intel

i860™ 64-BIT MICROPROCESSOR PROGRAMMER'S REFERENCE MANUAL

i860™ 64-BIT MICROPROCESSOR PROGRAMMER'S REFERENCE MANUAL

1989

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, 386, 386SX, 387, 387SX, 486, 4-SITE, Above, BITBUS, COMMputer, CREDIT, Data Pipeline, ETOX, Genius, i, $\overset{\Delta}{i}$, i860, ICE, iCEL, iCS, iDBP, iDIS, I^{2}ICE, iLBX, i$_m$, iMDDX, iMMX, Inboard, Insite, Intel, int$_e$l, Intel376, Intel386, int$_e$lBOS, Intel Certified, Intelevision, int$_e$ligent Identifier, int$_e$ligent Programming, Intellec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, KEPROM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, ONCE, OpenNET, OTP, PC BUBBLE, Plug-A-Bubble, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, UPI, and VLSiCEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

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.

* MULTIBUS is a patented Intel bus.

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.

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

Intel Corporation
Literature Sales
P.O. Box 58130
Santa Clara, CA 95052-8130

©INTEL CORPORATION 1989

CG-020389

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 quite 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 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 Phone 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).

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.

NETWORK MANAGEMENT SERVICES

Today's networking products are powerful and extremely flexible. The return they can provide on your investment via increased productivity and reduced costs can be very substantial.

Intel offers complete network support, from definition of your network's physical and functional design, to implementation, installation and maintenance. Whether installing your first network or adding to an existing one, Intel's Networking Specialists can optimize network performance for you.

Preface

The Intel i860™ Microprocessor (part number 80860) delivers supercomputer level performance in a single VLSI component. The 64-bit design of the i860 Microprocessor balances integer, floating point, and graphics performance for applications such as engineering workstations, scientific computing, 3-D graphics workstations, and multiuser systems. Its parallel architecture achieves high throughput with RISC design techniques, pipelined processing units, wide data paths, large on-chip caches, and fast one micron CHMOS IV silicon technology.

This book is the basic source of the detailed information that enables software designers and programmers to use the i860 Microprocessor. 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 only to read 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 Microprocessor "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, page-oriented virtual memory, and on-chip caches. Systems designers and administrators may choose to read the introductory sections of each topic.

- Chapter 5, "Core Instructions," presents detailed information about those instructions that deal with memory addressing, integer arithmetic, and control flow.

- Chapter 6, "Floating-Point Instructions," presents detailed information about those instructions that deal with floating-point arithmetic, long-integer arithmetic, and 3-D graphics support. Explains how extremely high performance can be achieved by utilizing the parallelism and pipelining of the i860 Microprocessor.

- Chapter 7, "Traps and Interrupts," deals 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 8, "Programming Model," defines standards for the use of many features of the i860 Microprocessor. 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 9, "Programming Examples," illustrates the use of the i860 Microprocessor 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 Microprocessor Assembler and Linker Reference Manual*, order number 240436
- *i860 Microprocessor Simulator-Debugger Reference Manual*, order number 240437

Notation and Conventions

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.

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 a another register.
- Do not depend on the ability to retain information written into any reserved or undefined bits.
- When loading a register, always load the reserved and undefined bits as zeros or reload them with values previously stored from the same register.

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

CHAPTER 1 **Page**

ARCHITECTURAL OVERVIEW

1.1 Overview 1-1
1.2 Integer Core Unit 1-2
1.3 Floating-Point Unit 1-3
1.4 Graphics Unit 1-4
1.5 Memory Management Unit 1-5
1.6 Caches 1-5
1.7 Parallel Architecture 1-5
1.8 Software Development Environment 1-6
1.8.1 Multiprocessing for High-Performance with Compatibility 1-6

CHAPTER 2

DATA TYPES

2.1 Integer 2-1
2.2 Ordinal 2-1
2.3 Single-Precision Real 2-1
2.4 Double-Precision Real 2-2
2.5 Pixel 2-3
2.6 Real-Number Encoding 2-4

CHAPTER 3

REGISTERS

3.1 Integer Register File 3-1
3.2 Floating-Point Register File 3-1
3.3 Processor Status Register 3-2
3.4 Extended Processor Status Register 3-5
3.5 Data Breakpoint Register 3-6
3.6 Directory Base Register 3-6
3.7 Fault Instruction Register 3-8
3.8 Floating-Point Status Register 3-8
3.9 KR, KI, T, and MERGE Registers 3-11

CHAPTER 4

ADDRESSING

4.1 Alignment 4-2
4.2 Virtual Addressing 4-2
4.2.1 Page Frame 4-2
4.2.2 Virtual Address 4-2
4.2.3 Page Tables 4-4
4.2.4 Page-Table Entries 4-4
4.2.4.1 Page Frame Address 4-4
4.2.4.2 Present Bit 4-5

4.2.4.3 Cache Disable Bit 4-5
4.2.4.4 Write-Through Bit 4-5
4.2.4.5 Accessed and Dirty Bits 4-6
4.2.4.6 Writable and User Bits 4-6
4.2.4.7 Combining Protection of Both Levels of Page Tables 4-7
4.2.5 Address Translation Algorithm 4-7
4.2.6 Address Translation Faults 4-8
4.2.7 Page Translation Cache 4-8
4.3 Caching and Cache Flushing 4-9

CHAPTER 5
CORE INSTRUCTIONS
5.1 Load Integer 5-2
5.2 Store Integer 5-3
5.3 Transfer Integer to F-P Register 5-3
5.4 Load Floating-Point 5-4
5.5 Store Floating-Point 5-5
5.6 Pixel Store 5-6
5.7 Integer Add and Subtract 5-6
5.8 Shift Instructions 5-8
5.9 Software Traps 5-9
5.10 Logical Instructions 5-9
5.11 Control-Transfer Instructions 5-11
5.12 Cache Flush 5-14
5.13 Control Register Access 5-16
5.14 Bus Lock 5-16

CHAPTER 6
FLOATING-POINT INSTRUCTIONS
6.1 Precision Specification 6-1
6.2 Pipelined and Scalar Operations 6-1
6.2.1 Scalar Mode 6-3
6.2.2 Pipelining Status Information 6-3
6.2.3 Precision in the Pipelines 6-4
6.2.4 Transition between Scalar and Pipelined Operations 6-4
6.3 Multiplier Instructions 6-4
6.3.1 Floating-Point Multiply 6-5
6.3.2 Floating-Point Multiply Low 6-6
6.3.3 Floating-Point Reciprocals 6-6
6.4 Adder Instructions 6-6
6.4.1 Floating-Point Add and Subtract 6-7
6.4.2 Floating-Point Compares 6-8
6.4.3 Floating-Point to Integer Conversion 6-9
6.5 Dual Operation Instructions 6-9
6.6 Graphics Unit 6-22

6.6.1 Long-Integer Arithmetic 6-22
6.6.2 3-D Graphics Operations 6-23
6.6.2.1 Z-Buffer Check Instructions 6-24
6.6.2.2 Pixel Add 6-25
6.6.2.3 Z-Buffer Add 6-28
6.6.2.4 OR with MERGE Register 6-30
6.7 Transfer F-P to Integer Register 6-31
6.8 Dual-Instruction Mode 6-31
6.8.1 Core and Floating-Point Instruction Interaction 6-32
6.8.2 Dual-Instruction Mode Restrictions 6-33

CHAPTER 7
TRAPS AND INTERRUPTS

7.1 Types of Traps 7-1
7.2 Trap Handler Invocation 7-1
7.2.1 Saving State 7-2
7.2.2 Returning from the Trap Handler 7-3
7.2.2.1 Determining Where to Resume 7-3
7.2.2.2 Setting KNF 7-4
7.3 Instruction Fault 7-4
7.4 Floating-Point Fault 7-4
7.4.1 Source Exception Faults 7-5
7.4.2 Result Exception Faults 7-6
7.5 Instruction-Access Fault 7-7
7.6 Data-Access Fault 7-7
7.7 Interrupt Trap 7-7
7.8 Reset Trap 7-8
7.9 Pipeline Preemption 7-8
7.9.1 Floating-Point Pipelines 7-8
7.9.2 Load Pipeline 7-9
7.9.3 Graphics Pipeline 7-9
7.9.4 Examples of Pipeline Preemption 7-9

CHAPTER 8
PROGRAMMING MODEL

8.1 Register Assignment 8-1
8.1.1 Integer Registers 8-1
8.1.2 Floating-Point Registers 8-3
8.1.3 Passing Mixed Integer and Floating-Point Parameters in Registers 8-3
8.1.4 Variable Length Parameter Lists 8-3
8.2 Data Alignment 8-3
8.3 Implementing a Stack 8-4
8.3.1 Stack Entry and Exit Code 8-5
8.3.2 Dynamic Memory Allocation on the Stack 8-6
8.4 Memory Organization 8-7

CHAPTER 9
PROGRAMMING EXAMPLES
9.1 Small Integers 9-1
9.2 Single-Precision Divide 9-1
9.3 Double-Precision Divide 9-2
9.4 Integer Multiply 9-3
9.5 Conversion from Signed Integer to Double 9-3
9.6 Signed Integer Divide 9-4
9.7 String Copy 9-5
9.8 Floating-Point Pipeline 9-5
9.9 Pipelining of Dual-Operation Instructions 9-6
9.10 Dual Instruction Mode 9-7
9.11 Cache Strategies for Matrix Dot Product 9-8

APPENDIX A
INSTRUCTION SET SUMMARY

APPENDIX B
INSTRUCTION FORMAT AND ENCODING

APPENDIX C
INSTRUCTION TIMINGS

APPENDIX D
INSTRUCTION CHARACTERISTICS

Figures

Figure	Title	Page
2-1	Pixel Format Example	2-4
3-1	Register Set	3-2
3-2	Processor Status Register	3-3
3-3	Extended Processor Status Register	3-5
3-4	Directory Base Register	3-6
3-5	Floating-Point Status Register	3-9
4-1	Memory Formats	4-1
4-2	Format of a Virtual Address	4-3
4-3	Address Translation	4-3
4-4	Format of a Page Table Entry	4-4
4-5	Invalid Page Table Entry	4-5
6-1	Pipelined Instruction Execution	6-2
6-2	Dual-Operation Data Paths	6-11
6-3	Data Paths by Instruction (1 of 8)	6-13
6-3	Data Paths by Instruction (2 of 8)	6-14
6-3	Data Paths by Instruction (3 of 8)	6-15
6-3	Data Paths by Instruction (4 of 8)	6-16
6-3	Data Paths by Instruction (5 of 8)	6-17
6-3	Data Paths by Instruction (6 of 8)	6-18
6-3	Data Paths by Instruction (7 of 8)	6-19
6-3	Data Paths by Instruction (8 of 8)	6-20
6-4	Data Path Mnemonics	6-21
6-5	PSR Fields for Graphics Operations	6-24
6-6	FADDP with 8-Bit Pixels	6-26
6-7	FADDP with 16-Bit Pixels	6-27
6-8	FADDP with 32-Bit Pixels	6-28
6-9	FADDZ with 16-Bit Z-Buffer	6-29
6-10	64-Bit Distance Interpolation	6-30
6-11	Dual-Instruction Mode Transitions (1 of 2)	6-32
6-11	Dual-Instruction Mode Transitions (2 of 2)	6-33
8-1	Register Allocation	8-2
8-2	Stack Frame Format	8-5
8-3	Example Memory Layout	8-7

Tables

Table	Title	Page
2-1	Pixel Formats	2-3
2-2	Single and Double Real Encodings	2-5
3-1	Values of PS	3-4
3-2	Values of RB	3-7

3-3	Values of RC	3-8
3-4	Values of RM	3-9
4-1	Combining Directory and Page Protection	4-8
5-1	Control Register Encoding	5-16
6-1	DPC Encoding	6-12
6-2	FADDP MERGE Update	6-26
7-1	Types of Traps	7-1
8-1	Register Allocation	8-1
A-1	FADDP MERGE Update	A-4

Examples

Example	Title	Page
5-1	Example of **bla** Usage	5-13
5-2	Cache Flush Procedure	5-15
5-3	Examples of **lock** and **unlock** Usage	5-18
7-1	Saving Pipeline States	7-10
7-2	Restoring Pipeline States (1 of 2)	7-11
7-2	Restoring Pipeline States (2 of 2)	7-12
8-1	Reading Misaligned 32-Bit Value	8-4
8-2	Subroutine Entry and Exit with Frame Pointer	8-6
8-3	Subroutine Entry and Exit without Frame Pointer	8-6
8-4	Possible Implementation of **alloca**	8-6
9-1	Sign Extension	9-1
9-2	Loading Small Unsigned Integers	9-1
9-3	Single-Precision Divide	9-2
9-4	Double-Precision Divide	9-2
9-5	Integer Multiply	9-3
9-6	Single to Double Conversion	9-3
9-7	Signed Integer Divide	9-4
9-8	String Copy	9-5
9-9	Pipelined Add	9-6
9-10	Pipelined Dual-Operation Instruction	9-7
9-11	Dual-Instruction Mode	9-9
9-12	Matrix Multiply, Cached Loads Only (sheet 1 of 2)	9-10
9-12	Matrix Multiply, Cached Loads Only (sheet 2 of 2)	9-11
9-13	Matrix Multiply, Cached and Pipelined Loads (sheet 1 of 2)	9-12
9-13	Matrix Multiply, Cached and Pipelined Loads (sheet 2 of 2)	9-13

Revision Information:

-002:

- Example 5-2, "Cache Flush Procedure" added 2 instructions.
- Flush instruction usage revised (pg. 5-15).
- Data cache not searched for Page Directories and Tables (pg. 4-9).
- Section 4.3 revised.
- Section 8.1.3 revised.

Architectural Overview 1

Chapter 1
Architectural Overview

The Intel i860™ 64-bit Microprocessor defines a complete architecture that balances integer, floating point, and graphics performance. Target applications include engineering workstations, scientific computing, 3-D graphics workstations, and multiuser systems. Its parallel architecture achieves high throughput with RISC design techniques, pipelined processing units, wide data paths, and large on-chip caches.

1.1 OVERVIEW

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

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

Having a data cache as an integral part of the architecture provides support for vector operations. The data cache supports integer 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, the i860 Microprocessor incorporates wide information paths that include:

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

Floating-point vector operations use all three busses.

To drive the graphics and floating point hardware, the i860 Microprocessor includes a RISC integer core processing unit with one-clock instruction execution. This unit also processes conventional integer programs. It provides complete support for standard operating systems, such as UNIX and OS/2.

The i860 Microprocessor supports vector floating-point operations without special vector instructions or vector registers. It accomplishes 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.
- Parallel integer core and floating-point processing units.
- Parallel multiplier and adder units within the floating-point unit.
- Pipelined floating-point hardware units, with both scalar (nonpipelined) and vector (pipelined) variants of floating-point instructions. Software can switch between scalar and pipelined modes.
- Large register set with 32 general-purpose integer registers, each 32-bits wide, and 32 floating-point registers, each 32-bits wide, that 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 pipelines.

There are two classes of instructions:

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

The processor has 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. Within the floating-point unit, special dual-operation instructions (add-and-multiply, subtract-and-multiply) use the adder and multiplier units in parallel. With both dual-instruction mode and dual operation instructions, the i860 Microprocessor 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, performing the Fast Fourier Transform (FFT), and performing graphics transformations.

1.2 INTEGER CORE UNIT

The core unit is the administrative center of the i860 Microprocessor. The core unit fetches both integer and floating-point instructions. It contains the integer register file, and decodes and executes load, store, integer, bit, and control-transfer operations. Its pipelined organization with extensive bypassing and scoreboarding maximizes performance.

A complete list of its instruction categories includes ...

- Loads and stores between memory and the integer and floating-point registers. Floating-point loads can be pipelined in three levels. 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. Arithmetic on large (128-bit or greater) integers can be implemented via short software macros or subroutines. (The graphics unit provides arithmetic for 64-bit integers.)
- Shifts of the integer registers.
- Logical operations on the integer registers.
- Control transfers. There are both direct and indirect branches, a call instruction, and a branch that can be used to form 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.

1.3 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.

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 and 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 the two-result-per-clock rate by overlapping administrative functions with floating point operations.

The RISC design philosophy minimizes circuit delays and enables using of all the available chip space to achieve the greatest performance for floating-point operations. Due to this fact, due to the use of pipelining and parallelism in the floating-point unit, and due to the wide on-chip caches, the i860 Microprocessor achieves extremely high levels of floating-point performance.

The use of RISC design principles implies that the i860 Microprocessor does not have high-level math macro-instructions. High-level math (and other) functions are implemented in software macros and libraries. For example, the i860 Microprocessor does not have a **sin** instruction. The **sin** function is implemented in software on the i860 Microprocessor. The **sin** routine for the i860 Microprocessor, however, will still be very fast due to the extremely 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. Due to the low-level instruction set of the i860 Microprocessor, not all functions defined by the standard are implemented directly by the hardware. The i860 Microprocessor 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 supplies a software library that provides programs for the i860 Microprocessor with full IEEE-compatible arithmetic.

1.4 GRAPHICS UNIT

The graphics unit has special 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 multiple additions on integers stored in the floating-point register file.

These special graphics features focus the chip's high performance 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 Microprocessor assume that:

- The surface of a solid object is drawn with polygon patches whose shapes approximate the original object.
- The color intensities of the vertices of the polygon and their distances from the viewer are known, but the distances and intensities of the other points must be calculated by interpolation.

The graphics instructions of the i860 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.

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

In addition to the special 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 . . .

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

. . . 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 Microprocessor.

1.5 MEMORY MANAGEMENT UNIT

The on-chip MMU of the i860 Microprocessor 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 a two-level structure of page directories and page tables of 1K entries each. Information from these tables is cached in a 64-entry, four-way set-associative memory. The i860 Microprocessor provides 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 386™ and 486™ microprocessors.

1.6 CACHES

In addition to the page translation cache mentioned previously, the i860 Microprocessor contains separate on-chip caches for data and instructions. Caching is transparent, except to systems programmers who must ensure that the data cache is flushed when switching tasks 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.

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.

1.7 PARALLEL ARCHITECTURE

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

- Conventional programs and conventional compilers can use the i860 Microprocessor as a scalar machine and still benefit from the high-performance of the i860 Microprocessor.
- Compilers designed for the vector model can treat the i860 Microprocessor as a vector machine.
- New 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 Microprocessor. The key to treating the i860 Microprocessor as a vector machine is choosing the appropriate vector primitives that the compiler assumes are available on the target machine. (Intel has defined a standard set 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 architecture of the i860 Microprocessor 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 Microprocessor, these programs can achieve even greater performance gain from the i860 Microprocessor.

Designers of compilers for the i860 Microprocessor will find that the i860 Microprocessor offers more flexibility than traditional vector processing. The instruction set of the i860 Microprocessor separates addressing functions from arithmetic functions. Two benefits result from this separation:

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 Microprocessor, it is no longer necessary to pass all elements of a vector several times to implement complex vector operations.

1.8 SOFTWARE DEVELOPMENT ENVIRONMENT

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

- Assembler, linker, C, and FORTRAN compilers, and FORTRAN vectorizer.
- Libraries of higher-level math functions and IEEE-standard exception support. Intel supplies such libraries in a form that can be utilized by a variety of compilers.
- Simulator and debugger.

1.8.1 Multiprocessing for High-Performance with Compatibility

Memory organization of the i860 Microprocessor is compatible with that of the 386™ and 486™ microprocessors (including addresses and page-table entries); all data types are compatible as well (both integers and floating-point numbers). The page-oriented virtual memory management of the i860 Microprocessor is also compatible with that of the 386 and 486 microprocessors. This level of compatibility facilitates use of the i860 Microprocessor in multiprocessor systems with a 386 or 486 microprocessor. Moreover, complete hardware and software support for such multiprocessor systems is available.

An i860 microprocessor can be used with a 386™, 386SX™, or 486™ microprocessor system. The i860 microprocessor extends system performance to supercomputer levels, while the 386/386SX/486 microprocessor provides binary compatibility with existing applications. The compatibility processor provides access to a huge software base supporting a wide variety of I/O devices, communications protocols, and human-interface methods. The computation-intensive applications enjoy the raw computational power of the i860 Microprocessor, while having access to all capabilities and resources of the compatibility processor.

Data Types 2

Chapter 2
Data Types

The i860 Microprocessor provides operations for integer and floating-point data. Integer operations are performed on 32-bit operands with some support also for 64-bit operands. Load and store instructions can reference 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit operands. Floating-point operations are performed on IEEE-standard 32- and 64-bit formats. Graphics oriented instructions operate on arrays of 8-, 16-, or 32-bit pixels.

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 2,147,438,647 ($+2^{31} - 1$). Arithmetic operations on 8- and 16-bit integers can be performed by sign-extending the 8- or 16-bit values to 32 bits, then using the 32-bit operations.

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

Load and store instructions may also reference (in addition to the 32- and 64-bit formats previously mentioned) eight- and 16-bit items in memory. 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

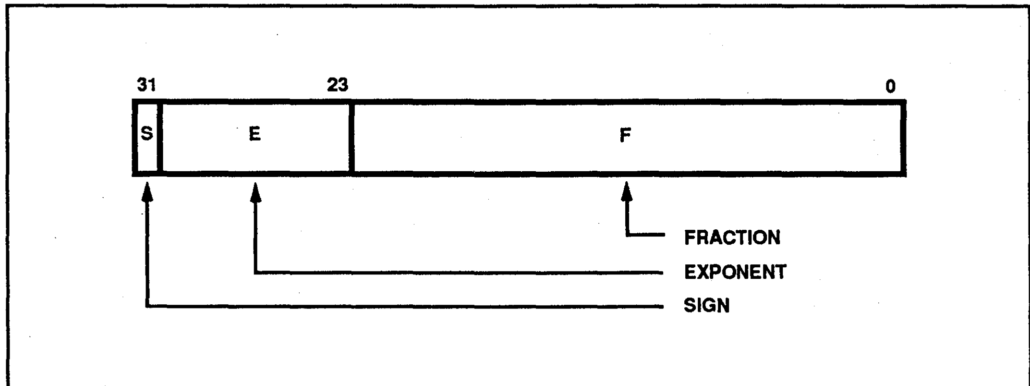
Arithmetic operations are available for 32-bit ordinals. An ordinal is an unsigned integer. An ordinal can represent values in the range 0 to 4,294,967,295 ($+2^{32} - 1$).

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. Bit 31 is the sign bit; bits 30..23 are the exponent; and bits 22..0 are the fraction. In accordance with ANSI/IEEE standard 754, the value of a single-precision real is defined as follows:

1. If **e** = 0 and **f** ≠ 0 or **e** = 255 then generate a floating-point source-exception trap when encountered in a floating-point operation.

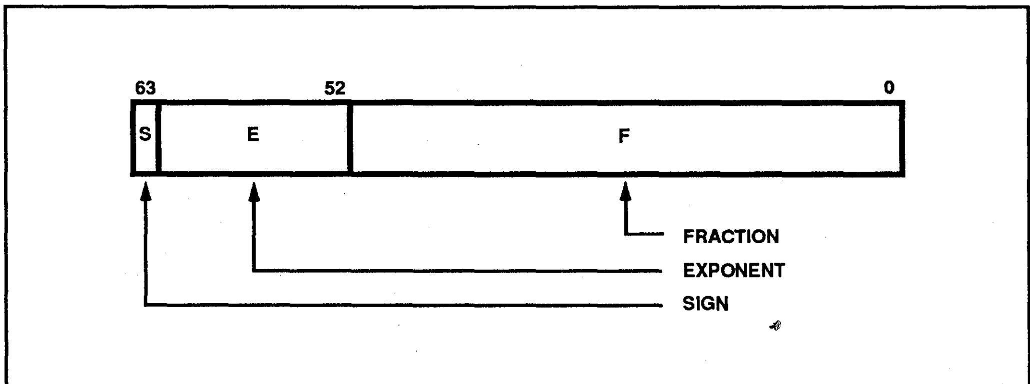


2. If $0 < \mathbf{e} < 255$, then the value is $-1^{s} \times 1.\mathbf{f} \times 2^{e-127}$. (The exponent adjustment 127 is called the *bias*.)

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

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. 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 $\mathbf{e} = 0$ and $\mathbf{f} \neq 0$ or $\mathbf{e} = 2047$, then generate a floating-point source-exception trap when encountered in a floating-point operation.

2. If $0 < \mathbf{e} < 2047$, then the value is $-1^{s} \times 1.\mathbf{f} \times 2^{e-1023}$. (The exponent adjustment 1023 is called the *bias*.)

3. If $\mathbf{e} = 0$ and $\mathbf{f} = 0$, 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 i860 Microprocessor always operates on 64 bits worth of pixels at a time. The pixel data type is used by two kinds of instructions:

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

To perform color intensity shading efficiently in a variety of applications, the i860 Microprocessor defines three pixel formats according to Table 2-1.

Table 2-1. Pixel Formats

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

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

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

Figure 2-1 illustrates one way of assigning meaning to the fields of pixels. These assignments are for illustration purposes only. The i860 Microprocessor defines only the field sizes, not the specific use of each field. Other ways of using the fields of pixels are possible.

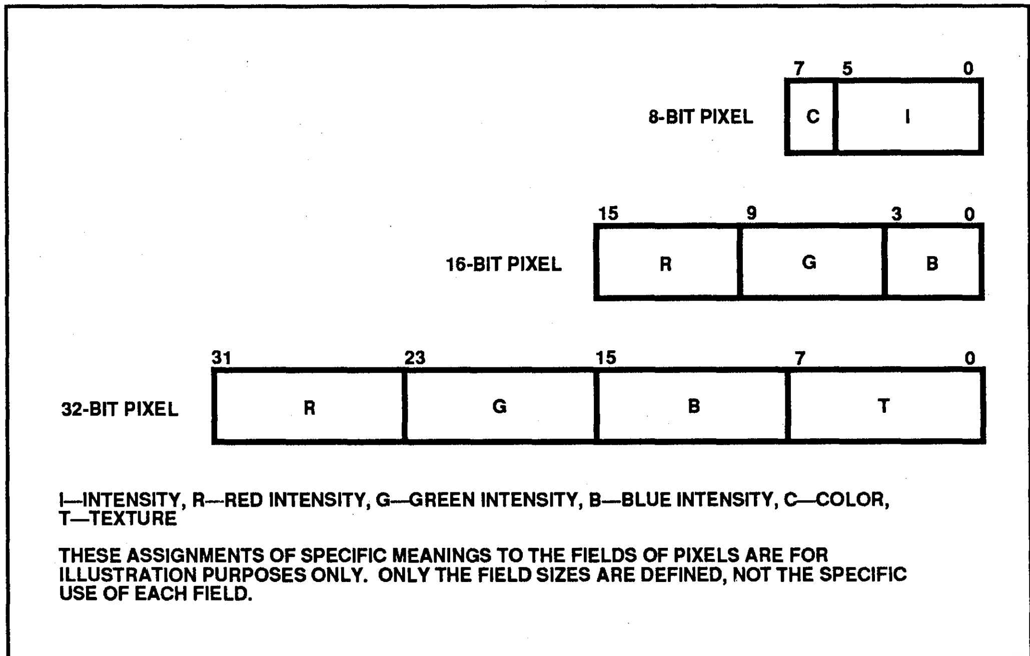


Figure 2-1. Pixel Format Example

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 i860 Microprocessor. The supported values are the normals and the zeros, both positive and negative. Other values are not generated by the i860 Microprocessor, 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 2-2. Single and Double Real Encodings

Class			Sign	Biased Exponent	Significand ff--ff*
POSITIVES	NANS	Quiet	0	11..11	11..11
			.	.	.
			0	11..11	10..00
		Signaling	0	11..11	01..11
			.	.	.
			0	11..11	00..01
		Infinity	0	11..11	00..00
	REALS	Normals	0	11..10	11..11
			.	.	.
			0	00..01	00..00
		Denormals	0	00..00	11..11
			.	.	.
			0	00..00	00..01
		Zero	0	00..00	00..00
NEGATIVES	REALS	Zero	1	00..00	00..00
		Denormals	1	00..00	00..01
			.	.	.
			1	00..00	11..11
		Normals	1	00..01	00..00
			.	.	.
			1	11..10	11..11
		Infinity	1	11..11	00..00
	NANS	Signaling	1	11..11	00..01
			.	.	.
			1	11..11	01..11
		Quiet	1	11..11	10..00
			.	.	.
			1	11..11	11..11
			Single:	< 8 bits>	<- 23 bits ->
			Double:	< 11bits>	<- 52 bits ->

* 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
- Six control registers (**psr**, **epsr**, **db**, **dirbase**, **fir**, and **fsr**)
- Four special-purpose registers (KR, KI, T, and MERGE)

The control registers are accessible only by load and store control-register instructions; the integer and floating-point registers are accessed by arithmetic operations and load and store instructions. The special-purpose registers KR, KI, T, and MERGE are used by a few specific instructions. For information about initialization of registers, refer to the reset trap in Chapter 7. For information about protection as it applies to registers, refer to the **st.c** instruction in Chapter 5.

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, independently of what is stored in it. This special behaviour 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, independently of what is stored in them. 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**, ... , **f30**). The instruction must designate the lowest register number of the set of registers containing 64- or 128-bit values. Misaligned register numbers produce undefined results. The register with the lowest number contains the least significant part of the value. For 128-bit values, the register pair with the lower number contains the 64 bits at the lowest memory address; the register pair with the higher number contains the 64 bits at the highest address.

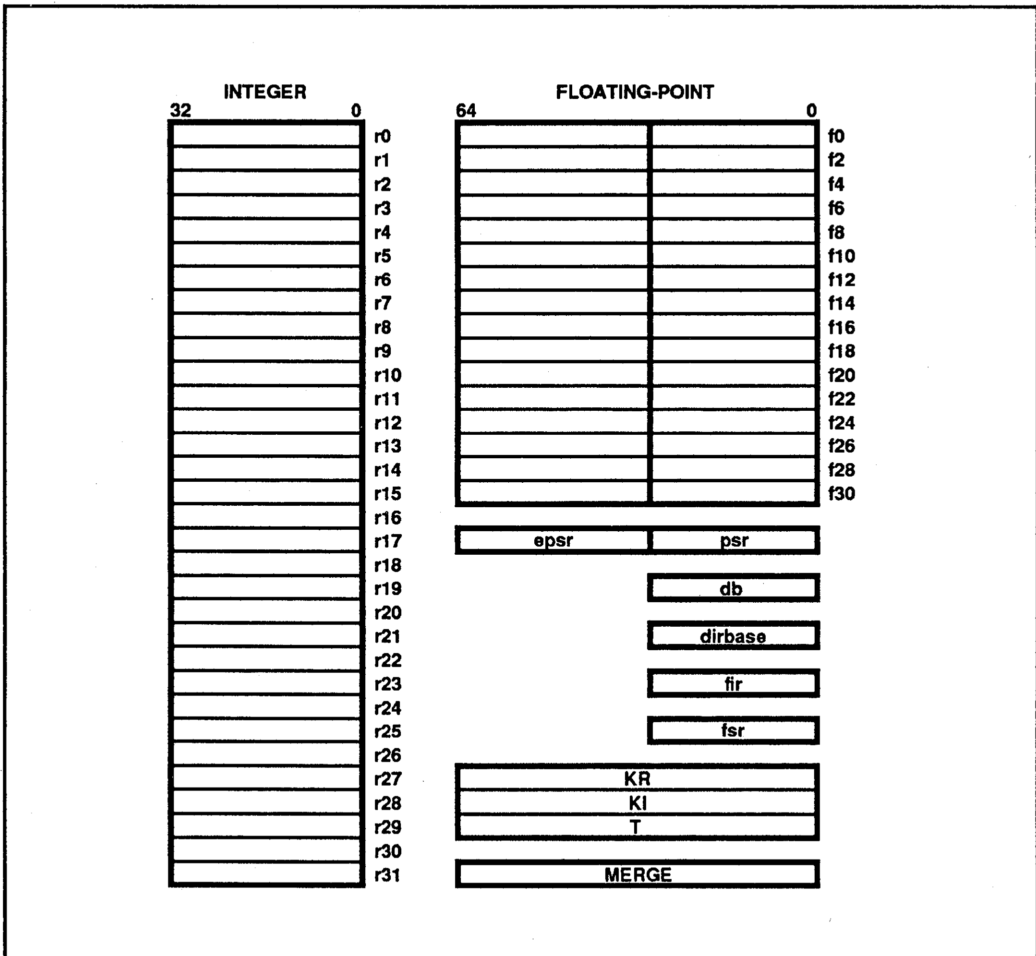


Figure 3-1. Register Set

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**. Fields marked by an asterisk in the figure can be changed only in supervisor mode.

