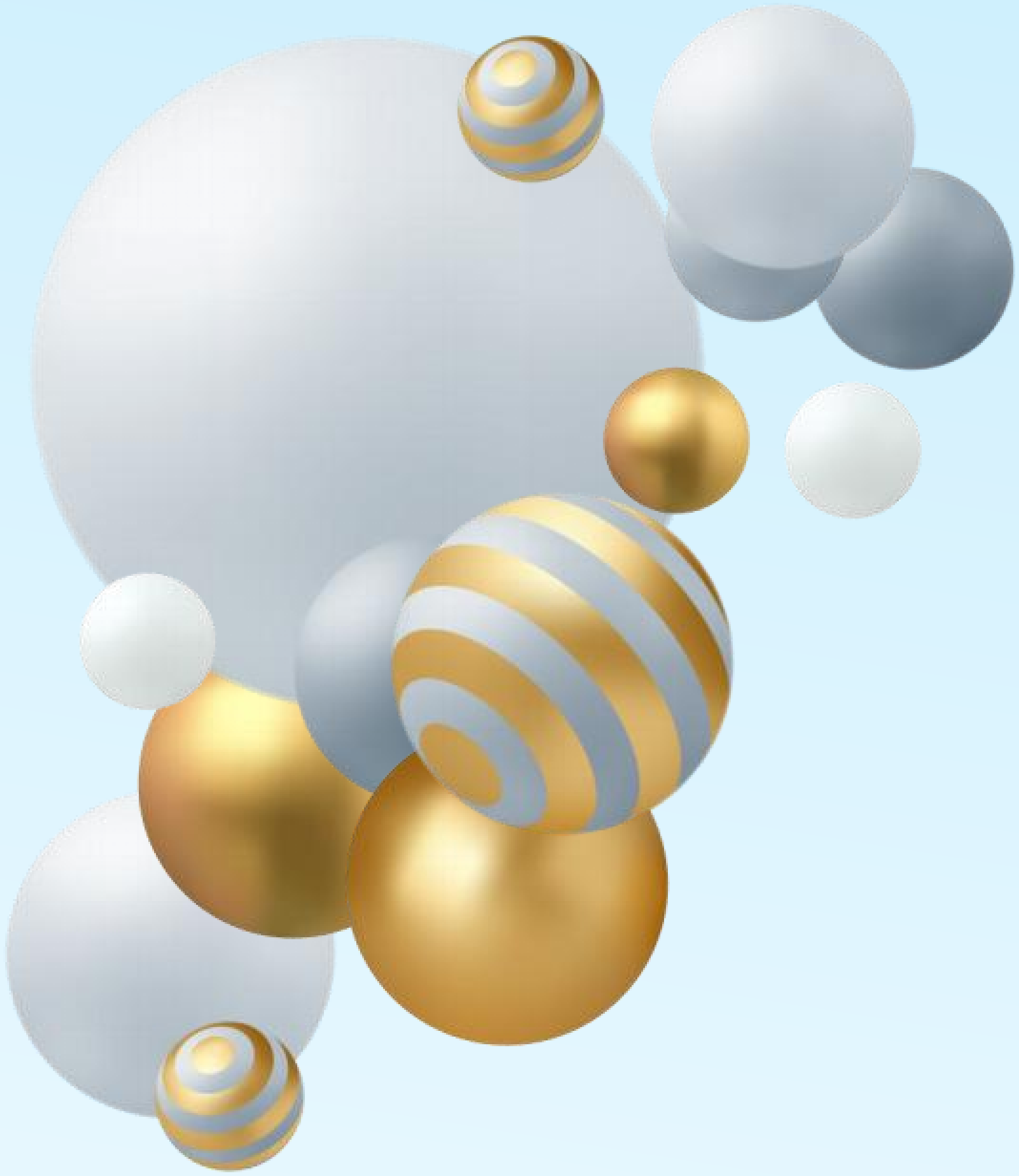


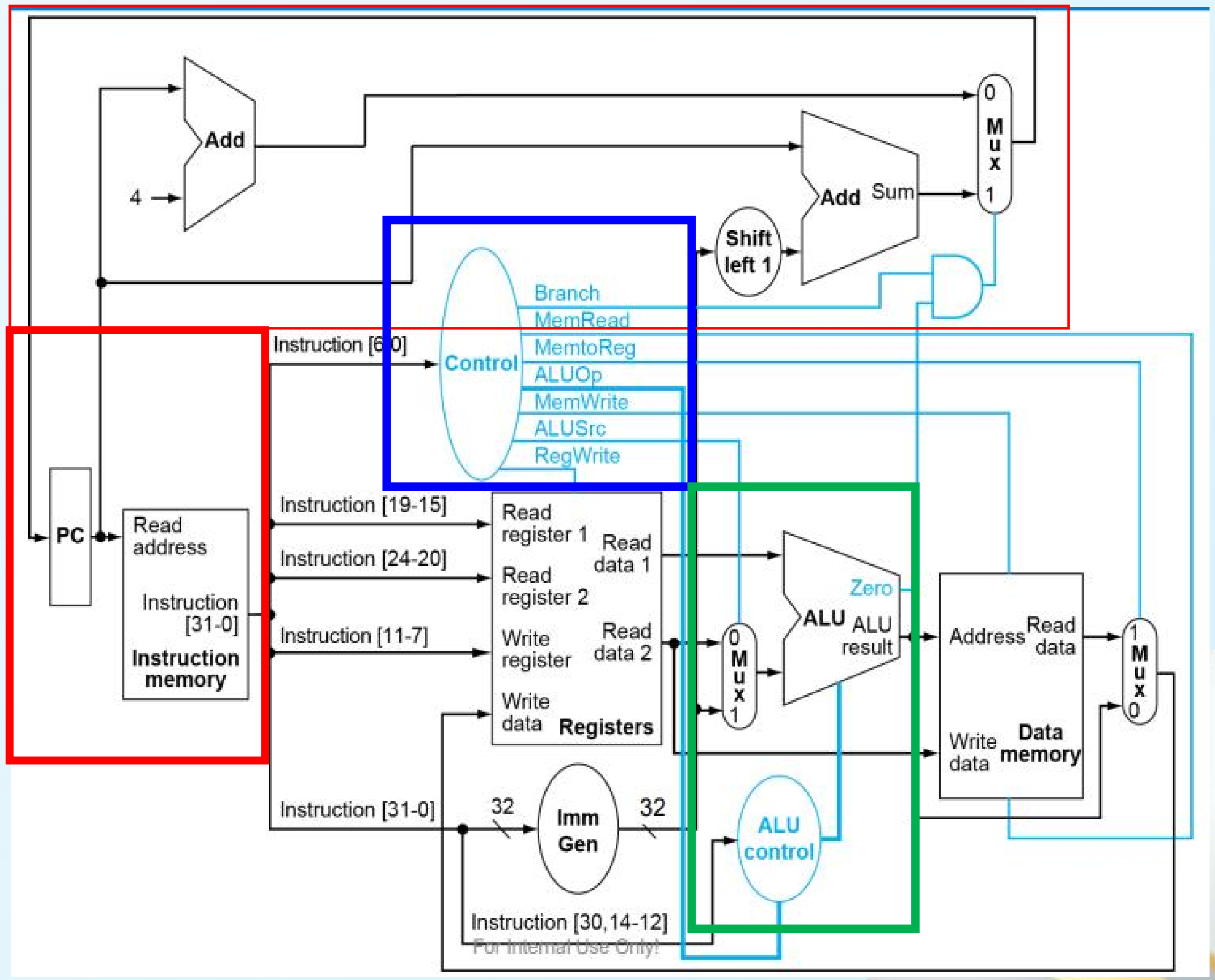
Computer Organization

Lab10 CPU Design(2)

IFetch, Controller, ALU, ‘Single cycle’ CPU

Topic

- CPU Design(2)
  - Instruction Fetch(2)
    - PC + Instruction Memory
  - Instruction Analysis(2)
    - Controller
  - Instruction Execution(2)
    - ALU
  - A ‘Single cycle’ CPU

