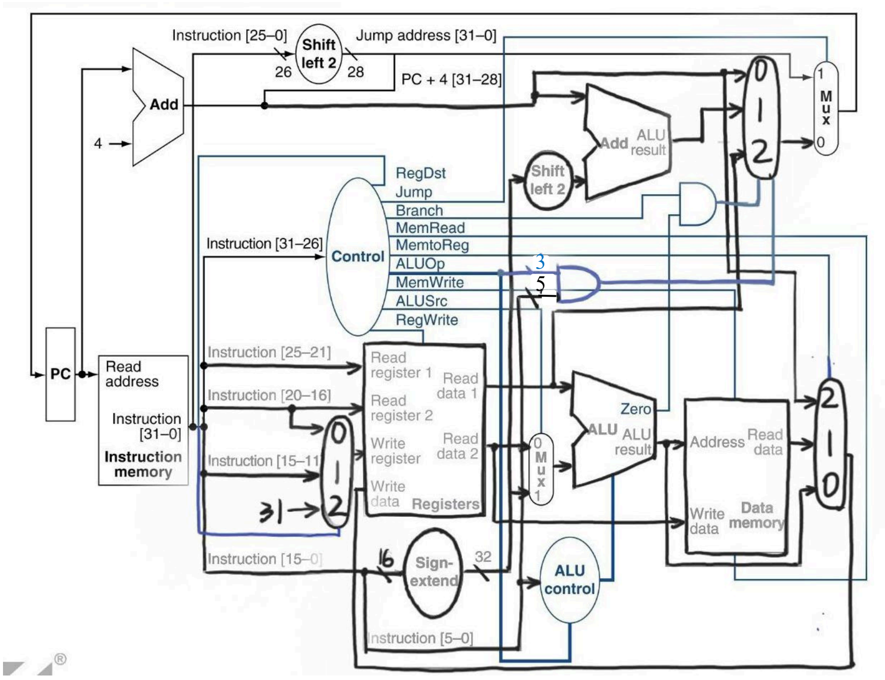
# Computer Organization Lab3

## Name:

## ID:

### <u>Architecture diagrams</u>:

(modified from Ch4 ppt)




### <u>Hardware module analysis</u>:

---------------------- same as Lab2 ----------------------

- Adder

  Composed of 32 full-adders, with "cin" of the first bit be 0.
  full-adder:

| Input | | | Output | |
|---|---|---|---|---|
| a | b | cin | result | cout |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

➔ result = a ^ b ^ cin
cout = (a & b) | (a & cin) | (b & cin)

## ➢ ALU

I use the behavioral 32-bit ALU from the textbook directly.

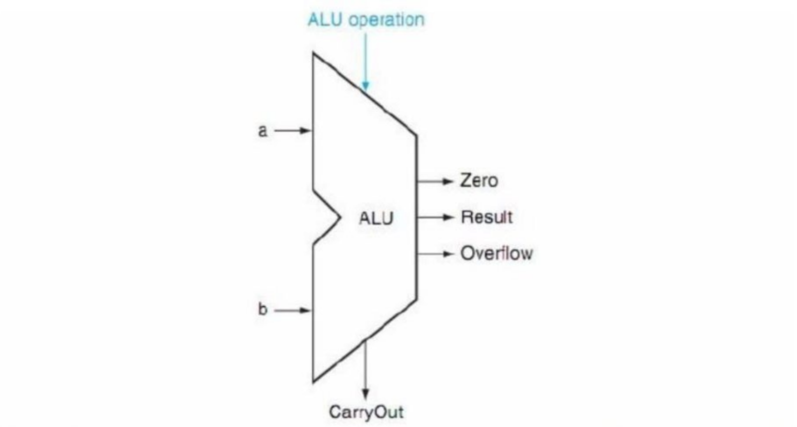


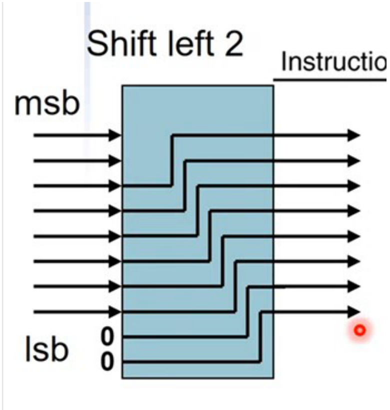

**FIGURE C.5.14 The symbol commonly used to represent an ALU, as shown in Figure C.5.12.** This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

## ➢ MUX_2to1

The function of this module is to choose one data from the given two data. So the output depends on how "select_i" specifies and chooses the corresponding data.

