CS 240 Administirivia, Sept. 9

- Everything is on the website: [http://cs.wellesley.edu/~cs240/](http://cs.wellesley.edu/~cs240/)
  - PS1 due Friday – how’s it going?
  - Readings – getting more relevant
  - Google Group: ask questions, link for anonymous feedback
- Double-check your textbook:
  - Computer Organization and Design: The Hardware/Software Interface
  - NOT: Computer Architecture: A Quantitative Approach, same authors.
  - Abacus, not columns.
- MIPS card in front cover
- Office Hours:
  - Monday 4-6, Tuesday 9-10, 3-4, Thursday 4-5:30, Friday 11:30-12:30
- Names
- What you do

Today

- MIPS instruction set (part 1)
  - Design principles
  - Arithmetic
  - Data transfer (memory access)
  - Translation from Java/C
  - Immediate operands
- Bitwise boolean algebra and logical operators
  - Java
  - MIPS
  - Bit vector manipulations

- MIPS instruction set (part 1)
- Bitwise boolean algebra and logical operators

- Everything is on the website: [http://cs.wellesley.edu/~cs240/](http://cs.wellesley.edu/~cs240/)
- Double-check your textbook:
  - Computer Organization and Design: The Hardware/Software Interface
  - NOT: Computer Architecture: A Quantitative Approach, same authors.
  - Abacus, not columns.
- MIPS card in front cover
- Office Hours:
  - Monday 4-6, Tuesday 9-10, 3-4, Thursday 4-5:30, Friday 11:30-12:30
- Names
- What you do
Instruction Set Architecture

- The ISA defines:
  - The system state (registers, memory, program counter, etc.)
  - The instructions the CPU can execute
  - The effect that each instruction has on the system state

- MIPS Memory

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v0</td>
<td></td>
</tr>
<tr>
<td>$s0</td>
<td></td>
</tr>
<tr>
<td>$a0</td>
<td></td>
</tr>
<tr>
<td>$t0</td>
<td></td>
</tr>
<tr>
<td>$t1</td>
<td></td>
</tr>
<tr>
<td>$t2</td>
<td></td>
</tr>
<tr>
<td>$s1</td>
<td></td>
</tr>
</tbody>
</table>

General ISA Design Decisions

- Registers
  - How many registers are there?
  - How wide are they?

- Memory
  - How do you specify a memory location?

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded?

MIPS

- Early 1980s: MIPS designed by John Hennessy et al. at Stanford.
- 1984: MIPS Computer Systems. (Later MIPS Technologies)
- 1996: Nintendo 64 – most famous MIPS processor?
- 2013: Sold to Imagination Technologies.

- Reduced Instruction Set Computer (RISC)
  - vs. Complex Instruction Set Computer (CISC)

Three Basic Kinds of Instructions

- Perform arithmetic or logic on register data. (Today, Friday)
  - $a = b + c$
  - $x = y << z$
  - $i = j \& k$

- Transfer data between memory and register. (Today)
  - Load data from memory into register
  - $\text{Register} = \text{Memory}[\text{address}]$
  - Store data from register into memory
  - $\text{Memory}[\text{address}] = \text{Register}$

- Control what instruction is executed next. (Next week)
  - Unconditional jumps
  - Conditional branches
One general instruction format:

- `<instruction> <operand>, <operand>, <operand>`

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1$, $s2$, $s3$</td>
<td>$s1 = s2 + s3$</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1$, $s2$, $s3$</td>
<td>$s1 = s2 - s3$</td>
</tr>
</tbody>
</table>

- Design principle: Simplicity favors regularity.
  - Many instructions naturally require three operands.
  - Variable numbers of operands necessitate more complex hardware.
- Why is there no `negate` instruction?

Design principle: Smaller is faster.

- How do we apply this principle in modern computer systems?

 registers!

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>a0</td>
<td>$s0-3$</td>
<td>2-3 values for results and expression evaluation</td>
</tr>
<tr>
<td>a1</td>
<td>$s4-6$</td>
<td>4-7 arguments</td>
</tr>
<tr>
<td>a2</td>
<td>$s7$</td>
<td>8-15 temporaries</td>
</tr>
<tr>
<td>a3</td>
<td>$s8-15$</td>
<td>16-23 saved</td>
</tr>
<tr>
<td>t0</td>
<td>$s16-19$</td>
<td>24-25 more temporaries</td>
</tr>
<tr>
<td>fp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

We will talk about the usage column when we cover procedures.

Translate to MIPS

```java
int f(int a0, int a1, int a2, int a3) {
    ...
    int v0 = -(a0 + a1 - (a2 + a3));
    ...
}
```

How?

Two steps:
1. Break into several simple Java/C statements, each corresponding to a single MIPS instruction. Introduce temporary local variables as needed.
2. Translate to MIPS instructions, representing temporary local variables with registers in $s0-$s7. Local variable $v0$ is represented by register $s0$, $a0$ by $s0$, etc.
Data transfer instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
</tbody>
</table>

Explored yesterday in lab...

