Verilog Implementation Report

Computer Organization Project 4

Jaewon Chung, 2015-11832
Due: 2019/04/15

1. Introduction

Through this project, I will:

- implement a single-cycle CPU module that supports five TSC instructions (ADD, ADI, LHI, JMP, WWD).
- understand the two main components of a CPU: **datapath** and **control unit**.
- understand the communication between the CPU and the memory.

2. Design

This section is replaced with the attached design documentation. There are two versions: design 1.0 and design 2.0. I wrote the former before implementing the CPU, and the latter during implementation. Design 2.0 updates the control unit of the CPU, while datapaths stay the same as 1.0. The key difference is that the ALU control module was removed, with the control unit operating the ALU.

The testbench 1. simulates the memory and 2. checks the output of the CPU. When the CPU requests access to an `address` in the memory by asserting `readM`, after `` `READ_TIME ``, and if `readM` is asserted, the memory sends the data to the CPU and notifies it by asserting `inputReady`. After `` `STABLE_TIME `` the data is removed. Instruction output check is done according to `num_inst`. This part is trivial.

3. Implementation

In general, I sticked with the structure in the 'Putting it all together' slide. In this section, I will only describe the parts that I did not.

1) ALU (different from project 1)
   - Input: 16 bit A , 16bit B, 4 bit OP
     Output: 16 bit C, 1 bit bcond
   - No carry in and carry out bits. The TSC ISA does not require that.
   - The operations that use the ALU are highlighted pink in the design. There are thirteen such operations, so there are thirteen OPs (0 ~ 12) plus one for undefined (15).

2) RF (same with project 2)
   - To be exact though, I changed synchronous active low reset to asynchronous active low reset to match the reset timing with the CPU module. Everything else is the same.
   - Just in case you need explanation on the RF, I attached the RF report from project 2 at the back.

3) Control

- Input: 4 bit opcode, 6 bit function code
  Output: RegDst, Jump, Branch, MemRead, MemtoReg, 3 bit ALUOp, MemWrite, ALUSrc, RegWrite, **OpenPort**
- The control module receives not only the opcode but also the function code of the instruction, and there is no separate ALU control module. The output ALUOp is directly connected to the ALU input OP.
- I spent some time looking at other instructions and tried to implement them too, although I didn't test them.
- The meaning of each output is described in detail at the module comment. Everything is the same with the slide, but I added an OpenPort output that is asserted only when the current instruction is WWD. The ouptut_port of the CPU is not z only when OpenPort is asserted.

4) CPU

- PC, nextPC: With the current PC, the nextPC is calculated and stored. At next posedge, PC receives nextPC. The instruction is fetched with nextPC too (i.e. address receives nextPC).
- Reset timeline

```
1. reset_n reaches negedge.
     (CPU) PC = nextPC = address = num_inst = 0
     (RF) register = 0
```

- A timeline of a single cycle

```
1. clk is low. Previous instruction is executed and nextPC is calculated.
2. clk reaches posedge.
     (CPU)
        PC <= nextPC
        address <= nextPC
        readM <= 1
        num_inst <= num_inst + 1
        reading_instruction <= 1
     (RF)
        if RegWrite is asserted, data is written to register.
3. memory senses posedge of readM.
     after READ_DELAY,
        data <= instruction
        inputReady <= 1
4. CPU senses posedge of inputReady. (with always @(posedge inputReady))
     if reading_instruction is asserted, memory access is in instruction fetch mode.
        instruction <= data
        readM <= 0
        reading_instruction <= 0
     else, memory access is in data fetch mode.
        dataReg <= data
        readM <= 0
5. Back to 1.
```

4. Discussion

- At first, I implemented the output_port as:
```
assign output_port = ReadData1;
```
  Although this implementation passes all the tests, I later realized this was a security flaw of the CPU, since the content of a certain register is always leaking. That's why I implemented the OpenPort control signal.
- As I was told that I will have to implement all other instructions of the TSC ISA, and since I have two exams next week, I tried to build the framework for implementing other instructions. Also, I implemented the data fetch part, not only the instruction fetch part. These parts are not complete yet, and will have flaws, but it's good enough to pass the given testbench.
- Feedback: Great pain, Great gain, Great fun. I loved it.

