Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress       #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return

- Machine format (J format):

0x03	26 bit address

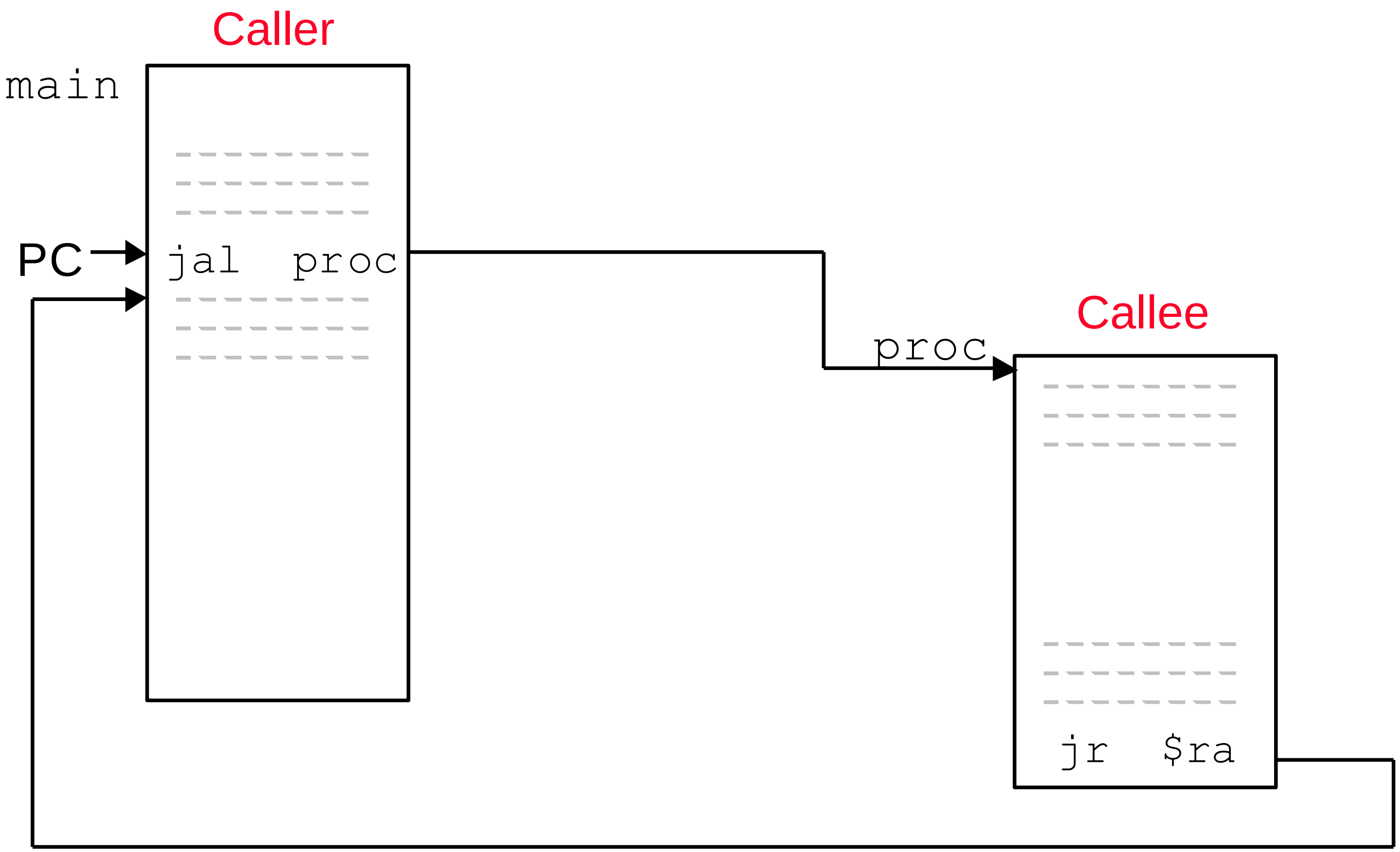
- Then can do procedure return with

```
jr    $ra                    #return
```

- Instruction format (R format):

