EECE 417 Computer Systems Architecture

Department of Electrical and Computer Engineering
Howard University

Charles Kim

Spring 2007

Computer Organization and Design (3rd Ed)

-The Hardware/Software Interface

by

David A. Patterson
John L. Hennessy

Chapter 2

Instructions: Language of the Computer

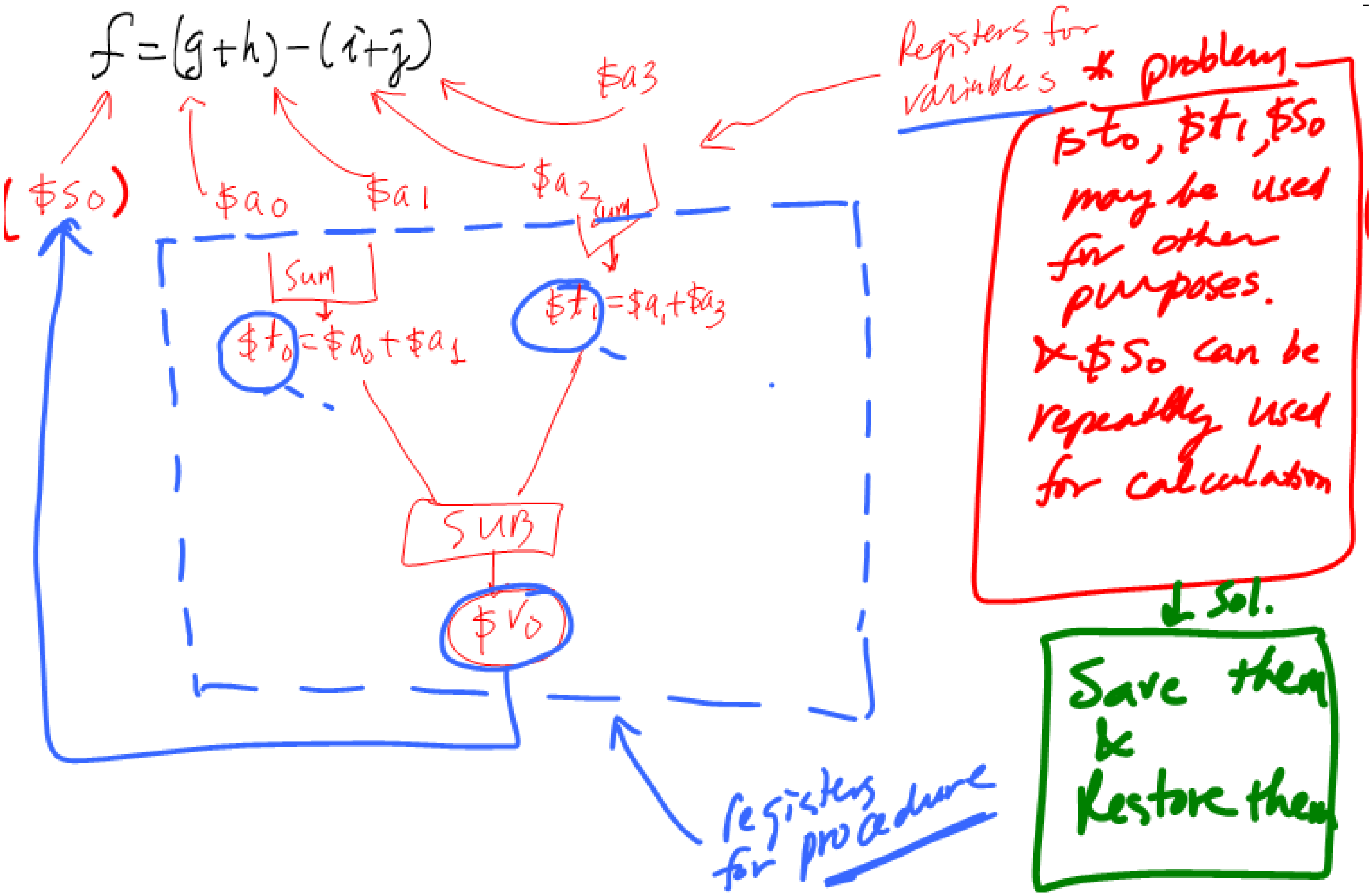
Chapter 2

Part C

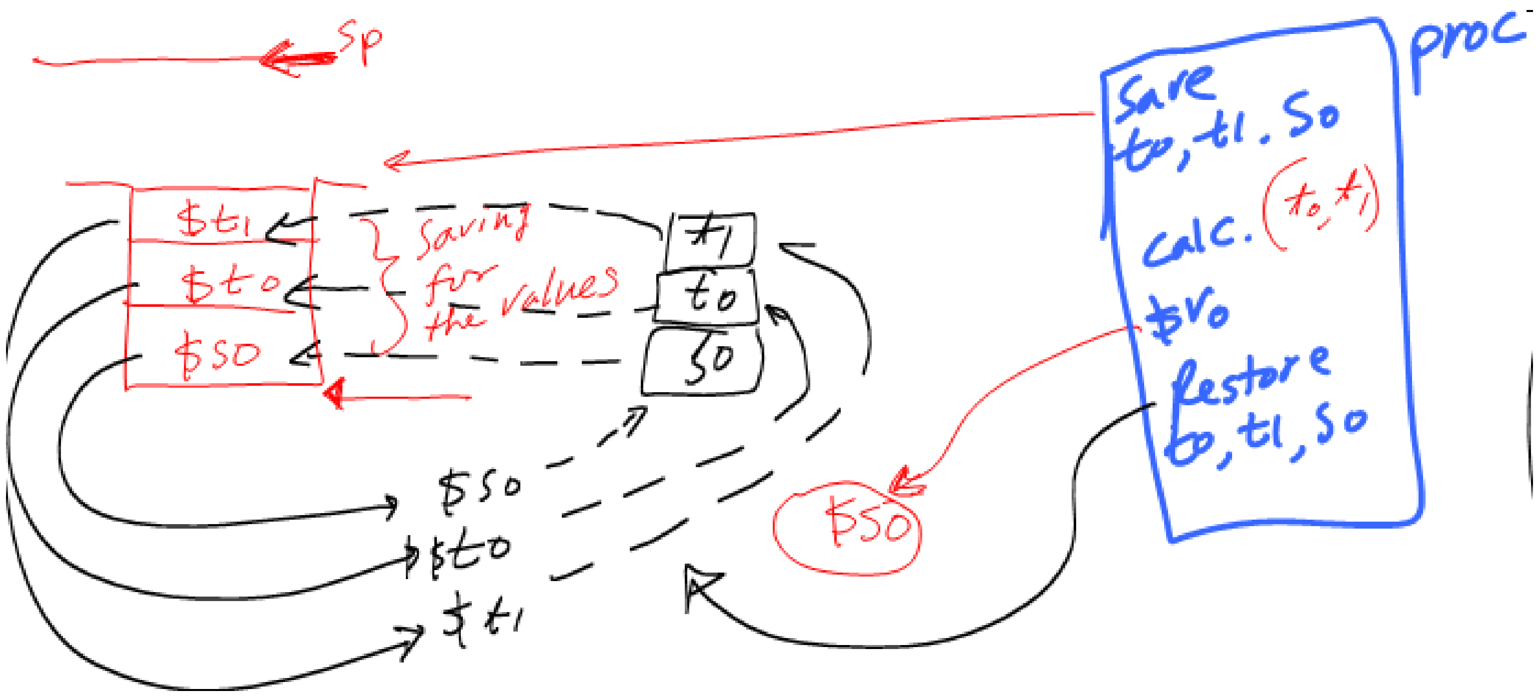
Procedure Calling

- Procedure [*subroutine*]
  - Tool for structuring programs
  - Code reuse
  - Pass values
  - Return results
- Register Convention in Procedure Calling
  - $a0 - $a3 : Parameter passing
  - $v0 - $v1: Result returning
  - $ra: Return address (automatically saved)
  - Stack is used when more arguments and more results are involved
- Saved and unsaved registers by the callee (*i.e.* subroutine) in the procedure calling
  - Saved: $s0 - $s7
  - Unsaved: $t0 - $t9
- Procedure calling instruction
  - `jal`
- Ending procedure
  - `jr  $ra`

Exercise of Procedure Calling



Using Stack for save/retrieve



Procedure calling (p81) p1/2

p81.asm - Notepad

File Edit Format View Help

```
#p81.asm
#Procedure calling exercise
#implements a leaf example
#    int leaf_example (int g, int h, int i, int j)
#    {
#        int f;
#        f = (g + h) - (i + j);
#        return f;
#    }
#assumes:
#  f is in $s0 and g, h, i and j are in registers $a0 through $a3
#
# the temprariy result of $t0 <--- $a0 + $a1 will be saved in the stack
# the temporary result of $t1 <----$a2+ $a3 will be saved in the stack
# The result $s0 will be saved in the stack

        .data 0x10010000
        .asciiz "\nThe value of f is: " #string to print
        .text
main:                                       #main has to be a global label
        addu    $s7, $zero, $ra             #save the return address to main
        addi    $s0, $zero, -1              #initialize $s0 to -1
        addi    $a0, $zero, 5               #g = 5
        addi    $a1, $zero, -20             #h = -20
        addi    $a2, $zero, 13              #i = 13
        addi    $a3, $zero, 3               #j = 3
        jal     leaf_example                #call the function leaf_example
        add     $s0, $zero, $v0             #set f to the computed value

                #Now print out f

