Appendix B

Instruction Set Principles and Examples
Computer Architecture’s Changing Definition

- 1950s to 1960s:
  Computer Architecture Course = Computer Arithmetic

- 1970s to mid 1980s:
  Computer Architecture Course = Instruction Set Design, especially ISA appropriate for compilers

- 1990s:
  Computer Architecture Course = Design of CPU, memory system, I/O system, Multiprocessors
Instruction Set Architecture (ISA)
Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
- Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)
- Separation of Programming Model from Implementation
  - High-level Language Based (B5000 1963)
  - Concept of a Family (IBM 360 1964)
- General Purpose Register Machines
  - Complex Instruction Sets (Vax, Intel 432 1977-80)
  - Load/Store Architecture (CDC 6600, Cray 1 1963-76)
  - RISC (Mips, Sparc, HP-PA, IBM RS6000, PowerPC . . . 1987)
  - LIW/”EPIC”? (IA-64 . . . 1999)
Instructions Can Be Divided into 3 Classes (I)

• Data movement instructions
  – Move data from a memory location or register to another memory location or register without changing its form
  – Load—source is memory and destination is register
  – Store—source is register and destination is memory

• Arithmetic and logic (ALU) instructions
  – Change the form of one or more operands to produce a result stored in another location
  – Add, Sub, Shift, etc.

• Branch instructions (control flow instructions)
  – Alter the normal flow of control from executing the next instruction in sequence
  – Br Loc, Brz Loc2,—unconditional or conditional branches
Classifying ISAs

Accumulator (before 1960):
1 address add A acc ← acc + mem[A]

Stack (1960s to 1970s):
0 address add tos ← tos + next

Memory-Memory (1970s to 1980s):
2 address add A, B mem[A] ← mem[A] + mem[B]
3 address add A, B, C mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present):
2 address add R1, A R1 ← R1 + mem[A]
   load R1, A R1 ← mem[A]

Register-Register (Load/Store) (1960s to present):
3 address add R1, R2, R3 R1 ← R2 + R3
   load R1, R2 R1 ← mem[R2]
   store R1, R2 mem[R1] ← R2
Classifying ISAs
Load-Store Architectures

- **Instruction set:**
  - add R1, R2, R3
  - sub R1, R2, R3
  - mul R1, R2, R3
  - load R1, R4
  - store R1, R4

- **Example: A*B - (A+C*B)**
  - load R1, &A
  - load R2, &B
  - load R3, &C
  - load R4, R1
  - load R5, R2
  - load R6, R3
  - mul R7, R6, R5 /* C*B */
  - add R8, R7, R4 /* A + C*B */
  - mul R9, R4, R5 /* A*B */
  - sub R10, R9, R8 /* A*B - (A+C*B) */
Load-Store: Pros and Cons

• Pros
  – Simple, fixed length instruction encoding
  – Instructions take similar number of cycles
  – Relatively easy to pipeline

• Cons
  – Higher instruction count
  – Not all instructions need three operands
  – Dependent on good compiler
Registers: Advantages and Disadvantages

• Advantages
  – Faster than cache (no addressing mode or tags)
  – Deterministic (no misses)
  – Can replicate (multiple read ports)
  – Short identifier (typically 3 to 8 bits)
  – Reduce memory traffic

• Disadvantages
  – Need to save and restore on procedure calls and context switch
  – Can’t take the address of a register (for pointers)
  – Fixed size (can’t store strings or structures efficiently)
  – Compiler must manage
General Register Machine and Instruction Formats

Instruction formats

- **load R8, Op1 (R8 ← Op1)**
  - Load R8 with the value at memory location Op1Addr.
  - Instruction format: `load R8 Op1Addr`

- **add R2, R4, R6 (R2 ← R4 + R6)**
  - Add the values in registers R4 and R6, store the result in R2.
  - Instruction format: `add R2 R4 R6`
General Register Machine and Instruction Formats

• It is the most common choice in today’s general-purpose computers

• *Which* register is specified by small “address” (3 to 6 bits for 8 to 64 registers)

