clear those broadcast registers that are used as synchronization counters. With the DCCU, no response is necessary. (Refer to the load and store operations on broadcast registers.)

ld.l l%cver (*isrc2*), *idest*	**Load Version**
idest ← 0	

The version and stepping numbers are returned as zero.

A store to **l%cget** and a load from **l%cclm** have undefined results.

Programming Notes

All DCCU accesses can cause the same data access traps as other **ld.x** and **st.x** operations (protection, breakpoint on read or write, page not present, misalignment).

The effect of addressing the DCCU with any of the instructions **pfld, fld,** or **fst** is undefined. Addressing the DCCU with **ldio, stio,** or **pst** has no effect.

Core Instructions 7

CHAPTER 7
CORE INSTRUCTIONS

Core instructions include loads and stores of the integer, floating-point, and control registers; arithmetic and logical operations on the 32-bit integer registers; control transfers; I/O; and system control functions. All these instructions are executed by the core unit.

The comments regarding optimum performance that appear in the subsections **Programming Notes** are recommendations only. If these recommendations are not followed, the processor automatically waits the necessary number of clocks to satisfy internal hardware requirements.

7.1 LOAD INTEGER

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

.x = **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)

The load integer instruction transfers an 8-, 16-, or 32-bit value from memory to the integer registers. The *isrc1* can be either a 16-bit immediate address offset or an index register. Loads of 8- or 16-bit values from memory place them in the low-order bits of the destination registers and sign-extend them to 32-bit values in the destination registers.

Traps

If the operand is misaligned, a data-access trap results.

Programming Notes

For best performance, observe the following guidelines:

1. The destination of a load should not be referenced as a source operand by the next instruction.

2. A load instruction should not directly follow a store that is expected to hit in the data cache.

Even though immediate address offsets are limited to 16 bits, loads using a 32-bit address offset may be implemented by the following sequence (**r31** is recommended for all such addressing calculations):

```
orh     ha%ADDRESS, r0, r31
ld.l    l%ADDRESS(r31), idest
```

The **l%** operator takes the low-order 16 bits of the address. Note that the processor uses signed addition when it adds **l%ADDRESS** to **r31**. If bit 15 is set, this has the effect of subtracting from **r31**. Therefore, when bit 15 of **l%ADDRESS** is set, the high-order part of the address must be derived by adding one to the high-order 16 bits, so that the net result is correct. This is precisely what the **ha%** operator does.

The assembler must align the immediate address offsets used in loads to the same boundary as the effective address, because the lower bits of the immediate offset are used to encode operand length information.

7.2 STORE INTEGER

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

.x = .b (8 bits), **.s** (16 bits), or **.l** (32 bits)

The store instruction transfers an 8-, 16-, or 32-bit value from the integer registers to memory. Stores do not allow an index register in the effective-address calculation, because *isrc1ni* is used to specify the register to be stored. The *#const* is a signed, 16-bit, immediate address offset. An absolute address may be formed by using **r0** for *isrc2*. Stores of 8- or 16-bit values store the low-order 8 or 16 bits of the register.

Traps

If the operand is misaligned, a data-access trap results.

Programming Notes

For best performance, a load instruction should not directly follow a store that is expected to hit in the data cache.

Even though immediate address offsets are limited to 16 bits, a store using a 32-bit immediate address offset may be implemented by the following sequence (**r31** is recommended for all such addressing calculations):

```
orh     ha%ADDRESS, r0, r31
st.l    isrc1ni, l%ADDRESS(r31)
```

The **l%** operator takes the low-order 16 bits of the address. Note that the processsor uses signed addition when it adds **l%ADDRESS** to **r31**. If bit 15 is set, this has the effect of subtracting from **r31**. Therefore, when bit 15 of **l%ADDRESS** is set, the high-order part of the address must be derived by adding one to the high-order 16 bits, so that the net result is correct. This is precisely what the **ha%** operator does.

The assembler must align the immediate address offsets used in stores to the same boundary as the effective address, because the lower bits of the immediate offset are used to encode operand length information.

7.3 TRANSFER INTEGER TO F-P REGISTER

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

The **ixfr** instruction transfers a bit pattern from the 32-bit integer register *isrc1ni* to the 32-bit floating-point register *fdest*. Assemblers and compilers should encode *fsrc2* as **f0**.

Programming Notes

For best performance, the destination of an **ixfr** should not be referenced as a source operand in the next two instructions.

7.4 LOAD FLOATING-POINT

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

Pipelined Floating-Point Load

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

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

.y = **.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)
pfld.q is not available with the i860 XR CPU

Floating-point loads transfer 32-, 64-, or 128-bit values from memory to the floating-point registers. These may be floating-point values or integers. An autoincrement option supports constant-stride vector addressing. If this option is specified, the processor stores the effective address into *isrc2*.

Floating-point loads may be either pipelined or not. Data that is expected to be used several times before being replaced in the cache should be loaded with the nonpipelined **fld** instruction. The **fld** instruction checks the data cache. If the required data is in the cache, no bus cycle is issued. On a cache miss, **fld** accesses the data via the bus, and places the data in the cache. The **fld** instruction does not advance the load pipeline and does not interact with outstanding **pfld** instructions.

The load pipeline has three stages. A **pfld** returns the data from the address calculated by the third previous **pfld**, thereby allowing three loads to be outstanding on the external bus. The **pfld** instruction, is optimized for use with uncached data; it does not place the data in the data cache after a cache miss. When the data is already in the cache, **pfld** reads the data from the cache only if it has been modified. A **pfld** should be used when the data is expected to be used only once in the near future.

Traps

If the operand is misaligned, a data-access trap results. No trap occurs when the data loaded is not a valid floating-point number. Executing **pfld.q** on the i860 XR microprocessor causes an instruction trap.

Programming Notes

In the i860 XR microprocessor, a **pfld** cannot load a 128-bit operand.

For the autoincrementing form of the instruction, the register coded as *isrc1* must not be the same register as *isrc2*.

For best performance, observe the following guidelines:

1. The destination of an **fld** or **pfld** should not be referenced as a source operand in the next two instructions.

2. An **fld** instruction should not directly follow a store instruction that is expected to hit in the data cache. There is no performance impact for a **pfld** following a store instruction.

3. A string of more than three successive **pfld** instructions causes internal delays due the fact that the bandwith of the processor bus is one **pfld** per two cycles.

The assembler must align the immediate address offsets used in loads to the same boundary as the effective address, because the lower bits of the immediate offset are used to encode operand length information.

To take greatest advantage of the NENE# pin and the faster access to paged DRAM that it provides, programmers should organize **pfld** instructions and their operands so that successive memory accesses have a high probability of hitting the same DRAM page.

7.5 STORE FLOATING-POINT

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

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

.y = **.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)

Floating-point stores transfer 32-, 64-, or 128-bit values from the floating-point registers to memory. These may be floating-point values or integers. Floating-point stores allow *isrc1* to be used as an index register. An autoincrement option supports constant-stride vector addressing. If this option is specified, the processor stores the effective address into *isrc2*.

Traps

If the operand is misaligned, a data-access trap results.

Programming Notes

For the autoincrementing form of the instruction, the register coded as *isrc1* must not be the same register as *isrc2*.

For best performance, observe the following guidelines:

1. An **ld** or **fld** instruction should not directly follow a store instruction that is expected to hit in the data cache. There is no performance impact for a **pfld** following a store instruction.

2. The *fdest* of an **fst.y** instruction should not reference the destination of the next instruction if that instruction is a pipelined floating-point operation.

The assembler must align the immediate address offsets used in stores to the same boundary as the effective address, because the lower bits of the immediate offset are used to encode operand length information.

7.6 PIXEL STORE

pst.d *fdest, #const(isrc2)* — **Pixel Store (Normal)**
pst.d *fdest, #const(isrc2)* + + — **(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
```

The pixel store instruction selectively updates the pixels in a 64-bit memory location. The pixel size is determined by the PS field in the **psr**. The pixels to be updated are selected by the low-order bits of the PM field in the **psr**. 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.

This instruction is typically used in conjunction with the **fzchks** or **fzchkl** instructions to implement Z-buffer hidden-surface elimination. When used this way, a pixel is updated only when it represents a point that is closer to the viewer than the closest point painted so far at that particular pixel location. Refer to Chapter 8 for more about **fzchks** and **fzchkl**.

Traps

If the operand is misaligned, a data-access trap results.

Programming Notes

For the autoincrementing form of the instruction, the register coded as *isrc1* must not be the same register as *isrc2*.

For best performance, observe the following guidelines:

1. An **ld** or **fld** instruction should not directly follow a store instruction that is expected to hit in the data cache. There is no performance impact for a **pfld** following a store instruction.

2. The *fdest* of a **pst.y** instruction should not reference the destination of the next instruction if that instruction is a pipelined floating-point operation.

Even if all bits of PM are zero, bus cycles are still issued; however, no byte-enable signals are asserted.

7.7 INTEGER ADD AND SUBTRACT

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

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

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

idest ← *isrc1* + *isrc2*
OF ← (bit 31 carry ≠ bit 30 carry)
Using signed comparison,
  CC set if *isrc2* + *isrc1* < 0
  CC clear if *isrc2* + *isrc1* ≥ 0

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

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

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

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

In addition to their normal arithmetic functions, the add and subtract instructions are also used to implement comparisons. For this use, **r0** is specified as the destination, so that the result is effectively discarded. Equal and not-equal comparisons are implemented with the **xor** instruction (refer to the section on logical instructions).

Add and subtract ordinal (unsigned) can be used to implement multiple-precision arithmetic.

Flags Affected

CC and OF as defined above.

Programming Notes

For optimum performance, a conditional branch should not directly follow an add or subtract instruction.

Refer to Chapter 12 for an example of how to handle the sign of 8- and 16-bit integers when manipulating them with 32-bit instructions.

An instruction of the form **subs –1**, *isrc2*, *idest* yields the one's complement of *isrc2*.

When *isrc1* is immediate, the immediate value is sign-extended to 32-bits even for the unsigned instructions **addu** and **subu**.

These instructions enable convenient encoding of a literal operand in a subtraction, regardless of whether the literal is the subtrahend or the minuend. For example:

	Calculation	Encoding
Signed	**r6 = 2 − r5** **r6 = r5 − 2**	**subs 2, r5, r6** **adds −2, r5, r6**
Unsigned	**r6 = 2 − r5** **r6 = r5 − 2**	**subu 2, r5, r6** **addu −2, r5, r6**

Note that the only difference between the signed and the unsigned forms is in the setting of the condition code CC and the overflow flag OF.

The various forms of comparison between variables and constants can be encoded as follows:

Condition	Encoding	Branch When True	
		Signed	Unsigned
var ≤ const	**subs** const, var **subu** const, var	**bnc**	**bc**
var < const	**adds** −const, var **addu** −const, var*	**bc**	**bnc**
var ≥ const	**adds** −const, var **addu** −const, var*	**bnc**	**bc**
var > const	**subs** const, var **subu** const, var	**bc**	**bnc**

NOTE: *Valid only when const > 0

The arithmetic instructions are recommended for moving a small integer constant to an integer register, because they move and sign-extend in one instruction. They do, however, affect the condition code. The following assembler pseudo-operation utilizes the **adds** instruction. The *const32* represents a signed constant expression in assembly language whose value is in the range: 0xFFFF8000 ≤ *const32* < 0x8000.

mov *const32, idest* — **Small Constant-to-Register Move**

Assembler pseudo-operation, equivalent to:
adds *l%const32*, **r0**, *idest*

7.8 SHIFT INSTRUCTIONS

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 *isrc1ni, isrc2, idest* — **Shift Right Double**

idest ← low-order 32 bits of *isrc1ni*:*isrc2* shifted right by SC bits

The arithmetic shift does not change the sign bit; rather, it propagates the sign bit to the right *isrc1* bits.

Shift counts are taken modulo 32. A **shrd** right-shifts a 64-bit value with *isrc1* being the high-order 32 bits and *isrc2* the low-order 32 bits. The shift count for **shrd** is taken from the shift count of the last **shr** instruction, which is saved in the SC field of the **psr**. Shift-left is identical for integers and ordinals.

Programming Notes

The shift instructions are recommended for the integer register-to-register move and for no-operations, because they do not affect the condition code. The **shrd** instruction is used as the floating-point no-op, because not only does it not affect the condition code, but it does not change the floating-point pipeline, either. The processor interprets the D-bit of **shrd** as if it were a floating-point instruction. The following assembler pseudo-operations utilize the shift instructions:

mov *isrc2, idest* — **Register-to-Register Move**

Assembler pseudo-operation, equivalent to:
shl r0, *isrc2, idest*

nop — **Core No-Operation**

Assembler pseudo-operation, equivalent to:
shl r0, r0, r0

fnop — **Floating-Point No-Operation**

Assembler pseudo-operation, equivalent to:
shrd r0, r0, r0

Rotate is implemented by:

```
shr    COUNT, r0, r0    // Only loads COUNT into SC of PSR
shrd   op, op, op       // Uses SC for shift count
```

7.9 SOFTWARE TRAPS

trap *isrc1ni, isrc2, idest* — **Software Trap**

Generate trap with IT set in **psr**

intovr — **Software Trap on Integer Overflow**

IF OF in **epsr** = 1
THEN generate trap with IT set in **psr**
FI

These instructions generate the instruction trap, as described in Chapters 9 and 10.

The **trap** instruction can be used to implement supervisor calls and code breakpoints. The *idest* should be zero, because its contents are undefined after the operation. The *isrc1ni* and *isrc2* fields can be used to encode the type of trap.

The **intovr** instruction generates an instruction trap if the OF bit (overflow flag) of **epsr** is set. It is used to test for integer overflow after the instructions **adds**, **addu**, **subs**, and **subu**. Assemblers and compilers should encode *isrc1, isrc2,* and *idest* as zero.

Programming Notes

The **trap** and **intovr** instructions must not be included in a locked sequence.

7.10 LOGICAL INSTRUCTIONS

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

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

xor *isrc1, isrc2, idest* — **Logical XOR**

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

xorh *#const, isrc2, idest* — **Logical XOR High**

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

The operation is performed bitwise on all 32 bits of *isrc1* and *isrc2*. When *isrc1* is an immediate constant, it is zero-extended to 32 bits.

The "h" variant signifies "high" and forms one operand by using the immediate constant as the high-order 16 bits and zeros as the low-order 16 bits. The resulting 32-bit value is then used to operate on the *isrc2* operand.

Flags Affected

CC is set if the result is zero, cleared otherwise.

Programming Notes

Bit operations can be implemented with logical operations by setting up *isrc1* as an immediate constant that contains a one in the bit position to be operated on and zeros elsewhere.

Bit Operation	Equivalent Logical Operation
Set bit	**or**
Clear bit	**andnot**
Complement bit	**xor**
Test bit	**and** (CC set if bit is clear)

The logical instructions are recommended for moving a 32-bit integer constant to an integer register. Note, however, that they *do* affect the condition code. The following assembler pseudo-operation utilizes the **or** and **orh** instruction. The *const32* represents a signed constant expression in assembly language whose value is in one of these ranges: *const32* < 0xFFFF8000, *const32* ≥ 0x8000.

mov *const32, idest* — **Large Constant-to-Register Move**

Assembler pseudo-operation, equivalent to:

orh *h%const32*, **r0**, *idest*
or *l%const32, idest, idest*

7.11 CONTROL-TRANSFER INSTRUCTIONS

Control transfers can branch to any location within the address space. However, if a relative branch offset, when added to the address of the control-transfer instruction plus four, produces an address that is beyond the 32-bit addressing range of the processor, the results are **undefined**.

Many of the control-transfer instructions are *delayed* transfers. They are delayed in the sense that the processor executes one additional instruction following the control-transfer instruction before actually transferring control. During the time used to execute the additional instruction, the processor refills the instruction pipeline by fetching instructions from the new instruction address. This avoids breaks in the instruction execution pipeline. It is generally possible to find an appropriate instruction to execute after the delayed control-transfer instruction even if it is merely the first instruction of the procedure to which control is passed.

Programming Notes

The sequential instruction following a delayed control-transfer instruction must not be another control-transfer instruction, nor a **trap** instruction, nor the target of a control-transfer instruction.

br *lbroff* **Branch Direct Unconditionally**

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

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
```

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
```

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
```

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

```
LCC_temp clear if isrc2 + isrc1ni < 0 (signed)
LCC_temp set if isrc2 + isrc1ni ≥ 0 (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
```

The instructions **bc.t** and **bnc.t** are delayed forms of **bc** and **bnc**. The delayed branch instructions **bc.t** and **bnc.t** should be used when the branch is taken more frequently than not; for example, at the end of a loop. The nondelayed branch instructions **bc**, **bnc**, **bte**,

and **btne** should be used when branch is taken less frequently than not; for example, in certain search routines.

If a trap occurs on a **bla** instruction or the next instruction, LCC is not updated. The trap handler resumes execution with the **bla** instruction, so the LCC setting is not lost. The i860 XP microprocessor sets the AI bit of **epsr** when a trap occurs on **bla**.

Programming Notes

The **bla** instruction is useful for implementing loop counters, where *isrc2* is the loop counter and *isrc1* is set to −1. In such a loop implementation, a **bla** instruction may be performed before the loop is entered to initialize the LCC bit of the **psr**. The target of this **bla** should be the sequential instruction after the next, so that the target instruction is executed regardless of the setting of LCC. Another **bla** instruction placed as the next to last instruction of the loop tests for loop completion and update the loop counter. The total number of iterations is the value of *isrc2* before the first **bla** instruction, plus one. Example 7-1 illustrates this use of **bla**.

Programmers should avoid calling subroutines from within a **bla** loop, because a subroutine may also use **bla** and change the value of LCC.

For the **bla** instruction, the register coded as *isrc1* must not be the same register as *isrc2*.


```
// EXAMPLE OF bla USAGE

// Write zeros to an array of 64 single-precision numbers
// Starting address of array is already in r4

    fmov.dd             f0,     f2  // f3, f2 <-- 0
    adds   -1           r0,     r5  // r5 <-- loop increment
    mov                 15,     r6  // r6 <-- loop count -1
    bla    r5,   r6,    CLEAR_LOOP  // One time to initialize LCC
    addu   -16,         r4,     r4  // Start one group lower to
                                    //  allow for autoincrement
CLEAR_LOOP:
    bla    r5,   r6,    CLEAR_LOOP  // Loop for the 16 times
    fst.q  f0,          16(r4)++    // Write and autoincrement
                                    //  to next group
```

Example 7-1. Example of bla Usage

call *lbroff* **Subroutine Call**

r1 ← address of next sequential instruction + 4 (or + 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 (or + 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.)

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 instruction pair
                     ELSE    enter single-instruction mode
                               for next instruction pair
                     FI
                FI
        FI
FI
```

Continue execution at address in *isrc1ni* (The original contents of *isrc1ni* are used even if the next instruction modifies *isrc1ni*. Does not trap if *isrc1ni* is misaligned.)

Return from a subroutine is implemented by branching to the return address with the indirect branch instruction **bri**.

Indirect branches are also used to resume execution from a trap handler (refer to Chapters 9 and 10). The need for this type of branch is indicated by set trap bits in the **psr** at the time **bri** is executed. In this case, the instruction following the **bri** must be a load or move that restores *isrc1ni* to the value it had before the trap occurred.

Programming Notes

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

Due to contention for the **psr** register, the following sequence is illegal if any trap bits are set when the **bri** instruction is executed:

```
bri any_address
st.c src1, psr
```

The register *isrc1ni* of the **calli** instruction must not be **r1**.

7.12 CONTROL REGISTER ACCESS

ld.c *csrc2, idest* *idest* ← *csrc2*	**Load from Control Register**
st.c *isrc1ni, csrc2* *csrc2* ← *isrc1ni*	**Store to Control Register**

Csrc2 specifies a control register that is transferred to or from a general-purpose register. The function of each control register is defined in Chapter 3. As shown below, some registers or parts of registers are write-protected when the U-bit in the **psr** is set. A store to those registers or bits is ignored when the processor is in user mode. The encoding of *isrc2* is defined by Table 7-1.

Programming Notes

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.

After a scalar floating-point operation, an **st.c** to **fsr** should not change the value of RR, RM, or FZ until the point at which result exceptions are reported. (Refer to Chapters 9 and 10 for more details.)

Only a trap handler should use the instruction **st.c** to set the trap bits (IT, IN, IAT, DAT, FT) of the **psr**.

Table 7-1. Control Register Encoding for Assemblers

Register		*Src2* Code	User-Mode Write-Protected?
fir	(Fault Instruction)	0	N/A[1]
psr	(Processor Status)	1	Yes[2]
dirbase	(Directory Base)	2	Yes
db	(Data Breakpoint)	3	Yes
fsr	(Floating-Point Status)	4	No
epsr	(Extended Processor Status)	5	Yes[3]
bear	(Bus Error Address Register)[4]	6	N/A[1]
ccr	(Concurrency Control Register)[4]	7	Yes
p0	(Privileged Register 0)[4]	8	Yes
p1	(Privileged Register 1)[4]	9	Yes
p2	(Privileged Register 2)[4]	10	Yes
p3	(Privileged Register 3)[4]	11	Yes

NOTES:
1. The **fir** and **bear** registers cannot be written by the **st.c** instruction.
2. Only the **psr** bits BR, BW, PIM, IM, PU, U, IT, IN, IAT, DAT, FT, DS, DIM, and KNF are write-protected.
3. The processor type, stepping number, and DCS cannot be changed from either user or supervisor level.
4. Available only with i860™ XP CPU. Using these encodings with the i860 XR CPU produces undefined results.

7.13 CACHE FLUSH

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

Write back modified data from the data-cache line residing at the cache location addressed by (*#const* + *isrc2*).
For 80860XR, the contents of the line are undefined, and the tag is set to (*#const* + *isrc2*); for 80860XP, the virtual tag and physical tag of the line are invalidated.
IF autoincrement
THEN *isrc2* ← *#const* + *isrc2*
FI

The **flush** instruction is used to force modified data from the data cache to external memory. The address *#const* + *isrc2* must be aligned on a 16-byte boundary. Any of the 32-byte lines in a cache set can be addressed *#const* + *isrc2*. The particular line (or "way") that is forced to memory is controlled by the RB field of **dirbase**. In user mode, execution of **flush** is suppressed; use it only in supervisor mode. Because the register designated by *idest* is undefined after **flush**, assemblers should encode *idest* as zero.

Example 7-2 shows how to use the **flush** instruction.

– XR –

To invalidate the virtual tags, the addresses used by the **flush** instruction refer to a reserved four-Kbyte memory area that is not used to store data. These addresses must be valid and writable in both the old and the new task's space.

– XP –

The processor invalidates virtual tags by clearing their V-bit. The physical tag state is set to I (invalid). For compatibility with the i860 XR CPU, it is recommended that the addresses used by **flush** be chosen from the same reserved four-Kbyte memory area.

Making one pass for each way ensures that all cache lines containing modified data are written back to memory. Each pass references every 32nd byte of the reserved area with the **flush** instruction. Before the first pass, the RC field in **dirbase** is set to two and RB is set to zero. This causes data-cache misses to flush line zero of each set. Before the each subsequent pass, RB is increased by one, causing the next line of each set to be flushed.

```
// CACHE FLUSH ROUTINE -- Common to 80860XR and 80860XP
// Rr, Rs, Rt, Ru, Rv, Rw, Rx, Ry represent integer registers
// Constant definitions for control registers masks:
   DIRB_rb = 0xc00;  DIRB_rc = 0x300

   .data
flush_area::          // Using method of your choice, reserve
   .byte [4096] 0     //  4 Kbytes of writable memory

   .text
flush::
   mov      r1,      Rr              // Save return address
   ld.c     dirbase,Rv               // Save dirbase
   andnot   0x0F00, Rv, Ru           // Clear RC, RB fields
   adds     -1,      r0,     Rx      // Loop decrement
   mov      flush_area-32,  Rt       // Starting flush address-32

// Get ready for first flush pass
    or      0x800,  Ru,     Rs       // Set RC = 2, RB = 0
    call    D_FLUSH
    st.c    Rs,     dirbase          // dirbase <-- RC = 2, RB = 0
// Get ready for second flush pass
    or      0x900,  Ru,     Rs       // Set RC = 2, RB = 1
    call    D_FLUSH
    st.c    Rs,     dirbase          // dirbase <-- RC = 2, RB = 1

// Check data cache size.     // EPSR (21:18) is DCS (3..0)
    ld.c    epsr,   Rs        // 0001=8K, 0010=16K
    shr     19,     Rs,   Rs // Rs <- DCS (3..1). Bit 0 discarded
    and     0x07,   Rs,   Rs // Isolate DCS (3..1)
    bc      RESTORE_DIRBASE  // Taken if data cache is 8K or less

// This data cache is > 8K. Flush more cache ways.

// Get ready for third flush pass
    or      0xa00,  Ru,   Rs         // Set RC = 2, RB = 2
    call    D_FLUSH
    st.c    Rs,     dirbase          // dirbase <-- RC = 2, RB = 2
// Get ready for fourth flush pass
    or      0xb00,  Ru,   Rs         // Set RC = 2 and RB = 3
    call    D_FLUSH
    st.c    Rs,     dirbase          // dirbase <-- RC = 2, RB = 3

RESTORE_DIRBASE:
// Change DTB, ATE or ITI fields in Rv here, if necessary...
    st.c    Rv,     dirbase          // Restore original dirbase
    nop;nop;nop;nop;nop;nop          // Required to drain instruction Q
// Return from flush
    bri     Rr
    nop

D_FLUSH:
    mov     127,    Ry               // Reset loop counter
    bla     Rx, Ry, D_FLUSH_LOOP     // One time to initialize LCC
    mov     Rt,     Rw               // Original flush address
D_FLUSH_LOOP:
    bla     Rx, Ry, D_FLUSH_LOOP     // Loop on next instruction 128 times
    flush   32(Rw)++                 // Flush and autoincrement
    bri     r1                       // Return after next instruction
    nop
```

Example 7-2. Cache Flush Procedure

– XR –

Note that the processor may direct actual write cycles to the page reserved for **flush** addresses. Each line of the data cache has two M (modified) bits, one for each half line. Although both half-lines are written back if both are modified, the **flush** instruction clears only the M-bit of the half line that corresponds to the target address. (If A4 = 0, the M-bit for the lower 16 bytes is cleared; otherwise, the M-bit for the higher 16 bytes is cleared.) The procedure in Example 7-2 clears only the lower M-bit. Because the other M-bit remains set, the processor may later perform 16-byte write-backs to the reserved page.

– XP –

Unlike the i860 XR microprocessor, the **flush** instruction of the i860 XP microprocessor invalidates the entire line by writing back modified data and invalidating both its virtual and its physical tag.

7.14 BUS LOCK

lock — **Begin Interlocked Sequence**

Causes the next data load or store that appears on the bus to assert the LOCK# signal, directing the external system to lock that location by preventing locked reads, locked writes, and unlocked writes to it from other processors. External interrupts are disabled from the first instruction after the **lock** until the location is unlocked.

unlock — **End Interlocked Sequence**

The next load or store (regardless of whether it hits in the cache) deasserts the LOCK# signal, directing the external system to unlock the location. Interrupts are enabled.

These instructions allow programs running in either user or supervisor mode to perform atomic read-modify-write sequences in multiprocessor and multithread systems. The lock protocol requires the following sequence of activities:

1. **lock**

2. Any load or store instruction that appears on the bus. For compatibility with future processor generations, this should be a load. With the 80860XR, software in a multiprocessing system should ensure that the first load or store instruction after a **lock** references noncacheable memory or a memory location not yet cached. If the load or store instruction hits the cache, the sequence is legal, but the bus of the 80860XR is not locked. For the 80860XP, the first load or store does not have to miss the cache; it (as well as all subsequent loads and stores in the locked sequence) are forced to the bus.

3. **unlock**

4. Any load or store instruction (regardless of whether it misses the cache). For compatibility with future processor generations, this should be a store.

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, an instruction trap occurs. The interlocked sequence must not branch outside of the 30 sequential instructions following the **lock** instruction. If the processor encounters another **lock** instruction before unlocking the bus or an **unlock** with no preceding **lock**, that instruction is ignored.

If a trap occurs after step 1 but before or during step 4, the processor sets the IL (interlock) bit of **epsr**. This is likely to happen, for example, during TLB miss processing, when the A-bit of the page table entry is not set. (Refer to Chapter 4.)

The sequence must be restartable from the **lock** instruction in case a trap occurs. Simple read-modify-write sequences are automatically restartable. For sequences with more than one store, the software must ensure that no traps occur after the first nonreexecutable store. To ensure that no instruction-access fault occurs, the instructions

that are not restartable should not span a page boundary. To ensure that no unrecoverable data access fault occurs, locations to be modified should first be read then written with unmodified data, so that any data access traps are triggered before modified data is written.

When multiple memory locations are accessed during a locked sequence, the system only needs to guarantee that the first location is locked against locked reads, locked writes, and unlocked writes by other processors. High-performance multiprocessor hardware systems can implement a fine-grained lock, during which other processors can use the bus for access to other memory locations. Simpler systems may implement more inclusive locking, even preventing other processors from using the system bus, but software should not count on such an inclusive lock. For each shared data structure, software must establish a single location that is the first location referenced by any locked sequence that requires that data. For example, the head of a doubly linked list should be referenced before accessing items in the middle of the list.

Between locked sequences, at least one cycle of LOCK# pin deactivation is guaranteed by the behavior of **unlock**.

If the load or store instruction of step 2 accesses a previously unaccessed page (A=0), the bus is locked briefly while the A bit is set, unlocked, then locked again to satisfy the **lock** instruction and start the locked sequence.

Example 7-3 shows how **lock** and **unlock** can be used in a variety of interlocked operations.


```
// LOCKED TEST AND SET
//  Value to put in semaphore is in r23

    lock                            //
    ld.b    semaphore,       r22    // Put current value of semaphore in r22
    unlock                          //
    st.b    r23, semaphore          //
// The prior value of the semaphore in r22 can now be tested.

// LOCKED LOAD-ALU-STORE

    lock                            //
    ld.l    word,             r22   //
    addu    1,          r22,  r22   // Can be any ALU operation
    unlock
    st.l    r22,              word  //

// LOCKED COMPARE AND SWAP
//  Swaps r23 with word in memory, if word = r21

    lock                            //
    ld.l    word,             r22   //
    bte     r22,        r21,  L1    //
    mov     r22,              r23   // Executed only if not equal
L1: unlock                          //
    st.l    r23,              word  //
```

Example 7-3. Examples of lock and unlock Usage

Programming Notes

During the lock protocol, a transition to or from dual-instruction mode is not permitted.

The **trap** and **intovr** instructions must not be included within the lock protocol.

7.15 INPUT AND OUTPUT (80860XP ONLY)

ldio.x *isrc2, idest*	**Load I/O**
idest ← *port.x (isrc2)*	
stio.x *isrc1ni, isrc2*	**Store I/O**
port.x (isrc2) ← *isrc1ni*	

.x = **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)
Not available with the i860 XR CPU

The **ldio** instruction transfers an 8-, 16- or 32-bit value from the I/O space to the integer registers. Eight- and 16-bit loads are sign-extended to 32 bits. *isrc1* must be zero.

The **stio** instruction transfers an 8-, 16- or 32-bit value from an integer register to the I/O space.

No address translation is done for **ldio** and **stio**, and the virtual address is driven directly to the external address bus. Data at the same address in the data cache is not accessed by **ldio** or **stio**.

The **ldio** and **stio** instructions are suppressed in user mode.

Traps

The address in *isrc2* must be aligned to match the operand size; otherwise, the i860 XP microprocessor generates a misalignment data access trap. Address translation exceptions do not occur, because the address is not translated. Breakpoint traps occur as in **ld.x** and **st.x** instructions.

Using **ldio** or **stio** on the i860 XR microprocessor causes an instruction fault.

7.16 LOAD INTERRUPT (80860XP ONLY)

ldint.x *isrc2, idest* **Load Interrupt Vector**

Generate a bus cycle with M/IO# = 0, W/R# = 0, and D/C# = 0.
idest ← *int_vector.x (isrc2)*

.x = **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)
Not available with the i860 XR CPU

The **ldint** instruction performs an 8-, 16-, or 32-bit interrupt acknowledge cycle using the address in *isrc2*. The interrupt vector (a datum from the external bus) is returned to *idest*. Eight- and 16-bit values are sign-extended to 32 bits. *Src1* should be zero. The address in *isrc2* is not translated, the data cache is not searched, and the bus cycle is not burstable. In user mode, the **ldint** instruction is treated as a no-op.

The **ldint** instruction can be used to emulate the interrupt acknowledge sequence of the Intel486 and Intel386 microprocessors, using the instruction sequence shown in Example 7-4. The Intel486 and Intel386 microprocessors generate two INTA cycles as a response to an external interrupt and insert four idle clocks in between. In order to generate a compatible interrupt acknowledge sequence, i860 XP microprocessor code must prevent an instruction-cache miss between the two INTA cycles. This is done by aligning the first **ldint** instruction on a 32-byte boundary.

For compatibility with INTA of the Intel486 CPU, the *isrc2* of the first **ldint** instruction should be a register containing the value 8, and the *isrc2* of the second **ldint** should be **r0**. The use of **lock** also matches the Intel486 CPU.

The **ldint** instruction can be executed only in supervisor mode; it is suppressed in user mode.

Traps

The addresses in *src2* must be aligned to match the operand size; otherwise, the i860 XP microprocessor generates a misalignment data access trap. Address translation exceptions do not occur, because the address is not translated. Data access traps for breakpoints can occur.

Using **ldint** on the i860 XR microprocessor causes an instruction fault.