        ori     $v0, $zero, 4              #print_str (system call 4)
        lui     $a0, 0x1001
        or      $a0, $a0, $zero            # takes the address of string as a
        syscall

        ori     $v0, $zero, 1              #print_int (system call 1)
        add     $a0, $zero, $s0            #put value to print in $a0
        syscall
```

Procedure calling (p81) p2/2

```
#Usual stuff at the end of the main
        addu    $ra, $zero, $s7         #restore the return address
        jr      $ra                     #return to the main program
        add     $zero, $zero, $zero     #nop



        .globl  leaf_example
leaf_example:
        addi    $sp, $sp, -12   #make space on the stack for three items
        sw      $t1, 8($sp)     #save register $t1
        sw      $t0, 4($sp)     #save register $t2
        sw      $s0, 0($sp)     #save register $s0
        add     $t0, $a0, $a1   #register $t0 contains g + h
        add     $t1, $a2, $a3   #register $t1 contains i + j
        sub     $s0, $t0, $t1   #f = (g + h) - (i + j)
        or      $v0, $s0, $zero #returns f
        lw      $s0, 0($sp)     #restore register $s0
        lw      $t0, 4($sp)     #restore register $t0
        lw      $t1, 8($sp)     #restore register $t1
        addi    $sp, $sp, 12    #adjust the stack before the return
        jr      $ra             #return to the calling program
```

Assignment

- Revise the code so that it receives the values of g, h, i, and j from keyboard

Nested Procedures

- Possible Conflicts
  - Argument Register Values
  - Return Address values
- One Solution?
  - Caller: Push any argument registers ($a0 - $a3), temporary registers ($t0 - $t9) that are need after call to stack. Upon return, restore the registers from the stack.
  - Callee: Push the return address ($ra) and saved registers ($s0 - $s7) to stack
- Example (p.83) - recursive procedure that calculates factorial

```
Int fact (int n)
{
    if (n<1) return (1);
         else  return (n* fact (n-1));
}
```

- Argument *n* ---> $a0
- Return Address ($ra)
- Output *f* --> $s0

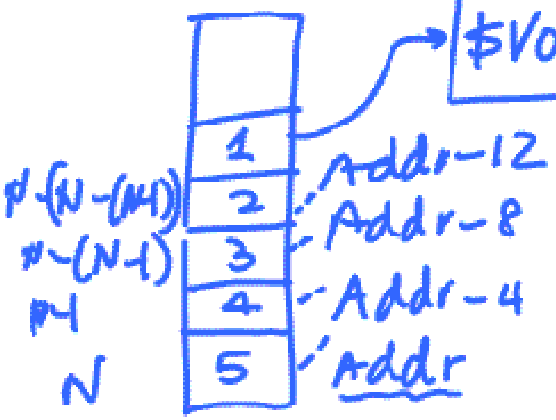
Factorial Calculation Details (1)

Factorial Calculation

$N = 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ $1! = 1$ $0! = 1$

in μP: Store 5, 4, 3, 2, 1 somewhere, then Restore them back one at a time, and multiply.

