

# Project1 Report

## 1. Modules explanation

- (a). Adder.v : Adder module reads data1\_in(32 bits) and data2\_in(32\_bits) as input, and then calculate the sum of data1\_in and data2\_in. Finally, outputs the result data\_o(32 bits).
- (b). MUX32.v : MUX32 module reads data1\_i(32 bits), data2\_i(32\_bits)and select\_i(1 bit) as input. If select\_i equals to 0, then outputs the result data\_o(32 bits) that equals to data1\_i. Otherwise, outputs the result data\_o(32 bits) equals to data2\_i.
- (c). Control.v : Control module reads Op\_i(7 bits) and NoOp\_i(1 bit) as input, and then outputs ALUOp\_o(2 bits), ALUSrc\_o(1 bit), RegWrite\_o(1 bit), MemtoReg\_o(1 bit), MemRead\_o(1 bit), MemWrite\_o(1 bit) and Branch\_o(1 bit). By reading the Opcode of instruction, we can determine the value of these outputs. If NoOp occurs, set all outputs to 0 but ALUOp\_o as 11, which is the state of bubble I defined.
- (d). ALU\_Control.v : ALU\_Control module reads funct\_i(7+3 bits), ALUOp\_i(2 bits) as input, and then outputs ALUCtrl\_o(4 bits). Here, I use the information in funct\_i and ALUOp\_i to distinguish which instructions(12 kinds, including NoOp) are, and then mark these instructions from 0000 to 1011, which is the output ALUCtrl\_o.
- (e). ALU.v : ALU module reads data1\_i(32 bits), data2\_i(32\_bits) and ALUCtrl\_i(3 bits) as input, and outputs data\_o(32 bits) and Zero\_o(1 bit). First, we can find which instruction is and the operator from the information in ALUCtrl\_i, then we can caluculate data\_o by the determined operator and the two input data.
- (f). IF\_ID.v : IF\_ID module acts as pipeline register. When the input clk\_i(1 bit) is positive edge, pass the signals down to the next stage. If condition stall occurs, pass the instruction(32 bits) as the previous one, rather than the new instruction. If condition flush occurs, pass the instruction(32 bits) as 32'b0 with a view to flushing wrong instruction that is passing down already.
- (g). ID\_EX.v : ID\_EX module acts as pipeline register. When the input clk\_i(1 bit) is positive edge, pass the signals down to the next stage.
- (h). EX\_MEM.v : EX\_MEM module acts as pipeline register. When the input clk\_i(1 bit) is positive edge, pass the signals down to the next stage.
- (i). MEM\_WB.v : MEM\_WB module acts as pipeline register. When the input clk\_i(1 bit) is positive edge, pass the signals down to the next stage.
- (j). ImmGen.v : ImmGen module reads data\_i(32 bits) as input, and find the information of immediate bits in data\_i. Then, extend it to 32 bits by filling the most significant immediate bit of data\_i. Finally, outputs this result data\_o(32 bits).
- (k). MUX4.v : MUX4 module reads data1\_i(32 bits), data2\_i(32\_bits), data3\_i(32\_bits)and select\_i(1 bit) as input. If select\_i equals to 00, then outputs the result data\_o(32 bits) that equals to data1\_i. If select\_i equals to 01, then outputs the result data\_o(32 bits) that equals to data2\_i. Otherwise, outputs the result data\_o(32 bits) equals to data3\_i.
- (l). And.v : And module reads data1\_i(1 bit), data2\_i(1 bit) as input. Output the result data\_o(1 bit) equals to data1\_i & data2\_i.
- (m). Forwarding\_unit.v : Forwarding\_unit module handle the forwarding issue. It read RegWrite and

register destination in MEM and WB stage, also rs1 and rs2 in EX stage. Then, it output the control signal Forward A(2 bits) and Forward B(2 bits).

- (n). Hazard\_Detection.v : Hazard\_Detection module handle the hazarding issue. If it detected hazard, it outputs NoOp\_o = 1, Stall\_o = 1, PCWrite = 0. Otherwise, it outputs NoOp\_o = 0, Stall\_o = 0, PCWrite = 1.
- (o). CPU.v : CPU module reads clock signals, reset bit, start bit as input, and then construct the whole circuit with all of the modules. Finally, a single-cycle RISC-V CPU is complete.

## 2. Difficulties Encountered and Solutions in This Project

- (a). At first, I was confused by the clock signal in Resisters.v and Data\_Memory.v. I can't realize why set clock in these two gate and fail to set the clock correctly. Then, I was finally solved the problem by drawing the signal wave in gtkwave. In this way, I can figure out the clock problem in each cycle.
- (b). When I was implementing the hazard unit, I failed to set NoOp operand correctly for a while. Then, I define a ALUOp = 2'b11 by myself, with a view to regarding NoOp as an operand, too. Therefore, I can handle this special case in the following gates afterward.

## 3. Development Environment

- (c). OS : Ubuntu 16.04
- (d). Compiler : Iverilog