- 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.)

- Various instructions set CC (Condition Code) according to tests they perform, as explained in Chapter 5. The conditional branch instructions test its value. The **bla** instruction described in Chapter 5 sets and tests LCC (Loop Condition Code).

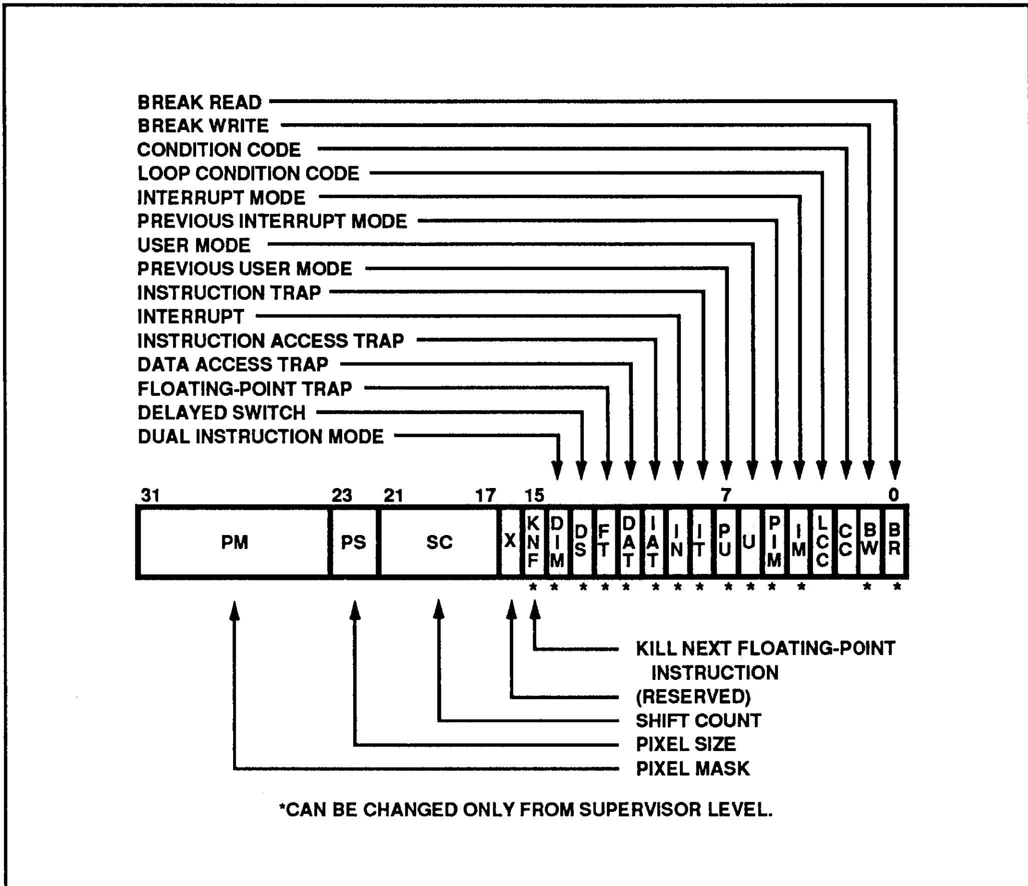


Figure 3-2. Processor Status Register

- IM (Interrupt Mode) enables external interrupts if set; disables interrupts if clear. (Chapter 7 covers 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**. (Chapter 7 provides the details about traps.)

- FT (Floating-Point Trap), DAT (Data Access Trap), IAT (Instruction Access Trap), IN (Interrupt), and IT (Instruction Trap) are trap flags. They are set when the corresponding trap

condition occurs. The trap handler examines these bits to determine which condition or conditions have caused the trap. Refer to Chapter 7 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. Chapter 7 explains 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 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 5.

- PS (Pixel Size) and PM (Pixel Mask) are used by the pixel-store instruction described in Chapter 5 and by the graphics instructions described in Chapter 6. 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**. Fields marked by an asterisk in the figure can be changed only in supervisor mode.

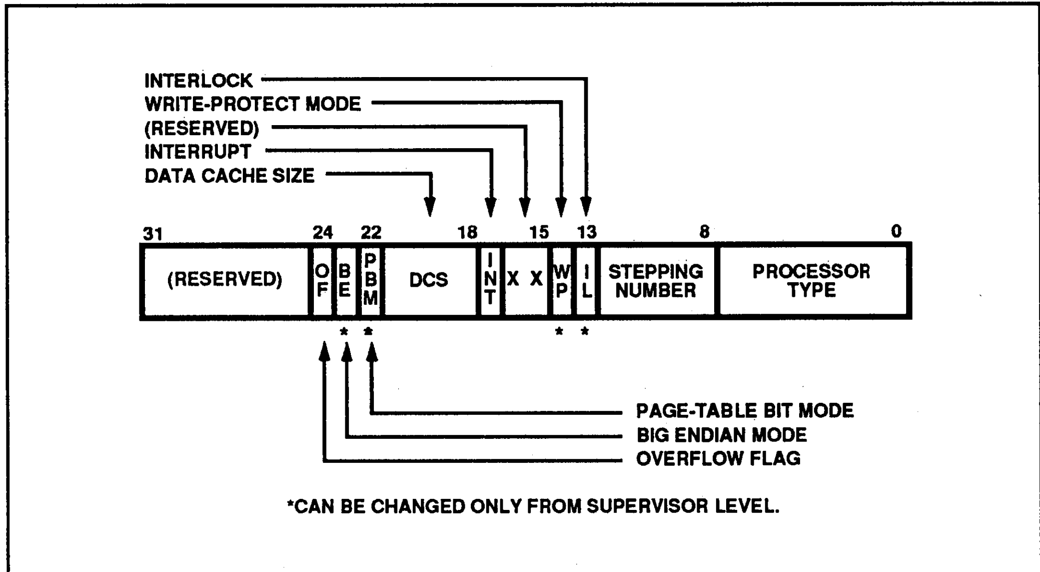


Figure 3-3. Extended Processor Status Register

- The processor type is one for the i860 Microprocessor.
- The stepping number has a unique value that distinguishes among different revisions of the processor.
- IL (Interlock) is set if a trap occurs after a **lock** instruction but before the load or store following the subsequent **unlock** instruction. IL indicates to the trap handler that a locked sequence has been interrupted.
- 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.
- INT (Interrupt) is the value of the INT input pin.
- DCS (Data Cache Size) is a read-only field that tells the size of the on-chip data cache. The number of bytes actually available is 2^{12+DCS}; therefore, a value of zero indicates 4 Kbytes, one indicates 8 Kbytes, etc.
- PBM (Page-Table Bit Mode) determines which bit of page-table entries is output on the PTB pin. When PBM is clear, the PTB signal reflects bit CD of the page-table entry used for the

current cycle. When PBM is set, the PTB signal reflects bit WT of the page-table entry used for the current cycle.

- BE (Big Endian) controls the ordering of bytes within a data item in memory. Normally (i.e. when BE is clear) the i860 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. Refer to Chapter 4 for more endian information.

- OF (Overflow Flag) is set by **adds**, **addu**, **subs**, and **subu** when integer overflow occurs. For **adds** and **subs**, OF is set if the carry from bit 31 is different than the carry from bit 30. For **addu**, OF is set if there is a carry from bit 31. For **subu**, OF is set if there is no carry from bit 31. Under all other conditions, it is cleared by these instructions. OF controls the function of the **intovr** instruction (refer to Chapter 5).

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 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.

3.6 DIRECTORY BASE REGISTER

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

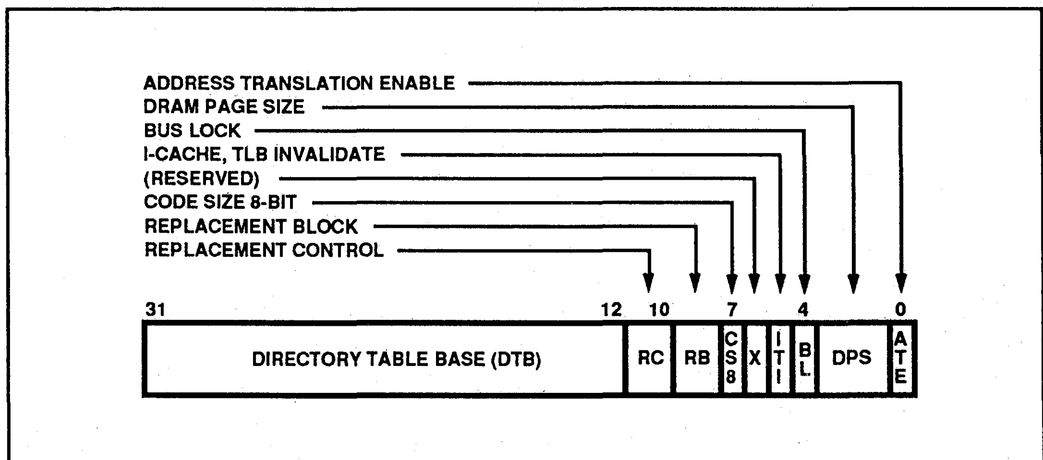


Figure 3-4. Directory Base Register

- ATE (Address Translation Enable), when set, enables the virtual-address translation algorithm described in Chapter 4. The data cache must be flushed before changing the ATE bit.

- DPS (DRAM Page Size) controls how many bits to ignore when comparing the current bus-cycle address with the previous bus-cycle address to generate the NENE# signal. This feature allows for higher speeds when using static column or page-mode DRAMs and consecutive reads and writes access the 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.

- When BL (Bus Lock) is set, external bus accesses are locked. The LOCK# signal is asserted the next bus cycle whose internal bus request is generated after BL is set. It remains set on every subsequent bus cycle as long as BL remains set. The LOCK# signal is deasserted on the next bus cycle whose internal bus request is generated after BL is cleared. Traps immediately clear BL and the LOCK# signal and set 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 5).

- ITI (Instruction-Cache, TLB Invalidate), when set in the value that is loaded into **dirbase**, causes the instruction cache and address-translation cache (TLB) to be flushed. The ITI bit does not remain set in **dirbase**. ITI always appears as zero when read from **dirbase**. The data cache must be flushed before invalidating the TLB.

- When CS8 (Code Size 8-Bit) is set, instruction cache misses are processed as 8-bit bus cycles. When this bit is clear, instruction cache misses are processed as 64-bit bus cycles. This bit can not be set by software; hardware sets this bit at initialization time. It can be cleared by software (one time only) to allow the system to execute out of 64-bit memory after bootstrapping from 8-bit EPROM. A nondelayed branch to code in 64-bit memory should directly follow the **st.c** instruction that clears CS8, in order to make the transition from 8-bit to 64-bit memory occur at the correct time. The branch must be aligned on a 64-bit boundary. Refer to the CS8 mode in the *i860 Hardware Reference Manual* for more information.

- RB (Replacement Block) identifies the cache block to be replaced by cache replacement algorithms. The high-order bit of RB is ignored by the instruction and data caches. RB conditions the cache flush instruction **flush**, which is discussed in Chapter 5. Table 3-2 explains the values of RB.

Table 3-2. Values of RB

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

- 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).

Table 3-3. Values of RC

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

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 Chapter 7. Saving **fir** anytime except the first time after a trap occurs saves the address of the **ld.c** instruction.

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.

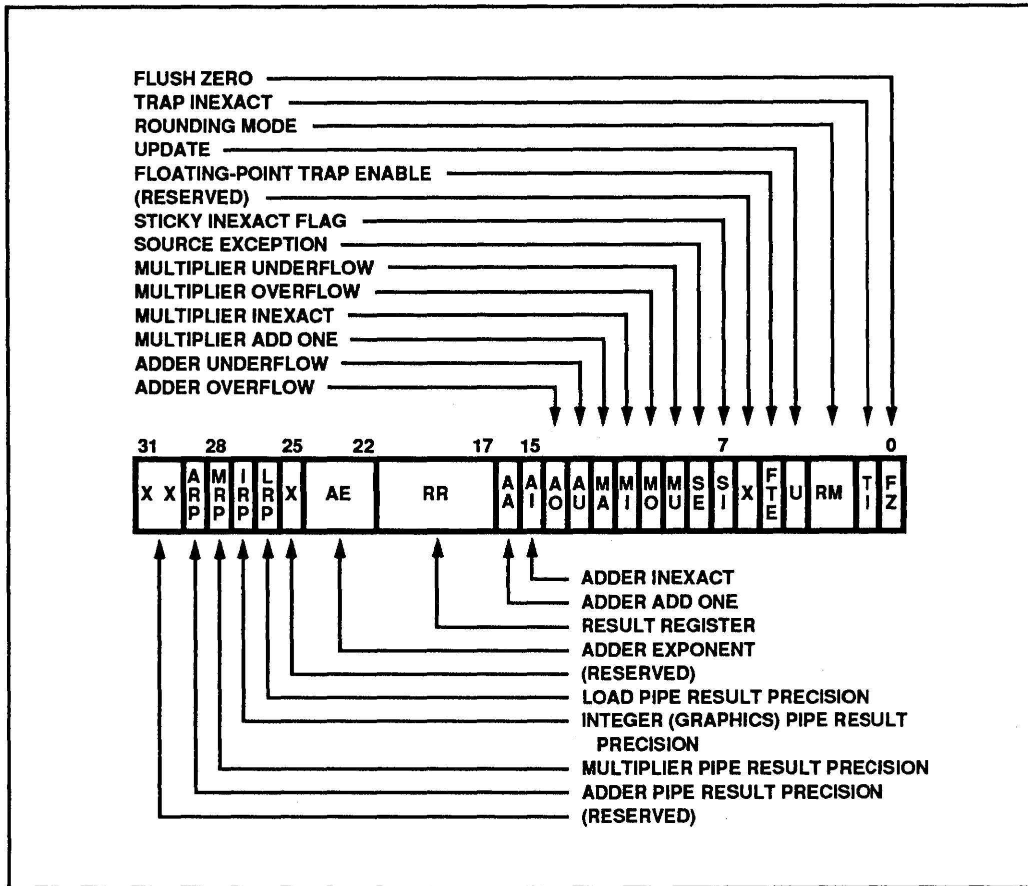


Figure 3-5. Floating-Point Status Register

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*.

- The U-bit (Update Bit), if set in the value that is loaded into **fsr** by a **st.c** instruction, enables updating of the result-status bits (AE, AA, AI, AO, AU, MA, MI, MO, and MU) in the first-stage of the floating-point adder and multiplier pipelines. If this bit is clear, the result-status bits are unaffected by a **st.c** instruction; **st.c** ignores the corresponding bits in the value that is being loaded. An **st.c** always updates **fsr** bits 21..17 and 8..0 directly. The U-bit does

not remain set; it always appears 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 Chapter 7 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 Chapter 7) 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.

- 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), if set, indicates that the adder rounded the result by adding one least significant bit.

- MA (Multiplier Add One), if set, indicates the multiplier rounded the result by one least significant bit.

- 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.

3.9 KR, KI, T, AND MERGE REGISTERS

The KR and KI (''Konstant'') registers and the T (Temporary) register are special-purpose registers used by the dual-operation floating-point instructions described in Chapter 6. The MERGE register is used only by the graphics instructions also presented in Chapter 6. Refer to this chapter for details of their use.

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, the i860 Microprocessor also offers big endian mode, in which the most significant byte of a data item is at the lowest address. Code accesses are always done with little endian addressing. Figure 4-1 shows the difference between the two storage modes. Big endian and little endian data areas should not be mixed within a 64-bit data word. Illustrations of data structures in this manual show data stored in little endian mode, i.e. the rightmost (low-order) byte is at the lowest memory address. The BE bit of **epsr** selects the mode, as described in Chapter 3.

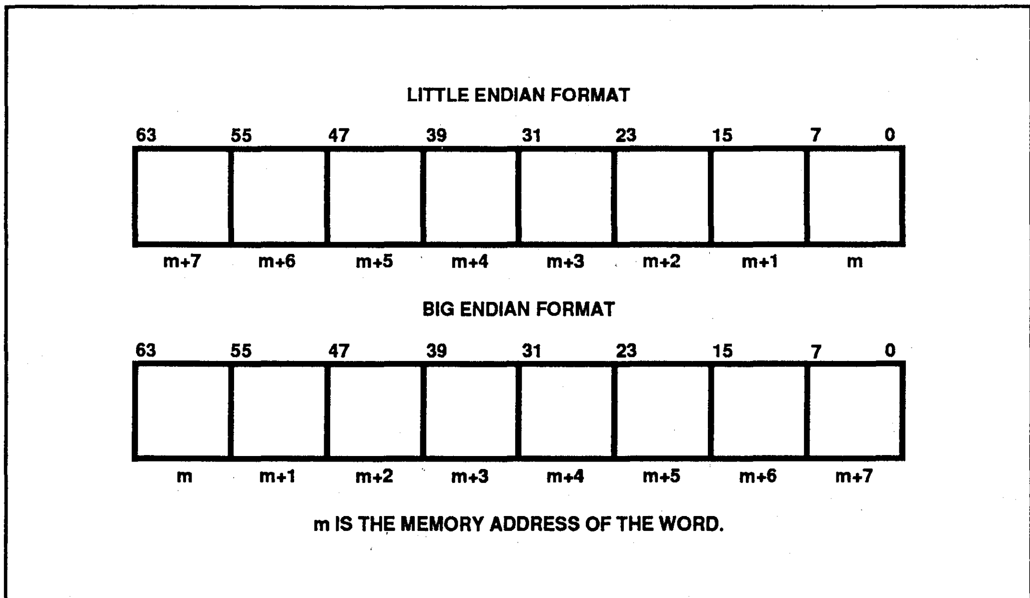


Figure 4-1. Memory Formats

4.1 ALIGNMENT

All data types are addressed by specifying their lowest-addressed byte. Alignment requirements are as follows:

- 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) or a data-access trap occurs.
- 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) or a data-access trap occurs.
- 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) or a data-access trap occurs.
- 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) or a data-access trap occurs.

4.2 VIRTUAL ADDRESSING

When address translation is enabled, the i860 Microprocessor maps instruction and data virtual addresses into physical addresses before referencing memory. This address transformation is compatible with that of the 386™ microprocessor and implements the basic features needed for page-oriented virtual-memory systems and page-level protection.

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

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

4.2.1 Page Frame

A **page frame** is a 4K-byte unit of contiguous addresses of physical main memory. Page frames begin on 4K-byte boundaries and are fixed in size. A **page** is a 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.

4.2.2 Virtual Address

A virtual address refers indirectly to a physical address by specifying a page table, a page within that table, and an offset within that page. Figure 4-2 shows the format of a virtual address.

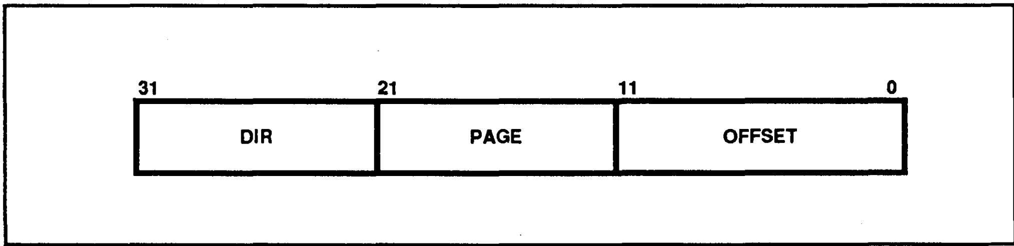


Figure 4-2. Format of a Virtual Address

Figure 4-3 shows how the i860 Microprocessor converts the DIR, PAGE, and OFFSET fields of a virtual address into the physical address by consulting two levels of page tables. The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

Figure 4-3. Address Translation

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 therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.

Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages (2^{20}). Because each page contains 4Kbytes (2^{12} bytes), the tables of one page directory can span the entire physical address space of the i860 Microprocessor ($2^{20} \times 2^{12} = 2^{32}$).

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

4.2.4 Page-Table Entries

Page-table entries (PTEs) in either level of page tables have the same format. Figure 4-4 illustrates this format.

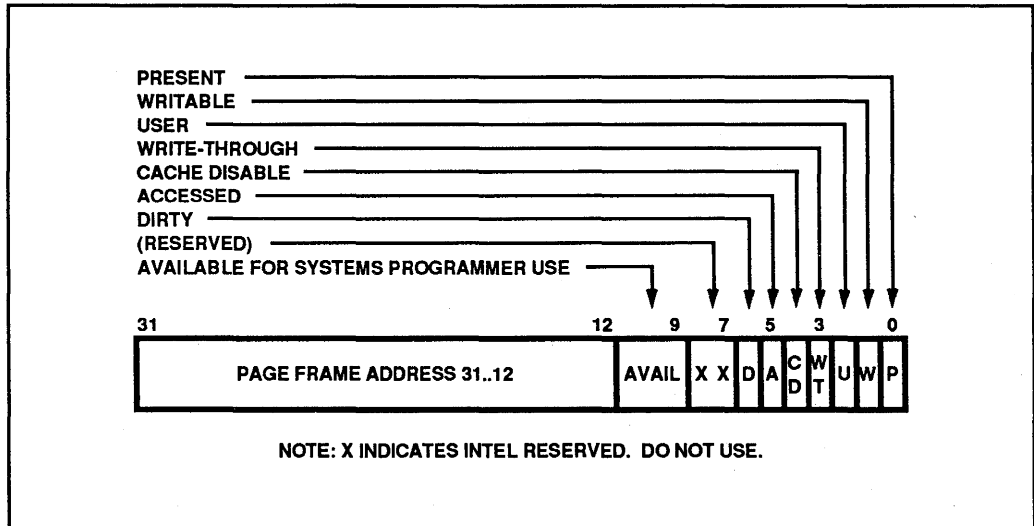


Figure 4-4. Format of a Page Table Entry

4.2.4.1 PAGE FRAME ADDRESS

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

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 4-5 illustrates the format of a page-table entry when P=0.

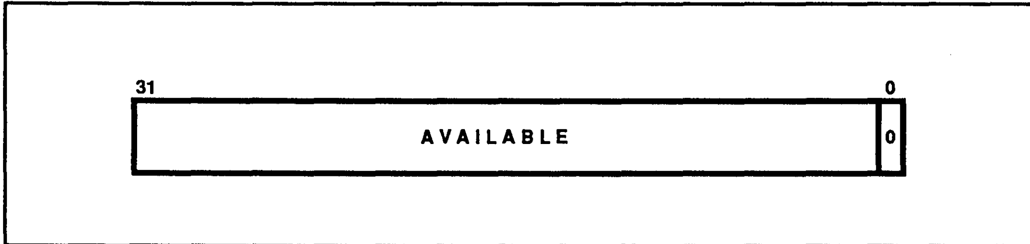


Figure 4-5. Invalid Page Table Entry

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 Chapter 7 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 CACHE DISABLE BIT

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

4.2.4.4 WRITE-THROUGH BIT

The i860 Microprocessor does not implement a write-through caching policy for the on-chip instruction and data caches; however, the WT (write-through) bit in the second-level page-table entry does determine internal caching policy. If WT is set in a PTE, on-chip caching from the corresponding page is inhibited. If WT is clear, the normal write-back policy is applied to data from the page in the on-chip caches. The WT bit of page directory entries is not referenced by the processor, but is **reserved**.

To control external caches, the chip outputs on its PTB pin either CD or WT. The PBM bit of **epsr** determines which bit is output, as described in Chapter 3.

4.2.4.5 ACCESSED AND DIRTY BITS

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

The i860 Microprocessor sets the corresponding accessed bits in both levels of page tables before a read or write operation to a page. The processor tests the dirty bit in the second-level page table before a write to an address covered by that page table entry, and, under certain conditions, causes traps. The trap handler then has the opportunity to maintain appropriate values in the dirty bits. The dirty bit in directory entries is not tested by the i860 Microprocessor. The precise algorithm for using these bits is specified in Section 4.2.5.

An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available. The D and A bits in the PTE (page-table entry) are normally initialized to zero by the operating system. The processor sets the A bit when a page is accessed either by a read or write operation. When a data- or instruction-access fault occurs, the trap handler sets the D bit if an allowable write is being performed, then 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 i860 Microprocessor automatically asserts the LOCK# signal while testing and setting the A bit.

4.2.4.6 WRITABLE AND USER BITS

The W (writable) and U (user) bits are used for page-level protection, which the i860 Microprocessor performs at the same time as address translation. The concept of privilege for pages is implemented by assigning each page to one of two levels:

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

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

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

- The i860 Microprocessor copies the **psr** PU bit into the U bit when an indirect branch is executed and one of the trap bits is set. If PU was one, the i860 Microprocessor enters user level.

- The i860 Microprocessor clears the **psr** U bit to indicate supervisor level when a trap occurs (including when the **trap** instruction causes the trap). The prior value of U is copied into PU. (The trap mechanism is described in Chapter 7; the **trap** instruction is described in Chapter 5.)

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

- When at user level, a read or write of a supervisor-level page causes a trap.

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

When the i860 Microprocessor is executing at supervisor level, all pages are addressable, but, when it is executing at user level, only pages that belong to the user-level are addressable.

When the i860 Microprocessor is executing at supervisor level, all pages are readable. Whether a page is writable depends upon the write-protection mode controlled by WP of **epsr**:

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

When the i860 Microprocessor is executing at user level, only pages that belong to user level and are marked writable are actually writable; pages that belong to supervisor level are neither readable nor writable from user level.

4.2.4.7 COMBINING PROTECTION OF BOTH LEVELS OF PAGE TABLES

For any one page, the protection attributes of its page directory entry may differ form those of its page table entry. The i860 Microprocessor computes the effective protection attributes for a page by examining the protection attributes in both the directory and the page table. Table 4-1 shows the effective protection provided by the possible combinations of protection attributes.

4.2.5 Address Translation Algorithm

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

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

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

10. Form the physical address as PFA2:OFFSET.

11. Deassert LOCK#.

Table 4-1. Combining Directory and Page Protection

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

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

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

4.2.6 Address Translation Faults

The address translation fault is one instance of the data-access fault. (Refer to Chapter 7 for more information on this and other faults.) The instruction causing the fault can be reexecuted by the return-from-trap sequence defined in Chapter 7.

4.2.7 Page Translation Cache

For greatest efficiency in address translation, the i860 Microprocessor stores the most recently used page-table data in an on-chip cache called the TLB (translation lookaside buffer). Only if the necessary paging information is not in the cache must both levels of page tables be referenced.

4.3 CACHING AND CACHE FLUSHING

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

Instruction, data, and address-translation caching on the i860 Microprocessor are not transparent. Writes do not immediately update memory, the TLB, nor the instruction cache. Writes to memory by other bus devices do not update the caches. Under certain circumstances, such as I/O references, self-modifying code, page-table updates, or shared data in a multiprocessing system, it is necessary to bypass or to flush the caches. i860 Microprocessor provides the following methods for doing this:

- **Bypassing Instruction and Data Caches.** If deasserted during cache-miss processing, the KEN# pin disables instruction and data caching of the referenced data. If the CD or WT bit from the associated second-level PTE is set, internal caching of data and instructions is disabled. The value of the CD or WT bit is output on the PTB pin for use by external caches.

- **Flushing Instruction and Address-Translation Caches.** Storing to the **dirbase** register with the ITI bit set invalidates the contents of the instruction and address-translation caches. This bit should be set when a page table or a page containing code is modified or when changing the DTB field of **dirbase**. Note that in order to make the instruction or address-translation caches consistent with the data cache, the data cache must be flushed *before* invalidating the other caches.

NOTE

The mapping of the page containing the currently executing instruction and the next 6 instructions should not be different in the new page tables when **st.c dirbase** changes DTB or activates ITI. The 6 instructions following the **st.c** should be **nops**, and should lie in the same page as the **st.c**.

- **Flushing the Data Cache.** The data cache is flushed by the software routine shown in Chapter 5 with the **flush** instruction. The data cache must be flushed prior to flushing the instruction or address-translation cache (as controlled by the ITI bit of **dirbase**) or enabling or disabling address translation (via the ATE bit).

The i860 CPU searches only external memory for Page Directories and Page Tables, in the translation process. The data cache is not searched. Thus Page Tables and Directories should be kept in non-cacheable memory, or flushed from the cache by any code which accesses them.

Core Instructions 5

Chapter 5
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; and control transfers. All these instructions are executed by the core unit.

Key to abbreviations in the following descriptions of core instructions:

src1	An integer 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.
src2	An integer register.
rdest	An integer register.
freg	A floating-point register.
mem.x(address)	The contents of the memory location indicated by *address* with a size of *x*.
#const	A 16-bit immediate constant or address offset that the i860 Microprocessor sign-extends to 32 bits when computing the effective address.
ctrlreg	One of the control registers **fir**, **epsr**, **psr**, **dirbase**, **db**, or **fsr**.
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.
src1s	An integer register or a 5-bit immediate constant that is zero-extended to 32 bits.
comp2	A function that returns the two's complement of its argument.

The comments regarding optimum performance that appear in the subsections **Programming Notes** are recommendations only. If these recommendations are not followed, the i860 Microprocessor automatically waits the necessary number of clocks to satisfy internal hardware requirements.

5.1 LOAD INTEGER

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

.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 *src1* 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    HIGH16a, r0, r31
ld.l   LOW16(r31), rdest
```

Note that the i860 Microprocessor uses signed addition when it adds LOW16 to **r31**. If bit 15 of LOW16 is set, this has the effect of subtracting from **r31**. Therefore, when bit 15 of LOW16 is set, HIGH16a must be derived by adding one to the high-order 16 bits, so that the net result is correct.

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.

5.2 STORE INTEGER

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

.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 *src1ni* 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 the zero register for *src2*. 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    HIGH16a, r0, r31
st.l   rdest, LOW16(r31)
```

Note that the i860 Microprocessor uses signed addition when it adds LOW16 to **r31**. If bit 15 of LOW16 is set, this has the effect of subtracting from **r31**. Therefore, when bit 15 of LOW16 is set, HIGH16a must be derived by adding one to the high-order 16 bits, so that the net result is correct.

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.

5.3 TRANSFER INTEGER TO F-P REGISTER

ixfr	*src1ni, freg*	**(Transfer Integer to F-P Register)**
freg ← src1ni		

The **ixfr** instruction transfers a 32-bit value from an integer register to a floating-point register.

Programming Notes

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

5.4 LOAD FLOATING-POINT

Floating-Point Load

fld.y	*src1(src2),freg*	**(Normal)**
fld.y	*src1(src2)++ freg*	**(Autoincrement)**

freg ← mem.y *(src1 + src2)*
IF autoincrement
THEN *src2* ← *src1* + *src2*
FI

Pipelined Floating-Point Load

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

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

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

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 i860 Microprocessor stores the effective address into *src2*.

Floating-point loads may be either pipelined or not. 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. When the data is already in the cache, both pipelined and nonpipelined forms of the load instruction read the data from the cache. The pipelined **pfld** instruction, however, does not place the data in the data cache on a cache miss. A **pfld** should be used only when the data is expected to be used once in the near future. 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 does not advance the load pipeline and does not interact with outstanding **pfld** instructions.

Traps

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

Programming Notes

A **pfld** cannot load a 128-bit operand.

For best performance, observe the following guidelines:

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

2. A **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 **pfld** instruction should not directly follow another **pfld**.

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.

5.5 STORE FLOATING-POINT

		Floating-Point Store
fst.y	*freg, src1(src2)*	**(Normal)**
fst.y	*freg, src1(src2)++*	**(Autoincrement)**

```
mem.y (src2 + src1) ← freg
IF autoincrement
THEN src2 ← src1 + src2
FI
```

.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 *src1* to be used as an index register. An autoincrement option supports constant-stride vector addressing. If this option is specified, the i860 Microprocessor stores the effective address into *src2*.

