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

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

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

### CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information.

<table>
<thead>
<tr>
<th>TITLE</th>
<th>LITERATURE ORDER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET OF 11 HANDBOOKS (Available in U.S. and Canada only)</td>
<td>231003</td>
</tr>
<tr>
<td>EMBEDDED APPLICATIONS</td>
<td>270648</td>
</tr>
<tr>
<td>8-BIT EMBEDDED CONTROLLERS</td>
<td>270645</td>
</tr>
<tr>
<td>16-BIT EMBEDDED CONTROLLERS</td>
<td>270646</td>
</tr>
<tr>
<td>16/32-BIT EMBEDDED PROCESSORS</td>
<td>270647</td>
</tr>
<tr>
<td>MEMORY</td>
<td>210830</td>
</tr>
<tr>
<td>MICROCOMMUNICATIONS (2 volume set)</td>
<td>231658</td>
</tr>
<tr>
<td>MICROCOMPUTER SYSTEMS</td>
<td>280407</td>
</tr>
<tr>
<td>MICROPROCESSORS</td>
<td>230843</td>
</tr>
<tr>
<td>PERIPHERALS</td>
<td>296467</td>
</tr>
<tr>
<td>PRODUCT GUIDE (Overview of Intel's complete product lines)</td>
<td>210846</td>
</tr>
<tr>
<td>PROGRAMMABLE LOGIC</td>
<td>296083</td>
</tr>
</tbody>
</table>

### ADDITIONAL LITERATURE

(Not included in handbook set)

<table>
<thead>
<tr>
<th>TITLE</th>
<th>LITERATURE ORDER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMOTIVE SUPPLEMENT</td>
<td>231792</td>
</tr>
<tr>
<td>COMPONENTS QUALITY/RELIABILITY HANDBOOK</td>
<td>210997</td>
</tr>
<tr>
<td>INTEL PACKAGING OUTLINES AND DIMENSIONS</td>
<td>231369</td>
</tr>
<tr>
<td>(Packaging types, number of leads, etc.)</td>
<td></td>
</tr>
<tr>
<td>INTERNATIONAL LITERATURE GUIDE</td>
<td>E00029</td>
</tr>
<tr>
<td>LITERATURE PRICE LIST (U.S. and Canada)</td>
<td>210620</td>
</tr>
<tr>
<td>(Comprehensive list of current Intel Literature)</td>
<td></td>
</tr>
<tr>
<td>MILITARY</td>
<td>210461</td>
</tr>
<tr>
<td>(2 volume set)</td>
<td></td>
</tr>
<tr>
<td>SYSTEMS QUALITY/RELIABILITY</td>
<td>231762</td>
</tr>
</tbody>
</table>
# U.S. and CANADA LITERATURE ORDER FORM

**NAME:** __________________________________________________________

**COMPANY:** ______________________________________________________

**ADDRESS:** ______________________________________________________

**CITY:** _________________________________ **STATE:** ______ **ZIP:** ______

**COUNTRY:** _______________________________________________________

**PHONE NO.:** ______________________

<table>
<thead>
<tr>
<th>ORDER NO.</th>
<th>TITLE</th>
<th>QTY.</th>
<th>PRICE</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtotal __________

Must Add Your Local Sales Tax __________

Postage: add 10% of subtotal

Postage __________

Total __________

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

- VISA
- MasterCard
- American Express

Expiration Date __________________________

**Account No.** _______________________________________________________

**Signature** _______________________________________________________

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

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

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

Prices good until 12/31/90.

**Source HB**
INTERNATIONAL LITERATURE ORDER FORM

NAME: ____________________________________________

COMPANY: ________________________________________________________

ADDRESS: _________________________________________________________

CITY: ________________________ STATE: _______ ZIP: ______

COUNTRY: _________________________________________________________

PHONE NO.: (____) 

<table>
<thead>
<tr>
<th>ORDER NO.</th>
<th>TITLE</th>
<th>QTY.</th>
<th>PRICE</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtotal __________
Must Add Your Local Sales Tax __________
Total __________

PAYMENT

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

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

The completed form should be marked to the attention of the LITERATURE COORDINATOR and returned to your local Intel Sales Office.
Intel 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, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE196, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, DVI, ETOX, FaxBACK, Genius, i, i, i486, i750, i860, ICE, ICEL, ICEVIEW, ICS, IDBP, IDIS, i, ICE, iLBX, iMDDX, iMMX, Inboard, Insate, Intel, inti, Intelx86, intiB0S, Intel Certified, Intelevision, intiElgent Identifier, intiElgent Programming, Intellec, Intellink, IOSP, IPAT, IPDS, IPSC, iRMK, iRMX, iSB3, IS3B, IS3M, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PRO750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSICEL, and ZapCode, and the combination of ICE, ICS, iRMX, ISBC, iSB3, ISXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PRO750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSICEL, and ZapCode, and the combination of ICE, ICS, iRMX, ISBC, iSB3, ISXM, Library Manager, MAPNET, 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.

OS/2 is a trademark of International Business Machines Corporation.

UNIX is a registered trademark of AT&T.

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

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

The Intel i860™ Microprocessor (part number 80860XR) delivers supercomputer 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. This chapter 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™ 64-Bit Microprocessor Assembler and Linker Reference Manual, order number 240436
• i860™ 64-Bit 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, 0xF is a hexadecimal number that is equivalent to decimal 15.

RESERVED BITS AND SOFTWARE COMPATIBILITY

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

• Do not depend on the states of any reserved or undefined bits when testing the values of registers that contain such bits. Mask out the reserved and undefined bits before testing.

• Do not depend on the states of any reserved or undefined bits when storing them in memory or in another register.
• Do not depend on the ability to retain information written into any reserved or undefined bits.

• When loading a control register, always load the reserved and undefined bits with values previously retrieved 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.**
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 to economical carry-in or mail-in factory service.

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

SOFTWARE SUPPORT SERVICES

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

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

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.
# TABLE OF CONTENTS

## CHAPTER 1

**ARCHITECTURAL OVERVIEW**

1.1 OVERVIEW ................................................................. 1-1
1.2 INTEGER CORE UNIT ................................................... 1-3
1.3 FLOATING-POINT UNIT ............................................... 1-4
1.4 GRAPHICS UNIT ....................................................... 1-4
1.5 MEMORY MANAGEMENT UNIT ...................................... 1-5
1.6 CACHES ........................................................................ 1-6
1.7 PARALLEL ARCHITECTURE ......................................... 1-6
1.8 SOFTWARE DEVELOPMENT ENVIRONMENT .................... 1-7
1.8.1 Multiprocessing for High-Performance with Compatibility ........................................ 1-7

## 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 .......................................... 2-3

## CHAPTER 3

**REGISTERS**

3.1 INTEGER REGISTER FILE ........................................... 3-2
3.2 FLOATING-POINT REGISTER FILE ............................... 3-2
3.3 PROCESSOR STATUS REGISTER ................................. 3-2
3.4 EXTENDED PROCESSOR STATUS REGISTER .................. 3-4
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 .................................................................. 4-1
4.2 VIRTUAL ADDRESSING ............................................... 4-2
4.2.1 Page Frame .......................................................... 4-3
4.2.2 Virtual Address ..................................................... 4-3
4.2.3 Page Tables .......................................................... 4-3
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 .................................. 4-6
4.2.4.4 WRITE-THROUGH BIT .................................. 4-6
4.2.4.5 ACCESSED AND DIRTY BITS ......................... 4-6
4.2.4.6 WRITABLE AND USER BITS ......................... 4-7
4.2.4.7 COMBINING PROTECTION OF BOTH LEVELS OF PAGE TABLES 4-8
4.2.5 Address Translation Algorithm ............................ 4-8
4.2.6 Address Translation Faults .................................... 4-9
4.2.7 Page Translation Cache ........................................ 4-9
4.3 CACHING AND CACHE FLUSHING .............................. 4-10
# Table of Contents

## Chapter 5: Core Instructions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Instructions</td>
<td></td>
</tr>
<tr>
<td>5.1 Load Integer</td>
<td>5-3</td>
</tr>
<tr>
<td>5.2 Store Integer</td>
<td>5-4</td>
</tr>
<tr>
<td>5.3 Transfer Integer to F-P Register</td>
<td>5-5</td>
</tr>
<tr>
<td>5.4 Load Floating-Point</td>
<td>5-6</td>
</tr>
<tr>
<td>5.5 Store Floating-Point</td>
<td>5-8</td>
</tr>
<tr>
<td>5.6 Pixel Store</td>
<td>5-9</td>
</tr>
<tr>
<td>5.7 Integer Add and Subtract</td>
<td>5-10</td>
</tr>
<tr>
<td>5.8 Shift Instructions</td>
<td>5-12</td>
</tr>
<tr>
<td>5.9 Software Traps</td>
<td>5-13</td>
</tr>
<tr>
<td>5.10 Logical Instructions</td>
<td>5-14</td>
</tr>
<tr>
<td>5.11 Control-Transfer Instructions</td>
<td>5-16</td>
</tr>
<tr>
<td>5.12 Control Register Access</td>
<td>5-20</td>
</tr>
<tr>
<td>5.13 Cache Flush</td>
<td>5-21</td>
</tr>
<tr>
<td>5.14 Bus Lock</td>
<td>5-23</td>
</tr>
</tbody>
</table>

## Chapter 6: Floating-Point Instructions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-Point Instructions</td>
<td></td>
</tr>
<tr>
<td>6.1 Precision Specification</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2 Pipelined and Scalar Operations</td>
<td>6-2</td>
</tr>
<tr>
<td>6.2.1 Scalar Mode</td>
<td>6-4</td>
</tr>
<tr>
<td>6.2.2 Pipelining Status Information</td>
<td>6-4</td>
</tr>
<tr>
<td>6.2.3 Precision in the Pipelines</td>
<td>6-4</td>
</tr>
<tr>
<td>6.2.4 Transition between Scalar and Pipelined Operations</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3 Multiplier Instructions</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3.1 Floating-Point Multiply</td>
<td>6-7</td>
</tr>
<tr>
<td>6.3.2 Floating-Point Multiply Low</td>
<td>6-8</td>
</tr>
<tr>
<td>6.3.3 Floating-Point Reciprocals</td>
<td>6-9</td>
</tr>
<tr>
<td>6.4 Adder Instructions</td>
<td>6-9</td>
</tr>
<tr>
<td>6.4.1 Floating-Point Add and Subtract</td>
<td>6-10</td>
</tr>
<tr>
<td>6.4.2 Floating-Point Compares</td>
<td>6-12</td>
</tr>
<tr>
<td>6.4.3 Floating-Point to Integer Conversion</td>
<td>6-13</td>
</tr>
<tr>
<td>6.5 Dual Operation Instructions</td>
<td>6-14</td>
</tr>
<tr>
<td>6.6 Graphics Unit</td>
<td>6-26</td>
</tr>
<tr>
<td>6.6.1 Long-Integer Arithmetic</td>
<td>6-28</td>
</tr>
<tr>
<td>6.6.2 3-D Graphics Operations</td>
<td>6-28</td>
</tr>
<tr>
<td>6.6.2.1 Z-Buffer Check Instructions</td>
<td>6-29</td>
</tr>
<tr>
<td>6.6.2.2 Pixel Add</td>
<td>6-32</td>
</tr>
<tr>
<td>6.6.2.3 Z-Buffer Add</td>
<td>6-36</td>
</tr>
<tr>
<td>6.6.2.4 Or with Merge Register</td>
<td>6-38</td>
</tr>
<tr>
<td>6.7 Transfer F-P to Integer Register</td>
<td>6-39</td>
</tr>
<tr>
<td>6.8 Dual-Instruction Mode</td>
<td>6-40</td>
</tr>
<tr>
<td>6.8.1 Core and Floating-Point Instruction Interaction</td>
<td>6-41</td>
</tr>
<tr>
<td>6.8.2 Dual-Instruction Mode Restrictions</td>
<td>6-42</td>
</tr>
</tbody>
</table>

## Chapter 7: Traps and Interrupts

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traps and Interrupts</td>
<td></td>
</tr>
<tr>
<td>7.1 Types of Traps</td>
<td>7-1</td>
</tr>
<tr>
<td>7.2 Trap Handler Invocation</td>
<td>7-1</td>
</tr>
<tr>
<td>7.2.1 Saving State</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2.2 Inside the Trap Handler</td>
<td>7-3</td>
</tr>
<tr>
<td>7.2.3 Returning from the Trap Handler</td>
<td>7-3</td>
</tr>
<tr>
<td>7.2.3.1 Determining Where to Resume</td>
<td>7-4</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

APPENDIX A
INSTRUCTION SET SUMMARY

APPENDIX B
INSTRUCTION FORMAT AND ENCODING

APPENDIX C
INSTRUCTION TIMINGS

APPENDIX D
INSTRUCTION CHARACTERISTICS

Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Registers and Data Paths</td>
<td>1-2</td>
</tr>
<tr>
<td>2-1</td>
<td>Pixel Format Examples</td>
<td>2-4</td>
</tr>
<tr>
<td>3-1</td>
<td>Register Set</td>
<td>3-1</td>
</tr>
<tr>
<td>3-2</td>
<td>Processor Status Register</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>Extended Processor Status Register</td>
<td>3-5</td>
</tr>
<tr>
<td>3-4</td>
<td>Directory Base Register</td>
<td>3-6</td>
</tr>
<tr>
<td>3-5</td>
<td>Floating-Point Status Register</td>
<td>3-9</td>
</tr>
<tr>
<td>4-1</td>
<td>Memory Formats</td>
<td>4-1</td>
</tr>
<tr>
<td>4-2</td>
<td>Big and Little Endian Memory Transfers</td>
<td>4-2</td>
</tr>
<tr>
<td>4-3</td>
<td>Format of a Virtual Address</td>
<td>4-3</td>
</tr>
<tr>
<td>4-4</td>
<td>Address Translation</td>
<td>4-4</td>
</tr>
<tr>
<td>4-5</td>
<td>Format of a Page Table Entry</td>
<td>4-5</td>
</tr>
<tr>
<td>4-6</td>
<td>Invalid Page Table Entry</td>
<td>4-5</td>
</tr>
<tr>
<td>6-1</td>
<td>Pipelined Instruction Execution</td>
<td>6-3</td>
</tr>
<tr>
<td>6-2</td>
<td>Dual-Operation Data Paths</td>
<td>6-16</td>
</tr>
<tr>
<td>6-3</td>
<td>Data Paths by Instruction (1 of 8)</td>
<td>6-18</td>
</tr>
<tr>
<td>6-4</td>
<td>Data Path Mnemonics</td>
<td>6-26</td>
</tr>
<tr>
<td>6-5</td>
<td>PSR Fields for Graphics Operations</td>
<td>6-29</td>
</tr>
<tr>
<td>6-6</td>
<td>FADDP with 8-Bit Pixels</td>
<td>6-33</td>
</tr>
<tr>
<td>6-7</td>
<td>FADDP with 16-Bit Pixels</td>
<td>6-34</td>
</tr>
<tr>
<td>6-8</td>
<td>FADDP with 32-Bit Pixels</td>
<td>6-35</td>
</tr>
<tr>
<td>6-9</td>
<td>FADDP with 64-Bit Z-Buffer</td>
<td>6-36</td>
</tr>
<tr>
<td>6-10</td>
<td>64-Bit Distance Interpolation</td>
<td>6-37</td>
</tr>
<tr>
<td>6-11</td>
<td>Dual-Instruction Mode Transitions (1 of 2)</td>
<td>6-40</td>
</tr>
<tr>
<td>8-1</td>
<td>Register Allocation</td>
<td>8-2</td>
</tr>
<tr>
<td>8-2</td>
<td>Stack Frame Format</td>
<td>8-6</td>
</tr>
<tr>
<td>8-3</td>
<td>Example Memory Layout</td>
<td>8-7</td>
</tr>
<tr>
<td>9-1</td>
<td>Z-Buffer Interpolation</td>
<td>9-23</td>
</tr>
<tr>
<td>9-2</td>
<td>faddz Operands</td>
<td>9-24</td>
</tr>
<tr>
<td>9-3</td>
<td>Pixel Interpolation for Gouraud Shading</td>
<td>9-27</td>
</tr>
<tr>
<td>9-4</td>
<td>faddp Operands</td>
<td>9-27</td>
</tr>
</tbody>
</table>
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Pixel Formats</td>
<td>2-3</td>
</tr>
<tr>
<td>2-2</td>
<td>Single and Double Real Encodings</td>
<td>2-5</td>
</tr>
<tr>
<td>3-1</td>
<td>Values of PS</td>
<td>3-4</td>
</tr>
<tr>
<td>3-2</td>
<td>Values of RB</td>
<td>3-7</td>
</tr>
<tr>
<td>3-3</td>
<td>Values of RC</td>
<td>3-8</td>
</tr>
<tr>
<td>3-4</td>
<td>Values of RM</td>
<td>3-9</td>
</tr>
<tr>
<td>4-1</td>
<td>Combining Directory and Page Protection</td>
<td>4-8</td>
</tr>
<tr>
<td>5-1</td>
<td>Control Register Encoding for Assemblers</td>
<td>5-20</td>
</tr>
<tr>
<td>6-1</td>
<td>Precision Specification</td>
<td>6-2</td>
</tr>
<tr>
<td>6-2</td>
<td>DPC Encoding</td>
<td>6-17</td>
</tr>
<tr>
<td>6-3</td>
<td>FADDP MERGE Update</td>
<td>6-32</td>
</tr>
<tr>
<td>7-1</td>
<td>Types of Traps</td>
<td>7-1</td>
</tr>
<tr>
<td>7-2</td>
<td>Register and Cache Values after Reset</td>
<td>7-9</td>
</tr>
<tr>
<td>8-1</td>
<td>Register Allocation</td>
<td>8-1</td>
</tr>
<tr>
<td>9-1</td>
<td><strong>faddz</strong> Visualization</td>
<td>9-25</td>
</tr>
<tr>
<td>9-2</td>
<td>Accumulator Initial Values</td>
<td>9-29</td>
</tr>
<tr>
<td>9-3</td>
<td>Accumulator Initialization Table</td>
<td>9-30</td>
</tr>
<tr>
<td>A-1</td>
<td>Precision Specification</td>
<td>A-2</td>
</tr>
<tr>
<td>A-2</td>
<td>FADDP MERGE Update</td>
<td>A-5</td>
</tr>
<tr>
<td>B-1</td>
<td>Register Encoding</td>
<td>B-1</td>
</tr>
</tbody>
</table>

Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Example of <strong>bla</strong> Usage</td>
<td>5-18</td>
</tr>
<tr>
<td>5-2</td>
<td>Cache Flush Procedure</td>
<td>5-22</td>
</tr>
<tr>
<td>5-3</td>
<td>Examples of <strong>lock</strong> and <strong>unlock</strong> Usage</td>
<td>5-24</td>
</tr>
<tr>
<td>7-1</td>
<td>Saving Pipeline States</td>
<td>7-12</td>
</tr>
<tr>
<td>7-2</td>
<td>Restoring Pipeline States (1 of 2)</td>
<td>7-13</td>
</tr>
<tr>
<td>8-1</td>
<td>Reading Misaligned 32-Bit Value</td>
<td>8-4</td>
</tr>
<tr>
<td>8-2</td>
<td>Subroutine Entry and Exit with Frame Pointer</td>
<td>8-6</td>
</tr>
<tr>
<td>8-3</td>
<td>Subroutine Entry and Exit without Frame Pointer</td>
<td>8-7</td>
</tr>
<tr>
<td>8-4</td>
<td>Possible Implementation of <strong>alloca</strong></td>
<td>8-8</td>
</tr>
<tr>
<td>9-1</td>
<td>Sign Extension</td>
<td>9-1</td>
</tr>
<tr>
<td>9-2</td>
<td>Loading Small Unsigned Integers</td>
<td>9-1</td>
</tr>
<tr>
<td>9-3</td>
<td>Single-Precision Divide</td>
<td>9-2</td>
</tr>
<tr>
<td>9-4</td>
<td>Double-Precision Divide</td>
<td>9-3</td>
</tr>
<tr>
<td>9-5</td>
<td>Integer Multiply</td>
<td>9-4</td>
</tr>
<tr>
<td>9-6</td>
<td>Single to Double Conversion</td>
<td>9-5</td>
</tr>
<tr>
<td>9-7</td>
<td>Signed Integer Divide</td>
<td>9-6</td>
</tr>
<tr>
<td>9-8</td>
<td>String Copy</td>
<td>9-7</td>
</tr>
<tr>
<td>9-9</td>
<td>Pipelined Add</td>
<td>9-8</td>
</tr>
<tr>
<td>9-10</td>
<td>Pipelined Dual-Operation Instruction</td>
<td>9-10</td>
</tr>
<tr>
<td>9-11</td>
<td>Pipelined Double-Precision Dual Operation</td>
<td>9-12</td>
</tr>
<tr>
<td>9-12</td>
<td>Dual-Instruction Mode</td>
<td>9-14</td>
</tr>
<tr>
<td>9-13</td>
<td>Matrix Multiply, Cached Loads Only (Sheet 1 of 2)</td>
<td>9-16</td>
</tr>
<tr>
<td>9-14</td>
<td>Matrix Multiply, Cached and Pipelined Loads (Sheet 1 of 2)</td>
<td>9-19</td>
</tr>
<tr>
<td>9-15</td>
<td>Setting Pixel Size</td>
<td>9-21</td>
</tr>
<tr>
<td>9-16</td>
<td>Register Assignments</td>
<td>9-22</td>
</tr>
<tr>
<td>9-17</td>
<td>Construction of Z Interpolants</td>
<td>9-26</td>
</tr>
<tr>
<td>Example</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9-18</td>
<td>Construction of Color Interpolants</td>
<td>9-28</td>
</tr>
<tr>
<td>9-19</td>
<td>Z Mask Procedure</td>
<td>9-28</td>
</tr>
<tr>
<td>9-20</td>
<td>Accumulator Initialization</td>
<td>9-30</td>
</tr>
<tr>
<td>9-21</td>
<td>3-D Rendering (1 of 2)</td>
<td>9-31</td>
</tr>
</tbody>
</table>
Architectural Overview
CHAPTER 1  
ARCHITECTURAL OVERVIEW

The Intel i860™ 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 applications programs in the conventional manner, without explicit programming. For vector operations, however, programmers can explicitly use the data cache as if it were a large block of vector registers.

To sustain high performance, 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.

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

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:
  - 32 general-purpose integer registers, each 32-bits wide.
  - 32 floating-point registers, each 32-bits wide, which can also be configured as 64- and 128-bit registers. The floating-point registers also serve as the staging area for data going into and out of the floating-point pipelines.

Figure 1-1 illustrates the registers and data paths of the i860 microprocessor.
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 $8 \times 128$-bit registers, $16 \times 64$-bit registers, or $32 \times 32$-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 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 offers 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...

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

...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 i486™ 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 to 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 its high-performance. Even when used as a scalar machine, the i860 microprocessor implements concurrency between integer and floating-point operations, as long as there are no conflicts for internal resources. An integer instruction that follows a floating-point instruction begins immediately, overlapping the floating-point instruction. A floating-point instruction that follows an integer instruction also begins immediately.

- Compilers designed for the vector model can treat 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 an even greater performance gain from the i860 microprocessor.
Designers of compilers will find that the i860 microprocessor offers more flexibility than traditional vector processors. 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 offers 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 i486 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 i486 microprocessors. This level of compatibility facilitates use of the i860 microprocessor in multiprocessor systems with a 386 or i486 microprocessor. Moreover, complete hardware and software support for such multiprocessor systems is available.

An i860 microprocessor can be used with a 386, 386 SX, or i486 microprocessor system. The i860 microprocessor extends system performance to supercomputer levels, while the 386/386 SX/i486 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
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 \(2,147,483,647 \ (+2^{31} - 1)\). Arithmetic operations on 8- and 16-bit integers can be performed by sign-extending the 8- or 16-bit values to 32 bits, then using the 32-bit operations.

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

When an eight- or 16-bit item is loaded into a register, it is converted to an integer by sign-extending the value to 32 bits. When an eight- or 16-bit item is stored from a register, the corresponding number of low-order bits of the register are used.