5. Conclusion

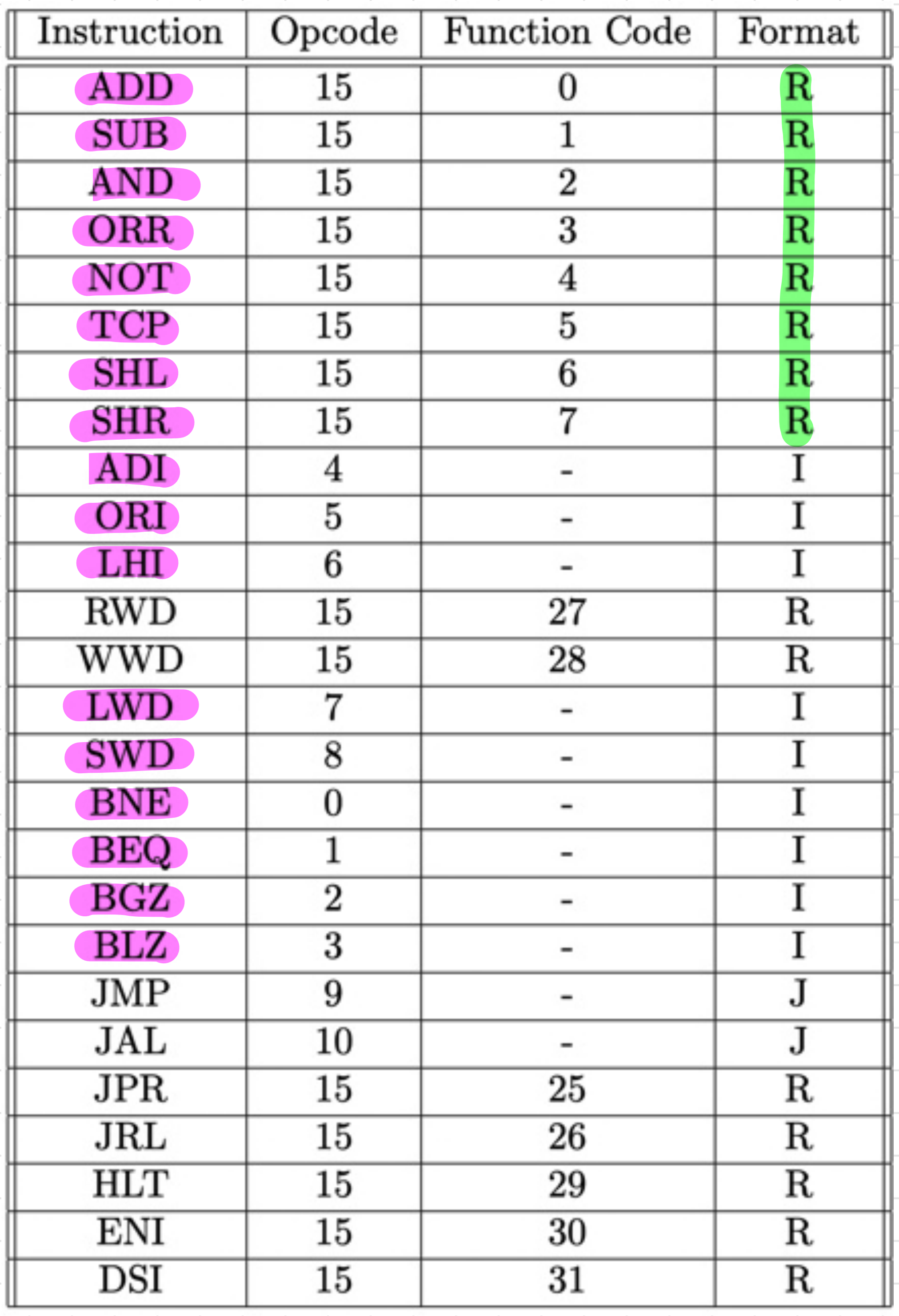
- I believe I have achieved every goal specified in the introduction.

