

# ECE3055

## Computer Architecture and Operating Systems

### Chapter 2: Procedure Calls & System Software

---



These lecture notes are adapted from those of Professor Sean Lee and Morgan Kaufman Pubs.

## Procedure Calls

---

- Basic functionality
  - Transfer of parameters to procedure
  - Transfer of results back to the calling program
  - Support for nested procedures
  
- What is so hard about this?
  - Consider independently compiled code modules

## Specifics

---

- Where do we pass data
  - Preferably registers → make the common case fast
  - Memory as an overflow area
  
- Nested procedures
  - The stack and \$sp and \$ra
  
- Set of rules that developers/compilers abide by
  - Which registers can am I permitted to use with no consequence?
  - Caller and callee save conventions for MIPS

3

## Our First Example

---

```

swap(int v[], int k);
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}

```

➔

```

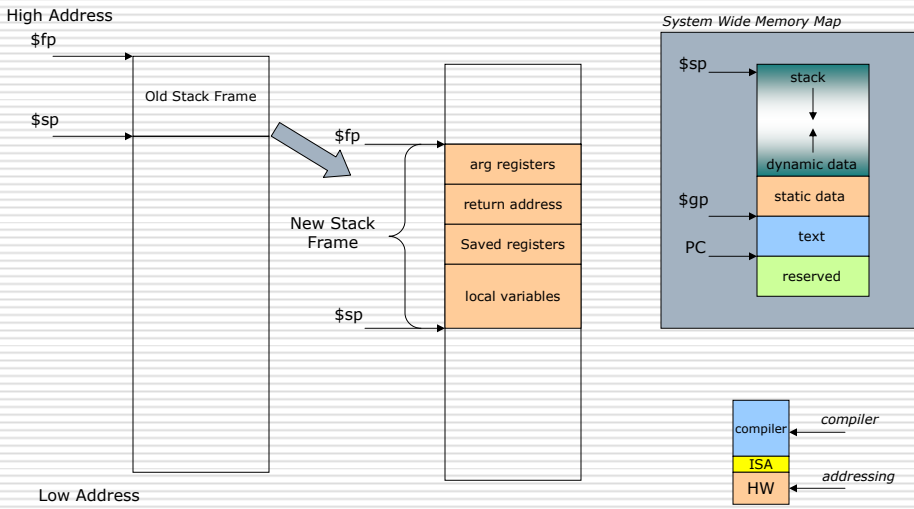
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31

```

- MIPS Software Convention
  - \$4, \$5, \$6, \$7 are used for passing arguments

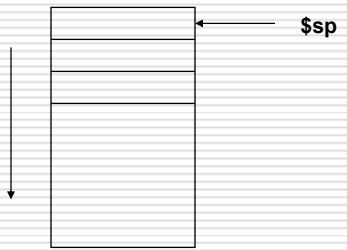
4

# Procedure Call Mechanics



# Parameter Passing and the Stack

- ❑ Register usage and need for state saving
- ❑ Nested calls and return addresses
- ❑ excess arguments



```

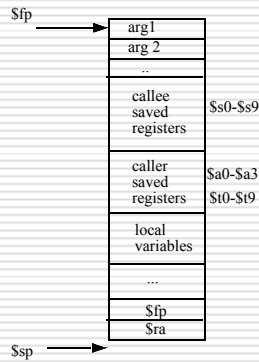
.data
arg1: .word 22, 20, 16, 4
arg2: .word 33,34,45,8

.text
addi $t0, $0, 4
move $t3, $0
move $t1, $0
move $t2, $0
loop: beq $t0, $0, exit
      addi $t0, $t0, -1
      lw $a0, arg1($t1)
      lw $a1, arg2($t2)
      jal func
      add $t3, $t3, $v0
      addi $t1, $t1, 4
      addi $t2, $t2, 4
      j loop

func: sub $v0, $a0, $a1
      jr $ra

exit: ---
    
```

