Computer Organization

Lab4 MIPS(3)

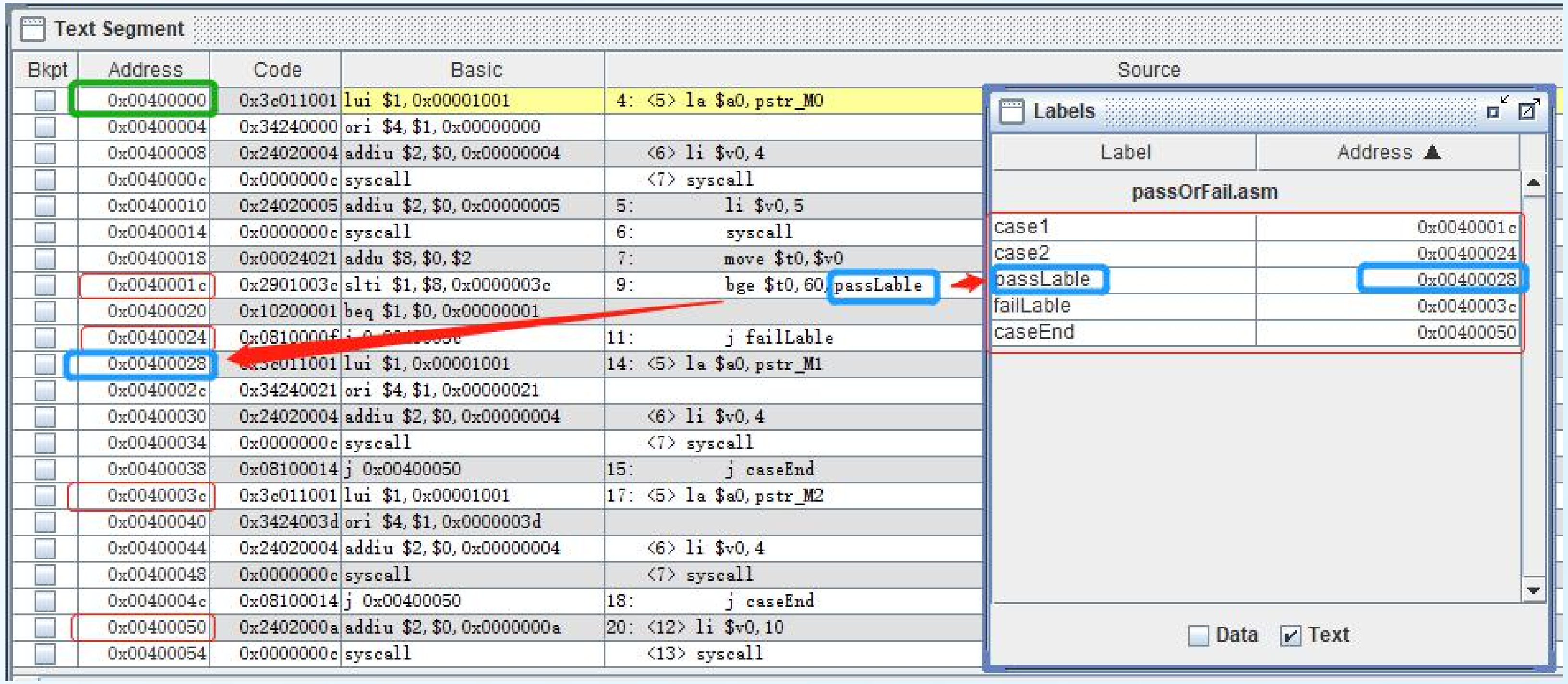
Instruction execution

Topics

- Instruction execution
  - PC register
  - PC+4 vs Jump（Branch、Loop）
- Procedure/Function
  - Defination、Call、Return

How does CPU execute the instructions ?

- Before executing an instruction, CPU **fetches** it from memory according to its address, then **analyze**, finally **execute**.
- Register **PC** stores **the address of the instruction** which is to be executed.