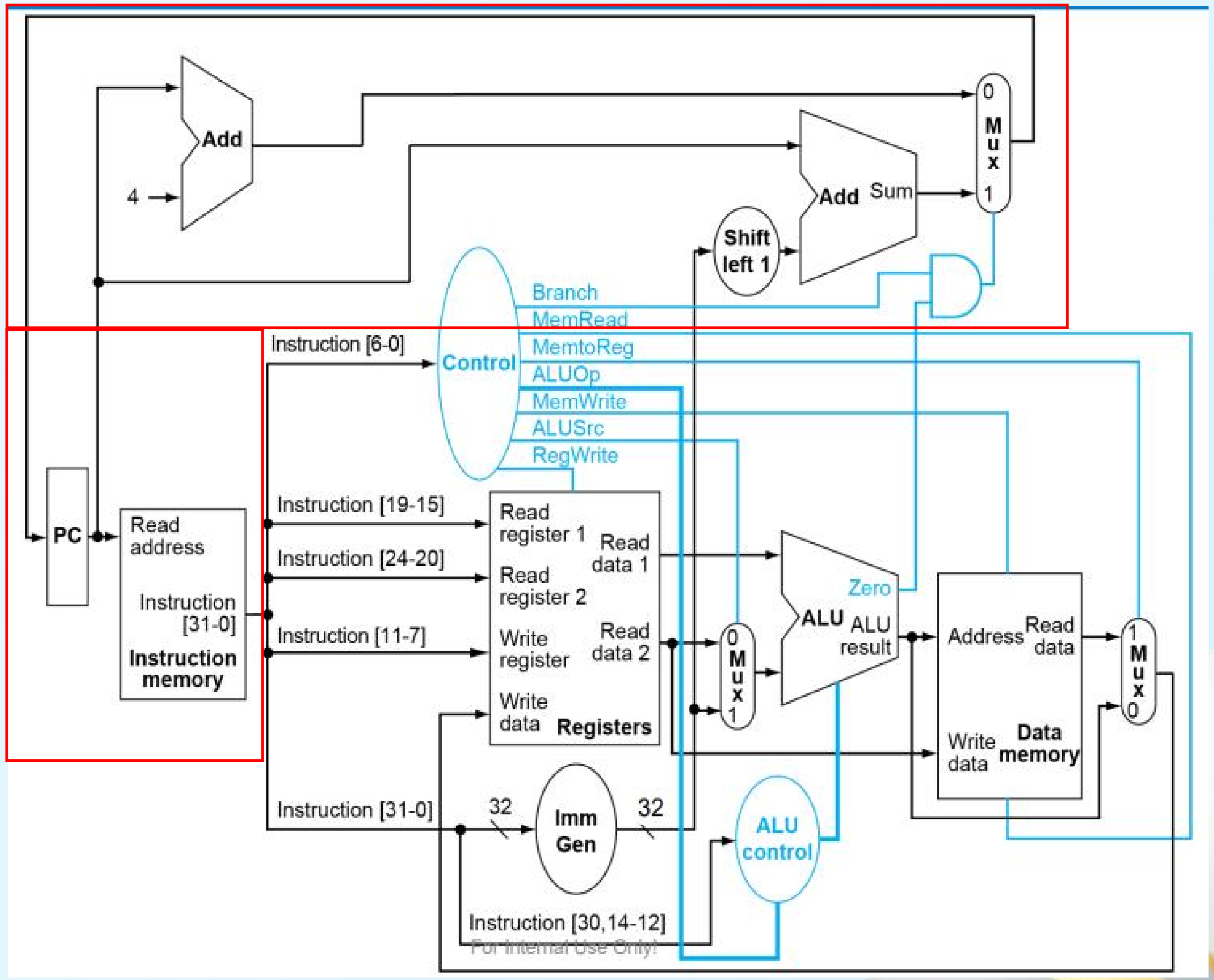


Instruction Fetch(2)-IFetch

- **IFetch(update PC, fetch instruction according to the value of PC)**
  - **input: clk, branch, zero, imm32, ?**
  - **output: instruction**
  - **PC regiester**
  - **Instruction Memory(finished in lab9)**
- **Task 1: using verilog to decribe IFetch module**

```
// Structural design in verilog
module IFetch(clk,branch, zero, imm32, instruction);

  reg [/*TBD*/:0] pc;
  prgrom urom(.clka(clk), .addra(pc[15:2]), .douta(instruction));

endmodule
```

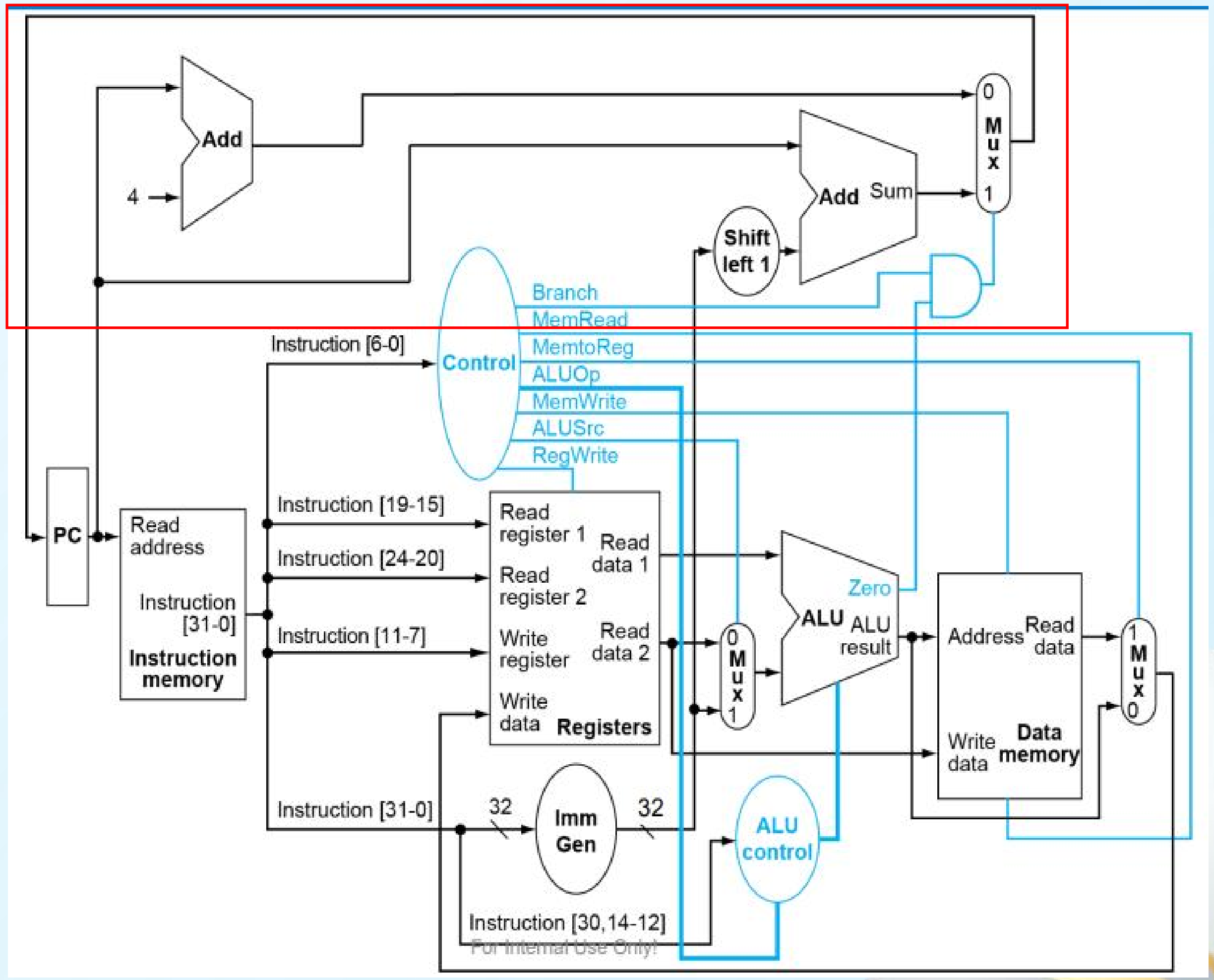
Q: Why using pc[15:2] to bind with port addra of prgrom?



IFetch continued

- Task 2: How to initialize and update PC register
  - How to initialize PC register?
    - What's the initial value
    - When to initialize
  - How to update PC register?
    - Where does the address come from
    - When to update

```
//Behavior design in verilog
always@ ( /*TBD*/) begin
        if( /*TBD*/ )                    //initialize
           pc<=/*TBD*/;
        else                             //update
          if(/*TBD*/)                    //branch
              pc<=pc+imm32;
          else                           //branch
              pc<=pc+4;                  //Sequential execution
   end
```



Practice1

- Implement the sub-module of a single cycle CPU: IFetch, which update PC register and fetch the instruction from the instruction memory according to the value of PC.
- Build the testbench to verify the function of IFetch.