```
// The following lock instruction must be on a 32-byte boundary:
   lock                          // Lock the bus
   ldint.b src2, rdest           // First INTA cycle. src2 contains 8.
   or     rdest, r0, rdest       // Wait for completion
   unlock                        // Unlock the bus after the next ldint
   nop                           // Insert 2 + <number of NOPs> idle
   nop                           //   clocks for 8259A recovery.
   ldint.b r0,       rdest       // Second INTA cycle
```

Example 7-4. Interrupt Acknowledge Sequence

7.17 SPECIAL CYCLES (80860XP ONLY)

scyc.b *isrc2*	**Special Cycles**
Generate a special bus cycle (D/C# = 0, W/R# = 1, M/IO# = 0) and set the address bus to the value contained in the register *isrc2*	

Not available with the i860 XR CPU

The **scyc** instruction generates special bus cycles that signal internal events to external devices over the bus. The value of *isrc2* is placed on the address bus and specifies the type of special cycle, as Table 7-2 defines. *isrc1* must be encoded as zero. The special cycles are compatible with the corresponding cycles of the Intel486 microprocessor.

The external cache invalidate cycle (INVD instruction of the Intel486 CPU) tells an external cache controller to invalidate its cache, without writing back to memory any modified lines. The external cache write-back cycle (WBINVD instruction of the Intel486 CPU) tells an external cache controller to write back all dirty lines and then invalidate the cache.

The behavior of external caches depends on the design of the external bus decoder; for example, the decoder may choose to make the INVD cycle flush modified data from the external cache to memory, and not invalidate the cache (a "synchronize" operation).

The **scyc** instruction can be executed only in supervisor mode; it is suppressed in user mode.

Traps

Data alignment traps can occur on **scyc**; so, if **scyc.s** or **scyc.l** are used, the contents of *isrc2* must be aligned to the operand length. Data breakpoint traps can occur on the *isrc2* value. The operands of **scyc** do not undergo address translation; therefore, address translation exceptions do not occur.

Using **scyc** on the i860 XR microprocessor causes an instruction fault.

Table 7-2. Encoding of Special Bus Cycles

isrc2	Special Bus Cycle	Corresponding Intel486™ Microprocessor Condition
0	Shutdown	Shutdown
1	Ext. Cache Invalidate	INV Instruction
2	Halt	HLT Instruction
3	Ext. Cache Write Back	WBINV Instruction

NOTE: All other encodings are *reserved.*

7.18 ASSEMBLER PSEUDO-OPERATIONS

mov *const32, idest* — **Small Constant-to-Register Move**

... where 0xFFFF8000 ≤ *const32* < 0x8000
Assembler pseudo-operation, equivalent to:
adds *l%const32*, **r0**, *idest*

mov *const32, idest* — **Large Constant-to-Register Move**

... where *const32* < 0xFFFF8000 or *const32* ≥ 0x8000
Assembler pseudo-operation, equivalent to:
orh *h%const32*, **r0**, *idest*
or *l%const32, idest, idest*

mov *isrc2, idest* — **Register-to-Register Move**

Assembler pseudo-operation, equivalent to:
shl r0, *isrc2, idest*

nop — **Core No-Operation**

Assembler pseudo-operation, equivalent to:
shl r0, r0, r0

fnop — **Floating-Point No-Operation**

Assembler pseudo-operation, equivalent to:
shrd r0, r0, r0

These assembly-language instructions are provided for programmer convenience and for their self-documenting value. They do not represent actual i860 microprocessor instructions; instead, they are implemented by the i860 microprocessor instructions or instruction sequences shown.

The *const32* represents a signed constant expression in assembly language.

Floating-Point Instructions 8

CHAPTER 8
FLOATING-POINT INSTRUCTIONS

The floating-point section of i860 microprocessors comprises the floating-point registers and three processing units:

1. The floating-point multiplier
2. The floating-point adder
3. The graphics unit

This section executes not only floating-point operations but also 32- and 64-bit integer operations and graphics operations that utilize the 64-bit internal data path of the floating-point section.

8.1 PIPELINED AND SCALAR OPERATIONS

The architecture of the floating-point unit uses parallelism to increase the rate at which operations can be started. One type of parallelism used is called "pipelining." The pipelined architecture treats each operation as a series of more primitive operations (called "stages") that execute in parallel. Consider 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 and the number of clocks per stage may be different.

Figure 8-1 illustrates three-stage pipelining as found in the floating-point adder (and in the floating-point multiplier for single-precision). Each stage of the pipeline holds intermediate results and also (when introduced into the 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 (in this case, the single-precision add instruction **pfadd.ss**), all of one type (i.e., all adder instructions or all single-precision multiplier instructions). Each time a pipelined operation is performed using the pipeline in question, the status of the last stage becomes available in **fsr**, 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 i860 microprocessors, the number of pipeline stages ranges from one to three. A pipelined instruction with a three-stage pipeline writes to its *fdest* the result of the third prior instruction. A pipelined instruction with a two-stage pipeline writes to its *fdest* the result of the second prior operation. A pipelined operation with a one-stage pipeline stores the result of the prior operation.

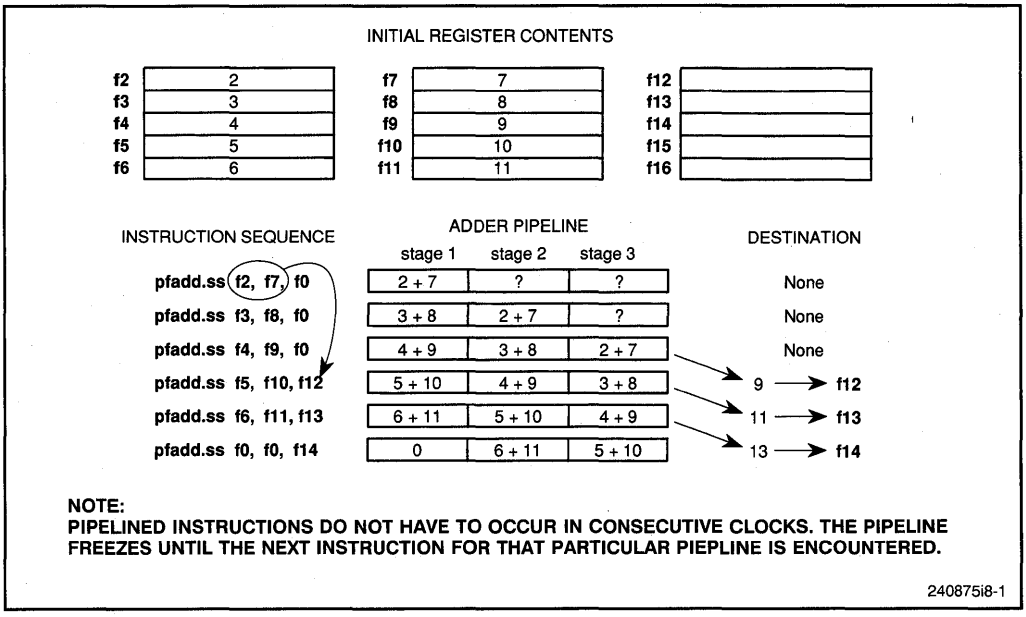


Figure 8-1. Pipelined Instruction Example

There are four floating-point pipelines: one for the multiplier, one for the adder, one for the graphics unit, and one for the pipelined floating-point load **pfld**. 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: two stages for double precision or three stages for single precision. 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.

8.1.1 Scalar Mode

In addition to the pipelined execution mode described above, i860 microprocessors also can execute floating-point and graphics instructions in "scalar" mode. Most floating-point and graphics instructions have both pipelined and scalar variants, distinguished by a bit in the instruction encoding. In scalar mode, the unit does not start a new operation until the previous 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).

8.1.2 Pipelining Status Information

Result status information in the **fsr** consists of the AA, AI, AO, AU, and AE bits for the adder, the MA, MI, MO, and MU bits for the multiplier, and the LRP0 and LRP1 bits (LRP bit on the 80860XR) for the load pipeline. This information arrives at the **fsr** via the pipeline in one of two ways:

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 **st.c** instruction updates the **fsr** and the U bit being written into the **fsr** is set, the store updates 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 processor traps if any of the status bits of the **fsr** indicate exceptions. Note that, in this case, the instruction that creates the exceptional condition is not the instruction at which the trap is reported.

8.1.3 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.

If *fdest* is the same register 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 operands are double-precision and three stages when they are 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 mixing precisions). The two-stage pipeline executes at two clocks per stage; the three-stage pipeline executes at one clock per stage.

8.1.4 Transition between Scalar and Pipelined Operations

When a scalar operation is executed in the adder, multiplier, or graphics unit, 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 drain 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.

Note that the **pfld** pipeline is not affected by scalar **fld** and **ld** instructions.

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

8.2 MULTIPLIER INSTRUCTIONS

The multiplier unit of the floating-point section performs not only the standard floating-point multiply operation but also provides reciprocal operations that can be used to implement floating-point division and square roots, and provides a special type of multiply that assists in integer multiply sequences. The multiply instructions can be pipelined.

Programming Notes

Complications arise with sequences of pipelined multiplier operations with mixed single- and double-precision inputs because the pipeline length is different for the two precisions. The complications can be avoided by not mixing the two precisions, i.e., by draining out all single-precision operations with dummy single-precision operations before starting double-precision operations, and *vice versa*. For the adventuresome, the rules for mixing precisions follow:

- **Single to Double Transitions.** When a pipelined multiplier operation with double-precision inputs is executed and the previous multiplier operation was pipelined with single-precision inputs, the third previous (last stage) result is stored, and the previous operation (first stage) is advanced to the second stage (now the last stage). The second previous operation (old second stage) is discarded. The next pipelined multiplier operation stores the single-precision result.
- **Double to Single Transitions.** When a pipelined multiplier operation with single-precision inputs is executed and the previous multiplier operation was pipelined with double-precision inputs, the result of the second previous multiplier operation advances to the second stage, the result of the previous multiplier operation advances to the second stage, and a single- or double-precision zero is placed in the last stage of the pipeline. The next pipelined multiplier operation stores zero instead of the result of the prior operation, and the MRP bit of **fsr** for that next operation is **undefined.**

8.2.1 Floating-Point Multiply

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

These instructions perform a standard multiply operation.

Programming Notes

Fsrc1 must not be the same as *fdest* for pipelined operations. For best performance when the prior operation is scalar, *fsrc1* should not be the same as the *fdest* of the prior operation.

The **pfmul3.dd** instruction is only for use by exception handlers in restoring pipeline contents (refer to "Pipeline Preemption" in Chapters 9 and 10). It should not be mixed in instruction sequences with other pipelined multiplier instructions.

8.2.2 Floating-Point Multiply Low

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 (*fsrc1* mantissa × *fsrc2* mantissa)

The **fmlow** instruction multiplies the mantissas of its floating-point operands. It operates only on double-precision operands.

A mantissa is a 53-bit binary integer of the form 1.**f**, where **f** is the 52-bit fractional part of a floating-point operand. Multiplying two 53-bit mantissas produces a 106-bit true result, which can be partitioned into the form **ij.gh**, where **i** and **j** are single bits, **g** is 51 bits, and **h** is 53 bits. As Figure 8-2 shows, the values **j.g** are not returned by **fmlow**; these values, in normalized form, form part of the result that would be returned by an **fmul.dd** operation on the same operands. With **fmlow**, bits 0–52 of *fdest* receive **h**. (In an **fmul.dd** operation these bits of the true result would be lost.) Bit 53 of *fdest* receives **i**, which is the amount by which the exponent would be increased in an **fmul.dd** operation as the first step of normalization. The high-order 10 bits of *fdest* are undefined.

An **fmlow** instruction can perform 32-bit integer multiplies. The two 32-bit operands should be placed in the low-order parts of *fsrc1* and *fsrc2*. The value returned in the low-order 53 bits of *fdest* is the same as that of the low-order 53 bits of an integer multiply.

The **fmlow** instruction does not update the result-status bits of **fsr** and does not cause source or result exceptions. (However, its execution may trigger a trap to report a result exception caused by a prior floating-point instruction.)

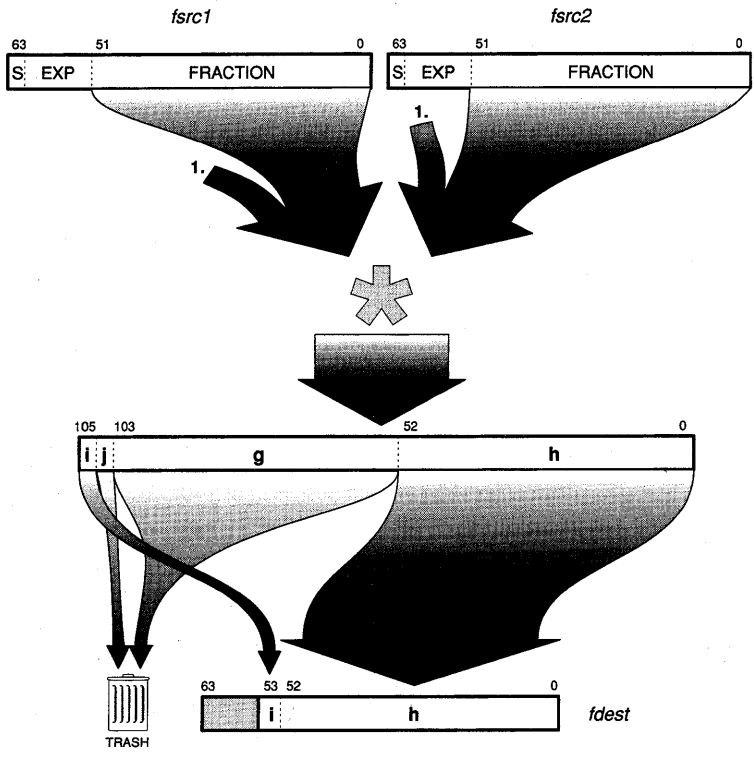


Figure 8-2. FMLOW Operation

8.2.3 Floating-Point Reciprocals

frcp.p *fsrc2, fdest* **Floating-Point Reciprocal**

fdest ← 1 / *fsrc2* with absolute mantissa error < 2^{-7}

frsqr.p *fsrc2, fdest* **Floating-Point Reciprocal Square Root**

fdest ← 1 / $\sqrt{}$(*fsrc2*) with absolute mantissa error < 2^{-7}

The **frcp** and **frsqr** instructions are intended to be used with algorithms such as the Newton-Raphson approximation to compute divide and square root. Assemblers and compilers must encode *fsrc1* as **f0**. A Newton-Raphson approximation may produce a result that is different from the IEEE standard in the two least significant bits of the mantissa. A library routine supplied by Intel may be used to calculate the correct IEEE-standard rounded result.

Traps

frcp causes a source-exception trap if *fsrc2* is zero. **frsqr** causes a source-exception trap if *fsrc2* ≤ 0.

8.3 ADDER INSTRUCTIONS

The adder unit of the floating-point section provides floating-point addition, subtraction, and comparison, as well as conversion from floating-point to integer formats.

8.3.1 Floating-Point Add and Subtract

fadd.p *fsrc1, fsrc2, fdest*	**Floating-Point Add**
fdest ← *fsrc1* + *fsrc2*	
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*	
fsub.p *fsrc1, fsrc2, fdest*	**Floating-Point Subtract**
fdest ← *fsrc1* − *fsrc2*	
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*	
famov.r *fsrc1, fdest*	**Floating-Point Adder Move**
fdest ← *fsrc1*	
pfamov.r *fsrc1, fdest*	**Pipelined Floating-Point Adder Move**
fdest ← last stage adder result Advance A pipeline one stage A pipeline first stage ← *fsrc1*	

These instructions perform standard addition and subtraction operations.

The **famov** and **pfamov** instructions send *fsrc1* through the floating-point adder, preserving the value of −0 (minus zero) when *fsrc1* is −0. (Note that **(p)fadd.p** *fsrc1*, **f0**, *fdest* may round −0 to +0, depending on the RM bits of **fsr**.) The **pfamov** instruction is used by the trap handler to restore pipeline states. *Fsrc2* for **(p)famov** must be encoded as **f0** by assemblers and compilers.

Programming Notes

In order to allow conversion from double precision to single precision, an **famov** or **pfamov** instruction may have double-precision inputs and a single-precision output. In assembly language, this conversion is specified using the **fmov** or **pfmov** pseudo-operation with the **.ds** suffix.

fmov.ds *fsrc1, fdest*	**Convert Double to Single**
Equivalent to **famov.ds** *fsrc1*, **f0**, *fdest*	
pfmov.ds *fsrc1, fdest*	**Pipelined Convert Double to Single**
Equivalent to **pfamov.ds** *fsrc1*, **f0**, *fdest*	

Conversion from single to double is accomplished by **famov.sd** or **pfamov.sd**. In assembly language, this conversion is specified by the **fmov** or **pfmov** pseudo-operation with the **.sd** suffix.

fmov.sd *fsrc1, fdest*	**Convert Single to Double**
Equivalent to **famov.sd** *fsrc1, fdest*	
pfmov.sd *fsrc1, fdest*	**Pipelined Convert Single to Double**
Equivalent to **pfamov.sd** *fsrc1, fdest*	

8.3.2 Floating-Point Compares

pfgt.p *fsrc1, fsrc2, fdest* **Pipelined Floating-Point Greater-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

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

(Assembler sets R-bit of instruction.)
fdest ← last stage adder result
CC cleared if *fsrc1* ≤ *fsrc2*, else set
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

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

There are no corresponding scalar versions of the floating-point compare instructions. The pipelined instructions can be used either within a sequence of pipelined instructions or within a sequence of nonpipelined (scalar) instructions.

pfgt.p should be used for A > B and A < B comparisons. **pfle.p** should be used for A ≥ B and A ≤ B comparisons. **pfeq.p** should be used for A = B and A ≠ B comparisons. The mnemonics **pfle.p** and **pfgt.p** refer to the same opcode; the only difference in instruction coding is the setting of the R-bit.

8

Traps

Compares never cause result exceptions when the result is stored. They do trap on invalid input operands.

Programming Notes

The only difference between **pfgt.p** and **pfle.p** is the encoding of the R bit of the instruction and the way in which the trap handler treats unordered compares. The R bit normally indicates result precision, but in the case of these instructions it is not used for that purpose. The trap handler can examine the R bit to help determine whether an unordered compare should set or clear CC to conform with the IEEE standard for unordered compares. For **pfgt.p** and **pfeq.p**, it should clear CC; for **pfle.p**, it should set CC.

For best performance, a **bc** or **bnc** instruction should not directly follow a **pfgt** or **pfeq** instruction. Be sure, however, that intervening instructions do not change CC.

8.3.3 Floating-Point to Integer Conversion

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

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

ftrunc.v *fsrc1, fdest* **Floating-Point to Integer Truncation**

fdest ← 64-bit value with low-order 32 bits equal to integer part of *fsrc1*

pftrunc.v *fsrc1, fdest* **Pipelined Floating-Point to Integer Truncation**

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*

The instructions **fix**, **pfix**, **ftrunc**, and **pftrunc** must specify double-precision results. The low-order 32 bits of the result contain the integer part of *fsrc1* represented in twos-complement form. The high-order 32 bits of the result are *undefined*. For **fix** and **pfix**, the integer is selected according to the rounding mode specified by RM in the **fsr**. The instructions **ftrunc** and **pftrunc** are identical to **fix** and **pfix**, except that RM is not consulted; rounding is always toward zero. Assembler and compilers should encode *fsrc2* as **f0**.

Traps

The instructions **fix**, **pfix**, **ftrunc**, and **pftrunc** signal overflow (AO bit of **fsr** set) if the integer part of *fsrc1* is bigger than what can be represented as a 32-bit twos-complement integer. Underflow and inexact are never signaled.

Adder overflow can occur due either to a true floating-point operation (for example, **pfadd.p** or **pfeq.p**) or to an integer conversion operation (**fix.v, pfix.v, ftrunc.v, pftrunc.v**). For a true floating-point operation, the exponent of the result will be all ones. For an integer conversion operation, the exponent of the result will be less than all ones. When adder overflow occurs, the trap handler can distinguish between the two cases by examining the exponent of the result.

8.4 DUAL OPERATION INSTRUCTIONS

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

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

pfmam.p *fsrc1, fsrc2, fdest* **Pipelined Floating-Point Multiply with Add**

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

pfmsm.p *fsrc1, fsrc2, fdest* **Pipelined Floating-Point Multiply with Subtract**

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

The instructions **pfam**, **pfsm**, **pfmam**, and **pfmsm** initiate both an adder (A-unit) operation and a multiplier (M-unit) operation. The source precision specified by **.p** applies to the source operands of the multiplication. The result precision normally specified by **.p** controls in this case both the precision of the source operands of the addition or subtraction and the precision of all the results.

Suffix	Precision of Source of Multiplication	Precision of Source of Add or Subtract and Result of all Operations
.ss	single	single
.sd	single	double
.dd	double	double

The instructions **pfmam** and **pfmsm** are identical to **pfam** and **pfsm** except that **pfmam** and **pfmsm** transfer the last stage result of the multiplier to *fdest*.

Six operands are required, but the instruction format can specify 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.
  - The constant registers KR and KI can store the value of *fsrc1* and subsequently supply that value to the M-pipeline in place of *fsrc1*.
  - 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*, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.
  3. Operand-1 of the adder can be *fsrc1*, the T-register, the last-stage result of the multiplier pipeline, 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 8-3 shows all the possible data paths surrounding the adder and multiplier. Table 8-1 shows how the various encodings of DPC select different data paths. Figure 8-4 illustrates the actual data path for each dual-operation instruction.

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 by this manual to designate classes of related instructions. Each value of DPC has a unique mnemonic associated with it. An initial "m" distinguishes the **pfmam.p**, and **pfmsm.p** classes from the **pfam.p**, and **pfsm.p** classes. Figure 8-5 explains how the rest of these mnemonics are derived.

Programming Notes

When *fsrc1* goes to M-unit *op1* or to KR or KI, *fsrc1* must not be the same as *fdest*. For best performance when the prior operation is scalar and the M-unit *op1* is *fsrc1*, *fsrc1* should not be the same as the *fdest* of the prior operation.

Dual-operation instructions that feed the adder result back into the multiplier or adder do so regardless of the register specified as *fdest*. In particular, even though *fdest* is **f0** or **f1**, the value fed back is not zero, but rather the actual multiplier output.

When dual-operation instructions are used with single-precision operands, all 64 bits of the T, KR, and KI registers are updated, but the values stored there are not converted to double-precision format. (The exponent bias is not adjusted for double precision.)

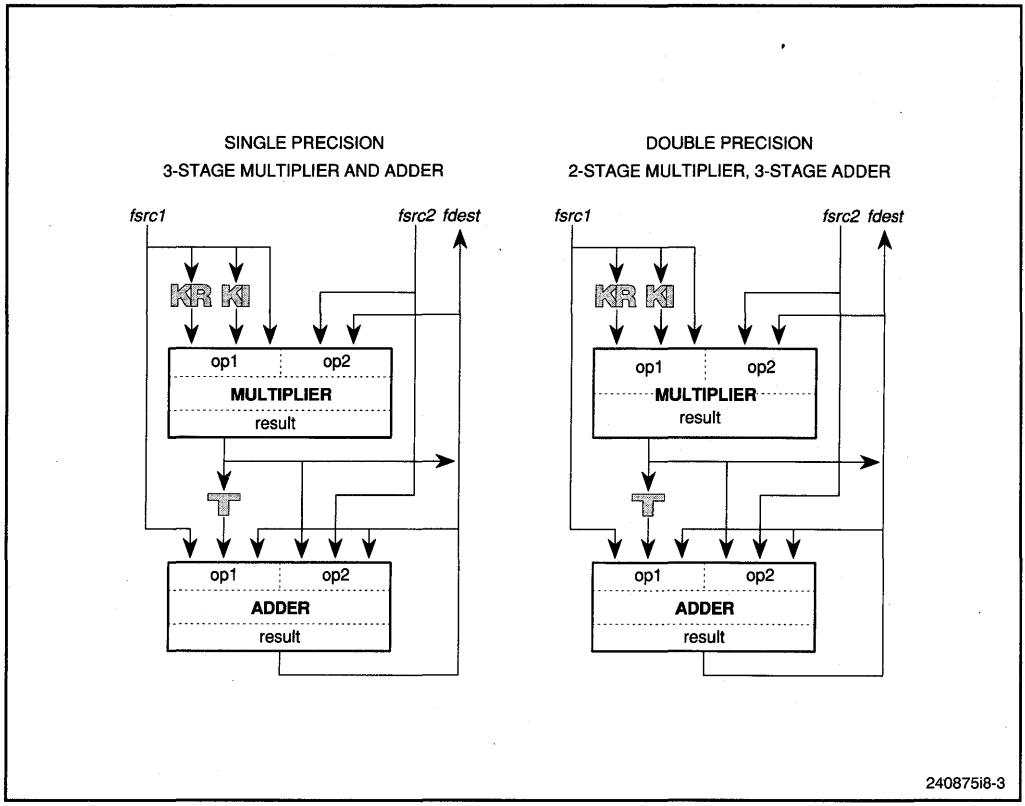


Figure 8-3. Dual-Operation Data Paths

Instead, zeros are inserted as pads in exponent bits 10..8 and as the fraction's least significant 29 bits (bits 28..0). All 64 bits of the T, KR, and KI registers can be initialized to zero with this sequence of instructions:

```
r2apt.ss f0, f0, f0
r2apt.ss f0, f0, f0
r2apt.ss f0, f0, f0
i2apt.ss f0, f0, f0
```

Single-precision values are stored in these 64-bit registers in a format that does not conform to the standard for double-precision numbers; therefore, leaving a single-precision value in T, KR, or KI can cause unexpected results if a later double-precision operation refers to one of these registers. Likewise, valid double-precision values left in T, KR, or KI can cause unexpected results if a later single-precision operation uses one of these registers. A trap may occur in some of these cases. Even if a trap does not occur, the bit patterns of one precision will represent a different value in the other precision. Therefore, programs that use dual-operation instructions should clear T, KR, and KI before switching precisions.

Table 8-1. 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	M 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**							

NOTE:
* 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.

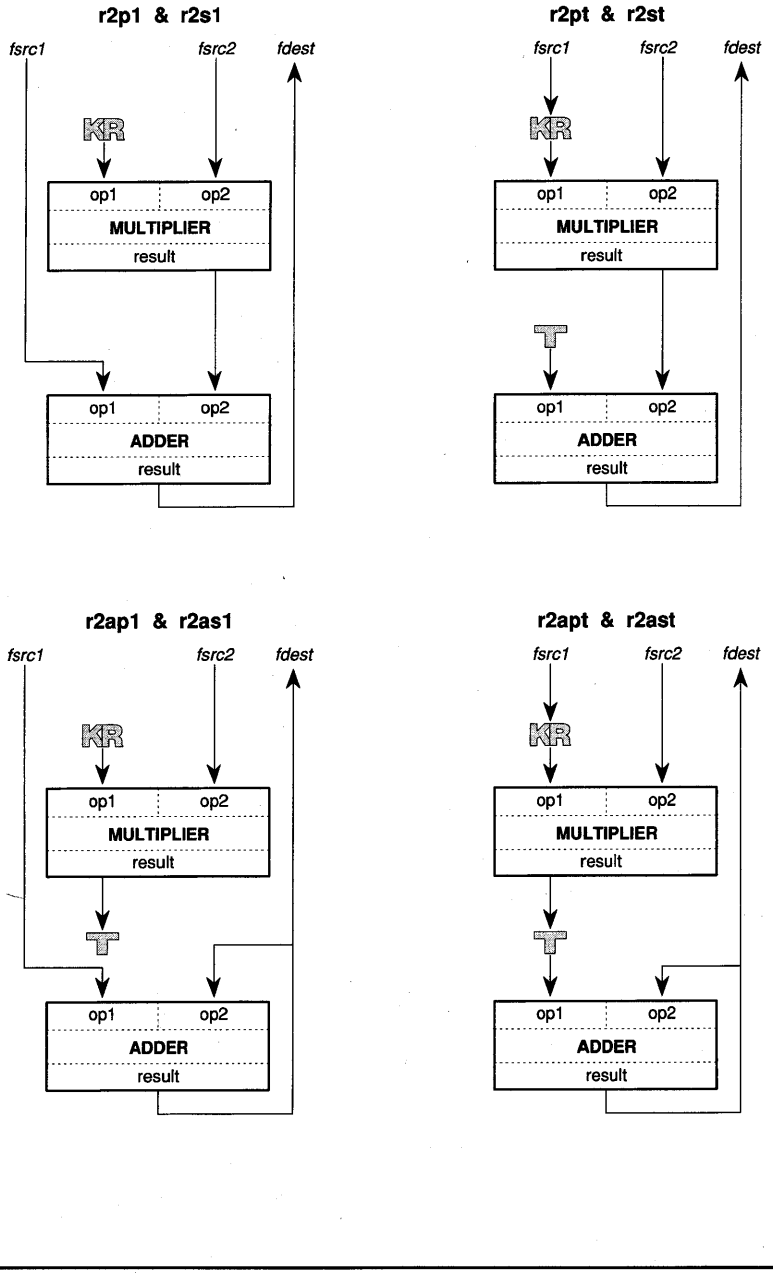


Figure 8-4. Data Paths by Instruction (1 of 8)

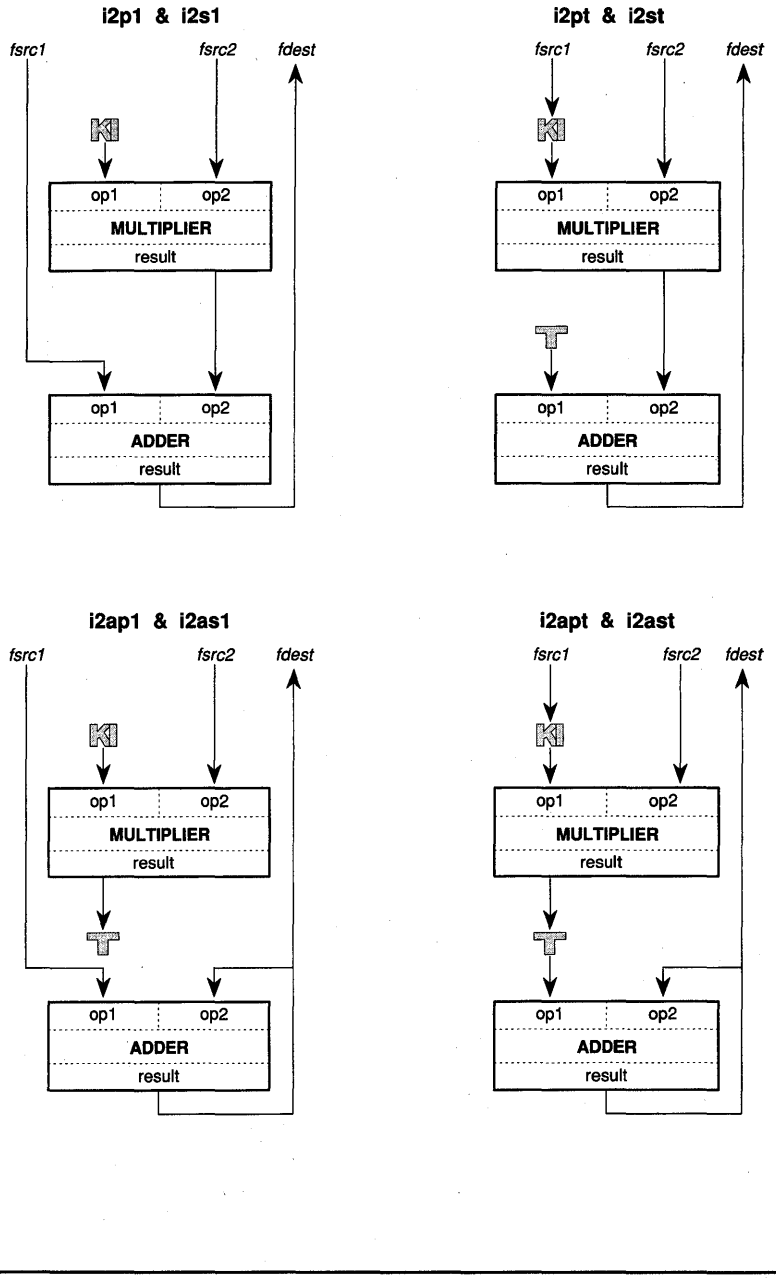


Figure 8-4. Data Paths by Instruction (2 of 8)

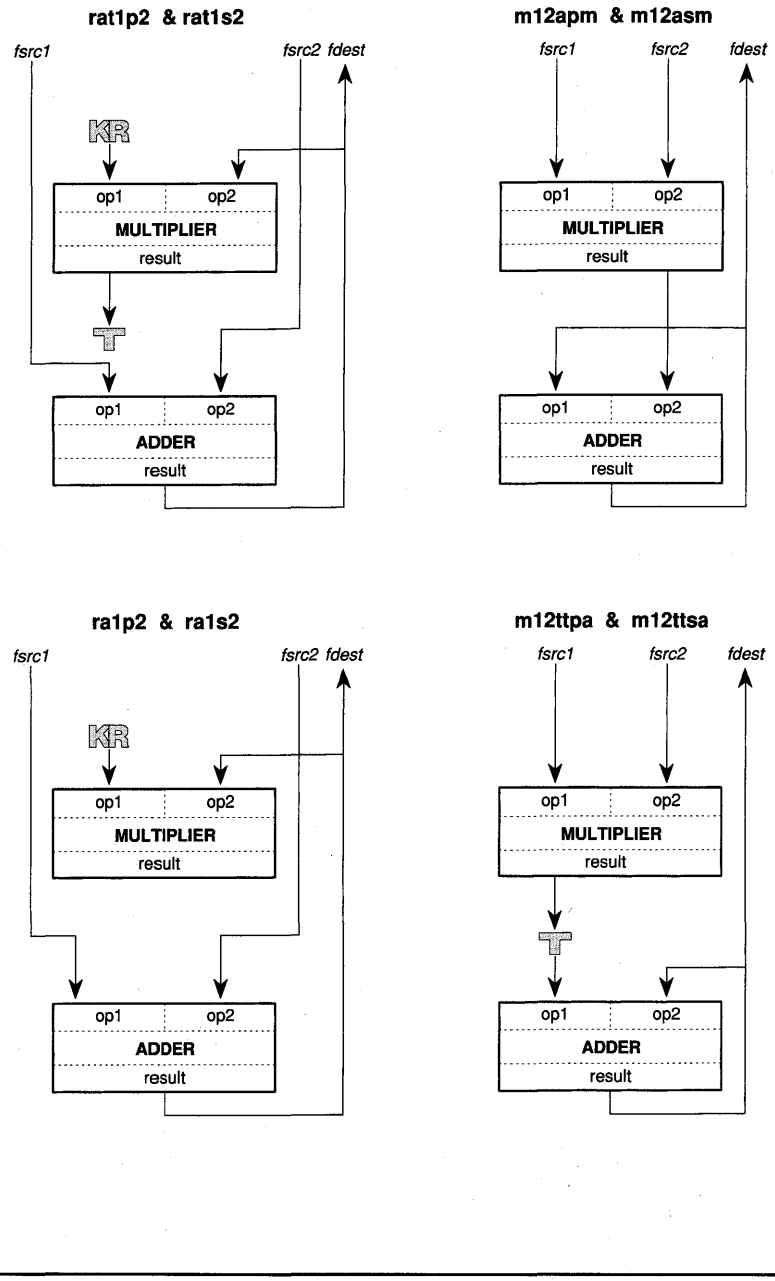


Figure 8-4. Data Paths by Instruction (3 of 8)

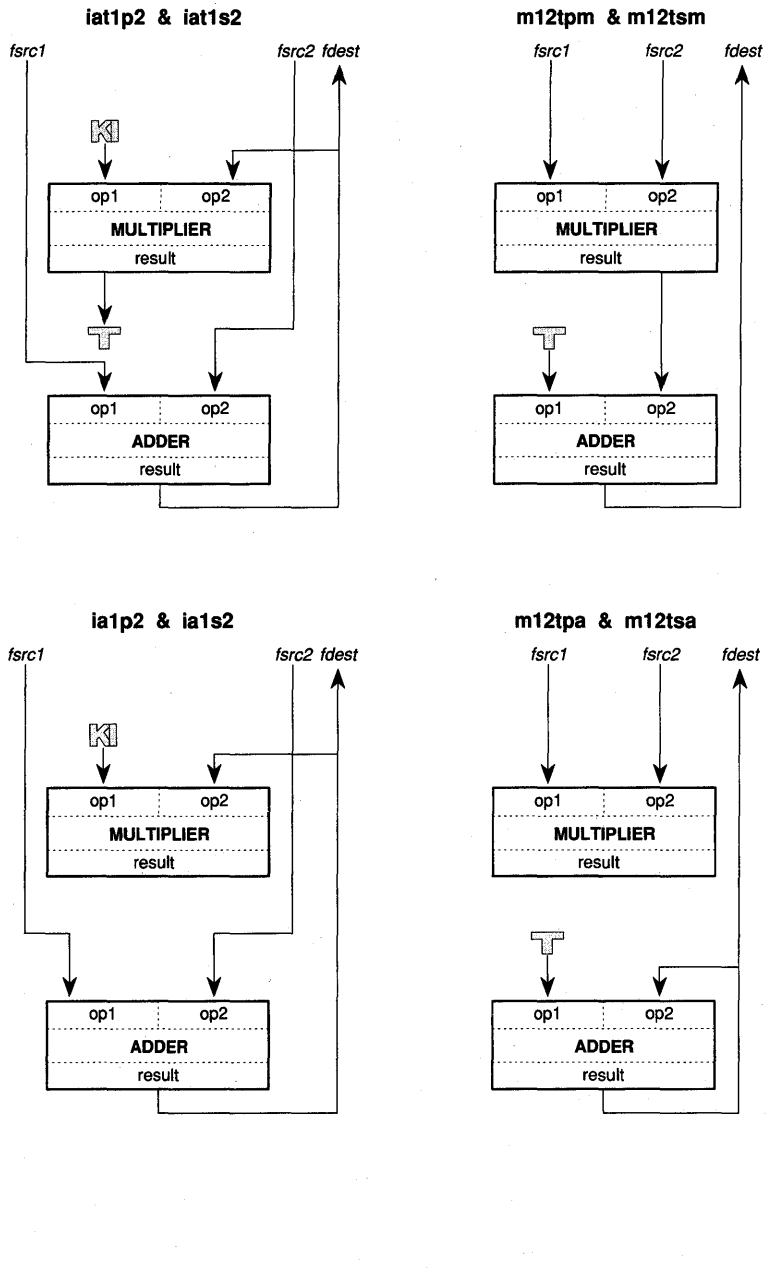


Figure 8-4. Data Paths by Instruction (4 of 8)

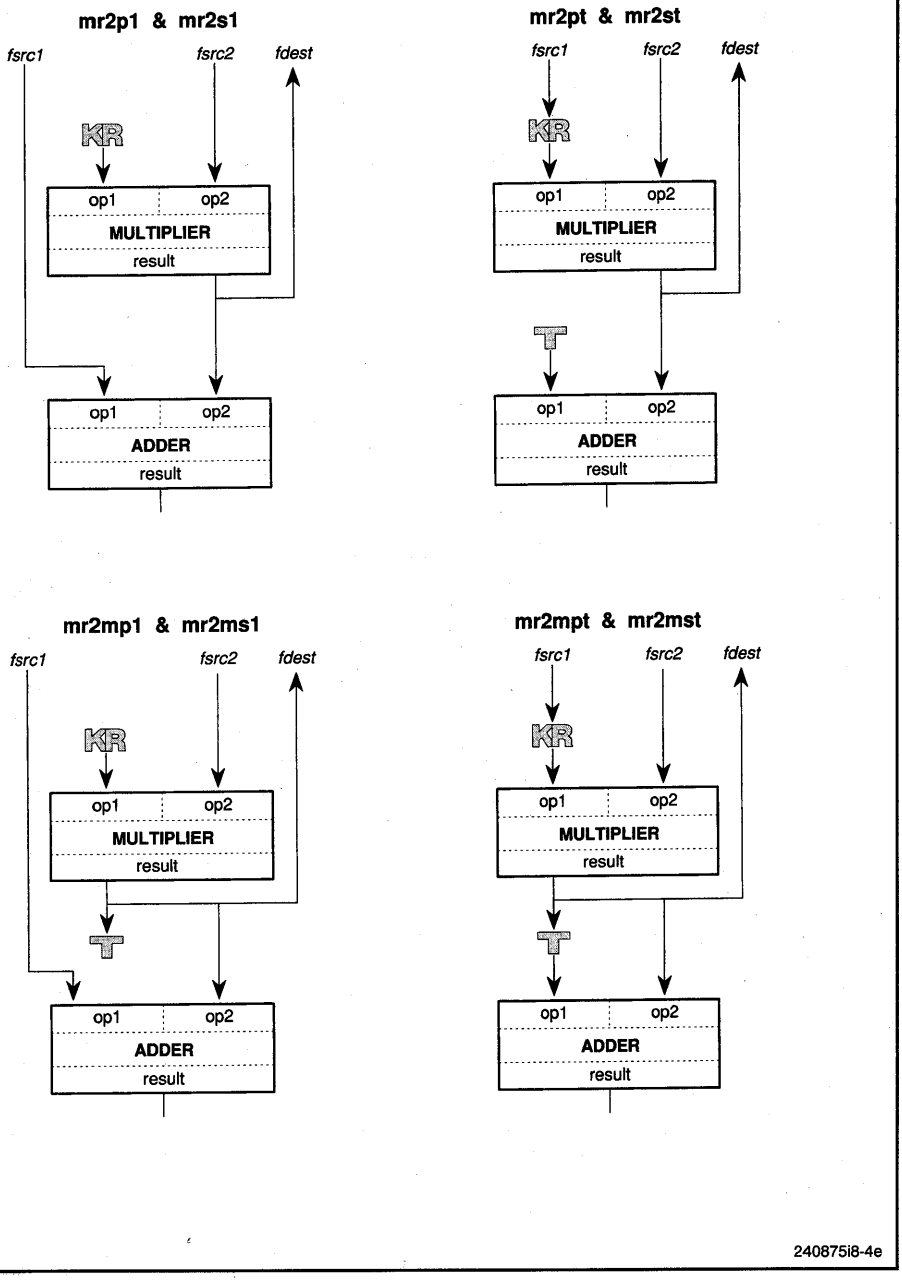


Figure 8-4. Data Paths by Instruction (5 of 8)

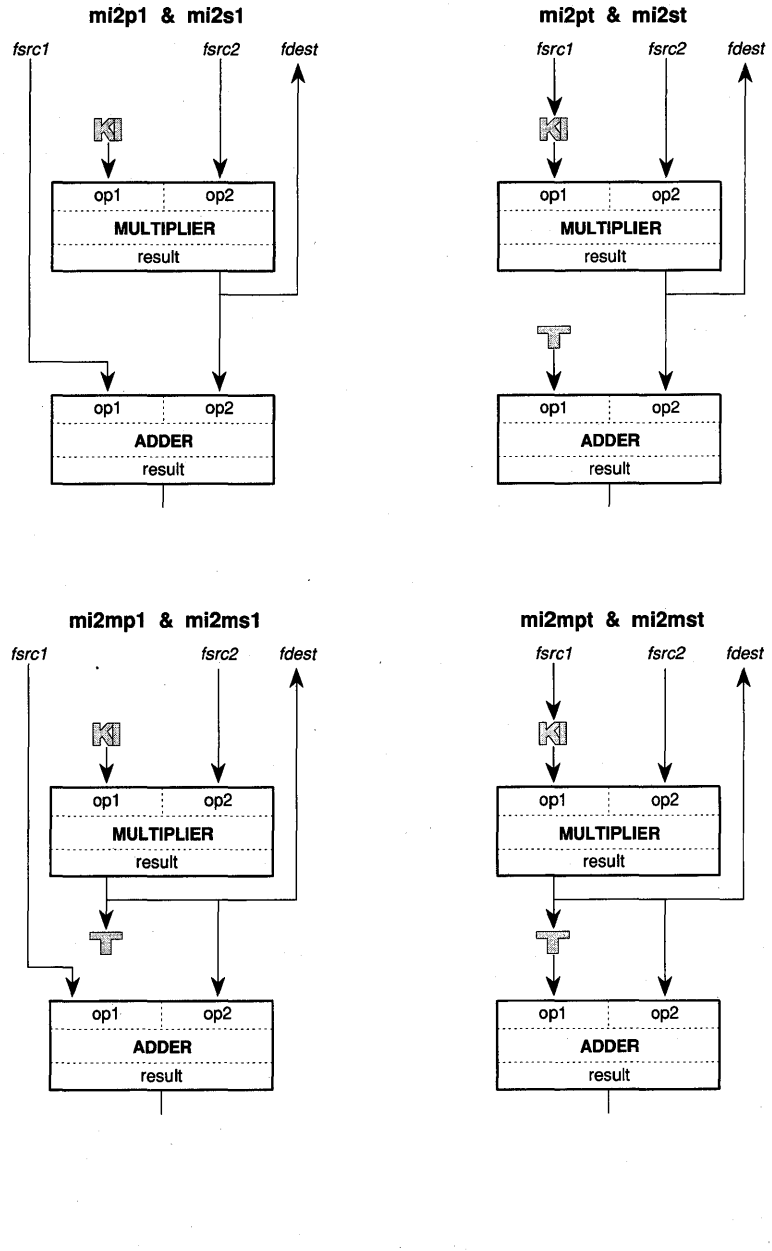


Figure 8-4. Data Paths by Instruction (6 of 8)

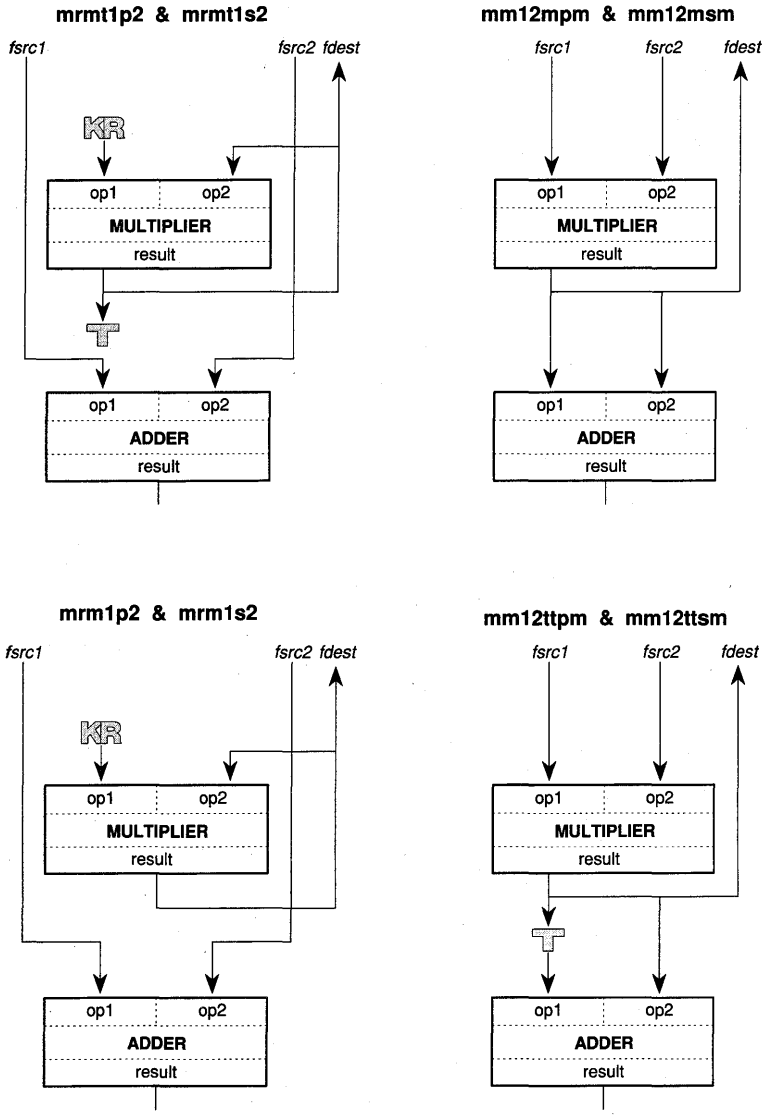
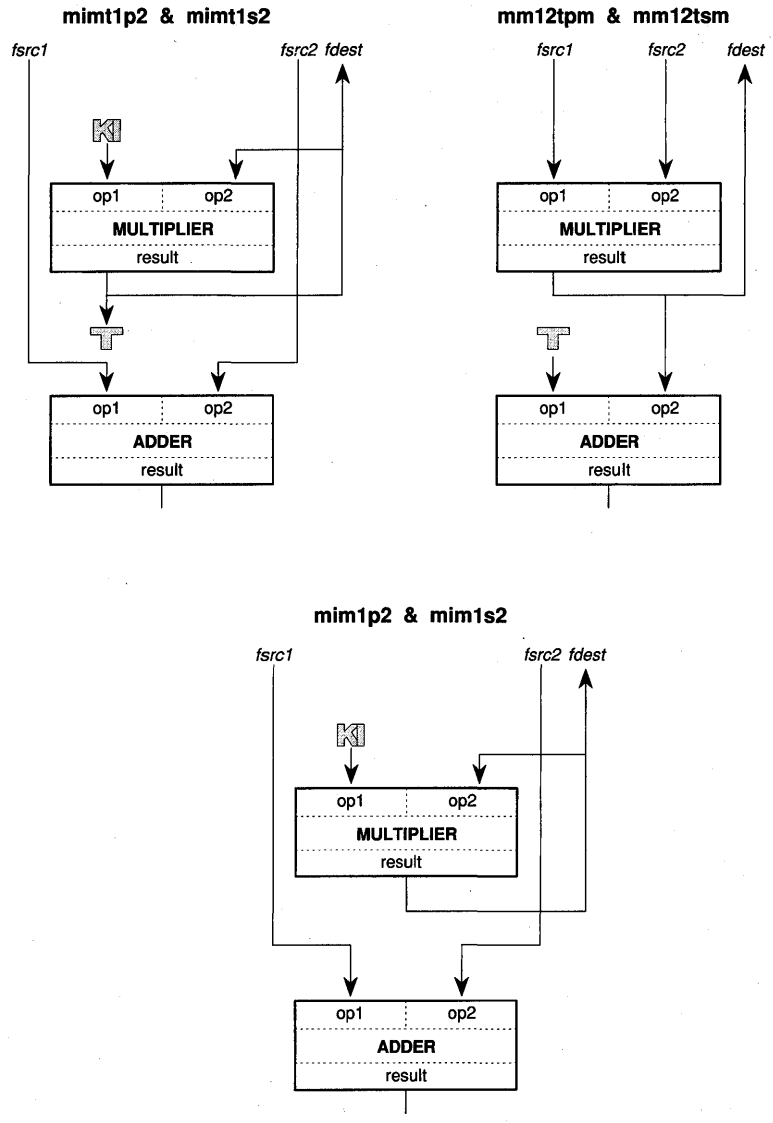


Figure 8-4. Data Paths by Instruction (7 of 8)



240875i8-4h

Figure 8-4. Data Paths by Instruction (8 of 8)

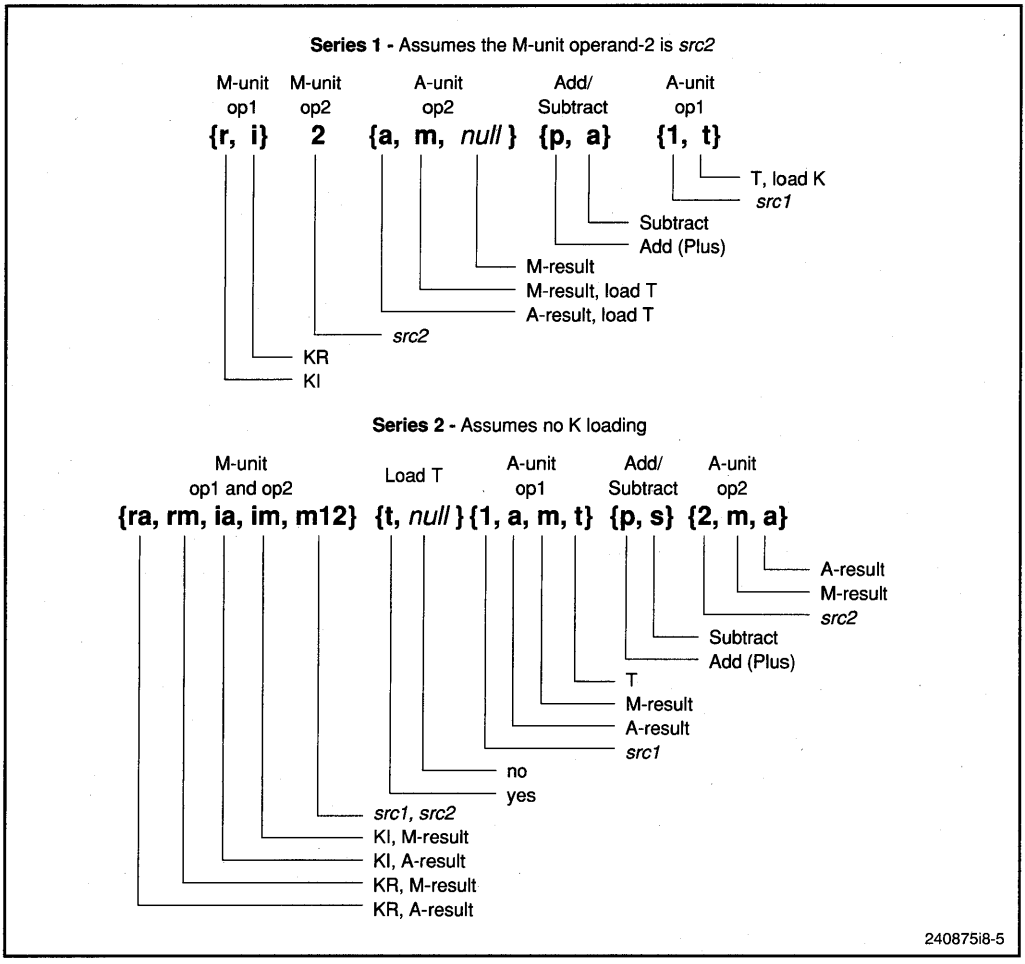


Figure 8-5. Data Path Mnemonics

8.5 GRAPHICS UNIT

The graphics unit operates on 32- and 64-bit integers stored in the floating-point register file. This unit supports long-integer arithmetic and 3-D graphics drawing algorithms. Operations are provided for pixel shading and for hidden surface elimination using a Z-buffer.

Programming Notes

In a pipelined graphics operation, if *fdest* is not **f0**, then *fdest* must not be the same as *fsrc1* or *fsrc2*.

For best performance, the result of a scalar operation should not be a source operand in the next instruction, unless the next instruction is a multiplier or adder operation.

8.5.1 Long-Integer Arithmetic

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

fdest ←*fsrc1* − *fsrc2* (2's complement integer arithmetic)

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

fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* − *fsrc2* (2's complement integer arithmetic)

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

fdest ← *fsrc1* + *fsrc2* (2's complement integer arithmetic)

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

fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2* (2's complement integer arithmetic)

.w = **.ss** (32 bits), or **.dd** (64 bits)

The **fiadd** and **fisub** instructions implement arithmetic on integers up to 64 bits wide. Such integers are loaded into the same registers that are normally used for floating-point operations. These instructions do not set CC nor do they cause floating-point traps due to overflow.

Programming Notes

In assembly language, **fiadd** and **pfiadd** are used to implement the **fmov.ss, fmov.dd, pfmov.ss,** and **pfmov.dd** pseudo instructions.

fmov.ss *fsrc1, fdest* — **Single Move**

Equivalent to **fiadd.ss** *fsrc1*, **f0**, *fdest*

pfmov.ss *fsrc1, fdest* — **Pipelined Single Move**

Equivalent to **pfiadd.ss** *fsrc1*, **f0**, *fdest*

fmov.dd *fsrc1, fdest* — **Double Move**

Equivalent to **fiadd.dd** *fsrc1*, **f0**, *fdest*

pfmov.dd *fsrc1, fdest* — **Pipelined Double Move**

Equivalent to **pfiadd.dd** *fsrc1*, **f0**, *fdest*

8.5.2 3-D Graphics Operations

i860 microprocessors support high-performance 3-D graphics applications by supplying operations that assist in the following common graphics functions:

1. Hidden surface elimination.
2. Distance interpolation.
3. 3-D shading using intensity interpolation.

The interpolation operations of i860 microprocessors support graphics applications in which the set of points on the surface of a solid object is represented by polygons. The distances from the viewer and color intensities of the vertices of the polygon are known, but the distances and intensities of other points must be calculated by interpolation between the known values.

Certain fields of the **psr** are used by the graphics instructions, as illustrated in Figure 8-6.

The merge instructions are those that utilize the 64-bit MERGE register. 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.

8.5.2.1 Z-BUFFER CHECK INSTRUCTIONS

A Z-buffer aids hidden-surface elimination by associating with a pixel a value that represents the distance of that pixel from the viewer. When painting a point at a specific pixel location, three-dimensional drawing algorithms calculate the distance of the point from the viewer. If the point is farther from the viewer than the point that is already represented by the pixel, the pixel is not updated. i860 microprocessors support distance values that are either 16-bits or 32-bits wide. The size of the Z-buffer values is independent of the pixel size. Z-buffer element size is controlled by whether the 16-bit instruction **fzchks** or the 32-bit instruction **fzchkl** is used; pixel size is controlled by the PS field of the **psr**.

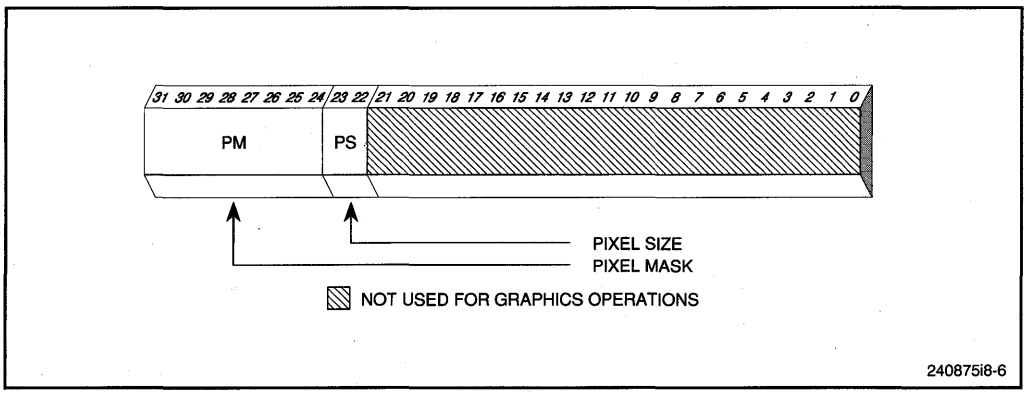


Figure 8-6. PSR Fields for Graphics Operations

All operands *fsrc1, fsrc2,* and *fdest* designate 64-bit register pairs.
Consider PM as an array of eight bits PM(7)..PM(0), where PM(0) is the least-significant bit.

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

Consider operands as arrays of four 16-bit fields *fsrc1*(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0) 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

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

Consider operands as arrays of four 16-bit fields *fsrc1*(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0) 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 ← last-stage graphics-unit result
  last-stage graphics-unit result(i) ← smaller of *fsrc2*(i) and *fsrc1*(i)
OD
MERGE ← 0

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

Consider operands as arrays of two 32-bit fields *fsrc1*(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0) 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

pfzchkl *fsrc1, fsrc2, fdest* **Pipelined 32-Bit Z-Buffer Check**

Consider operands as arrays of two 32-bit fields *fsrc1*(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0) 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-unit result
  last-stage graphics-unit result ← smaller of *fsrc2*(i) and *fsrc1*(i)
OD
MERGE ← 0

The instructions **fzchks** and **fzchkl** perform multiple unsigned-integer (ordinal) comparisons. The inputs to the instructions **fzchks** and **fzchkl** are normally taken from two arrays of values, each of which typically represents the distance of a point from the viewer. One array contains distances that correspond to points that are to be drawn; the other contains distances that correspond to points that have already been drawn (a Z-buffer). The instructions compare the distances of the points to be drawn against the values in the Z-buffer and set bits of PM to indicate which distances are smaller than those in the Z-buffer. Previously calculated bits in PM are shifted right so that consecutive **fzchks** or **fzchkl** instructions accumulate their results in PM. Subsequent **pst.d** instructions use the bits of PM to determine which pixels to update.

8.5.2.2 PIXEL ADD

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

fdest ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift, then load MERGE register from *fsrc1* + *fsrc2* as defined in Table 8-2

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

fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift, then load MERGE register from *fsrc1* + *fsrc2* as defined in Table 8-2

The **faddp** instruction implements interpolation of color intensities. The 8- and 16-bit pixel formats use 16-bit intensity interpolation. Being a 64-bit instruction, **faddp** does four 16-bit interpolations at a time. The 32-bit pixel formats use 32-bit intensity interpolation; consequently, **faddp** performs them two at a time. By itself **faddp** implements linear interpolation; combined with **fiadd**, nonlinear interpolation can be achieved.

Figure 8-7 illustrates **faddp** when PS is set for 8-bit pixels. Each operand can be treated as four fixed-point numbers, each with an 8-bit integer portion and an 8-bit fractional portion. Each fixed-point sum is rounded to 8 integer bits by truncation when it is loaded into the MERGE register. With each **faddp** instruction, the MERGE register is shifted right by 8 bits. Two **faddp** instructions should be executed consecutively, one to interpolate for even-numbered pixels, the next to interpolate for odd-numbered pixels. The shifting of the MERGE register has the effect of merging the results of the two **faddp** instructions.

Figure 8-8 illustrates **faddp** when PS is set for 16-bit pixels. Each operand can be treated as four fixed-point numbers, each with a 6-bit integer portion and a 10-bit fractional portion. Each fixed-point sum is rounded to 6 bits by truncation when it is loaded into the MERGE register. With each **faddp**, the MERGE register is shifted right by 6 bits. Normally, three **faddp** instructions are executed consecutively, one for each color represented in a pixel. The shifting of MERGE causes the results of consecutive **faddp** instructions to be accumulated in the MERGE register. Note that each one of the first set of 6-bit values loaded into MERGE is further truncated to 4-bits when it is shifted to the extreme right of the 16-bit pixel.

Figure 8-9 illustrates **faddp** when PS is set for 32-bit pixels. Each operand can be treated as two fixed-point numbers, each with an 8-bit integer portion and a 24-bit fractional portion. Each fixed-point sum is rounded to 8 bits by truncation when it is loaded into

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

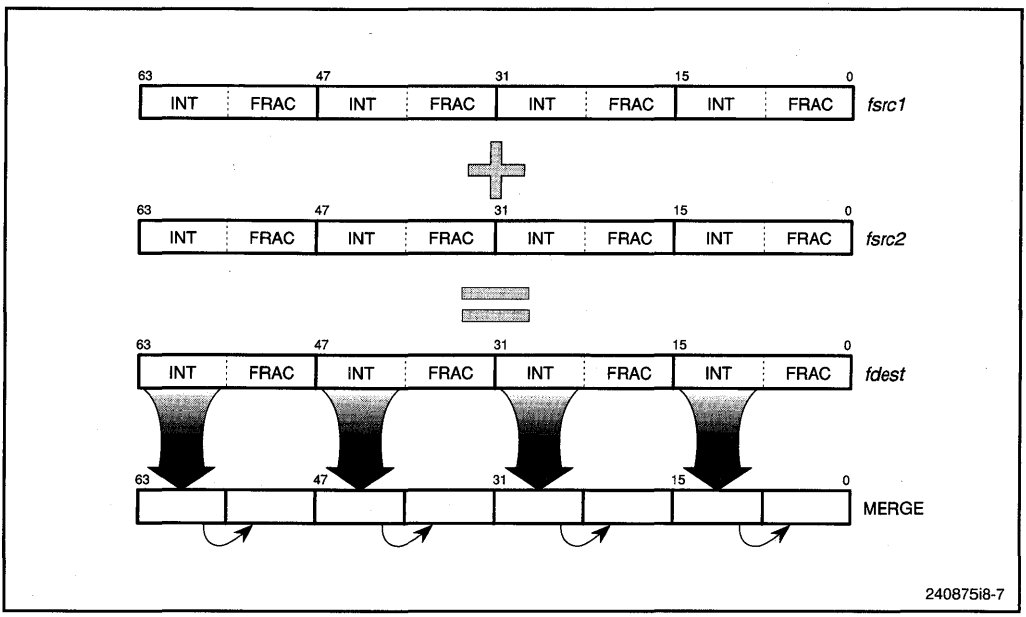


Figure 8-7. FADDP with 8-Bit Pixels

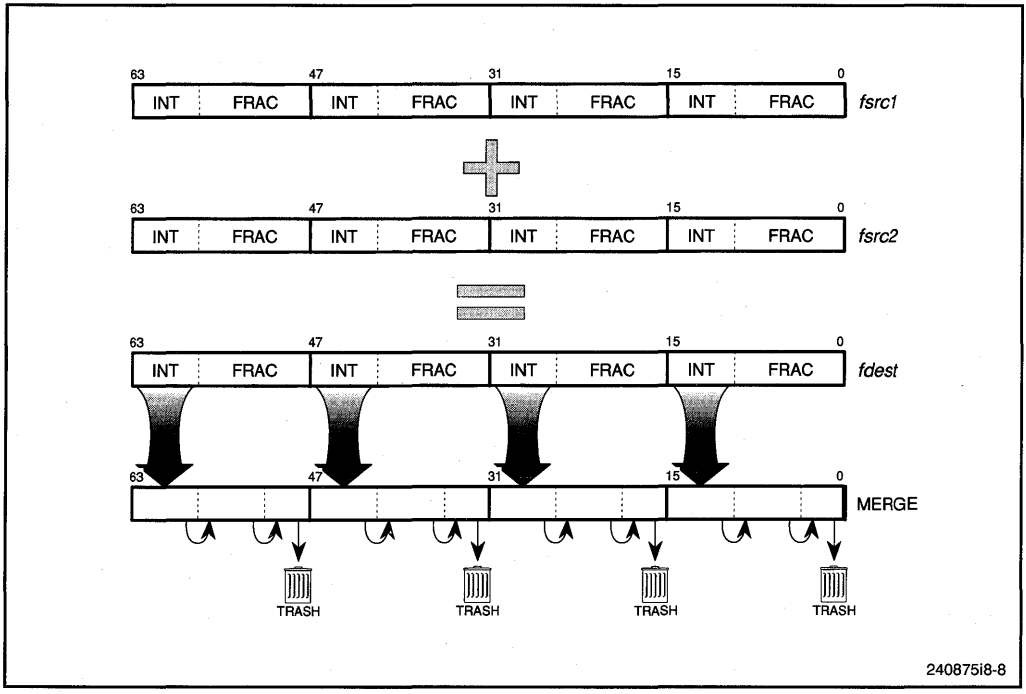


Figure 8-8. FADDP with 16-Bit Pixels

8

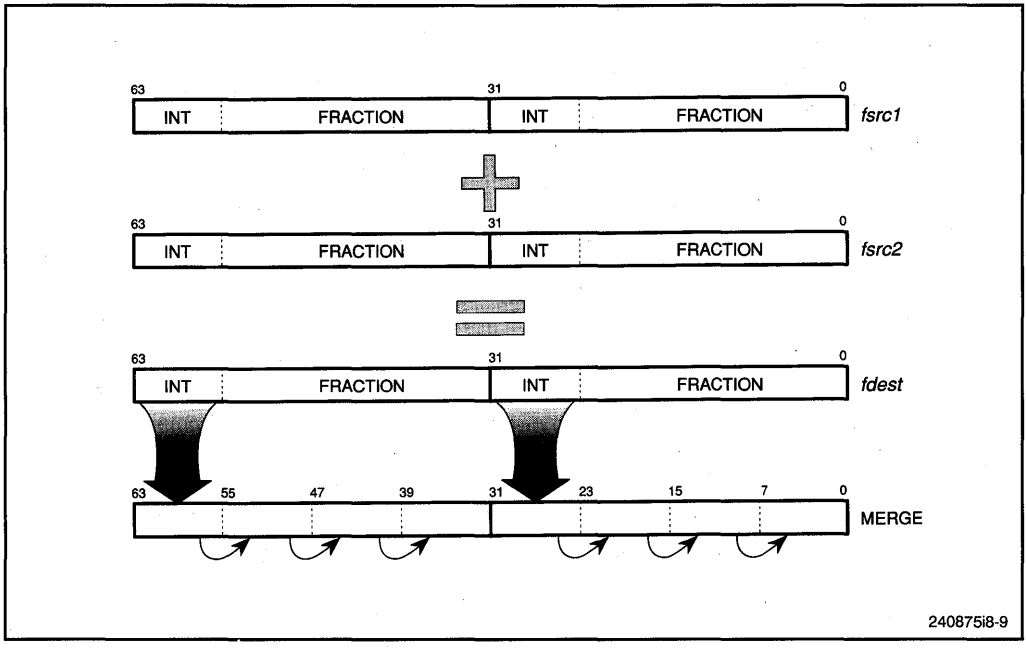


Figure 8-9. FADDP with 32-Bit Pixels

the MERGE register. With each **faddp**, the MERGE register is shifted right by 8 bits. Normally, three **faddp** instructions are executed consecutively, one for each color represented in a pixel. The shifting of MERGE causes the results of consecutive **faddp** instructions to be accumulated in the MERGE register.

Programming Notes

When interpolating with a negative slope, one pixel's most-significant bit may carry into the least-significant bit of the fraction of the neighboring pixel on the left. The carry is due to the fact that **faddp** instruction does not treat pixels individually; instead, it adds their operands exactly as **fiadd.dd** does—in one 64-bit operation. (The only difference between **fiadd.dd** and **faddp** is the effect on the MERGE register.) Interpolation algorithms should compensate for the carry.

8.5.2.3 Z-BUFFER ADD

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

fdest ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16, then load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

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

fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16, then load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

The **faddz** instruction implements linear interpolation of distance values such as those that form a Z-buffer. With **faddz**, 16-bit Z-buffers can use 32-bit distance interpolation, as Figure 8-10 illustrates. Each operand can be treated as two fixed-point numbers, each with a 16-bit integer portion and a 16-bit fractional portion. Each fixed-point sum is rounded to 16 bits by truncation when it is loaded into the MERGE register. With each **faddz**, the MERGE register is shifted right by 16 bits. Normally, two **faddz** instructions are executed consecutively. The shifting of MERGE causes the results of consecutive **faddz** instructions to be accumulated in the MERGE register.

32-bit Z-buffers can use 32-bit or 64-bit distance interpolation. For 32-bit interpolation, no merge instructions are required. Instead, two 32-bit adds can be performed simultaneously by the 64-bit add instruction.

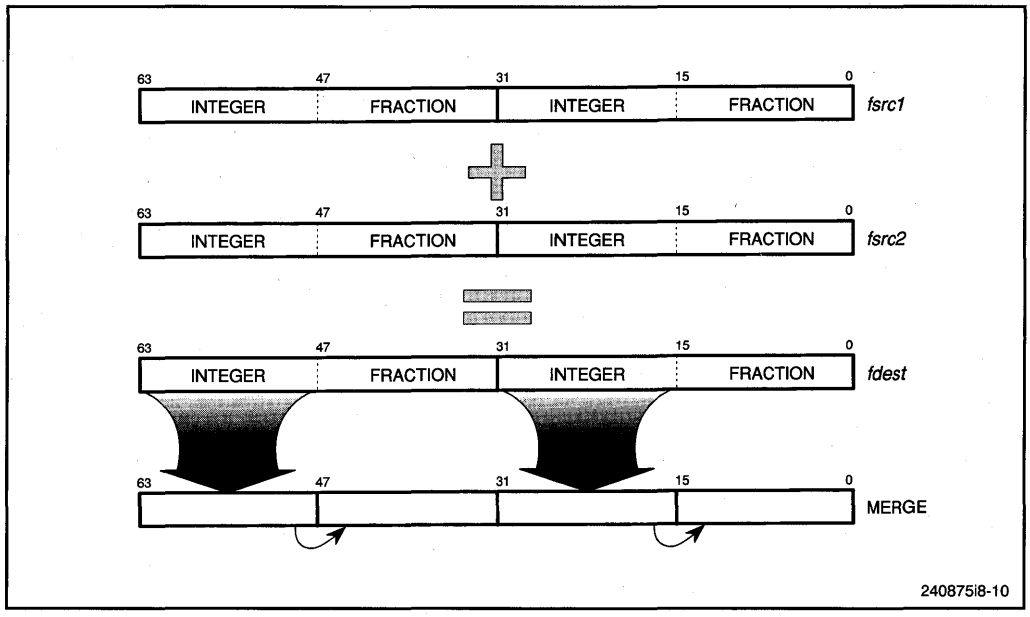


Figure 8-10. FADDZ with 16-Bit Z-Buffer

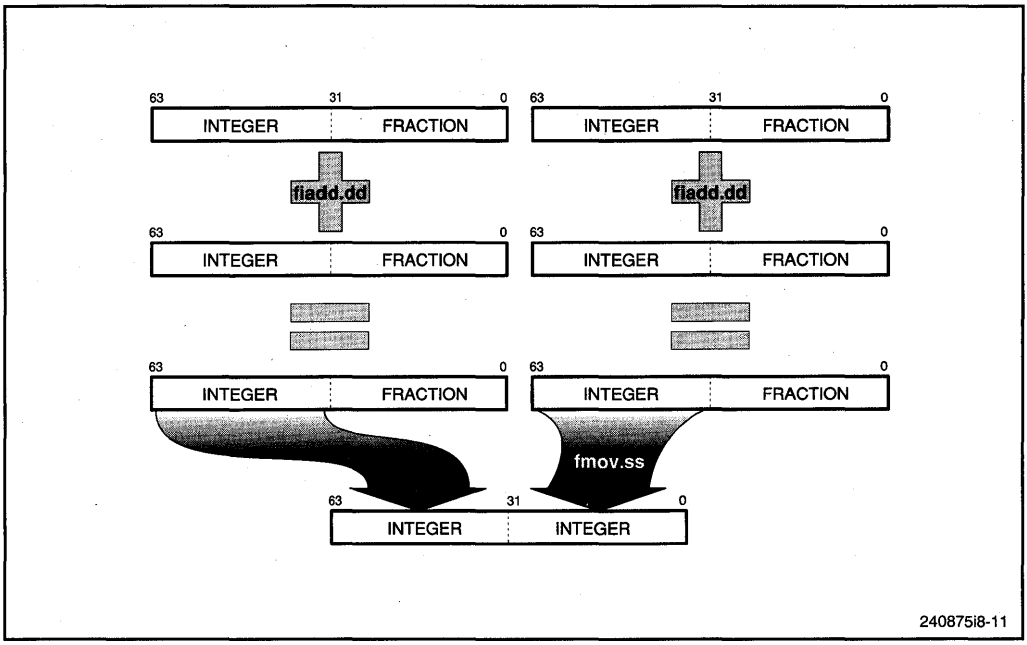


Figure 8-11. 64-Bit Distance Interpolation

For 32-bit Z-buffers, 64-bit distance interpolation is implemented (as Figure 8-11 shows) with two 64-bit **fiadd** instructions. The merging is implemented with the 32-bit move **fmov.ss** *fsrc1*, *fdest*.

Programming Notes

When interpolating with a negative slope, one pixel's most-significant bit may carry into the least-significant bit of the fraction of the neighboring pixel on the left. The carry is due to the fact that the **faddz** instruction does not treat pixels individually; instead, it adds their operands exactly as **fiadd.dd** does—in one 64-bit operation. (The only difference between **fiadd.dd** and **faddz** is the effect on the MERGE register.) Interpolation algorithms should compensate for the carry.

8.5.2.4 OR WITH MERGE REGISTER

form *fsrc1, fdest* **OR with MERGE Register**

fdest ← *fsrc1* OR MERGE (64 bits)
MERGE ← 0

pform *fsrc1, fdest* **Pipelined OR with MERGE Register**

fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* OR MERGE (64 bits)
MERGE ← 0

For intensity interpolation, the **form** instruction fetches the partially completed pixels from the MERGE register, sets any additional bits that may be needed in the pixels (e.g., texture values), and loads the result into a floating-point register. *Fsrc1* (when a register) and *fdest* are floating-point register pairs; the *fsrc2* field of the instruction should contain zero.

For distance interpolation or for intensity interpolation that does not require further modification of the value in the MERGE register, the *fsrc1* operand of **form** may be **f0**, thereby causing the instruction to simply load the contents of the MERGE register into a floating-point register.

8.5.3 Transfer F-P to Integer Register

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

The bit pattern in the 32-bit floating-point register *fsrc1* is stored into the 32-bit integer register *idest*. Assemblers and compilers should encode *fsrc2* as **f0**.

Programming Notes

This scalar instruction is performed by the graphics unit. When it is executed, the result in the graphics-unit pipeline is lost. However, executing this instruction does not impact performance, even if the next instruction is a pipelined operation whose *fdest* is nonzero (refer to Section 8.1).

For best performance, *idest* should not be referenced in the next instruction, and *fsrc1* should not reference the result of the prior instruction if the prior instruction is scalar.

8.6 DUAL-INSTRUCTION MODE

i860 microprocessors can execute a floating-point and a core instruction in parallel. Such parallel execution is called **dual-instruction mode**. When executing in dual-instruction mode, the instruction sequence consists of 64-bit aligned instruction pairs with a floating-point instruction in the lower 32 bits and a core instruction in the upper 32 bits.

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 processor 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 processor 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 8-12 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.

The recommended floating-point NOP for dual-instruction mode is **shrd r0,r0,r0**, because this instruction does not affect the states of the floating-point pipelines. Even though this is a core instruction, bit 9 is interpreted as the dual-instruction mode control bit. In assembly language, this instruction is specified as **fnop** or **d.fnop**. Traps are not reported on **fnop**. Because it is a core instruction, **d.fnop** cannot be used to initiate entry into dual-instruction mode.

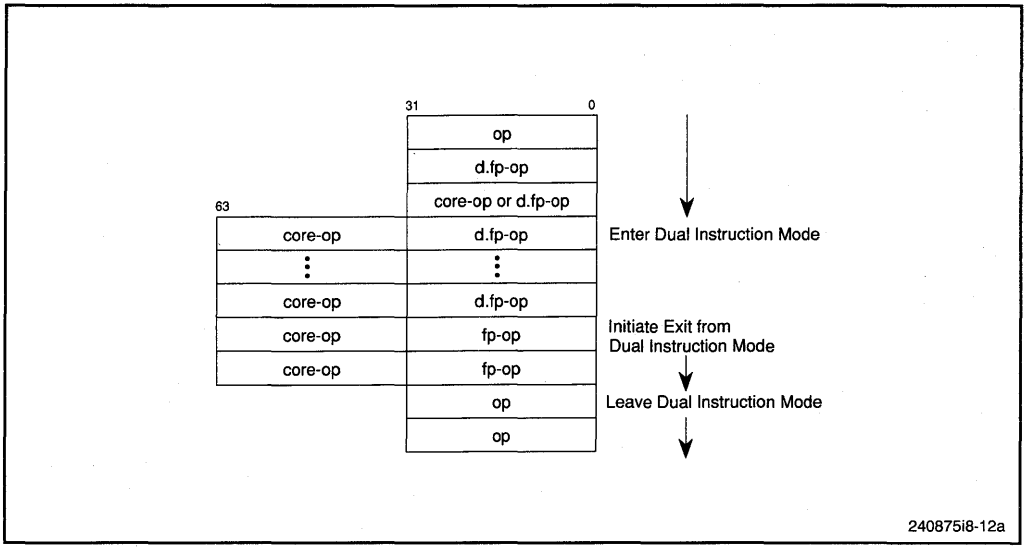


Figure 8-12. Dual-Instruction Mode Transitions (1 of 2)

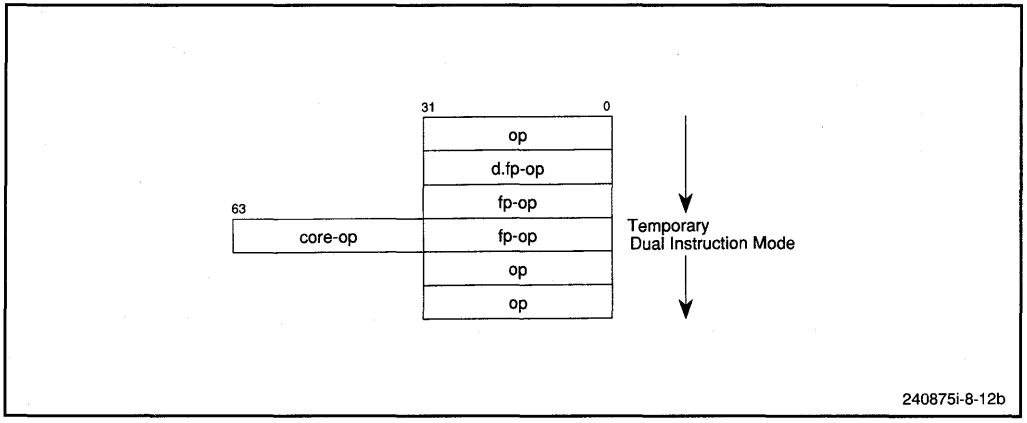


Figure 8-12. Dual-Instruction Mode Transitions (2 of 2)

8.6.1 Core and Floating-Point Instruction Interaction

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 the **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 (or when an IT trap occurs on the i860 XR microprocessor due to **ldio, stio, scyc, ldint,** or **pfld.q**), the floating-point companion instruction has completely finished execution before the trap is taken.

8.6.2 Dual-Instruction Mode Restrictions

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)fzchkl**.

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**, **ldio**, **ldint**, **ld.c**, **st**, **stio**, **st.c**, **call**, **calli**, **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 same register as the companion floating-point instruction unless the destination is **f0** or **f1**. 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.q    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 the lock protocol, a transition to or from dual-instruction mode is not permitted.

Traps and Interrupts (80860XR)

9

CHAPTER 9
TRAPS AND INTERRUPTS (80860XR)

Traps are caused either 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 9-1.

9.1 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 as described in Section 9.1.3. 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).

Table 9-1. Types of Traps (80860XR)

Type	Indication			Caused by	
	psr	epsr	fsr	Condition	Instruction
Instruction Fault	IT	OF IL		Software traps Missing **unlock**	**intovr** **trap, scyc, ldio, stio, ldint, pfld.q** 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, **pst, fst,** and sometimes **fld, pfld,** and **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 signal on INT pin	
Reset	None			Hardware RESET signal	

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

4. Sets IM to zero (interrupts disabled). This guards against further interrupts until the trap information can be saved.

5. If the processor is in dual instruction mode, it sets DIM; otherwise DIM is cleared.

6. DS is set under either of the following conditions:
   - The processor is in single-instruction mode and the next instruction will be executed in dual-instruction mode.
   - The processor is in dual-instruction mode and the next instruction will be executed in single-instruction mode.

7. The appropriate trap type bits in **psr** and **epsr** are set (IT, IN, IAT, DAT, FT, IL, OF). 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.

9. Clears the BL bit of **dirbase** and deasserts LOCK#.

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

9.1.1 Saving State

To support nesting of traps, the trap handler must save the current state before another trap occurs. An interrupt stack can be implemented in software (refer to the section on stack implementation in Chapter 11). Interrupts can then be reenabled by clearing the trap-type bits and setting IM to the value of PIM. The branch-indirect instruction is sensitive to the trap-type bits; therefore, clearing the trap-type bits allows normal indirect branches to be performed within the trap handler.

The items that make up the current state may include any of the following:

1. The **fir**.

2. The **psr**.

3. The **epsr**.
4. The **fsr**.
5. The **dirbase** register.
6. The MERGE register.
7. The KR, KI, and T registers.
8. Any of the four pipelines (refer to Section 9.8).
9. The floating-point and integer register files.

While the floating-point registers are being saved, the FTE bit of the **fsr** must be temporarily cleared, so that no floating-point traps are triggered. FTE must be restored to its original value before returning from the trap handler.

9.1.2 Inside the Trap Handler

While most activities of trap handlers are application dependent (and, therefore, are beyond the scope of this manual), programmers should be aware of the following requirements that are imposed by the i860 microprocessor architecture:

1. For all types of traps, the trap handler must check the IL bit of **epsr** to determine if a locked sequence is being interrupted.
2. The trap handler must execute **ld.c fir**, *rdest* once for each trap. Failure to do so prevents **fir** from receiving the address of the next trap.
3. When the interrupted program is in dual-instruction mode, KNF may be set upon entry to the trap handler. The handler must clear KNF (after saving its former value) before it executes a floating-point instruction; otherwise, that floating-point instruction would be killed.

9.1.3 Returning from the Trap Handler

Returning from a trap handler involves the following steps.

1. Restoring the pipeline states, including the **fsr**, KR, KI, T, and MERGE registers, where necessary.
2. Subtracting *src1* from *src2*, when a data-access fault occurred on an autoincrementing load/store instruction and a floating-point trap did not also occur.
3. Determining where to resume execution by inspecting the instruction at **fir** − 4. The details for this determination are given in Section 9.1.3.1.

4. Restoring the integer and floating-point register files (except for the register that holds the resumption address).

5. Updating the **psr** with the value to be used after return. It may be necessary to set the KNF bit in the **psr**. The requirements for KNF are given in Section 9.1.3.2. The trap handler must ensure that no trap occurs between the **st.c** to the **psr** and the indirect branch that exits the trap handler.

6. Executing an indirect branch to the resumption address, making sure that at least one of the trap bits is set in the **psr**. Neither the indirect branch nor the following instruction may be executed in dual-instruction mode.

7. Restoring the register that holds the resumption address. (This is executed before the delayed indirect branch is completed.)

Once restoration of the initial state has begun, the trap handler must ensure that no trap occurs before returning to the interrupted procedure.

9.1.3.1 DETERMINING WHERE TO RESUME

To determine where to resume execution upon leaving the trap handler, the trap handler should normally examine the instruction at **fir** − 4 to determine whether the instruction at that address is a delayed control-transfer instruction (i.e., one that executes the next sequential instruction on branch taken). However, examining **fir** − 4 may cause a page fault. If the location in **fir** is at the beginning of a page, then **fir** − 4 is in the prior page. If the prior page is not present, then examining **fir** − 4 will cause a page fault. In this case, however, the instruction at **fir** − 4 could not have been a delayed control-transfer instruction, and it is not necessary to examine **fir** − 4. Note that, when determining whether the prior page is present, it is necessary to inspect the P (present) bit in both the page table and its page directory entry.

If the instruction at **fir** − 4 is not a delayed control-transfer instruction, then execution normally resumes at the address in **fir**. However, if the trap was caused by a **trap** instruction, execution should resume at the address in **fir** + 4 in single-instruction mode or at the address in **fir** + 8 in dual-instruction mode.

If, on the other hand, the instruction at **fir** − 4 is a delayed control-transfer instruction, execution normally resumes at **fir** − 4 so that the control-transfer instruction (which did not finish because of the trap) is also reexecuted. If the instruction at **fir** − 4 is a **bla** instruction, then *src1* should be subtracted from *src2* before reexecuting the **bla**.

The one variance from this strategy occurs when the instruction at **fir** − 4 is a conditional delayed branch (**bc.t** or **bnc.t**), the instruction at **fir** is a **pfgt**, **pfle**, or **pfeq**, and a source exception has occurred. To implement the IEEE standard for unordered compares, the trap handler may need to change the value of CC. In this case it cannot resume at **fir** − 4, because the new value of CC might cause an incorrect branch. Instead, the trap handler must interpret the conditional branch instruction and resume at its target.

If the processor was in dual-instruction mode and execution is to resume at **fir** − 4, the trap handler should set DS and clear DIM in the **psr** before resuming execution of the interrupted procedure. Clearing DIM prevents the floating-point instruction associated with the control-transfer instruction at **fir** − 4 from being reexecuted. Setting DS forces the processor back to dual-instruction mode after executing the control-transfer instruction.

Every code section should begin with a **nop** instruction so that **fir** − 4 is defined even in case a trap occurs on the first real instruction of the code section. Furthermore, this **nop** should not be the target of any branch or call.

9.1.3.2 SETTING KNF

The trap handler should set the KNF bit of **psr** if the trapped instruction is a floating-point instruction that should not be reexecuted; otherwise, KNF is left unchanged. Floating-point instructions should not be reexecuted under either of the following conditions:

- The trap was caused in dual-instruction mode by a data-access fault or an **intovr** instruction and there are no other trap conditions. In this case, the floating-point instruction has already been executed.
- The trap was caused by a source exception on any floating-point instruction (except when a **pfgt**, **pfle**, or **pfeq** follows a conditional branch, as already explained in Section 9.1.3.1). The trap handler determines the result that corresponds to the exceptional inputs; therefore, the instruction should not be reexecuted.

9.2 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. Note that when **trap** is executed in dual-instruction mode, the floating-point companion of the **trap** instruction is not executed before the trap is taken. 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** instruction by returning to user code at the address in **fir** + 8. In this case, the trap handler must emulate the companion 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.

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. Refer to the **intovr** instruction in Chapter 7. When **intovr** causes a trap in dual-instruction mode, the floating-point companion of the **intovr** instruction has completely finished execution before the trap is taken.

3. By violation of the lock/unlock protocol explained in Chapter 7. In this case, IL is also set, and the instruction pointed to by **fir** may or may not have been executed.

4. By executing an instruction implemented only in the i860 XP microprocessor. Execution of **ldio, stio, scyc, ldint,** or **pfld.q** on the i860 XR microprocessor causes an instruction trap.

The **trap** and **intovr** instructions must not be used within a locked sequence.

To distinguish between cases 1 and 2, the trap handler must examine the instruction addressed by **fir**.

9.3 FLOATING-POINT FAULT

The floating-point faults of i860 microprocessors support the floating-point exceptions defined by the IEEE standard as well as some other useful classes of exceptions. The i860 microprocessors divide these into two classes:

1. **Source exceptions.** This class includes:
   - All the invalid operations defined by the IEEE standard (including operations on signaling NaNs).
   - Division by zero.
   - Operations on quiet NaNs, denormals and infinities. (These data types are implemented by software.)

2. **Result exceptions.** This class includes the overflow, underflow, and inexact exceptions defined by the IEEE standard.

Software supplied by Intel provides the IEEE standard default handling for all these exceptions.

Floating-point faults are reported only on floating-point instructions, and on **pst, fst, fld, pfld,** and **ixfr**.

No floating-point fault occurs when **pst, fst, fld, pfld,** or **ixfr** transfers an operand that is not a valid floating-point value.

9.3.1 Source Exception Faults

When used as inputs to the floating-point adder or multiplier, 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 trap handler can reference both source operands and the operation by decoding the instruction specified by **fir**.

In the case of dual operations, the trap handler has to determine which special registers the source operands are stored in and inspect all four source operands to see if one or both operations need to be fixed up. It can then compute the appropriate result and store the result in *fdest*, in the case of a scalar operation, or replace the appropriate first-stage result, in the case of a pipelined operation.

Note that, in the following sequence, inappropriate use of the FTE bit of the **fsr** can produce an invalid operand that does not cause a source exception:

1. Floating-point traps are masked by clearing the FTE bit.
2. A dual-operation instruction causes underflow or overflow leaving an invalid result in the T register.
3. Floating-point traps are enabled by setting the FTE bit.
4. The invalid result in the T register is used as an operand of a subsequent instruction.

Even though the result of an operation would normally cause a source exception, it can be inserted into the pipeline as follows:

1. Disable traps by clearing FTE.
2. Perform a pipelined add of the value with zero or a multiply by one.
3. Set the result-status bits of **fsr** to "normal" by loading **fsr** with the U-bit set and zeros in the appropriate unit's result-status bits. The other unit's status must be set to the saved status for the first pipeline stage.
4. Reenable traps by setting FTE.
5. Set KNF in the **psr** to avoid reexecuting the instruction.

The trap handler should ignore the SE bit 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. The SE value is undefined in this case.

The trap handler should process result exceptions as described below and reexecute the instruction before processing source exceptions.

9.3.2 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 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 finite number in the destination format.
- Inexact result (when TI is set). The result is not exactly representable in the destination format. For example, the fraction 1/3 cannot be precisely represented in binary form. This exception occurs frequently and indicates that some (generally acceptable) accuracy has been lost.

The point at which a result exception is reported depends upon whether it is caused by a pipelined operation:

- **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. The result is also written to the register indicated by the RR field of the **psr**.
- **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.

To define the cases in which the instructions **fld**, **pfld**, and **ixfr** report exceptions, let *A* be any floating-point instruction that causes a result exception, and let *B* be **fld**, **pfld**, or **ixfr**, the next floating-point instruction executed after *A* after any number of intervening non-floating-point instructions.

- If the *fdest* of *B* overlaps with the *fdest* of *A*, then *B* always traps.
- If the *fdest* of *B* does not overlap with the *fdest* of *A*, then:
  - If *A* finishes executing before *B* executes, then *B* traps.
  - If *A* does not finish executing before *B* executes, then *B* does not trap.

To calculate the time for *A* to execute, refer to the instruction timings listed in Appendix C.

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 **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, RM, and FZ bits are not changed.

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

For both scalar and pipelined modes, if a result exception occurs, the trap handler must calculate the desired result. In either mode, the result supplied by the CPU has the same mantissa as the true result and has an exponent which is the low-order bits of the true result. The trap handler can inspect the supplied result, calculate the result appropriate for that instruction (a NaN or an infinity, for example), and store the calculated result. The trap handler must store the calculated result in the register specified by RR (if nonzero) or (if RR = 0) must load the calculated result into the last stage of the pipeline in place of the saved result.

Adder overflow can occur due either to a true floating-point operation (for example, **pfadd.p** or **pfeq.p**) or to an integer conversion operation (**fix.v, pfix.v, ftrunc.v, pftrunc.v**). For a true floating-point operation, the exponent of the result will be all ones. For an integer conversion operation, the exponent of the result will be less than all ones. When adder overflow occurs, the trap handler can distinguish between the two cases by examining the exponent of the result.

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.

9.4 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 directory or 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 will occur 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 will occur. No protection violation results, because the processor does not permit data accesses to supervisor pages while running in user mode.

9.5 DATA-ACCESS FAULT

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

- An attempt is being made to write to a page whose D-bit is clear.
- A memory operand is misaligned (is not located at an address that is a multiple of the length of the data).
- The address stored in the **db** (data breakpoint) register is equal to one of the addresses spanned by the operand.
- The operand is in a not-present page.
- A memory access is being attempted in violation of the memory protection scheme defined in Chapter 4.
- A-bit is zero during address translation within a locked sequence.

When a data-access trap occurs and the next instruction is a pipelined floating-point instruction, the destination register of the pipelined floating-point instruction may be partially updated. This condition only affects debuggers, not applications software. A debugger should somehow indicate that the contents of that register are invalid. Correct execution will occur when the trap handler resumes execution after handling the data-access trap, because the pipelined floating-point instruction will then correctly update its destination register.

9.6 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.

9.7 RESET TRAP

When the i860 XR microprocessor is reset, execution begins in single-instruction mode at virtual 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 **psr** trap bits are set.

Table 9-2 shows the initial settings of all registers and caches.

Software must ensure that the data cache is flushed (refer to Chapter 5) and control registers are properly initialized before performing operations that depend on the values of the cache or registers. The **fir** must be initialized with an **ld.c fir, r0** instruction.

Table 9-2. Register and Cache Values after Reset (80860XR)

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	All entries invalid
Data Cache	*Undefined.* All modified bits = 0.
TLB	All entries invalid

Reset code must initialize the floating-point pipeline states and the KR, KI, and T registers to zero, using dummy pipelined instructions. Floating-point traps must be 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. (Refer to the **bri** instruction in Chapter 7.)

9.8 PIPELINE PREEMPTION

Each of the four pipelines (adder, multiplier, load, graphics) contains state information. The pipeline state must be saved when a process is preempted or when a trap handler performs pipelined operations using the same pipeline. The state must be restored when resuming the interrupted code.

9.8.1 Floating-Point Pipelines

The floating-point pipeline state consists of the following items:

1. The current contents of the floating-point status register **fsr** (including the third-stage result status).

2. Unstored results from the first, second, and third stages. The number of stages that exist in the multiplier pipeline depends on the sizes of the operands that occupy the pipeline. The MRP bit of **fsr** helps determine how many stages are in the multiplier pipeline.

3. The result-status bits for the first two stages.

4. The contents of the KR, KI, and T registers.

While the floating-point pipelines are being saved and restored, the FTE bit of the **fsr** must be temporarily cleared, so that no floating-point traps are triggered. FTE must be restored to its original value before returning from the trap handler.

9.8.2 Load Pipeline

The pipeline state for **pfld** instructions can be saved by performing three **pfld** instructions to a dummy address. Thus, the pipeline is advanced three stages, causing the last three real operands to be stored from the pipeline into registers that are then saved in some memory area. The size of each saved value is indicated by the value of the LRP bit of the **fsr**. Note that, when changing between big and little endian modes, the load pipeline must be saved *before* changing the BE bit.

The load pipeline can be restored performing three **pfld** instructions using the memory addresses of the saved values. The pipeline will then contain the same three values it held before the preemption.

9.8.3 Graphics Pipeline

The graphics pipeline has only one stage. To flush the pipeline, execute a **pfiadd f0, f0,** *fdest*. The only other state information for the graphics unit resides in the PM bits of **psr**, the IRP bit of the **fsr**, and in the MERGE register. Store the MERGE register with a **form** instruction. Restore the MERGE register by using **faddz** instructions.

Traps and Interrupts (80860XP)

10

CHAPTER 10
TRAPS AND INTERRUPTS (80860XP)

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 10-1.

10.1 TRAP HANDLER INVOCATION

This section applies to traps other than reset. When a trap occurs, execution of the current instruction is aborted. Except for bus and parity errors, the instruction is restartable as described in Section 10.1.4. 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).

Table 10-1. Types of Traps (80860XP)

Type	Indication			Caused by	
	psr	epsr	fsr	Condition	Instruction
Instruction Fault	IT	OF IL PT & PI		Software traps Missing **unlock** Pipeline usage	**intovr** **trap** Any Any scalar or pipelined instruction that uses a pipeline
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, **pst, fst,** and sometimes **fld, pfld,** and **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
Parity Error Fault	IN	PEF		Parity error on data pins during bus read operation when PEN# pin active	
Bus Error Fault	IN	BEF		External interrupt signal on BERR pin	
Interrupt	IN	INT		External interrupt signal on INT pin	
Reset	None	PEF, BEF		Hardware RESET signal	

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

4. Sets IM to zero (interrupts disabled). This guards against further interrupts until the trap information can be saved.

5. If the processor is in dual instruction mode, it sets DIM; otherwise DIM is cleared.

6. DS is set under either of the following conditions:
   - The processor is in single-instruction mode and the next instruction will be executed in dual-instruction mode.
   - The processor is in dual-instruction mode and the next instruction will be executed in single-instruction mode.

   Otherwise, it is cleared.

7. The appropriate trap type bits in **psr** and **epsr** are set (IT, IN, IAT, DAT, FT, IL, OF, PI, AI, DI, PT, BEF, PEF, BS). Several bits may be set if the corresponding trap conditions occur simultaneously. If PEF or BEF is set, the processor places the bus address in **bear**.

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. To determine the cause or causes of the trap, the trap handler must examine the trap-type bits in **psr** (IT, IN, IAT, DAT, FT) and **epsr** (IL, PI, PT, BEF, PEF) as well as the instruction addressed by **fir**.

10.1.1 Saving State

To support nesting of traps, the trap handler must save the current state before another trap occurs. An interrupt stack can be implemented in software (refer to the section on stack implementation in Chapter 11). Interrupts can then be reenabled by clearing the trap-type bits and setting IM to the value of PIM. The branch-indirect instruction is sensitive to the trap-type bits; therefore, clearing the trap-type bits allows normal indirect branches to be performed within the trap handler.

The items that make up the current state may include any of the following:

1. The **fir**.

2. The **psr**.

3. The **epsr**.
4. The **fsr**.
5. The **dirbase** register.
6. The MERGE register.
7. The KR, KI, and T registers.
8. Any of the four pipelines (refer to Section 10.10).
9. The floating-point and integer register files.
10. The privileged registers **p0, p1, p2, p3**.
11. **bear**
12. **ccr**

Items not used by the system or not altered by the trap handler need not be saved. Refer also to Section 10.10.4 for a mechanism that the i860 XP microprocessor provides to help avoid unnecessarily saving the pipelines.

While the floating-point registers are being saved, the FTE bit of the **fsr** must be temporarily cleared, so that no floating-point traps are triggered. FTE must be restored to its original value before returning from the trap handler.

10.1.2 Inside the Trap Handler

While most activities of trap handlers are application dependent (and, therefore, are beyond the scope of this manual), programmers should be aware of the following requirements that are imposed by the i860 microprocessor architecture:

1. For all types of traps, the trap handler must check the IL bit of **epsr** to determine if a locked sequence is being interrupted. If so, the trap handler must execute a load or store operation to unlock the bus (no **unlock** instruction is required), and must restart at the beginning of the locked sequence instead of at the trapping instruction. (Refer to the **lock** instruction in Chapter 7.)

2. The trap handler must execute **ld.c fir**, *rdest* once for each trap. Failure to do so prevents **fir** from receiving the address of the next trap.

3. When the interrupted program is in dual-instruction mode, KNF may be set upon entry to the trap handler. The handler must clear KNF (after saving its former value) before executing a floating-point instruction; otherwise, that floating-point instruction would be killed.

10.1.3 Fatal Errors

When a bus error trap (BEF bit of **epsr** set) or parity error trap (PEF bit of **epsr** set) occurs, the interrupted instruction is not restartable. The BS bit of **epsr** indicates whether the trap occurred while in supervisor mode, in which case the operating system should reboot, or in user mode, in which the operating system should discontinue the interrupted task.

10.1.4 Returning from the Trap Handler

If the trap is not due to a bus or parity error and no locked sequence was interrupted, the interrupted program is restartable. Returning from a trap handler involves the following steps.

1. Restoring the pipeline states, including the **fsr**, KR, KI, T, and MERGE registers, where necessary.

2. Subtracting *src1* from *src2*, when a data-access fault occurred on an autoincrementing load/store instruction and a floating-point trap did not also occur. The AI bit of **epsr** indicates when an autoincrementing instruction causes a data-access trap; the handler must determine whether a floating-point trap occurred.

3. Determining where to resume execution by inspecting the DI bit of **epsr** and possibly the instruction at **fir** − 4. The details for this determination are given in Section 10.1.4.1.

4. Restoring the integer and floating-point register files (except for the register that holds the resumption address).

5. Updating the **psr** with the value to be used after return. It may be necessary to set the KNF bit in the **psr**. The requirements for KNF are given in Section 10.1.4.2. The trap handler must ensure that no trap occurs between the **st.c** to the **psr** and the indirect branch that exits the trap handler.

6. Executing an indirect branch to the resumption address, making sure that at least one of the trap bits is set in the **psr**. Neither the indirect branch nor the following instruction may be executed in dual-instruction mode.

7. Restoring the register that holds the resumption address. (This is executed before the delayed indirect branch is completed.)

Once restoration of the initial state has begun, the trap handler must ensure that no trap occurs before returning to the interrupted procedure.

10.1.4.1 DETERMINING WHERE TO RESUME

To determine where to resume execution upon leaving the trap handler, the trap handler should first examine the DI bit of **epsr**. DI lets the trap handler avoid unnecessary interpretation of the instruction at **fir** − 4.

If DI is clear, the instruction at **fir** − 4 is not a delayed control-transfer instruction, and execution normally resumes at the address in **fir**. However, if the trap was caused by a **trap** instruction, execution should resume at the address **fir** + 4 in single-instruction mode or at the address **fir** + 8 in dual-instruction mode.

If DI is set, the instruction at **fir** − 4 is a delayed control-transfer instruction (i.e., one that executes the next sequential instruction on branch taken), and the trap handler must interpret that instruction to distinguish among these cases:

1. The instruction at **fir** − 4 is a conditional delayed branch (**bc.t** or **bnc.t**), the instruction at **fir** is a **pfgt**, **pfle**, or **pfeq**, and a source exception has occurred. To implement the IEEE standard for unordered compares, the trap handler may need to change the value of CC. In this case it cannot resume at **fir** − 4, because the new value of CC might cause an incorrect branch. Instead, the trap handler must interpret the conditional branch instruction and resume at its target.

2. All other cases. Execution resumes at **fir** − 4 so that the control-transfer instruction (which did not finish because of the trap) is also reexecuted. If the instruction at **fir** − 4 is a **bla** instruction, then *src1* should be subtracted from *src2* before reexecuting the **bla**.

If the processor was in dual-instruction mode and execution is to resume at **fir** − 4, the trap handler should set DS and clear DIM in the **psr** before resuming execution of the interrupted procedure. Clearing DIM prevents the floating-point instruction associated with the control-transfer instruction at **fir** − 4 from being reexecuted. Setting DS forces the processor back to dual-instruction mode after executing the control-transfer instruction.

10.1.4.2 SETTING KNF

The trap handler should set the KNF bit of **psr** if the trapped instruction is a floating-point instruction that should not be reexecuted; otherwise, KNF is left unchanged. Floating-point instructions should not be reexecuted under either of the following conditions:

- The trap was caused in dual-instruction mode by a data-access fault or an **intovr** instruction and there are no other trap conditions. In this case, the floating-point instruction has already been executed.
- The trap was caused by a source exception on any floating-point instruction (except when a **pfgt**, **pfle**, or **pfeq** follows a conditional branch, as already explained in Section 10.1.4.1). The trap handler determines the result that corresponds to the exceptional inputs; therefore, the instruction should not be reexecuted.

10.2 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. Note that when **trap** is executed in dual-instruction mode, the floating-point companion of the **trap** instruction is not executed before the trap is taken. 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** instruction by returning to user code at the address in **fir** + 8. In this case, the trap handler must emulate the companion 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.

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. Refer to the **intovr** instruction in Chapter 7. When **intovr** causes a trap in dual-instruction mode, the floating-point companion of the **intovr** instruction has completely finished execution before the trap is taken.

3. By violation of the lock/unlock protocol explained in Chapter 7. In this case, IL is also set, and the instruction pointed to by **fir** may or may not have been executed.

4. By execution of an instruction that uses a pipeline when the PT bit of **epsr** is set.

The **trap** and **intovr** instructions must not be used within a locked sequence.

To distinguish between cases 1 and 2, the trap handler must examine the instruction addressed by **fir**.

10.3 FLOATING-POINT FAULT

The floating-point faults of i860 microprocessors support the floating-point exceptions defined by the IEEE standard as well as some other useful classes of exceptions. The i860 microprocessors divide these into two classes:

1. **Source exceptions.** This class includes:
   - All the invalid operations defined by the IEEE standard (including operations on signaling NaNs).
   - Division by zero.
   - Operations on quiet NaNs, denormals and infinities. (These data types are implemented by software.)

2. **Result exceptions.** This class includes the overflow, underflow, and inexact exceptions defined by the IEEE standard.

Software supplied by Intel provides the IEEE standard default handling for all these exceptions.

Floating-point faults are reported only on floating-point instructions, and on **pst**, **fst**, **fld**, **pfld**, and **ixfr**.

No floating-point fault occurs when **pst**, **fst**, **fld**, **pfld**, or **ixfr** transfers an operand that is not a valid floating-point value.

10.3.1 Source Exception Faults

When used as inputs to the floating-point adder or multiplier, 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 trap handler can reference both source operands and the operation by decoding the instruction specified by **fir**.

In the case of dual operations, the trap handler has to determine which special registers the source operands are stored in and inspect all four source operands to see if one or both operations need to be fixed up. It can then compute the appropriate result and store the result in *fdest*, in the case of a scalar operation, or replace the appropriate first-stage result, in the case of a pipelined operation.

Note that, in the following sequence, inappropriate use of the FTE bit of the **fsr** can produce an invalid operand that does not cause a source exception:

1. Floating-point traps are masked by clearing the FTE bit.
2. An dual-operation instruction causes underflow or overflow leaving an invalid result in the T register.
3. Floating-point traps are enabled by setting the FTE bit.
4. The invalid result in the T register is used as an operand of a subsequent instruction.

Even though the result of an operation would normally cause a source exception, it can be inserted into the pipeline as follows:

1. Disable traps by clearing FTE.
2. Perform a pipelined add of the value with zero or a multiply by one.

3. Set the result-status bits of **fsr** to "normal" by loading **fsr** with the U-bit set and zeros in the appropriate unit's result-status bits. The other unit's status must be set to the saved status for the first pipeline stage.

4. Reenable traps by setting FTE.

5. Set KNF in the **psr** to avoid reexecuting the instruction.

The trap handler should ignore the SE bit 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. The SE value is undefined in this case.

The trap handler should process result exceptions as described below and reexecute the instruction before processing source exceptions.

10.3.2 Result Exception Faults

The result exceptions include:

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

The point at which a result exception is reported depends upon whether it is caused by a pipelined operation:

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

To define the cases in which the instructions **fld**, **pfld**, and **ixfr** report exceptions, let *A* be any floating-point instruction that causes a result exception, and let *B* be **fld**, **pfld**, or **ixfr**, the next floating-point instruction executed after *A* after any number of intervening non-floating-point instructions.

- If the *fdest* of *B* overlaps with the *fdest* of *A*, then *B* always traps.
- If the *fdest* of *B* does not overlap with the *fdest* of *A*, then:
  - If *A* finishes executing before *B* executes, then *B* traps.
  - If *A* does not finish executing before *B* executes, then *B* does not trap.

To calculate the time for *A* to execute, refer to the instruction timings listed in Appendix C.

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 **pfadd** is started, a trap occurs and MO is set.

For scalar operations, the RR bits of **fsr** report in which register the result was stored. RR is updated when the scalar instruction is initiated. The result exception 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, RM, and FZ bits are not changed.

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

For both scalar and pipelined modes, if a result exception occurs, the trap handler must calculate the desired result. In either mode, the result supplied by the CPU has the same mantissa as the true result and has an exponent which is the low-order bits of the true result. The trap handler can inspect the supplied result, calculate the result appropriate for that instruction (a NaN or an infinity, for example), and store the calculated result. If RR is nonzero, the trap handler must store the calculated result in the register specified by RR; if RR is zero, it must load the calculated result into the last stage of the pipeline in place of the saved result.

Adder overflow can occur due either to a true floating-point operation (for example, **pfadd.p** or **pfeq.p**) or to an integer conversion operation (**fix.v, pfix.v, ftrunc.v, pftrunc.v**). For a true floating-point operation, the exponent of the result will be all ones. For an integer conversion operation, the exponent of the result will be less than all ones. When adder overflow occurs, the trap handler can distinguish between the two cases by examining the exponent of the result.

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.

10.4 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 directory or 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 will occur 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 will occur. No protection violation results, because the processor does not permit data accesses to supervisor pages while running in user mode.

10.5 DATA-ACCESS FAULT

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

- An attempt is being made to write to a page whose D-bit is clear.
- A memory operand is misaligned (is not located at an address that is a multiple of the length of the data).
- The address stored in the **db** (data breakpoint) register is equal to one of the addresses spanned by the operand.
- The operand is in a not-present page.
- A memory access is being attempted in violation of the memory protection scheme defined in Chapter 4.
- A-bit is zero during address translation within a locked sequence.

When a data-access trap occurs and the next instruction is a pipelined floating-point instruction, the destination register of the pipelined floating-point instruction may be partially updated. This condition only affects debuggers, not applications software. A debugger should somehow indicate that the contents of that register are invalid. Correct execution will occur when the trap handler resumes execution after handling the data-access trap, because the pipelined floating-point instruction will then correctly update its destination register.

10.6 PARITY ERROR TRAP

If the PEN# pin is active and the bus unit detects a parity error during a bus read operation, the processor sets PEF and generates a trap. Further parity error traps are masked as soon as PEF is set. To reenable such traps, software must clear PEF and unfreeze BEAR by executing **ld.c bear,** *rdest*.

BS (bus or parity error trap in supervisor mode) is set by the i860 XP microprocessor when a parity error occurs while the processor is in supervisor mode. The operating system can use this bit to decide, for example, whether to abort the process (user mode) or reboot the system (supervisor mode).

10.7 BUS ERROR TRAP

When external hardware asserts the BERR pin, the processor sets BEF (bus error flag) and traps. Further BERR traps are masked as soon as BEF is set by hardware. To reenable such traps, software must clear BEF and unfreeze BEAR by executing **ld.c bear,** *rdest*.

BS (bus or parity error trap in supervisor mode) is set by the i860 XP microprocessor when a bus error occurs while the processor is in supervisor mode. The operating system can use this bit to decide, for example, whether to abort the process (user mode) or reboot the system (supervisor mode).

10.8 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.

10.9 RESET TRAP

When the i860 XP microprocessor is reset, execution begins in single-instruction mode at virtual 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 **psr** trap bits are set.

Table 10-2 shows the initial settings of all registers and caches.

Software must ensure that control registers are properly initialized before performing operations that depend on the values of the registers. The **fir** must be initialized with a **ld.c fir, r0** instruction. The **bear** must be initialized with a **ld.c bear, r0** instruction.

Reset code must initialize the floating-point pipeline states and the KR, KI, and T registers to zero, using dummy pipelined instructions. Floating-point traps must be disabled to ensure that no spurious floating-point traps are generated.

Table 10-2. Register and Cache Values after Reset (80860XP)

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, PT = 0; BEF, PEF = 1; Processor Type, Stepping Number, DCS, SO are read only; others are *undefined*
db	*Undefined*
dirbase	DPS, BL, LB, ATE = 0; others are *undefined*
fir	*Undefined*
fsr	*Undefined*
bear	*Undefined*
p3–p0	*Undefined*
ccr	CO, DO = 0; others are *undefined*
KR, KI, T, MERGE	*Undefined*
NEWCURR	*Undefined*
STATUS	InLoop, Nested, Detached = 0
Caches	**Initial Value**
Instruction Cache	All entires invalid
Data Cache	All entries invalid
TLB	All entries invalid

After a RESET the i860 XP 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. (Refer to the **bri** instruction in Chapter 7.)

10.10 PIPELINE PREEMPTION

Each of the four pipelines (adder, multiplier, load, graphics) contains state information. The pipeline state must be saved when a process is preempted or when a trap handler performs pipelined operations using the same pipeline. The state must be restored when resuming the interrupted code.

10.10.1 Floating-Point Pipelines

The floating-point pipeline state consists of the following items:

1. The current contents of the floating-point status register **fsr** (including the third-stage result status).

2. Unstored results from the first, second, and third stages. The number of stages that exist in the multiplier pipeline depends on the sizes of the operands that occupy the pipeline. The MRP bit of **fsr** helps determine how many stages are in the multiplier pipeline.

3. The result-status bits for the first two stages.

4. The contents of the KR, KI, and T registers.

While the floating-point pipelines are being saved and restored, the FTE bit of the **fsr** must be temporarily cleared, so that no floating-point traps are triggered. FTE must be restored to its original value before returning from the trap handler.

10.10.2 Load Pipeline

The pipeline state for **pfld** instructions can be saved by performing three **pfld** instructions to a dummy address. Thus, the pipeline is advanced three stages, causing the last three real operands to be stored from the pipeline into registers that are then saved in some memory area. The size of each saved value is indicated by the values of the LRP0 and LRP1 bits of the **fsr**. Note that, when changing between big and little endian modes, the load pipeline must be saved *before* changing the BE bit.

The load pipeline can be restored performing three **pfld** instructions using the memory addresses of the saved values. The pipeline will then contain the same three values it held before the preemption.

10.10.3 Graphics Pipeline

The graphics pipeline has only one stage. To flush the pipeline, execute a **pfiadd f0, f0,** *fdest*. The only other state information for the graphics unit resides in the PM bits of **psr**, the IRP bit of the **fsr**, and in the MERGE register. Store the MERGE register with a **form** instruction. Restore the MERGE register by using **faddz** instructions.

10.10.4 Using PI and PT Bits

The PI and PT bits are provided to help the trap handler avoid unnecessarily saving and restoring the pipelines.

Trap handlers that use PI or PT must initially examine **fsr**. If a pending trap exists—that is, if the FTE (floating-point trap enable) bit is set and any of the floating-point exception bits (AI, AO, AU, MI, MO, MU) is active—the trap handler must save the pipelines. The i860 XP microprocessor may set an **fsr** exception bit before the floating-point trap is reported, and this pending trap relies on information in the pipeline. For example, an external interrupt might invoke the trap handler between the scalar floating-point instruction that produces an overflow and the next floating-point operation—the one that would cause a branch to the trap handler for the floating-point trap.

If no pending trap exists, the handler can follow either of the following two methods:

- **Using both PI and PT:** Upon invocation for any reason, the trap handler saves the state of PI and PT (in **epsr**), but does not save the pipes. If PI is found set (which means that the interrupted code needs the state information currently in the floating-point pipelines), the handler sets PT and clears PI (with the **st.c** instruction), then continues with trap processing. If the pipes are used during trap handling (even by a scalar instruction), a trap will be generated with IT and PI set by hardware. The trap handler may then check PI and PT, and if both are set, clear PT, PI, and IT; save the pipes, set an indication that they were saved; and restart execution from the instruction that caused the trap. At the end of trap handling, the trap handler restores the pipes if they were saved, and restores PI and PT to their values before the trap. This method, which avoids saving and restoring the pipes, assumes that most trap handling sequences do not alter the pipes, and that therefore a trap for PT = 1 will not happen very often.
- **Using only PI:** Another approach is to leave PT = 0, using only the PI bit, which the processor sets each time a pipelined instruction or **pfld** is executed. The trap handler saves PI, saves the pipes if PI is set, sets an indication that they were saved, and clears PI. At the end of trap handling, the trap handler restores the pipes if they were saved, and restores PI to its value before the trap. With this method, the pipes are sometimes saved and restored unnecessarily if the trap handler code does not use the pipes. This method is advised when it is known that the trap handler uses the pipes.

Programming Model 11

CHAPTER 11
PROGRAMMING MODEL

This chapter defines compiler and assembly language conventions for the use of certain aspects of the i860 architecture. These standards must be followed to guarantee that compilers, applications programs, and operating systems written by different people and organizations will work together. The conventions here implement the proposed application binary interface (ABI) as defined in the *Intel i860™ Architecture ABI Supplement* and the *Unix System V Interface Definition,* Issue 3.

11.1 REGISTER ASSIGNMENT

Table 11-1 defines the standard for register allocation. Figure 11-1 presents the same information graphically.

NOTE

The dividing point between locals and parameters in the floating-point registers is now set at 8. Some earlier software (prior to the *Intel i860™ Architecture ABI Supplement*) used a dividing point at 16.

11.1.1 Integer Registers

Up to 12 integral parameters (including pointers and characters) can be passed to subroutines in the integer registers. The first (leftmost) parameter is passed in **r16**; the rest

Table 11-1. Register Allocation

Register	Purpose	Left Unchanged by a Subroutine?
r0	Always zero	Yes
r1	Return address	No
r2	Stack pointer	Yes[1]
r3	Frame pointer	Yes
r4-r15	Local values	Yes
r16-r17	Return value	No
r16-r27	Parameters and temporaries	No
r28	Memory parameter pointer	No
r29	Environment pointer	No
r28-r30	Temporaries	No
r31	Addressing temporary	No
f0-f1	Always zero	Yes
f2-f7	Local values	Yes
f8-f15	Return value	No
f8-f15	Parameters and temporaries	No
f16-f31	Temporaries	No

NOTE:
1. The stack pointer is normally kept unchanged across a subroutine call. However, some subroutines may allocate stack space and return with a different value in **r2**.

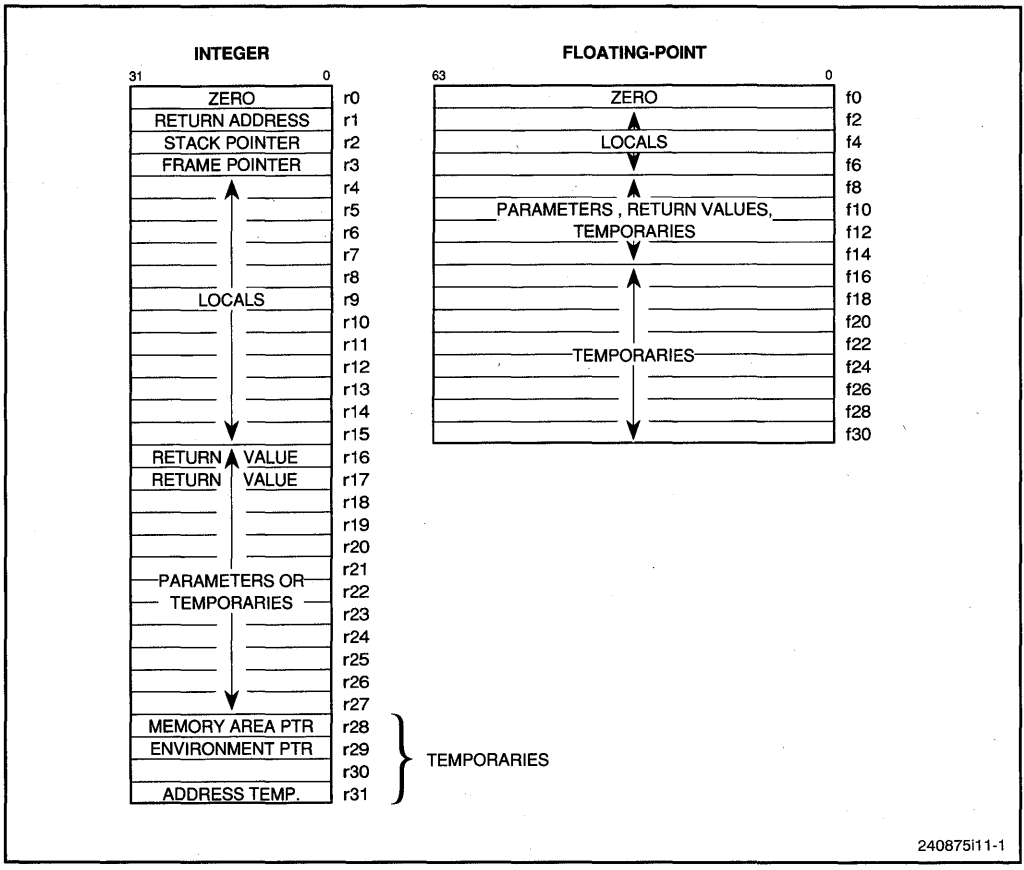


Figure 11-1. Register Allocation

in successively higher-numbered registers. If fewer parameters are required than the number of parameter registers available, the remaining registers can be used for temporary variables. If there are more integral parameters than will fit in the integer parameter registers, the remaining integral parameters are placed in a memory area, properly aligned, in their proper order, and possibly interspersed with other nonintegral parameters.

Registers **r16** and **r17** are both parameter registers and return value registers. If a subroutine has an integral return value, it loads the return value into **r16** before returning control to the caller. Register **r17** is used for return values that require more than one register. Sixty-four-bit integer values are returned in floating-point registers.

Register **r1** is the return-address register, because the **call** and **calli** instructions store the return address in it. Subroutines must therefore use the value in **r1** to return to the caller. If a subroutine saves **r1**, it may then use it as a temporary until it returns.

A separate addressing temporary register (**r31**) is allocated for construction of 32-bit address temporaries by assemblers. Assemblers may use **r31** by default to construct 32-bit addresses from 16-bit literals.

11.1.2 Floating-Point Registers

Floating-point and 64-bit integer values in the floating-point registers must use **f8–f15** when passed by value if space permits. The leftmost such parameter is passed in **f8** or **f8–f9**; the rest in successively higher-numbered registers. Single-precision parameters use one register; double-precision parameters use two properly aligned registers.

The first floating-point parameter is placed in the register pair **f8** and **f9**, if it is double precision, or in register **f8**, if it is single precision. The second floating-point parameter, if it is double-precision, is placed in the register pair **f10** and **f11**, regardless of the type of the first floating-point parameter. If the second floating-point parameter is single precision, it is placed in register **f9** if the first floating-point parameter is single precision, or in register **f10** if the first floating-point parameter is double precision. When single- and double-precision parameters are interspersed, it is possible for some floating-point parameter registers to remain unused in order to preserve alignment. This argument-to-register mapping does not change even if two floating-point arguments are separated by integral or aggregate arguments.

If there are more floating-point parameters than will fit in the floating-point parameter registers, the remaining parameters are placed in a memory area, properly aligned, in their proper order, and possibly interspersed with other non-floating-point parameters. The last (rightmost) parameter is at the highest stack address (i.e., it is pushed first using a grow-down stack).

11.1.3 Passing Structure Parameters in Memory

When passed by value, structure parameters are always placed in a memory area. The minimum alignment for a structure parameter passed in memory is 16 bits, even if the natural alignment requirement for the parameter is less.

11.1.4 Memory Parameter Area

When parameters are placed in memory, either because there are more parameters than fit in the allocated registers or because there are structure parameters, register **r28** is set to point to this area in memory by the caller. The memory parameter area must be properly aligned to preserve data alignment for the parameters within it. The minimum alignment for the area is 16 bits, even if the natural alignment of its parameters is less. Within the called procedure, register **r28** a local scratch register, which is not preserved for the caller.

11.1.5 Environment Pointer

For block-structured languages that allow up-level data references or for other languages or implementations that require that an environment be established, an environment pointer is placed in register **r29** before the call. Within the called procedure, this is a local scratch register, which is not preserved for the caller.

11.1.6 Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C language uses a special method to access variable-count parameters. The **stdarg.h** and **varargs.h** files define several functions to get at these parameters in a way that is independent of stack growth direction and of whether parameters are passed in registers or on the stack. A C subroutine with variable parameters must use the **va_start** macro to set up a data structure before the parameters can be used. The **va_arg** macro must be used to access successive parameters.

11.1.7 Returning Structures

If a called procedure returns a structure, the caller provides space for the return value and places the address of this space in **r16** before the **call** instruction is executed. Having the caller supply the return object's space allows reentrancy. The space provided by the caller must be aligned properly for the type of structure being returned.

11.2 DATA ALIGNMENT

Compilers and assemblers must do their best to keep data aligned. It is acceptable to have holes in data structures to keep all items aligned. In some cases (e.g., FORTRAN programs with overlaid data), it is necessary to have misaligned data. A run-time trap handler can be provided to handle misaligned data; however, such data imposes a performance penalty on the application. If a compiler must reference data that is known to be misaligned, the compiler should generate separate instructions to access the data in smaller units that will not generate misaligned-data traps. Accessing 16-bit misaligned data requires two byte loads plus a shift. Storing to 32-bit misaligned data may require four byte stores and three shifts. The code example in Example 11-1 is the recommended method for reading a misaligned 32-bit value whose address is in **r8**.

11.3 IMPLEMENTING A STACK

In general, compilers or programmers have to maintain a software stack. Register **r2** (called **sp** in assembly language) is the suggested stack pointer. Register **r2** is set by the operating system for the application when the program is started. The stack must be a grow-down stack, so as to be compatible with that of the Intel386 and Intel486 architecture. If a subroutine call requires placing parameters on the stack, then the caller is