2.2 ORDINAL

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

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

2.3 SINGLE-PRECISION REAL

![Diagram of single-precision real data format]

2-1
A single-precision real (also called “single real”) data type is a 32-bit binary floating-point number. Bit 31 is the sign bit; bits 30..23 are the exponent; and bits 22..0 are the fraction. In accordance with ANSI/IEEE standard 754, the value of a single-precision real is defined as follows:

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

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

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

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

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

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

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

The special values infinity, NaN, indefinite, and denormal generate a trap when encountered. The trap handler implements IEEE-standard results. (Refer to Table 2-2 for encoding of these special values.)
A double real value occupies an even/odd pair of floating-point registers. Bits 31..0 are stored in the even-numbered floating-point register; bits 63..32 are stored in the next higher odd-numbered floating-point register.

2.5 PIXEL

A pixel may be 8, 16, or 32 bits long depending on color and intensity resolution requirements. Regardless of the pixel size, the 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.

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.

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 2-1. Pixel Formats

<table>
<thead>
<tr>
<th>Pixel Size (in bits)</th>
<th>Bits of Color 1* Intensity</th>
<th>Bits of Color 2* Intensity</th>
<th>Bits of Color 3* Intensity</th>
<th>Bits of Other Attribute (Texture)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>N (&lt; 8) bits of intensity**</td>
<td>8</td>
<td>8</td>
<td>8 – N</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

* 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.
DATA TYPES

8-BIT PIXEL

I

C

16-BIT PIXEL

R
G
B

32-BIT PIXEL

R
G
B
T

I—INTENSITY, R—RED INTENSITY, G—GREEN INTENSITY, B—BLUE INTENSITY, C—COLOR, T—TEXTURE

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

Figure 2-1. Pixel Format Examples
Table 2-2. Single and Double Real Encodings

<table>
<thead>
<tr>
<th>Class</th>
<th>Sign</th>
<th>Biased Exponent</th>
<th>Fraction ff–ff*</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaNs</td>
<td>0</td>
<td>11..11</td>
<td>11..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11..11</td>
<td>10.00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11..11</td>
<td>01..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11..11</td>
<td>00..01</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Infinity</td>
<td>0</td>
<td>11..11</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>11..10</td>
<td>11..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00..01</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00..00</td>
<td>11..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>00..00</td>
<td>00..01</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td>0</td>
<td>00..00</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00..00</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00..00</td>
<td>00..01</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>00..01</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11..10</td>
<td>11..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11..11</td>
<td>00..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11..11</td>
<td>00..01</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11..11</td>
<td>10..00</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11..11</td>
<td>11..11</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

*Integer bit is implied and not stored.
Registers
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.e 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, ... , f28). The instruction must designate the lowest register number of the set of registers containing 64- or 128-bit values. Misaligned register numbers produce undefined results. The register with the lowest number contains the least significant part of the value.

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 the value of the result, as explained in Chapter 5. The conditional branch instructions test CC. The bna instruction described in Chapter 5 sets and tests LCC (Loop Condition Code).
- IM (Interrupt Mode) enables external interrupts if set; disables interrupts if clear. (Chapter 7 covers interrupts.)

- 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.)

- 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.

- IT (Instruction Trap), IN (Interrupt), IAT (Instruction Access Trap), DAT (Data Access Trap), and FT (Floating-Point Trap) are trap flags. They are set when the corresponding trap condition occurs. The trap handler examines these bits to determine which condition or conditions have caused the trap. Refer to 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 DIM if it is executing in single-instruction mode. If DIM is set, the i860 microprocessor resumes execution in dual-instruction mode after returning from the trap handler.

• When KNF (Kill Next Floating-Point Instruction) is set, the next floating-point instruction is suppressed (except that its dual-instruction mode bit is interpreted). A trap handler sets KNF if the trapped floating-point instruction should not be reexecuted. KNF is especially useful for returning from a trap that occurred in dual-instruction mode, because it permits the core instruction to be executed while the floating-point instruction is suppressed. KNF is automatically reset by the i860 microprocessor when the instruction has been successfully bypassed. It is possible that the core instruction may cause a trap when the floating-point instruction is suppressed. In this case KNF remains set, permitting retry of the core instruction.

• SC (Shift Count) stores the shift count used by the last right-shift instruction. It controls the number of shifts executed by the double-shift instruction, as described in Chapter 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.

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.

• The processor type is one for the i860 microprocessor.

<table>
<thead>
<tr>
<th>Table 3-1. Values of PS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11 (undefined)</td>
</tr>
</tbody>
</table>
- 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 $2^{12 + \text{DCS}}$; 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 information on byte ordering.
• OF (Overflow Flag) is set by `adds`, `addu`, `subs`, and `subu` when integer overflow occurs. For `adds` and `subs`, OF is set if the carry from bit 31 is different than the carry from bit 30. For `addu`, OF is set if there is a carry from bit 31. For `subu`, OF is set if there is no carry from bit 31. Under all other conditions, it is cleared by these instructions. OF 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. The trap occurs before the register or memory update by the load or store instruction.

3.6 DIRECTORY BASE REGISTER

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

- ATE (Address Translation Enable), when set, enables the virtual-address translation 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

![Figure 3-4. Directory Base Register](image-url)
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. A trap that occurs during a locked sequence immediately clears BL and the LOCK# signal and sets IL in epsr. In this case the trap handler should resume execution at the beginning of the locked sequence. The lock and unlock instructions control the BL bit (refer to Chapter 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 (except for the case of setting the D- or P-bit in a PTE that is not itself in the data cache).

- 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 stc 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™ 64-Bit Microprocessor Hardware Design Guide 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.

- 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.

### Table 3-2. Values of RB

<table>
<thead>
<tr>
<th>Value</th>
<th>Replace TLB Block</th>
<th>Replace Instruction and Data Cache Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3-3. Values of RC

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Selects the normal replacement algorithm where any block in the set may be replaced on cache misses in all caches.</td>
</tr>
<tr>
<td>01</td>
<td>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.</td>
</tr>
<tr>
<td>10</td>
<td>Data cache misses replace the block selected by the low-order bit of RB.</td>
</tr>
<tr>
<td>11</td>
<td>Disables data cache replacement.</td>
</tr>
</tbody>
</table>

- DTB (Directory Table Base) contains the high-order 20 bits of the physical address of the page directory when address translation is enabled (i.e. ATE = 1). The low-order 12 bits of the address are zeros (therefore the directory must be located on a 4K boundary).

3.7 FAULT INSTRUCTION REGISTER

When a trap occurs, this register (the fir) contains the address of the instruction that caused the trap, as described in Chapter 7. Reading fir anytime except the first time after a trap occurs only yields the address of the ld.c instruction. The fir cannot be modified by the st.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.
- The U-bit (Update Bit), if set in the value that is loaded into fsr by a st.c instruction, enables updating of the result-status bits (AE, AA, AI, AO, AU, MA, MI, MO, and MU) in the first-stage of the floating-point adder and multiplier pipelines. If this bit is clear, the result-status bits are unaffected by a st.c instruction; st.c ignores the corresponding bits in the value that is being loaded. A st.c always updates fsr bits 21..17
Figure 3-5. Floating-Point Status Register

Table 3-4. Values of RM

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

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

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

• RR (Result Register) specifies which floating-point register (f0-f31) was the destination register when a result-exception trap occurs due to a scalar operation.

• LRP (Load Pipe Result Precision), IRP (Integer (Graphics) Pipe Result Precision), MRP (Multiplier Pipe Result Precision), and ARP (Adder Pipe Result Precision) aid in restoring pipeline state after a trap or process switch. Each defines the precision of the last-stage result in the corresponding pipeline. One of these bits is set when the result in the last stage of the corresponding pipeline is double precision; it is cleared if the result is single precision. These bits cannot be changed by software.
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.
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. The BE bit of epsr selects the mode, as Chapter 3 describes. Code accesses and page directory/page table accesses are always done with little endian addressing. Figure 4-1 shows the difference between the two storage modes. Figure 4-2 defines by example how data is transferred from memory over the bus into a register in both modes. Big endian and little endian data areas should not be mixed within a 64-bit data word. Illustrations of data structures in this manual show data stored in little endian mode, i.e. the rightmost (low-order) byte is at the lowest memory address.

4.1 ALIGNMENT

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.

![Figure 4-1. Memory Formats](image-url)
• 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 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 4-3 shows the format of a virtual address.

Figure 4-4 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.

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 the DTB field of the `dirbase` register. Memory management software has the option of using one page directory for all processes, one page directory for each process, or some combination of the two.

### 4.2.4 Page-Table Entries

Page-table entries (PTEs) in either level of page tables have the same format. Figure 4-5 illustrates this format.

#### 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 4-6 illustrates the format of a page-table entry when \( P = 0 \).

If \( P = 0 \) in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals either a data-access fault or an instruction-access fault. In software systems that support paged virtual memory, the trap handler can bring the required page into physical memory. Refer to 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 data caching from the corresponding page is inhibited (note, however, that instruction caching is not inhibited). If WT is clear, the normal write-back policy is applied to data from the page in the on-chip caches. (Future implementations of the architecture may provide a write-through policy, in which case pages that have WT set will be written to cache as well as to memory.) The WT bit of page directory entries is not referenced by the processor, but is reserved.

To control external caches, the PTB output pin reflects either CD or WT depending on the PBM bit of epsr (refer to 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 (except during a locked sequence, when a trap occurs instead). 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 uses the LOCK# signal to coordinate its testing and setting of 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:

\[
\begin{align*}
    WP = 0 & \quad \text{All pages are writable.} \\
    WP = 1 & \quad \text{A write to a page whose W bit is not set causes a trap.}
\end{align*}
\]

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 from 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.

Table 4-1. Combining Directory and Page Protection

<table>
<thead>
<tr>
<th>Page Directory Entry</th>
<th>Page Table Entry</th>
<th>Combined Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U-bit</td>
<td>W-bit</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>1 1</td>
</tr>
</tbody>
</table>

NOTES:
N = No Access Allowed
R = Read Access Only
R/W = Both Reads and Writes Allowed
X = Don't Care
1. Read the PTE (page table entry) at the physical address formed by DTB:DIR:00. Note that the data cache is *not* accessed during PTE fetches; therefore, the operating system must ensure that the page table is not in the cache.

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

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

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

5. If A in the PTE is zero and if the TLB miss occurred while the bus was locked, generate a data- or instruction-access fault. (The trap allows software to set A to one and restart the sequence. This avoids ambiguity in determining what address corresponds to a locked semaphore for external bus hardware use.)

6. If A in the PTE is zero and if the TLB miss occurred while the bus was not locked, assert LOCK#, refetch the PTE, set A, and store the PTE, deasserting LOCK# during the store.

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

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

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

10. Form the physical address as PFA2:OFFSET.

### 4.2.6 Address Translation Faults

An address translation fault can be signalled as either an instruction-access fault or a data-access fault. (Refer to 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. When address translation is enabled (ATE = 1), caching uses virtual-address tags. The effects of mapping two different virtual addresses in the same address space to the same physical address are undefined.

The caching policy employed is write-back; i.e., writes to memory locations that are cached update only the cache and do not update memory until the corresponding cache block is needed to cache newly read data.

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. The 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 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 (except for the case of setting the D-, P- or A-bit in a PTE that is not itself in the data cache).

**NOTE**

When an st.c dirbase changes DTB or activates ITI, the mapping of the page containing the currently executing instruction and the next six instructions should not be different in the new page tables. The next six instructions 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 before using the ITI bit of dirbase to flush the instruction or address-translation cache (except for the case of setting the D-, P- or A-bit in a PTE that is not itself in the data cache), before enabling or disabling address translation (via the ATE bit), and before changing the page frame address field of any PTE.

In the translation process, the i860 microprocessor searches only external memory for page directories and page tables. The data cache is not searched; therefore, page tables and directories should be kept in noncacheable memory or flushed from the cache by any code that modifies them.
Core Instructions
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; control transfers; and system control functions. All these instructions are executed by the core unit.

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

- $c$: One of the control registers $f_{ir}$, $psr$, $epsr$, $dirbase$, $db$, or $fsr$
- $f$: One of the floating-point registers: $f_0$ through $f_{31}$
- $i$: One of the integer registers: $r_0$ through $r_{31}$

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

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

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

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

- $#const$: A 16-bit immediate constant or address offset that the i860™ microprocessor sign-extends to 32 bits when computing the effective address.
- $lbroff$: A signed, 26-bit, immediate, relative branch offset.
**CORE INSTRUCTIONS**

**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.

**\(mem.x(address)\)**
The contents of the memory location indicated by \(address\) with a size of \(x\).

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

<table>
<thead>
<tr>
<th>ld.x isrc1(isrc2), idest</th>
<th>(Load Integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>idest ← mem.x(isrc1 + isrc2)</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Traps

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

Programming Notes

For best performance, observe the following guidelines:

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

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

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

```
orh   HIGH16a, r0, r31
ld.1   LOW16(r31), idest
```

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 STORING INTEGER

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

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

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

### Traps

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

### Programming Notes

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

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

```
orh  HIGH16a, r0, r31
st.1  isrc1ni, 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

\[
\text{ixfr } \text{isrc1ni}, \text{fdest} \\
\text{fdest } \leftarrow \text{isrc1ni}
\]

(Transfer Integer to F-P Register)

The \text{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 \text{ixfr} should not be referenced as a source operand in the next two instructions.
5.4 LOAD FLOATING-POINT

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 isrc2.

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. No trap occurs when the data loaded is not a valid floating-point number.

Programming Notes

A pfld cannot load a 128-bit operand.

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

.y = .I (32 bits), .d (64 bits), or .q (128 bits); .z = .I or .d
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 string of successive pfld instructions causes internal delays due the fact that the bandwidth of the i860 microprocessor bus is one transfer per two cycles.

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

Floating-point stores transfer 32-, 64-, or 128-bit values from the floating-point registers to memory. These may be floating-point values or integers. Floating-point stores allow isrc1 to be used as an index register. An autoincrement option supports constant-stride vector addressing. If this option is specified, the i860 microprocessor stores the effective address into isrc2.

Traps

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

Programming Notes

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

For best performance, observe the following guidelines:

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

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

<table>
<thead>
<tr>
<th>pst.d fdest, #const (isrc2)</th>
<th>Pixel Store (Normal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pst.d fdest, #const (isrc2) + +</td>
<td>Pixel Store (AutoIncrement)</td>
</tr>
</tbody>
</table>

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

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

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{addu} isrc1, isrc2, idest & (Add unsigned) \\
\hline
\textit{idest} $\leftarrow$ isrc1 $+$ isrc2 \\
\textit{OF} $\leftarrow$ bit 31 carry \\
\textit{CC} $\leftarrow$ bit 31 carry \\
\hline
\textbf{adds} isrc1, isrc2, idest & (Add signed) \\
\hline
\textit{idest} $\leftarrow$ isrc1 $+$ isrc2 \\
\textit{OF} $\leftarrow$ (bit 31 carry $\geq$ bit 30 carry) \\
Using signed comparison, \\
\textit{CC} set if isrc2 $<$ \textit{comp2} (isrc1) \\
\textit{CC} clear if isrc2 $\geq$ \textit{comp2} (isrc1) \\
\hline
\textbf{subu} isrc1, isrc2, idest & (Subtract unsigned) \\
\hline
\textit{idest} $\leftarrow$ isrc1 $-$ isrc2 \\
\textit{OF} $\leftarrow$ NOT (bit 31 carry) \\
\textit{CC} $\leftarrow$ bit 31 carry \\
\textit{(i.e., using unsigned comparison,} \\
\textit{CC} set if isrc2 $\geq$ isrc1 \\
\textit{CC} clear if isrc2 $>$ isrc1 \\
\hline
\textbf{subs} isrc1, isrc2, idest & (Subtract signed) \\
\hline
\textit{idest} $\leftarrow$ isrc1 $-$ isrc2 \\
\textit{OF} $\leftarrow$ (bit 31 carry $\geq$ bit 30 carry) \\
Using signed comparison, \\
\textit{CC} set if isrc2 $>$ isrc1 \\
\textit{CC} clear if isrc2 $\leq$ isrc1 \\
\hline
\end{tabular}
\end{center}

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

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

**Flags Affected**

\texttt{CC} and \texttt{OF} as defined above.

**Programming Notes**

For optimum performance, a conditional branch should not directly follow 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 \texttt{subs -1, isrc2, idest} yields the one's complement of \texttt{isrc2}.

When \texttt{isrc1} is immediate, the immediate value is sign-extended to 32-bits even for the unsigned instructions \texttt{addu} and \texttt{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:

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</td>
<td></td>
</tr>
<tr>
<td>$r6 = 2 - r5$</td>
<td><code>subs 2, r5, r6</code></td>
</tr>
<tr>
<td>$r6 = r5 - 2$</td>
<td><code>adds -2, r5, r6</code></td>
</tr>
<tr>
<td>Unsigned</td>
<td></td>
</tr>
<tr>
<td>$r6 = 2 - r5$</td>
<td><code>subu 2, r5, r6</code></td>
</tr>
<tr>
<td>$r6 = r5 - 2$</td>
<td><code>addu -2, r5, r6</code></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Encoding</th>
<th>Branch When True</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Signed</td>
</tr>
<tr>
<td>$var \leq const$</td>
<td><code>subs const, var</code></td>
<td><code>bnc</code></td>
</tr>
<tr>
<td></td>
<td><code>subu const, var</code></td>
<td></td>
</tr>
<tr>
<td>$var &lt; const$</td>
<td><code>adds -const, var</code></td>
<td><code>bc</code></td>
</tr>
<tr>
<td></td>
<td><code>addu -const, var*</code></td>
<td></td>
</tr>
<tr>
<td>$var \geq const$</td>
<td><code>adds -const, var</code></td>
<td><code>bnc</code></td>
</tr>
<tr>
<td></td>
<td><code>addu -const, var*</code></td>
<td></td>
</tr>
<tr>
<td>$var &gt; const$</td>
<td><code>subs const, var</code></td>
<td><code>bc</code></td>
</tr>
<tr>
<td></td>
<td><code>subu const, var</code></td>
<td></td>
</tr>
</tbody>
</table>

* Valid only when const > 0
5.8 SHIFT INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shl isrc1, isrc2, idest</td>
<td>Shift left: idest ← isrc2 shifted left by isrc1 bits</td>
</tr>
<tr>
<td>shr isrc1, isrc2, idest</td>
<td>Shift right: idest ← isrc2 shifted right by isrc1 bits</td>
</tr>
<tr>
<td>shra isrc1, isrc2, idest</td>
<td>Shift right arithmetic: idest ← isrc2 arithmetically shifted right by isrc1 bits</td>
</tr>
<tr>
<td>shrd isrc1ni, isrc2, idest</td>
<td>Shift right double: idest ← low-order 32 bits of isrc1ni:isrc2 shifted right by SC bits</td>
</tr>
</tbody>
</table>

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

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

Programming Notes

The shift instructions are recommended for the integer register-to-register move and for no-operations, because they do not affect the condition code. The following assembler pseudo-operations utilize the shift instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov isrc2, idest</td>
<td>Register-to-register move: Assembler pseudo-operation, equivalent to: shl r0, isrc2, idest</td>
</tr>
<tr>
<td>nop</td>
<td>Core no-operation: Assembler pseudo-operation, equivalent to: shl r0, r0, r0</td>
</tr>
<tr>
<td>fnop</td>
<td>Floating-point no-operation: Assembler pseudo-operation, equivalent to: shrd r0, r0, r0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trap isrc1ni, isrc2, idest</td>
<td>(Software trap) Generate trap with IT set in psr</td>
</tr>
<tr>
<td>intovr</td>
<td>(Software trap on integer overflow)</td>
</tr>
<tr>
<td>IF OF in epsr = 1</td>
<td>THEN generate trap with IT set in psr</td>
</tr>
<tr>
<td>FI</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and isrc1, isrc2, idest</code></td>
<td>(Logical AND)</td>
</tr>
<tr>
<td><code>idest ← isrc1 AND isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>andh #const, isrc2, idest</code></td>
<td>(Logical AND high)</td>
</tr>
<tr>
<td><code>idest ← (#const shifted left 16 bits) AND isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>andnot isrc1, isrc2, idest</code></td>
<td>(Logical AND NOT)</td>
</tr>
<tr>
<td><code>idest ← NOT isrc1 AND isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>andnoth #const, isrc2, idest</code></td>
<td>(Logical AND NOT high)</td>
</tr>
<tr>
<td><code>idest ← NOT (#const shifted left 16 bits) AND isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>or isrc1, isrc2, idest</code></td>
<td>(Logical OR)</td>
</tr>
<tr>
<td><code>idest ← isrc1 OR isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>orh #const, isrc2, idest</code></td>
<td>(Logical OR high)</td>
</tr>
<tr>
<td><code>idest ← (#const shifted left 16 bits) OR isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>xor isrc1, isrc2, idest</code></td>
<td>(Logical XOR)</td>
</tr>
<tr>
<td><code>idest ← isrc1 XOR isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
<tr>
<td><code>xorh #const, isrc2, idest</code></td>
<td>(Logical XOR high)</td>
</tr>
<tr>
<td><code>idest ← (#const shifted left 16 bits) XOR isrc2</code></td>
<td>CC set if result is zero, cleared otherwise</td>
</tr>
</tbody>
</table>

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

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

Flags Affected

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

Programming Notes

Bit operations can be implemented using logical operations. `Isrc1` is an immediate constant which contains a one in the bit position to be operated on and zeros elsewhere.
<table>
<thead>
<tr>
<th>Bit Operation</th>
<th>Equivalent Logical Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set bit</td>
<td>or</td>
</tr>
<tr>
<td>Clear bit</td>
<td>andnot</td>
</tr>
<tr>
<td>Complement bit</td>
<td>xor</td>
</tr>
<tr>
<td>Test bit</td>
<td>and (CC set if bit is clear)</td>
</tr>
</tbody>
</table>
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 \texttt{bc.t} and \texttt{bnc.t} are delayed forms of \texttt{bc} and \texttt{bnc}. The delayed branch instructions \texttt{bc.t} and \texttt{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 \texttt{bc}, \texttt{bnc}, \texttt{bte}, \texttt{btne} should be used when branch is taken less frequently than not; for example, in certain search routines.

If a trap occurs on a \texttt{bla} instruction or the next instruction, LCC is not updated. The trap handler resumes execution with the \texttt{bla} instruction, so the LCC setting is not lost.
Programming Notes

The **bla** instruction is useful for implementing loop counters, where **isrc2** is the loop counter and **isrc1** is set to -1. In such a loop implementation, a **bla** instruction may be performed before the loop is entered to initialize the LCC bit of the **psr**. The target of this **bla** should be the sequential instruction after the next, so that the next sequential instruction is executed regardless of the setting of LCC. Another **bla** instruction placed as the next to last instruction of the loop can test for loop completion and update the loop counter. The total number of iterations is the value of **isrc2** before the first **bla** instruction, plus one. Example 5-1 illustrates this use of **bla**.

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

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

---

```plaintext
// 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 \texttt{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 \texttt{psr} at the time \texttt{bri} is executed. In this case, the instruction following the \texttt{bri} must be a load that restores \texttt{isrc1ni} to the value it had before the trap occurred.

\textbf{Programming Notes}

When using \texttt{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).

The register \texttt{isrc1ni} of the \texttt{calli} instruction must not be \texttt{r1}.
5.12 CONTROL REGISTER ACCESS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.c csrc2, idest</td>
<td>Load from control register</td>
</tr>
<tr>
<td>idest ← csrc2</td>
<td></td>
</tr>
<tr>
<td>st.c isrc1ni, csrc2</td>
<td>Store to control register</td>
</tr>
<tr>
<td>csrc2 ← isrc1ni</td>
<td></td>
</tr>
</tbody>
</table>

Csrc2 specifies a control register that is transferred to or from a general-purpose register. The function of each control register is defined in Chapter 3. As shown below, some registers or parts of registers are write-protected when the U-bit in the psr is set. A store to those registers or bits is ignored when the i860 microprocessor is in user mode. The encoding of csrc2 is defined by Table 5-1.