Load word

```
lw $s0, 16($s1)
```

Java/C-ish description

```
$s0 = M[$s1+16]
```

Destination register

Immediate offset

Source register

Store word

```
sw $s0, 16($s1)
```

```
M[$s1+16] = $s0
```

Helpful pseudo-instructions

Load immediate

```
li $s0, 240
```

```
$s0 = 240
```

Java/C-ish description

Load address

```
la $s0, count
```

```
$s0 = &count
```

Java/C-ish description

Move

```
move $s0, $s1
```

```
$s0 = $s1
```

Translate \( x = x + x; \) to MIPS

\( x \) is a static (global) variable stored in memory:

```
data
x: .word 0x00000003
```

Two steps:

1. Break into several simple Java/C statements, each corresponding to a single MIPS instruction. Introduce temporary local variables as needed.
2. Translate to MIPS instructions, representing temporary local variables with registers.
Translate $x = x + x$;

$\text{MIPS Memory}$

$\begin{array}{|c|c|c|c|c|c|}
\hline
& 2^32 - 1 & & & & \\
\hline
\text{A} & +8 & & & & \\
\text{A} & +4 & & & & \\
\text{A} & 0 & & & & \\
\hline \end{array}$

int[] fibs = new int[6];

$\text{MIPS Memory}$

$\begin{array}{|c|c|c|c|c|c|}
\hline
& 2^32 - 1 & & & & \\
\hline
\text{A} & +24 & & & & \\
\text{A} & +20 & & & & \\
\text{A} & +16 & & & & \\
\hline \end{array}$

Array Representation

- (The basics - minor additions/differences for full details of Java or C)
- Store in registers or memory? How? \textit{(draw)}
  - Where is the base/start?
  - Where is end? (How big is the array?)
  - Where is index $i$?
  - Why is zero-indexing such a good idea?

int[] fibs = new int[6];
Translation

Translate the following code to MIPS assembly code, assuming:
• $a0 represents the local variable \texttt{fibs} and holds the address of a 6-element array of type \texttt{int}.
\[
\]

Two steps:
1. Break into several simple Java/C statements, each corresponding to single MIPS instruction. Introduce temporary local variables as needed.
2. Translate to MIPS instructions, representing temporary local variables with registers.

Design principle: Make the common case fast.

```java
for (int i = 0; i < 1000; i++) {
    ...
}
```

```mips

```

Why no \texttt{subi}? (RISC!)
How could we implement \( li, la \)?

Logical operations

**Boolean Algebra**

- George Boole, 19th Century
- Algebraic representation of logic
- Encode boolean as bit: True = 1, False = 0
- AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
- OR: \( A \| B = 1 \) when either \( A \) is 1 or \( B \) is 1
- XOR: \( A \oplus B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
- NOT: \( \sim A = 1 \) when \( A = 0 \) and vice-versa
- DeMorgan’s Law: \( \sim (A \| B) = \sim A \& \sim B \)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**General Boolean Algebras**

Operate bitwise on bit vectors

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0101001</th>
<th>01101001</th>
<th>01101001</th>
<th>~ 0101001</th>
</tr>
</thead>
</table>

\[ A \& 0101001 \]
\[ \| 0101001 \]
\[ ^ 0101001 \]
\[ \~ 0101001 \]

All of the properties of Boolean algebra apply

\[ \forall a \in A, a \oplus 0 = 0101001 \]

How does this relate to set operations?
Sets as bit vectors

Representation
A \( w \)-bit vector represents subsets of \( \{0, ..., w-1\} \)
\( a_j = 1 \) if and only if \( j \in A \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>(0, 3, 5, 6)</td>
<td></td>
</tr>
<tr>
<td>01010101</td>
<td>(0, 2, 4, 6)</td>
<td></td>
</tr>
</tbody>
</table>

Operations
- \& Intersection
- | Union
- ^ Symmetric difference
- ~ Complement

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01000001</td>
<td>{0, 6}</td>
<td></td>
</tr>
<tr>
<td>01111101</td>
<td>{0, 2, 3, 4, 5, 6}</td>
<td></td>
</tr>
<tr>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
<td></td>
</tr>
<tr>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
<td></td>
</tr>
</tbody>
</table>

Bit-Level Operations in Java

- \& | ^ ~
  - Apply to any "integral" data type
  - long, int, short, byte, also char
  - View arguments as bit vectors
- Examples (byte data type)
  - \(~0x41\)
  - \(~0x00\)
  - \(0x69 \& 0x55\)
  - \(0x69 \mid 0x55\)
- Bit-manipulation puzzles in PS 2

Contrast: Java boolean logic operations

- Contrast to logical operators
  - \&\& \|\| \!
  - Applies to arguments of type boolean only
  - Early termination: a.k.a. short-circuit evaluation

I've shifted <<, >>, >>> to the next lecture.