```
    andnot  3,      r8,     r9  // Get address aligned on 4-byte boundary
    ld.l    0(r9),  r10         // Get low 32-bit value
    ld.l    4(r9),  r11         // Get high 32-bit value
    and     3,      r8,     r9  // Get byte offset in 8-byte field
    shl     3,      r9,     r9  // Convert to bit offset
    shr     r9,     r0,     r0  // Set shift count
    shrd    r11,    r10,    r9  // Put 32-bit value into R9

// If the misalignment offset (m) is known in advance, this code can be
//   optimized. Assume r8 points to next aligned address less than
//   address of misaligned field.
    ld.l    0(r8),  r10         // Get low value
    ld.l    4(r8),  r11         // Get high value
    shr     m*8,    r0,     r0  // Set shift count
    shrd    r11,    r10,    r9  // Put 32-bit value into R9
```

Example 11-1. Reading Misaligned 32-Bit Value

responsible for adjusting the stack pointer upon return. The caller must also allocate space on the stack for the overflow parameters (i.e., parameters that exceed the capacity of the registers reserved for passing parameters) and store them there before the call operation.

A separate frame pointer is used to allow calls to subroutines that change the stack pointer to allocate space on the stack at run-time (e.g., **alloca** and **va_start**). Some languages may also return values from a subroutine allocated on stack space below the original top-of-stack pointer. Such a subroutine prevents the caller from using **sp**-relative addressing to access values on the stack. If the compiler knows that it does not call subroutines that leave **sp** in an altered state when they return, then no frame pointer is necessary.

The stack must be kept aligned on 16-byte boundaries to keep data arrays aligned. Each subroutine must use stack space in multiples of 16 bytes. The frame pointer **r3** (called **fp** in assembly language) must at all times point to a 16-byte boundary.

Figure 11-2 shows the stack-frame format. A fixed format is necessary to allow stack-frame analysis by a low-level debugger or multiple-level stack-frame unwinding.

11.3.1 Stack Entry and Exit Code

Example 11-2 shows the recommended entry and exit code sequences. The stack pointer is restored to the value it had on entry into the subroutine. Assuming the subroutine needs to call another subroutine, it must save the frame pointer and its return address. It probably also needs to save some of its internal values across that call to another subroutine; therefore, the example saves one local register into the stack frame and subsequently reloads it.

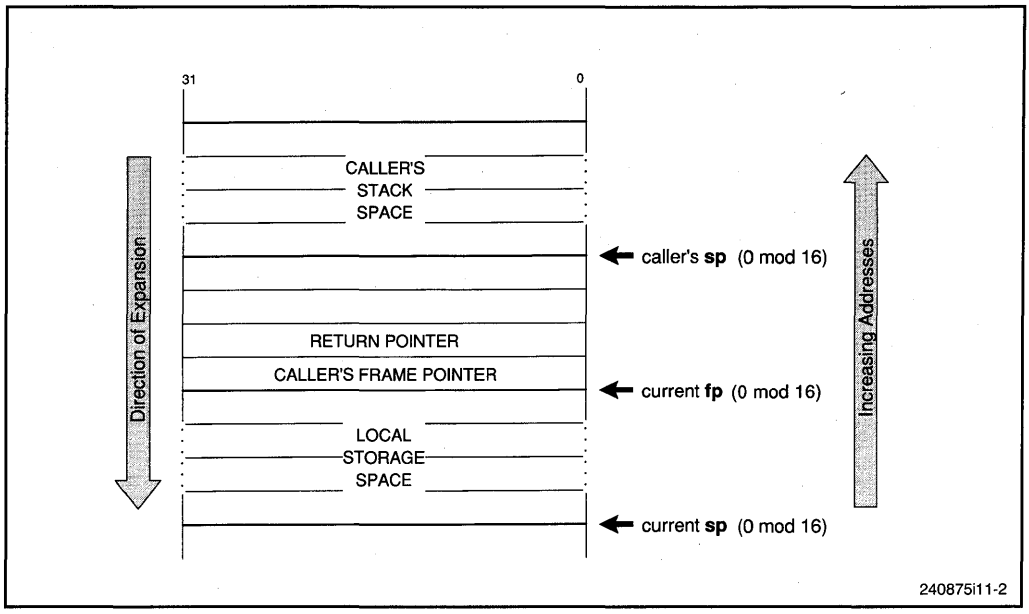


Figure 11-2. Stack Frame Format

```
// Subroutine entry
    adds -(Locals+16), sp, sp  // Allocate space for local variables
                               // Locals must be a multiple of 16
    st.l   fp,     Locals(sp)  // Save old frame pointer below old SP
    st.l   r1,   Locals+4(sp)  // Save return address
    adds   Locals,  sp,    fp  // Set new frame pointer
    st.l   r5,     -4(fp)      // Save a local register

// Subroutine exit
    ld.l    -4(fp), r5         // Restore a local register
    mov     fp,     sp         // Deallocate stack frame
    ld.l    4(fp),  r1         // Restore return address
    ld.l    0(fp),  fp         // Restore old frame pointer
    bri     r1                 // Return to caller after next instruc
    adds    16,     sp,    sp  // Deallocate frame pointer save area
```

Example 11-2. Subroutine Entry and Exit with Frame Pointer

Languages such as Pascal that need to maintain activation records on the stack can put them below the frame pointer in the program-specific area. The frame pointer is optional. All stack references can be made relative to **sp**. The code example in Example 11-3 shows the recommended entry and exit sequences when no frame pointer is required.

A lowest-level subroutine need not perform any stack accesses if it can run completely from the temporary registers. No entry/exit code is required by such a subroutine.

```
// Subroutine entry
    addu   -Locals,   sp, sp  // Allocate space for local variables
                              // -Locals must be a multiple of 16

// Subroutine exit
    bri    r1                 // Return to caller after next instruc
    addu   Locals,   sp, sp  // Restore stack pointer
```

Example 11-3. Subroutine Entry and Exit without Frame Pointer

11.3.2 Dynamic Memory Allocation on the Stack

Consider a function **alloca** that allocates space on the stack and returns a pointer to the space. The allocated space is lost when the caller returns. The function **alloca** could be implemented as shown in Example 11-4. For any function calling **alloca**, a separate stack pointer and frame pointer are required.

11.4 MEMORY ORGANIZATION

Every code section should begin with a **nop** instruction so that the trap handler can always examine the instruction at **fir** − 4 even in case a trap occurs on the first instruction of a section.

The memory-mapped I/O devices should also be placed in the upper operating-system data space. The paging hardware allows logical addresses to be different from their corresponding physical addresses. The I/O device logical address area may be located anywhere convenient.

11.5 INPUT/OUTPUT SPACE (80860XP ONLY)

The i860 XP microprocessor provides a four-Gbyte I/O space, which programs access via the **ldio** and **stio** instructions. The processor distinguishes cycles generated by the I/O instructions from memory accesses by driving the M/IO# output pin low. Generally, using a separate I/O space yields a simpler system design, because I/O mapped devices can have simpler address decoders. However, the choice between a separate I/O space and memory-mapped I/O may be dictated by existing software or by the need to interface with existing memory-mapped devices or with other processors in the same system.

```
alloca::
                                  // r16 has size requested
    adds    15,    r16,   r16 // Round size to 0 mod 16
    andnot  15,    r16,   r16 //
    subs    sp,    r16,   sp  // Adjust stack downwards
    bri     r1                // Return to caller after next instruc
    mov     sp,    r16        // Set return value to allocated space
```

Example 11-4. Possible Implementation of alloca

Other factors that programmers should consider in the use of the processor's I/O features include:

1. Protection.
   - The **ldio** and **stio** instructions are privileged; they can only be executed at supervisor level. (At user level they are treated as no-ops.)
   - Protection of memory-mapped I/O is under software control. Through page table allocation, I/O addresses may be reserved for the operating system or may be given selectively to user-level programs.

2. Address Translation.
   - The address operand of **ldio** and **stio** instructions is a physical I/O address. It is not translated by page tables.
   - The addresses of memory-mapped I/O operations *are* translated. This is advantageous if user-level programs are given access to I/O addresses. The physical addresses can be changed without affecting the user-level programs.

3. Cacheability.
   - The memory-mapped I/O space must be noncacheable so that all accesses to that space are seen by the devices and so that the possibility of write reordering is eliminated. Software can make the space noncacheable by setting the CD (cache disable) bit of page tables. External hardware can make the space noncacheable by deasserting the KEN# signal for cycles that access I/O addresses.
   - The processor never caches nor searches the data cache for the operands of I/O instructions.

4. Reserved I/O Addresses.
   - I/O addresses 0xF8–0xFF are reserved in the x86 family of architectures. Programs should avoid using these addresses so as not to interfere with other processors that might be in the same system.

Programming Examples 12

CHAPTER 12
PROGRAMMING EXAMPLES

12.1 SMALL INTEGERS

The 32-bit arithmetic instructions can be used to implement arithmetic on 8- or 16-bit ordinals and integers. The integer load instruction places 8- or 16-bit values in the low-order end of a 32-bit register and propagates the sign bit through the high-order bits of the register.

Occasionally, it is necessary to sign extend 8- or 16-bit integers that are generated internally, not loaded from memory. Example 12-1 shows how.

```
// SIGN-EXTEND 8-BIT INTEGER TO 32 BITS
// Assume the operand is already in r16
   shl     24,     r16,    r16 // left-justify
   shra    24,     r16,    r16 // right-justify all but sign bit
```

Example 12-1. Sign Extension

Example 12-2 shows how to load a small unsigned integer, converting the sign-extended form created by the load instruction to a zero-extended form.

```
// LOADING OF 8-BIT UNSIGNED INTEGERS
// Assume the address is already in r19

   // Load the operand (sign-extended) into r20
   ld.b   0(r19), r20

   // Mask out the high-order bits
   and    0x000000FF, r20, r20
```

Example 12-2. Loading Small Unsigned Integers

12.2 SINGLE-PRECISION DIVIDE

Example 12-3 computes Z = X ÷ Y for single-precision variables. The algorithm begins by using the reciprocal instruction **frcp** to obtain an initial guess for the value of 1/Y. The **frcp** instruction gives a result that can differ from the true value of 1/Y by as much as 2^{-8}. The algorithm then continues to make guesses based on the prior guess, refining each guess until the desired accuracy is achieved. Let G represent a guess, and let E represent the error, i.e., the difference between G and the true value of 1/Y. For each guess ...

$G_{new} = G_{old}(2 - G_{old}*Y)$.
$E_{new} = 2(E_{old})^2$.

This algorithm is optimized for high performance and does not produce results that are rounded according to the IEEE standard. Worst case error is about two least-significant bits.

12.3 DOUBLE-PRECISION DIVIDE

Example 12-4 computes Z = X ÷ Y for double-precision variables. The algorithm is similar to that shown previously for single-precision divide. For double-precision divide, one more iteration is needed to achieve the required accuracy.

This algorithm is optimized for high performance and does not produce results that are rounded according to the IEEE standard. Worst case error is about two least-significant bits.

