Instruction Set Design

Which is easier to change/design???
Instruction Set Architecture: What must be specified?

- Instruction Format or Encoding
  - how is it decoded?

- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?

- Data type and Size

- Operations
  - what are supported
  - *fetch-decode-execute is implicit!*

- Successor instruction
  - jumps, conditions, branches

Instruction Sets

- Instruction Set
  - An agreement between architects and machine language programmers

- Aspects
  - Operations
    - Arithmetic and logical +, -, X, /
    - Data Transfer - load/store
    - Control - branch, jump, call, return
    - Floating Point operations - FADD, FDIV
    - String
    - System support
    - HLL (high-level languages)
  - Operands
    - operand storage
    - number of operands
    - addressing operands
    - Type and size of operands
    - Implicit/Explicit operands
Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
  - CISC vs RISC

- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems

- And they in turn influence these factors

What is good for instruction set?

- 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
  - Resource needs easily determined

- Ease of compilation (programming?)

- Ease of implementation

- Scalability
Classifying Instruction Set Architecture

- **Stack**: 0 address
  
  \[ \text{add } (\text{sp}) \leftarrow (\text{sp}) + (\text{sp}-1) \]

- **Accumulator**: 1 address, 1+x address
  
  \[ \begin{align*}
  \text{add A} & \quad \text{acc} = \text{acc} + \text{mem}[A] \\
  \text{addx A} & \quad \text{acc} = \text{acc} + \text{mem}[A + x]
  \end{align*} \]

- **General Purpose Register**: 2,3 address
  
  \[ \begin{align*}
  \text{add A,B} & \quad \text{EA}(A) = \text{EA}(A) + \text{EA}(B) \\
  \text{add A B C} & \quad \text{EA}(A) = \text{EA}(B) + \text{EA}(C)
  \end{align*} \]

- **Load/Store**: 3 address
  
  \[ \begin{align*}
  \text{add Ra Rb Rc} & \quad \text{Ra} = \text{Rb} + \text{Rc} \\
  \text{load Ra Rb} & \quad \text{Ra} = \text{mem}[\text{Rb}] \\
  \text{store Ra Rb} & \quad \text{mem}[\text{Rb}] = \text{Ra}
  \end{align*} \]

  - access memory only with load and store instructions

Comparing Number of Instructions

Code sequence for \((C = A + B)\) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C,R3</td>
</tr>
</tbody>
</table>
Issues of Operands

- **Operand Storage**
  - either registers, memory, implicit

- **Number of operands**
  - 0: stack machine
  - 1: accumulator
  - 2,3: register set/memory

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg-reg (load/store) (0,3)</td>
<td>simple, fixed-length, fixed cycles</td>
<td>larger instr count, must be loaded into registers</td>
</tr>
<tr>
<td>reg-mem (1,2)</td>
<td>not necessarily loaded easy encoding good code density</td>
<td>inequivalent operands clocks may vary</td>
</tr>
<tr>
<td>mem-mem (3,3)</td>
<td>most compact, no register limit</td>
<td>large variation of instruction length, bottleneck in memory</td>
</tr>
</tbody>
</table>

Issues of Operands (Cont)

- **Addressing Operands**
  - **Endian Convention**
    - Ordering the bytes within a word
  - **Big Endian**: MSB at xx00

<table>
<thead>
<tr>
<th>word addr</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

  ex. : IBM, Motorola

  - **Little Endian**: LSB at xx00

<table>
<thead>
<tr>
<th>word addr</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

  ex. Intel, Dec

  - Aligned vs. Misaligned
Issues of Operands (Cont)

- **Addressing Mode**: determination of effective address
  - register \( \text{Ri} \)
  - Immediate \( \#n \)
  - Base + displacement \( \text{M[Ri + #n]} \)
  - Register indirect \( \text{M[Ri]} \)
  - Indexed \( \text{M[Ri, Rj]} \)
  - Absolute \( \text{M[#n]} \)
  - Memory indirect \( \text{M[M[Ri]]} \)
  - PC-relative \( \text{M[PC + #n]} \)
  - Auto-Increment \( \text{M[Ri]; Ri+=d} \)
  - Auto-Decrement \( \text{Ri-=d; M[Ri]} \)

Implicit/Explicit operands

- **Compiler issue**
- **Take branch on conditional code as example**

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condition code</strong></td>
<td>- be set for free</td>
<td>- Constrains code reordering, - Extra state to save</td>
</tr>
<tr>
<td><strong>Condition register</strong></td>
<td>- Simple, - No special state to save</td>
<td>- Use up a register</td>
</tr>
<tr>
<td><strong>Compare &amp; branch</strong></td>
<td>- No extra compare, - No state passed between instructions</td>
<td>- May be too much work per instruction</td>
</tr>
</tbody>
</table>
**Instruction Set Architecture**

- **CISC vs. RISC**

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction set</td>
<td>Large instr variety</td>
<td>Small instr. Set</td>
</tr>
<tr>
<td></td>
<td>Variable formats</td>
<td>Fixed format</td>
</tr>
<tr>
<td></td>
<td>Variable instr length</td>
<td>Fixed length</td>
</tr>
<tr>
<td>Operand storage</td>
<td>reg-mem</td>
<td>reg-reg</td>
</tr>
<tr>
<td></td>
<td>mem-reg</td>
<td></td>
</tr>
<tr>
<td>Addressing mode</td>
<td>complex</td>
<td>simple</td>
</tr>
<tr>
<td>GP registers</td>
<td>8-24 + special regs</td>
<td>Large number</td>
</tr>
<tr>
<td>CPU control</td>
<td>Microcode, hardwired</td>
<td>Hardwired</td>
</tr>
<tr>
<td>Cache/TLB</td>
<td>External cache</td>
<td>On-chip cache &amp; TLB</td>
</tr>
</tbody>
</table>

- **Instruction Format Widths**

Variable: ...

