Lecture 3
Machine Language

Speaking computer before voice recognition interfaces

Instructions:
- Language of the Machine
- More primitive than higher level languages
e.g., no sophisticated control flow
- Very restrictive
e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time

Instruction Execution cycle

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

- operations: how many?
- which ones
- operands: how many?
- location
- types
- how to specify?
- instruction format: size
- how many formats?

\[ y = x + b \]

In the end, we must have:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Destination</th>
<th>Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>011010010</td>
<td>000010010</td>
<td>1010.01010010</td>
</tr>
</tbody>
</table>

What an architect can change:
- type of operations: add, branch, find roots of \( ax^2 + bx + c \)
- sources and destinations: memory, registers, I/O
- the way we see memory: program and data or separated

To understand these trade-offs, let us make some designs!

Crafting an ISA-I

Suppose: memory is very slow, 100 X slower than CPU, and also, REALLY expensive. What would you do:

a) code the instructions in a simple way, so that decoding them in the CPU takes less hardware;
b) code instructions so that they use less memory.

Answer (b) seems to be in the good direction!
Crafting an ISA-II

And what seems more reasonable:
- have few instructions, so that there are not too many options to the programmer, and so he/she won't get confused;
- have many complex instructions, so that the CPU spends less time going to memory and more time all by itself.

And again, what is more reasonable:
- have simple and few instructions, so that the assembly programmer will have few options to mistake, and the memory will be accessed many times;
- have complex instructions, so that the assembly programmer, just by calling the instruction, can solve a complex task (polynomial foot or copy a whole array).

If you always chose (b), you're a great architecture designer of the 60s.

Remember

- An architect should ALWAYS have an eye on the available technology!
  - When silicon memory came, the time difference was reduced, and also the price. It's time for a change!
  - Going to memory is easier, but there is still a large difference in speed;
  - memory is cheaper, can have lots of it!
  - Compiler can not use complex instructions.
  - People are less and less programming using assembly (only time critical code or bad management)

People in the 60s-80s were not dummies, they had a different technological environment!

Crafting an ISA-III

- We'll look at some of the decisions facing an instruction set architect, and
- how those decisions were made in the design of the MIPS instruction set.
- MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
  - fixed instruction length
  - few instruction formats
  - load/store architecture
Crafting an ISA-IV

- RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism (more on this in this course).
- Changes on technology that allowed RISCs:
  - memory speed and cost
  - compiler technology increase
- New changes to come:
  - many transistors more (increase cache, branch prediction, floating point, etc);
  - low power applications: back to complex coding?
  - extensive use of dynamic RAM: IRAM chips?
  - reconfigurable devices?
  - Java machines?

Crafting an ISA-V

- For the moment, let’s focus on a simple machine, simple ISA, the MIPS
- More on technology dependencies and design decisions during the course.

How Many Operands?

- Most instructions have three operands
  - (e.g., \( z = x + y \)).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicitly.
## How Many Operands? (basic ISA classes)

<table>
<thead>
<tr>
<th>Category</th>
<th>Address Size</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>1 address</td>
<td>add A</td>
<td>acc = acc + mem[A]</td>
</tr>
<tr>
<td>Accumulator</td>
<td>5 address</td>
<td>add</td>
<td>mem = mem + int</td>
</tr>
<tr>
<td>Register</td>
<td>2 address</td>
<td>add A B</td>
<td>acc = A + B (addressing might be more complex)</td>
</tr>
<tr>
<td>Load/Store</td>
<td>3 address</td>
<td>add A B C</td>
<td>B + C</td>
</tr>
<tr>
<td>Hybrid</td>
<td>3 address</td>
<td>add B C D</td>
<td>B + C + D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store R A</td>
<td>mem[R] = R</td>
</tr>
</tbody>
</table>

## Comparing the Number of Instructions

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td>Load R1A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2B</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>Store C,R3</td>
<td></td>
</tr>
</tbody>
</table>

Watch! For a sequence of operations, who is favored? Try: \( a^2 + b^2 + c^2 \); again with only 2 registers.

## Instruction Length

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Fixed</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Length

- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.
- Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism (believe me by now; see for yourself in the lab!).

All MIPS instructions are 32 bits long. This decision impacts every other ISA decision we make because it makes instruction bits scarce.