```
// SINGLE-PRECISION DIVIDE

//      The dividend X is in f6
//      The divisor Y is in f2
//      The result Z is in f3
//      f5 contains single-precision floating-point 2.

   frcp.ss f2,      f3          // first guess has 2**-8 error
   fmul.ss f2,      f3,     f4  // guess * divisor
   fsub.ss f5,      f4,     f4  // 2 - guess * divisor
   fmul.ss f3,      f4,     f3  // second guess has 2**-15 error
   fmul.ss f2,      f3,     f4  // avoid using f3 as src1
   fsub.ss f5,      f4,     f4  // 2 - guess * divisor
   fmul.ss f6,      f3,     f5  // second guess * dividend
   fmul.ss f4,      f5,     f3  // result = second guess * dividend
```

Example 12-3. Single-Precision Divide

```
// DOUBLE-PRECISION DIVIDE

//      The dividend X is in f2
//      The divisor Y is in f4
//      The result Z is in f8

   frcp.dd f4,     f6           // first guess has 2**-8 error
   fmul.dd f4,     f6,     f8   // guess * divisor
   fld.d   dbltwo, f10          // load double-precision floating 2
// The fld.d is free.   It completely overlaps the preceding fmul.dd
   fsub.dd f10,    f8,     f8   // 2 - guess * divisor
   fmul.dd f6,     f8,     f6   // second guess has 2**-15 error
   fmul.dd f4,     f6,     f8   // avoid using f6 as src1
   fsub.dd f10,    f8,     f8   // 2 - guess * divisor
   fmul.dd f6,     f8,     f6   // third guess has 2**-29 error
   fmul.dd f4,     f6,     f8   // avoid using f6 as src1
   fsub.dd f10,    f8,     f8   // 2 - guess * divisor
   fmul.dd f6,     f2,     f6   // guess * dividend
   fmul.dd f8,     f6,     f8   // result = third guess * dividend
```

Example 12-4. Double-Precision Divide

```
// INTEGER MULTIPLY

//      The multiplier is in r4
//      The multiplicand is in r5
//      The product is left in r6
//      The registers f2, f4, and f6 are used as temporaries.

   ixfr    r4,    f2
   ixfr    r5,    f4
// Two core instructions can be inserted here without penalty.
   fmlow.dd    f4, f2, f6
// Four core instructions can be inserted here without penalty.
   fxfr    f6,    r6
// One core instruction can be inserted here without penalty.
```

Example 12-5. Integer Multiply

12.4 INTEGER MULTIPLY

A 32-bit integer multiply is implemented in Example 12-5 by transferring the operands to floating-point registers and using the **fmlow** instruction. If the result is referenced in the next instruction, eleven clocks are required. Seven clocks can be overlapped with other operations.

12.5 CONVERSION FROM SIGNED INTEGER TO DOUBLE

The strategy used in Example 12-6 is to use the bits of the integer to construct a value in double-precision format. The double-precision value constructed contains two biases:

BC — A bias that compensates for the fact that the signed integer is stored in two's complement format. The value of this bias is 2^{31}.

BN — A bias that produces a normalized number, so that the algorithm does not cause a floating-point exception. The value of this bias is 2^{52}.

If the desired value is **x**, then the constructed value is **x** + BC + BN. By later subtracting BC + BN, the value **x** is left in double precision format, properly normalized by the processor. The value of BC + BN is $2^{52}+2^{31}$ (0x4330_0000_8000_0000).

The conversion requires 7 clocks if the result is referenced in the next instruction. Three clocks can be overlapped with other operations. If a single-precision result is required, add an **famov.ds** instruction at the end.

12.6 SIGNED INTEGER DIVIDE

Example 12-7 combines the techniques of Section 12.3 and 12.5.

12.7 STRING COPY

Example 12-8 shows how to avoid the freeze condition that might occur when using a load in a tight loop such as that commonly used for copying strings. A performance penalty is incurred if the destination of a load is referenced in the next instruction. In order to avoid this condition, Example 12-8 juggles characters of the string between two registers.

```
// CONVERT SIGNED INTEGER TO DOUBLE

//      The integer is in r4
//      The double-precision floating-point result is left in f7:f6
//      The register f5:f4 contains BN+BC

    xorh 0x8000, r4, r4 // Complement sign bit (equivalent to adding BC).
    ixfr    r4,  f6     // Construct low half.
    fmov.ss f5,  f7     // Set exponent in high half (includes BN)
//  One instruction can be inserted here without penalty.
    fsub.dd f6,     f4,     f6  // (x + BN + BC) - (BN + BC) = x
//  Two core instructions can be inserted here without penalty.
```

Example 12-6. Signed Integer to Double Conversion

```
// SIGNED INTEGER DIVIDE
//      The denominator is in r4
//      The numerator is in r5
//      The quotient is left in r6
//      The remainder is left in r7
//      The registers f2 through f11 are used as temporaries.
// Convert Denominator and Numerator
    fld.d   two52two31,     f6  // load constant 2**52 + 2**31
    xorh    0x8000, r4,     r19 //
    ixfr    r19,    f4          //
    fmov.ss f7,     f5          //
    xorh    0x8000, r5,     r20 //
    fsub.dd f4,     f6,     f4  //
    ixfr    r20,    f2          //
    fmov.ss f7,     f3          //
    fsub.dd f2,     f6,     f2  //
// Do Floating-Point Divide
    fld.d   dbltwo, f10         // load floating-point two
    frcp.dd f4,     f6          // first guess has 2**-8 error
    fmul.dd f4,     f6,     f8  // guess * divisor
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f8,     f6  // second guess has 2**-15 error
    fmul.dd f4,     f6,     f8  // avoid using f6 as src1
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f8,     f6  // third guess has 2**-29 error
    fmul.dd f4,     f6,     f8  // avoid using f6 as src1
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f2,     f6  // guess * dividend
    fmul.dd f8,     f6,     f8  // result = third guess * dividend
// Convert Quotient to Integer
    fld.d   onepluseps,     f10 // load value 1 + 2**-40
    fmul.dd f8,     f10,    f8  // force quotient to be bigger than integer
    ixfr    r4,     f10         // get denominator for remainder computation
    ftrunc.dd       f8,     f8  // convert to integer
// Compute Remainder (optional)
    fmlow.dd f10,   f8,     f10 // quotient * denominator
    fxfr    f10,    r4
    fxfr    f8,     r6          // transfer quotient
    subs    r5,     r4,     r7  // remainder = numerator - quotient *
denominator
```

Example 12-7. Signed Integer Divide

12.8 FLOATING-POINT PIPELINE

Most instruction sequences that use pipelined instructions can be divided into three phases:

Priming — Filling a pipeline with known intermediate results while disposing of previous pipeline contents.

```
// STRING COPY
//  Assumptions:
//      Source address alignment unknown
//      Destination address alignment unknown
//      End of string indicated by NUL
//  r17 - address of source string
//  r16 - address of destination string

copy_string::
    ld.b    0(r17),         r26     // Load one character
    bte     0,      r26,    done    // Test for NUL character
    adds    1,      r17,    r17     // Bump pointer to source string
    ld.b    0(r17),         r27     // Load one more character
    subs    r17,    r16,    r18     // Use constant offset to avoid
                                    //  incrementing two indexes
loop::
    st.b    r26,            0(r16)  // Store previous character
    adds    1,      r16,    r16     // Bump common index
    or      r0,     r27,    r26     // Test for NUL character
    bnc.t                   loop    // If not NUL, branch after loading
    ld.b    r18(r16),       r27     //  next character. r18(r16) = 0(r17)
done::
    bri     r1                      // Return after storing
    st.b    r26,            0(r16)  //  the NUL character, too
```

Example 12-8. String Copy

Continuous Operation — Receiving expected results with the initiation of each new pipelined instruction.

Draining — Retrieving the results that remain in the pipeline after the pipelined instruction sequence has terminated.

Example 12-9 shows one strategy for using the floating-point adder, which has a three-stage pipeline. This example assumes that the prior contents of the adder's pipeline are unimportant, and discards them by specifying register **f0** as the destination of the first three instructions. After performing the intended calculations, it drains the pipeline by executing three dummy addition instructions with **f0** (which always contains zero) as the operands.

12.9 PIPELINING OF DUAL-OPERATION INSTRUCTIONS

When using dual-operation instructions (all of which are pipelined), code that primes and drains the pipelines must take into account both the adder and multiplier pipelines. Example 12-10 illustrates pipeline usage for a simple single-precision matrix operation: the dot product of a 1×8 row matrix **A** with an 8×1 column matrix **B**. For the purpose of

```
// PIPELINED FLOATING-POINT ADD

//  Calculates  f10 = f4 + f5,  f11 = f6 + f7
//              f12 = f8 + f9,  f13 = f5 + f6

//  Assume  f4 = 1.0,   f5 = 2.0,   f6 = 3.0
//          f7 = 4.0,   f8 = 5.0,   f9 = 6.0

//                                Stage 1   Stage 2   Stage 3   Result
// Priming phase
    pfadd.ss f4, f5, f0      //    1+2       ??        ??       Discard
    pfadd.ss f6, f7, f0      //    3+4       1+2       ??       Discard
    pfadd.ss f8, f9, f0      //    5+6       3+4        3       Discard

// Continuous operation phase
    pfadd.ss f5, f6, f10     //    2+3       5+6        7       f10= 3
// For longer pipelined sequences, include more instructions here

// Draining phase
    pfadd.ss f0, f0, f11     //    0+0       2+3       11       f11= 7
    pfadd.ss f0, f0, f12     //    0+0       0+0        5       f12=11
    pfadd.ss f0, f0, f13     //    0+0       0+0        0       f13= 5
```

Example 12-9. Pipelined Add

tracking values through the pipelines, assume that the actual matrices to be multiplied have the following values:

A = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0] **B** = $\begin{bmatrix} 8.0 \\ 7.0 \\ 6.0 \\ 5.0 \\ 4.0 \\ 3.0 \\ 2.0 \\ 1.0 \end{bmatrix}$

Assume further that the two matrices are already loaded into registers thus:

A:		**B:**	
	f4 = 1.0		f12 = 8.0
	f5 = 2.0		f13 = 7.0
	f6 = 3.0		f14 = 6.0
	f7 = 4.0		f15 = 5.0
	f8 = 5.0		f16 = 4.0
	f9 = 6.0		f17 = 3.0
	f10 = 7.0		f18 = 2.0
	f11 = 8.0		f19 = 1.0

The calculation to perform is 1.0*8.0 + 2.0*7.0 + ... + 8.0*1.0 – a series of multiplications followed by additions. The dual-operation instructions are designed precisely to execute this type of calculation efficiently by using the adder and multiplier in parallel. At the heart of Example 12-10 is the dual-operation instruction **m12apm**, which multiplies its operands and adds the multiplier result to the result of the adder.

The priming phase is somewhat different in Example 12-10 than in Example 12-9. Because the result of the adder is fed back into the adder, it is not possible to simply ignore the prior contents of the adder pipeline; and because the result of the multiplier is automatically fed into the adder, it is important to consider the effect of the multiplier on the adder pipeline as well. This example waits until unknown results have been drained from the multiplier pipeline, then puts zeros in all stages of the adder pipeline.

Because the adder pipeline has three stages, the draining phase produces three partial results that must be added together.

```
// PIPELINED DUAL-OPERATION INSTRUCTION

//                              Multiplier          Adder
//                                Stages            Stages
//                              1    2   3     1       2    3   Result

// Priming phase
    pfmul.ss f4, f12, f0  // 1*8  ??   ??   ??      ??    ??  Discard
    pfmul.ss f5, f13, f0  // 2*7  1*8  ??   ??      ??    ??  Discard
    pfmul.ss f6, f14, f0  // 3*6  2*7  8    ??      ??    ??  Discard

    pfadd.ss  f0, f0 ,f0  //                 0      ??    ??  Discard
    pfadd.ss  f0, f0 ,f0  //                 0       0    ??  Discard
    pfadd.ss  f0, f0 ,f0  //                 0       0     0  Discard

// Continuous operation phase
    m12apm.ss f7, f15,f0  // 4*5  3*6  14   8+0     0+0    0  Discard
    m12apm.ss f8, f16,f0  // 5*4  4*5  18  14+0     8+0    0  Discard
    m12apm.ss f9, f17,f0  // 6*3  5*4  20  18+0    14+0    8  Discard
    m12apm.ss f10,f18,f0  // 7*2  6*3  20  20+8    18+0   14  Discard
    m12apm.ss f11,f19,f0  // 8*1  7*2  18  20+14   20+8   18  Discard
// For larger matrices, include more instructions here

// Draining phase
    m12apm.ss f0, f0, f0  // 0*0  8*1  14  18+18   20+14  28  Discard
    m12apm.ss f0, f0, f0  // 0*0  0*0  8   14+28   18+18  34  Discard
    m12apm.ss f0, f0, f0  // 0*0  0*0  0    8+34   14+28  36  Discard
    // Sum the partial results
    pfadd.ss  f0, f0, f20 //                 0+0     8+34  42  f20=36
    pfadd.ss  f20,f21,f21 //                42+36    0+0   42  f21=42
    pfadd.ss  f0, f0, f20 //                 0+0    42+36   0  f20=42
    pfadd.ss  f0, f0, f0  //                 0+0     0+0   78  Discard
    pfadd.ss  f0, f0, f21 //                 0+0     0+0    0  f21=78
    fadd.ss   f20,f21,f20 //                                   f20=120
```

Example 12-10. Pipelined Dual-Operation Instruction

12.10 PIPELINING OF DOUBLE-PRECISION DUAL OPERATIONS

Example 12-11 illustrates how pipeline usage for double-precision differs from the single-precision Example 12-10. Example 12-11 performs the dot product of a 1×6 row matrix **A** with a 6×1 column matrix **B**. For the purpose of tracking values through the pipelines, assume that the actual matrices to be multiplied have the following values:

A = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0,] **B** = $\begin{bmatrix} 6.0 \\ 5.0 \\ 4.0 \\ 3.0 \\ 2.0 \\ 1.0 \end{bmatrix}$

Assume further that the two matrices are already loaded into registers thus:

A:	f5:f4 = 1.0	**B:**	f17:f16 = 6.0
	f7:f6 = 2.0		f19:f18 = 5.0
	f9:f8 = 3.0		f21:f20 = 4.0
	f11:f10 = 4.0		f23:f22 = 3.0
	f13:f12 = 5.0		f25:f24 = 2.0
	f15:f14 = 6.0		f27:f26 = 1.0

Example 12-11 differs from Example 12-10 in that, with double precision, the multiplier pipeline has only two stages; therefore, the priming and draining phases use fewer instructions.

12.11 DUAL INSTRUCTION MODE

The previous Example 12-9 and Example 12-10 showed how i860 microprocessors can deliver up to two floating-point results per clock by using the pipelining and parallelism of the adder and multiplier units. These examples, however, assume that the data is already loaded in registers. Example 12-12 goes one step further and shows how to maintain the high throughput of the floating-point unit while simultaneously loading the data from main memory and controlling the logical flow.

The problem is to sum the single-precision elements of an arbitrarily long vector. The procedure uses dual-instruction mode to overlap loading, decision making, and branching with the basic pipelined floating-point add instruction **pfadd.ss**. To make obvious the pairing of core and floating-point instructions in dual-instruction mode, the listing in Example 12-12 shows the core instruction of a dual-mode pair indented with respect to the corresponding floating-point instruction.

Elements are loaded two at a time into alternating pairs of registers: one time at **loop1** into **f20** and **f21**, the next time at **loop2** into **f22** and **f23**. Performance would be slightly degraded if the destination of a **fld.d** were referenced as a source operand in the next

```
// PIPELINED DUAL-OPERATION INSTRUCTION -- DOUBLE PRECISION

//                          Multiplier          Adder
//                            Stages            Stages
//                             1    2        1      2    3   Result

// Priming phase
    m12apm.dd f4, f16,f0  // 1*6  ??        ??     ??   ??   Discard
    m12apm.dd f6, f18,f0  // 2*5  1*6       ??     ??   ??   Discard

    pfadd.dd f0, f0 ,f0   //                 0     ??   ??   Discard
    pfadd.dd f0, f0 ,f0   //                 0      0   ??   Discard
    pfadd.dd f0, f0 ,f0   //                 0      0    0   Discard

// Continuous operation phase
    m12apm.dd f8, f20,f0  // 3*4  2*5       6+0    0    0   Discard
    m12apm.dd f10,f22,f0  // 4*3  3*4      10+0    6+0  0   Discard
    m12apm.dd f12,f24,f0  // 5*2  4*3      12+0   10+0  6   Discard
    m12apm.dd f14,f26,f0  // 6*1  5*2      12+6   12+0  10  Discard
// For larger vectors, include more instructions here

// Draining phase
    m12apm.dd f0, f0, f0  // 0*0  6*1      10+10  12+6  12  Discard
    m12apm.dd f0, f0, f0  // 0*0  0*0       6+12  10+10 18  Discard
//  Three partial sums are now in the adder pipeline.
    pfadd.dd  f0 ,f0 ,f28 //                 0     6+12  20  f28 = 18
    pfadd.dd  f28,f30,f30 //                18+20   0    18  f30 = 20
    pfadd.dd  f0 ,f0 ,f28 //                 0    18+20  0  f28 = 18
    pfadd.dd  f0 ,f0 ,f0  //                 0     0    38  Discard
    pfadd.dd  f0 ,f0 ,f30 //                 0     0     0  f30 = 38
    fadd.dd   f28,f30,f30 //                                 f30 = 56
```

Example 12-11. Pipelined Double-Precision Dual Operation

two instructions. The strategy of alternating registers avoids this situation and maintains maximum performance. Some extra logic is needed at **sumup** to account for an odd number of elements.

12.12 CACHE STRATEGIES FOR MATRIX DOT PRODUCT

Calculations that use (and reuse) massive amounts of data may achieve significantly less than optimum performance unless their memory access demands are carefully taken into consideration during algorithm design. The prior Example 12-12 easily executes at near the theoretical maximum speed of the processor because it does not make heavy demands on the memory subsystem. This section considers a more demanding calculation, the dot product of two matrices, and analyzes two memory access strategies as they apply to this calculation.

The product of matrix $\mathbf{A}=A_{i,j}$ of dimension $L\times M$ with matrix $\mathbf{B}=B_{i,j}$ of dimension $M\times N$ is the matrix $\mathbf{C}=C_{i,j}$ of dimension $L\times N$, where ...

$$C_{i,j} = A_{i,1}B_{1,j}+A_{i,2}B_{2,j}+\ldots +A_{i,M}B_{M,j} \text{ (for } 1 \le i \le L,\ 1 \le j \le N\text{)}$$

```
// SINGLE-PRECISION VECTOR SUM
//  input:  r16 - vector address, r17 - vector size (must be > 5)
//  output: f16 - sum of vector elements
      fld.d     r0(r16),    f20     // Load first two elements
      mov       -2,         r21     // Loop decrement for bla
                    // Initiate entry into dual-instruction mode
      d.pfadd.ss f0,    f0,     f0  // Clear adder pipe (1)
      adds      -6,     r17,    r17 // Decrement size by 6
                        // Enter into dual-instruction mode
    d.pfadd.ss  f0,     f0,     f0  // Clear adder pipe (2)
      bla       r21,    r17,    L1  // Initialize LCC
    d.pfadd.ss  f0,     f0,     f0  // Clear adder pipe (3)
      fld.d     8(r16)++,       f22 // Load 3rd and 4th elements
L1::d.pfadd.ss  f20,    f30,    f30 // Add f20 to pipeline
      bla       r21,    r17,    L2  // If more, go to L2 after
    d.pfadd.ss  f21,    f31,    f31 // adding f21 to pipeline and
      fld.d     8(r16)++,       f20 // loading next f20:f21
    // If we reach this point, at least one element remains to be loaded.
    // r17 is either -4 or -3.
    // f20, f21, f22, and f23 still contain vector elements.
    // Add f20 and f22 to the pipeline, too.
    d.pfadd.ss  f20,    f30,    f30
      br        S                   // Exit loop after adding
    d.pfadd.ss  f21,    f31,    f31 // f21 to the pipeline
      nop
L2::d.pfadd.ss  f22,    f30,    f30 // Add f22 to pipeline
      bla       r21,    r17,    L1  // If more, go to L1 after
    d.pfadd.ss  f23,    f31,    f31 // adding f23 to pipeline and
      fld.d     8(r16)++,       f22 // loading next f22:f23
    // If we reach this point, at least one element remains to be loaded.
    // r17 is either -4 or -3.
    // f20, f21, f22, and f23 still contain vector elements.
    // Add f20 and f21 to the pipeline, too.
    d.pfadd.ss  f20,    f30,    f30
      nop
    d.pfadd.ss  f21,    f31,    f31
      nop
S::                         // Initiate exit from dual mode
    pfadd.ss    f22,    f30,    f30 // Still in dual mode
      mov       -4,             r21
    pfadd.ss    f23,    f31,    f31 // Last dual-mode pair
      bte       r21,    r17,   DONE // If there is one more
      fld.l     8(r16)++,       f20 // element, load it and
      pfadd.ss  f20,    f30,    f30 // add to pipeline
      // Intermediate results are sitting in the adder pipeline.
      // Let A1:A2:A3 represent the current pipeline contents
DONE::pfadd.ss  f0,     f0,     f30 // 0:A1:A2       f30=A3
      pfadd.ss  f30,    f31,    f31 // A2+A3:0:A1    f31=A2
      pfadd.ss  f0,     f0,     f30 // 0:A2+A3:0     F30=A1
      pfadd.ss  f0,     f0,     f0  // 0:0:A2+A3
      pfadd.ss  f0,     f0,     f31 // 0:0:0         F31=A2+A3
      fadd.ss   f30,    f31,    f16 // f16 = A1+A2+A3
```

Example 12-12. Dual-Instruction Mode

The basic algorithm for calculation of a dot product appears in Example 12-10. To extend this algorithm to the current problem requires adding instructions to:

1. Load the entries of each matrix from memory at appropriate times.

2. Repeat the inner loop as many times as necessary to span matrices of arbitrary *M* dimension.

3. Repeat the entire algorithm *L*N* times to produce the $L \times N$ product matrix.

Each of the Examples 12-13 and 12-14 accomplishes the above extensions through straightforward programming techniques. Each example uses dual-instruction mode to perform the loading and loop control operations in parallel with the basic floating-point calculations. The examples differ in their approaches to memory access and cache usage. To eliminate needless complexity, the examples require that the *M* dimension be a multiple of eight and that the **B** matrix be stored in memory by column instead of by row. Data is fetched 32 bytes beyond the higher-address end of both matrices. In real applications, programmers should ensure that no page protection faults occur due to these accesses.

- Example 12-13 depends solely on cached loads.
- Example 12-14 depends on a mix of cached and pipelined loads.

Example 12-13 uses the **fld** instruction for all loads, which places all elements of both matrices **A** and **B** in the cache. This approach is ideal for small matrices. Accesses to all elements (after the first access to each) retrieve elements from the cache at the rate of one per clock. Using **fld.q** instructions to retrieve four elements at a time, it is possible to overlap all data access as well as loop control with **m12apm** instructions in the inner loop.

Note, however, that Example 12-13 is "cache bound"; i.e., if the combined size of the two matrices is greater than that of the cache, cache misses will occur, degrading performance. The larger the matrices, the more the misses that will occur.

Example 12-14 uses **fld** for all the elements of each row of **A**, and uses **pfld** to pass all columns of **B** against each row of **A**. This example is less cache bound, because only rows of **A** are placed in the cache. More load instructions are required, because a **pfld** can load at most two single-precision operands. Still, with pipelined memory cycles, it remains possible to overlap the loading of the eight items from matrix **A**, the eight items from matrix **B**, and the loop control with the eight **m12apm** instructions in the inner loop.

The strategy of Example 12-14 is suitable for larger matrices than the strategy in Example 12-13 because, even in the extreme case where only one row of **A** fits in the cache, cache misses occur only the first time each row is processed. However, if dimension *M* is so great that not even one row of **A** fits entirely in the cache, cache misses will still occur. On the other side, for small matrices, Example 12-14 may not perform as well as Example 12-13, because, even when there is sufficient space in the cache for elements of matrix **B**, Example 12-14 does not use it.

```
// MATRIX MULTIPLY, C = A * B, CACHED LOADS ONLY
// Registers loaded by calling routine
A=r16   // pointer into A, stored in memory by rows
B=r17   // pointer into B, stored in memory by columns
C=r18   // pointer into C, stored in memory by rows
L=r19   // the number of rows in A
M=r20   // the number of columns in A and rows in B
N=r21   // the number of columns in B
// Registers used locally
RC=r28  // row/column counter decremented by bla for loop control
DEC=r27 // decrementor for row/column pointers
Ar=r26  // counter of rows in A
Bc=r25  // counter of columns in B
Bp=r24  // temporary pointer into B
SIZ=r23 // number of bytes in row of A or column of B
A1=f4; A2=f5; A3=f6; A4=f7; A5=f8; A6=f9; A7=f10;A8=f11 // A row
B1=f12;B2=f13;B3=f14;B4=f15;B5=f16;B6=f17;B7=f18;B8=f19 // B column
T1=f20;T2=f21;T3=f22                                  // temporary results
  shl           2, M, SIZ       // Number of bytes in M entries
  adds          -8, r0, DEC     // Set decrementor for bla
  adds          -8, M, RC       // Initialize row/column counter
  adds          -4, C, C        // Start C index one entry low
  d.fiadd.dd    f0, f0, f0      // Initiate dual-instruction mode
  adds          -1, L, Ar       // Make row counter zero relative
  d.fnop                        // First dual-mode pair
    bla     DEC, RC, start_row  // Initialize LCC
  d.fnop                        //
    subs          A, SIZ, A     // Start pointer to A one row low
start_row::                     // Executed once per row of A
  d.pfmul.ss    f0, f0, f0      //
    mov           B, Bp         // Point to first col of B
  d.pfmul.ss    f0, f0, f0      //
    adds          SIZ, A, A     // Point to next row of A
  d.pfmul.ss    f0, f0, f0      //
    fld.q         16(Bp), B5    // Load 4 entries of B
  d.pfadd.ss    f0, f0, f0      //
    fld.q       16(A), A5       // Load 4 entries of A
  d.pfadd.ss    f0, f0, f0      //
    adds          -1, N, Bc     // Initialize column counter
  d.pfadd.ss    f0, f0, f0      //
    fld.q         0(A), A1      // Load 4 entries of A
```

Example 12-13. Matrix Multiply, Cached Loads Only (1 of 2)

```
inner_loop:: // Process eight entries of row of A with eight of col of B
  d.m12apm.ss   A5,  B5, T1      //
    fld.q         0(Bp), B1      // Load 4 entries of B
  d.m12apm.ss   A6, B6, T1       //
    adds          32, A, A       // Bump pointer to A by 8 entries
  d.m12apm.ss   A7, B7, T1       //
    adds          32, Bp, Bp     // Bump pointer to B by 8 entries
  d.m12apm.ss   A8, B8, T1       //
    fld.q         16(Bp), B5     // Load 4 entries of B
  d.m12apm.ss   A1, B1,T1        //
    fld.q         16(A), A5      // Load 4 entries of A
  d.m12apm.ss   A2, B2,T1        //
    nop                          //
  d.m12apm.ss   A3, B3, T1       //
    bla   DEC, RC, inner_loop    // Loop until end of row/column
  d.m12apm.ss   A4, B4, T2       //
    fld.q         0(A), A1       // Load 4 entries of A
// End Inner Loop.  End of row/column
  d.m12apm.ss   f0, f0, T3       //
    subs          A, SIZ, A      // Set A pointer back to beginning of row
  d.m12apm.ss   f0, f0, T1       //
    adds          -8, M, RC      // Reinitialize row/column counter
  d.m12apm.ss   f0, f0, T2       //
    nop                          //
  d.pfadd.ss    f0, f0, T3       //
    bla     DEC, RC, inner_loop  // Won't branch; initializes LCC
  d.pfadd.ss    f0, f0, T1       //
    fld.q         16(A), A5      // Load 4 entries of A
  d.pfadd.ss    f0, f0, T2       //
    fld.q         16(Bp), B5     // Load 4 entries of B
  d.fadd.ss     T1, T3, T3       //
    fld.q         0(A), A1       // Load 4 entries of A
  d.fadd.ss     T2, T3, T3       //
    adds          -1, Bc, Bc     // Decrement column counter
  d.pfadd.ss    f0, f0, f0       //
    fst.l         T3, 4(C)++     // Store row/column product in C
// Continue with next column of B?
  d.pfadd.ss    f0, f0, f0       //
    bnc.t       inner_loop       // CC controlled by prior adds
  d.pfadd.ss    f0, f0, f0       //
    nop                          //
// Continue with next row of A?
  d.fnop                         //
    xor           Ar, r0, r0     // Is row counter zero?
  d.fnop                         //
    bnc.t         start_row      // Taken if row counter not zero
  d.fnop                         //
    adds          -1, Ar, Ar     // Decrement row counter
  fnop                           // Initiate exit from dual mode
    nop                          //
  fnop                           // Last dual-mode pair
    nop                          // End
```

Example 12-13. Matrix Multiply, Cached Loads Only (2 of 2)

```
// MATRIX MULTIPLY, C = A * B, CACHED AND PIPELINED LOADS MIXED
// Registers loaded by calling routine
A=r16   // pointer into A, stored in memory by rows
B=r17   // pointer into B, stored in memory by columns
C=r18   // pointer into C, stored in memory by rows
L=r19   // the number of rows in A
M=r20   // the number of columns in A and rows in B
N=r21   // the number of columns in B
// Registers used locally
Ap=r29  // temporary pointer into A
RC=r28  // row/column counter decremented by bla for loop control
DEC=r27 // decrementor for row/column pointers
Ar=r26  // counter of rows in A
Bc=r25  // counter of columns in B
Bp=r24  // temporary pointer into B
SIZ=r23 // number of bytes in row of A or column of B
A1=f4; A2=f5; A3=f6; A4=f7; A5=f8; A6=f9; A7=f10;A8=f11 // A row
B1=f12;B2=f13;B3=f14;B4=f15;B5=f16;B6=f17;B7=f18;B8=f19 // B column
T1=f20;T2=f21;T3=f22                                  // temporary results
  mov           B, Bp           // Pointer to B
  shl           2, M, SIZ       // Number of bytes in M entries
  adds          -8, r0, DEC     // Set decrementor for bla
  adds          -8, M, RC       // Initialize row/column counter
  d.fiadd.dd    f0, f0, f0      // Initiate dual-instruction mode
  adds          -4, C, C        // Start C index one entry low
  d.fnop                        // First dual-mode pair
    adds          -1, L, Ar     // Make row counter zero relative
  d.fnop                        //
    bla     DEC, RC, start_row  // Initialize LCC
  d.fnop                        //
    mov           A, Ap         // Pointer to A
start_row::                     // Executed once per row of A
  d.pfmul.ss    f0, f0, f0      //
    pfld.d        0(Bp), f0     // Load 2 entries of B into load pipe
  d.pfmul.ss    f0, f0, f0      //
    pfld.d        8(Bp)++, f0   // Load 2 entries of B into load pipe
  d.pfmul.ss    f0, f0, f0      //
    pfld.d        8(Bp)++, f0   // Load 2 entries of B into load pipe
  d.pfadd.ss    f0, f0, f0      //
    fld.q         0(Ap), A1     // Load 4 entries of A
  d.pfadd.ss    f0, f0, f0      //
    pfld.d        8(Bp)++, B1   // Load 2 entries of B
  d.pfadd.ss    f0, f0, f0      //
    adds          -1, N, Bc     // Initialize column counter
  d.fnop                        //
    pfld.d        8(Bp)++, B3   // Load 2 entries of B
inner_loop:: // Process eight entries from A row with eight from B col
  d.m12apm.ss   A1, B1, f0      //
    fld.q         16(Ap)++, A5  // Load 4 entries of A
  d.m12apm.ss   A2, B2, f0      //
    pfld.d        8(Bp)++, B5   // Load 2 entries of B
  d.m12apm.ss   A3, B3, f0      //
    pfld.d        8(Bp)++, B7   // Load 2 entries of B
```

Example 12-14. Matrix Multiply, Cached and Pipelined Loads (1 of 2)

```
  d.m12apm.ss   A4, B4, f0       //
    fld.q         16(Ap)++, A1   // Load 4 entries of A
  d.m12apm.ss   A5, B5, f0       //
    nop                          //
  d.m12apm.ss   A6, B6, f0       //
    pfld.d        8(Bp)++, B1    // Load 2 entries of B
  d.m12apm.ss   A7, B7, f0       //
    bla     DEC, RC, inner_loop // Loop until end of row/column
  d.m12apm.ss   A8, B8, f0       //
    pfld.d        8(Bp)++, B3    // Load 2 entries of B
// End Inner Loop. End of row/column
  d.m12apm.ss   f0, f0, f0       //
    nop                          //
  d.m12apm.ss   f0, f0, f0       //
    adds          -8, M, RC      // Reinitialize row/column counter
  d.m12apm.ss   f0, f0, f0       //
    mov           A, Ap          // Set A pointer back to beginning of row
  d.pfadd.ss    f0, f0, T3       //
    fld.q         0(Ap), A1      // Load first 4 entries of row of A
  d.pfadd.ss    f0, f0, T1       //
    bla     DEC, RC, inner_loop // Won't branch; initializes LCC
  d.pfadd.ss    f0, f0, T2       //
   nop                           //
  d.fadd.ss     T1, T3, T3       //
   nop                           //
  d.fadd.ss     T2, T3, T3       //
   adds           -1, Bc, Bc     // Decrement column counter
  d.pfadd.ss    f0, f0, f0       //
    fst.l         T3, 4(C)++     // Store row/column product in C
// Continue with next column of B?
  d.pfadd.ss    f0, f0, f0       //
    bnc.t         inner_loop     // CC controlled by prior adds
  d.pfadd.ss    f0, f0, f0       //
    nop                          //
// End of all columns of B
  d.fnop                         //
    mov           B, Bp          // Point to first col of B
  d.fnop                         //
    adds          A, SIZ, A      // Bump pointer to A by one row
  d.fnop                         //
    mov           A, Ap          // Set A index to beginning of next row
// Continue with next row of A?
  d.fnop                         //
    xor           Ar, r0, r0     // Is row counter zero?
  d.fnop                         //
    bnc.t         start_row      // Taken if row counter not zero
  d.fnop                         //
    adds          -1, Ar, Ar     // Decrement row counter
  fnop                           // Initiate exit from dual mode
    nop                          //
  fnop                           // Last dual-mode pair
    nop                          // End
```

Example 12-14. Matrix Multiply, Cached and Pipelined Loads (2 of 2)

12.13 3-D RENDERING

This series of examples are routines that might be used at the lowest level of a graphics software system to convert a machine-independent description of a 3-D image into values for the frame buffer of a color video display. Typically, higher-level graphics routines represent an object as a set of polygons that together roughly describe the surfaces of the objects to be displayed. The graphics system maintains a database that describes these polygons in terms of their colors, properties of reflectance or translucence, and the locations in 3-D space of their vertices. Due to the roughness of the representation, the amount of information in the database is considerably less than that which must be delivered to the video display. A rendering procedure, such as Example 12-21, uses interpolation to derive the detailed information needed for each pixel in the graphics frame buffer. The rendering procedure also performs pixel-by-pixel hidden-surface elimination.

The focus of this series of examples is Example 12-21, which operates on a segment of a scan line. The segment is bounded by two points of given location and color: from point (*X1, Y0, Z1*) with color intensities *Red1, Grn1, Blu1* to point (*X2, Y0, Z2*) with color intensities *Red2, Grn2, Blu2*. The points and color intensities are determined by higher-level graphics software. The points represent the intersection of the scan line with two edges of the projected image of a polygon. For a given scan line, the rendering procedure is executed once for each polygon that projects onto that scan line. The higher-level graphics software is responsible for orienting the objects with respect to the viewer, for making perspective calculations, for scaling, and for determining the amount of light that falls on each polygon vertex.

The 16-bit pixel format is used, giving ample resolution for color shading: 2^6 intensity values for red, 2^6 intensity values for green, and 2^4 intensity values for blue. Example 12-15 shows how to set the pixel size. For hidden-surface elimination, the Z-buffer (or depth buffer) technique is employed, each Z value having a resolution of 16-bits.

Because the examples presented here use almost all of the registers of the processor, the registers are given symbolic names, as defined by Example 12-16. In a real application, it is likely that some of the inputs to the rendering procedure would be passed in floating-point registers instead of the integer registers employed here. The register allocation shown in Example 12-16 simplifies the examples by avoiding the need to use any register for multiple purposes.

```
// SET PIXEL SIZE TO 16
    ld.c      psr,     Ra       // Work on psr
    andnoth   0x00C0,  Ra,   Ra // Clear PS
    orh       0x0040,  Ra,   Ra // PS = 16-bit pixels
    st.c      Ra,      psr      //
```

Example 12-15. Setting Pixel Size

```
// REGISTER DEFINITIONS FOR RENDERING PROCEDURE
//      INTEGER LOCALS
    Ra   = r4   // Temporary
    Rb   = r5   // Temporary
    Rc   = r6   // Temporary
    Rd   = r7   // Temporary
//      INTEGER INPUTS
    X1   = r16  // X coord of starting point of line seg in pixels
    dX   = r17  // Width of scan line segment in number of pixels
    ZBP  = r18  // Z-buffer pointer to the current line segment
    Z1   = r19  // Initial Z value, fixed-point 16.16 format
    mZ   = r20  // Z slope, fixed-point 16.16 format
    FBP  = r21  // Graphics frame buffer pointer to the current line seg
    Red1 = r22  // Initial red intensity, fixed-point 6.10 format, + .5
    Grn1 = r23  // Initial green intensity, fixed-point 6.10 format, + .5
    Blu1 = r24  // Initial blue intensity, fixed-point 6.10 format, + .5
    mR   = r25  // Red slope, fixed-point 6.10 format
    mG   = r26  // Green slope, fixed-point 6.10 format
    mB   = r27  // Blue slope, fixed-point 6.10 format
//      REAL LOCALS
    aZ   = f2   // Accumulated Z values
    aZh  = f3   //
    iZ1  = f4   // Z interpolant, coefficient 1.0
    iZ1h = f5   //
    iZ3  = f6   // Z interpolant, coefficient 3.0
    iZ3h = f7   //
    oldz = f8   // Original values from the Z-buffer
    newz = f10  // New Z-buffer values
    newzh = f11 //
    newi = f12  // New pixel values
    iR   = f14  // Red interpolant, coefficient 4.0
    iRh  = f15  //
    aR   = f16  // Accumulated red intensities
    aRh  = f17  //
    iG   = f18  // Green interpolant, coefficient 4.0
    iGh  = f19  //
    aG   = f20  // Accumulated green intensities
    aGh  = f21  //
    iB   = f22  // Blue interpolant, coefficient 4.0
    iBh  = f23  //
    aB   = f24  // Accumulated blue intensities
    aBh  = f25  //
    lZmask   = f26  // left-end Z mask
    lZmaskh  = f27  //
    rZmask   = f28  // right-end Z mask
    rZmaskh  = f29  //
```

Example 12-16. Register Assignments

12.13.1 Distance Interpolation

To perform hidden surface elimination at each pixel, the rendering routine first interpolates the value of Z associated with each pixel. Distance interpolation consists of calculating the slope of Z over the given line segment, then increasing the Z value of each successive pixel by that amount, starting from *X1*. The width of the line segment in pixels is ...

$$dX = X2 - X1$$

Calculate the reciprocal of *dX*:

$$RdX = 1/dX$$

The value of *dX* is used several times as a divisor. It is most efficient to calculate its reciprocal once, then, instead of dividing by *dX*, multiply by *RdX*. The slope of Z is ...

$$mZ = (Z2 - Z1)*RdX$$

Because each polygon is a plane, the value of *mZ* is constant for all scan lines that intersect the polygon's projection; therefore, *mZ* needs to be calculated only once for each polygon. Example 12-21 assumes that *dX* and *mZ* have already been calculated, and all that remains is to apply *mZ* to successive pixels. Let *Z(Xn)* be the Z value at pixel *Xn*. Then ...

$$Z(X1) = Z1$$
$$Z(X1 + 1) = Z1 + mZ$$
$$Z(X1 + 2) = Z1 + 2*mZ$$
$$\vdots$$
$$Z(X1 + N) = Z1 + N*mZ$$
$$\vdots$$
$$Z(X1 + dX) = Z1 + dX*mZ = Z(X2)$$

Figure 12-1 illustrates this Z-value interpolation.

The **faddz** instruction helps to perform the above calculations 64 bits at a time. Because a Z value is 16 bits wide, Example 12-21 operates on the Z buffer in groups of four. The **faddz** instruction, however, treats the interpolation values (*N*mZ*) as 32-bit fixed-point numbers; therefore, two **faddz** instructions are executed for each group of four pixels. Because of the way the **faddz** shifts the MERGE register, the first **faddz** corresponds to even-numbered pixels, while the second corresponds to odd-numbered pixels. Instead of starting with the value for the first pixel (*Z(X1)*) and adding *mZ* to each pixel to produce the value for the next pixel, the example procedure starts with the values for the first two even-numbered pixels and adds 1**mZ* to each of these values to produce the values for

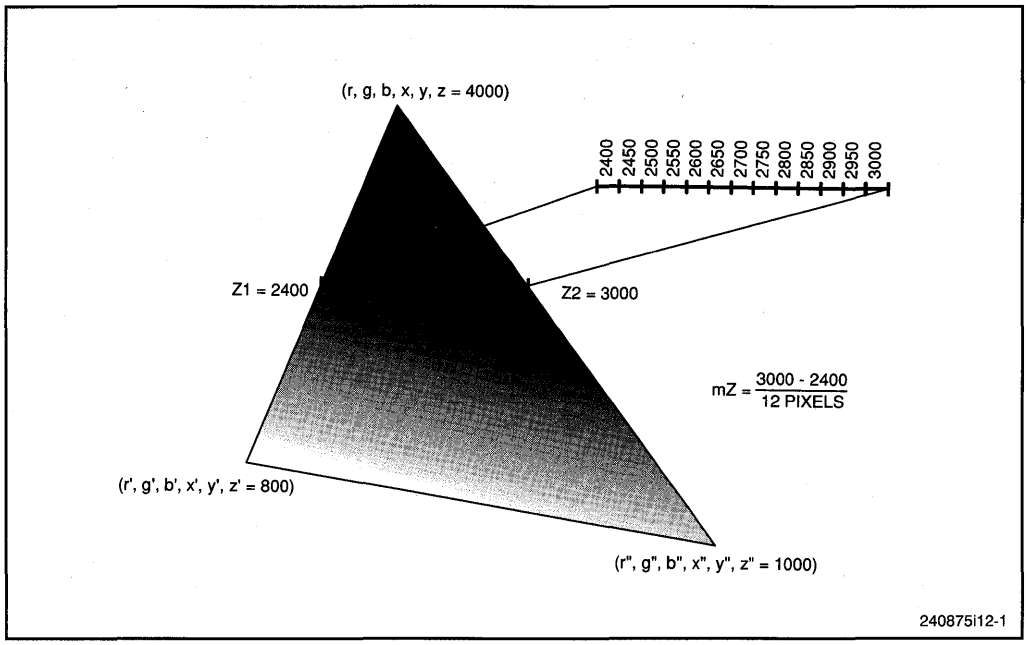


Figure 12-1. Z-Buffer Interpolation

the adjacent odd-numbered pair. Adding 3**mZ* to each of the Z values of an odd-numbered pair produces the values for the next even-numbered pair. Figure 12-2 shows one way of constructing the operands before starting the distance interpolations. (The initial value given to *fsrc1* depends on the alignment of the first pixel.) Table 12-1 helps to visualize the process.

After two **faddz** instructions, the MERGE register holds the Z values for four adjacent pixels (in the correct order). The **form** instruction copies MERGE into one of the 64-bit floating-point registers, because the MERGE register cannot be directly accessed by **pst.d**.

The same register is used as both *fsrc1* and *fdest* in all **faddz** instructions. This register serves to accumulate Z values for successive pixels; therefore, it is called an *accumulator*. The registers used as *fsrc2* are called *interpolants*. The code in Example 12-17 constructs the interpolants; it needs to be executed only once for each polygon.

12.13.2 Color Interpolation

To determine the RGB color intensities at each pixel, the rendering routine interpolates between the color intensities at the end points. (This rendering technique is called "Gouraud shading" after H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, C-20(6), June 1971, pp. 623-628.) Let the symbol C (color)

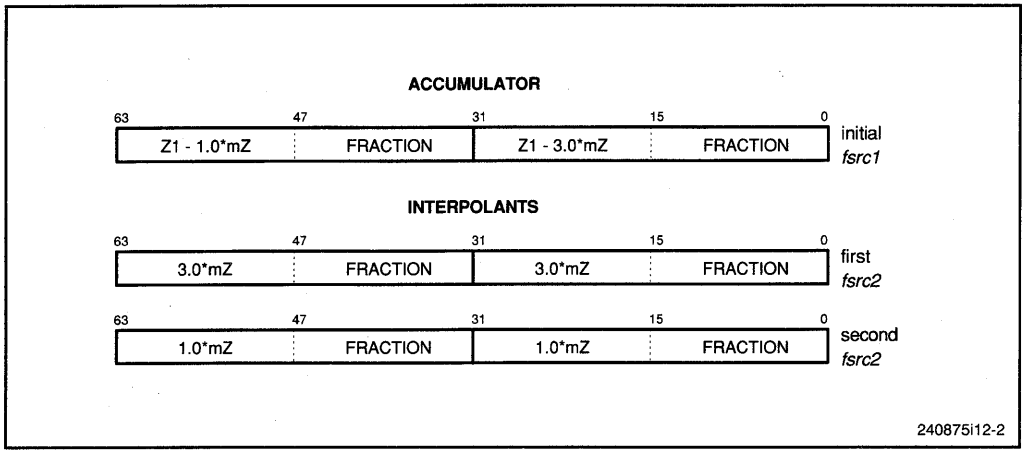


Figure 12-2. faddz Operands

represent either R (red), G (green), or B (blue). Color interpolation consists of calculating the slope of C over the given line segment, then increasing the C values of each successive pixel by that amount, starting from the values for *X1*. This must be done for C=R, C=G, and C=B. The slope of C is ...

$$mC = (C2 - C1)*RdX$$

... where $RdX = 1/dX$

The value of *mC* is constant for all scan lines that intersect a given pair of polygon edges; therefore *mC* needs to be calculated only once for each such pair. Example 12-21 assumes that *mC* has already been calculated for all colors, and all that remains is to apply *mC* to successive pixels. Let *C(Xn)* be a C value at pixel *Xn*. Then ...

$$C(X1) = C1$$
$$C(X1 + 1) = C1 + mC$$
$$C(X1 + 2) = C1 + 2*mC$$
$$\vdots$$
$$C(X1 + N) = C1 + N*mC$$
$$\vdots$$
$$C(X1 + dX) = C1 + dX*mC = C(X2)$$

Figure 12-3 illustrates Gouraud shading of a triangle.

The **faddp** instruction performs the above calculations 64 bits at a time. Because a pixel is 16 bits wide, Example 12-21 operates on pixels in groups of four. Instead of starting with the value for the first pixel (*C(X1)*) and adding *mC* to each pixel to produce the

Table 12-1. faddz Visualization

Operands	63-32	31-0	MERGE Register			
			63-48	47-32	31-16	15-0
src1	−1.0	−3.0				
src2	3.0	3.0				
rdest/src1	2.0	0.0	2		0	
src2	1.0	1.0				
rdest/src1	3.0	1.0	3	2	1	0
src2	3.0	3.0				
rdest/src1	6.0	4.0	6		4	
src2	1.0	1.0				
rdest/src1	7.0	5.0	7	6	5	4
src2	3.0	3.0				
rdest/src1	10.0	8.0	10		8	
src2	1.0	1.0				
rdest/src1	11.0	9.0	11	10	9	8
src2	3.0	3.0				
rdest/src1	14.0	12.0	14		12	
src2	1.0	1.0				
rdest	15.0	11.0	15	14	13	12

NOTE: Because the values of *Z1* and *mZ* are constant for each loop through the rendering routine, the numbers shown here are the values of the coefficient *N*, where the actual operands have the values *Z1* + *N*mZ*. For each execution of **faddz**, *src1* is the same as *rdest* of the prior **faddz**. After every two **faddz** instructions, a **form** instruction empties the MERGE register.

value for the next pixel, the example procedure starts with the values for the first four pixels and adds 4**mC* to each group of four to produce the values for the next four. Three **faddp** instructions are executed for each group of four pixels. The first increments the blue values; the second, green; the third, red. Figure 12-4 shows one way of constructing the operands for each color before starting the color interpolations. (The initial value given to *fsrc1* depends on the alignment of the first pixel.)

Setup of the accumulator and interpolants is similar to that of the Z-buffer. The code in Example 12-18 constructs the interpolants; it needs to be executed only once for each pair of edges in each polygon.