$\$V_0 = 1$ when (if) $N = 0$



$\overset{N}{[\text{Addr}]} * \$V_0 \rightarrow \$V_0$, # of MULT

$\overset{N-1}{[\text{Addr}-4]} * \$V_0 \rightarrow \$V_0$, Track of N-1

$\overset{(N-1)-1}{[\text{Addr}-8]} * \$V_0 \rightarrow \$V_0$

$\overset{(N-1)-1)-1}{[\text{Addr}-12]} * \$V_0 \rightarrow \underbrace{\$V_0}_{\text{Final Ans.}}$

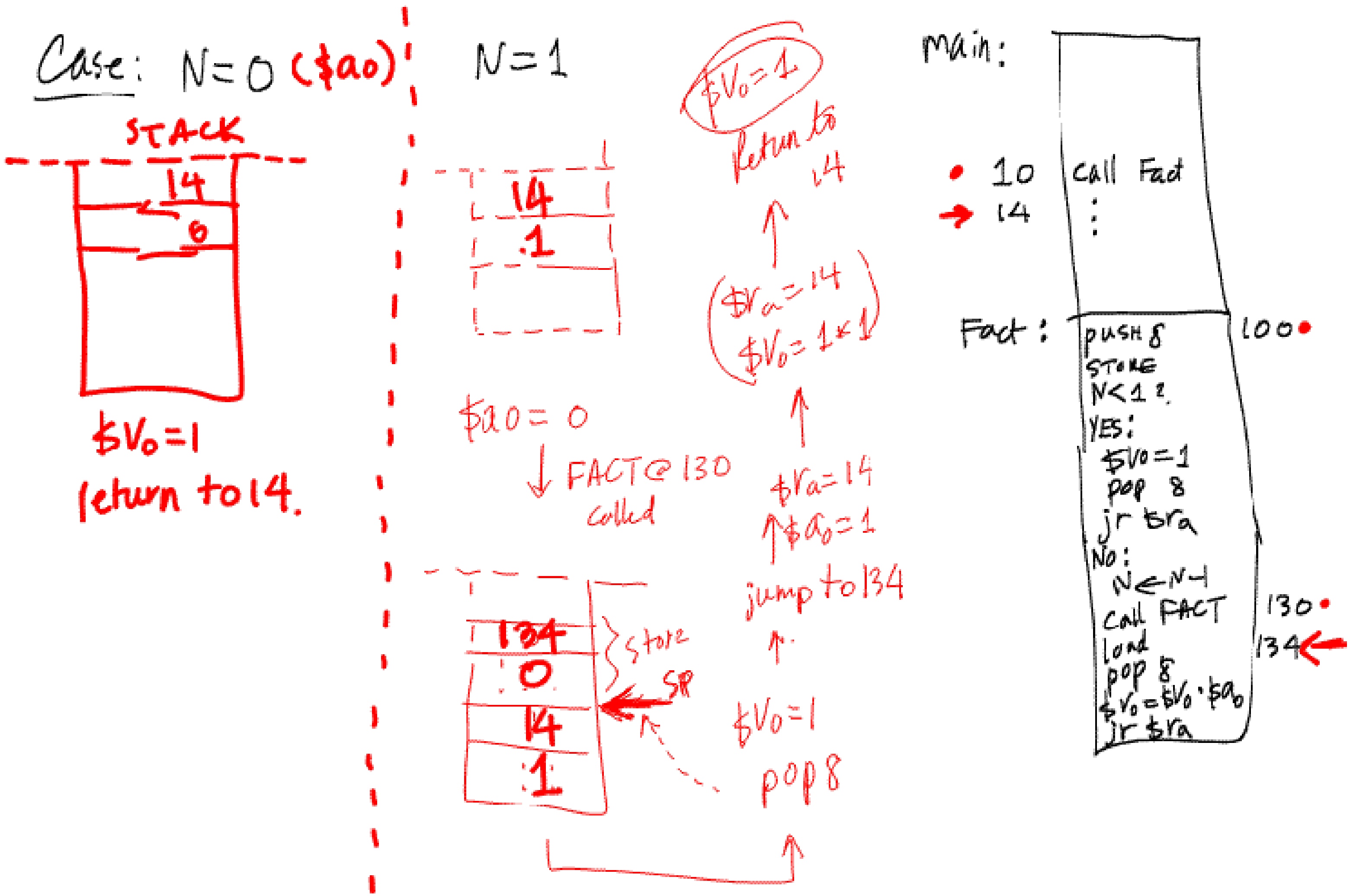
Factorial Calculation Details (2)

FACT (procedure)