Traps

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

Programming Notes

For best performance, observe the following guidelines:

1. A **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 *freg* 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.

5.6 PIXEL STORE

pst.d	*freg, #const(src2)*	**(Pixel store)**
pst.d	*freg, #const(src2)++*	**(Pixel store autoincrement)**

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

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**. Each bit of PM corresponds to one pixel, with bit 0 corresponding to the pixel at the lowest address.

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 6 for more about **fzchks** and **fzchkl**.

Traps

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

5.7 INTEGER ADD AND SUBTRACT

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.

Programming Notes

For optimum performance, do not perform a conditional branch in the instruction following an add or subtract instruction.

Refer to Chapter 9 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**, *src2*, *rdest* yields the one's complement of *src2*.

addu *src1, src2, rdest* **(Add unsigned)**

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

adds *src1, src2, rdest* **(Add signed)**

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

subu *src1, src2, rdest* **(Subtract unsigned)**

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

subs *src1, src2, rdest* **(Subtract signed)**

rdest ← *src1* − *src2*
OF ← (bit 31 carry ≠ bit 30 carry)
Using signed comparison,
CC set if *src2* > *src1*
CC clear if *src2* ≤ *src1*

When *src1* 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.

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**

*Valid only when const > 0

5.8 SHIFT INSTRUCTIONS

shl *src1, src2, rdest* **(Shift left)**

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

shr *src1, src2, rdest* **(Shift right)**

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

shra *src1, src2, rdest* **(Shift right arithmetic)**

rdest ← *src2* arithmetically shifted right by *src1* bits

shrd *src1ni, src2, rdest* **(Shift right double)**

rdest ← low-order 32 bits of *src1ni:src2* shifted right by SC bits

The arithmetic shift does not change the sign bit; rather, it propagates the sign bit to the right *src1* bits.

Shift counts are taken modulo 32. A **shrd** right-shifts a 64-bit value with *src1* being the high-order 32 bits and *src2* 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 following assembler pseudo-operations utilize the shift instructions:

mov	*src2, rdest*	**(Register-to-register move)**

Assembler pseudo-operation, equivalent to:
shl r0, *src2, rdest*

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
```

5.9 SOFTWARE TRAPS

trap	*src1, src2, rdest*	**(Software trap)**

Generate trap with IT set in **psr**

intovr		**(Software trap on integer overflow)**

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

These instructions generate the instruction trap, as described in Chapter 7.

The **trap** instruction can be used to implement supervisor calls and code breakpoints. The *rdest* should be zero, because its contents are undefined after the operation. The *src1* and *src2* fields can be used to encode the type of trap.

The **intovr** instruction generates an instruction trap if OF bit (overflow flag) of **epsr** is set. It is used to test for integer overflow after the instructions **adds**, **addu**, **subs**, and **subu**.

5.10 LOGICAL INSTRUCTIONS

The operation is performed bitwise on all 32 bits of *src1* and *src2*. When *src1* 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 *src2* operand.

and *src1, src2, rdest* **(Logical AND)**

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

andh *#const, src2, rdest* **(Logical AND high)**

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

andnot *src1, src2, rdest* **(Logical AND NOT)**

rdest ← NOT *src1* AND *src2*
CC set if result is zero, cleared otherwise

andnoth *#const, src2, rdest* **(Logical AND NOT high)**

rdest ← NOT (*#const* shifted left 16 bits) AND *src2*
CC set if result is zero, cleared otherwise

or *src1, src2, rdest* **(Logical OR)**

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

orh *#const, src2, rdest* **(Logical OR high)**

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

xor *src1, src2, rdest* **(Logical XOR)**

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

xorh *#const, src2, rdest* **(Logical XOR high)**

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

Flags Affected

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

Programming Notes

Bit operations can be implemented using logical operations. *Src1* is an immediate constant which 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)

5.11 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 i860 Microprocessor, the results are **undefined**.

Many of the control-transfer instructions are *delayed* transfers. They are delayed in the sense that the i860 Microprocessor executes one additional instruction following the control-transfer instruction before actually transferring control. During the time used to execute the additional instruction, the i860 Microprocessor 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 may be neither another control-transfer instruction, nor a **trap** instruction, nor the target of a control-transfer instruction.

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**, **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.

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 *src1s, src2, sbroff* **(Branch if equal)**

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

btne *src1s, src2, sbroff* **(Branch if not equal)**

IF $src1s \neq src2$
THEN continue execution at *brx(sbroff)*
FI

bla **(Branch on LCC and add)**

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

Programming Notes

The **bla** instruction is useful for implementing loop counters, where *src2* is the loop counter and *src1* 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 next sequential instruction is executed regardless of the setting of LCC. Another **bla** instruction placed as the next to the last instruction of the loop can test for loop completion and update the loop counter. The total number of iterations is the value of *src2* before the first **bla** instruction, plus one. Example 5-1 illustrates this use of **bla**.

Programs should avoid calling subroutines while within a **bla** loop, because a subroutine may use **bla** also and change LCC.

```
// EXAMPLE OF bla USAGE

// Write zeros to an array of 16 single-precision numbers
// Starting address of array is already in r4

    adds    -1,             r0,      r5  // r5 <-- loop increment
    or      15,             r0,      r6  // r6 <-- loop count
    bla     r5,     r6,     CLEAR_LOOP   // One time to initialize LCC
    addu    -4,             r4,      r4  // Start one lower to
                                         //  allow for autoincrement
CLEAR_LOOP:
    bla     r5,     r6,     CLEAR_LOOP   // Loop for the 16 times
    fst.l   f0,             4(r4)++      // Write and autoincrement
                                         //  to next word
```

Example 5-1. Example of bla Usage

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 Chapter 7). 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 that restores *src1ni* 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).

call **(Subroutine call)**

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

calli [*src1ni*] **(Indirect subroutine call)**

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

bri [*src1ni*] **(Branch indirect unconditionally)**

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

5.12 CACHE FLUSH

The **flush** instruction is used to force modified data in the data cache to external memory. Because the contents of *rdest* are undefined after **flush**, translators should encode it as zero. The address *#const* + *src2* must be aligned on a 16-byte boundary. There are two 32-byte blocks in the cache which can be replaced by the address *#const* + *src2*. The particular block that is forced to memory is controlled by the RB field of **dirbase**. When flushing the cache before a task switch, the addresses used by the **flush** instruction should reference non-user-accessible memory to ensure that cached data from the old task is not transferred to the new task. These addresses must be

(Cache flush)

flush	*#const(src2)*	**(Normal)**
flush	*#const(src2)++*	**(Autoincrement)**

Replace the block in data cache that has address (*#const* + *src2*).
Contents of block undefined.
IF autoincrement
THEN *src2* ◄— *#const* + *src2*
FI

Example 5-2 shows how to flush the data cache using the **flush** instruction. The code depends on having reserved a 4 Kbyte memory area that is not used to store data. Cache elements containing modified data are written back to memory by making two passes, each of which references every 32nd byte of this 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 element zero of each set. Before the second pass, RB is changed to one, causing element one of each set to be flushed.

The **flush** instruction must only be used as in Example 5-2. Any other usage of **flush** has undefined results.

```
// CACHE FLUSH PROCEDURE
// Rw, Rx, Ry, Rz represent integer registers
// FLUSH_P_H is the high-order 16 bits of a pointer to reserved area
// FLUSH_P_L is the low-order 16 bits of the pointer, minus 32

