ISA Issues;
Performance Considerations
Key ISA decisions

• Instruction format – Length? Variable? Fixed? Fields?
• How many registers?
• Where do instruction operands reside?
  – e.g., can you add contents of memory to a register?
• Operands
  – how many? how big?
  – how are memory addresses computed?
**Instruction Length**

**Variable:**

- x86 – Instructions vary from 1 to 17 Bytes long

**Fixed:**

- VAX – from 1 to 54 Bytes

MIPS, PowerPC, and most other RISC’s:

all instruction are 4 Bytes long
Instruction Length

- Variable-length instructions (x86, VAX):
  - require multi-step, complex fetch and decode (-)
  - allow smaller binary programs that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency (+)

- Fixed-length instructions (RISC’s)
  - allow easy fetch and decode (+)
  - simplify pipelining and parallelism (+)
  - result in larger binary programs that require more disk storage, more DAM at runtime, more memory bandwidth and lower cache efficiency (-)
ARM Case Study

• ARM (Advanced RISC Machine)
  – Started with fixed, 32-bit instruction length
  – Added Thumb instructions
    • A subset of the 32-bit instructions
    • All encoded in 16 bits
    • All translated into equivalent 32-bit instructions within the processor pipeline at runtime
    • Can access only 8 general purpose registers
  – Motivated by many resource constrained embedded applications that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency
How many registers?

All computers have a small set of registers. Memory to hold values that will be used soon. Typical instruction will use 2 or 3 register values.

Advantages of a small number of registers:
- It requires fewer bits to specify which one.
- Less hardware
- Faster access (shorter wires, fewer gates)
- Faster context switch (when all registers need saving)

Advantages of a larger number:
- Fewer loads and stores needed
- Easier to do several operations at once

In 411, “load” means moving data from memory to register, “store” is reverse.
Where do operands reside (when the ALU needs them)?

**Stack machine:**

“Push” loads memory into 1st register (“top of stack”), moves other regs down

“Pop” does the reverse.

“Add” combines contents of first two registers, moves rest up.

**Accumulator machine:**

Only 1 register (called the “accumulator”)

Instruction include “store” and “acc ← acc + mem”

**Register-Memory machine:**

Arithmetic instructions can use data in registers and/or memory

**Load-Store Machine** (aka **Register-Register Machine**):

Arithmetic instructions can only use data in registers.
Comparing the ISA classes

Code sequence for $C = A + B$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load-Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Add C, A, B</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></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>

Java VMs   DSPs      VAX, x86 partially
Comparing the ISA classes

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Reg-Mem</th>
<th>Load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ A = X \times Y + X \times Z \]

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Stack</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C 5</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>temp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Load-store architectures
can do:
  \[
  \begin{align*}
  &\text{add } r1 = r2 + r3 \\
  &\text{load } r3, M(\text{address}) \\
  &\text{store } r1, M(\text{address})
  \end{align*}
\]
⇒ forces heavy dependence on registers, which is exactly what you want in today’s CPUs

can’t do:
  \[
  \begin{align*}
  &\text{add } r1 = r2 + M(\text{address})
  \end{align*}
\]
- more instructions
+ fast implementation (e.g., easy pipelining)
+ easier to keep instruction lengths fixed
Key ISA decisions

**Instruction length**
- are all instructions the same length?

**How many registers?**
- e.g., can you add contents of memory to a register?

**Instruction format**
- which bits designate what??

**Operands**
- how many? how big?
- how are memory addresses computed?

**Operations**
- what operations are provided??
Instruction formats

-what does each bit mean?

Machine needs to determine quickly,

- “This is a 6-byte instruction”
- “Bits 7-11 specify a register”
- ...
- Serial decoding bad

Having many different instruction formats...

- complicates decoding
- uses instruction bits (to specify the format)

What would be a good thing about having many different instruction formats?
LC-3b Instruction Formats

- **ADD, AND (without Immediate)**

![Instruction Format Diagram](Diagram1)

- **ADD, AND (with Immediate), NOT**

![Instruction Format Diagram](Diagram2)
MIPS Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>r format</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
<tr>
<td>i format</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j format</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>

• for instance, “add r1, r2, r3” has
  – OP=0, rs=2, rt=3, rd=1, sa=0, funct=32
  – 0000000 00010 00011 00001 00000 100000

• opcode (OP) tells the machine which format
MIPS ISA Tradeoffs

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
<tr>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>

What if?

- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. Y = AX + B)

Think about how sparsely the bits are used
Conditional branch

• How do you specify the destination of a branch/jump?
Conditional branch

• How do you specify the destination of a branch/jump?

• studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
Conditional branch

• How do you specify the destination of a branch/jump?

• studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
  – we can specify a relative address in much fewer bits than an absolute address
  – e.g., beq $1, $2, 100 => if ($1 == $2) PC = PC + 100 * 4
Conditional branch

• How do you specify the destination of a branch/jump?

• studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else).
  – we can specify a relative address in much fewer bits than an absolute address
  – e.g., beq $1, $2, 100 => if ($1 == $2) PC = PC + 100 * 4

• How do we specify the condition of the branch?
MIPS conditional branches

- `beq, bne`  
  \[ \text{beq } r1, r2, \text{addr} \Rightarrow \text{if } (r1 == r2) \text{ goto addr} \]
- `slt $1, $2, $3`  
  \[ \text{if } ($2 < $3) \text{ } $1 = 1; \text{ else } $1 = 0 \]
- these, combined with $0, can implement all fundamental branch conditions
  
  Always, never, !=, ==, >, <=, >=, <, > (unsigned), <= (unsigned), ...

```plaintext
if (i<j)
    w = w+1;
else
    w = 5;
```

```plaintext
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
Beq $0, $0, L2
L1: add $w, $0, #5
L2:
```
Jumps

• need to be able to jump to an absolute address sometime
• need to be able to do procedure calls and returns

• jump -- j 10000 => PC = 10000
• jump and link -- jal 100000 => $31 = PC + 4; PC = 10000
  -- used for procedure calls
  | OP | target |

• jump register -- jr $31 => PC = $31
  -- used for returns, but can be useful for lots of other things.
Computer Performance
Computer performance: time!, time!, time!

• Response time (latency)
  – How long does it take to execute a job?
  – How long must I wait for the database query?

• Throughput
  – How many jobs can the machine complete in a minute?
  – What is the average execution rate?
  – How much work is getting done?
A Throughput Oriented Approach

• Each car still experience the same latency at the toll

• Many cars are served to improve the number of cars processed per time unit (sec/min/hr)
Case Study

- How can we improve the throughput of I-57 b/w Chicago and Champaign?
  - Increase the number of lanes
- How can we improve the latency of driving from Chicago to Champaign on I-57?
  - Increase the driving speed
Aspects of CPU Performance

<table>
<thead>
<tr>
<th></th>
<th>Inst Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

CPU time = Seconds = Instructions \times Cycles \times Seconds
Definition of Performance

• For some program running on machine X
  – PerformanceX = 1 / Execution timeX
• "X is n times faster than Y“
  – PerformanceX / PerformanceY = n
• Problem:
  – machine A runs a program in 20 seconds
  – machine B runs the same program in 25 seconds
  – A is ________ times faster than B?
How to Improve Performance

\[ \frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} \]

- So, to improve performance (everything else being equal) you can either
  
  _______ the # of required cycles for a program, or
  _______ the clock cycle time or, said another way.
Performance Related Metrics

• An execution of a given program will require
  – some number of instructions (machine instructions)
  – some number of cycles
  – some number of seconds

• We have a vocabulary of metrics that relate these quantities:
  – Cycle time (seconds per cycle)
  – Clock rate (cycles per second)
  – CPI (cycles per instruction)
Instruction Set Metrics Example

• If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will likely be identical during a comparison?
CPI Example

- Suppose we have two implementations of the same instruction set architecture (ISA). For some program, A has a clock cycle time of 10ns and a CPI of 2.5. B has a clock cycle time of 20ns and a CPI of 1.

- What machine is faster for this program, and by how much?
Cycles and CPI Example

• A compiler designer is trying to decide b/w two code sequences for a particular machine. Based on the hardware implementation, there are 3 different classes of instructions: Class A, B, and C, and they require 1, 2, and 3 cycles. The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C. The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C. Which sequence will be faster? How much?
  – First: $2*1 + 1*2 + 2*3 = 10$ cycles
  – Second: $4*1 + 1*2 + 1*3 = 9$ cycles
MIPS example

- Two different compilers are being tested for a 100MHz. Machine with three different classes of instructions: Class A, B, and C, which require 1, 2, and 3 cycles. Both compilers are used to produce code for a large piece of software. The first compiler's code uses 5M Class A instructions, 1M Class B instructions, and 1M Class C instructions. The second compiler's code uses 10M Class A instructions, 1M Class B instructions, and 1M Class C instructions.
MIPS example (cont.)

- Two different compilers are being tested for a 100MHz. Machine with three different classes of instructions: Class A, B, and C, which require 1, 2, and 3 cycles. Both compilers are used to produce code for a large piece of software. The first compiler's code uses 5M Class A instructions, 1M Class B instructions, and 1M Class C instructions. The second compiler's code uses 10M Class A instructions, 1M Class B instructions, and 1M Class C instructions.