• Load and store have one long & one short address: One and half addresses

• Arithmetic instruction has 3 “half” addresses
Real Machines Are Not So Simple

• Most real machines have a mixture of $3, 2, 1, 0,$ and $1$-address instructions

• A distinction can be made on whether arithmetic instructions use data from memory

• If ALU instructions only use registers for operands and result, machine type is **load-store**
  – Only load and store instructions reference memory

• Other machines have a mix of register-memory and memory-memory instructions
Alignment Issues

• If the architecture does not restrict memory accesses to be aligned then
  – Software is simple
  – Hardware must detect misalignment and make 2 memory accesses
  – Expensive detection logic is required
  – All references can be made slower

• Sometimes unrestricted alignment is required for backwards compatibility

• If the architecture restricts memory accesses to be aligned then
  – Software must guarantee alignment
  – Hardware detects misalignment access and traps
  – No extra time is spent when data is aligned

• Since we want to make the common case fast, having restricted alignment is often a better choice, unless compatibility is an issue
Types of Addressing Modes
(VAX)

1. Register direct  Ri
2. Immediate (literal)  #n
3. Displacement  M[Ri + #n]
4. Register indirect  M[Ri]
5. Indexed  M[Ri + Rj]
6. Direct (absolute)  M[#n]
7. Memory Indirect  M[M[Ri]]
8. Autoincrement  M[Ri++]
9. Autodecrement  M[Ri - -]
10. Scaled  M[Ri + Rj*d + #n]
Summary of Use of Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>TeX</th>
<th>spice</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory indirect</td>
<td>1%</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>16%</td>
<td>6%</td>
</tr>
<tr>
<td>Register indirect</td>
<td>3%</td>
<td>24%</td>
<td>11%</td>
</tr>
<tr>
<td>Immediate</td>
<td>17%</td>
<td>43%</td>
<td>39%</td>
</tr>
<tr>
<td>Displacement</td>
<td>32%</td>
<td>55%</td>
<td>40%</td>
</tr>
</tbody>
</table>

© 2003 Elsevier Science (USA). All rights reserved.
Distribution of Displacement Values

© 2003 Elsevier Science (USA). All rights reserved.
Frequency of Immediate Operands

- ** Loads: **
  - Floating-point average: 22%
  - Integer average: 23%

- ** ALU operations: **
  - Floating-point average: 19%
  - Integer average: 25%

- ** All instructions: **
  - Floating-point average: 16%
  - Integer average: 21%

© 2003 Elsevier Science (USA). All rights reserved.
Types of Operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control: BRANCH, JUMP, CALL
- System: OS CALL, VM
- Floating Point: ADDF, MULF, DIVF
- Decimal: ADDD, CONVERT
- String: MOVE, COMPARE
- Graphics: (DE)COMPRESS
Distribution of Data Accesses by Size

- Double word (64 bits): 70% (floating-point), 59% (integer)
- Word (32 bits): 29% (floating-point), 26% (integer)
- Half word (16 bits): 0% (floating-point), 5% (integer)
- Byte (8 bits): 1% (floating-point), 10% (integer)

© 2003 Elsevier Science (USA). All rights reserved.
80x86 Instruction Frequency
(SPECint92, Fig. B.13)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>register move</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>
Relative Frequency of Control Instructions

- **Call/return**: Floating-point average 8%, Integer average 19%
- **Jump**: Floating-point average 10%, Integer average 6%
- **Conditional branch**: Floating-point average 82%, Integer average 75%
Control instructions (cont’d)

• Addressing modes
  – PC-relative addressing (independent of program load & displacements are close by)
    • Requires displacement (how many bits?)
    • Determined via empirical study. [8-16 works!]
  – For procedure returns/indirect jumps/kernel traps, target may not be known at compile time.
    • Jump based on contents of register
    • Useful for switch/(virtual) functions/function ptrs/dynamically linked libraries etc.
Branch Distances (in terms of number of instructions)
Frequency of Different Types of Compares in Conditional Branches

- Not equal: 2% (Floating-point), 5% (Integer)
- Equal: 0% (Floating-point), 16% (Integer)
- Greater than or equal: 0% (Floating-point), 11% (Integer)
- Greater than: 0% (Both)
- Less than or equal: 0% (Floating-point), 33% (Integer)
- Less than: 34% (Floating-point), 35% (Integer)

