Lecture 4: MIPS Instruction Set

• Today’s topic:
  ▪ More MIPS instructions for math and control
  ▪ Code examples
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:

  \texttt{addi \$s0, \$zero, 1000}
**Example**

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

addi $s0, $zero, 1000   # the program has base address
#  1000 and this is saved in $s0
# $zero is a register that always
# equals zero
addi $s1, $s0, 0          # this is the address of variable a
addi $s2, $s0, 4          # this is the address of variable b
addi $s3, $s0, 8          # this is the address of variable c
addi $s4, $s0, 12        # this is the address of variable d[0]
```
Memory Instruction Format

• The format of a load instruction:

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

destination register

source address

lw $t0, 8($t3)

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

Convert to assembly:

Example

Convert to assembly:


Assembly: # addi instructions as before
          lw $t0, 8($s4)  # \textit{d}[2] is brought into $t0
          lw $t1, 0($s1)  # \textit{a} is brought into $t1
          add $t0, $t0, $t1  # the sum is in $t0
          sw $t0, 12($s4)  # $t0 is stored into \textit{d}[3]

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
Another Version

Convert to assembly:

C code: 

Assembly:

\[
\begin{align*}
\text{lw} & \quad \$t0, 20($gp) & \# & \text{d}[2] \text{ is brought into } \$t0 \\
\text{lw} & \quad \$t1, 0($gp) & \# & \text{a is brought into } \$t1 \\
\text{add} & \quad \$t0, \$t0, \$t1 & \# & \text{the sum is in } \$t0 \\
\text{sw} & \quad \$t0, 24($gp) & \# & \$t0 \text{ is stored into } d[3]
\end{align*}
\]
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)**
  \[0x 23 \text{ or } 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0\]

0-15 (decimal) → 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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>1100</td>
<td>0c</td>
<td>13</td>
<td>1101</td>
<td>0d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>1110</td>
<td>0e</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

<table>
<thead>
<tr>
<th>R-type instruction</th>
<th>add</th>
<th>$t0, $s1, $s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 10001 10010 01000 00000 100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits 5 bits 5 bits 5 bits 5 bits 6 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op rs rt rd shamt funct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode source source dest shift amt function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I-type instruction</th>
<th>lw</th>
<th>$t0, 32($s3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits 5 bits 5 bits 16 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode rs rt constant</td>
<td></td>
<td></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><code>&lt;&lt;</code></td>
<td><code>&lt;&lt;</code></td>
<td><code>sll</code></td>
</tr>
<tr>
<td>Shift Right</td>
<td><code>&gt;&gt;</code></td>
<td><code>&gt;&gt;&gt;</code></td>
<td><code>srl</code></td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td><code>&amp;</code></td>
<td><code>&amp;</code></td>
<td><code>and, andi</code></td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td>`</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td><code>~</code></td>
<td><code>~</code></td>
<td><code>nor</code></td>
</tr>
</tbody>
</table>
Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2:  
  ```
  beq register1, register2, L1
  ```
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  ```
  j L1
  jr $s0  (useful for large case statements and big jumps)
  ```

Convert to assembly:
```java
if (i == j)
    f = g+h;
else
    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;
else  
  f = g-h;

bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit

Else:  
  sub $s0, $s1, $s2
Exit:
Example

Convert to assembly:

```assembly
while (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6
Example

Convert to assembly:

```assembly
while (save[i] == k)
    i += 1;

i and k are in $s3 and $s5 and base of array save[] is in $s6
```

Loop:

```
sll      $t1, $s3, 2
add      $t1, $t1, $s6
lw       $t0, 0($t1)
bne      $t0, $s5, Exit
addi     $s3, $s3, 1
```

Exit:

```
j         Loop
```

```
Registers

• The 32 MIPS registers are partitioned as follows:
  
  - Register 0 : $zero        always stores the constant 0
  - Regs 2-3 : $v0, $v1       return values of a procedure
  - Regs 4-7 : $a0-$a3        input arguments to a procedure
  - Regs 8-15 : $t0-$t7       temporaries
  - Regs 16-23: $s0-$s7       variables
  - Regs 24-25: $t8-$t9       more temporaries
  - Reg  28 : $gp             global pointer
  - Reg  29 : $sp             stack pointer
  - Reg  30 : $fp             frame pointer
  - Reg  31 : $ra             return address
Title

• Bullet