    ld.c    dirbase,     Rz
    or      0x800,       Rz,      Rz  // RC <-- 0b10 (assuming was 00)
    adds    -1,          r0,      Rx  // Rx <-- -1 (loop increment)
    call    D_FLUSH
    st.c    Rz,          dirbase      // Replace in block 0
    or      0x900,       Rz,      Rz  // RB <-- 0b01
    call    D_FLUSH
    st.c    Rz,          dirbase      // Replace in block 1
    xor     0x900,       Rz,      Rz  // Clear RC and RB
// Change DTB, ATE, or ITI fields here, if necessary
    st.c    Rz,          dirbase
D_FLUSH:
    orh     FLUSH_P_H,   r0,      Rw  // Rw <-- address minus 32
    or      FLUSH_P_L,   Rw,      Rw  //  of flush area
    or      127,         r0,      Ry  // Ry <-- loop count
    ld.l    32(Rw),      r31          // Clear any pending bus writes
    shl     0,           r31,    r31  // Wait until load finishes
    bla     Rx, Ry, D_FLUSH_LOOP      // One time to initialize LCC
    nop
D_FLUSH_LOOP:
    bla     Rx, Ry, D_FLUSH_LOOP      // Loop; execute next instruction
                                      //   for 128 lines in cache block
    flush   32(Rw)++            // Flush and autoincrement to next line
    bri     r1                        // Return after next instruction
    ld.l    -512(Rw),    r0     // Load from flush area to clear pending
                                //  writes. A hit is guaranteed.
```

Example 5-2. Cache Flush Procedure

5.13 CONTROL REGISTER ACCESS

ld.c	*ctrlreg, rdest*	**(Load from control register)**
rdest ← *ctrlreg*		
st.c	*src1ni, ctrlreg*	**(Store to control register)**
ctrlreg ← *src1ni*		

Ctrlreg 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 i860 Microprocessor is in user mode. *Ctrlreg* is specified by a code in the *src2* field of the instruction, as defined by Table 5-1.

Table 5-1. Control Register Encoding

Register	*Src2* Code	User-Mode Write-Protected?
Fault Instruction	0	N/A
Processor Status	1	Yes*
Directory Base	2	Yes
Data Breakpoint	3	Yes
Floating-Point Status	4	No
Extended Process Status	5	Yes**

* Only the **psr** bits BR, BW, PIM, IM, PU, U, IT, IN, IAT, DAT, FT, DS, DIM, and KNF are write-protected.
** The processor type, stepping number, and cache size cannot be changed from either user or supervisor level.

Programming Notes

Saving **fir** (the fault instruction register) anytime except the first time after a trap occurs saves the address of the **ld.c** instruction.

After a scalar floating-point operation, a **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 Chapter 7 for more details.)

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

5.14 BUS LOCK

These instructions allow programs running in either user or supervisor mode to perform read-modify-write sequences in multiprocessor and multithread systems. The interlocked sequence must not branch outside of the 32 sequential instructions following the **lock** instruction. 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 non-reexecutable store. To insure that no data access fault occurs, it must first store unmodified values in the other store locations. To insure that no instruction access fault occurs, the code that is not restartable should not span a page boundary.

lock **(Begin interlocked sequence)**

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

unlock **(End interlocked sequence)**

Clear BL in **dirbase**. The next load or store that misses the cache unlocks the bus.

After a **lock** instruction, the bus is not locked until the first data access that misses the data cache. Software in a multiprocessing system should ensure that the first load instruction after a **lock** references noncacheable memory. Likewise, after an **unlock** instruction, the bus is not unlocked until the first data access that misses the data cache. Software in a multiprocessing system should ensure that the first load or store instruction after an **unlock** references noncacheable memory.

If a trap occurs after a **lock** instruction and before the load or store that follows the corresponding **unlock**, the processor clears BL and sets the IL (interlock) bit of **epsr**.

If the processor encounters another **lock** instruction before unlocking the bus, that instruction is ignored.

If, following a **lock** instruction, the processor does not encounter a load or store following an **unlock** instruction by the time it has executed 32 instructions, it triggers an instruction fault on the 32nd instruction. In such a case, the trap handler will find both IL and IT set.

Example 5-3 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    //

// 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 5-3. Examples of lock and unlock Usage

6 Floating-Point Instructions

Chapter 6
Floating-Point Instructions

The floating-point section of the i860 Microprocessor 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 of the i860 Microprocessor executes not only floating-point operations but also 64-bit integer operations and graphics operations that utilize the 64-bit internal data path of the floating-point section.

Floating-point instruction operands *src1*, *src2*, and *rdest* refer to one of the 32 floating-point registers; *ireg* refers to one of the integer registers.

6.1 PRECISION SPECIFICATION

Unless otherwise specified, floating-point operations accept single- or double-precision source operands and produce a result of equal or greater precision. Both input operands must have the same precision. The source and result precision are specified by a two-letter suffix to the mnemonic of the operation, as shown below. In this manual, the suffix *.p* refers to the precision specification. In an actual program, *.p* is to be replaced by the appropriate two-letter suffix.

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

6.2 PIPELINED AND SCALAR OPERATIONS

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

Figure 6-1 illustrates three-stage pipelining as found in the floating-point adder (also in the floating-point multiplier when single-precision input operands are employed). The columns of the

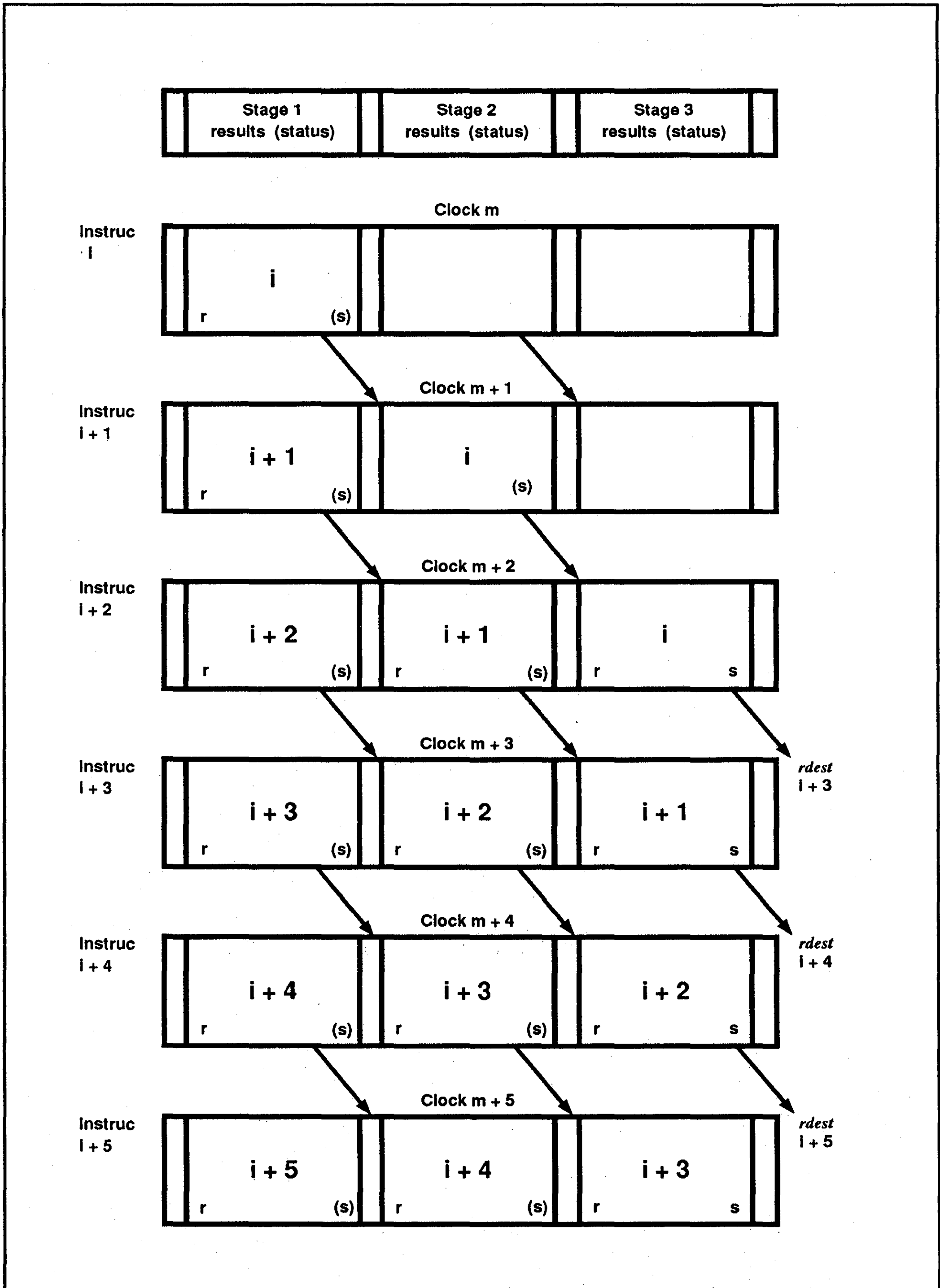


Figure 6-1. Pipelined Instruction Execution

figure represent the three stages of the pipeline. Each stage 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, all of one type (i.e. all adder instructions or all single-precision multiplier instructions). The instructions are represented as **i**, **i+1**, etc. The rows of the figure represent the states of the unit at successive clock cycles. Each time a pipelined operation is performed, the status of the last stage becomes available in **fsr**, the result of the last stage of the pipeline is stored in the destination register *rdest*, the pipeline is advanced one stage, and the input operands *src1* and *src2* are transferred to the first stage of the pipeline.

In the i860 Microprocessor, the number of pipeline stages ranges from one to three. A pipelined operation with a three-stage pipeline stores the result of the third prior operation. A pipelined operation with a two-stage pipeline stores the result of the second prior operation. A pipelined operation with a one-stage pipeline stores the result of the prior operation.

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

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

6.2.1 Scalar Mode

In addition to the pipelined execution mode described above, the i860 Microprocessor also can execute floating-point instructions in "scalar" mode. Most floating-point instructions have both pipelined and scalar variants, distinguished by a bit in the instruction encoding. In scalar mode, the floating-point unit does not start a new operation until the previous floating-point operation is completed. The scalar operation passes through all stages of its pipeline before a new operation is introduced, and the result is stored automatically. Scalar mode is used when the next operation depends on results from the previous few floating-point operations (or when the compiler or programmer does not want to deal with pipelining).

6.2.2 Pipelining Status Information

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

1. It is calculated by the last stage of the pipeline. This is the normal case.

2. It is propagated from the first stage of the pipeline. This method is used when restoring the state of the pipeline after a preemption. When a store instruction updates the **fsr** and the 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 i860 Microprocessor traps if any of the status bits of the **fsr** indicate exceptions. Note that the instruction that creates the exceptional condition is not the instruction at which the trap occurs.

6.2.3 Precision in the Pipelines

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

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

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

6.2.4 Transition between Scalar and Pipelined Operations

When a scalar operation is executed in the adder, multiplier, or graphics units, 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 extract 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** or **ld** instructions.

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

6.3 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 provides a special type of multiply that assists in coding integer multiply sequences. The multiply instruction 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 flushing 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 previous multiplier operation is advanced 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.

6.3.1 Floating-Point Multiply

fmul.p	*src1, src2, rdest*	**(Floating-Point Multiply)**
rdest ← *src1* × *src2*		
pfmul.p	*src1, src2, rdest*	**(Pipelined Floating-Point Multiply)**
rdest ← last M-stage result Advance M pipeline one stage M pipeline first stage ← *src1* × *src2*		
pfmul3.dd	*src1, src2, rdest*	**(Three-Stage Pipelined Multiply)**
rdest ← last M-stage result Advance 3-stage M pipeline one stage M pipeline first stage ← *src1* × *src2*		

These instructions perform a standard multiply operation.

Programming Notes

Src1 must not be the same as *rdest* for pipelined operations. For best performance when the prior operation is scalar, *src1* should not be the same as the *rdest* of the prior operation.

The **pfmul3.dd** instruction is intended primarily for use by exception handlers in restoring pipeline contents (refer to "Pipeline Preemption" in Chapter 7). It should not be mixed in instruction sequences with other pipelined multiplier instructions.

6.3.2 Floating-Point Multiply Low

fmlow.dd	*src1, src2, rdest*	**(Floating-Point Multiply Low)**

rdest ← low-order 53 bits of (*src1* significand × *src2* significand)
rdest bit 53 ← most significant bit of (*src1* significand × *src2* significand)

The **fmlow** instruction multiplies the low-order bits of its operands. It operates only on double-precision operands. The high-order 10 bits of the result are undefined.

An **fmlow** can perform 32-bit integer multiplies. Two 64-bit values are formed, with the integers in the low-order 32 bits. The low-order 32-bits of the result are the same as the low-order 32 bits of an integer multiply. The **fmlow** instruction does not update the result-status bits of **fsr** and does not cause source- or result-exception traps.

6.3.3 Floating-Point Reciprocals

frcp.p	*src2, rdest*	**(Floating-Point Reciprocal)**

rdest ← 1 / *src2* with absolute significand error $< 2^{-7}$

frsqr.p	*src2, rdest*	**(Floating-Point Reciprocal Square Root)**

rdest ← $1 / \sqrt{src2}$ with absolute significand 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 set *src1* to zero. 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

The instructions **frcp** and **frsqr** cause the source-exception trap if *src2* is zero. An **frsqr** causes the source-exception trap if $src2 < 0$.

6.4 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.

6.4.1 Floating-Point Add and Subtract

fadd.p	*src1, src2, rdest*	**(Floating-Point Add)**
rdest ← *src1* + *src2*		
pfadd.p	*src1, src2, rdest*	**(Pipelined Floating-Point Add)**
rdest ← last A-stage result Advance A pipeline one stage A pipeline first stage ← *src1* + *src2*		
fsub.p	*src1, src2, rdest*	**(Floating-Point Subtract)**
rdest ← *src1* − *src2*		
pfsub.p	*src1, src2, rdest*	**(Pipelined Floating-Point Subtract)**
rdest ← last A-stage result Advance A pipeline one stage A pipeline first stage ← *src1* − *src2*		

These instructions perform standard addition and subtraction operations.

Programming Notes

In order to allow conversion from double precision to single precision, an **fadd** or **pfadd** instruction may have double-precision inputs and a single-precision output, as long as one of its input operands is **f0**. In assembly language, this conversion is specified using the **fmov** or **pfmov** pseudoinstruction with the **.ds** suffix.

fmov.ds	*src1*, *rdest*	**(Convert Double to Single)**
Equivalent to **fadd.ds** *src1*, **f0**, *rdest*		
pfmov.ds	*src1*, *ireg*	**(Pipelined Convert Double to Single)**
Equivalent to **pfadd.ds** *src1*, **f0**, *rdest*		

Conversion from single to double is accomplished by **fadd.sd** or **pfadd.sd** with **f0** as one input operand. In assembly language, this conversion is specified by the **fmov** or **pfmov** pseudoinstruction with the **.sd** suffix.

fmov.sd	*src1*, *rdest*	**(Convert Single to Double)**
Equivalent to fadd.sd *src1*, **f0**, *rdest*		
pfmov.sd	*src1*, *ireg*	**(Pipelined Convert Single to Double)**
Equivalent to **pfadd.sd** *src1*, **f0**, *rdest*		

6.4.2 Floating-Point Compares

pfgt.p *src1, src2, rdest* **(Pipelined Floating-Point Greater-Than Compare)**

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

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

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

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

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

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.

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.

6.4.3 Floating-Point to Integer Conversion

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

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

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

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

ftrunc.p *src1, rdest* **(Floating-Point to Integer Truncation)**

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

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

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

The instructions **fix** and **pfix** must specify double-precision results. The low-order 32 bits of the result contain the integer part of *src1* represented in twos-complement form. 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. *Src2* should contain zero.

Traps

The instructions **fix**, **pfix**, **ftrunc**, and **pftrunc** signal overflow if the integer part of *src1* is bigger than what can be represented as a 32-bit twos-complement integer. Underflow and inexact are never signaled.

6.5 DUAL OPERATION INSTRUCTIONS

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.

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

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

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

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

pfmam.p *src1, src2, rdest* **(Pipelined Floating-Point Multiply with Add)**

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

pfmsm.p *src1, src2, rdest* **(Pipelined Floating-Point Multiply with Subtract)**

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

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 *rdest* (the adder result is lost).

Six operands are required, but the instruction format specifies only three operands; therefore, there are special provisions for specifying the operands. These special provisions consist of:

- Three special registers (KR, KI, and T), that can store values from one dual-operation instruction and supply them as inputs to subsequent dual-operation instructions.

— The constant registers KR and KI can store the value of *src1* and subsequently supply that value to the M-pipeline in place of *src1*.

— The transfer register T can store the last-stage result of the multiplier pipeline and subsequently supply that value to the adder pipeline in place of *src1*.

- A four-bit data-path control field in the opcode (DPC) that specifies the operands and loading of the special registers.

1. Operand-1 of the multiplier can be KR, KI, or *src1*.
2. Operand-2 of the multiplier can be *src2*, 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 *src1*, 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 *src2*, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.

Figure 6-2 shows all the possible data paths surrounding the adder and multiplier. Table 6-1 shows how the various encodings of DPC select different data paths. Figure 6-3 illustrates the actual data path for each dual-operation instruction.

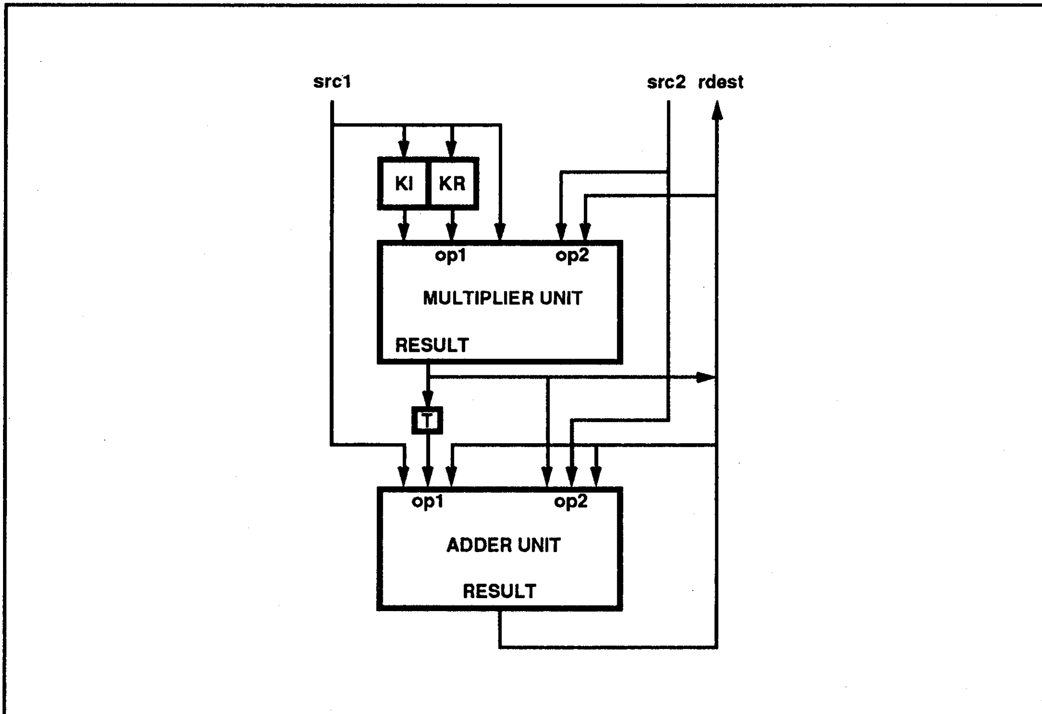


Figure 6-2. Dual-Operation Data Paths

Table 6-1. 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	mm12tpm	mm12tsm	src1	src2	T	M result	No	No

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

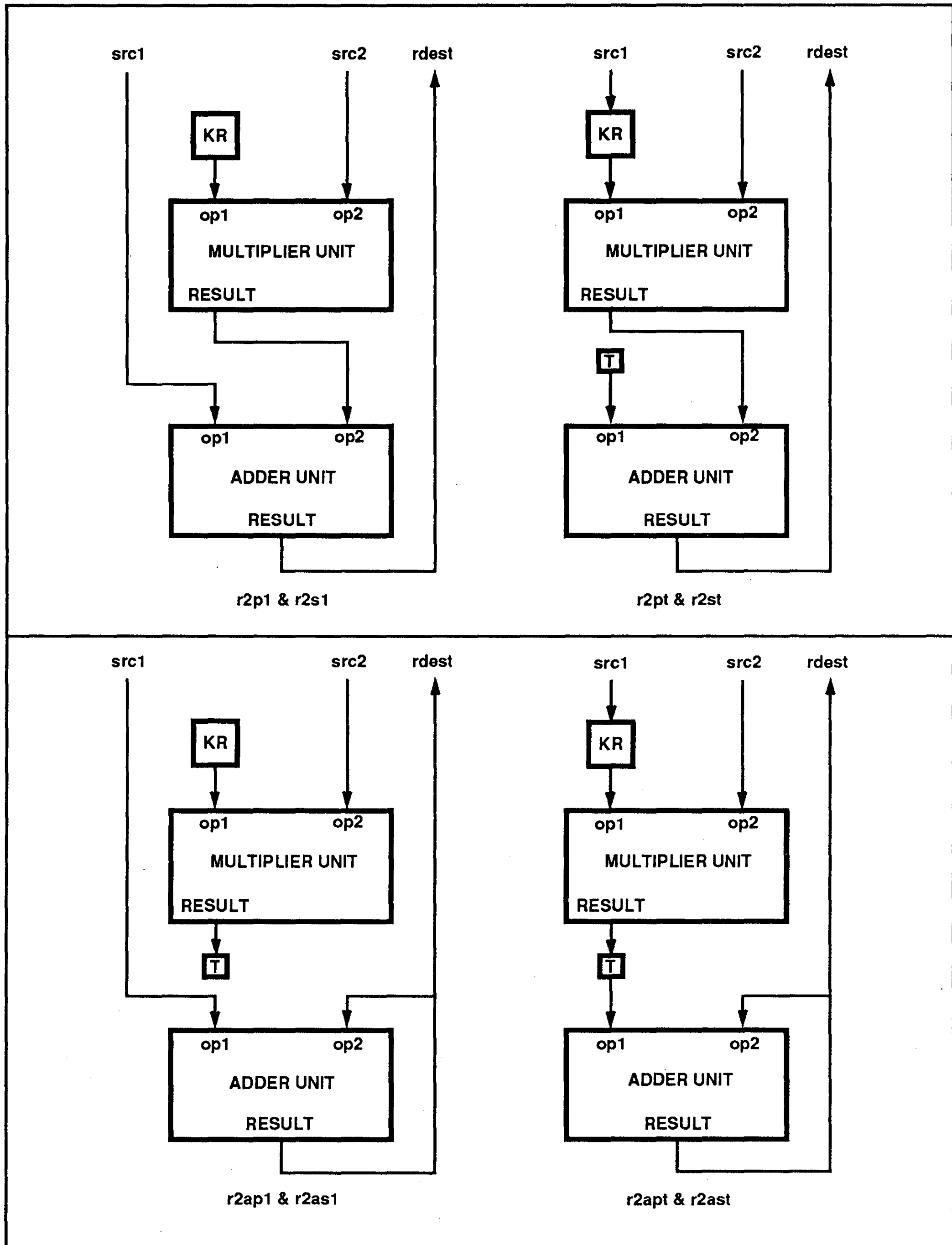


Figure 6-3. Data Paths by Instruction (1 of 8)

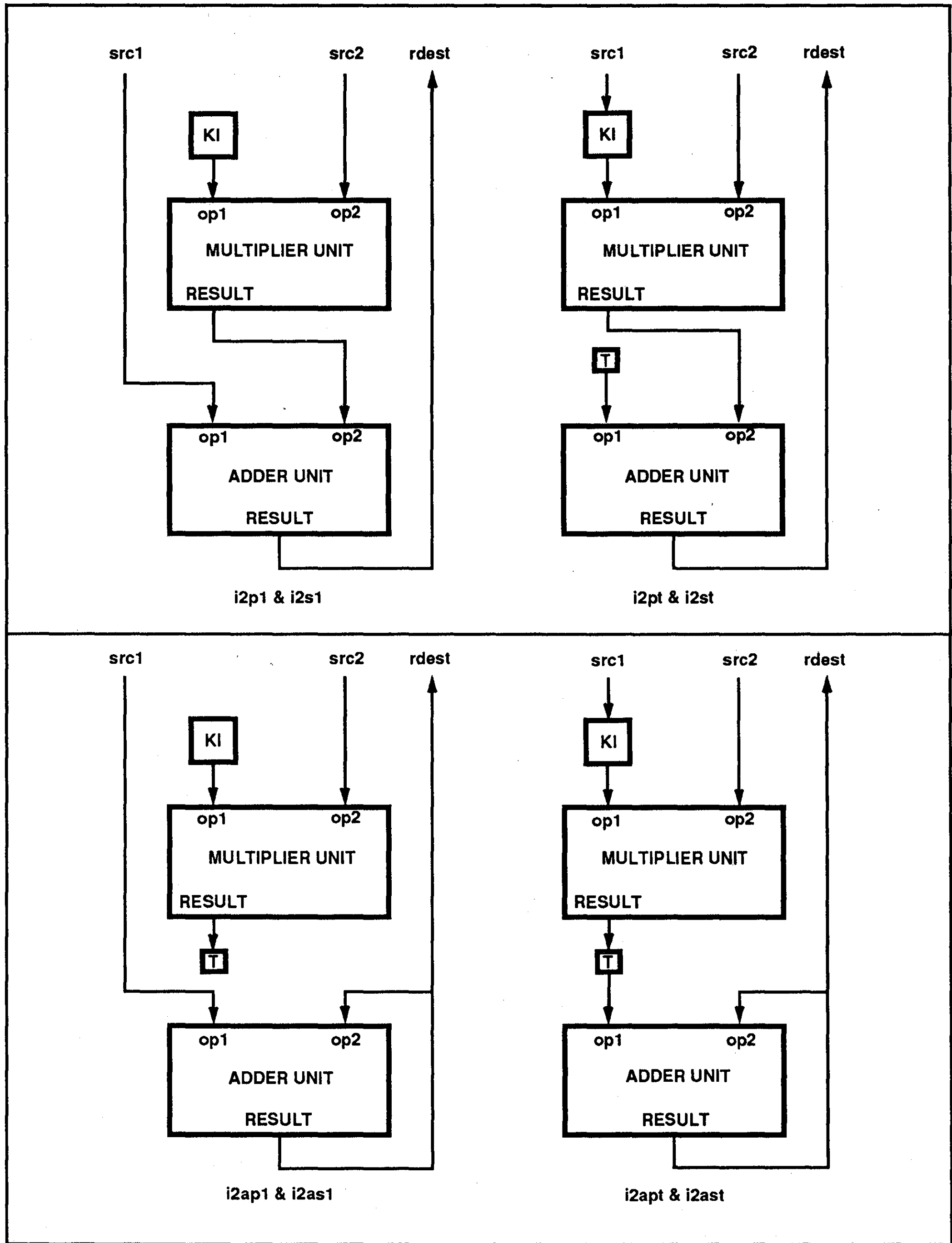


Figure 6-3. Data Paths by Instruction (2 of 8)

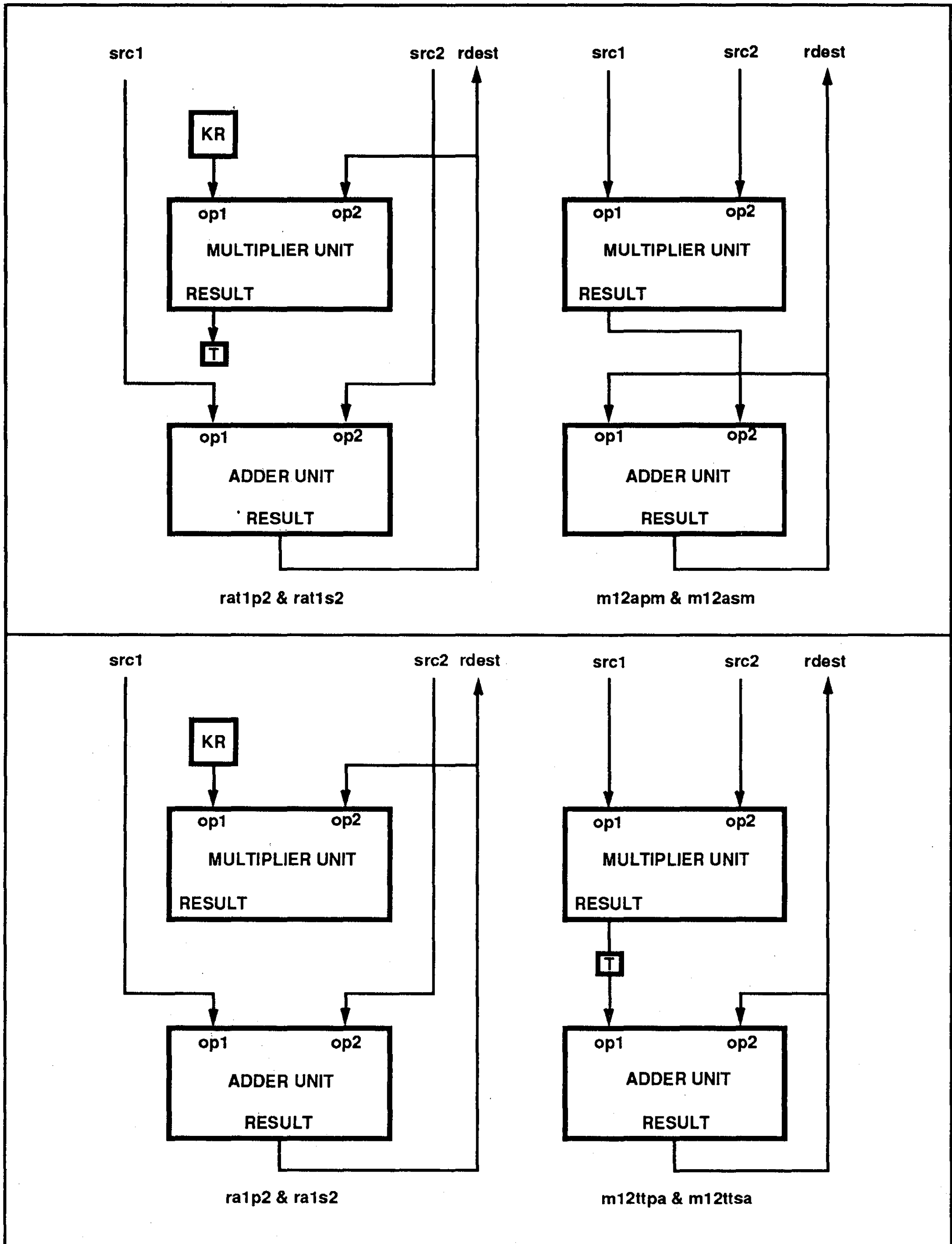


Figure 6-3. Data Paths by Instruction (3 of 8)

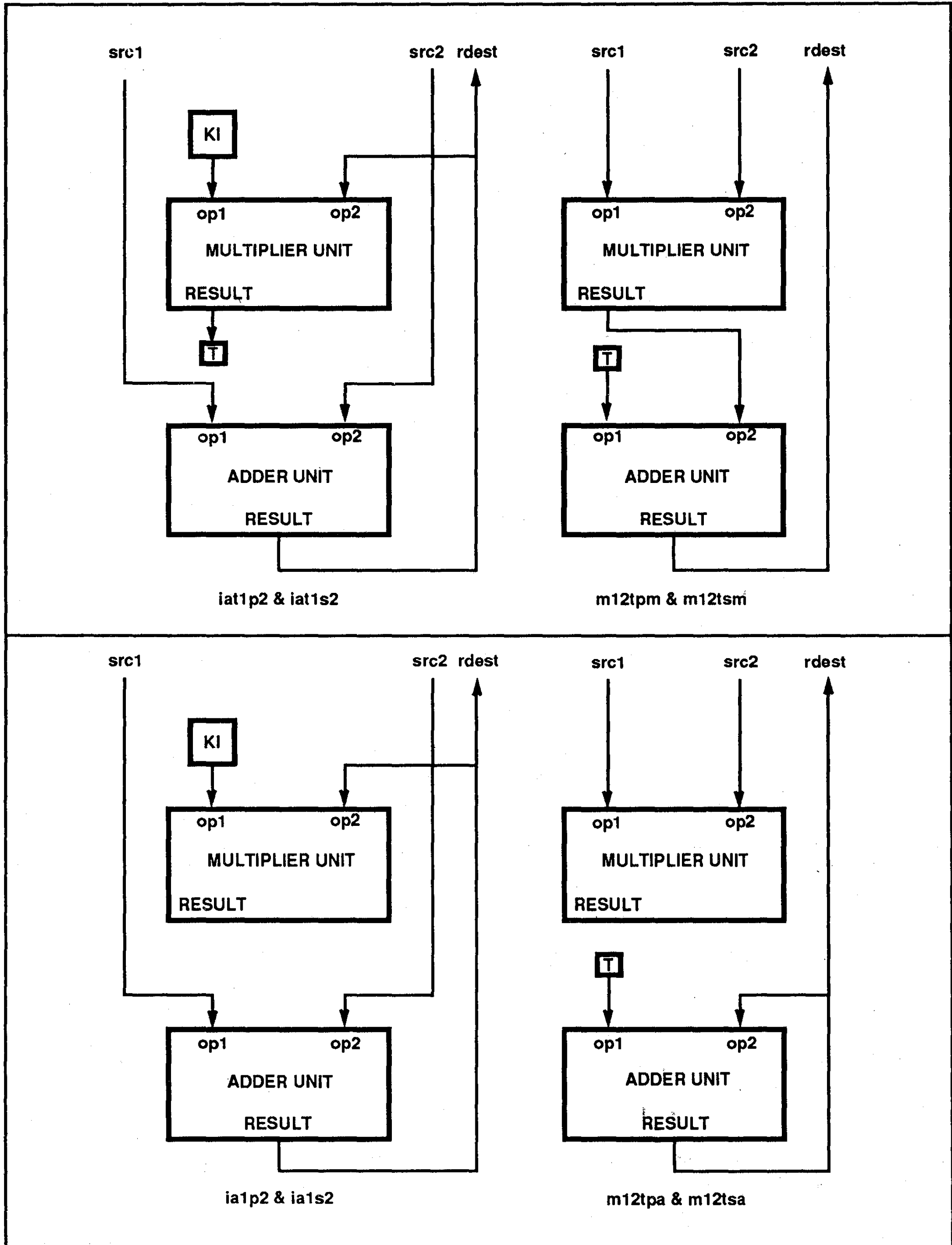


Figure 6-3. Data Paths by Instruction (4 of 8)

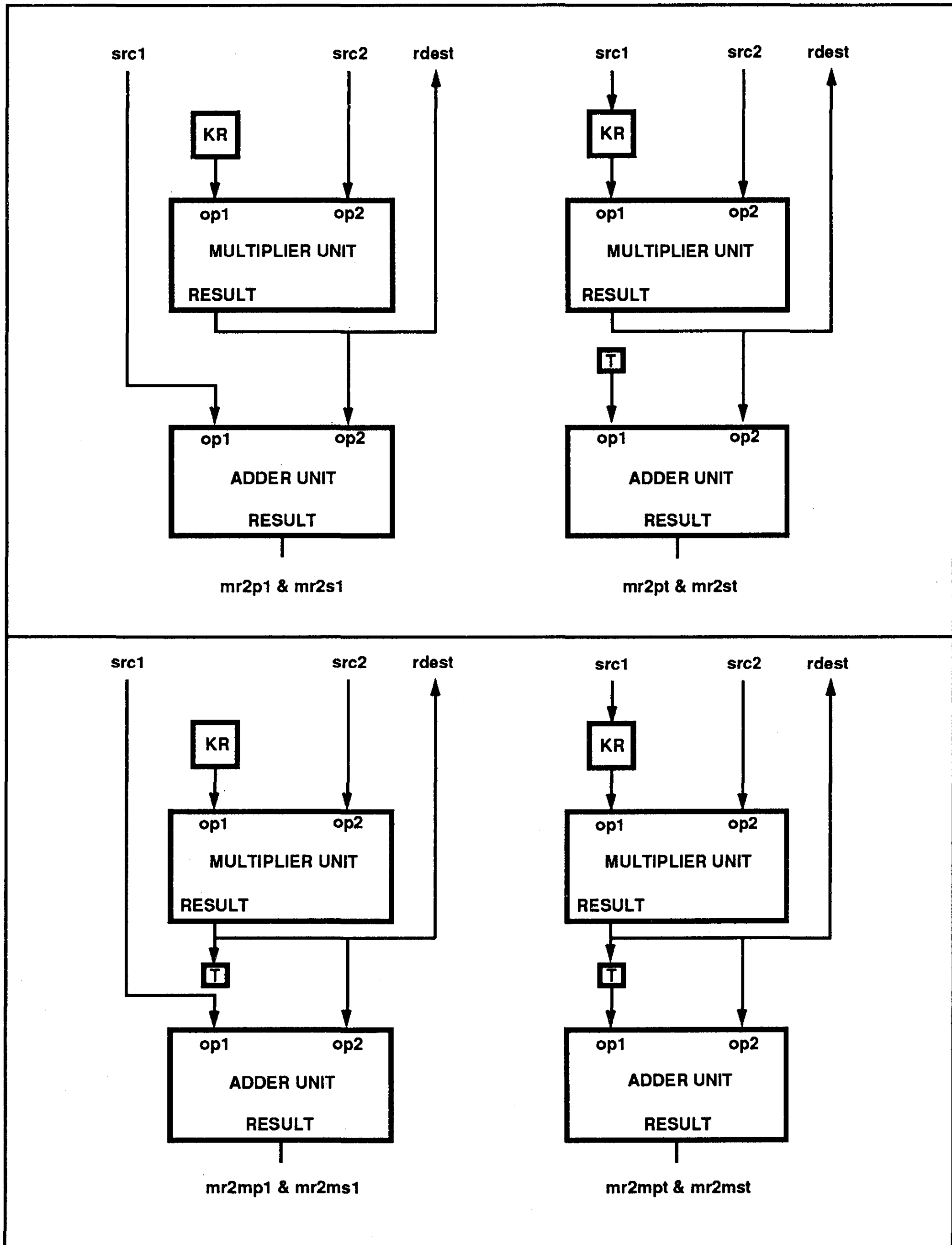


Figure 6-3. Data Paths by Instruction (5 of 8)

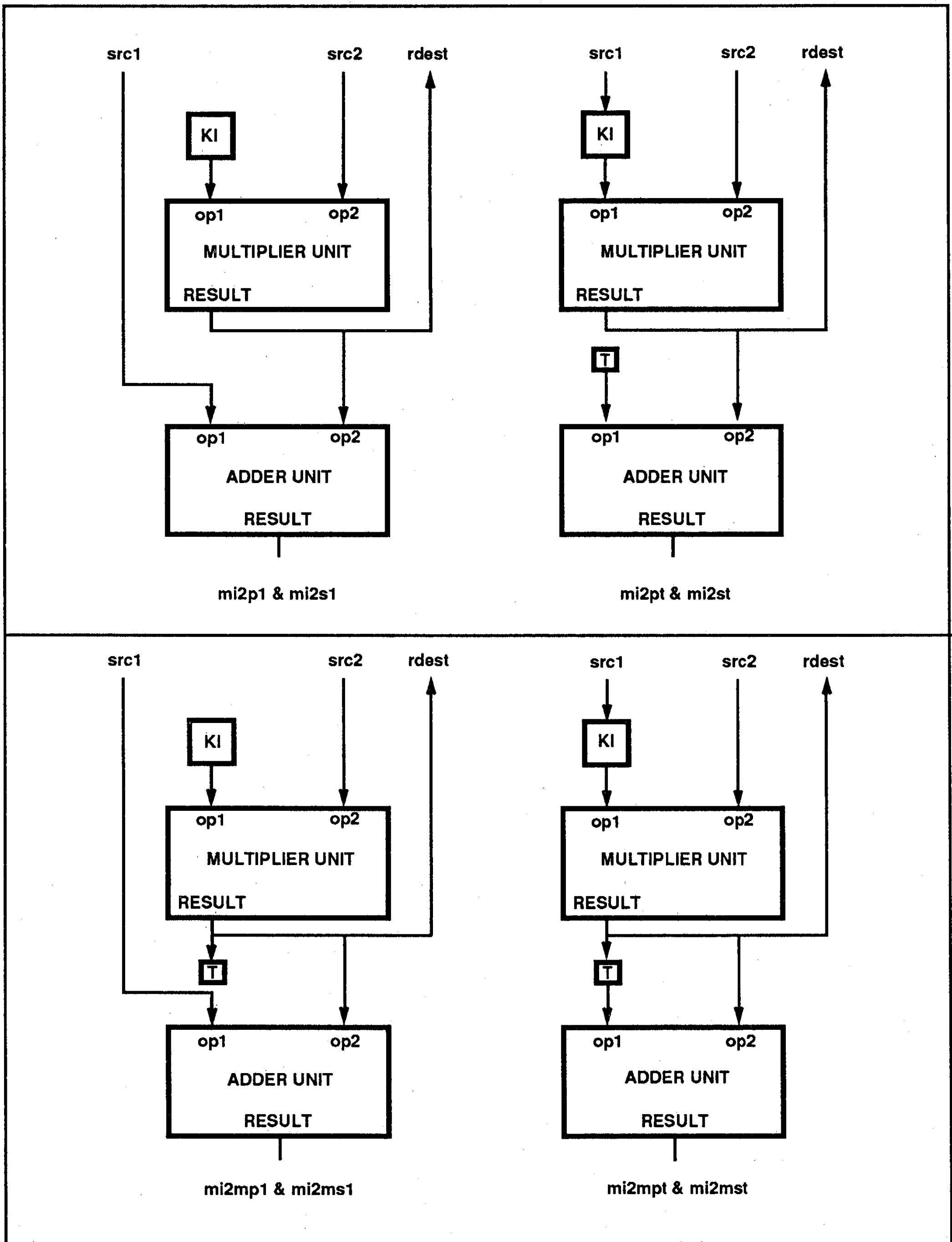


Figure 6-3. Data Paths by Instruction (6 of 8)

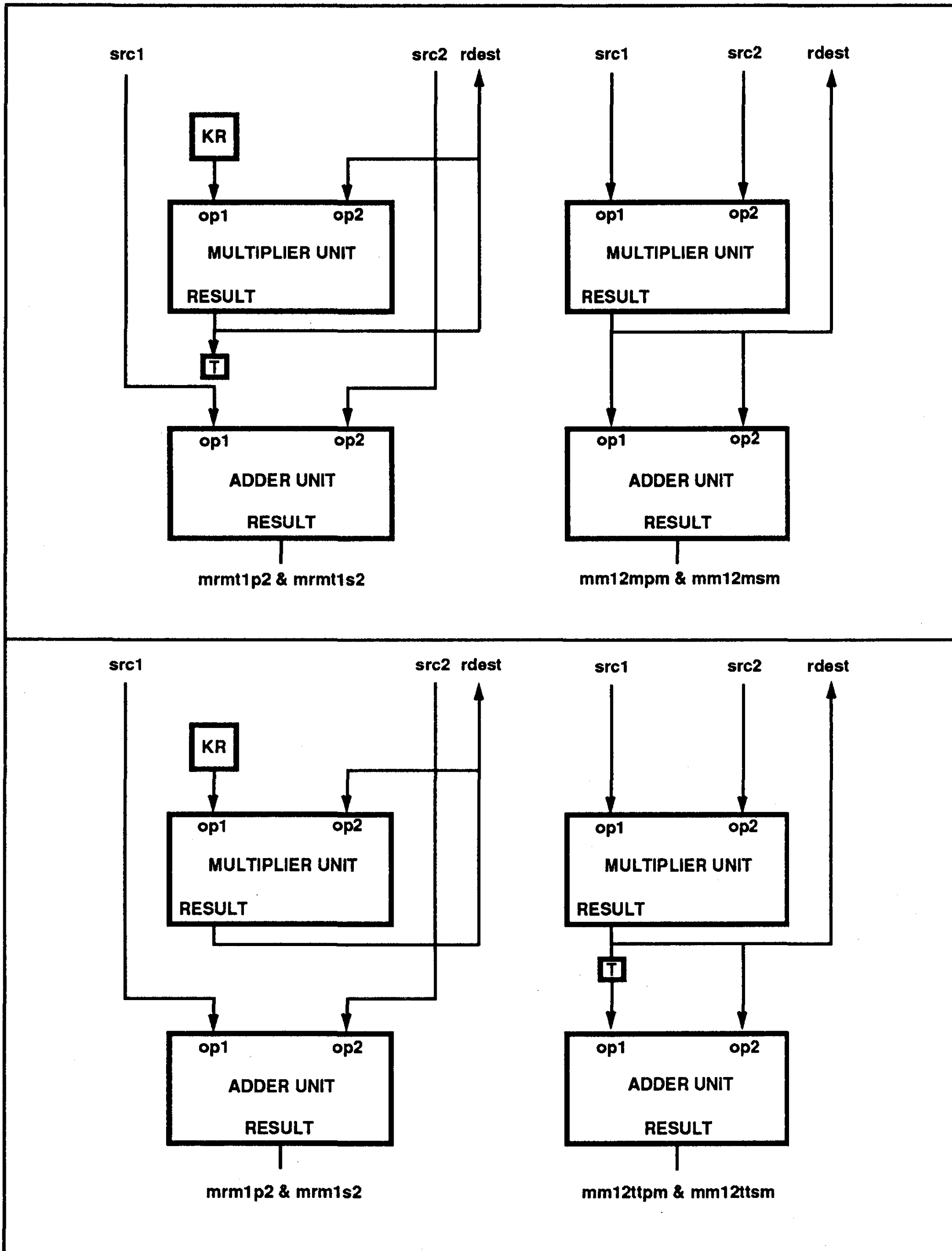


Figure 6-3. Data Paths by Instruction (7 of 8)

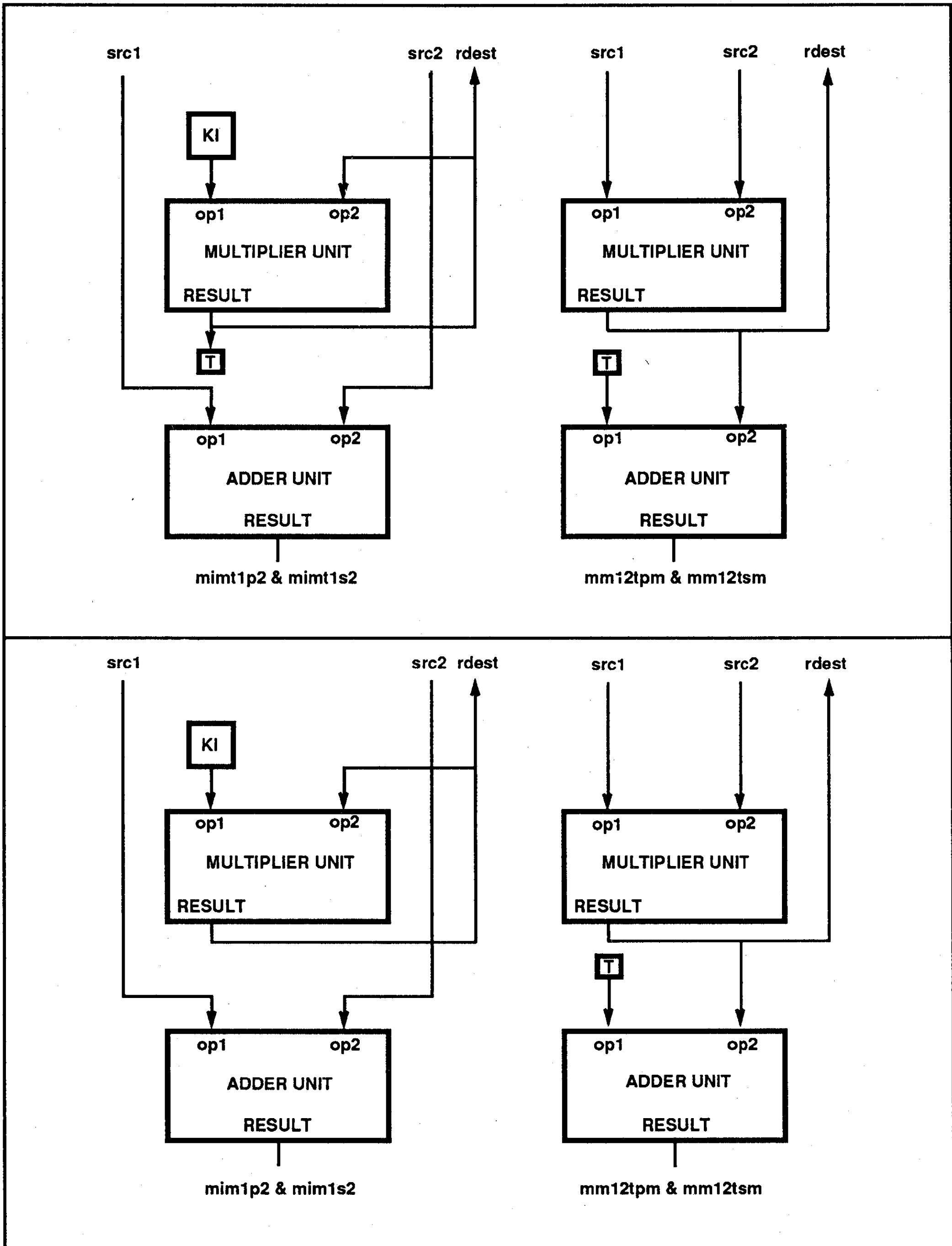


Figure 6-3. Data Paths by Instruction (8 of 8)

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 6-4 explains how the rest of these mnemonics are derived.

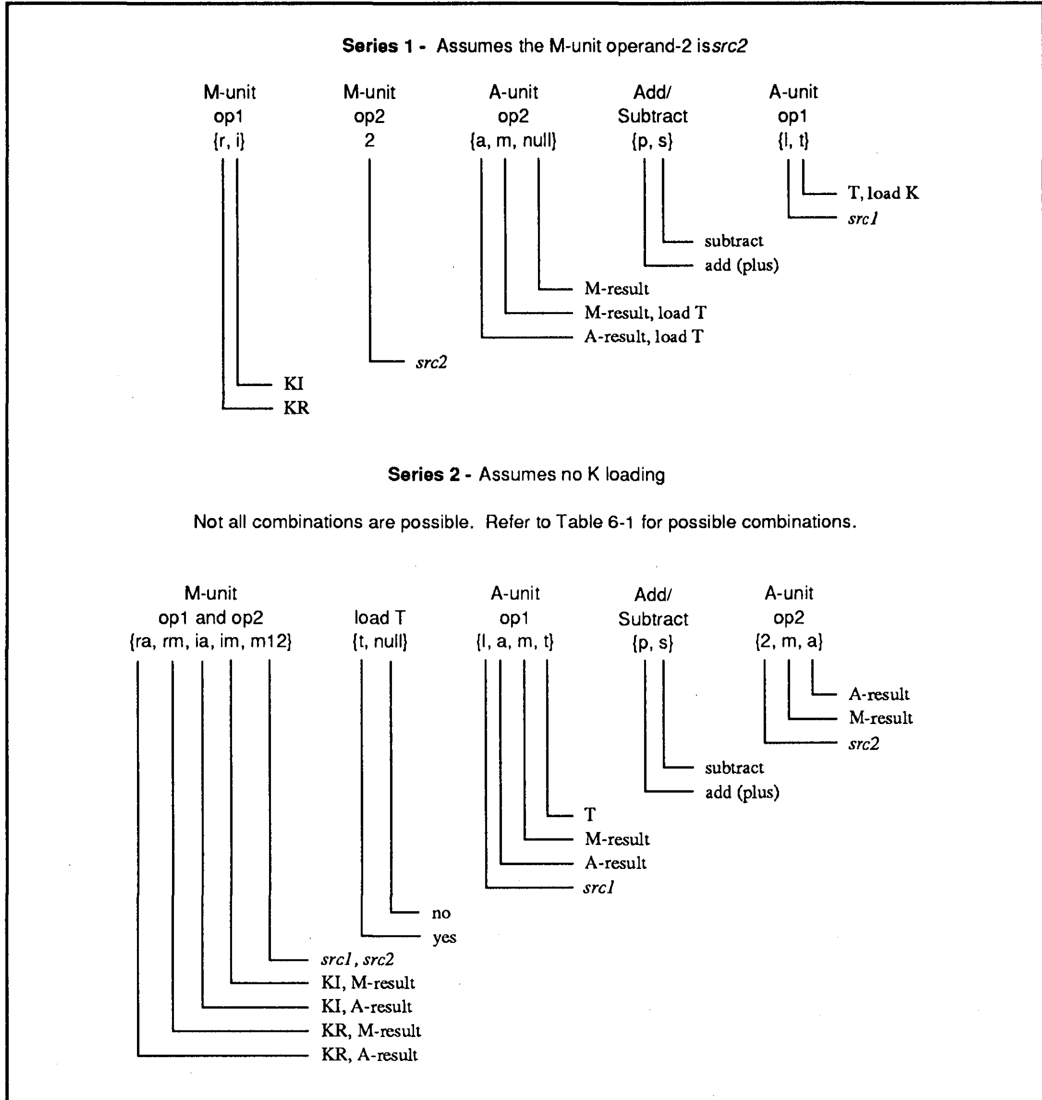


Figure 6-4. Data Path Mnemonics

Programming Notes

When the M-unit *op1* is *src1*, *src1* must not be the same as *rdest*. For best performance when the prior operation is scalar and M-unit *op1* is *src1*, *src1* should not be the same as the *rdest* of the prior operation.

6.6 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 *rdest* is not **f0**, then *rdest* must not be the same as *src1* or *src2*.

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.

6.6.1 Long-Integer Arithmetic

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

rdest ← *src1* − *src2*

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

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

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

rdest ← *src1* + *src2*

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

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

.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** and **pfmov** pseudoinstructions.

fmov.ss	*src1*, *rdest*	**(Single Move)**
Equivalent to **fiadd.ss** *src1*, **f0**, *rdest*		
pfmov.ss	*src1*, *ireg*	**(Pipelined Single Move)**
Equivalent to **pfiadd.ss** *src1*, **f0**, *rdest*		
fmov.dd	*src1*, *rdest*	**(Double Move)**
Equivalent to **fiadd** *src1*, **f0**, *rdest*		
pfmov.dd	*src1*, *ireg*	**(Pipelined Double Move)**
Equivalent to **pfiadd** *src1*, **f0**, *rdest*		

6.6.2 3-D Graphics Operations

The i860 Microprocessor supports 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 the i860 Microprocessor support graphics applications in which the set of points on the surface of a solid object is represented by polygons. The distances 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 i860 Microprocessor's graphics instructions, as illustrated in Figure 6-5.

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.

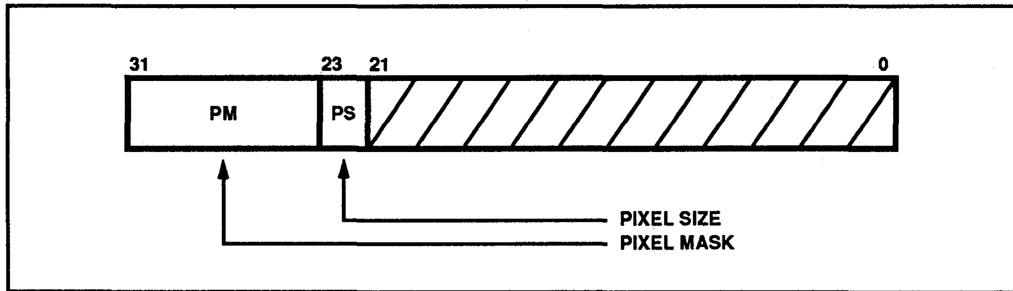


Figure 6-5. PSR Fields for Graphics Operations

6.6.2.1 Z-BUFFER CHECK INSTRUCTIONS

Consider PM as an array of eight bits PM(0)..PM(7),
where PM(0) is the least-significant bit.

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

Consider *src1*, *src2*, and *rdest* as arrays of four 16-bit fields *src1*(0)..*src1*(3), *src2*(0)..*src2*(3), and *rdest*(0)..*rdest*(3) where zero denotes the least-significant field.