Frequency of comparison types in branches

© 2003 Elsevier Science (USA). All rights reserved.
Encoding an Instruction set

• a desire to have as many registers and addressing mode as possible
• the impact of size of register and addressing mode fields on the average instruction size and hence on the average program size
• a desire to have instruction encode into lengths that will be easy to handle in the implementation
Three choice for encoding the instruction set

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Variable (e.g., VAX, Intel 80x86)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2003 Elsevier Science (USA). All rights reserved.
Compilers and ISA

• Compiler Goals
  – All correct programs compile correctly
  – Most compiled programs execute quickly
  – Most programs compile quickly
  – Achieve small code size
  – Provide debugging support

• Multiple Source Compilers
  – Same compiler can compile different languages

• Multiple Target Compilers
  – Same compiler can generate code for different machines
Compilers Phases

**Dependencies**
- Language dependent; machine independent
- Somewhat language dependent; largely machine independent
- Small language dependencies; machine dependencies slight (e.g., register counts/types)
- Highly machine dependent; language independent

**Function**
- Front end per language: Transform language to common intermediate form
- High-level optimizations: For example, loop transformations and procedure inlining (also called procedure integration)
- Global optimizer: Including global and local optimizations + register allocation
- Code generator: Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

© 2003 Elsevier Science (USA). All rights reserved.
Compiler Based Register Optimization

• Assume small number of registers (16-32)
• Optimizing use is up to compiler
• HLL programs have no explicit references to registers
  – usually – is this always true?
• Assign symbolic or virtual register to each candidate variable
• Map (unlimited) symbolic registers to real registers
• Symbolic registers that do not overlap can share real registers
• If you run out of real registers some variables use memory
• Uses graph coloring approach
Designing ISA to Improve Compilation

- Provide enough general purpose registers to ease register allocation (more than 16).
- Provide regular instruction sets by keeping the operations, data types, and addressing modes orthogonal.
- Provide primitive constructs rather than trying to map to a high-level language.
- Simplify trade-off among alternatives.
- Allow compilers to help make the common case fast.
ISA Metrics

• Orthogonality
  – No special registers, few special cases, all operand modes available with any data type or instruction type

• Completeness
  – Support for a wide range of operations and target applications

• Regularity
  – No overloading for the meanings of instruction fields

• Streamlined Design
  – Resource needs easily determined. Simplify tradeoffs.

• Ease of compilation (programming?), Ease of implementation, Scalability
MIPS Processor
MIPS Registers

- **Main Processor (integer manipulations):**
  - 32 64-bit general purpose registers – GPRs (R₀ – R₃₁);
    - R₀ has fixed value of zero. Attempt to writing into R₀ is not illegal, but its value will not change;
  - two 64-bit registers – Hi & Lo, hold results of integer multiply and divide
  - 64-bit program counter – PC;
- **Coprocessor 1 (Floating Point Processor — real numbers manipulations):**
  - 32 64-bit floating point registers – FPRs (f₀ – f₃₁);
  - five control registers;
- **Coprocessor 0 – CP0** is incorporated on the MIPS CPU chip and it provides functions necessary to support operating system: exception handling, memory management scheduling and control of critical resources.
MIPS Registers (continued)

- Coprocessor 0 (CP0) registers (partial list):
  - Status register (CP0reg12) – processor status and control;
  - Cause register (CP0reg13) – cause of the most recent exception;
  - EPC register (CP0reg14) – program counter at the last exception;
  - BadVAddr register (CP0reg08) – the address for the most recent address related exception;
  - Count register (CP0reg09) – acts as a timer, incrementing at a constant rate that is a function of the pipeline clock;
  - Compare register (CP0reg11) – used in conjunction with Count register;
  - Performance Counter register (CP0reg25);
MIPS Data Types

• MIPS64 operates on:
  – 64-bit (unsigned or 2’s complement) integers,
  – 32-bit (single precision floating point) real numbers,
  – 64-bit (double precision floating point) real numbers;

• 8-bit bytes, 16-bit half words and 32-bit words loaded into GPRs are either zero or sign bit expanded to fill the 64 bits.

