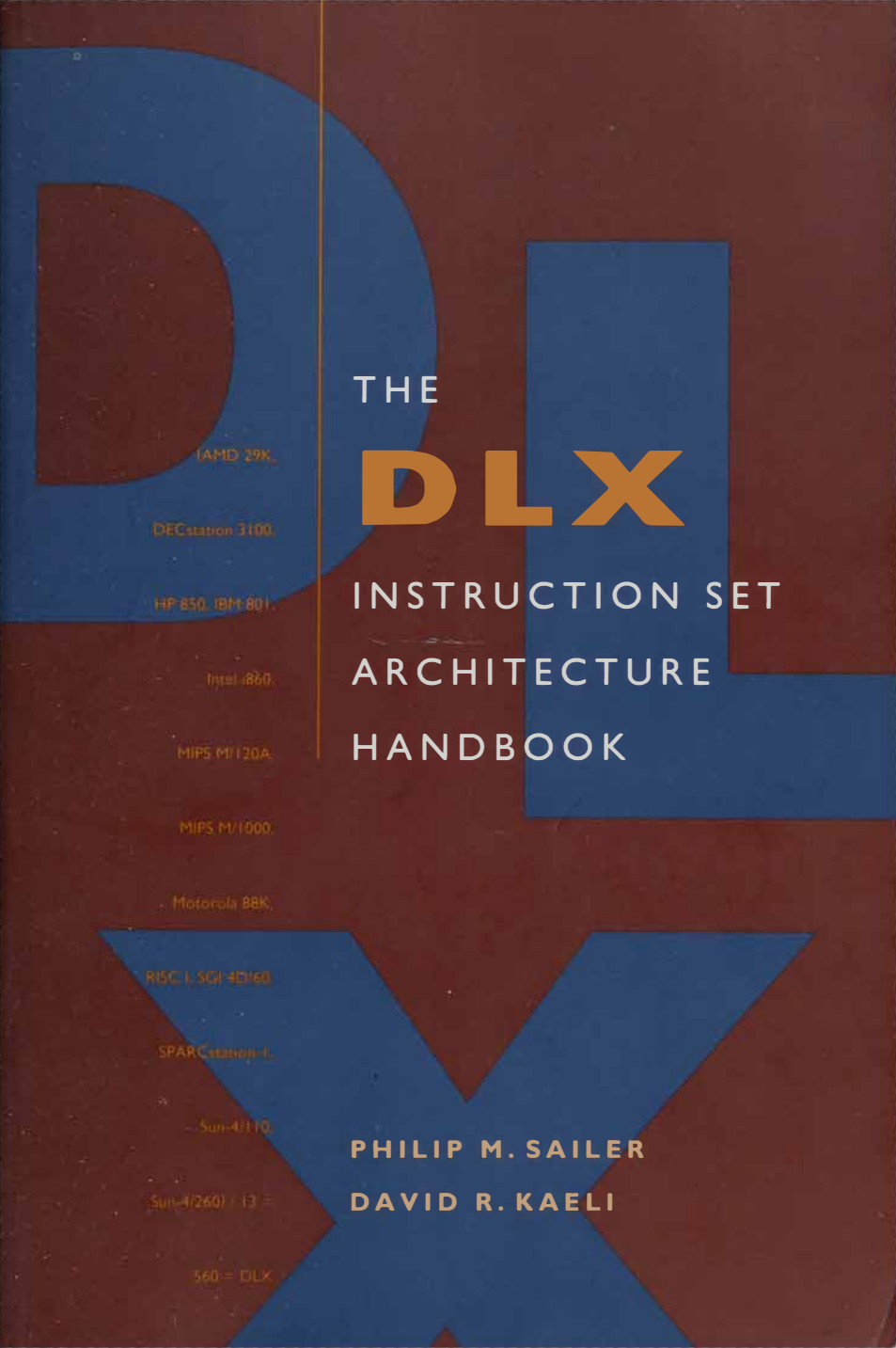


THE DLX INSTRUCTION SET ARCHITECTURE HANDBOOK

PHILIP M. SAILER
DAVID R. KAELI

THE DLX INSTRUCTION SET ARCHITECTURE HANDBOOK

THE DLX INSTRUCTION SET ARCHITECTURE HANDBOOK

PHILIP M. SAILER

DAVID R. KAELI

NORTHEASTERN UNIVERSITY

Morgan Kaufmann Publishers, Inc., San Francisco, California

Sponsoring Editor Jennifer Mann
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Assistant Jane Elliott
Copyeditor Ken DellaPenta
Cover Design Carron Design
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet mkp@mkp.com
Order toll free 800/745-7323

Printed in the United States of America

00 99 98 97 96 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available for thi book.

ISBN 1-55860-371-9

Contents

Figures

Tables

Preface

The DLX instruction set architecture was first presented by John Hennessy and Dave Patterson in their popular text *Computer Architecture: A Quantitative Approach.* They developed this architecture because it fully embodies the philosophy of RISC architecture. The instructions included in the DLX instruction set represent those that are most frequently used in programs compiled on other, more sophisticated RISC machines. The DLX architecture is a great choice for teaching the principles of processor architecture and pipelining to tomorrow's computer scientists and engineers.

The DLX instruction set architecture is currently used in a variety of computer science and computer engineering courses at many educational institutions. DLX is especially well suited as an example to demonstrate the fundamental principles underlying RISC architecture, while still remaining easy enough to understand. Surprisingly, though, there has been no comprehensive guide to the DLX architecture until the publication of this text. Therefore, we consider it a privilege to provide the computer architecture community with this book—the first, definitive *DLX Instruction Set Architecture Handbook.*

Have you ever had to debug an assembly language program and forgotten what a particular opcode stands for? Have you ever been confused about which field in a particular instruction is the source register and which is the destination? Or have you ever needed to know how to interpret the contents of an immediate field? Anyone who's taken a computer architecture course has experienced the frustration of not being able to find this information in an organized and easy-to-understand reference. It was just this reason that prompted us to write this handbook.

The purpose of this handbook is to provide students, computer scientists, and computer architects with a complete reference to the DLX instruction set architecture. This book is not meant to be a primer on using the architecture, nor does it suggest how to construct a DLX processor.

Instead, we strove to make this book implementation-independent, just as DLX is implementation-independent. This means that, although you will find much important and useful information about DLX, you will not find the details of such topics as interrupt/exception handling and usage of traps. By limiting discussion of such topics we avoid imposing needless restrictions upon implementations of the architecture.

Handbook Organization

The first chapter of this book introduces the reader to DLX, presenting its history and those topics that form the foundation of the instruction set, including registers, data formats, addressing, and interrupt handling. The second chapter provides an overall summary of the DLX instruction set organized into six distinct categories. This provides a natural index into the intended usage of the instruction set architecture. The third and final chapter is a strict reference providing the type, format, operation, and detailed description of each DLX instruction. It is this chapter that will invariably become the primary source of reference information for DLX users.

The book includes two appendices. The first is a quick reference to DLX instructions, providing tables of the instructions, organized by their function, including the mnemonic and required operands for each. The second appendix is a report on the DLXsim simulator from the University of California at Berkeley.

Supplemental Tools and Software

There are two sources of information, courseware and software, to accompany this handbook. Both are provided by the publishers of this book. This material is located at the Morgan Kaufmann Publishers ftp site, which may be accessed anonymously at `mkp.com`, and a World-Wide Web site, accessed using the URL: `http://MKP.COM`. At the time of publication of this book, the resources available at these locations include the DLXsim simulator and the dlxcc C language compiler for the DLX instruction set architecture. A pipelined implementation of DLX in synthesizable VHDL will be made available in 1996.

Acknowledgements

We would like to acknowledge the contributions of a number of people. First, we would like to thank Peter Ashenden for his careful technical review of preliminary versions of this handbook. Next, we would like to thank the staff at Morgan Kaufmann, including Jennifer Mann, Yonie Overton, Julie Pabst, Doug Sery, and Bruce Spatz, for their patience, enthusiasm, and careful copyediting. We would like to thank Ed Sznyter of Babel Press for his assistance in producing our high-resolution output. Finally, we would like to acknowledge the assistance of Navid Atoofi, our system administrator at Northeastern University, who guided us through the many trials and tribulations encountered while using LaTeX to prepare this handbook.

Phil Sailer and Dave Kaeli

Chapter 1

DLX Architecture Overview

This first chapter of the DLX handbook is to serve as an introduction to DLX,[1] including its origins and history as well as the essential topics of registers, data formats, addressing, and interrupt handling.

1.1 Introduction

> *The authors believe DLX to be the world's second polyunsaturated computer—the average of a number of recent experimental and commercial machines that are very similar in philosophy to DLX. Like [Donald] Knuth [1], we derived the name of our machine from an average expressed in Roman numerals:*
> *(AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260) / 13 = 560 = DLX. [2]*

DLX (pronounced "deluxe") is a simple load/store architecture that has its origin in the text by Hennessy & Patterson, cited above. Its architecture is an amalgam of other similar (i.e., of the RISC—Reduced Instruction Set Computer—type) and more sophisticated load/store architectures. It is based on observations made about the most frequently used

[1] Usage of the acronym "DLX" found in this introduction, as well as the remainder of this handbook, may refer to either "the DLX computer" or "the DLX instruction set architecture," depending upon the context in which it is used, for the purposes of brevity and readability.

primitives in programs. Functions that are used less often are considered less critical in terms of performance and are therefore not implemented directly in DLX. Emphasis is clearly placed upon four architectural concepts:

- the simplicity of a load/store instruction set
- the importance of pipelining capability
- the benefits of an easily decoded instruction set
- the compilation of high-level programs into efficient machine-language code

DLX has 32 general-purpose registers and 32 single-precision floating-point registers, as well as a number of miscellaneous registers used for interrupt handling and floating-point exceptions. The DLX word length is 32 bits, while memory is byte addressable in Big Endian mode with a 32-bit address.

The DLX instruction set was initially introduced in the first edition of *Computer Architecture: A Quantitative Approach* by John Hennessy and David Patterson [2], and its instructions operate most similarly to those of the MIPS R2000 instruction set architecture [3]. This handbook provides a clear and concise description of the architecture, without attempting to discuss any particular *implementation* of the architecture.[2]

The primary objective of this handbook is to define the DLX instruction set architecture (ISA) used throughout the Hennessy & Patterson computer architecture textbook [2], as well as in the VHDL design book by Peter Ashenden [5]. In this chapter, a description of general-purpose registers, floating-point registers, miscellaneous registers, data formats, addressing modes, and interrupt handling is provided.

1.2 Registers

1.2.1 General-Purpose Registers

The DLX ISA contains 32 (R0–R31) 32-bit general-purpose registers. Registers R1–R30 are true general-purpose registers. While particular

[2] A popular implementation of DLX is *DLXsim—A Simulator for DLX*, created by Larry B. Hostetler and Brian Mirtich [4]. The documentation for this simulator is provided in Appendix B.

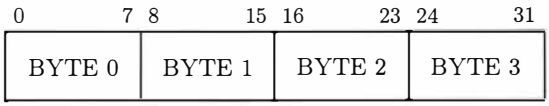


Figure 1.1: General-Purpose Register Format

compiler implementations may limit their generality, the architecture does not restrict their general use in any way.

Register R0 always contains a 0 value. Register R0 cannot be modified but can be used as a destination operand if the result of the executed instruction will not be needed. To perform a load immediate operation (i.e., load a fixed value into a register), R0 is used as one of the operands of the ADDI (add immediate) instruction.

Register R31 is used for remembering the return address for the JAL and JALR instructions. One should be careful when using R31 as an operand to ensure that the return address is preserved.

A register may be loaded with a byte (8-bit), halfword (16-bit) or fullword (32-bit) quantity. Register bits are numbered 0–31, from back to front (i.e., bit 0 is the most significant bit (MSB) and bit 31 is the least significant bit (LSB)). Byte ordering is done in a similar manner: BYTE 0 is the most significant byte and BYTE 3 is the least significant byte (see Figure 1.1).[3]

ALU operations that operate on register operands expect fullword register operands (they only expect 16-bit sign-extended immediate operands) and produce a fullword result. Store instructions can store a byte, halfword, or fullword from a register. Byte-register operands address [BYTE 3], halfword-register operands address [BYTE 2 BYTE 3], and fullword-register operands address [BYTE 0 BYTE 1 BYTE 2 BYTE 3] (see Figure 1.1).

Halfword and byte load instructions fill the least significant bytes of the destination register. The most significant bytes (that were not filled) are filled with either the sign bit of the loaded values or zeroes, depending upon the load opcode (i.e., signed or unsigned load). The LHI (load high immediate) instruction loads the top half of a register with a halfword quantity and sets the bottom half of the register to zero.

[3]While this ordering may seem awkward to some readers, it is consistent with the ordering used in the Hennessy & Patterson textbook [2].

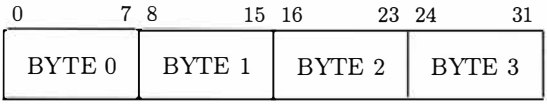


Figure 1.2: Single-Precision Floating-Point Register Format

The general-purpose registers also hold the result of set-on-comparison instructions that are used for conditional control transfer (i.e., conditional branches).

1.2.2 Floating-Point Registers

The DLX ISA provides 32 32-bit single-precision registers (F0, F1, F2, F3, ..., F28, F29, F30, F31). This same set of registers can also be addressed as 16 64-bit double-precision registers (F0, F2, F4, ..., F26, F28, F30). Double-precision operations can only be performed on even/odd pairs of floating-point registers (e.g., F10 & F11), but only the even-numbered register name is used.

The floating-point format used by DLX is the IEEE Standard 754, a discussion of which may be found in Appendix A of the Hennessy & Patterson textbook [2]. Floating-point data can be transferred between double-precision and single-precision registers (using a pair of single-precision registers). Transfers can also take place between general-purpose registers and floating-point registers. The smallest addressable unit in a floating-point register is 32 bits. Figures 1.2 and 1.3 illustrate the formats of the single- and double-precision floating-point registers for DLX.

The DLX instructions for multiplication and division operate on 32-bit values stored in individual floating-point registers. These instructions treat the contents of the source registers as either signed (two's complement) or unsigned integers, depending upon the opcode.

1.2.3 Miscellaneous Registers

There are also three miscellaneous registers defined in the DLX instruction set: the program counter (PC), the interrupt address register (IAR), and the floating-point status register ($FPSR$). Their formats are illus-

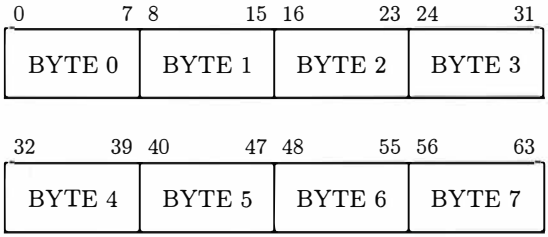


Figure 1.3: Double-Precision Floating-Point Register Format

trated in Figure 1.4. The *IAR* and *FPSR* are also referred to as "special registers."

The 32-bit program counter (*PC*) contains the address of the instruction currently being retrieved from memory for execution. The *PC* is an implicit operand for all jump (non-register), branch, and trap instructions. If the current instruction is not a taken branch, jump, or trap instruction, the PC is incremented ($PC \leftarrow_{32} [(PC) + 4]$).[4] The return from exception (RFE) instruction will restore the *PC* saved in *IAR*.

The 32-bit interrupt address register (*IAR*) maintains the 32-bit return address of the interrupted program when a TRAP instruction is encountered, and is used by the return from exception (RFE) instruction to restore the 32-bit *PC*. The contents of the *IAR* may also be loaded from or stored to a GPR.

To provide for conditional branching based on the result of floating-point operations, a single-bit floating-point status register (*FPSR*) is provided. Floating-point compares will set this bit, and then conditional branch instructions are provided that branch based on the value of the bit (true '1' or false '0').

1.3 Data Formats

The DLX instruction set architecture defines the following data formats: a fullword is 32 bits, a halfword is 16 bits, and a byte is 8 bits. A byte is the smallest addressable unit. Byte ordering adheres to the Big Endian ordering, as shown in Figure 1.5. The most significant byte is always at

[4] Refer to Tables 2.1 and 2.2 for an explanation of this notation.

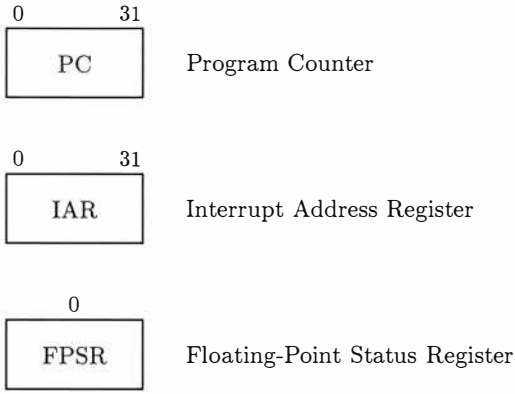


Figure 1.4: Miscellaneous Register Formats

the lowest byte address in a word or halfword. Each word or halfword is addressed by the byte address of its most significant byte.

DLX provides for signed and unsigned loads of byte, halfword, and fullword quantities. A signed load of a halfword or a byte will place the loaded quantity in the lower portion of the register, filling the upper portion with the value of the sign bit. An unsigned load of a halfword or a byte will place the loaded quantity in the lower portion of the register, filling the upper portion with zeroes. To support the loading and storing of single- and double-precision floating-point numbers to and from floating-point registers, DLX provides special opcodes. A single-precision

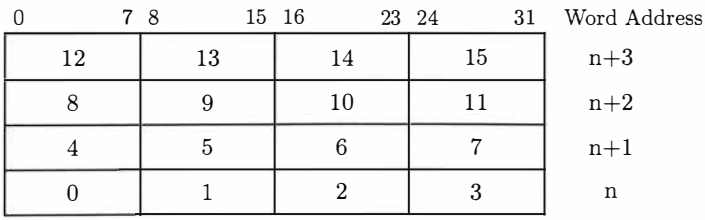


Figure 1.5: Big Endian Byte Ordering

load or store will transfer 32 bits, and a double-precision load or store will transfer 64 bits.

1.4 Addressing

Memory is byte addressable, and strict address alignment is enforced. Halfword memory accesses are restricted to even memory addresses. Fullword memory accesses are restricted to memory addresses divisible by 4 (i.e., ending in `0x0`, `0x4`, `0x8`, or `0xC`). Virtual addressing is not architected in DLX.

Because DLX is a pure load/store architecture, the only instructions that access memory are explicit load and store instructions. (This adheres to the pure RISC philosophy that is one of the principles of the DLX architecture.) An effective address for a load or a store instruction is computed as follows:

$$\textit{32-bit Unsigned Effective Address} = \textit{32-bit Register} + \textit{Signed 16-bit Offset}$$

1.5 Interrupt Handling

There are two architected interrupts in the DLX instruction set: the arithmetic overflow exception and the TRAP instruction.[5]

The arithmetic overflow exception may occur during an ADD, ADDI, SUB, or SUBI instruction as a result of a two's complement overflow. For this type of interrupt, the program counter (PC) is loaded into the interrupt address register (IAR), and the interrupt address is loaded into the program counter (PC). As a result, control is transferred to the interrupt handling routine located at the interrupt address. The return from exception (RFE) instruction restores the PC from the IAR.

The TRAP instruction causes the program counter (PC) to be loaded with the specified, absolute address, and the interrupt address register (IAR) to be loaded with the return address. The return from exception (RFE) instruction restores the PC from the IAR.

[5]The discussion of the TRAP instruction found in this handbook, although simplistic, follows that presented in the Hennessy & Patterson textbook [2]. A more realistic suggestion of how the TRAP instruction might be architected can be found in the DLXsim simulator [4].

