Lecture 3
Instruction Set Architecture

Computer Architectures
521480S
Hot Topics in Computer Architecture

- **1950s and 1960s:**
  - Computer Arithmetic

- **1970 and 1980s:**
  - Instruction Set Design
  - ISA Appropriate for Compilers

- **1990s:**
  - Design of CPU
  - Design of memory system
  - Instruction Set Extensions
    - e.g. MMX for x86 instruction set: SIMD (single instruction multiple data) support
    - use for graphics, video and audio, for example.

- **2000s:**
  - Computer Arithmetic
  - Design of I/O system
  - Parallelism
Instruction Set Architecture

• “Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.”
  – Source: IBM in 1964 when introducing the IBM 360 architecture, which eliminated 7 different IBM instruction sets.

• The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.
Interface Design

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
Instruction Set Architecture

- The instruction set architecture serves as the interface between software and hardware.
- It provides the mechanism by which the software tells the hardware what should be done.

High level language code: C, C++, Java, Fortran,
  \[\text{compiler}\]
Assembly language code: architecture specific statements
  \[\text{assembler}\]
Machine language code: architecture specific bit patterns

software

instruction set

hardware
ISA Metrics

- **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 (or assembly language programming)**

- **Ease of implementation**
Instruction Set Design Issues

- Instruction set design issues include:
  - Instruction format
    » how is decoded? how many bits? fixed or variable length?
  - Where are operands stored?
    » registers, memory, stack, accumulator
  - How many explicit operands are there?
    » 0, 1, 2, or 3
  - How is the operand location specified?
    » register, immediate, indirect, . . .
  - What type & size of operands are supported?
    » byte, int, float, double, string, vector. . .
  - What operations are supported?
    » add, sub, mul, move, compare . . .
**Evolution of Instruction Sets**

- **Single Accumulator** (EDSAC 1950)
- **Accumulator + Index Registers** (Manchester Mark I, IBM 700 series 1953)
- **Separation of Programming Model from Implementation**
- **High-level Language Based** (B5000 1963)
- **Concept of a Family** (IBM 360 1964)
- **General Purpose Register Machines**
- **Complex Instruction Sets** (Vax, Intel 8086 1977-80)
- **Load/Store Architecture** (CDC 6600, Cray 1 1963-76)
- **RISC** (Mips, Sparc, 88000, IBM RS6000, . . . 1987+)
Classifying ISAs

Accumulator (before 1960):
1 address  add A       acc ← acc + mem[A]

Stack (1960s to 1970s):
0 address  add          tos ← tos + next
(tos = top-of-stack)

Memory-Memory (1970s to 1980s):
2 address  add A, B     mem[A] ← mem[A] + mem[B]
3 address  add A, B, C  mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present):
2 address  add R1, A   R1 ← R1 + mem[A]
load R1, A  R1 ← mem[A]

Register-Register (Load/Store) (1960s to present):
3 address  add R1, R2, R3  R1 ← R2 + R3
load R1, R2  R1 ← mem[R2]
store R1, R2 mem[R1] ← R2
Comparison of ISA Classes

- Code Sequence for $C = A + B$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-Mem)</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>

- Code density (memory efficiency)?
  Instruction access? Data access?
Accumulator Architectures

• Instruction set:
  add A, sub A, mult A, div A, . . .
  load A, store A

• Example: A*B - (C*B+A)
  load B
  mul C
  add A
  store D
  load A
  mul B
  sub D

  B
  B*C
  B*C+A
  A
  A*B
  A*B – B*C+A
Accumulators: Pros and Cons

• Pros
  – Very low hardware requirements
  – Easy to design and understand

• Cons
  – Accumulator becomes the bottleneck
    – intermediate results must be stored to memory (slow)
  – Little ability for parallelism or pipelining
    – structural/data hazards (many instructions use the same accumulator): lot of dependencies in a block of code
  – High memory traffic
    – no registers for temporarily storing data
Stack Architectures

- **Instruction set:**
  - add, sub, mult, div, . . .
  - push A, pop A

- **Example: A*B - (A+C*B)**
  - push A
  - push B
  - mul
  - push A
  - push C
  - push B
  - mul
  - add
  - sub
Stacks: Pros and Cons

