
SI232
SlideSet #4: Procedures
(more Chapter 2)

Stack Example

<u>Action</u>	<u>Stack</u>	<u>Output</u>
push(3)		
push(2)		
push(1)		
pop()		
pop()		
push(6)		
pop()		
pop()		
pop()		

Procedure Example & Terminology

```
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;
}
```

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

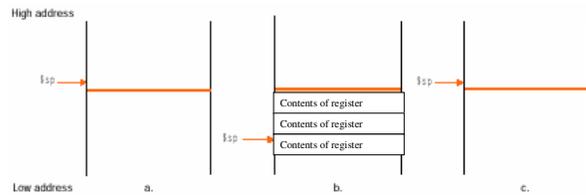
Step #3: Acquire storage resources needed by callee

- 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



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

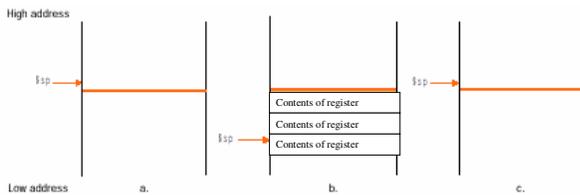
Step #6: Return control to caller – Part A

- 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

Name	Reg#	Usage	Preserved on Call
\$zero	0	constant value 0	N/A
\$at	1	assembler temporary	N/A
\$v0 - \$v1	2-3	returned values from functions (\$v0 used to set value for system call)	No
\$a0 - \$a3	4-7	arguments passed to function (or system call)	No
\$t0 - \$t7	8-15	temporary registers (functions)	No
\$s0 - \$s7	16-23	saved registers (main program)	Yes
\$t8 - \$t9	24-25	temporary registers (functions)	No
\$k0 - \$k1	26-27	reserved for OS	N/A
\$gp	28	global pointer	Yes
\$sp	29	stack pointer	Yes
\$fp	30	frame pointer	Yes
\$ra	31	return address (function call)	Yes

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

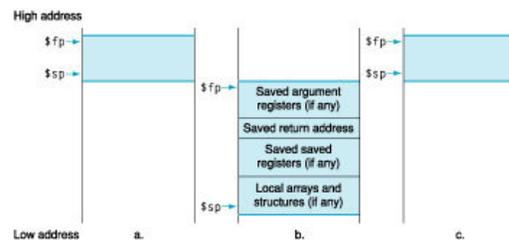
- What if the callee wants to call another procedure – any problems?

• 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

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

- Call this function to compute cloak(6):

Example – putting it all together

```
cloak:
    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  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?

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

Exercise #1

- Suppose you are given the code for the following function:
int function1(int a, int b);
Write MIPS code to call function1(3, 7) and then store the result in \$s0

Exercise #2

- Now you have this definition for function1:

```
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

```
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

```
int function3(int a, int b)  
    { return function6(a) + function7(b); }
```

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

(extra space)

Addressing in Conditional Branches

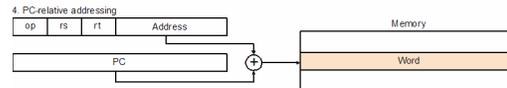
- Instructions:
 - `bne $t4, $t5, Label` Next instruction is at Label if $t4 \neq t5$
 - `beq $t4, $t5, Label` Next instruction is at Label if $t4 = t5$
- Format:

I	op	rs	rt	16 bit Branch offset
---	----	----	----	----------------------
- 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

Example: Addressing in Conditional Branches

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

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



Addressing in Jumps

- Instructions:
 - `j Label` Next instruction is at Label
 - `jal Label` Next instruction is at Label
- Format:

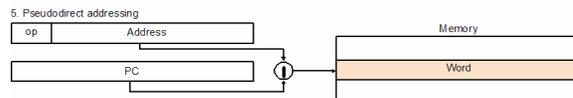
J	op	26 bit Branch address
---	----	-----------------------
- How do we get a 32 bit address?
 - "Pseudodirect addressing"
 - 4 most significant bits:
 - 28 other bits:

Example: Addressing in Jumps

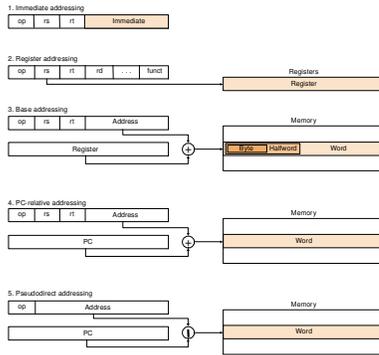
target field of jump instruction
 jump instruction 00001010110001010001010001100010

PC 010101100111011001110011001010010110
 Copy high-order four bits from PC

26-bit target field from jump instruction
 32-Bit Jump Address 01011011000101000101000110001000



MIPS Addressing Modes



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)