[Attachments]

(1) CPU Design 1.0
(2) CPU Design 2.0
(3) Verilog Implementation Report for Register File (Project 2)

Design 1.0

ALU input A, B, OP . output C, bcond



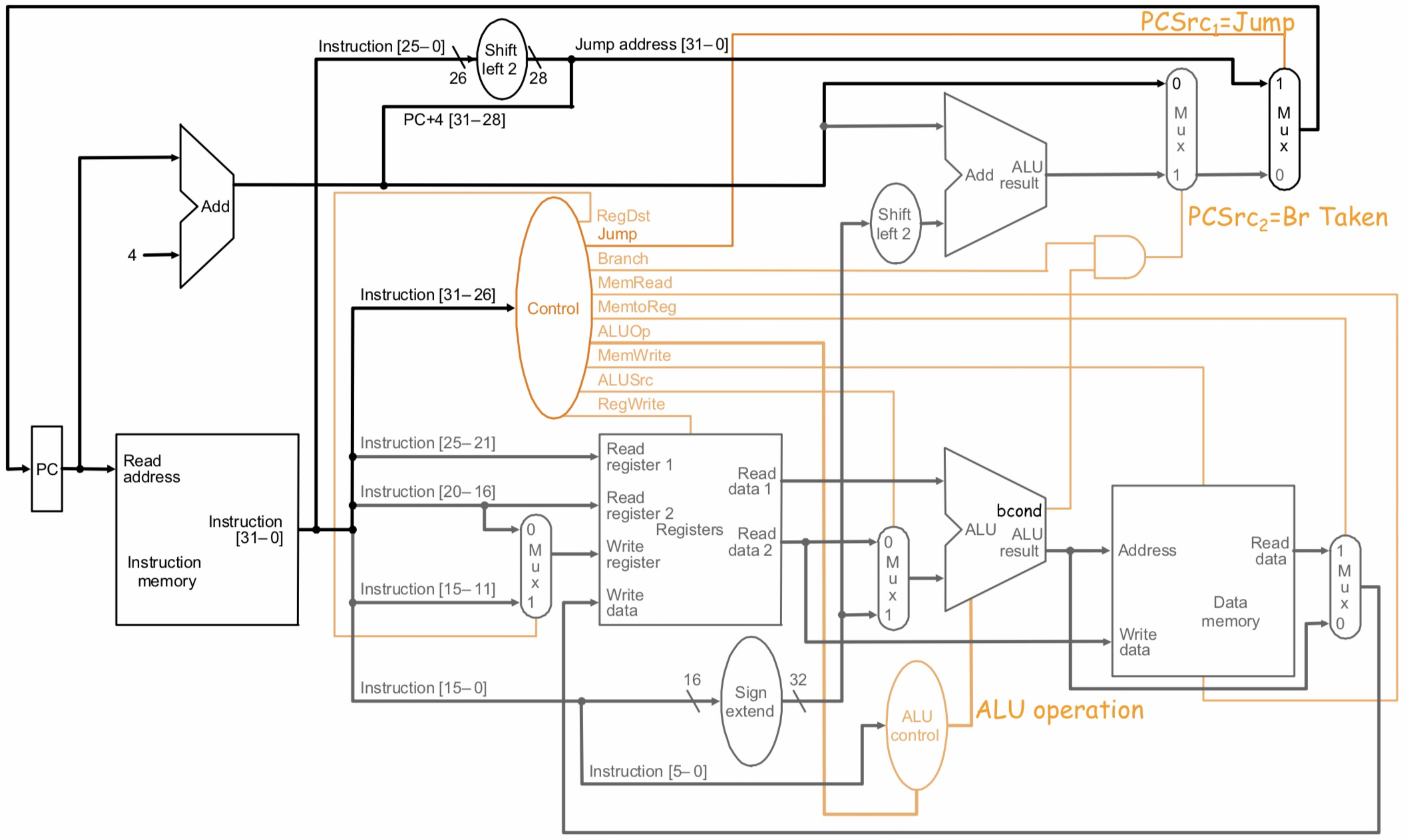
Instruction	Opcode	Function Code	Format	OP (4 bits)	
ADD	15	0	R	0	
SUB	15	1	R	1	
AND	15	2	R	2	
ORR	15	3	R	3	
NOT	15	4	R	4	
TCP	15	5	R	5	
SHL	15	6	R	6	
SHR	15	7	R	7	⇒ C = A>>>1 arithmetic!
ADI	4	-	I	0	
ORI	5	-	I	3	
LHI	6	-	I	8	⇒ C = {B[7:0], 8'b0}
RWD	15	27	R		
WWD	15	28	R		
LWD	7	-	I	0	
SWD	8	-	I	0	
BNE	0	-	I	9	bcond = (C != 0)
BEQ	1	-	I	10	bcond = (C == 0)
BGZ	2	-	I	11	bcond = (C > 0)
BLZ	3	-	I	12	bcond = (C < 0)
JMP	9	-	J		
JAL	10	-	J		
JPR	15	25	R		
JRL	15	26	R		
HLT	15	29	R		
ENI	15	30	R		
DSI	15	31	R		

JPR, JRL
another jump signal?

