IC220
SlideSet #4: Procedures
(Chapter 2 finale)

Addressing in Conditional Branches

- Read Section 2.9 of text!
- You should understand the basics of “PC-relative” addressing

Stack Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push(6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Procedure Example & Terminology

```c
void function1() {
    int a, b, c, d;
    ...
    a = function2(b, c, d);
    ...
}
```

```c
int function2(int b, int c, int d) {
    int x, y, z;
    ...
    return x;
}
```
Big Picture – Steps for Executing a Procedure

1. Place parameters where the callee procedure can access them

2. Transfer control to the callee procedure

3. (Maybe) Acquire the storage resources needed for the callee procedure

4. Callee performs the desired task

5. Place the result somewhere that the “caller” procedure can access it

6. Return control to the point of origin (in caller)

Step #1: Placement of Parameters

- Assigned Registers: _____, _____, _____, & _____
- If more than four are needed?
- Parameters are not “saved” across procedure call

Step #2: Transfer Control to the Procedure

- jal –
  - Jumps to the procedure address AND links to return address
- Link saved in register _____
  - What exactly is saved?
  - Why do we need this?
    - Allows procedure to be called at ________ points in code, ________ times, each having a ________ return address

Step #3: Acquire storage resources needed by callee

- Suppose callee wants to use registers $$s_1$$, $$s_2$$, and $$s_3$$
  - But caller still expects them to have same value after the call
  - Solution: Use stack to

- Saving Registers $$s_1$$, $$s_2$$, $$s_3$$
  addi _____,_____, ____#
  sw $$s_1$$, ____($sp) #
  sw $$s_2$$, ____($sp) #
  sw $$s_3$$, ____($sp) #
Step #3 Storage Continued

Step #4: Callee Execution

- Use parameters from ________________ and ________________ (setup by caller)

- Temporary storage locations to use for computation:
  1. Temporary registers ($t0-$t9)
  2. Argument registers ($a0-$a3)
     - if...
  3. Other registers
     - but...
  4. What if still need more?

Step #5: Place result where caller can get it

- Placement of Result
  - Must place result in appropriate register(s)
    - If 32-bit value:
    - If 64-bit value:

- Often accomplished by using the $zero register
  - If result is in $t0 already then
    add ____, ____, $zero

Step #6: Return control to caller – Part A

- Part I – Restore appropriate registers before returning from the procedure
  - lw $s3, 0($sp) # restore register $s0 for caller
  - lw $s2, 4($sp) # restore register $t0 for caller
  - lw $s1, 8($sp) # restore register $t1 for caller
  - add $sp, $sp, _____ # adjust stack to delete 3 items
Step #6: Return control to caller – Part B

- Part II – Return to proper location in the program at the end of the procedure
  - Jump to stored address of next instruction after procedure call
    
    \[
    \text{jr} \quad \text{_____}
    \]

Recap – Steps for Executing a Procedure

1. Place parameters where the callee procedure can access them
2. Transfer control to the callee procedure
3. (Maybe) Acquire the storage resources needed for the callee procedure
4. Callee performs the desired task
5. Place the result somewhere that the “caller” procedure can access it
6. Return control to the point of origin (in caller)

Example – putting it all together

- Write assembly for the following procedure

  \[
  \text{int dog (int } n) \\
  \{ \\
  \quad n = n + 7; \ 
  \quad \text{return } n; \ 
  \}
  \]

- Call this function to compute dog(5):

Register Conventions

- Register Convention – for “Preserved on Call” registers (like $s0):
  1. If used, the callee must store and return values for these registers
  2. If not used, not saved

<table>
<thead>
<tr>
<th>Name</th>
<th>Reg#</th>
<th>Usage</th>
<th>Preserved on Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>constant value 0</td>
<td>N/A</td>
</tr>
<tr>
<td>lr</td>
<td>1</td>
<td>assembler temporary</td>
<td>N/A</td>
</tr>
<tr>
<td>$t0$ - $t1$</td>
<td>2-3</td>
<td>saved values from functions / (call used to set value for system call)</td>
<td>No</td>
</tr>
<tr>
<td>$a0$ - $a3$</td>
<td>4-7</td>
<td>arguments passed to function (or system call)</td>
<td>No</td>
</tr>
<tr>
<td>$s0$ - $s7$</td>
<td>8-15</td>
<td>temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$t8$ - $t9$</td>
<td>16-17</td>
<td>saved registers (main program)</td>
<td>Yes</td>
</tr>
<tr>
<td>$k0$ - $k1$</td>
<td>18-19</td>
<td>temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$fp$ - $sp$</td>
<td>20-25</td>
<td>temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$s0$ - $s1$</td>
<td>26-27</td>
<td>saved for OS</td>
<td>N/A</td>
</tr>
<tr>
<td>$gp$</td>
<td>28</td>
<td>global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp$</td>
<td>29</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$gp$</td>
<td>30</td>
<td>frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra$</td>
<td>31</td>
<td>return address (function call)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Nested Procedures

• What if the callee wants to call another procedure – any problems?