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

### Table 5-1. Control Register Encoding for Assemblers

<table>
<thead>
<tr>
<th>Register</th>
<th>Src2 Code</th>
<th>User-Mode Write-Protected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir</td>
<td>0</td>
<td>N/A***</td>
</tr>
<tr>
<td>psr</td>
<td>1</td>
<td>Yes*</td>
</tr>
<tr>
<td>dirbase</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>db</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>fsr</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>epsr</td>
<td>5</td>
<td>Yes**</td>
</tr>
</tbody>
</table>

* Only the psr bits BR,BW, PIM, IM, PU, U, IT, IN, IAT, DATA, 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.
*** The fir register cannot be written by the st.c instruction.
5.13 CACHE Flush

<table>
<thead>
<tr>
<th>flush #const(isrc2)</th>
<th>(Cache flush)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flush #const(isrc2) + +</td>
<td>(Normal)</td>
</tr>
</tbody>
</table>

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

The flush instruction is used to force modified data in the data cache to external memory. Because the register designated by idest is undefined after flush, assemblers should encode idest as zero. The address #const + isrc2 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 + isrc2. The particular block that is forced to memory is controlled by the RB field of dirbase. In user mode, execution of flush is suppressed; use it only in supervisor mode.

Example 5-2 shows how to use the flush instruction. The addresses used by the flush instruction refer to a reserved 4 Kbyte memory area that is not used to store data. This ensures that, when flushing the cache before a task switch, cached data items from the old task are not transferred to the new task. These addresses must be valid and writable in both the old and the new task's space. Any other usage of flush has undefined results.

Cache elements containing modified data are written back to memory by making two passes, each of which references every 32nd byte of the reserved area with the flush instruction. Before the first pass, the RC field in dirbase is set to two and RB is set to zero. This causes data-cache misses to flush element zero of each set. Before the second pass, RB is changed to one, causing element one of each set to be flushed.
Example 5-2. Cache Flush Procedure
5.14 BUS LOCK

<table>
<thead>
<tr>
<th>lock</th>
<th>(Begin interlocked sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set BL in dirbase. The next load or store that misses the cache locks that location, preventing locked access to it by other processors. External interrupts are disabled from the first instruction after the lock until the location is unlocked.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unlock</th>
<th>(End interlocked sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear BL in dirbase. The next load or store unlocks the location (regardless of whether it hits in the cache). Interrupts are enabled.</td>
<td></td>
</tr>
</tbody>
</table>

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 30 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 ensure that no data access fault occurs, it must first store unmodified values in the other store locations. To ensure that no instruction-access fault occurs, the code that is not restartable should not span a page boundary.

After a lock instruction, the location 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.

If a trap occurs after a lock instruction but before the load or store that follows the corresponding unlock, the processor clears BL and sets the IL (interlock) bit of epsr. This is likely to happen, for example, during TLB miss processing, when the A-bit of the page table entry is not set.

If the processor encounters another lock instruction before unlocking the bus or an unlock with no preceding lock, 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 30-33 instructions, it triggers an instruction fault. In such a case, the trap handler will find both IL and IT set. The instruction pointed to by fir may or may not have been executed.

When multiple memory locations are accessed during a locked sequence, only the first location with a cache miss is guaranteed to be locked against access by other processors.

For high-performance multiprocessors, this allows a read-for-ownership policy, instead of locking the system bus.

Between locked sequences, at least one cycle of LOCK# deactivation is guaranteed by the behavior of unlock.
Note that, for each shared data structure, software must establish a single location that is the first location referenced by any locked sequence that requires that data. For example, the head of a doubly linked list should be referenced before accessing items in the middle of the list.

Example 5-3 shows how `lock` and `unlock` can be used in a variety of interlocked operations.

**Programming Notes**

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

```plaintext
// LOCKED TEST AND SET
// Value to put in semaphore is in r23
lock  //
1d.b  semaphore, r22  // Put current value of semaphore in r22
unlock //
st.b  r23, semaphore //

// LOCKED LOAD-ALU-STORE
lock  //
1d.l  word, r22  // Can be any ALU operation
addu 1, r22, r22 //
unlock //
st.l  r22, word //

// LOCKED COMPARE AND SWAP
// Swaps r23 with word in memory, if word = r21
lock  //
1d.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
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.

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

- $f$: One of the floating-point registers: $f_0$ through $f_{31}$
- $i$: One of the integer registers: $r_0$ through $r_{31}$

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

- $src_1$: The first of the two source-register designators.
- $src_2$: The second of the two source-register designators.
- $dest$: The destination register designator.

Thus, the operand specifier $fsrc_2$, for example, means that a floating-point register is used and that the encoding of that register must be placed in the $src_2$ field of the machine instruction.

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 in Table 6-1. In this manual, the suffixes .p and .r refer to the precision specification. In an actual program, .p is to be replaced by the precision specification .s, .sd, or .dd (.ds not permitted). Likewise, .r is to be replaced by the precision specification .s, .sd, .ds, or .dd.
FLOATING-POINT INSTRUCTIONS

Table 6-1. Precision Specification

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Source Precision</th>
<th>Result Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ss</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>.sd</td>
<td>single</td>
<td>double</td>
</tr>
<tr>
<td>.dd</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>.ds</td>
<td>double</td>
<td>single</td>
</tr>
</tbody>
</table>

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 A_1, A_2, and A_3. The stages are designed such that A_{i+1} for one adder instruction can execute in parallel with A_i for the next adder instruction. Furthermore, each A_i can be executed in just one clock. The pipelining within the multiplier and graphics units can be described similarly, except that the number of stages and the number of clocks per stage may be different.

Figure 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 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 fsrc, the result of the last stage of the pipeline is stored in the destination register fdest, the pipeline is advanced one stage, and the input operands fsrc1 and fsrc2 are transferred to the first stage of the pipeline.

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

There are four floating-point pipelines: one for the multiplier, one for the adder, one for the graphics unit, and one for floating-point loads. The adder pipeline has three stages. The number of stages in the multiplier pipeline depends on the precision of the source operands in the pipeline: two stages for double precision or three stages for single precision. The graphics unit has one stage for all precisions. The load pipeline has three stages for all precisions.
Figure 6-1. Pipelined Instruction Execution
Changing the FZ (flush zero), RM (rounding mode), or RR (result register) bits of \texttt{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 \texttt{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 \texttt{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 \texttt{fsr} and the the U bit being written into the \texttt{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 \texttt{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 \texttt{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 \texttt{fdest} is that which was specified by the instruction that initiated that operation.
If \( fdest \) is the same as \( fsrcl \) or \( fsrc2 \), the value being stored in \( fdest \) is used as the input operand. In this case, the precision of \( fdest \) must be the same as the source precision.

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

### 6.2.4 Transition between Scalar and Pipelined Operations

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

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

Note that the \( pfld \) pipeline is not affected by scalar \( fld \) and \( ld \) instructions.

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

### 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 instructions can be pipelined.

**Programming Notes**

Complications arise with sequences of pipelined multiplier operations with mixed single- and double-precision inputs because the pipeline length is different for the two precisions. The complications can be avoided by not mixing the two precisions; i.e., by 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, and the MRP bit of *fsr* for that next operation is *undefined*. 
6.3.1 Floating-Point Multiply

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmul.p fsrcl, fsrcl2, fdest</td>
<td>(Floating-Point Multiply)</td>
</tr>
<tr>
<td>fdest ← fsrcl1 × fsrcl2</td>
<td></td>
</tr>
<tr>
<td>pfmul.p fsrcl1, fsrcl2, fdest</td>
<td>(Pipelined Floating-Point Multiply)</td>
</tr>
<tr>
<td>fdest ← last stage multiplier result</td>
<td></td>
</tr>
<tr>
<td>Advance M pipeline one stage</td>
<td></td>
</tr>
<tr>
<td>M pipeline first stage ← fsrcl1 × fsrcl2</td>
<td></td>
</tr>
<tr>
<td>pfmul3.dd fsrcl1, fsrcl2, fdest</td>
<td>(Three-Stage Pipelined Multiply)</td>
</tr>
<tr>
<td>fdest ← last stage multiplier result</td>
<td></td>
</tr>
<tr>
<td>Advance 3-stage M pipeline one stage</td>
<td></td>
</tr>
<tr>
<td>M pipeline first stage ← fsrcl1 × fsrcl2</td>
<td></td>
</tr>
</tbody>
</table>

These instructions perform a standard multiply operation.

**Programming Notes**

Fsrcl must not be the same as fdest for pipelined operations. For best performance when the prior operation is scalar, fsrcl should not be the same as the fdest 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

<table>
<thead>
<tr>
<th>fmlow.dd fsr1, fsr2, fdest</th>
<th>(Floating-Point Multiply Low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fdest \leftarrow$ low-order 53 bits of $(fsr1 \text{ mantissa} \times fsr2 \text{ mantissa})$</td>
<td></td>
</tr>
<tr>
<td>$fdest$ bit 53 $\leftarrow$ most significant bit of $(fsr1 \text{ mantissa} \times fsr2 \text{ mantissa})$</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>frcp.p fsrcl, fdest</code></td>
<td>(Floating-Point Reciprocal) ( fdest \leftarrow \frac{1}{fsrcl} ) with absolute mantissa error ( &lt; 2^{-7} )</td>
</tr>
<tr>
<td><code>frsqr.p fsrcl, fdest</code></td>
<td>(Floating-Point Reciprocal Square Root) ( fdest \leftarrow \frac{1}{\sqrt{fsrcl}} ) with absolute mantissa error ( &lt; 2^{-7} )</td>
</tr>
</tbody>
</table>

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

**Traps**

The instructions `frcp` and `frsqr` cause the source-exception trap if `fsrcl` is zero. An `frsqr` causes the source-exception trap if `fsrcl < 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadd.p fsr1, fsr2, fdest</td>
<td>(Floating-Point Add) fdest ← fsr1 + fsr2</td>
</tr>
<tr>
<td>pfadd.p fsr1, fsr2, fdest</td>
<td>(Pipelined Floating-Point Add) fdest ← last stage adder result A pipeline first stage ← fsr1 + fsr2</td>
</tr>
<tr>
<td>fsub.p fsr1, fsr2, fdest</td>
<td>(Floating-Point Subtract) fdest ← fsr1 - fsr2</td>
</tr>
<tr>
<td>pfsub.p fsr1, fsr2, fdest</td>
<td>(Pipelined Floating-Point Subtract) fdest ← last stage adder result A pipeline first stage ← fsr1 - fsr2</td>
</tr>
<tr>
<td>famov.r fsr1, fdest</td>
<td>(Floating-Point Adder Move) fdest ← fsr1</td>
</tr>
<tr>
<td>pfamov.r fsr1, fdest</td>
<td>(Pipelined Floating-Point Adder Move) fdest ← last stage adder result A pipeline first stage ← fsr1</td>
</tr>
</tbody>
</table>

These instructions perform standard addition and subtraction operations.

The `famov` and `pfamov` instructions send `fsr1` through the floating-point adder, preserving the value of −0 (minus zero) when `fsr1` is −0. (Note that `(p)fadd.p fsr1, f0, fdest` may round −0 to +0, depending on the RM bits of `fsr`.) The `pfamov` instruction is used by the trap handler to restore pipeline states. `Fsrc2` for `(p)famov` must be encoded as `f0` by assemblers and compilers.

#### Programming Notes

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmov.ds fsr1, fdest</td>
<td>(Convert Double to Single) Equivalent to <code>famov.ds fsr1, fdest</code></td>
</tr>
<tr>
<td>pfmov.ds fsr1, fdest</td>
<td>(Pipelined Convert Double to Single) Equivalent to <code>pfamov.ds fsr1, fdest</code></td>
</tr>
</tbody>
</table>
Conversion from single to double is accomplished by `famov.sd` or `pfamov.sd`. In assembly language, this conversion can be specified by the `fmov` or `pfmov` pseudo-operation with the `.sd` suffix.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmov.sd fsrc1, fdest</code></td>
<td>Convert Single to Double</td>
</tr>
<tr>
<td>Equivalent to <code>famov.sd fsrc1, fdest</code></td>
<td></td>
</tr>
<tr>
<td><code>pfmov.sd fsrc1, fdest</code></td>
<td>Pipelined Convert Single to Double</td>
</tr>
<tr>
<td>Equivalent to <code>pfamov.sd fsrc1, fdest</code></td>
<td></td>
</tr>
</tbody>
</table>
6.4.2 Floating-Point Compares

\begin{verbatim}
\begin{tabular}{ll}
\textbf{pfgt.p} & \textbf{fsrc1, fsrc2, fdest} \\
& (Pipelined Floating-Point Greater-Than Compare) \\
& (Assembler clears R-bit of instruction) \\
& \textit{fdest} \leftarrow \text{last stage adder result} \\
& CC set if \textit{fsrc1} > \textit{fsrc2}, else cleared \\
& \text{Advance A pipeline one stage} \\
& \text{A pipeline first stage is undefined, but no result exception occurs} \\
\end{tabular}
\end{verbatim}

\begin{verbatim}
\begin{tabular}{ll}
\textbf{pfle.p} & \textbf{fsrc1, fsrc2, fdest} \\
& (Pipelined F-P Less-Than or Equal Compare) \\
& (Identical to \textbf{pfgt.p} except that \\
& \quad assembler sets R-bit of instruction.) \\
& \textit{fdest} \leftarrow \text{last stage adder result} \\
& CC cleared if \textit{fsrc1} \leq \textit{fsrc2}, else set \\
& \text{Advance A pipeline one stage} \\
& \text{A pipeline first stage is undefined, but no result exception occurs} \\
\end{tabular}
\end{verbatim}

\begin{verbatim}
\begin{tabular}{ll}
\textbf{pfeq.p} & \textbf{fsrc1, fsrc2, fdest} \\
& (Pipelined Floating-Point Equal Compare) \\
& \textit{fdest} \leftarrow \text{last stage adder result} \\
& CC set if \textit{fsrc1} = \textit{fsrc2}, else cleared \\
& \text{Advance A pipeline one stage} \\
& \text{A pipeline first stage is undefined, but no result exception occurs} \\
\end{tabular}
\end{verbatim}

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.

\textbf{pfgt.p} should be used for A > B and A < B comparisons. \textbf{pfle.p} should be used for A \geq B and A \leq B comparisons. \textbf{pfeq.p} should be used for A = B and A \equiv B comparisons.

\textbf{Traps}

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

\textbf{Programming Notes}

The only difference between \textbf{pfgt.p} and \textbf{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 \textbf{pfgt.p} and \textbf{pfeq.p}, it should clear CC; for \textbf{pfle.p}, it should set CC.

For best performance, a \texttt{bc} or \texttt{bnc} instruction should not directly follow a \textbf{pfgt} or \textbf{pfeq} instruction. Be sure, however, that intervening instructions do not change CC.
### 6.4.3 Floating-Point to Integer Conversion

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix.p fsrcl, fdest</code></td>
<td>(Floating-Point to Integer Conversion) fdest ← 64-bit value with low-order 32 bits equal to integer part of fsrcl rounded</td>
</tr>
<tr>
<td><code>pfix.p fsrcl, fdest</code></td>
<td>(Pipelined Floating-Point to Integer Conversion) fdest ← last stage adder result Advance A pipeline one stage A pipeline first stage ← 64-bit value with low-order 32 bits equal to integer part of fsrcl rounded</td>
</tr>
<tr>
<td><code>ftrunc.p fsrcl, fdest</code></td>
<td>(Floating-Point to Integer Truncation) fdest ← 64-bit value with low-order 32 bits equal to integer part of fsrcl</td>
</tr>
<tr>
<td><code>pftrunc.p fsrcl, fdest</code></td>
<td>(Pipelined Floating-Point to Integer Truncation) fdest ← last stage adder result Advance A pipeline one stage A pipeline first stage ← 64-bit value with low-order 32 bits equal to integer part of fsrcl</td>
</tr>
</tbody>
</table>

The instructions `fix`, `pfix`, `ftrunc`, and `pftrunc` must specify double-precision results. The low-order 32 bits of the result contain the integer part of `fsrcl` represented in two's-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. Assembler and compilers should encode `fsrcl` as `f0`.

**Traps**

The instructions `fix`, `pfix`, `ftrunc`, and `pftrunc` signal overflow if the integer part of `fsrcl` is bigger than what can be represented as a 32-bit two's-complement integer. Underflow and inexact are never signaled.
6.5 DUAL OPERATION INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfam.p fsrc1, fsrc2, fdest</td>
<td>(Pipelined Floating-Point Add and Multiply)</td>
</tr>
<tr>
<td></td>
<td>fdest ← last stage adder result</td>
</tr>
<tr>
<td></td>
<td>Advance A and M pipeline one stage (operands accessed before advancing pipeline)</td>
</tr>
<tr>
<td></td>
<td>A pipeline first stage ← A-op1 + A-op2</td>
</tr>
<tr>
<td></td>
<td>M pipeline first stage ← M-op1 × M-op2</td>
</tr>
<tr>
<td>pfsm.p fsrc1, fsrc2, fdest</td>
<td>(Pipelined Floating-Point Subtract and Multiply)</td>
</tr>
<tr>
<td></td>
<td>fdest ← last stage adder result</td>
</tr>
<tr>
<td></td>
<td>Advance A and M pipeline one stage (operands accessed before advancing pipeline)</td>
</tr>
<tr>
<td></td>
<td>A pipeline first stage ← A-op1 - A-op2</td>
</tr>
<tr>
<td></td>
<td>M pipeline first stage ← M-op1 × M-op2</td>
</tr>
<tr>
<td>pfmam.p fsrc1, fsrc2, fdest</td>
<td>(Pipelined Floating-Point Multiply with Add)</td>
</tr>
<tr>
<td></td>
<td>fdest ← last stage multiplier result</td>
</tr>
<tr>
<td></td>
<td>Advance A and M pipeline one stage (operands accessed before advancing pipeline)</td>
</tr>
<tr>
<td></td>
<td>A pipeline first stage ← A-op1 + A-op2</td>
</tr>
<tr>
<td></td>
<td>M pipeline first stage ← M-op1 × M-op2</td>
</tr>
<tr>
<td>pfmsm.p fsrc1, fsrc2, fdest</td>
<td>(Pipelined Floating-Point Multiply with Subtract)</td>
</tr>
<tr>
<td></td>
<td>fdest ← last stage multiplier result</td>
</tr>
<tr>
<td></td>
<td>Advance A and M pipeline one stage (operands accessed before advancing pipeline)</td>
</tr>
<tr>
<td></td>
<td>A pipeline first stage ← A-op1 - A-op2</td>
</tr>
<tr>
<td></td>
<td>M pipeline first stage ← M-op1 × M-op2</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Precision of Source of Multiplication</th>
<th>Precision of Source of Add or Subtract and Result of All Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ss</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>.sd</td>
<td>single</td>
<td>double</td>
</tr>
<tr>
<td>.dd</td>
<td>double</td>
<td>double</td>
</tr>
</tbody>
</table>

The instructions **pfmam** and **pfmsm** are identical to **pfam** and **pfsm** except that **pfmam** and **pfmsm** transfer the last stage result of the multiplier to fdest.
Six operands are required, but the instruction format 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 $fsrcl$ and subsequently supply that value to the M-pipeline in place of $fsrcl$.
  - 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 $fsrcl$.
- 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 $fsrcl$.
  2. Operand-2 of the multiplier can be $fsrc2$, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.
  3. Operand-1 of the adder can be $fsrcl$, the T-register, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.
  4. Operand-2 of the adder can be $fsrc2$, the last-stage result of the multiplier pipeline, or the last-stage result of the adder pipeline.

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

Note that the mnemonics pfam.p, pfsm.p, pfmanp, 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 pfmanp, and pfmsm.p classes from the pfam.p, and pfsm.p classes. Figure 6-4 explains how the rest of these mnemonics are derived.

Programming Notes

When $fsrcl$ goes to M-unit op1 or to KR or KI, $fsrcl$ must not be the same as fdest. For best performance when the prior operation is scalar and the M-unit op1 is $fsrcl$, $fsrcl$ should not be the same as the fdest of the prior operation.
When dual operation instructions are used in single-precision mode, all 64 bits of the T, KR, and KI registers are updated, but the values stored there are not converted to double-precision format (the exponent bias is not adjusted for double precision). Instead, zeros are inserted as pads in exponent bits 11:9 and as the fraction's least significant 29 bits (bits 28:0). All 64 bits of the T, KR, and KI registers can be initialized to zero using 3 single-precision r2apt.ss f0,f0,f0 instructions and 1 i2apt.ss f0,f0,f0. Because single-precision values are stored in these 64-bit registers in a format which does not conform to the standard for double-precision numbers, leaving a valid single-precision value in T, KR, or KI can cause floating-point traps if a double-precision operation is later performed referencing one of these registers. Likewise, valid double-precision values left in T, KR, or KI can cause traps if a single precision operation is later performed using one of these registers. Therefore, programs should clear T, KR, and KI before switching precisions.

Figure 6-2. Dual-Operation Data Paths
### Table 6-2. DPC Encoding

<table>
<thead>
<tr>
<th>DPC</th>
<th>PFAM Mnemonic</th>
<th>PFAM Mnemonic</th>
<th>M-Unit op1</th>
<th>M-Unit op2</th>
<th>A-Unit op1</th>
<th>A-Unit op2</th>
<th>T Load</th>
<th>K Load*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>r2p1</td>
<td>r2s1</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0001</td>
<td>r2pt</td>
<td>r2st</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>0010</td>
<td>r2ap1</td>
<td>r2as1</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>A result</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0011</td>
<td>r2apt</td>
<td>r2ast</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>A result</td>
<td>Yes</td>
</tr>
<tr>
<td>0100</td>
<td>i2p1</td>
<td>i2s1</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0101</td>
<td>i2pt</td>
<td>i2st</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>0110</td>
<td>i2ap1</td>
<td>i2as1</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>A result</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0111</td>
<td>i2apt</td>
<td>i2ast</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>A result</td>
<td>Yes</td>
</tr>
<tr>
<td>1000</td>
<td>rat1p2</td>
<td>rat1s2</td>
<td>KR</td>
<td>A result</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>1010</td>
<td>m12apm</td>
<td>m12asm</td>
<td>src1</td>
<td>src2</td>
<td>A result</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>1011</td>
<td>m12tpa</td>
<td>m12ttsa</td>
<td>src1</td>
<td>src2</td>
<td>T</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>1100</td>
<td>iat1p2</td>
<td>iat1s2</td>
<td>Ki</td>
<td>A result</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>1110</td>
<td>ia1p2</td>
<td>ia1s2</td>
<td>Ki</td>
<td>A result</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>1111</td>
<td>m12tspa</td>
<td>m12tsa</td>
<td>src1</td>
<td>src2</td>
<td>T</td>
<td>A result</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DPC</th>
<th>PFMAM Mnemonic</th>
<th>PFMSM Mnemonic</th>
<th>M-Unit op1</th>
<th>M-Unit op2</th>
<th>A-Unit op1</th>
<th>A-Unit op2</th>
<th>T Load</th>
<th>K Load*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>mr2p1</td>
<td>mr2s1</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0001</td>
<td>mr2pt</td>
<td>mr2st</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>0010</td>
<td>mr2mp1</td>
<td>mr2ms1</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0011</td>
<td>mr2mp2</td>
<td>mr2ms2</td>
<td>KR</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>0100</td>
<td>mi2p1</td>
<td>mi2s1</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0101</td>
<td>mi2pt</td>
<td>mi2st</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>0110</td>
<td>mi2mp1</td>
<td>mi2ms1</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>M result</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0111</td>
<td>mi2mp2</td>
<td>mi2ms2</td>
<td>Ki</td>
<td>src2</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>1000</td>
<td>mmrmt1p2</td>
<td>mmrmt1s2</td>
<td>KR</td>
<td>M result</td>
<td>src1</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>1010</td>
<td>mm12mpm</td>
<td>mm12msm</td>
<td>src1</td>
<td>src2</td>
<td>M result</td>
<td>src1</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>1011</td>
<td>mm12tpm</td>
<td>mm12ttsm</td>
<td>src1</td>
<td>src2</td>
<td>M result</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>1100</td>
<td>mimt1p2</td>
<td>mimt1s2</td>
<td>src1</td>
<td>src2</td>
<td>M result</td>
<td>T</td>
<td>M result</td>
<td>Yes</td>
</tr>
<tr>
<td>1110</td>
<td>mm12tpm</td>
<td>mm12ttsm</td>
<td>src1</td>
<td>src2</td>
<td>M result</td>
<td>T</td>
<td>M result</td>
<td>No</td>
</tr>
<tr>
<td>1111</td>
<td>mim1p2</td>
<td>mim1s2</td>
<td>Ki</td>
<td>M result</td>
<td>src1</td>
<td>src2</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* 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)
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 $f_{dest}$ is not $f_{0}$, then $f_{dest}$ must not be the same as $f_{src1}$ or $f_{src2}$.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fisub.w</td>
<td>Long-Integer Subtract</td>
</tr>
<tr>
<td>fsrc1, fsrc2, fdest</td>
<td>( fdest ← fsrc1 - fsrc2 )</td>
</tr>
<tr>
<td>pfiisub.w</td>
<td>Pipelined Long-Integer Subtract</td>
</tr>
<tr>
<td>fsrc1, fsrc2, fdest</td>
<td>last stage graphics result ← fsrc1 - fsrc2</td>
</tr>
<tr>
<td>fiadd.w</td>
<td>Long-Integer Add</td>
</tr>
<tr>
<td>fsrc1, fsrc2, fdest</td>
<td>( fdest ← fsrc1 + fsrc2 )</td>
</tr>
<tr>
<td>pfiadd.w</td>
<td>Pipelined Long-Integer Add</td>
</tr>
<tr>
<td>fsrc1, fsrc2, fdest</td>
<td>last stage graphics result ← fsrc1 + fsrc2</td>
</tr>
</tbody>
</table>

\(.w = .ss (32 \text{ bits}), \text{ or } .dd (64 \text{ bits})\)

The \texttt{fiadd} and \texttt{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, \texttt{fiadd} and \texttt{pfiadd} are used to implement the \texttt{fmov} and \texttt{pfmov} pseudoinstructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmov.ss</td>
<td>Single Move</td>
</tr>
<tr>
<td>fsrc1, fdest</td>
<td>Equivalent to \texttt{fiadd.ss fsrc1, f0, fdest}</td>
</tr>
<tr>
<td>pfmov.ss</td>
<td>Pipelined Single Move</td>
</tr>
<tr>
<td>fsrc1, fdest</td>
<td>Equivalent to \texttt{pfiadd.ss fsrc1, f0, fdest}</td>
</tr>
<tr>
<td>fmov.dd</td>
<td>Double Move</td>
</tr>
<tr>
<td>fsrc1, fdest</td>
<td>Equivalent to \texttt{fiadd.dd fsrc1, f0, fdest}</td>
</tr>
<tr>
<td>pfmov.dd</td>
<td>Pipelined Double Move</td>
</tr>
<tr>
<td>fsrc1, fdest</td>
<td>Equivalent to \texttt{pfiadd.dd fsrc1, f0, fdest}</td>
</tr>
</tbody>
</table>

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.

6.6.2.1 Z-BUFFER CHECK INSTRUCTIONS

A Z-buffer aids hidden-surface elimination by associating with a pixel a value that represents the distance of that pixel from the viewer. When painting a point at a specific pixel location, three-dimensional drawing algorithms calculate the distance of the point from the viewer. If the point is farther from the viewer than the point that is already represented by the pixel, the pixel is not updated. 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.

![Figure 6-5. PSR Fields for Graphics Operations](image-url)
Consider PM as an array of eight bits PM(0) .. PM(7), where PM(0) is the least-significant bit.

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

Consider fsrc1, fsrc2, and fdest as arrays of four 16-bit fields fsrc1(0) .. fsrc1(3), fsrc2(0) .. fsrc2(3), and fdest(0) .. fdest(3) where zero denotes the least-significant field.  

PM ← PM shifted right by 4 bits  
FOR i = 0 to 3  
DO  
  PM [i + 4] ← fsrc2(i) ≤ fsrc1(i) (unsigned)  
  fdest(i) ← smaller of fsrc2(i) and fsrc1(i)  
OD  
MERGE ← 0

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

Consider fsrc1, fsrc2, and fdest as arrays of four 16-bit fields fsrc1(0) .. fsrc1(3), fsrc2(0) .. fsrc2(3), and fdest(0) .. fdest(3) where zero denotes the least-significant field.  

PM ← PM shifted right by 4 bits  
FOR i = 0 to 3  
DO  
  PM [i + 4] ← fsrc2(i) ≤ fsrc1(i) (unsigned)  
  fdest ← last stage graphics result  
  last stage graphics result (i) ← smaller of fsrc2(i) and fsrc1(i)  
OD  
MERGE ← 0

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

Consider fsrc1, fsrc2, and fdest as arrays of two 32-bit fields fsrc1(0) .. fsrc1(1), fsrc2(0) .. fsrc2(1), and fdest(0) .. fdest(1) where zero denotes the least-significant field.  

PM ← PM shifted right by 2 bits  
FOR i = 0 to 1  
DO  
  PM [i + 6] ← fsrc2(i) ≤ fsrc1(i) (unsigned)  
  fdest(i) ← smaller of fsrc2(i) and fsrc1(i)  
OD  
MERGE ← 0

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

Consider fsrc1, fsrc2, and fdest as arrays of two 32-bit fields fsrc1(0) .. fsrc1(1), fsrc2(0) .. fsrc2(1), and fdest(0) .. fdest(1) where zero denotes the least-significant field.  

PM ← PM shifted right by 2 bits  
FOR i = 0 to 1  
DO  
  PM [i + 6] ← fsrc2(i) ≤ fsrc1(i) (unsigned)  
  fdest(i) ← last stage graphics result  
  last stage graphics result ← smaller of fsrc2(i) and fsrc1(i)  
OD  
MERGE ← 0

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

\[
\text{faddp } \text{frsrc1, frsrc2, fdest} \quad \text{(Add with Pixel Merge)}
\]

\[
fdest \leftarrow \text{frsrc1} + \text{frsrc2}
\]
Shift and load MERGE register from \text{frsrc1} + \text{frsrc2} as defined in Table 6-3

\[
\text{pfaddp } \text{frsrc1, frsrc2, fdest} \quad \text{(Pipelined Add with Pixel Merge)}
\]

\[
fdest \leftarrow \text{last stage graphics result}
\]

\[
\text{last stage graphics result} \leftarrow \text{frsrc1} + \text{frsrc2}
\]
Shift and load MERGE register from \text{frsrc1} + \text{frsrc2} as defined in Table 6-3

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

### Table 6-3. FADDP MERGE Update

<table>
<thead>
<tr>
<th>Pixel Size (from PS)</th>
<th>Fields Loaded From Result into MERGE</th>
<th>Right Shift Amount (Field Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>63..56, 47..40, 31..24, 15..8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>63..58, 47..42, 31..26, 15..10</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>63..56, 31..24</td>
<td>8</td>
</tr>
</tbody>
</table>
Figure 6-6 illustrates \texttt{faddp} when PS is set for 8-bit pixels. Since \texttt{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 8-bit fractional portion. The real numbers are rounded to 8 bits by truncation when they are loaded into the MERGE register. With each \texttt{faddp} instruction, the MERGE register is shifted right by 8 bits. Two \texttt{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 \texttt{faddp} instructions.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{faddp_8bit}
\caption{FADDP with 8-Bit Pixels}
\end{figure}
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 a 6-bit integer portion and a 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](image)

63: TRUNCATE 63: TRUNCATE 63: TRUNCATE 63: TRUNCATE
Figure 6-8 illustrates \texttt{faddp} when PS is set for 32-bit pixels. Since \texttt{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 a 24-bit fractional portion. The real numbers are rounded to 8 bits by truncation when they are loaded into the MERGE register. With each \texttt{faddp}, the MERGE register is shifted right by 8 bits. Normally, three \texttt{faddp} instructions are executed consecutively, one for each color represented in a pixel. The shifting of MERGE causes the results of consecutive \texttt{faddp} instructions to be accumulated in the MERGE register.

![Figure 6-8. FADDP with 32-Bit Pixels](image-url)
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.

![Figure 6-9. FADDZ with 16-Bit Z-Buffer](image-url)
Normally, two faddz instructions are executed consecutively. The shifting of MERGE causes the results of consecutive faddz instructions to be accumulated in the MERGE register.

32-bit Z-buffers can use 32-bit or 64-bit distance interpolation. For 32-bit interpolation, no 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 fladd instructions. The merging is implemented with the 32-bit move fmov.ss fsrl, fdest.
6.6.2.4 OR WITH MERGE REGISTER

<table>
<thead>
<tr>
<th>form.dd fsrc1, fdest</th>
<th>(OR with MERGE Register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdest ← fsrc1 OR MERGE</td>
<td></td>
</tr>
<tr>
<td>MERGE ← 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pform.dd fsrc1, fdest</th>
<th>(Pipelined OR with MERGE Register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdest ← last stage graphics result</td>
<td></td>
</tr>
<tr>
<td>last stage graphics result ← fsrc1 OR MERGE</td>
<td></td>
</tr>
<tr>
<td>MERGE ← 0</td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
\begin{align*}
\text{fxfr } &\text{ fsrc1, idest} \\
\text{idest} &\leftarrow \text{ fsrc1}
\end{align*}
\]

(Transfer F-P to Integer Register)

The 32-bit floating-point register selected by \(fsrc1\) is stored into the (32-bit) integer
register selected by \(idest\). Assemblers and compilers should encode \(fsrc2\) as \(f0\).

Programming Notes

This scalar instruction is performed by the graphics unit. When it is executed, the result
in the graphics-unit pipeline is lost. However, executing this instruction does not impact
performance, even if the next instruction is a pipelined operation whose \(fdest\) is nonzero
(refer to Section 6.2).

For best performance, \(idest\) should not be referenced in the next instruction, and \(fsrc1\)
should not reference the result of the prior instruction if the prior instruction is scalar.
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 \texttt{d.} prefix or by using the Assembler directives \texttt{.dual ... .end-dual}. 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.

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

![Figure 6-11. Dual-Instruction Mode Transitions (1 of 2)](image_url)
The recommended floating-point NOP for dual-instruction mode is \texttt{shrd r0,r0,r0}, because this instruction does not affect the states of the floating-point pipelines. Even though this is a core instruction, bit 9 is interpreted as the dual-instruction mode control bit. In assembly language, this instruction is specified as \texttt{fnop} or \texttt{d.fnop}. Traps are not reported on \texttt{fnop}. Because it is a core instruction, \texttt{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 \texttt{bc} or \texttt{bnc} is paired with a floating-point compare, the branch tests the value of the condition code prior to the compare.

2. If an \texttt{ixfr}, \texttt{fld}, or \texttt{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 \texttt{fst} or \texttt{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 an 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**, **pfle** or **pfeq**, CC is set according to the result of the **pfgt**, **pfle** 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, calli, ixfr**, or any instruction that updates an integer register (including autoincrement indexing). Furthermore, the core instruction cannot be a **fld, fst, pst, or pfld** that uses as **isrc1** or **isrc2** the same register as the **idest** of the **fxfr**.

7. A **bri** must not be executed in dual-instruction mode if any trap bits are set.

8. When the core operation is **bc.t** or **bnc.t**, the floating point operation cannot be **pfeq, pfle** or **pfgt**. The floating point operation in the sequentially following instruction pair cannot be **pfeq, pfle** or **pfgt**, either.

9. A transition to or from dual-instruction mode cannot be initiated on the instruction following a **bri**.
10. An *inxfr, *fld, or *pfl d cannot update the same register as the companion floating-point instruction unless the destination is *f0 or *f1. No overlap of register destinations is permitted; for example, the following instructions must not be paired:

```
d.fmul.ss  f9, f10, f5
fld.q f4
```

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

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

1. Copies U (user mode) of the psr into PU (previous U).
2. Copies IM (interrupt mode) into PIM (previous IM).
3. Sets U to zero (supervisor mode).

<table>
<thead>
<tr>
<th>Type</th>
<th>Indication</th>
<th>Caused by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>psr, epsr</td>
<td>Condition</td>
</tr>
<tr>
<td></td>
<td>fsr</td>
<td>Instruction</td>
</tr>
<tr>
<td>Instruction Fault</td>
<td>IT OF IL</td>
<td>Software traps Missing unlock</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trap, intovr Any</td>
</tr>
<tr>
<td>Floating Point Fault</td>
<td>FT</td>
<td>Floating-point source exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Floating-point result exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>underflow inexact result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any M- or A-unit except fmlow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any M- or A-unit except fmlow, pfgt,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pfl, and pfeq Reported on any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F-Pinstruction plus pst, fst, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sometimes fid, pfd, ixfr</td>
</tr>
<tr>
<td>Instruction Access Fault</td>
<td>IAT</td>
<td>Address translation exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>during instruction fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any</td>
</tr>
<tr>
<td>Data Access Fault</td>
<td>DAT*</td>
<td>Load/store address translation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Misaligned operand address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand address matches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>db register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any load/store</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any load/store</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any load/store</td>
</tr>
<tr>
<td>Interrupt</td>
<td>IN</td>
<td>External interrupt</td>
</tr>
<tr>
<td>Reset</td>
<td>No trap bits set</td>
<td>Hardware RESET signal</td>
</tr>
</tbody>
</table>
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.

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

The processor begins executing the trap handler by transferring execution to virtual address 0xFFFFFFFF00. 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, OF) 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. Further, the trap handler must ensure that no trap may occur once the restoration of the initial state (described in Section 7.2.3) has begun prior to returning from the trap handler. The branch-indirect instruction is sensitive to the trap-type bits; therefore, clearing the trap-type bits allows normal indirect branches to be performed within the trap handler.

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

1. The fir.
2. The psr.
3. The epsr.
4. The fsr.

5. The dirbase register.

6. The MERGE register.


8. Any of the four pipelines (refer to Section 7.9).

9. The floating-point and integer register files.

7.2.2 Inside the Trap Handler

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

1. For all types of traps, the trap handler must check the IL bit of epsr to determine if a locked sequence is being interrupted.

2. The trap handler must execute ld.c fir, isrc1 once for each trap. Failure to do so prevents fir from receiving the address of the next trap.

7.2.3 Returning from the Trap Handler

Returning from a trap handler involves the following steps:

1. Restoring the pipeline states, including the fsr, KR, KI, T, and MERGE registers, where necessary.

2. Subtracting src1 from src2, when a data-access fault occurred on an autoincrementing load/store instruction and a floating-point trap did not also occur.

3. Determining where to resume execution by inspecting the instruction at fir − 4. The details for this determination are given in Section 7.2.3.1.

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

5. 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 7.2.3.2. The trap handler must ensure that no trap occurs between the st.c to the psr and the indirect branch that exits the trap handler.
6. Executing an indirect branch to the resumption address, making sure that at least one of the trap bits is set in the psr. Neither the indirect branch nor the following instruction may be executed in dual-instruction mode.

7. Restoring the register that holds the resumption address. (This is executed before the delayed indirect branch is completed.)

**7.2.3.1 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 (bct or bnc.t), the instruction at fir is a pfgt, plge, 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.

When examining fir - 4, take care not to cause a page fault. If the location in fir is at the beginning of a page, then fir - 4 is in the prior page. If the prior page is not present, then examining fir - 4 will cause a page fault. In this case, however, the instruction at fir - 4 could not have been a delayed control instruction; therefore it is not necessary to examine fir - 4. Note that, when determining whether the prior page is not present, it is necessary to inspect both the page table and its page directory entry.

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. 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 of the code section. Furthermore, this nop should not be the target of any branch or call.
7.2.3.2 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 or an intovr instruction and there are no other trap conditions. In this case, the floating-point instruction has already been executed.
- The trap was caused by a source exception on any floating-point instruction (except when a pfgt, pfle, or pfeq follows a conditional branch, as already explained in Section 7.2.3.1). 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.

1. By the trap instruction. Refer to the trap instruction in Chapter 5.

2. By the intovr instruction. The trap occurs only if OF in epsr is set when intovr is executed. The trap handler should clear OF before returning. Refer to the intovr instruction in Chapter 5.

3. By the lack of an unlock instruction (and subsequent load or store) within 30-33 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 30-33 instructions of the trap indicates a programming error. Refer to the lock instruction in Chapter 5.

Note that trap and intovr should not be used within a locked sequence; otherwise, it would not be possible to distinguish among the above cases.

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 signaling NaNs).
   - Division by zero.
   - Operations on quiet NaNs, denormals and infinities. (These data types are implemented by software.)
2. **Result exceptions.** This class includes the overflow, underflow, and inexact exceptions defined by the IEEE standard.

Software available from Intel provides the IEEE standard default handling for all these exceptions.

The floating-point fault occurs only on floating-point instructions, and on `pst`, `fst`, `fld`, `pfld`, and `ixfr`. No floating-point fault occurs when `pst`, `fst`, `fld`, `pfld`, or `ixfr` transfers an operand that is not a valid floating-point value.

### 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 `fdest`, in the case of a scalar operation, or replace the appropriate first-stage result, in the case of a pipelined operation.

Note that, in the following sequence, inappropriate use of the FTE bit of the `fsr` can produce an invalid operand that does not cause a source exception:

1. Floating-point traps are masked by clearing the FTE bit.
2. An dual-operation instruction causes underflow or overflow leaving an invalid result in the T register.
3. Floating-point traps are enabled by setting the FTE bit.
4. The invalid result in the T register is used as an operand of a subsequent instruction.

Even though the result of an operation would normally cause a source exception, it can be inserted into the pipeline as follows:

1. Disable traps by clearing FTE.
2. Perform a pipelined add of the value with zero or a multiply by one.
3. Set the result-status bits of `fsr` to “normal” by loading `fsr` with the U-bit set and zeros in the appropriate unit’s result-status bits. The other unit’s status must be set to the saved status for the first pipeline stage.
4. Reenable traps by setting FTE.

5. Set KNF in the psr to avoid reexecuting the instruction.

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

The trap handler should process result exceptions as described below and reexecute the instruction before processing source exceptions.

### 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. The instructions fld, pfld and ixfr report result exceptions when the fdest of these instructions overlap the fdest of the instruction that caused the exception. When a trap occurs, the last stage of the affected unit contains the result of the scalar operation. The result is also written to the register indicated by the RR field of the psr.

- **Pipelined operations.** Result exceptions are reported when the result is in the last stage and the next floating-point, fst.x or pst.x (and sometimes fld, pfld, ixfr) instruction is executed. The instructions fld, pfld and ixfr report result exceptions when the fdest of these instructions overlap the fdest of the instruction that caused the exception. 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 \texttt{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 \texttt{pfadd} is started, a trap occurs and MO is set.

For scalar operations, the RR bits of \texttt{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 \texttt{fsr} from modifying the RR bits. Prevention may take one of the following forms:

- Before any store to \texttt{fsr} when a result exception may be pending, execute a dummy floating-point operation to trigger the result-exception trap.
- Always read from \texttt{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 mantissa as the true result and has an exponent which is the low-order bits of the true result. The trap handler can inspect the 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.

\textbf{7.5 INSTRUCTION-ACCESS FAULT}

This trap results from a page-not-present exception during instruction fetch or an attempt to access a supervisor-level page while in user mode. Protection checking for instruction accesses occurs only during instruction fetches from external memory (i.e., I-cache miss).

\textbf{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 only to one of the following causes:

- An attempt is being made to write to a page whose D-bit is clear.
- A memory operand is misaligned (is not located at an address that is a multiple of the length of the data).
- The address stored in the \texttt{db} (data breakpoint) register is equal to one of the addresses spanned by the operand.
- The operand is in a not-present page.
TRAPS AND INTERRUPTS

- A memory access is being attempted in violation of the memory protection scheme defined in Chapter 4.
- A-bit is zero during address translation within a locked sequence.

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 0xFFFFFFF00. 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, PBM, and BE 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. Refer to Table 7-2 for a summary of these initial settings.

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.

Table 7-2. Register and Cache Values after Reset

<table>
<thead>
<tr>
<th>Registers</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Registers</td>
<td>Undefined</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>Undefined</td>
</tr>
<tr>
<td>psr</td>
<td>U, IM, BR, BW = 0; others = undefined</td>
</tr>
<tr>
<td>epsr</td>
<td>IL, WP, PBM, BE = 0; Processor Type, Stepping Number, DCS are read only; others are undefined</td>
</tr>
<tr>
<td>db</td>
<td>Undefined</td>
</tr>
<tr>
<td>dirbase</td>
<td>DPS, BL, ATE = 0</td>
</tr>
<tr>
<td>fir</td>
<td>Undefined</td>
</tr>
<tr>
<td>tsr</td>
<td>Undefined</td>
</tr>
<tr>
<td>KR, KI, MERGE</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Caches</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Cache</td>
<td>Flushed</td>
</tr>
<tr>
<td>Data Cache</td>
<td>Undefined. All modified bits = 0.</td>
</tr>
<tr>
<td>TLB</td>
<td>Flushed</td>
</tr>
</tbody>
</table>
Reset code must initialize the floating-point pipeline states to zero, using dummy \texttt{pfadd}, \texttt{pfmul}, \texttt{pfiadd} instructions. Floating-point traps must be 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 \texttt{PU} and a trap bit so that an indirect branch instruction will copy \texttt{PU} to \texttt{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 \texttt{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 \texttt{fsr} helps determine how many stages are in the multiplier pipeline.

3. The result-status bits for the first two stages.


#### 7.9.2 Load Pipeline

The pipeline state for \texttt{pfld} instructions can be saved by performing three \texttt{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 \texttt{fsr}. Note that the load pipeline must be saved before changing the BE bit.

The load pipeline can be restored performing three \texttt{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 \texttt{pfiadd f0, f0, fdest}. The only other state information for the graphics unit resides in the PM bits of \texttt{psr}, the IRP bit of the \texttt{fsr}, and in the MERGE register. Store the MERGE register with a \texttt{form} instruction. Restore the MERGE register by using \texttt{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 \texttt{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 \texttt{fsr} always reflect the last-stage result status and cannot be directly set by software.
// The symbols Mres3, Ares3, Mres2, Ares2, Mres1, Ares1, Ires1, Lres1m 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 f0, Fsr3 // save third stage result status
andnot 0x20, Fsr3, Temp // clear FTE bit
st.c Temp, f0 // 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 f0, 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 f0, 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
pfadd.dd f0, f0, Ires1 // save vector-integer result
andnot 0x2C, Fsr1, Temp // clear RM, clear FTE
or d 4, Temp, Temp // set RM=01, round down, so -0
st.c Temp, f0 // is preserved when added to f0
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 f3, Mergelo32
fxfr f3, Mergehi32

Example 7-1. Saving Pipeline States
TRAPS AND INTERRUPTS

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.

Example 7-2. Restoring Pipeline States (1 of 2)
Example 7-2. Restoring Pipeline States (2 of 2)
CHAPTER 8
PROGRAMMING MODEL

This chapter defines standards for compiler and assembly language conventions 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.

NOTE

The dividing point between locals and parameters in the floating-point registers is now set at 8. Earlier software used a dividing point at 16.

Table 8-1. Register Allocation

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Left Unchanged by a Subroutine?</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>Always zero</td>
<td>Yes</td>
</tr>
<tr>
<td>r1</td>
<td>Return address</td>
<td>No</td>
</tr>
<tr>
<td>r2</td>
<td>Stack pointer</td>
<td>Note 1</td>
</tr>
<tr>
<td>r3</td>
<td>Frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>r4-r15</td>
<td>Local values</td>
<td>Yes</td>
</tr>
<tr>
<td>r16-r27</td>
<td>Parameters and temporaries</td>
<td>No</td>
</tr>
<tr>
<td>r16</td>
<td>Return value</td>
<td>No</td>
</tr>
<tr>
<td>r28</td>
<td>Memory parameter pointer</td>
<td>No</td>
</tr>
<tr>
<td>r28-r30</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>r31</td>
<td>Addressing temporary</td>
<td>No</td>
</tr>
<tr>
<td>f0-f1</td>
<td>Always zero</td>
<td>Yes</td>
</tr>
<tr>
<td>f2-f7</td>
<td>Local values</td>
<td>Yes</td>
</tr>
<tr>
<td>f8-f15</td>
<td>Parameters and temporaries</td>
<td>No</td>
</tr>
<tr>
<td>f8-f9</td>
<td>Return value</td>
<td>No</td>
</tr>
<tr>
<td>f16-f31</td>
<td>Temporaries</td>
<td>No</td>
</tr>
</tbody>
</table>

NOTE:
1. The stack pointer is normally kept unchanged across a subroutine call. However, some subroutines may allocate stack space and return with a different value in r2.
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 register. If a subroutine has an integral or pointer return value, it loads the return value into r16 before returning control to the caller.

Register r1 is the required return-address register, because the call and calli instructions use it to save the return address. Subroutines are therefore required to use the value in r1 to return to the caller. If a subroutine saves r1, it may then use it as a temporary until it returns.
A separate addressing temporary register (r31) is allocated to allow construction of 32-bit address temporaries. Assemblers may use r31 by default to construct 32-bit addresses from 16-bit literals.

If there are memory parameters, either because there are more parameters than will fit in the registers or because there are structure parameters, they should be put in the caller's stack frame properly aligned. Register r28 is set to point to this area in memory by the caller.

8.1.2 Floating-Point Registers

Floating-point and 64-bit integer values in the floating-point registers must use f8-f15 when passed by value. The leftmost such parameter is passed in f8-f9; the rest in successively higher-numbered registers. Single-precision parameters use one register, double-precision parameters use two properly aligned registers. A single-precision floating-point value can be converted to double-precision with the fmov.sd fx, fy pseudoinstruction.

Parameters beyond f15 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

Integer and floating-point parameter registers are allocated independently. If parameter N (N is less than or equal to 12) is an integral parameter, then it is placed in integer register 16 + N, with no effect on the floating-point register usage. If parameter M is the first floating-point parameter, then it is placed in the register pair f8 and f9 if it is double precision, or in register f8 if it is single precision. If parameter M + 1 is the second floating-point parameter, then it is placed in register pair f10 and f11 if it is double precision, regardless of the type of the first floating-point parameter. If parameter M + 1 is single precision, then it is placed in register f9 if the first floating-point parameter is single precision, or in register f10 if the first floating-point parameter is double precision.

NOTE
The conventions in Sections 8.1.1 through 8.1.3 remain tentative.

8.1.4 Variable Length Parameter Lists

Parameter passing in registers can handle a variable number of parameters. The C programming language uses a special method to access variable-count parameters. The stdarg.h and varargs.h files define several functions to get at these parameters in a way that is independent of stack growth direction and of whether parameters are passed in
registers or on the stack. A subroutine with variable parameters must use the `va_start` macro to set up a data structure before the parameters can be used. The `va_arg` macro must be used to access the successive parameters. 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 known to be misaligned, the compiler should generate separate instructions to access the data in smaller units that will not generate misaligned-data traps. Accessing 16-bit misaligned data requires two byte loads plus a shift. Storing a 32-bit misaligned data item may require 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`.

### 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™ architecture. 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.