15 output all z.

ALU control (input 4 bit ALUOp, 6 bit Func. output 4 bit OP)

Instruction (opcode)	ALUOp (from control)	OP (to ALU)
R-type (15)	0	Func < 8 ? Func : 15
BNE (0)	1	9
BEQ (1)	2	10
BGZ (2)	3	11
BLZ (3)	4	12
ADI (4)	5	0
ORI (5)	6	3
LHI (6)	7	8
LWD (7)	8	0
SWD (8)	9	0
default	15	15



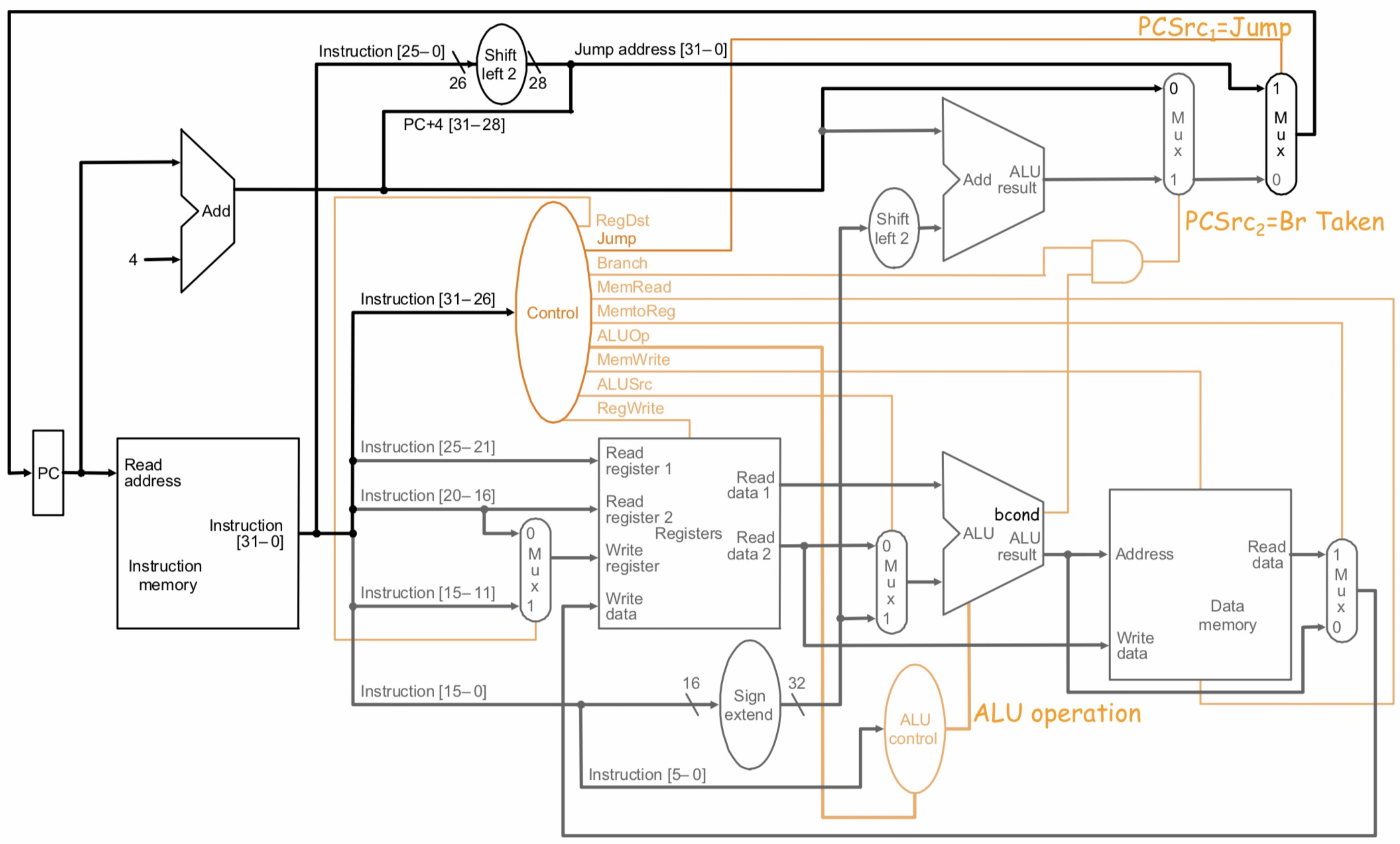
RF : no delay read, clk posedge write → need to prepare write data in one cycle.