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1, 0x00001001	4: <5> la $a0, pstr_M0
	0x00400004	0x34240000	ori $4, $1, 0x00000000	
	0x00400008	0x24020004	addiu $2, $0, 0x00000004	<6> li $v0, 4
	0x0040000c	0x0000000c	syscall	<7> syscall
	0x00400010	0x24020005	addiu $2, $0, 0x00000005	5: li $v0, 5
	0x00400014	0x0000000c	syscall	6: syscall
	0x00400018	0x00024021	addu $8, $0, $2	7: move $t0, $v0
	0x0040001c	0x2901003c	slti $1, $8, 0x0000003c	9: bge $t0, 60, passLable
	0x00400020	0x10200001	beq $1, $0, 0x00000001	
	0x00400024	0x0810000f	[illegible]	11: j failLable
	0x00400028	[illegible]	lui $1, 0x00001001	14: <5> la $a0, pstr_M1
	0x0040002c	0x34240021	ori $4, $1, 0x00000021	
	0x00400030	0x24020004	addiu $2, $0, 0x00000004	<6> li $v0, 4
	0x00400034	0x0000000c	syscall	<7> syscall
	0x00400038	0x08100014	j 0x00400050	15: j caseEnd
	0x0040003c	0x3c011001	lui $1, 0x00001001	17: <5> la $a0, pstr_M2
	0x00400040	0x3424003d	ori $4, $1, 0x0000003d	
	0x00400044	0x24020004	addiu $2, $0, 0x00000004	<6> li $v0, 4
	0x00400048	0x0000000c	syscall	<7> syscall
	0x0040004c	0x08100014	j 0x00400050	18: j caseEnd
	0x00400050	0x2402000a	addiu $2, $0, 0x0000000a	20: <12> li $v0, 10
	0x00400054	0x0000000c	syscall	<13> syscall

Labels

Label	Address ▲
passOrFail.asm	
case1	0x0040001c
case2	0x00400024
passLable	0x00400028
failLable	0x0040003c
caseEnd	0x00400050

Data ☑ Text

Q1: What is the difference between the addresses of two adjacent instructions?

Q2: How does the value in **$PC** change?

Conditional Branch & Unconditional Jump

- **Conditional branch**
  - **beq \$t0,\$t1,lable** *# branch to instruction addressed by the label if \$t1 and \$t2 are equal*
  - **bne \$t0,\$t1,lable** *# branch to instruction addressed by the label if \$t1 and \$t2 are NOT equal*
  - **blt, ble, bltu, bleu, bgt, bge, bgtu, bgeu**
- **Unconditional jump**

Jump (j)	Unconditionally jumps to a specified location. A symbolic address or a general register specifies the destination. The instruction j \$31 returns from the a jal call instruction.
Jump And Link (jal)	Unconditionally jumps to a specified location and puts the return address in a general register. A symbolic address or a general register specifies the target location. By default, the return address is placed in register \$31. If you specify a pair of registers, the first receives the return address and the second specifies the target. The instruction jal procname transfers to procname and saves the return address. For the two-register form of the instruction, the target register may not be the same as the return-address register. For the one-register form, the target may not be \$31.

Branch

Are the running results of two demos the same ?

*Modify **the more efficient one** without changing the result by using **ble** or **blt** instead of **bge** .*

```
.include "macro_print_str.asm"
.text
        print_string("please input your score (0~100):")
        li $v0,5
        syscall
        move $t0,$v0
case1:
        bge $t0,60,passLable
case2:
        j failLable

passLable:
        print_string("\nPASS (exceed or equal 60) ")
        j caseEnd
failLable:
        print_string("\nFaild(less than 60)")
        j caseEnd
caseEnd:
        end
```

```
.include "macro_print_str.asm"
.text
        print_string("please input your score (0~100):")
        li $v0,5
        syscall
        move $t0,$v0
case1:
        bge $t0,60,passLable
        j case2
case2:
        j failLable

passLable:
        print_string("\nPASS (exceed or equal 60) ")
        j caseEnd
failLable:
        print_string("\nFaild(less than 60)")
        j caseEnd
caseEnd:
        end
```

Loop

Compare the operations of loop which calculats the sum from 1 to 10 in java and MIPS.