Chapter 2

DLX Instruction Set Summary

This chapter of the DLX handbook describes the DLX instruction set architecture. Included is a general discussion of the ISA, followed by the specifics of DLX instruction types, format notation, and operational notation.[1] The entire set of DLX instructions is separated into six classes and discussed in detail with respect to this classification.

2.1 General Description

The instructions that were chosen to be part of DLX are those that were determined to resemble the most frequently used (and, therefore, performance-critical) primitives in programs. More complex data manipulations (such as those found in CISC—Complex Instruction Set Computer—architectures) can be accomplished by a sequence of several DLX instructions. The 92 instructions provided by DLX are separated into six distinct classes:

- load and store instructions
- move instructions
- arithmetic and logical instructions
- floating-point instructions

[1]The notational conventions introduced in this section will apply consistently throughout the remainder of the handbook.

- jump and branch instructions
- special instructions

2.2 Instruction Types

All DLX instructions are 32 bits and must be aligned in memory on a word (32-bit) boundary. There are only three instruction formats used in DLX:

- I-type (immediate)
- R-type (register)
- J-type (jump)

Limiting the number of instruction formats provides for efficient decoding of instructions. I-type instructions manipulate data provided by a 16-bit signed or unsigned *immediate* field (and possibly a register). The *immediate* data is either an ALU operand or a 16-bit *offset* that is sign-extended and added to the program counter (for conditional branches). I-type instructions also include unconditional control transfers (i.e., jumps) that use a register operand to specify the branch target address. R-type instructions manipulate data from one or two registers. Both I-type and R-type instructions, with the exception of store and branch instructions, place their results into a register. J-type instructions provide for the execution of jumps that do not use a register operand to specify the branch target address. The branch target address is instead either obtained in an implied register (for the RFE instruction) or computed using a 26-bit signed *immediate* field that is added to the program counter (PC).

2.2.1 I-Type

The I-type instructions include load and store instructions of bytes, halfwords, full words, and immediates. They also include all immediate ALU operations and conditional branch instructions. I-type instructions include the unconditional control transfers jump register (JR) and jump and link register (JALR). The format of an I-type instruction is illustrated in Figure 2.1.

The opcode field specifies which DLX instruction is being executed. The rs_1 field may specify a source register for ALU operations, a base

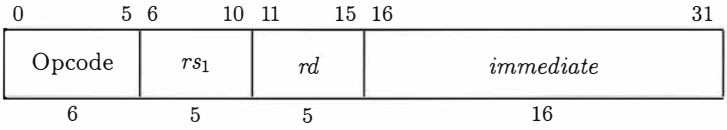


Figure 2.1: I-Type Instruction Format

register for memory address generation for loads and stores, the location of the register to test for conditional branches, or the destination address register for the jump register (JR) and jump and link register (JALR) instructions. The destination register *rd* is used as the location to either place the data retrieved from memory on a load instruction or provide the data to be written to memory on a store instruction. The destination register is also used for storing the result of ALU operations. The *rd* field is unused for conditional branches and the jump register (JR) and jump and link register (JALR) instructions. The *immediate* field contains the offset used to compute the sign-extended memory address for loads and stores, the *immediate* operand for ALU operations, or the sign-extended offset that is added to the program counter (*PC*) to compute the branch target address for a conditional branch. The *immediate* field is unused for the jump register (JR) and jump and link register (JALR) instructions.

2.2.2 R-Type

The R-type instructions are used for register-to-register ALU operations, reads and writes to and from the special registers (*IAR* and *FPSR*), and moves between the general-purpose registers and/or the floating-point registers. The R-type instruction format is illustrated in Figure 2.2.

The opcode field, along with the *func* field, specifies the operation to be performed. The number of bits used for the *func* field depends upon the instruction. Bits 26–31 are used for integer arithmetic and logic instructions, while bits 27–31 are used for floating-point instructions. The rs_1 field is used as a source register operand for ALU operations and move instructions. The rs_2 field is used as a second source register operand for ALU operations. The *rd* field specifies the destination register operand for ALU operations and move instructions.

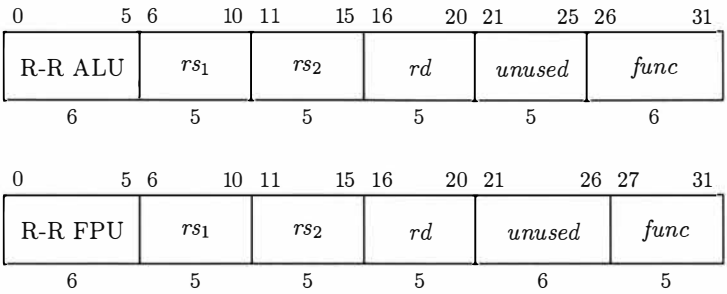


Figure 2.2: R-Type Instruction Format

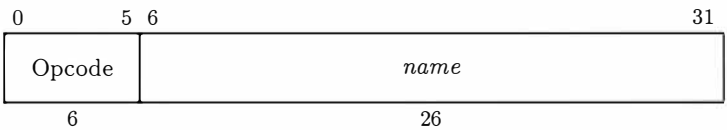


Figure 2.3: J-Type Instruction Format

2.2.3 J-Type

The J-type instructions include jump (J), jump and link (JAL), TRAP, and return from exception (RFE). The format for J-type instructions is illustrated in Figure 2.3.

The opcode field specifies the operation to be performed. For jump instructions, the *name* field is a 26-bit signed offset that is added to the address of the instruction in the delay slot (i.e., $[(PC) + 4]$) to generate the target address. For a trap instruction, the *name* field specifies an unsigned 26-bit absolute address that is used as the target address.

2.3 Instruction Format Notation

When specifying DLX instructions in the form of a mnemonic followed by one, two, or three operands, certain notation is consistently used to represent registers, bit fields, etc. The notation presented here will help the reader interpret the remaining information and examples presented

in this handbook, as well as provide a basis for properly writing DLX assembly programs.[2]

- The DLX mnemonics are always capitalized (e.g., ADDUI, LW).
- The specifiers rs_1 and rs_2 (and *rd*, for store instructions, only) represent locations where *source registers* should be specified.
- The specifier *rd* (except for store instructions) represents a location where a *destination register* should be specified.
- If a register is to be a GPR, it is specified by the lowercase letter 'r' immediately followed by a number in the range 0..31 (e.g., r0).
- If a register is to be a FPR, it is specified by the lowercase letter 'f' immediately followed by a number in the range 0..31 (e.g., f0).
- Instructions requiring registers to be specified are expecting registers of the GPR type, except when explicitly stated otherwise.
- The specifier *immediate* represents a location where a 16-bit number should be specified. This number may be treated as a signed or unsigned value, depending upon the instruction in which it is being used.
- The specifier *name* represents a location where a 16-bit number (for branch instructions) or a 26-bit number (for jump and trap instructions) should be specified. This number is treated as a signed value, except in the case of a trap instruction.
- The specifier *offset* represents a location where a 16-bit number should be specified. This number is always treated as a signed value.
- The *offset* specifier is always found immediately preceding a parenthesized GPR specifier (rs_1 for load and store instructions) in order to represent the fact that the 16-bit *offset* value will be sign-extended and added to the contents of the specified GPR to form a 32-bit unsigned effective address.

[2] Programs such as the DLXsim simulator [4] may provide additional facilities for specifying numbers, registers, and addresses; when writing DLX code for input to these programs, it is strongly advised to read the corresponding documentation in addition to this handbook.

2.4 Instruction Operation Notation

When expressing the operation of DLX instructions, certain notational conventions are consistently used to represent data transfers, memory accesses, register accesses, bit fields, values, and operations on bit fields. These conventions are presented (with illustrative examples) in Tables 2.1 and 2.2. They are essential for correct interpretation of the remaining information and examples presented in this handbook and, therefore, should be studied carefully and thoroughly.

2.5 Load and Store Instructions

The DLX load and store instructions are divided into two categories: those that load and store general-purpose registers (i.e., integer loads and stores), and those that load and store floating-point registers (i.e., floating-point loads and stores). All of these instructions are in the I-type format. All memory references are performed using these load and store instructions. All memory accesses must be aligned on a byte, halfword, or word boundary, as appropriate. Double-precision floating-point operands must be aligned on a word boundary. All effective addresses for all memory accesses are created by adding a 16-bit signed *offset* to the contents of one of the general-purpose registers.

2.5.1 General-Purpose Registers

The DLX instructions for loading to and storing from general-purpose registers are LB, LBU, SB, LH, LHU, SH, LW, and SW.

Integer loads may be performed with any of the general-purpose registers as the destination register, except that loading R0 has no effect. The byte and halfword loads (LB, LBU, LH, and LHU) place the loaded byte or halfword into the least significant byte or halfword, respectively, of the destination register. The remaining portion of the destination register is filled with either the sign extension (for LB and LH) or '0's (for LBU and LHU). The load word (LW) instruction fills the entire destination register with the 32-bit loaded word.

Integer stores may use any of the general-purpose registers as the source of the data to be stored. Byte and halfword stores (SB and SH, respectively) write the least significant byte or halfword, respectively, of the contents of the source register to memory at the effective address. The store word (SW) instruction writes the entire 32-bit contents of the source register to memory at the effective address.

Notation	Example	Meaning
$\leftarrow_n$	R1 $\leftarrow_{32}$ (R0)	Data transfer of an n-bit field (e.g., transfer the 32-bit contents of GPR R0 to GPR R1).
M{*addr*}	R1 $\leftarrow_{32}$ M{100}	Memory access of one, two, or four bytes (as required), beginning at address *addr* (e.g., transfer the bytes in memory at locations 100, 101, 102, and 103 into GPR R1).
(X)	(R1)	The full contents of register X (e.g., the 32-bit contents of GPR R1).
X_n	$offset_0$	Select bit n from the bit field X (e.g., the sign bit of the 16-bit *offset*). (Bits are numbered from the most significant bit (MSB) starting at 0.)
$X_{m..n}$	$(R1)_{24..31}$	Select the range of bits m through n from the bit field X (e.g., the least significant byte of the 32-bit contents of GPR R1). (Bits are numbered from the most significant bit (MSB) starting at 0.)
X^n	$(name_0)^{16}$	Replicate the bit field X n times (e.g., the sign bit of the *name* field is replicated 16 times to form a 16-bit field).
X+1	(F2+1)	The register following register X in ascending numerical order (e.g., the contents of FPR F3).
$X \parallel Y$	$(offset_0)^{16} \parallel offset$	Concatenate two fields (e.g., the 16-bit *offset* field is sign-extended to 32 bits).
0	(R1) = 0	The numerical value zero (e.g., this condition is true when the value of the contents of GPR R1 is zero).
'0', '1'	$'0'^{16} \parallel immediate$	The single-bit fields '0' and '1' (e.g., the 16-bit *immediate* field is zero-extended to 32 bits).

Table 2.1: Instruction Operation Notation—Fields

Notation	Example	Meaning
$+, -, \times, \div$	(R1) − (R2)	Arithmetic operations on signed, unsigned, and floating-point values (e.g., arithmetically subtract the contents of GPR R2 from the contents of GPR R1).
$\&, \mid, \oplus$	(R1) $\oplus$ (R2)	Bitwise logical operations (and, or, and exclusive-or) on 32-bit fields (e.g., compute the bitwise exclusive-or of the contents of GPR R1 and the contents of GPR R2).
$=, \neq, <, \leq, >, \geq$	(R1) $\neq$ (R2)	Relational operators for signed, unsigned, and floating-point values (e.g., this condition is true when the contents of GPR R1 are not equal to the contents of GPR R2).
if *cond* then *oper*	if (R1) = 0 then $PC \leftarrow_{32}$ 200	If condition *cond* is true, then execute operation *oper* (e.g., if the value of the contents of GPR R1 is zero, then branch to the instruction in memory at location 200).
if *cond* then $oper_1$ else $oper_2$	if (R1) = (R2) then R3 $\leftarrow_{32}$ ('0'^{31} || '1') else R3 $\leftarrow_{32}$ ('0'^{32})	If condition *cond* is true, then execute operation $oper_1$, otherwise execute operation $oper_2$ (e.g., if the contents of GPR R1 and the contents of GPR R2 are equal, then place a value of 1 into GPR R3, otherwise place a value of 0 into GPR R3).
fmt2{*fmt1*[*X*]}	FxPI{SPFP[F3]}	Interpreting the bit field X in *fmt1*, convert it to *fmt2*, where *fmt1* and *fmt2* may be FxPI (fixed-point integer), SPFP (single-precision floating-point), or DPFP (double-precision floating-point) (e.g., interpreting the contents of FPR F3 as a single-precision floating-point value, convert this value to fixed-point integer format).

Table 2.2: Instruction Operation Notation—Operations

2.5.2 Floating-Point Registers

The DLX load and store instructions for single-precision and double-precision floating-point registers are LF, SF, LD, and SD.

Floating-point loads may be performed with any of the floating-point registers. Single-precision loads (LF) place 32 bits from memory into a single floating-point register. Double-precision loads (LD) place 64 bits from memory into an even/odd pair of floating-point registers.

Floating-point stores may use any individual (for single-precision, SF) or even/odd pair (for double-precision, SD) of floating-point register(s) to supply the data to be stored. Single-precision stores (SF) write the entire 32-bit contents of the source register to memory at the effective address. Double-precision stores (SD) write 64 bits, obtained from the concatenation of the contents of an even/odd pair of floating-point registers specified by the source register, to memory at the effective address.

2.6 Move Instructions

The DLX move instructions for general-purpose and floating-point registers are MOVI2S, MOVS2I, MOVF, MOVD, MOVFP2I, and MOVI2FP. All of these instructions are in the R-type format.

32-bit data transfers between the general-purpose registers and the special registers (*IAR* and *FPSR*) are performed with the MOVI2S and MOVS2I instructions. For the MOVI2S instruction, the source register is specified and the destination register is *IAR*. For the MOVS2I instruction, the source register is *IAR*, the destination register is specified.

Data contained in the floating-point registers may be transferred to other floating-point registers either 32 or 64 bits at a time. The MOVF instruction transfers the 32-bit contents of any individual floating-point register to any other individual floating-point register. The MOVD instruction transfers 64 bits, obtained from the concatenation of the contents of any even/odd pair of floating-point registers, to another even/odd pair of floating-point registers. The pairs of registers are specified by the even register number only.

32-bit data transfers between the floating-point registers and the general-purpose registers are performed with the MOVFP2I and MOVI2FP instructions. These instructions transfer the contents of any individual floating-point register to any general-purpose register (MOVFP2I), and the contents of any general-purpose register to any indi-

vidual floating-point register (MOVI2FP). Therefore, moving a double-precision floating-point value to two general-purpose registers requires two MOVFP2I instructions.

2.7 Arithmetic and Logical Instructions

The DLX arithmetic and logical instructions are divided into four categories: arithmetic, logical, shift, and set-on-comparison. With the exception of those for multiplication and division, the DLX arithmetic and logical instructions operate on signed and unsigned integer values stored in the general-purpose registers and 16-bit signed or unsigned immediates supplied in the instructions. The DLX instructions for multiplication and division operate on signed and unsigned integer values stored in the floating-point registers only. Excluding LHI, each of the DLX arithmetic and logical instructions are provided in both the R-type format and the I-type format (indicated by the suffix 'I' appended to the mnemonic for the R-type version of each instruction). The LHI (load high immediate) instruction is in the I-type format and has no R-type counterpart.

2.7.1 Arithmetic Instructions

The DLX integer arithmetic instructions are ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI, MULT, MULTU, DIV, and DIVU.

Addition and subtraction operations on the contents of two general-purpose registers are performed with the ADD, ADDU, SUB, and SUBU instructions. All of these instructions are in the R-type format. The ADD and SUB instructions treat the contents of the source registers as signed (two's complement) integers, while the ADDU and SUBU instructions treat the contents of the source registers as unsigned integers. The ADD and SUB instructions generate an arithmetic overflow exception when the result of their operations is greater than $2^{31} - 1$ (i.e., `> 0x7fffffff`) or less than -2^{31} (i.e., `< 0x80000000`), respectively. The ADDU and SUBU instructions do not generate an arithmetic overflow exception under any circumstances.

Addition and subtraction operations on the contents of a general-purpose register and a 16-bit immediate are performed with the ADDI, ADDUI, SUBI, and SUBUI instructions. All of these instructions are in the I-type format. The ADDI and SUBI instructions treat the contents of the source register as a signed (two's complement) integer, and sign-extend the 16-bit *immediate* field to form the second 32-bit signed

operand. The ADDUI and SUBUI instructions treat the contents of the source register as an unsigned integer, and zero-extend the 16-bit *immediate* field to form the second 32-bit unsigned operand. The ADDI and SUBI instructions generate an arithmetic overflow exception when the result of their operations is greater than $2^{31} - 1$ (i.e., `> 0x7fffffff`) or less than -2^{31} (i.e., `< 0x80000000`), respectively. The ADDUI and SUBUI instructions do not generate an arithmetic overflow exception under any circumstances.

Multiplication and division of integer values are performed with the MULT, MULTU, DIV, and DIVU instructions. All of these instructions are in the R-type format, and may obtain their operands from individual floating-point registers only. The result of these operations must be placed into an individual floating-point register. Therefore, multiplication and division of integer values stored in general-purpose registers require the additional use of the MOVI2FP and MOVFP2I instructions. The MULT and DIV instructions treat the contents of the source registers as signed (two's complement) integers, while the MULTU and DIVU instructions treat the contents of the source registers as unsigned integers.

2.7.2 Logical Instructions

The DLX logical instructions are AND, ANDI, OR, ORI, XOR, XORI, and LHI.

Bitwise logical and, or, and xor operations on the contents of two general-purpose registers are performed with the AND, OR, and XOR instructions, respectively. These instructions are all in the R-type format. They perform their respective bitwise logical operations on the full contents of two general-purpose registers.

Bitwise logical and, or, and xor operations on the contents of a general-purpose register and a 16-bit unsigned immediate are performed with the ANDI, ORI, and XORI instructions, which are all in the I-type format. These instructions perform their respective bitwise logical operations on the full contents of a general-purpose register with the 16-bit *immediate* field after it is zero-extended to form a 32-bit operand.

The LHI (load high immediate) instruction places its 16-bit *immediate* field into the most significant portion of the destination register and fills the remaining portion of the destination register with '0's. This makes it possible to create a full 32-bit constant in a general-purpose register in two instructions (accomplished by an LHI instruction followed by an ADDI instruction, where the source and destination registers of

the ADDI instruction are the same as the destination register of the LHI instruction).

2.7.3 Shift Instructions

The DLX shift instructions are SLL, SLLI, SRL, SRLI, SRA, and SRAI. Shift amounts are specified by either the value of the contents of a general-purpose register (for SLL, SRL, and SRA) or the value of a 16-bit *immediate* field. In either case, only the five low-order bits are considered, so that the maximum shift amount is 31 bits.

Logical shifts are performed with the SLL, SLLI, SRL, and SRLI instructions. The SLL and SRL instructions shift the contents of a general-purpose register left and right, respectively, by the number of bits specified by the value of the contents of any general-purpose register. The SLLI and SRLI instructions shift the contents of a general-purpose register left and right, respectively, by the number of bits specified by the value of the 16-bit *immediate* field. The SLL and SLLI instructions place '0's into the number of low-order bits specified by the shift amount. The SRL and SRLI instructions place '0's into the number of high-order bits specified by the shift amount.