```
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

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

Consider *src1*, *src2*, and *rdest* as arrays of four 16-bit fields *src1*(0)..*src1*(3), *src2*(0)..*src2*(3), and *rdest*(0)..*rdest*(3) where zero denotes the least-significant field.

```
PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
    PM [i + 4] ← src2(i) ≤ src1(i) (unsigned)
    rdest ← last-stage I-result
    last-stage I-result(i) ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

fzchkl *src1, src2, rdest* **(32-Bit Z-Buffer Check)**

Consider *src1*, *src2*, and *rdest* as arrays of two 32-bit fields *src1*(0)..*src1*(1), *src2*(0)..*src2*(1), and *rdest*(0)..*rdest*(1) where zero denotes the least-significant field.

```
PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
    PM [i + 6] ← src2(i) ≤ src1(i) (unsigned)
    rdest(i) ← smaller of src2(i) and src1(i)
OD
MERGE ← 0
```

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

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. The i860 Microprocessor supports 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**.

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.

6.6.2.2 PIXEL ADD

```
faddp            src1, src2, rdest      (Add with Pixel Merge)

    rdest ← src1 + src2
    Shift and load MERGE register from src1 + src2 as defined in Table 6-2

pfaddp           src1, src2, rdest      (Pipelined Add with Pixel Merge)

    rdest ← last-stage I-result
    last-stage I-result ← src1 + src2
    Shift and load MERGE register from src1 + src2 as defined in Table 6-2
```

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, **is t tz** performs them two at a time. By itself **faddp** implements linear interpolation; combined with **fiadd**, nonlinear interpolation can be achieved.

Table 6-2. 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 6-6 illustrates **faddp** when PS is set for 8-bit pixels. Since **faddp** adds 16-bit values in this case, each value can be treated as a fixed-point real number with an 8-bit integer portion and an

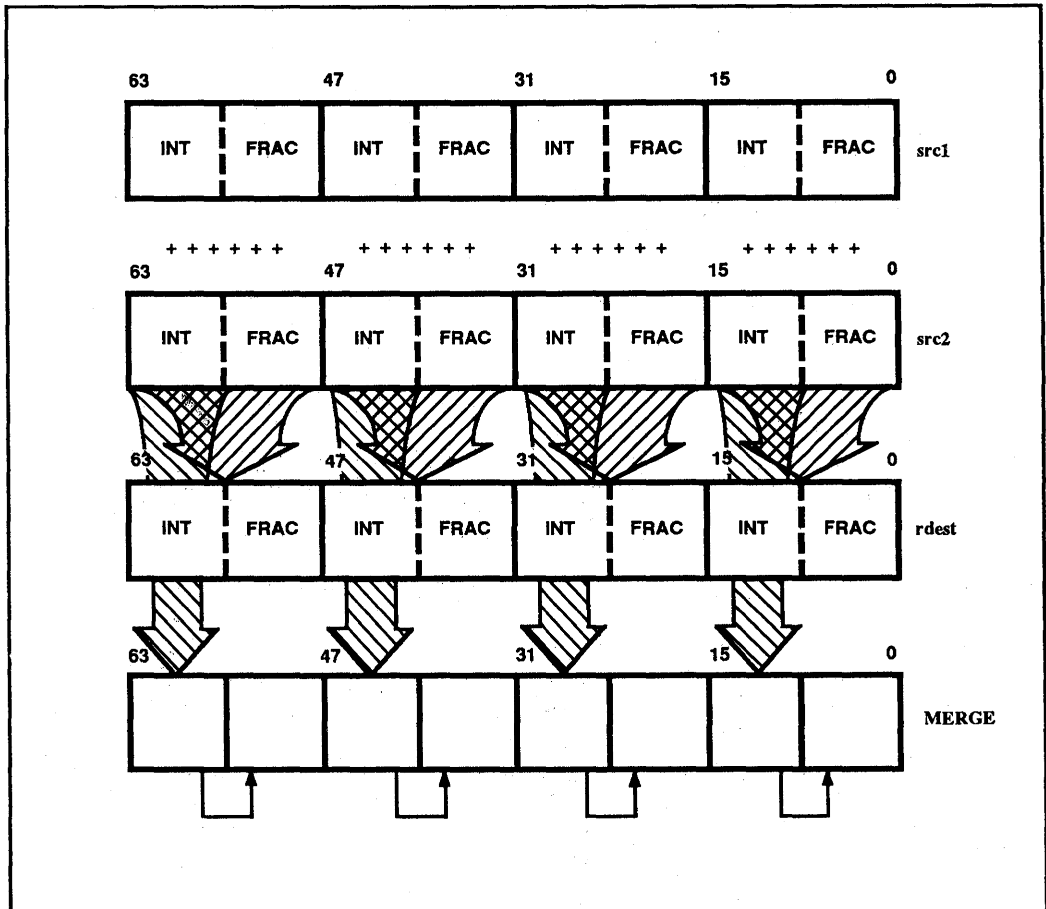


Figure 6-6. FADDP with 8-Bit Pixels

8-bit fractional portion. The real numbers are rounded to 8 bits by truncation when they are 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 6-7 illustrates **faddp** when PS is set for 16-bit pixels. Since **faddp** adds 16-bit values in this case, each value can be treated as a fixed-point real number with an 6-bit integer portion and an 10-bit fractional portion. The real numbers are rounded to 6 bits by truncation when they are 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.

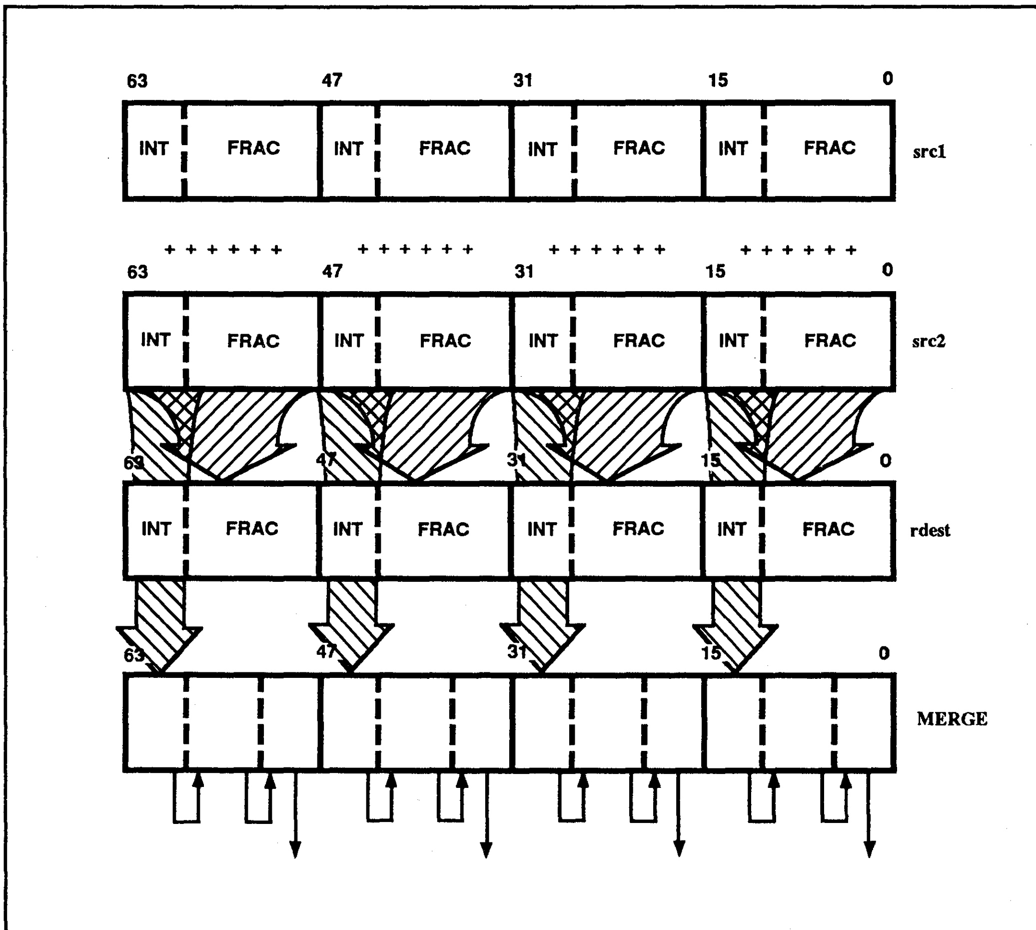


Figure 6-7. FADDP with 16-Bit Pixels

Figure 6-8 illustrates **faddp** when PS is set for 32-bit pixels. Since **faddp** adds 32-bit values in this case, each value can be treated as a fixed-point real number with an 8-bit integer portion and an 24-bit fractional portion. The real numbers are rounded to 8 bits by truncation when they are loaded into 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.

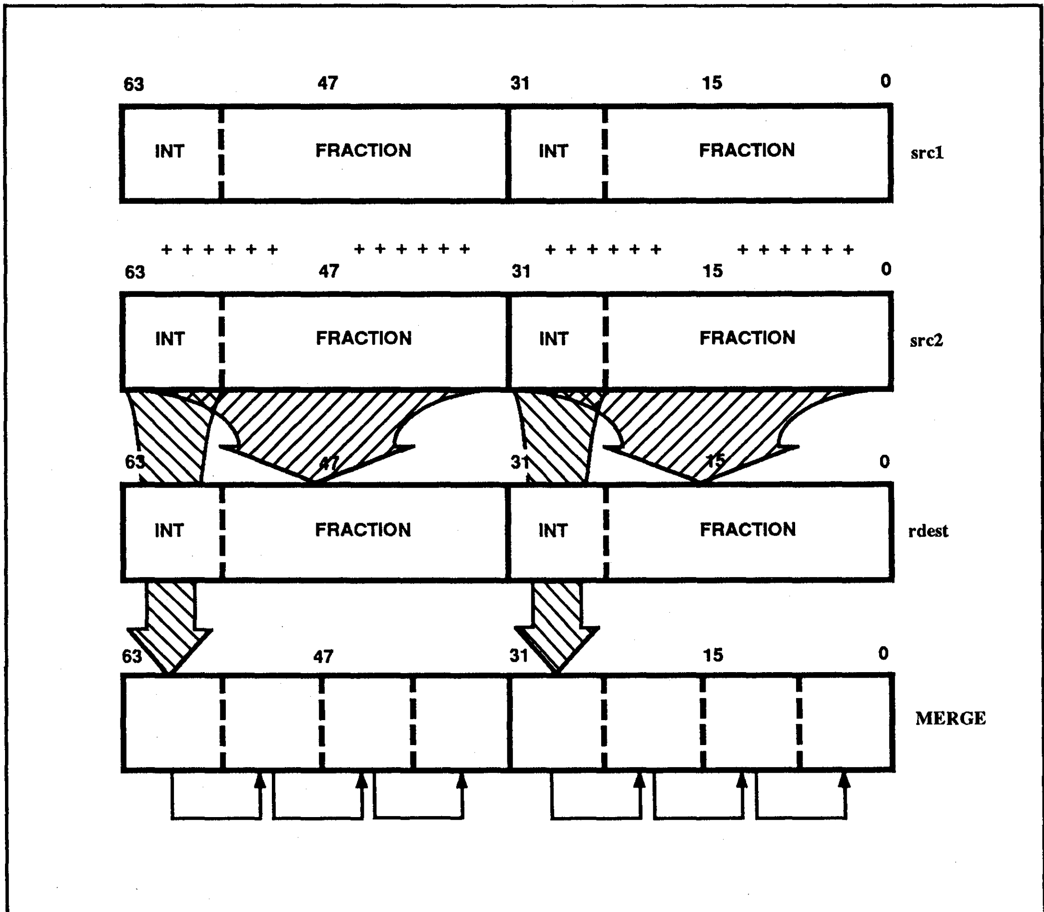


Figure 6-8. FADDP with 32-Bit Pixels

6.6.2.3 Z-BUFFER ADD

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 6-9 illustrates. Since **faddz** adds 32-bit values, each value can be treated as a fixed-point real number with an 16-bit integer portion and a 16-bit fractional portion. The real numbers are rounded to 16

bits by truncation when they are 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.

faddz *src1, src2, rdest* **(Add with Z Merge)**

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

pfaddz *src1, src2, rdest* **(Pipelined Add with Z Merge)**

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

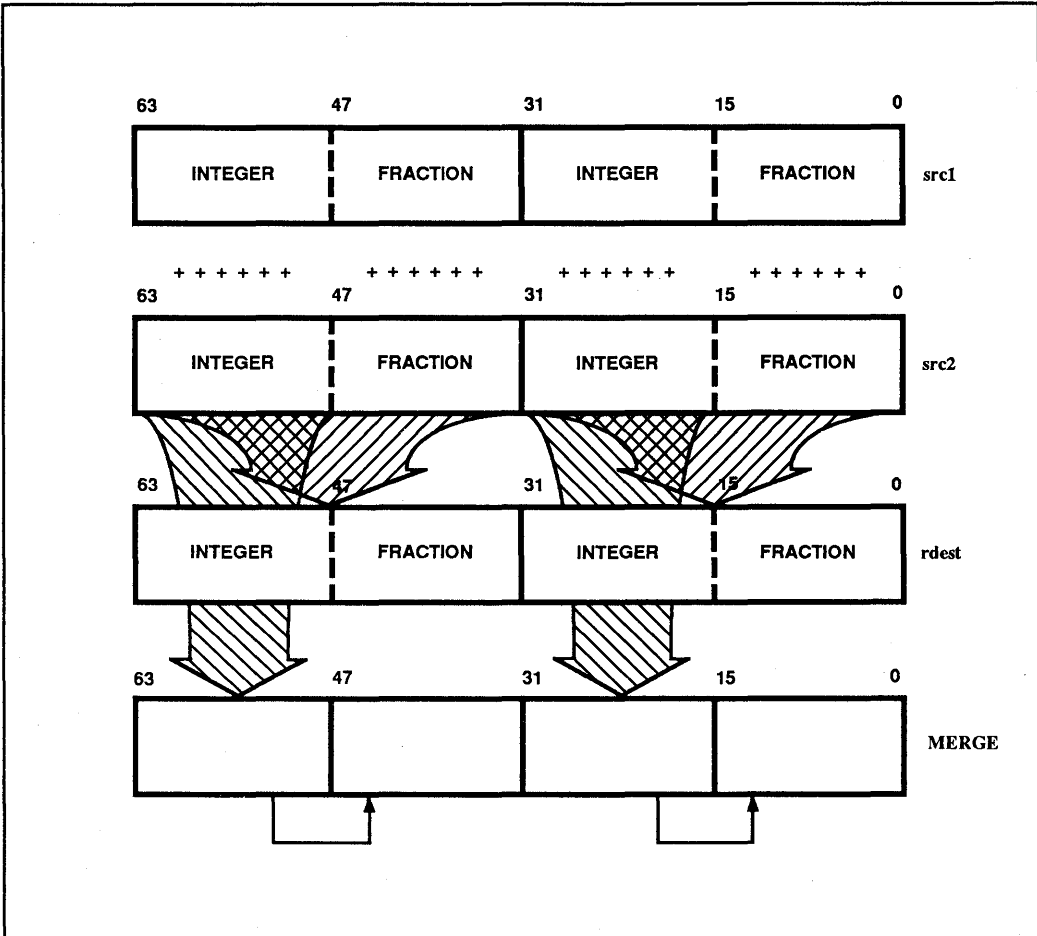


Figure 6-9. FADDZ with 16-Bit Z-Buffer

32-bit Z-buffers can use 32-bit or 64-bit distance interpolation. For 32-bit interpolation, no special instructions are required. Two 32-bit adds can be performed as an 64-bit add instruction. The fact that data is carried from the low-order 32-bits into the high-order 32-bits may introduce an insignificant distortion into the interpolation.

For 32-bit Z-buffers, 64-bit distance interpolation is implemented (as Figure 6-10 shows) with two 64-bit **fiadd** instructions. The merging is implemented with the 32-bit move **fmov.ss** *src1*, *rdest*.

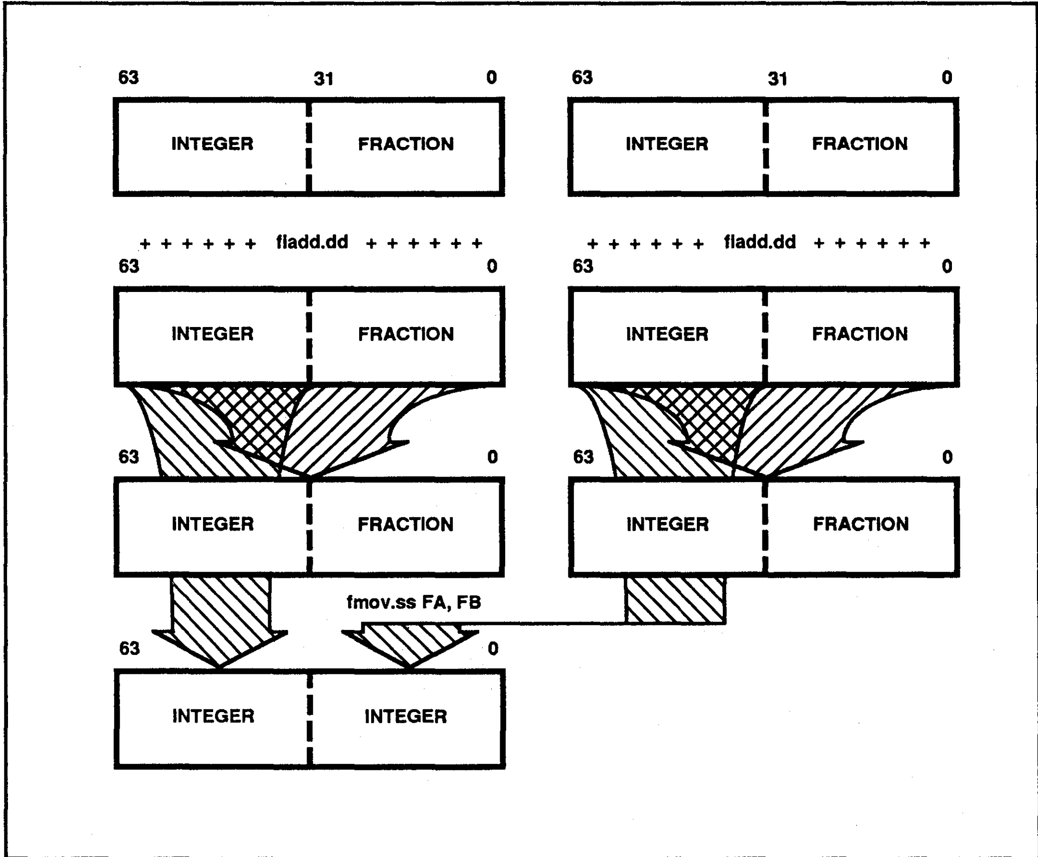


Figure 6-10. 64-Bit Distance Interpolation

6.6.2.4 OR WITH MERGE REGISTER

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. *Src2* should contain zero.

For distance interpolation or for intensity interpolation that does not require further modification of the value in the MERGE register, the *src1* operand of **form** may be **f0**, thereby causing the instruction to simply load the MERGE register into a floating point register.

form	*src1, rdest*	**(OR with MERGE Register)**
rdest ← *src1* OR MERGE MERGE ← 0		
pform	*src1, rdest*	**(Pipelined OR with MERGE Register)**
rdest ← last-stage I-result last-stage I-result ← *src1* OR MERGE MERGE ← 0		

6.7 TRANSFER F-P TO INTEGER REGISTER

fxfr	*src1, ireg*	**(Transfer F-P to Integer Register)**
ireg ← *src1*		

The 32-bit floating-point register selected by *src1* is stored into the (32-bit) integer register selected by *ireg*. Assemblers and compilers should set *src2* to zero.

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 *rdest* is nonzero (refer to section 6.2).

For best performance, *ireg* should not be referenced in the next instruction, and *src1* should not reference the result of the prior instruction if the prior instruction is scalar.

6.8 DUAL-INSTRUCTION MODE

The i860 Microprocessor 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 instructions 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 i860 Microprocessor is executing in single-instruction mode and encounters a floating-point instruction with the D-bit set,

one more 32-bit instruction is executed before dual-mode execution begins. If the i860 Microprocessor is executing in dual-instruction mode and a floating-point instruction is encountered with a clear D-bit, then one more pair of instructions is executed before resuming single-instruction mode. Figure 6-11 illustrates two variations of this sequence of events: one for extended sequences of dual-instructions and one for a single instruction pair.

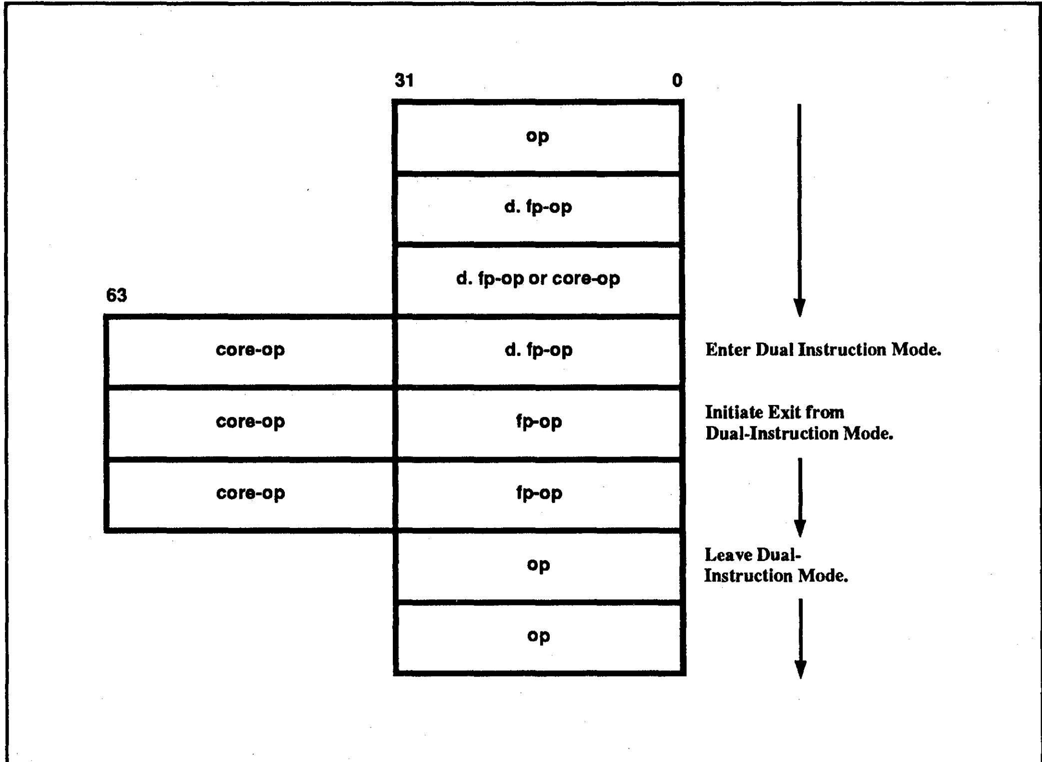


Figure 6-11. Dual-Instruction Mode Transitions (1 of 2)

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**. 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.

6.8.1 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.

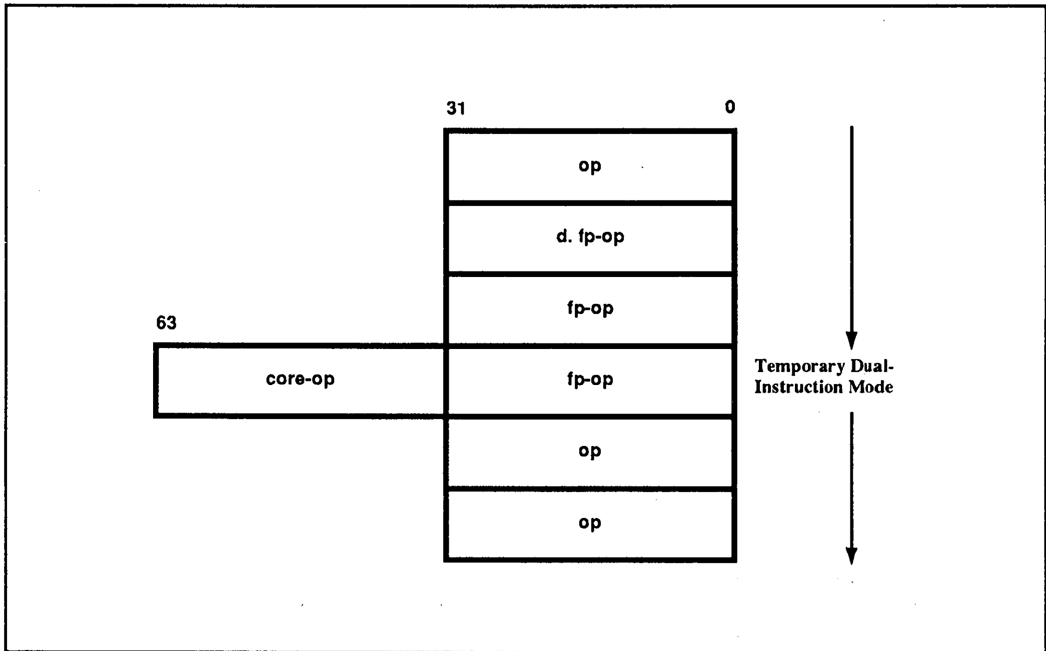


Figure 6-11. Dual-Instruction Mode Transitions (2 of 2)

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. An **fxfr** instruction that transfers to a register referenced by the companion core instruction will update the register after the core instruction accesses the register. The destination of the core instruction will not be updated if it is any if the integer register. Likewise, if the core instruction uses autoincrement indexing, the index register will not be updated.

5. When the core instruction sets CC and the floating-point instruction is **pfgt** or **pfeq**, CC is set according to the result of the **pfgt** or **pfeq**.

6.8.2 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**, **ld.c**, **st**, **st.c**, **call**, **ixfr**, or any instruction that updates an integer register (including autoincrement indexing).

7. In dual-instruction mode when the core instruction is an indirect branch, the **psr** trap bits cannot be 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**.

Traps and Interrupts 7

Chapter 7
Traps and Interrupts

Traps are caused by exceptional conditions detected in programs or by external interrupts. Traps cause interruption of normal program flow to execute a special program known as a trap handler.

7.1 TYPES OF TRAPS

Traps are divided into the types shown in Table 7-1

Table 7-1. Types of Traps

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

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

7.2 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 7.2.2. The processor takes the following steps while transferring control to the trap handler:

1. Copies U (user mode) of the **psr** into PU (previous U).

2. Copies IM (interrupt mode) into PIM (previous IM).

3. Sets U to zero (supervisor mode).

4. Sets IM to zero (interrupts disabled). 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. If the processor is in single-instruction mode and the next instruction will be executed in dual-instruction mode or if the processor is in dual-instruction mode and the next instruction will be executed in single-instruction mode, DS is set; otherwise, it is cleared.

7. The appropriate trap type bits in **psr** and **epsr** are set (IT, IN, IAT, DAT, FT, IL). Several bits may be set if the corresponding trap conditions occur simultaneously.

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

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

7.2.1 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 8). 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 MERGE register.
6. The KR, KI, and T registers.
7. Any of the four pipelines (refer to section 7.9).
8. The floating-point and integer register files.
9. The **dirbase** register.

7.2.2 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 7.2.2.1.

4. Updating **psr** with the value to be used after return. It may be necessary to set the KNF bit in **psr**. The requirements for KNF are given in section 7.2.2.2.

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

6. Executing an indirect branch to the resumption address. 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.)

7.2.2.1 DETERMINING WHERE TO RESUME

To determine where to resume execution upon leaving the trap handler, examine the instruction at address **fir** − 4. If this instruction is not a delayed control instruction, then execution resumes at the address in **fir**.

If, on the other hand, the instruction at **fir** − 4 is a delayed control instruction (i.e. one that executes the next sequential instruction on branch taken), the normal action is to resume at **fir** − 4 so that the control 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 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 i860 Microprocessor was in dual-instruction mode and execution is to resume at **fir** − 4, DS should be set and DIM cleared in the **psr** used to resume execution. Clearing DIM prevents the floating-point instruction associated with the control instruction from being reexecuted. Setting DS forces the processor back to dual-instruction mode after executing the control 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. Also, that **nop** should not be the target of any branch or call.

7.2.2.2 SETTING KNF

The KNF bit of **psr** should be set 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 the following conditions:

- The trap was caused in dual-instruction mode by a data-access fault and there are no other trap conditions. In this case, the the floating-point instruction has already been executed. (The one exception is the **fxfr** instruction. An **fxfr** must be reexecuted; so do not set KNF).

- 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 7.2.2.1). The trap handler determines the result that corresponds to the exceptional inputs; therefore, the instruction should not be reexecuted.

7.3 INSTRUCTION FAULT

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

- By the **trap** instruction. Refer to the **trap** instruction in Chapter 5.

- 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 5.

- By the lack of an **unlock** instruction and a subsequent load or store within 32 instructions of a **lock**. In this case IL is also set. When the trap handler finds IL set, it should scan backwards for the **lock** instruction and restart at that point. The absence of a **lock** instruction within 32 instructions of the trap indicates a programming error. Refer to the **lock** instruction in Chapter 5.

7.4 FLOATING-POINT FAULT

The floating-point faults of the i860 Microprocessor support the floating-point exceptions defined by the IEEE standard as well as some other useful classes of exceptions. The i860 Microprocessor divides these into two classes:

1. **Source exceptions.** This class includes:

    - All the invalid operations defined by the IEEE standard (including operations on trapping 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.

The floating-point fault occurs only on floating-point instructions, **pst, fst, fld, pfld,** and **ixfr.** However, no fault occurs when **pst, fst, fld, pfld,** or **ixfr** transfers an invalid floating-point format.

Software supplied by Intel provides the IEEE standard default handling for all these exceptions.

7.4.1 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 *rdest*, 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 case, 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.

7.4.2 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 pipelined operations are being used:

- **Scalar (nonpipelined) operations.** Result exceptions are reported on the next floating-point, **fst.x**, or **pst.x** (and sometimes **fld**, **pfld**, **ixfr**) instruction after the scalar operation. When a trap occurs, the last-stage of the affected unit contains the result of the scalar operation.

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

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

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

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

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

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

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

7.5 INSTRUCTION-ACCESS FAULT

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

7.6 DATA-ACCESS FAULT

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

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

7.7 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.

7.8 RESET TRAP

When the i860 Microprocessor is reset, execution begins in single-instruction mode at address 0xFFFFFF00. This is the same address as for other traps. The reset trap can be distinguished from other traps by the fact that no trap bits are set. The instruction cache is flushed. The bits DPS, BL, and ATE in **dirbase** are cleared. CS8 is initialized by the value at the INT pin just before the end of RESET. The read-only fields of the **epsr** are set to identify the processor, while the IL, WP, and PBM bits are cleared. The bits U, IM, BR, and BW in **psr** are cleared. All other bits of **psr** and all other register contents are **undefined**.

The software must ensure that the data cache is flushed (refer to Chapter 4) 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 a **ld.c fir, r0** instruction.

Reset code must initialize the floating-point pipeline states to zero, using dummy **pfadd**, **pfmul**, **pfiadd** instructions. Floating-point traps must disabled to ensure that no spurious floating-point traps are generated.

After a RESET the i860 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.

7.9 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.

7.9.1 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.

7.9.2 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**.

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.

7.9.3 Graphics Pipeline

The graphics pipeline has only one stage. To flush the pipeline, execute a **pfiadd f0, f0,** *rdest*. 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 (see Example 7-2).

7.9.4 Examples of Pipeline Preemption

Example 7-1 shows how to save the pipeline state.

Example 7-2 shows how to restore the pipeline state. Trap handlers manipulate the result-status bits in the floating-point pipelines while preparing for pipeline resumption. When storing to **fsr** with the U-bit set, the result-status bits are loaded into the first stage of the pipelines of the floating-point adder and multiplier. The updated result-status bits of a particular unit (multiplier or adder) are propagated one stage for each pipelined floating-point operation for that unit. When they reach the last stage, they override the normal result-status bits computed from the last-stage result. The result-status bits in the **fsr** always reflect the last-stage result status and cannot be directly set by software.