Code in Java:

```
public class CalculateSum{
   public static void main(String [] args){
      int i = 0;
      int sum = 0;
      for(i=0;i<=10;i++)
               sum = sum + i;
      System.out.print("The sum from 1 to 10 : " + sum );
   }
}
```

Code in MIPS:

```
.include "macro_print_str.asm"
.data
        #....
.text
        add $t1,$zero,$zero     #sum=0
        addi $t0,$zero,0        #i=0
        addi $t7,$zero,10
calcu:
        addi $t0,$t0,1          #i++
        add $t1,$t1,$t0         #sum+=i
        bgt $t7,$t0,calcu       #while(t7>t0)

        print_string ("The sum from 1 to 10 : ")
        move $a0,$t1
        li $v0,1
        syscall

        end
```

Demo #1

The following code is expected to get 10 integers from the input device, and print it as the following sample.
Is the result of the code running the same as the screenshot in the lower right corner?
If not, what happened ?

#piece 1/3

```
.include "macro_print_str.asm"
.data
        arrayx:     .space      10
        str:        .asciiz     "\nthe arrayx is:"
.text
main:
        print_string("please input 10 integers: ")

        add $t0,$zero,$zero
        addi $t1,$zero,10
        la $t2,arrayx
```

#piece 2/3

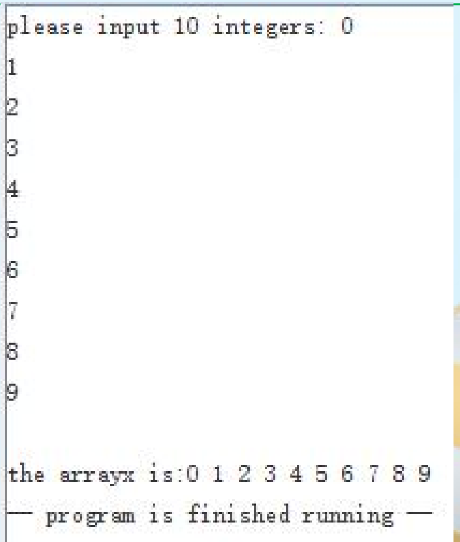
```
loop_r:
        li $v0,5
        syscall
        sw $v0,($t2)
        addi $t2,$t2,4
        addi $t0,$t0,1
        bne $t0,$t1,loop_r

        la $a0,str
        li $v0,4
        syscall

        addi $t0,$zero,0
        la $t2,arrayx
```

#piece 3/3

```
loop_w:
        lw $a0,($t2)
        li $v0,1
        syscall
        print_string(" ")
        addi $t2,$t2,4
        addi $t0,$t0,1
        bne $t0,$t1,loop_w
        end
```



The function of following code is to get 5 integers from input device, and find the min value and max value of them.
There are 4 pieces of code, write your code based on them.
Can the code find the real min and max among the 5 integers?

