Data declaration: Program code: Comments : #

.data .text

System calls

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $v0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $v0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $v0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

SPIM提供了一系列syscall操作数

指令数 → $v0

参数 → $a0 ~ $a3 或 $f12 (浮点数)

返回值 → $v0 或 $f0 (浮点数)

Data types & Literals: (MIPS32)

1 word = 32 bit = 4 byte = 2 half word

一个基本寄存器的位宽是 32 bit

MIPS指令转成机器码后都长为32bit = 4byte

- Instruction and Data Storage:
  - Instructions are all 32 bits(1 word)
  - A character requires 1 byte of storage
- Literals:
  - Characters enclosed in single quotes. e.g. 'C'
  - Strings enclosed in double quotes. E.g. "a String"
  - Numbers in code. e.g. 10

Data delaration

```
        name:       storage_type    value(s)
example

var1:       .word   3       # create a single integer:
                            #variable with initial value 3

array1:     .byte   'a','b' # create a 2-element character
                            # array with elements initialized:
                            # to  a  and  b

array2:     .space  40      # allocate 40 consecutive bytes,
                            # with storage uninitialized
                            # could be used as a 40-element
                            # character array, or a
                            # 10-element integer array;
                            # a comment should indicate it.

string1:    .asciiz "Print this.\n"         #declare a string
```

.space 后的数字n 如申请 n byte 长度的内存.

.ascii结尾无\0，但.asciiz结尾有\0.

	register	memory
执行速度	faster	slower
大小	smaller	bigger
寻址	name+ID	address

```
.data
    str: .ascii "A"
    d1: .byte 10,58,96
    c1: .byte 'B'
    ax: .space 8
```

Address
str: 0x10010000
d1: 0x10010001
c1: 0x10010004
ax: 0x10010005

.data区内的变量都存放于内存中.

内存存放的基本单位是 byte

所有的MIPS指令都必须操作在寄存器上，MIP32中register 大小为32 bit

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

始终为0，不可修改

系统调用返回值

系统调用参数

保存数据

内核

栈、堆

函数调用返回

Load & Store

唯二能访问内存中的数据的指令

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, register $zero always equals 0, and register $at is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Load (to register)

```
lw        register_destination, RAM_source
                          # copy word (4 bytes) at
                          # source_RAM location
                          # to destination register.
                          # load word -> lw

lb        register_destination, RAM_source
                          # copy byte at source RAM
                          # location to low-order byte of
                           # destination register,
                           # and sign -e.g. tend to
                           # higher-order bytes
                           # load byte -> lb

li        register_destination, value
                           #load immediate value into
                           #destination register
                           #load immediate --> li
```

标签

la $s0, str　　load address to register

立即数

li $v0, 4　　load immediate to register

内存

lb $t0, ($a0)　　load byte

1) get value(1) of $a0
2) use the value(1) as address to get memory unit
3) get value(2) in this memory unit
4) load the value(2) to $t0

lh　　load half word (2 byte)

lw　　load word

Store (to memory)

```
sw        register_source, RAM_destination
                          #store word in source register
                          # into RAM destination

sb        register_source, RAM_destination
                          #store byte (low-order) in
                          #source register into RAM
                          #destination
```

sb $t0, 1($a0)

1) get value(1) of $a0
2) s = value(1)+1
3) use s as address to get memory unit
4) get value(2) of $t0
5) store value(2) to the memory unit

label 实际上是内存里的地址.

- Load the address which is labeled by "var1" into the Register "t0".

```
la        $t0, var1
```

- NOTIC : "var1" could be either a label of data or a label of a MIPS instruction.

```
.data
    str:    .asciiz "the answer = "
.text
        li $v0, 4               # system call code for print_str
        la $a0, str             # address of string to print
        syscall                 # print the string
```

Baseline + offset

- Load the word from the memory unit whose **address is in the sum of 4 and the value in register "t0"** to the register "t2".

```
lw        $t2, 4($t0)
```

- Store the word in register "t2" to the memory unit whose **address is the sum of -12 and the value in the register "t0"**.

```
sw        $t2, -12($t0)
```

- **Baselin + offset is applied widely while process on the Array and stack.**

sw　$t0, str($v1)

label　偏移量存在v1中

Arithmetic Instructions

```
  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
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
```

mult结果的高32 bit在hi中，低32 bit在lo中.

div 商在lo中，余数在hi中.
mfhi 将hi中内容到$to
mflo 将lo中内容到$t1

宏：macro_print.asm

```
.macro print_string(%str)
.data
  ...
.text
  ...
.end_macro
```

遇到同名直接替换

使用宏：

```
.include "macro_print.asm"
.data
  ...
.text
main:
  print_string("\n")
```

宏实际上是做了一个字符串替换.