0	31				0x08

Illustrating a Procedure Call



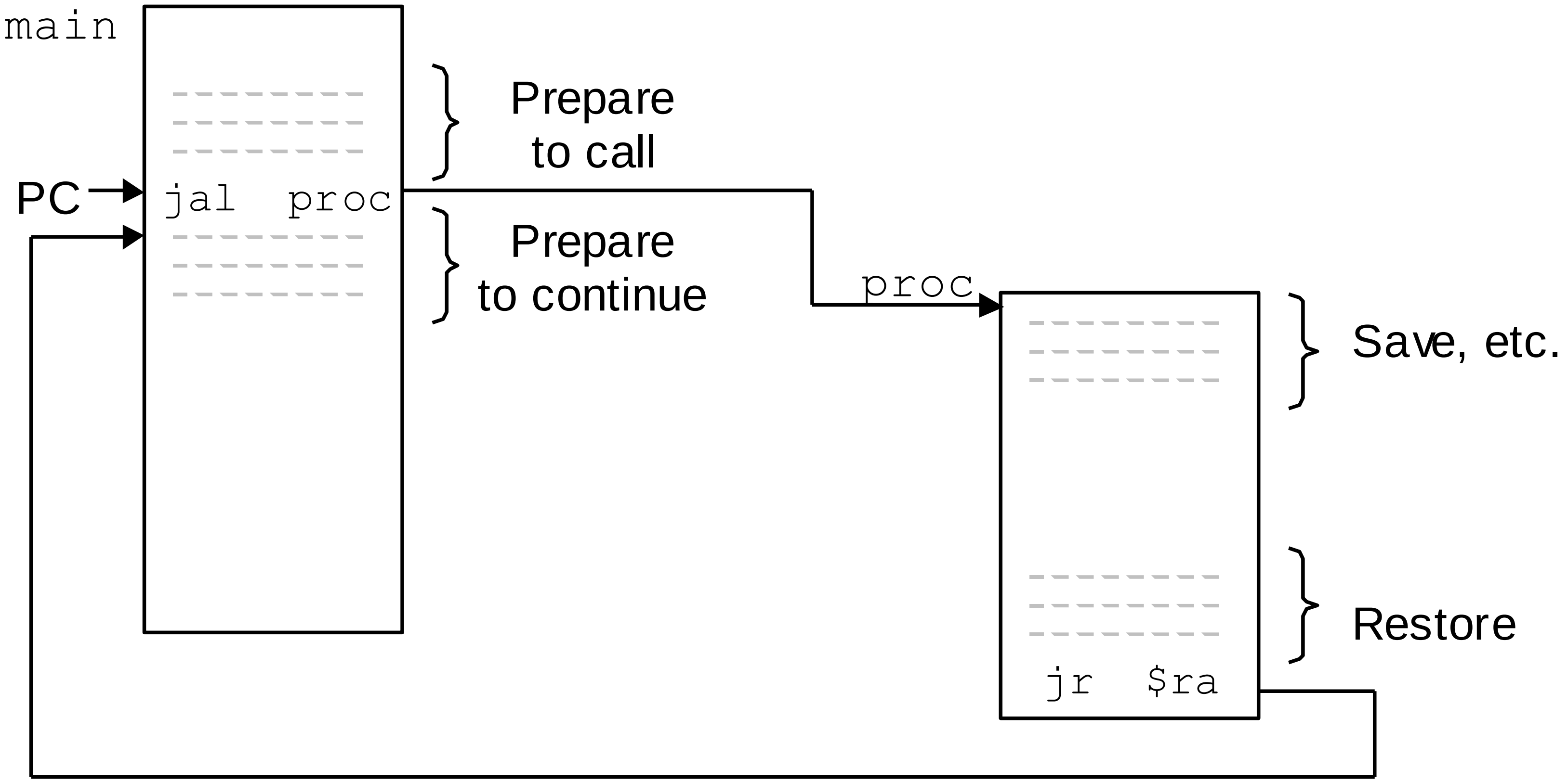
Instructions for accessing procedures

How to pass arguments and get return value?

Six Steps in the Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee (jal)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller (jr)
   - `$ra`: one return address register to return to the point of origin