```assembly
andnot 3, r8, r9 // Get address aligned on 4-byte boundary
ld.l 0(r9), r10 // Get low 32-bit value
ld.l 4(r9), r11 // Get high 32-bit value
and 3, r8, r9 // Get byte offset in 8-byte field
shr 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
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 sp-relative addressing to get at values on the stack. If the compiler knows that it does not call subroutines that leave sp in an altered state when they return, then no frame pointer is necessary.

The stack must be kept aligned on 16-byte boundaries to keep data arrays aligned. Each subroutine must use stack space in multiples of 16 bytes. The frame pointer r3 (called fp in assembly language) need not point to a 16-byte boundary, as long as the compiler keeps data correctly aligned when assigning positions relative to fp.

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.

### 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.

![Figure 8-2. Stack Frame Format](image-url)
PROGRAMMING MODEL

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

```
// 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 -1, -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 8-3. Subroutine Entry and Exit without Frame Pointer

```
// Subroutine entry
addu -Locals, sp, sp // 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, sp, sp // Restore stack pointer
```

Languages such as Pascal that need to maintain activation records on the stack can put them below the frame pointer in the program-specific area. The frame pointer is optional. All stack references can be made relative to sp. The code example in Example 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.

8.3.2 Dynamic Memory Allocation on the Stack

Consider a function `alloca` that allocates space on the stack and returns a pointer to the space. The allocated space is lost when the caller returns. The function `alloca` could be implemented as shown in Example 8-4. For any function calling `alloca` a separate stack pointer and frame pointer are required.

8.4 MEMORY ORGANIZATION

Figure 8-3 illustrates 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
Example 8-4. Possible Implementation of alloca

```
alloca::
    // r1b has size requested
    adds 15, r1b, r1b  // Round size to 0 mod 1b
    andnot 15, r1b, r1b  // Adjust stack downwards
    subs sp, r1b, sp  // Return to caller after next instruction
    bri r1
    mov sp, r1b  // Set return value to allocated space
```

Figure 8-3. Example Memory Layout
operating system share a single four-gigabyte address space. The illustrated memory map assumes paging is being used to place DRAM-resident code in the upper 256 Mbytes of the address space.

In this example, 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 a \texttt{0(r0)} assembly-language address expression) 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 \texttt{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.
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 r1b
shl 24, r1b, r1b  // left-justify
shra 24, r1b, r1b  // 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...

\[
G_{\text{new}} = G_{\text{old}}(2 - G_{\text{old}}Y).
\]
\[
E_{\text{new}} = 2(E_{\text{old}})^2.
\]

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.

```assembly
// 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 src1
fsub.ss f5, f4, f4 // 2 - guess * divisor
fmul.ss f6, f3, f5 // second guess * dividend
fmul.ss f4, f5, f3 // result = second guess * dividend
```

Example 9-3. Single-Precision Divide
9.3 DOUBLE-PRECISION DIVIDE

Example 9-4 computes \( Z = X \div Y \) 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, f6 // 2 - guess * divisor
fmul.dd f6, f8, f8 // second guess has 2**-15 error
fmul.dd f4, f8, f8 // avoid using f6 as src1
fsub.dd f10, f8, f6 // 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, f6 // 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, eleven clocks are required. Seven clocks can be overlapped with other operations.

```
// INTEGER MULTIPLY
// The multiplier is in r4
// The multiplicand is in r5
// The product is left in rb
// The registers f2, f4, and fb are used as temporaries.

ixfr r4, f2
ixfr r5, f4
// Two core instructions can be inserted here without penalty.
fmlow.dd f4, f2, fb
// Four core instructions can be inserted here without penalty.
fxfr fb, rb
// 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 $2^{52} + 2^{31} (0x4330_0000_8000_0000)$.

The conversion requires 7 clocks if the result is referenced in the next instruction. Three clocks can be overlapped with other operations. If a single-precision result is required, add an `famov.ds` instruction at the end.

```assembly
// 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
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).

```plaintext
// SIGNED INTEGER DIVIDE
// The denominator is in r4
// The numerator is in r5
// The quotient is left in rb
// The remainder is left in r7
// The registers f2 through f11 are used as temporaries.
// Convert Denominator and Numerator
fld.d two52two31, fb  // load constant 2**52 + 2**31
xorh 0x8000, r4, r19  //
ixfr r19 f4  //
fmov.ss f7, f5  //
oxrh 0x8000, r5, r20  //
fsub.dd f4, fb, f4  //
ixfr r20, f2  //
fmov.ss f7, f3  //
fsub.dd f2, fb, f2  //
// Do Floating-Point Divide
fld.d fdtwo, f10  // load floating-point two
frcp.dd f4, fb  // first guess has 2**-8 error
fmul.dd f4, fb, f8  // guess * divisor
fsub.dd f10, f8, f8  // 2 - guess * divisor
fmul.dd fb, f8, f8  // second guess has 2**-15 error
fmul.dd f4, f8, f8  // avoid using fb as src1
fsub.dd f10, f8, f8  // 2 - guess * divisor
fmul.dd fb, f8, f8  // third guess has 2**-29 error
fmul.dd f4, f8, f8  // avoid using fb as src1
fsub.dd f10, f8, f8  // 2 - guess * divisor
fmul.dd fb, f8, f8  // guess * dividend
fmul.dd f8, f8, 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, rb  // 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
// r1b - address of destination string

copy_string::
  ld.b 0(r17), r2b  // Load one character
  bte 0, r2b, 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, r1b, r18  // Use constant offset to avoid
                        // incrementing two indexes
loop::
  st.b r2b, 0(r1b)  // Store previous character
  adds 1, r1b, r1b  // Bump common index
  or r0, r27, r2b  // Test for NUL character
  bnc.t loop       // If not NUL, branch after loading
  ld.b r18(r1b), r27  // next character. r18(r1b) = 0(r17)
done::
  bri r1          // Return after storing
  st.b r2b, 0(r1b)  // 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.

```plaintext
// 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 9-11 illustrates pipeline usage for a simple single-precision matrix operation: the dot product of a $1 \times 8$ row matrix $A$ with an $8 \times 1$ column matrix $B$. For the purpose of tracking values through the pipelines, assume that the actual matrices to be multiplied have the following values:

$$A = \begin{bmatrix} 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 \end{bmatrix} \quad B = \begin{bmatrix} 8.0 \\ 7.0 \\ 6.0 \\ 5.0 \\ 4.0 \\ 3.0 \\ 2.0 \\ 1.0 \end{bmatrix}$$

Assume further that the two matrices are already loaded into registers thus:

$$A: \quad f4 = 1.0 \quad f5 = 2.0 \quad f6 = 3.0 \quad f7 = 4.0 \quad f8 = 5.0 \quad f9 = 6.0 \quad f10 = 7.0 \quad f11 = 8.0$$
$$B: \quad f12 = 8.0 \quad f13 = 7.0 \quad f14 = 6.0 \quad f15 = 5.0 \quad f16 = 4.0 \quad f17 = 3.0 \quad f18 = 2.0 \quad f19 = 1.0$$

The calculation to perform is $1.0 \times 8.0 + 2.0 \times 7.0 + \ldots + 8.0 \times 1.0$ — 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 puts zeros in all stages of the adder pipeline.

Because the adder pipeline has three stages, the flushing phase produces three partial results that must be added together.
### Example 9-10. Pipelined Dual-Operation Instruction

<table>
<thead>
<tr>
<th>Stages</th>
<th>Multiplier</th>
<th>Adder</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Priming phase</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m12apm.ss f4, f12, f0 // 1*8</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>m12apm.ss f5, f13, f0 // 2*7</td>
<td>1*8</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>m12apm.ss f6, f14, f0 // 3*6</td>
<td>2*7</td>
<td>6</td>
<td>??</td>
</tr>
<tr>
<td>pfadd.ss f0, f0, f0 //</td>
<td>0</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>pfadd.ss f0, f0, f0 //</td>
<td>0</td>
<td>0</td>
<td>??</td>
</tr>
<tr>
<td>pfadd.ss f0, f0, f0 //</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Continuous operation phase |
| m12apm.ss f7, f15, f0 // 4*5 | 3*6 | 14 | 8+0 | 0+0 | 0 | Discard |
| m12apm.ss f8, f16, f0 // 5*4 | 4*5 | 16 | 14+0 | 8+0 | 0 | Discard |
| m12apm.ss f9, f17, f0 // 6*3 | 5*4 | 20 | 18+0 | 14+0 | 8 | Discard |
| m12apm.ss f10, f18, f0 // 7*2 | 6*3 | 20 | 20+6 | 14+0 | 14 | Discard |
| m12apm.ss f11, f19, f0 // 8*1 | 7*2 | 16 | 20+14 | 20+6 | 18 | Discard |

// For larger matrices, include more instructions here

| Flushing phase |
| m12apm.ss f0, f0, f0 // 0*0 | 0*0 | 14 | 16+18 | 20+14 | 28 | Discard |
| m12apm.ss f0, f0, f0 // 0*0 | 0*0 | 0 | 16+28 | 18+18 | 34 | Discard |
| m12apm.ss f0, f0, f0 // 0*0 | 0*0 | 0 | 6+34 | 14+28 | 36 | Discard |

// Sum the partial results
| pfadd.ss f0, f0, f20 // | 0+0 | 8+34 | 42 | f20=34 |
| pfadd.ss f20, f21, f21 // | 0+0 | 0+0 | 42 | f21=42 |
| pfadd.ss f0, f0, f20 // | 0+0 | 0+0 | 42 | f20=42 |
| pfadd.ss f0, f0, f21 // | 0+0 | 0+0 | 78 | Discard |
| fadd.ss f20, f21, f20 // | 0+0 | 0+0 | 0 | f21=78 |

fadd.ss f20, f21, f20 // f20=120
9.10 PIPELINING OF DOUBLE-PRECISION DUAL OPERATIONS

Example 9-11 illustrates how pipeline usage for a double-precision differs from the single-precision Example 9-10. Example 9-11 performs the dot product of a $1 \times 6$ row matrix $A$ with an $6 \times 1$ column matrix $B$. For the purpose of tracking values through the pipelines, assume that the actual matrices to be multiplied have the following values:

$$A = \begin{bmatrix} 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, \end{bmatrix} \quad B = \begin{bmatrix} 6.0 \\ 5.0 \\ 4.0 \\ 3.0 \\ 2.0 \\ 1.0 \end{bmatrix}$$

Assume further that the two matrices are already loaded into registers thus:

$\begin{array}{ll}
A: & f4:f5 = 1.0 \\
    & f6:f7 = 2.0 \\
    & f8:f9 = 3.0 \\
    & f10:f11 = 4.0 \\
    & f12:f13 = 5.0 \\
    & f14:f15 = 6.0 \\
B: & f16:f17 = 6.0 \\
    & f18:f19 = 5.0 \\
    & f20:f21 = 4.0 \\
    & f22:f23 = 3.0 \\
    & f24:f25 = 2.0 \\
    & f26:f27 = 1.0 \\
\end{array}$

Example 9-11 differs from Example 9-10 in that, with double precision, the multiplier pipeline has only two stages; therefore the priming and flushing phases use fewer instructions.
// PIPELINED DUAL-OPERATION INSTRUCTION -- DOUBLE PRECISION

<table>
<thead>
<tr>
<th></th>
<th>Multiplier Stages</th>
<th>Adder Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 1 2 3</td>
<td>0</td>
</tr>
</tbody>
</table>

// Priming phase
m12apm.dd f4, f16, f0 // 1*6 ?? ?? ?? ?? Discard
m12apm.dd f6, f18, f0 // 2*5 1*6 ?? ?? ?? ?? Discard
pfadd.dd f0, f0, f0 // 0 ?? ?? ?? Discard
pfadd.dd f0, f0, f0 // 0 0 ?? ?? Discard
pfadd.dd f0, f0, f0 // 0 0 0 Discard

// Continuous operation phase
m12apm.dd f8, f20, f0 // 3*4 2*5 6+0 0 0 Discard
m12apm.dd f10, f22, f0 // 4*3 3*4 10+0 6+0 0 Discard
m12apm.dd f12, f24, f0 // 5*2 4*3 12+0 10+0 6 Discard
m12apm.dd f14, f26, f0 // 6*1 5*2 12+6 12+0 10 Discard

// For larger vectors, include more instructions here

// Flushing phase
m12apm.dd f0, f0, f0 // 0*0 6*1 10+10 12+6 12 Discard
m12apm.dd f0, f0, f0 // 0*0 6*0 6+12 10+10 18 Discard

// Three partial sums are now in the adder pipeline.
pfadd.dd f0, f0, f28 // 0 6+12 20 0 0 20 = f28
pfadd.dd f28, f30, f30 // 18+20 0 18 0 18 = f30
pfadd.dd f0, f0, f28 // 0 18+20 0 0 18 = f28
pfadd.dd f0, f0, f0 // 0 0 35 Discard
pfadd.dd f0, f0, f30 // 0 0 0 35 = f30
fadd.dd f28, f30, f30 // 0 0 56

Example 9-11. Pipelined Double-Precision Dual Operation
9.11 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 already loaded in registers. Example 9-12 goes one step further and shows how to maintain the high throughput of the floating-point unit while simultaneously loading the data from main memory and controlling the logical flow.

The problem is to sum the single-precision elements of an arbitrarily long vector. The procedure uses dual-instruction mode to overlap loading, decision making, and branching with the basic pipelined floating-point add instruction pfadd.ss. To make obvious the pairing of core and floating-point instructions in dual-instruction mode, the listing in Example 9-12 shows the core instruction of a dual-mode pair indented with respect to the corresponding floating-point instruction.

Elements are loaded two at a time into alternating pairs of registers: one time at loop1 into f20 and f21, the next time at loop2 into f22 and f23. Performance would be slightly degraded if the destination of a fld.d were referenced as a source operand in the next 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.
PROGRAMMING EXAMPLES

II SINGLE-PRECISION VECTOR SUM
II input: r16 - vector address, r17 - vector size (must be > 5)
II output: f16 - sum of vector elements
fld.d r0(r16), f20 // Load first two elements
mov -2, r21 // Loop decrement for bla
; // Initiate entry into dual-instruction mode
d.pfadd.ss f0, f0, f0 // Clear adder pipe (1)
adds -6, r17, r17 // Decrement size by 6
; // Enter into dual-instruction mode
d.pfadd.ss f0, f0, f0 // Clear adder pipe (2)
bla r21, r17, L1 // Initialize LCC
d.pfadd.ss f0, f0, f0 // Clear adder pipe (3)

L1::
d.pfadd.ss f20, f30, f30 // Add f20 to pipeline
bla r21, r17, L2 // If more, go to L2 after
d.pfadd.ss f21, f31, f31 // adding f21 to pipeline and
fld.d &r16++, f20 // loading next f20:f21
; // If we reach this point, at least one element remains to be loaded.
; // r17 is either -4 or -3.
; // f20, f21, f22, and f23 still contain vector elements.
; // Add f20 and f22 to the pipeline, too.
d.pfadd.ss f20, f30, f30
br S // Exit loop after adding
d.pfadd.ss f21, f31, f31 // f21 to the pipeline
nop

L2::
d.pfadd.ss f22, f30, f30 // Add f22 to pipeline
bla r21, r17, L1 // If more, go to L1 after
d.pfadd.ss f23, f31, f31 // adding f23 to pipeline and
fld.d &r16++, f32 // loading next f22:f23
; // If we reach this point, at least one element remains to be loaded.
; // r17 is either -4 or -3.
; // f20, f21, f22, and f23 still contain vector elements.
; // Add f20 and f21 to the pipeline, too.
d.pfadd.ss f20, f30, f30
nop
d.pfadd.ss f21, f31, f31
nop

S:: // Initiate exit from dual mode
pfadd.ss f22, f30, f30 // Still in dual mode
mov -4, r21
pfadd.ss f23, f31, f31 // Last dual-mode pair
bte r21, r17, DONE // If there is one more
fld.1 &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
pfadd.ss f30, f31, f31 // A2:A3:0:A1
pfadd.ss f0, f0, f30 // 0:A2+A3:0
pfadd.ss f0, f0, f0 // 0:0:A2+A3
pfadd.ss f0, f0, f31 // 0:0:0
fadd.ss f30, f31, f16 // f16 = A1+A2+A3

Example 9-12. Dual-Instruction Mode
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 9-12 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 $A = A_{ij}$ of dimension $L \times M$ with matrix $B = B_{ij}$ of dimension $M \times N$ is the matrix $C = C_{ij}$ of dimension $L \times N$, where ...

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

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 $L \times N$ times to produce the $L \times N$ product matrix.

Each of the examples 9-13 and 9-14 accomplishes the above extensions through straightforward programming techniques. Each example uses dual-instruction mode to perform the loading and loop control operations in parallel with the basic floating-point calculations. The examples differ in their approaches to memory access and cache usage. To eliminate needless complexity, the examples require that the $M$ dimension be a multiple of eight and that the $B$ matrix be stored in memory by column instead of by row. Data is fetched 32 bytes beyond the higher-address end of both matrices. In real applications, programmers should ensure that no page protection faults occur due to these accesses.

- Example 9-13 depends solely on cached loads.
- Example 9-14 depends on a mix of cached and pipelined loads.

Example 9-13 uses the `fld` instruction for all loads, which places all elements of both matrices $A$ and $B$ in the cache. This approach is ideal for small matrices. Accesses to all elements (after the first access to each) retrieve elements from the cache at the rate of one per clock. Using `fld.q` instructions to retrieve four elements at a time, it is possible to overlap all data access as well as loop control with `m12apm` instructions in the inner loop.

Note, however, that Example 9-13 is "cache bound"; i.e., if the combined size of the two matrices is greater than that of the cache, cache misses will occur, degrading performance. The larger the matrices, the more the misses that will occur.
PROGRAMMING EXAMPLES

// MATRIX MULTIPLY, \( C = A \times B \), CACHED LOADS ONLY
// Registers loaded by calling routine
A=r16 // pointer into A, stored in memory by rows
B=r17 // pointer into B, stored in memory by columns
C=r18 // pointer into C, stored in memory by rows
L=r19 // the number of rows in A
M=r20 // the number of columns in A and rows in B
N=r21 // the number of columns in B
// Registers used locally
RC=r28 // row/column counter decremented by bla for loop control
DEC=r27 // decrementor for row/column pointers
Ar=r26 // counter of rows in A
Bc=r25 // counter of columns in B
Bp=r24 // temporary pointer into B
SIZ=r23 // number of bytes in row of A or column of B
A1=f4; A2=f5; A3=f6; A4=f7; A5=f8; A6=f9; A7=f10; A8=f11 // matrix A row values
B1=f12; B2=f13; B3=f14; B4=f15; B5=f16; B6=f17; B7=f18; B8=f19 // matrix B column vals
T1=f20; T2=f21; T3=f22 // temporary results

Example 9-13. Matrix Multiply, Cached Loads Only (Sheet 1 of 2)
Example 9-13. Matrix Multiply, Cached Loads Only (Sheet 2 of 2)
Example 9-14 uses fld for all the elements of each row of A, and uses pfld to pass all columns of B against each row of A. This example is less cache bound, because only rows of A are placed in the cache. More load instructions are required, because a pfld can load at most two single-precision operands. Still, with pipelined memory cycles, it remains possible to overlap the loading of the eight items from matrix A, the eight items from matrix B, and the loop control with the eight m12apm instructions in the inner loop.

The strategy of Example 9-14 is suitable for larger matrices than the strategy in Example 9-13 because, even in the extreme case where only one row of A fits in the cache, cache misses occur only the first time each row is processed. However, if dimension M is so great that not even one row of A fits entirely in the cache, cache misses will still occur. On the other side, for small matrices, Example 9-14 may not perform as well as Example 9-13, because, even when there is sufficient space in the cache for elements of matrix B, Example 9-14 does not use it.
PROGRAMMING EXAMPLES

Example 9-14. Matrix Multiply, Cached and Pipelined Loads (Sheet 1 of 2)
Example 9-14. Matrix Multiply, Cached and Pipelined Loads (Sheet 2 of 2)
9.13 3-D RENDERING

This series of examples are routines that might be used at the lowest level of a graphics software system to convert a machine-independent description of a 3-D image into values for the frame buffer of a color video display. Typically, higher-level graphics routines represent an object as a set of polygons that together roughly describe the surfaces of the objects to be displayed. The graphics system maintains a database that describes these polygons in terms of their colors, properties of reflectance or translucence, and the locations in 3-D space of their vertices. Due to the roughness of the representation, the amount of information in the database is considerably less than that which must be delivered to the video display. A rendering procedure, such as Example 9-21, uses interpolation to derive the detailed information needed for each pixel in the graphics frame buffer. The rendering procedure also performs pixel-by-pixel hidden-surface elimination.

The focus of this series of examples is Example 9-21, which operates on a segment of a scan line. The segment is bounded by two points of given location and color: from point \((X1, Y0, Z1)\) with color intensities \(\text{Red}1, \text{Grn}1, \text{Blu}1\) to point \((X2, Y0, Z2)\) with color intensities \(\text{Red}2, \text{Grn}2, \text{Blu}2\). The points and color intensities are determined by higher-level graphics software. The points represent the intersection of the scan line with two edges of the projected image of a polygon. For a given scan line, the rendering procedure is executed once for each polygon that projects onto that scan line. The higher-level graphics software is responsible for orienting the objects with respect to the viewer, for making perspective calculations, for scaling, and for determining the amount of light that falls on each polygon vertex.

The 16-bit pixel format is used, giving ample resolution for color shading: \(2^6\) intensity values for red, \(2^6\) intensity values for green, and \(2^4\) intensity values for blue. Example 9-15 shows how to set the pixel size. For hidden-surface elimination, the Z-buffer (or depth buffer) technique is employed, each Z value having a resolution of 16-bits.

Because the examples presented here use almost all of the registers of the i860 microprocessor, the registers are given symbolic names, as defined by Example 9-16. In a real application, it is likely that some of the inputs to the rendering procedure would be passed in floating-point registers instead of the integer registers employed here. The register allocation shown in Example 9-16 simplifies the examples by avoiding the need to use any register for multiple purposes.