Signed vs. Unsigned

```
.include "macro_print_str.asm"
.data
    tdata: .byte 0x80
.text
main:
    lb $a0,tdata
    li $v0,1
    syscall

    print_string("\n")
    lb $a0,tdata
    li $v0,36
    syscall

    end
```

```
.include "macro_print_str.asm"
.data
    tdata: .byte 0x80
.text
main:
    lbu $a0,tdata
    li $v0,1
    syscall

    print_string("\n")
    lbu $a0,tdata
    li $v0,36
    syscall

    end
```

lb有符号数，以符号位做填充
lbu无符号数，以0做填充

```
.include "macro_print_str.asm"
.data
.text
main:
    print_string("\n -1 less than 1 using slt:")
    li $t0,-1
    li $t1,1
    slt $a0,$t0,$t1
    li $v0,1
    syscall

    print_string("\n -1 less than 1 using sltu:")
    sltu $a0,$t0,$t1
    li $v0,1
    syscall
    end
```

1) slt $t1,$t2,$t3

set less than: if $t2 is less than $t3, then set $t1 to 1 else set $t1 to 0

2) sltu $t1,$t2,$t3

set less than unsigned: if $t2 is less than $t3 using unsigned comparision, then set $t1 to 1 else set $t1 to 0

```
.include "macro_print_str.asm"
.data
    tdata: .word 0x11111111
.text
main:
    lw $t0,tdata
    addu $a0,$t0,$t0
    li $v0,1
    syscall

    print_string("\n")
    add $a0,$t0,$t0
    li $v0,1
    syscall

    end
```

```
.include "macro_print_str.asm"
.data
    tdata: .word 0x71111111
.text
main:
    lw $t0,tdata
    addu $a0,$t0,$t0
    li $v0,1
    syscall

    print_string("\n")
    add $a0,$t0,$t0
    li $v0,1
    syscall

    end
```

→会出报错 arithmetic overflow
两正数相加变成负数

Big-endian vs. Little-endian

The CPU's **byte ordering scheme** (or **endian issues**) affects memory organization and defines the relationship between address and byte position of data in memory.

- a **Big-endian** system means byte 0 is always the most-significant (leftmost) byte.
- a **Little-endian** system means byte 0 is always the least-significant (rightmost) byte.

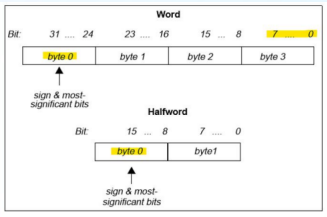


Figure 1-1: Big-endian Byte Ordering

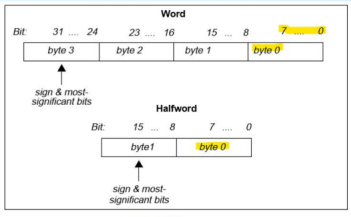


Figure 1-2: Little-endian Byte Ordering

验证大/小端存储

```
.include "macro_print_str.asm"
.data
    tdata0: .byte  0x11,0x22,0x33,0x44
    tdata:  .word  0x44332211
.text
main:
    lb $a0,tdata
    li $v0,34
    syscall

    end
```

```
.include "macro_print_str.asm"
.data
    tdata0: .byte  0x11,0x22,0x33,0x44
    tdata:  .word  0x44332211
.text
main:
    lh $a0,tdata
    li $v0,34
    syscall

    end
```

print integer in hexadecimal	34	$a0 = integer to print	Displayed value is 8 hexadecimal digits, left-padding with zeroes if necessary.

Common Operations

Description	Op-code	Operand
Add with Overflow	add	destination, src1, src2
Add without Overflow	addu	destination, src1, src2
AND	and	destination, src1, immediate
Divide Signed	div	destination/src1, immediate
Divide Unsigned	divu	
Exclusive-OR	xor	
Multiply	mul	
Multiply with Overflow	mulo	
Multiply with Overflow Unsigned	mulou	
NOT OR	nor	
OR	or	
Set Equal	seq	
Set Greater	sgt	
Set Greater/Equal	sge	
Set Greater/Equal Unsigned	sgeu	
Set Greater Unsigned	sgtu	
Set Less	slt	
Set Less/Equal	sle	
Set Less/Equal Unsigned	sleu	
Set Less Unsigned	sltu	
Set Not Equal	sne	
Subtract with Overflow	sub	
Subtract without Overflow	subu	

Description	Op-code	Operand
Rotate Left	rol	
Rotate Right	ror	
Shift Right Arithmetic	sra	
Shift Left Logical	sll	
Shift Right Logical	srl	
Absolute Value	abs	destination,src1
Negate with Overflow	neg	destination/src1
Negate without Overflow	negu	
NOT	not	
Move	move	destination,src1
Multiply	mult	src1,src2
Multiply Unsigned	multu	

move ← add
la ← {lui / ori
not ← nor
ror ←

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.

```
.data
    dvalue1: .byte 27
    dvalue2: .byte 4
.text
    lb $t0,dvalue1
    lb $t1,dvalue2

    div $t0,$t1
    mfhi $a0

    li $v0,1
    syscall

    li $v0,10
    syscall
```

```
.data
    dvalue1: .byte 27
    dvalue2: .byte 4
.text
    lb $t0,dvalue1
    lb $t1,dvalue2

    sub $t1,$t1,1
    and $a0,$t0,$t1

    li $v0,1
    syscall

    li $v0,10
    syscall
```

两者用于求余的结果相同
当dvalue2是2的幂时，左边可被右边所替代.

