THE K-1 ARCHITECTURE MANUAL

THE K-1 ARCHITECTURE MANUAL

Version 3.0

June 5, 1989

CONFIDENTIAL CONFIDENTIAL

This document contains information proprietary to Amdahl Key Computer Laboratories (KCL). Use or disclosure without the written permission of an officer of KCL is expressly forbidden. Copyright © KCL 1987, 1988, 1989.

Report number KC-101

Contents

List of Figures vii
List of Tables ix

CHAPTER 1. INTRODUCTION 1-1

CHAPTER 2. K-1 ARCHITECTURE OVERVIEW 2-1
- 2.1 Introduction 2-1
- 2.2 Conventions 2-3
- 2.3 Data Types 2-3
  - 2.3.1 Integers 2-3
  - 2.3.2 Floating-Point Numbers 2-4
- 2.4 Instruction Formats 2-6
- 2.5 General Registers 2-7
- 2.6 Flags and Conditional Branching 2-7
- 2.7 ELF Flags 2-7
- 2.8 Memory 2-7
  - 2.8.1 Precision and Alignment 2-8
  - 2.8.2 Addressing Modes 2-8
- 2.9 Processor States 2-9
- 2.10 Virtual Memory 2-9
  - 2.10.1 Address Mapping 2-10
    - 2.10.1.1 Instruction Mapping 2-10
    - 2.10.1.2 Data Mapping 2-11
- 2.11 Cache Effects 2-14
  - 2.11.1 Cache Organization and Addressing 2-14
  - 2.11.2 Instruction Stache 2-15
  - 2.11.3 Cache Coherence 2-15
  - 2.11.4 Interprocessor Synchronization 2-16
- 2.12 Program Execution 2-16
  - 2.12.1 Delayed Branching 2-17
  - 2.12.2 Conditional Execution 2-18
- 2.13 Early Load 2-18
- 2.14 Processor Status 2-19
- 2.15 Input/Output 2-23
  - 2.15.1 I/O Interrupts 2-23
- 2.16 Timers 2-23
- 2.17 Traps, Interrupts, and Machine Checks 2-24
  - 2.17.1 The Trap Sequence 2-26

2.18 Reset Operation 2-29

CHAPTER 3. K-1 Instructions 3-1

3.1 Instruction Formats 3-1

CHAPTER 4. Floating-Point Instructions 4-1

4.1 Floating-Point Compare Instructions 4-1
4.2 Floating-Point Conversion Instructions 4-5
4.3 Floating-Point Computation Instructions 4-13

CHAPTER 5. Load and Store Instructions 5-1

5.1 Referencing Memory 5-1
5.2 Memory-Referencing Instruction Traps 5-3
5.3 Load Instructions 5-5
5.4 Store Instructions 5-34
5.5 Special Load/Store Instructions 5-43
5.6 ELF Flag Instructions 5-49
5.7 Data Watchpoint 5-53

CHAPTER 6. Integer Instructions 6-1

6.1 Integer Arithmetic Instructions 6-1
6.2 Integer Compare Instructions 6-16
6.3 Data Moving Instructions 6-29
6.4 Boolean Instructions 6-39
6.5 Shift Instructions 6-48
6.6 Bit Count and Reverse Instructions 6-54
6.7 Flag Instructions 6-58
6.8 Check Instructions 6-61

CHAPTER 7. Transfer of Control Instructions 7-1

CHAPTER 8. Processor Status Register and Timer Instructions 8-1

CHAPTER 9. Virtual Memory and Cache Instructions 9-1

CHAPTER 10. Trap Instructions 10-1

CHAPTER 11. I/O Instructions 11-1

CHAPTER 12. Miscellaneous Instructions 12-1

CHAPTER 13. Undefined Opcodes 13-1

Appendix A. Instruction Index (Alphabetic) A-1

Appendix B. Instruction Index (Numeric) B-1

Appendix C. Instruction Timing Considerations C-1

C.1 Clock Cycles C-1
C.2 Instruction Issue C-1
C.3 Pipelining C-2
C.4 Functional Unit Latency and Interlocks C-2
C.5 K-1 Interlocks C-3
C.5.1 Register Interlocks C-3
C.5.2 Flag Interlocks C-4
C.5.3 I1/I0 Interlocks C-5
C.5.3.1 I0 Interlocked C-5
C.5.3.2 Serial Instructions C-5
C.5.3.3 I1/I0 Read Port Conflicts C-5
C.5.3.4 I1/I0 Register and Flag Conflicts C-6
C.5.4 Floating-Point Divide/Square Root Unit C-6
C.5.5 Load/Store Unit C-7
C.6 Branching C-7
C.6.1 Branching to a PC-Relative or Absolute Address C-7
C.6.2 Branching to an Address in a Register C-8
C.7 Special Instructions C-8
C.7.1 nop C-8
C.7.2 rps, wps, spl, and srm C-9
C.7.3 Trap Instructions and Instructions Which Trap C-9
C.7.4 exts C-10
C.7.5 ickill C-10
C.7.6 iskill C-10
C.8 Load/Store Timing C-11
C.8.1 load instructions C-11
C.8.2 store instructions C-11
C.8.3 ldpage C-11
C.8.4 pcl and zcl C-11
C.8.5 dflush C-12
C.8.6 rfec and wfec C-12
C.8.7 rios and wios C-12

Appendix D. Trap Handling D-1

D.1 Trap Data and Trap Recovery D-1
D.1.1 Primary Trap Data D-1
D.1.1.1 Trap Summary D-1
D.1.1.2 Trap Locators D-5
D.1.1.2.1 IMA Trap Locators D-8
D.1.1.2.2 L/S Trap Locators D-10
D.1.1.2.3 Divide Trap Locators D-10
D.1.2 Reading Primary Trap Data D-10
D.1.3 Load/Store Trap Data D-11
D.1.4 Storing Load/Store Trap Data D-14
D.2 Use of exts Instructions D-15

D.3 Example Trap Handler D-15

Appendix E. Memory System Specifics E-1

E.1 Introduction E-1
E.2 Data Order E-1
E.3 Error-Correcting Code E-3
E.4 CPU Features for Diagnosing the Memory System E-3
E.5 Console Interactions and Error Logging E-3
E.6 Cache Coherency E-3
E.7 Memory System Timing E-3

Appendix F. I/O System Specifics F-1

F.1 Introduction F-1
F.2 Instruction Descriptions F-3
F.2.1 wios F-3
F.2.2 rios F-7
F.3 Control and Status Register Formats F-11
F.3.1 Channel Control Register Format F-11
F.3.2 Channel Status Register Format F-12
F.3.3 IOCA Status Register Format F-13

Appendix G. Console Specifics G-1

G.1 Introduction G-1
G.2 Interrupts G-1
G.3 TTY Interface G-1
G.4 Performance Counters G-2
G.5 Clock Control G-2
G.6 Scan Control and Hidden State G-2
G.7 Bootstrap Procedure G-2

Appendix H. Implementation Dependencies H-1

H.1 The Version 1 Implementation H-1

Appendix I. Floating-Point Operation Details I-1

I.1 Abbreviations I-1
I.2 Use of NaN I-1
I.3 Floating-Point Overflow and Underflow I-2
I.4 Floating-Point Operation Tables I-2
I.4.1 Floating-Point Addition I-3
I.5 Floating-Point Negation I-5
I.6 Floating-Point Subtraction I-5
I.7 Floating-Point Multiplication I-6
I.8 Floating-Point Division I-7
I.9 Floating-Point Square Root I-8
I.10 Floating-Point Conversions I-9
I.10.1 Conversion Between Floating-Point Formats I-9

I.10.2 Conversion Between Floating-Point and Integer Formats....................I-11

INDEX..Index-1

List of Figures

Figure 2-1. The K-1 Processor and Its Interconnections 2-2
Figure 2-2. Signed 64-Bit Integer Data Type 2-4
Figure 2-3. IEEE 32-Bit Precision Floating-Point Format 2-5
Figure 2-4. IEEE 64-Bit Precision Floating-Point Format 2-5
Figure 2-5. Address Calculation for Load/Store Instructions 2-8
Figure 2-6. Virtual Address Format 2-10
Figure 2-7. Instruction Page Table Format 2-11
Figure 2-8. Virtual Page Number 2-13
Figure 2-9. Data Page Table Format 2-13
Figure 2-10. Uptime Counter 2-24
Figure 2-11. Interval Timer Register 2-24
Figure 2-12. Restart PC 2-26
Figure 3-1. Register Instruction Format 3-2
Figure 3-2. Unconditional Short Constant Instruction Format 3-3
Figure 3-3. Conditional Short Constant Instruction Format 3-3
Figure 3-4. Long Constant Instruction Format 3-4
Figure 3-5. PC-Relative Branch Instruction Format 3-4
Figure 3-6. Absolute Branch Instruction Format 3-5
Figure 3-7. Register Branch Instruction Format 3-6
Figure 3-8. exts Instruction Format 3-6
Figure 4-1. Floating-Point Conversion srcb Argument 4-5
Figure 5-1. Data Watchpoint Table Entry Format 5-53
Figure 6-1. boof srca Argument 6-59
Figure D-1. Trap Summary D-2
Figure D-2. Divide Trap Locator D-5
Figure D-3. Load/Store Trap Locator D-5
Figure D-4. IMA Trap Locator D-5
Figure D-5. IMA Trap Byte D-6
Figure D-6. IMA Trap Word D-6
Figure D-7. Load/Store Trap Data D-12
Figure D-8. addr Data Field Format D-13
Figure D-9. Combined srca/srcb Data Field Format D-13
Figure D-10. ldpage Data Field Format D-13

Figure D-11. slstrpd srcb Operand Format D-14
Figure E-1. ECC Groupings Within a Sub-Line E-2
Figure F-1. srca and srcb Operand Format F-3
Figure F-2. CRP and NCRP Pointer Structure F-5
Figure F-3. Type 0001 rdst Register Format F-8
Figure F-4. Type 0010 rdst Register Format F-8
Figure F-5. Type 0011 rdst Register Format F-9
Figure F-6. Type 0100 rdst Register Format F-9
Figure F-7. Type 100x rdst Register Format CRP Queue Pointers F-10
Figure F-8. Type 100x rdst Register Format NCRP Queue Pointers F-10
Figure F-9. Type 100x rdst Register Format CRP Locations (0-4;6-7;15) F-10

List of Tables

Table 2-1. IEEE 32-Bit Precision Floating-Point Values ... 2-5
Table 2-2. IEEE 64-Bit Precision Floating-Point Values ... 2-6
Table 2-3. Instruction Page Table Size Specifier ... 2-11
Table 2-4. Memory Reference Modes ... 2-12
Table 2-5. Processor Status Register ... 2-20
Table 2-6. Arithmetic Trap Enables and Exception Flags ... 2-20
Table 2-7. Rounding Modes ... 2-21
Table 2-8. Low-to-High Byte Addressing ... 2-22
Table 2-9. External Interrupt Priority Levels ... 2-27
Table 2-10. Trap Vectoring ... 2-27
Table 2-11. D0 and D1 Decoding for 32-bit Instructions ... 2-28
Table 2-12. D0 and D1 Decoding for a 64-bit Instruction ... 2-28
Table 3-1. Source Operand Control ... 3-2
Table 3-2. Delayed Execution Control Decoding ... 3-5
Table 4-1. Floating-Point Comparisons ... 4-2
Table 5-1. Memory Referencing Instruction Traps ... 5-3
Table 6-1. boof Code (BC) Decoding ... 6-59
Table 9-1. dflush srcb Control Functions ... 9-7
Table C-1. Functional Unit Latencies ... C-3
Table D-1. Trap Encodings ... D-3
Table D-2. Machine Check Encoding ... D-4
Table D-3. Trap Bit Field Encodings ... D-7
Table D-4. I0 and I1 Decoding for 32-bit Instructions ... D-7
Table D-5. I0 and I1 Decoding for a 64-bit Instruction ... D-8
Table D-6. rtrpd srca encoding ... D-11
Table D-7. TY Encoding ... D-14
Table I-1. Addition Results ... I-4
Table I-2. Multiplication Results ... I-6
Table I-3. Division Results ... I-7
Table I-4. Square Root Results ... I-8
Table I-5. Double to Single Conversion Results ... I-9
Table I-6. Single to Double Conversion Results ... I-10
Table I-7. Signed or Unsigned Integer to Floating-Point Conversion Results ... I-10
Table I-8. Floating-Point to Signed Integer Conversion Results ... I-11

CHAPTER 1. INTRODUCTION

The Amdahl Key Computer Laboratories K-1 family of supercomputers is designed to meet the needs of the modern large computer user. Equipped with an extremely large address space, very high I/O and memory bandwidths, and pipelined functional units, the K-1 is ideally suited to tackling today's highly compute intensive problems. At the same time, a simple, straightforward, yet powerful instruction set makes the K-1 architecture one of the best possible targets for optimizing compilers. A complete set of virtual memory features rounds out the architecture and makes it possible to run modern operating systems smoothly and efficiently.

This manual describes the architecture of the central processing unit of the K-1 family. Different members of this family may contain different numbers of central processing units or have different physical memory sizes or attached I/O processors, but the central processing units all function identically. The main body of this manual contains all the information necessary for programming the K-1, with the exception of instruction timing information which is provided in Appendix C.

The following sections describe the main features of the K-1 architecture and are followed by individual instruction descriptions. Instruction indexes are provided in Appendices A and B for quick reference. Appendix C gives detailed instruction timing information and Appendix D gives trap handling details along with an example of a trap handler. Appendices E, F and G give memory system, I/O system, and Front-End system specifics, respectively. Implementation dependent aspects of the K-1 architecture are noted in the text with square brackets ([]) and are references to Appendix H, which explains features particular to this implementation.

The examples used in this manual to describe K-1 machine instructions conform to the K-1 assembly language syntax, which is described in a separate manual, *The K-1 Assembly Language Reference Manual.*

Other documents of interest are:

KC-126: K-1/IOP Software Interface Specification
KC-109: The K-1 Assembly Language Reference Manual
ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic

CHAPTER 2. K-1 ARCHITECTURE OVERVIEW

2.1 Introduction

The K-1 architecture is designed for general purpose/scientific computing. It has been optimized to allow extremely high performance implementations such as the Amdahl Key Computer Laboratories K-1 system. It provides an extensive set of high-precision, IEEE-compatible, floating-point instructions, as well as a full complement of integer, logical and addressing operations. In addition, there are instructions to manipulate the virtual memory system, to control the caching of main memory data, and to control input/output.

The K-1 computer can be divided into three main parts: the Central Processing Units (CPUs, or simply "processors"), the memory subsystem, and the I/O subsystem. Each CPU can be further divided into three main subsections: the instruction fetch and issue units, the register file, and the functional units. Figure 2-1 shows the major interconnections within a K-1 CPU, and its connections to the memory and I/O subsystems.

The most central part of a K-1 CPU is the register file, containing up to 64 general-purpose registers of 64 bits each. The register file is used to store data items of all types. Functional units take their inputs from registers, or from constants that are part of the instruction. Functional unit results are always stored into registers. Most instructions can specify three independent register addresses. For example, the **add** instruction adds two registers together and stores the result into a third register.

There are five different types of functional units which process information from the registers: the **integer**, **load/store**, **floating-point add**, **floating-point multiply**, and **floating-point divide/square root** units. With the exception of a few special instructions that affect the internal state of the CPU, each instruction in the architecture is executed by exactly one functional unit. Timings for the individual functional units are given in Appendix C. The number of each type of functional unit present in a given K-1 CPU is model-dependent; however, every K-1 CPU has at least one unit of each type. A program will give identical results regardless of the number of units; only the execution time will be affected.

The K-1 main memory system consists of a very large, uniformly addressed memory space. Two large caches insulate each K-1 processor from the access time of main memory. One cache is used exclusively to hold instructions and the other to hold data. The caches operate transparently, but there is no hardware coherence between the instruction and data caches. The hardware does, however, maintain cache coherence among the data caches of all CPUs in a multiprocessor system. The architecture provides instructions for manipulating the caches and for updating memory.

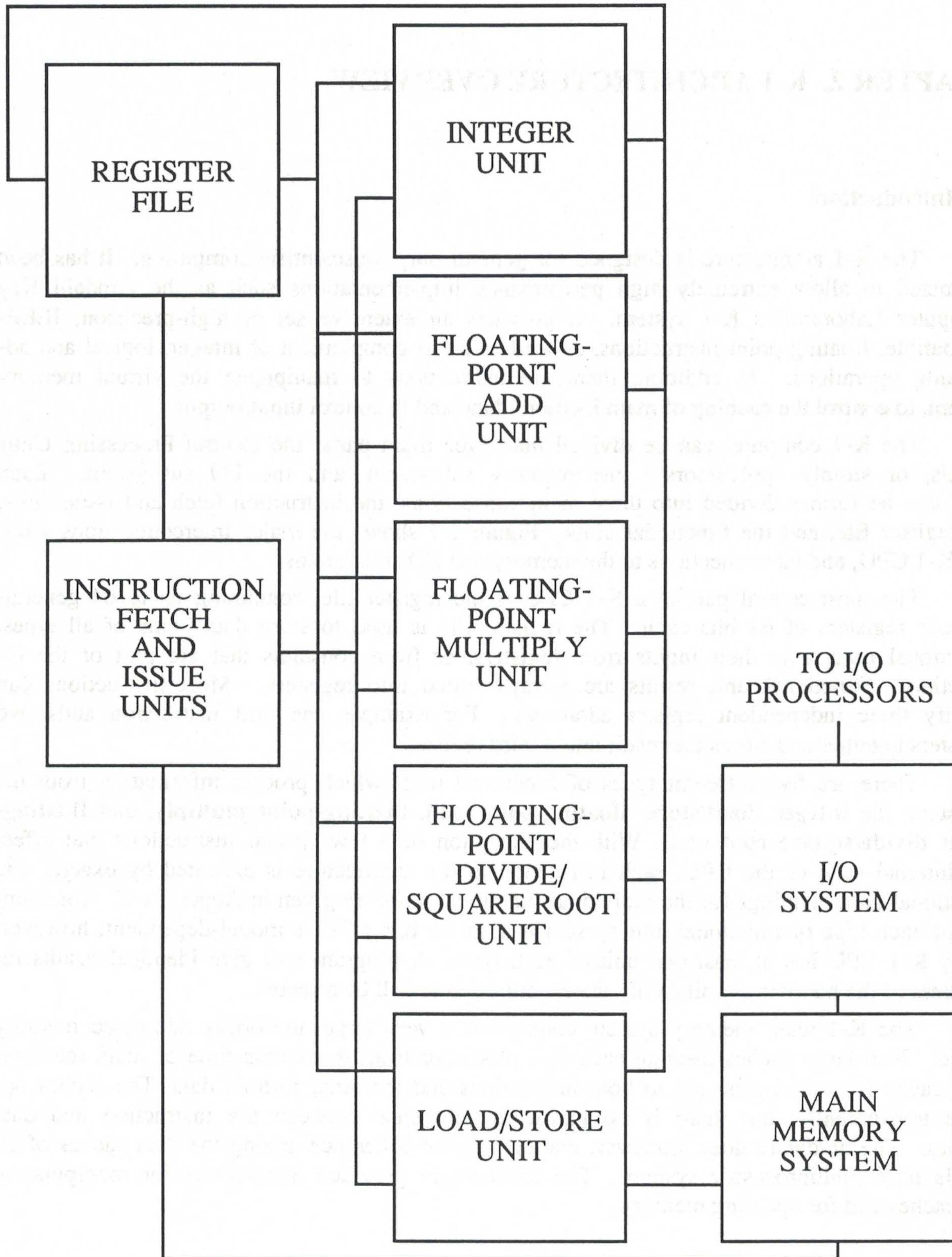


Figure 2-1. The K-1 Processor and Its Interconnections

I/O in the K-1 architecture is performed through attached I/O processors that interface via a number of very high speed I/O busses. These busses are operated by I/O controllers within the K-1, which transfer data directly to and from main memory. Instructions are provided to send control information to, and receive status from, the I/O subsystem.

2.2 Conventions

Every numeric data type in this manual is assumed to have its most significant bit on the left and its least significant bit on the right. Bits are numbered in ascending order from least significant to most significant, right-to-left, starting with zero. Bytes are numbered in the reverse order from most significant to least significant, left-to-right. This numbering scheme is commonly referred to as "big endian". The only exception to this is related to the **Byte Order Low-to-High** feature which allows the ordering of bytes in memory to be in ascending order from least significant to most significant, right-to-left (commonly referred to as "little endian"). However, even in this case, the bit numbering remains right-to-left.

A data type can be *zero-extended* from its natural size to a larger size by appending sufficient high-order zero bits to make up the difference. For example, a byte can be zero-extended to 64 bits by appending 56 high-order zero bits. Similarly, a signed quantity can be *sign-extended* by appending sufficient high-order copies of the sign bit. For example, a 32-bit data type can be sign-extended to 64 bits by appending 32 high-order copies of bit 31.

The letters **K** and **M** indicate 1,024 and 1,048,576 units of something, usually bytes.

The notation *m..n* indicates a contiguous range of bits within a word or field. Bit number *m* is the most significant end of the range, and bit number *n* is the least significant end. If the name of the field is **NAME**, then such a range is indicated by **NAME<*m..n*>**.

2.3 Data Types

The K-1 architecture supports a number of data types and precisions. When a data item is in a register, it may be a signed or unsigned integer of length 8, 16, 32, or 64 bits, or a 32-bit or 64-bit IEEE floating-point number. When in memory, only the precision of the data item (8, 16, 32, or 64 bits) is important. Quantities smaller than 64 bits are always right-adjusted within a register.

2.3.1 Integers

Integers are 8-, 16-, 32-, 33-, 53-, or 64-bit quantities and may be either signed or unsigned. Signed integers are two's complement values whose most significant bit is the sign bit, S. Figure 2-2 shows the format of a signed 64-bit integer. The 53-bit integer format is used only for the integer multiply instructions; the 33-bit integer format is used only for the integer divide instructions.

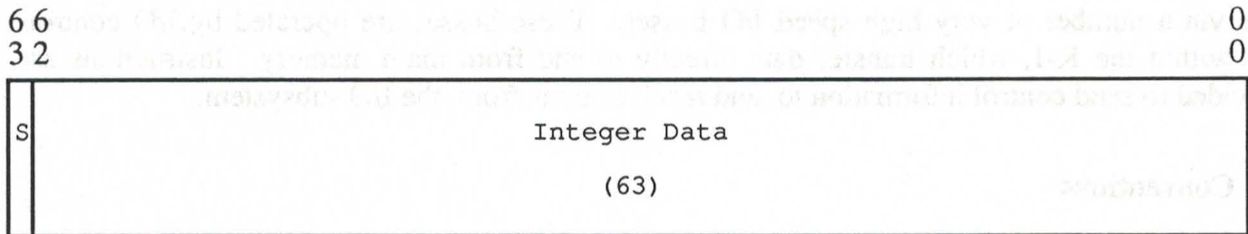


Figure 2-2. Signed 64-Bit Integer Data Type

2.3.2 Floating-Point Numbers

Two precisions of floating-point numbers are supported: 32-bit (single) and 64-bit (double). Separate instructions are provided for each floating-point operation for each precision. The formats conform to the ANSI/IEEE Standard 754-1985 floating-point single and double formats.

Both floating-point precisions contain a sign bit, an exponent field, and a fraction field. The value represented is always a sign-magnitude mantissa times a power of two determined by the exponent. The mantissa consists of a "hidden" bit followed by the fraction. The floating-point format also provides certain special values, such as NaN (Not-a-Number), infinity, negative zero and denormalized numbers (allowing for gradual underflow).

The IEEE standard specifies two types of NaNs: signaling and quiet. Both types of NaNs must have a maximum exponent and a non-zero fraction. The K-1 recognizes a NaN as quiet if the most significant fraction bit is a one, and as signaling if it is zero. When given a NaN as input, all non-trapping floating-point operations (except compares and negate) will generate a quiet NaN. When given as operands to floating-point instructions, signaling NaNs cause an invalid operation trap if enabled. Whenever a NaN is output by a floating-point operation (except negate), it will be a quiet NaN in the form shown in Tables 2-1 and 2-2 with the sign bit set to zero. The floating-point negate function is considered to be a data moving operation, and thus never changes its input (except for the sign bit), and never causes a trap.

Most floating-point numbers are normalized, meaning that their most significant mantissa bit (called the hidden bit) is a one. This bit, therefore, does not need to be present in the representation and is omitted. When computing the value represented by a floating-point number, the hidden bit must be reinserted. However, in order to extend the negative exponent range and allow for gradual underflow, the floating-point format provides a class of numbers called denormalized numbers. These numbers, which are very close to zero in value, have a hidden bit of zero (and are therefore not normalized).

32-bit (single) precision floating-point format numbers have a sign bit **S**, an 8-bit exponent **EXP**, and a 23-bit fraction **F** (Figure 2-3). The value, **V**, represented by this format is computed as in Table 2-1. Note that Figure 2-3 shows this format as it would be stored in a 64-bit register; 32-bit precision floating-point numbers can be stored in 32 bits in memory.

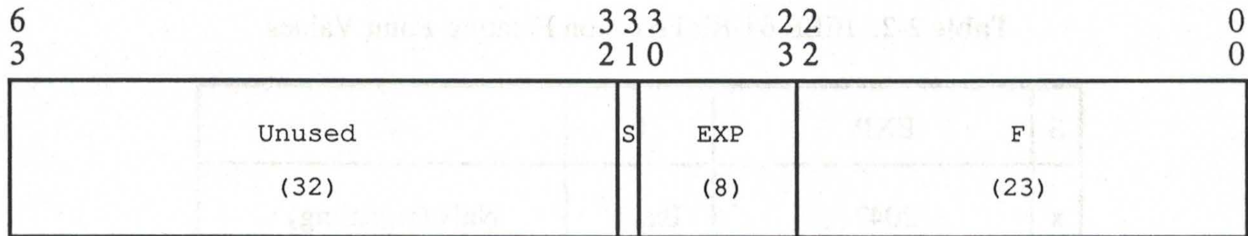


Figure 2-3. IEEE 32-Bit Precision Floating-Point Format

Table 2-1. IEEE 32-Bit Precision Floating-Point Values

S	EXP	F	V
x	255	0x..x†	NaN (signaling)
x	255	1y..y‡	NaN (quiet)
S	255	0	$(-1)^S$ INFINITY
S	0 < EXP < 255	F	$(-1)^S\ 2^{EXP-127}$ (1.F)
S	0	≠ 0	$(-1)^S\ 2^{-126}$ (0.F)
S	0	0	$(-1)^S$ 0

64-bit (double) precision floating-point format numbers have a sign bit **S**, an 11-bit exponent **EXP**, and a 52-bit fraction **F** (Figure 2-4). The value, **V**, represented by this format is computed as in Table 2-2.

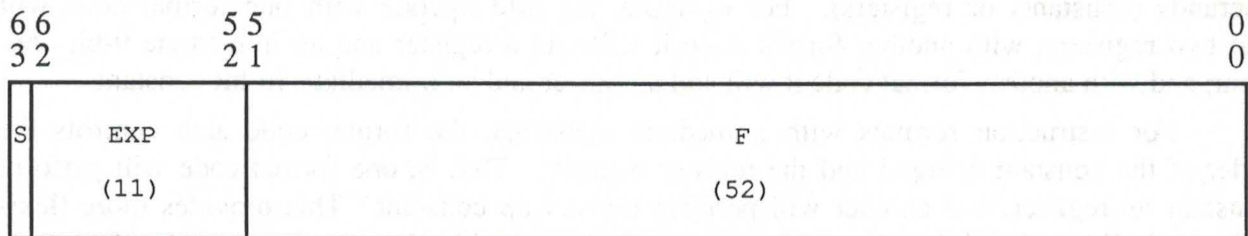


Figure 2-4. IEEE 64-Bit Precision Floating-Point Format

---

† x..x is any non-zero bit pattern on input. Signaling NaNs are never generated by the K-1.

‡ y..y is any bit pattern on input, and is all ones if generated by the K-1.

Table 2-2. IEEE 64-Bit Precision Floating-Point Values

S	EXP	F	V
x	2047	0x..x†	NaN (signaling)
x	2047	1y..y‡	NaN (quiet)
S	2047	0	$(-1)^S$ INFINITY
S	0 < EXP < 2047	F	$(-1)^S\ 2^{EXP-1023}\ (1.F)$
S	0	≠0	$(-1)^S\ 2^{-1022}\ (0.F)$
S	0	0	$(-1)^S\ 0$

Note that due to the sign-magnitude nature of the floating-point formats, there are distinct representations for both positive and negative zero.

Refer to Appendix I for details on floating-point computations and exceptions in the K-1.

2.4 Instruction Formats

The K-1 uses a three-address instruction format: most instructions require three register addresses, two of which specify source operands and the third of which specifies the destination for the result. The way in which the K-1 specifies the use of immediate constants as operands, however, is quite different from other machines.

Whereas most machines use different opcodes to distinguish instructions which allow an immediate constant operand from instructions which only have register operands, the K-1 distinguishes these types of instructions with a **format code**. The format code is a separate field in the instruction from the opcode; it regularizes the instruction set by separating the *function* of an instruction (e.g., addition, loading from memory, etc.), from the *sources* of its operands (constants or registers). For example, the **add** opcode with one format code will add two registers; with another format code it will add a register and an immediate 9-bit constant; and with another format code it will add a register and an immediate 36-bit constant.

For instruction formats with immediate constants, the format code also controls the order of the constant operand and the register operand. That is, one format code will perform constant *op* register, and another will perform register *op* constant. This provides more flexibility and allows the K-1 to have fewer opcodes than would otherwise be needed. (Only one form of asymmetric instructions such as subtract and magnitude comparison is required).

---

† x..x is any non-zero bit pattern on input. Signaling NaNs are never generated by the K-1.

‡ y..y is any bit pattern on input, and is all ones if generated by the K-1.

Instruction formats are fully explained in Chapter 3.

2.5 General Registers

The register file comprises up to 64 registers containing 64 bits each, and are referred to as **r0** to **r63**. Implementations of the K-1 architecture may support less than 64 registers [2-1]. All of the registers are general purpose and may be used to hold the operands or the results of any operation. The architecture treats all registers identically.

2.6 Flags and Conditional Branching

The **Processor Status** register contains seven flags, named **f0** through **f6**, that can each store a binary value; an additional flag, **f7**, always contains the value one. These flags control conditional branching, and allow conditional execution of instructions in most formats. (See Chapter 3 for a description of instruction formats). A number of different instructions, such as the compare instructions, may set or clear a flag; a field in the instruction determines which flag is affected. In addition, boolean operations may be performed on flags and the result may be written to a flag or to a register. During a conditional branch instruction, any flag may determine if the branch should be taken. The flag to be used and its polarity are specified by fields in the branch instruction. There are actually no unconditional branch instructions in the K-1 architecture. "Unconditional" branches are accomplished by specifying flag **f7** as the branch condition. Similarly, in most instruction formats, unconditional execution can only be achieved by conditional execution with respect to **f7**. As will be seen in Chapter 3, conditional branching and conditional execution of instructions are specified in exactly the same fashion.

2.7 ELF Flags

Associated with each general register is an **Early Load Fault** (ELF) flag [2-2]. These flags are set and cleared by **load** and **eload** instructions, and may be interrogated by an **echk** instruction. The **ELF** flags are used to indicate that an **eload** instruction encountered an illegal condition. (See the section on **Early Load** below for a description of the **eload** instruction and its differences from normal **load** instructions). The **relf** and **welf** instructions can be used to save and restore the **ELF** flags.

2.8 Memory

Though most operations in a K-1 program will be of the register-to-register type, there must still be some way to move data in registers into and out of memory. The K-1 load/store instructions serve this purpose. These instructions operate on a number of different sizes of data in memory, and are blind to the data types being moved. For example, a 64-bit integer and a 64-bit floating-point number are treated identically by the load/store instructions.

Main memory can be thought of as a very large array of 8-bit bytes which are numbered starting from zero. The architectural limit to the size of memory is the number of bytes that can be addressed in 48 bits, approximately 281 trillion bytes. Implementations of the K-1 architecture may support smaller address spaces [2-3].

2.8.1 Precision and Alignment

When a load/store instruction references memory, it specifies the precision in which the operation is to be done: either 1, 2, 4, or 8 bytes. The address given must be aligned according to the precision. This means that the address for a 2-byte operation must be evenly divisible by 2; for a 4-byte operation, it must be evenly divisible by 4; and for an 8-byte operation, it must be evenly divisible by 8. A load/store instruction with an address that is not properly aligned will cause a trap (except for **eload** instructions, as explained in the section on **Early Load**, below).

The address given in a load/store instruction normally specifies the most significant byte of the data item. However, there is a bit in the **Processor Status** register called **Byte Order Low-to-High**, which, if set, causes the address to refer to the least significant byte of the data item. In either case, the address of a data item of any precision always refers to byte 0 of the data item. (**Byte Order Low-to-High** is described in more detail in the section on the **Processor Status** register.)

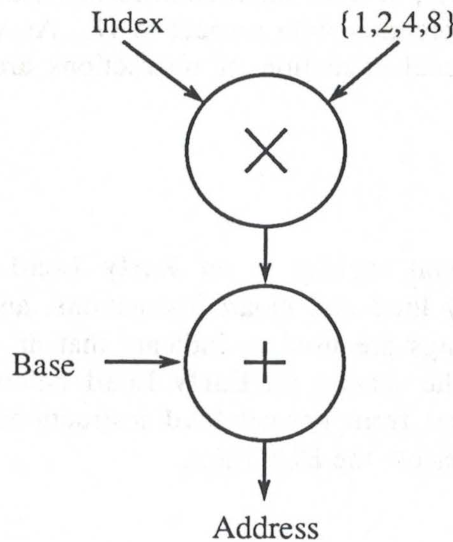


Figure 2-5. Address Calculation for Load/Store Instructions

2.8.2 Addressing Modes

Addresses in load/store instructions are computed in one of two ways. The address may come directly from a single register or constant operand, or it may be calculated as the

sum of two terms: the **base** and the **index**. The index is multiplied by an additional factor, the **memory precision** (Figure 2-5).

The memory precision is either 1, 2, 4, or 8, and is implicitly specified in the opcode of the load/store instruction; it cannot come from a register. The base and index can come from registers or constants depending upon the instruction formats and opcodes used.

2.9 Processor States

The processor can be in one of three modes of operation (states) depending on the settings of some bits in the **Processor Status** register and whether a trap has just occurred. These states are called **user** mode, **supervisor** mode, and **Trap State**. **User** mode has the least privileges; all applications will normally run in **user** mode. **Supervisor** mode is intended for use by the operating system; it provides more capabilities than **user** mode, such as the ability to set more bits in the **Processor Status** register, the ability to write data that is read-only in **user** mode, and the ability to execute privileged instructions.

Whenever a trap is taken, the processor enters **Trap State**, which provides all the privileges of **supervisor** mode, plus the ability to read and modify special internal state information which aids in trap diagnosis and recovery.

Trap State, **user** mode, and **supervisor** mode are described more fully in the sections on **Processor Status** and **Traps, Interrupts, and Machine Checks.**

2.10 Virtual Memory

Data addresses (produced by the address calculations of load/store instructions) and instruction addresses are called **virtual addresses**, and are the only type of address that an application programmer ever uses. The virtual memory system allows multiple users to coexist in a common physical memory by providing hardware support for sharing between users, protection from other users, and efficient execution of modern operating systems such as the UNIX® operating system.†

In order to facilitate porting application programs from an all-32-bit environment, a **Small Address Compatibility** mode is provided. In this mode, all user program and data addresses are assumed to be 32-bit addresses: the high-order bits of any address are set to zero and the low-order 32 bits are retained. This mode is controlled by the **Small Address Compatibility Mode** bit in the **Processor Status** register.

The term *reference* is used throughout this manual to mean a memory operation (generally a load or a store) at a given address. Memory references can be made in either **supervisor** mode or **user** mode. The **mode** of the reference controls the mapping of the virtual addresses used in the processor to the physical addresses used in the memory system. **Supervisor** mode references, used by the operating system, employ a different mapping scheme than **user** mode references.

---

† UNIX is a registered trademark of AT&T.

2.10.1 Address Mapping

The address of an instruction or of a data operand in a load/store instruction is translated to a physical address before being sent to the memory system. This process is known as virtual-to-physical address mapping.

The mapping of a contiguous block of memory, called a **page**, is specified by a single entry in a table, called a **page table** or **page map**. Specifically, a page is a 64K-byte block of addresses starting at an address that is evenly divisible by 64K. This is equivalent to saying that the address of the first byte of a page has 16 low-order zeros in binary. There are separate page tables for instructions and for data.

When a memory data reference is a **supervisor** mode reference (see the section on **Data Mapping**, below), the mapping from virtual to physical addresses is the identity function. In other words, there is no distinction between virtual and physical addresses for **supervisor** mode data references.

As part of the mapping process, a 48-bit virtual address is divided into two pieces. The high-order 32 bits are called the virtual page number and the low-order 16 bits are called the page offset (Figure 2-6) [2-4]. The virtual page number is mapped into a physical page number while the page offset remains the same. The physical page number and the page offset are concatenated to produce the physical address.

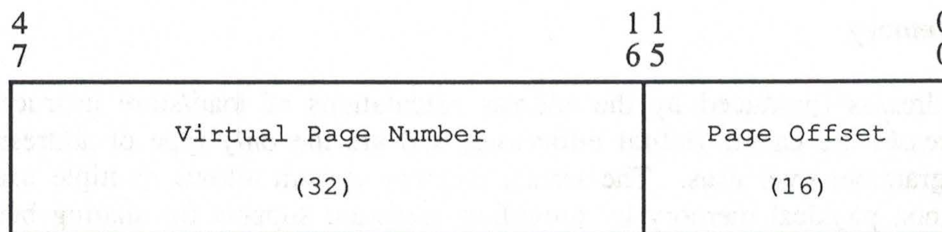


Figure 2-6. Virtual Address Format

2.10.1.1 Instruction Mapping

The instruction page table consists of eight entries, individually loadable with the privileged instruction **lipage**, that specify the mapping of eight different virtual regions for **user** mode instruction references. In **supervisor** mode, instruction mapping is done in an implementation-dependent fashion [2-5]. In **user** mode, instruction mapping is done using the instruction page table. Figure 2-7 shows the format of an instruction page table entry (the operand to a **lipage** instruction) [2-6].

The **N** field, sometimes called the **Instruction Page ID**, specifies which of the eight instruction page table entries is to be affected. If the valid bit, **V**, is a zero, then page table entry **N** is invalid and the rest of the fields are ignored. If the valid bit is a one, then page table entry **N** maps a virtual page range into a physical page range. The virtual and physical page ranges begin with the pages specified by the **Virtual Page Number** and **Physical Page Number** fields, respectively [2-7]. These page ranges must always be aligned on a bound-

ary which is a multiple of their size. Note that an instruction virtual page number is architecturally limited to 25 bits, and therefore instruction addresses are architecturally limited to 41 bits.

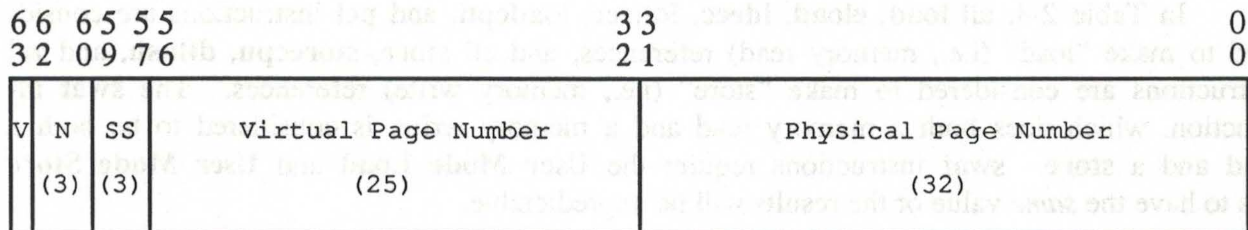


Figure 2-7. Instruction Page Table Format

The extent of the page ranges is determined by the size specifier, **SS,** field. It indicates how many contiguous pages are referenced by this instruction map entry. The interpretation of this field is implementation-dependent (Table 2-3) [2-8].

Note that it does not matter which page table entry (which value of **N**) is used for a particular mapping entry, but mapping a given virtual address with more than one page table entry will produce unpredictable results. Therefore, the virtual page ranges specified by the valid page table entries must be non-overlapping.

Table 2-3. Instruction Page Table Size Specifier

SS	# of Pages Mapped
0-7	See Implementation Dependencies, Appendix H

2.10.1.2 Data Mapping

It is intended that the operating system and applications program(s) occupy separate address spaces. Whether a reference is treated as a **supervisor** mode reference or a **user** mode reference is a function of the **User Mode Load**, **User Mode Store**, and **User Protection** bits of the **Processor Status** register, as well as the type of reference (Table 2-4). When the processor is in **user** mode, it can only make **user** mode references to memory. But when the processor is in **supervisor** mode (and not in **Trap State**), it can make *either* **user** mode references *or* **supervisor** mode references depending on the settings of the **User Mode Load** and **User Mode Store** bits and the type of reference. This allows the operating system to perform memory references with the user's address mapping. For example, if the operating system wanted to copy data into the user's area at a virtual address provided by the user (such as in response to an I/O request), it could set the **User Mode Store** bit; **load**

instructions would then use the operating system's address mapping, but **store** instructions would use the user's address mapping and protection. Using the user's protection prevents malicious users from providing invalid addresses.

In Table 2-4, all **load, eload, ldecc, ldnecc, loadcpu,** and **pcl** instructions are considered to make "load" (i.e., memory read) references, and all **store, storecpu, dflush,** and **zcl** instructions are considered to make "store" (i.e., memory write) references. The **swat** instruction, which does both a memory read and a memory write, is considered to be both a **load** and a **store**. **swat** instructions require the **User Mode Load** and **User Mode Store** bits to have the *same* value or the results will be unpredictable.

Table 2-4. Memory Reference Modes

User Mode Load	User Mode Store	Load Reference Mode	Store Reference Mode
Off	Off	Supervisor	Supervisor
Off	On	Supervisor	User
On	Off	User	Supervisor
On	On	User	User

User mode references always use the data page table to perform the mapping from virtual to physical addresses and to determine the legal access modes (read or write) and sharability of that page. **Supervisor** mode references do not use the data page table; both read and write **supervisor** mode references are always permitted. **Supervisor** mode references are always shared. Only shared pages participate in the multiprocessor cache-coherence scheme. Thus, all **supervisor** mode references and any user mode references which are designated as shared in the data page table are cache-coherent.

The page control bits of a data page table entry (described below) control the types of access (read and write) that are allowed for **user** mode references to that page. In addition, they control whether the page can be shared among processes. For **user** mode references when the **User Protection** bit in the **Processor Status** register is *off*, the page control bits are ignored: *both* read and write references are permitted if *either* type of reference is permitted. This allows the supervisor to modify user data without the user's access restrictions.

Data page table information is stored differently from instruction page table entries. The data page table is a cache, while the instruction page table is a fully-associative memory [2-9].

Each page table entry contains the **Process Key**, a virtual tag, the physical page number, and a number of page control bits. The data page table is addressed by hashing the virtual page number being mapped and the **Process Key** field of the **Processor Status** register [2-10]. Therefore, the same virtual page numbers for two different processes (which, of course, must have different **Process Keys**) are probably mapped by different entries in the

page table. It is not necessary to evict one entire process to load the page table for another because the **Process Key** values stored with the page table entries are used to distinguish the entries for different processes.

A **data map miss** trap occurs on a **user** mode reference when the virtual address calculated by a load/store instruction does not have a valid entry with the proper **Process Key** and address tags in the data page table.

