Chapter 2: Instructions
How we talk to the computer

Pages not responsible for:
2.7 pg79-89
2.10-2.15 pg 106-134
DO READ 2.16 “Real Stuff”

Notes:

• !!!CHECK CLASS WEBPAGE!!!!!
• Homework due next Tues!
• LectureTA office hours T10-11 Th 10-15
• Extra lab TA hours Friday 1-3pm 2444
• Move my WED office hours next week to Monday 12:30-2pm 4018
• Questions?
• Office Hours - come see me!

The Instruction Set Architecture
The agreed-upon interface between:
the software that runs on a computer and
the hardware that executes it.

The Instruction Execution Cycle

Obtain instruction from program storage
Determine required actions and instruction size
Locate and obtain operand data
Compute result value or status
Deposit results in storage for later use
Determine successor instruction
Key ISA decisions

- Instruction length
  - Are all instructions the same length?
- How many registers?
- Where do operands reside?
  - 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??

Where do operands reside?
(or alternate ISAs)

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

Load-store architectures

can do:

- add r1=r2+r3
- load r3, M(address)
- store r1, M(address)

⇒ forces heavy dependence on registers, which is exactly what you want in today's CPUs

don't do:

- add r1=r2+M(address)

- more instructions
- fast implementation (e.g., easy pipelining)

Where do operands reside?

VAX: register-memory
- Very general. 0, 1, 2, or 3 operands can be in registers

x86: register-memory...
- But floating-point registers are a stack.
- Not as general as VAX instructions

RISC machines:
- Always load-store machines

- Stack, Accumulator machines?
**Comparing the Number of Instructions**

Code sequence for \( C = A + B \times B \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load-Store</th>
</tr>
</thead>
</table>

**Can You Do?**

\( A = X \times Y + X^2 Z \)

---

**Key ISA decisions**

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

- **How many registers?**

- **Where do operands reside?**
  - 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?

Having many different instruction formats...

- Complicates decoding
- Uses instruction bits (to specify the format)

Machine needs to determine quickly,

- "This is a 6-byte instruction"
- "Bits 7-11 specify a register"
- ...

---

**MIPS Instruction Formats**

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For instance, "add $r1, $r2, $r3" has
  - 00000 00010 00011 00001 00000 100000
- Opcode (OP) tells the machine which format
VAX Instruction Formats

1 Byte OP code
specifies number, type and length of operands
Each operand is 1 to many bytes
first byte specifies addressing mode

```
+---------+---------+---------+---------+
| disp r  | autoinc | disp r  | autoinc |
| register| disp    | register| disp    |
| A r     | byte    | A r     | byte    |
| C r half word |       | C r half word |       |
| index A r m l displacement |
```

"move" can be load, store, mem copy, jump, depending on operands

How Many Operands?

- Two-address code: target is same as one operand
  - E.g. \( x = x + y \)
- Three-address code: target can be different
  - E.g. \( x = y + z \)
  - x86 doesn't have three-address instructions; others do
- Some operands are also specified implicitly
  - "condition code" setting shows if result was +, 0, or -
  - PowerPC's "branch on count" uses special "count register"
- Well-known ISA's have 0-4 (explicit) operands
  - PowerPC has "float multiply add", \( r = x + y^z \)

Addressing Modes

**how do we specify the operand we want?**

#25

The operand (25) is part of the instruction

R3
(sometimes written $3)
The operand is the contents of register 3

\[ M[R3] \]

Use contents of R3 as address into memory; find the operand there. This is a special case of...

\[ M[R3 + 160] \]

Add the displacement (160) to contents of R3, look in that memory location for the operand.
- If register is PC, this is "PC-relative addressing"

All our example ISA's have the above modes

More Addressing Modes

(Not included in MIPS ISA)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base+Index</td>
<td>M[R3 + R4]</td>
</tr>
<tr>
<td></td>
<td>Add contents of R3 and R4 to get memory address</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>M[R3++] (or M[R3+d])</td>
</tr>
<tr>
<td></td>
<td>Add value in memory location designated by R3, but also increment R3</td>
</tr>
<tr>
<td></td>
<td>- Useful for accessing array elements</td>
</tr>
<tr>
<td></td>
<td>- Autodecrement is similar</td>
</tr>
<tr>
<td>Scaled Index</td>
<td>M[R3 + R4*d]</td>
</tr>
<tr>
<td></td>
<td>(VAX and x86)</td>
</tr>
<tr>
<td></td>
<td>Multiply R4 by d (d is typically 1, 2, 4, or 8), then add R3 to get memory address</td>
</tr>
<tr>
<td>Memory Direct</td>
<td>M[10000]</td>
</tr>
<tr>
<td>Memory Indirect</td>
<td>M[M[R3]]</td>
</tr>
<tr>
<td>(only VAX)</td>
<td>Find number in memory location R3, use THAT as address into memory to find</td>
</tr>
</tbody>
</table>
**VAX addressing mode usage**

- Half of all references were register-mode
- Remaining half distributed as follows:

<table>
<thead>
<tr>
<th>Program</th>
<th>Base + Displacement</th>
<th>Immediate</th>
<th>Scaled Index</th>
<th>Memory Indirect</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEX</td>
<td>56%</td>
<td>43%</td>
<td>0</td>
<td>1%</td>
<td>0</td>
</tr>
<tr>
<td>Spice</td>
<td>58%</td>
<td>17%</td>
<td>16%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>GCC</td>
<td>51%</td>
<td>39%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>

- Similar measurements show that 16 bits is enough for the immediate field 75% to 80% of the time.
- And 16 bits is enough for displacement 99% of the time.

