Verilog Implementation Report

Computer Organization Project 5

Jaewon Chung, 2015-11832
Due: 2019/04/29

1. Introduction

- Above the single cycle CPU I built last project, I will upgrade it to a multicycle version. Hence I will have to newly implement registers, datapath, and especially the control module.

2. Design

- Design details can be found at the design document attached at the end of this report.
- This CPU implements a vertical micro-code controller. In other words, the control module sends the ROM which register transfers will occur at its current state (or cycle), and the ROM module returns appropriate control signals by referencing its hard-coded truth table.
- There are 17 states (0~16) for the control module (thus it is a FSM), where each at state at most two register transfers are conducted. There are 14 types (0~13) of register transfers, each requiring 16 bit control signals.
- The ROM module has register transfers hard-coded in it. Since there are 14 types of register transfers and each have 16 bit control signals, the size of the ROM memory is 14*16. When there are two register transfers in a single state, the **elementwise-OR** of the two control signals is returned.
- The ALU and RF modules were not changed.

3. Implementation

- At every positive edge of the clock, the following happens.
  - `ReadData1` from RF is latched into register A.
  - `ReadData2` from RF is latched into register B.
  - `ALUResult` from ALU is latched into register `ALUOut`.
  - Control module `state` is advanced. (Refer to p.3 in design document)
  - PC is updated to `nextPC` if PC write is enabled.
- I separated each part of the CPU (Declarations and Instantiations, CPU reset, Outward signals, Memory access, PC update logic, RF logic, and ALU logic) with comment blocks. It won't be difficult to read.

4. Discussion

- My CPU took 3488 cycles. You said you won't take points off for this, right?
- **<u>Feedback</u>**: The testbench evaluates the CPU with `WWD` and `num_inst`. This was fine when the CPU handled only one instrcution per cycle, but it wasn't for this project. Since the testbench checks for the test `num_inst` EVERY cycle, and since `output_port` is only available at the ID stage, testbench `$displays` were triggered multiple times for one instruction. Thus I had to develop some extra logic to show the real `num_inst` for exactly one cycle for each instruction, and keep an `internal_num_inst` register. A possible solution for this would be to require a `output_port_ready` signal coming out of the CPU, which is asserted only when valid information is coming out of the `output_port`. Now the testbench can test the output only when both `num_inst` matches the test `num_inst` AND `output_port_ready` is high.