Illustrating a Procedure Call

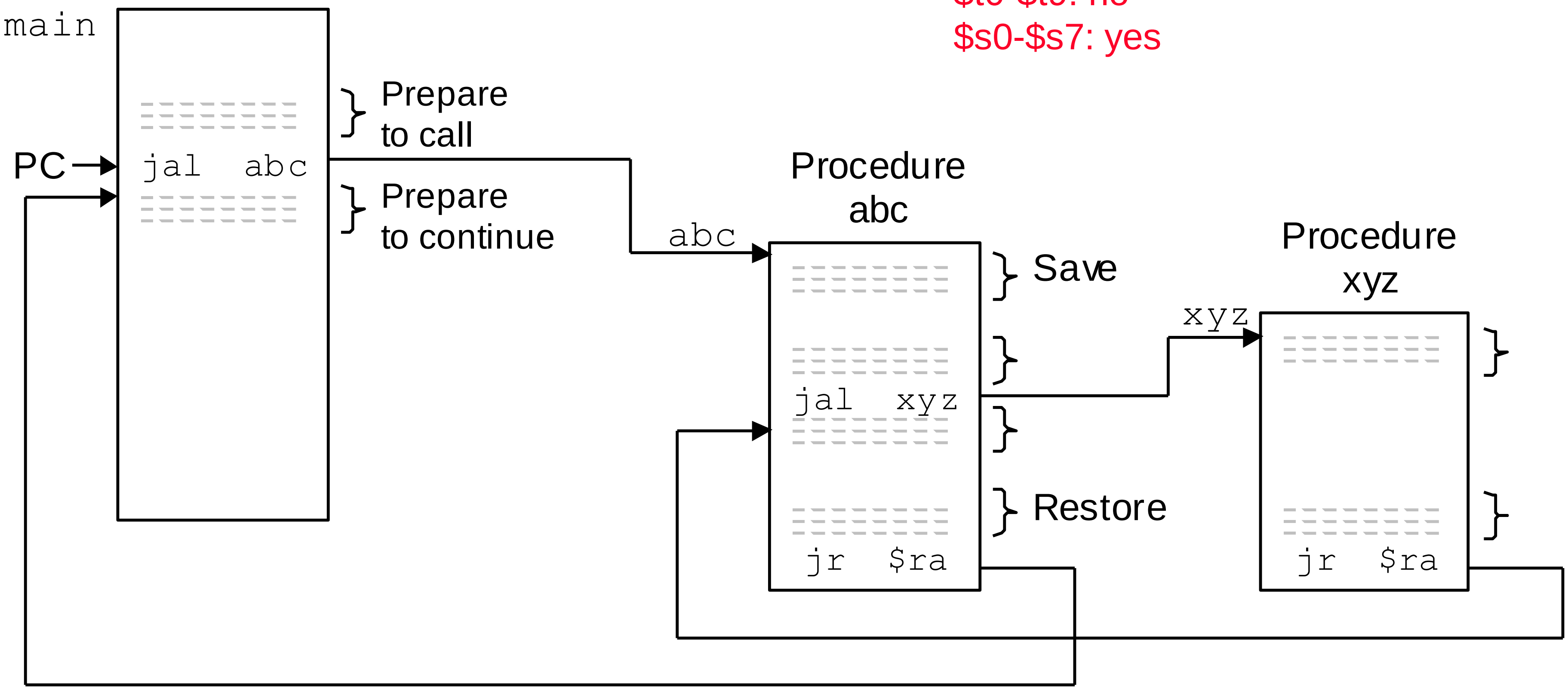


Relationship between the main program and a procedure.

How can main and proc share the same registers?

Nested Procedure Calls

Registers preserved on call
$t0-$t9: no
$s0-$s7: yes



Example of nested procedure calls.

Procedure that does not call another proc.

- C code:

```
int leaf_example (int g, h, i, j)
{

    int f;
    f = (g + h) - (i + j);
    return f;
}
```

  - g, h, i, j stored in \$a0, \$a1, \$a2, \$a3
  - f in \$s0 (need to be saved)
  - \$t0 and \$t1 used for temporary data, also need to be saved
  - Result in \$v0

Sample code

```
leaf_example:
        addi  $sp, $sp, -12     # room for 3 items
        sw    $t1, 8($sp)       # save $t1
        sw    $t0, 4($sp)       # save $t0
        sw    $s0, 0($sp)       # save $s0
        add   $t0, $a0, $a1     # $t0 = g+h
        add   $t1, $a2, $a3     # $t1 = i+j
        sub   $s0, $t0, $t1     # $s0 = (g+h)-(i+j)
        add   $v0, $s0, $zero   # return value in $v0
        lw    $s0, 0($sp)       # restore $s0
        lw    $t0, 4($sp)       # restore $t0
        lw    $t1, 8($sp)       # restore $t1
        addi  $sp, $sp, 12      # shrink stack
        jr    $ra               # return to caller
```