```
// The symbols Mres3, Ares3, Mres2, Ares2, Mres1, Ares1,
//  Ires1, Lres, KR, KI, and T refer to 64-bit FP registers.
// The symbols Fsr3, Fsr2, Fsr1, Mergelo32, Mergehi32, and Temp
//  refer to integer registers.
// The symbols Lres3m, Lres2m, and Lres1m refer to memory locations.
// The symbol Dummy represents an addressing mode that refers to some
//   readable location that is always present (e.g. 0(r0)).

// Save third, second, and first stage results
  fld.d       DoubOne,          f4  // get double-precision 1.0
  ld.c        fsr,     Fsr3         // save third stage result status
  andnot      0x20,    Fsr3, Temp   // clear FTE bit
  st.c        Temp,    fsr          // disable FP traps
  pfmul.ss    f0,      f0,   Mres3  // save third stage M result
  pfadd.ss    f0,      f0,   Ares3  // save third stage A result
  pfld.d      Dummy,   Lres         // save third stage pfld result
  fst.d       Lres,    Lres3m       //  ... in memory
  ld.c        fsr,     Fsr2         // save second stage result status
  pfmul.ss    f0,      f0,   Mres2  // save second stage M result
  pfadd.ss    f0,      f0,   Ares2  // save second stage A result
  pfld.d      Dummy,   Lres         // save second stage pfld result
  fst.d       Lres,    Lres2m       //  ... in memory
  ld.c        fsr,     Fsr1         // save first stage result status
  pfmul.ss    f0,      f0,   Mres1  // save first stage M result
  pfadd.ss    f0,      f0,   Ares1  // save first stage A result
  pfld.d      Dummy,   Lres         // save first stage pfld result
  fst.d       Lres,   Lres1m        //  ... in memory
  pfiadd.dd   f0,      f0,   Ires1  // save vector-integer result
// Save KR, KI, T, and MERGE
  r2apt.dd    f0,      f4,      f0  // M first stage contains KR
                                    // A first stage contains T
  i2p1.dd     f0,      f4,      f0  // M first stage contains KI
  pfmul.dd    f0,      f0,      KR  // Save KR register
  pfmul.dd    f0,      f0,      KI  // Save KI register
  pfadd.dd    f0,      f0,      f0  // adder third stage gets T
  pfadd.dd    f0,      f0,      T   // save T-register
  form        f0,               f2  // save MERGE register
  fxfr        f2,      Mergelo32
  fxfr        f3,      Mergehi32
```

Example 7-1. Saving Pipeline States

```
// The symbols Mres3, Ares3, Mres2, Ares2, Mres1, Ares1,
//  Ires1, KR, KI, and T refer to 64-bit FP registers.
// The symbols Fsr3, Fsr2, Fsr1, Mergelo32, Mergehi32, and Temp
//  refer to integer registers.
// The symbols Lres3m, Lres2m, and Lres1m refer to memory locations.

    st.c        r0,      fsr            // clear FTE
// Restore MERGE
    shl         16, Mergelo32,  r1      // move low 16 bits to high 16
    ixfr        r1,      f2
    shl         16, Mergehi32,  r1      // move low 16 bits to high 16
    ixfr        r1,      f3
    ixfr        Mergelo32,      f4
    ixfr        Mergehi32,      f5
    faddz       f0,      f2,    f0      // merge low 16s
    faddz       f0,      f4,    f0      // merge high 16s
// Restore KR, KI, and T
    fld.l       SingOne,        f2      // get single-precision 1.0
    fld.d       DoubOne,        f4      // get double-precision 1.0
    pfmul.dd    f4,      T,     f0      // put value of T in M 1st stage
    r2pt.dd     KR,      f0,    f0      // load KR, advance t
    i2apt.dd    KI,      f0,    f0      // load KI and T
// Restore 3rd stage
    andh        0x2000, Fsr3,   r0      // test adder result precision ARP
    bc.t        L0                      // taken if it was single
    pfadd.ss    Ares3,   f0,    f0      // insert single result
    pfadd.dd    Ares3,   f0,    f0      // insert double result
L0: orh         ha%Lres3m, r0,  r31
    andh        0x400,   Fsr3,  r0      // test load result precision LRP
    bc.t        L1                      // taken if it was single
    pfld.l      l%Lres3m(r31),  f0      // insert single result
    pfld.d      l%Lres3m(r31),  f0      // insert double result
L1: andh        0x1000, Fsr3,   r0      // test multiplier result precision MRP
    bc.t        L2                      // taken if it was single
    pfmul.ss    Mres3,   f2,    f0      // insert single result
    pfmul3.dd   Mres3,   f4,    f0      // insert double result
L2: or          0x10,    Fsr3,  Temp    // set U (update) bit so that st.c
                                        //   will update status bits in pipeline
    andnot      0x20,    Temp,  Temp    // clear FTE bit so as not to cause traps
    st.c        Temp ,   fsr            // update stage 3 result status
```

Example 7-2. Restoring Pipeline States (1 of 2)

```
// Restore 2nd stage
    andh        0x2000, Fsr2,    r0  // test adder result precision ARP
    bc.t        L3                   // taken if it was single
    pfadd.ss    Ares2,  f0,      f0  // insert single result
    pfadd.dd    Ares2,  f0,      f0  // insert double result
L3: orh         ha%Lres2m, r0,   r31
    andh        0x400,  Fsr2,    r0  // test load result precision LRP
    bc.t        L4                   // taken if it was single
    pfld.l      l%Lres2m(r31),   f0  // insert single result
    pfld.d      l%Lres2m(r31),   f0  // insert double result
L4: or          0x10,   Fsr2,  Temp  // set update bit
    andnot      0x20,   Temp,  Temp  // clear FTE
    andh        0x1000, Fsr2,    r0  // test multiplier result precision MRP
    bc.t        L5                   // taken if it was single
    pfmul.ss    Mres2,  f2,      f0  // insert single result
    pfmul3.dd   Mres2,  f4,      f0  // insert double result
L5: st.c        Temp,   fsr          // update stage 2 result status
// Restore 1st stage
    andh        0x1000, Fsr1,    r0  // test multiplier result precision MRP
    bc.t        L6                   // skip next if double
    pfmul.ss    Mres1,  f2,      f0  // insert single result
    pfmul3.dd   Mres1,  f4,      f0  // insert double result
L6: andh        0x2000, Fsr1,    r0  // test adder result precision ARP
    bc.t        L7                   // taken if it was single
    pfadd.ss    Ares1,  f0,      f0  // insert single result
    pfadd.dd    Ares1,  f0,      f0  // insert double result
L7: orh         ha%Lres1m, r0,   r31
    andh        0x400,  Fsr1,    r0  // test load result precision LRP
    bc.t        L8                   // taken if it was single
    pfld.l      l%Lres1m(r31),   f0  // insert single result
    pfld.d      l%Lres1m(r31),   f0  // insert double result
L8: andh        0x800,  Fsr1,    r0  // test vector-integer result precision IRP
    bc.t        L9                   // taken if it was single
    pfiadd.ss   f0, Ires1,  f0       // insert single result
    pfiadd.dd   f0, Ires1,  f0       // insert double result
L9: or          0x10,   Fsr1,  Fsr1  // set U (update) bit
    st.c        Fsr1,   fsr          // update stage 1 result status
    st.c        Fsr3,   fsr          // restore nonpipelined FSR status
```

Example 7-2. Restoring Pipeline States (2 of 2)

Programming Model 8

Chapter 8
Programming Model

This chapter defines standards for the use of certain aspects of the architecture of the i860 Microprocessor. These standards must be followed to guarantee that compilers, applications programs, and operating systems written by different people and organizations will work together.

8.1 REGISTER ASSIGNMENT

Table 8-1 defines the standard for register allocation. Figure 8-1 presents the same information graphically.

Table 8-1. Register Allocation

Register	Purpose	Left Unchanged by a Subroutine?
r0	Always zero	Yes
r1	Return address	Yes
r2	Stack pointer	Note[1]
r3	Frame pointer	Yes
r4-r15	Local values	Yes
r16-r27	Parameters and temporaries	No
r16	Return value	No
r28-r30	Temporaries	No
r31	Addressing temporary	No
f0-f1	Always zero	
f2-f15	Local values	Yes
f16-f27	Parameters and temporaries	No
f16-f17	Return value	No
f28-f31	Temporaries	No

[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**.

NOTE

The dividing point between locals and parameters and return value in the floating-point registers is not yet firm. For the purpose of illustration, the dividing point is shown at f16, but this may change to f8.

8.1.1 Integer Registers

Up to 12 parameters can be passed in the integer registers. The first (leftmost) parameter is passed in **r16** (if it is an integer), the rest in successively higher-numbered registers. If fewer parameters are required, the remaining registers can be used for temporary variables. If more than 12 parameters are required, the overflow can be passed in memory on the stack.

Register **r16** is both a parameter register and a return value. If a subroutine has an integer return value, the value is put into **r16** before control is returned to the caller.

Register **r1** is the required return-address register, because the **call** instruction uses it to save the return address. Subroutines are therefore required to use **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 to allow construction of 32-bit absolute-address temporaries. The assembler uses **r31** by default to construct 32-bit absolute addresses from 16-bit literals.

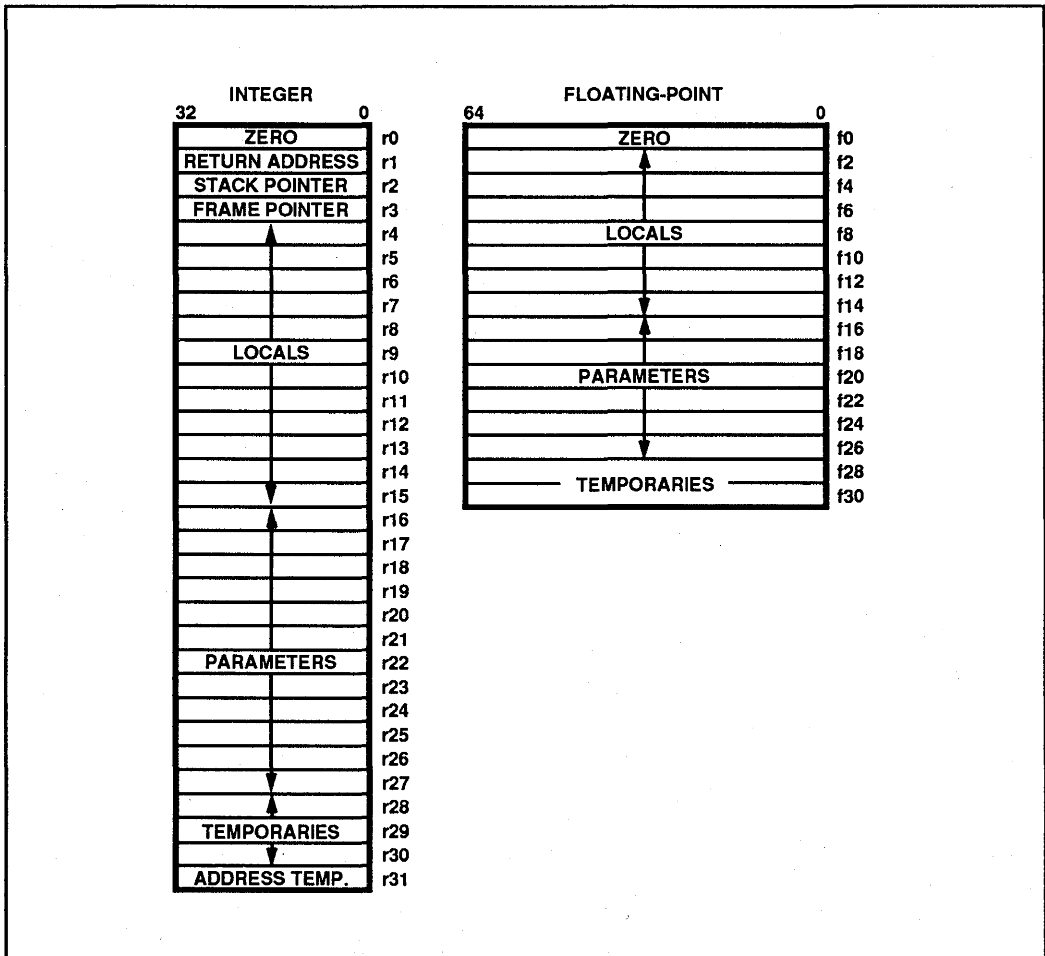


Figure 8-1. Register Allocation

8.1.2 Floating-Point Registers

Floating-point and 64-bit integer values in the floating-point registers must use **f16-f27** when passed by value. The leftmost parameter is passed in **f17-f16** (if it is floating-point); the rest in successively higher-numbered registers. Single-precision parameters use two registers, just as do double-precision parameters. The single-precision value must be in the even-numbered register; the corresponding odd-numbered register is left unused in this case. A single-precision floating-point value can be converted to double-precision with the **fmov.sd** *fx, fy* pseudoinstruction.

Parameters beyond **f26-f27** are passed in memory on the stack. The last (i.e. rightmost) parameter is at the highest stack address (i.e is pushed first assuming a grow-down stack). The same registers used to pass the first parameter are used for the return value when the return value is a floating-point value or 64-bit integer. A subroutine may need to save the first parameter to make room for the return value.

8.1.3 Passing Mixed Integer and Floating-Point Parameters in Registers

If parameter N is an integer parameter, then it is placed in integer register 16 + N, and the double-precision register at 16 + 2N is available for use as a local variable. If parameter M is a floating-point parameter, then it is placed in the floating-point register pair at 16 + 2M, and the integer register 16 + M is available for use as a local variable.

NOTE

This convention remains tentative. It may change to allow all integer and floating parameter registers to contain parameter values.

8.1.4 Variable Length Parameter Lists

Parameter passing in registers can handle variable parameters. UNIX* System V uses a special method to access variable-count parameters. The **varargs.h** file defines 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 subroutine with variable parameters calls **va_start** to force them onto the stack before they can be used. The routine **va_start** must be called at the beginning of a subroutine. This method works with current C standards.

8.2 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 would impose a performance penalty on the application. If a compiler must reference data that is misaligned, the compiler must 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 requires four byte stores and three shifts. The code example in Example 8-1 is the recommended method for reading a misaligned 32-bit value whose address is in **r8**.

```
    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 value 32-bit
    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 8-1. Reading Misaligned 32-Bit Value

8.3 IMPLEMENTING A STACK

In general, compilers and 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™. If a subroutine call requires placing parameters on the stack, then the caller is 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 directly for the call operation.

A separate frame pointer is used because C allows calls to subroutines that change the stack pointer to allocate space on the stack at run-time (e.g. **alloca** and **va_start**). Other 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 **r2**-relative addressing to get at values on the stack. If the compiler knows that it does not call subroutines that leave **r2** 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) need not point to a 16-byte boundary, as long as the compiler keeps data correctly aligned when assigning positions relative to **r3**.

Figure 8-2 shows the stack-frame format. A fixed format is necessary to allow some minimal stack-frame analysis by a low-level debugger.

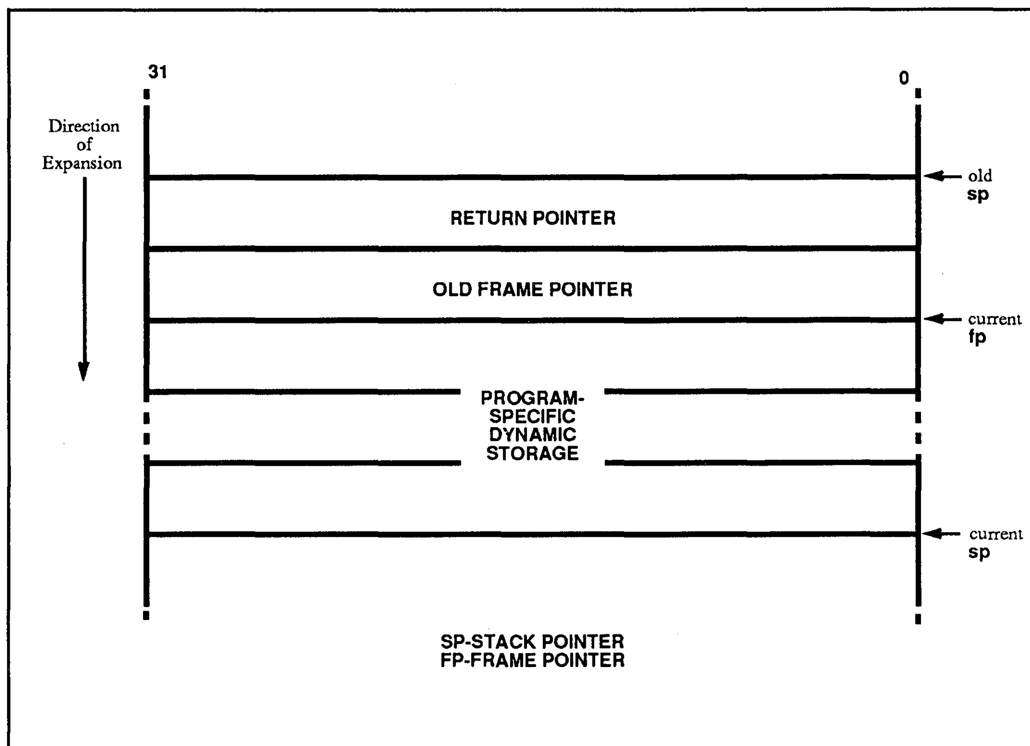


Figure 8-2. Stack Frame Format

8.3.1 Stack Entry and Exit Code

Example 8-2 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.

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 **r2**. The code example in Example 8-3 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 a lowest-level subroutine.

```
// Subroutine entry
    adds    -(Locals+8), sp,      sp  // Allocate stack space for local variables
                                      // Locals+8 must be a multiple of 16
    st.l       fp,    Locals(sp)      // Save old frame pointer below old SP
    adds     Locals,    sp,       fp  // Set new frame pointer
    st.l       r1,      4(fp)         // Save return address
    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 instruction
    adds       8,      sp,        sp  // Deallocate frame pointer save area
```

Example 8-2. Subroutine Entry and Exit with Frame Pointer

```
// Subroutine entry
    addu       -Locals,     r2, r2  // Allocate stack space for local variables
                                    // -Locals must be a multiple of 16

// Subroutine exit
    bri        r1                   // Return to caller after next instruction
    addu       Locals,      r2, r2  // Restore stack pointer
```

Example 8-3. Subroutine Entry and Exit without Frame Pointer

8.3.2 Dynamic Memory Allocation on the Stack

Consider a function **alloca** which 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 8-4, and a separate stack pointer and frame pointer are required.

```
_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 instruction
   mov        sp,     r16           // Set return value to allocated space
```

Example 8-4. Possible Implementation of alloca

8.4 MEMORY ORGANIZATION

Figure 8-3 suggests an overall memory layout. The i860 Linker needs to know by default where to assign code and data inside a program. The output of the linker must normally be executable without fixups. Code and data of both the application and operating system can share a single four-gigabyte address space. The example memory map assumes paging is being used to place DRAM-resident code in the upper 256 Mbytes of the address space.

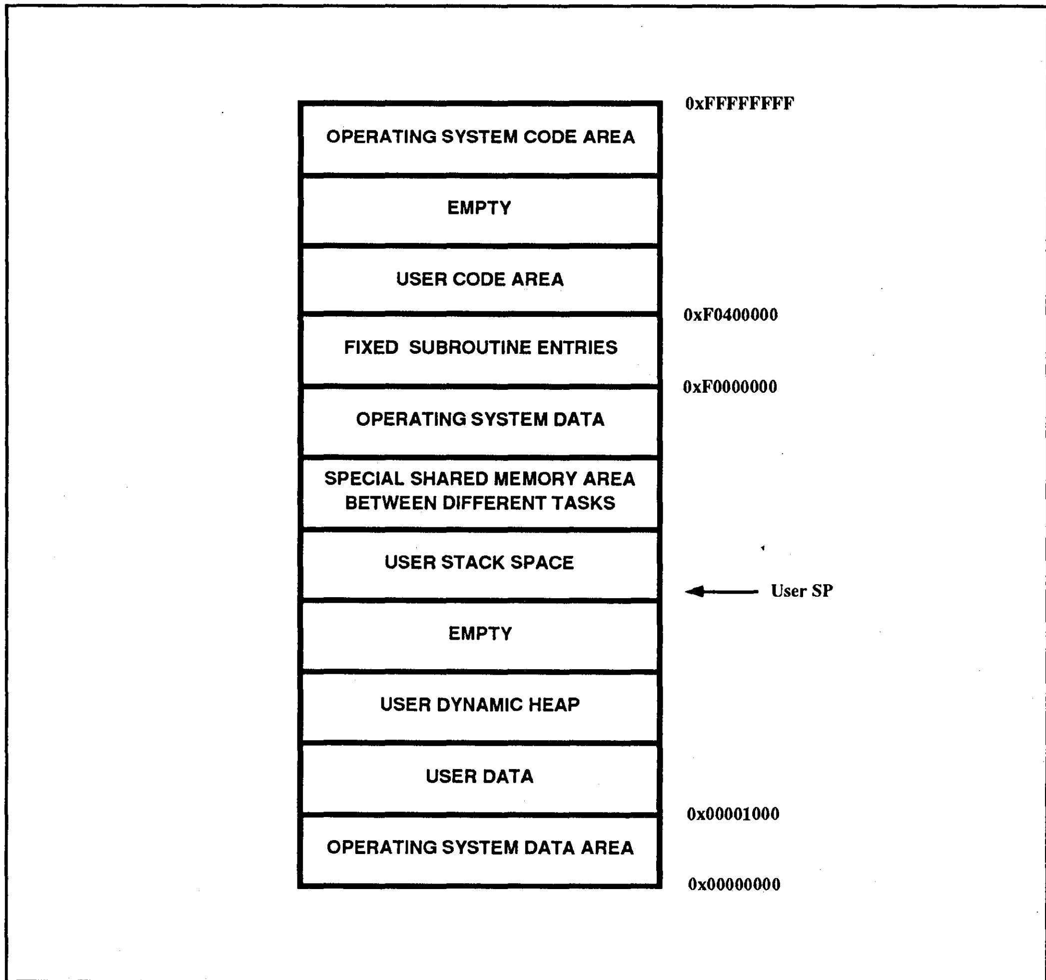


Figure 8-3. Example Memory Layout

The first four Kbytes (first page) of the address space are reserved for the operating system. It should be a supervisor-only page and should not be swappable. Uninitialized external address references in user programs (which are equivalent to an assembly-language address expression of the form **0(r0)**) reference this first page and cause a trap.

The data space for the application begins at 0x1000 (second page). It is all readable and writable. The total data address space available to the application should be over 3500 Mbytes. The user's data space has the following sections:

- A user-data portion whose size and content is defined by the program and development tools.
- A section called the heap whose size is determined at run time and can change as the program executes.
- A stack section.

The application's stack area starts at some address set by the OS and grows downward. The starting address of the stack would normally be at a four-Mbyte boundary to allow easy page-table formatting. The stack's starting address is not known in advance. It depends on how much address space is used by the operating system at the top of the address space.

The operating system may also want to reserve some portion of the application's address space for shared memory areas with other tasks. UNIX System V allows such shared memory areas. The empty areas on the diagram if Figure 8-3 would normally be marked as not-present in the page table entries. Some special flag in the page table entry could allow the operating system to determine that the page is not usable instead of just not present in memory.

A four-Mbyte area of code space is reserved starting at 0xF0000000 for a set of entry addresses to subroutines commonly used by all application programs (math libraries and vector primitives, for example). These code sections are shared by all application programs. The code in this area is directly callable from user-level code and executes at user level. Standard i860 Microprocessor calling conventions are used for these subroutines. The size of this area is chosen as four Mbytes, because that size corresponds to a directory-level page table entry that all applications tasks can share. It should be large enough to contain all desirable shared code.

The application program code area starts at 0xF0400000. It can be as large as 248 Mbytes. The application code is write-protected. The operating system and application code spaces lie in the upper 256 Mbytes of the address space. The operating system code is in the upper part of the 256 Mbyte code space. The operating system code is protected from application programs. Because it is easier for the operating system to divide up the address space in four-Mbyte blocks, the minimum operating-system code allocation from the address space is probably four Mbytes. Additional space would be allocated in four-Mbyte increments.

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.

Programming Examples 9

Chapter 9
Programming Examples

9.1 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 9-1 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 9-1. Sign Extension

Example 9-2 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 9-2. Loading Small Unsigned Integers

9.2 SINGLE-PRECISION DIVIDE

Example 9-3 computes $Z = X \div Y$ 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. If the result is referenced by the next instruction, 22 clocks are required to perform the divide.

```
// SINGLE-PRECISION DIVIDE

//      The dividend X is in f6
//      The divisor Y is in f2
//      The result Z is left 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 srcl
    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 9-3. Single-Precision Divide

9.3 DOUBLE-PRECISION DIVIDE

Example 9-4 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. If the result is referenced by the next instruction, 38 clocks are required to perform the divide.

```
// DOUBLE-PRECISION DIVIDE

//      The dividend X is in f2
//      The divisor Y is in f4
//      The result Z is left in f8

    frcp.dd f4,     f6              // first guess has 2**-8 error
    fmul.dd f4,     f6,     f8      // guess * divisor
    fld.d   flttwo, 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 srcl
    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 srcl
    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 9-4. Double-Precision Divide

9.4 INTEGER MULTIPLY

A 32-bit integer multiply is implemented in Example 9-5 by transferring the operands to floating-point registers and using the **fmlow** instruction. If the result is referenced in the next instruction, nine clocks are required. Five clocks can be overlapped with other operations.

```
// 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
//  Two core instructions can be inserted here without penalty.
    fxfr    f6,     r6
//  One core instruction can be inserted here without penalty.
```

Example 9-5. Integer Multiply

9.5 CONVERSION FROM SIGNED INTEGER TO DOUBLE

The strategy used in Example 9-6 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 i860 Microprocessor. The value of BC + BN is $2^{52}+2^{31}$ (0x4330__0000__8000__0000).

```
// 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 9-6. Single to Double Conversion

The conversion requires 7 clocks if the result is referenced in the next instruction. Three clocks can be overlapped with other operations.

9.6 SIGNED INTEGER DIVIDE

Example 9-7 combines the techniques of Section 9.3 and 9.5. It requires 62 clocks (59 clocks without remainder).

```
// 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,     r4  //
  ixfr    r4,     f4          //
  fmov.ss f7,     f5          //
  xorh    0x8000, r5,     r5  //
  fsub.dd f4,     f6,     f4  //
  ixfr    r5,     f2          //
  fmov.ss f7,     f3          //
  fsub.dd f2,     f6,     f2  //

// Do Floating-Point Divide
  fld.d   fdtwo,  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
  fmlow.dd f10,   f8,     f10 // quotient * denominator
  fxfr    f10,    r4
  fxfr    f8,     r6          // transfer quotient
  subs    r5,     r7,     r7  // remainder = numerator - quotient * denominator
```

Example 9-7. Signed Integer Divide

9.7 STRING COPY

Example 9-8 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 9-8 juggles characters of the string between two registers.

```
// 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 9-8. String Copy

9.8 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.

Continuous Operation — Receiving expected results with the initiation of each new pipelined instruction.

Flushing — Retrieving the results that remain in the pipeline after the pipelined instruction sequence has terminated.

Example 9-9 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 flushes the pipeline by executing three dummy addition instructions with **f0** (which always contains zero) as the operands.

```
// 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

// Flushing 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 9-9. Pipelined Add

9.9 PIPELINING OF DUAL-OPERATION INSTRUCTIONS

When using dual-operation instructions (all of which are pipelined), code that primes and flushes the pipelines must take into account both the adder and multiplier pipelines. Example 9-10 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 tracking values through the pipelines, assume that the actual matrices to be multiplied have the following values:

$$\mathbf{A} = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0] \qquad \mathbf{B} = \begin{bmatrix} 8.0 \\ 7.0 \\ 6.0 \\ 5.0 \\ 4.0 \\ 3.0 \\ 2.0 \\ 1.0 \end{bmatrix}$$

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 9-10 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 9-10 than in Example 9-9. 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 flushed from the multiplier pipeline, then uses **pfadd** instructions to put zeros in all stages of the adder pipeline.

9.10 DUAL INSTRUCTION MODE

The previous Example 9-9 and Example 9-10 showed how the i860 Microprocessor 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 are not realistic, because they assume that the data is

```
// PIPELINED DUAL-OPERATION INSTRUCTION

//                              Multiplier          Adder
//                                Stages            Stages
//                              1    2    3     1      2      3    Result

// Priming phase
    ml2apm.ss f4, f12,f0   // 1*8  ??   ??    ??     ??     ??   Discard
    ml2apm.ss f5, f13,f0   // 2*7  1*8  ??    ??     ??     ??   Discard
    ml2apm.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
    ml2apm.ss f7, f15,f0   // 4*5  3*6  14   8+0    0+0    0    Discard
    ml2apm.ss f8, f16,f0   // 5*4  4*5  18   14+0   8+0    0    Discard
    ml2apm.ss f9, f17,f0   // 6*3  5*4  20   18+0   14+0   8    Discard
    ml2apm.ss f10,f18,f0   // 7*2  6*3  20   20+8   18+0   14   Discard
    ml2apm.ss f11,f19,f0   // 8*1  7*2  18   20+14  20+8   18   Discard
// For larger matrices, include more instructions here

// Flushing phase
    ml2apm.ss f0, f0, f0   // 0*0  8*1  14   18+18  20+14  28   Discard
    ml2apm.ss f0, f0, f0   // 0*0  0*0  8    14+28  18+18  34   Discard
    ml2apm.ss f0, f0, f0   // 0*0  0*0  0     8+34  14+28  36   Discard

    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 9-10. Pipelined Dual-Operation Instruction

already loaded in registers. Example 9-11 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 9-11 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 slighty degraded if the destination of a **fld.d** were referenced as a source operand in the next 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.

9.11 CACHE STRATEGIES FOR MATRIX DOT PRODUCT

Calculations that use (and reuse) massive amounts of data may render significantly less than optimum performance unless their memory access demands are carefully taken into consideration during algorithm design. The prior Example 9-11 easily executes at near the theoretical maximum speed of the i860 Microprocessor 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 ...

$$C_{i,j} = A_{i,1}B_{1,j}+A_{i,2}B_{2,j}+ \ldots +A_{i,M}B_{M,j} \text{ (for } 1 \leq i \leq L,\ 1 \leq j \leq N)$$

The basic algorithm for calculation of a dot product appears in Example 9-10. 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 9-12 and 9-13 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.

```
// SINGLE-PRECISION VECTOR SUM
//      input:  r16 - vector address
//              r17 - vector size (must be > 5)
//      output: f16 - sum of vector elements
spvsum::
    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,  loop1 // Initialize LCC
  d.pfadd.ss    f0,     f0,     f0  // Clear adder pipe (3)
    fld.d       8(r16)++,       f22 // Load 3rd and 4th elements
loop1::
  d.pfadd.ss    f20,    f30,    f30 // Add f20 to pipeline
    bla         r21,    r17,  loop2 // If more, go to loop2 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          sumup               // Exit loop after adding
  d.pfadd.ss    f21,    f31,    f31 // f21 to the pipeline
    nop
loop2::
  d.pfadd.ss    f22,    f30,    f30 // Add f22 to pipeline
    bla         r21,    r17,  loop1 // If more, go to loop1 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

sumup::                             // 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 9-11. Dual-Instruction Mode

- Example 9-12 depends solely on cached loads.
- Example 9-13 depends on a mix of cached and pipelined loads.

Example 9-12 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 9-12 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.

