

組別： _ _ _ _ _ 簽名： _ _ _ _ _

[\[group 1\]](#)

1. Qustion:

Suppose the memory byte at the address stored in \$s0 contains the value 0xF7. Execute the following two MIPS instructions:

`lb $t0, 0($s0)``lbu $t1, 0($s0)`

Give the 32-bit hexadecimal values stored in \$t0 and \$t1 after execution, and briefly explain why the two results are different.

Ans:

`0xF7 = 1111 0111` # 8-bit binary representation`lb $t0, 0($s0)` # Load byte with sign extension`$t0 = 11111111 11111111 11111111 11110111``$t0 = 0xFFFFFFFF` # MSB = 1, so upper bits are filled with 1s`lbu $t1, 0($s0)` # Load byte unsigned with zero extension`$t1 = 00000000 00000000 00000000 11110111``$t1 = 0x000000F7` # Upper bits are filled with 0s

Therefore:

`$t0 = 0xFFFFFFFF` # lb → sign extension`$t1 = 0x000000F7` # lbu → zero extension

[group 1]

2. Question:

In MIPS register conventions, what is the key difference between temporary registers (\$t0–\$t9) and saved registers (\$s0–\$s7) during a procedure call?

Ans:

\$t0–\$t9 are caller-saved (a called function can freely overwrite them without saving them).

\$s0–\$s7 are callee-saved (if a function uses them, it must save their original values on the stack and restore them before returning).

[group 5]

3. Question:

Given:

a -> \$s0, b -> \$s1

Write MIPS code to call a function sum(a, b) and return a + b.

Use:

\$a0, \$a1 for arguments

\$v0 for the return value

jal to call the function

jr \$ra to return

Ans:

add \$a0, \$s0, \$zero

add \$a1, \$s1, \$zero

jal sum

add \$s2, \$v0, \$zero

j Exit

sum:

add \$v0, \$a0, \$a1

jr \$ra

Exit:

[group 6]

4. Question:

請問 jr、jal、j 的用法跟意義有甚麼不同？

Ans:

j label(jump)會無條件的跳去該 label 繼續執行，執行後會直接把程式指向 Counter(PC)該標籤的位置，不會紀錄是從哪裡跳來的。常用於寫迴圈、if-else 內部的跳躍。

jr \$ra(jump register)跳到 ra 儲存的記憶體地址，常跟 \$ra 一起使用，當用 jal 呼叫函式時，系統會把「下一行指令的地址」記在 \$ra 裡；當函式跑完後，執行 jr \$ra 就可以跳回原本呼叫它的地方。常用於函式 (Function) 的結尾，相當於 C 語言的 return;。

jal label(jump and link)會跳到該 label 繼續執行，並在跳過去之前會記錄下一行程式的地址到 \$ra 中，這樣和 jr \$ra 搭配使用時可以回到原來程式呼叫的地方並繼續執行下去

[group 13]

5. Question:

Translate the following high-level C code into MIPS assembly language, adhering strictly to standard MIPS calling conventions.

C code:

```
int square(int n) {  
    return n * n;  
}
```

```
int sum_of_squares(int a, int b) {  
    int x = square(a);  
    int y = square(b);  
    return x + y;  
}
```

Requirements & Constraints

1. Follow standard MIPS register usage rules:
 - o Arguments: \$a0, \$a1
 - o Return values: \$v0
 - o Saved registers: \$s0 – \$s7 (callee-saved)
 - o Temporary registers: \$t0 – \$t9 (caller-saved)
 - o Return address: \$ra
2. Stack management in sum_of_squares:
 - o Save \$ra before calling square.
 - o Save any \$s registers used across sub-procedure calls, or use stack slots directly.
 - o Clean up the stack frame before returning.

Ans:

square:

```
mul    $v0, $a0, $a0
```

```
jr     $ra
```

sum_of_squares:

```
addi $sp, $sp, -12
```

```
sw    ra, 8(sp)
```

```
sw    s0, 4(sp)
```

```
sw    s1, 0(sp)
```

```
move $s0, $a1
```

```
jal   square
```

```
move $s1, $v0
```

```
move $a0, $s0
```

```
jal   square
```

```
add   $v0, $s1, $v0    # $v0 = x + y
```

```
lw    s1, 0(sp)
```

```
lw    s0, 4(sp)
```

```
lw    ra, 8(sp)
```

```
addi $sp, $sp, 12
```

```
jr     $ra
```

[group 7]

6. Question:

Consider the following MIPS recursive procedure that computes the factorial of an integer:

fact:

```
addi    $sp, $sp, -8
```

```

sw $ra, 4($sp)

sw $a0, 0($sp)

slti $t0, $a0, 1

beq $t0, $zero, L1

addi    $v0, $zero, 1

addi    $sp, $sp, 8

jr $ra

L1:

addi    $a0, $a0, -1

jal fact

lw $a0, 0($sp)

lw $ra, 4($sp)

addi    $sp, $sp, 8

mul     $v0, $a0, $v0

jr $ra

```

Suppose the procedure is called with $\$a0 = 3$ and the initial value of $\$sp$ is 1000. Assume that the stack grows toward lower memory addresses and that one word occupies 4 bytes.