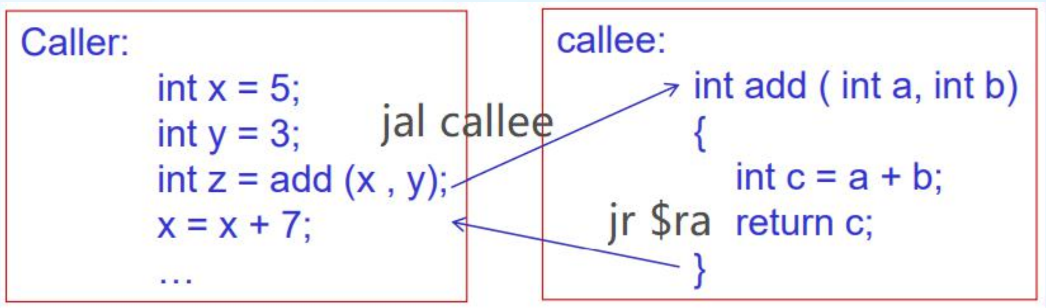
```
#piece ?/4
.include "macro_print_str.asm"
.data
	min: .word 0
	max: .word 0
.text
	lw $t0,min		#min
	lw $t1,max		#max
	li $t7,5
	li $t6,0
	print_string("please input 5 integer:")
loop:
	li $v0,5
	syscall
	bgt $v0,$t1,get_max
	j get_min
```

```
#piece ?/4
get_max:
	move $t1,$v0
	j get_min
get_min:
	bgt $v0,$t0,judge_times
	move $t0,$v0
	j judge_times
```

```
#piece ?/4
judge_times:
	addi $t6,$t6,1
	bgt $t7,$t6,loop
```

```
#piece ?/4
	print_string("min : ")
	move $a0,$t0
	li $v0,1
	syscall
	print_string("max : ")
	move $a0,$t1
	li $v0,1
	syscall
	end
```

Procedure/Function



- **jal** callee_lable
  - Used in **Caller** while calling the function
  - **Save** the address of the next instruction in register **$ra**, **Unconditionally jump** to the instruction at callee_lable
- **lw** / **sw** with **$sp** in callee
  - Protects register data by using **stack** in memory
- **jr $ra**
  - Used in **Callee** while returning to the caller
  - **Unconditionally jump** to the instruction according the address stored in register **$ra**

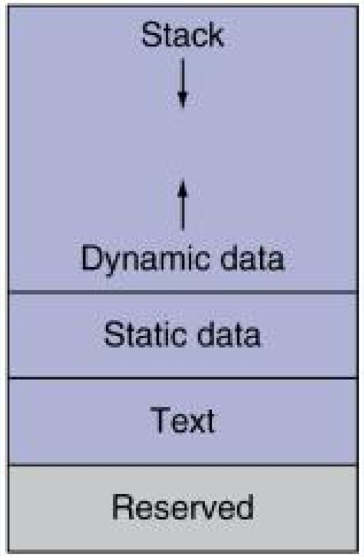
Stack Segment

Stack segment:
The portion of memory used by a program to hold procedure call frames.

The program ***stack segment***, resides **at the top of the virtual address space**.

Like dynamic data, the maximum size of a program's stack is not known in advance.

As the program **pushes values on the stack**, the operating system **expands** the stack segment **down, toward the data segment**.



Demo #2

What's the value of $ra while jumping and linking to the print_string (at line 12,15,18,21)?

```
print_string:
        addi $sp,$sp,-8
        sw $a0,4($sp)
        sw $v0,0($sp)

        addi $v0,$zero,4
        syscall

        lw $v0,0($sp)
        lw $a0,4($sp)
        addi $sp,$sp,8

        jr $ra
```

Text Segment

Bkpt	Address	Code	Basic		Source
	0x0040001c	0x0c100013	jal 0x0040004c	12:	jal print_string
	0x00400020	0x3c011001	lui $1, 0x00001001	14:	la $a0, tdata
	0x00400024	0x34240000	ori $4, $1, 0x00000000		
	0x00400028	0x0c100013	jal 0x0040004c	15:	jal print_string
	0x0040002c	0x3c011001	lui $1, 0x00001001	17:	la $a0, str2
	0x00400030	0x3424001e	ori $4, $1, 0x0000001e		
	0x00400034	0x0c100013	jal 0x0040004c	18:	jal print_string
	0x00400038	0x3c011001	lui $1, 0x00001001	20:	la $a0, tdata+3
	0x0040003c	0x34240003	ori $4, $1, 0x00000003		
	0x00400040	0x0c100013	jal 0x0040004c	21:	jal print_string
	0x00400044	0x2002000a	addi $2, $0, 0x0000000a	23:	addi $v0, $zero, 10
	0x00400048	0x0000000c	syscall	24:	syscall