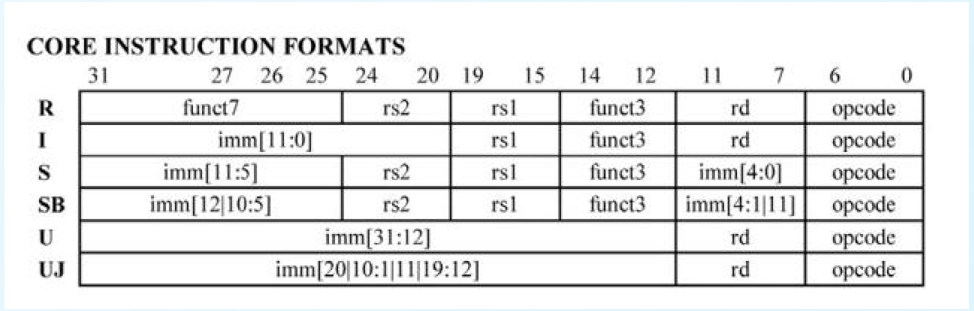
- Tips
  - Is port list in demo ok? if not, complete it.
  - The Instruction Memory and Data Memory introduced in lab9 are different from that in Rars.

```
// Structural design in verilog
module IFetch(clk,branch, zero, imm32, instruction);

/*To be complete*/


  reg [/*TBD*/:0] pc;
  prgrom urom(.clka(clk), .addra(pc[15:2]), .douta(instruction));

/*To be complete*/



//Behavior design in verilog
always@ ( /*TBD*/) begin
        if( /*TBD*/ )                   //initialize
            pc<=/*TBD*/;
        else                            //update
           if(/*TBD*/)                  //branch
                pc<=pc+imm32;
           else                         //branch
                pc<=pc+4;               //Sequential execution
  end


endmodule
```

Instruction Analysis(2) -Controller

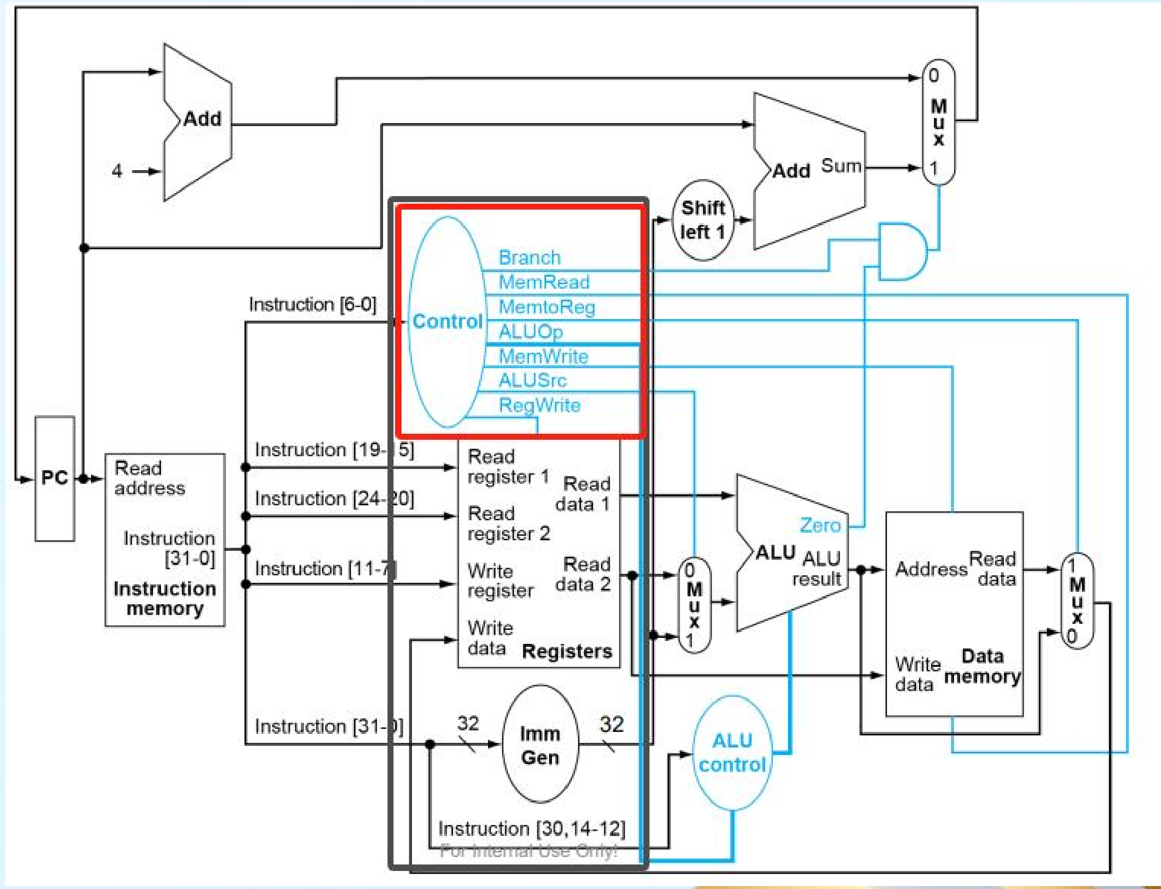
- Task 1: get information about the data from the instruction (Decoder in Data Path)
  - address of **registers**: **rs2**(Instruction[24:20]), **rs1**(Instruction[19:15]) and **rd**(Instruction[11:7])
  - **shift mount**(instruction[24:20]) (R type and I type)
  - **immediate(12bits for I/S/SB, 20bits for U/UJ)**

- **Task 2: generated control signals according to the instruction (Controller in Control Path)**
  - **get Operation code**(inst[**6:0**]) **and function code**(funct7,funct3) **in the instruction**
  - **generate control signals** to submodules of Data Path in CPU



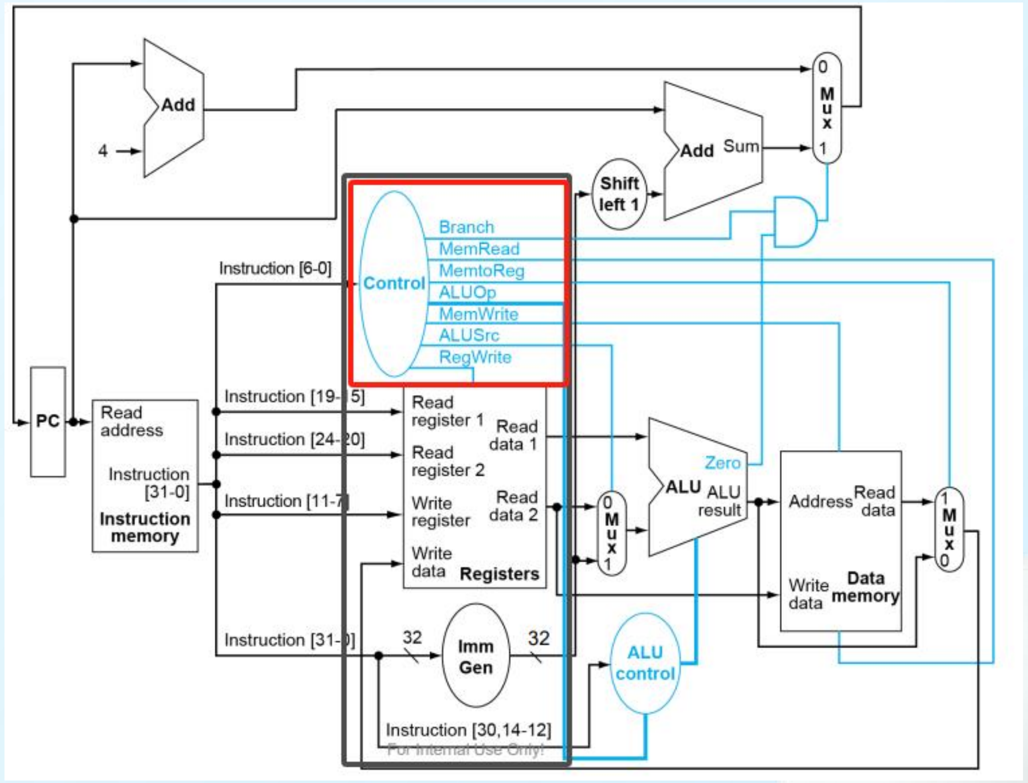
CORE INSTRUCTION FORMATS

	31 … 27 26 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode



Controller continued

- **inputs**
  - opcode: **instruction[6:0],Q: Are funct3 or funct7 needed here?**
- **outputs**
  - **Branch**: 1bit, 1’b1 while the instruction is beq, otherwise not.
  - **MemRead**:1bit, 1’b1 while need to read from Data Mem(load), otherwise not.
  - **MemtoReg**:1bit, 1’b1 while select the data read from memory to send to the Registers, otherwise not.
  - **ALUOp**: 2bit, Need to be considered together with the ALU.
  - **MemWrite**:1bit, 1’b1 while need to write to Data Mem(store), otherwise not.
  - **ALUSrc**:1bit, 1’b1 while select Immediate as the operand of ALU, otherswise select the read data2 from Registers.
  - **RegWrite**:1bit, 1’b1 while need to write to Reisters, otherwise not.



