Project2 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 distination 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.

(p). dcache_controller.v: dcache_controller module communicate between CPU, dcache_sram and Data_Memory these three modules. It has five stage, including IDLE, READMISS, READMISSOK, WRITEBACK and MISS, to implement cache control. In this way, we can decide when to send signal depending on the current state. Finally, we can successfully and correctly to handle the communication between each interface. During IDLE state, if there is CPU request but no hit on cache, then transmit to MISS state, or it might stay at IDLE state. During, MISS state, if cache is dirty, then set mem_enable, mem_write, write_back signals to 1 and transmit to WRITEBACK state. Otherwise, set mem_enable to 1 and transmit to READMISS state. During READMISS state, if mem_ack flip to 1, set cache_write to 1 and transmit to READMISSOK state. Otherwise, it might stay at remaining state. During READMISSOK state, flush all signals and transmit to IDLE state. During WRITEBACK state, if mem_ack flip to 1, set cache_write to 1 and transmit to READMISS state. Otherwise, it might stay at remaining state.

(q). Dcache_sram.v: dcache_sram module is the place to save cache data. Here, we implement 2-way associative cache. There is 16 cache lines and each cache line has two blocks. Also, 32 bytes per block. Furthermore, the replacement policy is LRU.

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.

(c). When I was implementing dcache controller, the one that spends most of my time was determining the corresponding signal during each state. At first, I didn't understand TA's sample codes well so that it is difficult to decide when to read and write memory. Then, after discussing with classmates and looking for some information, I could successfully claim reasonable signals during each state.

3. Development Environment

(d). OS : Ubuntu 16.04

(e). Compiler : Iverilog