Computer Organization

Lab3 MIPS(2)

Execution order of Instructions

Topics

- Execution order of instructions
  - PC register， the updation of PC register
  - Conditional jump vs Unconditional jump
    - beq, bne vs j
- Function
  - Defination、Call、Return
    - jal, jr （$31 / $ra）

'Who' determine the execution order of instructions

- The CPU takes the value of the PC register as the address and fetches the corresponding instruction from the memory.
  - **PC register maintains the address of the instruction currently being executed**.
  - **After** the current instruction is executed, the value of the PC register will be **updated** to determine the next instruction to be executed.

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020008	addiu $2, $0, 0x00000008	7: li $v0, 8 #to get a string
	0x00400004	0x3c011001	lui $1, 0x00001001	8: la $a0, sid
	0x00400008	0x34240008	ori $4, $1, 0x00000008	
	0x0040000c	0x24050009	addiu $5, $0, 0x00000009	9: li $a1, 9
	0x00400010	0x0000000c	syscall	10: syscall
	0x00400014	0x24020004	addiu $2, $0, 0x00000004	13: li $v0, 4 #to print a string
	0x00400018	0x3c011001	lui $1, 0x00001001	14: la $a0, s1
	0x0040001c	0x34240000	ori $4, $1, 0x00000000	
	0x00400020	0x0000000c	syscall	15: syscall
	0x00400024	0x2402000a	addiu $2, $0, 0x0000000a	16: li $v0, 10 #to exit
	0x00400028	0x0000000c	syscall	17: syscall

pc		0x00400000
pc		0x00400004
pc		0x00400008
pc		0x0040000c

pc		0x00400028

How to update the value of PC register?

- Check if the current instruction is non-jump
  - If the current instruction is non-jump instruction: PC = PC+4
  - If the current instruction is jump instruction
    - If the current instruction is unconditional jump   pc = destination address
    - If the current instruction is conditional jump
      - If the condition is met： PC = destination address
      - If the condition is not met: PC = PC + 4

Conditional Jump

basic instruction	usage
beq $t0,$t1,labelx	*branch to instruction addressed by the labelx if $t0 and $t1 are equal*
bne $t0,$t1,labelx	*branch to instruction addressed by the labelx if $t0 and $t1 are NOT equal*

pseudo instruction	basic instruction	usage
blt $t0,$t1,lable	slt **$1**, $t0, $t1 bne **$1**,$0, lable	*# branch to instruction addressed by the label if $t0 is less than $t1, data in $t0 and $t1 are taken as signed number*
ble $t0,$t1,lable	slt **$1**,$t1,$t0 beq **$1**,$0,lable	*# branch to instruction addressed by the label if $t0 is less or equal than $t1, data in $t0 and $t1 are taken as signed number*
bltu $t0,$t1,lable	sltu **$1**, $t0, $t1 bne **$1**,$0, lable	*# branch to instruction addressed by the label if $t0 is less than $t1, data in $t0 and $t1 are taken as unsigned number*
bleu $t0,$t1,lable	sltu **$1**,$t1,$t0 beq **$1**,$0,lable	*# branch to instruction addressed by the label if $t0 is less or equal than $t1, data in $t0 and $t1 are taken as unsigned number*
bgt, bge, bgtu, bgeu……		

Unconditional Jump

- **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.

Registers | Coproc 1 | Coproc

Name	Number
$zero	0
$at	1
$v0	2
$v1	3
$a0	4
$a1	5
$a2	6
$a3	7
$t0	8
$t1	9
$t2	10
$t3	11
$t4	12
$t5	13
$t6	14
$t7	15
$s0	16
$s1	17
$s2	18
$s3	19
$s4	20
$s5	21
$s6	22
$s7	23
$t8	24
$t9	25
$k0	26
$k1	27
$gp	28
$sp	29
$fp	30
$ra	31

Branch

Are the running results of two demos the same ?
*Modify them without changing the result by using **ble** or **blt** instead*

```
.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
        addi $t0,$zero,0
        addi $t7,$zero,10
calcu:
        addi $t0,$t0,1      #i++
        add $t1,$t1,$t0    #sum+=i
        bgt $t7,$t0,calcu  #if(t7>t0)  t0==t7

        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.
Will the code get desired result?
If not, what happened ?

#piece 1/3

```
.include "macro_print_str.asm"
.data
        arrayx:     .space      10      40
        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 $t0,$t0,1
        addi $t2,$t2,4
        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
```

```
please input 10 integers: 0
1
2
3
4
5
6
7
8
9

the arrayx is:0 1 2 3 4 5 6 7 8 9
-- program is finished running --
```

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 it find the real min and max?