NOTES: Here is just a demo CPU which implements a subset of RISC-VI : load, store, beq, add, sub, and, or.
Larger instruction sets require deeper and more analysis and implementation.

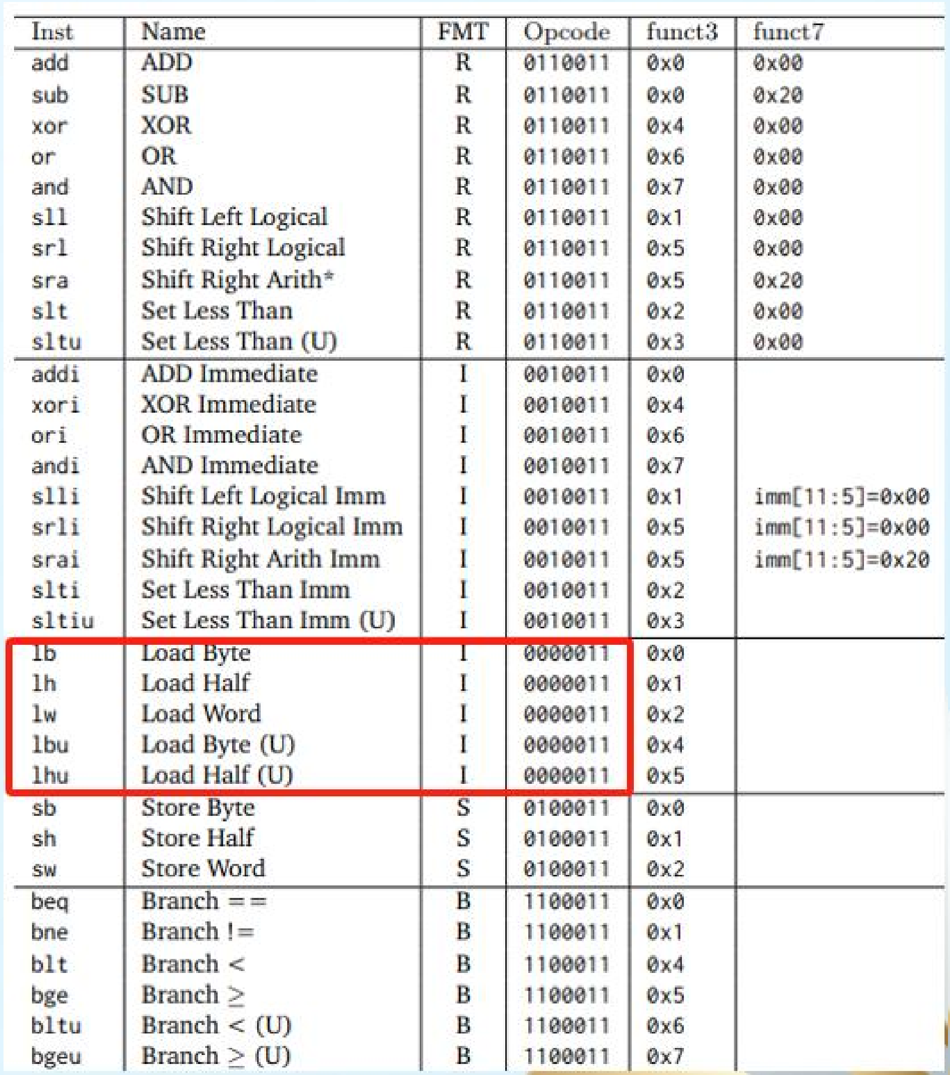
Controller continued

- **Circuit analysis**: After fetching the instruction, it is necessary to immediately align it for analysis, which does not involve storage. Therefore, a combinational logic circuit is used
- **outputs(demo1)**
  - MemRead:1bit, 1’b1 while need to read from Data Mem(load), otherwise not.
    - option1:

```
output MemRead;
assign MemRead = (instruction[6:0]==7′ h03)? 1′ b1:1′ b0;
```

    - option2:

```
output reg MemRead;
always @ *
        if( instruction[6:0]==7′ h03 )
                MemRead = 1′ b1;
        else
                MemRead = 1′ b0;
```

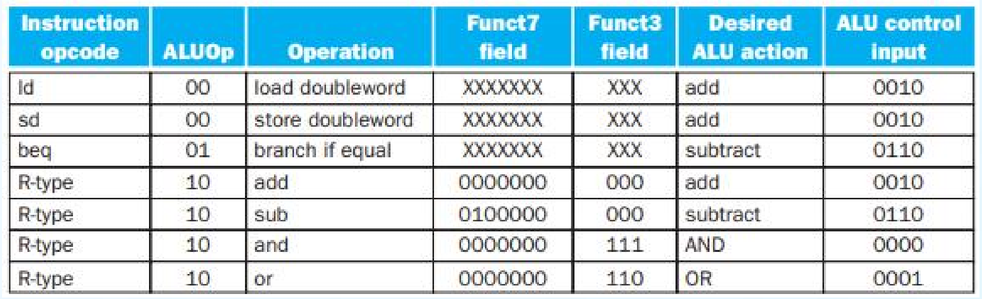


Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00
srai	Shift Right Arith Imm	I	0010011	0x5	imm[11:5]=0x20
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch <	B	1100011	0x4	
bge	Branch ≥	B	1100011	0x5	
bltu	Branch < (U)	B	1100011	0x6	
bgeu	Branch ≥ (U)	B	1100011	0x7	

Controller continued

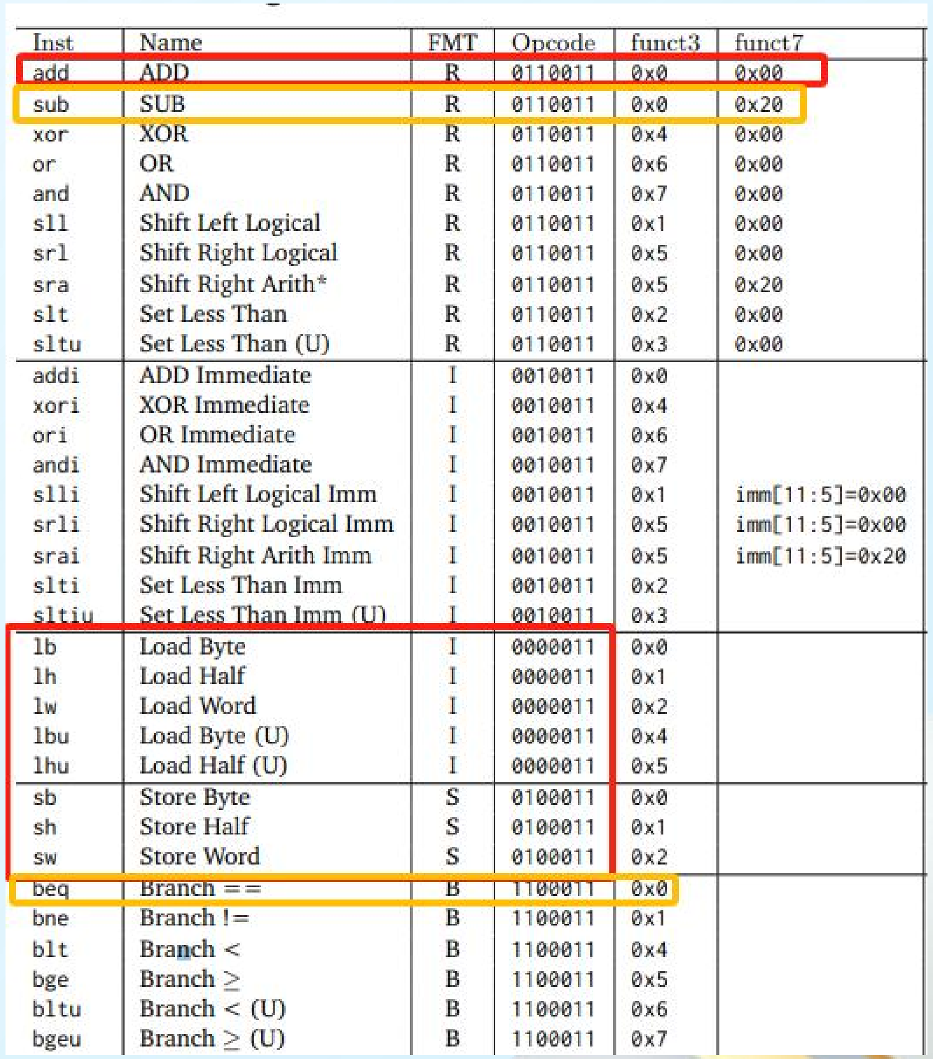
- **outputs(demo2)**
  - **ALUOp**: 2bit
    - 2'b00 : load/store
    - 2'b01: beq
    - 2'b10: R-type