# Example of the Stack Frame



## Call Sequence

1. place excess arguments
2. save caller save registers (\$a0-\$a3, \$t0-\$t9)
3. jal
4. allocate stack frame
5. save callee save registers (\$s0-\$s9, \$fp, \$ra)
6. set frame pointer

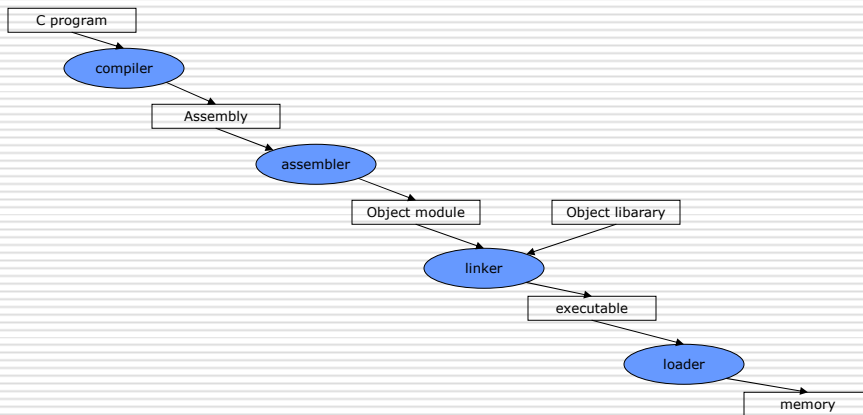
## Return

1. place function argument in \$v0
2. restore callee save registers
3. restore \$fp
4. pop frame
5. jr \$31

# Policy of Use Conventions

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

# The Complete Picture



# The Assembler

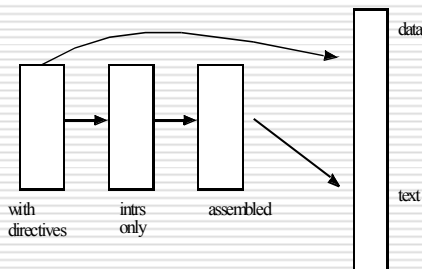
- Create a binary encoding of all native instructions
  - Translation of all pseudo-instructions
  - Computation of all branch offsets and jump addresses
  - Symbol table for unresolved (library) references
  
- Create an object file with all pertinent information

Header (information)
Text segment
Data segment
Relocation information
Symbol table

# Assembly Process

- ❑ One pass vs. two pass assembly
- ❑ effect of fixed vs. variable length instructions
- ❑ time, space and one pass assembly
- ❑ local labels, global labels, external labels and the symbol table
- ❑ absolute addresses and re-location

# Assembly Process

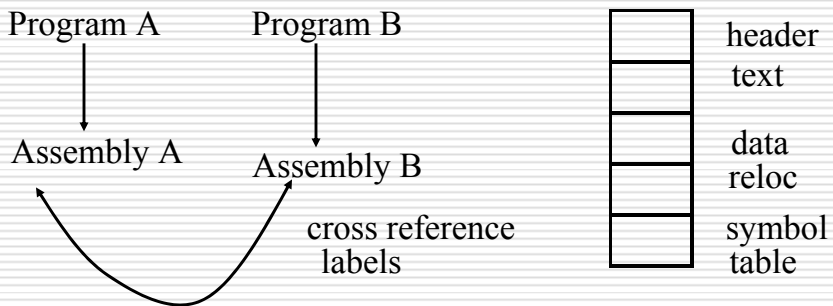


```
.data                                lui $1, 4097
.word 12, 15                          lw $8, 0($1)
                                        addi $9, $0, 4
                                        lui $1, 4097
.text
loop: lw $t0, L($zero)                 lw $9, 0($1)
    addi $t1, $zero, 4                 slt $11, $8, $9
    lw $t1, L($t1)                     beq $8, $0, 8
    slt $t3, $t0, $t1                   addi $8, $8, 8
    beq $t0, $zero, loop2              j 0x00400000
    addi $t0, $t0, 8                    addu $8, $0, $0
                                        ori $2, $0, 10
                                        syscall
loop2: move $t0, zero
```