The **ldpage** instruction is used to load entries into the data page table. Figures 2-8 and 2-9 show the formats of the operands to a **ldpage** instruction [2-11]. A table entry is written to map the given **Virtual Page Number** to the given **Physical Page Number**. The **Virtual Page Number** and the **Process Key** field of the **Processor Status** register are hashed to produce the data page table address to be written.

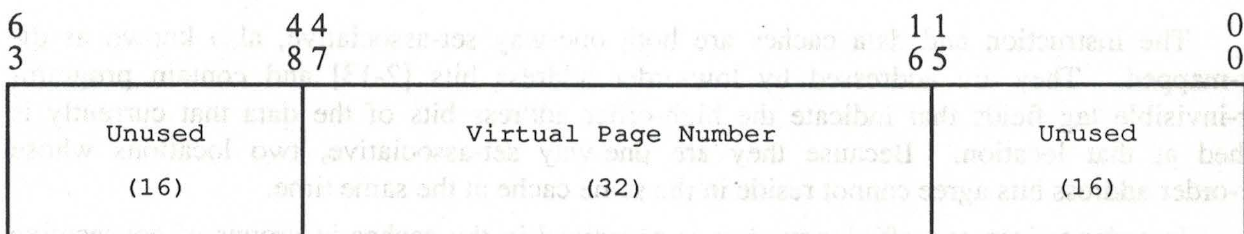


Figure 2-8. Virtual Page Number

The **R** and **W** page control bits define the types of **user** mode references permitted to the page being mapped (unless overridden by clearing the **User Protection** bit of the **Processor Status** register; see below). If **R** is set, then **load**, **eload**, and **pcl** instructions are permitted. If **W** is set, then **store**, **dflush**, and **zcl** instructions are permitted. **swat** instructions, which do both a read and a write, require both **R** and **W** to be set. If neither bit is set, the page is **invalid** and *no* reference to that page is permitted in **user** mode *regardless* of the setting of the **User Protection** bit. Note that referencing an invalid page will cause an illegal access trap, not a data map miss trap.

The **S** bit indicates that the page is **shared.** The unused fields in the data page table format are reserved for future expansion and must be zero. An illegal access trap occurs when the user virtual address calculated by a load/store instruction does not have the proper page control bits for the requested reference.

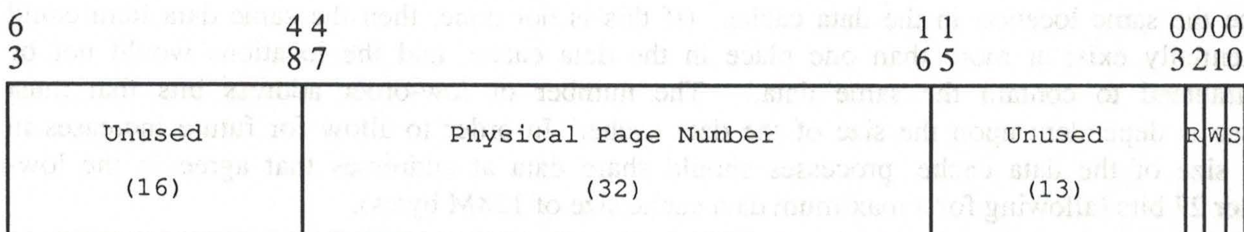


Figure 2-9. Data Page Table Format

2.11 Cache Effects

Each K-1 processor contains two independent cache systems, one for instructions and the other for data. The caches are hardware managed, but without regard for whether a single data item exists in both caches at the same time. Therefore, when writing applications that depend on the execution of data as programs, that write into the instruction stream, or that require that main memory contain an up-to-date copy of the data (such as I/O processing), the programmer is required to use instructions that manipulate the caches. These instructions can be used to remove data from either cache and/or force modified cache data to be written back to main memory.

2.11.1 Cache Organization and Addressing

The instruction and data caches are both one-way set-associative, also known as direct-mapped. They are addressed by low-order address bits [2-13] and contain programmer-invisible tag fields that indicate the high-order address bits of the data that currently is cached at that location. Because they are one-way set-associative, two locations whose low-order address bits agree cannot reside in the same cache at the same time.

In order to improve efficiency, data is organized in the caches in groups of consecutive addresses called **cache lines** (or sometimes just **lines**). Lines are the minimal tagged unit in the cache. Transfers between the memory system and a CPU are always of entire cache lines, but are broken up into smaller pieces called **memory transfer sub-lines** (or just **sub-lines**). A sub-line can be thought of as the largest unit that can be read from or written to the cache at one time (or as the width of the memory bus). For example, writing a cache line into the cache involves a series of writes of individual sub-lines. The size of cache lines and sub-lines are implementation-dependent, but are always powers of 2 [2-12]. Cache lines and sub-lines are always aligned on a boundary that is a multiple of their size.

The instruction cache is addressed by physical addresses, while the data cache uses a combination of virtual and physical addresses. That is, the instruction cache is addressed and tagged by the physical address, whereas the data cache is addressed by the low-order bits of the virtual address, but is tagged with the physical address. This means that lines of data at the same *virtual* address in two different processes will compete for the same place in the data cache.

An additional implication of this data cache organization is that shared data (data that is referenced by more than one process) should be addressed by all sharing processes with addresses that agree in sufficient low-order virtual address bits to ensure that they reference the same location in the data cache. (If this is not done, then the same data item could potentially exist at more than one place in the data cache, and the locations would not be guaranteed to contain the same data.) The number of low-order address bits that must agree is dependent upon the size of the data cache. In order to allow for future increases in the size of the data cache, processes should share data at addresses that agree in the low-order 27 bits (allowing for a maximum data cache size of 128M bytes).

It is also possible for processes to share data at arbitrary virtual addresses. However, this requires that the operating system "sweep" the cache after running such a process.

It also requires that two such sharing processes never be run simultaneously on two different processors in a multiprocessor system. Naturally, this will result in some performance penalty.

The sizes of the instruction and data caches are implementation-dependent [2-13].

2.11.2 Instruction Stache

The K-1 provides an additional level of instruction caching called the instruction **stache**. This much smaller and faster cache lies between each CPU and its instruction cache. The instruction stache is addressed by low-order instruction address bits, but instead of recording the entire physical page number, it remembers only the **Instruction Page ID** (the **N** field in an instruction page table entry). This means that the entire instruction stache must be invalidated whenever the instruction page table is changed. The entire instruction stache may be invalidated by executing an **iskill** instruction, or by entering or exiting **Trap State** (i.e., by trapping or by returning from a trap).

The line size of the instruction stache is not necessarily the same as that of the instruction cache. The total size and the line size of the instruction stache are both implementation-dependent [2-14].

2.11.3 Cache Coherence

The K-1 implements a multiprocessor **cache-coherence** scheme that allows CPUs in the same system to share data without software knowing about the effects of caches. That is, if one processor (A) modifies some data, cache coherence guarantees that the next processor (B) to read that data will see the modified value (even if the old value was in B's data cache prior to A's write). It is still necessary for software running on different CPUs to synchronize the modification of shared data. Otherwise, two CPUs might modify the same data at nearly the same time. Since there is no guarantee (without synchronization) which CPU would modify the data first, the result would be unpredictable.

The K-1 cache-coherence scheme is handled through the memory system. It supports a single-writer, multiple-reader model. That is, any number of processors can have a read-only copy of a particular piece of shared data in their data caches at the same time. But as soon as one processor tries to modify shared data, it must be granted sole ownership. When another processor tries to read the modified result, the writer will lose its sole ownership (and its write privileges).

The K-1 distinguishes between shared data and non-shared data; only shared data participates in the cache-coherence scheme. As explained in the section on **Data Mapping** above, all **supervisor** mode references are shared; **user** mode references are shared only if they are marked as shared in the data page table.

The instruction caches do not participate in the cache-coherence scheme. Cache coherence between instructions and data must be provided by software, even between the instruction cache and the data cache of a single processor.

The I/O system does not participate in the cache-coherence scheme and because of this, care must be taken when writing low-level code. Before initiating an I/O write, the data being written must be flushed from the data caches of all processors. Before an I/O read completes, the old "stale" data in all the processors' caches must be flushed (so that the new data can be read from memory). A special *shared* version of the **dflush** instruction is provided for this purpose. This shared **dflush** is "broadcast" to all the CPUs in a multiprocessor system, and can flush a given cache line from all processors' data caches at once. (See the **dflush** instruction description for more details.)

2.11.4 Interprocessor Synchronization

The K-1 has two *semaphore* schemes to allow different processors to synchronize their activities. First, the **swat** instruction is an atomic operation on shared memory. Second, the I/O system implements a limited number of registers whose bits can be atomically set or cleared with the **wios** instruction. Both of these methods can be used to implement binary semaphores, which can be used for interprocessor synchronization.

The two independent synchronization schemes use different system resources. While the **swat** instruction allows the implementation of a huge number of semaphores (limited only by the size of memory), its use of the memory system (which is optimized to transfer large amounts of data) can make it slow. **swat** instructions must also "compete" for memory bandwidth with other operations. The I/O system implements a smaller number of semaphores, but they are faster to access and their use does not consume memory bandwidth. The semaphores used most frequently by the operating system should be implemented in the I/O system. See Appendix F for more details on the I/O system.

2.12 Program Execution

K-1 instructions are either 32 or 64 bits long and reside in main memory at addresses that must be properly aligned; a 32-bit instruction must start at an address evenly divisible by 4 and a 64-bit instruction must start at an address evenly divisible by 8. Program execution involves the repeated fetching of 64-bit instruction words, and the **issue** of instructions from them. (Instruction words start at an address that is evenly divisible by 8.)

Each instruction word contains either two 32-bit instructions or one 64-bit instruction. If there are two 32-bit instructions packed into an instruction word, they are designated as **I0** and **I1**, with I0 occupying the most significant half (lower address) and I1 the least significant (higher address). If there is one 64-bit instruction in an instruction word, it is designated as I0, and there is no I1 instruction.

The byte address of the current instruction being fetched is contained in the **Program Counter (PC)**, which is incremented by 4 or 8 depending upon the size of the instruction being executed. Programs are executed sequentially until a trap condition or a branch instruction is encountered. Branch instructions select a new program counter either as an absolute address or as a signed offset from the address of the instruction word containing the branch instruction. If a new program counter points to an I1 instruction (i.e., its bit 2 is set) then the

entire 64-bit instruction word will be fetched, but only the I1 instruction in that word will be executed.

There are two **disable** bits, **D0** and **D1**, associated with each **PC**. The D0 bit inhibits the execution of the I0 instruction, and the D1 bit inhibits the execution of the I1 instruction. Normally, when an instruction word is first fetched, both the D0 and D1 bits will be clear. The D0 bit is set after the I0 instruction has executed, and the D1 bit is set after the I1 instruction has executed. For 64-bit instructions, both bits are set when the instruction executes. Note that a branch to an I1 instruction actually branches to the I0 instruction (since the K-1 always fetches 64-bit instruction words), but sets the D0 bit, disabling the I0 half of the instruction word. The **D0** disable bit can thus be seen to be the same as bit 2 of the program counter. The disable bits are used primarily in the features of the architecture relating to trapping. When an instruction word is fetched from a **PC** with both disable bits set (as can happen due to delayed branching or **exts** instructions), instruction map miss traps are suppressed.

2.12.1 Delayed Branching

Branch instructions have the ability to disable the execution of instructions from the subsequently-fetched one or two 64-bit words. This is done by specifying the conditions under which the two instruction words following a branch will be executed: in case of branch, in case of fall-through, or always (in case of branch or fall-through). The execution of a few more instructions after a branch can be thought of as delaying the actual branching (as opposed to delaying the branch decision). Delayed branching allows compilers to compensate for the execution time penalty incurred by branches in implementations of the architecture. Instructions whose execution is controlled by a branch are called **delay instructions**. The first 64-bit word fetched following a branch instruction is called the **first delay slot,** and the 64-bit word fetched following that is called the **second delay slot**.

Branch instructions control whether or not instructions that occupy the first and second delay slots are to be executed. For each delay slot there are three cases: execute the instructions in that slot if the branch is taken (branch), not taken (fall-through), or regardless of whether the branch is taken (always). The letters **b**, **f**, and **a** are used to refer to these cases, respectively. Of the nine possible independently controllable cases for *two* delay slots, eight may be specified by the delayed execution control field (**DC**) of a branch instruction (Table 3-2). The ninth case, **ba**, indicates that the instructions in the first slot are executed only on the branch path, while those from the second slot are executed always (on both paths). This is thought to be the least useful case, and is practically equivalent to the **ab** case, which is provided.

There are no restrictions on the types of instructions that can be executed as delay instructions. In particular, another branch instruction may be executed as a delay instruction. In order to clearly describe the sequence of instructions that will be executed in such a case, it is necessary to use a slightly unusual model of how program branching works. **The key rule to remember is that the execution of each 64-bit instruction word determines the address of the *third following* 64-bit instruction word to be fetched.** It is also important to know that all delay instructions are fetched, even if they are not going to be executed. Such a

nullified delay instruction word is effectively disabled by asserting both the D0 and D1 disable bits associated with that instruction's **PC**.

When the first in a sequence of branch instructions is encountered, the addresses of the next two instruction words have already been determined. If the branch is taken, the address of the third following instruction word fetch will be the branch target address. However, it is the execution of the first delay instruction that controls the fetch address to be used immediately after the target of the branch (the fourth following instruction word fetch).

As an example consider the following program:

```
A:      jump  XYZ, aa
B:      add   %r0,%r1,%r2;    br    QQQ, bb
C:      nop
        ...
XYZ:    move  %r1,%r2
        ...
QQQ:    ...
```

The sequence of instruction fetches will be from addresses A, B, C, XYZ, QQQ, and will then continue with QQQ+8, QQQ+16, etc. Note that the delay instructions of a branch are not necessarily located in the addresses sequentially following the branch. In the above example, the first delay slot for the branch to QQQ is at C, but the second delay slot for that branch is at XYZ.

2.12.2 Conditional Execution

Most of the K-1 instruction formats allow the specification of a conditional execution field: instructions can be conditionally executed based on the value of a flag. Since flag **f7** is always true, instructions can always be unconditionally executed. The benefit of conditional execution is that in many cases branches can be avoided by conditionally executing the code that would have been jumped around. For a small number of conditionally-executed instructions, this is more efficient than branching; the exact break-even point depends upon the specific instruction timings and interlocks. (See Appendix C for more details on these topics.) Conditionally-disabled instructions will never trap, even in cases such as illegal instruction/privilege violation traps. Note that traps that occur before an instruction is interpreted, such as instruction page map misses and trace traps, will still affect conditionally-disabled instructions.

2.13 Early Load

In order for compilers to produce highly efficient code despite the latency of **load** instructions, they must be able to move the **load** instructions "backwards" in the code, i.e., well before the use of the **load**'s result. However, a **load** may have the serious side effect of trapping. If a **load** instruction was moved before a check for an illegal address, for example,

the **load** result would probably have been ignored after the check and so an illegal address trap would have been superfluous and should have been suppressed.

The **early load** mechanism is designed to solve this problem. For each **load** instruction there is a corresponding **eload** instruction. The **eload** instruction performs the same address calculation and mapping function as the corresponding **load**, and, if there is no trap, loads the proper value into the destination register. **eloads**, however, behave differently if there is an error. For illegal access errors, illegal address errors (when the **Early Load Alignment Trap** bit in the **Processor Status** register is not set), and **supervisor** mode non-existent memory errors, the **eload** will *not* trap but will instead store zero into the destination register and set the **ELF** flag corresponding to that register. For all other types of errors, **eload** instructions behave the same as the corresponding **load**. In this case, it is then up to the operating system to either abort the program or set the result and **ELF** flag accordingly and resume its execution.

The **ELF** flag is cleared by any **load** that does not trap, and set by any **load** that does trap. **eload** instructions set the **ELF** flag identically to the corresponding **load**, whether the **eload** traps or not. A **load** that might have an unwanted side effect should be replaced with an **eload**.

In order to verify that the results of an **eload** are indeed valid, the programmer may use the **echk** instruction. This instruction tests one **ELF** flag and traps if it is set. It should be noted that **echk** instructions may be omitted for efficiency; correct programs will always work either with or without **echk** instructions, although the behavior of incorrect programs can be radically different.

In **Trap State** (see below), **load** instructions do not affect the **ELF** flags and **eload** instructions act the same as **loads**. The **relf** and **welf** instructions can be used either in or out of **Trap State** to save and restore the **ELF** flags.

2.14 Processor Status

Much of the K-1 processor's operation is controlled by a register called the **Processor Status** register. The **rps, wps, srm,** and **spl** instructions read and write the **Processor Status** register. In **user** mode, only the high-order 32 bits of the **Processor Status** register can be modified. The **Processor Status** register is overridden while in **Trap State** and a default value for some of the bits is used. The format of the **Processor Status** register is given in Table 2-5. Unused bits of the **Processor Status** register read as zero, but application programs must not rely on this fact since this may not be the case in other versions of the K-1 architecture. Attempts to set (write a '1' into) unused bits of the **Processor Status** register cause a trap [2-16].

The **Processor Version Number<7..0>** field is a read-only field giving the implementation level for this model of the K-1 processor. It may be used by operating systems to tailor code for specific implementations.

The **f<6..0>** field refers to the six flag bits **f6** through **f0** (where **f<6>** refers to **f6**, etc.). These bits are primarily set and cleared by compare and flag instructions and tested by conditional branch instructions. They can also be used to control conditional execution in

most instruction formats. Flag **f7** is always set and is therefore not part of the **Processor Status** register.

Table 2-5. Processor Status Register

Bits	Function	Act as if 0 in Trap State
63..56	Processor Version Number<7..0>	No
52..46	f<6..0>	No
45..41	Arithmetic Exception Flags<4..0>	No
40..36	Arithmetic Trap Enables<4..0>	Yes
35	Integer Divide Trap Enable	Yes
34..33	Rounding Mode<1..0>	No
32	Byte Order Low-to-High	Yes
28..16	Process Key<12..0>	No
15	Trace Enable	Yes
14	Trace Pending	Yes
13	User Protection	Yes
12	User Mode Store	Yes
11	User Mode Load	Yes
10	Small Address Compatibility Mode	Yes
9	Early Load Alignment Trap	Yes
3..0	Processor Priority Level<3..0>	Yes

Table 2-6. Arithmetic Trap Enables and Exception Flags

Bit	Trap/Exception Type
0	Invalid Operation
1	Division by Zero
2	Floating-Point Overflow
3	Floating-Point Underflow
4	Inexact Result

If certain exception conditions are detected during floating-point calculations, then one or more of the **Arithmetic Exception Flags** will be set, and an arithmetic trap may occur depending upon the **Arithmetic Trap Enables<4..0>** field. These five bits independently enable floating-point exceptions to trap as shown in Table 2-6. Setting **Arithmetic Exception Flags** or **Arithmetic Trap Enables** with the **wps** (write processor status) instruction will not cause a trap. The **Arithmetic Exception Flags** can be cleared only with the **wps** instruc-

tion. Note that floating-point instructions issued in Trap State can affect the **Arithmetic Exception Flags**. If preservation of the state of the trapping program is desired, then care must be taken in Trap State to save the **Processor Status** register before issuing any floating-point instructions.

Although the K-1 supports floating-point traps, they are not IEEE compatible. It should be noted, however, that the IEEE Floating-Point standard does not require any support for traps.

The **Integer Divide Trap Enable** bit controls the ability of integer divide instructions (**divsst** and **divssr**) to generate traps when division by zero is attempted. If this bit is not set, then integer divides will not trap on division by zero and will just return 0.

During floating-point calculations, it is sometimes necessary to store an inexact result. The process by which the exact answer is transformed into an inexact result is called *rounding*. Four different methods of rounding are available as controlled by the **Rounding Mode<1..0>** field and described in Table 2-7. The **srm** instruction is provided to change the **Rounding Mode** field of the **Processor Status** register.

Table 2-7. Rounding Modes

Field Value	Rounding Mode	Name
0	Round to Nearest	Nearest
1	Round Towards Zero	Truncate
2	Round Towards Positive Infinity	Ceiling
3	Round Towards Negative Infinity	Floor

The **Byte Order Low-to-High** bit controls the low-order address bits generated during load/store instructions. If this bit is set, then the low three address bits (2..0) are transformed based on the precision of the load/store instruction (Table 2-8). **Supervisor** mode references are not affected by the setting of the **Byte Order Low-to-High** bit.

The **Process Key<12..0>** field distinguishes the virtual page numbers of the currently running process from the identically-numbered pages of another process in the data page map.

The **Trace Enable** and **Trace Pending** bits control trace trapping. **Trace Enable** enables the setting of **Trace Pending** *after* the execution of the current instruction. **Trace Pending** actually causes a trace trap. These bits can be used to single step a program, or to advance a program past a breakpoint.

The **Trace Enable** bit can only be cleared or set by a **wps** instruction when the processor is in **Trap State.** The **Trace Pending** bit can only be cleared by a **wps** instruction when the processor is in **Trap State**, and can be set by a **wps** instruction in **Trap State**. But most importantly, the **Trace Pending** bit is set whenever a non-**Trap State** instruction is executed while the **Trace Enable** bit is set. A trace trap occurs when **Trace Pending** is

set, the processor is not in **Trap State**, and the instruction about to be issued has not been disabled by a branch or by **exts**. (**Trace Pending** *will* cause a trace trap on an instruction which is disabled by conditional execution.)

Table 2-8. Low-to-High Byte Addressing

Precision	Input <2..0>	Output <2..0>
1-byte	0 1 2 3 4 5 6 7	7 6 5 4 3 2 1 0
2-byte	0 2 4 6	6 4 2 0
4-byte	0 4	4 0
8-byte	0	0

The **User Mode Load** and **User Mode Store** bits control whether memory references are treated as **user** mode references or supervisor mode references. This topic is discussed in the section on **Data Mapping,** and is illustrated in Table 2-4. Note that the **Byte Order Low-to-High** and **Small Address Compatibility Mode** bits affect only **user** mode references – both bits are disabled in **supervisor** mode. Since the **User Mode Load** and **User Mode Store** bits always act as zero in **Trap State**, there can only be **supervisor** mode references in **Trap State.**

The **User Protection** bit controls whether **user** mode references use the protections provided in the data page table, or whether they get supervisor "over-ride" privileges. This topic is discussed in the section on **Data Mapping**.

The processor is considered to be in **user** mode only if all three of the **User Mode Load, User Mode Store**, and **User Protection** bits are on. If any of these bits is off, the processor is in **supervisor** mode. Any attempt to switch from **supervisor** mode to **user** mode *other* than when in **Trap State** will produce unpredictable results. Note that even when the processor is in **supervisor** mode (and not in **Trap State**), it can still make **user** mode references to memory by controlling which of the **User Mode Load, User Mode**

Store, and **User Protection** bits are set. Instruction references are in **user** mode only if the processor is in **user** mode (all three of the **User Mode Load**, **User Mode Store**, and **User Protection** bits are on).

The **Small Address Compatibility Mode** bit forces both **user** mode instruction and user mode data virtual addresses to be truncated to 32 bits. The higher-order bits of any user address will be set to zero. This mode is provided to ease the burden of porting programs from a 32-bit environment. **Supervisor** mode references are not affected.

The **Early Load Alignment Trap** bit enables **eload** instructions to trap on misaligned address errors (instead of just setting an **ELF** flag). The normal operation, if this bit is clear, is for **eloads** to "ignore" the alignment error (setting the **ELF** flag and returning 0). If this bit is set, however, an **eload** with a misaligned address will take an illegal address trap (as would the corresponding **load** instruction).

The **Processor Priority Level<3..0>** field gives the interrupt priority level of the processor. All interrupts have an associated priority level. If the priority level of an interrupt is *higher* than the priority level of the processor, then the interrupt will happen. If the interrupt priority is the same or lower than the priority level of the processor, then the interrupt will remain pending. The interrupt will occur as soon as the processor lowers its priority to be less than that of the interrupt. The **Processor Priority Level** field is encoded with zero as the highest priority level (masking out all interrupts), and fifteen as the lowest level (allowing all interrupts). The **spl** instruction is provided to change the **Processor Priority Level** field of the **Processor Status** register.

2.15 Input/Output

I/O controller(s) perform input/output via direct memory access over a number of high-speed I/O busses. The sequencing of I/O operations is controlled by messages in memory and by interrupts. The **rios** and **wios** instructions are provided to exchange control and status information between a CPU and the I/O subsystem.

2.15.1 I/O Interrupts

Interrupts from the I/O system are assigned priority levels from zero to fifteen, with zero being the highest priority. The level of the highest-priority, pending I/O interrupt is compared with the **Processor Priority Level<3..0>** field of the **Processor Status** register and, if the I/O interrupt has higher priority (i.e., is numerically less than the processor's priority), the processor will perform an I/O interrupt sequence. The **Trap Summary** contains the level of the highest-priority pending I/O interrupt. All I/O interrupts are disabled in **Trap State**. (See the section on **Traps, Interrupts, and Machine Checks**, below.)

2.16 Timers

The K-1 has an **Uptime Counter** and an **Interval Timer** register, whose formats are shown in Figures 2-10 and 2-11. The Uptime Counter continuously counts the number of cy-

cles since the processor was last reset. (See Appendix C for a description of processor cycles.) The Interval Timer register contains an **interval** field, that is constantly being compared with the low-order 32 bits of the Uptime Counter. If there is a match and the clock interrupt enable (**CIE**) bit is set and the processor is not in **Trap State**, then an interval timer interrupt will be generated; this interrupt has priority level 0. If the processor is in **Trap State**, the interrupt is held pending until the processor exits **Trap State**. Generation of an interrupt effectively clears the **CIE** bit; the Interval Timer register must be loaded again if another interrupt is desired.

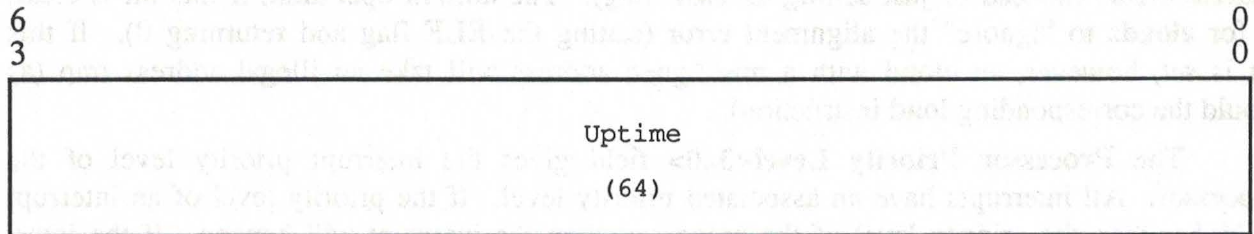


Figure 2-10. Uptime Counter

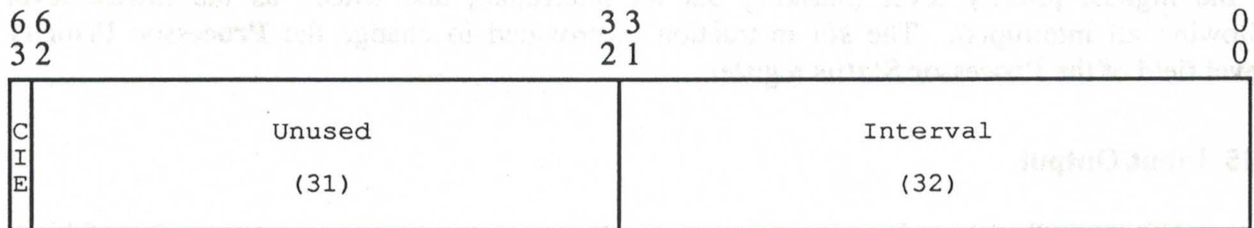


Figure 2-11. Interval Timer Register

The **rut** instruction can be used to read the Uptime Counter, and the **wit** instruction can be used to write the Interval Timer register.

The size of the fields in the Uptime Counter and the Interval Timer register are implementation-dependent [2-15].

2.17 Traps, Interrupts, and Machine Checks

A variety of exceptional conditions that prevent further progress may arise during the execution of a program and cause a **trap** to occur. During a trap, certain processor state information is saved and execution is transferred to a different address with (possibly) different processor status. Some of the exceptional conditions that can cause traps are data map misses, I/O, Interval Timer, and Console interrupts, machine checks and the execution of certain instructions whose purpose is to cause a trap. When a trap occurs, the processor enters **Trap State**.

With the exception of load/store unit instructions, all instructions that write results will still do so even if they trap. The results written, however, may be different depending on

whether there was a trap or not. See Chapter 5 for a description of the effects of traps on load/store unit instructions, and Appendix I for a description of the results returned by floating-point instructions in the presence or absence of traps.

The K-1 architecture does not guarantee that instructions will be completed in the order in which they are encountered in the program. The order of operations may be changed if it does not affect the results of the calculations. While this is normally completely invisible to the programmer, it can become visible when a trap occurs.

While execution can be continued after a trap, the imprecise nature of most traps means that results can not always be repaired before returning from the trap. In more detail, traps occurring before instruction issue (instruction map miss, illegal instruction/privilege violations, trace traps, interrupts, trap instructions such as **bpt**, **trap**, etc.) are **precise**, and traps occurring after instruction issue (floating-point exceptions, data map miss, memory errors, etc.) are **imprecise**. A precise trap implies that execution has not proceeded past the instruction that trapped. With imprecise traps, some number of instructions after the trapping instruction may have been issued and completed. For example, if a floating-point multiply instruction traps, several instructions after the multiply may have issued and completed before the multiply trap is detected (suspending further instruction issue). The instructions following the multiply may have altered the input operands of the multiply, making recovery from the trap difficult, if not impossible. Furthermore, instructions that use the result of the trapping instruction could have been issued before the trap suspended instruction issue.

Imprecise traps are a consequence of a highly pipelined machine with moderate functional unit latencies. Precise traps can be guaranteed in software by not modifying the operands of an instruction and by not using its result until after any trap it could have issued has taken affect. (The special long constant form of the **nop** instruction is useful in this context – it can be used to wait for various events such as the completion of non-fixed-latency instructions). See Appendix C for a discussion of functional unit latencies.

When a K-1 CPU decides to trap, all new instruction execution is suspended. A summary of all the "simultaneously" occurring traps is accumulated, as well as specific information about each one. This **Trap Summary** is explained in Appendix D.

Regardless of the cause of a trap, certain restart information, referred to as the **Restart PCs**, is also accumulated. The **Restart PCs** (Figure 2-12) are the first three instruction addresses to fetch to resume execution of the program that trapped, and contain execution disable bits to possibly void the execution of instructions fetched at those addresses (due to the effects of prior delayed branches and previously issued instructions). For both precise and imprecise traps, the **Restart PCs** allow the program to be resumed from the point at which execution was suspended. The exact location of the restart point depends upon the cause of the trap. (See the description of decode traps, below and in Appendix D.) Three **Restart PCs** are necessary because of the K-1 architecture's two-cycle delayed branching.

In a **Restart PC**, the **Restart IWA** (Instruction Word Address) field contains bits 40..3 of the address of an instruction word [2-3]. **D0** and **D1** are the disable bits associated with the instructions at address **Restart IWA*8**. If a disable bit is set for a particular **Restart PC**, the corresponding I0 or I1 instruction will be disabled when it is fetched. It is pos-

sible that both **D0** and **D1** will be set, indicating that no instructions from that word are to be executed upon restart.

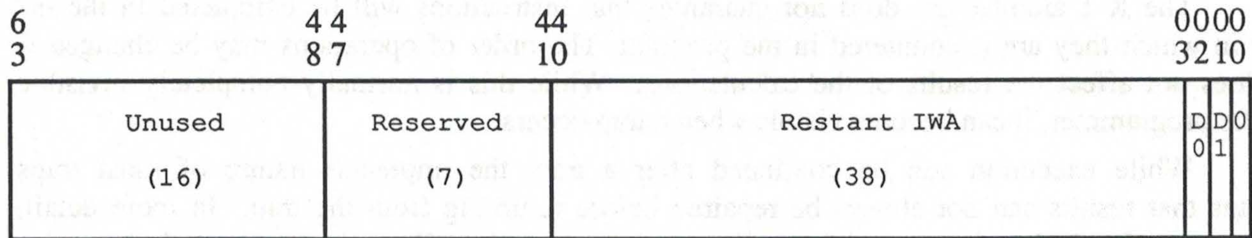


Figure 2-12. Restart PC

Traps from load/store instructions, although imprecise, are treated specially. In order to support paging, traps from load/store instructions must be recoverable. A further complication is that load/store instructions are also executed serially, and at the time of a trap, there may be quite a few load/store instructions already in the pipeline. To recover from an imprecise load/store instruction trap, it is necessary to simulate in software *all* of the load/store instructions that entered the pipeline before the trap suspended further instruction issue. All of the information needed to simulate these instructions is contained in a memory called the **load/store queue**. A copy of the contents of the load/store queue is frozen upon entering **Trap State**, and can be stored to memory with the **slstrpd** (store load/store trap data) instruction. (See Appendix D for more information on recovering from load/store traps.)

2.17.1 The Trap Sequence

At the start of a trap sequence, the K-1 saves certain trap recovery information, which can be read with the **rtrpd** and **slstrpd** instructions, in internal memories in the CPU. This trap data includes the **Trap Summary**, a set of **Trap Locators** for each pipeline stage of each functional unit, the **load/store queue**, and the three **Restart PC**s. Next, the processor enters a special state called **Trap State**. In **Trap State**, a number of fields in the **Processor Status** register act as if they are zero, except that a **rps** instruction reads out the correct processor status. The affected fields are indicated in the last column of Table 2-5. In **Trap State**, as in **supervisor** mode, instruction mapping is done in an implementation dependent fashion [2-5]. Any trap while in **Trap State** will cause the system to halt and the Console to be notified.

There are five types of CPU-internal traps that can occur in any combination: decode traps, instruction fetch traps, integer overflow or check traps, floating-point traps, and load/store traps. In addition, a special CPU-internal trap, called a **reset** trap, occurs at system initialization time. (See the section on **Reset Operation**.)

There are five types of external interrupts that are recognized by the K-1: **NMI** is a non-maskable interrupt that indicates that an over-temperature condition or other Console panic has been detected; **CKI** indicates an interval timer interrupt; **RPI** and **WPI** are interrupts that indicate that the Console interface's read port and write port are full and empty, respectively; and **IOI** indicates an I/O system interrupt. Table 2-9 lists the priority level corresponding to each external interrupt.

Lastly, machine checks are error conditions that, in a correctly operating machine, should never occur. Unlike traps or external interrupts, they are more indicative of component, connector, or design failures, possibly of an intermittent or one-time nature. A number of units within the K-1 CPU cooperate to detect such conditions and facilitate graceful (if possible) recovery.

Table 2-9. External Interrupt Priority Levels

Interrupt Type	Priority Level
NMI	-1 †
CKI	0
RPI	1
WPI	1
IOI	0-14

† Non-maskable (except in **Trap State**), **Processor Priority Level** ignored.

All decode traps, instruction fetch traps, and external interrupts are considered precise traps since they are detected prior to instruction issue. All other traps and machine checks are imprecise traps as they occur after instruction issue. The **Trap Summary** reports all traps, interrupts, and machine checks that may have occurred during the time the CPU was *not* in **Trap State.**

The format of the **Trap Summary**, the **Trap Locators**, and the **load/store queue** are implementation dependent; please see Appendix D for more information on these structures.

To begin the Trap Sequence, the processor first vectors to either address 0 or 128 as determined by the type of trap (Table 2-10). General registers may be saved in memory using **store** instructions. (The special **storecpu** instruction is provided for this purpose.) Following that, the processor internal state may be moved to the registers with **rtrpd** instructions, and load/store state may be saved with **slstrpd** instructions.

Table 2-10. Trap Vectoring

Trap Type	Vector Address
Reset	0
All other traps	128

A return from a trap is accomplished by reloading the three **Restart PCs** with three **exts** instructions. Since these instructions must be executed while the processor is in **Trap State**, the operating system must first cause a trap (using the **xtrap** instruction, for example) to enter **Trap State**. Appendix D gives an example of trap state software.

Note that using **exts** with a delayed execution control field that disables either of the following two instructions will produce unpredictable results. Also, while the hardware would never produce a **Restart PC** with I0 enabled and I1 disabled, returning from **Trap State** with such a **PC** is legal and produces the obvious result (I0 will get executed and I1 will not).

Table 2-11. D0 and D1 Decoding for 32-bit Instructions

D0	D1	Trapping Instruction	Instruction Executed Next
0	0	none	I0
0	1	not possible	not possible
1	0	I0	I1
1	1	I0 and/or I1†	‡

† Decode traps can only be caused by I1 in this case.
‡ both I0 and I1 have finished - apply same decoding to next PC

Table 2-12. D0 and D1 Decoding for a 64-bit Instruction

D0	D1	Trapping Instruction	Instruction Executed Next
0	0	none	I0
0	1	not possible	not possible
1	0	not possible	not possible
1	1	I0	‡

‡ I0 has finished - apply same decoding to next PC

For decode traps and instruction fetch traps, which are mutually exclusive, the first **Restart PC**'s **D0** and **D1** bits give an indication of which instruction(s) caused the trap or would have been executed next had there not been a trace trap or an instruction fetch trap. These bits should be interpreted according to Tables 2-11 and 2-12. For the **bpt**, **trap**, **strap**, and **xtrap** instructions, and for an illegal instruction/privilege violation, the disable bit

of the offending instruction will be set. Therefore, upon return, the same instruction word will be fetched, but the offending instruction will be disabled. For breakpoints this is not appropriate. In this case, the original instruction should be put back and the disable bit corresponding to the **bpt** instruction should be cleared. To be able to unambiguously determine which half of the instruction word executed the breakpoint, all **bpt** instructions must use a 32-bit format.

2.18 Reset Operation

When the K-1 is reset, the **Processor Status** register is undefined, **Trap State** is entered, and processing begins with a reset trap. Reset traps are distinguished from other traps by having a different vector address. In the case of a reset trap, *all* other trap information should be ignored.

After first applying power to the K-1, the state of the memory, caches, and page tables will be undefined, and in fact may contain parity or other uncorrectable errors. The memory and data cache should be cleared using **zcl** and **dflush** instructions, the instruction cache should be cleared using **ickill** instructions, and the instruction and data page tables should be cleared using **lipage** and **ldpage** instructions. The **ELF flags** and the general registers must also be initialized. Bootstrap programs that initialize the state may be loaded by the Console into a small section of the instruction cache. The Console may leave some interesting information in certain general registers upon start-up. See Appendix G for details of the reset state of the machine.

CHAPTER 3. K-1 Instructions

The K-1 architecture has been designed with simplicity and regularity as important goals. This is most evident in the way K-1 instructions have been organized. For each possible **operation** there is a corresponding unique **opcode**. For each opcode, a number of different instruction **formats** are available. These formats differ in their lengths and in terms of what operands they provide to the functional units. A given opcode always performs the same operation regardless of which instruction format is used. It is not always true, however, that a given opcode uses all of the fields available in a given format.

PC-relative branches use a special format that does not require an opcode. Other types of branches require an opcode and are restricted to using specific formats.

In the individual instruction descriptions in the following chapters, the functional unit that executes the instruction is explicitly named. The descriptions refer to the register file, flags, and **ELF** flags as if they were arrays, **R[]**, **F[]**, and **ELF[]**. Opcodes are given in hexadecimal. Timing information can be found in Appendix C. The memory access function mem(*size*, *paddr*) and the address mapping function map(*addr*), used in the descriptions of the memory referencing instructions, are described at the beginning of Chapter 5.

3.1 Instruction Formats

The K-1 architecture provides eight different instruction formats. Four of these are 32-bit, and four are 64-bit formats. The long (64-bit) formats provide long constants and target addresses for non-PC-relative branches. 32-bit instructions must be aligned on a 4-byte boundary, and 64-bit instructions must be aligned on an 8-byte boundary.

This chapter makes use of the terms "I0" and "I1", which were defined in Chapter 2. An I0 instruction resides in the most significant (lowest address) half of a 64-bit instruction word (at a byte address that is evenly divisible by 8). I1 instructions reside in the least significant (highest address) half of a 64-bit instruction word (at a byte address that is congruent to 4 modulo 8). All long (64-bit) formats are legal only as I0 instructions. One of the 32-bit formats, the **PC-relative branch** format, is legal only as an I1 instruction.

The instruction format specifies the origin of operands and the location where the result is to be stored. Data operands to instructions are given the names *srca*, *srcb*, *srcc*, and *fsrc*. Not all instructions require all of these operands. The result, if any, is stored in a register, *rdst*, or a flag, *fdst*.

Most formats specify a 2-bit format control field, **FC**, an 8-bit opcode field, **OP**, and a 6-bit register address/flag number field, **RC**. *srcc* and *rdst* are specified by **R[RC]**, *fdst* is specified by **F[RC_{2-0}]** (where **RC_{2-0}** is the low three bits of the **RC** field), and *fsrc* is specified by **F[OP_{2-0}]** (where **OP_{2-0}** is the low three bits of the opcode). *srca* and *srcb* are speci-

fied according to Table 3-1. In this table, as elsewhere, the term "**short constant**" format is used to refer to either the **unconditional short constant** format or the **conditional short constant** format.

Table 3-1. Source Operand Control

FC	CA	*srca*	*srcb*	Format	Legal in	
					I0	I1
10	-	R[RA]	R[RB]	Register	Yes	Yes
0X[†]	0	R[RAB]	S(SCON)[‡]	Short constant	Yes	Yes
0X[†]	1	S(SCON)[‡]	R[RAB]	Short constant	Yes	Yes
11	0	R[RAB]	S(LCON)[‡]	Long constant	Yes	No
11	1	S(LCON)[‡]	R[RAB]	Long constant	Yes	No
11	-	-	-	PC-relative branch	No	Yes
11	-	-	-	Absolute branch	Yes	No
11	-	R[RA]	-	Register branch	Yes	No
11	-	R[RA]	-	**exts**	Yes	No

The **register** format (Figure 3-1) has an **FC** field with the binary value 10. In addition to the standard **RC** field, this format contains two additional register address fields, **RA** and **RB**, that are the register addresses of *srca* and *srcb*, respectively, and two fields, **I** and **FLG**, that control conditional execution of the instruction. If the **I** field is a zero, the instruction is executed only if the flag addressed by the **FLG** field is set (has the value one). If the **I** field is a one, the instruction is executed only if the addressed flag is clear. By specifying flag **f7**, which is always set, and **I** = 0, the instruction can be unconditionally executed. Using the **register** format, an instruction can have two register operands and can write its result to a third register. Some instructions read a third operand instead of, or in addition to, writing a result.

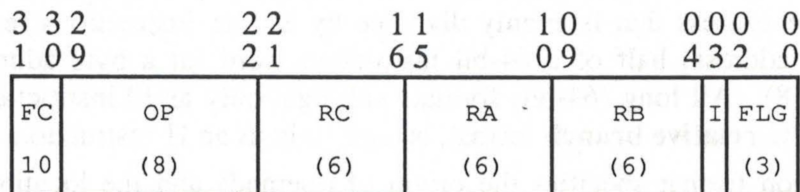


Figure 3-1. Register Instruction Format

† There are two versions of the **short constant** format; **FC<0>** controls whether this format is conditionally executed

‡ S(x) performs format-dependent extension and shifting of a constant to 64 bits.

The **short constant** format has two forms – unconditional and conditional (Figures 3-2 and 3-3). Both are similar to the **register** format except that an immediate, signed constant, **SCON**, can be used in place of either of the register operands. The **RA** field is renamed to **RAB** because **R[RAB]** can be directed to either *srca* or *srcb* under the control of the **CA** field (Table 3-1). If **CA** is a zero, then **R[RAB]** is *srca* and the constant is *srcb*. If **CA** is a one, then **R[RAB]** is *srcb* and the constant is *srca*. If **FC<0>** is a 0, then the instruction is in **unconditional short constant** format (Figure 3-2); if **FC<0>** is a 1, then the instruction is in **conditional short constant** instruction format (Figure 3-3). The **unconditional short constant** format provides a 9-bit signed short constant (**SCON9**); the **conditional short constant** format provides a 5-bit signed short constant (**SCON5**), but allows conditional execution as in the **register** format.