  Q: What's the problem in the reference code below, fix it:



Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

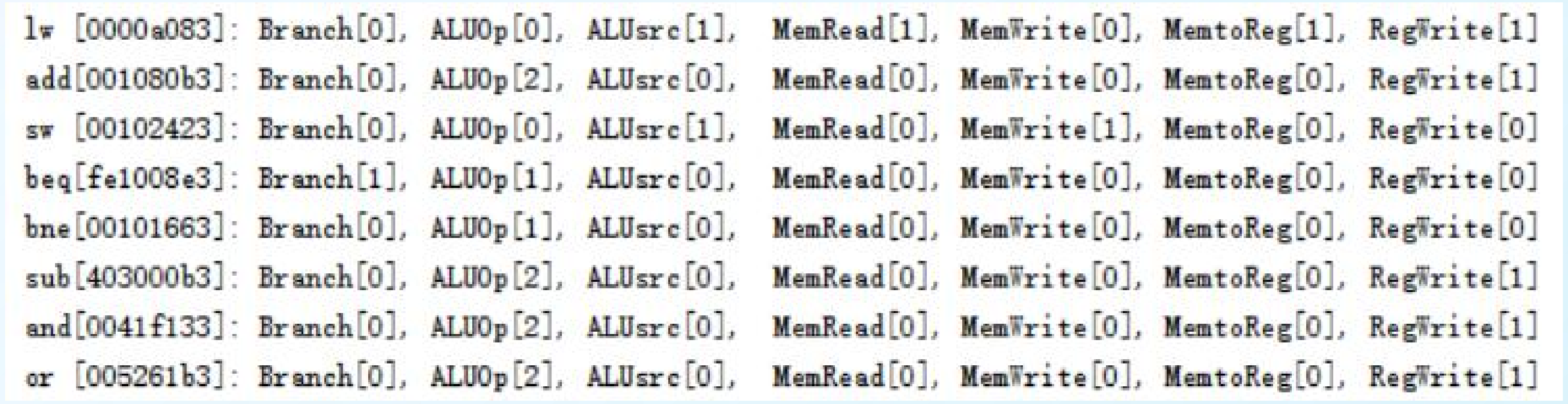
```
output [1:0] reg  ALUOp;

always @ *
        case( instruction[6:0])
                7'h03,7'h23: ALUOp = 2'b00;
                7'h33,7'h63: ALUOp = 2'b01;
                default: ALUOp = 2'b10;
        endcase
```



Inst	Name	FMT	Opcode	funct3	funct7
add	ADD	R	0110011	0x0	0x00
sub	SUB	R	0110011	0x0	0x20
xor	XOR	R	0110011	0x4	0x00
or	OR	R	0110011	0x6	0x00
and	AND	R	0110011	0x7	0x00
sll	Shift Left Logical	R	0110011	0x1	0x00
srl	Shift Right Logical	R	0110011	0x5	0x00
sra	Shift Right Arith*	R	0110011	0x5	0x20
slt	Set Less Than	R	0110011	0x2	0x00
sltu	Set Less Than (U)	R	0110011	0x3	0x00
addi	ADD Immediate	I	0010011	0x0	
xori	XOR Immediate	I	0010011	0x4	
ori	OR Immediate	I	0010011	0x6	
andi	AND Immediate	I	0010011	0x7	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00
srai	Shift Right Arith Imm	I	0010011	0x5	imm[11:5]=0x20
slti	Set Less Than Imm	I	0010011	0x2	
sltiu	Set Less Than Imm (U)	I	0010011	0x3	
lb	Load Byte	I	0000011	0x0	
lh	Load Half	I	0000011	0x1	
lw	Load Word	I	0000011	0x2	
lbu	Load Byte (U)	I	0000011	0x4	
lhu	Load Half (U)	I	0000011	0x5	
sb	Store Byte	S	0100011	0x0	
sh	Store Half	S	0100011	0x1	
sw	Store Word	S	0100011	0x2	
beq	Branch ==	B	1100011	0x0	
bne	Branch !=	B	1100011	0x1	
blt	Branch <	B	1100011	0x4	
bge	Branch ≥	B	1100011	0x5	
bltu	Branch < (U)	B	1100011	0x6	
bgeu	Branch ≥ (U)	B	1100011	0x7	

Practice2

- Implement the sub-module of CPU: Controller
  - The controller is expected to support the specified set (add, sub, and, or, lw, sw, beq), demos on the courseware can serve as a reference.
  - Build the testbench to verify the function of Controller(here is a reference):



```
lw  [0000a083]: Branch[0], ALUOp[0], ALUsrc[1],  MemRead[1], MemWrite[0], MemtoReg[1], RegWrite[1]
add[001080b3]: Branch[0], ALUOp[2], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
sw  [00102423]: Branch[0], ALUOp[0], ALUsrc[1],  MemRead[0], MemWrite[1], MemtoReg[0], RegWrite[0]
beq[fe1008e3]: Branch[1], ALUOp[1], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[0]
bne[00101663]: Branch[0], ALUOp[1], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[0]
sub[403000b3]: Branch[0], ALUOp[2], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
and[0041f133]: Branch[0], ALUOp[2], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
or  [005261b3]: Branch[0], ALUOp[2], ALUsrc[0],  MemRead[0], MemWrite[0], MemtoReg[0], RegWrite[1]
```

- it's strongly recommended to think about that which input and output ports, as well as internal processing logic, the controller needs to add if it needs to support instructions other than the specified set (add, sub, and, or, lw, sw, beq) in RISC-V32I.

