Chapter 2
Instruction Set Principle and Examples

Instruction Set Architecture

- Instruction set architecture is the *interface* of a computer that a machine language programmer, an operating system designer, or a compiler writer must understand to write a correct machine program, an operating system, a compiler, respectively for that machine.

- The instruction set architecture is also the *interface* that a hardware designer must understand to design a correct implementation of the computer.

Towards Evaluation of ISA and Organization

- A good interface:
  - Lasts through many implementations (portability, compatibility)
  - Is used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Interface Design
Machine Instruction

- Carries out a step of processing
- Must specify:
  - Function - OPCODE
  - Input Operand(s)
  - What to do with results
  - What to do next?

Four Address Instruction

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
<th>Next ?</th>
</tr>
</thead>
</table>

Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs. GPR (System 360)
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
  - And they in turn influence these
**Internal Storage:**
Stack, Accumulator, and Register

<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>Store C,R3</td>
<td></td>
</tr>
</tbody>
</table>

\[ C = A + B \]

**Machine Types**

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Simple Model, Good code</td>
<td>Can not address randomly</td>
</tr>
<tr>
<td></td>
<td>density</td>
<td>Stack Bottleneck</td>
</tr>
<tr>
<td>Accumulator</td>
<td>Minimizes internal states</td>
<td>Highest memory traffic</td>
</tr>
<tr>
<td></td>
<td>Short Instructions</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Most general model for</td>
<td>All operands must be named</td>
</tr>
<tr>
<td></td>
<td>code generation</td>
<td>longer instructions</td>
</tr>
</tbody>
</table>

**Emergence of GPR machines**

**Registers**
- Fastest memory
- Easier for compilers to manipulate
  - Hold variable
    - Reduce memory traffic
    - Speedup programs
    - Improves code density
- Register Load-Store – most popular
- How many registers?

**Classifying Instruction Set Architecture**

**Internal Storage**
- Stack
- Accumulator
- Register

**Operands**
- Number : (0,1,2,3)
- Type, Size : byte, int, float
- Location : memory or register (effective address)

**Operations**
- Type : add, sub, mul, ...
- How is it specified?
### Characterizing GPR architectures
(AlU instructions)

<table>
<thead>
<tr>
<th>No. of Mem Addresses per ALU Instruction</th>
<th>Max NO of Ops allowed</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>IBM RT</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>SPARC, MIPS, PowerPC, Alpha</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PDP-10, M68000, IBM 360</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>IBM 360 (RS Inst)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>PDP 11, National 32x32, IBM 360</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>VAX</td>
</tr>
</tbody>
</table>

### General Purpose Register Arch.

**Register-Register (0,3)**

- **Advantages**
  - Simple, Fixed Length Instruction Encoding
  - Simple Code Generation Model
  - Similar CPI values
- **Disadvantages**
  - Higher Instruction Count
  - Bit Encoding may be wasteful

**Register-Memory (1,2)**

- **Advantages**
  - No loading first required
  - Easy to encode with good density
- **Disadvantages**
  - Source operand is destroyed
  - CPI Varies
  - Encoding a register No. in each inst - limit Reg No.

**Memory-Memory (3,3)**

- **Advantages**
  - Most compact
  - Does not waste registers for temp info
- **Disadvantages**
  - Large variation in instruction size
  - Large variation in work per instruction
  - Memory access may create memory bottleneck
Memory Addressing

- Ordering of bytes within a word
- Alignment
- Addressing modes

Byte Ordering: Big/Little Endian

byte whose address is “x…x00” at

<table>
<thead>
<tr>
<th>Big endian: most-significant position</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM mainframes, MIPS, 680x0, SPARC</td>
</tr>
<tr>
<td>[31 0 1 2 3]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Big Endian</td>
</tr>
<tr>
<td>[0 0 1 2 3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little endian: least-significant position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vax, Intel 80x86</td>
</tr>
<tr>
<td>[31 0 1 2 3]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Little Endian</td>
</tr>
<tr>
<td>[0 0 1 2 3]</td>
</tr>
</tbody>
</table>

Alignment

Access to an object of size $s$ bytes at byte address $A$ is aligned if $A \mod s = 0$

Addressing modes

Addressing mode: how architectures specify the (address of an) object that they will access

Effective address: actual memory address specified

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R1, R2</td>
<td>opcodes</td>
<td>When a value is in a register.</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADD R1, 32</td>
<td>opcodes</td>
<td>For constants.</td>
</tr>
<tr>
<td>Displacement</td>
<td>ADD R1, 10H</td>
<td>opcodes</td>
<td>Accessing local variables.</td>
</tr>
<tr>
<td>Register defined in indirect</td>
<td>ADD R1, R1</td>
<td>opcodes</td>
<td>Accessing a pointer or a computed address.</td>
</tr>
<tr>
<td>Indirect</td>
<td>ADD R1, [R0,R1]</td>
<td>opcodes</td>
<td>Sometimes useful in array addressing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F = #data pointer</td>
</tr>
</tbody>
</table>