```
// CONSTRUCT INTERPOLANTS iZ1 AND iZ3 GIVEN mZ
    ixfr        mZ,     iZ1         // Join each half in 64-bit register
    shl         1,      mZ,     Ra  // Ra = 2*mZ
    adds        Ra,     mZ,     Ra  // Ra = 3*mZ
    ixfr        Ra,     iZ3         // Join each half in 64-bit register
    fmov.ss     iZ1,    iZ1h        // Join each half in 64-bit register
    fmov.ss     iZ3,    iZ3h        // Join each half in 64-bit register
```

Example 12-17. Construction of Z Interpolants

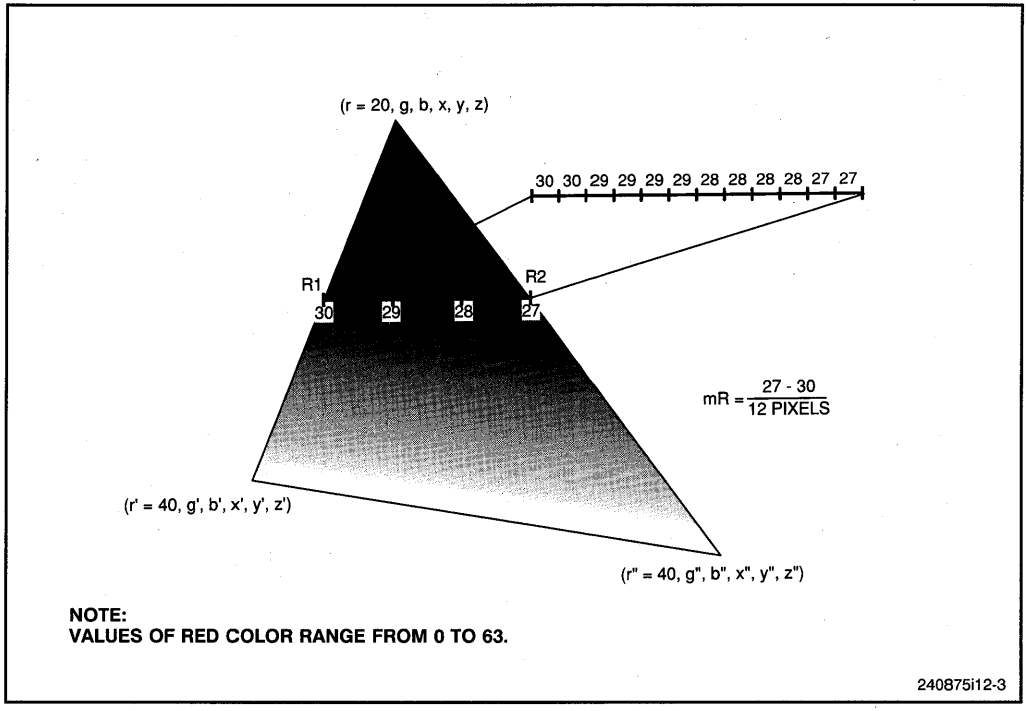


Figure 12-3. Pixel Interpolation for Gouraud Shading

12.13.3 Boundary Conditions

i860 microprocessors operate on 64-bit quantities that are aligned on 8-byte boundaries. The code in this example takes full advantage of this design, handling four 16-bit pixels in each loop. However, if the first or last pixel of a line segment is not on an 8-byte boundary, two kinds of special considerations are required:

1. Masking of Z values near the end points.

2. Initialization of the accumulators.

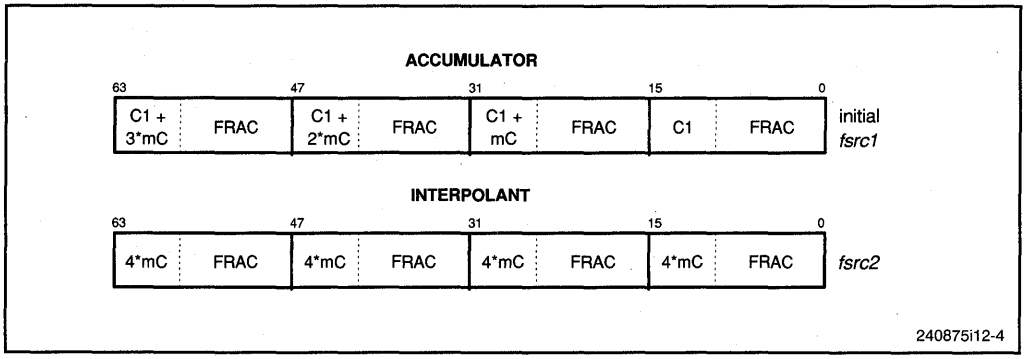


Figure 12-4. faddp Operands

```
// CONSTRUCT INTERPOLANTS iR, iG, iB GIVEN mR, mG, mB
    shl         18,     mR,     Ra  // Multiply each color slope by four,
    shl         18,     mG,     Rb  //  then shift by 16 to put
    shl         18,     mB,     Rc  //  significant bits into high half
    shr         16,     Ra,     mR  // Return significant 16 bits
    shr         16,     Rb,     mG  // to low-order half. Any sign bits
    shr         16,     Rc,     mB  //  in high-order half are gone.
    or          mR,     Ra,     Ra  // Join 16-bit quarters
    or          mG,     Rb,     Rb  //  in 32-bit register
    or          mB,     Rc,     Rc  //
    ixfr        Ra,     iR          // Join 32-bit halves
    ixfr        Rb,     iG          //  in 64-bit register
    ixfr        Rc,     iB          //
    fmov.ss     iR,     iRh         //
    fmov.ss     iG,     iGh         //
    fmov.ss     iB,     iBh         //
```

Example 12-18. Construction of Color Interpolants

12.13.3.1 Z-BUFFER MASKING

When either the first or last pixel of the line segment is not at an 8-byte boundary, the rendering procedure must mask the first or last set of new Z-buffer values (**newz**) so that the Z-buffer and the frame buffer are not erroneously updated. Sometimes both the first and last pixels are in the same 4-pixel set, in which case either one may not be on an 8-byte boundary. A function that looks up and calculates masks is shown in Example 12-19.

Because the value 0xFFFF is used for masking, the Z-buffer is initialized with 0xFFFE, so that the **fzchks** instruction always finds the mask to be greater than any Z-buffer contents.

12.13.3.2 ACCUMULATOR INITIALIZATION

When the first pixel of the line segment is not at an 8-byte boundary, initial values placed in the accumulators (*aZ, aB, aG,* and *aR*) must be selected so that *Z1, Red1,*

```
.macro zmask l_align, r_align, Rx, Ry
//  l_align -- left-end alignment in two-byte units
//  r_align -- right-end alignment in two-byte units
//  Rx, Ry  -- scratch registers
//          Left-end OR masks                Right-end OR masks
//  Input         Output          Input           Output
// l_align        lZmask         r_align          rZmask
//    0     0000 0000 0000 0000     0       FFFF FFFF FFFF 0000
//    1     0000 0000 0000 FFFF     1       FFFF FFFF 0000 0000
//    2     0000 0000 FFFF FFFF     2       FFFF 0000 0000 0000
//    3     0000 FFFF FFFF FFFF     3       0000 0000 0000 0000
// If the first and last pixels are contained in the same 64-bit
// aligned set, then lZmask = lZmask OR rZmask.
.endm
```

Example 12-19. Z Mask Procedure

Grn1, and *Blu1* correspond to the correct pixel. The desired result is that shown by Table 12-2. However, each value is a composite of two terms: one that is constant for each edge pair (*n*mZ, n*mR, n*mG, n*mB*) and one that can vary with each scan line (*Z1, Red1, Grn1, Blu1*). The example assumes that the constant values have all been calculated and stored in a memory table of the format shown by Table 12-3. At the beginning of each line segment the values appropriate to the alignment of the line segment are retrieved from the table and added to the initial Z and color values, as shown in Example 12-20.

12.13.4 The Inner Loop

Once the proper preparations have been made, only a minimal amount of code is needed to render each scanline segment of a polygon. The code shown in Example 12-21 operates on four pixels in each loop. The left and right ends of the line segment go through different logic paths so that the Z-buffer masks can be applied by the **form** instruction. All the interior points are handled by the tight inner loop.

Table 12-2. Accumulator Initial Values

Alignment	Initial Z Accumulator Values	
0 2 4 6	Z1 – 1*mZ Z1 – 2*mZ Z1 – 3*mZ Z1 – 4*mZ	Z1 – 3*mZ Z1 – 4*mZ Z1 – 5*mZ Z1 – 6*mZ

Alignment	Initial Color Accumulator Values C = R, G, B			
0 2 4 6	C1 – 1*mC C1 – 2*mC C1 – 3*mC C1 – 4*mC	C1 – 2*mC C1 – 3*mC C1 – 4*mC C1 – 5*mC	C1 – 3*mC C1 – 4*mC C1 – 5*mC C1 – 6*mC	C1 – 4*mC C1 – 5*mC C1 – 6*mC C1 – 7*mC

Table 12-3. Accumulator Initialization Table

Alignment	Table Values			
	*mZ	*mR	*mG	*mB
0	−1, −3	−1, −2, −3, −4	−1, −2, −3, −4	−1, −2, −3, −4
2	−2, −4	−2, −3, −4, −5	−2, −3, −4, −5	−2, −3, −4, −5
4	−3, −5	−3, −4, −5, −6	−3, −4, −5, −6	−3, −4, −5, −6
6	−4, −6	−4, −5, −6, −7	−4, −5, −6, −7	−4, −5, −6, −7

The controlling variable **dX** is zero-relative and is expressed as a number of pixels. The value of **dX** also indicates alignment of the end-points with respect to the 4-pixel groups. Unaligned left-end pixels are subtracted from **dX** before entering the inner loop; therefore, subsequent values of **dX** indicate the alignment of the right end. A value that is 3 mod 4 indicates that the right end is aligned, which explains the test for a value of –5 near the end of the loop (–5 mod 4 = 3). The fact that the value –5 is loaded into register **Rb** on every execution of the loop does not represent a programming inefficiency, because there is nothing else for the core unit to do at that point anyway.

12.14 GRAPHICS TRANSFORMATION

Example 12-22 transforms each of a list of 1×4 row vectors with a 4×4 matrix. This calculation is typical of 3-D graphics transformations where objects are represented as a polygon mesh. Typically, each element in the object database is associated with a list of all the vertices of the polygons in the mesh that forms the object.

Interactive graphics systems that provide highly realistic images generally require that the representation of an object be transformed in the following ways:

- Local scaling—making objects narrower or wider in any dimension.
- Rotation—a circular movement.
- Translation—linear relocation.
- Perspective projection—making parallel lines converge to give the impression of depth.
- Overall scaling—zooming in or out.

Any transformation of an object, reduces to a transformation of every vertex in that object's vertex list. The object database may also have lists of polygons and lists of edges; however, these lists are not used by the transformation procedure.

The theory of graphics is reviewed here only to the degree necessary to explain the example procedures. For a more detailed study, refer to a graphics text, such as:

- James D. Foley and Andries Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982.
- David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York, 1976.

```
// ACCUMULATOR INITIALIZATION TABLE
    .data; .align .double
acc_init_tab::  .double [16] 0
    .dsect
aBi:    .double     // Four initial 16-bit blue values
aGi:    .double     // Four initial 16-bit green values
aRi:    .double     // Four initial 16-bit red values
aZi:    .double     // Two initial 32-bit Z values
    .end
    .text
// INITIALIZE ACCUMULATORS
.macro acc_init  Lalign, Rtab, Rx, Ry, Fx, Fxh
//  Lalign -- left-end alignment (0..3) in two-byte units
//  Rtab   -- register to use for addressing the table
//  Rx, Ry, Fx, Fxh -- scratch registers
    mov         acc_init_tab,   Rtab    //
    shl         5,      Lalign, Lalign  // Multiply by row width
    adds        Lalign, Rtab,   Rtab    // Index row corresponding
                                        //   to alignment
    fld.d       aZi(Rtab),      aZ      // Z
    ixfr        Z1,     Fx              // Z
    fld.d       aRi(Rtab),      aR      // R--Load constant values
    shl         16,     Red1,   Rx      // R--Shift start val to hi-order
    fmov.ss     Fx,     Fxh             // Z
    shr         16,     Rx,     Ry      // R--Red1 stripped of sign bits
    fiadd.dd    Fx,     aZ,     aZ      // Z
    or          Rx,     Ry,     Ry      // R--Form (Red1,Red1)
    ixfr        Ry,     Fx              // R--Put in 64-bit register
    fld.d       aGi(Rtab),      aG      // G
    shl         16,     Grn1,   Rx      // G
    fmov.ss     Fx,     Fxh             // R--Form (Red1,Red1,Red1,Red1)
    shr         16,     Rx,     Ry      // G
    fiadd.dd    Fx,     aR,     aR      // R--Add variables to constants
    or          Rx,     Ry,     Ry      // G
    ixfr        Ry,     Fx              // G
    fld.d       aBi(Rtab),      aB      // B
    shl         16,     Blu1,   Rx      // B
    fmov.ss     Fx,     Fxh             // G
    shr         16,     Rx,     Ry      // B
    fiadd.dd    Fx,     aG,     aG      // G
    or          Rx,     Ry,     Ry      // B
    ixfr        Ry,     Fx              // B
    fmov.ss     Fx,     Fxh             // B
    fiadd.dd    Fx,     aB,     aB      // B
.endm
```

Example 12-20. Accumulator Initialization

```
//  RENDERING PROCEDURE
//      16-bit pixels, 16-bit Z-buffer
    and         3,      X1,     Ra  // Determine alignment of start-point
    acc_init  Ra, Rb, Rc, Rd, Fa, Fah   // Initialize accumulators
    subs        4,      Ra,     Rb  // 4 - alignment
    subs        dX,     Rb,     dX  // Adjust dX by X1 alignment
    // If dX <= 0, then right end is in same set as left end
    and         3,      dX,     Rb  // Determine alignment of right end
    zmask       Ra, Rb, Rc, Rd      // Prepare left- and right-end masks
left_end::  // Handle boundary conditions
    d.faddz     aZ,     iZ3,    aZ  // Interpolate 2 even Z values
      adds      -8,     FBP,    FBP // Anticipate autoincrement
    d.faddz     aZ,     iZ1,    aZ  // Interpolate 2 odd Z values
      adds      -8,     ZBP,    ZBP // Anticipate autoincrement
    d.form      1Zmask, newz        // Mask 4 new Z values
      fld.d     8(ZBP), oldz        // Fetch 4 old Z values
    d.faddp     aB,     iB,     aB  // Interpolate 4 blue intensities
      mov       -4,     Ra          // Loop increment: 4 pixels
    d.faddp     aG,     iG,     aG  // Interpolate 4 green intensities
      adds      -4,     dX,     dX  // Prepare dX for bla at end of loop
    d.faddp     aR,     iR,     aR  // Interpolate 4 red intensities
      bla       Ra,     dX,     L1  // Initialize LCC
    d.form      f0,     newi        // Move 4 new pixels to 64-bit reg
      adds      5,      dX,     r0  // Any whole sets (dX < -5)?
L1: d.fzchks    oldz,   newz,  newz // Mark closer points in PM[7..4]
      bc        short_segment       // Get out now if no whole set
    d.fnop                          //
      fld.d     16(ZBP),    oldz    // Fetch 4 old Z values
inner_loop:: // Handle all interior points
    d.faddz     aZ,     iZ3,    aZ  // Interpolate 2 even Z values
      nop                           //
    d.faddz     aZ,     iZ1,    aZ  // Interpolate 2 odd Z values
      fst.d     newz,   8(ZBP)++    // Update Z buf from prior loop
    d.form      f0,     newz        // Move 4 new Z values to 64-bit reg
      nop                           //
    d.fzchks    f0,     f0,     f0  // Shift PM[7..4] to PM[3..0]
      mov       -5,     Rb          // -5 mod 4 = 3, aligned right end
    d.faddp     aB,     iB,     aB  // Interpolate 4 blue intensities
      pst.d     newi,   8(FBP)++    // Store pixels indicated by PM[3..0]
    d.faddp     aG,     iG,     aG  // Interpolate 4 green intensities
      xor       Rb,     dX,     r0  // Are we at an aligned right end?
    d.faddp     aR,     iR,     aR  // Interpolate 4 red intensities
      bc        aligned_end         // Taken if at an aligned right end
    d.form      f0,     newi        // Move 4 new pixels to 64-bit reg
      bla       Ra, dX, inner_loop  // Loop if not at end of line segment
    d.fzchks    oldz,   newz,  newz // Mark closer points in PM[7..4]
      fld.d     16(ZBP),    oldz    // Fetch 4 old Z values for next loop
// End of inner_loop.  Right end not aligned
```

Example 12-21. 3-D Rendering (1 of 2)

```
right_end:: // Handle boundary conditions
    d.faddz     aZ,     iZ3,    aZ  // Interpolate 2 even Z values
      nop                           //
    d.faddz     aZ,     iZ1,    aZ  // Interpolate 2 odd Z values
      fst.d     newz,   8(ZBP)++    // Update Z buf from prior loop
    d.form      rZmask, newz        // Mask 4 new Z values
      nop                           //
    d.fzchks    f0,     f0,     f0  // Shift PM[7..4] to PM[3..0]
      nop                           //
    d.faddp     aB,     iB,     aB  // Interpolate 4 blue intensities
      pst.d     newi,   8(FBP)++    // Store pixels indicated by PM[3..0]
    d.faddp     aG,     iG,     aG  // Interpolate 4 green intensities
      nop                           //
    d.faddp     aR,     iR,     aR  // Interpolate 4 red intensities
      nop                           //

aligned_end::  // No special boundary conditions
    d.form      f0,     newi        // Move 4 new pixels to 64-bit reg
      br        wrap_up             //
    d.fzchks    oldz,   newz,   newz // Mark closer points in PM[7..4]
      nop                           //

short_segment::
    d.fnop                          //
      adds      8,      dX,     r0  // Is right end in same set as left?
    d.fnop                          //
      bnc.t     right_end           // Branch taken if no.
    d.fnop                          //
      fld.d     16(ZBP),    oldz    // Fetch 4 old Z values

wrap_up:: // Store the unstored and leave dual mode.
    fzchks      f0,     f0,     f0  // Shift PM[7..4] to PM[3..0]
      fst.d     newz,   8(ZBP)++    // Update Z buf from prior loop
    fnop
      pst.d     newi,   8(FBP)++    // Store pixels indicated by PM[3..0]
```

Example 12-21. 3-D Rendering (2 of 2)

```
// GRAPHICS TRANSFORM
//   Multiplies each element of a list of vertices in
//   homogeneous coordinates by a single transformation matrix.

// Parameters
    Vp=r16  // Pointer to single-precision input vertex list
    Mp=r17  // Pointer to single-precision xform matrix M
    Up=r18  // Pointer to single-precision output vertex list
    VN=r19  // Number of vertices (assumed >= 1)
// C syntax: xform (input_list, matrix, output_list, vertex_count);
    XW=f2   // Temporary

// Tranformation matrix M. Assumes that M is stored row-major, as in C.
//  col 1      col 2      col 3     col 4
    M11=f4;   M12=f5;   M13=f6;  M14=f7    // row 1
    M21=f8;   M22=f9;   M23=f10; M24=f11   // row 2
    M31=f12;  M32=f13;  M33=f14; M34=f15   // row 3
    M41=f16;  M42=f17;  M43=f18; M44=f19   // row 4

// Input vertex V
//   Even    Odd     --  Ping-pong input registers
    Vx=f20; VX=f28  // x component of vertex
    Vy=f21; VY=f29  // y component of vertex
    Vz=f22; VZ=f30  // z component of vertex
    Vw=f23; VW=f31  // w component of vertex

// Transformed vertex U
    Ux=f24  // x component of transformed vertex
    Uy=f25  // y component of transformed vertex
    Uz=f26  // z component of transformed vertex
    Uw=f27  // w component of transformed vertex

// This procedure modifies all floating-point registers. Save on the
// stack any floating point registers that the caller assumes preserved.

// Assume that adder pipe and T register contain ''safe'' values (values
//   that will not cause a source exception) from calling procedure or
//   from prior calls to this procedure.  If not, execute the following:
// pfmul.ss f0, f0, f0; pfmul.ss f0, f0, f0; pfmul.ss f0, f0, f0;
// pfadd.ss f0, f0, f0; pfadd.ss f0, f0, f0; pfadd.ss f0, f0, f0;
// i2apt f0, f0, f0;

xform:: ; _xform::
  fld.q  0(Vp), Vx      // Load first even vertex
  fld.q  0(Mp), M11     // Load transformation matrix M, row 1
  fld.q 48(Mp), M41     // Load row 4

// For brevity in pipeline diagrams, let a symbol of the form ''apq''
//  represent Va*Mpq. For example, y21 represents Vy*M21.
.align .double
```

Example 12-22. Graphics Transform (1 of 5)

```
//                        //       Multiplier       -- Adder ---
//                        //         Stages            Stages
//                        // KR KI *1  *2  *3  T   +1   +2   +3   Rslt  Ins#
d.i2pt.ss   Vy, f0,  f0 // -- y  --  --  --  --  --   --   --   --    (aa)
  fld.q 32(Mp), M31     // Load row 3
d.m12apm.ss Vx, M11, f0 // -- y  x11 --  --  --  --   --   --   --    (01)
  fld.q 16(Mp), M21     // Load row 2
d.m12apm.ss Vw, M41, f0 // -- y  w41 x11 --  --  --   --   --   --    (02)
  adds     -16, Up,  Up // Compensate for autoincrement store
d.m12apm.ss Vx, M12, f0 // -- y  x12 w41 x11 --  --   --   --   --    (03)
  nop
d.m12ttpa.ss Vw,M42, f0 // -- y  w42 x12 w41 x11 --   --   --   --    (04)
  nop

d.m12tpm.ss Vy, M21, f0 // -- y  y21 w42 x12 --  x11+ --   --   --    (05)
  nop                   //                     w41
d.i2ap1.ss  f0, M22, f0 // -- y  y22 y21 w42 x12 --   x11+ --   --    (06)
  nop                   //                          w41
d.m12tpm.ss Vx, M13, f0 // -- y  x13 y22 y21 --  x12+ --   x11+ --    (07)
  nop                   //                     w42       w41

d.m12apm.ss Vz, M31, f0 // -- y  z31 x13 y22 --  x11+ x12+ --   --    (08)
  nop                   //                     y21+ w42
                        //                     w41
d.m12ttpa.ss Vw,M43, f0 // -- y  w43 z31 x13 y22 --   x11+ x12+ --    (09)
  nop                   //                          y21+ w42
                        //                          w41

d.m12ttpa.ss Vz,M32, f0 // -- y  z32 w43 z31 x13 x12+ --   x11+ --    (10)
  adds -2, VN, VN       //                     y22+       y21+
  // -1 indicates done  //                     w42        w41

d.m12apm.ss Vx, M14, f0 // -- y  x14 z32 w43 x13 x11+ x12+ --   --    (11)
  fld.d  16(Vp)++, VX   //                     y21+ y22+ \
  // Next vertex (odd)  //                     z31+ w42
                        //                     w41
d.m12tpm.ss Vy, M23, f0 // -- y  y23 x14 z32 --  x13+ x11+ x12+ --    (12)
  nop                   //                     w43  y21+ y22+
                        //                          z31+ w42
                        //                          w41
d.m12apm.ss Vw, M44, f0 // -- y  w44 y23 x14 --  x12+ x13+ x11+ --    (13)
  bc end_even_xform     //                     y22+ w43  y21+
// Avoid using data     //                     z32+      z31+
// beyond end of list   //                     w42       w41

d.i2apt.ss VY, M24, Ux  // -- Y  y24 w44 y23 x14 --   x12+ x13+ x11+  (14)
  fld.d 8(Vp), VZ       //                          y22+ w43  y21+
  // Rest of vertex     //                          z32+      z31+
                        //                          w42       w41
d.m12apm.ss Vz, M33, f0 // -- Y  z33 y24 w44 x14 x13+ --   x12+ --    (15)
  nop                   //                     y23        y22+
                        //                     w43        z32+
                        //                                w42
d.m12tpm.ss Vz, M34, Uy // -- Y  z34 z33 y24 --  x14+ x13+ --   x12+  (16)
  nop                   //                     w44  y23+          y22+
                        //                          w43           z32+
                        //                                        w42
```

Example 12-22. Graphics Transform (2 of 5)

```
xformloop: //----------------------DDD VERTEX-------------------------//
//                     // KR KI *1  *2  *3  T   +1    +2    +3    Rslt  Ins#
d.m12ttpa.ss VX,M11, f0 // -- Y  X11 z34 z33 y24 --    x14+  x13+  --    (01)
  nop                  //                        w44   y23+
                       //                              w43
d.m12apm.ss VW, M41, f0 // -- Y  W41 X11 z34 y24 x13+  --    x14+  --    (02)
  nop                  //                  y23+        w44
                       //                  z33+
                       //                  w43
d.m12tpm.ss VX, M12, XW // -- Y  X12 W41 X11 --  y24+  x13+  --    x14+  (03)
  nop                  //                  z34   y23+        w44
                       //                        z33+
                       //                        w43
d.m12ttpa.ss VW,M42, f0 // -- Y  W42 X12 W41 X11 --    y24+  x13+  --    (04)
  nop                  //                        z34   y23+
                       //                              z33+
                       //                              w43
d.m12tpm.ss VY, M21, Uz // -- Y  Y21 W42 X12 --  X11+ --    y24+  x13+  (05)
  nop                  //                  W41         z34   y23+
                       //                                    z33+
                       //                                    w43
d.i2ap1.ss XW, M22, f0  // -- Y  Y22 Y21 W42 X12 x14+ X11+ --    --    (06)
  nop                  //                  y24+  W41
                       //                  z34+
                       //                  w44
d.m12tpm.ss VX, M13, f0 // -- Y  X13 Y22 Y21 --  X12+ x14+ X11+ --    (07)
  nop                  //                  W42   y24+  W41
                       //                        z34+
                       //                        w44
d.m12apm.ss VZ, M31, f0 // -- Y  Z31 X13 Y22 --  X11+ X12+ x14+ --    (08)
  nop                  //                  Y21+  W42   y24+
                       //                  W41         z34+
                       //                              w44
d.m12ttpa.ss VW,M43, Uw // -- Y  W43 Z31 X13 Y22 --    X11+ X12+ x14+  (09)
  fst.q Ux, 16(Up)++   //                              Y21+  W42   y24+
// Store xformed vertex //                              W41         z34+
                       //                                          w44
d.m12ttpa.ss VZ,M32, f0 // -- Y  Z32 W43 Z31 X13 X12+ --    X11+ --    (10)
  adds -1, VN, VN      //                  Y22+              Y21+
                       //                  W42               W41
d.m12apm.ss VX, M14, f0 // -- Y  X14 Z32 W43 X13 X11+ X12+ --    --    (11)
  fld.d 16(Vp)++, Vx   //                  Y21+  Y22+
  // Next even vertex  //                  Z31+  W42
                       //                  W41
d.m12tpm.ss VY, M23, f0 // -- Y  Y23 X14 Z32 --  X13+ X11+ X12+ --    (12)
  nop                  //                  W43   Y21+  Y22+
                       //                        Z31+  W42
                       //                              W41
d.m12apm.ss VW, M44, f0 // -- Y  W44 Y23 X14 --  X12+ X13+ X11+ --    (13)
  bc end_odd_xform     //                  Y22+  W43   Y21+
// Avoid using data     //                  Z32+        Z31+
// beyond end of list   //                  W42         W41
d.i2apt.ss Vy, M24, Ux  // -- y  Y24 W44 Y23 X14 --    X12+ X13+ X11+  (14)
  fld.d 8(Vp), Vz      //                              Y22+  W43   Y21+
  // Rest of vertex    //                              Z32+        Z31+
                       //                              W42         W41

d.m12apm.ss VZ, M33, f0 // -- y  Z33 Y24 W44 X14 X13+ --    X12+ --    (15)
  nop                  //                  Y23               Y22+
                       //                  W43               Z32+
                       //                                    W42
d.m12tpm.ss VZ, M34, Uy // -- y  Z34 Z33 Y24 --  X14+ X13+ --    X12+   (16)
  nop                  //                  W44   Y23+              Y22+
                       //                        W43               Z32+
                       //                                          W42
```

Example 12-22. Graphics Transform (3 of 5)

```
//-----EVEN VERTEX------// KR KI *1  *2  *3  T   +1   +2   +3   Rslt  Ins#
d.m12ttpa.ss Vx,M11, f0 // -- y  x11 Z34 Z33 Y24 --   X14+ X13+ --    (01)
  nop                   //                       W44  Y23+
                        //                            W43
d.m12apm.ss Vw, M41, f0 // -- y  w41 x11 Z34 Y24 X13+ --   X14+ --    (02)
  nop                   //                       Y23+      W44
                        //                       Z33+
                        //                       W43
d.m12tpm.ss Vx, M12, XW // -- y  x12 w41 x11 --  Y24+ X13+ --   X14+  (03)
  nop                   //                       Z34  Y23+      W44
                        //                            Z33+
                        //                            W43
d.m12ttpa.ss Vw,M42, f0 // -- y  w42 x12 w41 x11 --   Y24+ X13+ --    (04)
  nop                   //                       Z34  Y23+
                        //                            Z33+
                        //                            W43
d.m12tpm.ss Vy, M21, Uz // -- y  y21 w42 x12 --  x11+ --   Y24+ X13+  (05)
  nop                   //                       w41       Z34  Y23+
                        //                                  Z33+
                        //                                  W43
d.i2ap1.ss XW, M22, f0  // -- y  y22 y21 w42 x12 X14+ x11+ --   --    (06)
  nop                   //                       Y24+ w41
                        //                       Z34+
                        //                       W44
d.m12tpm.ss Vx, M13, f0 // -- y  x13 y22 y21 --  x12+ X14+ x11+ --    (07)
  nop                   //                       w42  Y24+ w41
                        //                            Z34+
                        //                            W44
d.m12apm.ss Vz, M31, f0 // -- y  z31 x13 y22 --  x11+ x12+ X14+ --    (08)
  nop                   //                       y21+ w42  Y24+
                        //                       w41       Z34+
                        //                                 W44
d.m12ttpa.ss Vw,M43, Uw // -- y  w43 z31 x13 y22 --   x11+ x12+ X14+  (09)
  fst.q Ux, 16(Up)++    //                            y21+ w42  Y24+
// Store xformed vertex //                            w41       Z34+
                        //                                      W44
d.m12ttpa.ss Vz,M32, f0 // -- y  z32 w43 z31 x13 x12+ --   x11+ --    (10)
  adds -1, VN, VN       //                       y22+      y21+
                        //                       w42       w41
d.m12apm.ss Vx, M14, f0 // -- y  x14 z32 w43 x13 x11+ x12+ --   --    (11)
  fld.d 16(Vp)++, VX    //                       y21+ y22+
  // Next odd vertex    //                       z31+ w42
                        //                       w41
d.m12tpm.ss Vy, M23, f0 // -- y  y23 x14 z32 --  x13+ x11+ x12+ --    (12)
  nop                   //                       w43  y21+ y22+
                        //                            z31+ w42
                        //                       w41
d.m12apm.ss Vw, M44, f0 // -- y  w44 y23 x14 --  x12+ x13+ x11+ --    (13)
  bc end_even_xform     //                       y22+ w43  y21+
// Avoid using data     //                       z32+      z31+
// beyond end of list   //                       w42       w41
d.i2apt.ss VY, M24, Ux  // -- Y  y24 w44 y23 x14 --   x12+ x13+ x11+  (14)
  fld.d 8(Vp), VZ       //                            y22+ w43  y21+
  // Rest of vertex     //                            z32+      z31+
                        //                            w42       w41
d.m12apm.ss Vz, M33, f0 // -- Y  z33 y24 w44 x14 x13+ --   x12+ --    (15)
  br        xformloop   //                       y23       y22+
                        //                       w43       z32+
                        //                                 w42
d.m12tpm.ss Vz, M34, Uy // -- Y  z34 z33 y24 --  x14+ x13+ --   x12+  (16)
  nop                   //                       w44  y23+      y22+
                        //                            w43       z32+
                        //                                      w42
```

Example 12-22. Graphics Transform (4 of 5)

```
end_even_xform::
d.i2apt.ss  f0, M24, Ux // -- -- y24 w44 y23 x14 --   x12+ x13+ x11+  (14)
  nop                   //                          y22+ w43  y21+
                        //                          z32+      z31+
                        //                          w42       w41
d.m12apm.ss Vz, M33, f0 // -- -- z33 y24 w44 x14 x13+ --   x12+ --    (15)
  br end_xformloop      //                          y23       y22+
                        //                          w43       z32+
                        //                                    w42
d.m12tpm.ss Vz, M34, Uy // -- -- z34 z33 y24 --  x14+ x13+ --   x12+  (16)
  nop                   //                          w44  y23+       y22+
                        //                               w43        z32+
                        //                                          w42
end_odd_xform::
d.i2apt.ss f0, M24, Ux  // -- -- Y24 W44 Y23 X14 --   X12+ X13+ X11+  (14)
  fld.d 8(Vp), Vz       //                          Y22+ W43  Y21+
  // Rest of vertex     //                          Z32+      Z31+
                        //                          W42       W41
d.m12apm.ss VZ, M33, f0 // -- -- Z33 Y24 W44 X14 X13+ --   X12+ --    (15)
  nop                   //                          Y23       Y22+
                        //                          W43       Z32+
                        //                                    W42
d.m12tpm.ss VZ, M34, Uy // -- -- Z34 Z33 Y24 --  X14+ X13+ --   X12+  (16)
  nop                   //                          W4Y  423+       Y22+
                        //                               W43        Z32+
                        //                                          W42
end_xformloop::         // Begin exit from dual-instruction mode
m12ttpa.ss  f0,  f0, f0 // -- -- 0   z34 z33 y24 --   x14+ x13+ --    (01)
  nop                   //                          w44  y23+
                        //                               w43
m12apm.ss   f0,  f0, f0 // -- -- 0   0   z34 y24 x13+ --   x14+ --    (02)
  nop                   //                          y23+      w44
                        //                          z33+
                        //                          w43
m12tpm.ss   f0,  f0, XW // -- -- 0   0   0   --  y24+ x13+ --   x14+  (03)
                        //                          z34  y23+       w44
                        //                               z33+
                        //                               w43
m12ttpa.ss  f0,  f0, f0 // -- -- 0   0   0   0   --   y24+ x13+ --    (04)
                        //                               z34  y23+
                        //                                    z33+
                        //                                    w43
m12tpm.ss   f0,  f0, Uz // -- -- 0   0   0   0   0    --   y24+ x13+  (05)
                        //                                    z34  y23+
                        //                                         z33+
                        //                                         w43
i2ap1.ss    XW,  f0, f0 // -- -- 0   0   0   0   x14+ 0    --   --    (06)
                        //                          y24+
                        //                          z34+
                        //                          w44
m12tpm.ss   f0,  f0, f0 // -- -- 0   0   0   0   0    x14+ 0    --    (07)
                        //                               y24+
                        //                               z34+
                        //                               w44
m12apm.ss   f0,  f0, f0 // -- -- 0   0   0   0   0    0    x14+ --    (08)
                        //                                    y24+
                        //                                    z34+
                        //                                    w44
m12ttpa.ss  f0,  f0, Uw // -- -- 0   0   0   0   0    0    0    x14+  (09)
                        //                                         y24+
                        //                                         z34+
                        //                                         w44
//bri r1                // Do bri here, if this code is a subroutine
fst.q Ux, 16(Up)++      // Store last xformed vertex
```

Example 12-22. Graphics Transform (5 of 5)

12.14.1 Representation of Vertices

A point in three-dimensional space is defined by its three coordinates along the X, Y, and Z axes, so that a point *P* is represented by the vector *(x, y, z)*. However, in graphics programming, it is convenient to represent points by *homogeneous coordinates*. In homogeneous coordinates, the point *P* is defined by a four-dimensional vector *(w*x, w*y, w*z, w)*. The additional factor *w* is called the *scaling factor*. To determine the actual coordinates of *P*, the scaling factor has to be divided out, leaving *(x, y, z, 1)*.

The use of homogeneous coordinates has two advantages:

1. The range of numbers represented by *x, y,* and *z* may be greater than the limits imposed by the processor's data types. This would be an advantage if, for example, the coordinates *x, y,* and *z* were each stored as 16-bit integers, but a greater resolution than 64K were desired. This is not an advantage in this example, where the floating-point data types of the i860 architecture provide adequate resolution for high-definition displays over a wide range of magnitudes.

2. The 1×4 vector formed by homogeneous coordinates can be multiplied by a 4×4 matrix. A 4×4 matrix is capable of representing all the transformations of a point.

12.14.2 Graphics Transformation Matrix

The formula for the product of a 1×4 row vector with a 4×4 matrix is:

$$[Vx,Vy,Vz,Vw] * \begin{bmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ M41 & M42 & M43 & M44 \end{bmatrix} = [Ux,Uy,Uz,Uw] = \quad (1)$$

$$\begin{bmatrix} (Vx*M11 + Vy*M21 + Vz*M31 + Vw*M41), \\ (Vx*M12 + Vy*M22 + Vz*M32 + Vw*M42), \\ (Vx*M13 + Vy*M23 + Vz*M33 + Vw*M43), \\ (Vx*M14 + Vy*M24 + Vz*M34 + Vw*M44) \end{bmatrix}$$

The components of a 4×4 matrix, when multiplied by a point in homogeneous representation, do not all have the same effect on that point. Figure 12-5 shows the various functions of different parts of the matrix. One matrix can specify a combination of transformations, depending on the values in the matrix.

The transformation procedure presented in this example assumes that any or all of the possible transformations may be specified in the transformation matrix, and, therefore, it performs the complete matrix multiplication. Some systems achieve high transformation rates by eliminating some capabilities. For example, by eliminating perspective and zooming, the transformation matrix can be reduced to 3×3, with translation performed separately as simple additions. The procedure presented here makes no such simplifying assumptions.

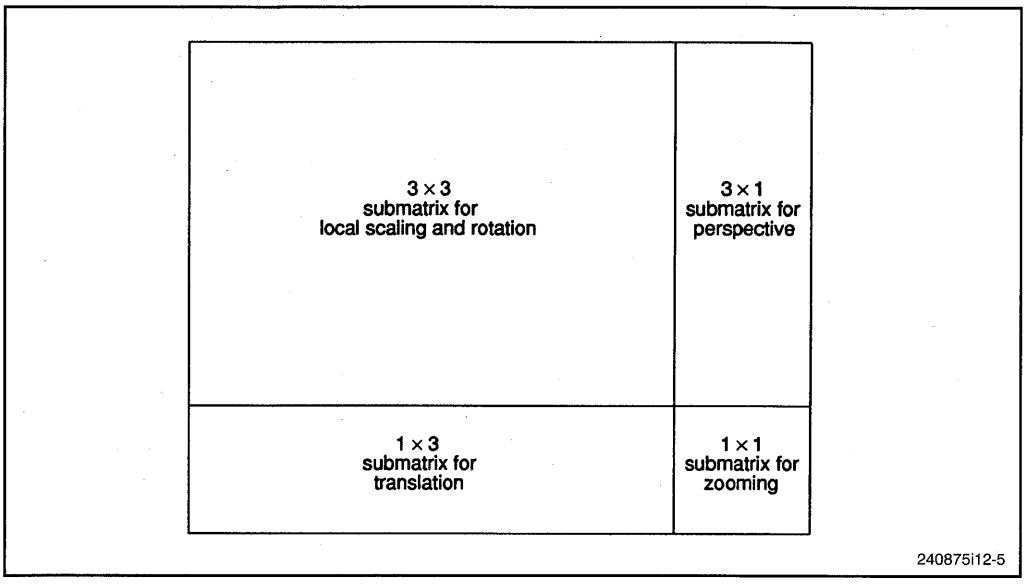


Figure 12-5. Functions of Parts of a Transformation Matrix

12.14.3 Transformation Code Design

As formula (1) indicates, the graphics transformation consists generally of a series of multiplications followed by additions. The dual-operation instructions of the i860 architecture are designed precisely to execute this type of calculation efficiently by using the pipelined adder and multiplier in parallel. For the single-precision operands used, the adder and multiplier pipelines both have three stages.

It is easiest to keep the adder and multiplier pipelines full when the input is treated as a vector and the same operation is applied repeatedly to consecutive elements of the vector. In this example, the vertex list is treated as a vector, and a single transformation is applied to each vertex.

The dual-operation mnemonics used in this example are (refer to Chapter 8):

Mnemonic	Multiplier	Adder	Register Loading
m12apm	*fsrc1***fsrc2*	Mout + Aout	
m12tpm	*fsrc1***fsrc2*	T + Mout	
m12ttpa	*fsrc1***fsrc2*	T + Aout	T←Mout
i2pt	KI**fsrc2*	T + Mout	KI←*fsrc1*
i2apt	KI**fsrc2*	T + Aout	KI←*fsrc1* *T←Mout*
i2ap1	KI**fsrc2*	*fsrc1* + Aout	T←Mout

The *row*column* dot products are calculated in an order determined by the needs of the pipelined instruction, not in the order one would choose for pencil and paper calculation. Instead of calculating *row1*column1*, followed by *row2*column2*, the code interleaves multiplies from *row1*column1* with those from *row2*column2* and *row3*column3*. Thus, when the third stage of the adder ejects a *row1*column1* component, another *row1*column1* component is ready at the output of the multiplier, and the two components can be immediately added by one of the instructions that feeds the adder output back into the adder input.

Note that the adder pipeline is used by only 12 of the 16 cycles of **xformloop**, while the multiplier is busy for all 16 cycles. This idle time in the adder is necessary, because the dot product operation requires only $M-1$ adds for M multiplies.

Dual-instruction mode is used, so that loop control and loading and storing of the vertices can be carried on in parallel with transformation calculations.

To achieve maximum throughput, it is necessary to load the next input vector before finishing with the current one. This is accomplished by alternating between two sets of input registers and by "unrolling" the loop; i.e., processing two inputs in each programmed loop. The section of code entitled "ODD VERTEX" is identical to the section entitled "EVEN VERTEX" except for the input registers used. Even with unrolling, the code fits easily within the 4-Kbyte instruction cache.

The load instructions read one entry beyond the end of the input vertex list; so, to avoid a page fault, storage should be allocated at that location. However, the data at that location are not used in the floating-point pipelines and therefore need not be valid.

The example procedure can be called by C syntax of the form:

```
xform (input_list, matrix, output_list, vertex_count);
```

Note that the same procedure can be used to combine two 4×4 transformation matrices. In this case the call from C has the form:

```
xform (matrix_A, matrix_B, result_matrix, 4);
```

The **result_matrix** has the same transformation effect on a vertex list as **matrix_A** and **matrix_B** would have if applied in succession.

12.14.4 Transformation Performance

At 40 MHz, the number of transforms per second is given by:

```
40 million clocks/sec ÷ 16 clocks/xform = 2.5 million xforms/sec
```

The number of floating-point operations per second is given by:

```
40 million clocks/sec × 28 flops/xform × 1 xform / 16 clocks = 70 Mflops
```

The latency from the first floating-point operation to storing the first result is 27 cycles.

These performance figures are not indicative of total graphics performance, because transformation code is just a small part of a graphics system. Transformation of surface or vertex normals, lighting calculations, factoring out **Vw**, clipping, and rendering must also be considered.

12.15 PERSPECTIVE DIVIDE

After the graphics transform shown in Section 12.14, the scaling factor *w* must be divided out of each of the transformed vertices sooner or later. The basic algorithm for division is shown in Sections 12.2 and 12.3. The perspective divide procedure in Example 12-23 expands on the basic algorithm in two respects:

1. It performs three divisions using one reciprocal calculation.

2. It takes advantage of the floating-point pipelines and dual-operation instructions.

The example is coded as a stand-alone function that treats the entire vertex list as a vector. While this structure is convenient for illustrating the algorithm, it is not necessarily the most efficient structure for any specific application. The overhead for function entry and exit, for loop setup, and for loading and storing the vertices may be reduced by integrating the perspective divide with another procedure, such as transformation.

The instructions at the heart of the loop operate on three vertices at a time, so as to best utilize the floating-point multiplier and adder pipelines. The dual-operation mnemonics used in this example are shown in the following table.

Mnemonic	Multiplier	Adder	Register Loading
m12apm	*fsrc1***fsrc2*	Mout + Aout	
i2s1	KI**fsrc2*	*src1*-Mout	

```
// PIPELINED DIVIDE FOR GRAPHICS PERSPECTIVE

// Inputs: (lists of vertices using single-precision
//          floating-point homogeneous coordinates)
Vp  = r16; // Points to list of input vertices, stored row major.
Up  = r17; // Points to list of output vertices, stored row major.
Len = r18; // Number of vertices (assumed >= 3, and a multiple of 3).