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.	

CPU如何执行指令

CPU会根据PC寄存器中的地址，从内存中抓取指令，分析并执行.

(32 bit)

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.

```
.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
```

左右两边代码执行结果一样

但左边的效率更高

case1、case2、caseEnd标签是可以不用的

把failLable移到bge后能不写failLable标签.

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
        add $t1,$t1,$t0
        bgt $t7,$t0,calcu

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

        end
```

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
        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 an array  (no more than 10 integer): 1
2
3
4
5
6
7
8
9
0

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

CS202 wangw6@sustech.edu.cn

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?

1

```
#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
```

2

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

3

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

4

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

不能得到右下角的结果，因为内存越界了，将str的字符串弹掉了.

Function

- **jal** function_lable
  - **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

Caller:

```
int x=5;
int y=3;
int z=add(x,y);
X=x+7;
...
```

jal callee → Callee:

```
int add(int a, int b){
    int c = a+b;
    return c;
}
```

jr $ra

jr使用时一定要保证$ra中是有效的地址，jal和jr要成对使用.

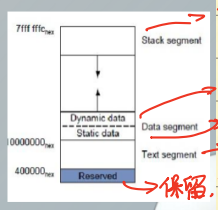
Stack Segment

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**.



栈空间

动态-数据：在运行时确定

静态-数据：在编译后已确定，空间大小不会再变化

→代码位置

→保留，系统指令

$sp是栈指针，一开始在最顶部

```
.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

```
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
```

申请8 byte的栈空间 → 先放到内存保护数据 → 重新放回去 → 回收栈空间.

*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" ?*

```
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
```

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

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?*

*Q2. How does the value of $a0 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
```

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.

Macro

Macros:

A **pattern-matching** and **replacement** facility that provide a simple mechanism to name a frequently used sequence of instructions.

- **Programmer invokes** the macro.
- **Assembler replaces** the macro call with the corresponding sequence of instructions.

Macros vs Subroutines:

- **Same:** permit a programmer to create and name a new abstraction for a common operation.
- **Difference:** Unlike subroutines, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled.

Demo #1

```
.text
print_string:
    addi $sp,$sp,-4
    sw $v0,($sp)

    li $v0,4
    syscall

    lw $v0,($sp)
    addi $sp,$sp,4

    jr $ra
```

Assembler replaces the macro call with the corresponding sequence of instructions.

Q1: What's the difference between macro and procedue?

Q2: While save the macro's defination(on the right hand in this slides) in an asm file, and assamble it,
what's the assembly result?
Is the macro's defination file runable?

Q3: While save the procedure's defination(on the left hand in this slides) in an asm file, and assemble it,
what's the assembly result?
Is this file runable?

```
.macro print_string(%str)
.data
    pstr:  .asciiz  %str
.text
    addi $sp,$sp,-8
    sw $a0,4($sp)
    sw $v0,($sp)

    la $a0,pstr
    li $v0,4
    syscall

    lw $v0,($sp)
    lw $a0,4($sp)
    addi $sp,$sp,8
.end_macro
```

宏只有完成替换后，编译后其内容才能生效，否则全是如空。
宏内部不能有 jal 和 jr，会报错。

Procedure

- In **CALLER**
  - **Before call the callee**
    - **Save caller-saved registers**
      - The called procedure can use these **registers($a0-$a3 and $t0-$t9)** without first saving their value.
      - If the **caller** expects to use one of these registers after a call, it must **save** its value **before the call**.
    - **Pass arguments**
      - By convention, the first four arguments are passed in **registers $a0-$a3**. Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure's stack **frame**.
  - **Execute a jal instruction**
    - **Saves** the return address in **register $ra** and **jumps** to **the callee's first instruction**.
- In **CALLEE(1)**
  - 1. **Allocate memory** for the stack by substracting the frame's size from the stack pointer(**$sp**).
  - 2. **Save callee-saved registers** in the frame.
    - A callee must **save** the values in these registers(**$s0-$s7 and $ra**) before altering them, since the caller expects to find these registers unchanged after the call.
    - **Register $ra ONLY needs to be saved if the callee itself makes a call**. The other callee-saved registers that are used also must be saved.

  In **CALLEE(2)**
  - 3. **before returning to caller**
    - If the callee is a function that **returns value(s)**, place the returned value(s) in register **$v0~$v1**
    - **Restore all callee-saved registers** that were saved upon procedure entry
    - **Pop** the stack frame by adding the frame size to **$sp**
  - 4. **Return** by jumping to the address in register **$ra**

在调用函数前需先传参，在函数内部需保护某些寄存器的内容
嵌套调用函数时一定要保护 $ra 的内容。
保护通过 stack 进行，$sp。
一般推荐 $t0~t9、$a0~a9 在 callee 存，$s0~s7 和 $ra 在 caller 存。

Local label vs. External label

- **Local label**
  - A label referring to an object that **can be used only within the file in which it is defined**.
- **External label**
  - A label referring to an object that **can be referenced from files other than the one in which it is defined**.

局部标签：只能在本文件生效
全局标签：能跨文件。

Find the usage of ".external" and ".globl" on page 10 and 11:
What's the relationship between globl main and the entrance of program?
What will happen if an external data have the same name with a local data?

.globl main 是程序入口。
同名时，局部数据会冲掉 .extern 数据。

Demo #2-1

