Lecture 5: More Instructions, Procedure Calls

• Today’s topics:
  ▪ Numbers, control instructions
  ▪ Procedure calls
Example

Convert to assembly:


Assembly (same assumptions as previous example):

\[
\begin{align*}
\text{lw} & \quad $s0, 0($gp) \quad \# \ a \text{ is brought into } $s0 \\
\text{lw} & \quad $s1, 20($gp) \quad \# \ d[2] \text{ is brought into } $s1 \\
\text{add} & \quad $s2, $s0, $s1 \quad \# \ \text{the sum is in } $s2 \\
\text{sw} & \quad $s2, 24($gp) \quad \# \ $s2 \text{ is stored into } d[3]
\end{align*}
\]

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)
  \(0x\ 23\ or\ 23_{hex} = 2 \times 16^1 + 3 \times 16^0\)

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>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Source</th>
<th>Source</th>
<th>Dest</th>
<th>Shift</th>
<th>Amount</th>
<th>Function</th>
</tr>
</thead>
</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>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Constant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>($s3)</td>
<td>($t0)</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>&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 (with $zero)</td>
</tr>
</tbody>
</table>
Control Instructions

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

- Unconditional branch:
  \texttt{j \ L1}
  \texttt{jr \ $s0}  \text{(useful for big jumps and procedure returns)}

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

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

• Unconditional branch:
  \texttt{j L1}
  \texttt{jr $s0}  \text{ (useful for big jumps and procedure returns)}

Convert to assembly:

\begin{align*}
\text{if (} & \text{i } \text{== } \text{j}) \\
& \quad \text{bne } \text{$s3, } \text{$s4, } \text{Else} \\
& \quad \text{add } \text{$s0, } \text{$s1, } \text{$s2} \\
\text{else} \\
& \quad \text{j } \text{End} \\
& \quad \text{Else: sub } \text{$s0, } \text{$s1, } \text{$s2} \\
& \quad \text{End:}
\end{align*}
Example

Convert to assembly:

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

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

Convert to assembly:

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

Values of 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
    j Loop

Exit:
    sll $t1, $s3, 2
    add $t1, $t1, $s6
    lw $t0, 0($t1)
    bne $t0, $s5, Exit
    addi $s3, $s3, 1
    addi $t1, $t1, 4
    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
Procedures

- Local variables, AR, $fp, $sp
- Scratchpad and saves/restores
- Arguments and returns
- jal and $ra