↳ ALU result (no delay) or Memory read (from data)

change address, change readM to 1.

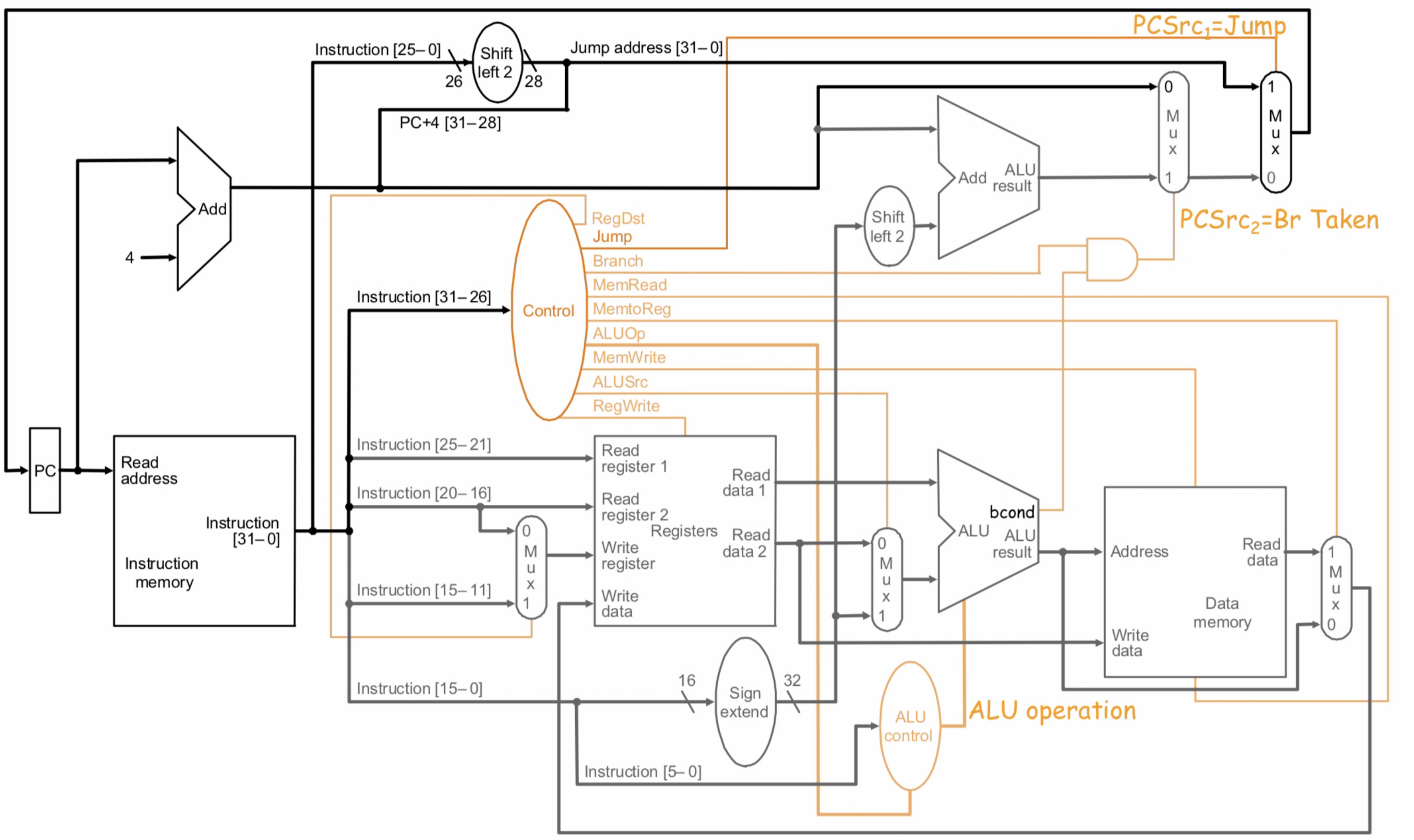
wait for input Ready to become 1. then we have data