```
it's in print callee.it's the default_str
it's in print caller.it's the default_str
```

Q1. Is the running result same as the sample snap?
Q2. How many "default_str" are defined in "lab5_print_callee.asm" ?
Q3. While executing the instruction "la $a0,default_str" in these two files, which "default_str" is used?

```
## "lab5_print_caller.asm" ##
.include "lab5_print_callee.asm"
.data
    str_caller:     .asciiz "it's in print caller."
.text
.globl main
main:
        jal print_callee

        addi $v0,$zero,4
        la  $a0,str_caller
        syscall
        la $a0,default_str   ###which one?
        syscall

        li $v0,10
        syscall
```

```
## "lab5_print_callee.asm" ##
.extern default_str 20
.data
        default_str:    .asciiz  "it's the default_str\n"
        str_callee:     .asciiz  "it's in print callee."
.text
print_callee:   addi $sp,$sp,-4
                sw $v0,($sp)

                addi $v0,$zero,4
                la  $a0,str_callee
                syscall
                la $a0,default_str   ###which one?
                syscall

                lw $v0,($sp)
                addi $sp,$sp,4
                jr $ra
```

结果是这个

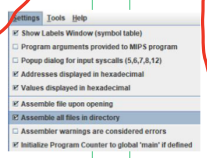
没有.globl main 时,会从头开始执行指令(即.include的内容)
有.globl main 就指定程序入口.

Demo #2-2

In Mars, set "Assemble all files in directory", put the following files in the same directory, then run it and answer the questions on page 10.

```
.data
    str_caller:     .asciiz     "it's in print caller."
.text
.globl  main   → 入口
main:
        jal print_callee

        addi $v0,$0,0x0a636261
        sw $v0,defaulte_str

        addi $v0,$zero,4
        la  $a0,str_caller
        syscall
        la $a0,defaulte_str
        syscall

        li $v0,10
        syscall
```

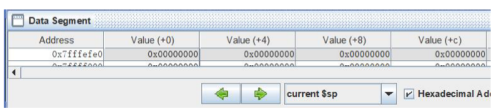
```
.data
        .extern     defaulte_str   20
        str_callee:         .asciiz     "it's in print callee."
        defaulte_str:       .asciiz     "ABC\n"
.text
.globl  print_callee   → 便于在其他文件识别到它
print_callee:   addi $sp,$sp,-4
                sw $v0,($sp)

                addi $v0,$zero,4
                la  $a0,str_callee
                syscall
                la $a0,defaulte_str
                syscall

                lw $v0,($sp)
                addi $sp,$sp,4
                jr $ra
```

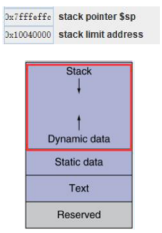


同时编译目录下所有文件,不等于include所有文件.
include后实际上只有一个文件,此时.extern的数据会被同名的局部数据冲掉
而同时编译,是多个文件,局部数据不能被其他文件调用,只有.extern可以.

Stack vs. Heap

- **Stack**: used to store the local variable, usually used in calle.
- **Heap**: The heap is reserved for **sbrk** and **break** system calls, and it not always present.





Demo #3-1 (1)

The following demo(composed of 4 pieces on page 12 and 13) is supposed to get and store the data from input device, get the minimal value among the data, the number of input data is determined by user.

```
.include "macro_print_str.asm"          #piece 1/4
.data
        min_value: .word 0
.text
        print_string("please input the number:")

        li $v0,5            #read an integer
        syscall
        move $s0,$v0    #s0 is the number of integers

        sll $a0,$s0,2       #new a heap with 4*$s0
        li $v0,9
        syscall
        move $s1,$v0    #$s1 is the start of the heap
        move $s2,$v0    #$s2 is the point
```

```
print_string("please input the array\n")     #piece 2/4

add $t0,$0,$0

loop_read:

        li $v0,5        #read the array
        syscall
        sw $v0,($s2)

        addi $s2,$s2,4
        addi $t0,$t0,1
        bne $t0,$s0,loop_read
```

While the 1st input number is 0 or 1,what will happen? why?
Modify this demo to make it better

Demo #3-1 (2)

