Computer Architecture

Fall, 2025
Week 3
2025.09.15

組別：__________ 簽名：____

[group 1]

1. Qustion:

The MIPS architecture has three instruction formats: R-type, I-type, and J-type.

(a) For the instruction below, determine which format applies and explain why.

add $t0, $s0, $s1 Ans: R-Type, with rs/rt/rd

(b) Using the format you identified in part (a), convert the instruction into 32-bit binary machine code and its hexadecimal representation. Show all steps clearly.

Ans: opcode, rs, rt, rd, shamt, func
bits 6 5 5 5 5 6

Ans:

(a) This is an arithmetic operation with three registers and no immediate constant, no memory address, no jump label. Therefore, it uses the R-type format.

R-type format fields: **[opcode(6)] [rs(5)] [rt(5)] [rd(5)] [shamt(5)] [funct(6)].**

Note:

- R-type (Register format): used for arithmetic/logic operations involving three registers (add, sub, and, slt). Fields:
  [opcode(6)] [rs(5)] [rt(5)] [rd(5)] [shamt(5)] [funct(6)]

- I-type (Immediate format): used for instructions with constants, offsets, or branches (addi, lw, sw, beq). Fields:
  [opcode(6)] [rs(5)] [rt(5)] [immediate(16)]

- J-type (Jump format): used for jump instructions with a target address (j, jal). Fields:
  [opcode(6)] [address(26)]

(b) Step 1. Recall the Instruction Format

MIPS **R-type format** is 32 bits:

opcode (6) rs (5) rt (5) rd (5) shamt (5) funct (6)

- opcode → tells the instruction type (R-type = 000000)
- rs → first source register
- rt → second source register
- rd → destination register
- shamt → shift amount (used only by shift instructions, otherwise all 00000)
- funct → specific operation (example: add = 100000, sub = 100010)

[group 3]

2. Question:

Convert the following two machine codes into 32-bit binary and decode them. Explain what they do.

(1) 0x02324020

(2) 0x00105100

Ans: (1) ✓ 0000 00 | 10 001 | 1 0010 | 0100 0 | 000 00 | 10 0000

0 17 18 8 0 32

add $s1 $s2 $t0

(2) ✓ 0000 00 | 00 000 | 1 0000 | 0101 0 | 001 00 | 00 0000

0 0 16 12 4 0

$zero $s0 $t2

sll $t2, $s0, 4

(Logical shift left by 4bits)

Ans:

(1) ✓

Binary: 000000 10001 10010 01000 00000 100000

Fields: op=000000, rs=10001($s1), rt=10010($s2), rd=01000($t0), shamt=00000, funct=100000(0x20)

Instruction: add $t0, $s1, $s2

Function: $t0 = $s1 + $s2

(2) ✓

Binary: 000000 00000 10000 01010 00100 000000

Fields: op=000000, rs=00000($zero), rt=10000($s0), rd=01010($t2), shamt=00100(4), funct=000000(0x00)

Instruction: sll $t2, $s0, 4

Function: Logical shift left by 4 bits ➞ $t2 = $s0 << 4 (multiplying by 2^4)

[group 4]

3. Question:

Encode the following instruction into its I-format binary and hexadecimal:

Ans: · I-Type opcode rs rt imm
6 5 5 16 bits

sw $t3, 40($s4)
|| ||
reg11 reg20

Ans:

X 1010|11 01|011 1|0100|0000|0000|0010|1000
A D 7 4 0 0 2 8

opcode	rs	rt	immediate
43	20	11	40

This ver. $t3 is "rs", $s4 is "rt"

But WRONG!!!!!

☆ $t3 = rt
$s4 = rs

Opcode for sw = 43 (**101011**)
Rs for $s4 = 20 (**10100**)
Rt for $t3 = 11 (**01011**)
Immediate = 40 (**0000 0000 0010 1000**)

Binary format:

101011 10100 01011 0000000000101000

Convert to hex:

1010	1110	1000	1011	0000	0000	0010	1000

A	E	8	B	0	0	2	8

Hex format:

0xAE8B0028

[group 7]

4. Question:

Why addi $t0, $t1, 1000000 will get error? And why this error exist, explain the behind priciple.

Ans: I-Type 6 5 5 16 (bits)

⇒ 16 bits signed number range -32678 ~ 32677

Ans:

I format Instruction can only have 2^16 immediate, it will out of the length. This follows the "Make the Common Case Fast" and "Simplicity favors regularity" principle. The 16-bit I-type field is sufficient for frequent operations (arithmetic with small constants, local branches). And the total instruction length is 32bits, 10 bits for rs rt, and 6 bits for operand. So we only have 16 bits for immediate.