```
// SET PIXEL SIZE TO 16
ld.c    psr, Ra          // Work on psr
andnoth 0x00C0, Ra, Ra   // Clear PS
orh     0x0040, Ra, Ra   // PS = 16-bit pixels
st.c    Ra, psr
```

Example 9-15. Setting Pixel Size
Example 9-16. Register Assignments

9.13.1 Distance Interpolation

To perform hidden surface elimination at each pixel, the rendering routine first interpo­lates the value of Z at each pixel. Distance interpolation consists of calculating the slope of Z over the given line segment, then increasing the Z value of each successive pixel by that amount, starting from \( X_1 \). The width of the line segment in pixels is \( dX = X_2 - X_1 \).
Calculate the reciprocal of $dX$:

$$RdX = 1/dX$$

The value of $dX$ is used several times as a divisor. It is most efficient to calculate its reciprocal once, then, instead of dividing by $dX$, multiply by $RdX$. The slope of $Z$ is...

$$mZ = (Z2 - Z1)*RdX$$

Because each polygon is a plane, the value of $mZ$ is constant for all scan lines that intersect the polygon; therefore $mZ$ needs to be calculated only once for each polygon. Example 9-21 assumes that $dX$ and $mZ$ have already been calculated, and all that remains is to apply $mZ$ to successive pixels. Let $Z(Xn)$ be the $Z$ value at pixel $Xn$. Then...

$$Z(X1) = Z1$$
$$Z(X1 + 1) = Z1 + mZ$$
$$Z(X1 + 2) = Z1 + 2*mZ$$

... 

$$Z(X1 + N) = Z1 + N*mZ$$

... 

$$Z(X1 + dX) = Z1 + dX*mZ = Z(X2)$$

Figure 9-1 illustrates this Z-value interpolation.
The `faddz` instruction helps to perform the above calculations 64 bits at a time. Because a Z value is 16 bits wide, Example 9-21 operates on the Z buffer in groups of four. The `faddz` instruction, however, treats the interpolation values \((N \times mZ)\) as 32-bit fixed-point numbers; therefore, two `faddz` instructions are executed for each group of four pixels. Because of the way the `faddz` shifts the MERGE register, the first `faddz` corresponds to even-numbered pixels, while the second corresponds to odd-numbered pixels. Instead of starting with the value for the first pixel \((Z(X1))\) and adding \(mZ\) to each pixel to produce the value for the next pixel, the example procedure starts with the values for the first two even-numbered pixels and adds \(1 \times mZ\) to each of these values to produce the values for the adjacent odd-numbered pair. Adding \(3 \times mZ\) to each of the Z values of an odd-numbered pair produces the values for the next even-numbered pair. Figure 9-2 shows one way of constructing the operands before starting the distance interpolations. (The initial value given to `fsrc1` depends on the alignment of the first pixel.) Table 9-1 helps to visualize the process.

After two `faddz` instructions, the MERGE register holds the Z values for four adjacent pixels (in the correct order). The `form` instruction copies MERGE into one of the 64-bit floating-point registers. the values \(Z1 + N \times mZ\). For each execution of `faddz`, `src1` is the same as `rdest` of the prior `faddz`. After every two `faddz` instructions, a `form` instruction empties the MERGE register.

The same register is used as both `fsrc1` and `fdest` in all `faddz` instructions. This register serves to accumulate Z values for successive pixels; therefore, it is called an `accumulator`. The registers used as `fsrc2` are called `interpolants`. The code in Example 9-17 constructs the interpolants; it needs to be executed only once for each polygon.
### Table 9-1. faddz Visualization

<table>
<thead>
<tr>
<th>Operands</th>
<th>63-32</th>
<th>31-0</th>
<th>MERGE Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63-48</td>
<td>47-32</td>
<td>31-16</td>
</tr>
<tr>
<td>fsr1</td>
<td>0.0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>fsr2</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>2.0</td>
<td>0.0</td>
<td>1</td>
</tr>
<tr>
<td>fsr2</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>3.0</td>
<td>1.0</td>
<td>2</td>
</tr>
<tr>
<td>fsr2</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>6.0</td>
<td>4.0</td>
<td>3</td>
</tr>
<tr>
<td>fsr2</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>7.0</td>
<td>5.0</td>
<td>4</td>
</tr>
<tr>
<td>fsr2</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>10.0</td>
<td>8.0</td>
<td>5</td>
</tr>
<tr>
<td>fsr2</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>11.0</td>
<td>9.0</td>
<td>6</td>
</tr>
<tr>
<td>fsr2</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>fdest/fsr1</td>
<td>14.0</td>
<td>12.0</td>
<td>7</td>
</tr>
<tr>
<td>fsr2</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>fdest</td>
<td>15.0</td>
<td>11.0</td>
<td>8</td>
</tr>
</tbody>
</table>

Because the values of \( Z_I \) and \( mZ \) are constant for each loop through the rendering routine, the numbers shown here are the values of the coefficient \( N \), where the actual operands have the values \( Z_I + N*mZ \). For each execution of faddz, fsr1 is the same as fdest of the prior faddz. After every two faddz instructions, a form instruction empties the MERGE register.

### 9.13.2 Color Interpolation

To determine the RGB color intensities at each pixel, the rendering routine interpolates between the color intensities at the endpoints. (This rendering technique is called “Gouraud shading” after H. Gouraud, “Continuous Shading of Curved Surfaces,” *IEEE Transactions on Computers*, C-20(6), June 1971, pp. 623-628.) Let the symbol C (color) represent either R (red), G (green), or B (blue). Color interpolation consists of calculating the slope of C over the given line segment, then increasing the C values of each pixel along the line.
Example 9-17. Construction of Z Interpolants

successive pixel by that amount, starting from the values for X1. This must be done for C=R, C=G, and C=B. The slope of C is...

\[ mC = (C_2 - C_1) \cdot RdX \]

...where \( RdX = 1/dX \)

The value of \( mC \) is constant for all scan lines that intersect a given pair of polygon edges; therefore \( mC \) needs to be calculated only once for each such pair. Example 9-21 assumes that \( mC \) has already been calculated for all colors, and all that remains is to apply \( mC \) to successive pixels. Let \( C(X_n) \) be a C value at pixel \( X_n \). Then...

\[
\begin{align*}
C(X_1) &= C_1 \\
C(X_1 + 1) &= C_1 + mC \\
C(X_1 + 2) &= C_1 + 2\cdot mC \\
&\vdots \\
C(X_1 + N) &= C_1 + N\cdot mC \\
&\vdots \\
C(X_1 + dX) &= C_1 + dX\cdot mC = C(X_2)
\end{align*}
\]

Figure 9-3 illustrates Gouraud shading of a triangle.

The `faddp` instruction performs the above calculations 64 bits at a time. Because a pixel is 16 bits wide, Example 9-21 operates on pixels in groups of four. Instead of starting with the value for the first pixel \( (C(X_1)) \) and adding \( mC \) to each pixel to produce the value for the next pixel, the example procedure starts with the values for the first four pixels and adds \( 4\cdot mC \) to each group of four to produce the values for the next four. Three `faddp` instructions are executed for each group of four pixels. The first increments the blue values; the second, green; the third, red. Figure 9-4 shows one way of constructing the operands for each color before starting the color interpolations. (The initial value given to `fsrcl` depends on the alignment of the first pixel.)

Setup of the accumulator and interpolants is similar to that of the Z-buffer. The code in Example 9-18 constructs the interpolants; it needs to be executed only once for each pair of edges in each polygon.
9.13.3 Boundary Conditions

The i860 microprocessor operates on 64-bit quantities that are aligned on 8-byte boundaries. The code in this example takes full advantage of this design, handling four 16-bit pixels in each loop. However, if the first or last pixel of a line segment is not on an 8-byte boundary, two kinds of special considerations are required:

1. Masking of Z values near the end points.
2. Initialization of the accumulators.
9.13.3.1 Z-BUFFER MASKING

When either the first or last pixel of the line segment is not at an 8-byte boundary, the rendering procedure must mask the first or last set of new Z-buffer values (newz) so that the Z-buffer and the frame buffer are not erroneously updated. Sometimes both the first and last pixels are in the same 4-pixel set, in which case either one may not be on an 8-byte boundary. A function that looks up and calculates masks is outlined in Example 9-19.

Because the value 0xFFFF is used for masking, the Z-buffer is initialized with 0xFFFE, so that the fzchks instruction always finds the mask to be greater than any Z-buffer contents.

```
// CONSTRUCT INTERPOLANTS IR, IG, IB GIVEN mR, mG, mB
shl 18, mR, Ra // Multiply each color slope by four, then
shl 18, mG, Rb // shift by 1b to put the significant
shl 18, mB, Rc // bits into the high-order half
shr 16, Ra, mR // Return significant 1b bits
shr 16, Rb, mG // to low-order half. Any sign bits
shr 16, Rc, mB // in high-order half are gone.
or mR, Ra, Ra // Join 16-bit quarters
or mG, Rb, Rb // in 32-bit register
or mB, Rc, Rc //
ixfr Ra, iR // Join 32-bit halves
ixfr Rb, iG // in 64-bit register
ixfr Rc, iB //
fmov.ss iR, iRh //
fmov.ss iG, iGh //
fmov.ss iB, iBh //
```

Example 9-18. Construction of Color Interpolants

```
.macro zmask l_align, r_align, Rx, Ry
// l_align -- left-end alignment in two-byte units
// r_align -- right-end alignment in two-byte units
// Rx, Ry -- scratch registers
// Left-end OR masks Right-end OR masks
// Input Output Input Output
// l_align 1Zmask r_align rZmask
// 0 0000 0000 0000 0000 0 0000 0000 0000 0000
// 1 0000 0000 0000 FFFF 1 FFFF 0000 0000 0000
// 2 0000 0000 FFFF FFFF 2 FFFF 0000 0000 0000
// 3 0000 FFFF FFFF FFFF 3 0000 0000 0000 0000
// If the first and last pixels are contained in the same 64-bit
// aligned set, then lZmask = lZmask OR rZmask.
.endm
```

Example 9-19. Z Mask Procedure
9.13.3.2 ACCUMULATOR INITIALIZATION

When the first pixel of the line segment is not at an 8-byte boundary, initial values placed in the accumulators \((aZ, aB, aG, \text{ and } aR)\) must be selected so that \(Z1, Red1, Grn1, \text{ and } Blu1\) correspond to the correct pixel. The desired result is that shown by Table 9-2. However, each value is a composite of two terms: one that is constant for each edge pair \((n*mZ, n*mR, n*mG, n*mB)\) and one that can vary with each scan line \((Z1, Red1, Grn1, Blu1)\). The example assumes that the constant values have all been calculated and stored in a memory table of the format shown by Table 9-3. At the beginning of each line segment the values appropriate to the alignment of the line segment are retrieved from the table and added to the initial \(Z\) and color values, as shown in Example 9-20.

9.13.4 The Inner Loop

Once the proper preparations have been made, only a minimal amount of code is needed to render each scanline segment of a polygon. The code shown in Example 9-21 operates on four pixels in each loop. The left and right ends of the line segment go through different logic paths so that the \(Z\)-buffer masks can be applied by the **form** instruction. All the interior points are handled by the tight inner loop.

The controlling variable \(dX\) is zero-relative and is expressed as a number of pixels. The value of \(dX\) also indicates alignment of the end-points with respect to the 4-pixel groups. Unaligned left-end pixels are subtracted from \(dX\) before entering the inner loop; therefore, subsequent values of \(dX\) indicate the alignment of the right end. A value that is 3 \(\mod 4\) indicates that the right end is aligned, which explains the test for a value of \(-5\) near the end of the loop \((-5 \mod 4 = 3\)). The fact that the value \(-5\) is loaded into register \(Rb\) on every execution of the loop does not represent a programming inefficiency, because there is nothing else for the core unit to do at that point anyway.

<table>
<thead>
<tr>
<th>Table 9-2. Accumulator Initial Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Alignment</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

9-29
Table 9-3. Accumulator Initialization Table

<table>
<thead>
<tr>
<th>Alignment</th>
<th>*mZ</th>
<th>*mR</th>
<th>*mG</th>
<th>*mB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1, -3</td>
<td>-1, -2, -3, -4</td>
<td>-1, -2, -3, -4</td>
<td>-1, -2, -3, -4</td>
</tr>
<tr>
<td>2</td>
<td>-2, -4</td>
<td>-2, -3, -4, -5</td>
<td>-2, -3, -4, -5</td>
<td>-2, -3, -4, -5</td>
</tr>
<tr>
<td>4</td>
<td>-3, -5</td>
<td>-3, -4, -5, -6</td>
<td>-3, -4, -5, -6</td>
<td>-3, -4, -5, -6</td>
</tr>
<tr>
<td>6</td>
<td>-4, -6</td>
<td>-4, -5, -6, -7</td>
<td>-4, -5, -6, -7</td>
<td>-4, -5, -6, -7</td>
</tr>
</tbody>
</table>

Example 9-20. Accumulator Initialization

```assembly
// ACCUMULATOR INITIALIZATION TABLE
.data; .align .double
acc_init_tab:: .double [16]

.dsect
aBi: .double // Four initial 16-bit blue values
aGi: .double // Four initial 16-bit green values
aRi: .double // Four initial 16-bit red values
aZi: .double // Two initial 32-bit Z values

.text
.INITIALIZE ACCUMULATORS
.macro acc_init Lalign, Rtab, Rx, Ry, Fx, Fxh

I--left-end alignment (0..3) in two-byte units
II--register to use for addressing the table
II--Rx, Ry, Fx, Fxh -- scratch registers