Instruction Execution(2) - ALU

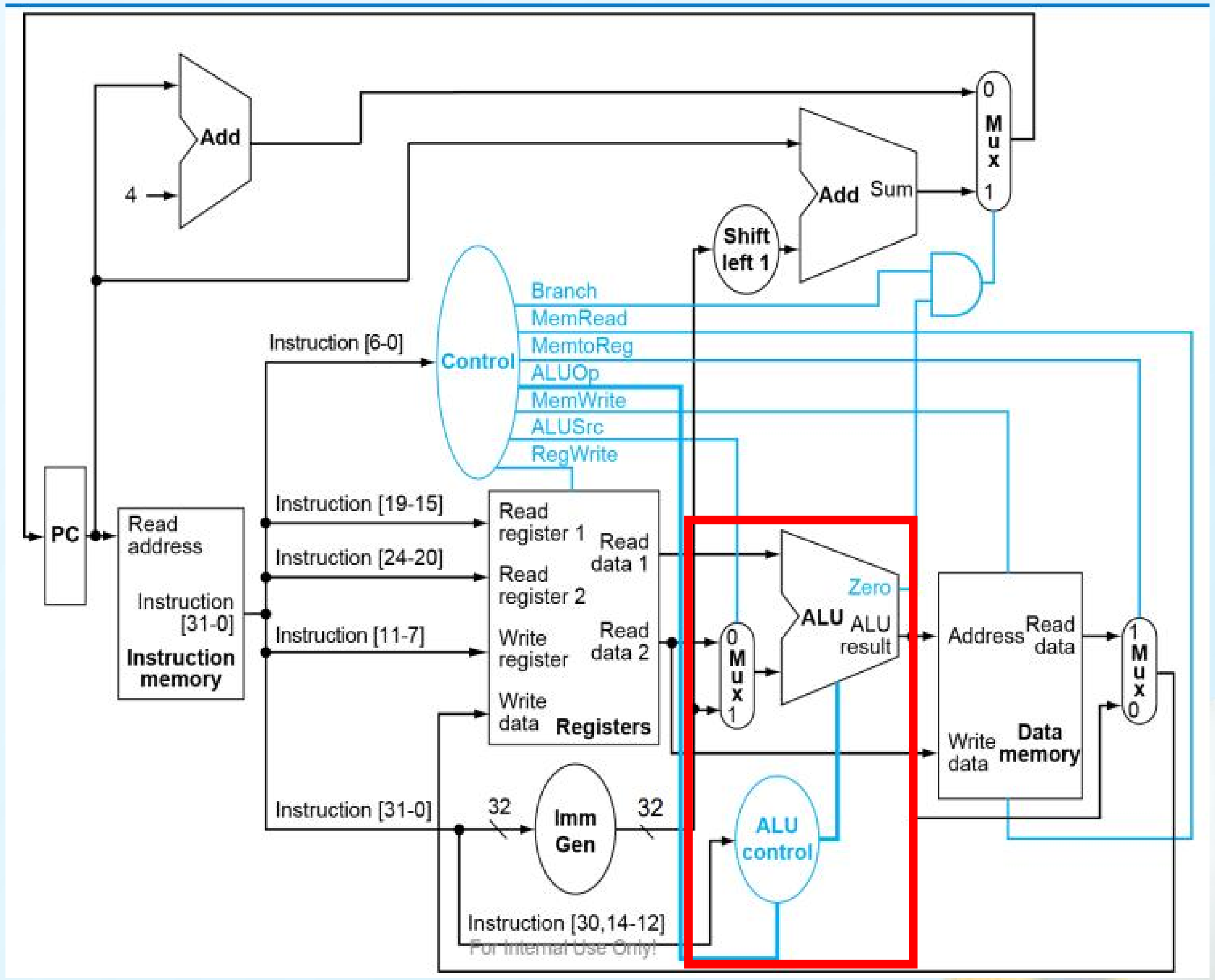
ALU:

- inputs:
  - ✓ datas from Decoder (**ReadData1**, **ReadData2**, **imm32**), all three have a 32-bit bit width
  - ✓ **ALUOp**: 2bit

  Q: Are funct3 or funct7 needed here?

- outputs:
  - ✓ **ALUResult**: 32bit, the result of calculation in ALU
  - ✓ **zero**:1bit, 1’ b1 while AULResult is zero, otherwise not

- functions:
  - ✓ 1. generate **ALUControl (4bit)**
  - ✓ 2. select **operand2** from ReadData2 and imm32
  - ✓ 3. calculate on ReadData1 and operand2 according to the **ALUControl**, assign the calculation result to **ALUResult**
  - ✓ 4. generate **zero** according to ALUResult



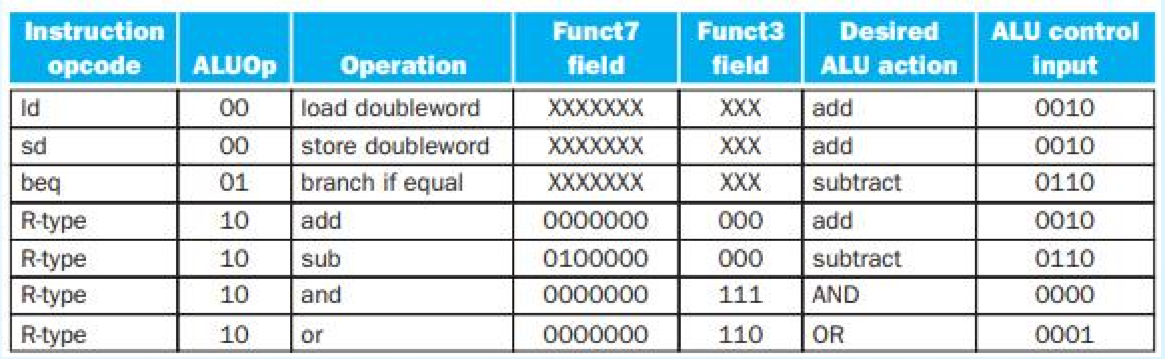
ALU continued

ALU:

- ➢ **Circuit analysis**: After get the control signals from controller and datas from Decoder, it is necessary to immediately calculate the result which does not involve storage. Therefore, a combinational logic circuit is used.

- ➢ Implements (Demo for load, store, beq, add, sub, and, or):
  - ✓ 1. generate **ALUControl**

```
reg [3:0] ALUControl;

always @ *
        case( ALUOp)
                2'b00,2'b01: ALUControl = { ALUOp, 2'b10};
                2'b10:
                //complete code here
        endcase
```



Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALU:

- ➢ Implements(Demo):
  - ✓ 3. calculate on ReadData1 and operand2 according to the ALUControl, assign the calculation result to **ALUResult**

```
output reg [31:0] ALUResult;

//operand2 is the 2nd operand for the calculation
always @ *
        case( ALUControl)
                4'b0010: ALUResult= ReadData1 + operand2;
                4'b0110:  ALUResult= ReadData1 - operand2;
                //complete code here
        endcase
```

ALU continued

ALU:

- Demos:

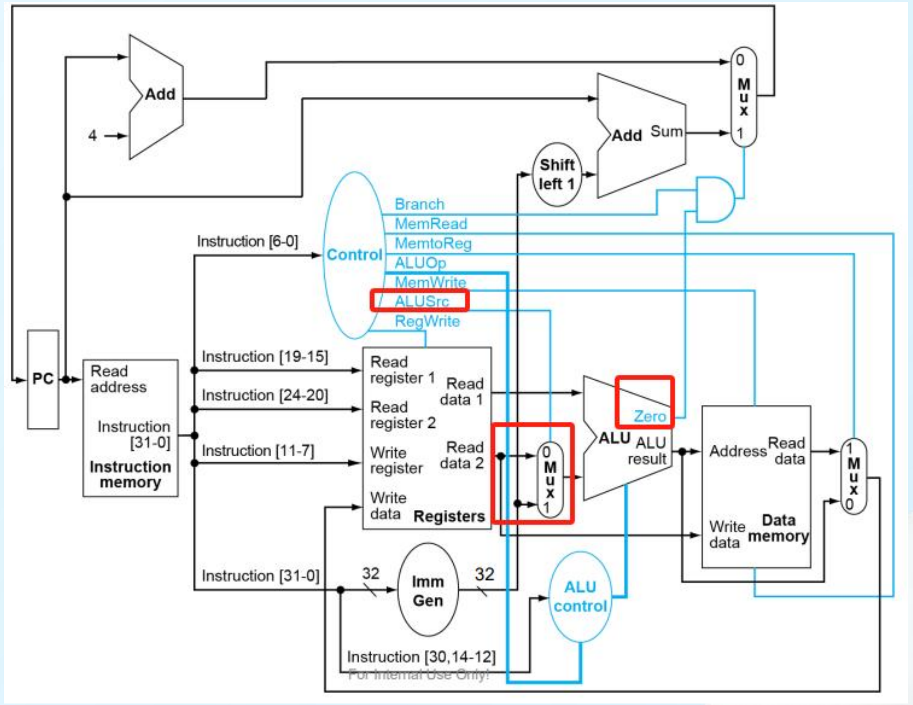
NOTES : there are some errors in the following demo codes, find them and correct

- ✓ 2. select **operand2** from ReadData2 and imm32

```
wire[31:0] operand2;

assign operand2 = (ALUSrc=1)? ReadData2 : imm32;
```