```
#piece ?/4
.include "macro_print_str.asm"
.data
        min: .word 0
        max: .word 0
.text
        lw $t0,min
        lw $t1,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
```

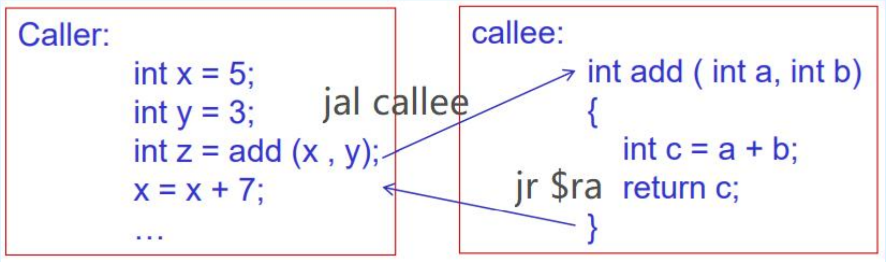
```
#piece ?/4
get_max:
        move $t1,$v0
        j get_min
get_min:
      bgt $v0,$t0,judge_times
        move $t0,$v0
        j judge_times
```

```
#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
```

Function

- **jal** function_lable #jump and link
  - **Save** the address of the next instruction in **register $ra**
  - **Unconditionally jump** to the instruction at function_lable.
  - Used in **caller** while calling the function



- **jr $ra**
  - **Read** the value in **register $ra**
  - **Unconditionally jump** to the instruction according the value in register $ra
  - Used in **callee** while returning to the caller
- **lw / sw** with $sp
  - Protects register data by using **stack** in memory

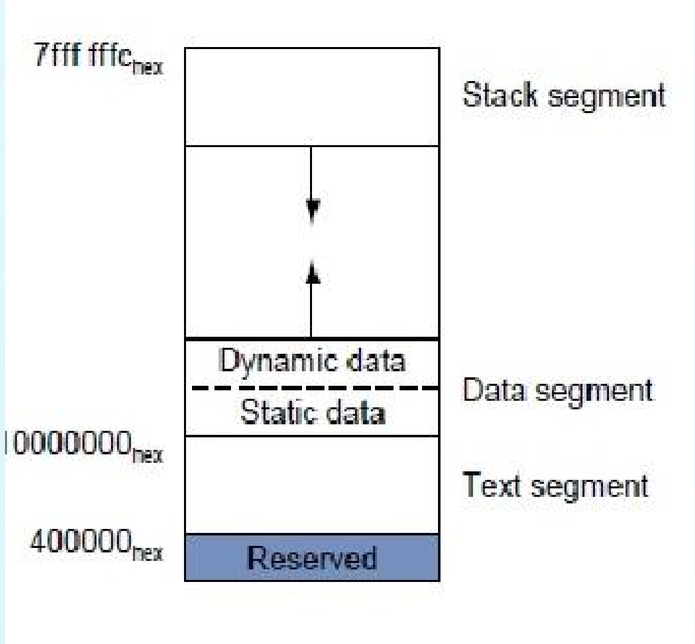
Stack Segment

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** (starting at address 7fffffff $_{hex}$).

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

```
.data                                  #piece 1/3
	tdata: .space 6
	str1:  .asciiz "the orignal string is: "
	str2:  .asciiz "\nthe last two character of the string 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 2/3
jal print_string

	la $a0,tdata
	jal print_string

	la $a0,str2
	jal print_string

	la $a0,tdata+3
	jal print_string

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

```
print_string:    #piece 3/3
	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** on the stack in "print_string" ?*

*Q2. Is it ok to remove the push and pop processing of **\$v0** on the stack in "print_string" ?*

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

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));
}
```