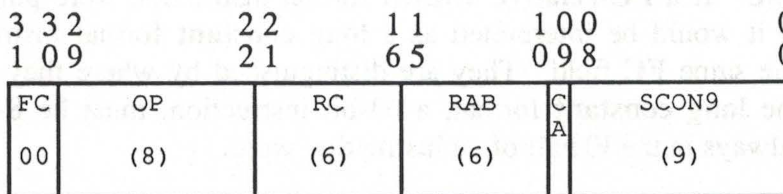


Figure 3-2. Unconditional Short Constant Instruction Format

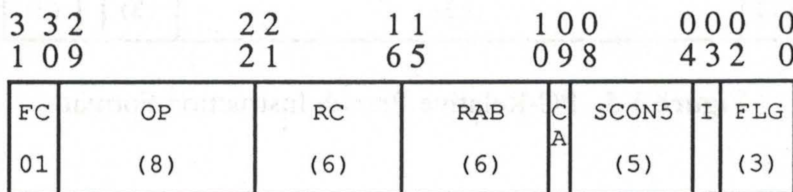


Figure 3-3. Conditional Short Constant Instruction Format

The **long constant** format (Figure 3-4) is the first of the 64-bit formats; it has an **FC** field with the binary value 11. The **long constant** format is similar to the **conditional short constant** format with the **SCON** field replaced by the 36-bit **LCON** field. As in the **short constant** formats, the **CA** field determines the ordering of **R[RAB]** and the constant (in this case **LCON**). The **LF** (LeFt) field controls the extending of **LCON** to 64 bits. If **LF** is a zero, then **LCON** is right-justified within 64 bits and sign-extended. However, if **LF** is a one, then **LCON** is left adjusted within 64 bits and the low 28 bits are set to zero. An instruction using this format may also be conditionally executed by using the **I** and **FLG** fields, as in the **register** format.

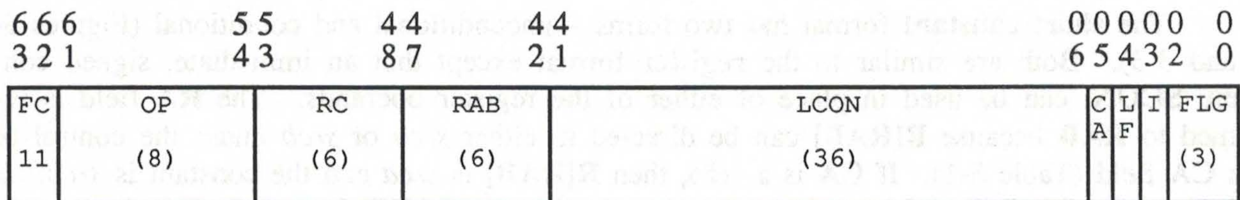


Figure 3-4. Long Constant Instruction Format

The **PC-relative branch** format (Figure 3-5), whose **FC** field has a binary value of 11, is a 32-bit format that is allowed only in the I1 (low-order) half of an instruction word. Therefore, there must be another 32-bit format instruction in the I0 (high-order) half of the same instruction word. If a **PC-relative branch** format instruction were put in the I0 half of an instruction word it would be interpreted as a **long constant** format instruction since both these formats use the same **FC** field. They are distinguished by where they occur in a 64-bit instruction word: the **long constant** format, a 64-bit instruction, must be 8-byte aligned and thus its **FC** field is always in the I0 half of an instruction word.

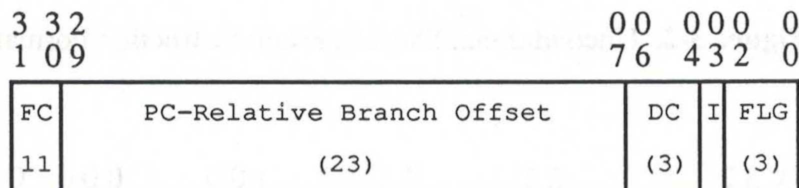


Figure 3-5. PC-Relative Branch Instruction Format

The remaining bits in the **PC-relative branch** format are a 23-bit signed **PC-Relative Branch Offset**, a 3-bit delayed execution control field **DC**, and the **I** and **FLG** fields. The **PC-Relative Branch Offset** is a signed offset from the address of the I0 half of the instruction word containing the PC-relative branch instruction, expressed as a number of 4-byte units. The branch address is calculated by summing the **PC-Relative Branch Offset** and the address of the first byte of the 64-bit instruction word containing the PC-relative branch.

The **I** and **FLG** fields in the **PC-relative branch** format are used to control branching: if the flag is set and **I** is zero, or if the flag is clear and **I** is one, then the branch is taken. (Note that this is identical to the way conditional execution is controlled in other formats.) Execution of the instructions following the branch is controlled by the **DC** field (Table 3-2).

Three variations of the **long constant** format are used for branches: if the opcode of an I0 instruction is **jump** or **call** with an address expression operand (as opposed to a register operand), then the instruction is decoded in **absolute branch** format rather than **long constant** format; if the opcode is **jump** or **call** with a register operand or is **ickill,** then the instruction is decoded in **register branch** format rather than **long constant** format; if the opcode is **exts,** then the instruction is decoded in **exts** format. In each of these cases the **FC** field must

have the binary value 11. Note that **jump (call)** with an address expression operand uses a different opcode from **jump (call)** with a register operand; if this was not the case, these two formats could not be distinguished. As noted in Table 3-2, the **DC** field also controls the PC stored for a **call** instruction, which is always an offset from the address of the **call** instruction.

Table 3-2. Delayed Execution Control Decoding

DC	Assembler Mnemonic	Stored PC if **call**: Address of **call** +	Branch taken		Branch not taken	
			1st	2nd	1st	2nd
0	ff	8	No	No	Yes	Yes
1	fa	24	No	Yes	Yes	Yes
2	af	16	Yes	No	Yes	Yes
3	aa	24	Yes	Yes	Yes	Yes
4	bf	16	Yes	No	No	Yes
5	bb	24	Yes	Yes	No	No
6	ab	24	Yes	Yes	Yes	No
7	fb	24	No	Yes	Yes	No

1st = execute from first delay word
2nd = execute from second delay word
a = always
b = if branch
f = if fall-through

The **absolute branch** format (Figure 3-6) has the same **FC** field as the **long constant** format, but is distinguished by the opcode being **jump** or **call** with an address expression operand. The **RC** field specifies the register in which to store the PC if the opcode is **call**. The instruction is conditionally executed based on the **I** and **FLG** fields. If the instruction is executed, then the branch address is the 41-bit **Absolute Branch Address** field of the instruction shifted left by two bit positions [3-1]. The delayed execution control bits operate as in the **PC-relative branch** format. (See Table 3-2).

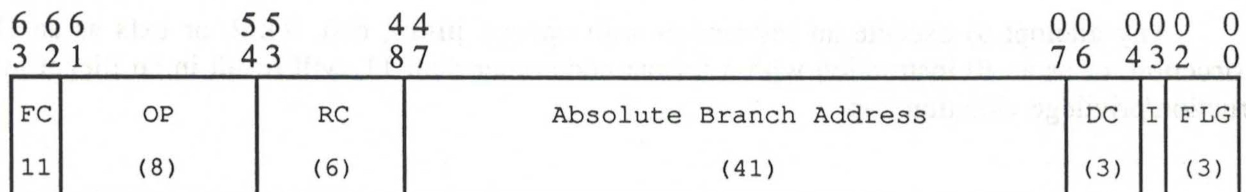


Figure 3-6. Absolute Branch Instruction Format

The **register branch** format (Figure 3-7) has the same **FC** field as the **long constant** format, and is used when the opcode is **jump** or **call** with a register operand, or is **ickill**. The **RC** field specifies the register in which to store the PC if the opcode is **call**. The instruction is conditionally executed based on the **I** and **FLG** fields. If the instruction is executed, then the branch address is taken from the register specified by the **RA** field. The delayed execution control bits operate as in the **PC-relative branch** format. (See Table 3-2).

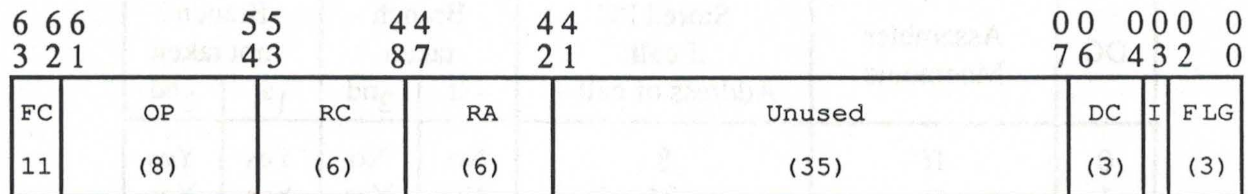


Figure 3-7. Register Branch Instruction Format

The **exts** instruction format (Figure 3-8) has the same **FC** field as the **long constant** format, and is used when the opcode is **exts**. The instruction is conditionally executed based on the **I** and **FLG** fields. If the instruction is executed, then the branch address is taken from the register specified by the **RA** field. The delayed execution control bits operate as in the **PC-relative branch** format, but using a delay code that could disable either of the following two instructions will produce unpredictable results. The **exts Load Address** field, when multiplied by eight, gives an absolute virtual memory address whose contents are to be loaded into the register specified by the **RC** field. Bits 7 and 8 are unused in this format and must be zero. (Note that the **exts Load Address** is transformed to make it unique among all the CPUs in a multiprocessor; see the **exts** instruction description for details.)

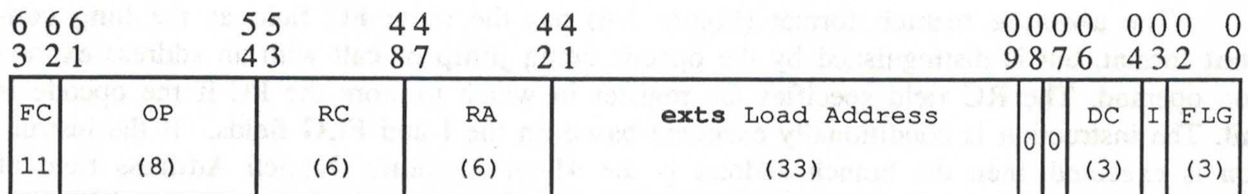


Figure 3-8. exts Instruction Format

Any attempt to execute an instruction with opcode **jump, call, ickill,** or **exts** as an I1 instruction, or as an I0 instruction with a format code other than 11, will result in an illegal instruction/privilege violation trap.

CHAPTER 4. Floating-Point Instructions

Floating-point operations can be performed on 32-bit (single) or 64-bit (double) precision numbers in IEEE format (*ANSI/IEEE 754-1985*). The precision is specified directly by the opcode, while the rounding mode, and ability to trap are controlled by bits in the **Processor Status** register. (Note that for convert instructions, the rounding mode can be specified either by the *srcb* operand, or by the **Processor Status** register, as explained in section 2 of this chapter). There are three classes of floating-point instructions: comparisons, conversions and computations. The computations are carried out in the floating-point add, floating-point multiply and floating-point divide/square root units, the comparisons are done in the integer unit, and the conversions are performed in the floating-point add unit. Exact descriptions of how the conversions and computations are carried out (and under what conditions different types of exceptions may result) can be found in Appendix I.

Many of these operations can result in floating-point exception conditions. If an exception occurs, then the corresponding bit in the **Arithmetic Exception Flags** field in the **Processor Status** register will be set. These flags are "sticky" in that they may be set but may never be cleared by a floating-point instruction. The only way to clear an exception flag is with a **wps** instruction, which can set the flags to any value. Note, however, that setting a bit in the **Arithmetic Exception Flags** field with **wps** will *not* cause the corresponding trap, even if enabled. The only way to cause an arithmetic trap is to execute an instruction that causes the desired exception while traps are enabled for that exception.

4.1 Floating-Point Compare Instructions

Floating-point numbers of the same precision may be compared using the floating-point compare instructions. The complete set of IEEE comparisons may be made with the understanding that the order of the operands may be reversed, and that conditional branches and conditional execution can test the negation of the flags. For example, if the programmer wishes to determine if the double precision floating-point number in register **r1** is negative and set flag **f3** accordingly, he may write:

```
cmpgt.d 0.0,%r1,%f3
```

since no **cmplt.d** instruction is available. The only exception to this is the **uge** (unordered or greater than or equal) comparison which requires two comparison instructions.

In accordance with the IEEE standard, if one or both of the operands being compared is a NaN, then the quantities are considered to be **unordered**. For any two quantities, therefore, exactly one of the four relationships: **greater than**, **less than**, **equal**, or **unordered** will be true. Invalid operation traps can be caused by floating-point comparisons if they are enabled by the appropriate bit in the **Arithmetic Trap Enables** field in the **Processor Status** register (see Table 2-6) and if one of two other conditions occurs:

(1) one or both of the operands is a signaling NaN, or

(2) the operands are unordered and the comparison test considers this to be an exception condition.

The complete list of K-1 floating-point comparison operations is given in Table 4-1, in which the mnemonics for the operations have the following meanings:

gt greater than
ge greater than or equal
lg less than or greater than
leg less than, equal, or greater than
eq equal
un unordered (i.e., at least one operand is a NaN)
ueq unordered or equal
ugt unordered or greater than

Table 4-1. Floating-Point Comparisons

OP	Greater Than	Less Than	Equal	Unordered	Exception if Unordered
gt	T	F	F	F	Yes
ge	T	F	T	F	Yes
lg	T	T	F	F	Yes
leg	T	T	T	F	Yes
eq	F	F	T	F	No
un	F	F	F	T	No
ueq	F	F	T	T	No
ugt	T	F	F	T	No

Instructions: **cmp{gt,ge,lg,leg,eq,un,ueq,ugt}.d** *srca,srcb,fdst*

Opcodes:

cmpgt.d	31
cmpge.d	37
cmplg.d	33
cmpleg.d	35
cmpeq.d	3B
cmpun.d	3D
cmpueq.d	3F
cmpugt.d	39

Operation: Compare 64-bit precision floating-point numbers and set flag.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Invalid operation

Description:

The two 64-bit precision floating-point numbers, *srca* and *srcb*, are compared and the result of the comparison is recorded in the flag *fdst*. *srca* is the left comparand and *srcb* is the right comparand. See Table 4-1 for the definitions of the individual tests. If one of the operands is a signaling NaN, or if one of the operands is a quiet NaN and Table 4-1 indicates an exception if unordered, then an invalid operation exception will occur and the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

Instructions: **cmp{gt,ge,lg,leg,eq,un,ueq,ugt}.s** *srca,srcb,fdst*

Opcodes:

cmpgt.s	30
cmpge.s	36
cmplg.s	32
cmpleg.s	34
cmpeq.s	3A
cmpun.s	3C
cmpueq.s	3E
cmpugt.s	38

Operation: Compare 32-bit precision floating-point numbers and set flag.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Invalid operation

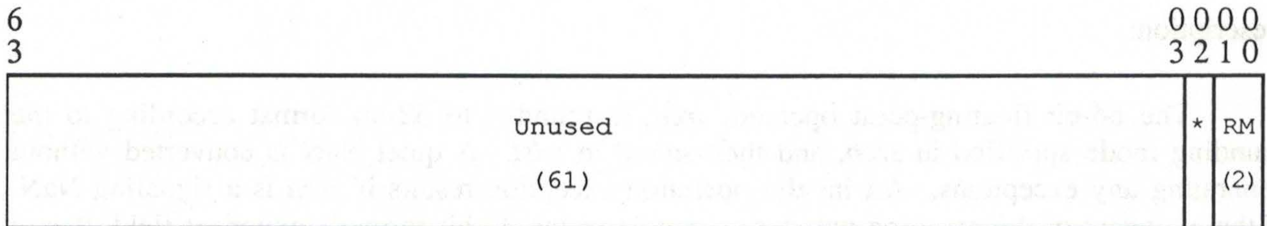
Description:

The two 32-bit precision floating-point numbers, *srca* and *srcb*, are compared and the result of the comparison is recorded in the flag *fdst*. *srca* is the left comparand and *srcb* is the right comparand. See Table 4-1 for the definitions of the individual tests. If one of the operands is a signaling NaN, or if one of the operands is a quiet NaN and Table 4-1 indicates an exception if unordered, then an invalid operation exception will occur and the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

4.2 Floating-Point Conversion Instructions

Numbers may be converted between 64-bit and 32-bit floating-point formats and between either of the floating-point precisions and 64-bit integer format. Conversion may require rounding and may result in floating-point overflow, underflow, invalid operation, or inexact exceptions (and possibly traps, if enabled). Conversion of a quiet NaN from one floating-point format to another is possible without causing an exception.

In order to provide a means for library routines to use a fixed rounding mode (instead of the **Rounding Mode<1..0>** field of the **Processor Status** register which could have been set arbitrarily by an application program), all but one of the floating-point conversion instructions take a *srcb* argument (shown in Figure 4-1) which can directly specify the rounding mode. The **RMB** bit, if set, causes the rounding mode to come from the **RM** field of the *srcb* argument, instead of the **Rounding Mode<1..0>** field of the **Processor Status** register. If the **RMB** bit is not set, the **RM** field is ignored and the rounding mode in the **Processor Status** register is used. Note that the **cvts.d** (convert from single precision to double precision) instruction will never have to round, and thus does not require a *srcb* argument.



* = RMB

Figure 4-1. Floating-Point Conversion *srcb* Argument

In the following instruction descriptions, the phrase "according to the rounding mode specified in *srcb*" should be understood to mean *either* the rounding mode specified in the **RM** field of *srcb* (if the **RMB** bit is set), *or* the rounding mode specified in the **Processor Status** register (if the **RMB** bit is not set).

Instruction: **cvtd.s** *srca,srcb,rdst*

Opcode: 94

Operation: Convert a 64-bit precision floating-point number to 32-bit precision.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The 64-bit floating-point operand, *srca,* is rounded to 32-bit format according to the rounding mode specified in *srcb*, and then stored in *rdst*. A quiet NaN is converted without generating any exceptions. An invalid operation exception results if *srca* is a signaling NaN. If the exponent of the resulting number cannot fit in the 32-bit format's exponent field, then a floating-point overflow or underflow exception occurs, as appropriate. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap.

Instruction: **cvts.d** *srca,rdst*

Opcode: 84

Operation: Convert a 32-bit precision floating-point number to 64-bit precision.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation

Description:

The 32-bit floating-point operand, *srca*, is converted to 64-bit format and stored in *rdst*. No rounding is necessary, nor can there be any overflow or underflow. A quiet NaN is converted without generating any exceptions. An invalid operation exception results if *srca* is a signaling NaN. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

Instruction: **cvtd.l** *srca,srcb,rdst*

Opcode: 95

Operation: Convert a 64-bit precision floating-point number to a 64-bit integer.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, inexact

Description:

The 64-bit floating-point number, *srca*, is rounded to an integer value according to the rounding mode specified in *srcb*, and the result is converted to integer format and stored in *rdst*. If the result for a finite input cannot be represented as a signed 64-bit integer, or if *srca* is positive or negative infinity, then an invalid operation exception is signaled and the largest integer of the same sign as *srca* is stored. An invalid operation exception also results if *srca* is a signaling or quiet NaN, and a zero is stored. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap.

Instruction: **cvts.l** *srca,srcb,rdst*

Opcode: 85

Operation: Convert a 32-bit precision floating-point number to a 64-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, inexact

Description:

The 32-bit floating-point number, *srca*, is rounded to an integer value according to the rounding mode specified in *srcb*, and the result is converted to integer format and stored in *rdst*. If the result for a finite input cannot be represented as a signed 64-bit integer, or if *srca* is positive or negative infinity, then an invalid operation exception is signaled and the largest integer of the same sign as *srca* is stored. An invalid operation exception also results if *srca* is a signaling or quiet NaN, and a zero is stored. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap.

Instruction: **cvtl.d** *srca,srcb,rdst*

Opcode: 97

Operation: Convert a signed 64-bit integer to a 64-bit precision floating-point number.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Inexact

Description:

The signed 64-bit integer, *srca*, is rounded to a 64-bit floating-point number according to the rounding mode specified in *srcb*. The result is stored in *rdst*. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

Instruction: **cvtul.d** *srca,srcb,rdst*

Opcode: 98

Operation: Convert an unsigned 64-bit integer to a 64-bit precision floating-point number.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Inexact

Description:

The unsigned 64-bit integer, *srca*, is rounded to a 64-bit floating-point number according to the rounding mode specified in *srcb*. The result is stored in *rdst*. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

Instruction: **cvtl.s** *srca,srcb,rdst*

Opcode: 96

Operation: Convert a signed 64-bit integer to a 32-bit precision floating-point number.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Inexact

Description:

The signed 64-bit integer, *srca*, is rounded to a 32-bit floating-point number according to the rounding mode specified in *srcb*. The result is stored in *rdst*. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enable** in the **Processor Status** register, this exception will cause a trap.

4.3 Floating-Point Computation Instructions

The floating-point computation instructions perform addition, subtraction, negation, multiplication, division, and square root. One of three floating-point functional units is involved in each calculation. All operations except negation generate rounded results according to the rounding mode in the **Processor Status** register, and, depending upon the operation and the input operands, can cause invalid operation, floating-point overflow, floating-point underflow, floating-point division by zero, and inexact exceptions. Negation is considered to be a data moving operation and therefore never causes any exceptions.

Instruction: **neg.d** *srca,rdst*

Opcode: 92

Operation: Compute the negative of a 64-bit precision floating-point number.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: none

Description:

The negative of the 64-bit floating-point operand, *srca,* is computed and stored in *rdst.* The operation is performed by complementing the sign bit of *srca* regardless of the value it represents. No exception can be generated by negation.

Instruction: **neg.s** *srca,rdst*

Opcode: 82

Operation: Compute the negative of a 32-bit precision floating-point number.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: none

Description:

The negative of the 32-bit floating-point operand, *srca*, is computed and stored in *rdst*. The operation is performed by complementing the sign bit of *srca* regardless of the value it represents. No exception can be generated by negation.

Instruction: **add.d** *srca,srcb,rdst*

Opcode: 90

Operation: Add two 64-bit precision floating-point numbers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded sum of the two 64-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 64-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if one of the operands is positive infinity while the other operand is negative infinity. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **add.s** *srca,srcb,rdst*

Opcode: 80

Operation: Add two 32-bit precision floating-point numbers.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded sum of the two 32-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 32-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if one of the operands is positive infinity while the other operand is negative infinity. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the corresponding **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **sub.d** *srca,srcb,rdst*

Opcode: 91

Operation: Subtract one 64-bit precision floating-point number from another.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded difference, *srcb* - *srca*, of the two 64-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Note that the order of the operands is backwards from what might be expected. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 64-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if both of the operands are infinities with the same sign. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **sub.s** *srca,srcb,rdst*

Opcode: 81

Operation: Subtract one 32-bit precision floating-point number from another.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Add

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded difference, *srcb - srca,* of the two 32-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Note that the order of the operands is backwards from what might be expected. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 32-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if both of the operands are infinities with the same sign. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **mult.d** *srca,srcb,rdst*

Opcode: A1

Operation: Multiply two 64-bit precision floating-point numbers.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Multiply

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded product of the two 64-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 64-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if one of the operands is infinity while the other operand is zero. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **mult.s** *srca,srcb,rdst*

Opcode: A0

Operation: Multiply two 32-bit precision floating-point numbers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Multiply

Exceptions: Invalid operation, floating-point overflow, floating-point underflow, inexact

Description:

The rounded product of the two 32-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 32-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if one of the operands is infinity while the other operand is zero. The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **div.d** *srca,srcb,rdst*

Opcode: A3

Operation: Divide one 64-bit precision floating-point number by another.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Divide/Square Root

Exceptions: Invalid operation, division by zero, floating-point overflow, floating-point underflow, inexact

Description:

The rounded quotient, *srca* ÷ *srcb*, of the two 64-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 64-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if both of the operands are zero, or if both of the operands are infinity. The result in any case will be a quiet NaN. If the numerator is not zero, NaN, or infinity while the denominator is zero, then a division by zero exception will occur and the correctly signed infinity will be produced as a result. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **div.s** *srca,srcb,rdst*

Opcode: A2

Operation: Divide one 32-bit precision floating-point number by another.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Divide/Square Root

Exceptions: Invalid operation, division by zero, floating-point overflow, floating-point underflow, inexact

Description:

The rounded quotient, *srca* ÷ *srcb*, of the two 32-bit floating-point operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. Floating-point overflow or floating-point underflow exceptions occur if the magnitude of a finite result is too big or too small to be represented in the 32-bit floating-point format. If traps are disabled, the result will be infinity or zero, respectively. If traps are enabled, the result will be as described in Appendix I. An invalid operation exception will occur if either or both of the operands is a signaling NaN, or if both of the operands are zero, or if both of the operands are infinity. The result in any case will be a quiet NaN. If the numerator is not zero, NaN, or infinity while the denominator is zero, then a division by zero exception will occur and the correctly signed infinity will be produced as a result. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, any of these exceptions will cause a trap. If none of the inputs is a signaling NaN, and one or more of the inputs is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **sqrt.d** *srca,rdst*

Opcode: B3

Operation: Compute the square root of a 64-bit precision floating-point number.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Divide/Square Root

Exceptions: Invalid operation, inexact

Description:

The square root of the 64-bit operand, *srca,* is computed and stored in *rdst.* Rounding is performed according to the rounding mode specified in the **Processor Status** register. If the input operand is finite and greater than or equal to zero, the result will be finite and non-negative. If the input operand is negative zero or positive infinity, the result will be negative zero or positive infinity, respectively. An invalid operation exception will occur if *srca* is either a signaling NaN, or a negative operand other than negative zero (including negative infinity). The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, either of these exceptions will cause a trap. If the input is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

Instruction: **sqrt.s** *srca,rdst*

Opcode: B2

Operation: Compute the square root of a 32-bit precision floating-point number.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-Point Divide/Square Root

Exceptions: Invalid operation, inexact

Description:

The square root of the 32-bit operand, *srca,* is computed and stored in *rdst*. Rounding is performed according to the rounding mode specified in the **Processor Status** register. If the input operand is finite and greater than or equal to zero, the result will be finite and non-negative. If the input operand is negative zero or positive infinity, the result will be negative zero or positive infinity, respectively. An invalid operation exception will occur if *srca* is either a signaling NaN, or a negative operand other than negative zero (including negative infinity). The result in either case will be a quiet NaN. If there was any loss of precision, then an inexact exception occurs. If an exception happens, the corresponding exception flag will be set. If enabled by the **Arithmetic Trap Enables** in the **Processor Status** register, either of these exceptions will cause a trap. If the input is a quiet NaN, then there will be no exception and the result will be a quiet NaN.

CHAPTER 5. Load and Store Instructions

This chapter describes load and store instructions, which are used to move data between the registers and main memory. It also describes instructions that read, write and test the **ELF** flags, instructions that are used for memory diagnostics, and instructions that write **Data Watchpoint Table** entries.

5.1 Referencing Memory

The operands of **load, eload** and **store** instructions are used to specify source or destination register addresses and to calculate memory addresses. A memory address may be specified directly by an operand, or it may be calculated by multiplying the index operand by 1, 2, 4, or 8 as specified by the opcode, and adding the result to the base operand (Figure 2-5). (The index multiplier, implicit in the opcode, is referred to as m in the instruction descriptions.) Memory addresses are architecturally limited to 48 bits, and may be restricted further by implementations of the architecture [5-1]. The opcode also implicitly specifies a memory precision (1, 2, 4, or 8 bytes). If the memory address is not a multiple of the precision, an illegal address condition will occur.

After the address is calculated, it may be modified if either of the **Byte Order Low-to-High** (Table 2-8) or **Small Address Compatibility Mode** bits in the **Processor Status** register is set. (See **Processor Status** in Chapter 2). In brief, **Byte Order Low-to-High** will transform the low three bits of the address according to Table 2-8, and **Small Address Compatibility Mode** will clear any bits in the address above bit 31. The **loadcpu**, **storecpu**, and **exts** instructions perform an additional translation to make the address unique among all of the CPUs in a multiprocessor [5-6]. The address resulting from these three possible transformations is a virtual address and must be mapped into a physical address before accessing any data. This translation process also produces a set of protection bits controlling read and write access, and a bit that indicates if the data is **shared.** (See the section on **Data Mapping** in Chapter 2). The virtual-to-physical address mapping function depends on whether the reference is a user or a supervisor mode reference.

The **User Mode Load** and **User Mode Store** bits control whether memory references are treated as **user** mode references or **supervisor** mode references. This topic is discussed in the section on **Data Mapping** in Chapter 2, and is illustrated in Table 2-4. The **User Protection** bit controls whether **user** mode references use the protections provided in the data page table, or whether they get supervisor "over-ride" privileges. This topic is also discussed in the section on **Data Mapping** in Chapter 2.

Supervisor mode references use the identity mapping to produce a physical address; this mapping also produces a default set of protection bits that grant both read and write access and indicate that the page is shared. For a user mode reference, the translation process

reads the data page table at an address that is a function of the given virtual address and the **Process Key<12..0>** field of the **Processor Status** register [5-2]. If the data page table entry that is read is *not* the entry corresponding to the given address, a data map miss trap will occur. (The data page table is a cache and cannot simultaneously map all possible pages). If the entry is found, then it contains the physical address, the access bits and the shared bit. Finally, for a user mode reference, if the **User Protection** bit in the **Processor Status** register is *off*, then the read and write access permission bits are both forced *on* if the page is valid (has either read or write access).

The above mapping process is performed for all memory reference instructions and can result in a data map miss for user mode references. The access bits that are produced are used to check whether the reference to be performed is legal. References that read memory (**load**, **eload**, **pcl**, etc.) require read access; references that write memory (**store**, **dflush**, etc.) require write access; **swat** instructions, which both read and write memory, require both read and write access. If the reference is not allowed and the instruction is not an **eload**, an illegal access trap will occur. For **eload** instructions, no illegal access trap is generated; this condition is signaled by storing zero in *rdst* and setting **ELF[RC]**.

If no problems occur during the computation and mapping of the memory address, then the requested load/store reference will take place. There are signed and unsigned versions of all **load** (and **eload**) instructions for precisions less than 8 bytes. The signed version of a **load** will sign-extend the data read to 64 bits; the unsigned version will zero-extend the data to 64 bits. For 64-bit data, the assembler accepts both signed and unsigned instruction mnemonics (though both refer to the same opcode); **load.l** and **loadu.l**, for example, both perform a load of 64 bits of data into a register.

Constants may be used in place of registers as operands (see Chapter 3), except that data to be stored in memory may not be a constant. To store a constant value in a memory location, the constant must first be loaded into a register, as in the following sequence:

```
move      <constant>, %r1
store.l   %r1, <memory address>
```

The function map(*addr*), which is referred to extensively in the instruction descriptions later in this chapter, performs all the transformations necessary to turn its virtual address argument into a physical address that can be used to reference memory. This includes the effects of the **Byte Order Low-to-High** and **Small Address Compatibility Mode** bits in the **Processor Status** register, the special transformations performed by the **loadcpu**, **storecpu**, and **exts** instructions, and virtual-to-physical address mapping. Note that the map function will not return a result larger than the size of a physical memory address [5-1]. Memory referencing instructions can trap during the mapping process (for traps such as illegal address, data map miss, and illegal access), or after referencing memory (for traps such as Nonexistent Memory and ECC errors).

The memory access function mem(*size*, *paddr*) refers to *size* consecutive bytes starting at physical address *paddr*. That is, *paddr* is a result of the map(*addr*) function.

5.2 Memory-Referencing Instruction Traps

A number of trapping conditions are common to major classes of memory-referencing instructions. Table 5-1 shows the traps that can occur for **load**-type, **eload**-type, **store**-type, **dflush**, and **zcl** instructions. Some special load/store instructions, such as **dflush**, **exts** and **slstrpd** are described in other chapters of this manual but refer to Table 5-1 to enumerate the conditions under which they trap. The individual instruction descriptions that follow refer to these trap classes instead of listing all the traps individually. Explanations of the circumstances under which these traps can occur also appear below and not with the individual instruction descriptions. See Appendix D for more information on load/store traps.

Table 5-1. Memory Referencing Instruction Traps

Trap/Error	load	eload	store	dflush	zcl
Data Map Miss	1	1	1	1	1
Illegal Address	3	2,3	3	no	4
Illegal Access	1	no	1	1	1
Data Watchpoint	5	5	5	no	5
PM/CT Parity	yes	yes	yes	yes	yes
ECC (1st sub-line)	yes	yes	yes	no	no
ECC (subsequent sub-line)	yes	yes	yes	no	no
Memory-related Parity	yes	yes	yes	no	no
Nonexistent Memory	yes	1	yes	no	no

[1] for **user** mode references only

[2] if the **Early Load Alignment Trap** bit in the **Processor Status** register is set, and ...

[3] if the precision is 2, 4 or 8 bytes

[4] if sufficient low-order address bits are not zero [5-5]

[5] Yes, except in **Trap State** which disables Data Watchpoint traps

A **data map miss** trap occurs when a **user** mode reference is not found in the data page table. A complete description of this issue is given in Chapter 2 in the section on **Data Mapping.**

An **illegal address** trap occurs when the address of the quantity being referenced is not a multiple of its precision. For example, a 4-byte **load** must use a 4-byte aligned address (the low two bits of the address must be zero). Note that the **zcl** instruction references an entire data cache line and thus requires more low-order address bits to be zero than **load** instructions (which reference eight bytes or less) [5-5].

An **illegal access** trap occurs when a **user** mode reference does not have the proper permissions in the data page table for the requested reference. (I.e., a **load** must have read permission). This topic is also discussed in Chapter 2 in the section on **Data Mapping**.

A **data watchpoint** trap occurs when the referenced address matches an address in one of several **Data Watchpoint Table** entries in the load/store unit. The **Data Watchpoint Table** entries (or just Data Watchpoints) can be set with the **wdwp** instruction. See the **wdwp** instruction description for details on setting Data Watchpoints and the conditions under which they will cause a trap (or stop the system clocks).

A **page map/cache tag parity error** can occur when reading the data page table (also called the page map) for a **user** mode reference, or when reading the tag fields of the data cache (which are read for all memory references). There are both recoverable (occurring with a data cache miss) and non-recoverable (occurring with a data cache hit) versions of this error.

A cache miss will transfer an entire cache line of data from the memory system into the cache, but this transfer actually comprises a series of smaller transfers of sub-lines (the largest unit that can be read from or written to the cache at one time). The K-1 distinguishes two types of ECC (Error Correcting Code) errors: **ECC errors on the first sub-line** transferred, and **ECC errors on subsequent sub-lines.** In order to decrease latency, the memory system always returns the sub-line with the referenced data first. Thus, for **load** and **eload** instructions (including **ldecc**, **ldnecc**, and **loadcpu**) the distinction between first and subsequent lines indicates whether the data written to the register file is correct or not. When data with ECC errors is returned from the memory system, the cache tags for that line are invalidated (so that the incorrect data cannot be accessed). Note that a **store** instruction can get ECC errors since a **store** that misses in the data cache must retrieve the referenced cache line from the memory system. **store** instructions, however, don't care which type of ECC error they receive – the data can not be marked as valid in the data cache so the store can not complete. Correctable and uncorrectable memory errors are logged as described in Appendices E and G.

A **memory-related parity error** is caused when the processor fails to complete a memory operation because of an internal error in the memory subsystem. This error is usually indicative of serious problems in the memory system.

A **nonexistent memory error** occurs when a reference outside the bounds of physical memory is made. For a user mode reference, this can only happen if the data page table has been set up incorrectly.

For all traps *except* the ECC error on subsequent sub-lines trap and the non-recoverable version of the page map/cache tag parity error trap, all **load** (including **ldecc**, **ldnecc**, and **loadcpu**) and **eload** instructions that trap will *not* modify their destination register. **eload** instructions that detect an error but do not trap (as for an illegal access) will always write their destination register to zero. The cases in which this can happen are: illegal access errors (which can only happen with **user** mode references), illegal address errors (but only if the **Early Load Alignment Trap** bit in the **Processor Status** register is clear), and nonexistent memory errors for supervisor mode references. These three cases will be referred to as the "special **eload** error conditions" in the instruction descriptions. The section on **Early Load** in Chapter 2 gives a complete description of the trap behavior of **eload** instructions.

5.3 Load Instructions

The following instruction descriptions are for all the “normal” **load** and **eload** instructions. Several special types of **load** instructions are covered in a later section.

Instruction: **loadu.b** (*srca*)[*srcb*:1],*rdst*

Opcode: D1

Operation: Load one byte of data from memory into a register.

Operands used: *srca*, *srcb*, mem(1, map(*srca* + *srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *srcb*). If no trap occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **loadu.b** *srca,rdst*

Opcode: D0

Operation: Load one byte of data from memory into a register.

Operands used: *srca*, mem(1, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instructions:	**loadu.h**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**loadu.h**	(*srca*)[*srcb*:1],*rdst*	D5
	loadu.h	(*srca*)[*srcb*:2],*rdst*	D6

Operation: Load two bytes of data from memory into a register.

Operands used: *srca*, *srcb*, mem(2, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **loadu.h** *srca,rdst*

Opcode: D4

Operation: Load two bytes of data from memory into a register.

Operands used: *srca*, mem(2, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instructions:	**loadu.w**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**loadu.w**	(*srca*)[*srcb*:1],*rdst*	D9
	loadu.w	(*srca*)[*srcb*:4],*rdst*	DA

Operation: Load four bytes of data from memory into a register.

Operands used: *srca*, *srcb*, mem(4, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **loadu.w** *srca,rdst*

Opcode: D8

Operation: Load four bytes of data from memory into a register.

Operands used: *srca*, mem(4, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instructions:	**load[u].l**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**load[u].l**	(*srca*)[*srcb*:1],*rdst*	CD
	load[u].l	(*srca*)[*srcb*:8],*rdst*	CE

Operation: Load eight bytes of data from memory into a register.

Operands used: *srca*, *srcb*, mem(8, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **load[u].l** *srca,rdst*

Opcode: CC

Operation: Load eight bytes of data from memory into a register.

Operands used: *srca*, mem(8, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **load.b** (*srca*)[*srcb*:1],*rdst*

Opcode: C1

Operation: Load one byte of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(1, map(*srca* + *srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *srcb*). If no trap occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **load.b** *srca,rdst*

Opcode: C0

Operation: Load one byte of data from memory into a register, extending the sign.

Operands used: *srca*, mem(1, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instructions:	**load.h**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**load.h**	(*srca*)[*srcb*:1],*rdst*	C5
	load.h	(*srca*)[*srcb*:2],*rdst*	C6

Operation: Load two bytes of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(2, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **load.h** *srca,rdst*

Opcode: C4

Operation: Load two bytes of data from memory into a register, extending the sign.

Operands used: *srca*, mem(2, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instructions:	**load.w**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**load.w**	(*srca*)[*srcb*:1],*rdst*	C9
	load.w	(*srca*)[*srcb*:4],*rdst*	CA

Operation: Load four bytes of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(4, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **load.w** *srca,rdst*

Opcode: C8

Operation: Load four bytes of data from memory into a register, extending the sign.

Operands used: *srca*, mem(4, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **eloadu.b** (*srca*)[*srcb*:1],*rdst*

Opcode: F1

Operation: Load one byte of data from memory into a register.

Operands used: *srca*, *srcb*, mem(1, map(*srca* + *srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *srcb*). If no error occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eloadu.b** *srca,rdst*

Opcode: F0

Operation: Load one byte of data from memory into a register.

Operands used: *srca*, mem(1, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instructions:	**eloadu.h**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**eloadu.h**	(*srca*)[*srcb*:1],*rdst*	F5
	eloadu.h	(*srca*)[*srcb*:2],*rdst*	F6

Operation: Load two bytes of data from memory into a register.

Operands used: *srca*, *srcb*, mem(2, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no error occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eloadu.h** *srca,rdst*

Opcode: F4

Operation: Load two bytes of data from memory into a register.

Operands used: *srca*, mem(2, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instructions:	**eloadu.w**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**eloadu.w**	(*srca*)[*srcb*:1],*rdst*	F9
	eloadu.w	(*srca*)[*srcb*:4],*rdst*	FA
Operation:	Load four bytes of data from memory into a register.		
Operands used:	*srca*, *srcb*, mem(4, map(*srca* + *m***srcb*))		
Results stored:	*rdst*, **ELF[RC]**		
Legal in:	User or Supervisor mode		
Functional unit:	Load/Store		
Exceptions:	**eload**-type (see Table 5-1)		

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no error occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eloadu.w** *srca,rdst*

Opcode: F8

Operation: Load four bytes of data from memory into a register.

Operands used: *srca*, mem(4, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, zero-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instructions::	**eload[u].l**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**eload[u].l**	(*srca*)[*srcb*:1],*rdst*	ED
	eload[u].l	(*srca*)[*srcb*:8],*rdst*	EE

Operation: Load eight bytes of data from memory into a register.

Operands used: *srca*, *srcb*, mem(8, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no error occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eload[u].l** *srca,rdst*

Opcode: EC

Operation: Load eight bytes of data from memory into a register.

Operands used: *srca*, mem(8, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eload.b** (*srca*)[*srcb*:1],*rdst*

Opcode: E1

Operation: Load one byte of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(1, map(*srca* + *srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *srcb*). If no error occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eload.b** *srca,rdst*

Opcode: E0

Operation: Load one byte of data from memory into a register, extending the sign.

Operands used: *srca*, mem(1, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then a single byte at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instructions:	**eload.h**	(*srca*)[*srcb*:*m*],*rdst*	
Opcodes:	**eload.h**	(*srca*)[*srcb*:1],*rdst*	E5
	eload.h	(*srca*)[*srcb*:2],*rdst*	E6

Operation: Load two bytes of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(2, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no error occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eload.h** *srca,rdst*

Opcode: E4

Operation: Load two bytes of data from memory into a register, extending the sign.

Operands used: *srca*, mem(2, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then two bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instructions: **eload.w** *(srca)[srcb:m],rdst*

Opcodes:

eload.w	*(srca)[srcb*:1],*rdst*	E9
eload.w	*(srca)[srcb*:4],*rdst*	EA

Operation: Load four bytes of data from memory into a register, extending the sign.

Operands used: *srca*, *srcb*, mem(4, map(*srca* + *m***srcb*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no error occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

Instruction: **eload.w** *srca,rdst*

Opcode: E8

Operation: Load four bytes of data from memory into a register, extending the sign.

Operands used: *srca*, mem(4, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **eload**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no error occurs during the mapping process, then four bytes at the resultant physical address will be read from memory, sign-extended to 64 bits, and stored in *rdst*. If one of the special **eload** error conditions occurs, then no trap will be taken and zero will be stored in *rdst*. **ELF[RC]** will be set to one if any error or trap occurs, and to zero if no error or trap occurs.

5.4 Store Instructions

The following instruction descriptions are for all the "normal" **store** instructions. Several special types of **store** instructions are covered in a later section.

Instruction: **store.b** *srcc*,(*srca*)[*srcb*:1]

Opcode: D3

Operation: Store one byte of data from a register into memory.

Operands used: *srcc*, *srca*, *srcb*

Results stored: mem(1, map(*srca* + *srcb*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *srcb*). If no trap occurs during the mapping process, then a single byte from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instruction: **store.b** *srcc,srca*

Opcode: C3

Operation: Store one byte of data from a register into memory.

Operands used: *srcc*, *srca*

Results stored: mem(1, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then a single byte from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instructions: **store.h** *srcc*,(*srca*)[*srcb*:*m*]

Opcodes: **store.h** *srcc*,(*srca*)[*srcb*:1] D7
store.h *srcc*,(*srca*)[*srcb*:2] E7

Operation: Store two bytes of data from a register into memory.

