Instruction Set Architecture

or

“How to talk to computers if you aren’t on Star Trek”

How to Speak Computer

| High Level Language Program | temp = v[k]; v[k] = v[k+1]; v[k+1] = temp; |
| Compiler                  | 1w $15, 0($2) |
| Assembly Language Program | lw $16, 4($2) |
| Assembler                 | sw $16, 0($2) |
| Machine Language Program  | sw $15, 4($2) |
| Machine Interpretation    | 100111000011000010000000000000000 |
|                           | 100111000011100010000000000000000 |
|                           | 101011001111001000000000000000000 |
|                           | 101011000111000100000000000001000 |

Crafting an ISA

- Designing an ISA is both an art and a science
- ISA design involves dealing in an extremely rare resource
  – instruction bits!
- Some things we want out of our ISA
  – completeness
  – orthogonality
  – regularity and simplicity
  – compactness
  – ease of programming
  – ease of implementation
Where are the instructions?

- Harvard architecture
- Von Neumann architecture

“stored-program” computer

Key ISA decisions

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

Choice 1: Operand Location

- Accumulator
- Stack
- Registers
- Memory

We can classify most machines into 4 types: accumulator, stack, register-memory (most operands can be registers or memory), load-store (arithmetic operations must have register operands).

Choice 1B: How Many Operands?

Basic ISA Classes

<table>
<thead>
<tr>
<th>Accumulator:</th>
<th>Stack:</th>
<th>General Purpose Register:</th>
<th>Load/Store:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 address</td>
<td>0 address</td>
<td>2 address</td>
<td>3 address</td>
</tr>
<tr>
<td>add A</td>
<td>add</td>
<td>add A B</td>
<td>add Ra Rb Rc</td>
</tr>
<tr>
<td>acc ← acc + mem[A]</td>
<td>tos ← tos + next</td>
<td>EA(A) ← EA(A) + EA(B)</td>
<td>Ra ← Rb + Rc</td>
</tr>
</tbody>
</table>

A load/store architecture has instructions that do either ALU operations or access memory, but never both.
Trade-offs

Stack
+ -

Accumulator
+ -

GPR
+ -

Load-store
+ -

Choice 2: Addressing Modes
how do we specify the operand we want?

- Register direct  $R3$
  $R6 = R5 + R3$

- Immediate (literal)  $#25$
  $R6 = R5 + 25$

- Direct (absolute)  $M[10000]$  
  $R6 = M[10000]$

- Register indirect  $M[R3]$  (a.k.a. register deferred)
  $R6 = M[R3]$

- Memory Indirect  $M[M[R3]]$

- Displacement  $M[R3 + 10000]$ ...

- Index  $M[R3 + R4]$

- Scaled  $M[R3 + R4 \times d + 10000]$

- Autoincrement  $M[R3++]$

- Autodecrement  $M[R3 - -]$

Addressing Mode Utilization

<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>4%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>1%</td>
<td>16%</td>
</tr>
<tr>
<td>Register deferred</td>
<td>3%</td>
<td>11%</td>
<td>24%</td>
</tr>
<tr>
<td>Immediate</td>
<td>17%</td>
<td>32%</td>
<td>40%</td>
</tr>
<tr>
<td>Displacement</td>
<td>32%</td>
<td>40%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Displacement Size

Conclusion?
Choice 3: Which Operations?

- arithmetic
  - add, subtract, multiply, divide
- logical
  - and, or, shift left, shift right
- data transfer
  - load word, store word
- control flow

Does it make sense to have more complex instructions?
- e.g., square root, mult-add, matrix multiply, cross product ...

Types of branches (control flow)

- conditional branch
- jump
- procedure call
- procedure return

Does it make sense to have more complex instructions?

Conditional branch

- How do you specify the destination (target) of a branch/jump?
- How do we specify the condition of the branch?

Branch distance

- Average distance (in bits needed to specify) from branch to target.
- Conclusions?
Branch condition

Condition Codes
Processor status bits are set as a side-effect of arithmetic instructions or explicitly by compare or test instructions.
ex: sub r1, r2, r3
bz label

Condition Register
Ex: cmp r1, r2, r3
bgt r1, label

Compare and Branch
Ex: bgt r1, r2, label

The Customer is Always Right

• Compiler is primary customer of ISA
• Features the compiler doesn’t use are wasted
• Register allocation is a huge contributor to performance
• Compiler-writer’s job is made easier when ISA has
  – regularity
  – primitives, not solutions
  – simple trade-offs
• Summary -> simplicity over power

Our desired ISA

• Registers, Load-store
• Addressing modes
  – immediate (8-16 bits)
  – displacement (12-16 bits)
  – register deferred (register indirect)
• Support a reasonable number of operations
• Don’t use condition codes
• Fixed instruction encoding/length for performance
• regularity (several general-purpose registers)

Choice 4: Instruction Format

Fixed (e.g., all RISC processors -- SPARC, MIPS, Alpha)

<table>
<thead>
<tr>
<th>opcode</th>
<th>addr1</th>
<th>addr2</th>
<th>addr3</th>
</tr>
</thead>
</table>

Variable (VAX, ...)

<table>
<thead>
<tr>
<th>opcode+</th>
<th>spec1</th>
<th>addr1</th>
<th>spec2</th>
<th>addr2</th>
<th>...</th>
<th>specn</th>
<th>addrm</th>
</tr>
</thead>
</table>

Hybrid

• Tradeoffs?
• Conclusions?
MIPS instruction set architecture

- 32 32-bit general-purpose registers
  - R0 always equals zero
  - 32 or 16 FP registers
- 8-, 16-, and 32-bit integers, 32- and 64-bit fp data types
- immediate and displacement addressing modes
  - register deferred is a subset of displacement
- 32-bit fixed-length instruction encoding

MIPS Instruction Format

MIPS Operations and ISA

RISC vs CISC

- MIPS is a classic RISC architectures (as are SPARC, Alpha, PowerPC, …)
- RISC stands for Reduced Instruction Set Computer. RISC architectures are load-store, few formats, minimal instruction sets.
- They were in contrast to the 70s and 80s which proliferated CISC ISAs (VAX, Intel x86, various IBM), which were characterized by complex and comprehensive instruction sets, and complex instruction decoding.
- RISC architectures thrived not because they supported fewer operations, but because they enabled parallelism.
A few sample instructions

\[
\text{lw R1, 1000(R2)}
\]
\[
\text{add R1, R2, R3}
\]
\[
\text{addi R1, R2, #53}
\]
\[
\text{jal label}
\]
\[
\text{jr R3}
\]
\[
\text{beq R1, R5, label}
\]

MIPS R2000 vs. VAX 8700

Or “Why RISC?”

\[
\text{ET = IC \times CPI \times CT}
\]
\[
\text{IC}_{\text{MIPS}} = 2 \text{IC}_{\text{VAX}}
\]
\[
\text{CPI}_{\text{VAX}} = 6 \text{CPI}_{\text{MIPS}}
\]

ISA Key Points

- Modern ISA’s typically sacrifice power and flexibility for regularity and simplicity; code density for parallelism and throughput.
- Instruction bits are extremely limited, particularly in a fixed-length instruction format.
- Registers are critical to performance – we want lots of them, and few strings attached.
- Displacement addressing mode handles the vast majority of memory reference needs.