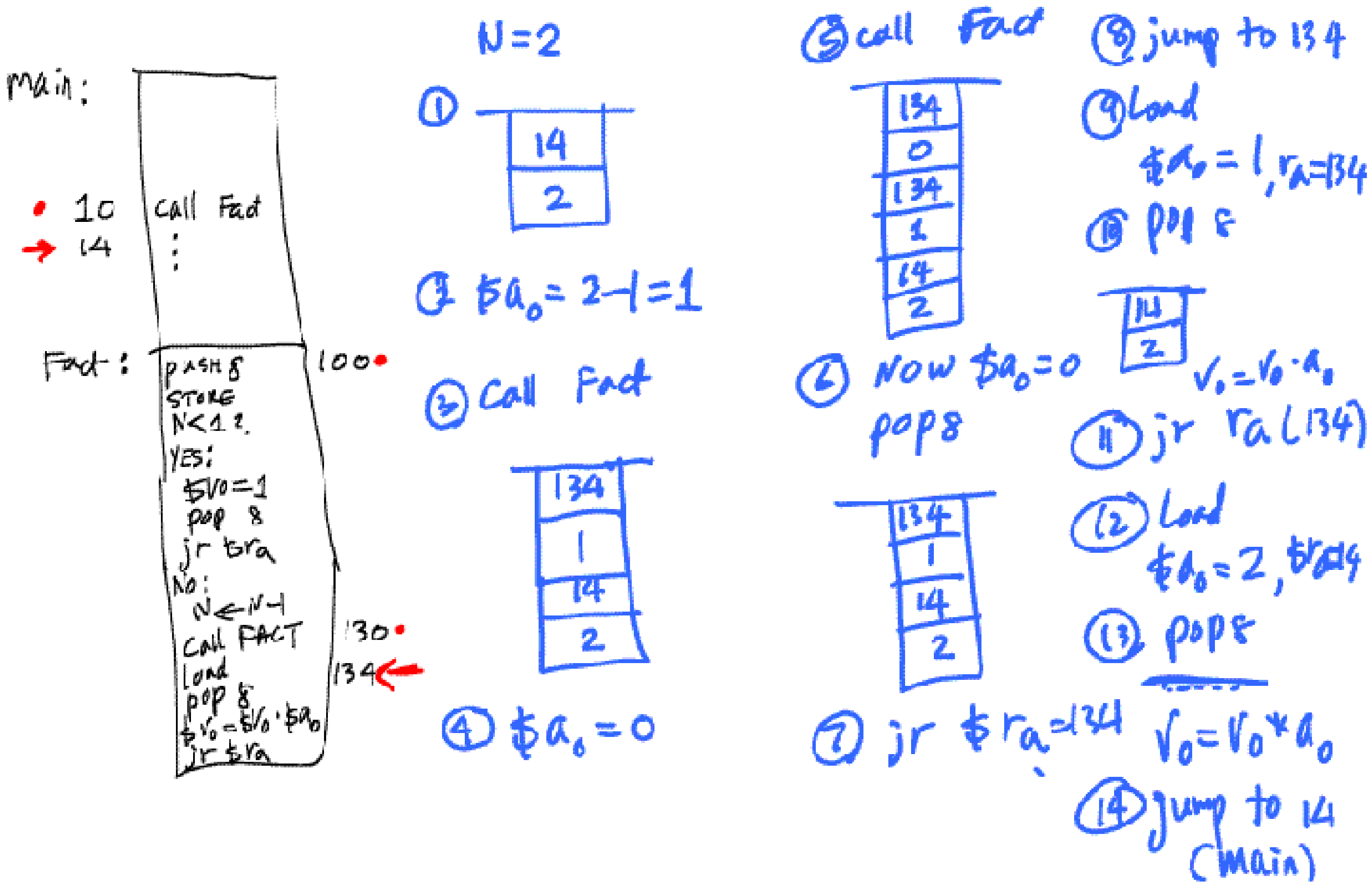
N ($a0) → [!] → f ($v0)

```
Fact:  push by 8      (to store $a0 & Return Addr)
       store $ra
       store $a0
       N < 1 ?
           YES :  $v0 = 1
                  pop by 8
                  return to [$ra]
           No :   N = N-1
                  call Fact
                  load $a0
                  load $ra
                  pop by 8
                  multi $a0 * $v0
                  return to [$ra]
```

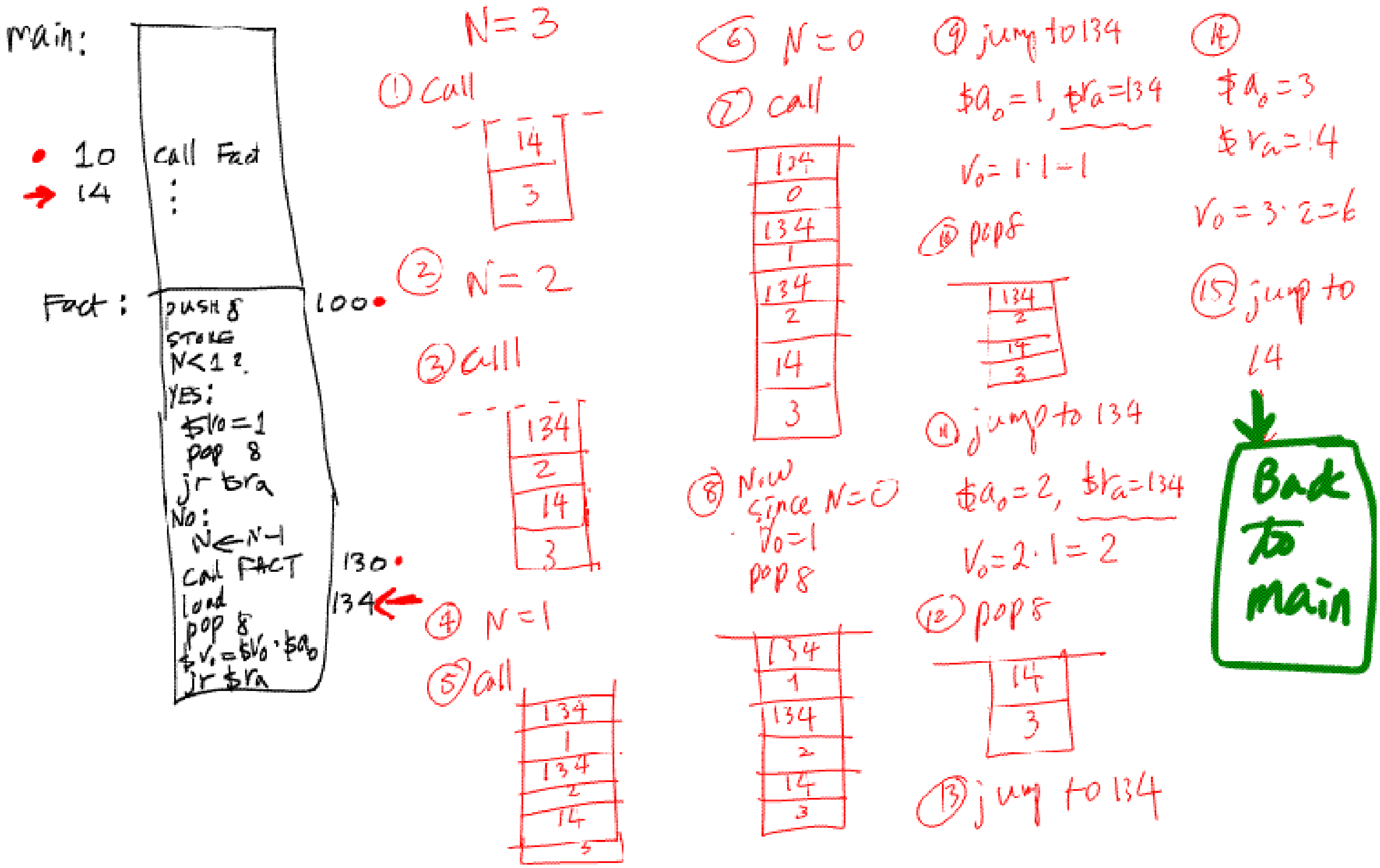
Factorial Calculation Details (3)