```

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.
 - Where does the bne go if taken?
 - Where does the beq go if taken?

(this program doesn't compute anything, just look at the branches)

```

(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

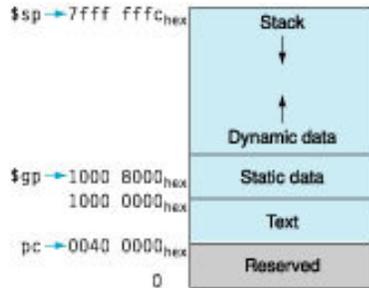
- Suppose the PC = 0x10cd 10f8 (0x = this is in hex) (this doesn't match the MIPS convention, but ignore that)

And we execute:

```
j 0x1230 (
```

What will the new PC be?

MIPS Memory Organization



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

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

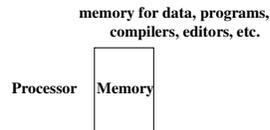
MIPS

MIPS operands				
Name	Example			Comments
32 registers	\$a0-\$a7, \$t0-\$t6, \$k0-\$k1, \$f0-\$f31, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at			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.
2 ³² memory words	Memory(0), Memory(4), ..., Memory(4294967296)			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.
MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$a1, \$a2, \$a3	$\$a1 = \$a2 + \$a3$	Three operands; data in registers
	sub	sub \$a1, \$a2, \$a3	$\$a1 = \$a2 - \$a3$	Three operands; data in registers
	add immediate	addi \$a1, \$a2, 100	$\$a1 = \$a2 + 100$	Used to add constants
Data transfer	add word	addw \$a1, 100(\$a2)	$\$a1 = \text{Memory}(\$a2 + 100)$	Word from memory to register
	store word	sw \$a1, 100(\$a2)	$\text{Memory}(\$a2 + 100) = \$a1$	Word from register to memory
	load byte	lb \$a1, 100(\$a2)	$\$a1 = \text{Memory}(\$a2 + 100)$	Byte from memory to register
	store byte	sb \$a1, 100(\$a2)	$\text{Memory}(\$a2 + 100) = \$a1$	Byte from register to memory
	load upper immediate	lui \$a1, 100	$\$a1 = 100 \cdot 2^16$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$a1, \$a2, 25	$\$a1 == \$a2$ go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$a1, \$a2, 25	$\$a1 \neq \$a2$ go to PC + 4 + 100	Not equal test; PC relative
	set less than	slt \$a1, \$a2, \$a3	$\$a1 = (\$a2 < \$a3) ? \$a1 + 1; \text{ else } \$a1 = 0$	Compare less than; for bcc, bne
	set less than immediate	slti \$a1, \$a2, 100	$\$a1 = (\$a2 < 100) ? \$a1 + 1; \text{ else } \$a1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 25000	Jump to literal address
	jump register	jr \$a1	go to \$a1	For switch, procedure return
	jump and link	jal 2500	$\$a0 = PC + 4; \text{ go to } 25000$	For procedure call

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



• 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

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

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
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 `sllr` – 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?

SCRAP
