SI232
SlideSet #4: Procedures
(more Chapter 2)

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;
}
```

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>

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. 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?
- 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

![Diagram showing allocation of registers on a stack](image-url)
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)
  3. Other registers
  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

- 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

- Return to proper location in the program at the end of the procedure
  - Jump to stored address of next instruction after procedure call
    jr ______
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>$ra</td>
<td>31</td>
<td>Return address</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame 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>28</td>
<td>Global pointer</td>
<td>N/A</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>Temporary registers (functions)</td>
<td>N/A</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>Temporary registers (functions)</td>
<td>Yes</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>Temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>Temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>Temporary registers (functions)</td>
<td>No</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>Assembler temporary constant</td>
<td>0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Preserved on Call</td>
<td>N/A</td>
</tr>
<tr>
<td>$zero</td>
<td>0</td>
<td>Preserved on Call</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Recap – Steps for Executing a Procedure

1. Place parameters where the callee procedure can access them
2. Transfer control to the callee procedure
3. 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)

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

- 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
example - putting it all together
```

Exercise #1

- Suppose you are given the code for the following function:

```c
int function1(int a, int b);
```

Write MIPS code to call function1(3, 7) and then store the result in $s0

```c
What does that function do?

int cloak (int n)
{
    if (n < 1) return 1;
    else return (n * dagger(n-1));
}
```
Exercise #2

- Now you have this definition for function1:
  ```c
  int function1(int a, int b) {
    return (a - b);
  }
  ```

Write MIPS code for function1.
(You won’t need to store anything on the stack – why not?)

Exercise #3

- Write the MIPS code for the following function
  ```c
  int function2(int a, int b) {
    return a + function1(a, b);
  }
  ```

(You will need to store something on the stack – why?)

Exercise #4

- Write the MIPS code for the following function
  ```c
  int function3(int a, int b) {
    return function6(a) + function7(b);
  }
  ```

(You will need to store something on the stack – why?)
**Addressing in Conditional Branches**

- Instructions:
  - bne $t4, $t5, Label Next instruction is at Label if $t4!=$t5
  - beq $t4, $t5, Label Next instruction is at Label if $t4==$t5
- Format:
  - 16 bits not enough to specify a complete address
  - Solution Part 1: “PC-relative addressing”
    - Offset is relative to Instruction Address Register
    - Most branches are local
  - Solution Part 2:
    - Last two bits of instruction address always ________
    - So, treat offset as plus/minus number of memory _______
- Random Nuance:
  - Final address relative to instruction following branch (PC+4), not PC

**Addressing in Jumps**

- Instructions:
  - j Label Next instruction is at Label
  - jal Label Next instruction is at Label
- Format:
  - J 26 bit Branch address
- How do we get a 32 bit address?
  - “Psuedodirect addressing”
    - 4 most significant. bits:
    - 28 other bits:

**Example: Addressing in Conditional Branches**

Instruction beq $s0, $s1, 25 means what?

- If $s0 == $s1 then the next instruction is

**Example: Addressing in Jumps**

32-bit Branch Address 00010110010100101001010010101010

Random Nuance:

- Jump instruction 00010110010100101001010010101010
  - Jump target is the current PC
  - Modified branch from jump instruction
### MIPS Addressing Modes

- **Immediate addressing**
  - `op rs rt`
- **Register addressing**
  - `op rs rt`
- **Base addressing**
  - `op rs rt`
- **PC-relative addressing**
  - `op`
- **Pseudodirect addressing**
  - `rd . . . funct`

### Exercise #1

- Label all the types of addressing that are used in this example:
  - (this program doesn’t compute anything, just look at the addressing)

```assembly
sub   $a0, $a1, $a2
addi $a1, $a0, 7
sll  $a2, $a1, 2
lw   $t0, $s0, $a2
bne  $a1, $a0, label1
j    label2
label1:      jr  $t0
label2:      add  $a0, $v0, $v1
```

### Exercise #2

- In the following code, instead of a label the branches have the actual number that goes in the machine language instruction.
  1. Where does the `bne` go if taken?
  2. Where does the `beq` go if taken?

```assembly
(100) add  $a1, $a0, $a2
(104) add  $a2, $a0, $a2
(108) add  $a3, $a0, $a2
(112) sub  $a0, $a1, $a2
(116) addi $a1, $a0, 7
(120) sll  $a2, $a1, 2
(124) lw   $t0, $s0, $a2
(128) bne  $a1, $a0, 1
(132) add  $a4, $a0, $a2
(136) add  $a1, $a0, $a2
(140) add  $a2, $a0, $a2
(144) add  $a3, $a0, $a2
(148) beq  $a2, $a1, -2
```

### Exercise #3

- Suppose the PC = 0x10cd 10f8  (0x = this is in hex)
  - (this doesn’t match the MIPS convention, but ignore that)
  - And we execute:

```assembly
j 0x1230
```

- What will the new PC be?
MIPS Memory Organization

- 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

Alternative Architectures

PowerPC

- Indexed addressing
  - example: lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4
  - What do we have to do in MIPS?

- Others:
  - load multiple/store multiple
  - a special counter register “bc Loop”
    decrement counter, if not 0 goto loop

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) 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

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

MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0-$s7</td>
<td></td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. 32 registers</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$zero</td>
<td></td>
<td>MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v0-$v1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$fp, $sp, $ra, $at</td>
<td></td>
<td>Memory[0], Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
<tr>
<td>$fp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$at</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lexicon

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

MIPS assembly language

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

- 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

Processor

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
</table>

(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>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>26 bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>26 bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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

Feedback

- Explain the jump and link instruction:

- From chapter 2, what topic(s) is still confusing to you?
End of Class Questions Results

• Explain the jump and link instruction:
  – A J-type instruction, used in conjunction with procedures, sets the PC to the first instruction of the procedure (i.e. does a jump) and stores (links) in register $ra the appropriate return address (PC + 4) to allow the procedure to return to correct memory location to continue the program after the procedure call.

• From chapter 2, what topic(s) is still confusing to you?
  – sll and slr – what do they do?
  – Stacks – how access in memory? How different from stack in C?
  – Ability to read/understand assembly programs
  – Nested function calls?
  – jal instruction – how do it all in one step?
  – 32 vs. 64 bit machines

Feedback

1. What was the most interesting part of Chapter 2? (assembly, instructions)

2. What is still confusing?