Stack usage

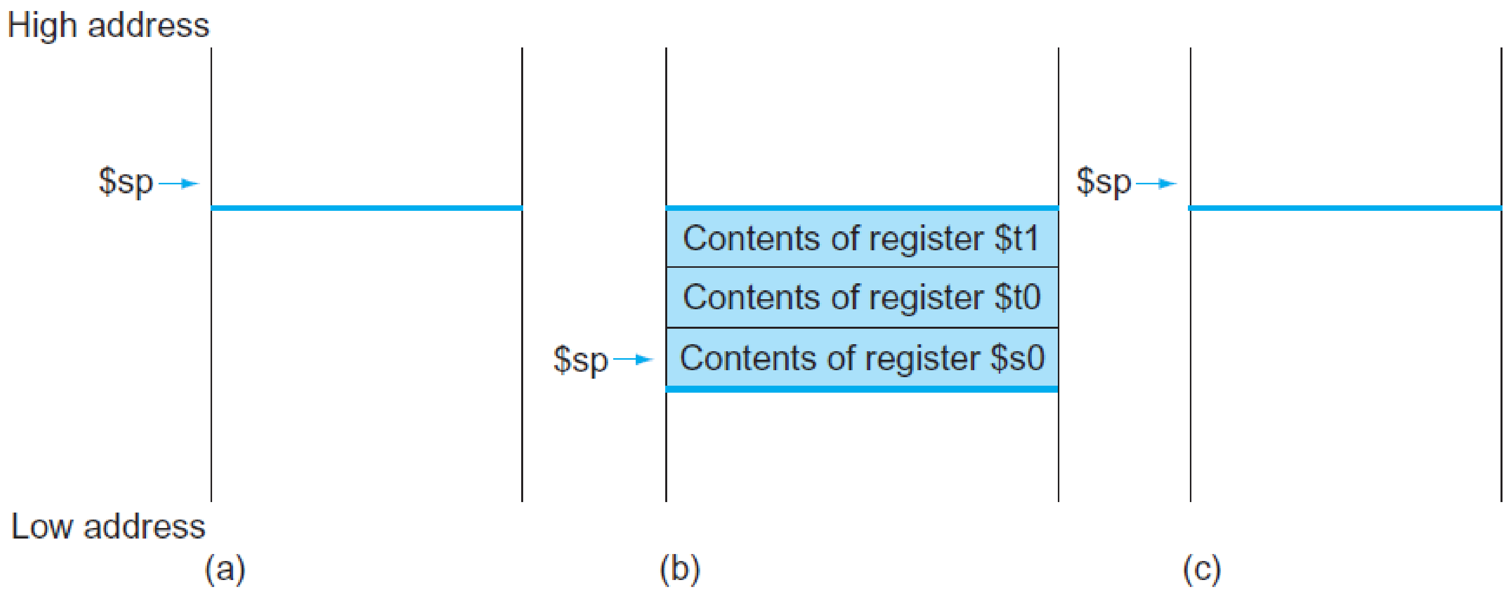


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Procedure with nested proc.

- C code:

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

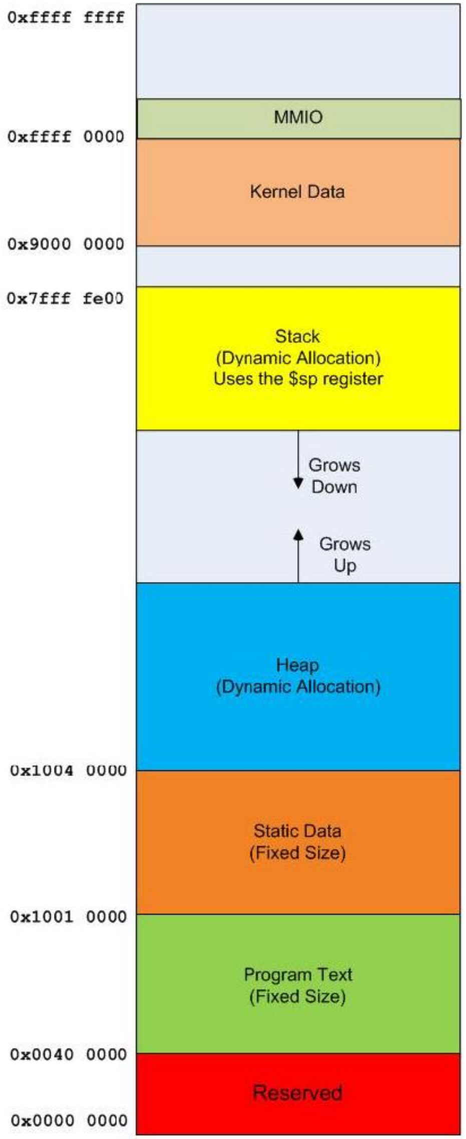
  - n in $a0
  - Result in $v0

Sample code

fact:				
	addi	$sp, $sp, -8	#2 items in stack
	sw	$ra, 4($sp)	#save return address
	sw	$a0, 0($sp)	#and current n
	slti	$t0, $a0, 1	#check base case
	beq	$t0, $zero, L1	#
	addi	$v0, $zero, 1	#value 1 for base case
	addi	$sp, $sp, 8	#then shrink stack
	jr	$ra	#and return
L1:	addi	$a0, $a0, -1	#otherwise reduce n
	jal	fact	#then call fact again
	lw	$a0, 0($sp)	#restore n
	lw	$ra, 4($sp)	#and return address
	addi	$sp, $sp, 8	#shrink stack
	mul	$v0, $a0, $v0	#value for normal case
	jr	$ra	#and return