```
#piece 3/4
        lw $t0,($s1)        #initialize the min_value
        sw $t0,min_value
        li $t0,1
        addi $s2,$s1,4    #$s1 is the start of the heap

loop_find_min:
        lw $a0,min_value
        lw $a1,($s2)
        jal find_min
        sw $v0,min_value
        addi $s2,$s2,4                  #$s2 is the point
        addi $t0,$t0,1
        bne $t0,$s0 loop_find_min  #s0 is the number of integers

        print_string("the min value : ")
        li $v0,1
        lw $a0,min_value
        syscall

        end         #end is defined in the file is macro_print_str.asm
```

```
#piece 4/4
find_min:

        move $v0,$a0
        blt $a0,$a1,not_update
        move $v0,$a1

not_update:

        jr $ra
```

```
please input the number:3
please input the array
-1
0
1
the min value : -1
-- program is finished running --
```

Exception

Exception vs Interruption

- An **exception** is an event that **disrupts the normal flow of the execution of your code.**
  - When an exception occurs, the CPU will figure out what is wrong by checking its status, see if it can be corrected and then continue the execution of the normal code like nothing happened.
  - E.g. Accessing to the 0x0 address in user mode will trigger an exception.
- An **interruption** is an event **caused by a device which is external to the CPU.**
  - E.g. 'syscall' is an interruption.

有些寄存器会记录异常信息.

中断不是异常

syscall中的IO也是一种中断

Common Exception

The following exceptions are the most common in the main processor:

- **Address error** exceptions
  - Which occur when the machine references a data item that is NOT on its proper **memory alignment** or when an **address is invalid** for the executing process.
- **Overflow** exceptions
  - Which occur when **arithmetic operations** compute **signed** values and the destination lacks the precision to store the result.
- **Bus exceptions**
  - Which occur when **an address is invalid** for the executing process.
- **Divide-by-zero** exceptions
  - Which occur when a **divisor is zero.**

The Register in Coprocessor 0

Register name	Register number	Usage
VAddr	8	memory address at which an offending memory reference occurred
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000030
$14 (epc)	14	0x00400010

不能对协处理器直接操作.

$14 (epc): 是触发异常的指令的地址

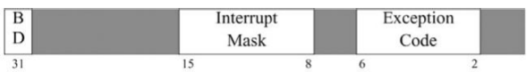
$13 (cause): 是异常的原因 (异常的种类编码)

$12 (status): 程序状态.

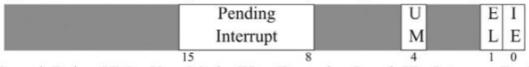
$8 (vaddr): 不总是用到. 显示错误地址

Exception Control Registers

CAUSE:



STATUS:



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

trap自陷：让程序从用户态进入到内核态 (userspace) (kernel space)

eret

Bad Address Exception

```
.text
print_string:
        addi $sp,$sp,-4
        sw $v0,($sp)

        li $v0,4
        syscall

        lw $v0,($sp)
        addi $sp,$sp,4

        jr $ra
```

Registers	Coproc 1	Coproc 0
Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000010
$14 (epc)	14	0x0040000c

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x23bdfffc	addi $29, $29, 0xfffffffc	5: addi $sp,$sp,-4
	0x00400004	0xafa20000	sw $2, 0x00000000($29)	6: sw $v0,($sp)
	0x00400008	0x24020004	addiu $2, $0, 0x00000004	9: li $v0,4
	0x0040000c	0x0000000c	syscall	10: syscall
	0x00400010	0x8fa20000	lw $2, 0x00000000($29)	12: lw $v0,($sp)
	0x00400014	0x23bd0004	addi $29, $29, 0x00000004	13: addi $sp,$sp,4
	0x00400018	0x201fffff	addi $31, $0, 0xffffffff	15: addi $ra,$zero,0xffffffff
	0x0040001c	0x03e00008	jr $31	16: jr $ra

```
Runtime exception at 0x0040000c: address out of range 0x00000000
```

能用的地址从 0x00400000 开始.
以下是保留地址，不能访问.

$a0's default value is 0x00000000, which is not allowed to access in user mode

```
.data
        str: .asciiz "hello"
.text
print_string:
        addi $sp,$sp,-4
        sw $v0,($sp)

        la $a0,str
        li $v0,4
        syscall

        lw $v0,($sp)
        addi $sp,$sp,4

        addi $ra,$zero,0xffffffff
        jr $ra
```

$ra	31	0xffffffff
pc		0xffffffff

Registers	Coproc 1	Coproc 0
Name	Number	Value
$8 (vaddr)	8	0xffffffff
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000010
$14 (epc)	14	0xffffffff

```
Error in : invalid program counter value: 0xffffffff
```

bs: .byte 1:10 申请3.10个连续的byte空间，每个初始化为1

ws: .word 2:10 申请3.10个连续的word空间，每个初始化为2.

half-word、word、double-word 都要求对齐

Which one will trigger the exception ?

Registers	Coproc 1	Coproc 0
Name	Number	Value
$8 (vaddr)	8	0x10010009
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000010
$14 (epc)	14	0x0040001c

没有以4的倍数为边界对齐

