Procedure Example & Terminology

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

int function2(int b, int c, int d) {
    int x, y, z;
    ...
    return x;
}
```

Addressing in Conditional Branches

- Read Section 2.10 of text!
- You should understand the basics of “PC-relative” addressing

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 $s1, s2, and $s3
  - But caller still expects them to have same value after the call
  - Solution: Use stack to

- Saving Registers $s1, $s2, $s3

  ```
  addi _____,_____,#
  sw $s1, ___($sp) #
  sw $s2, ___($sp) #
  sw $s3, ___($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
  – addi $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
  jr _______
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
  
  ```c
  int dog (int n)
  {
      n = n + 7;
      return n;
  }
  ```

- Call this function to compute dog(5):

  EX: 2-31 to 2-33

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>$zero</td>
<td>0</td>
<td>constant value 0</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
<td>N/A</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values from functions</td>
<td>(also used to set value for system call)</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments passed to function (or system call)</td>
<td>No</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved registers (main)</td>
<td>Yes</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>24-25</td>
<td>temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</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
Nested Procedures

- "Activation record" – part of stack holding procedures saved values and local variables
- $fp$ – points to first word of activation record for procedure

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):

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

What does that function do?

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

- **MIPS Addressing**
  - Immediate addressing
  - Register addressing
  - Base addressing
  - PC-relative addressing
  - Pseudoaddress addressing

MIPS Memory Organization

- **MIPS Memory Organization**
  - $sp \rightarrow 7fff \text{ fff}_{hEx}$
  - $gp \rightarrow 1000 \text{ 8000}_{hEx}$
  - $1000 \text{ 0000}_{hEx}$
  - $pc \rightarrow 0040 \text{ 0000}_{hEx}$

Alternative Architectures

- **MIPS philosophy**
  - Small number of fast, simple operations
    - Name:
    - Others: ARM, Alpha, SPARC

- **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: 80x86, Motorola 68000
  - Danger?

- Virtually all new instruction sets since 1982 have been

The Intel x86 ISA

- **Evolution with backward compatibility**
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA

• Further evolution...
  – i486 (1989): pipelined, on-chip caches and FPU
    • Compatible competitors: AMD, Cyrix, ...
  – Pentium (1993): superscalar, 64-bit datapath
    • Later versions added MMX (Multi-Media eXtension) instructions
    • The infamous FDIV bug
    • New microarchitecture (see Colwell, The Pentium Chronicles)
  – Pentium III (1999)
    • Added SSE (Streaming SIMD Extensions) and associated registers
  – Pentium 4 (2001)
    • New microarchitecture
    • Added SSE2 instructions

• And further...
  – AMD64 (2003): extended architecture to 64 bits
    • EM64T – Extended Memory 64 Technology (2004)
      • AMD64 adopted by Intel (with refinements)
      • Added SSE3 instructions
    – Intel Core (2006)
      • Added SSE4 instructions, virtual machine support
    – AMD64 (announced 2007): SSE5 instructions
      • Intel declined to follow, instead...
    – Advanced Vector Extension (announced 2008)
      • Longer SSE registers, more instructions
  • If Intel didn’t extend with compatibility, its competitors would!
    • Technical elegance ≠ market success

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”

Summary – Chapter Goals

• (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>$s0-$s7, $t0-$t9, $zero</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
<td></td>
</tr>
<tr>
<td>$fp, $sp, $ra, $at</td>
<td>Memory locations. Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data and spilled registers, such as those saved on procedure calls.</td>
<td></td>
</tr>
</tbody>
</table>

MIPS assembly language

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

Summary – Chapter Goals

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

memory for data, programs, compilers, editors, etc.

Processor

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

Summary – Chapter Goals

(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

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td>26 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary – Chapter Goals

(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