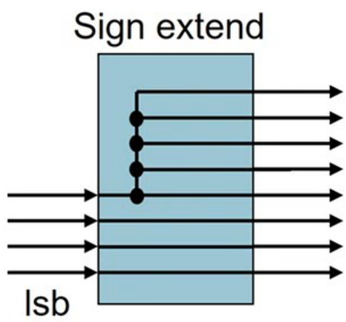
## ➢ Shift_Left_Two_32

Every bit shifts left by two bits, with the last two lsb given 0s.



## ➢ Sign_Extend

Bit 16~31 are the duplicate of msb of the input, while the remaining bits are the same as the input.




---

## ➢ ALU_Ctrl

| | Input | | | | | | | | | Output | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | fc5 | fc4 | fc3 | fc2 | fc1 | fc0 | op2 | op1 | op0 | ctr3 | ctr2 | ctr1 | ctr0 |
| add | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| sub | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| and | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| slt | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| jr | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| slti | x | x | x | x | x | x | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| beq | x | x | x | x | x | x | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| lw | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| sw | x | x | x | x | x | x | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| jp | x | x | x | x | x | x | x | x | x | x | x | x | x |
| jal | x | x | x | x | x | x | x | x | x | x | x | x | x |

given function field (fc5–fc0)    ALU opcode designed by me (op2–op0)    ALU control (ctr3–ctr0)

➔ ctr3 = 0

ctr2 = (fc1 & op1) | op0

ctr1 = (fc5 & ~fc2) | (~op1)

ctr0 = (fc2 & fc0 & op1) | (fc3 & fc1 & op1) | (op2 & op0)

- Decoder

| | Input | | | | | | Output | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | op5 | op4 | op3 | op2 | op1 | op0 | RW | alu op2 | alu op1 | alu op0 | Src | RD | Brh |
| R-f | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 01 | 0 |
| addi | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 00 | 0 |
| slti | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 00 | 0 |
| beq | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | xx | 1 |
| lw | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 00 | 0 |
| sw | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | xx | 0 |
| jp | 0 | 0 | 0 | 0 | 1 | 0 | 0 | x | x | x | x | xx | 0 |
| jal | 0 | 0 | 0 | 0 | 1 | 1 | 1 | x | x | x | x | 10 | 0 |

given op field | RegWrite | ALU opcode designed by me | ALUSrc | RegDst | Branch

➔ RW = (~op3 & op1 & op0) | (op3 & ~op0) | (~op3 & ~op2 & ~op1)
alu_op2 = (~op3 & ~op2 & ~op1) | (~op2 & op1 & ~op0)
alu_op1 = ~op3 & ~op2 & ~op1
alu_op0 = ~op5 & (op2 | op1)
Src = op5 | op3
RD[1] = ~op5 & op0
RD[0] = ~op3 & ~op2 & ~op1
Brh = op2

| | Input | | | | | | Output | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | op5 | op4 | op3 | op2 | op1 | op0 | jump | memread | memwr | memto |
| R-f | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 00 |
| addi | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 00 |
| slti | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 00 |
| beq | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | xx |
| lw | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 01 |
| sw | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | xx |
| jp | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | xx |
| jal | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 10 |

given op field | Jump | MemRead | MemWrite | MemtoReg

➔ jump = ~op5 & ~op3 & op1
memread = op5 & ~op3
memwr = op5 & op3
memto[1] = ~op5 & op0
memto[0] = op5 & ~op3

## ➢ MUX_3to1

Similar to MUX_2to1. The difference is that there are three data to choose from, so "select_i" should be two bits.

## ➢ Simple_Single_CPU

By reference to the architecture diagram, include all the required modules, and carefully connect the wires between them, done!