```
.include "../macro_print_str.asm"
.data
        str:    .asciiz "data is:"
        bs:     .byte 1:10
        ws:     .word 2:10
.text
        print_string("data is:")
        add $t0,$zero,$zero
        addi $t1,$zero,10
loop_out:
        lw $a0,bs
        li $v0,1
        syscall
        add $t0,$t0,1
        bne $t0,$t1,loop_out

        end
```

```
.include "../macro_print_str.asm"
.data
        str:    .asciiz "data is:"
        bs:     .byte 1:10
        ws:     .word 2:10
.text
        print_string("data is:")
        add $t0,$zero,$zero
        addi $t1,$zero,10
loop_out:
        lw $a0,ws
        li $v0,1
        syscall
        add $t0,$t0,1
        bne $t0,$t1,loop_out

        end
```

```
Runtime exception at 0x0040001c: fetch address not aligned on word boundary 0x10010009
```

→要求以1 word (即4 byte)对齐.

str: 0x10010000
bs: 0x10010009
ws: 0x10010014 → 不在0x10010013是因为内存对齐.

Arithmetic Exception

Will the 'addu' trigger an exception? How about 'sub', 'div'? How about 'addiu $a0, $t0, -1'?

```
.data
        addend1: .word 0x7fffffff
        addend2: .word 0x7fffffff
.text
print_string:
        lw $t0,addend1
        lw $t1,addend2
        add $a0,$t0,$t1

        li $v0,1
        syscall

        li $v0,10
        syscall
```

Registers	Coproc 1	Coproc 0
Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000030
$14 (epc)	14	0x00400010

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1, 0x00001001	6: lw $t0,addend1
	0x00400004	0x8c280000	lw $8, 0x00000000($1)	
	0x00400008	0x3c011001	lui $1, 0x00001001	7: lw $t1,addend2
	0x0040000c	0x8c290004	lw $9, 0x00000004($1)	
	0x00400010	0x01092020	add $4, $8, $9	8: add $a0,$t0,$t1
	0x00400014	0x24020001	addiu $2, $0, 0x00000001	10: li $v0,1
	0x00400018	0x0000000c	syscall	11: syscall
	0x0040001c	0x2402000a	addiu $2, $0, 0x0000000a	13: li $v0,10
	0x00400020	0x0000000c	syscall	14: syscall

→这些不能直接修改.

```
Runtime exception at 0x00400010: arithmetic overflow
```

div $t1,$t2,$t3 做了一个检查，看是否除0.（里面会先检查除数是否为零，若是则break）

div $t1,$t2 不会做检查

How MIPS Acts When Taking An Exception?

- **Step1.** It sets up the **EPC** to point to the restart location
- **Step2.** CPU changes into **kernel mode** and **disables the interrupts** (MIPS does this by setting EXL bit of **Status register**)
- **Step3.** Set up **the Cause register** to indicate which is wrong so that software can tell the reason for the exception. If it is for **address exception**, for example, TLB miss and so on, the **BadVaddr register** is set.
- **Step4.** CPU starts fetching instructions from the exception entry point and then goes to the **exception handler.**

Up to MIPS III, **eret** instruciton is used to return to the original location before falling into the exception. Note that **eret** behavior is: **clear the SR[EXL] bit and return control to the adress stored in EPC.**

① EPC会存储发生问题的地址
② CPU会进入内核模式，使中断无效
③ cause 会存异常发生的原因
④ CPU进入异常句柄.

内核态权限很大.

.ktext 0x80000000 .kdata 0x90000000
exception handler 0x80000180

若满足条件，就中断程序，执行ktext中的内容.

Exception Related Instructions

- **Conditional trap**
  - teq $s0, $s1 ##trap(jump to the ktext), if s0==s1
  - tne $s0, $s1 ##trap(jump to the ktext), if s0!=s1
  - teqi $s0, 1 ##trap(jump to the ktext), if s0==1
- **mfc0,mtc0**
  - mfc0 $k0,$14 ##Move from coproc0 reg#14(epc) to $k0
  - mtc0 $k0,$14 ##Move from $k0 to coproc0 reg#14(epc)
- **eret**
  - Returns from an interrupt, exception or error trap.
  - Similar to a branch or jump instruction, eret executes the next instruction before taking effect. Use this on R4000 processor machines in place of rfe.

exception return
在.ktext中将EPC的下一行更新到PC中.

Demo

```
.data
        dmsg:   .asciiz "\ndata over"
.text
main:   li $v0,5
        syscall
        teqi $v0,0
        la $a0,dmsg
        li $v0,4
        syscall
        li $v0,10
        syscall
```