// For each input vertex (x, y, z, w), calculate (x/w, y/w, z/w, 1).
// Because pipelines have three stages, work on three vertices at a time.

// Symbolic register definitions:

G1    = f28; G2    = f29; G3    = f30       // Guesses at reciprocals
W1rcp = f19; W2rcp = f23; W3rcp = f27       // True reciprocals

// Registers to hold coordinates of three vertices:
X1 = f16; Y1 = f17; Z1 = f18; W1 = f19
X2 = f20; Y2 = f21; Z2 = f22; W2 = f23
X3 = f24; Y3 = f25; Z3 = f26; W3 = f27

tmp = f31
Dcr  = r19      // Loop decrement
float1 = f8     // Single-precision 1.0
float2 = f9     // Single-precision 2.0

.data; .align .double; one_two::
one:    .float 1.0              // One for normalized W
two:    .float 2.0              // Two for floating-point divide usage

.text; .align .quad
pdivide::  ;  _pdivide::
  fld.d one_two, float1
  mov   -3, Dcr                 // Set loop decrement
  addu  -16, Up, Up             // Compensate for autoincrement
  fld.q  0(Vp), X1
  addu  -1, Len, Len
  fld.q 16(Vp)++, X2
  bla  Dcr, Len, pdiv_loop      // Initialize LCC
  fld.q 16(Vp)++, X3
```

Example 12-23. Perspective Divide (1 of 2)

```
pdiv_loop::
// First calculate reciprocals for 3 vertices: 1/W1, 1/W2, 1/W3.
// Pipelined floating-point division with 15 bits of precision in result.
//  (Such precision is plenty, as 1/w used only for creating screen-image
//   coordinates, and screen has less than 2K pixels/dimension.)
// Newton-Raphson formula for divide: Gnew = Guess*(2-(Guess*W))
  frcp.ss W1, G1                // Guess1 (approx. 1/w1)
  frcp.ss W2, G2                // Guess2 (1/w2)
  frcp.ss W3, G3                // Guess3

  pfmul.ss W1, G1, f0           // Guess1*W1
  pfmul.ss W2, G2, f0
  pfmul.ss W3, G3, f0

  i2s1.ss   float2, f0, f0      // 2-(Guess1*W1)
  i2s1.ss   float2, f0, f0      // 2-(Guess2*W2)
  i2s1.ss   float2, f0, f0      // 2-(Guess3*W3)

  m12apm.ss  G1, tmp, tmp       // Start calc of Guess*(2-Guess*W1)
  m12apm.ss  G2, tmp, tmp       // Guess*(2-Guess*W2)
  m12apm.ss  G3, tmp, tmp       // Guess*(2-Guess*W3)

//If 23-bit precision divide is required, insert following instructions:
// pfmul.ss W1, G1, G1;    pfmul.ss W2, G2, G2;    pfmul.ss W3, G3, G3
// i2s1.ss float2, f0, f0; i2s1.ss float2, f0, f0; i2s1.ss float2, f0, f0
// m12apm.ss G1, tmp, tmp; m12apm.ss G2, tmp, tmp; m12apm.ss G3, tmp, tmp

  pfmul.ss  Y1, W1rcp, W1rcp    // Start Y1/W1
  // W1rcp has 1/W1 (15 bits precision)
  pfmul.ss  X1, W1rcp, W2rcp    //       X1/W1
  pfmul.ss  Z1, W1rcp, W3rcp    //       Z1/W1

  pfmul.ss W2rcp, Y2, Y1        // Start Y2/W2
  pfmul.ss W2rcp, X2, X1        //       X2/W2
  pfmul.ss W2rcp, Z2, Z1        //       Z2/W2

  pfmul.ss W3rcp, Y3, Y2        // Start Y3/W3
  pfmul.ss W3rcp, X3, X2        //       X3/W3
  pfmul.ss W3rcp, Z3, Z2        //       Z3/W3

// NOTE: The following 3 multiplies could begin a scaling operation,
// in which case src1 and src2, instead of being f0, would multiply
// a scale factor for screen size by Y1, Y2, and Y3

  pfmul.ss f0, f0, Y3
  pfmul.ss f0, f0, X3
    .align .double
d.pfmul.ss f0, f0, Z3           // Initiate dual-instruction mode

d.fmov.ss  float1, W1
  fmov.ss  float1, W2   ;   fst.q X1, 16(Up)++  // New W value is 1.0
  fmov.ss  float1, W3   ;   fst.q X2, 16(Up)++  // Last dual-mode pair
  fst.q X3, 16(Up)++

  fld.q 16(Vp)++, X1
  fld.q 16(Vp)++, X2
  bla Dcr, Len, pdiv_loop
  fld.q 16(Vp)++, X3