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
```

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

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

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 13,14 and15.

Tips1

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.

Tips2： Arithmetic Instructions(1)

```
add     $t0,$t1,$t2    #  $t0 = $t1 + $t2;   add as signed
                       # (2's complement) integers
sub     $t2,$t3,$t4    #  $t2 = $t3 Ð $t4
addi    $t2,$t3, 5     #  $t2 = $t3 + 5;   "add immediate"
                       # (no sub immediate)
addu    $t1,$t6,$t7    #  $t1 = $t6 + $t7;
addu    $t1,$t6,5      #  $t1 = $t6 + 5;
                       #  add as unsigned integers
subu    $t1,$t6,$t7    #  $t1 = $t6 - $t7;
subu    $t1,$t6,5      #  $t1 = $t6 - 5
                       #  subtract as unsigned integers
```

Tips2: Arithmetic Instructions(2)

```
mult    $t3,$t4    # multiply 32-bit quantities in $t3
                   # and $t4, and store 64-bit
                   # result in special registers Lo
                   # and Hi:  (Hi,Lo) = $t3 * $t4

div     $t5,$t6    # Lo = $t5 / $t6   (integer quotient)
                   # Hi = $t5 mod $t6   (remainder)
mfhi    $t0        # move quantity in special register Hi
                   # to $t0:   $t0 = Hi
mflo    $t1        # move quantity in special register Lo
                   # to $t1:   $t1 = Lo,  used to get at
                   # result of product or quotient
```

Tips3： Shift Operation

Type	Instruction name	description	
shift	**sll** (Shift Left Logical)	**Shifts the contents of a register left** (toward the sign bit) and inserts zeros at the least-significant bit.	The contents of **src1 specify the value to shift**, and the contents of **src2 or the immediate value specify the amount to shift**. If src2 (or the immediate value) is greater than 31 or less than 0, src1 shifts by the result of src2 MOD 32.
	sra (Shift Right Arithmetic)	**Shifts the contents of a register right** (toward the least-significant bit) and inserts the sign bit at the most-significant bit.	
	srl (Shift Right Logical)	**Shifts the contents of a register right** (toward the least-significant bit) and inserts zeros at the most-significant bit.	
rotate	**rol** (Rotate Left)	**Rotates the contents of a register left** (toward the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit.	The contents of **src1 specify the value to shift**, and the contents of **src2 (or the immediate value) specify the amount to shift**. Rotate Left/right puts the result in the destination register. If src2 (or the immediate value) is greater than 31, src1 shifts by the result of src2 MOD 32.
	ror (Rotate Right)	**Rotates the contents of a register right** (toward the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit.	

Tips4: Bit Logic Operation

Instruction name	description
and (AND) *and dst,sr1,sr2(im)*	Computes the **Logical AND** of two values. This instruction ANDs (bit-wise) the contents of src1 with the contents of src2, or it can AND the contents of src1 with the immediate value. The immediate value is **NOT** sign extended. AND puts the result in the destination register.
or (OR) *or dst,sr1,sr2(im)*	Computes the **Logical OR** of two values. This instruction ORs (bit-wise) the contents of src1 with the contents of src2, or it can OR the contents of src1 with the immediate value. The immediate value is **NOT** sign extended. OR puts the result in the destination register
xor (Exclusive-OR) *xor dst,sr1,sr2(im)*	Computes the **XOR** of two values. This instruction XORs (bit-wise) the contents of src1 with the contents of src2, or it can XOR the contents of src1 with the immediate value. The immediate value is **NOT** sign extended. Exclusive-OR puts the result in the destination register
not (NOT) *not dst,src1*	Computes the **Logical NOT** of a value. This instruction complements (bit-wise) the contents of src1 and puts the result in the destination register.
nor (NOT OR) *nor dst,sr1,sr2*	Computes the **NOT OR** of two values. This instruction combines the contents of src1 with the contents of src2 (or the immediate value). NOT OR complements the result and puts it in the destination register.