mov acc_init_tab, Rtab //
shl 5, Lalign, Lalign // Multiply by row width
adds Lalign, Rtab, Rtab // Index row corresponding to alignment
fld.d aZi(Rtab), aZ // Z
ixfr Zl, Fx // Z
fld.d aRi(Rtab), aR // R--Load constant values
shl 1b, Red1, Rx // R--Shift starting value to hi-order
fmov.ss Fx, Fxh // Z
shr 1b, Rx, Ry // R--Red1 stripped of sign bits
fiadd.dd Fx, aZ, aZ // Z
or Rx, Ry, Ry // R--Form (Red1,Red1)
ixfr Ry, Fx // R--Put in 64-bit register
fld.d aGi(Rtab), aG // G
shl 1b, Grn1, Rx // G
fmov.ss Fx, Fxh // R--Form (Red1,Red1,Red1,Red1)
shr 1b, Rx, Ry // G
fiadd.dd Fx, aR, aR // R--Add variables to constants
or Rx, Ry, Ry // G
ixfr Ry, Fx // G
fld.d aBi(Rtab), aB // B
shl 1b, Blu1, Rx // B
fmov.ss Fx, Fxh // G
shr 1b, Rx, Ry // B
fiadd.dd Fx, aG, aG // G
or Rx, Ry, Ry // B
ixfr Ry, Fx // B
fmov.ss Fx, Fxh // B
fiadd.dd Fx, aB, aB // B
.endm
```

9-30
// RENDERING PROCEDURE
// 16-bit pixels, 16-bit Z-buffer
and 3, X1, Ra // Determine alignment of starting-point
acc_init Ra, Rb, Rc, Rd, Fa, Fah // Initialize accumulators
subs 4, Ra, Rb // 4 - alignment
subs dx, Rb, dx // Adjust dx by X1 alignment
// If dx <= 0, then right end is in same set as left end
and 3, dx, Rb // Determine alignment of right end
zmask Ra, Rb, Rc, Rd // Prepare both left- and right-end masks
left_end:: // Handle boundary conditions
d.faddz aZ, iZ3, aZ // Interpolate 2 even Z values
adds -8, FBP, FBP // Anticipate autoincrement
d.faddz aZ, iZ1, aZ // Interpolate 2 odd Z values
adds -8, ZBP, ZBP // Anticipate autoincrement
d.faddz FBP, iZ1, FBP // Prepare left-end mask
d.faddz ZBP, iZ3, ZBP // Prepare right-end mask
left_end:
left_end:: // Handle boundary conditions

Example 9-21. 3-D Rendering (1 of 2)
right_end:: // Handle boundary conditions
  d.faddz aZ, iZ3, aZ // Interpolate 2 even Z values
  nop
  d.faddz aZ, iZ1, aZ // Interpolate 2 odd Z values
  fst.d newz, \delta(ZBP)++ // Update Z buf from prior loop
  d.form rZmask, newz // Mask 4 new Z values
  nop
  d.fzchks f0, f0, f0 // Shift PM[7..4] to PM[3..0]
  nop
  d.faddp aB, iB, aB // Interpolate 4 blue intensities
  pst.d newi, \delta(FBP)++ // Store pixels indicated by PM[3..0]
  d.faddp aG, iG, aG // Interpolate 4 green intensities
  nop
  d.faddp aR, iR, aR // Interpolate 4 red intensities
  nop

aligned_end:: // No special boundary conditions
  d.form f0, newi // Move 4 new pixels to 64-bit reg
  br wrap_up //
  d.fzchks oldz, newz, newz // Mark closer points in PM[7..4]
  nop

short_segment::
  d.fnop
  adds \delta, dX, r0 // Is right end in same set as left?
  d.fnop
  bnc.t right_end // Branch taken if no.
  d.fnop
  fld.d 1\delta(ZBP), oldz // Fetch 4 old Z values

wrap_up:: // Store the unstored and leave dual mode.
  fzchks f0, f0, f0 // Shift PM[7..4] to PM[3..0]
  fst.d newz, \delta(ZBP)++ // Update Z buf from prior loop
  fnop
  pst.d newi, \delta(FBP)++ // Store pixels indicated by PM[3..0]
APPENDIX A
INSTRUCTION SET SUMMARY

Key to abbreviations:

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

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

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

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

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

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

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

Other abbreviations include:

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

.r  Precision specification .ss, .sd, .ds, or .dd. Refer to Table A-1.

.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)

mem.x(address)  The contents of the memory location indicated by address with a size of x.

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

Instruction Definitions in Alphabetical Order

adds isrc1, isrc2, idest ..................................................Add Signed
idest ← isrc1 + isrc2
OF ← (bit 31 carry ≥ bit 30 carry)
CC set if isrc2 < −isrc1 (signed)
CC clear if isrc2 ≥ −isrc1 (signed)

addu isrc1, isrc2, idest ..................................................Add Unsigned
idest ← isrc1 + isrc2
OF ← bit 31 carry
CC ← bit 31 carry

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Source Precision</th>
<th>Result Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ss</td>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>.sd</td>
<td>single</td>
<td>double</td>
</tr>
<tr>
<td>.dd</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>.ds</td>
<td>double</td>
<td>single</td>
</tr>
</tbody>
</table>

Table A-1. Precision Specification
and isrc1, isrc2, idest ...................................................... Logical AND
  idest ← isrc1 and isrc2
  CC set if result is zero, cleared otherwise

andh #const, isrc2, idest ................................................. Logical AND High
  idest ← (#const shifted left 16 bits) and isrc2
  CC set if result is zero, cleared otherwise

andnot isrc1, isrc2, idest ................................................ Logical AND NOT
  idest ← not isrc1 and isrc2
  CC set if result is zero, cleared otherwise

andnoth #const, isrc2, idest ........................................... Logical AND NOT High
  idest ← not (#const shifted left 16 bits) and isrc2
  CC set if result is zero, cleared otherwise

bc lbwoff ................................................................. Branch on CC
  IF CC = 1
  THEN continue execution at brx(lbwoff)
  FI

bc.t lbwoff .............................................................. Branch on CC, Taken
  IF CC = 1
  THEN execute one more sequential instruction
      continue execution at brx(lbwoff)
  ELSE skip next sequential instruction
  FI

bla isrc1ni, isrc2, sbwoff .............................................. Branch on LCC and Add
  LCC-temp clear if isrc2 < −isrc1ni (signed)
  LCC-temp set if isrc2 ≥ −isrc1ni (signed)
  isrc2 ← isrc1ni + isrc2
  Execute one more sequential instruction
  IF LCC
  THEN LCC ← LCC-temp
       continue execution at brx(sbwoff)
  ELSE LCC ← LCC-temp
  FI

bnc lbwoff ............................................................... Branch on Not CC
  IF CC = 0
  THEN continue execution at brx(lbwoff)
  FI

bnc.t lbwoff ............................................................. Branch on Not CC, Taken
  IF CC = 0
  THEN execute one more sequential instruction
      continue execution at brx(lbwoff)
  ELSE skip next sequential instruction
  FI
INSTRUCTION SET SUMMARY

br lbroff ................................................................. Branch Direct Unconditionally
  Execute one more sequential instruction.
  Continue execution at brx(lbroff).

bri [isrc1ni] ............................................................. Branch Indirect Unconditionally
  Execute one more sequential instruction
  IF any trap bit in psr is set
  THEN copy PU to U, PIM to IM in psr
      clear trap bits
      IF DS is set and DIM is reset
      THEN enter dual-instruction mode after executing one
          instruction in single-instruction mode
      ELSE IF DS is set and DIM is set
      THEN enter single-instruction mode after executing one
          instruction in dual-instruction mode
      ELSE IF DIM is set
      THEN enter dual-instruction mode
          for next instruction pair
      ELSE enter single-instruction mode
          for next instructions pair
  FI
  FI
  FI
  Continue execution at address in isrc1ni
  (The original contents of isrc1ni is used even if the next instruction
  modifies isrc1ni. Does not trap if isrc1ni is misaligned.)

bte isrc1s, isrc2, sbroff ......................................... Branch If Equal
  IF isrc1s = isrc2
  THEN continue execution at brx(sbroff)
  FI

btne isrc1s, isrc2, sbroff ....................................... Branch If Not Equal
  IF isrc1s ≠ isrc2
  THEN continue execution at brx(sbroff)
  FI

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

calli [isrc1ni] ......................................................... Indirect Subroutine Call
  r1 ← address of next sequential instruction + 4
  Execute one more sequential instruction
  Continue execution at address in isrc1ni
  (The original contents of isrc1ni is used even if the next instruction
  modifies isrc1ni. Does not trap if isrc1ni is misaligned. The
  register isrc1ni must not be r1.)
INSTRUCTION SET SUMMARY

fadd.p fsrl, fsr2, fdest ......................................................... Floating-Point Add
fdest ← fsrl + fsr2

faddp fsrl, fsr2, fdest .......................................................... Add with Pixel Merge
fdest ← fsrl + fsr2
Shift and load MERGE register from fsrl + fsr2 as defined in Table A-2

faddz fsrl, fsr2, fdest .......................................................... Add with Z Merge
fdest ← fsrl + fsr2
Shift MERGE right 16 and load fields 31..16 and 63..48 from fsrl + fsr2

famov.r fsrl, fdest ......................................................... Floating-Point Adder Move
fdest ← fsrl

fiadd.w fsrl, fsr2, fdest ..................................................... Long-Integer Add
fdest ← fsrl + fsr2

fisub.w fsrl, fsr2, fdest ..................................................... Long-Integer Subtract
frdest ← fsrl – fsr2

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

fld.y isrel(isr2), fdest ....................................................... (Normal)
fld.y isrel(isr2) + + , fdest .............................................. (Autoincrement)
fdest ← mem.y (isrel + isr2)
IF autoincrement
THEN isr2 ← isrel + isr2
FI

flush #const (isr2) .......................................................... (Normal)
flush #const(isr2) + + ................................................... (Autoincrement)
Replace block in data cache with address (#const + isr2).
Contents of block undefined.
IF autoincrement
THEN isr2 ← #const + isr2
FI

Table A-2. FADDP MERGE Update

<table>
<thead>
<tr>
<th>Pixel Size (from PS)</th>
<th>Fields Loaded from Result into MERGE</th>
<th>Right Shift Amount (Field Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>63..56, 47..40, 31..24, 15..8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>63..58, 47..42, 31..26, 15..10</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>63..56, 31..24</td>
<td>8</td>
</tr>
</tbody>
</table>
**INSTRUCTION SET SUMMARY**

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

fmov.r fsrcl, fdest ........................................................................ Floating-Point Reg-Reg Move
Assembler pseudo-operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmov.ss</td>
<td>fiadd.ss fsrcl, f0, fdest</td>
</tr>
<tr>
<td>fmov.dd</td>
<td>fiadd.dd fsrcl, f0, fdest</td>
</tr>
<tr>
<td>fmov.sd</td>
<td>famov.sd fsrcl, fdest</td>
</tr>
<tr>
<td>fmov.ds</td>
<td>famov.ds fsrcl, fdest</td>
</tr>
</tbody>
</table>

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

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

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

frcp.p fsrcl, fdest ............................................................... Floating-Point Reciprocal
fdest ← 1 / fsrc2 with maximum mantissa error < 2⁻⁷

frsqr.p fsrcl, fdest ............................................................. Floating-Point Reciprocal Square Root
fdest ← 1 / \(\sqrt{fsrc2}\) with maximum mantissa error < 2⁻⁷

Floating-Point Store

fst.y fdest, fsrcl(fsrc2) ...................................................... (Normal)
fst.y fdest, fsrcl(fsrc2) + + ............................................... (Autoincrement)
mem.y (fsrc2 + fsrcl) ← fdest
IF autoincrement
THEN fsrc2 ← fsrc1 + fsrc2
FI

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

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

fxfr fsrcl, idest ................................................................. Transfer F-P to Integer Register
idest ← fsrc1
INSTRUCTION SET SUMMARY

**fzchk1** fsrl, fsr2, fdest ................................................................. 32-Bit Z-Buffer Check

Consider fsrl, fsr2, and fdest as arrays of two 32-bit
fields fsrl(0)..fsrl(1), fsr2(0)..fsr2(1), and fdest(0)..fdest(1)
where zero denotes the least-significant field.

PM ← PM shifted right by 2 bits
FOR i = 0 to 1
DO
  PM[i + 6] ← fsr2(i) ≤ fsrl(i) (unsigned)
  fdest(i) ← smaller of fsr2(i) and fsrl(i)
OD
MERE ← 0

**fzchks** fsrl, fsr2, fdest ................................................................. 16-Bit Z-Buffer Check

Consider fsrl, fsr2, and fdest as arrays of four 16-bit
fields fsrl(0)..fsrl(3), fsr2(0)..fsr2(3), and fdest(0)..fdest(3)
where zero denotes the least-significant field.

PM ← PM shifted right by 4 bits
FOR i = 0 to 3
DO
  PM[i + 4] ← fsr2(i) ≤ fsrl(i) (unsigned)
  fdest(i) ← smaller of fsr2(i) and fsrl(i)
OD
MERE ← 0

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

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

**ixfr** isrl1ni, fdest ........................................................................... Transfer Integer to F-P Register

fdest ← isrl1ni

**ld.c** csrl2, idest ............................................................................. Load from Control Register

idest ← csrl2

**ld.x** isrl(isrl2), idest ....................................................................... Load Integer

idest ← mem.x (isrl1 + isrl2)

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

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

**mov** isrl2, idest ............................................................................. Register-Register Move

Assembler pseudo-operation

mov isrl2, idest = shl r0, isrl2, idest

A-7
nop ........................................................................................................Core-Unit No Operation
    Assembler pseudo-operation
    nop = shi r0, r0, r0

or isrc1, isrc2, idest .................................................................Logical OR
    idest ← isrc1 OR isrc2
    CC set if result is zero, cleared otherwise

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

pfadd.p fsrcl, fsrc2, fdest .................................................. Pipelined Floating-Point Add
    fdest ← last stage adder result
    Advance A pipeline one stage
    A pipeline first stage ← fsrc1 + fsrc2

pfaddp fsrcl, fsrc2, fdest .................................................. Pipelined Add with Pixel Merge
    fdest ← last stage graphics result
    last stage graphics result ← fsrc1 + fsrc2
    Shift and load MERGE register from fsrc1 + fsrc2 as defined in Table A-2

pfaddz fsrcl, fsrc2, fdest ........................................ Pipelined Add with Z Merge
    fdest ← last stage graphics result
    last stage graphics result ← fsrc1 + fsrc2
    Shift MERGE right 16 and load fields 31..16 and 63..48 from fsrc1 + fsrc2

pfam.p fsrcl, fsrc2, fdest ................................... Pipelined Floating-Point Add and Multiply
    fdest ← last stage adder result
    Advance A and M pipeline one stage
        (operands accessed before advancing pipeline)
        A pipeline first stage ← A-op1 + A-op2
        M pipeline first stage ← M-op1 × M-op2

pfamov.r fsrcl, fdest .................................................. Pipelined Floating-Point Adder Move
    fdest ← last stage adder result
    Advance A pipeline one stage
    A pipeline first stage ← fsrcl

pfeq.p fsrcl, fsrc2, fdest .................................. Pipelined Floating-Point Equal Compare
    fdest ← last stage adder result
    CC set if fsrcl = fsrc2, else cleared
    Advance A pipeline one stage
    A pipeline first stage is undefined, but no result exception occurs

pfgt.p fsrcl, fsrc2, fdest ........... Pipelined Floating-Point Greater-Than Compare
    (Assembler clears R-bit of instruction)
    fdest ← last stage adder result
    CC set if fsrcl > fsrc2, else cleared
    Advance A pipeline one stage
    A pipeline first stage is undefined, but no result exception occurs
pfiadd.w src1, src2, dest .................................................. Pipelined Long-Integer Add
dest ← last stage graphics result
last stage graphics result ← src1 + src2

pfisub.w src1, src2, dest .................................................. Pipelined Long-Integer Subtract
dest ← last stage graphics result
last stage graphics result ← src1 - src2

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

Pipelined Floating-Point Load

pfld.z isrc1(isrc2), dest .................................................. (Normal)
pfld.z isrc1(isrc2) + + , dest ........................................ (Autoincrement)

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

pfire.p src1, src2, dest .................................................. Pipelined F-P Less-Than or Equal Compare

(Idential to pfgt.p except that
assembler sets R-bit of instruction.)
dest ← last stage adder result
CC clear if src1 ≤ src2, else set
Advance A pipeline one stage
A pipeline first stage is undefined, but no result exception occurs

pfiadd.p src1, src2, dest .................................................. Pipelined Floating-Point Add and Multiply
dest ← last stage multiplier result
Advance A and M pipeline one stage
operands accessed before advancing pipeline
A pipeline first stage ← A-op1 + A-op2
M pipeline first stage ← M-op1 × M-op2

pfmov.r src1, dest .................................................. Pipelined Floating-Point Reg-Reg Move
Assembler pseudo-operation

pfmov.ss src1, dest = pfiadd.ss src1, f0, dest
pfmov.dd src1, dest = pfiadd.dd src1, f0, dest
pfmov.sd src1, dest = pfamov.sd src1, dest
pfmov.ds src1, dest = pfamov.ds src1, dest

pfisub.p src1, src2, dest .................................................. Pipelined Floating-Point Subtract and Multiply
dest ← last stage multiplier result
Advance A and M pipeline one stage
operands accessed before advancing pipeline
A pipeline first stage ← A-op1 - A-op2
M pipeline first stage ← M-op1 × M-op2
INSTRUCTION SET SUMMARY

\texttt{pfmul.p} \texttt{fsrcl, fsrc2, fdest} .................................................. Pipelined Floating-Point Multiply
\texttt{fdest} ← last stage multiplier result
Advance M pipeline one stage
M pipeline first stage ← \texttt{fsrcl} \times \texttt{fsrc2}

\texttt{pfmul3.p} \texttt{fsrcl, fsrc2, fdest} .................................................... Three-Stage Pipelined Multiply
\texttt{fdest} ← last stage multiplier result
Advance 3-Stage M pipeline one stage
M pipeline first stage ← \texttt{fsrcl} \times \texttt{fsrc2}

\texttt{pform} \texttt{fsrcl, fdest} ..................................................... Pipelined OR to MERGE Register
\texttt{fdest} ← last stage graphics result
last stage graphics result ← \texttt{fsrcl} OR MERGE
MERGE ← 0

\texttt{pfsm.p} \texttt{fsrcl, fsrc2, fdest} ........................................ Pipelined Floating-Point Subtract and Multiply
\texttt{fdest} ← last stage adder result
Advance A and M pipeline one stage
\texttt{fsrcl} (i) \texttt{fsrc2} (i) \texttt{M-op1} \times \texttt{M-op2}

\texttt{pfsub.p} \texttt{fsrcl, fsrc2, fdest} ......................................... Pipelined Floating-Point Subtract
\texttt{fdest} ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← \texttt{fsrcl} \times \texttt{fsrc2}

\texttt{pftrunc.p} \texttt{fsrcl, fdest} ........................................ Pipelined Floating-Point to Integer Conversion
\texttt{fdest} ← last stage adder result
Advance A pipeline one stage
A pipeline first stage ← 64-bit value with low-order 32 bits
equal to integer part of \texttt{fsrcl}

\texttt{pfzchk} \texttt{fsrcl, fsrc2, fdest} ........................................ Pipelined 32-Bit Z-Buffer Check
Consider \texttt{fsrcl}, \texttt{fsrc2}, and \texttt{fdest} as arrays of two 32-bit
fields \texttt{fsrcl(0).fsrcl(1), fsrc2(0).fsrc2(1), and fdest(0).fdest(1)}
where zero denotes the least-significant field.
PM ← PM shifted right by 2 bits
FOR \texttt{i} = 0 to 1
DO
PM \texttt{[i + 6]} ← \texttt{fsrc2(i) \leq fsrc1(i) (unsigned)}
\texttt{fdest(i) ← last stage graphics result}
later stage graphics result ← smaller of \texttt{fsrc2(i)} and \texttt{fsrc1(i)}
OD
MERGE ← 0
pfzchks fsrcl, fsr2, fdest................................. Pipelined 16-Bit Z-Buffer Check

Consider fsrcl, fsr2, and fdest as arrays of four 16-bit fields fsrcl(0), fsrcl(3), fsr2(0), fsr2(3), and fdest(0), fdest(3) where zero denotes the least-significant field.

PM ← PM shifted right by 4 bits
FOR i = 0 to 3
  DO
    PM[i + 4] ← fsrc2(i) ≤ fsrcl(i) (unsigned)
    fdest ← last stage graphics result
    last stage graphics result(i) ← smaller of fsrc2(i) and fsrcl(i)
  OD
MERGE ← 0

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

Shift Left
shl isrc1, isrc2, idest ................................................................. isrc2 shifted left by isrc1 bits

Shift Right
shr isrc1, isrc2, idest ................................................................. isrc2 shifted right by isrc1 bits

Shift Right Arithmetic
shra isrc1, isrc2, idest ............................................................... isrc2 arithmetically shifted right by isrc1 bits

Shift Right Double
shrd isrc1ni, isrc2, idest ............................................................. low-order 32 bits of isrc1ni:isrc2 shifted right by SC bits

Store to Control Register
st.c src1ni, csrc2 ................................................................. csrc2 ← src1ni

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

Subtract Signed
subs isrc1, isrc2, idest ............................................................... isrc1 − isrc2
  OF ← (bit 31 carry ⊕ bit 30 carry)
  CC set if isrc2 > isrc1 (signed)
  CC clear if isrc2 ≤ isrc1 (signed)
INSTRUCTION SET SUMMARY

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

**trap** isrc1ni, isrc2, idest ............................................................... Software Trap
  Generate trap with IT set in psr

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

**xor** isrc1, isrc2, idest ................................................................. Logical Exclusive OR
  idest ← isrc1 XOR isrc2
  CC set if result is zero, cleared otherwise

**xorh** #const, isrc2, idest ............................................................. Logical Exclusive OR High
  idest ← (#const shifted left 16 bits) XOR isrc2
  CC set if result is zero, cleared otherwise
Instruction Format and Encoding
All instructions are 32 bits long and begin on a four-byte boundary. When operands are registers, the encodings shown in Table B-1 are used.

Among the core instructions, there are two general formats: REG-format and CTRL-format. Within the REG-format are several variations.

<table>
<thead>
<tr>
<th>Register</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td>31</td>
</tr>
<tr>
<td>f0</td>
<td>0</td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td></td>
</tr>
<tr>
<td>f31</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Instruction</td>
<td>0</td>
</tr>
<tr>
<td>Processor Status</td>
<td>1</td>
</tr>
<tr>
<td>Directory Base</td>
<td>2</td>
</tr>
<tr>
<td>Data Breakpoint</td>
<td>3</td>
</tr>
<tr>
<td>Floating-Point Status</td>
<td>4</td>
</tr>
<tr>
<td>Extended Processor Status</td>
<td>5</td>
</tr>
</tbody>
</table>
The src2 field selects one of the 32 integer registers (most instructions) or one of the control registers (st.e and ld.e). 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:

<table>
<thead>
<tr>
<th>Bit 28</th>
<th>Bit 0</th>
<th>Operand Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8-bits</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8-bits</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>16-bits</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>32-bits</td>
</tr>
</tbody>
</table>
When \textit{src1} is immediate and bit 28 is set, bit zero of the immediate value is forced to zero.

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

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Operand Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>64-bits</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>128-bits</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>32-bits</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>32-bits</td>
</tr>
</tbody>
</table>

When \textit{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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.x</td>
<td>Load Integer</td>
<td>0000L01</td>
</tr>
<tr>
<td>st.x</td>
<td>Store Integer</td>
<td>0001L11</td>
</tr>
<tr>
<td>ixfr</td>
<td>Integer to F-P Reg Transfer</td>
<td>0001010</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>0001110</td>
</tr>
<tr>
<td>fld.x, fst.x</td>
<td>Load/Store F-P</td>
<td>00100LS1</td>
</tr>
<tr>
<td>flush</td>
<td>Flush</td>
<td>0011011</td>
</tr>
<tr>
<td>pst.d</td>
<td>Pixel Store</td>
<td>0011111</td>
</tr>
<tr>
<td>id.c, st.c</td>
<td>Load/Store Control Register</td>
<td>00111LS0</td>
</tr>
<tr>
<td>bri</td>
<td>Branch Indirect</td>
<td>1000000</td>
</tr>
<tr>
<td>trap</td>
<td>Trap</td>
<td>1010000</td>
</tr>
<tr>
<td>bte, btne</td>
<td>Branch Equal or Not Equal</td>
<td>011001E1</td>
</tr>
<tr>
<td>pfld.y</td>
<td>Pipelined F-P Load</td>
<td>0111011</td>
</tr>
<tr>
<td>(CTRL-Format Instructions)</td>
<td></td>
<td>01111xx</td>
</tr>
<tr>
<td>addu, -s, subu, -s,</td>
<td>Add/Subtract</td>
<td>1000SOAS1</td>
</tr>
<tr>
<td>shl, shr</td>
<td>Logical Shift</td>
<td>101010LR1</td>
</tr>
<tr>
<td>shrd</td>
<td>Double Shift</td>
<td>1011000</td>
</tr>
<tr>
<td>bla</td>
<td>Branch LCC Set and Add</td>
<td>1011101</td>
</tr>
<tr>
<td>shra</td>
<td>Arithmetic Shift</td>
<td>1011111</td>
</tr>
<tr>
<td>and(h)</td>
<td>AND</td>
<td>11000H1</td>
</tr>
<tr>
<td>andnot(h)</td>
<td>ANDNOT</td>
<td>11101H1</td>
</tr>
<tr>
<td>or(h)</td>
<td>OR</td>
<td>11100H1</td>
</tr>
<tr>
<td>xor(h)</td>
<td>XOR</td>
<td>1111H1</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>111x0x1</td>
</tr>
</tbody>
</table>

### Bit Descriptions
- **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, andnot h, 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

![Instruction Format and Encoding](240329i)

### Core Escape Opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>calli</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>intovr</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unlock</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **lock**: Begin Interlocked Sequence
- **calli**: Indirect Subroutine Call
- **intovr**: Trap on Integer Overflow
- **unlock**: End Interlocked Sequence
CTRL-Format Instructions

![Diagram showing the format of CTRL-Format Opcodes]

CTRL-Format Opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>28</th>
<th>27</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>br</td>
<td>Branch Direct</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>call</td>
<td>Call</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bc(.t)</td>
<td>Branch on CC Set</td>
<td>1</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>bnc(.t)</td>
<td>Branch on CC Clear</td>
<td>1</td>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

T = Taken
0 = -bc or bnc
1 = -bc.t or bnc.t
## Floating-Point Instruction Encoding

![Instruction Format and Encoding Diagram](image)

 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

<table>
<thead>
<tr>
<th>P</th>
<th>Pipelining</th>
<th>1 — Pipelined instruction mode</th>
<th>0 — Scalar instruction mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Dual-Instruction Mode</td>
<td>1 — Dual-instruction mode</td>
<td>0 — Single-instruction mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>Source Precision</th>
<th>1 — Double-precision source operands</th>
<th>0 — Single-precision source operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Result Precision</td>
<td>1 — Double-precision result</td>
<td>0 — Single-precision result</td>
</tr>
</tbody>
</table>

240329
## Floating-Point Opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfam</td>
<td>Add and Multiply*</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>pfmam</td>
<td>Multiply with Add*</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>pfsm</td>
<td>Subtract and Multiply*</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>pfmsm</td>
<td>Multiply with Subtract*</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>(p)fmul</td>
<td>Multiply</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>fnlow</td>
<td>Multiply Low</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>frcp</td>
<td>Reciprocal</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>frsqr</td>
<td>Reciprocal Square Root</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>pfmul3.dd</td>
<td>3-Stage Pipelined Multiply</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>(p)fadd</td>
<td>Add</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>(p)fsub</td>
<td>Subtract</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>(p)fix</td>
<td>Fix</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>(p)famov</td>
<td>Adder Move</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>pfgt/pfle**</td>
<td>Greater Than</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>pfeq</td>
<td>Equal</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>(p)frunc</td>
<td>Truncate</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>fxfr</td>
<td>Transfer to Integer Register</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>(p)fiadd</td>
<td>Long-Integer Add</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>(p)fisub</td>
<td>Long-Integer Subtract</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>(p)fzchkl</td>
<td>Z-Check Long</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>(p)fzchks</td>
<td>Z-Check Short</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>(p)faddp</td>
<td>Add with Pixel Merge</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>(p)faddz</td>
<td>Add with Z Merge</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>(p)form</td>
<td>OR with MERGE Register</td>
<td>1 0 1 0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Freeze Condition</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction-cache miss</td>
<td>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.</td>
</tr>
<tr>
<td>Reference to destination of ld instruction that misses fild miss</td>
<td>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)</td>
</tr>
<tr>
<td>call, calli, fxfr, ld.c, or st.c and data cache load miss processing in progress</td>
<td>One plus number of clocks from ADS to first (or second in the case of fld.q) READY returned</td>
</tr>
<tr>
<td>ld, st, pfld, fld, fst, or ixfr and data cache load miss processing in progress</td>
<td>One plus number of clocks until first (or second in the case of 128-bit loads) READY returned</td>
</tr>
<tr>
<td>Reference to dest of ld, call, calli, fxfr, or ld.c in the next instruction. (Dest of call and calli is r1)</td>
<td>One plus number of clocks until last READY returned</td>
</tr>
<tr>