[group 8]

5. Question:

Translate the following high-level C code into MIPS assembly:

for (i = 0; i < 5; i++) sum = sum + i;

Assume i is stored in $t0, and sum = 0 is initially stored in $t1.

```
Ans:
  addi $t0, $zero, 0  # i
  addi $t2, $zero, 5
Loop:
  beq $t0, $t2, End
  add $t1, $t1, $t0 #sum+i
  addi $t0, $t0, 1
  j Loop
End:
```

Ans:

```
    addi $t0, $zero, 0      # i = 0
    addi $t2, $zero, 5      # constant 5


Loop:
    beq  $t0, $t2, End      # if i == 5, exit
    add  $t1, $t1, $t0      # sum = sum + i
    addi $t0, $t0, 1        # i++
    j    Loop

End:
```

[group 9]

6. Question:

The following MIPS loop counts from 10 down to 1. It uses basic instructions.

```
    addi $t0, $zero, 10      # $t0 = 10 (counter)
loop:
    addi $t0, $t0, -1        # $t0 = $t0 - 1
    bne   $t0, $zero, loop # if $t0 != 0, goto loop
done:
```

The bne instruction's format is: opcode (6 bits) | rs (5 bits) | rt (5 bits) | immediate (16 bits)

Why did the designers of MIPS choose to make the immediate field only 16 bits instead of a larger size, like 20 bits? Explain the trade-off they had to consider.

✓ Ans:

An I-type instruction must fit into 32 bits. A larger immediate field (e.g., 20 bits) would require smaller rs/rt fields, reducing the number of addressable registers from 32 to 16. Designers prioritized access to all registers over a larger branch range, as most branches are to nearby targets.

[group 10]

7. Question:

Assume register $S2 contains the base address of a memory location. The byte stored in that memory address is 1010 1101. Find what value is loaded into $to if we use: 1. lb $to, 0($S2) 2. lbu $to, 0($S2)

Ans : ✓① lb → 0x FF FF FF AD = -83

signed extend

✓ Ans:

1. lb(signed):1010 1101 =173 (unsigned decimal) signed value = 173-256=-83

32 bit result:0xFFFFFFAD = -83

✓② lbu → 0x AD = 173

zero-extend

2. Zero-extend 10101101 -> 0xAD

32 bit result:0x000000AD = 173

[group 11]

8. Question:

1. Given register information: Assign $s0 value 0, $s2 to base address of array A, and $s1 value = 100 in decimal. Then translate the following MIPS instructions into C code:

```
add $t0, $zero, $s0

Loop:

sll $t1, $t0, 2

add $t1, $s2, $t1

sw $t0, 0($t1)

addi $t0, $t0, 1

slt $t2, $t0, $s1

bne $t2, $zero, Loop

Exit:
```

$t0 = 0   # i

$s0 = 0

$s2 = &A

$s1 = 100

loop →

$t1 = $t0 << 2

$t1 = $s2 + $t1

A[$t0] = $t0

$t0 = $t0 + 1   # i++

✓ for (int i = 0; i < 100; i++)

A[i] = i;

✓

Ans:

```
for(i = 0; i<100; i++){
        A[i] = i;
}
```

[group 12]

9. Question:

Please indicate the true or false of the following statements and explain why.

(a) The reason why we use shifting to multiply is that it's cheaper and faster.

(b) In C, we have " a* =32 " which would compile to " sll $s0,$s0,32 " in MIPS.

(c) Use beq and bne instead of blt, bge, etc is an example of good design compromise.

(d) The maximum and minimum immediate that can be implemented in I-Format are ±2^16.

(e) In MIPS, R-format instructions always use the funct field to determine the exact operation.

✓

Ans: (a) true

(b) false, 32 → 5 ⇒ $2^5 = 32$

(c) true

(d) false, $-2^{15} \sim 2^{15}-1$

(e) true

Ans:

(a) T

(b) F, it should be "sll $s0,$s0,5"

(c) T

(d) F, −2^15 ~ + 2^15 − 1

(e) F (T), R-format instructions use both the opcode (always 0 for R-type) and the funct field to determine the exact operation, so it is not determined by the funct field alone. Suggest T

[group 13]

10. Question:

The MIPS instruction set includes several shift instructions. They include logical-shift-left, logical-shift-right, and arithmetic-shift-right. Why doesn't MIPS offer an "arithmetic-shift-left" code?

Ans:

Left shifting is equivalent to multiplication by 2 for both positive and negative numbers in two's complement representation. Whether logical or arithmetic, the operation is identical: shift all bits left and fill the rightmost position with zero.

For positive numbers:

- 21 (0001 0101) → 42 (0010 1010) = 21 × 2

For negative numbers:

- -17 (1110 1111) → -34 (1101 1110) = -17 × 2

In both cases, left shifting by one position doubles the value regardless of sign, making LSL and ASL functionally identical operations.

[group 14]

11. Question:

Many programs have more variables than computers have registers. What does the compiler do to conquer this problem?

Ans:

Registers spilling:

The compiler tries to keep the most frequently used variables in registers and places the rest in memory, using load and store to move variables between registers and memory.