- ✓ 4. generate **zero** according to ALUResult

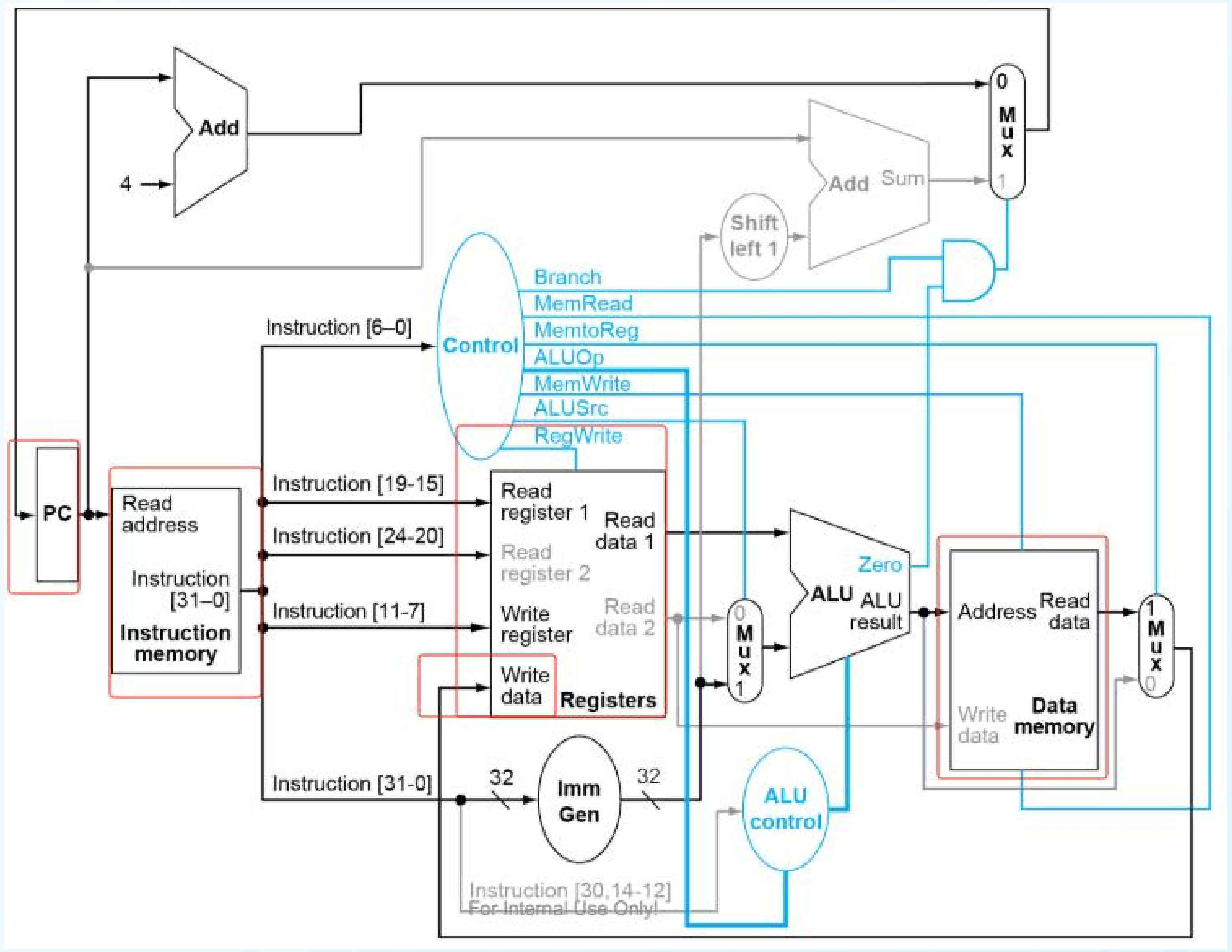
```
output reg[1:0] zero;

assign zero = (ALUResult=1)? 1’b1: 1’b0;
```



Practice3

- Implement the sub-module of CPU: ALU
  - The ALU is expected to support the specified set (add, sub, and, or, lw, sw, beq), demos on the courseware can serve as a reference.
  - Build the testbench to verify the function of ALU
    - TIPs: It’s suggested that integrating controller and alu into one module for testing after finish the function verification on Controller.
- it’s strongly recommended to think about that which input and output ports, as well as internal processing logic, the ALU needs to add if it needs to support instructions other than the specified set (add, sub, and, or, lw, sw, beq) in RISC-V32I.

A ‘single cycle’ CPU



Q1. Does it **take time** for signals to be processed and transmitted within the module, as well as between modules?

Q2. Which sub modules within CPU **need the trigger from the clock**? When does the following event occur in a clock cycle?

1-1） **IFetch**: **update** the value of PC register
1-2） **IFetch**: **fetch** the instruction according to the value of PC

2-1-1） **Controller**: generate the **control signals**
2-2-1） **Decoder**: **get** the value of register(s)
2-2-2） **Decoder**: **generate** the extended **immediate**

3-1） **ALU**: **get** the **operands**
3-2） **ALU**: **generate** the **calculation result**

4-1） **Dmemory**: **get** the **address**(from ALU) and **data**(from Decoder)
4-2） **Dmemory**: **read out the data**

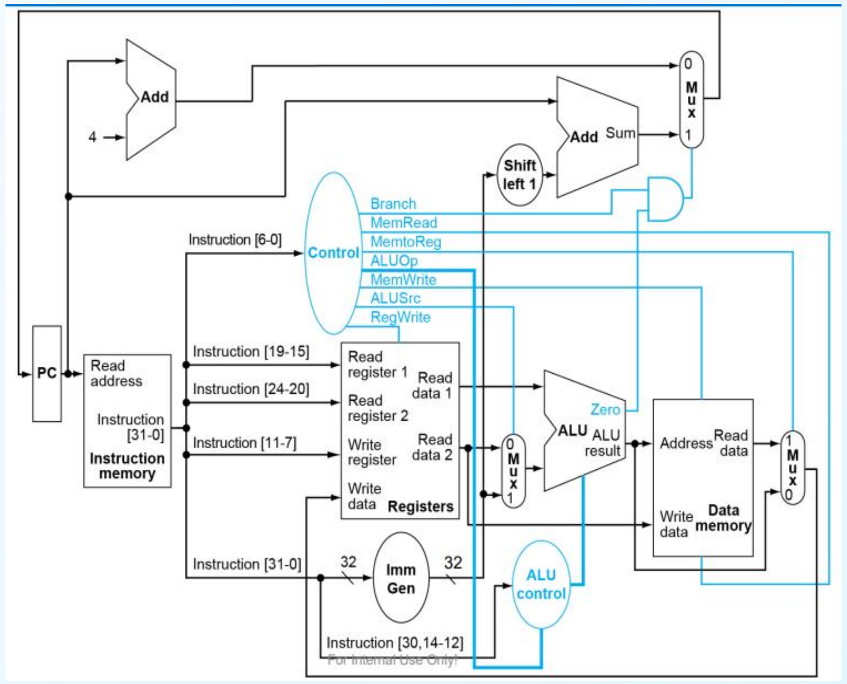
5-1） **Decoder**: **write back the data**

Build a ‘single cycle’ CPU

1) Define a CPU top module, determine its **port list.**

2) **Instantiating** the sub-modules: **Decoder**, execution unit/**ALU**, **IFetch, Controller, Data-Memory and other sub-modules if possible**.

3) Complete the **inter-module connection** inside the CPU and the **binding** to the CPU **top port.**



```
// Structural design in verilog
// JUST  a demo, need to be completed

module CPUv1(  input clk, rst );
    wire [31:0] inst, imm32;
    wire branch,zero;

    IFetch uIF( .clk(clk), .rst(rst), .branch(branch), .zero(zero), );

    Controller uCtrl(.inst(inst), .Branch(branch), );

    Decoder uDecd( .clk(clk), .rst(rst), .regWrite(RegWrite),  );

    ALU uALU( .rs1D(rs1Data), . rs2D(rs2Data), .imm32(imm32),  );

   DMem uDMem( .clk(clk),  .addr(alu_out), .din(rs2Data),  );

endmodule
```

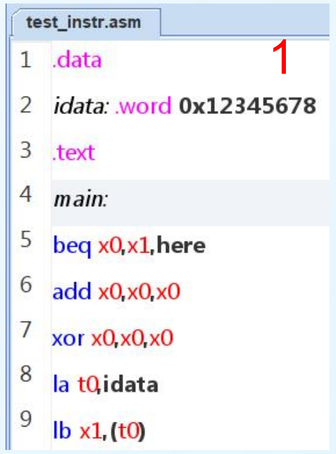
Test a ‘single cycle’ CPU(1)

Q1. How to test the CPU ?
how to determine the program, the data, how to check the result?

