Lecture 4: MIPS Instruction Set

• No class on Tuesday

• Today’s topic:
  - MIPS instructions
  - Code examples
Instruction Set

• Understanding the language of the hardware is key to understanding the hardware/software interface

• A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently

• Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)

• What are important design principles when defining the instruction set architecture (ISA)?
Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

We will later discuss RISC vs CISC
A Basic MIPS Instruction

C code:  
\[ a = b + c; \]

Assembly code: (human-friendly machine instructions)  
\[
\text{add } a, b, c \quad \# \text{ a is the sum of b and c}
\]

Machine code: (hardware-friendly machine instructions)  
000000100011001001000000000100000

Translate the following C code into assembly code:  
\[ a = b + c + d + e; \]
Example

C code    a = b + c + d + e;
translates into the following assembly code:

add a, b, c
add a, a, d or add f, d, e
add a, a, e

• Instructions are simple: fixed number of operands (unlike C)
• A single line of C code is converted into multiple lines of assembly code
• Some sequences are better than others… the second sequence needs one more (temporary) variable f
Subtract Example

C code \[ f = (g + h) - (i + j); \]

Assembly code translation with only add and sub instructions:
Subtract Example

C code    \( f = (g + h) - (i + j); \)
translates into the following assembly code:

\[
\begin{align*}
&\text{add } t0, g, h & \text{add } f, g, h \\
&\text{add } t1, i, j & \text{or } \text{sub } f, f, i \\
&\text{sub } f, t0, t1 & \text{sub } f, f, j
\end{align*}
\]

• Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later
Operands

• In C, each “variable” is a location in memory

• In hardware, each memory access is expensive – if variable \( a \) is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)

• To simplify the instructions, we require that each instruction (add, sub) only operate on registers

• Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so many scratchpad registers
Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?

- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)

- A 32-bit entity (4 bytes) is referred to as a word

- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw  $t0, memory-address

Store word
sw  $t0, memory-address

How is memory-address determined?
The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions.
Immediate Operands

• An instruction may require a constant as input

• An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

• Putting a constant in a register requires addition to register $zero (a special register that always has zero in it) -- since every instruction requires at least one operand to be a register

• For example, putting the constant 1000 into a register:

  addi $s0, $zero, 1000
Memory Instruction Format

• The format of a load instruction:

```
lw $t0, 8($t3)
```

- destination register
- source address
- any register
- a constant that is added to the register in brackets
Memory Instruction Format

• The format of a store instruction:

```
sw $t0, 8($t3)
```

- **source register**
- **source address**
- **any register**
- a constant that is added to the register in brackets
Example

```c
int a, b, c, d[10];
```

```assembly
addi $t0, $zero, 1000   # assume that data is stored at
                       # base address 1000; placed in $t0;
                       # $zero is a register that always
                       # equals zero
lw $s1, 0($t0)          # brings value of a into register $s1
lw $s2, 4($t0)          # brings value of b into register $s2
lw $s3, 8($t0)          # brings value of c into register $s3
lw $s4, 12($t0)         # brings value of d[0] into register $s4
lw $s5, 16($t0)         # brings value of d[1] into register $s5
```
Example

Convert to assembly:

Example

Convert to assembly:


Assembly (same assumptions as previous example):
\[
\begin{align*}
\text{lw} & \quad \$s0, 0($t0) \quad \# \ a \text{ is brought into } \$s0 \\
\text{lw} & \quad \$s1, 20($t0) \quad \# \ d[2] \text{ is brought into } \$s1 \\
\text{add} & \quad \$t1, \$s0, \$s1 \quad \# \ \text{the sum is in } \$t1 \\
\text{sw} & \quad \$t1, 24($t0) \quad \# \ \$t1 \text{ is stored into } d[3]
\end{align*}
\]

Assembly version of the code continues to expand!
Memory Organization

• The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
• $gp points to area in memory that saves global variables
• Dynamically allocated storage (with malloc()) is placed on the heap
Recap – Numeric Representations

- Decimal: \(35_{10} = 3 \times 10^1 + 5 \times 10^0\)

- Binary: \(00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0\)

- Hexadecimal (compact representation): \(0x\ 23 \quad \text{or} \quad 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0\)

0-15 (decimal) \(\rightarrow\) 0-9, a-f (hex)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>00</td>
<td>4</td>
<td>0100</td>
<td>04</td>
<td>8</td>
<td>1000</td>
<td>08</td>
<td>12</td>
<td>1100</td>
<td>0c</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>01</td>
<td>5</td>
<td>0101</td>
<td>05</td>
<td>9</td>
<td>1001</td>
<td>09</td>
<td>13</td>
<td>1101</td>
<td>0d</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>02</td>
<td>6</td>
<td>0110</td>
<td>06</td>
<td>10</td>
<td>1010</td>
<td>0a</td>
<td>14</td>
<td>1110</td>
<td>0e</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>03</td>
<td>7</td>
<td>0111</td>
<td>07</td>
<td>11</td>
<td>1011</td>
<td>0b</td>
<td>15</td>
<td>1111</td>
<td>0f</td>
</tr>
</tbody>
</table>
Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

**R-type instruction**

```
add    $t0, $s1, $s2
```

<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>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

**I-type instruction**

```
lw     $t0, 32($s3)
```

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>constant</td>
</tr>
</tbody>
</table>
# Logical Operations

<table>
<thead>
<tr>
<th>Logical ops</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift Right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>
Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2:
  \[\text{beq register1, register2, L1}\]
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  \[\text{j L1}\]
  \[\text{jr }$s0\]
  (useful for large case statements and big jumps)

Convert to assembly:
\[
\text{if } (i == j) \\
  \text{f = g+h;}
\text{else} \\
  \text{f = g-h;}
\]
Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: \[ \text{beq register1, register2, L1} \]
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  \[ \text{j L1} \]
  \[ \text{jr $s0} \] (useful for large case statements and big jumps)

Convert to assembly:

- if (i == j)
  \[ f = g+h; \]
  \[ \text{else} \]
  \[ f = g-h; \]
- \[ \text{bne $s3, $s4, Else} \]
- \[ \text{add $s0, $s1, $s2} \]
- \[ \text{j Exit} \]
- \[ \text{Else: sub $s0, $s1, $s2} \]
- \[ \text{Exit:} \]
Title

• Bullet