• Solution?

• This also applies to recursive procedures

Example – putting it all together (again)

• Write assembly for the following procedure

```c
int cloak (int n)
{
    if (n < 1) return 1;
    else return (n * dagger(n-1));
}
```

• Call this function to compute cloak(6):

```assembly
cloak:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slt $t0, $a0, 1
beq $t0, zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra

L1:
addi $a0, $a0, -1
jal dagger
lw $a0, 0($sp)
mul $v0, $a0, $v0 # pretend
lw $ra, 4($sp)
addi $sp, $sp, 8
jr $ra
```
What does that function do?

```c
int cloak(int n)
{
    if (n < 1) return 1;
    else return (n * dagger(n-1));
}
```

MIPS Addressing Summary

- Immediate addressing
- Register addressing
- Base addressing
- PC-relative addressing
- Pseudodirect addressing

MIPS Memory Organization

- Stack
- Dynamic data
- Static data
- Text
- Reserved

Alternative Architectures

- MIPS philosophy – small number of fast, simple operations
  - Name:

- Design alternative:
  - Name:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - Example VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
  - Others: PowerPC, 80x86
  - Danger?

- Virtually all new instruction sets since 1982 have been
80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, -instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - Hardware: the most frequently used instructions are...
  - Software: compilers avoid the portions of the architecture...

“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”

Chapter Goals

1. Teach a subset of MIPS assembly language
2. Introduce the stored program concept
3. Explain how MIPS instructions are represented in machine language
4. Illustrate basic instruction set design principles

Summary – Chapter Goals

1. (1) Teach a subset of MIPS assembly language
   - Show how high level language constructs are expressed in assembly
     - Demonstrated selection (if, if/else) and repetition (for, while) structures
     - MIPS instruction types
     - Various MIPS instructions & pseudo-instructions
     - Register conventions
     - Addressing memory and stack operations
### MIPS Operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>All registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Example**: Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.

### MIPS Instruction Language

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td><code>$s1, $s2, $s3</code></td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td><code>sub</code></td>
<td><code>$s1, $s2, $s3</code></td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td><code>addi</code></td>
<td><code>$s1, $s2, 100</code></td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td><code>lw</code></td>
<td><code>$s1, 100($s2)</code></td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td><code>sw</code></td>
<td><code>$s1, 100($s2)</code></td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td><code>lb</code></td>
<td><code>$s1, 100($s2)</code></td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td><code>sb</code></td>
<td><code>$s1, 100($s2)</code></td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td><code>lui</code></td>
<td><code>$s1, 100</code></td>
<td>$s1 = 100 * 2</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td><code>beq</code></td>
<td><code>$s1, $s2, 25</code></td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td><code>bne</code></td>
<td><code>$s1, $s2, 25</code></td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td><code>slt</code></td>
<td><code>$s1, $s2, $s3</code></td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td><code>slti</code></td>
<td><code>$s1, $s2, 100</code></td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td><code>j</code></td>
<td><code>2500</code></td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td><code>jr</code></td>
<td><code>$ra</code></td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td><code> jal</code></td>
<td><code>2500 $ra</code></td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

### Summary – Chapter Goals

1. **(2) Stored Program Concept**
   - Instructions are composed of bits / bytes / words
   - Programs are stored in memory — to be read or written just like data

2. **Memory**
   - Fetch & Execute Cycle
     - Instructions are fetched and put into a special register
     - Bits in the register “control” the subsequent actions
     - Fetch the “next” instruction and continue

3. **(3) Explain how MIPS instructions are represented in machine language**
   - Instruction format and fields
   - Differences between assembly language and machine language
   - Representation of instructions in binary

4. **(4) Illustrate basic instruction set design principles**
   1. Instructions similar size, register field in same place in each instruction format
   2. Only 32 registers rather than many more
   3. Providing for larger addresses and constants in instructions while keeping all instructions the same length
   4. Immediate addressing for constant operands