Instruction Set Architecture

or

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

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?

Operation

\[ y = x + b \]

Source operands

destination operand

How many? Which ones?

How does the computer know what
0001 0100 1101 1111
means?

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

**Accumulator:**

| 1 address | add A | acc ← acc + mem[A] |

**Stack:**

| 0 address | add | tos ← tos + next |

**General Purpose Register:**

| 2 address | add A B | EA(A) ← EA(A) + EA(B) |
| 3 address | add A B C | EA(A) ← EA(B) + EA(C) |

**Load/Store:**

| 3 address | add Ra Rb Rc | Ra ← Rb + Rc |
|           | load Ra Rb   | Ra ← mem[Rb] |
|           | store Ra Rb  | mem[Rb] ← Ra |

Alternative ISA’s

- \[ A = X*Y - B*C \]

| Stack Architecture | Accumulator | GPR | GPR (Load-store) |

Accumulator

Stack

Memory

A load/store architecture has instructions that do either ALU operations or access memory, but not both.
Choice 2: Addressing Modes

How do we specify the operand we want?

- Register direct
- Immediate (literal)
- Direct (absolute)
- Register indirect
- Memory indirect
- Displacement
- Index
- Scaled
- Autoincrement
- Autodecrement

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>gcc</th>
<th>spice</th>
<th>TeX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register direct</td>
<td>6%</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td>Immediate (literal)</td>
<td>1%</td>
<td>1%</td>
<td>0%</td>
</tr>
<tr>
<td>Direct (absolute)</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Register indirect (a.k.a register deferred)</td>
<td>24%</td>
<td>11%</td>
<td>6%</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Displacement</td>
<td>43%</td>
<td>59%</td>
<td>32%</td>
</tr>
<tr>
<td>Index</td>
<td>59%</td>
<td>43%</td>
<td>32%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>55%</td>
<td>55%</td>
<td>55%</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>55%</td>
<td>55%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Addressing Mode Utilization

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

Types of branches:

- Call/return
- Jump
- Conditional branch

Frequency of branch classes:

- Call/return: 13%
- Jump: 4%
- Conditional branch: 81%

Conditional branch

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

Branch distance

- 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
bg t r1, label

Compare and Branch
Ex: bgt r1, r2, label

Choice 4: Instruction Format

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

Variable (VAX, ...)

Hybrid

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)

DLX 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
DLX Instruction Format

I - type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rel</th>
<th>rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

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

R - type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs1</th>
<th>rs2</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

Register - register ALU operations: rs1, rs2, func
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
Trip and return from exception

DLX Operations

- Read on your own!
- Get comfortable with DLX instructions and formats

A few sample instructions

- lw R1, 1000(R2)
- add R1, R2, R3
- addi R1, R2, #53
- JAL label
- JR R3
- BEQZ R5, label

MIPS R2000 vs. VAX 8700

ET = IC * CPI * CT

- \( IC_{MIPS} = 2 IC_{VAX} \)
- \( CPI_{VAX} = 6 CPI_{MIPS} \)
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.