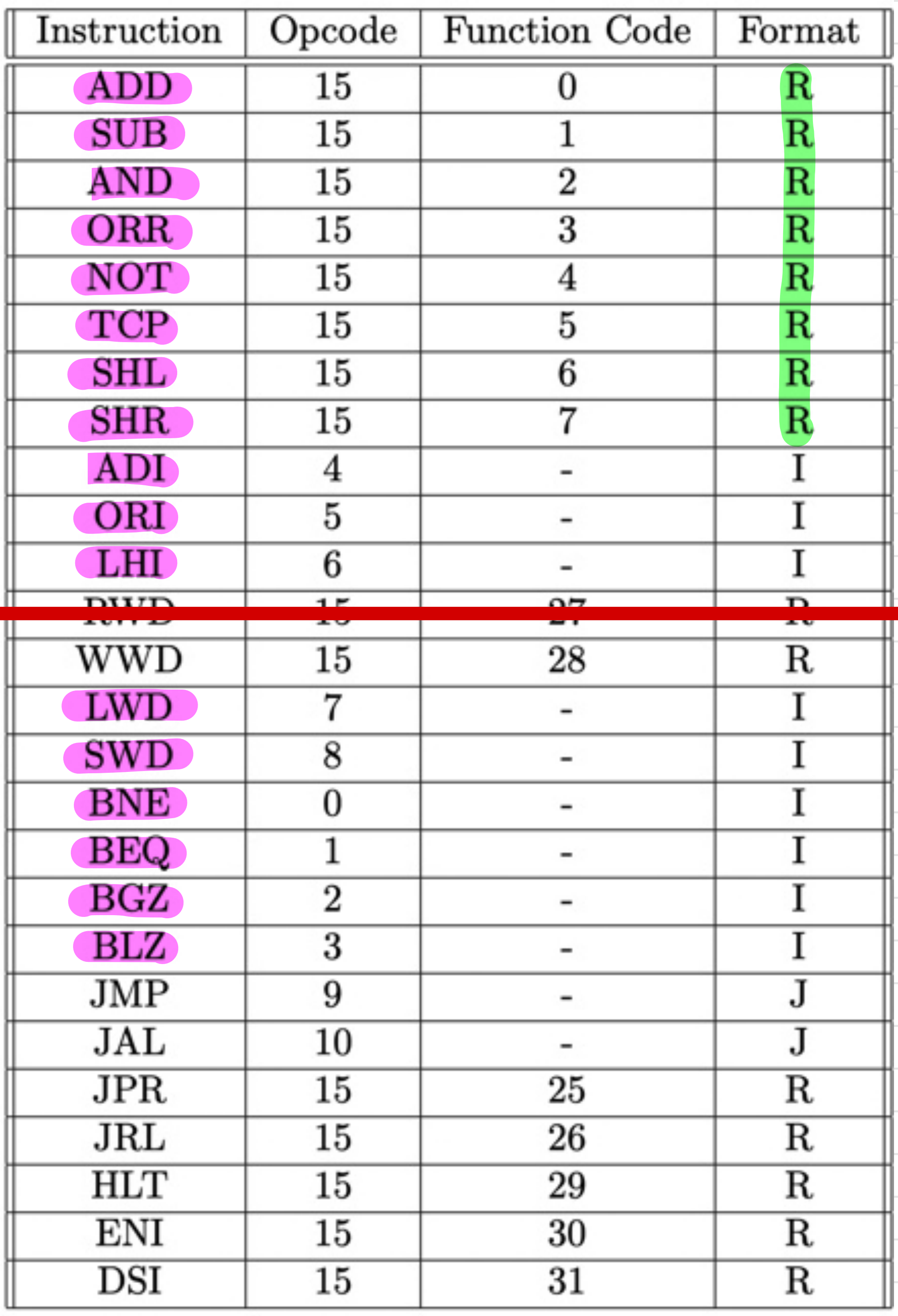
5. Conclusion

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

[Attachment]
1. Design document

Multi cycle

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	branch_cond = (C != 0)
BEQ	1	-	I	10	branch_cond = (C == 0)
BGZ	2	-	I	11	branch_cond = (C > 0)
BLZ	3	-	I	12	branch_cond = (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

RF

input : write : control signal RegWrite

clk
reset_n
$\rangle$ same with outside

2 bit addr1 : instruction [11:10]

2 bit addr2 : instruction [9:8]

2 bit addr 3 : RegDst ? instruction [7:6] : instruction [9:8]

16 bit data3 : MemtoReg ? ALUresult : dataReg

output : 16 bit data1

16 bit data2

Others

MUX : just use ?:

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

address adder : just use + operator. ← should use ALU.
(and PC ← PC+1)

shift left two : out = in << 2 ;

PC : 16 bit reg.

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

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

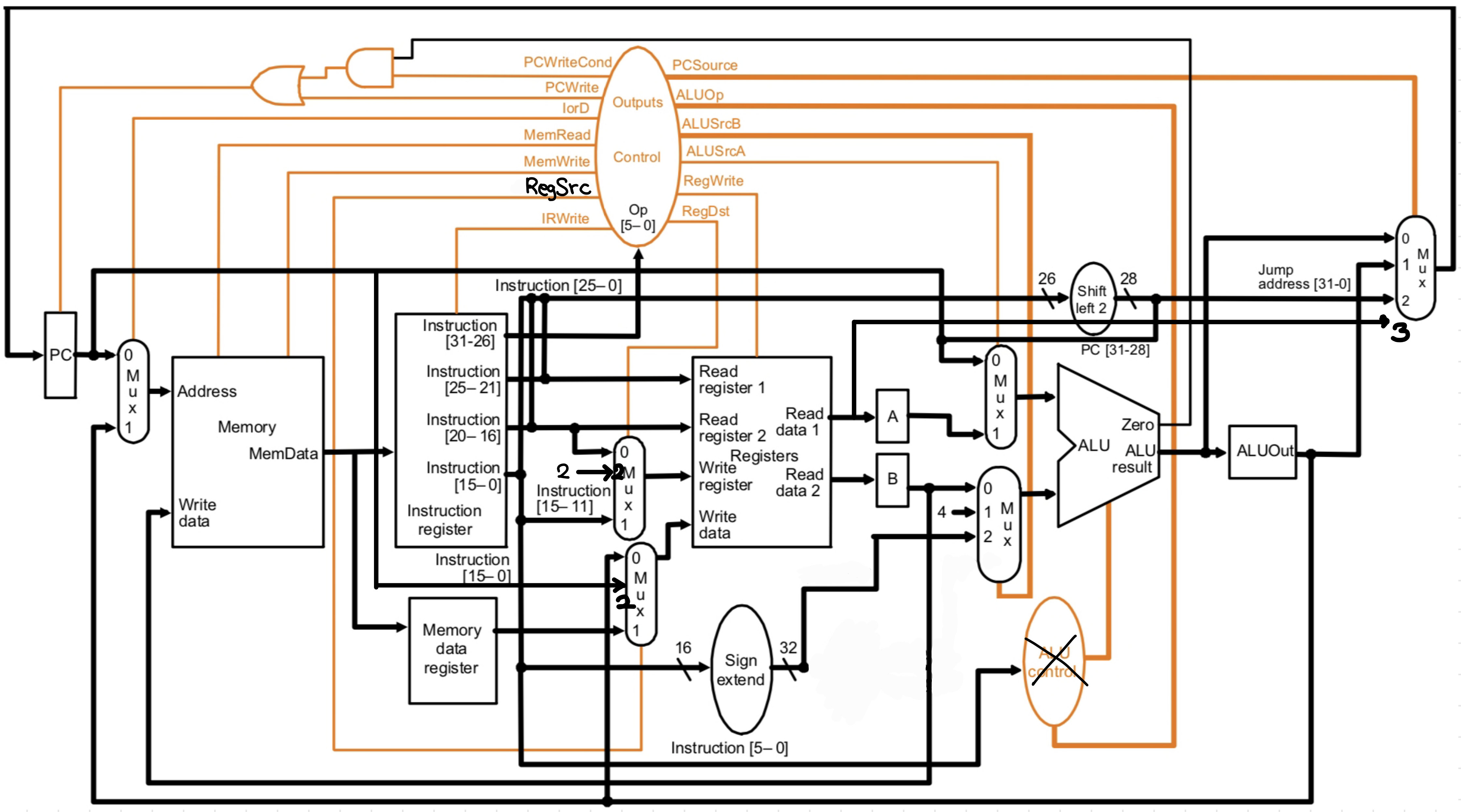
output_port : just assign to <u>data1</u> from RF.