  bri r1
  nop
```

Example 12-23. Perspective Divide (2 of 2)

Instruction Set Summary

A

APPENDIX A
INSTRUCTION SET SUMMARY

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**, **fsr**, **bear**, **ccr**, **p0**, **p1**, **p2**, or **p3**
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 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.

Other abbreviations include:

.p — Precision specification **.ss**, **.sd**, or **.dd** (**.ds** not permitted). Refer to Table A-1.

.r — Precision specification **.ss**, **.sd**, **.ds**, or **.dd**. Refer to Table A-1.

.v — **.sd** or **.dd** Refer to Table A-1.

.w — **.ss** or **.dd**. Refer to Table A-1.

.x — **.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)

.y — **.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)

mem.x(address) — The contents of the memory location indicated by *address* with a size of *x*.

port.x(address) — The I/O port indicated by *address* with a size of *x*.

int_vector.x(address) — The interrupt vector with a size of *x* returned from I/O port *address*.

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.

Table A-1. Precision Specification

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

NOTE: 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.

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* < 0 (signed)
CC clear if *isrc2* + *isrc1* ≥ 0 (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 < 0 (signed)
LCC-temp set if isrc2 + isrc1ni ≥ 0 (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 instruction pair
                        ELSE    enter single-instruction mode
                                  for next instruction pair
                        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 (or + 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 (or + 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* (using integer arithmetic; 8-byte operands and destination)
Shift and load MERGE register from *fsrc1* + *fsrc2* as defined in Table A-2

faddz *fsrc1, fsrc2, fdest* ..... **Add with Z Merge**
fdest ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16 and load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

Table A-2. FADDP MERGE Update

Pixel Size (from PS)	Fields Load 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

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

fiadd.w *fsrc1, fsrc2, fdest* ..... **Long-Integer Add**
fdest ← *fsrc1* + *fsrc2* (2's complement integer arithmetic)

fisub.w *fsrc1, fsrc2, fdest* ..... **Long-Integer Subtract**
frdest ← *fsrc1* − *fsrc2* (2's complement integer arithmetic)

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)**
Write back (if modified) the line in data cache that has address (*#const* + *isrc2*)
80860XR: and set tag value to (*#const* + *isrc2*).
80860XP: and invalidate its virtual and physical tags.
Contents of line 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 (*fsrc1* mantissa × *fsrc2* 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 the 64-bit operands as arrays of two 32-bit
fields *fsrc1*(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0)
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 the 64-bit operands as arrays of four 16-bit
fields *fsrc1*(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0)
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 = 1
THEN generate trap with IT set in **psr**
FI

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*)

ldint.x *isrc2, idest* ..... **Load Interrupt Vector**
idest ← *int_vector.x (isrc2)*
NOTE: Not available with the i860 XR CPU

ldio.x *isrc2, idest* ..... **Load I/O**
idest ← *port.x (isrc2)*
NOTE: Not available with the i860 XR CPU

lock ..... **Begin Interlocked Sequence**
Set BL in **dirbase**.
The next data load or store that appears on the bus 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 0xFFFF8000 ≤ *const32* < 0x8000

orh *h%const32*, **r0**, *idest*
or *l%const32, idest, idest*
... otherwise

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-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2* (using integer arithmetic; 8-byte operands and destination)
Shift and load MERGE register from *fsrc1* + *fsrc2* as defined in Table A-2

pfaddz *fsrc1, fsrc2, fdest* ..... **Pipelined Add with Z Merge**
frdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2*
(using integer arithmetic; 8-byte operands and destination)
Shift MERGE right 16 and load fields 31..16 and 63..48 from *fsrc1* + *fsrc2*

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 Greater-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-unit result
last-stage graphics-unit result ← *fsrc1* + *fsrc2* (2's complement integer arithmetic)

pfisub.w *fsrc1, fsrc2, fdest* ..... **Pipelined Long-Integer Subtract**
fdest ← last-stage graphics-unit result
last-stage graphics-unit result ← *fsrc1* − *fsrc2* (2's complement integer arithmetic)

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.y *isrc1*(*isrc2*), *fdest* ..... **(Normal)**
pfld.y *isrc1*(*isrc2*)++, *fdest* ..... **(Autoincrement)**
fdest ← mem.y (third previous **pfld**'s (*isrc1* + *isrc2*))
(where .y is precision of third previous **pfld.y**)
IF autoincrement
THEN *isrc2* ← *isrc1* + *isrc2*
FI
NOTE: pfld.q is not available with the i860 XR CPU

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-unit result
last-stage graphics-unit 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 the 64-bit operands as arrays of two 32-bit fields *fsrc1*(1)..*fsrc1*(0), *fsrc2*(1)..*fsrc2*(0), and *fdest*(1)..*fdest*(0) 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-unit result
last-stage graphics-unit result ← smaller of *fsrc2*(i) and *fsrc1*(i)
OD
MERGE ← 0

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

Consider the 64-bit operands as arrays of four 16-bit fields *fsrc1*(3)..*fsrc1*(0), *fsrc2*(3)..*fsrc2*(0), and *fdest*(3)..*fdest*(0) 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 ← last-stage graphics-unit result
last-stage graphics-unit result(i) ← 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

scyc.x *isrc2* ..... **Special Cycles**

Generate a special bus cycle (D/C# = 0, W/R# = 1, M/IO# = 0) and set BE7#–BE0# according to the value contained in the register *isrc2*
NOTE: Not available with the i860 XR CPU

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 *isrc1ni, isrc2, idest* ..... **Shift Right Double**

idest ← low-order 32 bits of *isrc1ni:isrc2* shifted right by SC bits

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

csrc2 ← *src1ni*

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

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

stio.x *isrc1ni, isrc2* ..... **Store I/O**

port.x (isrc2) ← *isrc1ni*
NOTE: Not available with the i860 XR CPU

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. Interrupts are enabled.

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 bits) XOR *isrc2*
CC set if result is zero, cleared otherwise

Instruction Format and Encoding

B

APPENDIX B
INSTRUCTION FORMAT AND ENCODING

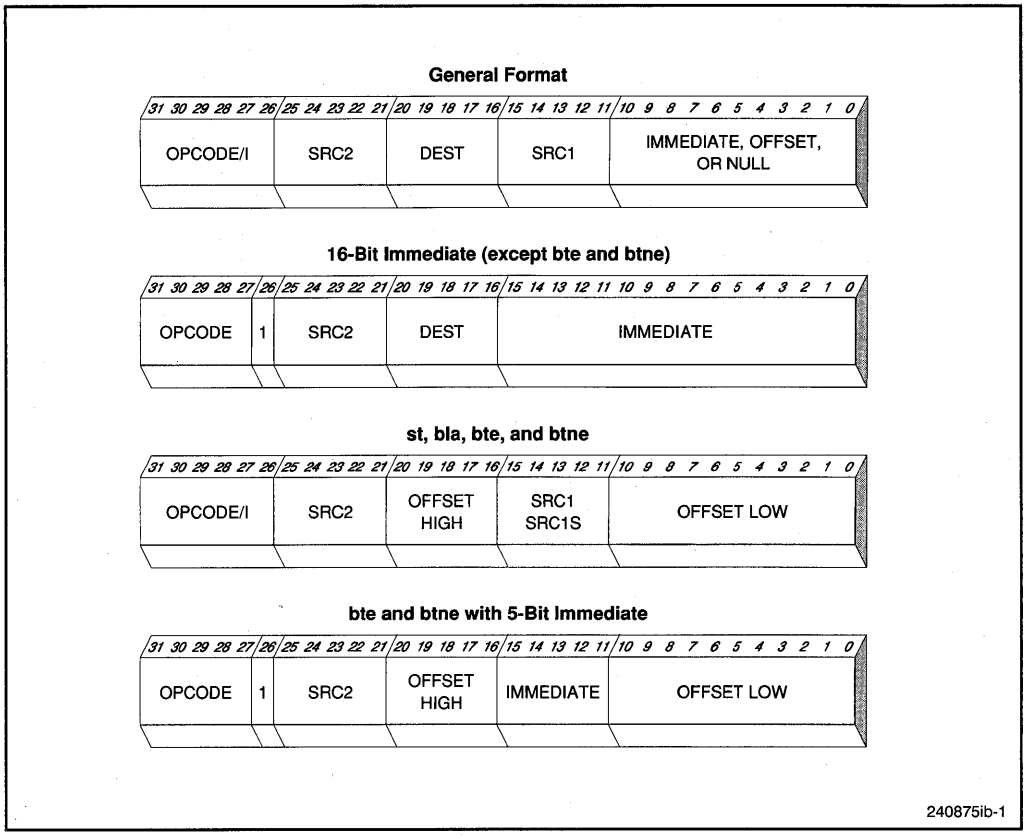
All instructions are 32 bits long and begin on a four-byte boundary. When operands are registers, the encodings shown in the following table are used:

Register	Encoding
r0 . . . **r31**	0 . . . 31
f0 . . . **f31**	0 . . . 31
Fault Instruction Processor Status Directory Base Data Breakpoint Floating-Point Status Extended Processor Status	0 1 2 3 4 5
Bus Error Address* Concurrency Control* **p0*** **p1*** **p2*** **p3***	6 7 8 9 10 11

NOTE:
* Available only with i860 XP CPU. Using these encodings with the i860 XR CPU produces undefined results.

Among the core instructions, there are two general formats: REG-format and CTRL-format. Within the REG-format are several variations.

REG-Format Instructions



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

For **ld** and **st**, bits 28 and 0 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 immediate and bit 28 is set, bit 0 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 1 and 2 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

NOTE: *The 128-bit encoding for **pfld** is not available with the i860 XR CPU.

For **flush**, bits 1 and 2 must be zero.

When *src1* is immediate, bits 0 and 1 of the immediate value are forced to zero to maintain alignment. When bit 1 of the immediate value is clear, bit 2 is also forced to zero.

REG-Format Opcodes

		31	30	29	28	27	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

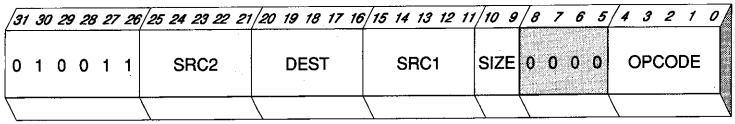
LR Left/Right
- 0 – Left Shift
- 1 – Right Shift

E Equal
- 0 – Branch on Unequal
- 1 – Branch on Equal

I Immediate
- 0 – *src1* is register
- 1 – *src1* is immediate

B

Core Escape Instructions



NOTE:
FIELDS NOT USED BY AN INSTRUCTION ARE *RESERVED* AND MUST BE SET TO ZERO.

240875ib-2

Core Escape Opcodes

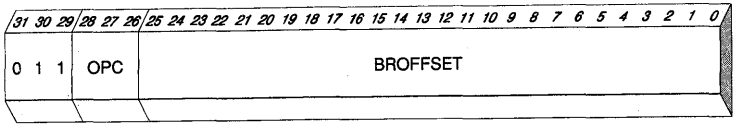
		4	3	2	1	0
—	(*reserved*)	0	0	0	0	0
lock	Begin Interlocked Sequence	0	0	0	0	1
calli	Indirect Subroutine Call	0	0	0	1	0
—	(*reserved*)	0	0	0	1	1
intovr	Trap on Integer Overflow	0	0	1	0	0
—	(*reserved*)	0	0	1	0	1
—	(*reserved*)	0	0	1	1	0
unlock	End Interlocked Sequence	0	0	1	1	1
ldio*	Load I/O	0	1	0	0	0
stio*	Store I/O	0	1	0	0	1
ldint*	Load Interrupt Vector	0	1	0	1	0
scyc*	Special Cycles	0	1	0	1	1
—	(*reserved*)	0	1	1	x	x
—	(*reserved*)	1	0	x	x	x
—	(*reserved*)	1	1	x	x	x

NOTE:
* Available only with i860 XP CPU, not with i860 XR CPU.

For the instructions **ldio, stio, ldint,** and **scyc,** the operand size is encoded by bits 9 and 10 as follows. For other instructions, these bits are *reserved* and should be set to zero.

Operand Size	Bit 10	Bit 9
8 Bits (.b)	0	0
16 Bits (.s)	0	1
32 Bits (.l)	1	0
reserved	1	1

CTRL-Format Instructions



NOTE:
BROFFSET IS A SIGNED 26-BIT RELATIVE BRANCH OFFSET

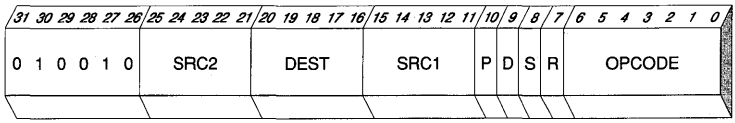
240875ib-3

CTRL-Format Opcodes

		28	27	26
br	Branch Direct	0	1	0
call	Call	0	1	1
bc(.t)	Branch on CC Set	1	0	T
bnc(.t)	Branch on CC Clear	1	1	T

T Taken
0 – **bc** or **bnc**
1 – **bc.t** or **bnc.t**

Floating-Point Instructions



SRC1, SRC2 – Source; one of 32 floating-point registers
DEST – Destination; one of 32 floating-point registers (except **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

240875ib-4

Floating-Point Opcodes

		6	5	4	3	2	1	0
pfam	Add and Multiply*	0	0	0	DPC			
pfmam	Multiply with Add*							
pfsm	Subtract and Multiply*	0	0	1	DPC			
pfmsm	Multiply with Subtract*							
(p)fmul	Multiply	0	1	0	0	0	0	0
fmlow	Multiply Low	0	1	0	0	0	0	1
frcp	Reciprocal	0	1	0	0	0	1	0
frsqr	Reciprocal Square Root	0	1	0	0	0	1	1
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

NOTE:
All opcodes not shown are *reserved*.
* **pfam** and **pfsm** have P-bit set; **pfmam** and **pfmsm** have P-bit clear.
** **pfgt** has R bit cleared; **pfle** has R bit set.

Data Path 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	scr2	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	M 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**							

NOTE: *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.

Instruction Timings

C

APPENDIX C
INSTRUCTION TIMINGS

Generally, i860 microprocessor instructions take one clock to execute unless a freeze condition is invoked. Detailed times, along with freeze conditions and their associated delays, are shown in the table on the following pages. The following symbols are used for brevity in the timing table:

$+ n$	n clocks must be added to the execution time if the stated conditions apply.
$\leftrightarrow n$	The processor requires at least n clocks between the indicated instructions. The actual delay will be n minus the number of clocks for executing intervening instructions (or dual-mode pairs). If the time for intervening instructions is $\geq n$, there is no delay.
n..m	Indicates a range of clocks. These cases are accompanied by a reference to a note where further explanation is available.
XR:	Applies to i860 XR microprocessors only.
XP:	Applies to i860 XP microprocessors only.
OA	The number of clocks to finish all outstanding accesses.
R1	The number of clocks from ADS# through the first READY# (80860XR) or BRDY# (80860XP) of the indicated bus activity.
R2	The number of clocks from ADS# through the second READY# or BRDY#.
RL	The number of clocks from ADS# through the last READY# or BRDY#.
RL1	XP: The number of clocks through last BRDY# of first access.
RN	XR: The 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).
RX	The number of clocks through READY# or BRDY# for the next 64-bit-or-less write cycle or second READY# or BRDY# for the next 128-bit write cycle.

Notes:

a. "Address path full" means one address internally waiting for bus while external bus pipeline full.

b. "Store path full" means two stores or one 256-bit write-back internally waiting for bus plus external bus pipeline full.

c. If a floating-point instruction, graphics-unit instruction, **fst**, or **pst** is executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress, the scalar operation must complete first: two additional clocks for **fadd**, **fix**, **fmlow**, **fmul.ss**, **fmul.sd**, **ftrunc**, and **fsub**; three additional clocks for **fmul.dd**. Add one if either or both of these 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 differs from the source precision of the floating-point operation.
2. The floating-point operation is pipelined and its destination is not **f0**.

TLB TLB miss. Five clocks plus the number of clocks to finish two reads plus the number of clocks to set A-bits (if necessary).

In addition, any instruction may be delayed due to an instruction cache miss or TLB miss during the instruction fetch. The time for a TLB miss is shown above in note **TLB**. An instruction cache miss adds the following delays:

- The number of clocks to get the next instruction from the bus (ADS# clock to first READY# or BRDY# clock, inclusive).
- XR: When any of the instructions in the new instruction-cache line is a branch or call or causes a freeze, the time through the last READY# for the new line.
- If the data cache is being accessed when the instruction-cache miss occurs, two clocks for data cache miss; one clock for hit.

Not included in the table is the delay caused by a trap. This depends on the trap handler.

In dual instruction mode, each pair of instructions requires the maximum of the times required by each individual instruction.

Instruction	Execution Clocks	Condition
adds	1	
addu	1	
and	1	
andh	1	
andnot	1	
andnoth	1	
bc	1	If branch not taken.
	2	If branch taken.
	+1	If the prior instruction is **addu, adds, subu, subs, pfeq,** or **pfgt**.
bc.t	1	If branch taken.
	2	If branch not taken.
	+1	If the prior instruction is **addu, adds, subu, subs, pfeq,** or **pfgt**.
bla	1	If branch taken.
	2	If branch not taken.
bnc	(same as **bc**)	
bnc.t	(same as **bc.t**)	
br	1	
bri	2	
bte	1	If branch not taken.
	3	If branch taken.
btne	(same as **bte**)	
call	1	
	+ 1	If **r1** referenced in next instruction.
	+1+R1	If data cache load miss in progress for a read of less than 128 bits.
	+1+R2	If data cache load miss in progress for 128-bit read.
calli	2	
	+ 1	If **r1** referenced in next instruction.
	+1+R1	If data cache load miss in progress for a read of less than 128 bits.
	+1+R2	If data cache load miss in progress for 128-bit read.
fadd.p	1	(... and all other A-unit instructions except dual operations)
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
faddp	1	(... and all other G-unit instructions except **fiadd.w, fxfr**)
	+1	If *fdest* is used by next instruction and next instruction is G-, M- or A-unit instruction
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
faddz	(same as **faddp**)	
famov.r	(same as **fadd.p**)	
fiadd.w	1	
	+ 1	If *fdest* is used by next instruction and next instruction is M- or A-unit instruction (except when **fiadd** is used for **fmov.dd** or **fmov.ss**).
	+1	If *fdest* is used by next instruction and next instruction is G-unit instruction.
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]

Instruction	Execution Clocks	Condition
fisub.w	(same as **faddp**)	
fix.v	(same as **fadd.p**)	
fld.y	1	
	+ 1	If this is the instruction after an **st, fst** or **pst** that hits the data cache.
	↔2	
	+1+R1	If *fdest* is referenced in the next two instructions.
	+1+R2	If 32-bit **fld.l** or 64-bit **fld.d** misses the data cache.
	+1+RL	If 128-bit **fld.q** misses the data cache.
	↔2	If data cache load miss in progress (except in the following case). XP: If this instruction follows a data cache access that misses in the virtual tags but hits in the physical tags.
	+2	XP: If the prior instruction is a **pfld.y** that hits a modified line in the data cache.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	+RN	XR: If address path full.[a]
	+RL1	XP: If address path full.[a]
	+TLB	If TLB miss.
flush	1	
	↔3	XR: If preceded by another **flush**.
	↔2	XP: If preceded by another **flush**.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	+1+RX	If flush to modified line when store path full.[b]
	+TLB	If TLB miss.
fmlow.dd	1	(... and all other M-unit instructions except dual operations)
	+1	If *fsrc1* refers to result of the prior operation (either scalar or pipelined).
	+1	If the prior operation is a double-precision multiply.
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
fmov.r		**fmov.ss** and **fmov.dd** same as **fiadd.w** **fmov.sd** and **fmov.ds** same as **fadd.p**
fmul.p	(same as **fmlow.dd**)	
fnop	1	
form	(same as **faddp**)	
frcp.p	(same as **fmlow.dd**)	
frsqr.p	(same as **fmlow.dd**)	
fst.y	1	
	+1	If followed by pipelined floating-point operation that overwrites the register being stored.
	+1+RL	If data cache load miss in progress.
	+2	XP: If the prior instruction is a **pfld.y** that hits a modified line in the data cache.
	↔2	XP: If this instruction follows a data cache access that misses in the virtual tags but hits in the physical tags.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
	+RN	XR: If address path full.[a]
	+RL1	XP: If address path full.[a]
	+1+RX	If cache miss when store path full.[b]
	+TLB	If TLB miss.

Instruction	Execution Clocks	Condition
fsub.p	(same as **fadd.p**)	
ftrunc.v	(same as **fadd.p**)	
fxfr	1	
	+1	If *idest* referenced in next instruction.
	+1+R1	If data cache load miss in progress for 64-bit read.
	+1+R2	If data cache load miss in progress for 128-bit read.
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
fzchkl	(same as **faddp**)	
fzchks	(same as **faddp**)	
intovr	1	
ixfr	1	
	+1+R1	If data cache load miss in progress for 64-bit read.
	+1+R2	If data cache load miss in progress for 128-bit read.
	↔2	If *fdest* is referenced in the next two instructions.
ld.c	1	
	+1	If *idest* referenced in next instruction.
	+1+R1	If data cache load miss in progress for 64-bit read.
	+1+R2	If data cache load miss in progress for 128-bit read.
ld.x	1	
	+1	If *idest* referenced in next instruction.
	+1	If this is the instruction after an **st, fst** or **pst** that hits the data cache.
	+1+RL	If data cache load miss in progress.
	↔1+R1	If **ld.x** misses the data cache and a subsequent instruction references the *idest* of the **ld.x** (except for following case).
	↔2	XP: If this instruction follows a data cache access that misses in the virtual tags but hits in the physical tags.
	+2	XP: If the prior instruction is a **pfld.y** that hits a modified line in the data cache.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	+RN	XR: If address path full.[a]
	+RL1	XP: If address path full.[a]
	+1+RX	If cache miss when store path full.[b]
	+TLB	If TLB miss.
ldint.x	1 + OA	
ldio.x	1 + OA	
lock	1	
mov	1	
nop	1	
or	1	
orh	1	
pfadd.p	(same as **fadd.p**)	
pfaddp	(same as **faddp**)	
pfaddz	(same as **faddp**)	

Instruction	Execution Clocks	Condition
pfam.p	1	(... and all other dual operations)
	+1	If *fsrc1* refers to result of the prior operation (either scalar or pipelined).
	+1	If the prior operation is a double-precision multiply.
	↔2..4	If executed when a scalar floating-point operation (other than **frcp** or **frsqr**) is in progress.[c]
pfamov.r	(same as **fadd.p**)	
pfeq.p	(same as **fadd.p**)	
pfgt.p	(same as **fadd.p**)	
pfiadd.w	(same as **faddp**)	
pfisub.w	(same as **faddp**)	
pfix.v	(same as **fadd.p**)	
pfld.y	1	
	+ 1 + RL	If data cache load miss in progress.
	↔2	If *fdest* is referenced in the next two instructions.
	+1+RL1	If three **pfld**'s are outstanding.
	+2+OA	XR: If **pfld** hits data cache.
	+2	XP: If the prior instruction is a **pfld.y** that hits a modified line in the data cache.
	↔2	XP: If this instruction follows a data cache access that misses in the virtual tags but hits in the physical tags.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	+RN	XR: If address path full.[a]
	+RL1	XP: If address path full.[a]
	+TLB	If TLB miss.
pfle.p	1	
pfmam.p	(same as **pfam.p**)	
pfmov.r		**pfmov.ss** and **pfmov.dd** same as **faddp** **pfmov.sd** and **pfmov.ds** same as **fadd.p**
pfmsm.p	(same as **pfam.dd**)	
pfmul.p	(same as **fmlow.dd**)	
pfmul3.dd	(same as **fmlow.dd**)	
pform	(same as **faddp**)	
pfsm.p	(same as **pfam.dd**)	
pfsub.p	(same as **fadd.p**)	
pftrunc.v	(same as **fadd.p**)	
pfzchkl	(same as **faddp**)	
pfzchks	(same as **faddp**)	
pst.d	(same as **fst.d**)	
scyc.x	1 + OA	
shl	1	
shr	1	
shra	1	

Instruction	Execution Clocks	Condition
shrd	1	
st.c	3	
	+ 1 + R1	If data cache load miss in progress for a read of less than 128 bits.
	+1+R2	If data cache load miss in progress for 128-bit read.
st.x	1	
	+ 1 + RL	If data cache load miss in progress.
	+2	XP: If the prior instruction is a **pfld.y** that hits a modified line in the data cache.
	↔2	XP: If this instruction follows a data cache access that misses in the virtual tags but hits in the physical tags.
	+R2	XP: If data-cache line write-back due to snoop is in progress.
	+RN	XR: If address path full.[a]
	+RL1	XP: If address path full.[a]
	+1+RX	If cache miss and store path full.[a]
	+TLB	If TLB miss.
stio.x	1 + OA	
subs	1	
subu	1	
trap	1	
unlock	1	
xor	1	
xorh	1	

C

Instruction Characteristics D

APPENDIX D
INSTRUCTION CHARACTERISTICS

The following table lists some of the characterisics 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 execution is suppressed in user mode. An *SU* marks supervisor-only instructions.
- Whether the instruction is available on both the i860 XR and i860 XP microprocessors. An *XL* marks instructions that are available only on the i860 XP microprocessor.
- 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 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, *src1* and *src2* of the current operation should not be the same as *dest* 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, nor the target of a control-transfer instruction.
  b. When using a **bri** to return from a trap handler, programmers should take care to prevent traps from occurring on that or on the next sequential instruction. IM should be zero (interrupts disabled) when the **bri** is executed.
  c. If *fdest* is not zero, *fsrc1* must not be the same as *fdest*.
  d. When *fsrc1* goes to multiplier *op1* or to KR or KI, *fsrc1* must not be the same as *fdest*.
  e. If *dest* is not zero, *src1* and *src2* must not be the same as *dest*.
  f. *Isrc1* must not be the same register as *isrc2* for the autoincrementing form of this instruction.
  g. *Isrc1* must not be the same register as *isrc2*.

Instruction	Execution Unit	Pipelined? Delayed? Supervisor? XP Only?	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.w	G				8	
fisub.w	G				8	
fix.p	A			SE, RE		
fld.y	E			DAT	2, 3	f
flush	E					
fmlow.dd	M				4	
fmul.p	M			SE, RE	4	
form	G				8	
frcp.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	
ldint.x	E	SU, XP		DAT		
ldio.x	E	SU, XP		DAT		
lock	E					
or	E		CC			
orh	E		CC			

Instruction	Execution Unit	Pipelined? Delayed? Supervisor? XP Only?	Sets CC?	Faults	Performance Notes	Programming Restrictions
pfadd.p	A	P		SE, RE*		
pfaddp	G	P		*	8	e
pfaddz	G	P		*	8	e
pfam.p	A&M	P		SE, RE*	7	d
pfamov.r	A	P		SE, RE*		
pfeq.p	A	P	CC	SE*	1	
pfgt.p	A	P	CC	SE*	1	
pfiadd.w	G	P		*	8	e
pfisub.w	G	P		*	8	e
pfix.p	A	P		SE, RE*		
pfld.y	E	P, (XP)**		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	5	f
scyc.x	E	SU, XP		DAT		
shl	E					
shr	E					
shra	E					
shrd	E					
st.c	E					
st.x	E			DAT		
stio.x	E	SU, XP		DAT		
subs	E		CC		1	
subu	E		CC		1	
trap	E			IT		
unlock	E					
xor	E		CC			
xorh	E		CC			

NOTES:
*On the i860 XP microprocessor, the pipelined instructions can generate IT with PI.
On the i860 XR microprocessor, the 128-bit **pfld.q is not available. If used it causes an instruction trap.

Compatibility Between i860™ XR and i860™ XP Microprocessors

E

APPENDIX E
COMPATIBILITY BETWEEN i860™ XR AND i860™ XP MICROPROCESSORS

REQUIRED CHANGES

To port existing systems software from the i860 XR microprocessor to the i860 XP microprocessor, the following changes may be required. Applications software does not require changes.

1. Data cache flush. All four ways of the data cache must be flushed on the i860 XP microprocessor. The cache flush routine can be modified to check processor type in **epsr** or the DCS field of **dirbase** and flush the appropriate number of ways.

2. Parity and bus error traps. If the i860 XP system signals these errors, the trap handler must be extended to handle them. Software must avoid testing the BEF and PEF bits unless executing on the i860 XP microprocessor.

3. LOCK# deactivation. On the i860 XP microprocessor, traps do not automatically deactivate the LOCK# signal, so the trap handler must do a data access to deactivate LOCK#. Trap handlers that already access data soon after invocation do not require this modification.

4. Load pipe precision. The precision of the last stage of the load pipeline is specified by the LRP bit on the i860 XR microprocessor but by the LRP0 and LRP1 bits on the i860 XP microprocessor. The procedure that restores the load pipe must check the processor type, use the appropriate bits, and restore the correct precision. Pipe restoration code for the i860 XR microprocessor will work correctly on the i860 XP microprocessor if **pfld.q** is not used.

5. Pre-accessed trap handler pages. Page-directory and page-table entries for the instruction pages of the trap handler and for the first data page accessed by the trap handler must always have A = 1. Software modified to allocate page tables this way works on both i860 XR and i860 XP microprocessors.

6. Page directory entry bit 7. On the i860 XP microprocessor, this bit determines whether the page size is four Mbytes or four Kbytes. On the i860 XR microprocessor, it is *reserved* and should be set to zero. It must be set to zero for four unmodified i860 XR microprocessor software to work correctly on the i860 XP microprocessor.

PERFORMANCE OPTIMIZATIONS

Software developers may wish to make the following performance enhancements in systems software for the i860 XP microprocessor. Systems software that must execute on both i860 XP and i860 XR systems can contain code both with and without the optimizations. By testing the processor type, the appropriate instruction path can be determined.

1. Data cache flush. On the i860 XP microprocessor, a complete flushing of the data cache is not needed when changing context or marking a page not present.

2. The **epsr** bits AI, DI, PI, and PT can be used on the i860 XP microprocessor to make trap handlers more efficient.

3. Four-Mbyte pages can be allocated to frame buffers and the operating-system kernel, thereby reducing the cost of TLB misses.

NEW FEATURES

Software that uses the new features available only on the i860 XP microprocessor will not be compatible with the i860 XR microprocessor unless alternate instruction paths are provided.

Systems software features:

1. New instructions **ldio, stio, ldint,** and **scyc**.

2. Four-Mbyte pages.

3. Privileged registers **p0, p1, p2,** and **p3**.

4. Concurrency control unit.

5. 128-bit load instruction **pfld.q**.

6. Support for virtual address aliases.

Applications software features:

1. Concurrency control unit.

2. 128-bit load instruction **pfld.q**. The i860 XR microprocessor traps on **pfld.q**; therefore, software has the opportunity to emulate a **pfld.q** with two **pfld.d** instructions. However, this strategy does not yield optimal performance on the i860 XR microprocessor.

NOTES

On the i860 XP microprocessor, pages with WT = 1 are cached with the write-through policy; whereas, on the i860 XR microprocessor, they are not cached at all. Because this change in the function of WT was anticipated in the i860 XR microprocessor documentation, no incompatibility should arise.

Index

INDEX

DOMESTIC SALES OFFICES

ALABAMA

Intel Corp.
015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010
FAX: (205) 837-2640

ARIZONA

Intel Corp.
10 North 44th Street
Suite 500
Phoenix 85008
Tel: (602) 231-0386
FAX: (602) 244-0446

Intel Corp.
225 N. Mona Lisa Rd.
Suite 215
Tucson 85741
Tel: (602) 544-0227
FAX: (602) 544-0232

CALIFORNIA

Intel Corp.
1515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500
FAX: (818) 340-1144

Intel Corp.
00 N. Continental Blvd.
Suite 100
El Segundo 90245
Tel: (213) 640-6040
FAX: (213) 640-7133

Intel Corp.
Sierra Gate Plaza
Suite 280C
Roseville 95678
Tel: (916) 782-8086
FAX: (916) 782-8153

Intel Corp.
1665 Chesapeake Dr.
Suite 325
San Diego 92123
Tel: (619) 292-8086
FAX: (619) 292-0628

Intel Corp.*
200 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114
FAX: (714) 541-9157

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

COLORADO

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (719) 594-6622
FAX: (303) 594-0720

Intel Corp.*
600 S. Cherry St.
Suite 700
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289
FAX: (303) 322-8670

CONNECTICUT

Intel Corp.
301 Lee Farm Corporate Park
83 Wooster Heights Rd.
Danbury 06810
Tel: (203) 748-3130
FAX: (203) 794-0339

FLORIDA

†Intel Corp.
800 Fairway Drive
Suite 160
Deerfield Beach 33441
Tel: (305) 421-0506
FAX: (305) 421-2444

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

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

GEORGIA

†Intel Corp.
20 Technology Parkway
Suite 150
Norcross 30092
Tel: (404) 449-0541
FAX: (404) 605-9762

ILLINOIS

†Intel Corp.*
Woodfield Corp. Center III
300 N. Martingale Road
Suite 400
Schaumburg 60173
Tel: (708) 605-8031
FAX: (708) 706-9762

INDIANA

†Intel Corp.
8910 Purdue Road
Suite 350
Indianapolis 46268
Tel: (317) 875-0623
FAX: (317) 875-8938

IOWA

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

KANSAS

†Intel Corp.
10985 Cody St.
Suite 140
Overland Park 66210
Tel: (913) 345-2727
FAX: (913) 345-2076

MARYLAND

†Intel Corp.*
10010 Junction Dr.
Suite 200
Annapolis Junction 20701
Tel: (301) 206-2860
FAX: (301) 206-3677
(301) 206-3678

MASSACHUSETTS

†Intel Corp.*
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (508) 692-0960
TWX: 710-343-6333
FAX: (508) 692-7867

MICHIGAN

†Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8096
FAX: (313) 851-8770

MINNESOTA

†Intel Corp.
3500 W. 80th St.
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867
FAX: (612) 831-6497

MISSOURI

†Intel Corp.
3300 Rider Trail South
Suite 170
Earth City 63045
Tel: (314) 291-1990
FAX: (314) 291-4341

NEW JERSEY

Intel Corp.
Arbor Circle South
8 Campus Drive
Parsippany 07054
Tel: (201) 455-1868
FAX: (201) 644-0680

†Intel Corp.*
Lincroft Office Center
125 Half Mile Road
Red Bank 07701
Tel: (908) 747-2233
FAX: (908) 747-0983

NEW YORK

Intel Corp.*
850 Crosskeys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391
FAX: (716) 223-2561

†Intel Corp.*
2950 Express Dr., South
Suite 130
Islandia 11722
Tel: (516) 231-3300
TWX: 510-227-6236
FAX: (516) 348-7939

†Intel Corp.
300 Westage Business Center
Suite 230
Fishkill 12524
Tel: (914) 897-3860
FAX: (914) 897-3125

Intel Corp.
Seventeen State Street
14th Floor
New York 10004
Tel: (212) 248-8086
FAX: (212) 248-0888

NORTH CAROLINA

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

†Intel Corp.
5540 Centerview Dr.
Suite 215
Raleigh 27606
Tel: (919) 851-9537
FAX: (919) 851-8974

OHIO

†Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528
FAX: (513) 890-8658

†Intel Corp.*
25700 Science Park Dr.
Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298
FAX: (804) 282-0673

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086
FAX: (405) 840-9819

OREGON

†Intel Corp.
15254 N.W. Greenbrier Pkwy.
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741
FAX: (503) 645-8181

PENNSYLVANIA

†Intel Corp.*
925 Harvest Drive
Suite 200
Blue Bell 19422
Tel: (215) 641-1000
FAX: (215) 641-0785

†Intel Corp.*
400 Penn Center Blvd.
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970
FAX: (412) 829-7578

PUERTO RICO

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

TEXAS

†Intel Corp.
8911 N. Capital of Texas Hwy.
Suite 4230
Austin 78759
Tel: (512) 794-8086
FAX: (512) 338-9335

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

†Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490
FAX: (713) 988-3660

UTAH

†Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051
FAX: (801) 268-1457

VIRGINIA

†Intel Corp.
9030 Stony Point Pkwy.
Suite 360
Richmond 23235
Tel: (804) 330-9393
FAX: (804) 330-3019

WASHINGTON

†Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002
FAX: (206) 451-9556

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086
FAX: (509) 928-9467

WISCONSIN

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

CANADA

BRITISH COLUMBIA

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

ONTARIO

†Intel Semiconductor of
Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
FAX: (613) 820-5936

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

QUEBEC

†Intel Semiconductor of
Canada, Ltd.
1 Rue Holiday
Suite 115
Tour East
Pt. Claire H9R 5N3
Tel: (514) 694-9130
FAX: 514-694-0064

·Sales and Service Office
Field Application Location

DOMESTIC DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc.
1015 Henderson Road
Huntsville 35805
Tel: (205) 837-6955
FAX: 205-751-1581

Hamilton/Avnet Computer
4930 I Corporate Drive
Huntsville 35805

Hamilton/Avnet Electronics
4940 Research Drive
Huntsville 35805
Tel: (205) 837-7210
FAX: 205-721-0356

MTI Systems Sales
4950 Corporate Drive
Suite 120
Huntsville 35806
Tel: (205) 830-9526
FAX: (205) 830-9557

Pioneer/Technologies Group, Inc.
4825 University Square
Huntsville 35805
Tel: (205) 837-9300
FAX: 205-837-9358

ALASKA

Hamilton/Avnet Computer
1400 W. Benson Blvd., Suite 400
Anchorage 99503

ARIZONA

†Arrow Electronics, Inc.
4134 E. Wood Street
Phoenix 85040
Tel: (602) 437-0750
TWX: 910-951-1550

Hamilton/Avnet Computer
30 South McKemy Avenue
Chandler 85226

Hamilton/Avnet Computer
90 South McKemy Road
Chandler 85226

†Hamilton/Avnet Electronics
505 S. Madison Drive
Tempe 85281
Tel: (602) 231-5140
TWX: 910-950-0077

Hamilton/Avnet Electronics
30 South McKemy
Chandler 85226
Tel: (602) 961-6669
FAX: 602-961-4073

Wyle Distribution Group
4141 E. Raymond
Phoenix 85040
Tel: (602) 249-2232
TWX: 910-371-2871

CALIFORNIA

Arrow Commercial System Group
1502 Crocker Avenue
Hayward 94544
Tel: (415) 489-5371
FAX: (415) 489-9393

Arrow Commercial System Group
14242 Chambers Road
Tustin 92680
Tel: (714) 544-0200
FAX: (714) 731-8438

†Arrow Electronics, Inc.
19748 Dearborn Street
Chatsworth 91311
Tel: (213) 701-7500
TWX: 910-493-2086

†Arrow Electronics, Inc.
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
FAX: 619-279-8062

†Arrow Electronics, Inc.
521 Weddell Drive
Sunnyvale 94086
Tel: (408) 745-6600
TWX: 910-339-9371

†Arrow Electronics, Inc.
2961 Dow Avenue
Tustin 92680
Tel: (714) 838-5422
TWX: 910-595-2860

Hamilton/Avnet Computer
3170 Pullman Street
Costa Mesa 92626

Hamilton/Avnet Computer
1361B West 190th Street
Gardena 90248

Hamilton/Avnet Computer
4103 Northgate Blvd.
Sacramento 95834

Hamilton/Avnet Computer
4545 Viewridge Avenue
San Diego 92123

Hamilton/Avnet Computer
1175 Bordeaux Drive
Sunnyvale 94089

Hamilton/Avnet Electronics
21150 Califa Street
Woodland Hills 91367

†Hamilton/Avnet Electronics
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4150
TWX: 910-595-2638

†Hamilton/Avnet Electronics
1175 Bordeaux Drive
Sunnyvale 94086
Tel: (408) 743-3300
TWX: 910-339-9332

†Hamilton/Avnet Electronics
4545 Ridgeview Avenue
San Diego 92123
Tel: (619) 571-7500
TWX: 910-595-2638

†Hamilton/Avnet Electronics
21150 Califa St.
Woodland Hills 91376
Tel: (818) 594-0404
FAX: 818-594-8233

†Hamilton/Avnet Electronics
10950 W. Washington Blvd.
Culver City 20230
Tel: (213) 558-2458
TWX: 910-340-6364

†Hamilton/Avnet Electronics
1361B West 190th Street
Gardena 90248
Tel: (213) 217-6700
TWX: 910-340-6364

†Hamilton/Avnet Electronics
4103 Northgate Blvd.
Sacramento 95834
Tel: (916) 920-3150

Pioneer/Technologies Group, Inc.
134 Rio Robles
San Jose 95134
Tel: (408) 954-9100
FAX: 408-954-9113

Wyle Distribution Group
124 Maryland Street
El Segundo 90254
Tel: (213) 322-8100

Wyle Distribution Group
7431 Chapman Ave.
Garden Grove 92641
Tel: (714) 891-1717
FAX: 714-891-1621

†Wyle Distribution Group
2951 Sunrise Blvd., Suite 175
Rancho Cordova 95742
Tel: (916) 638-5282

†Wyle Distribution Group
9525 Chesapeake Drive
San Diego 92123
Tel: (619) 565-9171
TWX: 910-335-1590

†Wyle Distribution Group
3000 Bowers Avenue
Santa Clara 95051
Tel: (408) 727-2500
TWX: 408-988-2747

†Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel: (714) 863-9953
TWX: 910-371-7127

†Wyle Distribution Group
26677 W. Agoura Rd.
Calabasas 91302
Tel: (818) 880-9000
TWX: 372-0232

COLORADO

Arrow Electronics, Inc.
7060 South Tucson Way
Englewood 80112
Tel: (303) 790-4444

Hamilton/Avnet Computer
9605 Maroon Circle, Ste. 200
Engelwood 80112

†Hamilton/Avnet Electronics
9605 Maroon Circle
Suite 200
Englewood 80112
Tel: (303) 799-0663
TWX: 910-935-0787

†Wyle Distribution Group
451 E. 124th Avenue
Thornton 80241
Tel: (303) 457-9953
TWX: 910-936-0770

CONNECTICUT

†Arrow Electronics, Inc.
12 Beaumont Road
Wallingford 06492
Tel: (203) 265-7741
TWX: 710-476-0162

Hamilton/Avnet Computer
Commerce Industrial Park
Commerce Drive
Danbury 06810

†Hamilton/Avnet Electronics
Commerce Industrial Park
Commerce Drive
Danbury 06810
Tel: (203) 797-2800
TWX: 710-456-9974

†Pioneer/Standard Electronics
112 Main Street
Norwalk 06851
Tel: (203) 853-1515
FAX: 203-838-9901

FLORIDA

†Arrow Electronics, Inc.
400 Fairway Drive
Suite 102
Deerfield Beach 33441
Tel: (305) 429-8200
FAX: 305-428-3991

†Arrow Electronics, Inc.
37 Skyline Drive
Suite 3101
Lake Marv 32746
Tel: (407) 323-0252
FAX: 407-323-3189

Hamilton/Avnet Computer
6801 N.W. 15th Way
Ft. Lauderdale 33309

Hamilton/Avnet Computer
3247 Spring Forest Road
St. Petersburg 33702

†Hamilton/Avnet Electronics
6801 N.W. 15th Way
Ft. Lauderdale 33309
Tel: (305) 971-2900
FAX: 305-971-5420

†Hamilton/Avnet Electronics
3197 Tech Drive North
St. Petersburg 33702
Tel: (813) 573-3930
FAX: 813-572-4329

†Hamilton/Avnet Electronics
6947 University Boulevard
Winter Park 32792
Tel: (407) 628-3888
FAX: 407-678-1878

†Pioneer/Technologies Group, Inc.
337 Northlake Blvd., Suite 1000
Alta Monte Springs 32701
Tel: (407) 834-9090
FAX: 407-834-0865

Pioneer/Technologies Group, Inc.
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
FAX: 305-481-2950

GEORGIA

Arrow Commercial System Group
3400 C. Corporate Way
Deluth 30139
Tel: (404) 623-8825
FAX: (404) 623-8802

†Arrow Electronics, Inc.
4250 E. Rivergreen Parkway
Deluth 30136
Tel: (404) 497-1300
TWX: 810-766-0439

Hamilton/Avnet Computer
5825 D. Peachtree Corners E.
Norcross 30092

†Hamilton/Avnet Electronics
5825 D Peachtree Corners
Norcross 30092
Tel: (404) 447-7500
TWX: 810-766-0432

Pioneer/Technologies Group, Inc.
3100 F Northwoods Place
Norcross 30071
Tel: (404) 448-1711
FAX: 404-446-8270

ILLINOIS

†Arrow Electronics, Inc.
1140 W. Thorndale
Itasca 60143
Tel: (708) 250-0500
TWX: 708-250-0916

Hamilton/Avnet Computer
1130 Thorndale Avenue
Bensenville 60106

†Hamilton/Avnet Electronics
1130 Thorndale Avenue
Bensenville 60106
Tel: (708) 860-7780
TWX: 708-860-8530

MTI Systems Sales
1100 W. Thorndale
Itasca 60143
Tel: (708) 773-2300

†Pioneer/Standard Electronics
2171 Executive Dr., Suite 200
Addison 60101
Tel: (708) 495-9680
FAX: 708-495-9831

INDIANA

†Arrow Electronics, Inc.
7108 Lakeview Parkway West Drive
Indianapolis 46268
Tel: (317) 299-2071
FAX: 317-299-0255

Hamilton/Avnet Computer
485 Gradle Drive
Carmel 46032

Hamilton/Avnet Electronics
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
FAX: 317-844-5921

†Pioneer/Standard Electronics
9350 Priority Way
West Drive
Indianapolis 46250
Tel: (317) 573-0880
FAX: 317-573-0979

†Certified Technical Distributor

DOMESTIC DISTRIBUTORS (Contd.)

WA

milton/Avnet Computer
5 33rd Avenue SW
dar Rapids 52404

milton/Avnet Electronics
5 33rd Avenue, S.W.
dar Rapids 52404
l: (319) 362-4757

ANSAS

row Electronics, Inc.
08 Melrose Dr., Suite 210
nexa 66214
l: (913) 541-9542
X: 913-541-0328

milton/Avnet Computer
313 W. 95th Street
nexa 61219

lamilton/Avnet Electronics
313 W. 95th
verland Park 66215
l: (913) 888-8900
X: 913-541-7951

ENTUCKY

milton/Avnet Electronics
5 A. Newtown Circle
xington 40511
l: (606) 259-1475

ARYLAND

rrow Electronics, Inc.
00 Guilford Drive
ite H, River Center
olumbia 21046
l: (301) 995-6002
X: 301-381-3854

milton/Avnet Computer
22 Oak Hall Lane
olumbia 21045

lamilton/Avnet Electronics
22 Oak Hall Lane
olumbia 21045
l: (301) 995-3500
X: 301-995-3593

lesa Technology Corp.
20 Patuxent Woods Dr.
olumbia 21046
l: (301) 290-8150
X: 301-290-6474

ioneer/Technologies Group, Inc.
00 Gaither Road
aithersburg 20877
l: (301) 921-0660
X: 301-921-4255

ASSACHUSETTS

ow Electronics, Inc.
Upton Dr.
lmington 01887
l: (508) 658-0900
/X: 710-393-6770

milton/Avnet Computer
D Centennial Drive
abody 01960

amilton/Avnet Electronics
D Centennial Drive
abody 01960
l: (508) 532-9838
X: 508-596-7802

ioneer/Standard Electronics
Hartwell Avenue
xington 02173
l: (617) 861-9200
X: 617-863-1547

le Distribution Group
Third Avenue
rlington 01803
l: (617) 272-7300
X: 617-272-6809

CHIGAN

rrow Electronics, Inc.
380 Haggerty Road
onia 48152
l: (313) 665-4100
/X: 810-223-6020

Hamilton/Avnet Computer
2215 S.E. A-5
Grand Rapids 49508

Hamilton/Avnet Computer
41650 Garden Rd., Ste. 100
Novi 48050

Hamilton/Avnet Electronics
2215 29th Street S.E.
Space A5
Grand Rapids 49508
Tel: (616) 243-8805
FAX: 616-698-1831

Hamilton/Avnet Electronics
41650 Garden Brook
Novi 48050
Tel: (313) 347-4271
FAX: 313-347-4021

†Pioneer/Standard Electronics
4505 Broadmoor S.E.
Grand Rapids 49508
Tel: (616) 698-1800
FAX: 616-698-1831

†Pioneer/Standard Electronics
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
FAX: 313-427-3720

MINNESOTA

†Arrow Electronics, Inc.
5230 W. 73rd Street
Edina 55435
Tel: (612) 830-1800
TWX: 910-576-3125

Hamilton/Avnet Computer
12400 Whitewater Drive
Minnetonka 55343

†Hamilton/Avnet Electronics
12400 Whitewater Drive
Minnetonka 55434
Tel: (612) 932-0600
TWX: 910-576-2720

†Pioneer/Standard Electronics
7625 Golden Triange Dr.
Suite G
Eden Prairie 55343
Tel: (612) 944-3355
FAX: 612-944-3794

MISSOURI

†Arrow Electronics, Inc.
2380 Schuetz
St. Louis 63141
Tel: (314) 567-6888
FAX: 314-567-1164

Hamilton/Avnet Computer
739 Goddard Avenue
Chesterfield 63005

†Hamilton/Avnet Electronics
741 Goddard
Chesterfield 63005
Tel: (314) 537-1600
FAX: 314-537-4248

NEW HAMPSHIRE

Hamilton/Avnet Computer
2 Executive Park Drive
Bedford 03102

Hamilton/Avnet Computer
444 East Industrial Park Dr.
Manchester 03103

NEW JERSEY

†Arrow Electronics, Inc.
4 East Stow Road
Unit 11
Marlton 08053
Tel: (609) 596-8000
FAX: 609-596-9632

†Arrow Electronics
6 Century Drive
Parsipanny 07054
Tel: (201) 538-0900
FAX: 201-538-0900

Hamilton/Avnet Computer
1 Keystone Ave., Bldg. 36
Cherry Hill 08003

Hamilton/Avnet Computer
10 Industrial Road
Fairfield 07006

†Hamilton/Avnet Electronics
1 Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
FAX: 609-751-2552

†Hamilton/Avnet Electronics
10 Industrial
Fairfield 07006
Tel: (201) 575-3390
FAX: 201-575-5839

†MTI Systems Sales
9 Law Drive
Fairfield 07006
Tel: (201) 227-5552
FAX: 201-575-6336

†Pioneer/Standard Electronics
14-A Madison Rd.
Fairfield 07006
Tel: (201) 575-3510
FAX: 201-575-3454

NEW MEXICO

Alliance Electronics Inc.
10510 Research Avenue
Albuquerque 87123
Tel: (505) 292-3360
FAX: 505-292-6537

Hamilton/Avnet Computer
5659 Jefferson, N.E. Suites A & B
Albuquerque 87109

†Hamilton/Avnet Electronics
5659A Jefferson N.E.
Albuquerque 87109
Tel: (505) 765-1500
FAX: 505-243-1395

NEW YORK

†Arrow Electronics, Inc.
3375 Brighton Henrietta Townline Rd.
Rochester 14623
Tel: (716) 427-0300
TWX: 510-253-4766

Arrow Electronics, Inc.
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
TWX: 510-227-6623

Hamilton/Avnet Computer
933 Motor Parkway
Haupauge 11788

Hamilton/Avnet Computer
2060 Townline
Rochester 14623

†Hamilton/Avnet Electronics
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
TWX: 510-224-6166

†Hamilton/Avnet Electronics
2060 Townline Rd.
Rochester 14623
Tel: (716) 272-2744
TWX: 510-253-5470

Hamilton/Avnet Electronics
103 Twin Oaks Drive
Syracuse 13206
Tel: (315) 437-0288
TWX: 710-541-1560

†MTI Systems Sales
38 Harbor Park Drive
Port Washington 11050
Tel: (516) 621-6200
FAX: 510-223-0846

Pioneer/Standard Electronics
68 Corporate Drive
Binghamton 13904
Tel: (607) 722-9300
FAX: 607-722-9562

Pioneer/Standard Electronics
40 Oser Avenue
Hauppauge 11787
Tel: (516) 231-9200
FAX: 510-227-9869

†Pioneer/Standard Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
FAX: 516-921-2143

†Pioneer/Standard Electronics
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
FAX: 716-381-5955

NORTH CAROLINA

†Arrow Electronics, Inc.
5240 Greensdairy Road
Raleigh 27604
Tel: (919) 876-3132
TWX: 510-928-1856

Hamilton/Avnet Computer
3510 Spring Forest Road
Raleigh 27604

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819
TWX: 510-928-1836

Pioneer/Technologies Group, Inc.
9401 L-Southern Pine Blvd.
Charlotte 28210
Tel: (919) 527-8188
FAX: 704-522-8564

Pioneer Technologies Group, Inc.
2810 Meridian Parkway
Suite 148
Durham 27713
Tel: (919) 544-5400
FAX: 919-544-5885

OHIO

Arrow Commercial System Group
284 Cramer Creek Court
Dublin 43017
Tel: (614) 889-9347
FAX: (614) 889-9680

†Arrow Electronics, Inc.
6238 Cochran Road
Solon 44139
Tel: (216) 248-3990
TWX: 810-427-9409

Hamilton/Avnet Computer
7764 Washington Village Dr.
Dayton 45459

Hamilton/Avnet Computer
30325 Bainbridge Rd., Bldg. A
Solon 44139

†Hamilton/Avnet Electronics
7760 Washington Village Dr.
Dayton 45459
Tel: (513) 439-6733
FAX: 513-439-6711

†Hamilton/Avnet Electronics
30325 Bainbridge
Solon 44139
Tel: (216) 349-5100
TWX: 810-427-9452

Hamilton/Avnet Computer
777 Brooksedge Blvd.
Westerville 43081
Tel: (614) 882-7004
FAX: 614-882-8650

Hamilton/Avnet Electronics
777 Brooksedge Blvd.
Westerville 43081
Tel: (614) 882-7004

MTI Systems Sales
23400 Commerce Park Road
Beachwood 44122
Tel: (216) 464-6688

†Pioneer/Standard Electronics
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
FAX: 513-236-8133

†Pioneer/Standard Electronics
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
FAX: 216-663-1004

DOMESTIC DISTRIBUTORS (Contd.)

OKLAHOMA

Arrow Electronics, Inc.
4719 South Memorial Dr.
Tulsa 74145

†Hamilton/Avnet Electronics
12121 E. 51st St., Suite 102A
Tulsa 74146
Tel: (918) 252-7297

OREGON

†Almac Electronics Corp.
1885 N.W. 169th Place
Beaverton 97005
Tel: (503) 629-8090
FAX: 503-645-0611

Hamilton/Avnet Computer
9409 Southwest Nimbus Ave.
Beaverton 97005

†Hamilton/Avnet Electronics
9409 S.W. Nimbus Ave.
Beaverton 97005
Tel: (503) 627-0201
FAX: 503-641-4012

Wyle
9640 Sunshine Court
Bldg. G, Suite 200
Beaverton 97005
Tel: (503) 643-7900
FAX: 503-646-5466

PENNSYLVANIA

Arrow Electronics, Inc.
650 Seco Road
Monroeville 15146
Tel: (412) 856-7000

Hamilton/Avnet Computer
2800 Liberty Ave., Bldg. E
Pittsburgh 15222

Hamilton/Avnet Electronics
2800 Liberty Ave.
Pittsburgh 15238
Tel: (412) 281-4150

Pioneer/Standard Electronics
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
FAX: 412-963-8255

†Pioneer/Technologies Group, Inc.
Delaware Valley
261 Gibralter Road
Horsham 19044
Tel: (215) 674-4000
FAX: 215-674-3107

TENNESSEE

Arrow Commercial System Group
3635 Knight Road
Suite 7
Memphis 38118
Tel: (901) 367-0540
FAX: (901) 367-2081

TEXAS

Arrow Electronics, Inc.
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
FAX: (214) 248-7208

Hamilton/Avnet Computer
1807A West Braker Lane
Austin 78758

Hamilton/Avnet Computer
Forum 2
4004 Beltline, Suite 200
Dallas 75244

Hamilton/Avnet Computer
4850 Wright Rd., Suite 190
Stafford 77477

†Hamilton/Avnet Electronics
1807 W. Braker Lane
Austin 78758
Tel: (512) 837-8911
TWX: 910-874-1319

†Hamilton/Avnet Electronics
4004 Beltline, Suite 200
Dallas 75234
Tel: (214) 308-8111
TWX: 910-860-5929

†Hamilton/Avnet Electronics
4850 Wright Rd., Suite 190
Stafford 77477
Tel: (713) 240-7733
TWX: 910-881-5523

†Pioneer/Standard Electronics
1826-D Kramer
Austin 78758
Tel: (512) 835-4000
FAX: 512-835-9829

†Pioneer/Standard Electronics
13710 Omega Road
Dallas 75244
Tel: (214) 386-7300
FAX: 214-490-6419

†Pioneer/Standard Electronics
10530 Rockley Road
Houston 77099
Tel: (713) 495-4700
FAX: 713-495-5642

†Wyle Distribution Group
1810 Greenville Avenue
Richardson 75081
Tel: (214) 235-9953
FAX: 214-644-5064

UTAH

Hamilton/Avnet Computer
1585 West 2100 South
Salt Lake City 84119

†Hamilton/Avnet Electronics
1585 West 2100 South
Salt Lake City 84119
Tel: (801) 972-2800
TWX: 910-925-4018

†Wyle Distribution Group
1325 West 2200 South
Suite E
West Valley 84119
Tel: (801) 974-9953

WASHINGTON

†Almac Electronics Corp.
14360 S.E. Eastgate Way
Bellevue 98007
Tel: (206) 643-9992
FAX: 206-643-9709

Hamilton/Avnet Computer
17761 Northeast 78th Place
Redmond 98052

†Hamilton/Avnet Electronics
17761 N.E. 78th Place
Redmond 98052
Tel: (206) 881-6697
FAX: 206-867-0159

Wyle Distribution Group
15385 N.E. 90th Street
Redmond 98052
Tel: (206) 881-1150
FAX: 206-881-1567

WISCONSIN

Arrow Electronics, Inc.
200 N. Patrick Blvd., Ste. 100
Brookfield 53005
Tel: (414) 792-0150
FAX: 414-792-0156

Hamilton/Avnet Computer
20875 Crossroads Circle
Suite 400
Waukesha 53186

†Hamilton/Avnet Electronics
28875 Crossroads Circle
Suite 400
Waukesha 53186
Tel: (414) 784-4510
FAX: 414-784-9509

CANADA

ALBERTA

Hamilton/Avnet Computer
2816 21st Street Northeast
Calgary T2E 6Z2

Hamilton/Avnet Electronics
2816 21st Street N.E. #3
Calgary T2E 6Z3
Tel: (403) 230-3586
FAX: 403-250-1591

Zentronics
6815 #8 Street N.E.
Suite 100
Calgary T2E 7H
Tel: (403) 295-8818
FAX: 403-295-8714

BRITISH COLUMBIA

†Hamilton/Avnet Electronics
8610 Commerce Ct.
Burnaby V5A 4N6
Tel: (604) 420-4101
FAX: 604-437-4712

Zentronics
108-11400 Bridgeport Road
Richmond V6X 1T2
Tel: (604) 273-5575
FAX: 604-273-2413

ONTARIO

Arrow Electronics, Inc.
36 Antares Dr., Unit 100
Nepean K2E 7W5
Tel: (613) 226-6903
FAX: 613-723-2018

†Arrow Electronics, Inc.
1093 Meyerside, Unit 2
Mississauga L5T 1M4
Tel: (416) 673-7769
FAX: 416-672-0849

Hamilton/Avnet Computer
Canada System Engineering Group
3688 Nashua Drive
Units 7 & 8
Mississauga L4V 1M5

Hamilton/Avnet Computer
3688 Nashua Drive
Units 9 & 10
Mississauga L4V 1M5

Hamilton/Avnet Computer
6845 Rexwood Road
Units 7, 8, & 9
Mississauga L4V 1R2

Hamilton/Avnet Computer
190 Colonade Road
Nepean K2E 7J5

†Hamilton/Avnet Electronics
6845 Rexwood Road
Units 3-4-5
Mississauga L4T 1R2
Tel: (416) 677-7432
FAX: 416-677-0940

†Hamilton/Avnet Electronics
190 Colonnade Road South
Nepean K2E 7L5
Tel: (613) 226-1700
FAX: 613-226-1184

†Zentronics
1355 Meyerside Drive
Mississauga L5T 1C9
Tel: (416) 564-9600
FAX: 416-564-8320

†Zentronics
155 Colonnade Road
Unit 17
Nepean K2E 7K1
Tel: (613) 226-8840
FAX: 613-226-6352

QUEBEC

Arrow Electronics Inc.
1100 St. Regis
Dorval H9P 2T5
Tel: (514) 421-7411
FAX: 514-421-7430

Arrow Electronics, Inc.
500 Boul. St-Jean-Baptiste
Suite 280
Quebec G2E 5R9
Tel: (418) 871-7500
FAX: 418-871-6816

Hamilton/Avnet Computer
2795 Rue Halpern
St. Laurent H4S 1P8

†Hamilton/Avnet Electronics
2795 Halpern
St. Laurent H2E 7K1
Tel: (514) 335-1000
FAX: 514-335-2481

†Zentronics
520 McCaffrey
St. Laurent H4T 1N3
Tel: (514) 737-9700
FAX: 514-737-5212

†Certified Technical Distributor

EUROPEAN SALES OFFICES

FINLAND

Intel Finland OY
Ruosilantie 2
00390 Helsinki
Tel: (358) 0 544 644
TLX: 123332

FRANCE

Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex
Tel: (33) (1) 30 57 70 00
TLX: 699016

ISRAEL

Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430
Tel: (972) 03-498080
TLX: 371215

ITALY

Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20094 Assago
Milano
Tel: (39) (02) 89200950
TLX: 341286

NETHERLANDS

Intel Semiconductor B.V.
Postbus 84130
3099 CC Rotterdam
Tel: (31) 10.407.11.11
TLX: 22283

SPAIN

Intel Iberia S.A.
Zurbaran, 28
28010 Madrid
Tel: (34) (1) 308.25.52
TLX: 46880

SWEDEN

Intel Sweden A.B.
Dalvagen 24
171 36 Solna
Tel: (46) 8 734 01 00
TLX: 12261

SWITZERLAND

Intel Semiconductor A.G.
Zuerichstrasse
8185 Winkel-Rueti bei Zuerich
Tel: (41) 01/860 62 62
TLX: 825977

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ
Tel: (44) (0793) 696000
TLX: 444447/8

WEST GERMANY

Intel GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen
Tel: (49) 089/90992-0
FAX: (49) 089/904/3948

Intel GmbH
Abraham Lincoln Strasse 16-18
6200 Wiesbaden
Tel: (49) 06121/7605-0
TLX: 4-186183

Intel GmbH
Zettachring 10A
7000 Stuttgart 80
Tel: (49) 0711/7287-280
TLX: 7-254826

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Electronics G.m.b.H.
Rotenmuehlgasse 26
1120 Wien
Tel: (43) (0222) 83 56 46
TLX: 31532

BELGIUM

Ineico Belgium S.A.
Av. des Croix de Guerre 94
1120 Bruxelles
Oorlogskruisenlaan, 94
1120 Brussel
Tel: (32) (02) 216 01 60
TLX: 64475 or 22090

DENMARK

ITT-Multikomponent
Naverland 29
2600 Glostrup
Tel: (45) (0) 2 45 66 45
TLX: 33 355

FINLAND

OY Fintronic AB
Melkonkatu 24A
00210 Helsinki
Tel: (358) (0) 6926022
TLX: 124224

FRANCE

Almex
Zone industrielle d'Antony
48, rue de l'Aubepine
BP 102
92164 Antony Cedex
Tel: (33) (1) 40 96 54 00
TLX: 250067

LEX Electronics
73-79, Rue des Solets
Silic 585
94663 Rungis Cedex
Tel: (33) (1) 49 78 48 78
TWX: 200485

Metrologie
Tour d'Asnieres
4, av. Laurent-Cely
92606 Asnieres Cedex
Tel: (33) (1) 47 90 62 40
TLX: 611448

Tekelec-Airtronic
Cite des Bruyeres
Rue Carle Vernet - BP 2
92310 Sevres
Tel: (33) (1) 45 34 75 35
TLX: 204552

IRELAND

Micro Marketing Ltd.
Glenageary Office Park
Glenageary
Co. Dublin
Tel: (21) (353) (01) 856288
FAX: (21) (353) (01) 857364
TLX: 31584

ISRAEL

Eastronics Ltd.
11 Rozanis Street
P.O.B. 39300
Tel-Aviv 61392
Tel: (972) 03-475151
TLX: 33638

ITALY

Intesi
Divisione ITT Industries GmbH
Viale Milanofiori
Palazzo E/5
20090 Assago (MI)
Tel: (39) 02/824701
TLX: 311351

Lasi Elettronica S.p.A.
V. le Fulvio Testi, 126
20092 Cinisello Balsamo (MI)
Tel: (39) 02/2440012
TLX: 352040

Telcom S.r.l.
Via M. Civitali 75
20148 Milano
Tel: (39) 02/4049046
TLX: 335654

ITT Multicomponents
Viale Milanofiori E/5
20090 Assago (MI)
Tel: (39) 02/824701
TLX: 311351

Silverstar
Via Dei Gracchi 20
20146 Milano
Tel: (39) 02/49961
TLX: 332189

NETHERLANDS

Koning en Hartman
Elektrotechniek B.V.
Energieweg 1
2627 AP Delft
Tel: (31) (1) 15/609906
TLX: 38250

NORWAY

Nordisk Elektronikk (Norge) A/S
Postboks 123
Smedsvingen 4
1364 Hvalstad
Tel: (47) (02) 84 62 10
TLX: 77546

PORTUGAL

ATD Portugal LDA
Rua Dr. Faria de Vasconcelos, 3 A
1900 Lisboa
Tel: 351 1 847 22 00
FAX: 351 1 847 21 97

SPAIN

ATD Electronica, S.A.
Plaza Ciudad de Viena, 6
28040 Madrid
Tel: (34) (1) 234 40 00
TLX: 42477

Metrologia Iberica, S.A.
Ctra. de Fuencarral, n.80
28100 Alcobendas (Madrid)
Tel: (34) (1) 653 86 11

SWEDEN

Nordisk Elektronik AB
Torshamnsgatan 39
Box 36
164 93 Kista
Tel: (46) 08-03 46 30
TLX: 105 47

SWITZERLAND

Industrade A.G.
Hertistrasse 31
8304 Wallisellen
Tel: (41) (01) 8328111
TLX: 56788

TURKEY

EMPA Electronic
Lindwurmstrasse 95A
8000 Muenchen 2
Tel: (49) 089/53 80 570
TLX: 528573

UNITED KINGDOM

Access Electronic Components Ltd.
Jubilee House, Jubilee Road
Letchworth, Herts SG6 1QH
Tel: (0462) 480888
FAX: (0462) 682467

Bytech Components Ltd.
12A Cedarwood
Chineham Business Park
Crockford Lane
Basingstoke
Hants RG24 0WD
Tel: (0256) 707107
FAX: 0256-707162

Conformix
Rapid House
Oxford Road
High Wycombe
Bucks HP11 2EE
Tel: (0494) 474147
FAX: (0494) 452144

Bytech Systems
Unit 3
The Western Centre
Western Road
Bracknell
Berks RG12 1RW
Tel: (0344) 55333
FAX: (0344) 867270

Jermyn
Vestry Estate
Otford Road
Sevenoaks
Kent TN14 5EU
Tel: (0732) 450144
FAX: (0732) 451251

MMD Ltd.
3 Bennet Court
Bennet Road
Reading
Berks RG2 0QX
Tel: (0734) 313232
FAX: (0734) 313255

Metro Systems
Rapid House
Oxford Road
High Wycombe
Bucks HP11 2EE
Tel: 0494 474171
FAX: 0494 21860

Micro Marketing
Taney Hall
Eglington Terrace
Dundrum
Dublin 14
Eire
Tel: 0001 989 400
FAX: 0001 989 828

Rapid Silicon
3 Bennet Court
Bennet Road
Reading
Berks RG2 0QX
Tel: 0734 752266
FAX: 0734 312728

WEST GERMANY

Electronic 2000 AG
Stahlgruberring 12
8000 Muenchen 82
Tel: (49) 089/42001-0
TLX: 522561

ITT Multikomponent GmbH
Postfach 1265
Bahnhofstrasse 44
7141 Moeglingen
Tel: (49) 07141/4879
TLX: 7264472

Jermyn GmbH
Im Dachsstueck 9
6250 Limburg
Tel: (49) 06431/508-0
TLX: 415257-0

Metrologie GmbH
Meglingerstrasse 49
8000 Muenchen 71
Tel: (49) 089/78042-0
TLX: 5213189

Proelectron Vertriebs GmbH
Max Planck Strasse 1-3
6072 Dreieich
Tel: (49) 06103/30434-3
TLX: 417903

YUGOSLAVIA

H.R. Microelectronics Corp.
2005 de la Cruz Blvd., Ste. 223
Santa Clara, CA 95050
U.S.A.
Tel: (1) (408) 988-0286
TLX: 387452

Rapido Electronic Components
S.p.a.
Via C. Beccaria, 8
34133 Trieste
Italia
Tel: (39) 040/360555
TLX: 460461

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW, 2086
Tel: 61-2975-3300
FAX: 61-2975-3375

BRAZIL

Intel Semiconductores do Brazil LTDA
Avenida Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.
Tel: 55-11-287-5899
TLX: 11-37-557-ISDB
FAX: 55-11-287-5119

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC
Tel: (1) 500-4850
TLX: 22947 INTEL CN
FAX: (1) 500-2953

Intel Semiconductor Ltd.*
10/F East Tower
Bond Center
Queensway, Central
Hong Kong
Tel: (852) 844-4555
FAX: (852) 868-1989

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001
Tel: 91-812-215773
TLX: 953-845-2646 INTEL IN
FAX: 091-812-215067

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26
Tel: 0298-47-8511
TLX: 3656-160
FAX: 0298-47-8450

Intel Japan K.K.*
Hachioji ON Bldg.
4-7-14 Myojin-machi
Hachioji-shi, Tokyo 192
Tel: 0426-48-8770
FAX: 0426-48-8775

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360
Tel: 0485-24-6871
FAX: 0485-24-7518

Intel Japan K.K.*
Kawa-asa Bldg.
2-11-5 Shin-Yokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222
Tel: 045-474-7661
FAX: 045-471-4394

Intel Japan K.K.*
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560
Tel: 06-863-1091
FAX: 06-863-1084

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-3201-3621
FAX: 03-3201-6850

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 450
Tel: 052-204-1261
FAX: 052-204-1285

KOREA

Intel Korea, Ltd.
16th Floor, Life Bldg.
61 Yoido-dong, Youngdeungpo-Ku
Seoul 150-010
Tel: (2) 784-8186
FAX: (2) 784-8096

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #08-03/06
United Square
Singapore 1130
Tel: (65) 250-7811
FAX: (65) 250-9256

TAIWAN

Intel Technology Far East Ltd.
Taiwan Branch Office
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei
Tel: 886-2-716-9660
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

Dafsys S.R.L.
Chacabuco, 90-6 Piso
1069-Buenos Aires
Tel: 54-1-34-7726
FAX: 54-1-34-1871

AUSTRALIA

Email Electronics
15-17 Hume Street
Huntingdale, 3166
Tel: 011-61-3-544-8244
TLX: AA 30895
FAX: 011-61-3-543-8179

NSD-Australia
205 Middleborough Rd.
Box Hill, Victoria 3128
Tel: 03 8900970
FAX: 03 8990819

BRAZIL

Elebra Componentes
Rua Geraldo Flausina Gomes, 78
7 Andar
04575 - Sao Paulo - S.P.
Tel: 55-11-534-9641
TLX: 55-11-54593/54591
FAX: 55-11-534-9424

CHINA/HONG KONG

Novel Precision Machinery Co., Ltd.
Room 728 Trade Square
681 Cheung Sha Wan Road
Kowloon, Hong Kong
Tel: (852) 360-8999
TWX: 32032 NVTNL HX
FAX: (852) 725-3695

INDIA

Micronic Devices
Arun Complex
No. 65 D.V.G. Road
Basavanagudi
Bangalore 560 004
Tel: 011-91-812-600-631
011-91-812-611-365
TLX: 9538458332 MDBG

Micronic Devices
No. 516 5th Floor
Swastik Chambers
Sion, Trombay Road
Chembur
Bombay 400 071
TLX: 9531 171447 MDEV

Micronic Devices
25/8, 1st Floor
Bada Bazaar Marg
Old Rajinder Nagar
New Delhi 110 060
Tel: 011-91-11-5723509
011-91-11-589771
TLX: 031-63253 MDND IN

Micronic Devices
6-3-348/12A Dwarakapuri Colony
Hyderabad 500 482
Tel: 011-91-842-226748

S&S Corporation
1587 Kooser Road
San Jose, CA 95118
Tel: (408) 978-6216
TLX: 820281
FAX: (408) 978-8635

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. 2-14-1 Asano
Kokurakita-ku
Kitakyushu-shi 802
Tel: 093-511-6471
FAX: 093-551-7861

CTC Components Systems Co., Ltd.
4-8-1 Dobashi, Miyamae-ku
Kawasaki-shi, Kanagawa 213
Tel: 044-852-5121
FAX: 044-877-4268

Dia Semicon Systems, Inc.
Flower Hill Shinmachi Higashi-kan
1-23-9 Shinmachi, Setagaya-ku
Tokyo 154
Tel: 03-3439-1600
FAX: 03-3439-1601

Okaya Koki
2-4-18 Sakae
Naka-ku, Nagoya-shi 460
Tel: 052-204-2916
FAX: 052-204-2901

Ryoyo Electro Corp.
Konwa Bldg.
1-12-22 Tsukiji
Chuo-ku, Tokyo 104
Tel: 03-3546-5011
FAX: 03-3546-5044

KOREA

J-Tek Corporation
Dong Sung Bldg. 9/F
158-24, Samsung-Dong, Kangnam-Ku
Seoul 135-090
Tel: (822) 557-8039
FAX: (822) 557-8304

Samsung Electronics
Samsung Main Bldg.
150 Taepyung-Ro-2KA, Chung-Ku
Seoul 100-102
C.P.O. Box 8780
Tel: (822) 751-3680
TWX: KORSST K 27970
FAX: (822) 753-9065

MEXICO

SSB Electronics, Inc.
675 Palomar Street, Bldg. 4, Suite A
Chula Vista, CA 92011
Tel: (619) 585-3253
TLX: 287751 CBALL UR
FAX: (619) 585-8322

Dicopel S.A.
Tochtli 368 Fracc. Ind. San Antonio
Azcapotzalco
C.P. 02760-Mexico, D.F.
Tel: 52-5-561-3211
TLX: 177 3790 Dicome
FAX: 52-5-561-1279

PSI S.A. de C.V.
Fco. Villa esq. Ajusco s/n
Cuernavaca – Morelos
Tel: 52-73-13-9412
FAX: 52-73-17-5333

NEW ZEALAND

Email Electronics
36 Olive Road
Penrose, Auckland
Tel: 011-64-9-591-155
FAX: 011-64-9-592-681

SAUDI ARABIA

AAE Systems, Inc.
642 N. Pastoria Ave.
Sunnyvale, CA 94086
U.S.A.
Tel: (408) 732-1710
FAX: (408) 732-3095
TLX: 494-3405 AAE SYS

SINGAPORE

Electronic Resources Pte, Ltd.
17 Harvey Road
#03-01 Singapore 1336
Tel: (65) 283-0888
TWX: RS 56541 ERS
FAX: (65) 289-5327

SOUTH AFRICA

Electronic Building Elements
178 Erasmus St. (off Watermeyet St.)
Meyerspark, Pretoria, 0184
Tel: 011-2712-803-7680
FAX: 011-2712-803-8294

TAIWAN

Micro Electronics Corporation
12th Floor, Section 3
285 Nanking East Road
Taipei, R.O.C.
Tel: (886) 2-7198419
FAX: (886) 2-7197916

Acer Sertek Inc.
15th Floor, Section 2
Chien Kuo North Rd.
Taipei 18479 R.O.C.
Tel: 886-2-501-0055
TWX: 23756 SERTEK
FAX: (886) 2-5012521

*Field Application Location

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

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

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

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

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

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

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

Order Number: 240875-001
Printed in U.S.A./15K/0591/RRD GC
MICROPROCESSORS

ISBN 1-55512-135-7