Factorial Calculation Details (4)



Factorial Calculation Details (5)



Nested Procedure Calling (p.83)

p83.asm - Notepad

File Edit Format View Help

```
#p83.asm
#implements a recursive version of N!
# Read the number N from keyboard
# Main program:
# void main()
#{   int N, f;
#    printf("\nFactorial Calculation\n");
#    printf("Enter N: ");
#    scanf("%d", N);
#    f = fact(N);
#    printf("\nN! = %d", f);
#    return;
#}
# int fact(int n)
# {
#    if (n < 1)
#        return 1;
#    else return (n * fact(n-1));
# }
# main assumes:
#   f is in $s0 and N is in $s1
main:
        .data   0x10010000
        .asciiz "\nFactorial Calculation\n"      #banner
        .data   0x10010020
        .asciiz "\nEnter N :"                   #Read key value N
        .data   0x10010030
        .asciiz "\nN! = "                       #Result
        .text
#       addu    $s7, $zero, $ra #save the return address in a global register
        ori     $v0, $zero, 4                   #print string
        lui     $a0, 0x1001
        ori     $a0, $a0, 0                     #$a0= 10010000
        syscall                                 #"Factorial Calculation"
again:  ori     $v0, $zero, 4                   #print str
        lui     $a0, 0x1001
        ori     $a0, $a0, 0x0020                #$a0= 10010020
        syscall                                 #"Enter N"
        ori     $v0, $zero,5                    #Integer Read
        syscall
```

Nested Procedure Calling (p.83)-Conti.

```
        or      $s1, $zero, $v0                 #$s1 <---N
        or      $a0, $zero, $s1                 #set the parameter ($a0) for N
        jal     fact                            #Call procedure FACT
        or      $s0, $zero, $v0                 #$s0 <---$v0= fact(N)
        ori     $v0, $zero, 4                   #print_str (system call 4)
        lui     $a0, 0x1001
        ori     $a0, $a0,0x0030
        syscall                                 #"N! = "
        ori     $v0, $zero, 1                   #Interger Print
        or      $a0, $zero, $s0
        syscall                                 #output f
        j       again
                                #Usual stuff at the end of the main
#        addu    $ra, $zero, $s7    #restore the return address
#        jr      $ra                #return to the main program
#        add     $zero, $zero, $zero      #nop

        .globl  fact                            #Procedure
fact:
        addi    $sp, $sp, -8            #make space on the stack for two items
        sw      $ra, 4($sp)             #save the return address on the stack
        sw      $a0, 0($sp)             #save the argument n on the stack
        slt     $t0, $a0, 1             #Set (i.e. $t=1) if $a0<1) to test for n < 1
        beq     $t0, $zero, L1          # If ($t=0 is not eqaual to 0) then jump to L1
        addi    $v0, $zero, 1           #otherwise return 1
        addi    $sp, $sp, 8             #  (just pop the saved items off stack since n
        jr      $ra                     #  and return)
L1:
        addi    $a0, $a0, -1            #when n >= 1:  decrement the argument
        jal     fact                    #call fact(n-1)
        lw      $a0, 0($sp)             #restore the value of argument n
        lw      $ra, 4($sp)             #restore the return address
        addi    $sp, $sp, 8             #release the save area on the stack
        mul     $v0, $a0, $v0           #(n*fact(n-1))
        jr      $ra                     #return
```

Coding Exercise (p.123) - Bubble Sort

- Bubble Sort
  - Simplest way of sorting array of objects
  - Maybe slowest too
  - Compare neighboring two objects, and swap them if the order in in the wrong way

```
for (i=0; i<n-1; i++) {   /* number of array is n */
    for (j=0; j<n-1-i; j++)
       if (a[j+1] < a[j]) { /* compare the two neighbors */
          tmp = a[j];       /* swap a[j] and a[j+1]       */
          a[j] = a[j+1];
          a[j+1] = tmp;
     }
 }
```

- Passes
  - Each pass moves the biggest number to the end of the array

Bubble Sort Exercise (p.123) - conti

Bubble Sort

A[0] [1] [2] [3] [4] ← j

$i = 1 \sim (n-1)$

$j = 0 \sim ((n-1)-i)$

n=5 five numbers in an array

8 6 3 7 4

i=0 1st pass

6 8 3 7 4

6 3 8 7 4

6 3 7 8 4

6 3 7 4 $\boxed{8}$ ← After 1st PASS (i=0)

[j=0 ~ 4] Largest Number — Inner loop

outer loop

i=1 2nd pass

6 3 7 4 8

3 6 7 4 8 — check only this numbers [j=0 ~ 3]

3 6 4 $\boxed{7}$ — After 2nd pass (i=1)

i=2 (3 4 $\boxed{6}$) [j=0 ~ 2] (i=2)

$j=(n-1)-i$

i=3 (3 $\boxed{4}$) [j=0 ~ 1] (i=3)

i=4 ($\boxed{3}$) [j=0] [i=4]

