

---

**IC220**  
**SlideSet #4: Procedures &**  
**Chapter 2 Finale**  
**(Sections 2.8)**

---

**Stack Example**

---

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

---

**Addressing in Conditional Branches**

---

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

---

**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. (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 #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 #1: Placement of Parameters

---

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

## 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

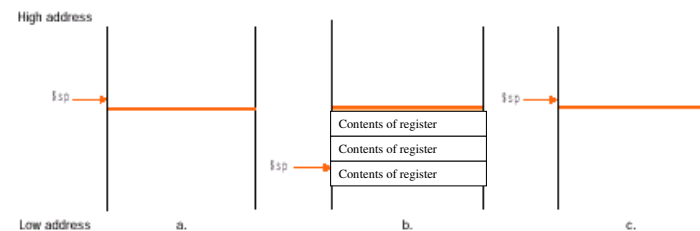
- Use parameters from \_\_\_\_\_ and \_\_\_\_\_ (setup by caller)
- Temporary storage locations to use for computation:
  1. Temporary registers (`$t0-$t9`)
  2. Argument registers (`$a0-$a3`)
  - if...
  3. Other registers
  - but...
  4. What if still need more?

## 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
  - `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

jr \_\_\_\_\_

## Example – putting it all together

EX: 2-31 to 2-33

- Write assembly for the following procedure

```
int dog (int n)
{
    n = n + 7;
    return n;
}
```

- Call this function to compute dog(5):

## Recap – Steps for Executing a Procedure

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

## Register Conventions

•Register Convention – for “Preserved on Call” registers (like \$s0):

- If used, the callee must store and return values for these registers
- 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

## 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)

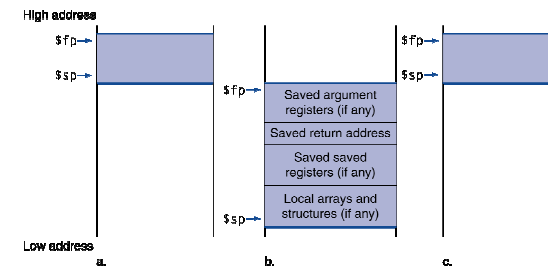
- Write assembly for the following procedure

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

- Call this function to compute cloak(6):

## 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

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

```
cloak:
    addi $sp, $sp, -8
    sw   $ra, 4($sp)
    sw   $a0, 0($sp)

    slli $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

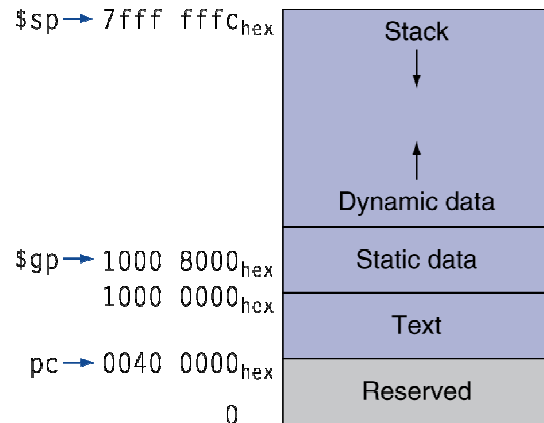
    lw   $ra, 4($sp)
    addi $sp, $sp, 8

    jr   $ra
```

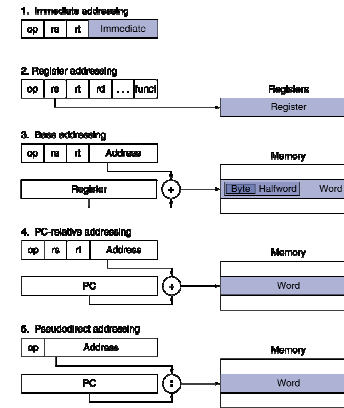
## What does that function do?

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

## MIPS Memory Organization



## MIPS Addressing Summary



## 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

---

- **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**

## 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
  - **Pentium Pro (1995), Pentium II (1997)**
    - 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

## 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

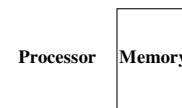
MIPS operands		
Name	Example	Comments
32 registers	\$s0-\$s7, \$s0-\$s9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2 <sup>30</sup> memory words	Memory[0], ..., Memory[4], ..., Memory[4294967295]	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 \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	\$s1 = \$s2 + 100	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	\$s1 = Memory[\$s2 + 100]	Word from memory to register
	store word	sw \$s1, 100(\$s2)	Memory[\$s2 + 100] = \$s1	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	\$s1 = Memory[\$s2 + 100]	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	Memory[\$s2 + 100] = \$s1	Byte from register to memory
	load upper immediate	lui \$s1, 100	\$s1 = 100 * 2 <sup>16</sup>	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if(\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if(\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if(\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	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

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

## 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