32 bits

To Processor
### Addressing modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct or absolute</td>
<td><code>Add R1, [R0]</code></td>
<td>Simplest form, no additional addressing information needed.</td>
</tr>
<tr>
<td>Memory indirect or memory deferred</td>
<td><code>Add R1, [R0, #5]</code></td>
<td>Access memory indirectly through a register.</td>
</tr>
<tr>
<td>Accumulator</td>
<td><code>Add R1, (R2)</code></td>
<td>Access memory indirectly through the accumulator.</td>
</tr>
<tr>
<td>Base Register – Displacement</td>
<td><code>Add R1, [R0, R2]</code></td>
<td>Uses both a base register and a displacement.</td>
</tr>
</tbody>
</table>

- Sometimes useful for accessing static data; address computation need to be large.
- May increase the average CPI.

---

### Addressing Modes

- Significantly reduce Instruction Count
- Add to the complexity of Building a Machine
- May Increase the Average CPI

**IMPORTANT FACTOR FOR AN ARCHITECT**

---

### Summary of Use of Memory Addressing Modes

![Summary of Use of Memory Addressing Modes](image)

- Memory indirect
- Scoped
- Register deferred
- Immediate
- Displacement

**Frequency of the addressing mode**

- **0%**
- **10%**
- **20%**
- **30%**
- **40%**
- **50%**
- **60%**

---

### Displacement Addressing Mode

- How large should the displacement field be?
  - Directly influence instruction length
- May have to depend on measurements taken from typical programs
Immediate or Literal Addressing Modes

- Used
  - in arithmetic operations
  - in comparisons (primarily for branches)
  - in moves where a constant is needed in a register
    - Constant written in code (Small)
    - address constants (Large)
- Range of Value for Immediates
  - Again, directly influence instruction length
- Again, may have to depend on measurements taken from typical programs

Summary: Memory Addressing

- Expect an architecture to support
  - displacement
  - immediate
  - register deferred

These modes represent 75% to 99% of modes used for some typical programs
- Displacement mode to be 12 to 16 bits
- Immediate field to be 8 to 16 bits
Classifying Instruction Set Architecture

- Internal Storage
- Stack
- Accumulator
- Register

Operands
- Number: (0, 1, 2, 3)
- Type, Size: byte, int, float
- Location: memory or register (effective address)

Operations
- Type: add, sub, mul, ...
- How is it specified?

Operations in the Instruction Set

- Instruction Categories
- Top 10 Instructions

Instruction Categories

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic and logical operations: add, and, subtract, or</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return, traps</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating-point operations: add, multiply</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal addition, decimal multiply, decimal-to-character conversions</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Print operations, compression/decompression operations</td>
</tr>
</tbody>
</table>

Top 10 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 instruction</th>
<th>Integer average of (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional 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 register-register</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>90%</td>
</tr>
</tbody>
</table>
Control-Flow

1. Conditional Branches
2. Jumps
3. Procedure Calls
4. Procedure Returns

![Diagram showing breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches.]

Destination Address

- PC Relative
  - Target often near the current instruction
  - Requires fewer bits
  - Also, makes the code Position Independent
- What branch offset should be supported?

![Diagram showing destination address distribution.]

Specifying Branch Condition

<table>
<thead>
<tr>
<th>Name</th>
<th>How condition is tested</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Special bits are set by ALU operations, possibly under program control.</td>
<td>Sometimes condition is set for free.</td>
<td>CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.</td>
</tr>
<tr>
<td>register</td>
<td>Test arbitrary register with the result of a comparison.</td>
<td>Simple.</td>
<td>Use up a register.</td>
</tr>
<tr>
<td>Compare and</td>
<td>Compare is part of the branch. Often compact is limited to subset.</td>
<td>One instruction rather than two for a branch.</td>
<td>May be too much work per instruction.</td>
</tr>
<tr>
<td>branch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Properties of branch instructions

- Large number of the comparisons are
  - Simple equalities or inequalities
  - Comparisons with zero

  ➔ treat as special cases

![Diagram showing frequency of comparison types in branches.]

Page 9
Destination Address Unknown

- Procedure returns or indirect jumps
- Need to specify target dynamically
- May use any addressing mode for destination address
- Use a register as address
  - Useful for
    - Case or Switch statements
    - Dynamically Shared Libraries
    - Loaded at run time as needed
    - Virtual Functions
    - Allow different routines to be called depending on data type

Procedure Call and Returns

Include control transfer and some state saving
- At a minimum return address has to be saved
- May also save registers
  - Through hardware action
  - Through software instructions
- Two conventions for saving registers
  - Caller Saving
    - Calling procedure must save the registers
  - Callee Saving
    - Called program must save the registers it wants to use
- Use depends on the program - Compilers use a combination