```
.ktext  0x80000180
        move $k0,$v0
        move $k1,$a0
        la $a0,msg
        li $v0,4
        syscall
        move $v0,$k0
        move $a0,$k1
        mfc0 $k0,$14
        addi $k0,$k0,4
        mtc0 $k0,$14
        eret
.kdata
msg: .asciiz "\nTrap generated"
```

$k0未加4

这个地址只能是exception handler

若是其他地址，则不会执行.ktext的指令.

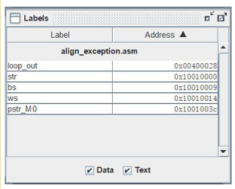
Q1. How to tigger the trap?

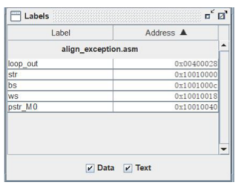
Q2. When will the string "\ndata over" be printed out?

Q3. Use "break" in text segment and ktext segment seperately, what happens?

(假如$k0不加4，eret后会一直重复.ktext.
此时可等待外设修改$v0后更新，循环teqi $v0,0的判断)

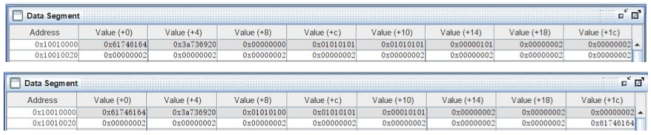
$v0为0时触发.
$v0不为0或执行完.ktext后eret返回后执行.
break不论在.text还是在.ktext，都会在执行后更新cause的内容(0x00000024)，都会跳到0x80000180处(exception handler)
但若break在.ktext的eret后，不会被执行.






```
.data
        str:   .asciiz "data is:"
        bs:    .byte 1:10
        ws:    .word 2:10
```

```
.data
        str:   .asciiz "data is:"
        .align 2
        bs:    .byte 1:10
        ws:    .word 2:10
```



.align Align the next datum on a 2 ^{n} byte boundary.

For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.kdata subsequent items are stored in the kernel data segment, If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.

.ktext subsequent items are stored in the kernel text segment, In SPIM, these items may only be instructions or words . If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.

.align 能够按内存对齐

IEEE 745 on Floating-point Number

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

single: 8 bits　　single: 23 bits
double: 11 bits　　double: 52 bits

S	Exponent (yyyy+Bias)	Fraction (xxxx)

$x = (-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$

For single-precision float data:
Exponents(8bit): **0000_0000** and **1111_1111** reserved
Bias in Exponent: **0111_1111**

For double-precision float data:
Exponents(11bit): **000_0000_0000** and **111_1111_1111** reserved
Bias in Exponent : **011_1111_1111**

```
.data
fneg1:    .float   -1
wneg1:    .word    -1
fpos1:    .float    1
wpos1:    .word     1
```

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$
single: 8 bits　double: 11 bits　single: 23 bits　double: 52 bits
S | Exponent (yyyy+Bias) | Fraction (xxxx)
$x = (-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$

Label	Address ▲
float_rw.asm	
fneg1	0x10010000
wneg1	0x10010004
fpos1	0x10010008
wpos1	0x1001000c

```
➢ -1 = (-1)¹ x (1+0) x 2⁰
    s: 1;  exponent: 0 + 0111_1111;      fraction: 0
➢ 1  = (-1)⁰ x (1+0) x 2⁰
    s: 0;  exponent: 0 + 0111_1111;      fraction: 0
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0xbf800000	0xffffffff	0x3f800000	0x00000001

float和word位宽一样，但word会以原数的补码存，float会以IEEE745标准存.

Infinite vs NaN

Which one will get an infinite value?

Which one will get the NaN?

	31 Sign	30–23 Exponent	22–0 Mantissa
93000000	0	0001 1010	101 1000 1011 0001 0001
0	0	0000 0000	000 0000 0000 0000 0000
+Infinity	0	1111 1111	000 0000 0000 0000 0000
−Infinity	1	1111 1111	000 0000 0000 0000 0000
Quiet NaN	x	1111 1111	0xx xxxx xxxx xxxx xxxx
Signaling NaN	x	1111 1111	1xx xxxx xxxx xxxx xxxx

```
.data
    sdata: .word 0xFF7F7FFF
    fneg1: .float -1
.text
    lw $t0,sdata
    mtc1 $t0,$f1
    mul.s $f12,$f1,$f1

    li $v0,2
    syscall

    lwc1 $f2,fneg1
    mul.s $f12,$f12,$f2

    li $v0,2
    syscall

    li $v0,10
    syscall
```

lwc1

```
.data
    sdata: .word 0xffff7fff
    fneg1: .float -1
.text
    lw $t0,sdata
    mtc1 $t0,$f1
    mul.s $f12,$f1,$f1

    li $v0,2
    syscall

    lwc1 $f2,fneg1
    div.s $f12,$f12,$f2

    li $v0,2
    syscall

    li $v0,10
    syscall
```

Coprocessor 1 in MIPS

Q1. What's the difference between 'lwc1' and 'ldc1'?
Q2. Which demo would trigger the exception?
Q3. Which demo would get the right answer?