MIPS memory configuration

- Program text: stores machine code of program, declared with *.text*
- Static data: data segment, declared with *.data*
- Heap: for dynamic allocation
- Stack: for local variable and dynamic allocation via push/pop
- Kernel: for OS's use
- MIMO: memory mapped IO for accessing input/output devices



Working with 32 bit immediates and addresses

- Operations that needs 32-bit literals
  - Loading 32-bit integers to registers
  - Loading variable addresses to registers
- I-format instructions only support 16-bit literals → combine two instructions
- Example: load the value 0x3D0900 into $s0

```
lui $s0, 0x003D          #$s0 ← 0x003D0000
ori #s0, $s0, 0x0900  #$s0 ← 0x003D0900
```

- Pseudo-instructions: combination of real instructions, for convenience
  - li, la, move…
  - bge, bgt, ble…

Accessing characters and string

- Accessing characters

  lb $s0, 0($s1)   #load byte with sign-extension

  lbu $s0, 0($s1)   #load byte with zero-extension

  sb $s0, 0($s1)   #store LSB to memory

- String is accessed as array of characters
- Example: string copy

```
void strcpy (char x[], char y[])
{
    int i = 0;
    while ((x[i] = y[i]) != ‘\0’)
        i += 1;
}
```

Accessing characters and string

```
#x and y are in $a0 and $a1, i in $s0
strcpy:
    addi $sp,$sp,-4         # adjust stack for 1 more item
    sw $s0, 0($sp)          # save $s0
    add $s0,$zero,$zero     # i = 0 + 0
L1: add $t1,$s0,$a1         # address of y[i] in $t1
    lbu $t2, 0($t1)         # $t2 = y[i]
    add $t3,$s0,$a0         # address of x[i] in $t3
    sb $t2, 0($t3)          # x[i] = y[i]
    beq $t2,$zero,L2        # if y[i] == 0, go to L2
    addi $s0, $s0,1         # i = i + 1
    j L1                    # go to L1
L2: lw $s0, 0($sp)          # y[i] == 0: end of string.
                            # Restore old $s0
    addi $sp,$sp,4          # pop 1 word off stack
    jr $ra                  # return
```

Interchange sort function

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1)
    {
        for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j-=1)
        {
            swap(v,j);
        }
    }
}
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Sorting function

Procedure body

```
swap: sll   $t1, $a1, 2        # reg $t1 = k * 4
      add   $t1, $a0, $t1      # reg $t1 = v + (k * 4)
                               # reg $t1 has the address of v[k]
      lw    $t0, 0($t1)        # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)        # reg $t2 = v[k + 1]
                               # refers to next element of v
      sw    $t2, 0($t1)        # v[k] = reg $t2
      sw    $t0, 4($t1)        # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr    $ra                # return to calling routine
```

Saving registers

	sort:	addi	$sp,$sp, -20	# make room on stack for 5 registers
		sw	$ra, 16($sp)# save $ra on stack	
		sw	$s3,12($sp)	# save $s3 on stack
		sw	$s2, 8($sp)# save $s2 on stack	
		sw	$s1, 4($sp)# save $s1 on stack	
		sw	$s0, 0($sp)# save $s0 on stack	

Procedure body

Move parameters		move	$s2, $a0 # copy parameter $a0 into $s2 (save $a0)	
		move	$s3, $a1 # copy parameter $a1 into $s3 (save $a1)	
Outer loop		move	$s0, $zero# i = 0	
	for1tst:	slt	$t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)	
		beq	$t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)	
Inner loop		addi	$s1, $s0, -1# j = i - 1	
	for2tst:	slti	$t0, $s1,0 #reg$t0=1if$s1<0(j<0)	
		bne	$t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)	
		sll	$t1, $s1, 2# reg $t1 = j * 4	
		add	$t2, $s2, $t1# reg $t2 = v + (j * 4)	
		lw	$t3, 0($t2)# reg $t3 = v[j]	
		lw	$t4, 4($t2)# reg $t4 = v[j + 1]	
		slt	$t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3	
		beq	$t0, $zero, exit2# go to exit2 if $t4 Š $t3	
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1 # 2nd parameter of swap is j	
		jal	swap	# swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1# j -= 1	
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop

Restoring registers

	exit1:	lw	$s0, 0($sp)	# restore $s0 from stack
		lw	$s1, 4($sp)# restore $s1 from stack	
		lw	$s2, 8($sp)# restore $s2 from stack	
		lw	$s3,12($sp)	# restore $s3 from stack
		lw	$ra,16($sp)	# restore $ra from stack
		addi	$sp,$sp, 20	# restore stack pointer