Operands used: *srcc*, *srca*, *srcb*

Results stored: mem(2, *srca* + map(*m***srcb*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then two bytes from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instruction: **store.h** *srcc,srca*

Opcode: C7

Operation: Store two bytes of data from a register into memory.

Operands used: *srcc*, *srca*

Results stored: mem(2, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then two bytes from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instructions:	**store.w**	*srcc*,(*srca*)[*srcb*:*m*]	
Opcodes:	**store.w**	*srcc*,(*srca*)[*srcb*:1]	DB
	store.w	*srcc*,(*srca*)[*srcb*:4]	EB

Operation: Store four bytes of data from a register into memory.

Operands used: *srcc*, *srca*, *srcb*

Results stored: mem(4, *srca* + *m***srcb*)

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then four bytes from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instruction: **store.w** *srcc,srca*

Opcode: CB

Operation: Store four bytes of data from a register into memory.

Operands used: *srcc*, *srca*

Results stored: mem(4, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then four bytes from the least significant part of *srcc* will be written to memory at the resultant physical address.

Instructions:	**store.l**	*srcc*,(*srca*)[*srcb*:*m*]	
Opcodes:	**store.l**	*srcc*,(*srca*)[*srcb*:1]	DF
	store.l	*srcc*,(*srca*)[*srcb*:8]	EF

Operation: Store eight bytes of data from a register into memory.

Operands used: *srcc*, *srca*, *srcb*

Results stored: mem(8, map(*srca* + *m***srcb*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca* + *m***srcb*). If no trap occurs during the mapping process, then all eight bytes of *srcc* will be written to memory at the resultant physical address.

Instruction: **store.l** *srcc,srca*

Opcode: CF

Operation: Store eight bytes of data from a register into memory.

Operands used: *srcc*, *srca*

Results stored: mem(8, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then all eight bytes of *srcc* will be written to memory at the resultant physical address.

5.5 Special Load/Store Instructions

The instructions described in this section are more specialized than the previous **load** and **store** instructions.

Instruction: **swat** *srca,srcc*

Opcode: FC

Operation: Swap atomically eight bytes of data between memory and a register.

Operands used: *srca*, *srcc*, mem(8, map(*srca*))

Results stored: *rdst* (same register as *srcc*), **ELF[RC]**, mem(8, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and exchanged with *srcc*. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

swat is the only instruction that is guaranteed to atomically read and write a memory location. The addressed location should be on a shared page since the operation is not guaranteed to be atomic between processors if the page is not shared. **swat** is also the only load/store instruction that requires both read and write access, and that requires both the **User Mode Load** and **User Mode Store** bits of the **Processor Status** register to have the same value.

Instruction: **loadcpu** *srca,rdst*

Opcode: 8D

Operation: Load eight bytes of data from memory into a register, transforming the address to be unique to one CPU in a multiprocessor.

Operands used: *srca*, mem(8, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*), which for this instruction includes an implementation-dependent transformation to make the address unique among all of the CPUs in a multiprocessor [5-6]. If no trap occurs during the mapping process, then eight bytes at the resultant physical address will be read from memory and stored in *rdst.* **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Instruction: **storecpu** *srcc,srca*

Opcode: 8F

Operation: Store eight bytes of data from a register into memory, transforming the address to be unique to one CPU in a multiprocessor.

Operands used: *srcc*, *srca*

Results stored: mem(8, map(*srca*))

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **store**-type (see Table 5-1)

Description:

A memory address is calculated as map(*srca*), which for this instruction includes an implementation-dependent transformation to make the address unique among all of the CPUs in a multiprocessor [5-6]. If no trap occurs during the mapping process, then all eight bytes of *srcc* will be written to memory at the resultant physical address.

Instruction: **ldecc** *srca,rdst*

Opcode: DC

Operation: Load ECC bits from memory.

Operands used: *srca*, mem(8, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1), but without ECC errors

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then the cache will be consulted to see if it contains the addressed data. If it does, then the instruction completes as if it were a **load.l** (loading whatever data is in the data cache). If it does not (if there is a cache miss), then the ECC bits for the line containing the resultant physical address will be read from memory and stored in the cache. The instruction will then complete as if there were no cache miss. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs. See Appendix E for details of how the ECC bits are stored and accessed.

Note that using this instruction on shared data will produce unpredictable results (if some other processor has the data). For this reason, **ldecc** should be used with great care as a **supervisor** mode reference (since **supervisor** mode references are always shared).

For diagnostics to properly use this instruction, the ECC bits should be verified a line at a time, and the desired line should be flushed from the data cache before the first **ldecc** instruction for that line.

Instruction: **ldnecc** *srca,rdst*

Opcode: BC

Operation: Load uncorrected data from memory.

Operands used: *srca*, mem(8, map(*srca*))

Results stored: *rdst*, **ELF[RC]**

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **load**-type (see Table 5-1), but without ECC errors

Description:

A memory address is calculated as map(*srca*). If no trap occurs during the mapping process, then the cache will be consulted to see if it contains the addressed data. If it does, then the instruction completes as if it were a **load.l**. If it does not (if there is a cache miss), then the line containing the resultant physical address will be read from memory and stored in the cache. However, the data will be uncorrected, i.e., the error correction possible using the ECC bits in memory will not be in effect. The instruction will then complete as if there were no cache miss. **ELF[RC]** will be set to one if any trap occurs, and to zero if no trap occurs.

Note that using this instruction on shared data will produce unpredictable results (if some other processor has the data). For this reason, **ldnecc** should be used with great care as a **supervisor** mode reference (since **supervisor** mode references are always shared).

For diagnostics to properly use this instruction, the ECC bits should be verified a line at a time, and the desired line should be flushed from the data cache before the first **ldnecc** instruction for that line.

5.6 ELF Flag Instructions

These instructions are used to test **ELF** flags, trapping if one is set, and to read and write the **ELF** flags for context switching.

Instruction:	**echk**	*rdst*
Opcode:	F3	
Operation:	Test an **ELF** bit and trap if it is set.	
Operands used:	**ELF[RC]**	
Results stored:	none	
Legal in:	User or Supervisor mode	
Functional unit:	Load/Store	
Exceptions:	echk	

Description:

If the **ELF** flag whose number is given by the instruction's **RC** field is set, then an echk trap occurs.

Instruction: **relf** *rdst*

Opcode: C2

Operation: Read early load fault bits into a register

Operands used: **ELF<63..0>**

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: none

Description:

The early load fault bits, **ELF[RC]<63..0>**, corresponding to registers **r63** through **r0**, are written to *rdst*.

Instruction:	**welf**	*srca,srcb*
Opcode:	D2	
Operation:	Write early load fault bits.	
Operands used:	*srca, srcb*	
Results stored:	**ELF<63..32>** or **ELF<31..0>**	
Legal in:	User or Supervisor mode	
Functional unit:	Load/Store	
Exceptions:	none	

Description:

If the low bit of *srcb* is a zero, then the early load fault bits, **ELF[RC]<31..0>**, corresponding to registers **r31** through **r0**, are written from the low-order half of *srca*. If the low bit of *srcb* is a one, then **ELF[RC]<63..32>**, corresponding to registers **r63** through **r32**, are written from the low-order half of *srca*.

5.7 Data Watchpoint

A **data watchpoint** is a breakpoint on load/store reference addresses. Data watchpoints are stored in the **Data Watchpoint Table.** A Data Watchpoint Table entry specifies a *virtual* address to be compared with addresses output by the Load/Store functional unit. (See Table 5-1, which lists the instruction classes that can take data watchpoint traps.) If enabled, an address match can cause a data watchpoint trap. The low three bits of the address are not compared, so care must be taken when setting up a data watchpoint for an 8-, 16-, or 32-bit quantity: spurious data watchpoint traps could occur on accesses to other parts of the same 64-bit word.

The format of a Data Watchpoint Table entry is shown in Figure 5-1.

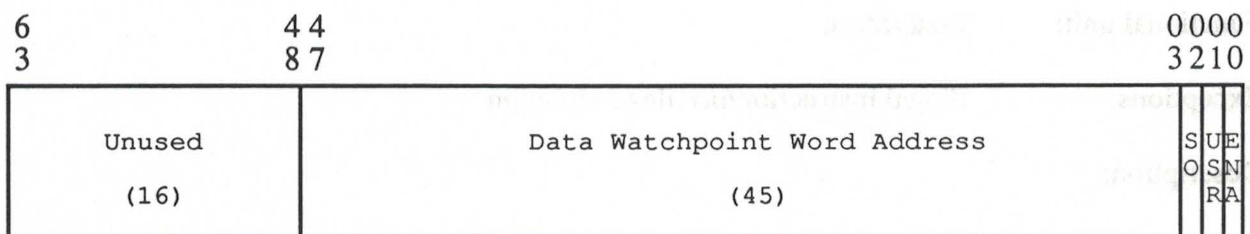


Figure 5-1. Data Watchpoint Table Entry Format

The **Data Watchpoint Word Address** field specifies bits <47..3> of the address to be compared [5-3]. The **ENA** bit must be set to enable this **Data Watchpoint Table** entry; if this bit is clear, this data watchpoint will not trap or cause the clocks to be stopped. The **USR** bit controls whether address matches are detected for user or for supervisor references. If the **USR** bit is set, then only user mode references will be watched; if the **USR** bit is clear, then only supervisor mode references will be watched. Data watchpoint traps will *not* occur in **Trap State**. The **SO** (store only) bit, if set, disables data watchpoint trapping except for **store, swat,** and **zcl** instructions. If the **SO** bit is clear, then both **loads** and **stores** can take data watchpoint traps.

The Console can enable data watchpoints to stop the processor's clocks and interrupt the Front-End Processor. See Appendix G for details of the Front-End Processor. Even though data watchpoint traps cannot be taken in **Trap State**, data watchpoint clocks stops can happen in **Trap State**.

A few instructions are treated specially. The **exts** and **slstrpd** instructions will never cause data watchpoint traps, though they can cause clock stops. The **dflush** and **pcl** instructions will never cause data watchpoint traps or clock stops. The **zcl** instruction compares only the cache line address (ignoring more low-order bits than other compares).

The size of the Data Watchpoint Table is implementation-dependent [5-4].

The contents of the Data Watchpoint Table cannot be read.

Instruction:	**wdwp**	*srca,srcb*
Opcode:	FB	
Operation:	Write a Data Watchpoint Table entry.	
Operands used:	*srca*, *srcb*	
Results stored:	Data Watchpoint Table[*srcb*]	
Legal in:	Supervisor mode only	
Functional unit:	Load/Store	
Exceptions:	Illegal instruction/privilege violation	

Description:

If the processor is in user mode, then an illegal instruction/privilege violation trap will occur. If the processor is in supervisor mode then the Data Watchpoint Table entry addressed by the low-order bits of *srcb* will be written from the *srca* operand. The format of a Data Watchpoint Table entry is shown in Figure 5-1.

CHAPTER 6. Integer Instructions

The integer instructions encompass a great variety of operations, including boolean, shift, compare, bit count and bit reverse, data moving, and signed and unsigned arithmetic operations. Operands are usually 64-bit integers, although certain integer operations work on 8-, 16-, 32-, 33-, and 53-bit quantities.

6.1 Integer Arithmetic Instructions

Integer operations use two's complement arithmetic. Whether the most significant bit is considered a sign bit is relevant only when comparisons are involved, or when the result has more bits of significance than the operands, such as in multiply. In those cases, both signed and unsigned versions of the operation are provided. All operations are performed in the integer functional unit except for signed and unsigned integer multiply and divide (which are done in the floating-point multiply and floating-point divide/square root units), and **move.d** (which is done in the floating-point add unit).

Certain common operations, such as **inc** (incrementing), **dec** (decrementing), and **neg** (negating), are not provided in hardware because they can be obtained using existing operations with constant operands at no cost in time or space. (See Chapter 3.) The assembler, however, performs these translations automatically, making it appear that these operations exist. Refer to *The K-1 Assembly Language Reference Manual* for more details.

Instruction:	**add**	*srca,srcb,rdst*
Opcode:	5A	
Operation:	Add two 64-bit integers.	
Operands used:	*srca, srcb*	
Results stored:	*rdst*	
Legal in:	User or Supervisor mode	
Functional unit:	Integer	
Exceptions:	none	

Description:

The low 64 bits of the sum of two 64-bit integer operands, *srca* and *srcb*, are stored in *rdst.*

Instruction: **addc** **f0**,*srca*,*srcb*,*rdst*

Opcode: 58

Operation: Add two 64-bit integers with carry in and out.

Operands used: **f0**, *srca*, *srcb*

Results stored: *rdst*, **f0**

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The sum, *srca* + *srcb* + 1 - **f0**, of two 64-bit integer operands, *srca* and *srcb*, and the complement of flag **f0** is computed and the low 64 bits of the result are stored in *rdst*. In addition, the complement of the carry out of the sum is returned to **f0**. This instruction is intended for use in multiple precision arithmetic.

Instruction: **addt** *srca,srcb,rdst*

Opcode: 59

Operation: Add two 64-bit signed integers with trap on overflow.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Integer overflow

Description:

The low 64 bits of the sum of two 64-bit integer operands, *srca* and *srcb*, are stored in *rdst*. An integer overflow trap will occur if the result overflows. Note that such a trap will occur regardless of the state of the **Arithmetic Trap Enables** in the **Processor Status** register.

This instruction can be used to implement range checking for languages such as Pascal and Ada.

Instruction: **sub** *srca,srcb,rdst*

Opcode: 4A

Operation: Subtract two 64-bit integers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The difference, *srcb* - *srca*, of two 64-bit integer operands, *srca* and *srcb*, is computed and the low 64 bits of the result are stored in *rdst*. Note that the order of the operands is backwards from what might be expected.

Instruction:	**subb**	**f0**,*srca*,*srcb*,*rdst*
Opcode:	48	
Operation:	Subtract two 64-bit integers with borrow in and out.	
Operands used:	**f0**, *srca*, *srcb*	
Results stored:	*rdst*, **f0**	
Legal in:	User or Supervisor mode	
Functional unit:	Integer	
Exceptions:	none	

Description:

The difference, *srcb* - *srca* - **f0**, of two 64-bit integer operands, *srca* and *srcb*, and a borrow from flag **f0** is computed and the low 64 bits of the result are stored in *rdst*. In addition, the borrow out of the sum is returned to **f0**. This instruction is intended for use in multiple precision arithmetic. It can also be used to maintain a decrementing loop counter, and will set flag **f0** when the counter is decremented from a positive number or zero to a negative number.

Instruction:	**subt**	*srca,srcb,rdst*
Opcode:	49	
Operation:	Subtract two 64-bit signed integers with trap on overflow.	
Operands used:	*srca, srcb*	
Results stored:	*rdst*	
Legal in:	User or Supervisor mode	
Functional unit:	Integer	
Exceptions:	Integer overflow	

Description:

The difference, *srcb* - *srca*, of two 64-bit signed integer operands, *srca* and *srcb*, is computed and the low 64 bits of the result are stored in *rdst*. An integer overflow trap will occur if the result overflows. Note that such a trap will occur regardless of the state of the **Arithmetic Trap Enables** in the **Processor Status** register.

This instruction can be used to implement range checking for languages such as Pascal and Ada.

Instruction: **multlss** *srca,srcb,rdst*

Opcode: A5

Operation: Multiply two 53-bit signed integers and return the low part of the result.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The least significant 64 bits of the product of two 53-bit signed integer operands, *srca* and *srcb*, is computed and stored in *rdst*. The sign bit of the operands is assumed to be in bit position 52, and bits 63 through 53 are ignored. A 106-bit signed product is calculated and the low 64 bits are stored as the result. The **multhss** instruction can be used to obtain the high-order portion of the product.

Instruction: **multhss** *srca,srcb,rdst*

Opcode: A7

Operation: Multiply two 53-bit signed integers and return the high part of the result.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The most significant 42 bits of the product of two 53-bit signed integer operands, *srca* and *srcb*, is computed and stored in *rdst*. The sign bit of the operands is assumed to be in bit position 52, and bits 63 through 53 are ignored. A 106-bit signed product is calculated and the high 42 bits are sign-extended to 64 bits and stored as the result. Note that the sign bit of the result gives the true sign of the product. The **multlss** instruction can be used to obtain the low-order portion of the product.

Instruction: **multluu** *srca,srcb,rdst*

Opcode: A9

Operation: Multiply two 53-bit unsigned integers and return the low part of the result.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The least significant 64 bits of the product of two 53-bit unsigned integer operands, *srca* and *srcb*, is computed and stored in *rdst*. A 106-bit product is calculated and the low 64 bits are stored as the result. The **multhuu** instruction can be used to obtain the high-order portion of the product.

Instruction: **multhuu** *srca,srcb,rdst*

Opcode: AB

Operation: Multiply two 53-bit unsigned integers and return the high part of the result.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The most significant 42 bits of the product of two 53-bit unsigned integer operands, *srca* and *srcb*, is computed and stored in *rdst*. A 106-bit unsigned product is calculated and the high 42 bits are zero-extended to 64 bits and stored as the result. The **multluu** instruction can be used to obtain the low-order portion of the product.

Instruction: **multlus** *srca,srcb,rdst*

Opcode: AD

Operation: Multiply a 53-bit unsigned integer and a 53-bit signed integer and return the low part of the result.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The least significant 64 bits of the product of a 53-bit unsigned integer operand, *srca*, and a 53-bit signed integer operand, *srcb*, is computed and stored in *rdst*. The sign bit of *srcb* is assumed to be in bit position 52, and bits 63 through 53 are ignored. A 106-bit signed product is calculated and the low 64 bits are stored as the result. The **multhus** instruction can be used to obtain the high-order portion of the product.

Instruction: **multhus** *srca,srcb,rdst*

Opcode: AF

Operation: Multiply a 53-bit unsigned integer and a 53-bit signed integer and return the high part of the result.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Multiply

Exceptions: none

Description:

The most significant 42 bits of the product of a 53-bit unsigned integer operand, *srca*, and a 53-bit signed integer operand, *srcb*, is computed and stored in *rdst*. The sign bit of *srcb* is assumed to be in bit position 52, and bits 63 through 53 are ignored. A 106-bit signed product is calculated and the high 42 bits are sign-extended to 64 bits and stored as the result. Note that the sign bit of the result gives the true sign of the product. The **multlus** instruction can be used to obtain the low-order portion of the product.

Instruction: **divsst** *srca,srcb,rdst*

Opcode: A4

Operation: Divide a 33-bit signed integer by a 33-bit signed integer and return a 64-bit signed integer quotient using **truncate** rounding mode.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Divide

Exceptions: Integer divide

Description:

The truncated 64-bit signed integer quotient, *srca* ÷ *srcb*, of the two 33-bit signed integer operands, *srca* and *srcb*, is computed and stored in *rdst*. If *srcb* is zero (in its low 33 bits), the result will be zero and, if enabled by the **Integer Divide Trap Enable** bit in the **Processor Status** register, an integer divide trap will occur.

Instruction: **divssr** *srca,srcb,rdst*

Opcode: A6

Operation: Divide a 33-bit signed integer by a 33-bit signed integer and return a 64-bit signed integer quotient.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Divide

Exceptions: Integer divide

Description:

The rounded 64-bit signed integer quotient, *srca* ÷ *srcb*, of the two 33-bit signed integer operands, *srca* and *srcb*, is computed and stored in *rdst*. Rounding is performed according to the **Rounding Mode<1..0>** field of the **Processor Status** register. If *srcb* is zero (in its low 33 bits), the result will be zero and, if enabled by the **Integer Divide Trap Enable** bit in the **Processor Status** register, an integer divide trap will occur.

6.2 Integer Compare Instructions

Only a very small number of compare instructions are necessary to provide a complete set of operations because the order of operands may be easily switched, and because the complements of flags may be tested as easily as the true version. (See Chapter 3). Therefore, there are only three types of integer compare instructions. One type tests for **equal** and may be used with either signed or unsigned operands. The other two types test for **greater than**, one with signed operands and the other with unsigned operands.

Each type of comparison is available in four precisions for operating on 1-, 2-, 4- or 8-byte operands. For precisions less than 8 bytes, the least significant part of the 64-bit register operands contain the data to be compared.

Instruction: **cmpeq.b** *srca,srcb,fdst*

Opcode: 27

Operation: Compare two 8-bit integers for equality.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 8-bit operands, *srca* and *srcb*, are compared. If they are equal, then the flag *fdst* is set. If they are different, then the flag is cleared.

Instruction: **cmpeq.h** *srca,srcb,fdst*

Opcode: 26

Operation: Compare two 16-bit integers for equality.

Operands used: *srca, srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 16-bit operands, *srca* and *srcb*, are compared. If they are equal, then the flag *fdst* is set. If they are different, then the flag is cleared.

Instruction: **cmpeq.w** *srca,srcb,fdst*

Opcode: 25

Operation: Compare two 32-bit integers for equality.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 32-bit operands, *srca* and *srcb*, are compared. If they are equal, then the flag *fdst* is set. If they are different, then the flag is cleared.

Instruction: **cmpeq.l** *srca,srcb,fdst*

Opcode: 24

Operation: Compare two 64-bit integers for equality.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 64-bit operands, *srca* and *srcb*, are compared. If they are equal, then the flag *fdst* is set. If they are different, then the flag is cleared.

Instruction: **cmpgt.b** *srca,srcb,fdst*

Opcode: 23

Operation: Compare two 8-bit signed integers.

Operands used: *srca, srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two signed 8-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpgt.h** *srca,srcb,fdst*

Opcode: 22

Operation: Compare two 16-bit signed integers.

Operands used: *srca, srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two signed 16-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpgt.w** *srca,srcb,fdst*

Opcode: 21

Operation: Compare two 32-bit signed integers.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two signed 32-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpgt.l** *srca,srcb,fdst*

Opcode: 20

Operation: Compare two 64-bit signed integers.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two signed 64-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpugt.b** *srca,srcb,fdst*

Opcode: 2B

Operation: Compare two 8-bit unsigned integers.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two unsigned 8-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpugt.h** *srca,srcb,fdst*

Opcode: 2A

Operation: Compare two 16-bit unsigned integers.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two unsigned 16-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction:	**cmpugt.w** *srca,srcb,fdst*
Opcode:	29
Operation:	Compare two 32-bit unsigned integers.
Operands used:	*srca*, *srcb*
Results stored:	*fdst*
Legal in:	User or Supervisor mode
Functional unit:	Integer
Exceptions:	none

Description:

The two unsigned 32-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

Instruction: **cmpugt.l** *srca,srcb,fdst*

Opcode: 28

Operation: Compare two 64-bit unsigned integers.

Operands used: *srca*, *srcb*

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two unsigned 64-bit operands, *srca* and *srcb*, are compared. If *srca* is greater than *srcb*, then the flag *fdst* is set, otherwise the flag is cleared.

6.3 Data Moving Instructions

The data moving instructions move quantities of any data type. One class of these instructions can move one of two operands based on the value of a flag.

Instruction: **move** *srca,rdst*

Opcode: 4B

Operation: Move a 64-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 64-bit integer, *srca*, is stored unchanged in *rdst*.

Instruction: **move.d** *srca,rdst*

Opcode: 93

Operation: Move a 64-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Floating-point Add

Exceptions: none

Description:

The 64-bit integer, *srca*, is stored unchanged in *rdst*. This is the same operation as **move** but is performed in the floating-point add functional unit, which can run in parallel with the integer unit, thus providing more bandwidth for copying data between registers. The timing of this instruction is different than that of **move**. See Appendix C for details.

Instruction: **zext.b** *srca,rdst*

Opcode: 2C

Operation: Zero-extend an 8-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 8-bit integer, *srca*, is zero-extended to 64 bits and stored in *rdst*.

Instruction: **zext.h** *srca,rdst*

Opcode: 2D

Operation: Zero-extend a 16-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 16-bit integer, *srca*, is zero-extended to 64 bits and stored in *rdst*.

Instruction: **zext.w** *srca,rdst*

Opcode: 2E

Operation: Zero-extend a 32-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 32-bit integer, *srca*, is zero-extended to 64 bits and stored in *rdst*.

Instruction: **sext.b** *srca,rdst*

Opcode: 4C

Operation: Sign-extend an 8-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 8-bit integer, *srca*, is sign-extended to 64 bits and stored in *rdst*.

Instruction: **sext.h** *srca,rdst*

Opcode: 4D

Operation: Sign-extend a 16-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 16-bit integer, *srca*, is sign-extended to 64 bits and stored in *rdst*.

Instruction: **sext.w** *srca,rdst*

Opcode: 4E

Operation: Sign-extend a 32-bit integer to 64 bits.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The 32-bit integer, *srca*, is sign-extended to 64 bits and stored in *rdst*.

Instructions:	**sel**	*fsrc,srca,srcb,rdst*	
Opcodes:	**sel**	**f0**,*srca,srcb,rdst*	40
	sel	**f1**,*srca,srcb,rdst*	41
	sel	**f2**,*srca,srcb,rdst*	42
	sel	**f3**,*srca,srcb,rdst*	43
	sel	**f4**,*srca,srcb,rdst*	44
	sel	**f5**,*srca,srcb,rdst*	45
	sel	**f6**,*srca,srcb,rdst*	46

Operation: Move one of two 64-bit integers.

Operands used: *fsrc*, *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

One of the 64-bit integers, *srca* or *srcb*, is stored unchanged in *rdst*. If *fsrc* is a zero, then *srca* is moved, otherwise *srcb* is moved.

6.4 Boolean Instructions

The boolean instructions perform **and**, **or**, and **xor** operations on 64-bit operands. Certain operations, such as **not** and the **or** of the second operand with the complement of the first operand, are not provided in hardware because they can be obtained using existing operations with constant operands or with the operands reversed at no loss in time or space. (See Chapter 3.) The assembler, however, performs these translations automatically, making it appear that all operations exist. Refer to *The K-1 Assembly Language Reference Manual* for more details.

Instruction: **and** *srca,srcb,rdst*

Opcode: 63

Operation: Perform the bitwise **and** of two 64-bit integers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **and** of two 64-bit integer operands, *srca* and *srcb*, is stored in *rdst*.

Instruction: **andtc** *srca,srcb,rdst*

Opcode: 69

Operation: Perform the bitwise **and** of two 64-bit integers, complementing one first.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **and** of the 64-bit integer operand, *srca*, and the complement of the 64-bit integer operand, *srcb*, is stored in *rdst*.

Instruction: **andcc** *srca,srcb,rdst*

Opcode: 65

Operation: Perform the bitwise **and** of the complements of two 64-bit integers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **and** of the complement of the 64-bit integer operand, *srca*, and the complement of the 64-bit integer operand, *srcb*, is stored in *rdst*.

Instruction: **or** *srca,srcb,rdst*

Opcode: 6A

Operation: Perform the bitwise **or** of two 64-bit integers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **or** of two 64-bit integer operands, *srca* and *srcb*, is stored in *rdst*.

Instruction: **ortc** *srca,srcb,rdst*

Opcode: 6F

Operation: Perform the bitwise **or** of two 64-bit integers, complementing one first.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **or** of the 64-bit integer operand, *srca*, and the complement of the 64-bit integer operand, *srcb*, is stored in *rdst*.

Instruction: **orcc** *srca,srcb,rdst*

Opcode: 6C

Operation: Perform the bitwise **or** of the complements of two 64-bit integers.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **or** of the complement of the 64-bit integer operand, *srca*, and the complement of the 64-bit integer operand, *srcb*, is stored in *rdst*.

Instruction: **xor** *srca,srcb,rdst*

Opcode: 68

Operation: Perform the bitwise **xor** of two 64-bit integers.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bitwise **exclusive-or** of two 64-bit integer operands, *srca* and *srcb*, is stored in *rdst*.

Instruction: **xnor** *srca,srcb,rdst*

Opcode: 67

Operation: Perform the bitwise **xor** of two 64-bit integers, and complement the result.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The complement of the bitwise **exclusive-or** of the 64-bit integer operand, *srca*, and the 64-bit integer operand, *srcb*, is stored in *rdst*.

6.5 Shift Instructions

This section describes instructions that shift and rotate 64-bit integers. In addition, special shift instructions, **dshfl** and **dshfr**, which return 64 bits of a shifted 128-bit operand (composed of two independent 64-bit operands), are described.

Instruction: **rot** *srca,srcb,rdst*

Opcode: 66

Operation: Rotate a 64-bit integer left or right.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The signed integer operand *srca* gives the number of places to rotate the 64-bit integer operand *srcb*. The result is stored in *rdst*. If *srca* is positive, then *srcb* is rotated left by *srca* places. If *srca* is negative, then *srcb* is rotated right by -*srca* places. Note that because two's complement arithmetic is used, and because the operand length (64 bits) is a power of two, only the low six bits of *srca* have any effect on the operation. Therefore, any value of *srca* is legal and the correct rotation will occur.

Instruction: **shf** *srca,srcb,rdst*

Opcode: 60

Operation: Logically shift a 64-bit integer left or right.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The signed integer operand *srca* gives the number of places to shift the 64-bit integer operand *srcb*. The result is stored in *rdst*. If *srca* is positive, then *srcb* is shifted left by *srca* places: zero bits are entered at the least significant end, and bits shifted off the most significant end are lost. If *srca* is negative, then *srcb* is shifted right by -*srca* places: zero bits are entered at the most significant end, and bits shifted off the least significant end are lost. Note that if the shift count is greater than 63 or less than -63, the result will be zero.

Instruction: **ashf** *srca,srcb,rdst*

Opcode: 61

Operation: Arithmetically shift a 64-bit integer left or right.

Operands used: *srca*, *srcb*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The signed integer operand *srca* gives the number of places to shift the 64-bit integer operand *srcb*. The result is stored in *rdst*. If *srca* is positive, then *srcb* is shifted left by *srca* places: zero bits are entered at the least significant end, and bits shifted off the most significant end are lost. If *srca* is negative then *srcb* is shifted right by -*srca* places: the sign bit is repeatedly entered at bit 63, and bits shifted off the least significant end are lost. Note that if the shift count is 64 or greater the result will be zero. But if the shift count is equal to or more negative than -63, then the result will be zero or minus one depending on whether *srcb* was positive or negative, respectively.

Instruction: **dshfl** *srca,srcb,srcc*

Opcode: 6D

Operation: Shift two 64-bit integers left 0 to 127 places.

Operands used: *srca*, *srcb*, *srcc*

Results stored: *rdst* (same register as *srcc*)

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 64-bit integer operands, *srcc* and *srcb* are concatenated to form a 128-bit operand; *srcc* contributes the most significant bits, and *srcb* the least significant bits. The combined entity is shifted left a number of places determined by the low seven bits of the integer operand *srca*. The high 64 bits of the result are stored in *rdst*. Zero bits are entered at the least significant end and bits shifted off the most significant end are lost. Note that *srcc* may not be a constant, and that *srcc* and *rdst* both refer to the same register, **R[RC]**.

Instruction: **dshfr** *srca,srcb,srcc*

Opcode: 6E

Operation: Shift two 64-bit integers right 1 to 128 places.

Operands used: *srca*, *srcb*, *srcc*

Results stored: *rdst* (same register as *srcc*)

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The two 64-bit integer operands, *srcb* and *srcc* are concatenated to form a 128-bit operand; *srcb* contributes the most significant bits and *srcc* the least significant bits. The combined entity is shifted right a number of places determined by the integer operand *srca*. This operand is intended to be in the range -1 to -128 and will result in right shifts of from 1 to 128 places. The low 64 bits of the result are stored in *rdst*. Zero bits are entered at the most significant end and bits shifted off the least significant end are lost. Note that *srcc* may not be a constant, and that *srcc* and *rdst* both refer to the same register, **R[RC]**.

In more detail, the shift count is computed by replacing bits 7 through 63 of *srca* with all ones and then taking the two's complement of the result. For example, if *srca* contains zero, then the shift count will be 128.

6.6 Bit Count and Reverse Instructions

The instructions in this category are used to reverse the bits in a 64-bit integer and to perform certain bit counting functions.

Instruction: **bitrev** *srca,rdst*

Opcode: 5C

Operation: Reverse the bits of a 64-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The bits of the 64-bit integer, *srca*, are reversed and stored in *rdst*. In other words, bit position *i* is sent to bit position *63 - i*.

Instruction: **bitcnt.w** *srca,rdst*

Opcode: 5D

Operation: Count the number of one bits in a 32-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The count of the number of bits in the 32-bit integer, *srca*, that are set to one is stored in *rdst*.

Instruction: **lzcnt** *srca,rdst*

Opcode: 5E

Operation: Count the number of leading zero bits in a 64-bit integer.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The count of the number of leading (most significant) bits of the 64-bit integer, *srca*, that are set to zero is stored in *rdst*. A zero input produces a result of 64.

6.7 Flag Instructions

Flags can be written by integer and floating-point compare instructions and by the **ad-dc** and **subb** instructions (all described elsewhere). In addition, they can be operated on directly with the **boof** instructions. The assembler supports a number of different mnemonics for **boof**, including the use of flags as sources and destinations with the usual boolean mnemonics (**and**, **or**, **andcc**, etc.). Refer to *The K-1 Assembly Language Reference Manual* for more details.

Instruction: **boof** *srca,fdst*

Opcode: 4F

Operation: Perform a boolean operation on flags, storing the result in a flag.

Operands used: *srca*, **F[*srca*<2..0>], F[*srca*<5..3>]**

Results stored: *fdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The low 9 bits of *srca* specify a boolean function and two flags on which to operate. The result of the operation is stored in flag *fdst*. Figure 6-1 shows the format of the *srca* operand, and Table 6-1 gives the encoding of the boolean function. In most applications, the *srca* operand will be a short constant, not a register.

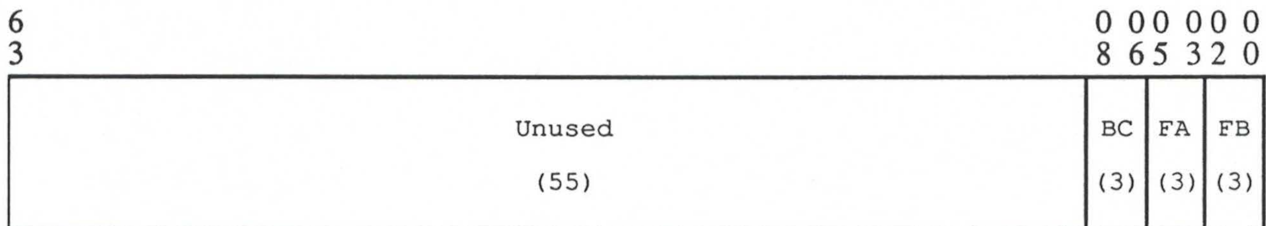


Figure 6-1. boof *srca* Argument

Table 6-1. boof Code (BC) Decoding

boof code (BC)	Operation
0	FA and (not FB)
1	(not FA) and (not FB)
2	FA xor FB
3	(not FA) or (not FB)
4	FA and FB
5	not (FA xor FB)
6	FA or FB
7	FA or (not FB)

Instruction: **boof** *srca,rdst*

Opcode: 2F

Operation: Perform a boolean operation on flags, storing the result in a register.

Operands used: *srca*, F[*srca*<2..0>], F[*srca*<5..3>]

Results stored: *rdst*

Legal in: User or Supervisor Mode

Functional Unit: Integer

Exceptions: none

Description:

The low 9 bits of *srca* specify a boolean function and two flags on which to operate. The single bit result of the operation is zero-extended to 64 bits and stored in *rdst*. Figure 6-1 shows the format of the *srca* operand, and Table 6-1 gives the encoding of the boolean function.

6.8 Check Instructions

The check instructions are used to determine if a 64-bit signed or unsigned integer can fit in a lesser precision. They do not store any results. Instead, if the operand will not fit in the smaller precision, a check trap occurs.

Instruction: **chk.b** *srca*

Opcode: 50

Operation: Check if a 64-bit signed integer can fit in one byte.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 56 bits of the signed 64-bit integer, *srca*, are examined. If they all have the same value as bit 7, the sign bit of the low byte, then nothing happens. If any bit is different, then a check trap occurs.

Instruction: **chk.h** *srca*

Opcode: 51

Operation: Check if a 64-bit signed integer can fit in two bytes.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 48 bits of the signed 64-bit integer, *srca*, are examined. If they all have the same value as bit 15, the sign bit of the low 2-byte group, then nothing happens. If any bit is different, then a check trap occurs.

Instruction: **chk.w** *srca*

Opcode: 52

Operation: Check if a 64-bit signed integer can fit in four bytes.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 32 bits of the signed 64-bit integer, *srca*, are examined. If they all have the same value as bit 31, the sign bit of the low 4-byte group, then nothing happens. If any bit is different, then a check trap occurs.

Instruction: **chku.b** *srca*

Opcode: 53

Operation: Check if a 64-bit unsigned integer can fit in one byte.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 56 bits of the unsigned 64-bit integer, *srca*, are examined. If they are all zero, then nothing happens. If any bit is different from zero, then a check trap occurs.

Instruction: **chku.h** *srca*

Opcode: 54

Operation: Check if a 64-bit unsigned integer can fit in two bytes.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 48 bits of the unsigned 64-bit integer, *srca*, are examined. If they are all zero, then nothing happens. If any bit is different from zero, then a check trap occurs.

Instruction: **chku.w** *srca*

Opcode: 55

Operation: Check if a 64-bit unsigned integer can fit in four bytes.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: Check

Description:

The high 32 bits of the unsigned 64-bit integer, *srca*, are examined. If they are all zero, then nothing happens. If any bit is different from zero, then a check trap occurs.

CHAPTER 7. Transfer of Control Instructions

The transfer of control instructions are used to change the value of the **Program Counter**, and include the subroutine call instructions and the absolute jump instructions. This chapter does not describe the assembly language **br** instruction. Rather than being an actual instruction, **br** indicates the use of **PC-relative branch** format, which does not require an opcode and is permitted only in the I1 half of an instruction word. All transfer of control instructions (other than **br**) must use the 64-bit **absolute branch** or **register branch** instruction formats or an illegal instruction/privilege violation trap will occur. (See Chapter 3 for more details on instruction formats.)

The subroutine call instructions move a value (the return **PC**) to a register and then branch. The return **PC** will not be stored if the subroutine call instruction is not actually executed (i.e., if it is conditionally disabled). A subroutine returns by branching to an address in a register (the linkage register used by the subroutine call instruction). The return **PC** is a function of the delayed branch control field of the subroutine call instruction, and will be that of the **call** plus a small offset, as given in Table 3-2. This is true even if the **call** is in a delay slot and regardless of what instructions are in the **call**'s delay slots; the return **PC** calculation is always based solely on the address of the **call**.

For all instructions described in this chapter, and for any PC-relative branch, the new **PC** (and the stored **PC** for **call** instructions) will be truncated to 32 bits if the processor is in **user** mode and the **Small Address Compatibility Mode** bit in the **Processor Status** register is on.

Instruction: **call** *address,rdst*

Opcode: 09

Operation: Move the PC to a register and jump to an absolute address.

Operands used: none

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation

Description:

A return **PC** is stored in *rdst*. The return **PC** is an offset from the address of the **call** instruction, and depends upon the **DC** field as given in Table 3-2. The third following instruction word will be fetched from *address*. This form of the **call** instruction must be used with the **absolute branch** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format. It is intended to be used as a subroutine call to an absolute address. The use of the **call** instruction in **Trap State** will produce an undefined return **PC**.

Instruction: **call** *(srca),rdst*

Opcode: 05

Operation: Move the PC to a register and jump to an address in a register.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation

Description:

A return **PC** is stored in *rdst*. The return **PC** is an offset from the address of the **call** instruction, and depends upon the **DC** field as given in Table 3-2. The third following instruction word will be fetched from an address specified by the operand *srca*. The low two bits of *srca* are treated as if they were zero. This form of the **call** instruction must be used with the **register branch** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format. It is intended to be used as a subroutine call instruction where the address of the subroutine is in a register. The use of the **call** instruction in **Trap State** will produce an undefined return **PC**.

Instruction: **jump** *address*

Opcode: 08

Operation: Jump to an absolute address.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation

Description:

The third following instruction word will be fetched from *address*. This form of the **jump** instruction must be used with the **absolute branch** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format.

Instruction: **jump** *(srca)*

Opcode: 04

Operation: Jump to an address in a register.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation

Description:

The third following instruction word will be fetched from an address specified by the operand *srca*. The low two bits of *srca* are treated as if they were zero. This form of the **jump** instruction must be used with the **register branch** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format. It is intended to be used as a subroutine return instruction or for computed branches (as with dispatch tables or dynamic linking).

CHAPTER 8. Processor Status Register and Timer Instructions

These instructions are used to read and write the **Processor Status** register (Table 2-5) and the timers. The **wps** instruction writes the entire **Processor Status** register in **supervisor** mode, but only writes the upper 32 bits in **user** mode. Two special instructions, **spl** and **srm**, more efficiently modify specific fields in the **Processor Status** register.

Instruction: **rps** *rdst*

Opcode: 0B

Operation: Read **Processor Status** register.

Operands used: none

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The **Processor Status** register is stored in *rdst*. Unused bits of the **Processor Status** register read as zero, but application programs must not rely on this fact since this may not be the case in other versions of the K-1 architecture.

Instruction: **wps** *srca*

Opcode: 0C

Operation: Write **Processor Status** register.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

If the processor is in **supervisor** mode then the entire operand, *srca*, is stored in the **Processor Status** register. If the processor is in **user** mode, then only the high-order 32 bits of *srca* are stored in the high-order 32 bits of the **Processor Status** register; the low-order 32 bits of the **Processor Status** register remain unaffected. A trap may occur if an attempt is made to set undefined bits in the **Processor Status** register [8-2]. Note that the **Processor Version Number** field is read-only and is not affected by this instruction. Not all of the effects of executing a **wps** instruction happen immediately. See Appendix C for a description of the delays involved.

Instruction: **spl** *srca,rdst*

Opcode: 0D

Operation: Set the **Processor Priority Level** field of the **Processor Status** register, returning its old value.

Operands used: *srca*

Results stored: *rdst*

Legal in: Supervisor mode only

Functional unit: Integer

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, then an illegal instruction/privilege violation trap occurs. Otherwise, the **Processor Priority Level<3..0>** field of the **Processor Status** register is set to the low four bits of *srca*. The previous value of this field is zero-extended to 64 bits and stored in *rdst*. Note that this instruction does not take effect immediately. See Appendix C for more details.

Instruction: **srm** *srca,rdst*

Opcode: 56

Operation: Set the **Rounding Mode** field of the **Processor Status** register, returning its old value.

Operands used: *srca*

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The **Rounding Mode<1..0>** field of the **Processor Status** register is set to the low two bits of *srca*. The previous value of this field is zero-extended to 64 bits and stored in *rdst*. Note that this instruction does not take effect immediately. See Appendix C for more details.

Instruction: **rut** *rdst*

Opcode: 5F

Operation: Read the Uptime Counter.

Operands used: none

Results stored: *rdst*

Legal in: User or Supervisor mode

Functional unit: Integer

Exceptions: none

Description:

The contents of the Uptime Counter are written to *rdst*. In any implementation where the Uptime Counter is less than 64 bits, the high-order bits of *rdst* are set to zero.

Instruction: **wit** *srca*

Opcode: 1F

Operation: Write the Interval Timer register.

Operands used: *srca*

Results stored: none

Legal in: Supervisor mode only

Functional unit: Integer

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, then an illegal instruction/privilege violation trap occurs. Otherwise, the Interval Timer register is written from the low-order bits of *srca*.

CHAPTER 9. Virtual Memory and Cache Instructions

These very special instructions are used to modify the processor page tables and caches. Some of them behave differently in **user** mode than in **supervisor** mode.

Instruction: **lipage** *srca*

Opcode: 18

Operation: Load Instruction Page Table entry.