Arithmetic shifts are performed with the SLL, SLLI, SRA, and SRAI instructions. The SLL and SLLI instructions double as left-arithmetic shifts since sign extension is not an issue when shifting bits in this direction. The SRA instruction shifts the contents of a general-purpose register to the right by the number of bits specified by the value of the contents of any general-purpose register. The SRAI instruction shifts the contents of a general-purpose register to the right by the number of bits specified by the value of its 16-bit *immediate* field. Both the SRA and SRAI instructions place the sign bit of the shifted register's contents into the number of high-order bits specified by the shift amount.

2.7.4 Set-On-Comparison Instructions

The DLX integer set-on-comparison instructions are SLT, SLTI, SGT, SGTI, SLE, SLEI, SGE, SGEI, SEQ, SEQI, SNE, and SNEI. All of these instructions set the destination register to a value of 1 when the comparison result is 'true,' and set the destination register to a value of 0 when the comparison result is 'false.' The six types of comparisons provided for by these instructions are less than ($<$), greater than ($>$), less than or equal to ($\leq$), greater than or equal to ($\geq$), equal to ($=$), and not equal to ($\neq$).

Comparisons between the contents of two general-purpose registers are performed with the SLT, SGT, SLE, SGE, SEQ, and SNE instructions. The register contents are treated as 32-bit signed (two's complement) integers and the values are arithmetically compared. Comparisons between the contents of a general-purpose register and a 16-bit *immediate* field are performed with the SLTI, SGTI, SLEI, SGEI, SEQI, and SNEI instructions. The register contents and the sign-extended 16-bit *immediate* field are treated as 32-bit signed integers and the values are arithmetically compared.

2.8 Floating-Point Instructions

The DLX floating-point instructions are divided into three categories: arithmetic, conversion, and set-on-comparison.[3] All floating-point instructions operate on floating-point values stored in either an individual (for single-precision) or an even/odd pair of (for double-precision) floating-point register(s), and all are in the R-type format.

2.8.1 Arithmetic Instructions

The DLX floating-point arithmetic instructions are ADDF, ADDD, SUBF, SUBD, MULTF, MULTD, DIVF, and DIVD.

Addition, subtraction, multiplication, and division operations on single-precision floating-point values are performed with the ADDF, SUBF, MULTF, and DIVF instructions. Each of these instructions interprets the contents of two individual floating-point registers as single-precision floating-point values, performs the appropriate operation on these values, rounds the result to single-precision floating-point format, and then places the rounded result into an individual floating-point register.[4]

Addition, subtraction, multiplication, and division operations on double-precision floating-point values are performed with the ADDD, SUBD, MULTD, and DIVD instructions. Each of these instructions interprets the contents of two even/odd pairs of floating-point registers as double-precision floating-point values, performs the appropriate operation on these values, rounds the result to double-precision floating-point

[3]For information regarding the IEEE 754 standard, refer to the *ANSI/IEEE Std 754-1985 Standard for Binary Floating-Point Arithmetic.*

[4]The method by which to specify which IEEE rounding mode should be used is not architected in the Hennessy & Patterson textbook [2]. Typically (for the MIPS R2000 architecture [3], for example), this is accomplished by setting two rounding mode bits of an FPU control/status register.

format, and then places the rounded result into an even/odd pair of floating-point registers.

2.8.2 Convert Instructions

The DLX integer/floating-point convert instructions are CVTF2D, CVTF2I, CVTD2F, CFTD2I, CVTI2F, and CVTI2D.

Conversion operations on single-precision floating-point values are performed with the CVTF2D and CVTF2I instructions. Both of these instructions interpret the contents of an individual floating-point register as a single-precision floating-point value. The CVTF2D instruction arithmetically converts this source value to double-precision floating-point format to form a 64-bit result, which is then placed into an even/odd pair of floating-point registers. The CVTF2I instruction arithmetically converts the source value to fixed-point integer format to form a 32-bit result, which is then placed into an individual floating-point register.

Conversion operations on double-precision floating-point values are performed with the CVTD2F and CVTD2I instructions. Both of these instructions interpret the contents of an even/odd pair of floating-point registers as a double-precision floating-point value. The CVTD2F instruction arithmetically converts this source value to single-precision floating-point format to form a 32-bit result, which is then placed into an individual floating-point register. The CVTD2I instruction arithmetically converts the source value to fixed-point integer format to form a 32-bit result, which is then placed into an individual floating-point register.

Conversion operations on fixed-point integer values are performed with the CVTI2F and CVTI2D instructions. Both of these instructions interpret the contents of an individual floating-point register as a fixed-point integer value. The CVTI2F instruction arithmetically converts this source value to single-precision floating-point format to form a 32-bit result, which is then placed into an individual floating-point register. The CVTI2D instruction arithmetically converts the source value to double-precision floating-point format to form a 64-bit result, which is then placed into an even/odd pair of floating-point registers.

2.8.3 Set-On-Comparison Instructions

The DLX floating-point set-on-comparison instructions are LTF, LTD, GTF, GTD, LEF, LED, GEF, GED, EQF, EQD, NEF, and NED. All

of these instructions set the floating-point status register (*FPSR*) to '1' when the comparison result is 'true,' and set the *FPSR* to '0' when the comparison result is 'false.' The six types of comparisons provided for by these instructions are less than ($<$), greater than ($>$), less than or equal to ($\leq$), greater than or equal to ($\geq$), equal to ($=$), and not equal to ($\neq$).

Comparisons between single-precision floating-point values are performed with the LTF, GTF, LEF, GEF, EQF, and NEF instructions. Each of these instructions interprets the contents of two individual floating-point registers as single-precision floating-point values and then arithmetically compares them. Comparisons between double-precision floating-point values are performed with the LTD, GTD, LED, GED, EQD, and NED instructions. Each of these instructions interprets the contents of two even/odd pairs of floating-point registers as double-precision floating-point values and then arithmetically compares them.

2.9 Jump and Branch Instructions

The DLX jump and branch instructions are J, JR, JAL, JALR, BEQZ, BNEZ, BFPT, and BFPF. The J and JAL instructions are in the J-type format, while the remaining jump and branch instructions are in the I-type format. All DLX control transfers (i.e., jumps and branches) occur after a delay of one instruction.[5] As a result, a *delay slot* is created immediately following each jump and branch instruction.

Conditional control transfers are performed with the branch instructions (BEQZ, BNEZ, BFPT, and BFPF). These instructions do not attempt to store any return address information. Therefore, they should not be used for procedure calls. The branch target address for these instructions is computed by sign-extending the 16-bit *name* field to a 32-bit signed (two's complement) value and adding this value to the address of the instruction in the delay slot (i.e., $(PC) + 4$). The branch condition for the BEQZ and BNEZ instructions may be based on the value of any general-purpose register. The BEQZ instruction executes a branch to the branch target address when the value of the source register is 0, and the BNEZ instruction executes a branch to the branch target address when the value of the source register is not equal to 0. The branch condition for the BFPT and BFPF instructions is based on the contents of the floating-point status register (*FPSR*). The BFPT instruction executes a branch to the branch target address when the *FPSR*

[5] This is commonly referred to as a *delayed branch*.

contains a '1' ('true'), and the BFPF instruction executes a branch to the branch target address when the *FPSR* contains a '0' ('false').

Unconditional control transfers are performed with the jump instructions (J, JR, JAL, and JALR). The target address for the J and JAL instructions is computed by sign-extending the 26-bit *name* field to a 32-bit signed (two's complement value) and adding this value to the address of the instruction in the delay slot (i.e., $(PC) + 4$). The target address for the JR and JALR instructions may be obtained from the 32-bit unsigned contents of any general-purpose register. All of the jump instructions (J, JR, JAL, and JALR) execute an unconditional jump to the target address. The jump and link instructions (JAL and JALR) place the address of the instruction after the delay slot (i.e., $(PC) + 8$) into the general-purpose register R31. Either of these instructions may, therefore, be used for procedure calls. The jump instructions J and JR do not attempt to store any return address information.

2.10 Special Instructions

The DLX special instructions are TRAP, RFE, and NOP.

The TRAP and RFE instructions are in the J-type format. For the TRAP instruction, a target address is generated by zero-extending its unsigned 26-bit *name* field to a 32-bit absolute address. The TRAP instruction executes an unconditional control transfer to the target address and places the address of the instruction after the delay slot (i.e., $(PC) + 8$) into the special register (interrupt address register, or IAR).[6] Returns from a control transfer executed by a TRAP instruction are performed with the RFE (return from exception) instruction, which transfers the 32-bit contents of IAR to the program counter.

The NOP instruction is in the R-type format, with the rs_1, rs_2, and rd fields being unused. The passing of an instruction cycle without any operation being performed is specified by the NOP instruction. This instruction is likely to be used most often in the delay slot following jump and branch instructions (when the delay slot cannot be filled with a useful instruction).

[6] The DLXsim simulator [4] implements the TRAP instruction in such a way that there exists no delay slot.

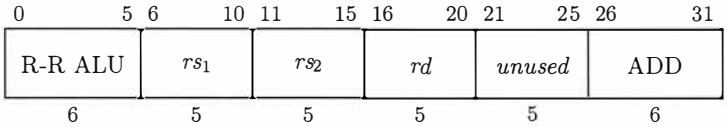
Chapter 3

DLX Instruction Set Details

This final chapter is a reference to the entire DLX instruction set. Its purpose is to provide a primary source of complete and concise reference information.

Each of the following pages covers a single DLX instruction. They are organized alphabetically by instruction mnemonic, and contain detailed information on every DLX instruction. For each instruction, a figure of the opcode format is provided, followed by the instruction's type, format, operation, and description.

Integer Add (Signed) **ADD**



Type: R

Format:

ADD rd, rs_1, rs_2

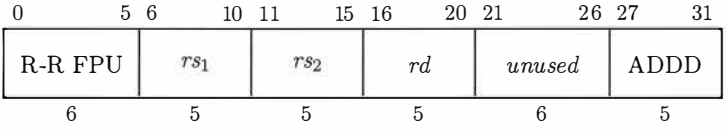
Operation:

$$rd \leftarrow_{32} (rs_1) + (rs_2)$$

Description:

The contents of GPR rs_1 and the contents of GPR rs_2 are arithmetically added to form a 32-bit two's complement result, which is then placed into GPR rd. An overflow exception occurs when the result of the addition operation is greater than $2^{31} - 1$ (i.e., `> 0x7fffffff`).

Add Double-Precision Floating-Point ADDD



Type: R

Format:

ADDD *rd*, rs_1, rs_2
where *rd*, rs_1, and rs_2 specify even-numbered FP registers

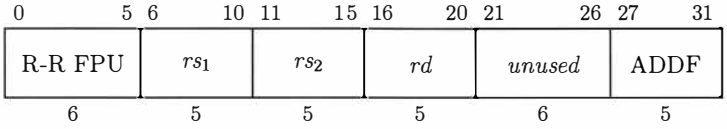
Operation:

$$rd \,\|\, rd{+}1 \leftarrow_{64} [(rs_1) \,\|\, (rs_1{+}1)] + [(rs_2) \,\|\, (rs_2{+}1)]$$

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and added using floating-point arithmetic to form a 64-bit result. The result is rounded to double-precision floating-point format and placed into the even/odd-numbered FP register pair specified by *rd*.

Add Floating-Point (Single-Precision) ADDF



Type: R

Format:

ADDF *rd*, rs_1, rs_2

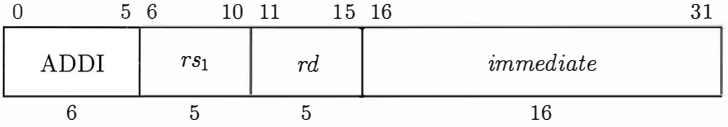
Operation:

$$rd \leftarrow_{32} (rs_1) + (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are interpreted in single-precision floating-point format and added using floating-point arithmetic to form a 32-bit result. The result is rounded to single-precision floating-point format and placed into FPR *rd*.

Integer Add Immediate (Signed) ADDI



Type: I

Format:

ADDI rd, rs_1, *immediate*

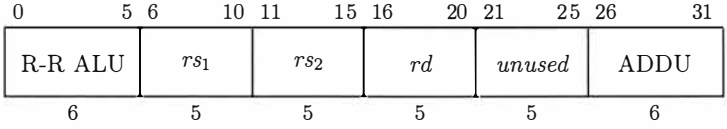
Operation:

$$rd \leftarrow_{32} (rs_1) + [(immediate_0)^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is sign-extended and arithmetically added to the contents of GPR rs_1 to form a 32-bit two's complement result, which is then placed into GPR rd. An overflow exception occurs when the result of the addition operation is greater than $2^{31} - 1$ (i.e., `> 0x7fffffff`).

Integer Add Unsigned — ADDU



Type: R

Format:

ADDU *rd*, rs_1, rs_2

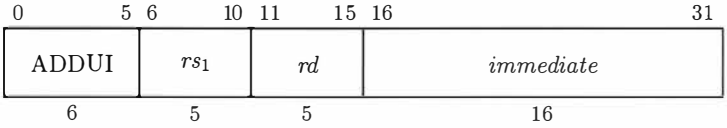
Operation:

$$rd \leftarrow_{32} (rs_1) + (rs_2)$$

Description:

The contents of GPR rs_1 and the contents of GPR rs_2 are arithmetically added to form a 32-bit unsigned result, which is then placed into GPR *rd*. No overflow exception occurs under any circumstances. As a result, this is the only difference between this instruction and the ADD instruction.

Integer Add Unsigned Immediate ADDUI



Type: I

Format:

ADDUI *rd*, rs_1, *immediate*

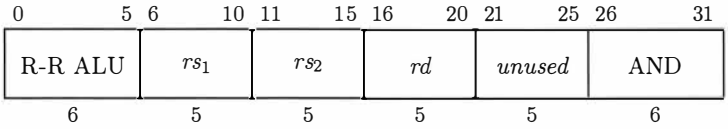
Operation:

$$rd \leftarrow_{32} (rs_1) + (\text{`0'}^{16} \parallel immediate)$$

Description:

The 16-bit *immediate* is zero-extended and arithmetically added to the contents of GPR rs_1 to form a 32-bit unsigned result, which is then placed into GPR *rd*. No overflow exception occurs under any circumstances. As a result, this is the only difference between this instruction and the ADDI instruction.

Logical And — AND



Type: R

Format:

AND *rd*, rs_1, rs_2

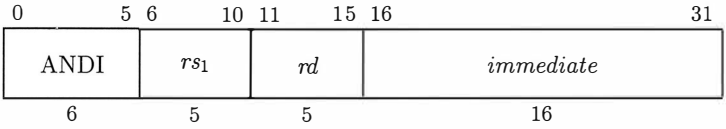
Operation:

$rd \leftarrow_{32} (rs_1)$ & (rs_2)

Description:

The contents of GPR rs_1 are combined with the contents of GPR rs_2 in a bitwise logical AND operation, and the result is placed into GPR *rd*.

Logical And Immediate (Signed) ANDI



Type: I

Format:

ANDI *rd*, rs_1, *immediate*

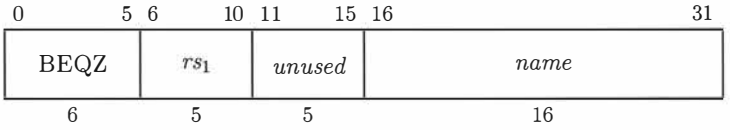
Operation:

$$rd \leftarrow_{32} (rs_1)\ \&\ [\text{`0'}^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of GPR rs_1 in a bitwise logical AND operation, and the result is placed into GPR *rd*.

Branch On Integer Equal To Zero — BEQZ



Type: I

Format:

BEQZ rs_1, *name*
where $-2^{15} \leq name < 2^{15}$

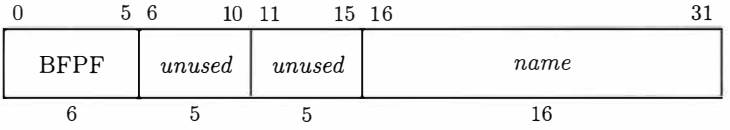
Operation:

if $(rs_1) = 0$ then $PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^{16} \parallel name]\}$

Description:

The 16-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit branch target address. If the contents of GPR rs_1 are equal to zero, then this branch target address is placed into the program counter.

Branch On Floating-Point False **BFPF**



Type: I

Format:

BFPF *name*
where $-2^{15} \leq name < 2^{15}$

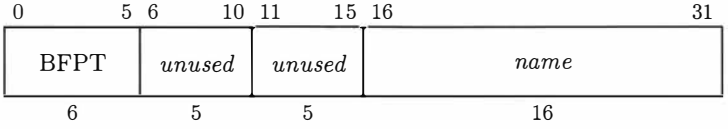
Operation:

if $(FPSR) = \text{'0'}$ then $PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^{16} \parallel name]\}$

Description:

The 16-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit branch target address. If the contents of *FPSR* are equal to zero, then this branch target address is placed into the program counter.

Branch On Floating-Point True **BFPT**



Type: I

Format:

BFPT *name*
where $-2^{15} \leq name < 2^{15}$

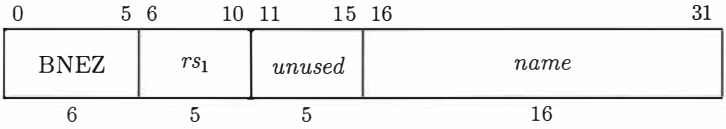
Operation:

if $(FPSR) =$ '1' then $PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^{16} \parallel name]\}$

Description:

The 16-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit branch target address. If the contents of *FPSR* are equal to one, then this branch target address is placed into the program counter.

Branch On Integer Not Equal To Zero **BNEZ**



Type: I

Format:

BNEZ rs_1, *name*
where $-2^{15} \leq name < 2^{15}$

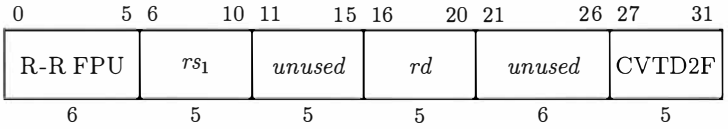
Operation:

if $(rs_1) \neq 0$ then $PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^{16} \parallel name]\}$

Description:

The 16-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit branch target address. If the contents of GPR rs_1 are not equal to zero, then this branch target address is placed into the program counter.

Convert DPFP To SPFP — CVTD2F



Type: R

Format:

CVTD2F rd, rs_1
where rs_1 specifies an even-numbered FP register

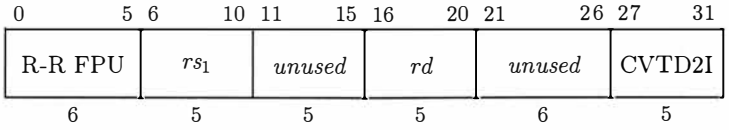
Operation:

$$rd \leftarrow_{32} \text{SPFP}\{\text{DPFP}[(rs_1) \parallel (rs_1+1)]\}$$

Description:

The contents of FPR rs_1 are concatenated with the contents of the following odd-numbered FP register to form a single 64-bit operand. This operand is interpreted in double-precision floating-point format and arithmetically converted to single-precision floating-point format to form a 32-bit result, which is then placed into FPR *rd*.

Convert DPFP To Integer — CVTD2I



Type: R

Format:

CVTD2I rd, rs_1
where rs_1 specifies an even-numbered FP register

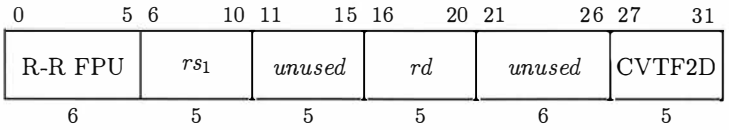
Operation:

$$rd \leftarrow_{32} \text{FxPI}\{\text{DPFP}[(rs_1) \parallel (rs_1{+}1)]\}$$

Description:

The contents of FPR rs_1 are concatenated with the contents of the following odd-numbered FP register to form a single 64-bit operand. This operand is interpreted in double-precision floating-point format and arithmetically converted to fixed-point integer format to form a 32-bit result, which is then placed into FPR *rd*.