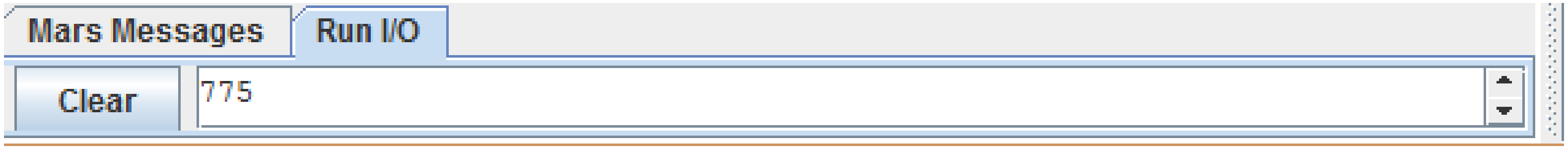
Procedure return

		jr	$ra	# return to calling routine

syscall

- **Print decimal** integer to standard output (the console).
- Argument(s):
  - $v0 = 1
  - $a0 = number to be printed
- Return value: none

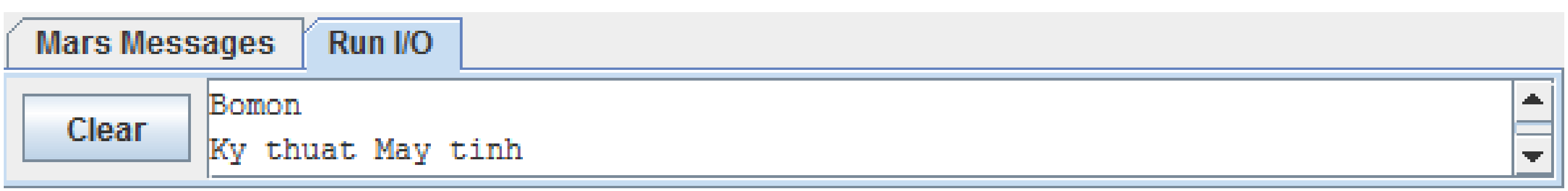
```
li   $v0, 1             # service 1 is print integer
li   $a0, 0x307         # the interger to be printed is 0x307
syscall                 # execute
```



syscall

- **Print string** to standard output (the console).
- Argument(s)
  - $v0 = 1
  - $a0 = address of null terminated string to print
- Return value: none

```
.data
Message: .asciiz "Bomon \nKy thuat May tinh"
.text
   li  $v0, 4
   la  $a0, Message
   syscall
```



syscall

- **Read integer** from standard input (the console).
- Argument
  - $v0 = 5
- Return value
  - $v0 = contains integer read

```
li      $v0, 5
syscall
```

syscall

- **Read string** from standard input
- Argument(s):
  - $v0 = 8
  - $a0 = address of input buffer
  - $a1 = maximum number of characters to read
- Return value: none
- Note: for specified length n, string can be no longer than n-1.
  - If less than that, adds newline to end.
  - In either case, then pads with null byte
- String can be declared with *.space*

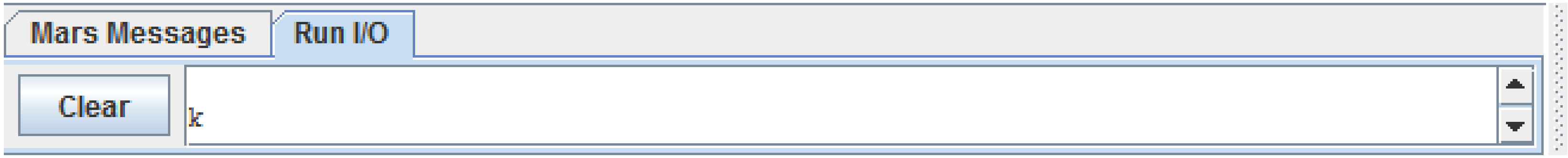
syscall

```
.data
Message: .space 100    # string with max len = 99
.text
   li  $v0, 8
   la  $a0, Message
   li  $a1, 100
   syscall
```

syscall

- **Print** a **character** to standard output.
- Arguments
  - $v0 = 11
  - $a0 = character to print (at LSB)
- Return value: none

```
li  $v0, 11
li  $a0, 'k'
syscall
```



syscall

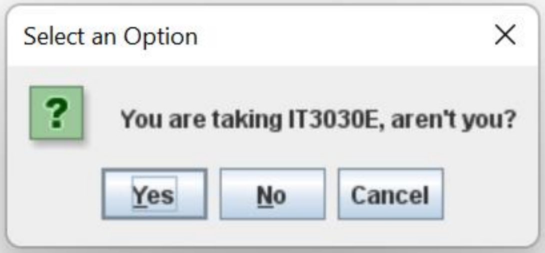
- **Read** a **character** from standard input.
- Argument(s):
  - $v0 = 12
- Return value:
  - $v0 contains the character read

syscall

- ConfirmDialog
- Argument(s):
  - $v0 = 50
  - $a0 = address of the null-terminated message string
- Return value: $a0 = value of selected option