---

**MIPS addressing modes**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$1, $2, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Immediate

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$1, $2, 35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Immediate

**MIPS ISA decisions**

- Instruction length
  - All instructions are 32 bits long (one word)

- How many registers?
  - 32 general purpose registers (R0 always 0)
  - 2 special purpose for multiply and divide

- Where do operands reside?
  - Load-store architecture.

- Instruction formats
  - Three (r-, i-, and j-format).

- Operands
  - 3-address code.
  - Immediate, register, and base-displacement modes.

---

**Intel x86 evolution**

- 1978: Intel 8086 announced (16-bit architecture)
- 1980: The 8087 floating point coprocessor added
- 1982: The 80286 - more ops, 24-bit address space
- 1985: The 80386 - 32-bit address space + new modes
- 1989-1995: The 80486, Pentium, and Pentium Pro add a few instructions
- 1997: MMX is added (Pentium II is P. Pro + MMX)
- 1999 Pentium III (same architecture)
- 2000 Pentium 4 (144 new multimedia instructions)
- 2001 Itanium - new ISA (still can execute x86 code)
Anthropology of ISA’s

- VAX design goal was small code size and simple compilation (since all combinations of addressing modes were possible).
  - Met goals, but cheap & fast memories and better compilers made goals less important.
  - But couldn’t pipeline well. Replaced by Alpha (a RISC).
- x86’s goal was to get to market quickly.
  - Ugly, but most common instructions can be implemented relatively efficiently.
- MIPS’ goal: simplicity. Reflects university heritage.
- PowerPC: super-RISC. Compiler group influence.

Recent developments

- VLIW - Very Long Instruction Word
  - 1 “packet” has multiple instructions
  - Tera MTA has 26, 21, and 14 bit-long RISC operations (plus 3 “lookahead” bits) in 64 bits
  - Intel Itanium has three 41-bit RISC ops (plus 5 “type” bits) in 128-bit packet
- JVM (Java Virtual Machine)
  - A new level of abstraction
    - Between Java language and ISA
  - Stack based - is this a good choice??

Break - 3 minutes

- Find a neighbor (or 2)
- Tell them what you think the point of learning about all the components of the ISA were (each 1 minute)
  - Can you remember the 6 ISA-related issues we looked at?

MIPS ISA Tradeoffs

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

What if?

- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. Y = X + AB)
I Type: Conditional Branches and J Type: Unconditional Jumps

- How do you specify the destination of a branch/jump?
  - Studies show that almost all conditional branches go short distances from current PC
  - What program constructs cause conditional branches?
  - Specify a ____________ address in fewer bits than an ________________ address

- What about the branch condition?

But MIPS only supports beq/bne!

- What if I want to emit code for
  - if ($R2 < $R3)

- Set less than instruction (slt)
  - Slit $1, $2, $3
  - If ($2 < $3)
    - $1 = 1
  - Else
    - $1 = 0

- So with these 3 insts we can implement all types of conditional branches
  - ==, !=, <, <=, >, >= (unsigned), etc.

Jumps

- Not all changes in PC will be "close" to the current PC.
  - When?

  - Answer: jump to an ABSOLUTE address
  - Jump - j 10000
  - Jump and link - jal 10000
    - Used for procedure calls

  - Jump registers (R type) -- in book
    - Used for returns
Branch and Jump Addressing Modes

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, target is PC-displacement mode.
- If opcode is 6 bits, how many bits are available for displacement? How far can you jump?
**Key Points**

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count.
- Four principles of IS architecture:
  - regularity produces simplicity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

**Time versus throughput**

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Time to Buy Area</th>
<th>Speed</th>
<th>Passengers</th>
<th>(pm/h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ferrari</td>
<td>3.1 hours</td>
<td>160 mph</td>
<td>2</td>
<td>320</td>
</tr>
<tr>
<td>Greyhound</td>
<td>7.7 hours</td>
<td>65 mph</td>
<td>60</td>
<td>3900</td>
</tr>
</tbody>
</table>

- Time to do the task from start to finish
- execution time, response time.
- Tasks per unit time
- throughput.
Time versus throughput

- Execution Time is measured in time units/job.
  - For a SINGLE PROGRAM to execute on a system, usually in a dedicated environment
- Throughput is measured in jobs/time unit.
  - Total amount of work (multiple jobs) done by a computer for a given amount of time.
- But "time = 1/throughput" may be false.
  - It takes 4 months to grow a tomato.
  - Can you only grow 3 tomatoes a year??
That's half, the other half...

CPU Time = \#CPU cycles executed \times Cycle time

\#CPU cycles = \text{Instructions executed} \times CPI

\text{Average Clock Cycles per Instruction}

Different codes compile into different numbers of instructions.

\begin{align*}
\text{COMP501: iteration for loop} & \quad \text{Windows OS} \\
5000 & \quad 5 \text{ billion}
\end{align*}

Each computer design takes a certain amount of time to execute an “average” instruction.

Putting it all together

One of P&H’s “big pictures”

Note: average CPI is somewhat artificial

(it’s computed from the other numbers using this formula)

but it’s an intuitive and useful concept

Note: Use dynamic instruction count (\#instructions executed), not static (\#instructions in compiled code)

Think for a minute

• “Insanity is doing the same thing and expecting a different result”
  - Albert Einstein

Dynamic Instruction Count versus Static Instruction Count

\begin{align*}
\text{int } x = 10; \\
\text{for (int } j = 0; j < x; j++) \\
\{ \\
\quad c[j] = a[j] + b[j]; \\
\}\n\end{align*}

• Static instruction count is determined by the code and the compiler

• Dynamic instruction count is determined by the “choices” made in the execution of the code

  - A video game doesn’t have the same execution time each run...
### Explaining Execution Time Variation

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same machine, different programs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same program, different machines, but same ISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same program, different ISA’s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU Execution Time = Instruction Count × CPI × Clock Cycle Time