input :		write	: control signal RegWrite
		clk	same with outside
		reset_n	
	2 bit	addr1	: instruction [11:10]
	2 bit	addr2	: instruction [9:8]
	2 bit	addr3	: RegDst ? instruction[7:6] : instruction[9:8]
	16 bit	data3	: MemtoReg ? ALUresult : dataReg
output :	16 bit	data1	
	16 bit	data2	



control   input 4bit opcode ← instruction[15:12]

RegDst :	opcode == OPCODE_R
Jump :	opcode == OPCODE_JMP, JAL
Branch :	opcode == OPCODE_BEQ, BNE, BGZ, BLZ
MemRead :	opcode == OPCODE_LWD, SWD
MemtoReg :	opcode == OPCODE_LWD
ALUOp :	defined in ALU page! (4 bit)
MemWrite :	opcode == OPCODE_SWD
ALUSrc :	opcode != OPCODE_R, BEQ, BNE
RegWrite :	opcode != OPCODE_SWD, BEQ, BNE, BGZ, BLZ JMP, (JPR, JRL)



MUX : just use ?:

sign extend : 8bit to 16bit. out = { 8{in[7]}, in} ;

address adder : just use + operator.

shift left two : out = in << 2 ;

PC : 16 bit reg.

instruction : 16 bit reg. $\Big\rangle$ need to distinguish when reading memory.