Convert SPFP To DPFP — CVTF2D



Type: R

Format:

CVTF2D rd, rs_1
where rd specifies an even-numbered FP register

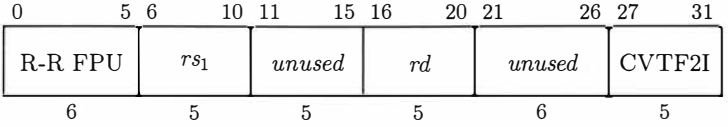
Operation:

$$rd \,||\, rd{+}1 \leftarrow_{64} \mathrm{DPFP}\{\mathrm{SPFP}[rs_1]\}$$

Description:

The contents of FPR rs_1 are interpreted in single-precision floating-point format and arithmetically converted to double-precision floating-point format to form a 64-bit result. This result is then placed into the even/odd-numbered FP register pair specified by rd.

Convert SPFP To Integer — CVTF2I



Type: R

Format:

CVTF2I *rd*, rs_1

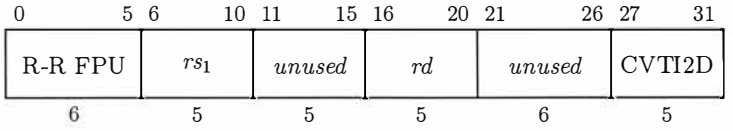
Operation:

$rd \leftarrow_{32} \text{FxPI}\{\text{SPFP}[rs_1]\}$

Description:

The contents of FPR rs_1 are interpreted in single-precision floating-point format and arithmetically converted to fixed-point integer format to form a 32-bit result, which is then placed into FPR *rd*.

Convert Integer To DPFP — CVTI2D



Type: R

Format:

CVTI2D *rd*, rs_1
where *rd* specifies an even-numbered FP register

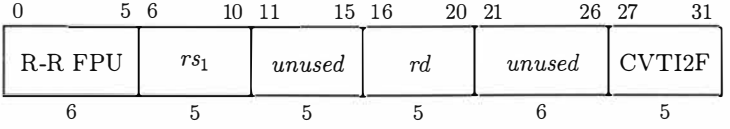
Operation:

$$rd \parallel rd{+}1 \leftarrow_{64} \text{DPFP}\{\text{FxPI}[rs_1]\}$$

Description:

The contents of FPR rs_1 are interpreted in fixed-point integer format and arithmetically converted to double-precision floating-point format to form a 64-bit result. This result is then placed into the even/odd-numbered FP register pair specified by *rd*.

Convert Integer To SPFP — CVTI2F



Type: R

Format:

CVTI2F *rd*, rs_1

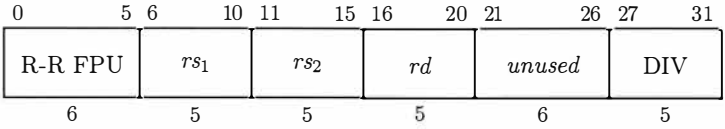
Operation:

$rd \leftarrow_{32} \text{SPFP}\{\text{FxPI}[rs_1]\}$

Description:

The contents of FPR rs_1 are interpreted in fixed-point integer format and arithmetically converted to single-precision floating-point format to form a 32-bit result, which is then placed into FPR *rd*.

Integer Divide (Signed) DIV



Type: R

Format:

DIV *rd*, rs_1, rs_2

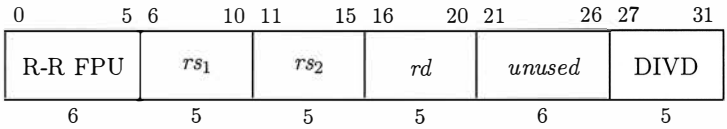
Operation:

$$rd \leftarrow_{32} (rs_1) \div (rs_2)$$

Description:

The contents of FPR rs_1 are arithmetically divided by the contents of FPR rs_2, treating both operands as 32-bit two's complement values, to form a 32-bit two's complement result, which is then placed into FPR *rd*.

Divide Double-Precision Floating-Point DIVD



Type: R

Format:

DIVD *rd*, rs_1, rs_2
where *rd*, rs_1, and rs_2 specify even-numbered FP registers

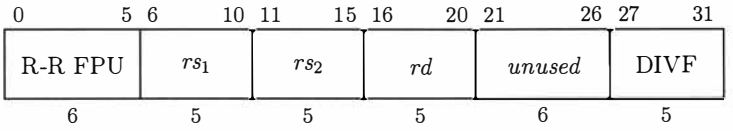
Operation:

$$rd \,||\, rd{+}1 \leftarrow_{64} [(rs_1) \,||\, (rs_1{+}1)] \div [(rs_2) \,||\, (rs_2{+}1)]$$

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and divided (the former by the latter) using floating-point arithmetic to form a 64-bit result. The result is rounded to double-precision floating-point format and placed into the even/odd-numbered FP register pair specified by *rd*.

Divide Floating-Point (Single-Precision) DIVF



Type: R

Format:

DIVF *rd*, rs_1, rs_2

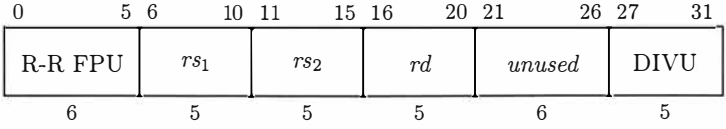
Operation:

$$rd \leftarrow_{32} (rs_1) \div (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are interpreted in single-precision floating-point format and divided (the former by the latter) using floating-point arithmetic to form a 32-bit result. The result is rounded to single-precision floating-point format and placed into FPR *rd*.

Integer Divide Unsigned — DIVU



Type: R

Format:

DIVU rd, rs_1, rs_2

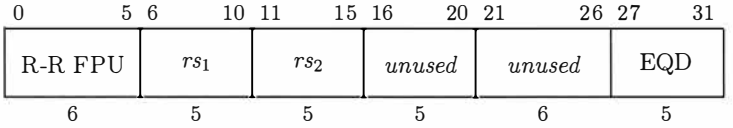
Operation:

$$rd \leftarrow_{32} (rs_1) \div (rs_2)$$

Description:

The contents of FPR rs_1 are arithmetically divided by the contents of FPR rs_2, treating both operands as 32-bit unsigned values, to form a 32-bit unsigned result, which is then placed into FPR rd.

Set On Equal To DPFP — EQD



Type: R

Format:

EQD rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

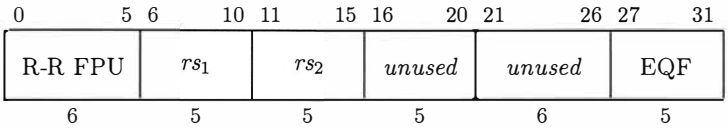
Operation:

if $[(rs_1) \parallel (rs_1+1)] = [(rs_2) \parallel (rs_2+1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Equal To SPFP **EQF**



Type: R

Format:

EQF *rd*, rs_1, rs_2

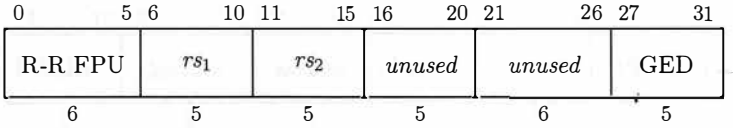
Operation:

if $(rs_1) = (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Greater Than Or Equal To DPFP GED



Type: R

Format:

GED rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

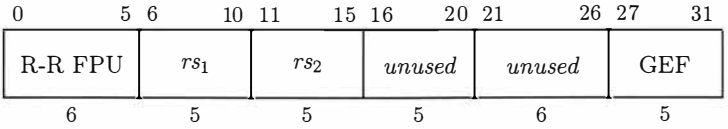
Operation:

if $[(rs_1) \parallel (rs_1+1)] \geq [(rs_2) \parallel (rs_2+1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is greater than or equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Greater Than Or Equal To SPFP — GEF



Type: R

Format:

GEF *rd*, rs_1, rs_2

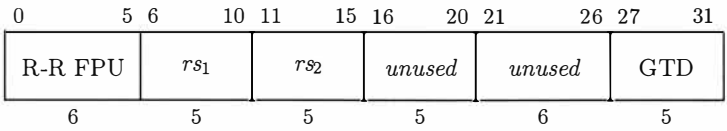
Operation:

if $(rs_1) \geq (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is greater than or equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Greater Than DPFP — GTD



Type: R

Format:

GTD rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

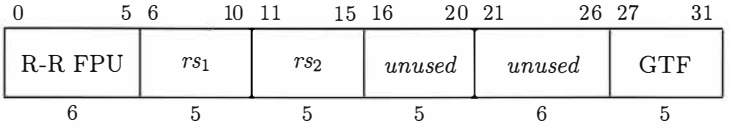
Operation:

if $[(rs_1) \parallel (rs_1+1)] > [(rs_2) \parallel (rs_2+1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is greater than the latter, the $FPSR$ is set to one, otherwise the $FPSR$ is set to zero.

Set On Greater Than SPFP — GTF



Type: R

Format:

GTF rd, rs_1, rs_2

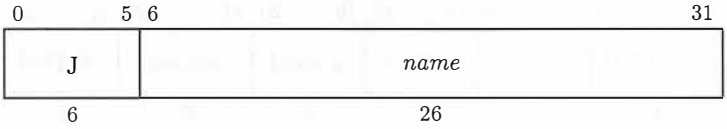
Operation:

if $(rs_1) > (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is greater than the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Jump J



Type: J

Format:

J *name*

where $-2^{25} \leq name < 2^{25}$

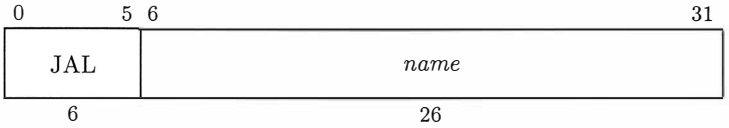
Operation:

$$PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^6 \parallel name]\}$$

Description:

The 26-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit target address. This target address is unconditionally placed into the program counter.

Jump And Link **JAL**



Type: J

Format:

JAL *name*
where $-2^{25} \leq name < 2^{25}$

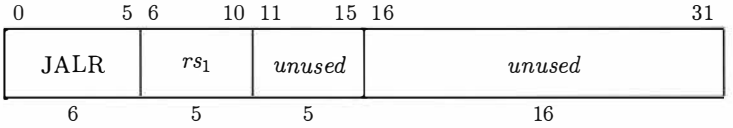
Operation:

$R31 \leftarrow_{32} [(PC) + 8]$
$PC \leftarrow_{32} \{[(PC) + 4] + [(name_0)^6 \parallel name]\}$

Description:

The 26-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit target address. This target address is unconditionally placed into the program counter. The address of the instruction after the delay slot is placed into GPR $R31$.

Jump And Link Register — JALR



Type: I

Format:

JALR rs_1

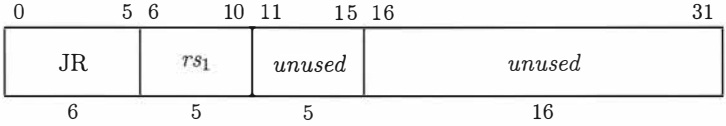
Operation:

$$R31 \leftarrow_{32} [(PC) + 8]$$
$$PC \leftarrow_{32} (rs_1)$$

Description:

The contents of GPR rs_1 are considered to be the target address, and they are unconditionally placed into the program counter. The address of the instruction after the delay slot is placed into GPR $R31$.

Jump Register **JR**



Type: I

Format:

JR rs_1

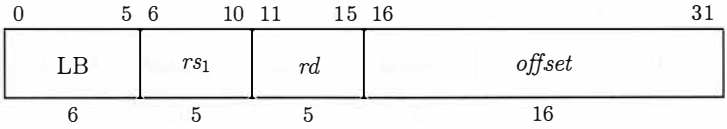
Operation:

$$PC \leftarrow_{32} (rs_1)$$

Description:

The contents of GPR rs_1 are considered to be the target address, and they are unconditionally placed into the program counter.

Load Byte (Signed) LB



Type: I

Format:

LB *rd*, *offset*(rs_1)

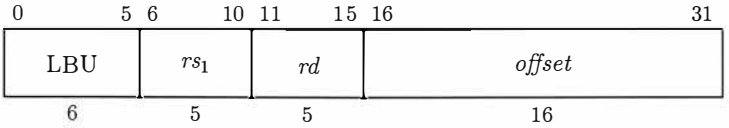
Operation:

$$rd \leftarrow_{32} \{\mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}_0\}^{24} \parallel \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the byte in memory at this effective address are sign-extended and loaded into GPR *rd*.

Load Byte Unsigned LBU



Type: I

Format:

LBU *rd*, *offset*(rs_1)

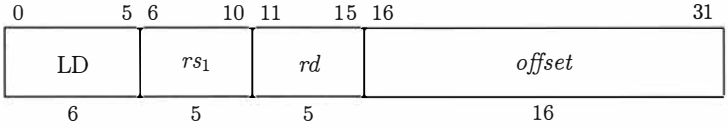
Operation:

$$rd \leftarrow_{32} \text{`0'}^{24} \,||\, \text{M}\{[(offset_0)^{16} \,\|\, offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the byte in memory at this effective address are zero-extended and loaded into GPR *rd*.

Load Double-Precision Floating-Point LD



Type: I

Format:

LD *rd*, *offset*(rs_1)
where *rd* specifies an even-numbered FP register

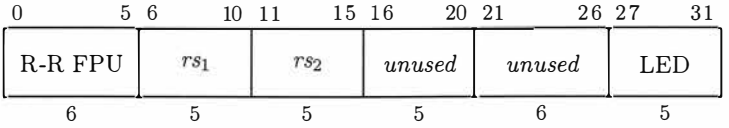
Operation:

$$rd \parallel rd{+}1 \leftarrow_{64} \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the two consecutive words in memory at this effective address are loaded into the even/odd-numbered FP register pair specified by *rd*.

Set On Less Than Or Equal To DPFP — LED



Type: R

Format:

LED rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

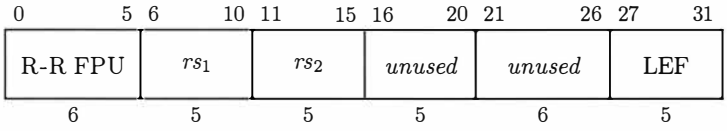
Operation:

if $[(rs_1) \parallel (rs_1+1)] \leq [(rs_2) \parallel (rs_2+1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is less than or equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Less Than Or Equal To SPFP — LEF



Type: R

Format:

LEF *rd*, rs_1, rs_2

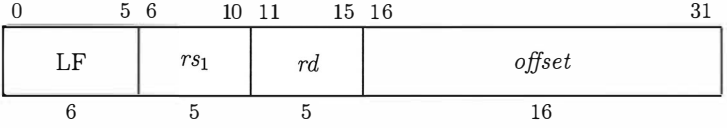
Operation:

if $(rs_1) \leq (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is less than or equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Load Floating-Point (Single-Precision) LF



Type: I

Format:

LF *rd*, *offset*(rs_1)

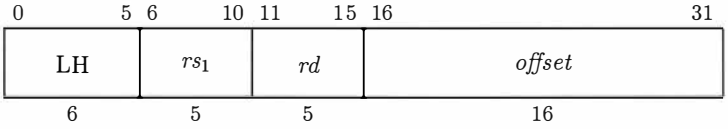
Operation:

$$rd \leftarrow_{32} \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the word in memory at this effective address are loaded into FPR *rd*.

Load Halfword (Signed) LH



Type: I

Format:

LH rd, $offset(rs_1)$

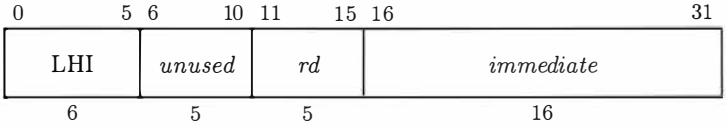
Operation:

$$rd \leftarrow_{32} \{\mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}_0\}^{16} \parallel \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the halfword in memory at this effective address are sign-extended and loaded into GPR rd.

Load High Immediate **LHI**



Type: I

Format:

LHI *rd*, *immediate*

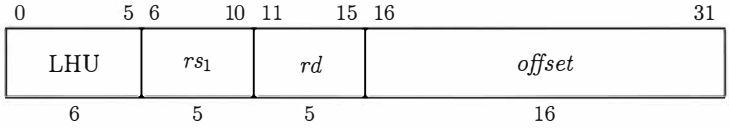
Operation:

$rd \leftarrow_{32} immediate \parallel \text{`0'}^{16}$

Description:

The 16-bit *immediate* is concatenated with 16 zero-bits, and the result is placed into GPR *rd*.

Load Halfword Unsigned — LHU



Type: I

Format:

LHU *rd*, *offset*(rs_1)

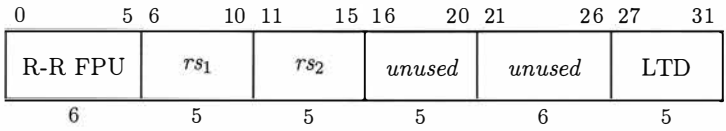
Operation:

$$rd \leftarrow_{32} \text{`0'}^{16} \parallel \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the halfword in memory at this effective address are zero-extended and loaded into GPR *rd*.

Set On Less Than DPFP — LTD



Type: R

Format:

LTD rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

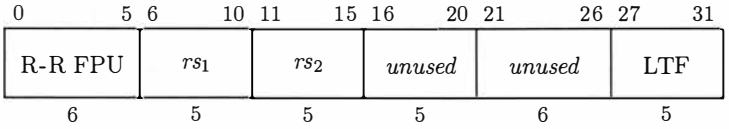
Operation:

if $[(rs_1) \parallel (rs_1{+}1)] < [(rs_2) \parallel (rs_2{+}1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is less than the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Set On Less Than SPFP — LTF



Type: R

Format:

LTF *rd*, rs_1, rs_2

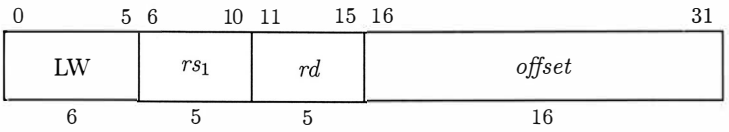
Operation:

if $(rs_1) < (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is less than the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

Load Word **LW**



Type: I

Format:

LW *rd*, *offset*(rs_1)

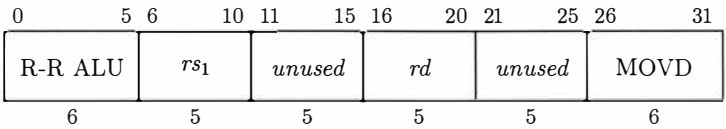
Operation:

$$rd \leftarrow_{32} \mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of the word in memory at this effective address are loaded into GPR *rd*.

Move Double-Precision Floating-Point MOVD



Type: R

Format:

MOVD *rd*, rs_1
where *rd* and rs_1 specify even-numbered FP registers

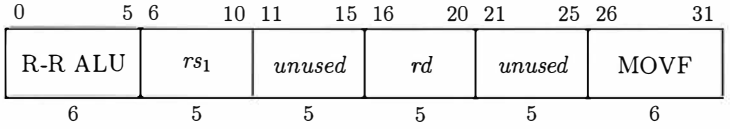
Operation:

$$rd \leftarrow_{32} (rs_1)$$
$$rd{+}1 \leftarrow_{32} (rs_1{+}1)$$

Description:

The contents of FPR rs_1 are loaded into FPR *rd*, and the contents of FPR $rs_1{+}1$ are loaded into FPR $rd{+}1$.

