Lecture 5: Procedure Calls

• Today’s topics:
  - Memory layout, numbers, control instructions
  - Procedure calls
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)
  \[ 0x \, 23 \text{ or } 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0 \]

  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>
<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
000000  10001  10010  01000  00000  100000
```

6 bits  5 bits  5 bits  5 bits  5 bits  6 bits

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>source</td>
<td>source</td>
<td>dest</td>
<td>shift</td>
<td>amt</td>
</tr>
</tbody>
</table>

**I-type instruction**

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

6 bits  5 bits  5 bits  16 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></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: \texttt{beq register1, register2, L1}
  Similarly, \texttt{bne} and \texttt{slt} (set-on-less-than)

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

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

```c
if (i == j)                           bne $s3, $s4, Else
  f = g+h;                               add $s0, $s1, $s2
else
  f = g-h;                               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:

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
        j         Loop

Exit:

        sll      $t1, $s3, 2
        add      $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
Each procedure (function, subroutine) maintains a scratchpad of register values – when another procedure is called (the callee), the new procedure takes over the scratchpad – values may have to be saved so we can safely return to the caller

- parameters (arguments) are placed where the callee can see them
- control is transferred to the callee
- acquire storage resources for callee
- execute the procedure
- place result value where caller can access it
- return control to caller
Jump-and-Link

• A special register (storage not part of the register file) maintains the address of the instruction currently being executed – this is the program counter (PC)

• The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure’s address (the PC is accordingly set to this address)
  
  jal NewProcedureAddress

• Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction

• How do we return control back to the caller after completing the callee procedure?
The register scratchpad for a procedure seems volatile – it seems to disappear every time we switch procedures – a procedure’s values are therefore backed up in memory on a stack.
Saves and Restores
Storage Management on a Call/Return

• A new procedure must create space for all its variables on the stack

• Before/after executing the jal, the caller/callee must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into the stack space

• Arguments are copied into $a0-$a3; the jal is executed

• After the callee creates stack space, it updates the value of $sp

• Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented

• On return, the caller/callee brings in stack values, ra, temps into registers

• The responsibility for copies between stack and registers may fall upon either the caller or the callee
int leaf_example (int g, int h, int i, int j)  
{  
  int f;  
  f = (g + h) – (i + j);  
  return f;  
}  

Notes:  
In this example, the callee took care of saving the registers it needs.  

The caller took care of saving its $ra and $a0-$a3.  

Could have avoided using the stack altogether.
Saving Conventions

• Caller saved: Temp registers $t0-$t9 (the callee won’t bother saving these, so save them if you care), $ra (it’s about to get over-written), $a0-$a3 (so you can put in new arguments)

• Callee saved: $s0-$s7 (these typically contain “valuable” data)

• Read the Notes on the class webpage on this topic
Title

• Bullet