• only 32- or 64-bit real numbers can be loaded into FPRs.
• 32-bit real number loaded into FPRs is zero-appended.
MIPS Addressing Modes

- register addressing;
- immediate addressing;
- register indexed is the only memory data addressing; (in MIPS terminology called base addressing):
  - memory address = register content plus 16-bit offset
- since R₀ always contains value 0:
  - R₀ + 16–bit offset → absolute addressing;
  - 16-bit offset = 0 → register indirect;
- branch instructions use PC relative addressing:
  - branch address = [PC] + 4 + 4×16-bit offset
- jump instructions use:
  - pseudo-direct addressing with 28-bit addresses (jumps inside 256MB regions),
  - direct (absolute) addressing with 64-bit addresses.
Instruction Layout for MIPS

I-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt → rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>

Jump and jump and link
Trap and return from exception
MIPS Alignment

- **MIPS supports byte addressability:**
  - it means that a byte is the smallest unit with its own address;
- **MIPS restricts memory accesses to be aligned as follows:**
  - 64-bit word has to start at byte address which is multiple of 8; thus, 64-bit word at address $8x$ includes eight bytes with addresses $8x$, $8x+1$, $8x+2$, … $8x+6$, $8x+7$.
  - 32-bit word has to start at byte address that is multiple of 4; thus, 32-bit word at address $4n$ includes four bytes with addresses: $4n$, $4n+1$, $4n+2$, and $4n+3$.
  - 16-bit half word has to start at byte address that is multiple of 2; thus, 16-bit word at address $2n$ includes two bytes with addresses: $2n$ and $2n+1$.
- **MIPS supports 64-bit addresses:**
  - it means that an address is given as 64-bit unsigned integer;
MIPS Instruction

- Instructions that move data:
  - load to register from memory (only base addressing),
  - store from register to memory (only base addressing),
  - move between registers in same and different coprocessors.

- ALU integer instructions; register – register and register-immediate computational instructions.

- Floating point instructions; register – register computational instructions and floating point to/from integer conversions.

- Control-related instruction:
  - (simple) branch instructions use PC relative addressing
  - jump instructions with 28-bit addresses (jumps inside 256MB regions), or absolute 64-bit addresses.

- Special control-related instructions.
## Load/Store Instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, 30(R2)</td>
<td>Load double word</td>
<td>Regs[R1] ← 64 Mem[30 + Regs[R2]]</td>
</tr>
<tr>
<td>LD R1, 1000(R0)</td>
<td>Load double word</td>
<td>Regs[R1] ← 64 Mem[1000 + 0]</td>
</tr>
<tr>
<td>LW R1, 60(R2)</td>
<td>Load word</td>
<td>Regs[R1] ← 64 (Mem[60 + Regs[R2]])&lt;sup&gt;32&lt;/sup&gt; # Mem[60 + Regs[R2]]</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
<td>Regs[R1] ← 64 (Mem[40 + Regs[R3]])&lt;sup&gt;56&lt;/sup&gt; # Mem[40 + Regs[R3]]</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
<td>Regs[R1] ← 64&lt;sup&gt;56&lt;/sup&gt; # Mem[40 + Regs[R3]]</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load half word</td>
<td>Regs[R1] ← 64 (Mem[40 + Regs[R3]])&lt;sup&gt;48&lt;/sup&gt; # Mem[40 + Regs[R3]]</td>
</tr>
<tr>
<td>LS F0, 50(R3)</td>
<td>Load FP single</td>
<td>Regs[F0] ← 64 Mem[50 + Regs[R3]] # 0&lt;sup&gt;32&lt;/sup&gt;</td>
</tr>
<tr>
<td>LD F0, 50(R2)</td>
<td>Load FP double</td>
<td>Regs[F0] ← 64 Mem[50 + Regs[R2]]</td>
</tr>
<tr>
<td>SD R3, 500(R4)</td>
<td>Store double word</td>
<td>Mem[500 + Regs[R4]] ← 64 Regs[R3]</td>
</tr>
<tr>
<td>SW R3, 500(R4)</td>
<td>Store word</td>
<td>Mcm[500 + Regs[R4]] ← 32 Regs[R3]</td>
</tr>
<tr>
<td>SS F0, 40(R3)</td>
<td>Store FP single</td>
<td>Mem[40 + Regs[R3]] ← 32 Regs[F0]&lt;sup&gt;0..31&lt;/sup&gt;</td>
</tr>
<tr>
<td>SD F0, 40(R3)</td>
<td>Store FP double</td>
<td>Mem[40 + Regs[R3]] ← 64 Regs[F0]</td>
</tr>
<tr>
<td>SH R3, 502(R2)</td>
<td>Store half</td>
<td>Mem[502 + Regs[R2]] ← 16 Regs[R3]&lt;sup&gt;48..63&lt;/sup&gt;</td>
</tr>
<tr>
<td>SB R2, 41(R3)</td>
<td>Store byte</td>
<td>Mem[41 + Regs[R3]] ← 9 Regs[R2]&lt;sup&gt;66..63&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