Move Floating-Point (Single-Precision) MOVF



Type: R

Format:

MOVF *rd*, rs_1

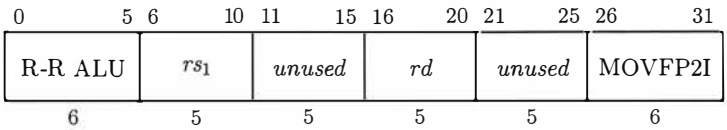
Operation:

$rd \leftarrow_{32} (rs_1)$

Description:

The contents of FPR rs_1 are loaded into FPR *rd*.

Move SPFP To Integer — MOVFP2I



Type: R

Format:

MOVFP2I *rd*, rs_1

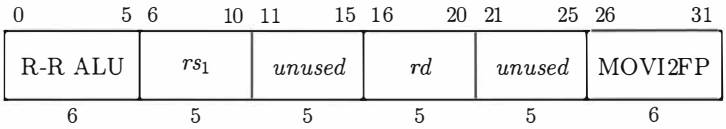
Operation:

$rd \leftarrow_{32} (rs_1)$

Description:

The contents of FPR rs_1 are loaded into GPR *rd*.

Move Integer To SPFP **MOVI2FP**



Type: R

Format:

MOVI2FP *rd*, rs_1

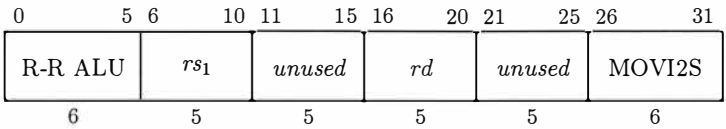
Operation:

$rd \leftarrow_{32} (rs_1)$

Description:

The contents of GPR rs_1 are loaded into FPR *rd*.

Move From GPR To Special Register **MOVI2S**



Type: R

Format:

MOVI2S *rd*, rs_1

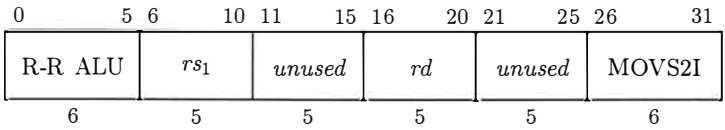
Operation:

$rd \leftarrow_{32} (rs_1)$

Description:

The contents of GPR rs_1 are loaded into special register *rd*.

Move From Special Register To GPR **MOVS2I**



Type: R

Format:

MOVS2I *rd*, rs_1

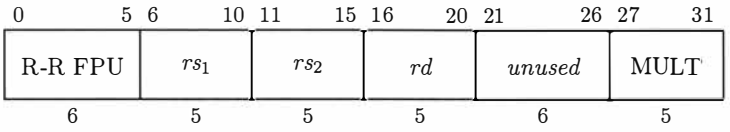
Operation:

$$rd \leftarrow_{32} (rs_1)$$

Description:

The contents of special register rs_1 are loaded into GPR *rd*.

Integer Multiply (Signed) MULT



Type: R

Format:

MULT *rd*, rs_1, rs_2

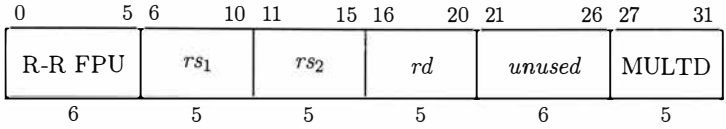
Operation:

$$rd \leftarrow_{32} (rs_1) \times (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are arithmetically multiplied, treating both operands as 32-bit two's complement values, to form a 32-bit two's complement result, which is then placed into FPR *rd*.

Multiply Double-Precision Floating-Point MULTD



Type: R

Format:

MULTD *rd*, rs_1, rs_2
where *rd*, rs_1, and rs_2 specify even-numbered FP registers

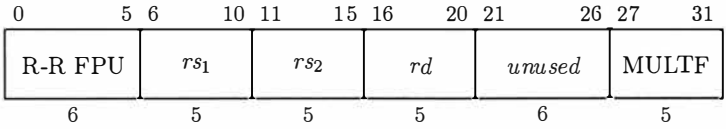
Operation:

$$rd \,||\, rd{+}1 \leftarrow_{64} [(rs_1) \,||\, (rs_1{+}1)] \times [(rs_2) \,||\, (rs_2{+}1)]$$

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and multiplied using floating-point arithmetic to form a 64-bit result. The result is rounded to double-precision floating-point format and placed into the even/odd-numbered FP register pair specified by *rd*.

Multiply Floating-Point (Single-Precision) MULTF



Type: R

Format:

MULTF rd, rs_1, rs_2

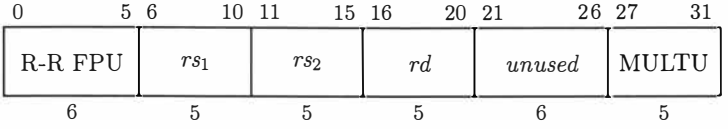
Operation:

$$rd \leftarrow_{32} (rs_1) \times (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are interpreted in single-precision floating-point format and multiplied using floating-point arithmetic to form a 32-bit result. The result is rounded to single-precision floating-point format and placed into FPR rd.

Integer Multiply Unsigned MULTU



Type: R

Format:

MULTU rd, rs_1, rs_2

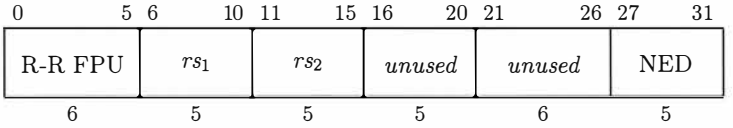
Operation:

$$rd \leftarrow_{32} (rs_1) \times (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are arithmetically multiplied, treating both operands as 32-bit unsigned values, to form a 32-bit unsigned result, which is then placed into FPR rd.

Set On Not Equal To DPFP — NED



Type: R

Format:

NED rs_1, rs_2
where rs_1 and rs_2 specify even-numbered FP registers

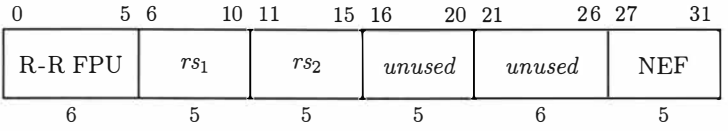
Operation:

if $[(rs_1) \parallel (rs_1+1)] \neq [(rs_2) \parallel (rs_2+1)]$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and arithmetically compared. If the former is not equal to the latter, the $FPSR$ is set to one, otherwise the $FPSR$ is set to zero.

Set On Not Equal To SPFP — NEF



Type: R

Format:

NEF *rd*, rs_1, rs_2

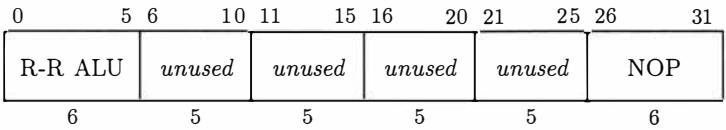
Operation:

if $(rs_1) \neq (rs_2)$ then $FPSR \leftarrow_1$ '1'
else $FPSR \leftarrow_1$ '0'

Description:

Interpreting the contents of GPR rs_1 and the contents of GPR rs_2 in single-precision floating-point format, if the former is not equal to the latter, the *FPSR* is set to one, otherwise the *FPSR* is set to zero.

No Operation — NOP



Type: R

Format:

NOP

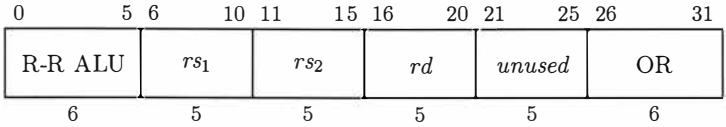
Operation:

(none)

Description:

No operation is performed.

Logical Or **OR**



Type: R

Format:

OR rd, rs_1, rs_2

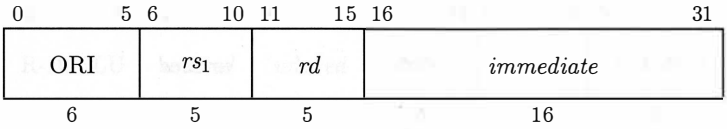
Operation:

$rd \leftarrow_{32} (rs_1) \mid (rs_2)$

Description:

The contents of GPR rs_1 are combined with the contents of GPR rs_2 in a bitwise logical OR operation, and the result is placed into GPR rd.

Logical Or Immediate (Signed) ORI



Type: I

Format:

ORI *rd*, rs_1, *immediate*

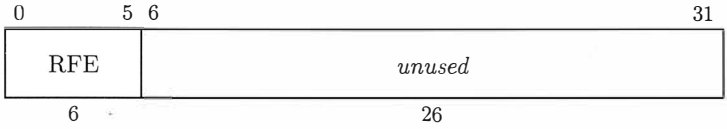
Operation:

$$rd \leftarrow_{32} (rs_1) \mid [\text{`0'}^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of GPR rs_1 in a bitwise logical OR operation, and the result is placed into GPR *rd*.

Return From Exception **RFE**



Type: J

Format:

RFE

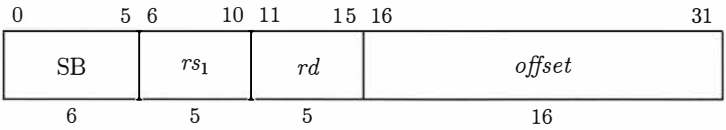
Operation:

$$PC \leftarrow_{32} (IAR)$$

Description:

The contents of special register *IAR* are considered to be the target address, and they are unconditionally placed into the program counter.

Store Byte — SB



Type: I

Format:

SB *offset*(rs_1), rd

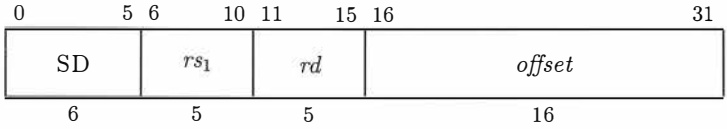
Operation:

$$M\{[(offset_0)^{16} \parallel offset] + (rs_1)\} \leftarrow_8 (rd)_{24..31}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The least significant byte of the contents of GPR rd is stored at this effective address.

Store Double-Precision Floating-Point SD



Type: I

Format:

SD $offset(rs_1), rd$
where rs_1 specifies an even-numbered FP register

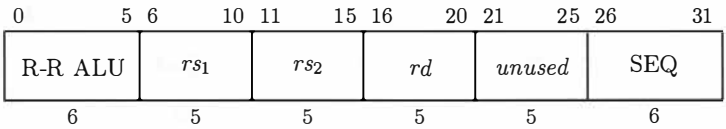
Operation:

$$M\{[(offset_0)^{16} \parallel offset] + (rs_1)\} \leftarrow_{64} (rd) \parallel (rd+1)$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of FPR *rd* and the contents of FPR *rd*+1 are concatenated to form a 64-bit result, which is then stored at the effective address.

Set On Equal To SEQ



Type: R

Format:

SEQ rd, rs_1, rs_2

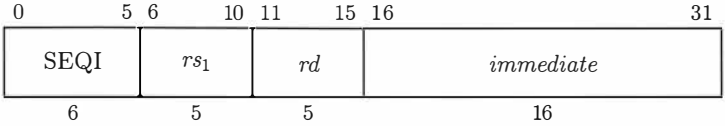
Operation:

$$\text{if } (rs_1) = (rs_2) \text{ then } rd \leftarrow_{-32} (\text{`0'}^{31} \,||\, \text{`1'})$$
$$\text{else } rd \leftarrow_{-32} (\text{`0'}^{32})$$

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR rd.

Set On Equal To Immediate **SEQI**



Type: I

Format:

SEQI rd, rs_1, $immediate$

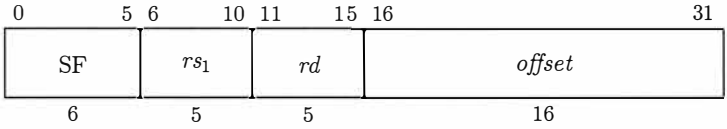
Operation:

$$\text{if } (rs_1) = (immediate_0)^{16} \parallel immediate \text{ then } rd \leftarrow_{32} (\text{`0'}^{31} \parallel \text{`1'}) \text{ else } rd \leftarrow_{32} (\text{`0'}^{32})$$

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR rd.

Store Floating-Point (Single-Precision) SF



Type: I

Format:

SF *offset*(rs_1), rd

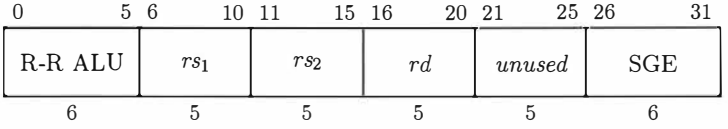
Operation:

$$M\{[(offset_0)^{16} \parallel offset] + (rs_1)\} \leftarrow_{32} (rd)$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of FPR rd are stored at this effective address.

Set On Greater Than Or Equal To SGE



Type: R

Format:

SGE *rd*, rs_1, rs_2

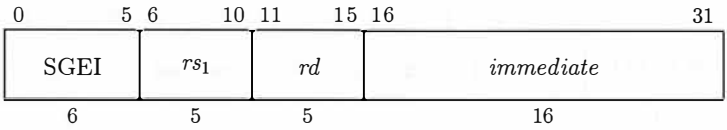
Operation:

if $(rs_1) \geq (rs_2)$ then $rd \leftarrow_{32}$ ('0'^{31} || '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is greater than or equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Greater Than Or Equal To Immediate **SGEI**



Type: I

Format:

SGEI *rd*, rs_1, *immediate*

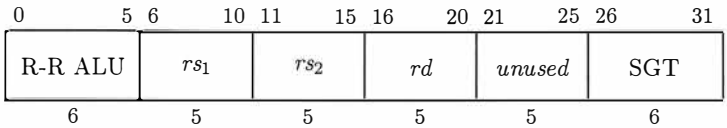
Operation:

if $(rs_1) \geq (immediate_0)^{16} \parallel immediate$ then $rd \leftarrow_{32}$ ('0'^{31} ‖ '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is greater than or equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Greater Than **SGT**



Type: R

Format:

SGT rd, rs_1, rs_2

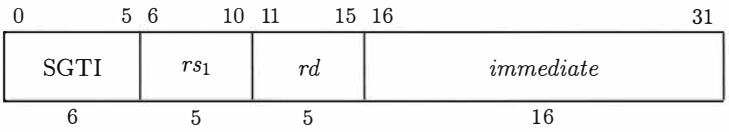
Operation:

if $(rs_1) > (rs_2)$ then $rd \leftarrow_{32}$ ('0'^{31} || '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is greater than the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Greater Than Immediate — SGTI



Type: I

Format:

SGTI *rd*, rs_1, *immediate*

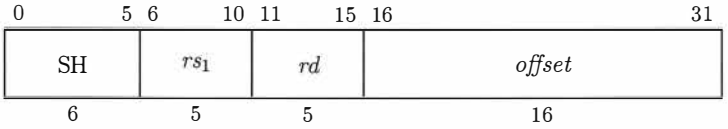
Operation:

if $(rs_1) > (immediate_0)^{16} \parallel immediate$ then $rd \leftarrow_{32}$ ('0'^{31} || '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is greater than the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Store Halfword **SH**



Type: I

Format:

SH $offset(rs_1)$, rd

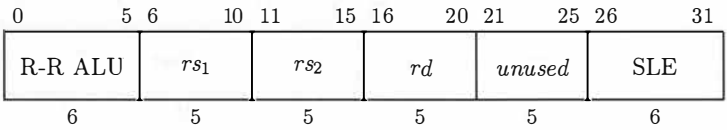
Operation:

$$M\{[(offset_0)^{16} \parallel offset] + (rs_1)\} \leftarrow_{16} (rd)_{16..31}$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The least significant halfword of the contents of GPR rd is stored at this effective address.

Set On Less Than Or Equal To **SLE**



Type: R

Format:

SLE *rd*, rs_1, rs_2

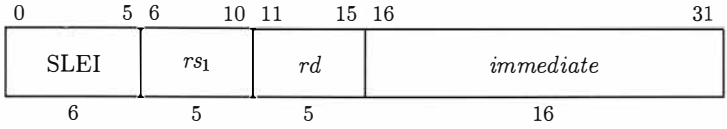
Operation:

if $(rs_1) \leq (rs_2)$ then $rd \leftarrow_{32}$ ('0'^{31} || '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is less than or equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Less Than Or Equal To Immediate **SLEI**



Type: I

Format:

SLEI rd, rs_1, *immediate*

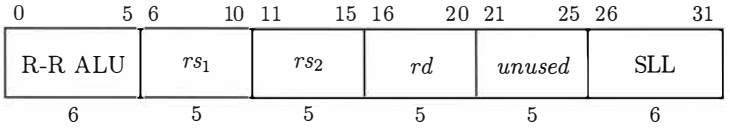
Operation:

if $(rs_1) \leq (immediate_0)^{16} \parallel immediate$ then $rd \leftarrow_{32}$ ('0'^{31} $\parallel$ '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is less than or equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Shift Left Logical — SLL



Type: R

Format:

SLL rd, rs_1, rs_2

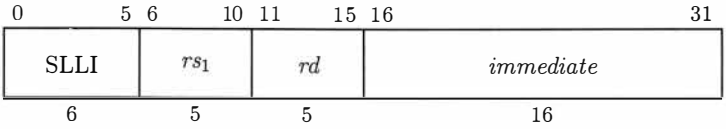
Operation:

$rd \leftarrow_{32} (rs_1)_{shamt_{..31}} \parallel {}^{\prime}0^{\prime shamt}$
where $shamt = (rs_2)_{27..31}$

Description:

The contents of GPR rs_1 are shifted left by the number of bits specified by the five low-order bits of the contents of GPR rs_2, inserting zeroes into the low-order bits, and the 32-bit result is placed into GPR rd.

Shift Left Logical Immediate — SLLI



Type: I

Format:

SLLI rd, rs_1, *immediate*

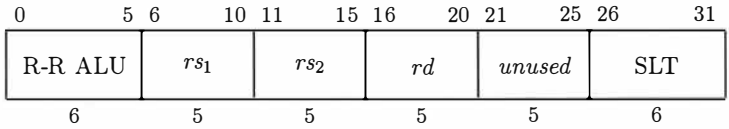
Operation:

$rd \leftarrow_{32} (rs_1)_{shamt..31} \parallel {}^{\prime}0^{\prime shamt}$
where $shamt = immediate_{27..31}$

Description:

The contents of GPR rs_1 are shifted left by the number of bits specified by the five low-order bits of *immediate*, inserting zeroes into the low-order bits, and the 32-bit result is placed into GPR rd.

Set On Less Than — SLT



Type: R

Format:

SLT rd, rs_1, rs_2

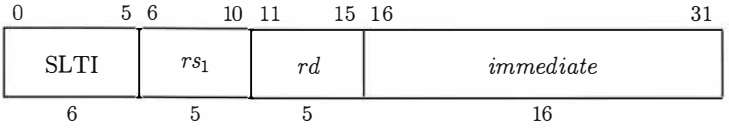
Operation:

$$\text{if } (rs_1) < (rs_2) \text{ then } rd \leftarrow_{32} (\text{`0'}^{31} \parallel \text{`1'})$$
$$\text{else } rd \leftarrow_{32} (\text{`0'}^{32})$$

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is less than the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR rd.