• Pros
  – Good code density
    » operands/results on stack
  – Low hardware requirements
  – Easy to write a simpler compiler for stack architectures

• Cons
  – Stack becomes the bottleneck
  – Little ability for parallelism or pipelining
    » instructions depend on stack and its current state
  – Data is not always at the top of stack when needed, so additional instructions like ROT and SWAP are needed
  – Difficult to write an optimizing compiler for stack architectures
Memory-Memory Architectures

• Instruction set:
  (3 operands)  add A, B, C  sub A, B, C  mul A, B, C
  (2 operands)  add A, B  sub A, B  mul A, B

• Example: A*B - (A+C*B)
  – 3 operands
    mul D, A, B
    mul E, C, B
    add E, A, E
    sub E, D, E
  2 operands
    mov D, A
    mul D, B
    mov E, C
    mul E, B
    add E, A
    sub E, D
Memory-Memory: Pros and Cons

• Pros
  – Requires fewer instructions (especially if 3 operands)
    – most compact
  – Easy to write compilers for (especially if 3 operands)

• Cons
  – Very high memory traffic (especially if 3 operands)
  – Variable instruction size
    – complex instruction decoding
Register-Memory Architectures

• Instruction set:
  - `add R1, A`
  - `sub R1, A`
  - `mul R1, B`
  - `load R1, A`
  - `store R1, A`

• Example: `A*B - (A+C*B)`
  - `load R1, A`
  - `mul R1, B /* A*B */`
  - `store R1, D`
  - `load R2, C`
  - `mul R2, B /* C*B */`
  - `add R2, A /* A + CB */`
  - `sub R2, D /* AB - (A + C*B) */`
Memory-Register: Pros and Cons

- **Pros**
  - Some data can be accessed without loading first
    - Good code density
  - Instruction format easy to encode

- **Cons**
  - Operands are not equivalent (poor orthogonal)
  - Variable number of clocks per instruction
  - one operand is destroyed
Load-Store Architectures

- Instruction set:
  - add R1, R2, R3
  - sub R1, R2, R3
  - mul R1, R2, R3
  - load R1, &A
  - store R1, &A
  - move R1, R2

- Example: \( A \times B - (A + C \times B) \)
  - load R1, &A
  - load R2, &B
  - load R3, &C
  - mul R7, R3, R2 /* \( C \times B \) */
  - add R8, R7, R1 /* \( A + C \times B \) */
  - mul R9, R1, R2 /* \( A \times B \) */
  - sub R10, R9, R8 /* \( A \times B - (A + C \times B) \) */
Load-Store: Pros and Cons

• Pros
  – Simple, fixed length instruction encodings
  – Instructions take similar number of cycles
  – Relatively easy to pipeline and make superscalar

• Cons
  – Higher instruction count
  – Not all instructions need three operands
    – reduces code density
  – Dependent on good compiler
    – e.g. optimizes register usage for pipelining
Registers: Advantages and Disadvantages

• Advantages
  – Faster than cache or main memory (no addressing modes or tags)
  – Deterministic (no misses)
  – Can replicate (multiple read ports)
  – Short identifier (typically 3 to 8 bits)
  – Reduce memory traffic

• Disadvantages
  – Need to save and restore on procedure calls and context switch
  – Can’t take the address of a register (for pointers)
  – Fixed size (can’t store strings or structures efficiently)
  – Compiler must manage
  – Limited number
Byte Ordering

• **Idea**
  - Bytes in long word numbered 0 to 3
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines

• **Big Endian: Byte 0 is most, 3 is least**
  - IBM 360/370, Motorola 68K, Sparc.

• **Little Endian: Byte 0 is least, 3 is most**
  - Intel x86, VAX

• **Alpha**
  - Chip can be configured to operate either way
  - DEC workstation are little endian
  - Cray T3E Alpha’s are big endian
Big Endian Addressing

• With Big Endian addressing, the byte binary address

  \[x \ldots x00\]

  is in the most significant position (big end) of a 32 bit word (IBM, Motorolla, Sun, HP).

<table>
<thead>
<tr>
<th>MSB</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Little Endian Addressing

• With Little Endian addressing, the byte binary address
  \[ x \ldots x00 \]
  is in the least significant position (little end) of a 32 bit word (DEC, Intel).