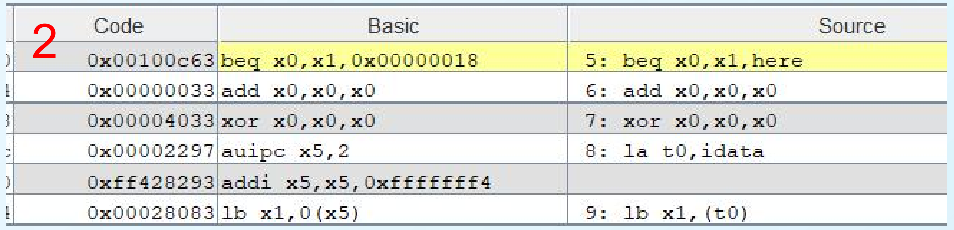
Solution 1: Simulation

```
module tb_cpu();

CPUv1 uCPU(.clk(clk), .rst(rst));
initial begin
    clk =1'b0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1'b0;
    #20 rst = 1'b1;
    #180 $finish;
end
endmodule
```

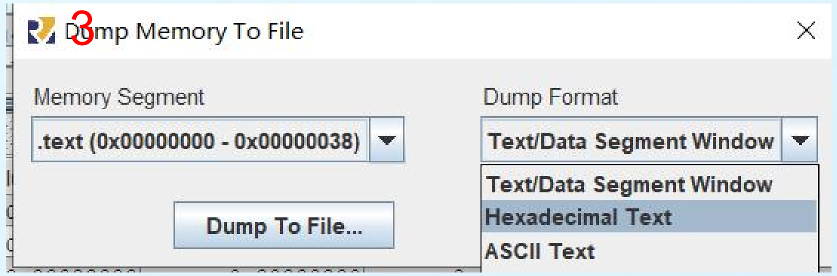
Tips: Tools in **labs/Tools** of blackboard sit are suggested to use to generate testcase.

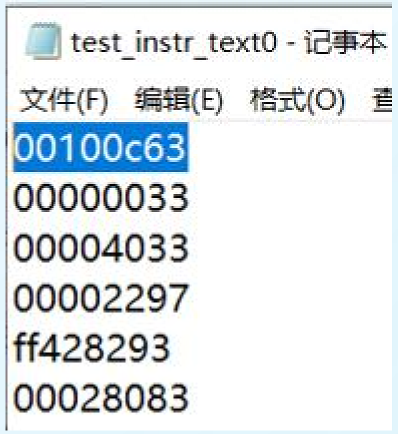


```
.data
idata: .word 0x12345678
.text
main:
beq x0,x1,here
add x0,x0,x0
xor x0,x0,x0
la t0,idata
lb x1,(t0)
```

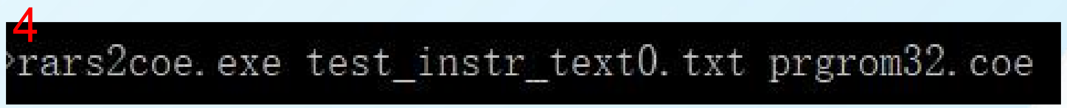


Code	Basic	Source
0x00100c63	beq x0,x1,0x00000018	5: beq x0,x1,here
0x00000033	add x0,x0,x0	6: add x0,x0,x0
0x00004033	xor x0,x0,x0	7: xor x0,x0,x0
0x00002297	auipc x5,2	8: la t0,idata
0xff428293	addi x5,x5,0xfffffff4	
0x00028083	lb x1,0(x5)	9: lb x1,(t0)



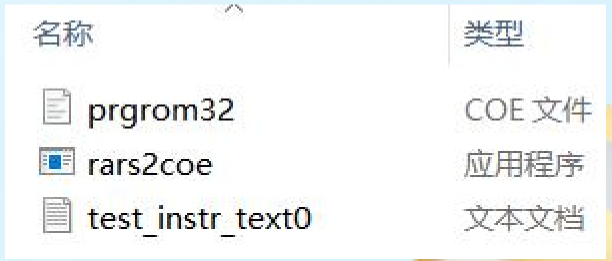


```
00100c63
00000033
00004033
00002297
ff428293
00028083
```



```
>rars2coe.exe test_instr_text0.txt prgrom32.coe
```

Tips: Method about how to generate coe file could be find in **labs/lab9**

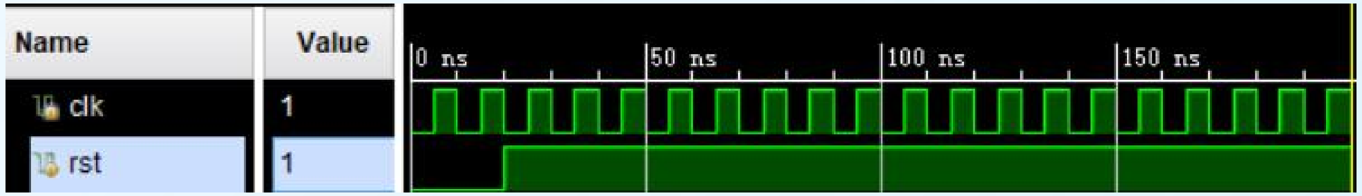


名称	类型
prgrom32	COE 文件
rars2coe	应用程序
test_instr_text0	文本文档

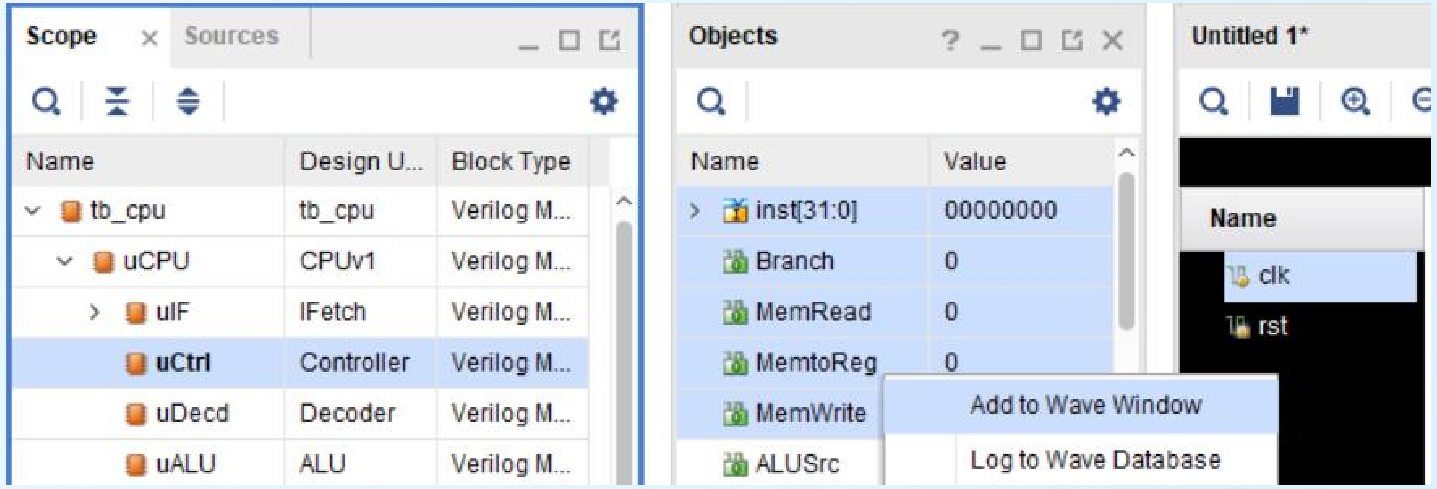
Test a 'single cycle' CPU(1) continued

Solution 1: Simulation

```
module tb_cpu();

CPUv1 uCPU(.clk(clk), .rst(rst));
initial begin
    clk =1'b0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1'b0;
    #20 rst = 1'b1;
    #180 $finish;
end
endmodule
```



The signals in testbench are not sufficient to determine whether the CPU is functioning properly.
Internal signals of the CPU could be added to the waveform to assist in the judgment.



Name	Design U...	Block Type
tb_cpu	tb_cpu	Verilog M...
uCPU	CPUv1	Verilog M...
uIF	IFetch	Verilog M...
uCtrl	Controller	Verilog M...
uDecd	Decoder	Verilog M...
uALU	ALU	Verilog M...

Name	Value
inst[31:0]	00000000
Branch	0
MemRead	0
MemtoReg	0
MemWrite	
ALUSrc	

Tips: First, select the **submodule** to be observed in the **scope** window, and then select the **signal** to be observed in the submodule in the **object** window. **Right click on these signals**, select "**Add to wave Window**", and **perform the simulation again.**