Set On Less Than Immediate — SLTI



Type: I

Format:

SLTI *rd*, rs_1, *immediate*

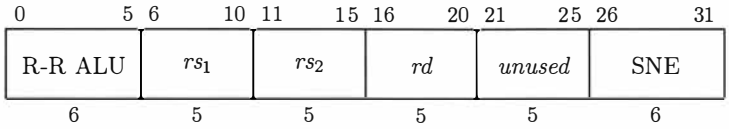
Operation:

if $(rs_1) < (immediate_0)^{16} \parallel immediate$ then $rd \leftarrow_{32}$ ('0'^{31} || '1')
else $rd \leftarrow_{32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is less than the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Not Equal To — SNE



Type: R

Format:

SNE *rd*, rs_1, rs_2

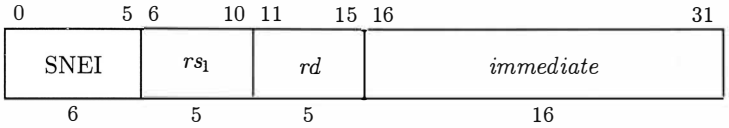
Operation:

$$\text{if } (rs_1) \neq (rs_2) \text{ then } rd \leftarrow_{32} (\text{`0'}^{31} \parallel \text{`1'})$$
$$\text{else } rd \leftarrow_{32} (\text{`0'}^{32})$$

Description:

Treating the contents of GPR rs_1 and the contents of GPR rs_2 as 32-bit two's complement integers, if the former is not equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Set On Not Equal To Immediate **SNEI**



Type: I

Format:

SNEI *rd*, rs_1, *immediate*

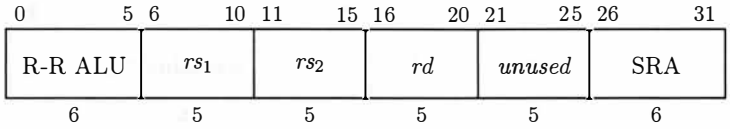
Operation:

if $(rs_1) \neq (immediate_0)^{16} \parallel immediate$ then $rd \leftarrow_{-32}$ ('0'^{31} || '1')
else $rd \leftarrow_{-32}$ ('0'^{32})

Description:

Treating the contents of GPR rs_1 and the sign-extended 16-bit *immediate* as 32-bit two's complement integers, if the former is not equal to the latter, the result is set to one, otherwise the result is set to zero. This 32-bit result is placed into GPR *rd*.

Shift Right Arithmetic SRA



Type: R

Format:

SRA rd, rs_1, rs_2

Operation:

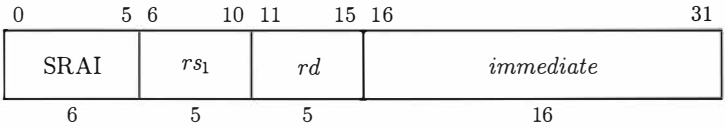
$$rd \leftarrow_{32} [(rs_1)_0]^{shamt} \parallel (rs_1)_{0..[31-shamt]}$$

where $shamt = (rs_2)_{27..31}$

Description:

The contents of GPR rs_1 are shifted right by the number of bits specified by the five low-order bits of the contents of GPR rs_2, sign-extending the high-order bits, and the 32-bit result is placed into GPR rd.

Shift Right Arithmetic Immediate — SRAI



Type: I

Format:

SRAI rd, rs_1, $immediate$

Operation:

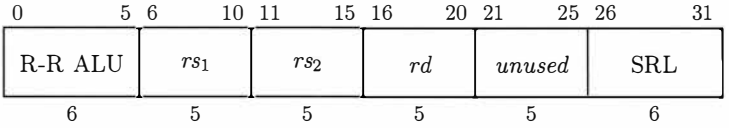
$$rd \leftarrow_{32} [(rs_1)_0]^{shami} \parallel (rs_1)_{0..[31-shamt]}$$

where $shamt = immediate_{27..31}$

Description:

The contents of GPR rs_1 are shifted right by the number of bits specified by the five low-order bits of *immediate*, sign-extending the high-order bits, and the 32-bit result is placed into GPR *rd*.

Shift Right Logical — SRL



Type: R

Format:

SRL rd, rs_1, rs_2

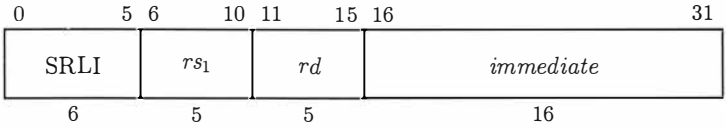
Operation:

$$rd \leftarrow_{32} \text{`0'}^{shamt} \parallel (rs_1)_{0..[31-shamt]}$$
where $shamt = (rs_2)_{27..31}$

Description:

The contents of GPR rs_1 are shifted right by the number of bits specified by the five low-order bits of the contents of GPR rs_2, inserting zeroes into the high-order bits, and the 32-bit result is placed into GPR rd.

Shift Right Logical Immediate — SRLI



Type: I

Format:

SRLI rd, rs_1, $immediate$

Operation:

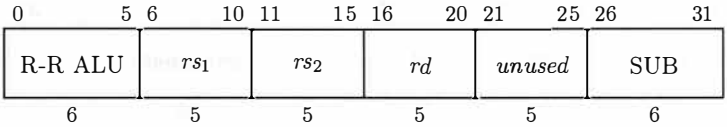
$$rd \leftarrow_{32} \text{`0'}^{shamt} \parallel (rs_1)_{0..[31-shamt]}$$

where $shamt = immediate_{27..31}$

Description:

The contents of GPR rs_1 are shifted right by the number of bits specified by the five low-order bits of *immediate*, inserting zeroes into the high-order bits, and the 32-bit result is placed into GPR *rd*.

Integer Subtract (Signed) SUB



Type: R

Format:

SUB rd, rs_1, rs_2

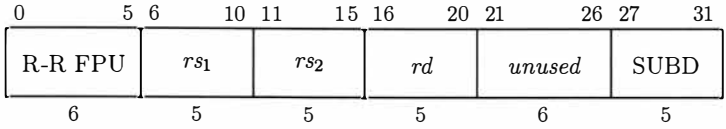
Operation:

$$rd \leftarrow_{32} (rs_1) - (rs_2)$$

Description:

The contents of GPR rs_2 are arithmetically subtracted from the contents of GPR rs_1 to form a 32-bit two's complement result, which is then placed into GPR rd. An overflow exception occurs when the result of the subtraction operation is less than -2^{31} (i.e., < 0x80000000).

Subtract Double-Precision Floating-Point SUBD



Type: R

Format:

SUBD *rd*, rs_1, rs_2

where *rd*, rs_1, and rs_2 specify even-numbered FP registers

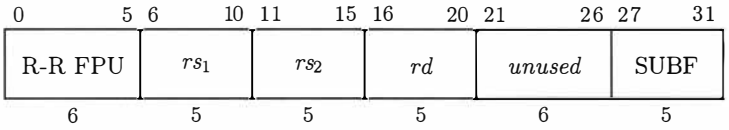
Operation:

$$rd \,||\, rd{+}1 \leftarrow_{64} [(rs_1) \,||\, (rs_1{+}1)] - [(rs_2) \,||\, (rs_2{+}1)]$$

Description:

The contents of the even-numbered FP registers rs_1 and rs_2 are individually concatenated with the contents of the respective following odd-numbered FP registers to form two 64-bit operands. These operands are interpreted in double-precision floating-point format and subtracted (the latter from the former) using floating-point arithmetic to form a 64-bit result. The result is rounded to double-precision floating-point format and placed into the even/odd-numbered FP register pair specified by *rd*.

Subtract Floating-Point (Single-Precision) SUBF



Type: R

Format:

SUBF rd, rs_1, rs_2

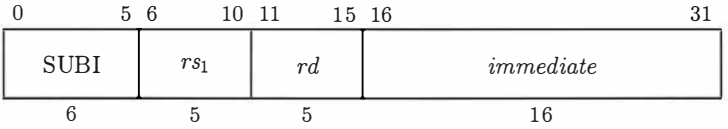
Operation:

$$rd \leftarrow_{32} (rs_1) - (rs_2)$$

Description:

The contents of FPR rs_1 and the contents of FPR rs_2 are interpreted in single-precision floating-point format and subtracted (the latter from the former) using floating-point arithmetic to form a 32-bit result. The result is rounded to single-precision floating-point format and placed into FPR *rd*.

Integer Subtract Immediate (Signed) SUBI



Type: I

Format:

SUBI *rd*, rs_1, *immediate*

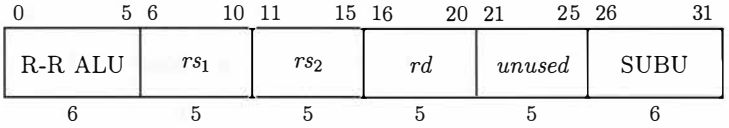
Operation:

$$rd \leftarrow_{32} (rs_1) - [(immediate_0)^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is sign-extended and arithmetically subtracted from the contents of GPR rs_1 to form a 32-bit two's complement result, which is then placed into GPR *rd*. An overflow exception occurs when the result of the subtraction operation is less than -2^{31} (i.e., < 0x80000000).

Integer Subtract Unsigned SUBU



Type: R

Format:

SUBU *rd*, rs_1, rs_2

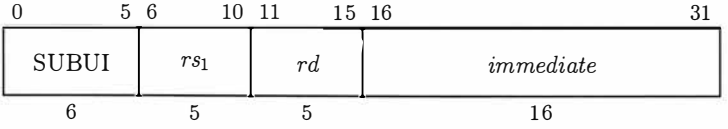
Operation:

$$rd \leftarrow_{32} (rs_1) - (rs_2)$$

Description:

The contents of GPR rs_2 are arithmetically subtracted from the contents of GPR rs_1 to form a 32-bit unsigned result, which is then placed into GPR *rd*. No overflow exception occurs under any circumstances. As a result, this is the only difference between this instruction and the SUB instruction.

Integer Subtract Unsigned Immediate SUBUI



Type: I

Format:

SUBUI *rd*, rs_1, *immediate*

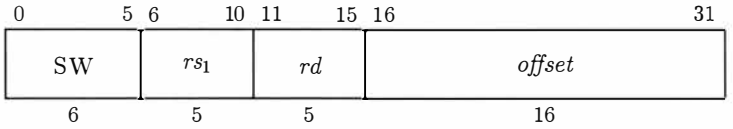
Operation:

$$rd \leftarrow_{32} (rs_1) - [\text{`0'}^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is zero-extended and arithmetically subtracted from the contents of GPR rs_1 to form a 32-bit unsigned result, which is then placed into GPR *rd*. No overflow exception occurs under any circumstances. As a result, this is the only difference between this instruction and the SUBI instruction.

Store Word **SW**



Type: I

Format:

SW $offset(rs_1)$, rd

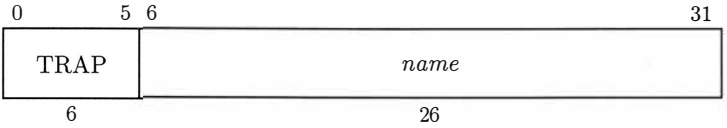
Operation:

$$\mathrm{M}\{[(offset_0)^{16} \parallel offset] + (rs_1)\} \leftarrow_{32} (rd)$$

Description:

The 16-bit *offset* is sign-extended and added to the contents of GPR rs_1 to form a 32-bit unsigned effective address. The contents of GPR rd are stored at this effective address.

Trap **TRAP**



Type: J

Format:

TRAP *name*
where $0 \leq name < (2^{26} - 1)$

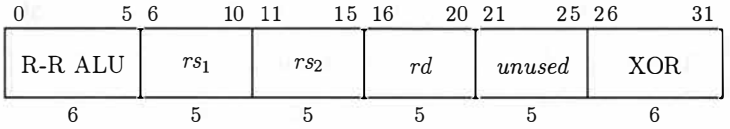
Operation:

$$IAR \leftarrow_{32} [(PC) + 8]$$
$$PC \leftarrow_{32} \text{`0'}^6 \parallel name$$

Description:

The 26-bit *name* is sign-extended and added to the address of the instruction in the delay slot to form a 32-bit target address. This target address is unconditionally placed into the program counter. The address of the instruction after the delay slot is placed into the special register *IAR*.

Logical Xor — XOR



Type: R

Format:

XOR rd, rs_1, rs_2

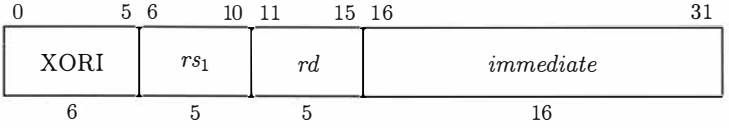
Operation:

$$rd \leftarrow_{32} (rs_1) \oplus (rs_2)$$

Description:

The contents of GPR rs_1 are combined with the contents of GPR rs_2 in a bitwise logical XOR operation, and the result is placed into GPR rd.

Logical Xor Immediate (Signed) XORI



Type: I

Format:

XORI *rd*, rs_1, *immediate*

Operation:

$$rd \leftarrow_{32} (rs_1) \oplus [\text{`0'}^{16} \parallel immediate]$$

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of GPR rs_1 in a bitwise logical XOR operation, and the result is placed into GPR *rd*.

Appendix A

DLX ISA Quick Reference

Description	Mnemonic	Operand(s)
load		
byte	LB	rd, $offset(rs_1)$
byte unsigned	LBU	
halfword	LH	
halfword unsigned	LHU	
word	LW	
SP floating-point	LF	
DP floating-point	LD	
store		
byte	SB	$offset(rs_1)$, rd
halfword	SH	
word	SW	
SP floating-point	SF	
DP floating-point	SD	
move		
GPR to special register	MOVI2S	rd, rs_1
special register to GPR	MOVS2I	
single FPR to single FPR	MOVF	
double FPR to double FPR	MOVD	
single FPR to GPR	MOVFP2I	
GPR to single FPR	MOVI2FP	

Table A.1: Load, Store, and Move Instructions

Description	Mnemonic	Operand(s)
integer arithmetic		
add (signed)	ADD	*rd*, rs_1, rs_2
add unsigned	ADDU	
subtract (signed)	SUB	
subtract unsigned	SUBU	
multiply (signed)	MULT	
multiply unsigned	MULTU	
divide (signed)	DIV	
divide unsigned	DIVU	
add immediate (signed)	ADDI	*rd*, rs_1, *immediate*
add unsigned immediate	ADDUI	
subtract immediate (signed)	SUBI	
subtract unsigned immediate	SUBUI	
logical		
and	AND	*rd*, rs_1, rs_2
or	OR	
xor	XOR	
and immediate	ANDI	*rd*, rs_1, *immediate*
or immediate	ORI	
xor immediate	XORI	
load high immediate	LHI	*rd*, *immediate*

Table A.2: Arithmetic and Logical Instructions

Description	Mnemonic	Operand(s)
shift		
left logical	SLL	*rd*, rs_1, rs_2
right logical	SRL	
right arithmetic	SRA	
left logical immediate	SLLI	*rd*, rs_1, *immediate*
right logical immediate	SRLI	
right arithmetic immediate	SRAI	
set-on-comparison		
less than	SLT	*rd*, rs_1, rs_2
greater than	SGT	
less than or equal to	SLE	
greater than or equal to	SGE	
equal to	SEQ	
not equal to	SNE	
less than immediate	SLTI	*rd*, rs_1, *immediate*
greater than immediate	SGTI	
less than or equal to imm.	SLEI	
greater than or equal to imm.	SGEI	
equal to immediate	SEQI	
not equal to immediate	SNEI	

Table A.3: Shift and Set-On-Comparison Instructions

Description	Mnemonic	Operand(s)
floating-point arithmetic		
add SP floating-point	ADDF	rd, rs_1, rs_2
add DP floating-point	ADDD	
subtract SP floating-point	SUBF	
subtract DP floating-point	SUBD	
multiply SP floating-point	MULTF	
multiply DP floating-point	MULTD	
divide SP floating-point	DIVF	
divide DP floating-point	DIVD	
convert		
SP to DP floating-point	CVTF2D	rd, rs_1
SP floating-point to integer	CVTF2I	
DP to SP floating-point	CVTD2F	
DP floating-point to integer	CVTD2I	
integer to SP floating-point	CVTI2F	
integer to DP floating-point	CVTI2D	
set-on-comparison		
less than SP floating-point	LTF	rs_1, rs_2
less than DP floating-point	LTD	
greater than SP floating-point	GTF	
greater than DP floating-point	GTD	
less than or equal to SPFP	LEF	
less than or equal to DPFP	LED	
greater than or equal to SPFP	GEF	
greater than or equal to DPFP	GED	
equal to SP floating-point	EQF	
equal to DP floating-point	EQD	
not equal to SP floating-point	NEF	
not equal to DP floating-point	NED	

Table A.4: Floating-Point Instructions

Description	Mnemonic	Operand(s)
jump		
jump	J	*name*
jump and link	JAL	
jump register	JR	rs_1
jump and link register	JALR	
branch		
on GPR equal to zero	BEQZ	rs_1, *name*
on GPR not equal to zero	BNEZ	
on FP status register true	BFPT	*name*
on FP status register false	BFPF	
special		
trap	TRAP	*name*
return from exception	RFE	
no operation	NOP	

Table A.5: Jump, Branch, and Special Instructions

Appendix B

DLXsim—A Simulator for DLX

by Larry B. Hostetler and Brian Mirtich

B.1 Introduction

Our project involved writing a simulator (DLXsim) for the DLX instruction set. DLXsim is an interactive program that loads DLX assembly programs and simulates the operation of a DLX computer on those programs, allowing both single-stepping and continuous execution through the DLX code. DLXsim also provides the user with commands to set breakpoints, view and modify memory and registers, and print statistics on the execution of the program allowing the user to collect various information on the run-time properties of a program. We expect that a major use for this tool will be in association with classes to aid in the understanding of this instruction set.

A complete overview of the interface provided by the simulator can be found in the user manual for DLXsim, which has been included after this section. Later in this appendix, a few sample runs of the simulator will also be given.

We decided that since the MIPS instruction set has many similarities with DLX, and a good MIPS simulator (available from Ousterhaut) already exists, it would be a better use of our time to modify that simulator to handle the DLX description. This simulator was built on top of the Tcl interface, providing a programming type environment for the user as well.

The main problem we encountered when rewriting the simulator was

that there are a couple of fundamental differences between the DLX and MIPS architectures. Following is a list of the main differences we identified between the two architectures.

- In MIPS, branch and jump offsets are stored as the number of words, where DLX stores the number of bytes. This has the effect of allowing jumps on MIPS to go four times as far.
- MIPS jumps have a non-obvious approach to determining the destination address: the bits in the offset part of the instruction simply replace the lower bits in the program counter. DLX chooses a more conventional approach in that the offset is sign-extended, and then added to the program counter.
- In the MIPS architecture, conditional branches are based on the result of a comparison between any two registers. DLX has only two main conditional branch operations that branch on whether a register is zero or non-zero.
- DLX provides load interlocks, while the MIPS 2000 does not.
- MIPS 2000 provides instructions for unaligned accesses to memory, while DLX does not.
- The result of a MIPS multiply or divide ends up in two special registers (HI and LO) allowing 64-bit results; the result of a DLX multiply is placed in the chosen general-purpose register, and must therefore fit into 32 bits.

Because of the large number of similarities between DLX and MIPS, we based our opcodes on those used by the MIPS machine (where MIPS had equivalent instructions). Where DLX had instructions with no MIPS equivalent, we grouped such similar DLX instructions and assigned to them blocks of unused opcodes. Below, you will find the opcode numbers used for the DLX instructions. Register-register instructions have the **special** opcode, and the instruction is specified in the lower six bits of the instruction word. Similarly, floating-point instructions have the **fparith** opcode, and the actual instruction is again found in the lower six bits of the word.

