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 Loc*,—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  
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
  - add R8, R7, R4
  - mul R9, R4, R5
  - sub R10, R9, R8

- C*B
- A + C*B
- A*B
- 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
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
Distribution of Displacement Values

Frequency of Immediate Operands

© 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

© 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

© 2003 Elsevier Science (USA). All rights reserved.
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

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

(a) Variable (e.g., VAX, Intel 80x86)

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

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 compiler different languages

• Multiple Target Compilers
  – Same compiler can generate code for different machines
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 \( R_0 \) always contains value 0:
  - \( R_0 + 16\text{-bit offset} \rightarrow \text{absolute addressing}; \)
  - 16-bit offset = 0 \( \rightarrow \) register indirect;
- branch instructions use PC relative addressing:
  - branch address = \([PC] + 4 + 4 \times 16\text{-bit offset} \)
- 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

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

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

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

Register-register ALU operations: \( rd \rightarrow rs \) funct \( rt \)
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and mems

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

### Figure B.23

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, R2(R0)</td>
<td>Load double word</td>
<td>Reg[R1] ← Mem[Reg[R2]]</td>
</tr>
<tr>
<td>LD R1, R2(R0)</td>
<td>Load double word</td>
<td>Reg[R1] ← Mem[1600+Reg[R2]]</td>
</tr>
<tr>
<td>LM R1, R2(R0)</td>
<td>Load word</td>
<td>Reg[R1] ← Mem<a href="Reg%5BR2%5D">Reg[R2]</a> +32 Mem[Reg<a href="Reg%5BR2%5D">R2</a>]</td>
</tr>
<tr>
<td>LH R1, R2(R0)</td>
<td>Load byte</td>
<td>Reg[R1] ← Mem[Reg<a href="Reg%5BR2%5D">R2</a> +1] Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32]</td>
</tr>
<tr>
<td>LHU R1, R2(R3)</td>
<td>Load byte unsigned</td>
<td>Reg[R1] ← Mem[Reg<a href="Reg%5BR2%5D">R2</a> +1] Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32]</td>
</tr>
<tr>
<td>LW R1, R2(R3)</td>
<td>Load half word</td>
<td>Reg[R1] ← Mem[Reg<a href="Reg%5BR2%5D">R2</a> +1] Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32]</td>
</tr>
<tr>
<td>LWU R1, R2(R3)</td>
<td>Load half word</td>
<td>Mem[Reg<a href="Reg%5BR2%5D">R2</a> +1] Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32]</td>
</tr>
<tr>
<td>1.5 FD_36(R1)</td>
<td>Load FP single</td>
<td>Reg[FD] ← Mem[Reg<a href="Reg%5BR1%5D">R1</a> +32]</td>
</tr>
<tr>
<td>1.0 FD_36(R2)</td>
<td>Load FP double</td>
<td>Reg[FD] ← Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32]</td>
</tr>
<tr>
<td>30 RS_300(R4)</td>
<td>Store double word</td>
<td>Mem[Reg<a href="Reg%5BR4%5D">R4</a> +32] ← Reg[R3]</td>
</tr>
<tr>
<td>SW RS_300(R4)</td>
<td>Store word</td>
<td>Mem[Reg<a href="Reg%5BR4%5D">R4</a> +32] ← Reg[R3]</td>
</tr>
<tr>
<td>5.5 FD_30(R1)</td>
<td>Store FP single</td>
<td>Mem[Reg<a href="Reg%5BR1%5D">R1</a> +32] ← Reg[FD]</td>
</tr>
<tr>
<td>5.0 FD_40(R3)</td>
<td>Store FP double</td>
<td>Mem[Reg<a href="Reg%5BR3%5D">R3</a> +32] ← Reg[FD]</td>
</tr>
<tr>
<td>SM RS_302(R2)</td>
<td>Store half</td>
<td>Mem[Reg<a href="Reg%5BR2%5D">R2</a> +32] ← Reg[R3]</td>
</tr>
<tr>
<td>SH RS_41(R3)</td>
<td>Store byte</td>
<td>Mem[Reg<a href="Reg%5BR3%5D">R3</a> +32] ← Reg[R3]</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

### Figure B.24

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>Reg[R1] ← Reg[R2] + Reg[R3]</td>
</tr>
<tr>
<td>ADDIU R1, R2, R3</td>
<td>Add immediate unsigned</td>
<td>Reg[R1] ← Reg[R2] + Reg[R3]</td>
</tr>
<tr>
<td>LUI R1, #R2</td>
<td>Load upper immediate</td>
<td>Reg[R1] ← 0(#R2)</td>
</tr>
<tr>
<td>DADD U R1, R2, R5</td>
<td>Shift left logical</td>
<td>Reg[R1] ← Reg[R2] &lt;&lt; Reg[R5]</td>
</tr>
<tr>
<td>DSLL R1, R2, #R1</td>
<td>Set less than</td>
<td>if (Reg[R2] &lt; Reg[R1]) Reg[R1] ← Reg[R1] else Reg[R1] ← 0</td>
</tr>
</tbody>
</table>

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

### 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.