Instruction Set Architectures

- Classification
- Addressing Modes
- Types of Instructions
- Encoding Instructions
- MIPS64 Instruction Set

ISA Classification (Fig. 2.1)
Addressing Modes

• There are a variety of ways a machine instruction can specify an address in memory.
• Effective address - the actual address used to access memory. May not be the address that appears in the instruction but is computed from the instruction address
• Impacts of having a variety of addr. modes:
  – can significantly reduce number of instructions needed
  – can increase average CPI
  – more complexity for hardware design and implementation

Usage of memory addressing modes

(Fig. 2.7)
Displacement addressing

Add R4, 100(R1)

<table>
<thead>
<tr>
<th>Add</th>
<th>R4</th>
<th>R1</th>
<th>100</th>
</tr>
</thead>
</table>

??? bits for this field of the instruction

Useful for accessing nearby data

How large should the displacement field be?

n bits \implies 0 \text{ to } 2^n \quad \text{if all positive}

\implies 0 \text{ to } 2^{(n-1)} \quad \text{if both positive and negative}

because one bit is used for the sign

Displacements used in benchmarks

(Fig. 2.8)

- x axis gives number of bits
- data is from a machine with max. 16 bit displacement
Immediate addressing

ADDI R1, R2, #2

Useful for many instructions that need a small constant
How large should the immediate field be?

\[ n \text{ bits } \implies 0 \text{ to } 2^n \text{ if all positive} \]
\[ \implies 0 \text{ to } 2^{n-1} \text{ if both positive and negative} \]

because one bit is used for the sign

Immediate data usage in benchmarks
(Fig. 2.9)

- Heavy use of immediate data
Immediate data usage in benchmarks

- Data is from a machine limited to 16 bit immediates

Types of instructions (by operation)

- arithmetic/logical
- data movement
- control
- floating point
- decimal
- string
- graphics
Top ten 80x86 instructions

<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>Cond. 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>Move R to R</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>
</tbody>
</table>

Control flow instructions

- conditional branches BEQZ HOME
- unconditional branches or jumps JMP ROPE
- procedure call
- procedure return

Target of the branch can be specified as
- explicit direct JMP ROPE
- explicit indirect JMP (R1)
- displacement JMP 100(R2)
- PC relative JMP 100(PC)
Use of branches in benchmarks (Fig. 2.19)

Displacements used in branches (Fig. 2.20)

- 4-8 bits captures most of the displacements
Instruction encoding issues

- total size(s) of instructions (halfword, word, doubleword)
- number of fields, number of operands
- number of bits for each field

Dependent on:
- number of registers
- types of instructions (operations)
- addressing modes
- etc.

Basic varieties of instruction encoding: variable, fixed, hybrid (Fig. 2.23)
MIPS64 Architecture

Registers
• 32 64-bit general purpose registers
  R0, R1, R2 ….. R31

• 32 64-bit floating points registers
  F0, F1, … F31

MIPS64 Architecture (cont)

Data types
• Integer data types
  byte (8 bits), halfword (16 bits),
  word (32 bits), doubleword (64 bits)

• Floating point data types
  single precision (32 bits)
  double precision (64 bits)
MIPS64 Architecture (cont)

Addressing modes (looks like four, only two)

- Immediate - 16 bits
  
  ADD R1, R2,#3

- Displacement - 16 bits
  
  LW R1,30(R2)

- Registered deferred/indirect - use 0 displacement
  
  LW R1,0(R2)

- Absolute - use R0 as the base, R0 = 0
  
  LW R1,508(R0)
### Deciphering MIPS64 notation

**LD R1,30(R2)**  
Regs[R1] ← 64 Mem[30 + Regs[R2]]

**LH R1,40(R3)**  
Regs[R1] ←
(Mem[40 + Regs[R3]] × 4)

Duplicate the sign bit, concatenate with the byte at Mem[40 + Regs[R3]] and concatenate with the byte at Mem[41 + Regs[R3]]. This makes a total of 64 bits.

### Deciphering MIPS64 notation

**JAL NAME**  
Regs[R31] ← PC + 4  
save return address

PC[36-63] ← NAME

Check bounds on value of NAME since this uses PC relative addressing

Used for subroutine calls