Main opcodes

	$00	$01	$02	$03	$04	$05	$06	$07
$00	SPECIAL	FPARITH	J	JAL	BEQZ	BNEZ	BFPT	BFPF
$08	ADDI	ADDUI	SUBI	SUBUI	ANDI	ORI	XORI	LHI
$10	RFE	TRAP	JR	JALR	SLLI		SRLI	SRAI
$18	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI		
$20	LB	LH		LW	LBU	LHU	LF	LD
$28	SB	SH		SW			SF	SD

Special opcodes (Main opcode = $00)

	$00	$01	$02	$03	$04	$05	$06	$07
$00					SLL		SRL	SRA
$08					TRAP			
$10								
$18								
$20	ADD	ADDU	SUB	SUBU	AND	OR	XOR	
$28	SEQ	SNE	SLT	SGT	SLE	SGE		
$30	MOVI2S	MOVS2I	MOVF	MOVD	MOVFP2I	MOVI2FP		

Floating-Point opcodes (Main opcode = $01)

	$00	$01	$02	$03	$04	$05	$06	$07
$00	ADDF	SUBF	MULTF	DIVF	ADDD	SUBD	MULTD	DIVD
$08	CVTF2D	CVTF2I	CVTD2F	CVTD2I	CVTI2F	CVTI2D	MULT	DIV
$10	EQF	NEF	LTF	GTF	LEF	GEF	MULTU	DIVU
$18	EQD	NED	LTD	GTD	LED	GED		

The manual entry for DLXsim follows.

B.2 DLXsim Manual

NAME

DLXsim—Simulator and debugger for DLX assembly programs

SYNOPSIS

`dlxsim`

OPTIONS

[-al#] [-au#] [-dl#] [-du#] [-ml#] [-mu#]

-al# Select the latency for a floating-point add (in clocks).

-au# Select the number of floating-point add units.

-dl# Select the latency for a floating-point divide.

-du# Select the number of floating-point divide units.

-ml# Select the latency for a floating-point multiply.

-mu# Select the number of floating-point multiply units.

DESCRIPTION

DLXsim is an interactive program that loads DLX assembly programs and simulates the operation of a DLX computer on those programs. When DLXsim starts up, it looks for a file named `.dlxsim` in the user's home directory. If such a file exists, DLXsim reads it and processes it as a command file. DLXsim also checks for a `.dlxsim` file in the current directory, and executes the commands in it if the file exists. Finally, DLXsim loops, forever reading commands from standard input and printing results on standard output.

NUMBERS

Whenever DLXsim reads a number, it will accept the number in either decimal notation, hexadecimal notation if the first two characters of the number are **0x** (e.g., 0x3acf), or octal notation if the first character is **0** (e.g., 0342). Two DLXsim commands accept only floating-point numbers from the user; these are **fget** and **fput** and will be described later.

ADDRESS EXPRESSIONS

Many of DLXsim's commands take as input an expression identifying a register or memory location. Such values are indicated with the term *address* in the command descriptions below. Where register names are acceptable, any of the names **r0** through **r31** and **f0** through **f31** may be used. The names **$0** through **$31** may also be used (instead of **r0** through **r31**), but the dollar signs are likely to cause confusion with Tcl variables, so it is safer to use **r** instead of **$**. The name **pc** may be used to refer to the program counter.

Symbolic expressions may be used to specify memory addresses. The simplest form of such an expression is a number, which is interpreted as a memory address. More generally, address expressions may consist of numbers, symbols (which must be defined in the assembly files currently loaded), the operators *, /, %, +, −, <<, >>, &, |, and ↑ (which have the same meanings and precedences as in C), and parentheses for grouping.

COMMANDS

In addition to all of the built-in Tcl commands, DLXsim provides the following application-specific commands:

asm *instruction* [*address*]

Treats *instruction* as an assembly instruction and returns a hexadecimal value equivalent to *instruction*. Some instructions, such as relative branches, will be assembled differently depending on where in memory the instruction will be stored. The *address* argument may be used to indicate where the instruction would be stored; if omitted, it defaults to 0.

fget *address* [*flags*]

Return the values of one or more memory locations or registers. *Address* identifies a memory location or register, and *flags*, if present, consists of a number and/or set of letters, all concatenated together. If the number is present, it indicates how many consecutive values to print (the default is 1). If flag characters are present, they have the following interpretation:

d Print values as double-precision floating-point numbers.

f Print values as single-precision floating-point numbers (default).

fput *address number* [*precision*]

Store *number* in the register or memory location given by *address*. If *precision* is **d**, the number is stored as a double-precision floating-point number (in two words). If *precision* is **f** or no *precision* is given, the number is stored as a single-precision floating-point number.

get *address* [*flags*]

Similar to **fget** above, this command is for all types except floating-point. If flag characters are present, they have the following interpretation:

B Print values in binary.

b When printing memory locations, treat each byte as a separate value.

c Print values as ASCII characters.

d Print values in decimal.

h When printing memory locations, treat each halfword as a separate value.

i Print values as instructions in the DLX assembly language.

s Print values as null-terminated ASCII strings.

v Instead of printing the value of the memory location referred to by *address*, print the address itself as the value.

w When printing memory locations, treat each word as a separate value.

x Print values in hexadecimal (default).

To interpret numbers as single- or double-precision floating-point, use the **fget** command.

go [*address*]

Start simulating the DLX machine. If *address* is given, execution starts at that memory address. Otherwise,

it continues from wherever it left off previously. This command does not complete until simulated execution stops. The return value is an information string about why execution stopped and the current state of the machine.

load *file file file ...*
Read each of the given *files*. Treat them as DLX assembly language files and load memory as indicated in the files. Code (text) is normally loaded starting at address 0x100, but the **codeStart** variable may be used to set a different starting address. Data is normally loaded starting at address 0x1000, but a different starting address may be specified in the **dataStart** variable. The return value is either an empty string or an error message describing problems in reading the files. A list of directives that the loader understands is in a later section of this manual.

put *address number*
Store *number* in the register or memory location given by *address*. The return value is an empty string. To store floating-point numbers (single or double precision), use the **fput** command.

quit Exit the simulator.

stats [**reset**] [**stalls**] [**opcount**] [**pending**] [**branch**] [**hw**] [**all**]
This command will dump various statistics collected by the simulator on the DLX code that has been run so far. Any combination of options may be selected. The options and their results are as follows:

reset Reset all of the statistics.

stalls Show the number of load stalls and stalls while waiting for a floating-point unit to become available or for the result of a previous operation to become available.

opcount Show the number of each operation that has been executed.

pending Show all floating-point operations current-

ly being handled by the floating-point units as well as what their results will be and where they will be stored.

branch Show the percentage of branches taken and not taken.

hw Show the current hardware setup for the simulated machine.

all Equivalent to choosing all options except **reset**. This is the default.

step [*address*]

If no *address* is given, the **step** command executes a single instruction, continuing from wherever execution previously stopped. If *address* is given, then the program counter is changed to point to *address*, and a single instruction is executed from there. In either case, the return value is an information string about the state of the machine after the single instruction has been executed.

stop [*option args*]

This command may take any of the forms described below:

stop Arrange for execution of DLX code to stop as soon as possible. If a simulation isn't in progress, then this command has no effect. This command is most often used in the *command* argument for the **stop at** command. Returns an empty string.

stop at *address* [*command*]

Arrange for *command* (a DLXsim command string) to be executed whenever the memory address identified by *address* is read, written, or executed. If *command* is not given, it defaults to **stop**, so that execution stops whenever *address* is accessed. A stop applies to the entire word containing *address*: the stop will be triggered whenever any byte of the word is accessed. Stops are not processed during the

step commands or the first instruction executed in a **go** command. Returns an empty string.

stop info
Return information about all stops currently set.

stop delete *number number number ...*
Delete each of the stops identified by the *number* arguments. Each *number* should be an identifying number for a stop, as printed by **stop info**. Returns an empty string.

trace [**on** *file*] [**off**]
This command toggles the writing of memory access information into a file for use with the dinero utility. The options and their results are as follows:

on *file* Start writing dinero trace information in the named *file*. If *file* already exists, the information will be appended to it. Reset all of the statistics.

off Stop writing dinero trace information.

ASSEMBLY FILE FORMAT

The assembler built into DLXsim, invoked using the **load** command, accepts standard format DLX assembly language programs. The file is expected to contain lines of the following form:

- Labels are defined by a group of non-blank characters starting with either a letter, an underscore, or a dollar sign, and followed immediately by a colon. They are associated with the address immediately following the last block of information stored. This has the bad effect that if you have code following a label following a block of data that does not end on a word boundary (multiple of 4), the label will not point to the first instruction in the code, but instead to 1 to 3 bytes before (since the address is only rounded when it is necessary to correctly align data). This is done so that if a label is found in the middle of a

data section, it will point to the start of the next section of data without the data having to be aligned to a word boundary. The work-around for this is to use the **.align** (see below) directive before labels that will not be aligned with the data following them. Labels can be accessed anywhere else within that file, and in files loaded after that if the label is declared as **.global** (see below).

- Comments are started with a semicolon and continue to the end of the line.
- Constants can be entered either with or without a preceding number sign.
- The format of instructions and their operands are as shown in Hennessy & Patterson [2].

While the assembler is processing an assembly file, the data and instructions it assembles are placed in memory based on either a text (code) or data pointer. Which pointer is used is selected not by the type of information, but by whether the most recent directive was **.data** or **.text**. The program initially loads into the text segment.

The assembler supports several directives that affect how it loads the DLX's memory. These should be entered in the place where you would normally place the instruction and its arguments. The directives currently supported by DLXsim are

.align n
: Cause the next data/code loaded to be at the next higher address with the lower n bits zeroed (the next closest address greater than or equal to the current address that is a multiple of 2^n).

.ascii "*string1*", "*string2*", ...
: Store the *strings* listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.

.asciiz "*string1*", "*string2*", ...
: Similar to **.ascii**, except each string is followed by a 0 byte (like C strings).

.byte "*byte1*", "*byte2*", ...
Store the *bytes* listed on the line sequentially in memory.

.data [*address*]
Cause the following code and data to be stored in the data area. If an *address* was supplied, the data will be loaded starting at that address, otherwise the last value for the data pointer will be used. If we were just reading code based on the text (code) pointer, store that address so that we can continue from there later (on a **.text** directive).

.double *number1*, *number2*, ...
Store the *numbers* listed on the line sequentially in memory as double-precision floating-point numbers.

.float *number1*, *number2*, ...
Store the *numbers* listed on the line sequentially in memory as single-precision floating-point numbers.

.global *label*
Make the *label* available for reference by code found in files loaded after this file.

.space *size*
Move the current storage pointer forward *size* bytes (to leave some empty space in memory).

.text [*address*]
Cause the following code and data to be stored in the text (code) area. If an *address* was supplied, the data will be loaded starting at that address, otherwise the last value for the text pointer will be used. If we were just reading data based on the data pointer, store that address so that we can continue from there later (on a **.data** directive).

.word *word1*, *word2*, ...
Store the *words* listed on the line sequentially in memory.

C LIBRARY FUNCTIONS

DLXsim allows the user access to a few simple C library functions through the use of the **TRAP** operation. Currently supported functions are **open()** (trap #1), **close()** (trap #2), **read()** (trap #3), **write()** (trap #4), **printf()** (trap #5). When the appropriate trap is invoked, the first argument should be located in the word starting at the address in r14, with the following arguments (as seen in a C statement calling the function) in words above that (r14+4, r14+8, ...). The result from the function call will be placed in r1 (this means there is currently no support for library functions that return floating-point values). If a double-precision floating-point value is to be passed to a library function, it will occupy two adjacent words with the lower word containing the value of the even-valued floating-point register, and the higher word containing the value of the odd-valued floating-point register (F0 in 0(r14), F1 in 4(r14)).

A call to a C library function currently only registers as a trap instruction in the statistics gathered by the simulator, and does not affect the instructions executed or cycles counted additionally.

NOTE: Any memory accessed by a trap function needed to perform its work is **not** currently placed in the **dinero** trace file (if one is active).

VARIABLES

DLXsim uses or sets the following Tcl variables:

codeStart
: If this variable exists, it indicates where to start loading code in **load** commands.

dataStart
: If this variable exists, it indicates where to start loading data in **load** commands.

insCount
: DLXsim uses this variable to keep a running count of the total number of instructions that have been simulated so far.

prompt

If this variable exists, it should contain a DLXsim command string. DLXsim will execute the command in this string before printing each prompt, and use the result as the prompt string to print. If this variable doesn't exist, or if an error occurs in executing its contents, then the prompt "(dlxsim)" is used.

FUTURE ENHANCEMENTS

Fix the label handling in the assembler so that a label is associated with the next address used in the assembler (not necessarily the address following the last memory altering line).

Modify the trap handler to note memory accesses to the **dinero** trace file when appropriate.

B.3 Interactive Sessions with DLXsim

To illustrate some of the features of DLXsim, this section describes two interactive sessions using examples taken from Chapter 4 of Hennessy & Patterson [2]. The programs used are on pages 225 and 227. The ADDD instructions have been replaced with MULTD instructions, however, to show the effects of a slightly longer latency. Also, TRAP instructions have been added to terminate execution of the programs when simulating.

B.3.1 Sample Datafile

The examples that follow operate on arrays of numbers. A common datafile is used for input to the programs. This datafile is named **fdata.s** and is shown below:

```
        .data    0
        .global  a
a:      .double  3.14159265358979
        .global  x
x:      .double  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
        .double  17,18,19,20,21,22,23,24,25,26,27
        .global  xtop
xtop:   .double  28
```

The **.data** directive specifies that the data should be loaded in at location

0. The **.global** directive adds the specified labels to a global symbol table so that other assembly files can access them. The **.double** directive stores double-precision data to memory.

B.3.2 First Example

The first example uses the program at the bottom of page 225 (with the ADDD replaced by MULTD). The program is shown below.

```
        ld      f2,a
        add     r1,r0,xtop
loop:   ld      f0,0(r1)
                        ; load stall occurs here
        multd   f4,f0,f2
                        ; 4 FP stalls
        sd      0(r1),f4
        sub     r1,r1,#8
        bnez    r1,loop
        nop             ; branch delay slot
        trap    #0      ; terminate simulation
```

The simulator is invoked by typing **dlxsim** at the system prompt.

```
% dlxsim
```

First the datafile is loaded, using the load command:

```
(dlxsim) load fdata.s
```

Next, the program may be loaded. The program above was created with an editor and saved in the file **f1.s**. It is loaded in the same way as the datafile.

```
(dlxsim) load f1.s
```

To verify that the program has been loaded, the **get** command can be used to examine memory. The program is loaded at location 256 by default. The second parameter to **get** indicates how many words to dump. The **i** suffix tells **get** to dump the contents in instruction format (i.e., produce a disassembly).

```
(dlxsim) get 256 9i
```

```
start: ld f2,a(r0)
start+0x4: addi r1,r0,0xe0
loop: ld f0,a(r1)
loop+0x4: multd f4,f0,f2
loop+0x8: sd a(r1),f4
loop+0xc: subi r1,r1,0x8
loop+0x10: bnez r1,loop
loop+0x14: nop
loop+0x18: trap 0x0
```

To make sure that the statistics are all cleared (as they should be when DLXsim is first invoked), use the **stats** command with the relevant parameters:

(dlxsim) `stats stalls branch pending hw`

```
Memory size: 65536 bytes.

Floating Point Hardware Configuration
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 1 multiply units,     latency =  5 cycles
Load Stalls = 0
Floating Point Stalls = 0

No branch instructions executed.

Pending Floating Point Operations:
none.
```

The **hw** specifier causes the memory size and floating-point hardware information to be dumped. The **stalls** specifier causes the total load stalls and floating-point stalls to be displayed. The **branch** specifier causes the branch information (taken vs. not taken) to be displayed; in this case no branches have been executed yet. Finally, the **pending** specifier causes the pending operations in the floating-point units to be displayed (none in this case). Below, the first four instructions are executed using the **step** command:

(dlxsim) `step 256`

```
stopped after single step, pc = start+0x4: addi r1,r0,0xe0
```

```
(dlxsim) step

stopped after single step, pc = loop: ld f0,a(r1)

(dlxsim) step

stopped after single step, pc = loop+0x4: multd f4,f0,f2

(dlxsim) step

stopped after single step, pc = loop+0x8: sd a(r1),f4
```

The **stats** command can produce some more interesting results at this point.

```
(dlxsim) stats stalls pending

Load Stalls = 1
Floating Point Stalls = 0

Pending Floating Point Operations:
multiplier #1 : will complete in  4 more cycle(s)  87.964594 ==> F4:F5
```

A load stall occurred between the third and fourth instructions because of the F0 dependency. The multiply instruction has issued and is being processed in multiplier unit #1. It will complete and store the double-precision value 87.96 into F4 and F5 in four more clock cycles.

The double-precision value in F4 can be displayed using the **fget** command with a **d** specifier (for double-precision).

```
(dlxsim) fget f4 d

f4: 0.000000
```

As expected, F4 hasn't received its value yet. Executing one more instruction will change the statistics:

```
(dlxsim) step

stopped after single step, pc = loop+0xc: subi r1,r1,0x8

(dlxsim) stats stalls pending

Load Stalls = 1
Floating Point Stalls = 4

Pending Floating Point Operations:
none.
```

Since the SD instruction used the result from the multiply instruction, the multiply was completed before the SD was executed. The four floating-point stalls required for the multiply to complete were recorded as well. If F4 is examined now, its value after the writeback is displayed.

(dlxsim) `fget f4 d`

```
f4: 87.964594
```

To execute the program to completion, the **go** command can be used. When the TRAP instruction is detected, the simulation will stop.

(dlxsim) `go`

```
TRAP #0 received
```

To view the cumulative stall and branch information, the **stats** command can be used.

(dlxsim) `stats stalls branch`

```
Load Stalls = 28
Floating Point Stalls = 112

Branches:  total 28, taken 27 (96.43%), untaken 1 (3.57%)
```

The loop executed 28 times. There was a single load stall per iteration, for a total of 28 load stalls. There were 4 floating-point stalls per iteration, for a total of 112 floating-point stalls. Finally, the conditional branch at the bottom of the loop was taken 27 times, and fell through on the final time. All these statistics are reflected above.

To verify the program operated properly, the memory locations containing the original data can be examined with the **fget** command. The original data was stored in the 28 double words beginning at location 8.

(dlxsim) `fget 8 28d`

```
x: 3.141593
x+0x8: 6.283185
x+0x10: 9.424778

... etc. ...

x+0xc8: 81.681409
```

```
x+0xd0: 84.823002
xtop: 87.964594
```

As expected, the initial integer values have all been multiplied by π.

B.3.3 Second Example

The second example is from page 227 of the aforementioned text. It demonstrates the effects of unrolling loops when multiple execution units are available. The program, which is shown below, performs the same operations on the list of numbers as the previous example program.

```
start:  ld      f2,a
        add     r1,r0,xtop
loop:   ld      f0,0(r1)
        ld      f6,-8(r1)
        ld      f10,-16(r1)
        ld      f14,-24(r1)
        multd   f4,f0,f2
        multd   f8,f6,f2
        multd   f12,f10,f2
        multd   f16,f14,f2
                                ; FP stall here
        sd      0(r1),f4
        sd      -8(r1),f8
        sd      -16(r1),f12
        sub     r1,r1,#32
        bnez    r1,loop
        sd      8(r1),f16       ; branch delay slot
        trap    #0
```

To take full advantage of this unwound loop, DLXsim can be invoked with a command line argument specifying four floating-point multiply units should be included in the hardware configuration.

```
% dlxsim -mu4

(dlxsim) stats hw

Memory size: 65536 bytes.

Floating Point Hardware Configuration
```

```
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 4 multiply units,     latency =  5 cycles
```

After loading the data and program files, the **step** instruction can be used to execute the first 10 instructions. At this point, the last MULTD instruction has just issued. The **stats** command can display the stalls and pending operations.