<table>
<thead>
<tr>
<th>MSB</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

• Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines
Operand Alignment

• An access to an operand of size $s$ bytes at byte address $A$ is said to be *aligned* if $A \mod s = 0$

<table>
<thead>
<tr>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
</tr>
</tbody>
</table>

$A \mod s = 0$

$A \mod s = 1$
Unrestricted Alignment

• If the architecture does not restrict memory accesses to be aligned then
  – Software is simple
  – Hardware must detect misalignment and make two memory accesses
  – Expensive logic to perform detection
  – Can slow down all references
  – Sometimes required for backwards compatibility
Restricted Alignment

• If the architecture restricts memory accesses to be aligned then
  – Software must guarantee alignment
  – Hardware detects misalignment access and traps
  – No extra time is spent when data is aligned

• Since we want to make the common case fast, having restricted alignment is often a better choice, unless compatibility is an issue.
### Types of Addressing Modes (VAX)

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Register direct</td>
<td>Add R4, R3</td>
<td>R4 &lt;- R4 + R3</td>
</tr>
<tr>
<td>2. Immediate</td>
<td>Add R4, #3</td>
<td>R4 &lt;- R4 + 3</td>
</tr>
<tr>
<td>3. Displacement</td>
<td>Add R4, 100(R1)</td>
<td>R4 &lt;- R4 + M[100 + R1]</td>
</tr>
<tr>
<td>4. Register indirect</td>
<td>Add R4, (R1)</td>
<td>R4 &lt;- R4 + M[R1]</td>
</tr>
<tr>
<td>5. Indexed</td>
<td>Add R4, (R1 + R2)</td>
<td>R4 &lt;- R4 + M[R1 + R2]</td>
</tr>
<tr>
<td>6. Direct</td>
<td>Add R4, (1000)</td>
<td>R4 &lt;- R4 + M[1000]</td>
</tr>
<tr>
<td>7. Memory Indirect</td>
<td>Add R4, @(R3)</td>
<td>R4 &lt;- R4 + M[M[R3]]</td>
</tr>
<tr>
<td>8. Autoincrement</td>
<td>Add R4, (R2)+</td>
<td>R4 &lt;- R4 + M[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 &lt;- R2 + d</td>
</tr>
<tr>
<td>9. Autodecrement</td>
<td>Add R4, (R2)-</td>
<td>R4 &lt;- R4 + M[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 &lt;- R2 - d</td>
</tr>
<tr>
<td>10. Scaled</td>
<td>Add R4, 100(R2)[R3]</td>
<td>R4 &lt;- R4 + M[100 + R2 + R3*d],</td>
</tr>
</tbody>
</table>

where d is typically 1, 2, 4, or 8

- Studies by [Clark and Emer] indicate that modes 1-4 account for 93% of all operands on the VAX.
Addressing Modes

Immediate
Add R4, #3

Register
Add R4, R3

Register Indirect
Add R4,(R1)

Operand

R3

Operand

R1

Registers

Operand

Registers

Memory
Addressing Modes (Cont.)

Direct
Add R4, (1001)

Memory Indirect
Add R4, @(R3)
Addressing Modes (Cont.)

Displacement
Add R4, 100(R1)

Scaled
Add R1, 100(R2) [R3]
Types of Operations

- **Arithmetic and Logical:**
  - add, subtract, and, or, etc.
- **Data transfer:**
  - Load, Store, etc.
- **Control**
  - Jump, branch, call, return, trap (= software interrupt), etc.
- **Synchronization:**
  - Test & Set
- **String:**
  - string move, compare, search.
80x86 Instruction Frequency (top 10)

<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>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>register-register move</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>96%</td>
</tr>
</tbody>
</table>

- Simple Instructions dominate instruction frequency.
Relative Frequency of Control Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>SPECint92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call/Return</td>
<td>13%</td>
<td>11%</td>
</tr>
<tr>
<td>Jumps</td>
<td>6%</td>
<td>4%</td>
</tr>
<tr>
<td>Branches</td>
<td>81%</td>
<td>87%</td>
</tr>
</tbody>
</table>

- Design hardware to handle branches quickly, since these occur most frequently
Methods of Testing Condition

• **Condition code**: Status bits are set by ALU operations.
  - Add r1, r2, r3 and bz label
  - Extra status bits: complicates HW

• **Condition register**:
  - cmp r1, r2, r3 and bgt r1, label
  - Simple, but use up a register

• **Compare and branch**
  - bgt r1, r2, label
  - One instruction
  - Too much work per instruction
Conditional Branch Distance

- Short displacement fields often sufficient for branch
Conditional Branch Addressing

- PC-relative, since most branches from current PC address
  - At least 8 bits.