Operands used: *srca*

Results stored: none

Legal in: Supervisor mode only

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, an illegal instruction/privilege violation trap will occur. If the processor is in **supervisor** mode, the operand, *srca,* is used to update the instruction page table. The format of the *srca* operand is given in Figure 2-7 [9-1].

Instruction: **ldpage** *srca,srcb*

Opcode: 19

Operation: Load Data Page Table entry.

Operands used: *srca, srcb*

Results stored: none

Legal in: Supervisor mode only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, an illegal instruction/privilege violation trap will occur. The address of the page table entry to be affected is calculated by hashing the virtual page number field of *srca* with the **Process Key<12..0>** field of the **Processor Status** register [9-3]. If the processor is in **supervisor** mode, the page table entry at the resulting table address is replaced by *srcb.* The format of the *srca* operand is given in Figure 2-8, and the format of the *srcb* operand is given in Figure 2-9 [9-2].

Instruction: **ickill** *(srca)*

Opcode: 06

Operation: Invalidate an instruction cache line and jump to an address in a register.

Operands used: *srca*

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: Illegal instruction/privilege violation, instruction page map miss

Description:

This instruction must be used with the **register branch** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format. **ickill** branches to the address given by *srca*; the low two bits of *srca* are treated as if they were zero. As is the case for all branch instructions, it is the third following instruction word fetch (called the **target** fetch) that is affected by this instruction. Prior to that fetch, however, a particular instruction cache line will be "killed". The instruction cache line to be killed is based on the address in *srca* in an implementation-dependent way [9-4], but will always contain the instruction word addressed by *srca*. It is important to note that **ickill** does *not* check for a hit in the instruction cache – the cache line addressed by *srca* is killed whether there is a hit or not. This can be used by the operating system to kill instructions at arbitrary physical addresses (even though instruction addressing in **supervisor** mode is limited), since only the low bits of the address matter for **ickill**.

During the target fetch, neither the instruction cache, instruction stache, nor memory will be referenced. Instead, a null instruction word will be created with both the I0 and I1 instructions disabled. An instruction page map miss *can* still result on the target instruction fetch. In case of an instruction page map miss, no line in the cache will be killed and an instruction page map miss trap will be taken.

The **ickill** instruction does not guarantee that the addressed line is killed in the instruction stache. The **iskill** instruction may be used to kill the entire contents of the instruction stache.

A branch instruction executed in the first delay slot of an **ickill** will prevent continuing execution at the address given in *srca*. In the following example, instruction words are fetched from the following locations: A, B, C, XYZ (which is killed in the instruction cache and is not executed), D, D+8, etc.

```
A:      ickill  XYZ, aa
B:      nop                     br      D, bb
C:      nop                     nop
D:      ...
        ...
XYZ:    ...
```

Instruction: **iskill**

Opcode: 16

Operation: Invalidate the entire instruction stache.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Fetch

Exceptions: none

Description:

The entire contents of the instruction stache are invalidated. Note that this instruction will *always* be executed, even when used with an instruction format that supports conditional execution. In other words, it is not possible to conditionally execute this instruction. This instruction does not take effect immediately [9-9].

Instruction:	**dflush**	*srca,srcb*
Opcode:	1D	
Operation:	Manipulate a data cache line.	
Operands used:	*srca, srcb*	
Results stored:	none	
Legal in:	User or Supervisor mode	
Functional unit:	Load/Store	
Exceptions:	**dflush**-type (see Table 5-1)	

Description:

The *srca* operand contains either the address of a memory location, or of a particular data cache line; there are no address alignment requirements for *srca.* This instruction performs some operation on the data cache line that contains the address *srca* (if there is such a line), or on the data cache line addressed directly by *srca.* Which of these two interpretations of *srca* is in effect is determined by a control bit in *srcb.* The operation to be performed is controlled by the low-order bits of *srcb* as enumerated in Table 9-1.

Table 9-1. dflush *srcb* Control Functions

Bit	Function
5	Write data only (don't write ECC bits)
4	Write ECC bits only (treat data as ECC bits)
3	Address a specific data cache line
2	Broadcast flush
1	Validate (update) the memory system
0	Kill data cache line

dflush is considered to be a **store** for purposes of access checking. Therefore, if the **User Mode Store** bit is on in the **Processor Status** register, the **dflush** is considered to be a **user** mode operation. In **user** mode operations, the **User Protection** bit in the **Processor Status** register controls the type of access granted.

If bit 3 of *srcb* is zero, then *srca* is considered to be a virtual address, which is then mapped, possibly resulting in a data map miss trap or an illegal access trap if write permission is not granted for the addressed page. If there is no trap, then the data cache address of *srca* will be computed. The addressed data cache line may or may not actually contain a valid

entry that may or may not be the entry for *srca*. (If *srca* does have an entry, it *will* be the addressed data cache line). The case where the addressed data cache line does not contain *srca* is called a **data cache miss.**

If bit 3 is a one, then the address in *srca* is used to address the data cache and the line selected is whatever line may be present (valid or not) at that location in the data cache. Which bits of *srca* give the data cache line address is implementation-dependent [9-5].

Bits 0 and 1 of *srcb* control the operation to be performed on the data cache line selected by the above process. In the case of a data cache miss, no operation is performed on this CPU's data cache. If bit 1 is set, then any modified data in the data cache line is written back to memory. If bit 0 is set, then the data cache line is invalidated (a subsequent reference to the address in *srca* will result in a data cache miss). Any combination of these bits (with at least one of them set) may be selected. Care should be take when killing a line without validating it as any modifications to the line will be lost.

If bits 0 and 1 are both clear, then a special **delayed write buffer** flush is done. In this case, bits 2, 3, 4, and 5 must all be zero. The least-significant bits of *srca* give the number of the delayed write buffer to be flushed. Delayed write buffers are used to perform **store** instructions more efficiently by delaying the **store** until the access can be verified in the data page table and the data cache tags. The ability to flush delayed write buffers is intended only for diagnostic purposes. The number of delayed write buffers is implementation-dependent [9-10].

Bit 2 of *srcb*, if set, forces bits 3, 4, and 5 to zero and causes the **dflush** request to be "broadcast" to all processors in the same multiprocessor system as the current processor. A **dflush** instruction is also broadcast whenever the data being referenced is shared and bits 3, 4, and 5 of *srcb* are all clear. Note that this means that all **dflush** instructions which are **supervisor** mode references (except those with bits 3, 4, or 5 of *srcb* set) will be broadcast.

The broadcast of a **dflush,** sometimes called a **shared dflush**, uses the cache coherence scheme to ensure data integrity across the cache's of all the processors in a multiprocessor system. All processors will be requested to perform the same operation on the same physical address. This broadcast happens even if there was a data cache miss on the access of *srca* and even if the page is not marked as shared in the data page table. The use of bit 2 to enable the broadcast feature allows the operating system to remove areas of memory on which I/O is being performed from the data caches of all CPUs, even if those areas are not marked as shared in the data page table.

Bits 4 and 5 of *srcb* are intended for use by diagnostic programs. Bit 4 causes any data writeback that occurs to be directed to the ECC bits rather than the data bits. Bit 5 prevents the ECC bits from being written. See Appendix E for details of referencing the ECC bits in memory. Unpredictable results will occur if either bits 4 and 5 are both on at the same time, or bits 0 and 1 are not both on if either bit 4 or 5 is on.

If the operation is in **user** mode (the **User Mode Store** bit is on in the **Processor Status** register), then bits 3, 4 and 5 of *srcb* will be forced to zero.

Instruction: **zcl** *(srca)*

Opcode: FF

Operation: Write zeroes to an entire cache line.

Operands used: *srca*

Results stored: mem(cache line size, *srca*)

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: **zcl**-type (see Table 5-1)

Description:

The address of a memory operand is given in *srca*. If the low-order bits of the address are not zero, then an illegal address trap will occur. The number of low-order bits that are checked is implementation-dependent [9-6]. The virtual address is mapped and the access is checked, possibly resulting in a data map miss or an illegal access trap. If no trap occurs, then the entire data cache line at the resultant physical address will be set to zero [9-7]. If a data cache miss occurs, then no access will be made to memory to bring in the old data cache line (since the line will be set to zero anyway). This instruction operates on the data cache only.

zcl is considered to be a **store** for purposes of access checking. Therefore, if the **User Mode Store** bit is on in the **Processor Status** register, the operation is considered to be a **user** mode operation. In a **user** mode operation, the **User Protection** bit in the **Processor Status** register controls the type of access granted.

Note that while **zcl** is a very efficient means of clearing data in the data cache, it is only efficient for non-shared references. All shared references must go through the memory system, and thus shared **zcls** run at memory speed. Non-shared **zcls** can be done locally in each CPU, and will run significantly faster. See Appendix C for more timing information.

Instruction: **pcl** *(srca)*

Opcode: F7

Operation: Preload cache line.

Operands used: *srca*, mem(cache line size, *srca*)

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Load/Store

Exceptions: none

Description:

The address of a memory operand is given in *srca*. The virtual address is mapped and the access is checked. If a map miss or access violation condition occurs, then the instruction completes without doing anything (and no trap will be taken). If no miss or access violation occurs, then the cache line containing the physical address will be read into the data cache. If the line is already in the data cache, then this instruction does nothing. If, in reading the line from the memory system, a non-existent memory or uncorrectable memory error is detected, then the operation is aborted and no error is reported.

pcl is considered to be a **load** for purposes of access checking. Therefore, if the **User Mode Load** bit is on in the **Processor Status** register, the operation is considered to be a **user** mode operation. In a **user** mode operation, the **User Protection** bit in the **Processor Status** register controls the type of access granted [9-8].

CHAPTER 10. Trap Instructions

These instructions cause traps, extract trap information from the processor after a trap occurs, and resume execution after a trap is handled.

Instruction: **bpt**

Opcode: 10

Operation: Cause a breakpoint trap.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Decode

Exceptions: **bpt**

Description:

This instruction causes a **bpt** trap to be taken. No following instructions will have been executed (i.e. this trap is precise). It is intended to be used by debuggers to cause execution to be cleanly suspended at any desired point in the program. Note that to unambiguously determine whether it was an I0 or an I1 instruction that took a **bpt** trap, **bpt** instructions should always use a 32-bit instruction format.

Instruction: **xtrap**

Opcode: 13

Operation: Cause an **xtrap** trap.

Operands used: none

Results stored: none

Legal in: Supervisor mode only

Functional unit: Decode

Exceptions: Illegal instruction/privilege violation, **xtrap**

Description:

This instruction causes an **xtrap** trap to be taken. No following instructions will have been executed (i.e. this trap is precise). This instruction is intended to be used by the operating system to enter **Trap State** prior to returning from a trap. If the processor is not in **supervisor** mode, then an illegal instruction/privilege violation trap occurs.

Instruction: **strap**

Opcode: 12

Operation: Cause a system call trap.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Decode

Exceptions: **strap**

Description:

This instruction causes an **strap** trap to be taken. No following instructions will have been executed (i.e. this trap is precise). This instruction is intended to be used by user programs requesting operating system services.

Instruction: **trap**

Opcode: 11

Operation: Cause a **trap** trap.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Decode

Exceptions: **trap**

Description:

This instruction causes a **trap** trap to be taken. No following instructions will have been executed (i.e. this trap is precise). This instruction is intended to be used by user programs to indicate certain abnormal conditions to the operating system.

Instruction: **rtrpd** *srca,rdst*

Opcode: 0F

Operation: Read 64 bits of trap data.

Operands used: *srca*

Results stored: *rdst*

Legal in: **Trap State** only

Functional unit: Integer

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, then an illegal instruction/privilege violation trap occurs. If the processor is in **supervisor** mode but not in **Trap State**, then the result will be unpredictable. A 64-bit unit of trap data is stored in *rdst.* Which unit of trap data is determined by *srca* in an implementation-dependent fashion. See Appendix D for details on trap handling.

Instruction: **slstrpd** *srca,srcb*

Opcode: 1C

Operation: Store 64 bits of load/store trap data to memory.

Operands used: *srca*, *srcb*

Results stored: mem(8, *srca)*

Legal in: **Trap State** only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation, **store**-type (see Table 5-1)

Description:

If the processor is in **user** mode, then an illegal instruction/privilege violation trap occurs. If the processor is in **supervisor** mode but not in **Trap State**, then the result will be unpredictable. The address of a memory operand is given in *srca*. A 64-bit unit of load/store trap data is stored in memory at the address specified by *srca*. Which unit of trap data is determined by *srcb* in an implementation-dependent fashion. See Appendix D for details on trap handling. The **slstrpd** instruction is subject to various traps associated with **store** instructions, and traps when in **Trap State** will halt the processor.

Instruction: **exts** *srca,address,rdst*

Opcode: 07

Operation: Exit trap state, load a register from an absolute address which is first transformed to be unique to one CPU in a multiprocessor, and jump to an address in a register.

Operands used: *srca*, mem(8, map(8***exts Load Address**))

Results stored: *rdst*

Legal in: **Trap State** only

Functional unit: Fetch, Decode, Load/Store

Exceptions: Illegal instruction/privilege violation, **load**-type (see Table 5-1) except no illegal address trap is possible

Description:

If the processor is in **user** mode, an illegal instruction/privilege violation trap occurs. If the processor is in **supervisor** mode but not in **Trap State**, the result will be unpredictable. If the processor is in **Trap State**, several operations are performed simultaneously by this instruction. First, the contents of *srca* specify the address of the third following instruction fetch. Second, a memory address is calculated as map(8***exts Load Address**), which for this instruction includes an implementation-dependent transformation to make the address unique among all of the CPUs in a multiprocessor [10-1]. The 64-bit word at the resultant address will be written to *rdst*. (The **exts Load Address** field specifies bits 35..3 of the virtual address of the 64-bit word to be loaded into *rdst*; the other bits of the virtual address are set to zero.) Finally, *before* the instruction at the address specified by *srca* is executed, the processor will have exited **Trap State**.

exts must be used in a very specific fashion to exit from **Trap State**. Once the first **exts** is executed, two more **exts**'s must be *immediately* executed with no intervening disabled instructions. Three sequential **exts** instructions are necessary to reload the three **Restart PC**s. See the section on **Traps, Interrupts, and Machine Checks** in Chapter 2 for more information on the **Restart PC**s, and Appendix D for details on trap handling.

This instruction must be used with the **exts** instruction format; an illegal instruction/privilege violation trap will result from its use with any other format. The **exts** instruction is subject to various traps associated with **load** instructions, and traps when in **Trap State** will halt the processor. The use of the **exts** format with either of bits 8 or 7 set to one, or with a delayed execution control field that disables *either* of the following two instructions, will produce unpredictable results.

CHAPTER 11. I/O Instructions

I/O instructions are privileged instructions that are used to communicate control information to and from I/O processors. Also listed here are instructions for communication with the Console.

Instruction: **rios** *srca,srcb,rdst*

Opcode: FE

Operation: Read I/O status.

Operands used: *srca,srcb*

Results stored: *rdst*

Legal in: Supervisor mode only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation

Description:

The operands *srca* and *srcb* are used to select an I/O status word. The selected I/O status is read and stored in the low-order 32 bits of *rdst*. The formats of *srca*, *srcb*, and of the I/O status read are described in Appendix F. The upper 32 bits of the result contain a status code indicating if the operation was successful or not. The status code is stored as a 2-bit number in bit positions 63 and 62. If the status code has the value 0, there were no errors (and the lower 32 bits of the result contain the I/O status). If the status code is non-zero, there was some type of error (and the lower 32 bits of the result should be ignored).

If the status code has the value 2, the I/O system was busy and the operation timed out before it could be sent. If the status code has the value 3, the operation was sent, but timed out before a reply was received. If the status code has the value 1, there was a protocol violation – the I/O system failed to respond to the **rios** request.

This instruction is legal only in **supervisor** mode. If a **user** mode program attempts to execute this instruction, an illegal instruction/privilege violation trap will result.

Instruction: **wios** *srca,srcb,rdst*

Opcode: FD

Operation: Write I/O status.

Operands used: *srca,srcb*

Results stored: *rdst*

Legal in: Supervisor mode only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation

Description:

The operands *srca* and *srcb* are used to send control information to the I/O system. The formats of *srca* and *srcb* are described in Appendix F. For **wios**, the lower 32 bits of the result are not used (and should be ignored). As with **rios**, the upper 32 bits of the result contain a status code indicating if the operation was successful or not. The status code is stored as a 2-bit number in bit positions 63 and 62. If the status code has the value 0, there were no errors (and the **wios** was successful). If the status code is non-zero, there was some type of error (and the results are unpredictable – the **wios** may or may not have modified state in the I/O system).

If the status code has the value 2, the I/O system was busy and the operation timed out before it could be sent. If the status code has the value 1, there was a protocol violation – the I/O system failed to acknowledge the **wios** request. The status code returned by a **wios** should never have the value 3.

This instruction is legal only in **supervisor** mode. If a **user** mode program attempts to execute this instruction, an illegal instruction/privilege violation trap will result.

Instruction: **rfec** *srca,rdst*

Opcode: E2

Operation: Read Console data.

Operands used: *srca*

Results stored: *rdst*

Legal in: Supervisor mode only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation

Description:

The Console data/control read port is read into the low-order 16 bits of *rdst*; the high-order 48 bits are set to zero. The low 2 bits of *srca* (*srca*<1..0>) control which Console read port is accessed – the low-priority read port (when *srca*<1..0> has the value 0), the debugger read port (when *srca*<1..0> has the value 1), or the high-priority read port (when *srca*<1..0> has the value 2). The next bit of *srca* (*srca*<2>) controls whether the read is destructive (if *srca*<2> is on), or non-destructive (if *srca*<2> is off). See Appendix G for details on the interface with the Console.

This instruction is legal only in **supervisor** mode. If a **user** mode program attempts to execute this instruction, an illegal instruction/privilege violation trap will result.

Instruction: **wfec** *srca*

Opcode: F2

Operation: Write Console data.

Operands used: *srca*

Results stored: none

Legal in: Supervisor mode only

Functional unit: Load/Store

Exceptions: Illegal instruction/privilege violation

Description:

The low-order 16 bits of the *srca* operand are written into the Console data/control write port, and an interrupt may be sent to the Console. The write port must not be full or unpredictable results will occur. See Appendix G for details on the interface to the Console.

This instruction is legal only in **supervisor** mode. If a **user** mode program attempts to execute this instruction, an illegal instruction/privilege violation trap will result.

CHAPTER 12. Miscellaneous Instructions

These instructions are used to perform certain miscellaneous control functions.

Instruction: **halt**

Opcode: 01

Operation: Halt.

Operands used: none

Results stored: none

Legal in: Supervisor mode only

Functional unit: Decode

Exceptions: Illegal instruction/privilege violation

Description:

If the processor is in **user** mode, an illegal instruction/privilege violation trap occurs. If the processor is in **supervisor** mode, it is halted. Execution may be continued at the instruction following the **halt** by the Console. Note that it *is* possible to continue from a **halt** in **Trap State**.

Instruction: **nop**

Opcode: 00

Operation: No operation.

Operands used: none

Results stored: none

Legal in: User or Supervisor mode

Functional unit: Decode

Exceptions: none

Description:

No operation is performed. If the **long constant** instruction format is used, several special delay features are invoked.

Several bits in the **LCON** field of a **long constant** format **nop** instruction can suspend execution until certain outstanding operations complete. These bits operate independently, so that if multiple bits are set, execution will be suspended until the last of the events completes.

Bit 19 of the **LCON** field, if set, cause the **nop** instruction's issue to be delayed until the load/store unit's pipeline is empty. In other words, the **nop** will issue no sooner than would an instruction that used a result from the *last* instruction sent to the load/store unit. Bit 18 of the **LCON** field has the same effect for the floating-point divide/square root unit, bit 17 for the floating-point multiply unit, and bit 16 for the floating-point add unit.

Certain memory requests, such as writing back a cache line due to a **dflush**, run "in the background" without holding up the load/store unit pipeline. Bit 20 of the **LCON** field, if set, causes the **nop** instruction's issue to be delayed until any outstanding memory request (other than an instruction cache miss read) is complete. This feature is most useful for I/O interactions, where it is necessary to wait for memory to be updated before initiating an I/O transfer.

In addition to bits 20 to 16, the low three bits of the **LCON** field of a **long constant** format **nop** instruction specify the minimum number of *extra* cycles (from 0 to 7) that the **nop** will take. If the low three bits are zero, then the **nop** will issue (barring any flag interlocks and the effects of bits 20 to 16) in one cycle.

See Appendix C for more details on instruction timing.

CHAPTER 13. Undefined Opcodes

All opcodes not defined in the individual instruction descriptions are illegal; executing one will give unspecified results. In the initial K-1 implementation, these opcodes cause an illegal instruction/privilege violation trap.

Appendix A. Instruction Index (Alphabetic)

Instruction	Opcode	Page
add	5A	6-2
add.d	90	4-16
add.s	80	4-17
addc	58	6-3
addt	59	6-4
and	63	6-40
andcc	65	6-42
andtc	69	6-41
ashf	61	6-51
bitcnt.w	5D	6-56
bitrev	5C	6-55
boof	2F	6-60
boof	4F	6-59
bpt	10	10-2
call	05	7-3
call	09	7-2
chk.b	50	6-62
chk.h	51	6-63
chk.w	52	6-64
chku.b	53	6-65
chku.h	54	6-66
chku.w	55	6-67
cmpeq.b	27	6-17
cmpeq.d	3B	4-3
cmpeq.h	26	6-18
cmpeq.l	24	6-20
cmpeq.s	3A	4-4
cmpeq.w	25	6-19
cmpge.d	37	4-3
cmpge.s	36	4-4
cmpgt.b	23	6-21
cmpgt.d	31	4-3
cmpgt.h	22	6-22
cmpgt.l	20	6-24
cmpgt.s	30	4-4
cmpgt.w	21	6-23
cmpleg.d	35	4-3
cmpleg.s	34	4-4
cmplg.d	33	4-3
cmplg.s	32	4-4
cmpueq.d	3F	4-3
cmpueq.s	3E	4-4
cmpugt.b	2B	6-25
cmpugt.d	39	4-3
cmpugt.h	2A	6-26
cmpugt.l	28	6-28
cmpugt.s	38	4-4
cmpugt.w	29	6-27
cmpun.d	3D	4-3
cmpun.s	3C	4-4
cvtd.l	95	4-8
cvtd.s	94	4-6
cvtl.d	97	4-10
cvtl.s	96	4-12
cvts.d	84	4-7
cvts.l	85	4-9
cvtul.d	98	4-11
dflush	1D	9-7
div.d	A3	4-22
div.s	A2	4-23
divssr	A6	6-15
divsst	A4	6-14
dshfl	6D	6-52
dshfr	6E	6-53
echk	F3	5-50
eload.b :1	E1	5-28
eload.b	E0	5-29
eload.h :1	E5	5-30
eload.h :2	E6	5-30
eload.h	E4	5-31
eload.l :1	ED	5-26
eload.l :8	EE	5-26
eload.l	EC	5-27
eload.w :1	E9	5-32
eload.w :4	EA	5-32
eload.w	E8	5-33
eload[u].l :1	ED	5-26
eload[u].l :8	EE	5-26
eload[u].l	EC	5-27
eloadu.b :1	F1	5-20
eloadu.b	F0	5-21
eloadu.h :1	F5	5-22
eloadu.h :2	F6	5-22
eloadu.h	F4	5-23

Instruction	Opcode	Page
eloadu.l :1	ED	5-26
eloadu.l :8	EE	5-26
eloadu.l	EC	5-27
eloadu.w :1	F9	5-24
eloadu.w :4	FA	5-24
eloadu.w	F8	5-25
exts	07	10-8
halt	01	12-2
ickill	06	9-4
iskill	16	9-6
jump	04	7-5
jump	08	7-4
ldecc	DC	5-47
ldnecc	BC	5-48
ldpage	19	9-3
lipage	18	9-2
load.b :1	C1	5-14
load.b	C0	5-15
load.h :1	C5	5-16
load.h :2	C6	5-16
load.h	C4	5-17
load.l :1	CD	5-12
load.l :8	CE	5-12
load.l	CC	5-13
load.w :1	C9	5-18
load.w :4	CA	5-18
load.w	C8	5-19
load[u].l :1	CD	5-12
load[u].l :8	CE	5-12
load[u].l	CC	5-13
loadcpu	8D	5-45
loadu.b :1	D1	5-6
loadu.b	D0	5-7
loadu.h :1	D5	5-8
loadu.h :2	D6	5-8
loadu.h	D4	5-9
loadu.l :1	CD	5-12
loadu.l :8	CE	5-12
loadu.l	CC	5-13
loadu.w :1	D9	5-10
loadu.w :4	DA	5-10
loadu.w	D8	5-11
lzcnt	5E	6-57
move	4B	6-30
move.d	93	6-31
mult.d	A1	4-20
mult.s	A0	4-21
multhss	A7	6-9
multhus	AF	6-13
multhuu	AB	6-11
multlss	A5	6-8
multlus	AD	6-12
multluu	A9	6-10
neg.d	92	4-14
neg.s	82	4-15
nop	00	12-3
or	6A	6-43
orcc	6C	6-45
ortc	6F	6-44
pcl	F7	9-10
relf	C2	5-51
rfec	E2	11-4
rios	FE	11-2
rot	66	6-49
rps	0B	8-2
rtrpd	0F	10-6
rut	5F	8-6
sel f0	40	6-38
sel f1	41	6-38
sel f2	42	6-38
sel f3	43	6-38
sel f4	44	6-38
sel f5	45	6-38
sel f6	46	6-38
sext.b	4C	6-35
sext.h	4D	6-36
sext.w	4E	6-37
shf	60	6-50

slstrpd	1C	10-7
spl	0D	8-4
sqrt.d	B3	4-24
sqrt.s	B2	4-25
srm	56	8-5
store.b :1	D3	5-35
store.b	C3	5-36
store.h :1	D7	5-37
store.h :2	E7	5-37
store.h	C7	5-38
store.l :1	DF	5-41
store.l :8	EF	5-41
store.l	CF	5-42
store.w :1	DB	5-39
store.w :4	EB	5-39
store.w	CB	5-40
storecpu	8F	5-46
strap	12	10-4
sub	4A	6-5
sub.d	91	4-18
sub.s	81	4-19
subb	48	6-6
subt	49	6-7
swat	FC	5-44
trap	11	10-5
wdwp	FB	5-54
welf	D2	5-52
wfec	F2	11-5
wios	FD	11-3
wit	1F	8-7
wps	0C	8-3
xnor	67	6-47
xor	68	6-46
xtrap	13	10-3
zcl	FF	9-9
zext.b	2C	6-32
zext.h	2D	6-33
zext.w	2E	6-34

Appendix B. Instruction Index (Numeric)

Opcode	Instruction	Page
00	nop	12-3
01	halt	12-2
04	jump	7-5
05	call	7-3
06	ickill	9-4
07	exts	10-8
08	jump	7-4
09	call	7-2
0B	rps	8-2
0C	wps	8-3
0D	spl	8-4
0F	rtrpd	10-6
10	bpt	10-2
11	trap	10-5
12	strap	10-4
13	xtrap	10-3
16	iskill	9-6
18	lipage	9-2
19	ldpage	9-3
1C	slstrpd	10-7
1D	dflush	9-7
1F	wit	8-7
20	cmpgt.l	6-24
21	cmpgt.w	6-23
22	cmpgt.h	6-22
23	cmpgt.b	6-21
24	cmpeq.l	6-20
25	cmpeq.w	6-19
26	cmpeq.h	6-18
27	cmpeq.b	6-17
28	cmpugt.l	6-28
29	cmpugt.w	6-27
2A	cmpugt.h	6-26
2B	cmpugt.b	6-25
2C	zext.b	6-32
2D	zext.h	6-33
2E	zext.w	6-34
2F	boof	6-60
30	cmpgt.s	4-4
31	cmpgt.d	4-3
32	cmplg.s	4-4
33	cmplg.d	4-3
34	cmpleg.s	4-4
35	cmpleg.d	4-3
36	cmpge.s	4-4
37	cmpge.d	4-3
38	cmpugt.s	4-4
39	cmpugt.d	4-3
3A	cmpeq.s	4-4
3B	cmpeq.d	4-3
3C	cmpun.s	4-4
3D	cmpun.d	4-3
3E	cmpueq.s	4-4
3F	cmpueq.d	4-3
40	sel f0	6-38
41	sel f1	6-38
42	sel f2	6-38
43	sel f3	6-38
44	sel f4	6-38
45	sel f5	6-38
46	sel f6	6-38
48	subb	6-6
49	subt	6-7
4A	sub	6-5
4B	move	6-30
4C	sext.b	6-35
4D	sext.h	6-36
4E	sext.w	6-37
4F	boof	6-59
50	chk.b	6-62
51	chk.h	6-63
52	chk.w	6-64
53	chku.b	6-65
54	chku.h	6-66
55	chku.w	6-67
56	srm	8-5
58	addc	6-3
59	addt	6-4
5A	add	6-2
5C	bitrev	6-55
5D	bitcnt.w	6-56

Opcode	Instruction	Page
5E	lzcnt	6-57
5F	rut	8-6
60	shf	6-50
61	ashf	6-51
63	and	6-40
65	andcc	6-42
66	rot	6-49
67	xnor	6-47
68	xor	6-46
69	andtc	6-41
6A	or	6-43
6C	orcc	6-45
6D	dshfl	6-52
6E	dshfr	6-53
6F	ortc	6-44
80	add.s	4-17
81	sub.s	4-19
82	neg.s	4-15
84	cvts.d	4-7
85	cvts.l	4-9
8D	loadcpu	5-45
8F	storecpu	5-46
90	add.d	4-16
91	sub.d	4-18
92	neg.d	4-14
93	move.d	6-31
94	cvtd.s	4-6
95	cvtd.l	4-8
96	cvtl.s	4-12
97	cvtl.d	4-10
98	cvtul.d	4-11
A0	mult.s	4-21
A1	mult.d	4-20
A2	div.s	4-23
A3	div.d	4-22
A4	divsst	6-14
A5	multlss	6-8
A6	divssr	6-15
A7	multhss	6-9
A9	multluu	6-10
AB	multhuu	6-11
AD	multlus	6-12
AF	multhus	6-13
B2	sqrt.s	4-25
B3	sqrt.d	4-24
BC	ldnecc	5-48
C0	load.b	5-15
C1	load.b :1	5-14
C2	relf	5-51
C3	store.b	5-36
C4	load.h	5-17
C5	load.h :1	5-16
C6	load.h :2	5-16
C7	store.h	5-38
C8	load.w	5-19
C9	load.w :1	5-18
CA	load.w :4	5-18
CB	store.w	5-40
CC	load[u].l	5-13
CD	load[u].l :1	5-12
CE	load[u].l :8	5-12
CF	store.l	5-42
D0	loadu.b	5-7
D1	loadu.b :1	5-6
D2	welf	5-52
D3	store.b :1	5-35
D4	loadu.h	5-9
D5	loadu.h :1	5-8
D6	loadu.h :2	5-8
D7	store.h :1	5-37
D8	loadu.w	5-11
D9	loadu.w :1	5-10
DA	loadu.w :4	5-10
DB	store.w :1	5-39
DC	ldecc	5-47
DF	store.l :1	5-41
E0	eload.b	5-29
E1	eload.b :1	5-28
E2	rfec	11-4
E4	eload.h	5-31
E5	eload.h :1	5-30
E6	eload.h :2	5-30
E7	store.h :2	5-37
E8	eload.w	5-33

E9	eload.w :1	5-32
EA	eload.w :4	5-32
EB	store.w :4	5-39
EC	eload[u].l	5-27
ED	eload[u].l :1	5-26
EE	eload[u].l :8	5-26
EF	store.l :8	5-41

F0	eloadu.b	5-21
F1	eloadu.b :1	5-20
F2	wfec	11-5
F3	echk	5-50
F4	eloadu.h	5-23
F5	eloadu.h :1	5-22
F6	eloadu.h :2	5-22
F7	pcl	9-10
F8	eloadu.w	5-25
F9	eloadu.w :1	5-24
FA	eloadu.w :4	5-24
FB	wdwp	5-54
FC	swat	5-44
FD	wios	11-3
FE	rios	11-2
FF	zcl	9-9

Appendix C. Instruction Timing Considerations

This appendix contains information that is applicable to the Version 1 implementation of the K-1 architecture. Future versions of the K-1 processor may make different implementation decisions, but should affect only those topics that are covered in this appendix.

C.1 Clock Cycles

The K-1 processor, memory, and I/O systems operate synchronously and are timed by a constant frequency main clock generator. All operations described in this appendix take an integral number of clock periods (usually referred to as **clock cycles**, or just **cycles**) to complete. The speed of the clock is given either as a frequency or as the duration of a single clock cycle. In the Version 1 implementation, the clock cycle is 6.5 nanoseconds.

C.2 Instruction Issue

Instruction words are fetched from the instruction stache according to the program flow. As explained in Chapter 2, an instruction word may contain a single 64-bit I0 instruction, or both 32-bit I0 and I1 instructions. Each instruction in an instruction word is decoded and, if appropriate, is **issued** for execution. To issue an instruction means to send the instruction to a particular functional unit to be executed. In this context, the fetch and decode units are considered to be functional units. Some number of clock cycles later, the functional unit will write its result (if any) to the register file and/or will cause a trap. Note that while the functional units send their results to the register file at the same time as they signal a trap, instructions that use the results of a trapping instruction may be issued for two cycles before the trap "takes effect" and stops instruction issue.

An important feature of the K-1 architecture is that, other than for trapping, it preserves the semantics of a "sequential" machine (one that issues no more than one instruction per clock cycle and waits for that instruction to complete before issuing another). Whenever the K-1 issues more than one instruction in a single cycle, it guarantees that those instructions do not conflict with each other in any way. Thus, instructions will never be issued "out of order" (in a different order than they would have been issued on a sequential machine), though often instructions that would have been issued on consecutive cycles on a sequential machine can be issued at the same time by the K-1.

The Version 1 implementation of the K-1 is capable of issuing two 32-bit instructions from the same instruction word, or one 64-bit instruction, in a single clock cycle. Conditions that can delay instruction issue are explained below.

C.3 Pipelining

The functional units in the K-1 are heavily **pipelined**. This means that they typically take more than one clock cycle to complete a single operation, but that they can begin a new operation every cycle. If a functional unit has a pipeline of length three, it can be doing three separate computations at the same time. Of course, the computations pass through the three separate phases of the calculation sequentially and independently of one another.

In order to take advantage of the pipelining of functional units in the K-1, it is necessary to issue instructions as soon as they become ready. In the optimal case, two instructions will be issued each cycle. At most one instruction can be issued to a given functional unit in any given cycle. Therefore, a pipelined functional unit is completely busy if it starts a new operation every cycle. The floating-point divide/square root unit is not pipelined; a maximum of two divides and one square root can be in progress at any one time. (See the subsection on the **Floating-Point Divide/Square Root Unit**.)

C.4 Functional Unit Latency and Interlocks

If an instruction is issued to a functional unit with a long pipeline, it is possible that the program may proceed to a point where it wishes to use the result of that instruction before the functional unit has written the result to the register file. Before describing how the K-1 deals with such a situation, two terms must be defined.

The **latency** of a functional unit is defined to be the number of cycles that must pass after the issue of a (result-returning) instruction to that functional unit before the result can be used. For example, in the following program:

```
add.d %r1,%r2,%r3        (0)
<instruction word>       (1)
<instruction word>       (2)
<instruction word>       (3)
<instruction word>       (4)
<instruction word>       (5)
move %r3,%r4             (6)
```

the **move** instruction is in the first position which can use the result of the **add.d** instruction. Since the **move** instruction can not issue sooner than relative cycle 6, the latency of the floating-point add functional unit must be 6. The latencies for the K-1 functional units are given in Table C-1.

If the **move** instruction in the above example was located at cycle 0 through 5 relative to the **add.d**, then it would be ready to issue before all its operands were ready. A hardware mechanism provided to detect this situation and delay the issue of the instruction until the operands are available is called an **interlock**. Interlocks can also delay the issue of instructions for reasons other than the unavailability of register operands. Interlocks are provided *wherever* they are needed. That is, a program will never be incorrect because it failed to wait long enough before using the result of a previous instruction.

Table C-1. Functional Unit Latencies

Functional Unit or Operation	Latency
Integer	3
Load/Store	9 †
Floating-Point Add	6
Floating-Point Multiply	8
Floating-Point Divide	20, 25, 35 ‡
Floating-Point Square Root	32, 61 ∇

C.5 K-1 Interlocks

This section enumerates the different types of interlocks in the K-1 and explains their effects on instruction timing.

C.5.1 Register Interlocks

Register interlocks are generated when an attempt is made to issue an instruction when one or more of its source or destination registers is currently **reserved**. A register is reserved when it is scheduled to be written by a previously issued but not yet completed instruction. An interlock begins on the cycle in which an instruction that writes a register is issued, and remains in place until the instruction completes. The duration of an interlock (in cycles) is the same as the functional unit latency of the instruction destined to write the register. An instruction cannot issue until all the registers it reads and writes are free of interlocks. (The exceptions to this rule are explained below).

The integer, floating-point multiply, and floating-point add functional units each have a fixed latency regardless of the type of operation the unit is performing. For example, all floating-point adds (both single and double precision) have a latency of six. Thus, any attempt to read *or to write* the destination register of a floating-point add will interlock until the add completes (the sixth cycle after the add was issued).

The load/store functional unit has a fixed latency for most normal load/store instructions that complete without data cache misses or traps. If there is a data cache miss, then the load/store functional unit pipeline is stopped until the miss is resolved (either by bringing the proper line into the data cache, or by trapping). Thus, load/store operations always complete in order. Data cache bank conflicts and store byte operations in some circumstances

† For **load** and **eload** instructions that hit in the data cache and complete without trapping.

‡ 20 for single precision, 25 for integer, and 35 for double precision operations.

∇ 32 for single precision, 61 for double precision.

can also extend the latency of load/store operations. Some special load/store instructions always have a longer latency. See the section on **Load/Store Timing** for more information.

The floating-point divide/square root functional unit is the only functional unit that has different latencies for different operations. In fact, divide/square root instructions can finish out of order with respect to other divides/square roots. For example, a double precision format floating-point divide could be started at cycle 0, and a single precision format floating-point divide could be started at cycle 10. The single precision divide would finish at cycle 29, five cycles before the double precision divide.

Register interlocks from the integer functional unit are handled specially. The integer functional unit implements **wrapping**. Wrapping allows the integer functional unit to utilize prior results without having to wait for them to be written into the register file. Thus, a sequence such as

```
add %r1,%r2,%r4
add %r3,%r4,%r5
```

does not pay the usual penalty for functional unit latency – the second **add** can be issued on the cycle following the first, even though it uses a result produced by the first. In effect, there are no interlocks on the results of integer instructions *when* they are used as operands to the integer functional unit. An attempt by another functional unit to use a result from the integer functional unit will still result in an interlock. Wrapping does not apply to the results or to the operands of the **call, rtrpd, wps, rps, spl, srm, rut,** and **wit** instructions. Also, the *srca* operand of the **dshfl** and **dshfr** instructions cannot be wrapped.

Another way in which the integer functional unit is treated specially is with regard to register "write" interlocks. A write interlock occurs when one instruction tries to write a register that is already scheduled to be written by an earlier instruction. In order to guarantee that the results will be written in the proper order, the second instruction is held up until the write of the first instruction completes. But, if the first instruction was issuing in the integer functional unit, then it is guaranteed to complete its write before the second – the integer functional unit has a shorter latency than any other functional unit. While this optimization could be done for all functional units, it would then greatly impact the complexity of the wrapping control logic. Thus, if an *integer* instruction tries to write a register that is already scheduled to be written by the integer functional unit, no interlock is generated.

C.5.2 Flag Interlocks

No interlock is generated for reading or writing a flag in the integer functional unit (i.e., as an argument of a **cmp, boof, addc, subb,** or **sel** instruction). This is because the flags are maintained in the integer unit and thus there is no latency to access them within that unit.

When flags are used for conditional execution (including branching), the latency of the integer functional unit must be taken into account. The use of a flag for conditional execution (or branching) will be delayed until the fifth cycle following the instruction that set the flag, as illustrated below.

```
cmpgt.l %r1,%r2,%f3         (0)
<instruction word>          (1)
<instruction word>          (2)
<instruction word>          (3)
<instruction word>          (4)
move <%f3> %r3,%r4          (5)
```

C.5.3 I1/I0 Interlocks

In the best case, two 32-bit instructions in a single instruction word can be issued in parallel. A number of **I1/I0 interlocks**, however, may prevent this from happening. The causes of I1/I0 interlocks are enumerated and explained in the following subsections. I1/I0 interlocks always cause the I1 instruction to wait until the interlock condition is removed – they never delay the issuing of an I0 instruction.

C.5.3.1 I0 Interlocked

Whenever the I0 half of an instruction word is interlocked, the I1 instruction is also interlocked. This prevents the I0 and I1 instructions from issuing out of order.

C.5.3.2 Serial Instructions

Certain instructions are serialized by the hardware because they affect machine state in more than one functional unit. These **serial** instructions are **halt**, **wps**, and **srm**. Whenever either of I0 or I1 is a serial instruction, they must issue in separate cycles.

C.5.3.3 I1/I0 Read Port Conflicts

Since the functional units in the K-1 are only capable of accepting one operation per clock cycle, there will be an I1/I0 interlock whenever the I0 and I1 instructions attempt to use the same functional unit. The actual cause of this interlock is slightly more complicated.

The register file in the K-1 has five **read ports** which provide the *srca* and *srcb* operands to the functional units. Each read port is connected to one or more functional units. If the I0 and I1 instructions need to use the same read port, this prevents the I1 instruction from issuing until after the I0 instruction issues. The floating-point add, floating-point multiply and load/store units each have their own read port. The integer, floating-point divide/square root, and fetch units share a read port (but there are very few instructions issued to the fetch unit). Thus, for example, if both the I0 and I1 instructions use the floating-point add functional unit, the I1 instruction will interlock until after the I0 instruction issues. Or, if the I0 instruction uses the integer functional unit and the I1 instruction uses the floating-point divide/square root functional unit, the I1 instruction will interlock until after the I0 instruction issues.

The **lipage** instruction is issued to the fetch unit on the same read port as the floating-point divide/square root and integer units, and thus cannot be issued at the same time as any

floating-point divide/square root or integer operation. Note that branches to an address in a register also issue to the fetch unit; these instructions cannot, however, interlock with floating-point divide/square root or integer instructions because *all* non-PC-relative branches are in 64-bit instruction formats.

In addition to the normal functional unit read ports, there is a special read port that is used only for **store** and **dshfl/dshfr** instructions. This read port allows these instructions to read three registers instead of the usual two. Because of this, a **store** cannot issue at the same time as a **dshfl** or **dshfr**.

C.5.3.4 I1/I0 Register and Flag Conflicts

Since the K-1 preserves the semantics of a sequential machine, an I1 instruction that reads or writes a register (or a flag) that is written by the I0 instruction in the same instruction word will interlock. This interlock is a special case of the register and flag interlocks described above, and serves to prevent the I1 instruction from issuing until after the I0 instruction has issued (at which point, the issue of the I1 instruction will be blocked by a register or flag interlock, unless they are both integer unit instructions).

C.5.4 Floating-Point Divide/Square Root Unit

The floating-point divide/square root unit is capable of executing only three instructions at one time – two divide instructions and one square root instruction. An attempt to issue a third divide before one of the two outstanding divides completes results in an interlock. An attempt to issue a second square root before the outstanding square root completes also results in an interlock. These interlocks last three cycles longer than the interlock on the result of the first completing divide (or square root). (I.e., the result of the floating-point divide/square root instruction is available three cycles before another such instruction can be issued.) Note that floating-point divide/square root unit instructions can complete out of order from each other, as, for example, when a single precision floating-point divide is issued the cycle after a double precision floating-point divide.