(dlxsim) `stats stalls pending`

```
Load Stalls = 0
Floating Point Stalls = 0

Pending Floating Point Operations:
multiplier #0 : will complete in  1 more cycle(s)  87.964594 ==> F4:F5
multiplier #1 : will complete in  2 more cycle(s)  84.823002 ==> F8:F9
multiplier #2 : will complete in  3 more cycle(s)  81.681409 ==> F12:F13
multiplier #3 : will complete in  4 more cycle(s)  78.539816 ==> F16:F17
```

It is interesting to see what happens after the next instruction is executed.

(dlxsim) `step`

```
stopped after single step, pc = loop+0x24: sd 0xfff8(r1),f8
```

(dlxsim) `stats stalls pending`

```
Load Stalls = 0
Floating Point Stalls = 1

Pending Floating Point Operations:
multiplier #2 : will complete in  1 more cycle(s)  81.681409 ==> F12:F13
multiplier #3 : will complete in  2 more cycle(s)  78.539816 ==> F16:F17
```

Since the SD instruction was dependent on the first MULTD instruction, a floating-point stall occurred so the MULTD could complete. This added stall cycle also caused the second MULTD to complete. The MULTDs have "caught up" with the SDs, and no more stalls will occur on this iteration. This is the reason loop unrolling works. To run the program to completion, the **go** command can be used.

(dlxsim) `go`

```
TRAP #0 received
```

To dump all the statistics gathered, the **stats** command is used without any parameters.

(dlxsim) stats

```
Memory size: 65536 bytes.

Floating Point Hardware Configuration
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 4 multiply units,     latency =  5 cycles
Load Stalls = 0
Floating Point Stalls = 7

Branches:  total 7, taken 6 (85.71%), untaken 1 (14.29%)

Pending Floating Point Operations:
none.

                        INTEGER OPERATIONS
                        ==================

     ADD       0      ADDI       1      ADDU       0     ADDUI       0
     AND       0      ANDI       0      BEQZ       0      BFPF       0
    BFPT       0      BNEZ       7       DIV       0      DIVU       0
       J       0       JAL       0      JALR       0        JR       0
      LB       0       LBU       0        LD      29        LF       0
      LH       0       LHI       0       LHU       0        LW       0
    MOVD       0      MOVF       0   MOVFP2I       0   MOVI2FP       0
   MOVI2S       0    MOVS2I       0      MULT       0     MULTU       0
      OR       0       ORI       0       RFE       1        SB       0
      SD      28       SEQ       0      SEQI       0        SF       0
     SGE       0      SGEI       0       SGT       0      SGTI       0
      SH       0       SLE       0      SLEI       0       SLL       0
SLLI/NOP       0       SLT       0      SLTI       0       SNE       0
    SNEI       0       SRA       0      SRAI       0       SRL       0
    SRLI       0       SUB       0      SUBI       7      SUBU       0
   SUBUI       0        SW       0      TRAP       1       XOR       0
    XORI       0
Total integer operations = 74

                     FLOATING POINT OPERATIONS
                     =========================

    ADDD       0      ADDF       0    CVTD2F       0    CVTD2I       0
  CVTF2D       0    CVTF2I       0    CVTI2D       0    CVTI2F       0
    DIVD       0      DIVF       0       EQD       0       EQF       0
     GED       0       GEF       0       GTD       0       GTF       0
     LED       0       LEF       0       LTD       0       LTF       0
   MULTD      28     MULTF       0       NED       0       NEF       0
```

```
    SUBD        0       SUBF        0
Total floating point operations = 28
Total operations = 102
Total cycles = 109
```

The dynamic counts for all instructions are shown, as well as the statistics previously discussed. The number of load stalls is seven in this case, compared to 28 in the first example. This is the result of unrolling the loop four times and providing four multiply units in hardware. An estimate of the clocks per instruction (CPI) can be obtained by dividing the total cycles (109) by the total operations (102).

The two examples above give only a flavor for the types of operations that may be done in DLXsim. The possibilities are endless.

B.4 Internal Operation

Some information concerning how DLXsim operates internally may be useful to some users, particularly those who wish to modify or enhance the simulator. This section provides an overview of the simulator and a discussion of the underlying data structures used. *This information is not necessary to use* DLXsim. All of the code discussed below is contained in the file `sim.c`.

B.4.1 Instruction Tables

DLXsim contains four tables that contain information about the DLX instruction set. The first is opTable. This table contains 64 entries corresponding to the 64 possible values of the opcode field. Each entry consists of an instruction-format pair. For example, the value of opTable[5] is {OP_BNEZ, IFMT}, indicating that opcode 5 is a branch not equal to zero instruction, which uses the I-type format. Several entries in this table have OP_RES as the instruction. These entries are reserved for future extensions to the DLX instruction set.

The zero opcode indicates a different table should be used to identify the instruction. A second table called specialTable handles this case. In this table are all the register-register operations. The format is not specified explicitly for these instructions (as it was in opTable) because they are all R-type format. These instructions all contain a zero in the opcode field and a function encoding in the lower six bits of the instruction word. There is also room in this table for expansion by using entries currently containing OP_RES.

An opcode of one indicates a floating-point arithmetic operation. A third table, FParithTable, handles these instructions. As with specialTable, all instructions in this table have R-type format. The exact operation is again specified by the lower six bits of the instruction word, which are used to index into this table. Currently 32 entries contain OP_RES and are available for future expansion to the floating-point instruction set.

The final table is operationNames. This table contains a list of all the integer instruction names followed by the floating-point instruction names. Each group is arranged alphabetically. These tables are used to print out the names of the instructions when a dynamic instruction count is requested.

B.4.2 Simulator Support Functions

This subsection describes the various routines that handle simulator commands and provide support for the main simulator code. The function Sim_Create initializes a DLX processor structure and is invoked when DLXsim is first started. The memory size of the machine along with the floating-point hardware specification (i.e., unit quantities and latencies) are specified as parameters.

Two functions, statsReset and Sim_DumpStats, process the `stats` command in DLXsim. The former resets all the statistics to zero, and the latter processes requests for various statistics. The statistics currently taken during simulation are load stalls, floating- point stalls, dynamic instruction counts, and conditional branch behavior. In addition, the floating-point hardware and pending floating-point operations can also be examined. See the description of the `stats` command for more information on how to request and reset the various statistics.

The functions Sim_GoCmd and Sim_StepCmd process the simulator's go and `step` commands, respectively. See the description of these commands for more information on using them.

The functions ReadMem and WriteMem provide the interface between the simulator and the DLX memory structure. They insure that the address accessed is valid, which means it must be within the memory's range and it must be on a word boundary. Otherwise, appropriate error handling occurs.

B.4.3 Compilation of Instructions

To improve efficiency, DLXsim "compiles" the instructions as it first encounters them. To understand how this works, it is necessary to examine the structure of a single word of the DLX memory. A single memory word contains several fields: value, opCode, rs1, rs2, rd, and extra. A DLX program to be simulated is written in DLX assembly language. Such a program is automatically assembled into machine code as it is loaded. The actual machine codes are stored in the value fields of the memory words. The value field represents the number actually stored at a particular memory word. The opCode field of each memory word is initially set to the special value OP_NOT_COMPILED.

When the simulator executes an instruction, it first examines the opCode field of the memory word pointed to by the program counter. If this field is a valid opcode (specified in the tables discussed above), the appropriate action for that instruction occurs. If the opCode field contains the value OP_NOT_COMPILED, the function Compile is invoked. This function looks at the actual word stored in the value field. The bits corresponding to the opcode and function fields are examined to determine what the instruction is. Depending on the instruction type, the two source register specifiers and destination register specifier may be extracted and stored in the fields rs1, rs2, and rd. If a 16-bit immediate value is present (for I-type instructions) or a 26-bit offset is present (for J-type instructions), this value is extracted and stored in the extra field of the memory word. The special code for the instruction is stored in the opCode field of the word, which previously contained the value OP_NOT_COMPILED. These special codes are not the real DLX opcodes, but rather the pseudo-opcodes defined in the file dlx.h.

When a compiled instruction is subsequently encountered, no shifting or masking operations are required to access the register specifiers or immediate values; the required information is already present in the appropriate fields of the memory words (rs1, rs2, rd, and extra). This allows the simulator to execute much faster. The actual machine code for the instruction can still be examined through the value field, and this is the value printed when the word is examined with the `get` command.

B.4.4 Main Simulation Loop

Simulate is the main function of the simulator. The heart of this function is basically a very large switch statement, based on the opCode field of the memory word pointed to by the program counter. There is a case

for each integer and floating-point instruction. Simulate loops through the basic fetch-decode-execute cycle until a stop command is received or some other exceptional condition occurs.

Load Stalls

DLX has a latency of one cycle on load instructions. In other words, the result is not yet present in the destination register on the cycle immediately following the load instruction. To address this problem, DLX has load interlocks that cause the pipeline to stall if an instruction immediately following a load instruction reads the value in the load's destination register. DLXsim records the occurrence of these load stall cycles for statistical purposes. Several variables are set during the processing of the following load instructions: LB, LBU, LH, LHU LW, LF, and LD. LHI is not included since the value to be loaded is contained in the instruction and there is no extra latency. For the other load instructions, the destination register (or registers in the case of load double) are stored in loadReg1 and loadReg2 (if this is a load double). The corresponding values to be stored in these registers (on the next cycle) are stored in loadValue1 and loadValue2.

When an instruction that reads registers (such as an ADD instruction) is encountered during simulation, the contents of loadReg1 and loadReg2 are examined before any other action occurs. If either of the registers specified by these variables were loaded in the previous instruction, a load stall is detected and tallied. Different register fields must be checked for different instructions. All the load stall detection logic is contained in the macros at the top of the Simulate function definition.

Of interest is the fact that while load stalls would slow down the execution speed of a real DLX machine, they do not affect the performance of the simulator. This is because load stall cycles are not actually simulated. Instead, it is simply noted that a load stall occurred at a particular point, and execution proceeds normally.

Dynamic Instruction Counts

Statistics on the number of each type of instruction executed are also recorded during simulation. This is a simple operation of incrementing the appropriate element of the array operationCount, which is indexed by the pseudo-opcodes discussed above. The information in the array can be accessed by the `stats` command.

Conditional Branch Behavior

DLXsim also keeps statistics on the conditional branch behavior during program execution. There are four instructions in this category: BEQZ, BNEZ, BFPT, and BFPF. The latter two instructions are branches based on the status of the floating-point condition register. Two fields of the DLX machine structure, branchYes and branchNo, record how many conditional branches were taken and not taken, respectively. These values are accessible via the `stats` command.

B.4.5 Floating-Point Execution Control

A large portion of the DLXsim code is devoted to the floating-point side of the machine. The floating-point scheme currently implemented requires instructions to issue in order, but they may complete out of order. In addition to managing the allocation of the floating-point units, DLXsim must also handle all the hazard checking associated with out-of-order completion of instructions. By requiring instructions to issue in order, the write-after-read (WAR) hazard is avoided. The three hazards that may occur are read-after-write (RAW) hazards, write-after-write (WAW) hazards, and structural hazards.

Floating-Point Data Structures

The variables and data structures that manage the floating-point execution are all declared in the file dlx.h as part of the basic DLX structure. The variables num_add_units, num_div_units, and num_mul_units specify how many of each type of floating-point execution unit are available on the machine. The variables fp_add_latency, fp_div_latency, and fp_mul_latency specify the corresponding latencies (in clock cycles) of each of the execution units. All six of these variables have default values that may be overridden via command line parameters when DLXsim is invoked.

The variable FPstatusReg is the status register that is examined on a BFPT or BFPF instruction. The various floating-point set instructions (EQF, NED, etc.) write to this register.

The array fp_add_units contains the status of all the floating-point adders during execution. If fp_add_units[i] is zero, adder i is available. A non-zero value means that the unit is currently performing an operation—the value specifies the clock cycle when the operation will complete. The array fp_div_units and fp_mul_units contain analogous information for the floating-point dividers and multipliers. All three

structures can be accessed through the array fp_units, which is an array of pointers to the three execution unit status arrays.

The array waiting_FPRs contains 32 elements, corresponding to the 32 floating-point registers in DLX. A zero in waiting_FPRs[i] means floating-point register Fi can be read from; it contains its most current value. A non-zero value means register Fi is the destination register of a pending floating-point operation (one that has issued but not yet completed). Attempting to read or write to such a register means a hazard condition exists. The non-zero value indicates the cycle at which the writeback to the register will occur.

The variable FPopsList points to the chain of pending floating-point operations. Each item in this chain is of type FPop, a structure with the following fields:

type — Indicates the type of operation. Normally this is implied by what type of floating-point unit is executing the operation; however adders can perform both additions and subtractions.

unit — The unit number of the execution unit that is executing the operation.

dest — The destination register for the operation. For a double-precision operation, this is the lower-numbered destination register.

isDouble — Indicates if the operation is single or double precision.

result — An array of two floats used to store the result of the operation (only the first element is used for single-precision operations). The result is actually computed at the time of issue.

ready — The cycle when the operation will complete and writeback will occur.

nextPtr — Points to the next FPop in the chain of pending operations.

To maximize performance, the list of pending floating-point operations is sorted based on when the operations will complete. The operation that will complete soonest is at the head of the list.

The variable checkFP is a copy of the ready field of the first floating-point operation on the pending operation list. If its value is zero, no floating-point operations are pending. Otherwise checkFP indicates

when the next (soonest) floating-point operation will complete. This provides for very quick checking in the fast-path of the simulator. Only one value needs to be checked in a cycle when no writebacks should occur.

Many of the previously discussed structures refer to a clock cycle count when a particular operation will complete. The current clock cycle is kept in the variable cycleCount. This variable is incremented each time the simulator executes its main loop. It is also incremented an extra time when a load stall is detected since the floating-point units are still executing during a load stall. When the cycle count reaches a large value specified by the constant CYC_CNT_RESET, cycleCount is "reset" back to a small number (5), and all references to clock cycles in the floating-point data structures are adjusted accordingly. This operation is necessary to prevent cycleCount from overflowing, becoming negative, and thereby wreaking havoc on the sorted list of pending operations. Making cycleCount an unsigned integer does not work, since there are still problems with sorting the pending operations when cycle counts "wrap around" to zero.

Issuing Floating-Point Operations

The function FPissue initiates a floating-point operation. It is called from eight of the switch cases in the main loop: ADDF, DIVF, MULF, SUBF, ADDD, DIVD, MULD, and SUBD. When a floating-point instruction issues, three hazard conditions must be checked. A structural hazard occurs if a floating-point unit of the required type is not available. A RAW hazard occurs if one of the source operands is the destination of a pending floating-point operation. Finally, a WAW hazard occurs if the destination register is the destination register of a pending floating-point operation. All three conditions can be checked by examing the floating-point data structures discussed above. If any of these hazards are present (and there may be more than one), the current instruction is not issued. Instead a non-zero value is returned, which indicates the soonest cycle when one of the hazard conditions will be over. This may be a cycle when one of the floating-point units will complete its current operation (eliminating a structural hazard), or when some register will be written back (eliminating a RAW or WAW hazard). When the caller receives a non-zero value from FPissue, the appropriate number of floating-point stalls are simulated by adjusting the variables cycleCount and FPstalls. The function FPwriteBack (see below) is called to perform any writebacks that may now occur. Then FPissue is re-invoked.

If another hazard condition exists, the whole process may be repeated, but eventually all of the hazard conditions will terminate.

If no hazards are present, the instruction is issued. That is, a new FPop structure is placed in the appropriate spot in the pending operations list. The appropriate elements of waiting_FPRs are also set to indicate that the destination registers are waiting for values to be written back. FPissue returns a zero value to indicate a successful issue, and the simulation continues.

Writing Back Floating-Point Results

The function FPwriteBack is the second function involved in floating-point execution. It is called whenever cycleCount reaches checkFP, indicating that a result is ready to be written back on the current cycle. FPwriteBack does exactly that. It removes the first FPop from the list of pending operations, and stores the result (computed at time of issue) in the appropriate register(s). It also zeroes the appropriate element(s) in waiting_FPRs. Since more than one operation may complete on the same cycle, FPwriteBack repeats this process until the value in the ready field of the operation at the head of the list exceeds the current value in cycleCount.

Handling RAW and WAW Hazards

The function FPissue (discussed above) handles the RAW and WAW hazards when a new floating-point operation is issued. However, several other instructions can generate such hazards. Any instruction that reads from or writes to a floating-point register must check that the register is not the destination of a pending operation. The following instructions fall into this class:

Loads	LF and LD
Stores	SF and SD
Moves	MOVFP2I, MOVI2FP, MOVF, MOVD
Converts	CVTD2FP, CVTD2I, CVTFP2D, CVTFP2I, CVTI2D, CVTI2FP
Sets	SEQF, SNEF, SLTF, SLEF, SGTF, SGEF, SEQD, SNED, SLTD, SLED, SGTD, SGED

When any of these instruction are executed, a call to FPwait is made. This is the third and final function for handling floating-point execution. It checks that all writebacks into the appropriate registers have occurred. The number of registers that need to be checked varies. For an LF instruction, only a single register needs to be checked, while four registers must be checked on a MOVD. If any of the registers are the destinations of pending operations, FPwait will adjust cycleCount and FPstalls appropriately, and call FPwriteBack to write the results back to the registers. When FPwait returns, all RAW and WAW hazard conditions will have passed.

Bibliography

[1] Donald E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2nd edition, 1995.

[3] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[4] Larry B. Hostetler and Brian Mirtich. "DLXsim—A Simulator for DLX." Technical report, University of California at Berkeley, 1990.

[5] Peter Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Francisco, 1995.

THE **DLX** INSTRUCTION SET ARCHITECTURE HANDBOOK

PHILIP M. SAILER AND DAVID R. KAELI

The definitive source for the DLX instruction set architecture introduced in Hennessy and Patterson's *Computer Architecture: A Quantitative Approach*. DLX is a selective amalgam of several sophisticated load/store architectures; it was developed to serve as a simple example of a pure RISC architecture and is invoked throughout *Computer Architecture* to demonstrate design principles. With its complete and up-to-date information on the details of DLX, this handbook is a valuable supplement for anyone studying *Computer Architecture*, whether independently or as part of a class. It also makes an informative addition to the library of any computer systems designer or RISC aficionado.

Beginning with the origin and history of DLX, the opening section of the handbook covers the essential topics of registers, data formats, addressing, and interrupt handling. Later sections provide a general description of the instruction set architecture, followed by the specifics of DLX instruction types, format notation, and operation notation. Appendices offer a quick reference to the instruction set and the latest available version of documentation for the DLXsim simulator.

DLXsim and a VHDL model of DLX are available online.

MORGAN KAUFMANN BOOKS FEATURING DLX AS AN EXAMPLE ARCHITECTURE

Computer Architecture: A Quantitative Approach, 2nd edition, by John L. Hennessy and David A. Patterson, 1995, ISBN 1-55860-329-8.

The standard reference for computer systems analysis and design. The focus is on fundamental techniques for designing real machines, with attention to maximizing cost/performance.

The Designer's Guide to VHDL by Peter J. Ashenden, 1995, 1-55860-270-4.

A comprehensive manual for the language and an authoritative reference on its use in hardware design at all levels, from the system level to the gate level. Requiring only a minimal background in programming, this is an excellent tutorial for anyone in computer architecture, digital systems engineering, or CAD.

Computer Systems and Design
Electrical Engineering

ISBN 1-55860-371-9

9 781558 603714 90000>

MK

Morgan Kaufmann Publishers, Inc.
San Francisco, California