# Linker & Loader

- Linker
  - “Links” independently compiler modules
  - Determines “real” addresses
  - Updates the executables with real addresses
  
- Loader
  - As the name implies
  - Specifics are operating system dependent

# Linking



- Why do we need independent compilation
- What are the issues with respect to independent compilation?
  - references across files
  - absolute addresses and relocation

## Example:

# separate file

```
.text          0x20040004
addi $4, $0, 4    0x20050005
addi $5, $0, 5    000011
jal func_add
done           0x0340200a
              0x0000000c
```

0x00400000	0x20040004
0x00400004	0x20050005
0x00400008	?
0x0040000c	0x0340200a
0x00400010	0x0000000c
0x00400014	0x008551020
0x00400018	0x03e00008

# separate file

```
.text
.globl func_add
func_add: add $2, $4, $5  0x00851020
jr $31                   0x03e00008
```

Ans: 0c100005

## Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy  
*instructions from 1 to 54 bytes long!*
- We'll look at PowerPC and 80x86

# 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*

17

# 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 (SIMD-INT) is added (PPMT and P-II)
- 1999: SSE (single prec. SIMD-FP and cacheability instructions) is added in P-III
- 2001: SSE2 (double prec. SIMD-FP) is added in P4
- 2004: Nocona introduced (compatible with AMD64 or once called x86-64)

“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”

18

# 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:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

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

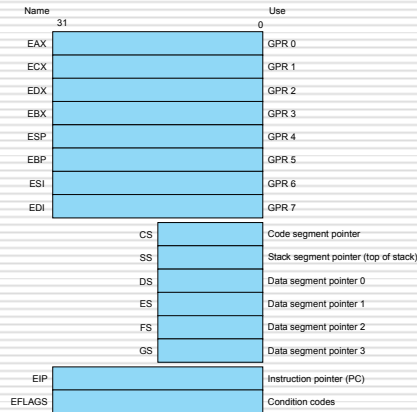
# IA-32 Overview

- 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:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

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

# IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



# IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	lw \$t0, 0(\$t1)
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	lw \$t0, 100(\$t1) # ≤16-bit # displacement
Base plus scaled index	The address is $\text{Base} + (2^{\text{Scale}} \times \text{Index})$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0, \$t2, 4 add \$t0, \$t0, \$t1 lw \$t0, 0(\$t0)
Base plus scaled index with 8- or 32-bit displacement	The address is $\text{Base} + (2^{\text{Scale}} \times \text{Index}) + \text{displacement}$ where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0, \$t2, 4 add \$t0, \$t0, \$t1 lw \$t0, 100(\$t0) # ≤16-bit # displacement

**FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code.** The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiples by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a mul to load the upper 16 bits of the displacement and an add to sum the upper address with the base register \$t1. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

# IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

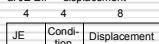
Instruction	Function
JE name	If equal (condition code) (EIP=name); EIP=EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOV EBX, [EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX, #6765	EAX=EAX+6765
TEST EDX, #42	Set condition code (flags) with EDX and 42
MOVS	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

**FIGURE 2.43** Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

# IA-32 instruction Formats

- Typical formats: (notice the different lengths)

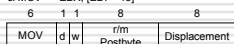
a. JE EIP + displacement



b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



## Summary

---

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!

## Study Guide

---

- Given the assembly of an independently compiled procedure, ensure that it follows the MIPS calling conventions, modifying it if necessary
- Given a SPIM program with nested procedures, ensure that you know what registers are stored in the stack as a consequence of a call
- Encode/disassemble *jal* and *jr* instructions
- What does it mean for a code segment to be relocatable?
- Computation of *jal* encodings for independently compiled modules
- For some instructions in the PowerPC and x86 ISAs, create equivalent sequences of MIPS instructions