Bubble Sort Code (1)

p123.asm - Notepad

File Edit Format View Help

```
#p123.asm (bubble sort)
#Bubble Sort of 10 number array
#print output numbers in ascending order
#
#
#   for (i=0; i<n-1; i++) {
#     for (j=0; j<n-1-i; j++)
#       if (A[j+1]<A[j]) {
#         tmp=A[j];
#         A[j]=A[j+1];
#         A[j+1]=tmp;
#     }
#   }
# n -- the size of array ($a1)
# i --$s0
# j -- $s1
# Base address of A[] ---($s2)
# tmp -- $t0
# k -- $s4
        .data   0x10010000
        .asciiz "\nBubble Sort\n"       #string to print
        .data   0x10010020
        .asciiz "\nType 10 numbers of array to sort\n"
        .data   0x10010050
        .asciiz "/"
        .data   0x10010100
        .space  10                       #allocate 10 word spaces
        .text
```

Bubble Sort Code (2)

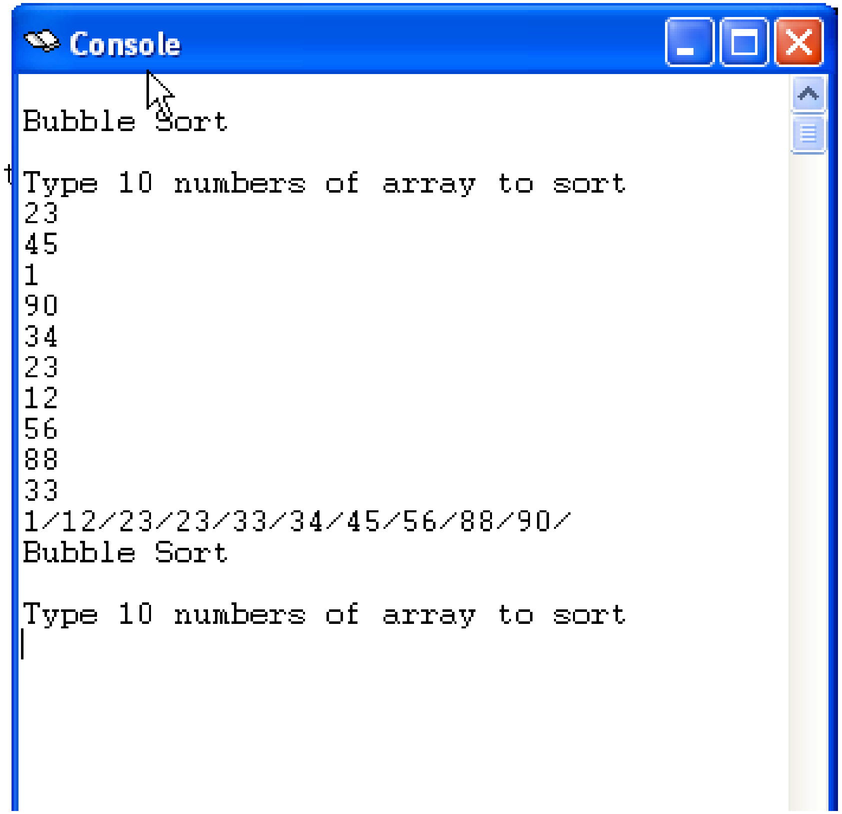
```
main:
        ori     $a1, $zero, 10          #$a1=n=10
#Read 10 numbers
        lui     $a0, 0x1001
        ori     $a0, $a0, 0x0100
        or      $s2, $a0, $zero         #$t2 is the base array addr

#string
        ori     $v0, $zero, 4          #print_str (system call 4)
        lui     $a0, 0x1001
        or      $a0, $a0, $zero        # takes the address of string a
        syscall
        ori     $v0, $zero, 4          #print_str (system call 4)
        lui     $a0, 0x1001
        ori     $a0, $a0, 0x0020        # takes the address of string
        syscall
#count the numbers entered in (k: 0 - 9)
        or      $s5, $s2, $zero         #s5 points memory now
        ori     $s4, $zero, 0
Keepread:
        ori     $v0, $zero, 5           #read input (b)
        syscall                         #now type-in is in v0
        or      $s1, $zero,$v0          #$s1
        sw      $s1, 0($s5)
        addi    $s4, $s4,1
        slti    $t3, $s4, 10
        beq     $t3, $zero, Readone
        addi    $s5,$s5, 4
        j       Keepread
```

Bubble Sort Code (3)

```
Readone:
        ori     $s0, $zero, 0           #i=0

        addi    $s3, $a1, -1            #n-1
#now sort part (outer loop)
# for i
Outer:  slt     $t0, $s0, $s3           #if i<(n-1) then $t0=1
        beq     $t0, $zero,exit1        #if not, exit1
#now inner loop
# for j
        sub     $t1, $s3,$s0            #t1 for max j which is (n-1)-i
        ori     $s1, $zero, 0           #j=0
Inner:  slt     $t0, $s1, $t1           #if j <(n-1-i) then $t0=1
        beq     $t0, $zero, exit2       #if not, exit2

#Now check if A[j+1]<A[j]
#get A[j]
#Add j*4 to the base address
        sll     $t6, $s1, 2             #t1=j*4
        add     $t2, $s2,$t6
        lw      $t3, 0($t2)             #A[j]
        lw      $t4, 4($t2)             #A[j+1]
#Compare
        slt     $t0, $t4, $t3           #if $t4<$t3, then $t0=1
        beq     $t0, $zero, exit3       #
#Now Swap
        sw      $t4, 0($t2)
        sw      $t3, 4($t2)
exit3:
        addi    $s1, $s1,1              #j=j+1
        j       Inner
exit2:
        addi    $s0, $s0, 1
        j       Outer
```

Bubble Sort Code (4)

```
exit1:
#print out

#count the numbers entered in (k: 0 - 9)
#         ori     $s2, $a0, 0               #$s2 has the original based addr
          ori     $s4, $zero, 0
Keept:
          ori     $v0, $zero, 1             #print
          lw      $t1, 0($s2)
          or      $a0, $zero, $t1
          syscall
#slash
          ori     $v0, $zero, 4            #print_st
          lui     $a0, 0x1001
          ori     $a0, $a0, 0x0050
          syscall
          addi    $s4, $s4,1
          slti    $t3, $s4, 10
          beq     $t3, $zero, exit4
          addi    $s2,$s2, 4
          j       Keept
#Usual stuff at the end of the main
exit4:
          j       main
```