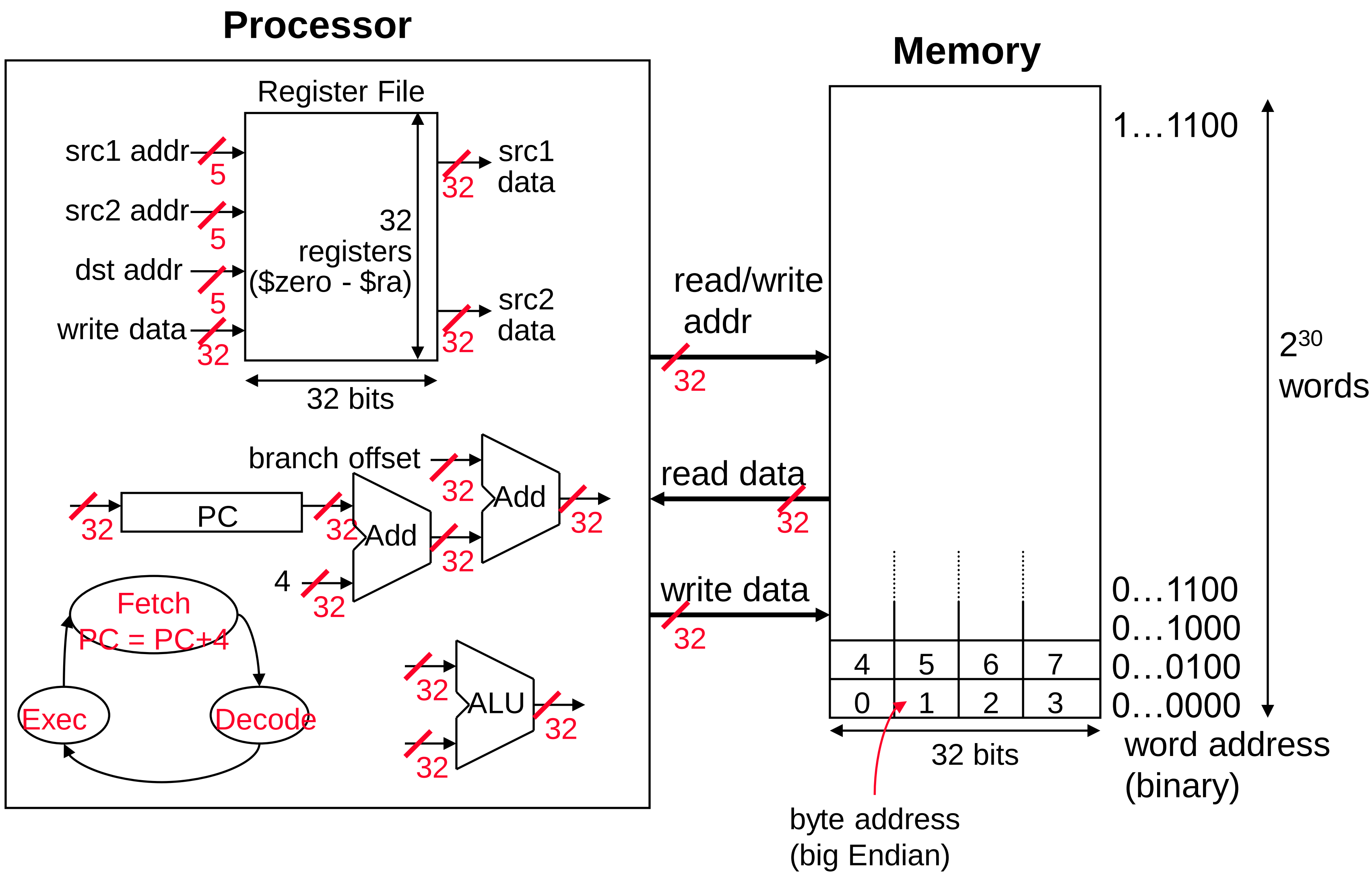
  0: Yes 1: No 2: Cancel

```
.data
Message: .asciiz "You are taking IT3030E, aren't you?"
.text
   li   $v0, 50
   la   $a0, Message
   syscall
```