Registers	Coproc 1	Coproc
Name	Float	
$f0	0x00000000	
$f1	0xbf800000	
$f2	0x00000000	
$f3	0x3f800000	

Runtime exception at 0x00400004: first register must be even-numbered

Runtime exception at 0x00400010: all registers must be even-numbered

```
.data       #demo1
    fneg1:   .float   -1
    fpos1:   .float    1
.text
    lwc1 $f1,fneg1
    lwc1 $f3,fpos1
    add.s $f12,$f1,$f3

    li $v0,2
    syscall
    li $v0,10
    syscall
```

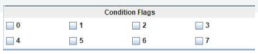
```
.data       #demo2
    fneg1:   .double  -1
    fpos1:   .double   1
.text
    ldc1 $f1,fneg1      ↙不能奇数
    ldc1 $f3,fpos1
    add.d $f12,$f1,$f3

    li $v0,3
    syscall
    li $v0,10
    syscall
```

```
.data       #demo3
    fneg1:   .double  -1
    fpos1:   .double   1
.text
    ldc1 $f0,fneg1
    ldc1 $f2,fpos1
    add.d $f11,$f0,$f2   ↑不能奇数

    li $v0,3
    syscall
    li $v0,10
    syscall
```

double会连续使用两个寄存器，且只能在偶数如$f0，则会用$f0、$f1，且前一个会存低位后一个存高位.

Floating-point Instructions

Type	Description	Instructions
Load and Store	**Load** values and move data **between memory and coprocessor registers**	**lwc1**,**ldc1**; **swc1**,**sdc1**; ...
Move	**Move** data between registers	**mtcl**, **mfc1**; **mov.s**,**mov.d**;
Computational	Do **arithmetic** operations on values in coprocessor 1 registers	add.**s**, add.**d**; sub.s, sub.d; mul.s, mul.d; div.s,div.d; ...
Relational	**Compare** two floating-point values and **set conditional flag**	c.eq.**s**, c.eq.**d**; c.le.s,c.le.d; c.lt.s,c.lt.d; ...
Conditional jumping	**Conditional jump** while **conditional flag** is 0(**f**alse)/1(**t**rue)	**bc1f**, **bc1t**
Convert	**Convert** the **data type**	**floor**.w.d,floor.w.s; **ceil**.w.d, ceil.w.s; **cvt**.d.s




```
.include "macro_print_str.asm"
.data
    f1: .float 12.625
.text
    lwc1 $f0,f1
    floor.w.s $f1,$f0
    ceil.w.s $f2,$f0
    round.w.s $f3,$f0

    print_string("orignal float: ")
    print_float($f0)

    print_string("\nafter floor:")
    print_float($f1)

    print_string("\nafter ceil:")
    print_float($f2)

    print_string("\nafter round:")
    print_float($f3)

    end
```

Q1. What's the output of current demo after running? Why?
Q2. How to change the code to get correct output?

```
.macro print_float(%fr)
    addi $sp,$sp,-8
    swc1 $f12,4($sp)
    sw $v0,0($sp)

    mov.s $f12,%fr
    li $v0,2
    syscall

    lw $v0,0($sp)
    lwc1 $f12,4($sp)
    addi $sp,$sp,8
.end_macro
```

```
orignal float: 12.625
after floor:1.7E-44
after ceil:1.8E-44
after round:1.8E-44
-- program is finished running --
```

```
orignal float: 12.625
after floor:12
after ceil:13
after round:13
-- program is finished running --
```

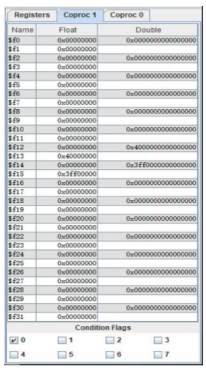
11

Tips:

Single	31	30 23	22 0
	Sign	Exponent	Mantissa
93000000	0	0001 1010	101 1000 1011 0001 0001
0	0	0000 0000	000 0000 0000 0000 0000
+Infinity	0	1111 1111	000 0000 0000 0000 0000
−Infinity	1	1111 1111	000 0000 0000 0000 0000
Quiet NaN	x	1111 1111	0xx xxxx xxxx xxxx xxxx
Signaling NaN	x	1111 1111	1xx xxxx xxxx xxxx xxxx

	High-order word																Low-order word															
Double	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Sign	Exponent											Mantissa																			
93000000	0	000	0001	1010	1011	0001	0110	0010	0010	1000	0000	....																				
0	0	000	0000	0000	0000	0000	0000	0000	0000	0000	0000	....																				
+Infinity	0	111	1111	1111	0000	0000	0000	0000	0000	0000	0000	....																				
−Infinity	1	111	1111	1111	0000	0000	0000	0000	0000	0000	0000	....																				
Quiet NaN	x	111	1111	1111	0xxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	....																				
Signaling NaN	x	111	1111	1111	1xxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	....																				

reference from "see in MIPS"



registers and flags in coprocessor 1