pay attention to the value of $pc

Demo #2

```
.data                                  #piece 1/4
        tdata: .space  6
        str1:  .asciiz "the 1st char's ascii value is: "
        str2:  .asciiz "\nthe last char's ascii value is: "
.text
        la $a0,tdata
        addi $a1,$zero,6
        addi $v0,$zero,8
        syscall
```

read string	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read

```
la $a0,str1            #piece 2/4
jal print_string
la $a0,tdata
jal print_ascii_value
la $a0,str2
jal print_string
la $a0,tdata+4
jal print_ascii_value

addi $v0,$zero,10
syscall
```

```
print_ascii_value:                     #piece 3/4
        addi $sp,$sp,-8
        sw $a0,4($sp)
        sw $v0,0($sp)
        lw $a0,($a0)
        addi $v0,$zero,1
        syscall
        lw $v0,0($sp)
        lw $a0,4($sp)
        addi $sp,$sp,8
        jr $ra
```

```
print_string:                          #piece 4/4
        addi $sp,$sp,-8
        sw $a0,4($sp)
        sw $v0,0($sp)

        addi $v0,$zero,4
        syscall
        lw $v0,0($sp)
        lw $a0,4($sp)
        addi $sp,$sp,8
        jr $ra
```

*Q1. Is it ok to remove the push and pop processing of **\$a0** and **\$v0** on the stack in "print_ascii_value" and "print_string" ? why?*

Q2. If the input string is "abcde" , could this demo get the ascii value of the 1st char and the last char in the string? If not, modify the code to make it work.

Recursion

"fact" is a function to calculate the Calculate the factorial.

Code in C:

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

*Q1. While calculate **fact(6)**, how many times does push and pop processing on stack happend? how many bytes are used in the stack?*

*Q2. How does the value of $a0 and $v0 change when calculate **fact(6)**?*

Code in MIPS:

```
fact:
        addi   $sp,$sp,-8          #adjust stack for 2 items
        sw     $ra, 4($sp)         #save the return address
        sw     $a0, 0($sp)         #save the argument n

        slti   $t0,$a0,1           #test for n<1
        beq    $t0,$zero,L1        #if n>=1,go to L1

        addi  $v0,$zero,1          #return 1
        addi  $sp,$sp,8            #pop 2 items off stack
        jr          $ra            #return to caller

L1:     addi  $a0,$a0,-1           #n>=1; argument gets(n-1)
        jal         fact           #call fact with(n-1)

        lw     $a0,0($sp)          #return from jal: restore argument n
        lw   $ra,4($sp)            #restore the return address
        addi   $sp,$sp,8           #adjust stack pointer to pop 2 items

        mul    $v0,$a0,$v0         #return n*fact(n-1)

        jr     $ra                 #return to the caller
```

Practice

1. Print out a 9*9 multiplication table.
    1. Define a function to print a*b = c , the value of “a” is from parameter $a0,the value of “b” is from parameter $a1.
    2. Less syscall is better(more effective).
2. Get a positive integer from input, calculate the sum from 1 to this value by using recursion, output the result in hexdecimal.
3. Get a positive integer from input, output an integer in reverse order using loop and recursion seperately.
4. Answer the questiones on page 5, 8, 12 and13.

Tips

caller-saved register A register saved by the routine being called.
callee-saved register A register saved by the routine making a procedure call.

- Registers **$a0~$a3** are used to **pass the first four arguments to routines** (remaining arguments are passed on the stack).
- Registers **$v0~$v1** are used to **return values from functions**.
- Registers **$t0~ $t9** are ***caller-saved registers*** that are used to hold temporary quantities that need not be preserved across calls.
- Registers **$s0~$s7** are ***callee-saved registers*** that hold long-lived values that should be preserved across calls.
- Register **$sp (29)** is the **stack pointer**, which points to the last location on the stack.
- Register **$fp (30)** is the frame pointer.
- The jal instruction writes register **$ra (31)**, the return address from a procedure call.