Instruction Formats: the role of each bit

- Having many different instruction formats...
  - complicates decoding
  - uses instruction bits (to specify the format)

VAX 11 instruction format

MIPS Instruction Formats

- the opcode tells the machine which format
- so add r1, r2, r3 has
  - opcode=0, rs=2, rt=3, rd=r1, sa=0, funct=32,
  - 000000 00010 00011 00001 00000 100000
Accessing the Operands

- Operands are generally in one of two places:
  - Registers (32 Int, 32 FP)
  - Memory (2^32 locations)
- Registers are:
  - Easy to specify
  - Close to the processor (fast access)
- The idea that we want to access registers whenever possible led to load-store architectures.
  - Normal arithmetic instructions only access registers
  - Only access memory with explicit loads and stores

Load-store architectures

- Can do:
  - Add: r1 = r2 + r3
  - Load: r3, M(address)
- Can't do:
  - Add: r1 = r2 + M(address)

⇒ Forces heavy dependence on registers, which is exactly what you want in today's CPUs
- More instructions (bad implementation)
  - Less predictable

A historical note

- Once upon a time, a big company invented a new microprocessor. It had NO registers at all, all operations were memory mapped.
- Since memory was slow, the microprocessor never sold very much, because the overall system performance was low.
- Although heads rolled, some engineers claimed that the memory would become faster, and a whole set of problems would become easier.
- The company used the same design strategy, applied to problems that required too many memory accesses and designed a processor tuned to a specific field: Digital Signal Processing. They made a fortune!
MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]
MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

C code:

\[ A = B + C + D; \]
\[ E = F - A; \]

MIPS code:

- `add $t0, $s1, $s2`
- `add $s0, $t0, $s3`
- `sub $s4, $s5, $s0`

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

Registers vs. Memory

- Arithmetic instructions operands must be registers, only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables
So, how do we get the operands?

Addressing modes!

- Register direct: R3
- Immediate (literal): $25
- Direct (absolute): M[10000]
- Register indirect: M[R3]
- Base+Displacement: M[R3 + 10000]
  - If register is the program counter, this is PC-relative
  - Base+Index: M[R3 + R4]
  - Scaled Index: M[R3 + R4*d + 10000]
  - Autoincrement: M[R3 + 1]
  - Autodecrement: M[R3 - 1]
- Memory Indirect: M[M[R3]]

MIPS addressing modes

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

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

<table>
<thead>
<tr>
<th>base + displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $1, disp($2)</td>
</tr>
</tbody>
</table>

Is this sufficient?

- Measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
- Similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time;
- and that 16 bits is enough of a displacement 99% of the time.
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>...</td>
</tr>
</tbody>
</table>

Memory Organization

- Bytes are nice, but most data items use larger "words" (remember, memory access is still an issue).
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>8 bits of data</td>
<td>Registers hold 32 bits of data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 2⁰ bytes with byte addresses from 0 to 2⁰-1
- 2⁴ words with byte addresses 0, 4, 8, ... 2⁴-4
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

What have we got so far for the MIPS ISA:

- Fixed 32-bit instructions
- 3 instruction formats
- 3-operand, load-store architecture
- 32 general purpose registers (R0=0), all 32-bits wide (word)
- register, immediate, and base-displacement addressing modes

Let’s see some actual instructions!
Instructions

- Load and store instructions
- Example (A = array 100 words, base address is in $s3):
  
  C code:
  \[ A[12] = h + A[8]; \]
  
  MIPS code:
  \[
  \begin{align*}
  &\text{lw}\ $t0, 32(\$s3) \\
  &\text{add}\ $t0, $s2, $t0 \\
  &\text{sw}\ $t0, 48(\$s3)
  \end{align*}
  \]
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

More lw and sw

\[
\begin{align*}
G &= h + A[i] \\
A &= \text{array 100 elements; base=}$s3 \\
G &= $s1 \\
h &= $s2 \\
I &= $s4 \\
\text{lw}\ $t0, 0(\$t1) # \text{offset of } A[i] \\
\text{add}\ $t1, $s4, $s4 # \text{temp reg } \times 2\text{'}
\end{align*}
\]

Notice:
- Offset + base addressing is good for structures:
  - one register points to the base, the offset can select the desired element.

Exercise:

- \( G[j] = h + A[i] \):
- \( G \) and \( A \) have 100 positions; use $s10-9 as scratch registers; base address of \( G \) is in $s3, of \( A \) in $s4.
**Remember:**

- **MIPS**
  - loading words but addressing bytes
  - arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>li $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s2, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>