L-type ALU	R-type ALU	LWD	SWD	Branch	JMP	JAL	JPR	JRL
				0 inst ← MEM[PC] $\boxed{0}$ 1 PC ← PC+1				
$\boxed{1}$ A ← RF[inst[11:10]]	$\boxed{2}$ A ← RF[inst[11:10]] B ← RF[inst[9:8]]	$\boxed{1}$ A ← RF[inst[11:10]]	$\boxed{2}$ A ← RF[inst[11:10]] B ← RF[inst[9:8]]	$\boxed{3}$ A ← RF[inst[11:10]] B ← RF[inst[9:8]] 2 ALUOut ← PC + sign-extend(inst[7:0])	$\boxed{4}$ 3 PC ← PC[15:12], inst[11:0]	$\boxed{5}$ 4 RF[2] ← PC 3 PC ← PC[15:12], inst[11:0]	$\boxed{6}$ 5 PC ← RF[inst[11:10]]	$\boxed{7}$ 4 RF[2] ← PC 5 PC ← RF[inst[11:10]]
$\boxed{8}$ 6 ALUOut ← ALU(A, inst[7:0]) ↳ ALUOp	$\boxed{9}$ 7 ALUOut ← ALU(A,B) ↳ ALUOp	$\boxed{10}$ 6 ALUOut ← A + sign-extend(inst[7:0])	$\boxed{10}$ 6 ALUOut ← A + sign-extend(inst[7:0])	$\boxed{11}$ 7 bcond ← ALU(A,B) 8 PC ← ALUOut if bcond				
$\boxed{12}$ 9 RF[inst[9:8]] ← ALUOut	$\boxed{13}$ 10 RF[inst[7:6]] ← ALUOut	$\boxed{14}$ 11 DataReg ← MEM[ALUOut]	$\boxed{15}$ 12 MEM[ALUOut] ← B					
		$\boxed{16}$ 13 RF[inst[9:8]] ← DataReg						

control module gets opcode and func.
its state code is $\boxed{n}$
→ sends RT code to ROM
(at most two each cycle)
control module takes care of ALUOp.

WWD :	OpenPort == 1	at	state	$\boxed{2}$
HLT :	Halt == 1	at	state	$\boxed{2}$

ROM gets at most two RT codes.
For each RT code, there are control signals hard-coded
⇒ All of them are "OR"ed and sent to the control unit.
⇒ Control unit splits the signals to each control signal.



0 IorD = 0, ReadM = 1, WriteM = 0, IRWrite = 1

1 PCWrite = 1, PCSrc = 00, ALUSrcA = 0, ALUSrcB = 01

2 ALUSrcA = 0, ALUSrcB = 10

3 PCWrite = 1, PCSrc = 10

4 RegSrc = 10, RegWrite = 1, RegDst = 10

5 PCWrite = 1, PCSrc = 11

6 ALUSrcA = 1, ALUSrcB = 10

7 ALUSrcA = 1, ALUSrcB = 00

8 PCWriteCond = 1, PCSrc = 01, ALUSrcA = 1, ALUSrcB = 00

9 RegSrc = 00, RegWrite = 1, RegDst = 00

10 RegSrc = 00, RegWrite = 1, RegDst = 01

11 IorD = 1, ReadM = 1, WriteM = 0

12 IorD = 1, ReadM = 0, WriteM = 1

13 RegSrc = 01, RegWrite = 1, RegDst = 00