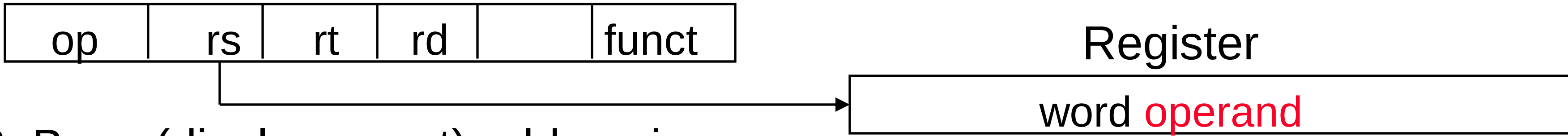
Exercise

MIPS Organization

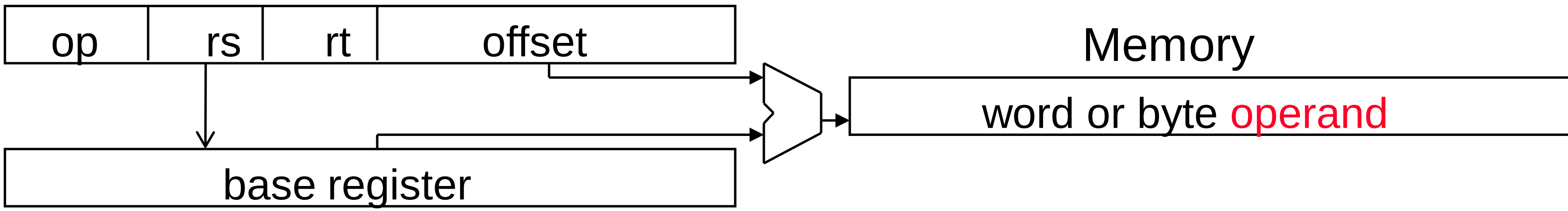


Addressing Modes Illustrated

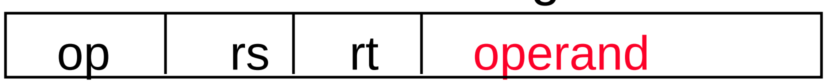
1. Register addressing



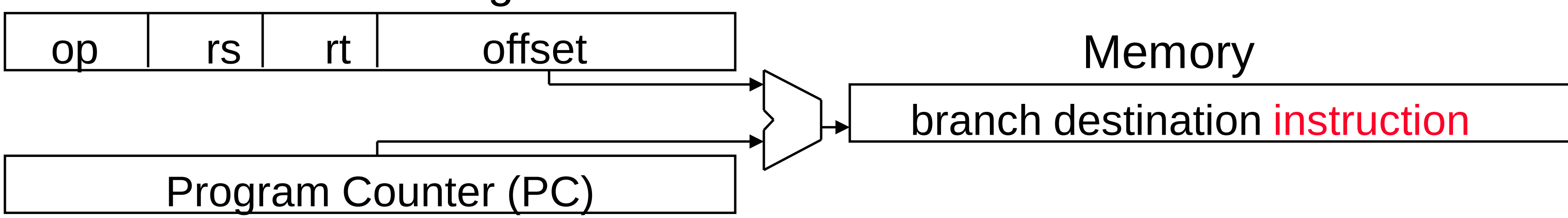
2. Base (displacement) addressing



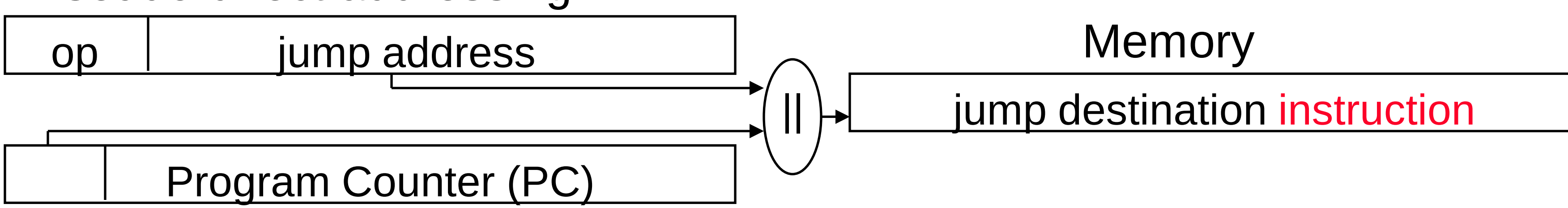
3. Immediate addressing



4. PC-relative addressing



5. Pseudo-direct addressing



Summary

- Provided one problem to be solved by computer
  - Can it be implemented?
  - Can it be programmed?
  - Which CPU is suitable?
- Metric of performance
  - How many bytes does the program occupy in memory?
  - How many instructions are executed?
  - How many clocks are required per instruction?
  - How much time is required to execute the program?

➔ Largely depend on Instruction Set Architecture (ISA)