**Figure B.23** The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.
Sample ALU Instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU R1,R2,R3</td>
<td>Add unsigned</td>
<td>Regs[R1] ← Regs[R2] + Regs[R3]</td>
</tr>
<tr>
<td>DADDIU R1,R2,#3</td>
<td>Add immediate unsigned</td>
<td>Regs[R1] ← Regs[R2] + 3</td>
</tr>
<tr>
<td>LUI R1,#42</td>
<td>Load upper immediate</td>
<td>Regs[R1] ← 0^{32}##42##0^{16}</td>
</tr>
<tr>
<td>DSLL R1,R2,#5</td>
<td>Shift left logical</td>
<td>Regs[R1] ← Regs[R2] &lt;&lt; 5</td>
</tr>
</tbody>
</table>
| DSLT R1,R2,R3       | Set less than            | if (Regs[R2] < Regs[R3])
                                       | Regs[R1] ← 1 else Regs[R1] ← 0              |

Figure B.24 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.
Control Flow Instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>J name</td>
<td>Jump</td>
<td>$PC_{36..63} \leftarrow name$</td>
</tr>
<tr>
<td>JAL name</td>
<td>Jump and link</td>
<td>$Regs[R31] \leftarrow PC+4; \quad PC_{36..63} \leftarrow name; \quad ((PC+4)-2^{27}) \leq name &lt; ((PC+4)+2^{27})$</td>
</tr>
<tr>
<td>JALR R2</td>
<td>Jump and link register</td>
<td>$Regs[R31] \leftarrow PC+4; \quad PC \leftarrow Regs[R2]$</td>
</tr>
<tr>
<td>JR R3</td>
<td>Jump register</td>
<td>$PC \leftarrow Regs[R3]$</td>
</tr>
<tr>
<td>BEQZ R4,name</td>
<td>Branch equal zero</td>
<td>if $(Regs[R4]==0) \quad PC \leftarrow name; \quad ((PC+4)-2^{17}) \leq name &lt; ((PC+4)+2^{17})$</td>
</tr>
<tr>
<td>BNE R3,R4,name</td>
<td>Branch not equal zero</td>
<td>if $(Regs[R3]!=Regs[R4]) \quad PC \leftarrow name; \quad ((PC+4)-2^{17}) &lt; name &lt; ((PC+4)+2^{17})$</td>
</tr>
<tr>
<td>MOVZ R1,R2,R3</td>
<td>Conditional move</td>
<td>if $(Regs[R3]==0) \quad Regs[R1] \leftarrow Regs[R2]$</td>
</tr>
</tbody>
</table>

