Lecture 4: MIPS Instruction Set

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

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
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
Base Address and Offsets

C code: a = b + c;

addi $gp, $zero, 1000 # putting base address 1000 into # the global pointer
lw $s2, 4($gp) # loading variable b into $s2
lw $s3, 8($gp) # loading variable c into $s3
add $s1, $s2, $s3 # sum in $s1
sw $s1, $gp # storing sum into variable a
addi $s4, $gp, 12 # $s4 now contains the start # address of array d[ ]
Example

Convert to assembly:


Assembly:

\[
\begin{align*}
\text{lw} & \quad \$t0, \ 8(\$s4) \quad \# \ d[2] \text{ is brought into } \$t0 \\
\text{add} & \quad \$t0, \ \$t0, \ \$s1 \quad \# \ \text{the sum is in } \$t0 \\
\text{sw} & \quad \$t0, \ 12(\$s4) \quad \# \ \$t0 \text{ is stored into } d[3]
\end{align*}
\]

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

  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** \( $t0, $s1, $s2 \)
- **opcode** 000000
- **rs** 10001
- **rt** 10010
- **rd** 01000
- **shamt** 00000
- **funct** 100000

### I-type instruction

- **lw** \( $t0, 32($s3) \)
- **opcode** 6 bits
- **rs** 5 bits
- **rt** 5 bits
- **constant** 16 bits
- **lw** 1 bit
# 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:  
  \[ \text{beq register1, register2, L1} \]
  Similarly,  \text{bne}  and  \text{slt}  (set-on-less-than)

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

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

• Conditional branch: Jump to instruction L1 if register1 equals register2: \( \text{beq register1, register2, L1} \)
  Similarly, \( \text{bne} \) and \( \text{slt} \) (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)                                   bne $s3, $s4, Else
  f = g+h;
else                                           j Exit
  f = g-h;

Else:  sub $s0, $s1, $s2
Exit:
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
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`
- `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.

Proc A’s values

Proc B’s values

Proc C’s values

Stack grows this way

High address

Low address

Proc A

call Proc B

... call Proc C

... return

return

return
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;
}
```

```
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
```
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

```c
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
```
Title

• Bullet