- Compare Equal/Not Equal most important for integer programs.

![Bar chart showing frequency of comparison types]

- **EQ/NEQ**: 37% for Integer, 87% for FP
- **GT/LE**: 7% for Integer, 23% for FP
- **LT/GE**: 7% for Integer, 40% for FP

*Frequency of comparison types*
Data Types and Usage

- Byte, half word (16 bits), word (32 bits), double word (64 bits).

- Arithmetic:
  - Decimal (BCD): 4bit per digit.
  - Integers: 2’s complement
  - Floating-point: IEEE-754 standard-- single, double, extended precision.

```
0 % 0 % 0 % 0 %
74 % 74 % 74 % 74 %
31 % 31 % 31 % 31 %
19 % 19 % 19 % 19 %
19 % 19 % 19 % 19 %
```

Double Word | 0 % | 69 %
---|---|---
Word | 31 % | 74 %
Half Word | 0 % | 19 %
Byte | 0 % | 7 %
Frequency of Operand Sizes on 32-bit Load-Store Machines

<table>
<thead>
<tr>
<th>Size</th>
<th>SPECint92</th>
<th>SPECfp92</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>0%</td>
<td>69%</td>
</tr>
<tr>
<td>32 bits</td>
<td>74%</td>
<td>31%</td>
</tr>
<tr>
<td>16 bits</td>
<td>19%</td>
<td>0%</td>
</tr>
<tr>
<td>8 bits</td>
<td>19%</td>
<td>0%</td>
</tr>
</tbody>
</table>

• For floating-point want good performance for 64 bit operands.
• For integer operations want good performance for 32 bit operands.
• Recent architectures also support 64-bit integers.
Instruction Encoding

• Variable
  – Instruction length varies based on opcode and address specifiers
  – For example, VAX instructions vary between 1 and 53 bytes, while x86 instruction vary between 1 and 17 bytes.
  – Good code density, but difficult to decode and pipeline

• Fixed
  – Only a single size for all instructions (including simple ones)
  – For example, DLX, MIPS, Power PC, Sparc all have 32 bit instructions
  – Not as good code density, but easier to decode and pipeline: simpler HW

• Hybrid
  – Have multiple format lengths specified by the opcode
  – For example, IBM 360/370, Intel x86
  – Compromise between code density and ease of decode

• Summary:
  – If code size is most important, use variable format.
  – If performance is most important, use fixed format.
Compilers and ISA

• **Compiler Goals**
  – All correct programs compile correctly
  – Most compiled programs execute quickly
  – Most programs compile quickly
  – Achieve small code size
  – Provide debugging support

• **Multiple Source Compilers**
  – Same compiler can compile different languages

• **Multiple Target Compilers**
  – Same compiler can generate code for different machines
Compilers Phases

- Compilers use phases to manage complexity
  - Front end
    - Convert language to intermediate form (RTL)
  - High level optimizer
    - Procedure inlining and loop transformations
  - Global optimizer
    - Global and local optimization, plus register allocation
  - Code generator (and assembler)
    - Dependency elimination, instruction selection, pipeline scheduling
Designing ISA to Improve Compilation

- Provide enough general purpose registers to ease register allocation (more than 16).
- Provide regular instruction sets by keeping the operations, data types, and addressing modes orthogonal
  - e.g. data types: 8-, 16-, 32-bit integer, IEEE FP standard
- Provide primitive constructs rather than trying to map to a high-level language.
- Simplify trade-offs among alternatives.
- Allow compilers to help make the common case fast
ISA Summary

• Use general purpose registers with a load-store architecture.
• Support these addressing modes: displacement, immediate, register indirect.
• Support these simple instructions: load, store, add, subtract, move register, shift, compare equal, compare not equal, branch, jump, call, return.
• Support these data size: 8-, 16-, 32-bit integer, IEEE FP standard.
• Provide at least 16 general purpose registers plus separate FP registers and aim for a minimal instruction set.