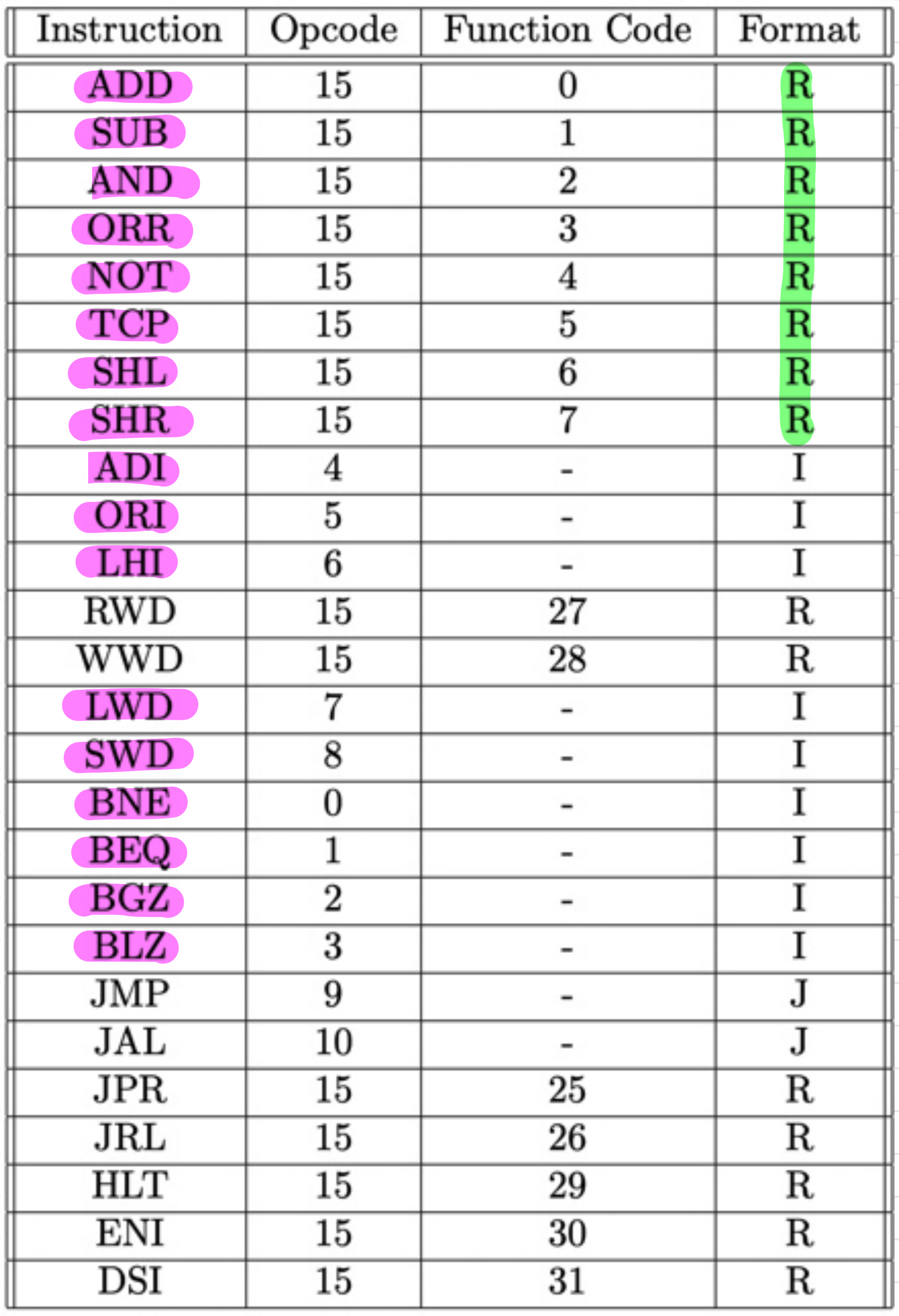
dataReg : 16 bit reg

num_inst : increment by 1 with always @ (posedge clk)

output_port : just assign to data1 from RF.

Design 2.0

ALU input A, B, OP . output C, bcond



Instruction	Opcode	Function Code	Format	OP (4 bits)	
ADD	15	0	R	0	
SUB	15	1	R	1	
AND	15	2	R	2	
ORR	15	3	R	3	
NOT	15	4	R	4	
TCP	15	5	R	5	
SHL	15	6	R	6	
SHR	15	7	R	7	⇒ C = A>>>1 arithmetic!
ADI	4	-	I	0	
ORI	5	-	I	3	
LHI	6	-	I	8	⇒ C = {B[7:0], 8'b0}
RWD	15	27	R		
WWD	15	28	R		
LWD	7	-	I	0	
SWD	8	-	I	0	
BNE	0	-	I	9	bcond = (C != 0)
BEQ	1	-	I	10	bcond = (C == 0)
BGZ	2	-	I	11	bcond = (C > 0)
BLZ	3	-	I	12	bcond = (C < 0)
JMP	9	-	J		
JAL	10	-	J		
JPR	15	25	R		
JRL	15	26	R		
HLT	15	29	R		
ENI	15	30	R		
DSI	15	31	R		

Control input 4bit opcode, 6 bit func → 4bit ALUOp

Instruction	ALUOp
opcode=15 & func<8	func
ADI	0
ORI	3
LHI	8
LWD	0
SWD	0
BNE	9
BEQ	10
BGZ	11
BLZ	12
default	15

Verilog Implementation Report

Jaewon Chung, 2015-11832
2019/03/28

Assignment 2-2: Register File

1. Introduction

- The assignment is to implement a register file, which will be used in the implementation of our single cycle cpu. It performs 2 reads and 1 writes according to the specified address and write enable signal.
- Write must be synchronized to the clock. In other words, write must be performed at the positive edge of the clock, and when write is enabled.
- The number of registers needed is 4, since the given address is 2 bits.

2. Design

- From the constraint that the smallest possible number of register should be used, I figured I could implement this with just one reg. Since I need four 16 bit registers, I can declare one 64 bit register and slice it into four parts.
- Similar to the 010 detector state transition, write should be done when the clock at its positive edge. Thus we can use the same code structure for writes.
- For reads, it seems easier to implement asynchronously, since I can use the assign statement.

3. Implementation

- A 64 bit register is declared. The register is addressed from the right; address 3, 2, 1, 0. Thus if the given address is `addr`, the data we are looking for is `register[addr*16+15 : addr*16]`, or `register[addr*16+: 16]`.
- Other parts are trivial. No need for explanation.

4. Discussion

- I planned to use four 16 bit registers, but I found a way to use only one. I believe this is of better analogy to the real register file, since physical memory is just a long array.

5. Conclusion

- I believe I have achieved every goal specified in the introduction.