Answer the following questions:

1. What is the value of $\$sp$ immediately after control returns from `fact(0)` to `fact(1)`?
2. At the moment described in Question 1, what value is stored at memory address 984?
3. What is the final value stored in $\$v0$ after the initial call to `fact` finishes?

Ans:

Each invocation of `fact` begins by allocating 8 bytes on the stack:

```
addi    $sp, $sp, -8
```

so the stack pointer decreases by 8 bytes for every recursive call.

Starting with $\$sp = 1000$, the recursive calls give:

fact(3): $\$sp = 992$

fact(2): $\$sp = 984$

fact(1): $\$sp = 976$

fact(0): $\$sp = 968$

Question 1:

For fact(0), the base case is reached. Before returning, it executes:

```
addi $sp, $sp, 8
```

Therefore: $968 + 8 = 976$

So immediately after fact(0) returns to fact(1): $\$sp = 976$

Question 2:

At this moment, $\$sp = 976$.

Earlier, when fact(2) was entered, $\$sp = 984$ and $\$a0 = 2$.

Therefore, `sw $a0, 0($sp)` stored the value 2 at address 984.

Question 3:

The recursive return values are:

fact(0) = 1

fact(1) = $1 \times 1 = 1$

fact(2) = $2 \times 1 = 2$

fact(3) = $3 \times 2 = 6$

Thus, the final value stored in $\$v0$ is 6.

[group 9]

7. Question:

Whats the difference between a leaf procedure and a non leaf procedure, give each a short example

Ans:

Leaf Procedure: Does not call any other procedures.

Example: `int add(int a, int b) { return a + b; }`

Non-Leaf Procedure: Calls other procedures.

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

[group 4]

8. Question:

Assume a MIPS beq instruction is at address 0x00013890:

- 1.Can the branch target be at address 0x0001B89B? Please explain
- 2.Can the branch target be at address 0x00033894? Please explain
- 3.What is the maximum possible branch target address?

Ans: (1) No (2) No (3) 0x00033890

1. $0x0001B89B = 0b11011100010011011$

The final two digits aren't 00, so it isn't a valid target address

2. $0x00033894 = 211092$, $0x00013890 = 80016$

$(211092 - (80016 + 4)) / 4 = 32768 > 2^{15} - 1 = 32767$

Not a valid target address

3. $0x00013890 = 80016$, $(80016 + 4) + 32767 * 4 = 211088$

$211088 = 0x00033890$

[group 8]

9. Question:

Assume the following MIPS code starts at address 80000:

```
Loop:  add  $t0, $t0, $t1
        beq  $t0, $zero, End
        addi $s0, $s0, 1
        j    Loop
```

End:

Given: \$t0 = register 8, \$zero = register 0, beq opcode = 4.

Find the decimal I-format fields for beq \$t0, \$zero, End
in the form: opcode | rs | rt | immediate

Ans:

Instruction addresses:

80000 add

80004 beq

80008 addi

80012 j

80016 End

$PC + 4 = 80008$

Target = 80016

$\text{immediate} = (80016 - 80008) / 4 = 2$

4 | 8 | 0 | 2

[group 10]

10. Question:

Let the value of the program counter (i.e., PC) be 0xCF01FC00. What is the target address of the instruction, j 0x20?

Ans: 0xC0000080

0xC0000080

PC+4 [31-28] 26 bits address (0x20) 2' b00

```
1100 000000000000000000000000100000 00
```

PC +4= 0xCF01FC00

For a MIPS jump instruction,

Target Address = { PC[31:28], target[25:0], 2'b00 }

$$\text{PC}+4[31:28] = 0\text{x}\text{C} = 1100_2$$

j 0x20:

target = 0x20

Since the target field represents a word address,

$$0x20 \ll 2 = 0x80$$

Therefore,

Target Address = 0xC0000000 | 0x00000080 = 0xC0000080

[group 11]

11. Question:

What is the jump range of the MIPS j instruction? How many bits are used to specify the target address, and how is the target address decoded? Why can't it jump to any arbitrary 32-bit address?

Ans:

The MIPS j instruction is a J-type instruction with a total length of 32 bits. It contains a 6-bit opcode and a 26-bit target address field.

The 32-bit target address is constructed as follows:

Target Address = $\{(PC+4), [31:28], \text{instr}[25:0]00\}$

The 26-bit target field is shifted left by 2 bits, and the upper 4 bits are taken from PC + 4. The lowest 2 bits are set to 00 because MIPS instructions are word-aligned.

Since $226 \times 4 = 228$ bytes, the j instruction can jump within the same 256 MiB address region as PC + 4.

It cannot jump to an arbitrary 32-bit address because the instruction does not encode all 32 address bits. The upper 4 bits are inherited from PC + 4, and the lower 2 bits are fixed at 00. To jump to an address outside the current region, a register-based jump such as jr can be used with the full target address stored in a register.