Lecture 3: MIPS Instruction Set

• Today’s topic:
  ▪ More MIPS instructions
  ▪ Procedure call/return

• Reminder: Assignment 1 is on the class web-page (due 9/7)
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?
Memory Address

• 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)

```plaintext
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:

\[ \text{destination register} \quad \text{source address} \]

\[ \text{any register} \quad \text{a constant that is added to the register in brackets} \]

\[ \text{lw} \quad $t0, \quad 8($t3) \]
Example

Convert to assembly:


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

Assembly version of the code continues to expand!
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)
  \[ 0 \times 23 \text{ or } 23_{\text{hex}} = 2 \times 16^1 + 3 \times 16^0 \]
  
  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></th>
<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></td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
<tr>
<td>opcode</td>
<td>source</td>
<td>source</td>
<td>dest</td>
<td>shift</td>
<td>amt</td>
<td>function</td>
</tr>
</tbody>
</table>

**I-type instruction**

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

<table>
<thead>
<tr>
<th></th>
<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>
<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</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:
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:

if (i == j)
    f = g+h;
else
    f = g-h;

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

Convert to assembly:

```c
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:
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
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
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 Stack

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.
Storage Management on a Call/Return

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

- Before executing the jal, the caller must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into its own 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 may bring in its stack values, ra, temps into registers

- The responsibility for copies between stack and registers may fall upon either the caller or the callee
Example 1

```c
int leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) - (i + j);
    return f;
}
```
Example 1

```c
int leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

leaf_example:

```
    addi  $sp, $sp, -12
    sw    $t1, 8($sp)
    sw    $t0, 4($sp)
    sw    $s0, 0($sp)
    add   $t0, $a0, $a1
    add   $t1, $a2, $a3
    sub   $s0, $t0, $t1
    add   $v0, $s0, $zero
    lw    $s0, 0($sp)
    lw    $t0, 4($sp)
    lw    $t1, 8($sp)
    addi  $sp, $sp, 12
    jr     $ra
```

Notes:
In this example, the procedure’s stack space was used for the caller’s variables, not the callee’s – the compiler decided that was better.

The caller took care of saving its $ra and $a0-$a3.
Example 2

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```
Example 2

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

```
fact:
    addi $sp, $sp, -8
    sw $ra, 4($sp)
    sw $a0, 0($sp)
    slti $t0, $a0, 1
    beq $t0, $zero, L1
    addi $v0, $zero, 1
    addi $sp, $sp, 8
    jr $ra
L1:
    addi $a0, $a0, -1
    jal fact
    lw $a0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    mul $v0, $a0, $v0
    jr $ra
```
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
Title

• Bullet