<td>Reference to dest of fild, pfld, or ixfr in the next two instructions</td>
<td>One clock</td>
</tr>
<tr>
<td>bc, bnc, bc.t, or bnc.t following addu, adds, subu, sub, pfeq, pfle, or pfgt</td>
<td>Two clocks in the first instruction; or one in the second instruction</td>
</tr>
<tr>
<td>Fsrt of multiplier operation refers to result of previous operation (either scalar or pipelined)</td>
<td>One clock</td>
</tr>
<tr>
<td>Floating-point operation or graphics-unit instruction or fst and scalar operation in progress other than frcp or frsqr</td>
<td>One clock</td>
</tr>
</tbody>
</table>

If the scalar operation is fadd, fix, fmlow, fmul.ss, fmul.sd, ftrunc, or fsub, two minus the number of instructions (or dual pairs) executed after the scalar operation. If the scalar operation is fmul.dd, three minus the number of instructions (or dual pairs) executed after it. Add one if one or both of the following situations occur:

a. There is an overlap between the result register(s) of the previous scalar operation, and the source of the floating-point operation, and the destination precision of the scalar operation is different from the source precision of the floating-point operation.

b. 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.
<table>
<thead>
<tr>
<th>Freeze Condition</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier operation preceded by a double-precision multiply</td>
<td>One clock</td>
</tr>
<tr>
<td>TLB miss</td>
<td>Five plus the number of clocks to finish two reads plus the number of clocks to set A-bits (if necessary)</td>
</tr>
<tr>
<td><strong>pfld</strong> when three <strong>pfld</strong>'s are outstanding</td>
<td>One plus the number of clocks to return data from first <strong>pfld</strong></td>
</tr>
<tr>
<td><strong>pfld</strong> hits in the data cache</td>
<td>Two plus the number of clocks to finish all outstanding accesses</td>
</tr>
<tr>
<td>Store pipe full (two store miss cycles pending or a 256-bit WB cycle pending plus external bus pipeline full) and <strong>st</strong> or <strong>fst</strong> miss, <strong>ld</strong> miss, or <strong>flush</strong></td>
<td>One plus the number of clocks until READY# active on next 64-bit write cycle or 2nd READY# of next 128-bit write cycle</td>
</tr>
<tr>
<td>Address pipe full (two internal bus cycles pending plus external bus pipeline full) and <strong>ld</strong>, <strong>fld</strong>, <strong>pfld</strong>, <strong>st</strong>, <strong>fst</strong></td>
<td>Number of clocks until next non-repeated address can be issued (i.e., an address which is not the 2nd-4th cycle of a cache fill, or the 2nd-8th cycle of a CS8 mode instruction fetch, or the 2nd cycle of an 128-bit write)</td>
</tr>
<tr>
<td><strong>Id</strong> or <strong>fld</strong> following <strong>st</strong> or <strong>fst</strong> hit</td>
<td>One clock</td>
</tr>
<tr>
<td>Delayed branch not taken</td>
<td>One clock</td>
</tr>
<tr>
<td>Nondelayed branch taken:</td>
<td>One clock</td>
</tr>
<tr>
<td><strong>bc</strong>, <strong>bnc</strong></td>
<td>Two clocks</td>
</tr>
<tr>
<td><strong>ble</strong>, <strong>btne</strong></td>
<td>One clock</td>
</tr>
<tr>
<td>Branch indirect <strong>bri</strong></td>
<td>Two clocks</td>
</tr>
<tr>
<td><strong>st.c</strong></td>
<td>One clock</td>
</tr>
<tr>
<td>Result of graphics-unit instruction (other than <strong>fmov.dd</strong>) used in next instruction when the next instruction is an adder or multiplier instruction</td>
<td>One clock</td>
</tr>
<tr>
<td>Result of graphics-unit instruction used in next instruction when the next instruction is a graphics-unit instruction</td>
<td>One clock</td>
</tr>
<tr>
<td><strong>flush</strong> followed by <strong>flush</strong></td>
<td>Three clocks minus the number of instructions between the two flush instructions</td>
</tr>
<tr>
<td><strong>fst</strong> followed by pipelined floating-point operation that overwrites the register being stored</td>
<td>One clock</td>
</tr>
<tr>
<td>Some multiplies, depending on data pattern and rounding mode. This delay occurs on 2 data patterns in every 256.</td>
<td>Two clocks</td>
</tr>
</tbody>
</table>
Instruction Characteristics
APPENDIX D
INSTRUCTION CHARACTERISTICS

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

- What processing unit executes the instruction. The codes for processing units are:
  
  A  floating-point adder unit
  E  Core execution unit
  G  Graphics unit
  M  Floating-point multiplier unit

- Whether the instruction is pipelined or not. A P indicates that the instruction is pipelined.

- Whether the instruction is a delayed branch instruction. A D marks the delayed branches.

- Whether the instruction changes the condition code CC. A CC marks those instructions that change CC.

- Which faults can be caused by the instruction. The codes used for exceptions are:

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

Note that this is not the same as specifying at which instructions faults may be reported. A fault is reported on the subsequent floating-point instruction plus pst, fst, and sometimes fld, pfld, and ixfr. See Section 7.4.2 for more information on result exception reporting.

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 dest of the prior operation.
  5. The freg should not reference the destination of the next instruction if that instruction is a pipelined floating-point operation.
6. The destination should not be a source operand of the next instruction.

7. When the prior operation is scalar and multiplier \textit{op1} is \textit{fsrl1}, \textit{fsrl2} should not be the same as the \textit{fdest} of the prior operation.

8. When the prior operation is scalar, \textit{srcl} and \textit{srcl2} of the current operation should not be the same as \textit{dest} of the prior operation.

9. A \textit{pfld} should not immediately follow a \textit{pfld}

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

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

b. When using a \textit{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 \textit{bri} is executed.

c. If \textit{dest} is not zero, \textit{fsrl1} must not be the same as \textit{dest}.

d. When \textit{fsrl1} goes to multiplier \textit{op1} or to KR or KI, \textit{fsrl1} must not be the same as \textit{rdest}.

e. If \textit{dest} is not zero, \textit{srcl} and \textit{srcl2} must not be the same as \textit{dest}.

f. \textit{Isrl1} must not be the same register as \textit{isrl2} for the autoincrementing form of this instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Unit</th>
<th>Pipelined? Delayed?</th>
<th>Sets CC?</th>
<th>Faults</th>
<th>Performance Notes</th>
<th>Programming Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>adds</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>addu</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>andh</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>andnot</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>andnoth</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc.t</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bla</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnc</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnc.t</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>br</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bri</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bte</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>btne</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>calli</td>
<td>E</td>
<td>D</td>
<td></td>
<td>CC</td>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>facd.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>SE, RE</td>
<td></td>
</tr>
</tbody>
</table>

D-2
## INSTRUCTION CHARACTERISTICS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Unit</th>
<th>Pipelined? Delayed?</th>
<th>Sets CC?</th>
<th>Faults</th>
<th>Performance Notes</th>
<th>Programming Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>faddp</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>faddz</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>famov.r</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fiadd.w</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fisub.w</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fix.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld.y</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flush</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>fmulw.dd</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmulp</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>form</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frcp.p</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frsqr.p</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fst.y</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>fsub.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ftrunc.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fxr</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fzchkl</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fzckhks</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>intovr</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>IT</td>
<td></td>
</tr>
<tr>
<td>ixfr</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>ld.c</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld.x</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orh</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td>pfiadd.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfadd.p</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfiaddz</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfamov.r</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfam.p</td>
<td>A&amp;M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfeq.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfgt.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfhadd.w</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfsub.w</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfld.z</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfle.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfamam.p</td>
<td>A&amp;M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfsm.p</td>
<td>A&amp;M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfsub.p</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: 
- 8
- 4
- 2, 3
- f
- 6, 8
- 5
- c
## INSTRUCTION CHARACTERISTICS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Unit</th>
<th>Pipelined? Delayed?</th>
<th>Sets CC?</th>
<th>Faults</th>
<th>Performance Notes</th>
<th>Notes</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pftrunc.p</td>
<td>A</td>
<td>P</td>
<td>P</td>
<td>SE, RE</td>
<td>8</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>pfzchkl</td>
<td>G</td>
<td>P</td>
<td>P</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pfzchks</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pst.d</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shrd</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st.c</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st.x</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subs</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td>DAT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>subu</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>trap</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td>IT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xorh</td>
<td>E</td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOMESTIC SALES OFFICES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALABAMA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>5015 Bradford Dr., #2</td>
<td>Huntsville 35805</td>
<td>Tel: (205) 830-4010</td>
<td>FAX: (205) 837-2940</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ARIZONA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>11225 N. 28th Dr.</td>
<td>Suite D-214</td>
<td>Phoenix 85029</td>
<td>Tel: (602) 869-4980</td>
<td>FAX: (602) 869-4294</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>1161 N. El Dorado Place</td>
<td>Suite 301</td>
<td>Tucson 85715</td>
<td>Tel: (520) 295-9815</td>
<td>FAX: (520) 295-6304</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CALIFORNIA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>21516 Vanowen Street</td>
<td>Suite 116</td>
<td>Canoga Park 91303</td>
<td>Tel: (818) 704-3500</td>
<td>FAX: (818) 340-1144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>2250 E. Imperial Highway</td>
<td>Suite 219</td>
<td>El Segundo 90244</td>
<td>Tel: (310) 640-6240</td>
<td>FAX: (310) 640-7133</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>1510 Arden Way</td>
<td>Suite 101</td>
<td>Sacramento 95815</td>
<td>Tel: (916) 920-8096</td>
<td>FAX: (916) 920-8253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>9665 Chesapeake Dr.</td>
<td>Suite 325</td>
<td>San Diego 92132</td>
<td>Tel: (619) 292-9066</td>
<td>FAX: (619) 292-0929</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>* 400 N. Tustin Avenue</td>
<td>Suite 450</td>
<td>Santa Ana 92705</td>
<td>Tel: (714) 635-9642</td>
<td>FAX: (714) 635-9642</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>* 2700 San Tomas Expressway</td>
<td>2nd Floor</td>
<td>Santa Clara 95051</td>
<td>Tel: (408) 886-8088</td>
<td>TWX: 910-983-0235</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>COLORADO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>4405 Eastgate Drive</td>
<td>Suite 100</td>
<td>Colorado Springs 80907</td>
<td>Tel: (719) 594-6652</td>
<td>FAX: (303) 594-0720</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>330 S. Cherry St.</td>
<td>Suite 915</td>
<td>Denver 80222</td>
<td>Tel: (303) 321-8086</td>
<td>FAX: (303) 321-2399</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>10000 Southpark Drive</td>
<td>Suite 200</td>
<td>Englewood 80112</td>
<td>Tel: (303) 260-2860</td>
<td>FAX: (303) 260-3762</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CONNECTICUT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>301 Lee Farm Corporate Park</td>
<td>83 Woodside Heights Rd.</td>
<td>Danbury 06810</td>
<td>Tel: (203) 748-3100</td>
<td>FAX: (203) 784-0339</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FLORIDA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>6353 N.W. 6th Way</td>
<td>Suite 100</td>
<td>Ft. Lauderdale 33309</td>
<td>Tel: (305) 711-0050</td>
<td>TWX: 510-856-9407</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>Orlando 32822</td>
<td>Suite 340</td>
<td>Orlando 32822</td>
<td>Tel: (407) 240-6000</td>
<td>FAX: (407) 240-6097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>5950 T.G. Lee Blvd.</td>
<td>Suite 340</td>
<td>Gainesville 32608</td>
<td>Tel: (352) 765-6565</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GEORGIA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>20 Technology Parkway, N.W.</td>
<td>Suite 130</td>
<td>Norcross 30092</td>
<td>Tel: (404) 635-9795</td>
<td>FAX: (404) 605-9762</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ILLINOIS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>300 N. Martingale Road</td>
<td>Suite 400</td>
<td>Schaumburg 60173</td>
<td>Tel: (847) 429-6031</td>
<td>FAX: (312) 706-9762</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INDIANA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>877 Purdue Road</td>
<td>Suite 125</td>
<td>Indianapolis 46220</td>
<td>Tel: (317) 875-0623</td>
<td>FAX: (317) 875-8938</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IOWA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>1930 S. Andrews Drive N.E.</td>
<td>2nd Floor</td>
<td>Cedar Rapids 52402</td>
<td>Tel: (319) 393-3920</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>KANSAS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>10990 Goodbye Dr.</td>
<td>Suite 140, 6th Fl.</td>
<td>Overland Park 66210</td>
<td>Tel: (913) 345-2372</td>
<td>FAX: (913) 345-2076</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MARYLAND</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>10000 Junction Dr.</td>
<td>Suite 200</td>
<td>Annapolis Junction 20701</td>
<td>Tel: (301) 206-2865</td>
<td>FAX: (301) 206-3767</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MASSACHUSETTS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>5540 Centerview Dr.</td>
<td>Suite 215</td>
<td>Raleigh 27606</td>
<td>Tel: (919) 851-9537</td>
<td>FAX: (919) 861-8974</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OHIO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>3401 Park Center Drive</td>
<td>Suite 220</td>
<td>Dayton 45414</td>
<td>Tel: (513) 890-5350</td>
<td>TWX: 810-450-2528</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>7071 Orchard Lake Road</td>
<td>Suite 100</td>
<td>West Bloomfield 48322</td>
<td>Tel: (248) 851-8096</td>
<td>FAX: (248) 851-8770</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MINNESOTA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>3500 W. 80th St.</td>
<td>Suite 366</td>
<td>Bloomington 55431</td>
<td>Tel: (612) 833-6722</td>
<td>TWX: 916-576-2697</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>4th Floor</td>
<td>Suite 220</td>
<td>Bloomington 55431</td>
<td>Tel: (612) 837-5417</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MISSOURI</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>4203 Earth City Expressway</td>
<td>Suite 131</td>
<td>Earth City 63045</td>
<td>Tel: (314) 291-1990</td>
<td>FAX: (314) 291-3441</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NEW JERSEY</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>Parkway 109 Office Center</td>
<td>328 Newman Springs Road</td>
<td>Red Bank 07701</td>
<td>Tel: (732) 273-2223</td>
<td>FAX: (201) 749-0989</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NEW YORK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>250 Corporate Center</td>
<td>75 Livingston Avenue</td>
<td>First Floor</td>
<td>Tel: (201) 740-0111</td>
<td>FAX: (201) 740-0626</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pennsylvania</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>850 Cross Keys Office Park</td>
<td>Suite 14450</td>
<td>Pittsburgh 15235</td>
<td>Tel: (412) 492-2500</td>
<td>FAX: (412) 492-2501</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Puerto Rico</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>850 Expressway Dr.</td>
<td>South Suite 112</td>
<td>San Juan 00927</td>
<td>Tel: (787) 523-7381</td>
<td>FAX: (787) 523-2551</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rutgers University</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>25 Expressway Dr.</td>
<td>South Suite 104</td>
<td>Islandia 11722</td>
<td>Tel: (631) 312-3300</td>
<td>TWX: 510 1327-6236</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>North Carolina</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>Westgate Business Center</td>
<td>300 Rosemary Road</td>
<td>Surfside Beach 2299</td>
<td>Tel: (910) 497-3600</td>
<td>FAX: (910) 497-3125</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Virginia</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>15254 N.W. Greenbrier Parkway</td>
<td>Building B</td>
<td>10001 Greenbrier Parkway</td>
<td>Tel: (703) 846-7000</td>
<td>FAX: (703) 846-7295</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Washington</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>455 Pennsylvania Avenue</td>
<td>Suite 230</td>
<td>Fort Washington 19034</td>
<td>Tel: (215) 641-1000</td>
<td>FAX: (215) 641-0768</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>West Virginia</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>841 Penny Center Blvd.</td>
<td>Suite 610</td>
<td>Charleston 25031</td>
<td>Tel: (304) 323-4000</td>
<td>FAX: (304) 323-4000</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Wisconsin</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Corp.</td>
<td>330 S. Executive Dr.</td>
<td>Suite 100</td>
<td>Brookfield 53005</td>
<td>Tel: (414) 784-8087</td>
<td>FAX: (414) 796-2115</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CANADA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Semiconductor of Canada, Ltd.</td>
<td>4585 Canada Way</td>
<td>Suite 100</td>
<td>Burnaby V5G 4L6</td>
<td>Tel: (604) 298-2037</td>
<td>FAX: (604) 298-2935</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ONTARIO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Semiconductor of Canada, Ltd.</td>
<td>2650 Queenstreet Drive</td>
<td>Suite 250</td>
<td>Otawa K2B 8H6</td>
<td>Tel: (613) 529-7970</td>
<td>FAX: (613) 529-5936</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Semiconductor of Canada, Ltd.</td>
<td>190 Atwell Drive</td>
<td>Suite 500</td>
<td>Rexdale MSG 6H6</td>
<td>Tel: (416) 675-2105</td>
<td>FAX: (416) 675-2439</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>QUEBEC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Semiconductor of Canada, Ltd.</td>
<td>620 St. Jean Boulevard</td>
<td>Suite 115</td>
<td>Laval H9R 3K2</td>
<td>Tel: (514) 549-6310</td>
<td>FAX: (514) 549-6004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>Address</td>
<td>Telephone</td>
<td>Fax</td>
<td>Email</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DENMARK</td>
<td>Intel Denmark A/S 2400 Copenhagen NV Tel: (45) (31) 19 60 33</td>
<td>19567</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINLAND</td>
<td>Intel Finland OY Ruosilantie 2 02090 Helsinki Tel: (0) 504 644 TLX: 123332</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRANCE</td>
<td>Intel Corporation S.A.R.L. 1, Rue Edison-BP 303 78703 Saint Quenin-en-Yvelines Cedex Tel: (33) (1) 30 57 70 50 TLX: 699016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEORGIA</td>
<td>Intel Semiconductors GmbH* Donachstrasse 1 80016 Friedenau bei Muenchen Tel: (49) 089/90092-0 TLX: 5-23177</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GERMANY</td>
<td>Intel Semiconductors GmbH Hohenzollernstrasse 5 20009 Hamburg 1 Tel: (49) 040/498080 TLX: 371215</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITALY</td>
<td>Intel Corporation Italia S.p.A. Milano 30500-Palazzo E 20090 Assago (MI) Tel: (39) (02) 89200950 TLX: 341269</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NETHERLANDS</td>
<td>Intel Semiconductors BV* Postbus 3099 CC Rotterdam Tel: (31) 10 407.11.11 TLX: 22263</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORWAY</td>
<td>Intel Norway A/S Hovemenen 4 PO Box 92 2013 Sottriten Tel: (47) (6) 842 420 TLX: 76518</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWITZERLAND</td>
<td>Intel Semiconductors A.G. Zuerichstrasse 8185 Winkel-Ruei bei Zuerich Tel: (41) 01/860 62 62 TLX: 625977</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNITED KINGDOM</td>
<td>Intel Corporation (U.K.) Ltd.* Pipers Way Swindon, Wiltshire SN3 1RJ Tel: (44) (0793) 696000 TLX: 4444478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUROPEAN SALES OFFICES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUSTRIA</td>
<td>Bacher Electronics G.m.b.H Rotenauerwegasse 25 1120 Wien Tel: (01) 362 89 90 20000 46 TLX: 31532</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BELGIUM</td>
<td>Inelo Belgium S.A. Av des Croix de Guerre 94 1120 Brussels Oorlogskruislaan 60, 1120 Brussels Tel: (32) (02) 616 01 60 TLX: 64479 or 22990</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DENMARK</td>
<td>ITT-Multicomponent Ravnifik 29 2600 Glostrup Tel: (39) (0) 62 45 65 46 TLX: 33 355</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINLAND</td>
<td>QY Fintronic AB Mekonkatu 24 00100 Helsinki Tel: (358) (0) 6926022 TLX: 124224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRANCE</td>
<td>Anex Zone industrielle d'Antony 48, rue de l'Europe BP 102 92164 Antony cedex Tel: (33) (1) 48 66 21 12 TLX: 250067</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISRAEL</td>
<td>Electrostar Ltd. 11 Ronshne Street P.O.B. 39380 Tel-Aviv 61382 Tel: (972) 03-475151 TLX: 33638</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITALY</td>
<td>Intel Divisione ITT Industries GmbH Milano Milanofiori E5 20090 Assago (MI) Tel: (39) 02/2420470 TLX: 311351</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NETHERLANDS</td>
<td>Koring en Hartman Elektrotechniek B.V. Energieweg 1 2677 AP Delft Tel: (31) (0) 15/609096 TLX: 38520</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORWAY</td>
<td>A.TD Portugal LDA Rua Dos Lusikados, 5 Sala B 1300 Lisbon Tel: (35) (1) 64 80 91 TLX: 61562</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWITZERLAND</td>
<td>Intel Semiconductors A.G. Zuerichstrasse 8185 Winkel-Ruei bei Zuerich Tel: (41) 01/860 62 62 TLX: 625977</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNITED KINGDOM</td>
<td>Intel Corporation (U.K.) Ltd.* Pipers Way Swindon, Wiltshire SN3 1RJ Tel: (44) (0793) 696000 TLX: 4444478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EUROPEAN DISTRIBUTORS/REPRESENTATIVES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUSTRIA</td>
<td>Tekelec-Atronic C/le des Bruyeres Rue Carl Vernet - BP 2 92310 Sevres Tel: (33) (1) 45 34 75 35 TLX: 204552</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BELGIUM</td>
<td>Electronic 2000 AG Stabilung7berg 12 8000 Muenchen 62 Tel: (49) 089/742001-0 TLX: 522551</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DENMARK</td>
<td>ITT Multicomponent Postfach 1265 Bahnhofstrasse 44 7141 Moeglingen Tel: (49) 07141/4579 TLX: 7264472</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINLAND</td>
<td>Jermyn GmbH Im Haralden 2 6250 Limburg Tel: (39) 069/110/508-0 TLX: 415257-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRANCE</td>
<td>Metrolage GmbH Meilingergasse 49 8000 Muenchen 71 Tel: (49) 09757/5042-0 TLX: 5213183</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISRAEL</td>
<td>Proelectron Vertriebs GmbH Max Planck Strasse 1-3 8070 Delsack Tel: (49) 06103/3043-3 TLX: 471903</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITALY</td>
<td>Micro Marketing Ltd. Gianagalea Office Park Gianagalea Co. Dublin Tel: (21) (353) (0) 85 63 25 TLX: 31564</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NETHERLANDS</td>
<td>Nordisk Elektronik (Norge) A/S Postbus 1239 8000 Muenchen 2 Tel: (49) 089/53 60 570 TLX: 203973</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORWAY</td>
<td>A.TD Portugal LDA Rua Dos Lusikados, 5 Sala B 1300 Lisbon Tel: (35) (1) 64 80 91 TLX: 61562</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWITZERLAND</td>
<td>Bytech-Conway Systems 3 The Western Centre Western Road Bracknell RG12 1RW Tel: (44) (0344) 55333 TLX: 847201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNITED KINGDOM</td>
<td>Jermyn Vesty Estate Olford Road Sevrcales Kent TN1 5EU Tel: (44) (0) 732 450144 TLX: 69142</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWEDEN</td>
<td>MMD Unit 8 Southview Park Caversham Reading Berkshire RG2 0AF Tel: (44) (0) 734146666 TLX: 846669</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DOMESTIC SERVICE OFFICES

ALABAMA
*Intel Corp. 5015 Bradford Dr., Suite 2 Huntsville 35805 Tel: (205) 833-4010

ALASKA
Intel Corp. c/o TransAlaska Data Systems 300 Old Steelee HwyFairbanks 99701-5120 Tel: (907) 452-4401

ARIZONA
*Intel Corp. 500 E. Dev Thru, Suite M-15 Sierra Vista 85633 Tel: (520) 459-5010

CALIFORNIA
**Intel Corp. 21515 Vancoven St., Ste. 116 Canoga Park 91303 Tel: (818) 724-6500
**Intel Corp. 2250 E. Imperial Hwy., Ste. 218 El Segundo 90245 Tel: (310) 200-6040
*Intel Corp. 1900 Prairie City Rd. Fontana 92335 Tel: (909) 351-6143 1-800-685-3548
Intel Corp. 9665 Chisapeake Dr., Suite 325 San Diego 92123-1206 Tel: (619) 252-8086
**Intel Corp. 400 N. Twin Avenue Suite 450 San Antonio 78205 Tel: (714) 835-9642

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

CONNECTICUT
**Intel Corp. 351 Lee Farm Corporate Park 83 Wooster Hegins Rd. Danbury 06810 Tel: (203) 748-3130

FLORIDA
**Intel Corp. 6363 N.W. 6th Way, Ste. 100 Ft. Lauderdale 33309 Tel: (305) 771-0600
*Intel Corp. 5850 T.G. Lee Blvd., Ste. 340 Brandon 33525 Tel: (813) 240-8000

GEORGIA
**Intel Corp. 3280 Poirite Pkwy., Ste. 200 Norcross 30092 Tel: (404) 449-0541

HAWAII

ILLINOIS
**Intel Corp. 300 N. Martingle Rd., Ste. 400 Schaumburg 60173 Tel: (312) 605-6031
**Intel Corp. 8777 Purdue Rd., Ste. 125 Indianapolis 46268 Tel: (317) 875-0623

INDIANA
**Intel Corp. 9665 Chisapeake Dr., Suite 325 San Diego 92123-1206 Tel: (619) 252-8086
**Intel Corp. 400 N. Twin Avenue Suite 450 San Antonio 78205 Tel: (714) 835-9642

KANSAS
*Intel Corp. 10885 Cody, Suite 140 Overland Park 66210 Tel: (913) 945-2727

MARYLAND
**Intel Corp. 10301 Juncton Dr., Suite 200 Annapolis Junction 20701 Tel: (301) 256-2860 FAX: 301-206-3677

MASSACHUSETTS
**Intel Corp. 3 Carlisle Rd., 2nd Floor Westford 01886 Tel: (978) 692-1060

MICHIGAN
**Intel Corp. 7071 Orchard Lake Rd., Ste. 100 West Bloomfield 48322 Tel: (313) 851-8905

MINNESOTA
**Intel Corp. 3500 W. 80th St., Suite 360 Bloomington 55431 Tel: (612) 835-8722

MISSOURI
**Intel Corp. 4203 Earth City Exp., Ste. 131 Earth City 63045 Tel: (314) 291-1990

NEW JERSEY
**Intel Corp. 300 Sylvan Avenue Englewood Cliffs 07632 Tel: (201) 567-0821
**Intel Corp. Parkay 109 Office Center 328 Newman Springs Road Red Bank 07701 Tel: (732) 747-2233

NEW YORK
**Intel Corp. 2950 Expressway Dr. South Islandia 11722 Tel: (516) 231-3300
**Intel Corp. Westage Business Center Bldg. 300, Route 9 Floriell 12524 Tel: (914) 897-3860

NORTH CAROLINA
*Intel Corp. 5800 Executive Dr., Ste. 105 Charlotte 28212 Tel: (704) 568-8966
**Intel Corp. 2700 Wycliff Road Suite 102 Raleigh 27607 Tel: (919) 781-8022

OHIO
**Intel Corp. 3401 Park Center Dr., Ste. 220 Dayton 45414 Tel: (513) 890-5390
**Intel Corp. 25700 Science Park Dr., Ste. 100 Beachwood 44122 Tel: (216) 464-2736

OREGON
Intel Corp. 15254 N.W. Greenbrier Parkway Building B Beaverton 97005 Tel: (503) 645-8051
**Intel Corp. 5200 N E. Elm Young Parkway Hillsboro 97123 Tel: (503) 681-8080

 PENNSYLVANIA
**Intel Corp. 455 Pennsylvania Ave., Ste. 230 Fort Washington 19034 Tel: (215) 641-1000
Intel Corp. 400 Penn Center Blvd., Ste. 610 Pittsburgh 15225 Tel: (412) 823-4970

CANADA
ONTARIO
Intel Semiconductor of Canada, Ltd. 2650 Gunterview Dr., Ste. 250 Ottawa K2B 8K8 Tel: (613) 829-9714 FAX: 613-820-5936
Intel Semiconductor of Canada, Ltd. 190 Atlantic Dr., Ste. 102 Rexdale M9W 6B6 Tel: (416) 675-2105 FAX: 416-675-2438

CUSTOMER TRAINING CENTERS

CALIFORNIA
2700 San Tomas Expy. Santa Clara 95051 Tel: (408) 870-1700 1-800-241-0306

ILLINOIS
300 N. Martingle Road Suite 300 Schaumburg 60173 Tel: (708) 705-5700 1-800-421-0386

MINNESOTA
3500 W. 86th Street Suite 360 Bloomington 55431 Tel: (952) 933-6722

NEW YORK
2950 Expressway Dr., South Islandia 11722 Tel: (506) 231-3300

SYSTEMS ENGINEERING MANAGERS OFFICES

MINNESOTA
3500 W. 86th Street Suite 360 Bloomington 55431 Tel: (952) 933-6722

NEW YORK
2950 Expressway Dr., South Islandia 11722 Tel: (506) 231-3300

**System Engineering locations
**Carry-in locations
**Carry-in/mail-in locations