Hybrid: ...

Fixed: ...

- **Encoding Instruction**

<table>
<thead>
<tr>
<th></th>
<th>Fixed</th>
<th>Hybrid</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L/S load/store</td>
<td>R/M Reg-mem</td>
<td>R+M Reg-plus-mem</td>
</tr>
<tr>
<td></td>
<td>32b instr size</td>
<td>2-addr format 16/32/64b instr sizes</td>
<td>2/3-addr format byte-variable size</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>IBM S/360</td>
<td>IBM 3033</td>
<td>VAX</td>
</tr>
<tr>
<td>IBM PowerPC</td>
<td>IBM S/360</td>
<td>Fujitsu</td>
<td>Motorola 680x0</td>
</tr>
<tr>
<td>MIPS R2000</td>
<td>IBM 3033</td>
<td>Hitachi</td>
<td></td>
</tr>
<tr>
<td>HP PA RISC</td>
<td>Fujitsu</td>
<td>Hitachi</td>
<td></td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>Hitachi</td>
<td>Intel x86</td>
<td></td>
</tr>
<tr>
<td>AMD 29000</td>
<td>Intel x86</td>
<td>(byte-variable size)</td>
<td></td>
</tr>
<tr>
<td>SPARC (reg window)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© by Tien-Fu Chen@CCU
**DLX**

- A RISC architecture related to MIPS
- 32-bit byte addresses
- Load/Store - only displacement

**Registers**
- 32 32-bit General-Purpose Registers
- 16 64-bit (32 32-bit) Floating-Point Register
- FP status register

**Emphasize**
- A simple load/store instruction set
- Design for pipelining efficiency
- An easily decoded instruction set
- Efficiency as a compiler target

---

**DLX (Cont)**

- Three fixed-length instruction formats
  - **I-Format**
    - ALU ops with immed: \( \text{rd} \leftarrow \text{rs1} \text{ op } \#\text{immed} \)
    - Load/store: \( \text{rd} \leftrightarrow \text{M[rs1+#imm]} \)
    - Conditional branch: \( \text{pc} \leftarrow \text{cond: pc }+=\#\text{im} \)
  - **R-Format**
    - Reg-Reg ALU operations: \( \text{rd }\leftarrow \text{rs1 func rs2} \)
  - **J-Format**
    - Unconditional jumps: \( \text{pc} += \#\text{offset} \)
**DLX Instruction Set**

- **Data transfers**
  - load/store full word
  - load/store byte/halfword
  - load/store FP single/double
  - moves between GPR and FP registers

- **Arithmetic/Logical**
  - Add/Subtract unsigned, immediate
  - MUL/DIV signed, unsigned
  - AND, OR, XOR immediate
  - Load high immediate
  - Shift left/right

- **Control**
  - Conditional branch
  - Conditional branch testing FP bit
  - Jump&link(JAL),Jump&link register(JALR)
  - Jump, Jump register

- **Floating Point**

**MIPS instruction format**

- **Register-Register**
  \[
  \begin{array}{cccccccccc}
  31 & 26 & 25 & 21 & 20 & 16 & 15 & 11 & 10 & 6 & 5 & 0 \\
  \hline
  Op & Rs1 & Rs2 & Rd & & & & & & & & & \\
  \end{array}
  \]

- **Register-Immediate**
  \[
  \begin{array}{cccccccccc}
  31 & 26 & 25 & 21 & 20 & 16 & 15 & & & & & 0 \\
  \hline
  Op & Rs1 & Rd & & & & & immediate & & & & & \\
  \end{array}
  \]

- **Branch**
  \[
  \begin{array}{cccccccccc}
  31 & 26 & 25 & 21 & 20 & 16 & 15 & & & & & 0 \\
  \hline
  Op & Rs1 & Rs2/Opx & & & & & immediate & & & & & \\
  \end{array}
  \]

- **Jump / Call**
  \[
  \begin{array}{cccccccccc}
  31 & 26 & 25 & & & & & & & & & target & 0 \\
  \hline
  Op & & & & & & & & & & & & \\
  \end{array}
  \]

- **A "Typical" RISC**
  - 3-address, reg-reg arithmetic instruction
  - Single address mode for load/store: base + displacement
  - no indirection
  - Simple branch conditions
  - Delayed branch
When does CPU need to sign extend?

- When value is sign extended, copy upper bit to full value:
  Examples of sign extending 8 bits to 16 bits:

\[
\begin{align*}
00001010 & \Rightarrow 00000000\ 00001010 \\
10001100 & \Rightarrow 11111111\ 10001100
\end{align*}
\]

- When is an immediate value sign extended?
  - Arithmetic instructions (add, sub, etc.) sign extend immediates even for the unsigned versions of the instructions!
  - Logical instructions do not sign extend

- Load/Store half or byte do sign extend, but unsigned versions do not.

---

**MIPS data transfer instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW 500(R4), R3</td>
<td>Store word</td>
</tr>
<tr>
<td>SH 502(R2), R3</td>
<td>Store half</td>
</tr>
<tr>
<td>SB 41(R3), R2</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW R1, 30(R2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU R1, 40(R3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LUI R1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

Why need LUI?

© by Tien-Fu Chen@CCU