Some special wires:

- ✓ jr_ctr = (ALU_op[2] & ALU_op[1] & ~ALU_op[0]) & (~instr_o[5] & ~instr_o[4] & instr_o[3] & ~instr_o[2] & ~instr_o[1] & ~instr_o[0])

→ 1, if the instruction is jr

- ✓ nextAdd = branch & zero

→ 1, if the instruction is branch

- ✓ jumpAdd = { seqAdd[31:28], jumpAdd_tmp[27:0] }

→ { PC[31:28], address << 2 }

## **Finished part:**

✓ test data1

The instructions do some addition among the registers, store r1 and r2 to the memory, and load the values from the memory back to r6, r7, and r9. We see that the results are correct. r29 represents the stack pointer register value, that is, 128.

| | | | |
|---|---|---|---|
| r0= | 0 | m0= | 1 |
| r1= | 1 | m1= | 2 |
| r2= | 2 | m2= | 0 |
| r3= | 3 | m3= | 0 |
| r4= | 4 | m4= | 0 |
| r5= | 5 | m5= | 0 |
| r6= | 1 | m6= | 0 |
| r7= | 2 | m7= | 0 |
| r8= | 4 | m8= | 0 |
| r9= | 2 | m9= | 0 |
| r10= | 0 | m10= | 0 |
| r11= | 0 | m11= | 0 |
| r12= | 0 | m12= | 0 |
| r13= | 0 | m13= | 0 |
| r14= | 0 | m14= | 0 |
| r15= | 0 | m15= | 0 |
| r16= | 0 | m16= | 0 |
| r17= | 0 | m17= | 0 |
| r18= | 0 | m18= | 0 |
| r19= | 0 | m19= | 0 |
| r20= | 0 | m20= | 0 |
| r21= | 0 | m21= | 0 |
| r22= | 0 | m22= | 0 |
| r23= | 0 | m23= | 0 |
| r24= | 0 | m24= | 0 |
| r25= | 0 | m25= | 0 |
| r26= | 0 | m26= | 0 |
| r27= | 0 | m27= | 0 |
| r28= | 0 | m28= | 0 |
| r29= | 128 | m29= | 0 |
| r30= | 0 | m30= | 0 |
| r31= | 0 | m31= | 0 |

✓ test data2

This is a Fibonacci function looking for the fifth number in Fibonacci sequence ( f(4) ). The answer is stored in r2, and we see that "5" is the correct answer.

| | | | |
|---|---|---|---|
| r0= | 0 | m0= | 0 |
| r1= | 0 | m1= | 0 |
| r2= | 5 | m2= | 0 |
| r3= | 0 | m3= | 0 |
| r4= | 0 | m4= | 0 |
| r5= | 0 | m5= | 0 |
| r6= | 0 | m6= | 0 |
| r7= | 0 | m7= | 0 |
| r8= | 0 | m8= | 0 |
| r9= | 1 | m9= | 0 |
| r10= | 0 | m10= | 0 |
| r11= | 0 | m11= | 0 |
| r12= | 0 | m12= | 0 |
| r13= | 0 | m13= | 0 |
| r14= | 0 | m14= | 0 |
| r15= | 0 | m15= | 0 |
| r16= | 0 | m16= | 0 |
| r17= | 0 | m17= | 0 |
| r18= | 0 | m18= | 0 |
| r19= | 0 | m19= | 0 |
| r20= | 0 | m20= | 68 |
| r21= | 0 | m21= | 2 |
| r22= | 0 | m22= | 1 |
| r23= | 0 | m23= | 68 |
| r24= | 0 | m24= | 2 |
| r25= | 0 | m25= | 1 |
| r26= | 0 | m26= | 68 |
| r27= | 0 | m27= | 4 |
| r28= | 0 | m28= | 3 |
| r29= | 128 | m29= | 16 |
| r30= | 0 | m30= | 0 |
| r31= | 16 | m31= | 0 |

## Problems you met and solutions:

At first I couldn’t get the correct answer for Fibonacci function, the result of r2 was very large. Since I had no problem for test data1, I guessed there was something wrong with my “jal” or “jr”.

I looked for the introduction to these two instructions on the Internet, and found that I misunderstood the function of “jr”, I assigned it a wrong “ALUCtrl_o” code. So the solution was to revise the truth table, then I could get the correct answer.

## Summary:

This lab was similar to the previous one. However, I had a hard time implementing it. It was mainly due to my ambiguous understanding of the function of “jump”, “jal”, and “jr”. I spent some time trying to realize those instructions, and found they should be nothing difficult at all.

Anyway, starting with some simple units, and implement a complete single cycle CPU step by step gave me a sense of achievement!