Floating-point divide and square root instructions can complete in less time than specified in Table C-1. Invalid operands, for example, will be detected almost immediately by the floating-point divide/square root unit, and the instruction will complete and return a trap code without waiting for the usual latency to elapse.

There is an additional interlock after a divide instruction (**div.s, div.d, divsst, divssr**) is issued – no divide or square root instructions can be issued on the two following cycles. Similarly, after a square root instruction (**sqrt.s, sqrt.d**), no divide or square root instructions can be issued on the following cycle. The hardware detects these cases, and will not issue any divide or square root instruction on the two cycles following the issue of a divide instruction, or the cycle following the issue of a square root instruction.

There are also additional delays when the results of divide or square root instructions are scheduled to complete at (or near) the same time. For example, a single precision divide issued five cycles after an integer divide would be scheduled to complete at the same time as

the integer divide. In cases such as this, one of the divides will complete a cycle later. The details of this are TBD.

C.5.5 Load/Store Unit

The load/store unit is capable of executing only eight instructions at one time. An attempt to issue a ninth load/store instruction before one of the eight outstanding ones completes results in an interlock. (Note that load/store instructions always complete in order.) One cause of this interlock is that certain load/store instructions, such as **wps**, wait for all the load/store instructions in front of them to complete before they proceed (and thus cause the load/store queue to back up behind them). This interlock can also be caused by data cache misses, and other dynamically varying events. If, for example, a **load** misses in the data cache, the load/store functional unit pipeline will stop until the data cache miss is satisfied. The dynamic nature of some of the delays makes this interlock difficult to predict statically. This may lead to program execution stopping for one of two reasons: an instruction tries to read or write the register that is being loaded, causing a register interlock, or, a ninth load/store instruction (the eighth after the "missing" instruction) tries to issue, causing a load/store unit interlock. (See the section on **Load/Store Timing** for more details.)

C.6 Branching

In addition to register and flag interlocks, branches can also experience additional delays.

C.6.1 Branching to a PC-Relative or Absolute Address

Branching per se incurs no absolute penalty measurable as a number of clock cycles. There are, however, dynamic penalties incurred by branching that are difficult to predict statically. To understand these penalties, a brief explanation of the K-1 hardware is necessary.

The instruction stache is a small cache that can be accessed very quickly. Branches that hit in the instruction stache incur absolutely no penalty. Branches that miss in the instruction stache, however, may or may not incur a penalty depending on how backlogged the issue unit is, and whether the "missing" data is in the instruction cache or must be retrieved from main memory.

Filling one line (64 bytes) of the instruction stache from the instruction cache takes only four additional cycles. This delay, however, does not necessarily slow down a running program. The fetch unit (which is responsible for retrieving instructions in the proper order) operates autonomously from the issue unit (which is responsible for issuing instructions in the proper order). Barring such events as instruction stache missing (and a few exceptional instructions such as branches to an address in a register, **wps**, **lipage**, etc.) the fetch unit will retrieve one instruction word from the instruction stache each clock cycle. Since four additional cycles are required for an instruction stache miss, the desired instruction word is retrieved at cycle $i+4$, instead of cycle i.

A queue of four instruction words, called the **Instruction Queue**, lies between the fetch and issue units. Normally, the fetch unit will be ahead of the issue unit (because of interlocks which slow down the issue process). While the fetch unit is handling an instruction stache miss, the issue unit can continue executing out of the Instruction Queue. Thus, a branch that misses in the instruction stache will only incur a cycle time penalty if the issue unit is able to empty the Instruction Queue before the fetch unit can retrieve the "missing" data.

Filling the instruction cache from main memory takes roughly 60 cycles, but this time is variable due to memory interference from other processors, different RAM speeds in the memory subsystem, etc. In this case, it is almost certain that the Instruction Queue will empty and some number of cycles will be lost. The large size of the instruction cache (1 MB), however, makes an instruction cache miss a rare event.

C.6.2 Branching to an Address in a Register

In addition to the general branch penalty described above, branching to an address in a register incurs an additional three cycle penalty in order to retrieve the contents of the register and affect the Program Counter. A further potential penalty comes from the fact that when a branch to an address in a register issues, there will be at most one other instruction word in the Instruction Queue.

C.7 Special Instructions

A number of instructions in the K-1 have special timing rules/constraints. These instructions are described below. Other than load/store instructions, which are described in the section on **Load/Store Timing**, any instruction *not* described below can be assumed to issue in a single cycle and to return its result (if any) after the latency of the functional unit in which it executes. Note that this section only considers instructions that have special timing rules and does not consider interlock issues (discussed in the section on **Interlocks**), or branch instructions (discussed in the section on **Branching**).

C.7.1 nop

nop instructions that are conditionally executed experience the same flag interlock as any other instruction. A 32-bit (**register** or **short constant** format) **nop** in the same instruction word as another instruction can be issued in parallel with any other instruction. Barring flag interlocks, two 32-bit **nop** instructions in the same instruction word will issue in the same cycle.

nop instructions in **long constant** format are treated specially. See the **nop** instruction description for details.

C.7.2 rps, wps, spl, and srm

The **wps** and **srm** instructions cannot be issued in parallel with any other instruction since they are serial instructions. In addition, no other instructions will issue until the fifth cycle following the issue of a register or **short constant** format I0 **wps** or the sixth cycle following the issue of a **long constant** format I0 or an I1 **wps**. (This additional penalty, which accounts for the time required to update the flags in the fetch and issue units, does not apply to **srm** or **spl**).

The **rps** instruction will not issue until all the arithmetic (integer, floating-point add, floating-point multiply, and floating-point divide/square root) functional unit pipelines have emptied. This allows the correct **Arithmetic Exception Flags<4..0>** field of the **Processor Status** register to be read. The timing of **rps** is such that it can be issued one cycle before the results of all the outstanding floating-point or integer instructions are available. (For instructions that do not return results, as for some integer instructions such as **wit**, **rps** will wait for these instructions as if they returned a result that the **rps** read.) Furthermore, **rps** does not have to wait any additional time for compare instructions (or other instructions that modify the flags). The integer unit is responsible for maintaining the flags and for returning the flag portion of the **Processor Status** register, so that any outstanding flag-modifying instructions will complete before the **rps**. To clarify these delays, a **rps** cannot be issued until the second cycle following the issue of an integer instruction, the fifth cycle following the issue of a floating-point add instruction, or the seventh cycle following the issue of a floating-point multiply instruction. That is, the **rps** waits one cycle less than it would if it were waiting for the result of the previously issued instruction(s).

The **spl** instruction is intended to be used only by the operating system in very special circumstances. Though it modifies the **Processor Status** register, it incurs no additional execution time penalties. But, the effect of the **spl** is delayed until the fifth cycle following its issue. If, for example, the priority level was being changed from 15 (the lowest priority level) to 2, it would be necessary to wait four cycles after issuing the **spl** to guarantee that no interrupts lower in priority than level 2 were about to occur. Note, however, that waiting only three cycles after issue of the **spl** is enough to guarantee that critical code is not interrupted *after it has begun.* The fourth cycle after the **spl** could receive an interrupt at the old level, but the instruction would be interrupted *before* it was issued, so that the critical section could be restarted. This three cycle wait can be accomplished with a **long constant** format **nop** with an operand of 2 immediately after the **spl**. Note that when changing from a high priority level to a lower one, no wait is needed since lower priority level interrupts are already blocked.

C.7.3 Trap Instructions and Instructions Which Trap

The trap instructions (**bpt**, **trap**, **strap**, and **xtrap**) all cause execution of the current process to be suspended *after* their issue. It does, however, require some amount of time to begin executing instructions in **Trap State**. The bulk of this penalty will be due to the time required to start fetching the instructions of the trap handler (which may involve an instruction cache miss). If the trap handler is resident in the instruction cache, however, there is

just a 12 cycle penalty to enter and start executing in **Trap State**. That is, if the trap instruction is issued at time *i*, the first instruction in **Trap State** will be issued at time $i+13$.

When instructions that have been issued to the functional units trap, the trap will suspend further issuing on the second cycle following the cycle when the result of the instruction could have been used. If instruction issue is suspended on cycle *i* by a functional unit trap, then the first instruction in **Trap State** will be issued at time $i+12$.

The time required to enter **Trap State** will be slightly longer when the functional unit pipelines are full when the trap is detected. The integer, floating-point add, and floating-point multiply pipelines will drain completely before **Trap State** is entered (and thus do not delay entering **Trap State**), but the load/store and floating-point divide/square root pipelines may not empty before **Trap State** is entered (due to their longer latencies). The issue unit will not issue the first **Trap State** instruction until the fourth cycle after *all* the pipelines have emptied.

C.7.4 exts

The **exts** instruction has the same timing as a branch to register combined with a **load.l**. Recall that three **exts** instructions in a row are required to exit **Trap State**; this is the only interesting timing case for **exts**, since any other use of it will produce undefined results. In the optimistic case where all three **exts** instructions hit in the instruction stache, and the first instruction returned to hits in the instruction cache, the first return instruction will issue on the tenth cycle following the issuing of the first **exts** instruction. (Whenever **Trap State** is entered or exited, the instruction stache is killed. Thus, there is always an instruction stache miss penalty associated with entering or exiting **Trap State**). Note that the first return instruction could interlock on the results of one of the **exts** instructions, because it could use as an operand (or a result) one of the registers being reloaded by the **exts** instructions. Also, because of the penalties associated with branching to an address in a register, the third **exts** instruction will not issue until the *fifth* cycle following the first.

C.7.5 ickill

The **ickill** instruction has the same timing as a branch to an address in a register, but incurs an additional three cycle instruction fetch penalty. Note that the destination of an **ickill** instruction will never miss in the instruction stache or instruction cache. Timing-wise, the **ickill** instruction acts like an instruction stache miss that takes only three cycles instead of the usual four.

C.7.6 iskill

The **iskill** instruction always causes the third following instruction word to miss in the instruction stache. Other than the stache miss penalty, it has no timing effects.

C.8 Load/Store Timing

The load/store unit has very complicated timing due to the banked nature of the data cache, the delayed nature of **store** instructions, the variable delays of the memory subsystem, and the unpredictable appearance of cache coherency requests.

C.8.1 load instructions

All **load** instructions that hit in the data cache *and* do not have **bank conflicts** have a latency of nine cycles. This includes **eload, ldecc, ldnecc**, etc.

A bank conflict results when two addresses that agree in bits five through three are used twice within three cycles. The second occurrence of such an address will wait until the third cycle following the first occurrence before it begins executing in the load/store unit. Note that the bank conflict is detected in the load/store unit as it is about to execute instructions. Due to data cache misses and other dynamically varying events in the load/store unit, it is possible that two load/store instructions that were issued more than three cycles apart will end up trying to execute on successive cycles in the load/store unit.

load instructions that miss in the data cache are subject to variable memory subsystem delays. Assuming that the read request for the **load** was handled immediately, the **load** instruction would have a latency of roughly 55 cycles. The load/store unit, however, remains busy (and will not start executing another instruction) for an additional 12 cycles after returning the result of the **load** instruction to the register file. (The additional 12 cycles is the time required to write the data returned from memory into the data cache).

There is no additional time penalty for data cache misses if the data present in the cache is modified and has to be written back to memory. In general, this time is overlapped with the memory access.

C.8.2 store instructions

In addition to the timing irregularities associated with **load** instructions (bank conflicts and data cache misses), **store** instructions have additional timing complications due to their use of **Delayed Write Buffers** (**DWB**s).

This section needs to be expanded, but it's real complicated.

C.8.3 ldpage

The **ldpage** instruction requires all load/store instructions in front of it to complete, and then prevents any other load/store instructions from executing for two additional cycles.

C.8.4 pcl and zcl

The **pcl** instruction will always issue in one clock cycle. The hardware, however, has the option of ignoring the **pcl**, or of cancelling it after it is started. (This cannot affect the cor-

rectness of programs, since **pcl** is simply an optimization). If the **pcl** is completed, it can reduce the delay on a data cache miss to under 12 cycles, with an additional 12 cycles during which no other load/store instructions can be started. (This additional time is required to write all the data retrieved from the memory subsystem into the data cache.)

The **zcl** instruction must write four sub-lines into the data cache, and requires a minimum of 12 extra cycles to do this. **zcl** must also wait for any instructions in front of it to complete.

C.8.5 dflush

A **dflush** that does not write data back to memory will take TBD cycles. If a memory write is involved, the data cache will be tied up for TBD cycles, thus preventing other load/store operations from executing, but not necessarily from issuing (which would have stopped the entire K-1).

C.8.6 rfec and wfec

rfec and **wfec** execute locally in the K-1 processor. Both these instructions wait for all load/store instruction ahead of them to complete before they issue.

C.8.7 rios and wios

The **rios** and **wios** instructions are executed with the cooperation of the I/O system. Their latency is discussed more fully in Appendix F.

Appendix D. Trap Handling

Many aspects of trap handling are implementation-dependent. This appendix describes the details of trap handling in the Version 1 implementation of the K-1 architecture. It contains specifics on the data preserved by the CPU on a trap, how to read it out, and how to restart after a trap. It concludes with an example of a K-1 assembly language program that is intended to be a model for the implementation of trap handlers.

D.1 Trap Data and Trap Recovery

Two major groups of trap data stored by the K-1 aid in recovery from a trap. The **primary trap data**, which can be read into registers with a sequence of **rtrpd** instructions, contains the Trap Summary, Trap Locators, and Restart PCs. The **load/store trap data**, which can be stored to memory with a sequence of **slstrpd** instructions, contains information used to recover from load/store traps. It is important to remember that load/store traps can occur in conjunction with other types of traps and/or interrupts. The following sections describe the two types of trap data.

D.1.1 Primary Trap Data

The primary trap data consists of the Trap Summary, the Trap Locators, and the Restart PCs. The Restart PCs, shown in Figure 2-12, are described in Chapter 2. The Trap Locators for the integer, floating-point multiply, and floating-point add units collectively are known as the **IMA Trap Locators**. The load/store and floating-point divide/square root units have Trap Locators called the **L/S Trap Locators** and the **Divide Trap Locators**, respectively.

D.1.1.1 Trap Summary

Since many traps (and interrupts) can occur simultaneously, the **Trap Summary**, shown in Figure D-1, indicates which **trap types** occurred in the current trap. The **Trap Summary** is divided into a Trap Report half (in the low-order 32 bits) and a Trap Status half (in the high-order 32 bits). The Trap Report contains all information about the occurrence of traps, interrupts, and machine checks. The Trap Status contains additional information that makes trap processing more efficient.

Five types of traps can occur: **DT** indicates a decode trap, **IFT** indicates an instruction fetch trap, **IT** indicates an integer overflow or check trap, **FT** indicates a floating-point trap including an invalid operation from a floating-point compare, and **LST** indicates a load/store trap. **DT**, **IFT**, and **LST** are decoded according to Table D-1.

43 41	40 37	36 32	31 28	27 23	22 20	19 18	17 16	15	14	13 12	11	10 8	7 5	4	3	2	1	0
DIVX (3)	IMAX (4)	LVL (5)	LST (4)	TCMC (5)	IFUE (3)	IPE (2)	EXTE (2)	FT	IT	00	ICPE	IFT (3)	DT (3)	IOI	WPI	RPI	CKI	NMI

Figure D-1. Trap Summary

Machine checks, in broad terms, are anomalous conditions that should never occur in a correctly functioning system. They are indicative of component, connector, or design failures, possibly of an intermittent or one-time nature. These errors' sporadic (and unexpected) nature makes them extraordinarily hard to diagnose and repair. The K-1 hardware detects a number of the most likely of these errors and logs them as part of the **Trap Summary** so that the operator can be notified of the problem. There are five classes of machine checks: **ICPE** indicates an instruction cache parity error, **EXTE** indicates an external error, **IPE** indicates an internal pipeline error, **IFUE** indicates a functional unit error, and **TCMC** indicates a trap-class machine check (an error detected along with a normal trap). **EXTE, IPE, IFUE,** and **TCMC** are decoded according to Table D-2.

The external interrupts, **NMI**, **CKI**, **RPI**, **WPI**, and **IOI**, are discussed in the subsection on **The Trap Sequence** in Chapter 2. Table 2-9 lists the priority level corresponding to each external interrupt.

There are three fields in the Trap Status half of the **Trap Summary**: **DIVX**, **IMAX**, and **LVL**. If the floating-point divide/square root unit trapped on a divide (floating-point or integer) or square root instruction, the **DIVX** field indexes the floating-point divide/square root **engine** that detected the first trap. The floating-point divide/square root unit contains two divide engines and one square root engine, each of which can be busy with a single instruction (of the appropriate type) at any given time. If no floating-point divide/square root instruction trapped, **DIVX** is guaranteed to be zero. See Appendix C for more information about the floating-point divide/square root functional unit. Similarly, the **IMAX** field provides an index into the **IMA Trap Locators** to help identify the first trapping integer, floating-point add, or floating-point multiply instruction. If no such trap was detected, **IMAX** will be zero. If a trap was detected, **IMAX** will take on a value between 1 and 10 which corresponds to the appropriate **IMA Trap Locator** entry. See sub-section D.1.1.2.1 for more information on the **IMA Trap Locators**. The **LVL** field records status information for I/O system interrupts and contains two subfields: **LVL<4>**, which indicates that a parity error was detected on the wires over which the I/O system signals interrupts to the CPU, and **LVL<3..0>**, which indicates the level of the highest priority (pending) interrupt. If **IOI** is not asserted, the **LVL** field is undefined. If **LVL<4>** is asserted, **LVL<3..0>** is undefined.

Table D-1. Trap Encodings

Trap Type	Hex Value	Description
DT	0	No trap
	1	Unused
	2	Illegal instruction/privilege violation
	3	Trace trap
	4	**bpt** executed
	5	**trap** executed
	6	**strap** executed
	7	**xtrap** executed
IFT	0	No trap
	1	Nonexistent memory error
	2	Uncorrectable ECC error
	3	Memory-system related parity errors
	4	Instruction cache parity error
	5	Instruction cache tag parity error
	6	Instruction map miss
	7	Unused
LST	0	No trap
	1	Data map miss
	2	Illegal address (misaligned)
	3	Illegal access (protection violation)
	4	**echk** trap
	5	Data watchpoint trap
	6	Page Map/Cache Tag parity error (R) †
	7	Page Map/Cache Tag parity error (NR) ‡
	8	Unused
	9	ECC error on first cache sub-line
	A	ECC error on subsequent cache sub-lines
	B	Memory-system related parity errors
	C	Nonexistent memory error
	D-F	Unused

† Recoverable trap

‡ Non-recoverable trap

Table D-2. Machine Check Encoding

Trap Type	Hex Value	Description
EXTE	0	No external machine check
	1	Memory read/data cache parity error
	2	I/O parity error
	3	Memory Interface Controller fatal error
IPE	0	No internal pipeline machine check
	1	Trap State PC when not in Trap State
	2	USED or ISSUE from DC with PC Pipe empty
	3	Valid PC with PC Pipe full
IFUE	0	No internal functional unit machine check
	1	Invalid exception from square root engine
	2	FMULT/FADD reports divide by zero exception
	3	Simultaneous IDIV and FDIV exceptions
	4	Load/Store DONE with L/S QUEUE empty
	5	Load/Store ISSUE with L/S QUEUE full
	6	Divide inconsistency
	7	Unused
TCMC<4>	N/A	Restart PC error
TCMC<3..0>	0	No trap-class machine check
	1	INT Trap without exception code
	2	FADD Trap without exception code
	3	FMULT Trap without exception code
	4	FDIV Trap without exception code
	5	FDIV Trap without FDIV DONE
	6	INT Trap without instruction issue
	7	FADD Trap without instruction issue
	8	FMULT Trap without instruction issue
	9	Second Load/Store Trap
	A	Load/Store Trap with L/S QUEUE empty
	B-F	Unused

D.1.1.2 Trap Locators

There are three types of Trap Locators. The floating-point divide and load/store units have Trap Locators called the **Divide Trap Locators** (Figure D-2) and the **L/S Trap Locators** (Figure D-3). The Trap Locators for the integer, floating-point multiply, and floating-point add units collectively are known as the **IMA Trap Locators** (Figure D-4). Associated with the **IMA Trap Locators** are the **IMA Trap Bits.**

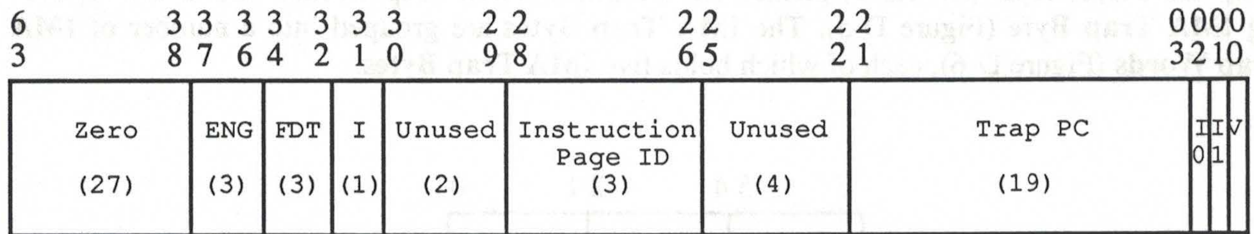


Figure D-2. Divide Trap Locator

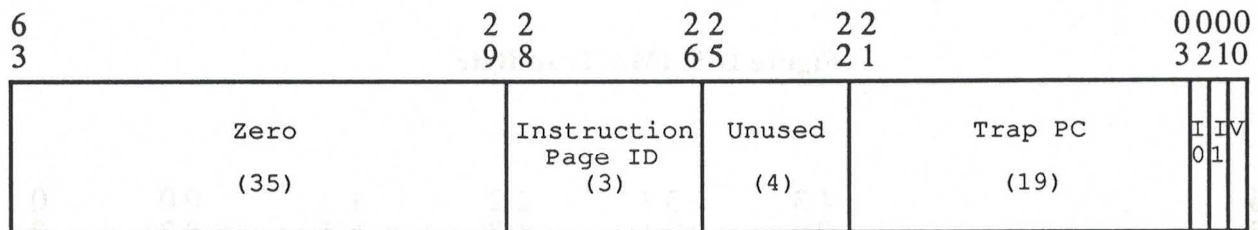


Figure D-3. Load/Store Trap Locator

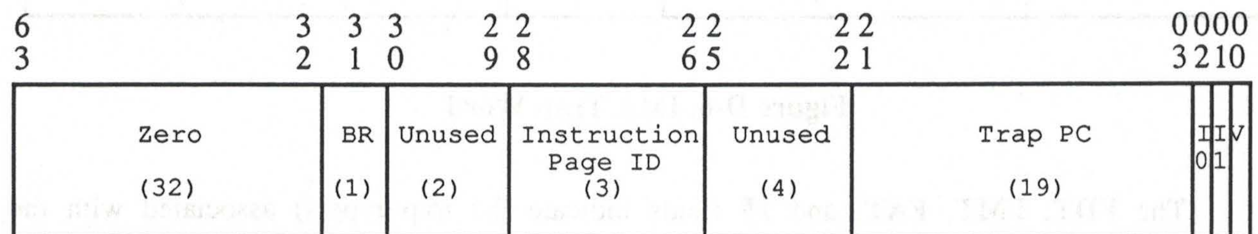


Figure D-4. IMA Trap Locator

Each Trap Locator contains a **Trap PC** and **Instruction Page ID**, **I0** and **I1** issue bits, a valid bit **V**, and zero or more trap type fields. If the **V** bit is not set, the Trap Locator should be ignored. The **Trap PC** and **Instruction Page ID** combine to indicate the address of the instruction word containing the offending instruction (i.e., the location of the trap). The **Trap PC** is simply the low-order address bits of the **PC** that can be mapped by a single instruction page table entry. The **Instruction Page ID** is a 3-bit field which represents one of

the eight instruction page table entries and uniquely identifies the high-order bits of the **PC**. (See the subsection on **Instruction Mapping** in Chapter 2). Each **Divide Trap Locator** has an additional 3-bit field, **ENG,** which indicates the divide/square root engine that executed the trapping instruction. Each **IMA Trap Locator** has an additional one-bit field, **BR,** that indicates whether a branch (i.e., **br, call, jump, exts,** or **ickill**) was executed by the instruction at the corresponding **PC**. Unused bits are provided for architectural expansion.

The **IMA Trap Bits** are a collection of the trap type fields associated with the **IMA Trap Locators,** separated for implementation reasons. Each Trap Locator has a corresponding **IMA Trap Byte** (Figure D-5). The **IMA Trap Bytes** are grouped into a number of **IMA Trap Words** (Figure D-6), each of which holds five **IMA Trap Bytes.**

7 5	4 2	1 0
FMT (3)	FAT (3)	IT (2)

Figure D-5. IMA Trap Byte

63 40	39 32	31 24	23 16	15 8	7 0
Zero (24)	Trap Byte 5 (8)	Trap Byte 4 (8)	Trap Byte 3 (8)	Trap Byte 2 (8)	Trap Byte 1 (8)

Figure D-6. IMA Trap Word

The **FDT, FMT, FAT,** and **IT** fields indicate the trap type(s) associated with the particular **Trap PC/Instruction Page ID** according to the decoding in Table D-3. Note that more than one trap may be associated with a **Trap PC,** hence more than one trap type field may be non-zero. There is no field indicating the type of load/store trap on a per Trap Locator basis because the first load/store trap that occurs freezes the load/store queue. Since only one load/store trap can occur, its type is reported in the Trap Summary.

The **I0** and **I1** fields in each type of Trap Locator help to determine which instruction in the instruction word is responsible for the trap. Table D-4 shows how to decode these fields when the instruction word at the **Trap PC** contains two 32-bit instructions. Table D-5 gives the decoding when one 64-bit instruction is at the **Trap PC**. In the case where there are two 32-bit instructions and both issue bits are set, the determination of which instruction(s) trapped is made by consulting the trap type fields and the opcodes of the two instruc-

tions. For example, if both floating-point add and floating-point multiply traps are reported in the same Trap Locator, then one of the opcodes of the instruction word addressed by the **Trap PC** must be a floating-point add and the other must be a floating-point multiply. If, for example, only an integer trap is reported, then the trapping instruction is whichever of the I0 and I1 instructions is an integer unit instruction. (Consult Appendix C for an explanation of why, when both issue bits are set, it cannot be the case that both the I0 and I1 instructions were executed by the same functional unit.)

Table D-3. Trap Bit Field Encodings

Trap Type	Encoding	Description
FDT, FMT, FAT	0	No trap
	1	Inexact
	2	Overflow
	3	Overflow and inexact
	4	Underflow
	5	Underflow and inexact
	6	Invalid operation (except floating-point compare)
	7	Divide by zero
IT	0	No trap
	1	Integer overflow † (not disabled by ATE bit in PS)
	2	Invalid operation (floating-point compare)
	3	Check trap (**chk** instruction)

† Due to **addt** or **subt** instructions detecting overflow.

Table D-4. I0 and I1 Decoding for 32-bit Instructions

I0	I1	Trapping Instruction
0	0	not possible
0	1	I1
1	0	I0
1	1	I0 and/or I1 †

† If the trap type is DT then only I1 could have caused the trap

Table D-5. I0 and I1 Decoding for a 64-bit Instruction

I0	I1	Trapping Instruction
0	0	none
0	1	not possible
1	0	I0
1	1	not possible

From Tables D-4 and D-5, it is apparent that the execution of an I1 instruction cannot be distinguished from the execution of a 64-bit I0 instruction, except by looking at the format control bits of the I0 instruction. For this reason, programmers should not put garbage in the unused I0 half of an instruction word. (In fact, good programming practice dictates that the I0 half of an instruction word should always contain a valid instruction.)

D.1.1.2.1 IMA Trap Locators

The **IMA Trap Locators** and **IMA Trap Bits** cooperate to maintain the PC, trap type, and issue bits of the last ten non-**Trap State** instruction issue cycles. An instruction issue cycle is defined as a single clock cycle in which one or two instructions are issued. IMA Trap Locator 1 stores information for the oldest such instruction(s) issued (if any), IMA Trap Locator 10 stores information for the youngest. The **IMA Trap Bits** are divided into two **IMA Trap Words**, each of which stores the trap type fields for five **IMA Trap Locators.** Each Trap Word maintains five **IMA Trap Bytes** labelled 1 through 5. IMA Trap Word 0 maintains the trap type information for IMA Trap Locators 1 through 5 in Trap Bytes 1 through 5, respectively. IMA Trap Word 1 maintains the trap type information for IMA Trap Locators 6 through 10 in Trap Bytes 1 through 5, respectively. These ten Trap Bytes are collectively referred to as IMA Trap Bytes 1 through 10, and are numbered identically to the **IMA Trap Locators.**

The Trap Locator in which the first trapping integer, floating-point multiply, and floating-point add unit instruction will be reported is determined by the number of instruction issue cycles that occurred after the trapping instruction is issued and until instruction issuing is suspended (due to the trap). If nine instruction issue cycles (the maximum) occurred, IMA Trap Locator 1 and IMA Trap Byte 1 hold the PC, trap type, and issue bits for the trapping instruction. If no instruction issue cycles occurred after the trapping instruction was issued (and before the trap was detected), IMA Trap Locator 10 and IMA Trap Byte 10 hold the PC, trap type, and issue bits for that instruction. At least one of the three fields: **IT**, **FAT**, or **FMT** will be nonzero in this Trap Byte. The **IT**, **FAT**, and **FMT** fields of all IMA Trap Bytes corresponding to instructions issued before the first trapping instruction will be zero. The **IT**, **FAT**, and **FMT** fields for instructions issued after the first trapping instruction depend on whether the corresponding instructions also trapped. Since one or two instructions

can be issued in a single clock cycle, zero, one or two of the **IT**, **FAT**, and **FMT** fields will be non-zero, indicating which instruction(s) trapped. The ninth clock cycle following the issue of the trapping instruction is the last cycle in which an instruction could have been issued prior to the reporting of traps (and the consequent suspension of instruction issuing).

The **IMAX** field in the Trap Summary indicates the number of the Trap Locator/Trap Byte containing the PC, trap type, and issue bits for the first issued trapping integer, floating-point add, or floating-point multiply instruction. A maximum of 4, 7, or 10 instruction issue cycles can occur after the issue of a trapping integer, floating-point add, or floating-point multiply instruction, respectively.

Once the K-1 exits **Trap State,** all ten IMA Trap Locators are invalidated. Subsequently, if another trap occurs and there have been less than ten instruction issue cycles since exiting **Trap State,** not all IMA Trap Locators will contain valid information. Invalid IMA Trap Locators will always have their V bit clear.

The **I0** and **I1** fields in the IMA Trap Locators serve to identify which instruction(s) (I0, I1, or both) were issued.

For example, in the following program:

```
XX:   move    %r24,%r1;          move    %r25,%r2
      move    %r26,%r4;          move    %r27,%r5
      add.d   %r1,%r2,%r3 ;      add.d   %r4,%r5,%r6
      mult.d  %r3,%r7,%r8 ;      add.d   %r9,%r10,%r11
      addt    %r12,%r13,%r14;    addt    %r15,%r16,%r17
      move    %r20,%r21 ;        nop
```

if we assume that all six **add.d**, **mult.d**, and **addt** instructions generate overflow traps, then the IMA Trap Locators/Trap Bytes will be as follows:

IMA Trap Locator	IT	FAT	FMT	Trap PC	I0	I1
1	0	0	0	XX	1	0
2	0	0	0	XX	0	1
3	0	0	0	XX+8	1	0
4	0	0	0	XX+8	0	1
5	0	2	0	XX+16	1	0
6	0	2	0	XX+16	0	1
7	0	2	2	XX+24	1	1
8	1	0	0	XX+32	1	0
9	1	0	0	XX+32	0	1
10	0	0	0	XX+40	1	1

Note: IMAX = 5

D.1.1.2.2 L/S Trap Locators

The eight **L/S Trap Locators** correspond to the maximum of eight possible load/store instructions that could have been issued but not completed at the time of a load/store trap. All of these instructions will have been stopped before altering either general registers or memory.

If there is no load/store trap reported in the Trap Summary, all L/S Trap Locators should be ignored (because their **V** bits will be off). If there is a load/store trap reported in the Trap Summary, then L/S Trap Locator 1 will give the **Trap PC/Instruction Page ID** and issue bits of the trapping instruction. L/S Trap Locators 2 through 8 will give the same information for the following load/store instructions that were issued (up to seven). Note that no information is delivered about whether or not these instructions would have trapped had they been allowed to complete. Part of the software simulation required in a trap handler is to determine if these instructions would have trapped, and if so, to simulate the appropriate load/store traps. If fewer than eight load/store instructions were in the load/store queue at the time of the load/store trap, then fewer than eight L/S Trap Locators will be valid.

D.1.1.2.3 Divide Trap Locators

The three **Divide Trap Locators** correspond to the two possible divide and one possible square root instructions that could be outstanding at the same time. Any number from zero to three simultaneous divide/square root traps are possible. A Divide Trap Locator, if valid, gives the location, trap type, issue bits, and engine number for one divide/square root instruction.

Divide Trap Locator 1, if valid, corresponds to the oldest trapping divide/square root instruction, Divide Trap Locator 2 to the next oldest, and Divide Trap Locator 3 to the youngest. One should note that the number of valid Divide Trap Locators will be equal to the number of trapping divide/square root instructions, something which is not true for either the IMA Trap Locators or the L/S Trap Locators. In addition, because of the variable latencies of the divide and square root instructions, the first divide instruction to trap may not be the oldest outstanding divide instruction.

D.1.2 Reading Primary Trap Data

Primary Trap Data is read one word at a time, using a sequence of **rtrpd** instructions. The *srca* operand of **rtrpd** controls which type of primary trap data is read as shown in Table D-6. Use of values for *srca* not shown in this table will lead to unpredictable results. With the exception of the **Trap Summary** and the **IMA Trap Words**, which can be read any number of times while in **Trap State**, Primary Trap Data can only be read once while in **Trap State** (i.e., the read operation is destructive) and must be read out in a pre-defined, type-dependent order.

The Restart PCs can only be read once while in **Trap State,** and they are read in the following order: First Restart PC, Second Restart PC, and Third Restart PC.

Table D-6. rtrpd *srca* encoding

srca value	Trap Data Read
0	Trap Summary
1	Restart PCs
2	IMA Trap Word 0
3	IMA Trap Word 1
4	IMA Trap Locators
5	L/S Trap Locators
6	Divide Trap Locators
> 6	Unpredictable results

The **L/S Trap Locators** can only be read once while in **Trap State**. The first entry read is L/S Trap Locator 1, which corresponds to the first (if any) trapping L/S instruction. Subsequent entries are read sequentially with L/S Trap Locator 8 read last.

The **Divide Trap Locators** also can only be read once while in **Trap State**. The first entry read is Divide Trap Locator 1; subsequent entries are read sequentially with Divide Trap Locator 3 being read last.

Similarly, the **IMA Trap Locators** can only be read once while in **Trap State**. The first entry read is IMA Trap Locator 1; subsequent entries are read sequentially with IMA Trap Locator 10 read last. IMA Trap Words 0 and 1, as indicated in Table D-6, are accessed separately using different *srca* arguments. As indicated earlier, the **IMA Trap Words** can be read any number of times while in **Trap State**.

D.1.3 Load/Store Trap Data

In the event of a trap, the K-1 attempts to empty all functional unit pipelines before starting the trap handler. This happens automatically for arithmetic pipelines by waiting enough cycles for any outstanding operations to complete. However, since load/store instructions must complete in order, if one load/store instruction traps, no load/store instructions after that can be allowed to finish. Therefore, the load/store pipeline may not be empty if there is a load/store trap.

There can be as many as eight load/store instructions in progress when a load/store trap is detected. The load/store trap data contains enough information about all of the instructions in the load/store pipeline, including the one that trapped (which will always be at the head of the load/store queue), to allow a software routine to recover from the trap by simulating these already-issued, but not executed, instructions. The type of load/store trap, if any, is contained in the Trap Summary. If there was no load/store trap, then the information contained in the load/store trap data will not be valid and should be ignored.

Note that it would not be correct to restart the program at the PC of the trapping load/store instruction (assuming that the cause of the trap had been rectified). Any non-

load/store instruction that appeared in the program between the trapping load/store instruction and the first Restart PC will have already completed before the trap. Restarting the program in this fashion would cause such non-load/store instructions to be re-executed, producing invalid results. For example, an integer instruction issued shortly after a trapping load/store instruction will complete *before* the load/store trap is detected. Execution could not be resumed at the location of the trapping load/store instruction without re-executing the integer instruction. This is why the instructions in the load/store pipeline at the time of a load/store trap must be software simulated.

For each of the eight possible load/store instructions in the load/store pipeline at the time of a load/store trap, certain trap data must be saved. This data, called **Load/Store Trap Data**, usually contains the address generated by the load/store instruction and other control information including the opcode and the destination register (if any). The **slstrpd** instruction is used to write this data to memory. For certain instructions, the address contained in Load/Store Trap Data is replaced by other information. The format of **Load/Store Trap Data** is shown in Figure D-7.

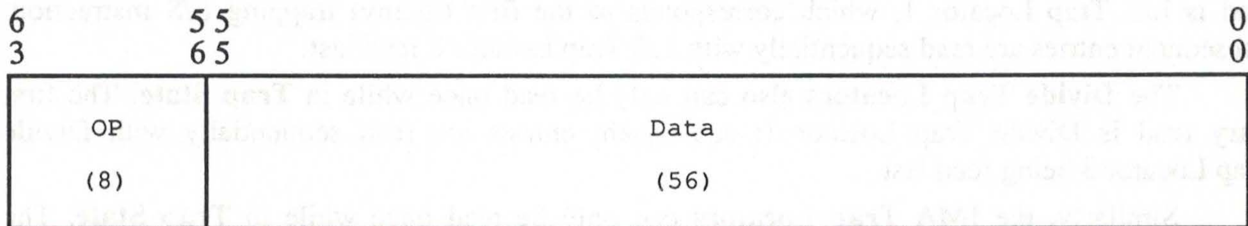


Figure D-7. Load/Store Trap Data

In Figure D-7, **OP** corresponds to the field of the same name in the instruction formats. It is possible that there were fewer than eight instructions in the load/store pipeline at the time of the trap. This situation is indicated by delivering zero in the **OP** field for those entries that correspond to the unfilled load/store queue slots. The unused entries will always be in a single group at the end of the **Load/Store Trap Data.**

For most load/store instructions, the **Data** field is interpreted in ***addr*** format as shown in Figure D-8. **RC** corresponds to the field of the same name in the instruction formats and will only be valid if the instruction uses that field. Bits 31..18 of the **Data** field are undefined, and the concatenation of bits 49..32 and 17..0 is the virtual address, **Addr**, referenced by the instruction. This interpretation of the **Data** field holds for all **load, eload, loadcpu, store, storecpu, ldecc, ldnecc, zcl, pcl, swat, wps, echk, relf, rfec,** and **wfec** instructions. For instructions such as **rfec** and **wfec**, which do not compute an address, the **Addr** field is identical to *srca* of the instruction.

For each **store** or **swat** instruction in the load/store pipeline at the time of a load/store trap, the K-1 remembers the data that was to be stored by that instruction. This data is kept in a sequential list, each entry of which (called **Load/Store Trap Store Data**) may be written directly to memory with the **slstrpd** instruction. The first entry in the list corresponds to the first **store** or **swat** instruction, the second entry corresponds to the second **store** or **swat** instruction, etc.

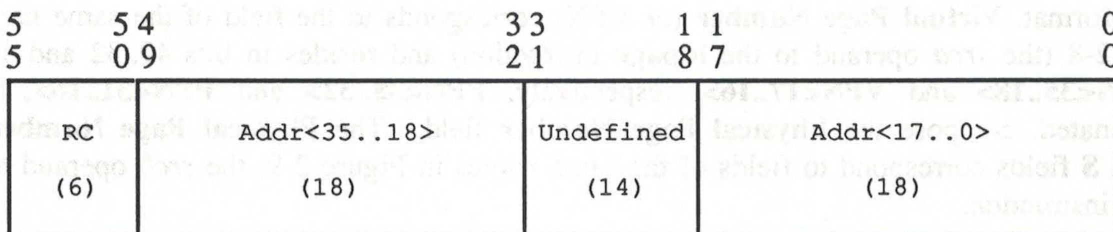


Figure D-8. *addr* Data Field Format

Note that for stores of less than 8-byte precision (i.e., byte, halfword, and word), only the least significant part of the Load/Store Trap Store Data for that instruction contains meaningful data. In other words, the data saved is not identical to the contents of *srcc* of the **store** instruction, except for 8-byte **stores** and **swat**.

For all **welf, wdwp, dflush, rios**, and **wios** instructions, the **Data** field is interpreted in **combined** ***srca/srcb*** format as shown in Figure D-9.

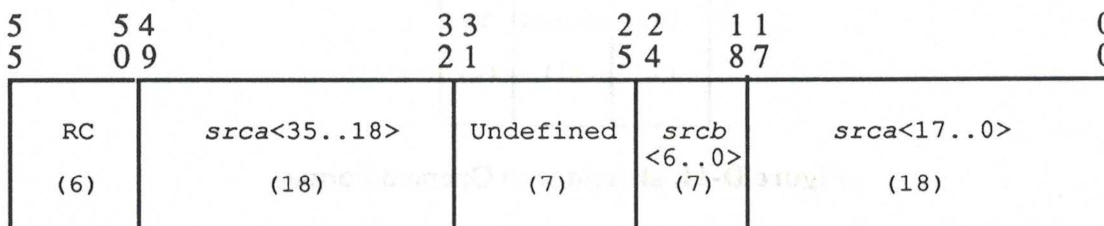


Figure D-9. Combined *srca/srcb* Data Field Format

In this format, **RC** has the same definition as in the *addr* format. The low-order 36 bits of the instruction's *srca* operand are packed into bits 49..32 and 17..0. The low-order 7 bits of its *srcb* operand reside in bits 24..18. *srcb*<6> is only defined for **rios** and **wios**. Bits 31..25 are undefined. These 36 bits of *srca* and 7 bits of *srcb* are provided regardless of the presence or size of the instructions' operands.

The **Data** field for the **ldpage** instruction is interpreted in **ldpage** format as shown in Figure D-10.

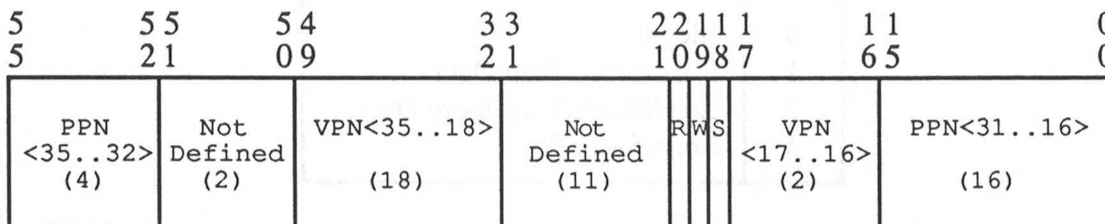


Figure D-10. ldpage Data Field Format

In this format, **Virtual Page Number** (or **VPN**) corresponds to the field of the same name in Figure 2-8 (the *srca* operand to the **ldpage** instruction) and resides in bits 49..32 and 17..16 as **VPN<35..18>** and **VPN<17..16>**, respectively. **PPN<35..32>** and **PPN<31..16>**, when concatenated, compose the **Physical Page Number** field. The **Physical Page Number**, **R**, **W**, and **S** fields correspond to fields of the same names in Figure 2-9, the *srcb* operand to the **ldpage** instruction.

D.1.4 Storing Load/Store Trap Data

Load/Store Trap Data is stored, one 64-bit word at a time, using a sequence of **slstrpd** instructions. Load/Store Trap Data can be stored any number of times while in **Trap State**. The *srca* operand of **slstrpd** specifies where in memory to write **Load/Store Trap Data**. The *srcb* operand selects which **Load/Store Trap Data** to store as shown in Figure D-11.