**Figure B.25** Typical control flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggest; since MIPS instructions are all 32 bits long, the byte branch address is multiplied by 4 to get a longer distance.
<table>
<thead>
<tr>
<th>Instruction type/opcode</th>
<th>Instruction meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data transfers</strong></td>
<td>Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR</td>
</tr>
<tr>
<td>LB, LBU, SB</td>
<td>Load byte, load byte unsigned, store byte (to/from integer registers)</td>
</tr>
<tr>
<td>LH, LHU, SH</td>
<td>Load half word, load half word unsigned, store half word (to/from integer registers)</td>
</tr>
<tr>
<td>LW, LWU, SW</td>
<td>Load word, load word unsigned, store word (to/from integer registers)</td>
</tr>
<tr>
<td>LD, SD</td>
<td>Load double word, store double word (to/from integer registers)</td>
</tr>
<tr>
<td>L.S, L.D, S.S, S.D</td>
<td>Load SP float, load DP float, store SP float, store DP float</td>
</tr>
<tr>
<td>MFC0, MTC0</td>
<td>Copy from/to GPR to/from a special register</td>
</tr>
<tr>
<td>MOV.S, MOV.D</td>
<td>Copy one SP or DP FP register to another FP register</td>
</tr>
<tr>
<td>MFC1, MTC1</td>
<td>Copy 32 bits from/to FP registers to/from integer registers</td>
</tr>
<tr>
<td><strong>Arithmetic/logical</strong></td>
<td>Operations on integer or logical data in GPRs; signed arithmetic trap on overflow</td>
</tr>
<tr>
<td>ADD, ADDI, DADDU, DADDIU</td>
<td>Add, add immediate (all immediates are 16 bits); signed and unsigned</td>
</tr>
<tr>
<td>DSUB, DSUBU</td>
<td>Subtract; signed and unsigned</td>
</tr>
<tr>
<td>DMUL, DMULU, DIV, DIVU, MADD</td>
<td>Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values</td>
</tr>
<tr>
<td>AND, ANDI</td>
<td>And, and immediate</td>
</tr>
<tr>
<td>OR, ORI, XOR, XORI</td>
<td>Or, or immediate, exclusive or, exclusive or immediate</td>
</tr>
<tr>
<td>LUI</td>
<td>Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends</td>
</tr>
<tr>
<td>DSLL, DSRL, DSRA, DSSLV, DSRLV, DSRAV</td>
<td>Shifts: both immediate (DS__) and variable form (DS__Y); shifts are shift left logical, right logical, right arithmetic</td>
</tr>
<tr>
<td>SLT, SLTI, SLTU, SLTIU</td>
<td>Set less than, set less than immediate; signed and unsigned</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>Conditional branches and jumps; PC-relative or through register</td>
</tr>
<tr>
<td>BNEZ, BNE</td>
<td>Branch GPR equal/not equal to zero; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>BEQ, BNE</td>
<td>Branch GPR equal/not equal to 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>BCT, BCF</td>
<td>Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4</td>
</tr>
<tr>
<td>MOVN, MOVZ</td>
<td>Copy GPR to another GPR if third GPR is negative, zero</td>
</tr>
<tr>
<td>J, JR</td>
<td>Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)</td>
</tr>
<tr>
<td>JAL, JALR</td>
<td>Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)</td>
</tr>
<tr>
<td>TRAP</td>
<td>Transfer to operating system at a vectored address</td>
</tr>
<tr>
<td>RET</td>
<td>Return to user code from an exception; returns user mode</td>
</tr>
<tr>
<td><strong>Floating point</strong></td>
<td>FP operations on DP and SP formats</td>
</tr>
<tr>
<td>ADD, ADD.S, ADD.PS</td>
<td>Add DP, SP numbers, and pairs of SP numbers</td>
</tr>
<tr>
<td>SUB, SUB.S, ADD.PS</td>
<td>Subtract DP, SP numbers, and pairs of SP numbers</td>
</tr>
<tr>
<td>MUL, MUL.S, MUL.PS</td>
<td>Multiply DP, SP floating point, and pairs of SP numbers</td>
</tr>
<tr>
<td>MADD, MADD.S, MADD.PS</td>
<td>Multiply-add DP, SP numbers and pairs of SP numbers</td>
</tr>
<tr>
<td>DIV, DIV.S, DIV.PS</td>
<td>Divide DP, SP floating point, and pairs of SP numbers</td>
</tr>
<tr>
<td>CVT.__D, CVT.__S</td>
<td>Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.</td>
</tr>
</tbody>
</table>

**Figure B.26** Subset of the instructions in MIPS64. Figure 2.27 lists the formats of these instructions. SP = single precision; DP = double precision. This list can also be found on the page preceding the back inside cover.