```
// MATRIX MULTIPLY, C = A * B, CACHED LOADS ONLY

// Registers loaded by calling routine
//  r16 - pointer into A, stored in memory by rows
//  r17 - pointer into B, stored in memory by columns
//  r18 - pointer into C, stored in memory by rows
//  r19 - L, the number of rows in A
//  r20 - M, the number of columns in A and rows in B
//  r21 - N, the number of columns in B

// Registers used locally
//  r28 - row/column counter decremented by bla for loop control
//  r27 - decrementor for row/column pointers
//  r26 - counter of rows in A
//  r25 - counter of columns in B
//  r24 - temporary pointer into B
//  r23 - number of bytes in row of A or column of B
//  f4..f11     - matrix A row values
//  f12..f19    - matrix B column values
//  f20..f22    - temporary results

  shl           2,r20,r23        // Number of bytes in M entries
  adds          -8,r0,r27        // Set decrementor for bla
  adds          -8,r20,r28       // Initialize row/column counter
  adds          -4,r18,r18       // Start C index one entry low
  d.fiadd.dd    f0,f0,f0         // Initiate dual-instruction mode
  adds          -1,r19,r26       // Make row counter zero relative
  d.fnop                         // First dual-mode pair
    bla     r27,r28,start_row    // Initialize LCC
  d.fnop                         //
    subs          r16,r23,r16    // Start pointer to A one row low
start_row::                    // Executed once per row of A
  d.pfmul.ss    f0,f0,f0         //
    mov           r17,r24        // Point to first col of B
  d.pfmul.ss    f0,f0,f0         //
    adds          r23,r16,r16    // Point to next row of A
  d.pfmul.ss    f0,f0,f0         //
    fld.q         16(r24),f16    // Load 4 entries of B
  d.pfadd.ss    f0,f0,f0         //
    fld.q         16(r16),f8     // Load 4 entries of A
  d.pfadd.ss    f0,f0,f0         //
    adds          -1,r21,r25     // Initialize column counter
  d.pfadd.ss    f0,f0,f0         //
    fld.q         0(r16),f4      // Load 4 entries of A
```

Example 9-12. Matrix Multiply, Cached Loads Only (sheet 1 of 2)

```
inner_loop:: // Process eight entries of row of A with eight of col of B
  d.m12apm.ss    f8, f16,f20      //
    fld.q          0(r24),f12     // Load 4 entries of B
  d.m12apm.ss    f9, f17,f20      //
    adds           32,r16,r16     // Bump pointer to A by 8 entries
  d.m12apm.ss    f10,f18,f20      //
    adds           32,r24,r24     // Bump pointer to B by 8 entries
  d.m12apm.ss    f11,f19,f20      //
    fld.q          16(r24),f16    // Load 4 entries of B
  d.m12apm.ss    f4, f12,f20      //
    fld.q          16(r16),f8     // Load 4 entries of A
  d.m12apm.ss    f5, f13,f20      //
    nop                           //
  d.m12apm.ss    f6, f14,f20      //
    bla    r27,r28,inner_loop     // Loop until end of row/column
  d.m12apm.ss    f7, f15,f21      //
    fld.q          0(r16),f4      // Load 4 entries of A
// End Inner Loop.  End of row/column
  d.m12apm.ss    f0, f0, f22      //
    subs           r16,r23,r16    // Set A pointer back to beginning of row
  d.m12apm.ss    f0, f0, f20      //
    adds           -8,r20,r28     // Reinitialize row/column counter
  d.m12apm.ss    f0, f0, f21      //
    nop                           //
  d.pfadd.ss     f0, f0, f22      //
    bla        r27,r28,inner_loop // Won't branch; initializes LCC
  d.pfadd.ss     f0, f0, f20      //
    fld.q          16(r16), f8    // Load 4 entries of A
  d.pfadd.ss     f0, f0, f21      //
    fld.q          16(r24), f16   // Load 4 entries of B
  d.fadd.ss      f20,f22,f22      //
    fld.q          0(r16), f4     // Load 4 entries of A
  d.fadd.ss      f21,f22,f22      //
    adds           -1,r25,r25     // Decrement column counter
  d.pfadd.ss     f0, f0, f0       //
    fst.l          f22, 4(r18)++  // 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            r26,r0,r0      // Is row counter zero?
  d.fnop                          //
    bnc.t          start_row      // Taken if row counter not zero
  d.fnop                          //
    adds           -1,r26,r26     // Decrement row counter
  fnop                            // Initiate exit from dual mode
    nop                           //
  fnop                            // Last dual-mode pair
    nop                           // End
```

Example 9-12. Matrix Multiply, Cached Loads Only (sheet 2 of 2)

```
// MATRIX MULTIPLY, C = A * B, CACHED AND PIPELINED LOADS MIXED

// Registers loaded by calling routine
//  r16 - pointer into A, stored in memory by rows
//  r17 - pointer into B, stored in memory by columns
//  r18 - pointer into C, stored in memory by rows
//  r19 - L, the number of rows in A
//  r20 - M, the number of columns in A and rows in B
//  r21 - N, the number of columns in B

// Registers used locally
//  r29 - temporary pointer into A
//  r28 - row/column counter decremented by bla for loop control
//  r27 - decrementor for row/column pointers
//  r26 - counter of rows in A
//  r25 - counter of columns in B
//  r24 - temporary pointer into B
//  r23 - number of bytes in row of A or column of B
//  f4..f11     - matrix A row values
//  f12..f19    - matrix B column values
//  f20..f22    - temporary results

  mov           r17,r24          // Pointer to B
  shl           2,r20,r23        // Number of bytes in M entries
  adds          -8,r0,r27        // Set decrementor for bla
  adds          -8,r20,r28       // Initialize row/column counter
  d.fiadd.dd    f0,f0,f0         // Initiate dual-instruction mode
  adds          -4,r18,r18       // Start C index one entry low
  d.fnop                         // First dual-mode pair
    adds          -1,r19,r26     // Make row counter zero relative
  d.fnop                         //
    bla     r27,r28,start_row    // Initialize LCC
  d.fnop                         //
    mov           r16,r29        // Pointer to A
start_row::                    // Executed once per row of A
  d.pfmul.ss    f0,f0,f0         //
    pfld.d        0(r24),f0      // Load 2 entries of B into load pipe
  d.pfmul.ss    f0,f0,f0         //
    pfld.d        8(r24)++,f0    // Load 2 entries of B into load pipe
  d.pfmul.ss    f0,f0,f0         //
    pfld.d        8(r24)++,f0    // Load 2 entries of B into load pipe
  d.pfadd.ss    f0,f0,f0         //
    fld.q         0(r29),f4      // Load 4 entries of A
  d.pfadd.ss    f0,f0,f0         //
    pfld.d        8(r24)++,f12   // Load 2 entries of B
  d.pfadd.ss    f0,f0,f0         //
    adds          -1,r21,r25     // Initialize column counter
  d.fnop                         //
    pfld.d        8(r24)++,f14   // Load 2 entries of B
inner_loop:: // Process eight entries from row of A with eight from col of B
  d.m12apm.ss   f4, f12,f0       //
    fld.q         16(r29)++,f8   // Load 4 entries of A
  d.m12apm.ss   f5, f13,f0       //
    pfld.d        8(r24)++,f16   // Load 2 entries of B
  d.m12apm.ss   f6, f14,f0       //
    pfld.d        8(r24)++,f18   // Load 2 entries of B
```

Example 9-13. Matrix Multiply, Cached and Pipelined Loads (sheet 1 of 2)

```
    d.ml2apm.ss    f7, f15,f0          //
      fld.q          16(r29)++,f4      // Load 4 entries of A
    d.ml2apm.ss    f8, f16,f0          //
      nop                              //
    d.ml2apm.ss    f9, f17,f0          //
      pfld.d         8(r24)++,f12      // Load 2 entries of B
    d.ml2apm.ss    f10,f18,f0          //
      bla      r27,r28,inner_loop      // Loop until end of row/column
    d.ml2apm.ss    f11,f19,f0          //
      pfld.d         8(r24)++,f14      // Load 2 entries of B
// End Inner Loop.  End of row/column
    d.ml2apm.ss    f0, f0, f0          //
      nop                              //
    d.ml2apm.ss    f0, f0, f0          //
      adds           -8,r20,r28        // Reinitialize row/column counter
    d.ml2apm.ss    f0, f0, f0          //
      mov            r16,r29           // Set A pointer back to beginning of row
    d.pfadd.ss     f0, f0, f22         //
      fld.q          0(r29), f4        // Load first 4 entries of row of A
    d.pfadd.ss     f0, f0, f20         //
      bla      r27,r28,inner_loop      // Won't branch; initializes LCC
    d.pfadd.ss     f0, f0, f21         //
      nop                              //
    d.fadd.ss      f20,f22,f22         //
      nop                              //
    d.fadd.ss      f21,f22,f22         //
      adds           -1,r25,r25        // Decrement column counter
    d.pfadd.ss     f0, f0, f0          //
      fst.l          f22, 4(r18)++     // 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            r17,r24           // Point to first col of B
    d.fnop                             //
      adds           r16,r23,r16       // Bump pointer to A by one row
    d.fnop                             //
      mov            r16,r29           // Set A index to beginning of next row
// Continue with next row of A?
    d.fnop                             //
      xor            r26,r0,r0         // Is row counter zero?
    d.fnop                             //
      bnc.t          start_row         // Taken if row counter not zero
    d.fnop                             //
      adds           -1,r26,r26        // Decrement row counter
    fnop                               // Initiate exit from dual mode
      nop                              //
    fnop                               // Last dual-mode pair
      nop                              // End
```

Example 9-13. Matrix Multiply, Cached and Pipelined Loads (sheet 2 of 2)

Example 9-13 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 9-13 is suitable for larger matrices than the strategy in Example 9-12 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 9-13 may not perform as well as Example 9-12, because, even when there is sufficient space in the cache for elements of matrix **B**, Example 9-13 does not use it.

Instruction Set Summary

A

Appendix A
Instruction Set Summary

Key to abbreviations:

src1	A register (integer or floating-point depending on class of instuction) 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 is permitted.
src2	A register (integer or floating-point depending on class of instruction).
rdest	A register (integer or floating-point depending on class of instruction).
freg	A floating-point register.
ireg	An integer register.
ctrlreg	One of the control registers **fir**, **psr**, **epsr**, **dirbase**, **db**, or **fsr**.
#const	A 16-bit immediate constant or address offset that the i860 Microprocessor sign-extends to 32 bits when computing the effective address.
mem.x(address)	The contents of the memory location indicated by *address* with a size of *x*.
.p	Precision specification. Unless otherwise specified, floating-point operations accept single- or double-precision source operands and produce a result of equal or greater precision. Both input operands must have the same precision. The source and result precision are specified by a two-letter suffix to the mnemonic of the operation, as shown in the table below.

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

.w	**.ss** (32 bits), or **.dd** (64 bits)
.x	**.b** (8 bits), **.s** (16 bits), or **.l** (32 bits)
.y	**.l** (32 bits), **.d** (64 bits), or **.q** (128 bits)
.z	**.l** (32 bits), or **.d** (64 bits)

lbroff — A signed, 26-bit, immediate, relative branch offset

sbroff — A signed, 16-bit, immediate, relative branch offset

brx — A function that computes the target address of a branch by shifting the offset (either *lbroff* or *sbroff*) left by two bits, sign-extending it to 32 bits, and adding the result to the address of the current control-transfer instruction plus four.

src1s — An integer register or a 5-bit immediate constant that is zero-extended to 32 bits.

comp2 — A function that returns the two's complement of its argument.

PM — The pixel mask, which is considered as an array of eight bits PM[0]..PM[7], where PM[0] is the least-significant bit.

Instruction Definitions in Alphabetical Order

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

bri [*src1ni*] .... **Branch Indirect Unconditionally**