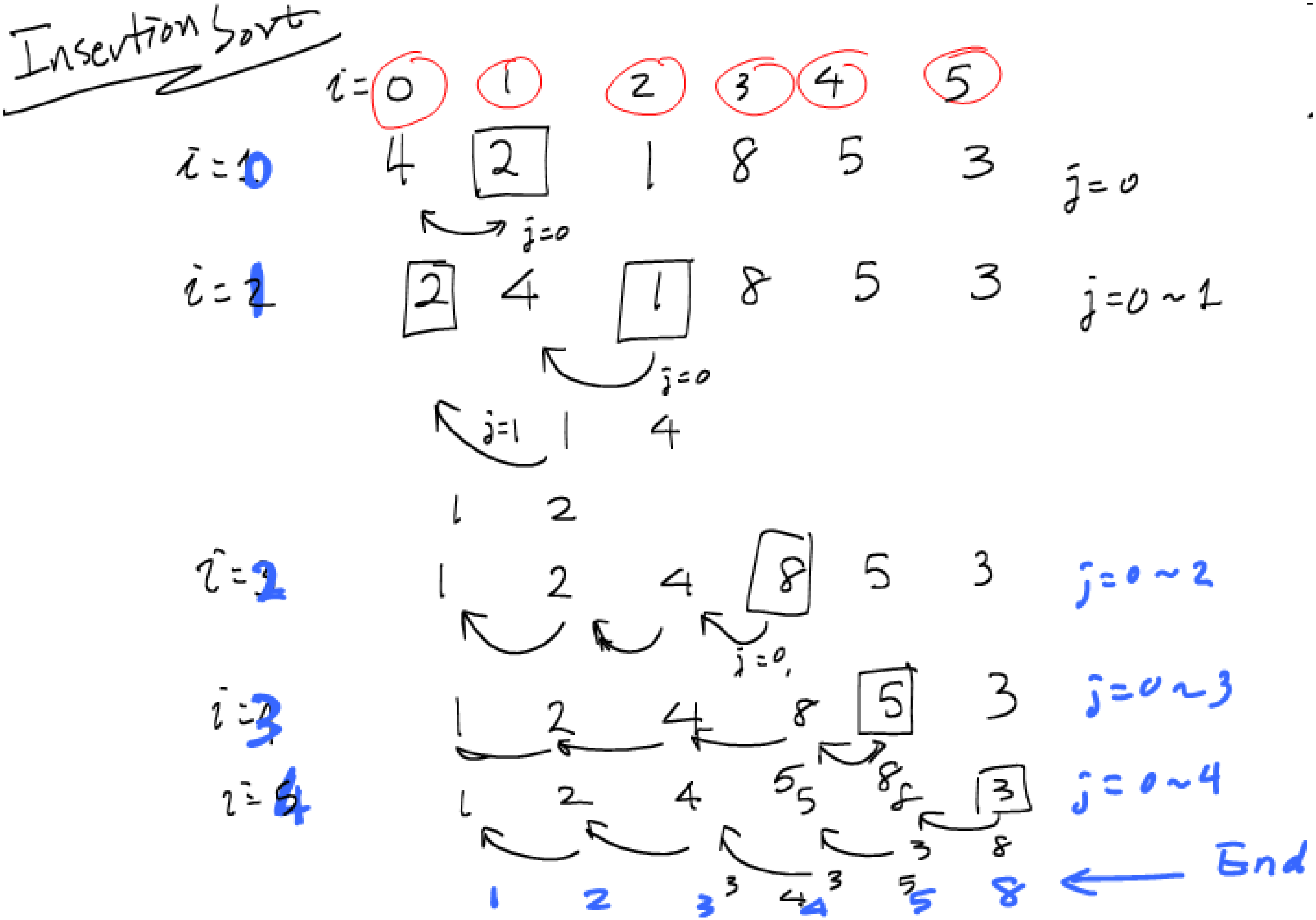


Insertion Sort

- it inserts each item into its proper place
- moving the current item past the already sorted items and repeatedly swapping it with the preceding item
- Twice as efficient as bubble sort

```
int i, j, tmp;
for (i=1; i < array_size; i++) {
    tmp = A[i];
    j = i;
    while ((j > 0) && (A[j-1] > tmp)) {
          A[j] = A[j-1];
          j = j - 1;
    }
    A[j] = tmp;
}
```

Insertion Code Illustration



Project 1 - Insertion Code

- **Read 10 numbers from keyboard**
- **Sort them by Insertion Sort algorithm**
- **Print the sorted numbers in the order**
- **Use only core MIPS instructions (no pseudo-instruction)**
- **Showing each step would earn extra points**
- **Due: in 2 weeks**

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

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

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- **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`

  `j Label` Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address

J	op	26 bit address

- **Addresses are not 32 bits**
  - — **How do we handle this with load and store instructions?**

Addresses in Branches

- **Instructions:**

```
bne $t4,$t5,Label
beq $t4,$t5,Label
```

Next instruction is at Label if $t4≠$t5
Next instruction is at Label if $t4=$t5

- **Formats:**

I	**op**	**rs**	**rt**	**16 bit address**

- **Could specify a register (like lw and sw) and add it to address**
  - **use Instruction Address Register (PC = program counter)**
  - **most branches are local (principle of locality)**
- **Jump instructions just use high order bits of PC**
  - **address boundaries of 256 MB**