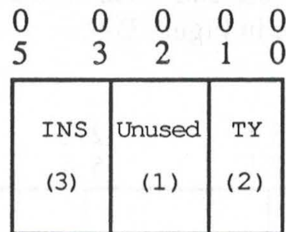


Figure D-11. slstrpd *srcb* Operand Format

The **TY** field controls which unit (**Load/Store Trap Data, Load/Store Trap Store Data**, or **Load/Store PS**) of **Load/Store Trap Data** to store, and the **INS** field selects the instruction in the load/store pipeline whose data is stored. The encoding of the **TY** field is given in Table D-7. For Load/Store Trap Data, the **INS** field has a simple binary encoding, with zero referring to the oldest instruction in the load/store pipeline, and seven referring to the youngest.

Table D-7. TY Encoding

TY	Unit of Load/Store Trap Data
0	Illegal
1	Load/Store Trap Data
2	Load/Store Trap Store Data
3	Load/Store PS

When the **TY** field selects Load/Store Trap Store Data, the **INS** field specifies which **store** or **swat** instruction's store data should be written to memory. A value of zero indi-

cates the first **store** or **swat** in the load/store queue, a value of one indicates the second, etc. For example, if only the third and fifth instructions in the load/store pipeline at the time of a trap were **store** or **swat** instructions, then only the first two Load/Store Trap Store Data words (**INS** = 0 and **INS** = 1) would be meaningful.

If the **TY** field is 3, then the **INS** field is ignored and **slstrpd** returns certain bits of the **Processor Status** register that were in effect at the time of the load/store trap. Since one or more **wps** instructions may have been in the load/store pipeline, these bits may not be the same as the *current* **Processor Status**. The trap recovery routine should use this **load/store PS** while simulating instructions up to the first **wps** instruction encountered in the load/store pipeline (if any). Upon encountering each **wps** instruction, the trap recovery routine should use the new PS (from the **wps** *srca* operand) for further simulation. The format of the load/store PS is the same as that returned by an ordinary **rps** instruction, except that only the following fields are defined:

Byte Order Low-to-High
Process Key<12..0>
User Protection
User Mode Store
User Mode Load
Small Address Compatibility Mode
Early Load Alignment Trap

D.2 Use of exts Instructions

The **exts** instruction is used to reload the Restart PCs and return from a trap. **exts** must be used in a very stylized manner to correctly restart the trapping program and exit from **Trap State**. Once the first **exts** is executed, two more **exts**'s must be *immediately* executed with no intervening instructions (enabled or disabled). Three sequential **exts** instructions are necessary to reload the three **Restart PCs**. *Before* the instruction at the address of the first **Restart PC** is executed, the processor will have exited **Trap State**. Since **exts** is restarting a trapping program, it both provides a way to return to the program, and to reload the registers needed to hold the Restart PCs. See the **exts** instruction description for more details.

D.3 Example Trap Handler

```
# include "~chris/notes/trap_state.s
```

Appendix E. Memory System Specifics

E.1 Introduction

The K-1 memory system is designed for very high memory bandwidths so that it can support a large number of high-speed processors and I/O Controllers. To obtain sufficient bandwidth, the datapath to/from the memory system is very wide, and the memory system itself is banked and pipelined. A request to the memory system will "tie up" one bank of RAMs during their access time, but will not tie up the entire memory system (so that other memory requests to different banks can be started in parallel). The number of memory requests that can be done in parallel in the initial implementation of the K-1 is four.

The memory itself is protected by an Error-Correcting Code (ECC). This code will correct any single-bit error in a 64-bit word, and will detect any double-bit error in a 64-bit word. All errors (both single- and double-bit) are also logged so that the Console can detect faulty memory and remove it from service. Note that the memory system's idea of which bytes in a cache line comprise a "word" differs significantly from the processor's. (See the section on **Data Order** for more details).

This appendix discusses the memory system from the point of view of a diagnostic programer.

At the time this manual went to press, this chapter was still incomplete.

E.2 Data Order

Due to the extreme width of the memory datapath in the K-1, the data between the processors and the memory system is *sliced* in an implementation dependent way. Certain bytes are grouped together logically in the memory system in a different fashion than the grouping in the processor. All discussions of data order will refer to cache lines and cache sub-lines, as defined in the section on **Cache Effects** in Chapter 2.

All data transfers from/to a CPU to/from the memory system consist of one cache line made up of four cache sub-lines. The cache sub-lines are always consecutive pieces of the cache line. The sub-line transferred first depends on the address of the reference. For data cache misses and writebacks, and for cache coherency responses, the sub-line containing the referenced word will always be transferred first. For instruction cache misses, the first sub-line (the one with the most low-order zero bits in its address) is always transferred first.

Within each sub-line, the bytes which are covered by common ECC bits in the memory system are not the obvious groups of consecutive bytes one would expect. If the bytes in a sub-line are numbered from 0 to 63, then bytes 0, 1, 8, 9, 16, 17, 23, and 24 are covered by the same ECC bits. That is, any single-bit error in these eight bytes will be corrected, and any double-bit error in these eight bytes will be detected. This pattern repeats seven more times in a sub-line, so that bytes 2, 4, 6, 32, 34, 36, and 38 start eight-byte groups with the same relative offsets. These ECC groupings within a sub-line are illustrated in Figure E-1.

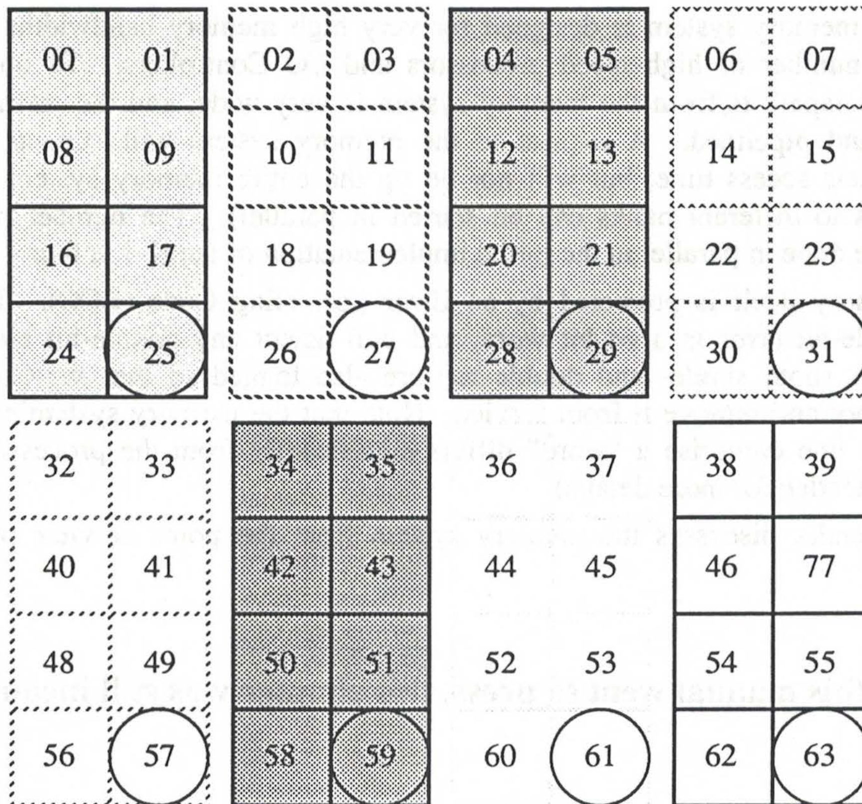


Figure E-1. ECC Groupings Within a Sub-Line

In Figure E-1, the rows of data are eight sequential bytes from the CPU's point of view (and the numbering shown is the byte address within a sub-line). The eight, shaded regions represent the groups of bytes that are covered by the same ECC bits. The circled locations show the positions used for reading and writing the ECC bits within each group using the **ldecc** instruction and the special ECC-writing form of **dflush**.

E.3 Error-Correcting Code

The memory system of the K-1 uses a **SECDED** (Single-Error Correcting, Double-Error Detecting) code. Mathematically, the code would be called an extended Hamming code. The overhead for this code is one byte for each 64 bits.

The code ...

The check bits can be calculated as...

E.4 CPU Features for Diagnosing the Memory System

uncorrectable memory error traps

ldecc

ldnecc

dflush

E.5 Console Interactions and Error Logging

MATR

error logging

E.6 Cache Coherency

Cache coherency

Bandwidth – normal and coherency

E.7 Memory System Timing

The memory system runs at a clock rate that is one third of the processor's clock. If there is no contention, it takes the memory system approximately 16 memory clock cycles to respond to a request from a processor.

From the time when the processor detects a miss in the data cache for a **load** instruction to the time when the result is written into the register file, approximately 52 processor clock cycles elapse. The load/store unit is tied up for an additional (approximately) 10 cycles while the rest of the data (the other sub-lines) are written into the data cache.

Note that shared requests, which use the cache coherency scheme, can take longer since the memory system waits for all processors to respond before it returns the data.

Appendix F. I/O System Specifics

F.1 Introduction

The I/O system of the K-1 provides very high bandwidths by using a large number of independent I/O channels, each of which is capable of very high I/O bandwidths. The K-1 system supports a number of I/O Controllers (IOCs), which interface to the memory system in much the same way that a processor does. In the initial implementation, there can be one or two IOCs, each IOC can support up to eight I/O Channel Adapters (IOCAs). Each IOCA can support up to eight I/O channels. The I/O channels themselves are compatible with the High-Speed Channel (HSC) standard, and operate full-duplex at speeds up to 100 Megabytes per second.

The **rios** and **wios** instructions provide communication with the IOCs and the IOCAs. Status information can be read from the I/O system with a **rios** instruction. The **wios** instruction can be used to change control settings, to initiate I/O operations, and for multiprocessor synchronization operations.

Table F-1 provides a summary of the **rios** and **wios** instructions. Section 2 describes in detail the operation of the **wios** and **rios** instructions. Section 3 explains the format of the control and status registers used in the I/O system.

Table F-1. rios/wios Instruction Summary

		TYPE *srca* <35..32>	DESCRIPTION	ADDR *srcb* <6..0>	SDATA *srca* <31..0>	RDATA *rdst* <31..0>
WIOS	IOCA	0000	System Service Request	IOC,IOCA<2..0>,CH<1..0>,X	Don't care	
WIOS	IOCA	0001	Write Channel Control Register	IOC,IOCA<2..0>,CH<1..0>,*	CCR Data 4 x (8)	
WIOS	IOCA	0010	Write Output Channel I-Field Register	IOC,IOCA<2..0>,CH<1..0>,X	Out I-Field (32)	
WIOS	IOCA	0011	Write Sys. Srvc. Req. Packet Pointer	IOC,IOCA<2..0>,CH<1..0>,X	SSRP Pointer	
WIOS	IOCA	0100	Channel Reset	IOC,IOCA<2..0>,CH<1..0>,X	Don't care	
WIOS	IOCA	0101	IOCA Reset	IOC,IOCA<2..0>,XXXX	Don't care	
WIOS	IOCA	0110	-----------			
WIOS	IOCA	0111	-----------			
WIOS	IOC	1000	Write CRP Register File	CPU<2..0>, LVL<3..0>	Q Address(32)	
WIOS	IOC	1001	-----------			
WIOS	IOC	1010	-----------			
WIOS	IOC	1011	-----------			
WIOS	IOC	1100	Interprocessor Interrupt	CPU<2..0>, XXXX	Don't care	
WIOS	IOC	1101	Set NCRP CPU	CPU<2..0>, XXXX	Don't care	
WIOS	IOC	1110	-----------			
WIOS	IOC	1111	I/O Subsystem Reset	Don't care	Don't care	
RIOS	IOCA	0000	Read Channel Status Register	IOC,IOCA<2..0>,REG<1..0>,X	Don't care	Ch. Status 4 x (8)
RIOS	IOCA	0001	-----------			
RIOS	IOCA	0010	Read Input Channel I-Field Register	IOC,IOCA<2..0>,CH<1..0>,X	Don't care	Input I-Field (32)
RIOS	IOCA	0011	Read Sys. Srvc. Req. Packet Pointer	IOC,IOCA<2..0>,CH<1..0>,X	Don't care	SSRP Pointer
RIOS	IOCA	0100	Read IOCA Status Register	IOC,IOCA<2..0>,REG<1..0>,X	Don't care	IOCA Status 4 x (8)
RIOS	IOCA	0101	-----------			
RIOS	IOCA	0110	-----------			
RIOS	IOCA	0111	-----------			
RIOS	IOC	1000	Read CRP Register File	CPU<2..0>, LVL<3..0>	Don't care	Q Address(32)
RIOS	IOC	1001	Read CRP Register File (clr LVL)	CPU<2..0>, LVL<3..0>	Don't care	Q Address(32)
RIOS	IOC	1010	Read MPSync flags (clr bits)	CPU<2..0>, LVL<3..0>	Bit mask(32)	MPSync flags(32)
RIOS	IOC	1011	Read MPSync flags (set bits)	CPU<2..0>, LVL<3..0>	Bit mask(32)	MPSync flags(32)
RIOS	IOC	1100	-----------			
RIOS	IOC	1101	-----------			
RIOS	IOC	1110	-----------			
RIOS	IOC	1111	-----------			

* "All-channels" flag

F.2 Instruction Descriptions

F.2.1 wios

Instruction: **wios** *srca,srcb,rdst*

Opcode: FD

Operation: Write I/O Status.

Operands used: *srca, srcb*

Results stored: *rdst*

Legal In: Supervisor mode only

Functional unit: Load/Store

Exceptions Illegal instruction/privilege violation

Description:

Four bits of the **wios** (Write I/O Status) instruction's *srca* operand (*srca*<35..32>) define the operation TYPE field. The *srcb* operand ADDR field (*srcb*<6..0>) supplies an address whose contents are dependent on the value contained in the TYPE field. The *srca*<63:36> and *srcb*<63..7> bit values currently are don't care and are ignored. The fields are shown in Figure F-1.

The different TYPE operations can be divided into two groups; those directed to the IOC and those directed to an IOCA. These two groups are indicated by the MSB of the TYPE field in the *srca* operand (*srca*<35>). A **wios** operation with *srca*<35>=0 is directed at a channel on an IOCA, and with *srca*<35>=1 is directed to an IOC.

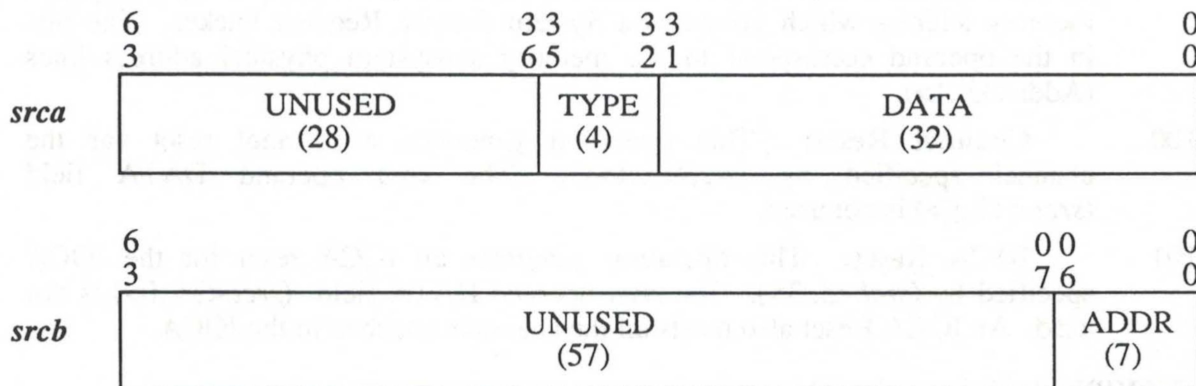


Figure F-1. ***srca*** **and** ***srcb*** **Operand Format**

The operations performed by the different TYPE designations are System Service Request, Write Channel Control Register, Write Output Channel I-Field Register, Write System Service Request Packet Pointer, Channel Reset, IOCA Reset, Write CRP Register File, Interprocessor Interrupt, Set NCRP CPU, and I/O Subsystem Reset. The description of each operation and the contents of the *srca* and *srcb* operands follows below.

IOCA-TYPE *wios* Instruction Operations

For all IOCA-type **wios** instructions, the *srcb* operand ADDR field specifies the address of the channel that processes the instruction. The channel address can be divided into three sub-fields: one IOC select bit (*srcb*<6>), three IOCA select bits (*srcb*<5..3>) and two channel select bits (*srcb*<2..1>). The LSB of the ADDR field (*srcb*<0>) is not currently used (and its value is ignored). If there is only one IOC in the system, the IOC select bit is checked against a configuration parameter in the IOC, and if there is a discrepancy, an error is detected.

Type	Description
0000	**System Service Request:** This operation indicates a System Service Request Packet is ready to be read from system memory. The *srca* operand DATA field (*srca*<31..0>) is not used.
0001	**Write Channel Control Register:** This operation writes one byte of the 32- bit *srca* operand DATA field (*srca*<31..0>) into the 8-bit Channel Control Register. The most significant byte is used for Channel 0, the next most significant byte for Channel 1, etc. The two channel select bits (*srcb*<2..1>) select the channel to be written. The definitions of the channel control bits and their functions are described later in section 3.
0010	**Write Output Channel I-Field Register:** This operation writes the *srca* operand DATA field (*srca*<31..0>) into the Output Channel I-Field Register. The DATA field is a 32-bit value which is presented on the output HSC channel during a connection request sequence. The function of the data contents is implementation specific.
0011	**Write System Service Request Packet Pointer:** This operation writes the *srca* operand DATA field (*srca*<31..0>) into the System Service Request Packet Pointer (SSRP Pointer). The DATA field is a 32-bit system memory address which points to a System Service Request Packet. The bits in the operand correspond to the memory subsystem physical address lines (Addr<35..4>).
0100	**Channel Reset:** This operation generates a channel reset for the channel specified by (*srcb*<2..1>). The *srca* operand DATA field (*srca*<31..0>) is not used.
0101	**IOCA Reset:** This operation generates an IOCA reset for the IOCA specified by (*srcb*<5..3>). The *srca* operand DATA field (*srca*<31..0>) is not used. An IOCA Reset also resets all the channels attached to the IOCA.

IOC-TYPE *wios* Instruction Operations

There are four different types of operations performed by the **wios** instruction to an IOC: Write CRP Register File, Interprocessor Interrupt, Set NCRP CPU, and I/O Subsystem

reset. The ADDR field is further subdivided into 2 fields, CPU=*srcb*<6..4> and LVL=*srcb*<3..0>. The most significant CPU bit (*srcb*<6>) determines the IOC address. If this bit is 0, then the operation is destined for IOC(0); if it's a 1, then the operation is destined for IOC(1). IOC(0) is physically connected to CPU(3..0) and IOC(1) is physically connected to CPU(7..4). The contents of the *srca* and *srcb* operands, and a description of each type of operation follows:

Type	Description
1000	**Write CRP Register File:** This operation writes to the CRP register file on the IOC. The CRP register file contains the Queue Address Pointers (LVL 8-14), the NCRP pointers (LVL 5), and the MPSync flags (LVL 0-3). The location, to be written, is addressed by the CPU (*srcb*<6..4>) and LVL (*srcb*<3..0>) variables contained in the ADDR field of the *srcb* operand. All Write CRP Register File operations **are** allowed to occur across IOCs. Figure F-2 illustrates the format of the CRP and NCRP pointers.

For LVL = 8-14, the operation will update the Response Packet (CRP) Queue Pointer from the DATA field (*srca*<31..0>) of the *srca* operand. **NOTE: The entire 32-bit value written, will appear in the 1st CRP link field. Subsequent CRP link fields will contain the previous packet's 30-bit memory address.** (CRP linkage forces the 2 LSBs to be loaded with zero due to the size of the I/O Bus Address)

For LVL = 5, the operation will update the Negative Response Packet (NCRP) Queue Pointer from the DATA field (*srca*<31..0>) of the *srca* operand. **NOTE: The entire 32-bit value written, will be used as the system address of the 1st NCRP. Even though all 32 bits are stored in the register file, the 4 lowest bits will not be used, since NCRP are 256-byte relative. Therefore the 4 low order bits should be considered as don't care other than for diagnostics. Subsequent NCRP addresses will be generated automatically in the IOC by incrementing the 4 bit address field <11..8> modulo 256 bytes, shown above. This field is independent of the actual NCRP counter value; so the NCRP queue may exist on any 64-byte boundary with a maximum packet size of 256 bytes.** When directed at the NCRP CPU location, this operation type, will cause an IOC internal flag, which blocks the writing of response packets to memory, to be

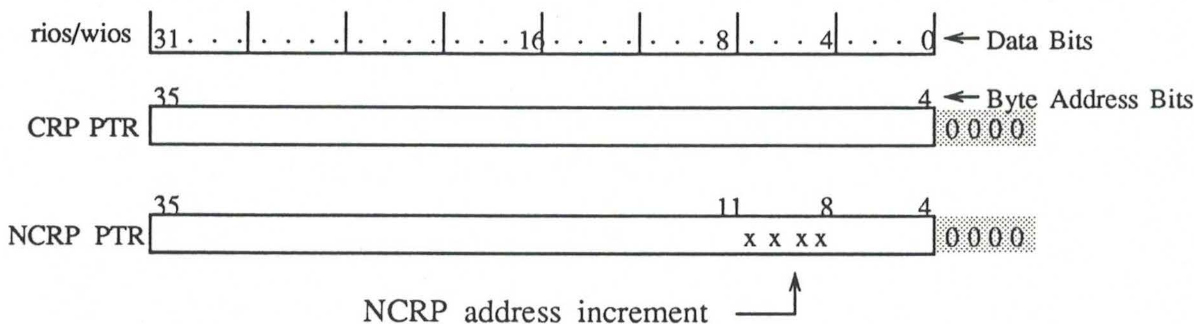


Figure F-2. CRP and NCRP Pointer Structure

reset (thus enabling NCRP writes to memory). This operation also clears the IOC internal NCRP counter to zero.

Non-aligned accesses for NCRPs wraparound a line in memory.

For LVL = 0-3, the operation will write the DATA (*srca*<31..0>) field of the *srca* operand into the MPSync locations. **NOTE: Do not use this TYPE for MPSync accesses, since consistency is not maintained. It is provided for Diagnostics only.**

For all other values of LVL, the operation will complete normally and write the location specified, although these locations are not currently used by the IOC.

1100 **Inter-Processor Interrupt:** This operation type will cause an interrupt at level 2 to be generated to the CPU indicated. The operand *srcb* defines the CPU (*srcb*<6..4>) which will be interrupted. These interrupts **are** also allowed to CPUs which cross IOCs.

1101 **Set NCRP CPU:** This operation type sets a pointer in **BOTH** IOCs to indicate; which CPUs' queue to put an NCRP (Negative Channel Response Packet) into, and then interrupt at level 5. The CPU indicated in the ADDR field (*srcb*<6..4>) of the *srcb* operand is latched as the CPU number to which an NCRP is directed.

This operation also clears both IOC's NCRP counters to zero. Therefore, when changing NCRP CPUs, be sure that the LVL5 pointer location for that CPU is properly initialized to handle 16 more NCRPs. **Any** CPU may set these registers.

1111 **I/O Subsystem Reset:** This operation type issues a reset to ***both*** IOCs and ***all*** IOCAs; which will completely clear control state in all IOCs, IOCAs, and channels. The I/O subsystem is then ready to resume normal functionality. The normal use of this operation is for error recovery.

F.2.2 rios

Instruction:	**rios** *srca,srcb,rdst*
Opcode:	FE
Operation:	Read I/O Status
Operands:	*srca, srcb*
Results stored:	*rdst*
Legal In:	Supervisor mode only
Functional unit:	Load/Store
Exceptions:	Illegal instruction/privilege violation

Description:

The **rios** (Read I/O Status) instruction is used to check status of the I/O subsystem. The *srca* and *srcb* operands use the same field definitions as the **wios** instruction (Figure F-1). The *srca* operand data (*srca*<31..0>) is used only by the IOC for MPSync operations. The result returned in *rdst* is dependent on the TYPE field.

As for the **wios** instruction, the MSB of the TYPE field (*srca*<35>) indicates whether the **rios** operation is directed to the IOCA or to the IOC.

The operations performed by the different TYPE designations are Read Channel Status Register, Read Input Channel I-Field Register, Read System Service Request Packet Pointer, Read IOCA Status Register, Read CRP Register File, Read CRP Register File and Clear Level, Clear MPSync bits, and Set MPSync bits. The description of each operation and the contents of the *srca* and *srcb* operands and the *rdst* result follows below.

IOCA-TYPE *rios* Instruction Operations

There are four different **rios** operations that are directed to the IOCAs: Read Channel Status Register, Read Input Channel I-Field Register, Read System Service Request Packet Pointer, and Read IOCA Status Register. For all IOCA-type **rios** instructions, the *srcb* operand ADDR field specifies the address of the channel that processes the instruction. The channel address can be divided into three sub-fields: one IOC select bit (*srcb*<6>), three IOCA select bits (*srcb*<5..3>) and two channel select bits (*srcb*<2..1>). The LSB of the ADDR field (*srcb*<0>) is not used (and its value is ignored). If there is only one IOC in the system, the IOC select bit is checked against a configuration parameter in the IOC, and if there is a discrepancy, an error is detected.

Type	Description
0001	**Read Channel Status Register:** This operation reads one byte from each of the four 32-bit Channel Status Registers and returns the 32-bit value to *rdst*. The channel select bits (*srcb*<2..1>) select the proper byte. The definition of the channel status bits and their functions are described in section 3. The format of *rdst* is shown in Figure F-3.

As an example, when a Read Channel Status Register *rios* is issued with *srcb*<2..1> = '00'; CSR Byte 0 is requested. The returned data will be as follows:

rdst Byte 0 - HSC 0, CSR Byte 0

rdst Byte 1 - HSC 1, CSR Byte 0

rdst Byte 2 - HSC 2, CSR Byte 0

rdst Byte 3 - HSC 3, CSR Byte 0

Likewise, when *srcb*<2..1> = '01', Byte 1 will be returned from the CSRs of all the channels.

0010 **Read Input Channel I-Field Register:** This operation reads the Input Channel I-Field Register and returns the 32-bit value to *rdst*. The I-Field data is received from the input HSC during a connect sequence. The function of the data contents is implementation specific. The format of *rdst* is shown in Figure F-4.

0011 **Read System Service Request Packet Pointer:** This operation reads the System Service Request Packet Pointer (SSRP Pointer) and returns the 32-bit value to *rdst*. The SSRP Pointer is a system memory address for the most recently executed System Service Request Packet. The bits in the

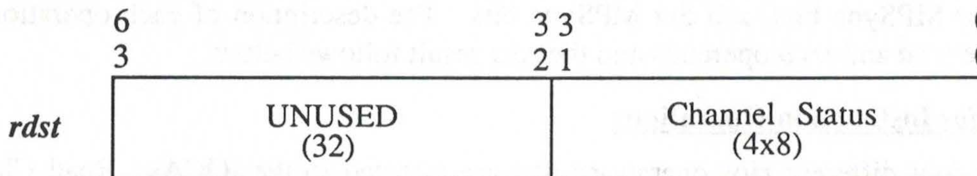


Figure F-3. Type 0001 *rdst* Register Format

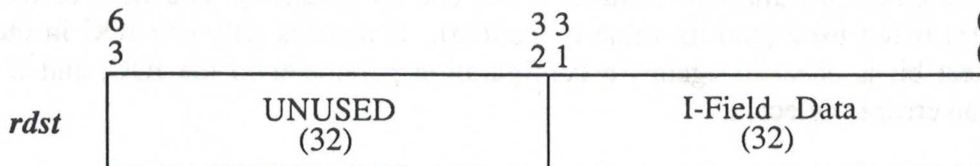


Figure F-4. Type 0010 *rdst* Register Format

operand correspond to the memory physical address lines (Addr<35..4>). The format of *rdst* is shown in Figure F-5.

0100 **Read IOCA Status Register:** This operation reads one of the four IOCA Status Registers and returns the 32-bit value to *rdst*. The two channel select bits (*srcb*<2..1>) are used to select the proper IOCA register. The definition of the IOCA status bits and their functions are described later in this section. The format of *rdst* is shown in Figure F-6.

IOC TYPE *rios* Instruction Operations

There are four different **rios** operations that are directed to the IOC: Read CRP Register File, Read CRP Register File and clear interrupt, MPSync and clear bit, and MPSync and set bit.

Type **Description**

1000 **Read CRP Register File:** This operation type reads the CRP register file on the IOC. The CRP register file contains the Response Packet (CRP) Queue Address Pointers (LVL 8-14), the NCRP pointers (LVL 5), and the MPSync flags (LVL 0-3). The location to be read, is addressed by the CPU (*srcb*<6..4>) and LVL (*srcb*<3..0>) variables contained in the ADDR field of the *srcb* operand. Interrupt levels to all CPUs remain unchanged. All read CRP Register File operations **are** allowed to occur across IOCs.

For LVL = 8-14, the operation will read the Response Packet (CRP) Queue Pointer. The *rdst* will contain the address stored in the register file for

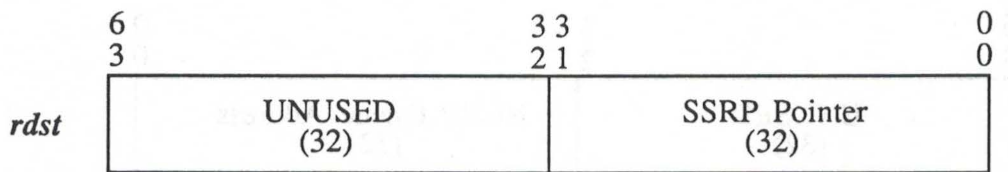


Figure F-5. Type 0011 *rdst* Register Format

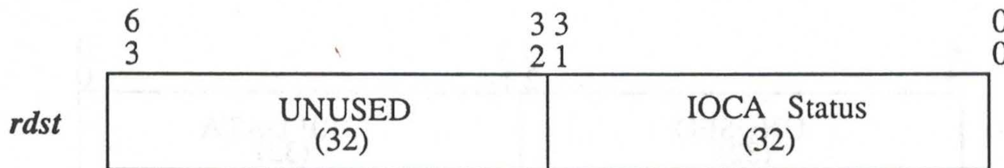


Figure F-6. Type 0100 *rdst* Register Format

the last response packet written to memory (This reflects the upper 32 bits of the byte address<35..4>). The format of *rdst* is shown in Figure F-7.

For LVL = 5, the operation will read the Negative Response Packet (NCRP) Queue Pointer. The *rdst* will contain the address of the next NCRP. The format of *rdst* is shown in Figure F-8.

For all other values of LVL, the operation will read all 32 bits of the data. **SOFTWARE NOTE: Do not use this TYPE for MPSync accesses. It is provided for Diagnostics only.** The format of *rdst* is shown in Figure F-9.

1001 **Read CRP Register File:** Same as ***rios*** 1000 above, except that for any location associated with an interrupt level (2-14), that interrupt level will be cleared.

When directed at the NCRP CPU location at LVL = 5, this operation type, will cause an IOC internal flag, which blocks the writing of response packets to memory, to be set (thus inhibiting NCRP writes to memory).

1010 **Read (and clear) MPSync:** This operation type performs an atomic read-modify-write of the MPSync locations. For each IOC, there are 16 MPSync locations in the CRP Register File and one external to the Register

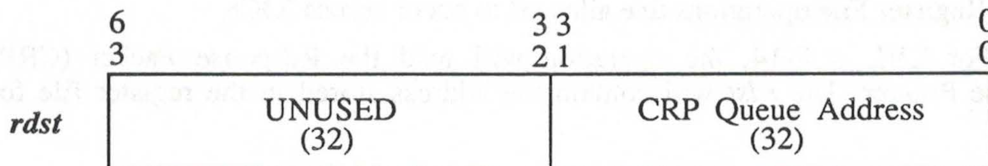


Figure F-7. Type 100x *rdst* Register Format
CRP Queue Pointers

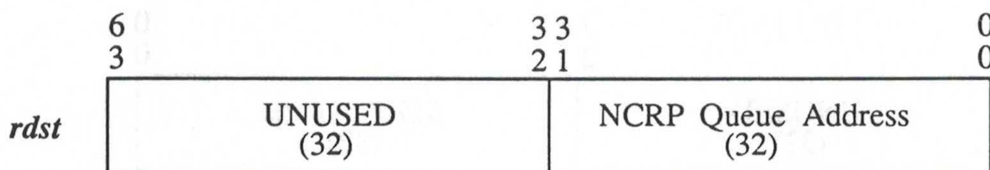


Figure F-8. Type 100x *rdst* Register Format
NCRP Queue Pointers

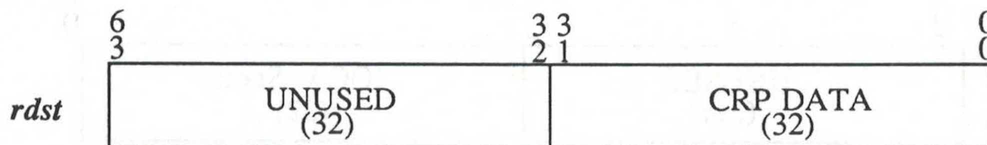


Figure F-9. Type 100x *rdst* Register Format
CRP Locations (0-4;6-7;15)

File. All registers are 32 bits in length, but the single 32-bit register has a shorter latency than the other 16 registers in the register File (shorter by 2 IOC cycles).

The CRP register file locations are addressed by the CPU (*srcb*<6..4>) and LVL (*srcb*<3..0>) variables contained in the ADDR field of the *srcb* operand. When the LVL bits (*srcb*<3..2>) = 0, the CPU bits (*srcb*<6..4>) concatenated with the LVL bits (*srcb*<1..0>) form the CRP Register File address of the MPSync locations. When LVL bits (*srcb*<3..2>) = 10, then the *single* 32-bit IOC register is accessed. DATA (*srca*<31..0>) provides a 32-bit enable mask for the modification operation. The results in *rdst* reflect the old contents of the MPSync location.

When this operation type occurs, the addressed MPSync location is read and returned to the CPU in *rdst*. In addition, the contents are modified and written back into the same location. Any bit whose corresponding mask enable bit is asserted (1) in the DATA (*srca*<31..0>) field, will be set to a zero. Any bit whose corresponding mask enable bit is not asserted (0) in the DATA (*srca*<31..0>) field, will be unchanged. All read MPSync operations **are** allowed to occur across IOCs, **but at a longer latency.**

1011 **Read (and set) MPSync:** This operation type operates identically to the **Read (and clr) MPSync** except that the enabled mask bits are set to one (instead of zero).

F.3 Control and Status Register Formats

This section provides descriptions of the control and status registers used in the I/O system.

F.3.1 Channel Control Register Format

Each HSC has an 8-bit Channel Control Register (CCR) associated with it. A K-1 processor is able to write the CCR in order to control the channel. The contents of the register are continuously sampled so modifications to the control bits are not synchronized with other IOCA operations. Bit definitions are:

Bit 7: **ICE** (Input Channel Enable): This bit enables input channel operations. All input channel operations are disabled if this bit is '0'.

Bit 6: **ICAC** (Input Channel Auto-Connect): This bit enables an auto-connect feature for the input channel. If no error conditions are present, an input channel request is honored and a connection is established. If error conditions are present, ICAC is ignored until there is another CCR WIOS instruction.

Bit 5: **ICC** (Input Channel Connect): This bit controls the connection status of the input channel. ICC is set to complete a connection sequence and cleared to disconnect the channel.

Bit 4: **ICRE** (Input Channel Request Enable): This bit controls how unsolicited input channel information is interpreted. If ICRE is set, the information is treated like

a request packet which contains the packet control bits used by the IOCA. If ICRE is clear, the information is treated like input data. The data is not used until a system request packet references it.

Bit 3: **OCE** (Output Channel Enable): This bit enables output channel operations. All output channel operations are disabled if this bit is '0'.

Bit 2: **OCR** (Output Channel Request): This bit controls the request signal of the output channel. If OCR is set and the connection is not established, a request sequence for the output channel is started. If OCR is set and the connection is already established, no action is taken. If OCR is clear and the connection is established, the connection is aborted. If OCR is clear and the connection is not established, no action is taken.

Bits 1..0: Not implemented.

All bits of *all* CCR's are reset by a Channel Reset, an IOCA reset, or an I/O Subsystem Reset.

F.3.2 Channel Status Register Format

The Channel Status Register (CSR) is a 32-bit wide register associated with each channel. A K-1 processor is able to read the CSR in order to monitor the channel. These registers are read in a byte fashion so that a portion of EACH CSR is read. Channel select bits (*srcb*<2..1>) select the proper byte for reading. Bit definitions are:

BYTE 0:

Bit 31: ANY Error - set for ANY error on this channel.

Bit 30: IC INTERCONNECT - value of the HSC INTERCONNECT signal for the input channel.

Bit 29: IC REQUEST - value of the HSC REQUEST signal for the input channel.

Bit 28: IC CONNECT - value of the HSC CONNECT signal for the input channel.

Bit 27: OC INTERCONNECT - value of the HSC INTERCONNECT for the output channel.

Bit 26: OC REQUEST - value of the HSC REQUEST signal for the output channel.

Bit 25: OC CONNECT - value of the HSC CONNECT signal for the output channel.

Bit 24: Direct/Indirect Connect Device - type of input channel device - scanned entry. '1' means Direct Connect Device.

BYTE 1:

Bit 23: IOC Channel Error - IOC-detected channel error.

Bit 22: IC BCF Error - BCF-detected error for the input channel.

Bit 21: OC BCF Error - BCF-detected error for the output channel.

Bit 20: IC CIO Error - CIO-detected error for the input channel.

Bit 19: OC CIO Error - CIO-detected error for the output channel.

Bit 18: BSI ICIF Read Error - BSI parity error on ICIF read from the IC CIO.

Bit 17: BSI CCR Write Error - CCR write to disabled channel.

Bit 16: SSR Counter Error - SSR *wios* received with counter full.

BYTE 2:

Bit 15: BCC Error - BCC-detected channel error.

Bit 14: BDI Error - BDI-detected channel error.

Bits 13..8: Not implemented.

BYTE 3:

Bits 7..0: Not implemented.

A '0' will be returned for bits that are not implemented.

F.3.3 IOCA Status Register Format

The IOCA Status Register may be viewed as four logical 32-bit wide registers associated with the IOCA and the HSC channels. A K-1 processor is able to read these registers in order to monitor the IOCA and channels. Register selection is via (*srcb*<2..1>). Register formats and bit definitions are:

IOCA STATUS REGISTER 0:

BYTE 0: returns the CCR associated with HSC 0.

BYTE 1: returns the CCR associated with HSC 1.

BYTE 2: returns the CCR associated with HSC 2.

BYTE 3: returns the CCR associated with HSC 3.

IOCA STATUS REGISTER 1:

BYTE 0: returns the SSR Counter associated with HSC 0.

BYTE 1: returns the SSR Counter associated with HSC 1.

BYTE 2: returns the SSR Counter associated with HSC 2.

BYTE 3: returns the SSR Counter associated with HSC 3.

IOCA STATUS REGISTER 2:

Bit 31: ANY IOCA error other than channel errors which are reported via CSR.

Bit 30: BSI-detected SDATA parity error.

Bit 29: BSI-detected RDATA parity error.

Bit 28: BCC-detected IOS error.

Bit 27: IOCA-detected grant tag error.

Bits 26..0: Not implemented.

IOCA STATUS REGISTER 3:

Bits 31..0: Not implemented.

A '0' will be returned for bits that are not implemented.

Appendix G. Console Specifics

G.1 Introduction

The Console interfaces to K-1 processors in two ways: through scan strings and through the TTY interface.

...

At the time this manual went to press, this chapter was still incomplete.

G.2 Interrupts

The Console is capable of interrupting a K-1 processor in several ways. First, the Console can send a NMI (Non-Maskable Interrupt). Second, the TTY interface will interrupt the processor when the processor's TTY read port is full (RPI), or when the processor's TTY write port is empty (WPI). The section on **Traps, Interrupts, and Machine Checks** in Chapter 2 discusses interrupts in more detail.

The NMI interrupt is intended for panic situations, such as an emergency shutdown due to an imminent power or cooling system failure. The TTY interface is discussed in more detail in the next section.

G.3 TTY Interface

The TTY interface provides a way for the Console and a K-1 processor to communicate. Each K-1 processor has a separate TTY port to the Console. While each processor only has one write port (the path from the processor to the Console), there are three different read ports available to each processor: the high-priority read port, the low-priority read port, and the debugger read port.

Read ports...

Write port...

The **rfec** and **wfec** instructions read and write a processor's TTY port.

...

G.4 Performance Counters

G.5 Clock Control

halt in supervisor mode

data watchpoint

trap in **Trap State**

...

G.6 Scan Control and Hidden State

reset state of machine

contents of registers:

Processor Number within MP

Number of processors in MP

Date/time

MATR (Memory Address Translation Register)

Software revision level of FE software

...

G.7 Bootstrap Procedure

bootstrap procedure

synchronous initialization of uptime counters in all processors

Appendix H. Implementation Dependencies

H.1 The Version 1 Implementation

This appendix describes those aspects of the K-1 architecture that are implemented differently in the Version 1 implementation than described in the main body of the K-1 Architecture Manual. Copies of the appropriate figures with the field sizes and restrictions for this implementation are included for clarity.

[2-1] Only the first 32 registers are supported (registers **r0** through **r31**). References to registers **r32** through **r63** are illegal and will produce unpredictable results.

[2-2] Only **ELF** flags corresponding to registers **r0** through **r31** exist.

[2-3] Addresses are 36 bits.

[2-4] Virtual and physical page numbers are 20 bits. This limits the maximum number of pages available to a process to 1024K. Figure 2-6 appears as follows:

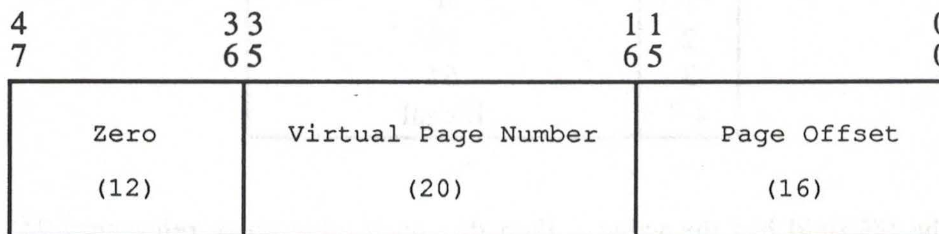


Figure 2-6. Virtual Address Format

[2-5] In **supervisor** mode (and in **Trap State**), the first 32M bytes of virtual instruction space are mapped to the first 32M bytes of physical memory. References outside of this range will cause an instruction map miss trap.

[2-6] Figure 2-7 appears as follows:

6 3	6 2	6 0	5 9	5 7	5 6	5 2	5 1	3 2	3 1	2 0	1 9	0 0

V	N (3)	SS (3)	Zero (5)	Virtual Page Number (20)	Zero (12)	Physical Page Number (20)

Figure 2-7. Instruction Page Table Format

[2-7] Both the **Virtual Page Number** and the **Physical Page Number** fields are 20 bits long and the remaining high-order bits must be set to zero.

[2-8] Table 2-3 appears as follows:

Table 2-3. Instruction Page Table Size Specifier

SS	# of Pages Mapped
0	1
1	4
2	16
3	64
4-7	Illegal

If the **SS** field has the value *i*, then this page table entry references 2^{2i} contiguous pages. The low 2*i* bits of both the virtual and physical page numbers must be zero. In other words, a 2^{2i}-page table entry must be 2^{2i}-page aligned. Each instruction page table entry can map up to 64, 64K-byte pages or 4M bytes. (That is, each entry can map as much as bits 21..3 of the **PC**.) With eight entries, a total of 32M bytes can be mapped.

[2-9] The data page table is a 16K-entry, one-way set-associative (direct-mapped) cache.