```
Execute one more sequential instruction
IF      any trap bit in psr is set
THEN    copy PU to U, PIM to IM in psr
        clear trap bits
        IF      DS is set and DIM is reset
        THEN    enter dual-instruction mode after executing one
                  instruction in single-instruction mode
        ELSE    IF      DS is set and DIM is set
                THEN    enter single-instruction mode after executing one
                          instruction in dual-instruction mode
                ELSE    IF      DIM is set
                        THEN    enter dual-instruction mode
                                  for next two instructions
```

ELSE enter single-instruction mode for next two instructions
FI
FI
FI
FI
Continue execution at address in *src1ni*

(The original contents of *src1ni* is used even if the next instruction modifies *src1ni*. Does not trap if *src1ni* is misaligned.)

bte *src1s, src2, sbroff* .... **Branch If Equal**
IF *src1s* = *src2*
THEN continue execution at *brx(sbroff)*
FI

btne *src1s, src2, sbroff* .... **Branch If Not Equal**
IF $src1s \neq src2$
THEN continue execution at *brx(sbroff)*
FI

call *lbroff* .... **Subroutine Call**
r1 ← address of next sequential instruction + 4
Execute one more sequential instruction
Continue execution at *brx(lbroff)*

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

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

faddp *src1, src2, rdest* .... **Add with Pixel Merge**
rdest ← *src1* + *src2*
Shift and load MERGE register as defined in Table A-1

Table A-1. FADDP MERGE Update

Pixel Size (from PS)	Field 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

faddz *src1, src2, rdest* .... **Add with Z Merge**
rdest ← *src1* + *src2*
Shift MERGE right 16 and load fields 31..16 and 63..48

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

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

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

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

fld.y *src1*(*src2*)++, *freg* .... **Floating-Point Load (Autoincrement)**
freg ← mem.y (*src1* + *src2*)
IF autoincrement
THEN *src2* ← *src1* + *src2*
FI

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

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

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

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

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

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

frcp.p *src2, rdest* .... **Floating-Point Reciprocal**

rdest ← 1 / *src2* with maximum mantissa error $< 2^{-7}$

frsqr.p *src2, rdest* .... **Floating-Point Reciprocal Square Root**

rdest ← $1 / \sqrt{src2}$ with maximum mantissa error $< 2^{-7}$

fst.y *freg, src1(src2)* ....**Floating-Point Store (Normal)**
fst.y *freg, src1(src2)++* .... **Floating-Point (Autoincrement)**

mem.y (*src2* + *src1*) ← *freg*
IF autoincrement
THEN *src2* ← *src1* + *src2*
FI

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

rdest ← *src1* − *src2*

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

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

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

ireg ← *src1*

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

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

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

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

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

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

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

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

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

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

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

nop .... **Core-Unit No Operation**
Assembler pseudo-operation
nop = **shl r0, r0, r0**

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

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

pfadd.p *src1, src2, rdest* .... **Pipelined Floating-Point Add**
rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← *src1* + *src2*

pfaddp *src1, src2, rdest* .... **Pipelined Add with Pixel Merge**
rdest ← last-stage I-result
last-stage I-result ← *src1* + *src2*
Shift and load MERGE register from *src1* + *src2* as defined in Table A-1

pfaddz *src1, src2, rdest* .... **Pipelined Add with Z Merge**
rdest ← last-stage I-result
last-stage I-result ← *src1* + *src2*
Shift MERGE right 16 and load fields 31..16 and 63..48 from *src1* + *src2*

pfam.p *src1, src2, rdest* .... **Pipelined Floating-Point Add and Multiply**
rdest ← last A-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 + A-op2
M pipeline first stage ← M-op1 × M-op2

pfeq.p *src1, src2, rdest* .... **Pipelined Floating-Point Equal Compare**
rdest ⟵ last A-stage result
CC set if *src1* = *src2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfgt.p *src1, src2, rdest* .... **Pipelined Floating-Point Greater-Than Compare**
(Assembler clears R-bit of instruction)
rdest ⟵ last A-stage result
CC set if *src1* > *src2*, else cleared
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfiadd.w *src1, src2, rdest* .... **Pipelined Long-Integer Add**
rdest ⟵ last-stage I-result
last-stage I-result ⟵ *src1* + *src2*

pfisub.w *src1, src2, rdest* ....**Pipelined Long-Integer Subtract**
rdest ⟵ last-stage I-result
last-stage I-result ⟵ *src1* − *src2*

pfix.p *src1, rdest* .... **Pipelined Floating-Point to Integer Conversion**
rdest ⟵ last A-stage result
Advance A pipeline one stage
A pipeline first stage ⟵ 64-bit value with low-order 32 bits
equal to integer part of *src1* rounded

Pipelined Floating-Point Load
pfld.z *src1*(*src2*), *freg* .... **(Normal)**
pfld.z *src1*(*src2*)++, *freg* .... **(Autoincrement)**
freg ⟵ mem.z (third previous **pfld**'s (*src1* + *src2*))
(where .z is precision of third previous **pfld.z**)
IF autoincrement
THEN *src2* ⟵ *src1* + *src2*
FI

pfle.p *src1, src2, rdest* .... **Pipelined F-P Less-Than or Equal Compare**
Assembler pseudo-operation, identical to **pfgt.p** except that
assembler sets R-bit of instruction.
rdest ⟵ last A-stage result
CC clear if *src1* ≤ *src2*, else set
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfmam.p *src1, src2, rdest* .... **Pipelined Floating-Point Add and Multiply**
rdest ⟵ last M-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ⟵ A-op1 + A-op2
M pipeline first stage ⟵ M-op1 × M-op2

pfmov.p *src1, rdest* .... **Pipelined Floating-Point Reg-Reg Move**
Assembler pseudo-operation
pfmov.ss *src1, rdest* = **pfiadd.ss** *src1*, **f0**, *rdest*
pfmov.dd *src1, rdest* = **pfiadd.dd** *src1*, **f0**, *rdest*
pfmov.sd *src1, rdest* = **pfadd.sd** *src1*, **f0**, *rdest*
pfmov.ds *src1, rdest* = **pfadd.ds** *src1*, **f0**, *rdest*

pfmsm.p *src1, src2, rdest* .... **Pipelined Floating-Point Subtract and Multiply**
rdest ← last M-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 − A-op2
M pipeline first stage ← M-op1 × M-op2

pfmul.p *src1, src2, rdest* .... **Pipelined Floating-Point Multiply**
rdest ← last M-stage result
Advance M pipeline one stage
M pipeline first stage ← *src1* × *src2*

pfmul3.p *src1, src2, rdest* .... **Three-Stage Pipelined Multiply**
rdest ← last M-stage result
Advance 3-Stage M pipeline one stage
M pipeline first stage ← *src1* × *src2*

pform *src1, rdest* .... **Pipelined OR to MERGE Register**
rdest ← last-stage I-result
last-stage I-result ← *src1* OR MERGE
MERGE ← 0

pfsm.p *src1, src2, rdest* .... **Pipelined Floating-Point Subtract and Multiply**
rdest ← last A-stage result
Advance A and M pipeline one stage (operands accessed before advancing pipeline)
A pipeline first stage ← A-op1 − A-op2
M pipeline first stage ← M-op1 × M-op2

pfsub.p *src1, src2, rdest* .... **Pipelined Floating-Point Subtract**
rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← *src1* − *src2*

pftrunc.p *src1, rdest* .... **Pipelined Floating-Point to Integer Conversion**
rdest ← last A-stage result
Advance A pipeline one stage
A pipeline first stage ← 64-bit value with low-order 32 bits
equal to integer part of *src1*

pfzchkl *src1, src2, rdest* .... **Pipelined 32-Bit Z-Buffer Check**
Consider *src1*, *src2*, and *rdest* as arrays of two 32-bit
fields *src1*(0)..*src1*(1), *src2*(0)..*src2*(1), and *rdest*(0)..*rdest*(1)
where zero denotes the least-significant field.

PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
    PM [i + 6] ← *src2*(i) ≤ *src1*(i) (unsigned)
    rdest(i) ← last-stage I-result
    last-stage I-result ← smaller of *src2*(i) and *src1*(i)
OD
MERGE ← 0

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

pst.d *freg*, #*const*(*src2*) .... **Pixel Store**
pst.d *freg*, #*const*(*src2*)++ .... **Pixel Store Autoincrement**
Pixels enabled by PM in mem.d (*src2* + #*const*) ← *freg*
Shift PM right by 8/pixel size (in bytes) bits
IF autoincrement THEN *src2* ← #*const* + *src2* FI

shl *src1, src2, rdest* .... **Shift Left**
rdest ← *src2* shifted left by *src1* bits

shr *src1, src2, rdest* .... **Shift Right**
SC (in **psr**) ← *src1*
rdest ← *src2* shifted right by *src1* bits

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

shrd *src1ni, src2, rdest* .... **Shift Right Double**
rdest ← low-order 32 bits of *src1ni:src2* shifted right by SC bits

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

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

subs *src1, src2, rdest* .... **Subtract Signed**
rdest ← *src1* − *src2*
OF ← (bit 31 carry ≠ bit 30 carry)

CC set if *src2* > *src1* (signed)
CC clear if *src2* ≤ *src1* (signed)

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

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

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

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

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

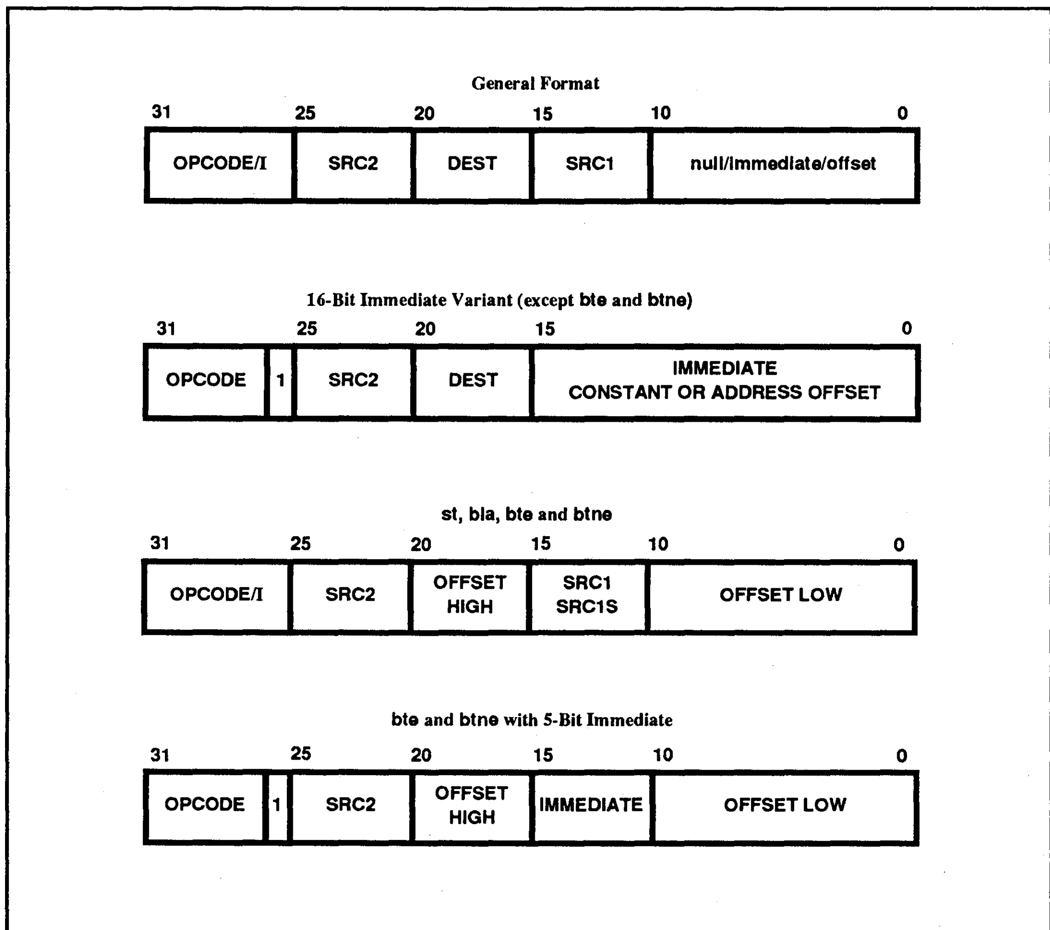
Instruction Format and Encoding

B

Appendix B
Instruction Format and Encoding

All instructions are 32 bits long and begin on a four-byte boundary. Among the core instructions, there are two general formats: REG-format and CTRL-format. Within the REG-format are several variations.

REG-Format Instructions



The *src2* field selects one of the 32 integer registers (most instructions) or one of the control registers (**st.c** and **ld.c**). *Dest* selects one of the 32 integer registers (most instructions) or floating-point registers (**fld**, **fst**, **pfld**, **pst**, **ixfr**). For instructions where *src1* is optionally an immediate 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 constant is contained in the *src1* field. For **st**, **bte**, **btne**, and **bla**, the upper five bits of the *offset* or *broffset* are contained in the *dest* field instead of *src1*, and the lower 11 bits of *offset* are the lower 11 bits of the instruction.

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

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

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

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

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

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

REG-Format Opcodes

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

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

LS Load/Store
- 0 —Load
- 1 —Store

SO Signed/Ordinal
- 0 —Ordinal
- 1 —Signed

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

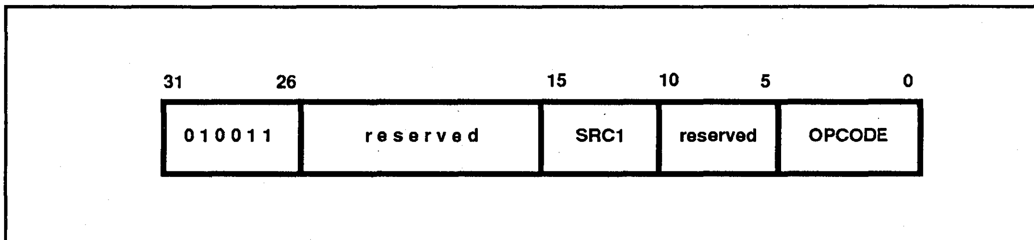
AS Add/Subtract
- 0 —Add
- 1 —Subtract

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

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

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

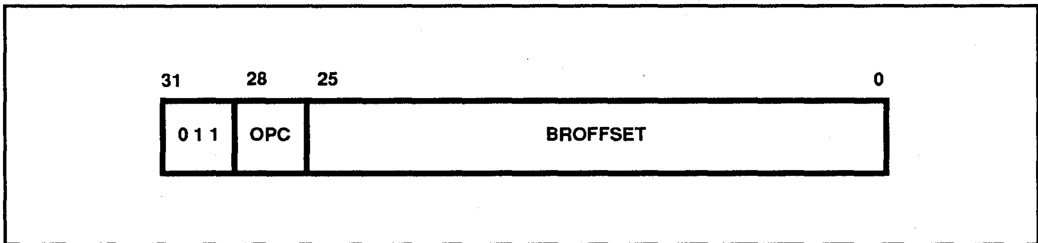
Core Escape Instructions



Core Escape Opcodes

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

CTRL-Format Instructions



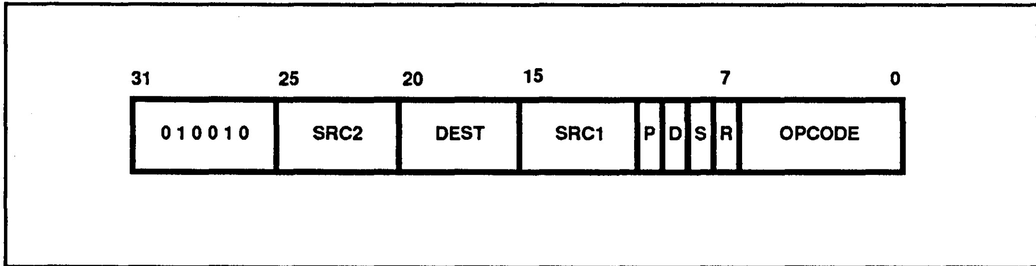
BROFFSET is a signed 26-bit relative branch offset.

CTRL-Format Opcodes

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

T Taken
0 —bc or bnc
1 —bc.t or bnc.t

Floating-Point Instruction Encoding



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

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

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

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

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

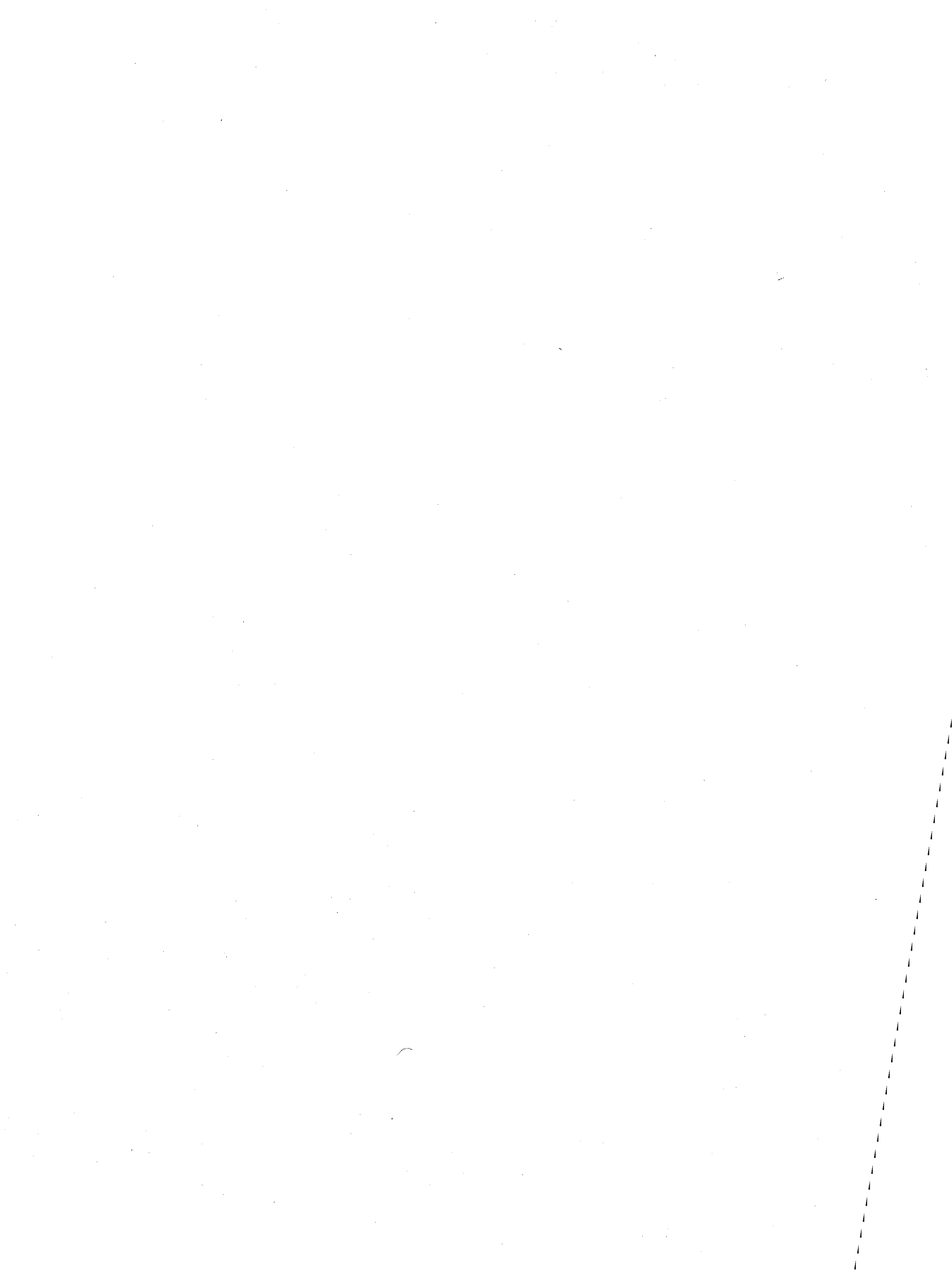
Floating-Point Opcodes

		6						0
pfam pfmam	Add and Multiply* Multiply with Add*	0	0	0	DPC			
pfsm pfmsm	Subtract and Multiply* Multiply with Subtract*	0	0	1	DPC			
(p)fmul	Multiply	0	1	0	0	0	0	0
fmlow	Multiply Low	0	1	0	0	0	0	1
frcp	Reciprocal	0	1	0	0	0	1	0
frsqr	Reciprocal Square Root	0	1	0	0	0	1	1
pfmul3.dd	3-Stage Pipelined Multiply	0	1	0	0	1	0	0
(p)fadd	Add	0	1	1	0	0	0	0
(p)fsub	Subtract	0	1	1	0	0	0	1
(p)fix	Fix	0	1	1	0	0	1	0
pfgt/pfle**	Greater Than	0	1	1	0	1	0	0
pfeq	Equal	0	1	1	0	1	0	1
(p)ftrunc	Truncate	0	1	1	1	0	1	0
fxfr	Transfer to Integer Register	1	0	0	0	0	0	0
(p)fiadd	Long-Integer Add	1	0	0	1	0	0	1
(p)fisub	Long-Integer Subtract	1	0	0	1	1	0	1
(p)fzchkl	Z-Check Long	1	0	1	0	1	1	1
(p)fzchks	Z-Check Short	1	0	1	1	1	1	1
(p)faddp	Add with Pixel Merge	1	0	1	0	0	0	0
(p)faddz	Add with Z Merge	1	0	1	0	0	0	1
(p)form	OR with MERGE Register	1	0	1	1	0	1	0

*pfam and pfsm have P-bit set; pfmuladd and pfmulsub have P-bit clear.
**pfgt has R bit cleared; pfle has R bit set.

Instruction Timings

C



Appendix C
Instruction Timings

i860 Microprocessor instructions take one clock to execute unless a freeze condition is invoked. Freeze conditions and their associated delays are shown in the table below. Freezes due to multiple simultaneous cache misses result in a delay that is the sum of the delays for processing each miss by itself. Other multiple freeze conditions usually add only the delay of the longest individual freeze.

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

continued

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

continued

Freeze Condition	Delay
st.c	Two clocks
Result of graphics-unit instruction (other than **fmov**) used in next instruction when the next instruction is an adder or multiplier instruction	One clock
Result of graphics-unit instruction used in next instruction when the next instruction is a graphics-unit instruction	One clock
flush followed by **flush**	Two clocks
fst followed by pipelined floating-point operation that overwrites the register being stored	One clock

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 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 fault is reported on the subsequent floating-point instruction plus **pst**, **fst**, and 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, *src1* should not be the same as the *rdest* of the prior operation.

5. The *freg* should not reference the destination of the next instruction if that instuction is a pipelined floating-point operation.

6. The destination should not be a source operand of the next instruction.

7. When the prior operation is scalar and multiplier *op1* is *src1*, *src2* should not be the same as the *rdest* of the prior operation.

8. When the prior operation is scalar, *src1* and *src2* of the current operation should not be the same as *rdest* of the prior operation.

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

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

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

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

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

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

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

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

DOMESTIC DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc.
1015 Henderson Road
Huntsville 35805
Tel: (205) 837-6955

†Hamilton/Avnet Electronics
4940 Research Drive
Huntsville 35805
Tel: (205) 837-7210
TWX: 810-726-2162

Pioneer/Technologies Group, Inc.
4825 University Square
Huntsville 35805
Tel: (205) 837-9300
TWX: 810-726-2197

ARIZONA

†Hamilton/Avnet Electronics
505 S. Madison Drive
Tempe 85281
Tel: (602) 231-5140
TWX: 910-950-0077

Hamilton/Avnet Electronics
30 South McKiemy
Chandler 85226
Tel: (602) 961-6669
TWX: 910-950-0077

Arrow Electronics, Inc.
4134 E. Wood Street
Phoenix 85040
Tel: (602) 437-0750
TWX: 910-951-1550

Wyle Distribution Group
17855 N. Black Canyon Hwy.
Phoenix 85023
Tel: (602) 249-2232
TWX: 910-951-4282

CALIFORNIA

Arrow Electronics, Inc.
10824 Hope Street
Cypress 90630
Tel: (714) 220-6300

Arrow Electronics, Inc.
19748 Dearborn Street
Chatsworth 91311
Tel: (213) 701-7500
TWX: 910-493-2086

†Arow Electronics, Inc.
521 Weddell Drive
Sunnyvale 94086
Tel: (408) 745-6600
TWX: 910-339-9371

Arrow Electronics, Inc.
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
TWX: 888-064

†Arrow Electronics, Inc.
2961 Dow Avenue
Tustin 92680
Tel: (714) 838-5422
TWX: 910-595-2860

†Avnet Electronics
350 McCormick Avenue
Costa Mesa 92626
Tel: (714) 754-6071
TWX: 910-595-1928

†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
9650 Desoto Avenue
Chatsworth 91311
Tel: (818) 700-1161

†Hamilton Electro Sales
10950 W. Washington Blvd.
Culver City 20230
Tel: (213) 558-2458
TWX: 910-340-6364

Hamilton Electro Sales
1361B West 190th Street
Gardena 90248
Tel: (213) 217-6700

†Hamilton/Avnet Electronics
3002 'G' Street
Ontario 91761
Tel: (714) 989-9411

†Avnet Electronics
20501 Plummer
Chatsworth 91351
Tel: (213) 700-6271
TWX: 910-494-2207

CALIFORNIA (Cont'd.)

†Hamilton Electro Sales
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4150
TWX: 910-595-2638

†Hamilton/Avnet Electronics
4103 Northgate Blvd.
Sacramento 95834
Tel: (916) 920-3150

Wyle Distribution Group
124 Maryland Street
El Segundo 90254
Tel: (213) 322-8100

Wyle Distribution Group
7382 Lampson Ave.
Garden Grove 92641
Tel: (714) 891-1717
TWX: 910-348-7140 or 7111

Wyle Distribution Group
11151 Sun Center Drive
Rancho Cordova 95670
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: 910-338-0296

†Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel: (714) 863-9953
TWX: 910-595-1572

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 Electronics
8765 E. Orchard Road
Suite 708
Englewood 80111
Tel: (303) 740-1017
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 Electronics
Commerce Industrial Park
Commerce Drive
Danbury 06810
Tel: (203) 797-2800
TWX: 710-456-9974

†Pioneer Electronics
112 Main Street
Norwalk 06851
Tel: (203) 853-1515
TWX: 710-468-3373

FLORIDA

†Arrow Electronics, Inc.
400 Fairway Drive
Suite 102
Deerfield Beach 33441
Tel: (305) 429-8200
TWX: 510-955-9456

Arrow Electronics, Inc.
37 Skyline Drive
Suite 3101
Lake Marv 32746
Tel: (407) 323-0252
TWX: 510-959-6337

†Hamilton/Avnet Electronics
6801 N.W. 15th Way
Ft. Lauderdale 33309
Tel: (305) 971-2900
TWX: 510-956-3097

†Hamilton/Avnet Electronics
3197 Tech Drive North
St. Petersburg 33702
Tel: (813) 576-3930
TWX: 810-863-0374

FLORIDA (Cont'd.)

†Hamilton/Avnet Electronics
6947 University Boulevard
Winter Park 32792
Tel: (305) 628-3888
TWX: 810-853-0322

†Pioneer/Technologies Group, Inc.
337 S. Lake Blvd.
Alta Monte Springs 32701
Tel: (407) 834-9090
TWX: 810-853-0284

Pioneer/Technologies Group, Inc.
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
TWX: 510-955-9653

GEORGIA

†Arrow Electronics, Inc.
3155 Northwoods Parkway
Suite A
Norcross 30071
Tel: (404) 449-8252
TWX: 810-766-0439

†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
TWX: 810-766-4515

ILLINOIS

Arrow Electronics, Inc.
1140 W. Thorndale
Itasca 60143
Tel: (312) 250-0500
TWX: 312-250-0916

†Hamilton/Avnet Electronics
1130 Thorndale Avenue
Bensenville 60106
Tel: (312) 860-7780
TWX: 910-227-0060

MTI Systems Sales
1100 W. Thorndale
Itasca 60143
Tel: (312) 773-2300

†Pioneer Electronics
1551 Carmen Drive
Elk Grove Village 60007
Tel: (312) 437-9680
TWX: 910-222-1834

INDIANA

†Arrow Electronics, Inc.
2495 Directors Row, Suite H
Indianapolis 46241
Tel: (317) 243-9353
TWX: 810-341-3119

Hamilton/Avnet Electronics
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
TWX: 810-260-3966

†Pioneer Electronics
6408 Castleplace Drive
Indianapolis 46250
Tel: (317) 849-7300
TWX: 810-260-1794

IOWA

Hamilton/Avnet Electronics
915 33rd Avenue, S.W.
Cedar Rapids 52404
Tel: (319) 362-4757

KANSAS

Arrow Electronics
8208 Melrose Dr., Suite 210
Lenexa 66214
Tel: (913) 541-9542

†Hamilton/Avnet Electronics
9219 Quivera Road
Overland Park 66215
Tel: (913) 888-8900
TWX: 910-743-0005

Pioneer/Tec Gr.
10551 Lockman Rd.
Lenexa 66215
Tel: (913) 492-0500

KENTUCKY

Hamilton/Avnet Electronics
1051 D. Newton Park
Lexington 40511
Tel: (606) 259-1475

MARYLAND

Arrow Electronics, Inc.
8300 Guilford Drive
Suite H, River Center
Columbia 21046
Tel: (301) 995-0003
TWX: 710-236-9005

Hamilton/Avnet Electronics
6822 Oak Hall Lane
Columbia 21045
Tel: (301) 995-3500
TWX: 710-862-1861

†Mesa Technology Corp.
9720 Patuxent Woods Dr.
Columbia 21046
Tel: (301) 290-8150
TWX: 710-828-9702

†Pioneer/Technologies Group, Inc.
9100 Gaither Road
Gaithersburg 20877
Tel: (301) 921-0660
TWX: 710-828-0545

MASSACHUSETTS

Arrow Electronics, Inc.
25 Upton Dr.
Wilmington 01887
Tel: (617) 935-5134

†Hamilton/Avnet Electronics
10D Centennial Drive
Peabody 01960
Tel: (617) 531-7430
TWX: 710-393-0382

MTI Systems Sales
83 Cambridge St.
Burlington 01813

Pioneer Electronics
44 Hartwell Avenue
Lexington 02173
Tel: (617) 861-9200
TWX: 710-326-6617

MICHIGAN

Arrow Electronics, Inc.
755 Phoenix Drive
Ann Arbor 48104
Tel: (313) 971-8220
TWX: 810-223-6020

Hamilton/Avnet Electronics
2215 29th Street S.E.
Space A5
Grand Rapids 49508
Tel: (616) 243-8805
TWX: 810-274-6921

Pioneer Electronics
4504 Broadmoor S.E.
Grand Rapids 49508
FAX: 616-698-1831

†Hamilton/Avnet Electronics
32487 Schoolcraft Road
Livonia 48150
Tel: (313) 522-4700
TWX: 810-282-8775

†Pioneer/Michigan
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
TWX: 810-242-3271

MINNESOTA

†Arrow Electronics, Inc.
5230 W. 73rd Street
Edina 55435
Tel: (612) 830-1800
TWX: 910-576-3125

†Hamilton/Avnet Electronics
12400 Whitewater Drive
Minnetonka 55434
Tel: (612) 932-0600

†Pioneer Electronics
7625 Golden Triange Dr.
Suite G
Eden Prairi 55343
Tel: (612) 944-3355

MISSOURI

†Arrow Electronics, Inc.
2380 Schuetz
St. Louis 63141
Tel: (314) 567-6888
TWX: 910-764-0882

†Hamilton/Avnet Electronics
13743 Shoreline Court
Earth City 63045
Tel: (314) 344-1200
TWX: 910-762-0684

NEW HAMPSHIRE

†Arrow Electronics, Inc.
3 Perimeter Road
Manchester 03103
Tel: (603) 668-6968
TWX: 710-220-1684

†Hamilton/Avnet Electronics
444 E. Industrial Drive
Manchester 03103
Tel: (603) 624-9400

NEW JERSEY

†Arrow Electronics, Inc.
Four East Stow Road
Unit 11
Marlton 08053
Tel: (609) 596-8000
TWX: 710-897-0829

†Arrow Electronics
6 Century Drive
Parsipanny 07054
Tel: (201) 538-0900

†Hamilton/Avnet Electronics
1 Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
TWX: 710-940-0262

†Hamilton/Avnet Electronics
10 Industrial
Fairfield 07006
Tel: (201) 575-5300
TWX: 710-734-4388

†MTI Systems Sales
37 Kulick Rd.
Fairfield 07006
Tel: (201) 227-5552

†Pioneer Electronics
45 Route 46
Pinebrook 07058
Tel: (201) 575-3510
TWX: 710-734-4382

NEW MEXICO

Alliance Electronics Inc.
11030 Cochiti S.E.
Albuquerque 87123
Tel: (505) 292-3360
TWX: 910-989-1151

Hamilton/Avnet Electronics
2524 Baylor Drive S.E.
Albuquerque 87106
Tel: (505) 765-1500
TWX: 910-989-0614

NEW YORK

†Arrow Electronics, Inc.
3375 Brighton Henrietta Townline Rd.
Rochester 14623
Tel: (716) 275-0300
TWX: 510-253-4766

Arrow Electronics, Inc.
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
TWX: 510-227-6623

Hamilton/Avnet
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
TWX: 510-224-6166

†Hamilton/Avnet Electronics
333 Metro Park
Rochester 14623
Tel: (716) 475-9130
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

†Pioneer Electronics
68 Corporate Drive
Binghamton 13904
Tel: (607) 722-9300
TWX: 510-252-0893

Pioneer Electronics
40 Oser Avenue
Hauppauge 11787
Tel: (516) 231-9200

†Microcomputer System Technical Distributor Center

DOMESTIC DISTRIBUTORS (Cont'd.)

NEW YORK (Cont'd.)

†Pioneer Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
TWX: 510-221-2184

†Pioneer Electronics
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
TWX: 510-253-7001

NORTH CAROLINA

†Arrow Electronics, Inc.
5240 Greensdairy Road
Raleigh 27604
Tel: (919) 876-3132
TWX: 510-928-1856

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819
TWX: 510-928-1836

Pioneer/Technologies Group, Inc.
9801 A-Southern Pine Blvd.
Charlotte 28210
Tel: (919) 527-8188
TWX: 810-621-0366

OHIO

Arrow Electronics, Inc.
7620 McEwen Road
Centerville 45459
Tel: (513) 435-5563
TWX: 810-459-1611

†Arrow Electronics, Inc.
6238 Cochran Road
Solon 44139
Tel: (216) 248-3990
TWX: 810-427-9409

†Hamilton/Avnet Electronics
954 Senate Drive
Dayton 45459
Tel: (513) 439-6733
TWX: 810-450-2531

Hamilton/Avnet Electronics
4588 Emery Industrial Pkwy.
Warrensville Heights 44128
Tel: (216) 349-5100
TWX: 810-427-9452

†Hamilton/Avnet Electronics
777 Brooksedge Blvd.
Westerville 43081
Tel: (614) 882-7004

†Pioneer Electronics
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
TWX: 810-459-1622

†Pioneer Electronics
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
TWX: 810-422-2211

OKLAHOMA

Arrow Electronics, Inc.
1211 E. 51st Street
Suite 101
Tulsa 74146
Tel: (918) 252-7537

†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
TWX: 910-467-8746

†Hamilton/Avnet Electronics
6024 S.W. Jean Road
Bldg. C, Suite 10
Lake Oswego 97034
Tel: (503) 635-7848
TWX: 910-455-8179

Wyle Distribution Group
5250 N.E. Elam Young Parkway
Suite 600
Hillsboro 97124
Tel: (503) 640-6000
TWX: 910-460-2203

PENNSYLVANIA

Arrow Electronics, Inc.
650 Seco Road
Monroeville 15146
Tel: (412) 856-7000

Hamilton/Avnet Electronics
2800 Liberty Ave.
Pittsburgh 15238
Tel: (412) 281-4150

Pioneer Electronics
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
TWX: 710-795-3122

†Pioneer/Technologies Group, Inc.
Delaware Valley
261 Gibralter Road
Horsham 19044
Tel: (215) 674-4000
TWX: 510-665-6778

TEXAS

†Arrow Electronics, Inc.
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
TWX: 910-860-5377

†Arrow Electronics, Inc.
10899 Kinghurst
Suite 100
Houston 77099
Tel: (713) 530-4700
TWX: 910-880-4439

†Arrow Electronics, Inc.
2227 W. Braker Lane
Austin 78758
Tel: (512) 835-4180
TWX: 910-874-1348

†Hamilton/Avnet Electronics
1807 W. Braker Lane
Austin 78758
Tel: (512) 837-8911
TWX: 910-874-1319

TEXAS (Cont'd.)

†Hamilton/Avnet Electronics
2111 W. Walnut Hill Lane
Irving 75038
Tel: (214) 550-6111
TWX: 910-860-5929

†Hamilton/Avnet Electronics
4850 Wright Rd., Suite 190
Stafford 77477
Tel: (713) 240-7733
TWX: 910-881-5523

†Pioneer Electronics
18260 Kramer
Austin 78758
Tel: (512) 835-4000
TWX: 910-874-1323

†Pioneer Electronics
13710 Omega Road
Dallas 75234
Tel: (214) 386-7300
TWX: 910-850-5563

†Pioneer Electronics
5853 Point West Drive
Houston 77036
Tel: (713) 988-5555
TWX: 910-881-1606

Wyle Distribution Group
1810 Greenville Avenue
Richardson 75081
Tel: (214) 235-9953

UTAH

Arrow Electronics
1946 Parkway Blvd.
Salt Lake City 84119
Tel: (801) 973-6913

†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
TWX: 910-444-2067

Arrow Electronics, Inc.
19540 68th Ave. South
Kent 98032
Tel: (206) 575-4420

†Hamilton/Avnet Electronics
14212 N.E. 21st Street
Bellevue 98005
Tel: (206) 643-3950
TWX: 910-443-2469

Wyle Distribution Group
15385 N.E. 90th Street
Redmond 98052
Tel: (206) 881-1150

WISCONSIN

Arrow Electronics, Inc.
200 N. Patrick Blvd., Ste. 100
Brookfield 53005
Tel: (414) 767-6600
TWX: 910-262-1193

Hamilton/Avnet Electronics
2975 Moorland Road
New Berlin 53151
Tel: (414) 784-4510
TWX: 910-262-1182

CANADA

ALBERTA

Hamilton/Avnet Electronics
2816 21st Street N.E.
Calgary T2E 6Z3
Tel: (403) 230-3586
TWX: 03-827-642

Zentronics
Bay No. 1
3300 14th Avenue N.E.
Calgary T2A 6J4
Tel: (403) 272-1021

BRITISH COLUMBIA

†Hamilton/Avnet Electronics
105-2550 Boundary
Burmalay V5M 3Z3
Tel: (604) 437-6667

Zentronics
108-11400 Bridgeport Road
Richmond V6X 1T2
Tel: (604) 273-5575
TWX: 04-5077-89

MANITOBA

Zentronics
60-1313 Border Unit 60
Winnipeg R3H 0X4
Tel: (204) 694-1957

ONTARIO

Arrow Electronics, Inc.
36 Antares Dr.
Nepean K2E 7W5
Tel: (613) 226-6903

Arrow Electronics, Inc.
1093 Meyerside
Mississauga L5T 1M4
Tel: (416) 673-7769
TWX: 06-218213

†Hamilton/Avnet Electronics
6845 Rexwood Road
Units 3-4-5
Mississauga L4T 1R2
Tel: (416) 677-7432
TWX: 610-492-8867

Hamilton/Avnet Electronics
6845 Rexwood Road
Unit 6
Mississauga L4T 1R2
Tel: (416) 277-0484

ONTARIO (Cont'd.)

†Hamilton/Avnet Electronics
190 Colonnade Road South
Nepean K2E 7L5
Tel: (613) 226-1700
TWX: 05-349-71

†Zentronics
8 Tilbury Court
Brampton L6T 3T4
Tel: (416) 451-9600
TWX: 06-976-78

†Zentronics
155 Colonnade Road
Unit 17
Nepean K2E 7K1
Tel: (613) 226-8840

Zentronics
60-1313 Border St.
Winnipeg R3H 0l4
Tel: (204) 694-7957

QUEBEC

†Arrow Electronics Inc.
4050 Jean Talon Quest
Montreal H4P 1W1
Tel: (514) 735-5511
TWX: 05-25590

Arrow Electronics, Inc.
909 Charest Blvd.
Quebec J1N 2C9
Tel: (418) 687-4231
TWX: 05-13388

Hamilton/Avnet Electronics
2795 Halpern
St. Laurent H2E 7K1
Tel: (514) 335-1000
TWX: 610-421-3731

Zentronics
817 McCaffrey
St. Laurent H4T 1M3
Tel: (514) 737-9700
TWX: 05-827-535

†Microcomputer System Technical Distributor Center

EUROPEAN SALES OFFICES

DENMARK

Intel Denmark A/S
Glentevej 61, 3rd Floor
2400 Copenhagen NV
Tel: (45) (01) 19 80 33
TLX: 19567

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

Intel Corporation S.A.R.L.
4, Quai des Etroits
69321 Lyon Cedex 05
Tel: (33) (16) 78 42 40 89
TLX: 305153

WEST GERMANY

Intel Semiconductor GmbH*
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen
Tel: (49) 089/90992-0
TLX: 5-23177
FAX: 904-3948

Intel Semiconductor GmbH
Hohenzollern Strasse 5
3000 Hannover 1
Tel: (49) 0511/344081
TLX: 9-23625

Intel Semiconductor GmbH
Abraham Lincoln Strasse 16-18
6200 Wiesbaden
Tel: (49) 06121/7605-0
TLX: 4-186183

Intel Semiconductor GmbH
Zettachring 10A
7000 Stuttgart 80
Tel: (49) 0711/728728-0
TLX: 7-254826

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
20090 Assago
Milano
Tel: (39) (02) 824 40 71
TLX: 341286

NETHERLANDS

Intel Semiconductor B.V.*
Marten Meesweg 93
3068 AV Rotterdam
Tel: (31) 10.407.11.11
TLX: 22283

NORWAY

Intel Norway A/S
Hvamveien 4-PO Box 92
2013 Skjetten
Tel: (47) (6) 842 420
TLX: 78018

SPAIN

Intel Iberia S.A.
Zurbaran, 28
28010 Madrid
Tel: (34) 410 40 04
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

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Electronics G.m.b.H.
Rotenmuehlgasse 26
1120 Wien
Tel: (43) (0222) 83 56 46-0
TLX: 131532

BELGIUM

Inelco Belgium S.A.
Av. des Croix de Guerre 94
1120 Bruxelles
Oorlogskruisenlaan, 94
1120 Brussel
Tel: (32) (02) 216 01 60
TLX: 64475

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

Generim
Z.A. de Courtaboeuf
Av. de la Baltique-BP 88
91943 Les Ulis Cedex
Tel: (33) (1) 69 07 78 78
TLX: 691700

Jermyn
73-79, rue des Solets
Silic 585
94663 Rungis Cedex
Tel: (33) (1) 49 78 49 00
TLX: 260967

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
92315 Sevres Cedex
Tel: (33) (1) 45 34 75 35
TLX: 204552

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) 06103/3040
TLX: 417903

IRELAND

Micro Marketing Ltd.
Glenageary Office Park
Glenageary
Co. Dublin
Tel: (21) (353) (01) 85 63 25
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
Milano
Tel: (39) 02/824701
TLX: 311351

Lasi Elettronica S.p.A.
V. le Fulvio Testi, 126
20092 Cinisello Balsamo
Milano
Tel: (39) 02/2440012
TLX: 352040

NETHERLANDS

Koning en Hartman
1 Energieweg
2627 AP Delft
Tel: (31) 15609906
TLX: 38250

NORWAY

Nordisk Elektronikk (Norge) A/S
Postboks 123
Smedsvingen 4
1364 Hvalstad
Tel: (47) (02) 84 62 10
TLX: 77546

PORTUGAL

Ditram
Avenida Miguel Bombarda, 133
1000 Lisboa
Tel: (351) (1) 734 884
TLX: 14182

SPAIN

ATD Electronica, S.A.
Plaza Ciudad de Viena, 6
28040 Madrid
Tel: (34) (1) 234 40 00
TLX: 42754

ITT-SESA
Calle Miguel Angel, 21-3
28010 Madrid
Tel: (34) (1) 419 54 00
TLX: 27461

SWEDEN

Nordisk Elektronik AB
Huvudstagatan 1
Box 1409
171 27 Solna
Tel: (46) 08-734 97 70
TLX: 105 47

SWITZERLAND

Industrade A.G.
Hertistrasse 31
8304 Wallisellen
Tel: (41) (801) 83 05 04 0
TLX: 56788

TURKEY

EMPA Electronic
Lindwurmstrasse 95A
8000 Muenchen 2
Tel: (49) 089/53 80 570
TLX: 528573

UNITED KINGDOM

Accent Electronic Components Ltd.
Jubilee House, Jubilee Road
Letchworth, Herts SG6 1TL
Tel: (44) (0462) 686666
TLX: 826293

Bytech-Comway Systems
3 The Western Centre
Western Road
Bracknell RG12 1RW
Tel: (44) (0344) 55333
TLX: 847201

Jermyn
Vestry Estate
Otford Road
Sevenoaks
Kent TN14 5EU
Tel: (44) (0732) 450144
TLX: 95142

MMD
Unit 8 Southview Park
Caversham
Reading
Berkshire RG4 0AF
Tel: (44) (0734) 48 16 66
TLX: 846669

Rapid Silicon
Rapid House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (44) (0494) 442266
TLX: 837931

Rapid Systems
Rapid House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (44) (0494) 450244
TLX: 837931

YUGOSLAVIA

Rapido Electronic Components S.p.a.
Via C. Beccaria, 8
34133 Trieste
Italia
Tel: (39) 040/360555
TLX: 460461

*Field Application Location

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.*
Spectrum Building
200 Pacific Hwy., Level 6
Crows Nest, NSE, 2065
Tel: 612-957-2744
FAX: 612-923-2632

BRAZIL

Intel Semicondutores do Brazil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.
Tel: 55-11-287-5899
TLX: 3911153146 ISDB
FAX: 55-11-287-5899

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: (5) 8444-555
TLX: 63869 ISHLHK HX
FAX: (5) 8681-989

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001
Tel: 91-812-567201
TLX: 9538452354 MACH
FAX: 091-812-563982

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26
Tel: 029747-8511
TLX: 3656-160
FAX: 029747-8450

Intel Japan K.K.*
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183
Tel: 0423-60-7871
FAX: 0423-60-0315

Intel Japan K.K.*
Flower-Hill Shin-machi Bldg.
1-23-9 Shinmachi
Setagaya-ku, Tokyo 154
Tel: 03-426-2231
FAX: 03-427-7620

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360
Tel: 0485-24-6871
FAX: 0485-24-7518

JAPAN (Cont'd.)

Intel Japan K.K.*
Mitsui-Seimei Musashi-kosugi Bldg.
915 Shinmaruko, Nakahara-ku
Kawasaki-shi, Kanagawa 211
Tel: 044-733-7011
FAX: 044-733-7010

Intel Japan K.K.
Nihon Seimei Atsugi Bldg.
1-2-1 Asahi-machi
Atsugi-shi, Kanagawa 243
Tel: 0462-29-3731
FAX: 0462-29-3781

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: 03-201-3621
FAX: 03-201-6850

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 Technology Asia, Ltd.
Business Center 16th Floor
61, Yoido-Dong, Young Deung Po-Ku
Seoul 150
Tel: (2) 784-8186, 8286, 8386
TLX: K29312 INTELKO
FAX: (2) 784-8096

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #21-06
Goldhill Square
Singapore 1130
Tel: 250-7811
TLX: 39921 INTEL
FAX: 250-9256

TAIWAN

Intel Technology Far East Ltd.
Taiwan Branch
10/F, No. 205, Tun Hua N. Road
Taipei, R.O.C.
Tel: 886-2-716-9660
TLX: 13159 INTELTWN
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

DAFSYS S.R.L.
Chacabuco, 90-6 PISO
1069-Buenos Aires
Tel: 54-1-334-7726
FAX: 54-1-334-1871

AUSTRALIA

Email Electronics
15-17 Hume Street
Huntingdale, 3166
Tel: 011-61-3-544-8244
TLX: AA 30895
FAX: 011-61-3-543-8179

BRAZIL

Elebra Microelectronica S.A.
Rua Geraldo Flausina Gomes, 78
10th Floor
04575 - Sao Paulo - S.P.
Tel: 55-11-534-9641
TLX: 55-11-54593/54591
FAX: 55-11-534-9424

CHILE

DIN Instruments
Suecia 2323
Casilla 6055, Correo 22
Santiago
Tel: 56-2-225-8139
TLX: 240.846 RUD

CHINA/HONG KONG

Novel Precision Machinery Co., Ltd.
Flat D, 20 Kingsford Ind. Bldg.
Phase 1, 26 Kwai Hei Street
N.T., Kowloon
Hong Kong
Tel: 852-0-223-222
TWX: 39114 JINMI HX
FAX: 852-0-261-602

INDIA

Micronic Devices
Arun Complex
No. 65 D.V.G. Road
Basavanagudi
Bangalore 560 004
Tel: 011-91-812-600-631
011-91-812-621-455
TLX: 9538458332 MDBG

Micronic Devices
Flat 403, Gagan Deep
12, Rajendra Place
New Delhi 110 008
Tel: 011-91-58-97-71
011-91-57-23509
TLX: 9533163235 MDND

Micronic Devices
No. 516 5th Floor
Swastik Chambers
Sion, Trombay Road
Chembur
Bombay 400 071
Tel: 011-91523963/527896
TLX: 9531 171447 MDEV

S&S Corporation
Camden Business Center
Suite 6
1610 Blossom Hill Rd.
San Jose, CA 95124
U.S.A.
Tel: (408) 978-6216
TLX: 820281

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. 2-14-1 Asano
Kokurakita-ku
Kitakyushu-shi 802
Tel: 093-511-6471
FAX: 093-551-7861

C. Itoh Techno-Science Co., Ltd.
4-8-1 Dobashi, Miyamae-ku
Kawasaki-shi, Kanagawa 213
Tel: 044-852-5121
FAX: 044-877-4268

JAPAN (Cont'd.)

Dia Semicon Systems, Inc.
Wacore 64, 1-37-8 Sangenjaya
Setagaya-ku, Tokyo 154
Tel: 03-487-0386
FAX: 03-487-8088

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: 03-546-5011
FAX: 03-546-5044

KOREA

J-Tek Corporation
6th Floor, Government Pension Bldg.
24-3 Yoido-Dong
Youngdeungpo-ku
Seoul 150
Tel: 82-2-782-8039
TLX: 25299 KODIGIT
FAX: 82-2-784-8391

Samsung Semiconductor &
Telecommunications Co., Ltd.
150, 2-KA, Tafpyung-ro, Chung-ku
Seoul 100
Tel: 82-2-751-3987
TLX: 27970 KORSST
FAX: 82-2-753-0967

MEXICO

Dicopel S.A.
Av. Federalismo Sur
268-2-PLSO
C.P. 44-100-Guadalajara
Tel: 52-36-26-1232
TLX: 681663 DICOME
FAX: 52-36-26-3966

Dicopel S.A.
Tochtli 368 Fracc. Ind. San Antonio
Azcapotzalco
C.P. 02760-Mexico, D.F.
Tel: 52-5-561-3211
TLX: 1773790 DICOME
FAX: 52-5-561-1279

NEW ZEALAND

Switch Enterprises
36 Olive Road
Penrose, Auckland
Tel: 011-64-9-591155
FAX: 64-9-592681

SINGAPORE

Electronic Resources Pte, Ltd.
17 Harvey Road #04-01
Singapore 1336
Tel: 283-0888, 289-1618
TWX: 56541 FRELS
FAX: 2895327

SOUTH AFRICA

Electronic Building Elements
178 Erasmus Street
Meyerspark, Pretoria, 0184
Tel: 011-2712-803-7680
FAX: 011-2712-803-8290

TAIWAN

Micro Electronics Corporation
No. 585, Ming Shen East Rd.
Taipei, R.O.C.
Tel: 886-2-501-8231
FAX: 886-2-501-4265

Sertek
5FL, 135 Sec. 2
Chien-Kuo N. Rd.
Taipei 10479
R.O.C.
Tel: (02) 5010055
FAX: (02) 5012521
(02) 5058414

VENEZUELA

P. Benavides S.A.
Avilanes a Rio
Residencia Kamarata
Locales 4 AL 7
La Candelaria, Caracas
Tel: 58-2-574-6338
TLX: 28450
FAX: 58-2-572-3321

*Field Application Location

intel

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

ISBN 1-55512-080-6

Order Number: 240329-002

Printed in U.S.A./SMD309/2K/0289/ML RJ
Microprocessors

©Intel Corporation, 1989