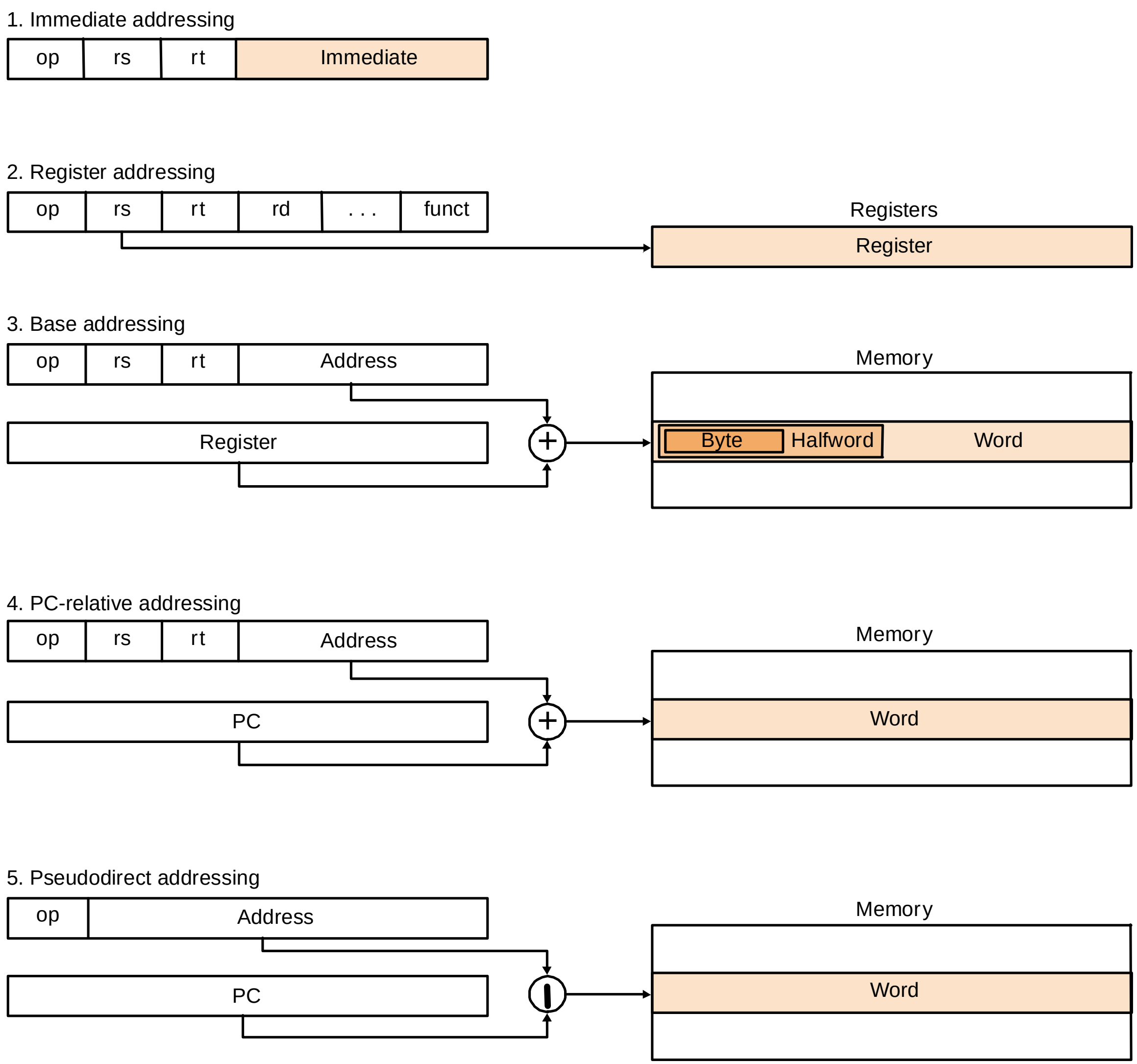
summary

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $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^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	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^{16}	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
Uncondi-tional 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


1. Immediate addressing
op
rs
rt
Immediate
2. Register addressing
op
rs
rt
rd
. . .
funct
Registers
Register
3. Base addressing
op
rs
rt
Address
Register
Memory
Byte
Halfword
Word
4. PC-relative addressing
op
rs
rt
Address
PC
Memory
Word
5. Pseudodirect addressing
op
Address
PC
Memory
Word

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

    – *"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"*

- Let's look (briefly) at IA-32

IA - 32

- 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: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

- "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"

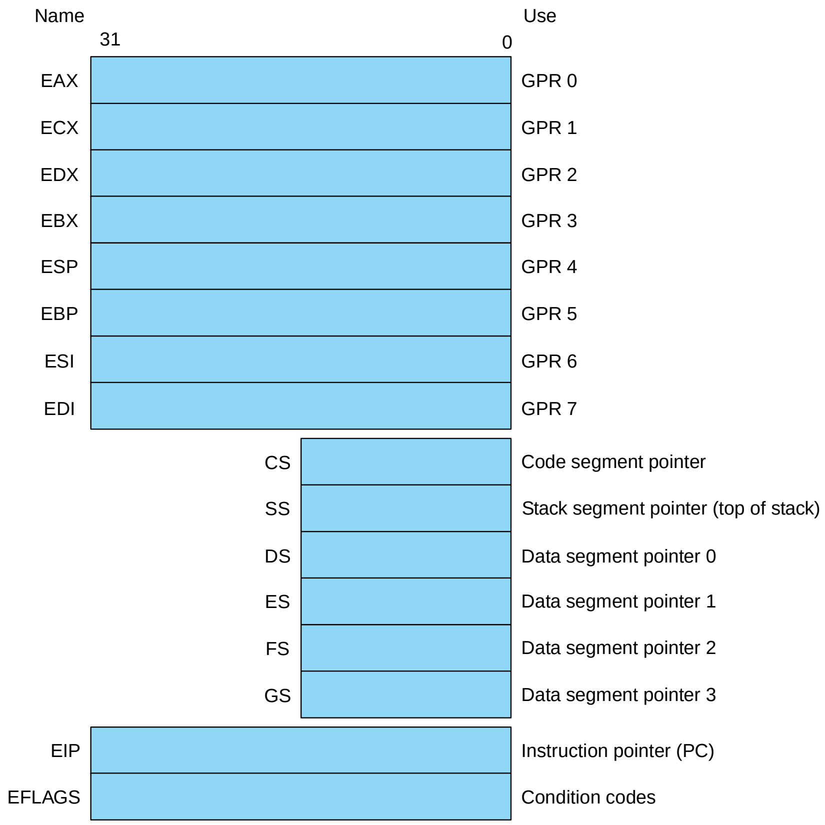
IA-32 Overview

- **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 $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1) # ≤16-bit` `# displacement`

Base plus scaled index	The address is
	Base + (2^{Scale} x Index)
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is
	Base + (2^{Scale} x Index) + displacement
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($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 multiplies 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 `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (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-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW 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
MOVSL	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.)

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!