[2-10] The data page table hashing function is the exclusive-OR of the low-order 14 bits of the virtual page number and the zero-extended **Process Key** field.

[2-11] Figures 2-8 and 2-9 appear as follows:

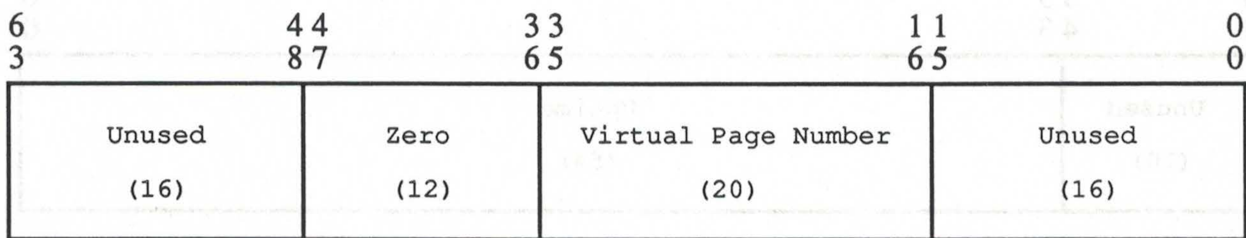


Figure 2-8. Virtual Page Number

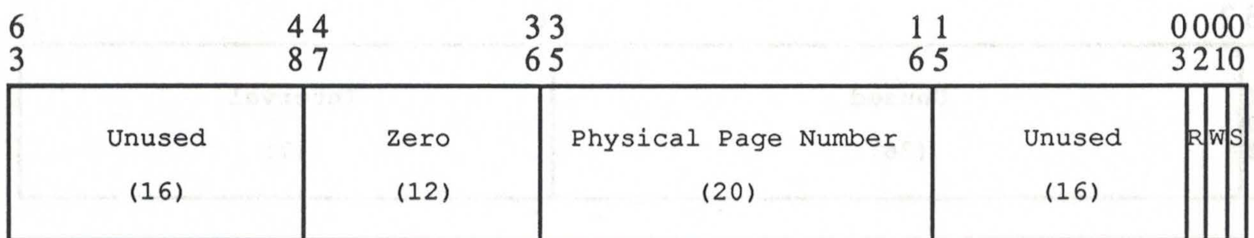


Figure 2-9. Data Page Table Format

Both the **Virtual Page Number** and the **Physical Page Number** are 20 bits long and the remaining high-order bits must be set to zero.

[2-12] The cache line size for the instruction and data caches is 256 bytes. The memory transfer sub-line size is 64 bytes.

[2-13] The instruction cache holds 4K lines (1M bytes) and is addressed by bits 19..0 of a physical address. The data cache holds 8K lines (2M bytes) and is addressed by bits 20..0 of a virtual address.

[2-14] The instruction stache line size is 64 bytes. The instruction stache stores 16 lines (1K bytes).

[2-15] The Uptime Counter is 54 bits. The **interval** field of the Interval Timer register is 27 bits. Only these 27 bits are compared with the low-order 27 bits of the Uptime Counter. Figures 2-10 and 2-11 appear as follows:

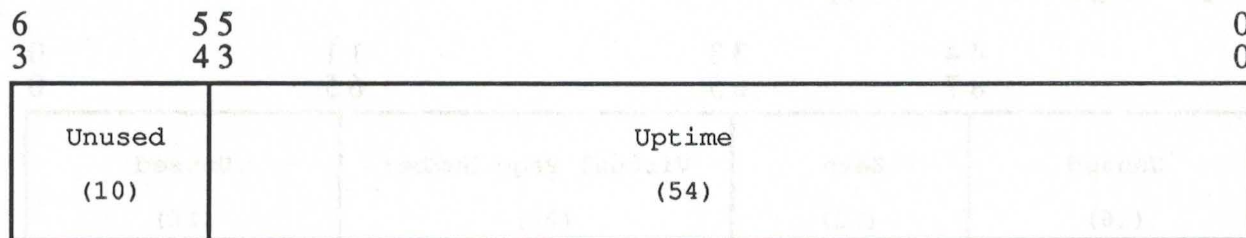


Figure 2-10. Uptime Counter

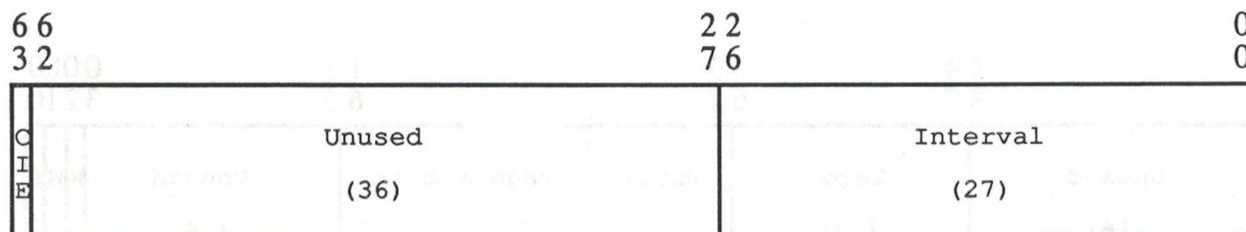


Figure 2-11. Interval Timer Register

[2-16] Attempts to write into unused bits of the **Processor Status** register may cause a trap. This issue was still open at the time this document was released.

[3-1] Only the low 34 bits of the **Absolute Branch Address** are implemented. The seven unused high-order bits must be zero. Figure 3-6 appears as follows:

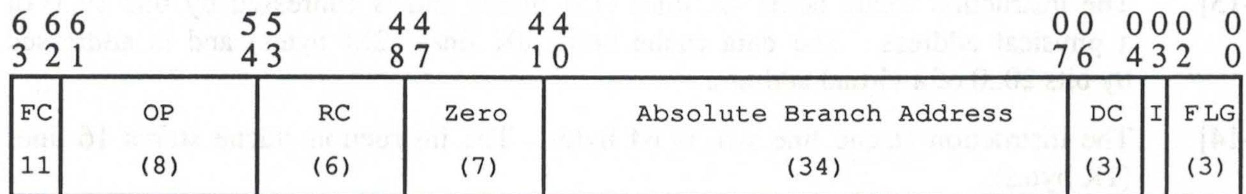


Figure 3-6. Absolute Branch Instruction Format

[5-1] Refer to [2-3].

[5-2] Refer to [2-10].

[5-3] Only the low 33 bits of the **Data Watchpoint Word Address** field are implemented. The 12 unused high-order bits must be zero. Figure 5-1 appears as follows:

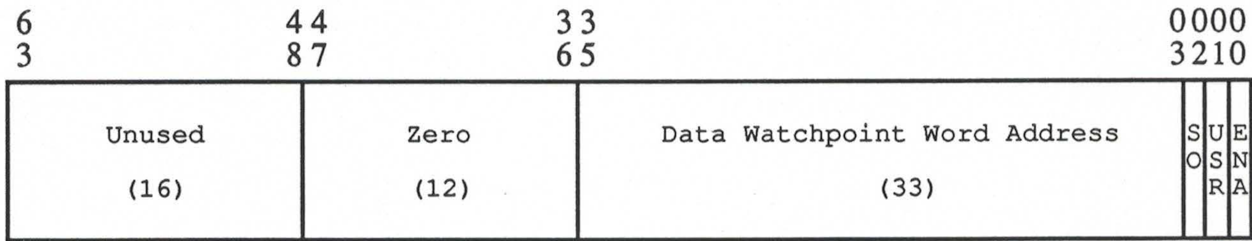


Figure 5-1. Data Watchpoint Table Entry Format

[5-4] The size of the Data Watchpoint Table is 2 (probably).

[5-5] The low-order 8 bits of the address must be zero or an illegal address trap will occur.

[5-6] Bits 10..8 of the address are replaced with the unique CPU ID.

[8-1] Refer to [2-15].

[8-2] Refer to [2-17].

[9-1] Refer to [2-6] through [2-8].

[9-2] Refer to [2-11].

[9-3] Refer to [2-10].

[9-4] The instruction cache is addressed by bits 19..8 of the physical address corresponding to *srca*, and the addressed cache line is killed. Since an entire 256 byte cache line is killed, bits 7..0 are not needed to address the instruction cache, though they are relevant if execution is continued sequentially from after the target of the **ickill**. In **supervisor** mode, since the physical address is the same as the virtual address, **ickill** may be used to kill physical memory, but only up to the 32 MB limit of **supervisor** mode addressing.

[9-5] Bits 20..8 of *srca* give the cache line index.

[9-6] The low-order 8 bits of *srca* must be zero, or else an illegal address trap occurs.

[9-7] Refer to [2-12].

[9-8] **pcl** is not implemented and acts like a **nop**.

[9-9] **iskill** takes effect on the third instruction word fetched after the instruction containing the **iskill**.

[9-10] There are two delayed write buffers.

[10-1] Refer to [5-6].

Appendix I. Floating-Point Operation Details

This appendix contains details of how floating-point operations are carried out in the K-1. Tables are presented showing the input/output relationships for all floating-point operations. In conjunction with the IEEE floating-point standard document, *ANSI/IEEE 754-1985*, this appendix completely specifies the behavior of K-1 floating-point.

I.1 Abbreviations

The following abbreviations are used throughout this appendix:

NaN	Not-a-Number (either quiet or signaling)
QNaN	Quiet NaN
SNaN	Signaling NaN
Z	Zero
FNZ	Finite nonzero representable floating-point value
∞	Infinity
BIAS	Exponent bias
INV OP	Invalid operation exception
DIV BY Z	Division by zero exception
FOVF	Floating-point overflow exception
FUDF	Floating-point underflow exception
INEXACT	Inexact exception

I.2 Use of NaN

As described in the main body of the K-1 Architecture Manual, there are two kinds of NaNs recognized by the K-1: quiet and signaling. Section 2.3.2 specifies which bit patterns are recognized as QNaNs and as SNaNs, and what bit pattern is generated for QNaNs. As described there, except for data moving instructions, when the K-1 generates a NaN, it is always a QNaN with a sign bit of zero. Note that data moving instructions (such as negate) will never "generate" a NaN (though they move an arbitrary NaN from their input to their output).

The floating-point negate function (**neg.d** and **neg.s**) is considered to be a data moving operation and therefore behaves differently than other floating-point instructions when presented with a NaN. Negate changes the sign of *any* input operand, including a NaN, and does not cause an invalid operation exception, even when given a signaling NaN.

Note that except for the negate function, no bits of the fraction part of an input NaN are preserved by any of the floating-point operations.

I.3 Floating-Point Overflow and Underflow

As mentioned in the section on **Processor Status** in Chapter 2, the K-1 supports floating-point exception traps, but they are not IEEE compatible. (The IEEE standard does not require any support for traps.) In particular, the behavior on floating-point overflow and underflow traps differs from that recommended by the IEEE standard. The results stored on floating-point overflow and underflow when trapping is not enabled are precisely in agreement with the standard.

The result stored for a floating-point operation that overflows when floating-point overflow traps are not enabled depends upon the sign of the infinitely precise result and the rounding mode in effect as described in section 7.3 of the standard. If floating-point overflow traps are enabled, then the result stored for an overflowed operation will contain the sign bit and the correctly rounded fraction, but a truncated exponent field. The "correct" unbiased exponent may be calculated by subtracting BIAS and then, if the result is negative, adding 2*(BIAS + 1).

Floating-point underflow is defined in terms of *tininess* and *loss of accuracy*. The K-1 defines a result to be tiny whenever the infinitely precise unrounded result, *r*, satisfies the condition $-2^{E_{min}} < r < 2^{E_{min}}$. Loss of accuracy is defined to occur when an inexact result is delivered.

When floating-point underflow traps are not enabled, a floating-point underflow exception is signaled whenever both tininess and loss of accuracy are detected. The correctly rounded result (which might be zero, $\pm 2^{E_{min}}$, or a denormalized number) will be stored and the floating-point underflow and inexact exception flags will be set. When floating-point underflow traps are enabled, floating-point underflow exceptions require only that tininess be detected. In this case, the correctly rounded denormalized result (which may be zero) is stored, instead of the rebiased result suggested by the IEEE standard. Either a floating-point underflow or a floating-point underflow and inexact trap occurs, depending on whether the result was inexact.

Note that floating-point underflow and overflow exceptions frequently occur along with inexact exceptions. If a trap occurs in such a case, the Trap Locators (see Chapter 2 and Appendix D) only store an indication that the *combined* condition (e.g., floating-point underflow *and* inexact) occurred, even though only one of the two exceptions may have been enabled to trap.

I.4 Floating-Point Operation Tables

This section contains a table for each floating-point operation showing the results and exceptions generated for all operands. For each table entry, the first line shows the result, and subsequent (upper case) lines indicate the exceptions. Results and exceptions that depend on the values of the FNZ inputs are given in parentheses. Footnotes explain the reasons for unusual results and exceptions.

I.4.1 Floating-Point Addition

The results for the floating-point addition instructions (**add.d** and **add.s**) are shown in Table I-1. Since addition is symmetric, only the upper triangular portion of the table is required.

Table I-1. Addition Results

srca \ *srcb*	+ Z	- Z	+ FNZ	- FNZ	+ ∞	- ∞	SNaN	QNaN
+ Z	+ Z	± Z note 1	+ FNZ	- FNZ	+ ∞	- ∞	QNaN INV OP	QNaN
- Z		- Z	+ FNZ	- FNZ	+ ∞	- ∞	QNaN INV OP	QNaN
+ FNZ			+ FNZ (or +∞) (FOVF) (INEXACT) note 2	± FNZ (or ± Z) (FUDF) (INEXACT) note 1,3	+ ∞	- ∞	QNaN INV OP	QNaN
- FNZ				- FNZ (or -∞) (FOVF) (INEXACT) note 2	+ ∞	- ∞	QNaN INV OP	QNaN
+ ∞					+ ∞	QNaN INV OP	QNaN INV OP	QNaN
- ∞						- ∞	QNaN INV OP	QNaN
SNaN							QNaN INV OP	QNaN INV OP
QNaN								QNaN

Note 1: Section 6.3 of the IEEE standard specifies that if the sum of two operands with opposite signs is exactly zero, the sign of the result depends on the rounding mode. If the floor rounding mode is used, the sign is negative; for all other rounding modes the sign is positive.

Note 2: Standard Section 7.5 specifies that if the sum overflows without an overflow trap, the INEXACT exception will also be signaled, and the result may be the same-signed infinity or the largest-magnitude number, depending on the rounding mode as specified in the IEEE standard section 7.3.

Note 3: Underflow can never be inexact in addition/subtraction. Thus, underflow is never signaled when its trap is not enabled: the condition for signaling in that case requires both underflow and inexact. A trapped underflow will never be accompanied by INEXACT.

I.5 Floating-Point Negation

The floating-point negate instructions (**neg.d** and **neg.s**) simply complement the sign bit of their input operand, regardless of the value it may represent. In particular, negate does not cause an invalid operation exception when given an SNaN, nor does it output a QNaN in this case.

I.6 Floating-Point Subtraction

The floating-point subtraction operation calculates *srcb* - *srca*. This operation is defined to be *srcb* plus the negation of *srca*. The results of floating-point subtraction instructions (**sub.d** and **sub.s**) can therefore be determined from the above rule for negation together with Table I-1.

I.7 Floating-Point Multiplication

The sign of the result for all input operands is always the **exclusive-or** of the sign bits of the input operands. The results for the floating-point multiplication instructions (**mult.d** and **mult.s**) are shown in Table I-2.

Table I-2. Multiplication Results

srca \ *srcb*	Z	FNZ	∞	SNaN	QNaN
Z	Z	Z	QNaN INV OP	QNaN INV OP	QNaN
FNZ		FNZ (or ∞ or Z) (FUDF) (FOVF) (INEXACT) note 1,2	∞	QNaN INV OP	QNaN
∞			∞	QNaN INV OP	QNaN
SNaN				QNaN INV OP	QNaN INV OP
QNaN					QNaN

Note 1: Section 7.5 of the IEEE standard specifies that if the result overflows without an overflow trap, the INEXACT exception will be signaled, and the result may be the same-signed infinity or the largest-magnitude number, depending on the rounding mode as specified in the IEEE standard section 7.3.

Note 2: An underflowed number may become zero through denormalization loss. Inexactness is one of the conditions detected for underflow (when traps are not enabled), so an FUDF exception could be

I.8 Floating-Point Division

The sign of the result for all input operands is always the **exclusive-or** of the sign bits of the input operands. The results for the operation *srca* ÷ *srcb* (the floating-point divide instructions **div.d** and **;div.s**) are shown in Table I-3.

Table I-3. Division Results

srca \ *srcb*	Z	FNZ	∞	SNaN	QNaN
Z	QNaN INV OP	Z	Z	QNaN INV OP	QNaN
FNZ	∞ DIV BY Z	FNZ (or ∞ or Z) (FUDF) (FOVF) (INEXACT) note 1, 2	Z	QNaN INV OP	QNaN
∞	∞ note 3	∞	QNaN INV OP	QNaN INV OP	QNaN
SNaN	QNaN INV OP	QNaN INV OP	QNaN INV OP	QNaN INV OP	QNaN INV OP
QNaN	QNaN	QNaN	QNaN	QNaN INV OP	QNaN

Note 1: Section 7.5 of the IEEE standard specifies that if the result overflows without an overflow trap, the INEXACT exception will be signaled, and the result may be the same-signed infinity or the largest-magnitude number, depending on the rounding mode as specified in the IEEE standard section 7.3.

Note 2: An underflowed number may become zero through denormalization loss. Inexactness is one of the conditions detected for underflow (depending on whether traps are enabled), so an FUDF exception could be accompanied by an INEXACT exception (IEEE standard section 7.4).

Note 3: The IEEE standard section 7.2 specifies that the DIV BY Z exception should not be signaled in the case of a divisor equal to zero and a dividend equal to infinity. The correctly signed infinity will be returned, and no exception will be signaled.

I.9 Floating-Point Square Root

The results for the operation *srca*$^{1/2}$ (the floating-point square root instructions **sqrt.d** and **sqrt.s**) are shown in Table I-4. Note that square root can never underflow.

Table I-4. Square Root Results

Operand	Result Value and Exceptions
+Z	+Z
-Z	-Z
FNZ in the range Z < FNZ < ∞	rounded result (INEXACT)
FNZ in the range -∞ ≤ FNZ < Z	QNaN INV OP
+∞	+∞
SNaN	QNaN INV OP
QNaN	QNaN

I.10 Floating-Point Conversions

Hardware support is provided for conversions between floating-point formats and for conversions between floating-point and integer formats.

I.10.1 Conversion Between Floating-Point Formats

The results of conversion from double precision to single precision (the **cvtd.s** instruction) are shown in Table I-5. The results of conversion from single precision to double precision (the **cvts.d** instruction) are shown in Table I-6.

Table I-5. Double to Single Conversion Results

Operand	Result Value and Exceptions
Z	Z
FNZ smaller than smallest representable nonzero single precision number (FNZS)	Z, or denorm, or smallest normalized number FUDF, (INEXACT) note 1
FNZ in the range FNZS ≤ |FNZ| ≤ FNZL	rounded result (INEXACT)
FNZ greater than largest representable finite single precision number (FNZL)	∞ or largest finite number FOVF, (INEXACT) note 2
∞	∞
SNaN	QNaN INV OP
QNaN	QNaN

Note 1: If an underflow trap is enabled, then the inexact exception is not a required condition for underflow. An INEXACT exception could also occur and be signaled (IEEE standard section 7.4).

Note 2: Section 7.5 of the IEEE standard specifies that if the result overflows without an overflow trap, the INEXACT exception will be signaled, and the result may be the same-signed infinity or the largest-magnitude number, depending on the rounding mode as specified in IEEE standard section 7.3.

Table I-6. Single to Double Conversion Results

Operand	Result Value and Exceptions
Z	Z
FNZ	FNZ
∞	∞
SNaN	QNaN INV OP
QNaN	QNaN

Table I-7. Signed or Unsigned Integer to Floating-Point Conversion Results

Operand	Result Value and Exceptions
Z	+ Z
any nonzero integer	FNZ (INEXACT) note 1

Note 1: Since the integer representation has 64 bits of precision and the floating-point format has 24 or 53, the floating-point format representation can be inexact, in which case an INEXACT exception is signaled.

I.10.2 Conversion Between Floating-Point and Integer Formats

The results of conversion from signed integer to double or single precision floating-point format (the **cvtl.d** and **cvtl.s** instructions) are shown in Table I-7. The results of conversion from unsigned integer to double precision floating-point (the **cvtul.d** instruction) can also be seen in Table I-7. The results of conversion from double or single precision floating-point to signed integer format (the **cvtd.l** and **cvts.l** instructions) are shown in Table I-8.

Table I-8. Floating-Point to Signed Integer Conversion Results

Operand	Result Value and Exceptions
Z	zero
FNZ of greater magnitude than can be represented in integer format (either positive or negative: note positive and negative maximum magnitudes will differ by 1)	largest-magnitude same-signed integer INV OP note 1
FNZ	possibly rounded integer (INEXACT) note 2
∞	largest magnitude same-signed integer INV OP note 1
QNaN	zero INV OP
SNaN	zero INV OP

Note 1: Conversion to integer of infinity and numbers too large to be represented in the destination format will be considered invalid operations. According to the IEEE standard section 7.1, if conversion of a floating-point number to an integer format results in an infinity, a NaN, or overflows and cannot be represented in the integer format, *and* this cannot be signaled any other way, then the INV OP exception will be signaled.

Note 2: If the result cannot be converted back into floating-point format and give the same original floating-point value, then INEXACT is signaled (IEEE standard section 7.5).

INDEX

absolute branch instruction format 3-4, 3-5, 7-1, 7-2, 7-4
access
  supervisor mode 2-9, 2-10, 2-11, 2-12, 2-15, 2-21, 2-22, 2-23, 2-26, 5-1, 5-4, 5-47, 5-48, 5-53, 9-8
  user mode 2-9, 2-10, 2-11, 2-12, 2-13, 2-15, 2-19, 2-22, 2-23, 5-1, 5-2, 5-3, 5-4, 5-53
add instruction C-4
add.d instruction C-2, I-3
add.s instruction I-3
addc instruction C-4
address
  absolute branch 3-5, 7-2, 7-4, H-4
  alignment 2-8
  mapping 2-10
  physical 2-10
  program 2-16, H-2
  reference 2-9
  virtual 2-9, 2-10
address mapping 2-10
  data 2-11, 5-1
  instruction 2-10
addressing modes 2-8
addt instruction D-7
alignment 2-8
Arithmetic Exception Flags 2-20, 2-21, 4-1, C-9
Arithmetic Trap Enables 2-20, 4-1

bank conflict C-3, C-11
base 2-9
binary semaphore 2-16
boof instruction C-4
bootstrap 2-29
bpt instruction 2-25, 2-28, 2-29, C-9, D-3
branch
  target 2-18, 9-4
branch address
  absolute 3-5, 7-2, 7-4, H-4
branching
  conditional 2-7, 4-1
  delayed 2-17
Byte Order Low-to-High 2-3, 2-8, 2-20, 2-21, 2-22, 5-1, 5-2, D-15

cache
  data 2-1, 2-14, C-3, C-7, C-11
  instruction 2-1, 2-14, C-7, C-10
cache coherence 2-1, 2-15, 2-16, 9-8, E-3
cache effects 2-14
cache line 2-14, 5-3, 5-4, 5-53, 9-7, 9-8, 9-9, 9-10, 12-3, E-1, H-3, H-5
cache miss 9-8, 9-9, C-3, C-7, C-10, C-11, C-12
call instruction 3-4, 3-5, 3-6, 7-1, C-4, D-6
check trap 6-61
chk instruction D-7
clock
  cycle 2-23, C-1
compare instructions C-4
  floating-point 4-1
  integer 6-16
conditional branching 2-7, 4-1
conditional execution 2-18, 4-1
conditional short constant instruction format 3-2, 3-3
console
  interrupt 2-24, 2-26
constants 2-1, 2-6, 2-8, 2-9, 5-2
  long 3-3, 12-3
  short 3-3
counter
  uptime 2-23
cvtd.l instruction I-11
cvtd.s instruction I-9
cvtl.d instruction I-11
cvtl.s instruction I-11
cvts.d instruction I-9
cvts.l instruction I-11
cvtul.d instruction I-11
cycle 2-23, C-1

data
  shared 2-14, 2-15, 5-1, 5-44, 5-47, 5-48, 9-8
data address mapping 2-11
data cache miss 9-8, 9-9, C-3, C-7, C-11, C-12
data map miss 2-13, 2-24, 5-2, 5-3, 9-9
data page table, see page table
data types 2-3
data watchpoint 5-1, 5-3, 5-4, 5-53
data watchpoint table 5-53
decode trap 2-25, 2-26, 2-27, 2-28, D-1, D-3, D-4

delay instructions 2-17, 2-18, 7-1
  first 2-17, 2-18
  second 2-17, 2-18
delay slot 2-17, 2-18, 7-1
  first 2-17, 2-18
  second 2-17, 2-18
delayed branching 2-17
delayed execution control field 2-17, 2-28, 3-4, 10-8
delayed write buffer 9-8, C-11, H-5
dflush instruction 2-12, 2-13, 2-16, 2-29, 5-2, 5-3, 5-53, C-12, D-13
  shared 2-16, 9-8
disable bit 2-17, 2-18, 2-25, 2-28, 2-29
div.d instruction C-6, I-7
div.s instruction C-6, I-7
divssr instruction 2-21, C-6
divsst instruction 2-21, C-6
dshfl instruction C-4, C-6
dshfr instruction C-4, C-6

early load 2-18
Early Load Alignment Trap 2-19, 2-20, 2-23, 5-3, 5-4, D-15
ECC 5-2, 5-3, 5-4
echk instruction 2-7, 2-19, D-3, D-12
echk trap 2-19, 5-49, 5-50
ELF flags 2-7, 2-19, 2-23, 2-29
eload instruction 2-7, 2-8, 2-12, 2-13, 2-19, 2-23, 5-1, 5-2, 5-4, D-12
  special error conditions 5-4
exception
  floating-point 2-20, 4-1, 4-5, 4-13
execution
  conditional 2-18
  program 2-16
exts instruction 2-17, 2-22, 2-28, 3-4, 3-6, 5-1, 5-2, 5-3, 5-53, C-10, D-6, D-15
exts instruction format 3-4, 3-6

fdst 3-1
flags
  Arithmetic Exception Flags 2-20, 2-21, 4-1
  ELF flags 2-7, 2-19, 2-23, 2-29
  Processor Status 2-7, 2-19, 2-20, 3-2, 4-1
floating-point
  addition I-3
  conversion I-9
  division I-7
  multiplication I-6
  negation I-5
  square root I-8
  subtraction I-5
  trap 2-26, 4-5, 4-13, D-1, D-3, D-4, D-7
floating-point add functional unit 2-1, 2-2, 6-1, C-2, C-3, C-5, C-9, C-10, D-1, D-5, D-8
floating-point divide/square root functional unit 2-1, 2-2, 6-1, C-4, C-5, C-9, C-10, D-1, D-2, D-5
floating-point multiply functional unit 2-1, 2-2, 6-1, C-3, C-5, C-9, C-10, D-1, D-8
floating-point multiply unit D-5
floating-point numbers 2-4
floating-point operations I-2
format code 2-6, 3-1, D-8
format, see instruction format
fsrc 3-1
functional unit 2-1, 2-26, C-1, C-2, C-3, C-5, C-10, D-2
  fetch, see instruction fetch unit
  floating-point 4-13
  floating-point add 2-1, 2-2, 6-1, C-2, C-3, C-5, C-9, C-10, D-1, D-5, D-8
  floating-point divide/square root 2-1, 2-2, 6-1, C-4, C-5, C-9, C-10, D-1, D-2, D-5
  floating-point multiply 2-1, 2-2, 6-1, C-3, C-5, C-9, C-10, D-1, D-5, D-8
  integer 2-1, 2-2, 6-1, C-3, C-4, C-5, C-9, C-10, D-1, D-5, D-8
  issue, see issue unit
  latency 2-25, C-2, C-3, C-4, C-8
  load/store 2-1, 2-2, C-3, C-5, C-7, C-10, C-11, D-1, D-5
  pipelines C-2, C-3, C-7, C-9, C-10, D-11
  read port C-5, C-6
  wrapping C-4

general registers 2-7

halt 2-26
halt instruction C-5

I/O, see input/output
I0 instruction 2-16, 2-25, 3-1, 3-2, 3-4, 3-6, C-1, C-5, C-6, C-9, D-6, D-7, D-8, D-9
I1 instruction 2-16, 2-25, 3-1, 3-2, 3-4, 3-6, C-1, C-5, C-6, C-9, D-6, D-7, D-8, D-9
ickill instruction 2-29, 3-4, 3-6, C-10, D-6

illegal access 2-13, 5-2, 5-3, 5-4, 9-9
illegal address 2-8, 2-19, 2-23, 5-1, 5-2, 5-3, 5-4
illegal instruction 2-18, 2-25, 3-6, 7-1, 13-1
index 2-9
input/output 2-1, 2-23
    interrupts 2-23, 2-24, 2-26
instruction 3-1
    disabled 2-18, 7-1
    I0 2-16, 2-25, 3-1, 3-2, 3-4, 3-6, C-1, C-5, C-6, C-9, D-6, D-7, D-8, D-9
    I1 2-16, 2-25, 3-1, 3-2, 3-4, 3-6, C-1, C-5, C-6, C-9, D-6, D-7, D-8, D-9
    issue 2-1, C-1, C-3, C-6
instruction address mapping 2-10
instruction cache C-7, C-10
instruction cache miss C-10
instruction fetch 2-16, 2-17, 2-18, 2-25
    trap 2-26, 2-27, 2-28, D-1
    unit 2-1, 2-2, C-1, C-5, C-6
instruction formats 2-6, 2-7, 3-1
    absolute branch 3-4, 3-5, 7-1, 7-2, 7-4
    conditional short constant 3-2, 3-3
    exts 3-4, 3-6
    long constant 2-25, 3-3, 3-4, 3-5, 3-6, 12-3, C-8, C-9
    PC-relative branch 3-1, 3-4, 3-5, 3-6, 7-1
    register 3-2, 3-3, C-8, C-9
    register branch 3-4, 3-6, 7-1, 7-3, 7-5, 9-4
    short constant 3-2, 3-3, C-8, C-9
    unconditional short constant 3-2, 3-3
instruction issue C-1, C-2, C-3, C-4, C-5, C-6, C-7, C-8, C-9, C-10, C-11
instruction map miss 2-17, 2-25, 2-28, 9-4, H-1
instruction opcode 3-1
instruction page ID 2-10, 2-15, D-5
instruction page table, see page table
Instruction Queue C-8
instruction stache 2-15, C-1, C-7, C-8, C-10
instruction stache miss C-7, C-8
instructions
    bit count 6-54
    boolean 6-39
    cache 9-1
    check 6-61
    data moving 6-29
    ELF flag 5-49
    flag 6-58
    floating-point 4-1
    floating-point compare 4-1
    floating-point computation 4-13
    floating-point conversion 4-5
    I/O 11-1
    integer 6-1
    integer arithmetic 6-1
    integer compare 6-16
    load/store 5-1
    miscellaneous 12-1
    processor status register 8-1
    reverse 6-54
    shift 6-48
    timer 8-1
    transfer of control 7-1
    trap 10-1
    virtual memory 9-1
Integer Divide Trap Enable 2-21, 6-14, 6-15
integer functional unit 2-1, 2-2, 6-1, C-3, C-4, C-5, C-9, C-10, D-1, D-5, D-8
integer trap 2-26, D-1, D-7
integers 2-3
interlock C-2, C-3
    flag 12-3, C-4
    I0/I1 C-5
    register C-3
interprocessor synchronization 2-16
interrupt 2-25, 2-26, D-1
    console 2-24, 2-26
    I/O 2-23, 2-24, 2-26
    interval timer 2-24, 2-26, D-1
interval timer interrupt 2-24, 2-26, D-1
iskill instruction 2-15, 9-4, C-10
issue 2-16, 2-21
    instruction 2-1, 2-21, 2-22, 2-25, 2-26, 2-27, C-1, C-2, C-3, C-4, C-5, C-6, C-7, C-8, C-9, C-10, C-11
    unit 2-1, 2-2, C-1, C-7, C-8, C-9, C-10

jump instruction 3-4, 3-5, 3-6, D-6

latency 2-25, C-2, C-3, C-4, C-8
ldecc instruction 5-4, D-12
ldnecc instruction 2-12, 5-4, D-12
ldpage instruction 2-13, 2-29, C-11, D-13, D-14
line, see cache line
lipage instruction 2-10, 2-29, C-5, C-7
load
    early 2-18
load instruction 2-7, 2-12, 2-13, 2-18, 2-19, 2-23, 5-1, 5-2, 5-4, C-7, C-11, D-12
load.l instruction C-10
load/store functional unit 2-1, 2-2, C-3, C-5, C-7, C-10, C-11, D-1, D-5
load/store queue 2-26, 2-27
load/store trap 2-26, D-1, D-3, D-4, D-10, D-11, D-12

loadcpu instruction 2-12, 5-1, 5-2, 5-4, D-12
long constant instruction format 2-25, 3-3, 3-4, 3-5, 3-6, 12-3, C-8, C-9

machine check 2-24, 2-27, D-1, D-2
  external D-2
  functional unit D-2
  instruction cache parity error D-2
  internal pipeline D-2
  trap-class D-2
map(addr) 3-1, 5-2
mem(size,paddr) 3-1, 5-2
memory 2-7
  virtual 2-9
memory precision 2-9
memory transfer sub-line, see sub-line
memory-related parity error 5-3, 5-4
miss
  data cache 9-8, 9-9, C-3, C-7, C-11, C-12
  data map 2-13, 5-2
  instruction cache C-10
  instruction stache C-7, C-8
mode
  addressing 2-8
  reference 2-9
  supervisor 2-9, 2-22
  user 2-9, 2-22, 2-23
move instruction C-2
move.d instruction 6-1
mult.d instruction I-6
mult.s instruction I-6

NaN
  quiet 2-4
  signaling 2-4
  usage I-1
  values defined 2-5
neg.d instruction I-1, I-5
neg.s instruction I-1, I-5
nonexistent memory 2-19, 5-2, 5-3, 5-4
nop instruction 2-25, C-8, C-9

opcode 3-1

page 2-10
  invalid 2-13
  shared 2-12, 2-13, 5-1, 5-44, 5-47, 5-48, 9-8
page control bit 2-12, 2-13
page map, see page table
page map/cache tag parity error 5-3, 5-4
page table 2-10
  data 2-10, 2-12, 2-13, 2-15, 2-22, 2-29, 5-1, 5-2, 5-3, 5-4, 9-3, 9-8
  instruction 2-10, 2-11, 2-15, 2-29, 9-2, D-5
PC 2-16, 2-17, 7-1, H-2
  Restart PC 2-25, 2-26, 2-28, 10-8, D-15
  return PC 7-1
PC-relative branch instruction 7-1
PC-relative branch instruction format 3-1, 3-4, 3-5, 3-6, 7-1
pcl instruction 2-12, 2-13, 5-2, 5-53, C-11, C-12, D-12
physical addresses 2-10
pipelining C-2
precision 2-8, 2-9
privilege violation 2-18, 2-25, 7-1, 13-1
Process Key 2-12, 2-13, 2-20, 2-21, 5-2, 9-3, D-15
Processor Priority Level 2-20, 2-23, 2-27, 8-4
Processor Status 2-8, 2-9, 2-11, 2-12, 2-19, 2-21, 2-29, 4-1, 4-5, 5-1, 5-2, 8-1, 8-2, 8-3, H-4
  Arithmetic Exception Flags 2-20, 2-21, 4-1, C-9
  Arithmetic Trap Enables 2-20, 4-1
  Byte Order Low-to-High 2-3, 2-8, 2-20, 2-21, 2-22, 5-1, 5-2, D-15
  Early Load Alignment Trap 2-19, 2-20, 2-23, 5-3, 5-4, D-15
  flags 2-7, 2-19, 2-20, 4-1
  Integer Divide Trap 2-21
  Integer Divide Trap Enable 6-14, 6-15
  Process Key 2-12, 2-13, 2-20, 2-21, 5-2, 9-3, D-15
  Processor Priority Level 2-20, 2-23, 2-27, 8-4
  Processor Version Number 2-19, 2-20, 8-3
  Rounding Mode 2-20, 2-21, 4-5, 6-15, 8-5
  Small Address Compatibility Mode 2-9, 2-20, 2-22, 2-23, 5-1, 5-2, 7-1, D-15
  Trace Enable 2-20, 2-21
  Trace Pending 2-20, 2-21, 2-22
  User Mode Load 2-11, 2-12, 2-20, 2-22, 2-23, 5-1, 5-44, 9-10, D-15
  User Mode Store 2-11, 2-12, 2-20, 2-22, 2-23, 5-1, 5-44, 9-7, 9-8, 9-9, D-15
  User Protection 2-11, 2-12, 2-13, 2-20, 2-22, 2-23, 5-1, 5-2, 9-7, 9-9, 9-10, D-15
Processor Version Number 2-19, 2-20, 8-3
program addresses 2-16, H-2
program counter, see PC
program execution 2-16

rdst 3-1
read port C-5, C-6
reference 2-9
mode 2-9
shared 2-12, 2-15
supervisor mode 2-9, 2-10, 2-11, 2-12, 2-15, 2-21, 2-22, 2-23, 2-26, 5-1, 5-4, 5-47, 5-48, 5-53, 9-8
user mode 2-9, 2-10, 2-11, 2-12, 2-13, 2-15, 2-19, 2-22, 2-23, 5-1, 5-2, 5-3, 5-4, 5-53
register branch instruction format 3-4, 3-6, 7-1, 7-3, 7-5, 9-4
register instruction format 3-2, 3-3, C-8, C-9
registers 2-1, 2-6, 2-9
general 2-7
reserved C-3
relf instruction 2-7, 2-19, D-12
reserved
register C-3
reset operation 2-29
reset trap 2-26, 2-27, 2-29
Restart PC 2-25, 2-26, 2-28, 10-8, D-1, D-15
return PC 7-1
rfec instruction C-12, D-12
rios instruction 2-23, C-12, D-13
rounding 2-21, I-2, I-4, I-6, I-7, I-9
Rounding Mode 2-20, 2-21, 4-5, 6-15, 8-5
rps instruction 2-19, 2-26, C-4, C-9, D-15
rtrpd instruction 2-26, 2-27, C-4, D-1, D-10, D-12, D-14
rut instruction 2-24, C-4

sel instruction C-4
semaphore 2-16
serial instructions C-5, C-9
shared
data 2-12, 2-14, 2-15, 5-1, 5-44, 5-47, 5-48, 9-8
page 2-12, 2-13
reference 2-12, 2-15
shared dflush 2-16, 9-8
short constant instruction format 3-2, 3-3, C-8, C-9
sign extended 2-3
slstrpd instruction 2-26, 2-27, 5-3, 5-53, D-1, D-12, D-14
Small Address Compatibility Mode 2-9, 2-20, 2-22, 2-23, 5-1, 5-2, 7-1, D-15
spl instruction 2-19, 2-23, C-4, C-9
sqrt.d instruction C-6, I-8
sqrt.s instruction C-6, I-8
srca 3-1, 3-2
srcb 3-1, 3-2
srcc 3-1
srm instruction 2-19, 2-21, C-4, C-5, C-9
stache
instruction 2-15, C-1, C-7, C-8, C-10
store instruction 2-12, 2-13, 2-27, 5-1, 5-2, 5-4, 5-53, C-6, C-11, D-12, D-13, D-14, D-15
storecpu instruction 2-12, 2-27, 5-1, 5-2, D-12
strap instruction 2-28, C-9, D-3, D-4
sub-line 2-14, 5-3, 5-4, E-1, E-2, H-3
sub.d instruction I-5
sub.s instruction I-5
subb instruction C-4
subt instruction D-7
supervisor mode 2-9, 2-22
swat instruction 2-12, 2-13, 2-16, 5-2, 5-53, D-12, D-13, D-14, D-15
synchronization 2-16

target
branch 2-18, 9-4
timers 2-23
timing C-1
Trace Enable 2-20, 2-21
Trace Pending 2-20, 2-21, 2-22
trace trap 2-25, 2-28
trap 2-24, 2-26
arithmetic 2-20, 2-21
bpt instruction 2-25, 2-28, 2-29, D-3
check 6-61
data map miss 2-13, 2-24, 5-2, 5-3, 9-9
data watchpoint 5-3, 5-4, 5-53
decode 2-25, 2-26, 2-27, 2-28, D-1, D-3, D-4
dflush-type 5-3
ECC 5-2, 5-3, 5-4
echk 2-19, 5-49, 5-50
eload-type 5-3
floating-point 2-26, 4-5, 4-13, D-1, D-3, D-4, D-7
handling D-1
illegal access 2-13, 5-2, 5-3, 5-4, 9-9
illegal address 2-8, 2-19, 2-23, 5-1, 5-2, 5-3, 5-4
illegal instruction/privilege violation 2-18, 2-25, 3-6, 7-1, 13-1
imprecise 2-25, 2-26, 2-27
in Trap State 2-26
instruction fetch 2-26, 2-27, 2-28, D-1
instruction map miss 2-17, 2-25, 2-28, 9-4, H-1
integer 2-26, 4-5, D-1, D-7
invalid operation 2-4, 4-1
load-type 5-3

load/store 2-26, D-1, D-3, D-4, D-10, D-11, D-12
memory-related parity error 5-3, 5-4
NaN causing 2-4
nonexistent memory 2-19, 5-2, 5-3, 5-4
page map/cache tag parity error 5-3, 5-4
precise 2-25, 2-27
reset 2-26, 2-27, 2-29
return from 2-28
simultaneous D-1
store-type 5-3
strap instruction 2-28, D-3, D-4
trace 2-21, 2-22, 2-25, 2-28
trap instruction 2-25, 2-28, D-3, D-4
vectoring 2-27, 2-29
xtrap instruction 2-28, D-3, D-4
zcl-type 5-3
trap data
load/store D-1
primary D-1
trap instruction 2-25, 2-28, C-9, D-3, D-4
Trap Locator 2-26, 2-27, D-1, D-5
trap sequence 2-26
Trap State 2-9, 2-11, 2-15, 2-19, 2-20, 2-21, 2-22, 2-23, 2-24, 2-26, 2-27, 2-28, 2-29, 5-53, 7-2, 7-3, 10-3, 10-8, C-9, C-10, D-8, D-9, D-10, D-14, D-15, H-1
Trap Summary 2-23, 2-25, 2-26, 2-27, D-1
trap type D-1

unconditional short constant instruction format 3-2, 3-3
undefined opcodes 13-1
unordered 4-1
uptime counter 2-23
user mode 2-9, 2-22, 2-23
User Mode Load 2-11, 2-12, 2-20, 2-22, 2-23, 5-1, 5-44, 9-10, D-15
User Mode Store 2-11, 2-12, 2-20, 2-22, 2-23, 5-1, 5-44, 9-7, 9-8, 9-9, D-15
User Protection 2-11, 2-12, 2-13, 2-20, 2-22, 2-23, 5-1, 5-2, 9-7, 9-9, 9-10, D-15

virtual addresses 2-9, 2-10
virtual memory 2-9

wdwp instruction 5-4, D-13
welf instruction 2-7, 2-19, D-13
wfec instruction C-12, D-12
wios instruction 2-23, C-12, D-13
wit instruction 2-24, C-4, C-9
wps instruction 2-19, 2-20, 2-21, 4-1, C-4, C-5, C-7, C-9, D-12, D-15
wrapping C-4

xtrap instruction 2-28, C-9, D-3, D-4

zcl instruction 2-12, 2-13, 2-29, 5-3, 5-53, C-12, D-12
zero extended 2-3