Computer Architecture Project 1 Report

B07902031 黃永雯、B07902043 許喬茵、B07902069 李哲宇

Modules Explanation

PC_MUX.v

PC_MUX module take branch signal as input. If the signal equals to 0, then it will take the output of PC_Adder module as output and connect to PC. On the other hand, if it equals to 1, then it will take the result of Branch_Adder.v as output and connect to PC.

PC_Adder.v

PC_Adder module will add the value of program counter. It reads two data sources as inputs and outputs their sum. The first input value is the current program counter value, and the second input value is 4 (in decimal) (which is assigned in CPU.v).

IF_ID.v

IF_ID module takes current PC, stall signal, flush signal and instruction as inputs. If the flush signal equals to 1, the current PC and instruction will both be set to zero. If the stall signal equals to 0, it will not change the value of current PC and instruction. Otherwise, it will output the current PC and instruction.

Branch_Adder.v

The purpose of this module is to get the target address of a branch instruction. It takes the PC address from IF_ID and the sign-extended immediate as inputs, and then outputs the sum of them.

ID_zero.v

ID_zero module is to check that whether ReadData1 and ReadData2 from the Registers.v module equals. If so, it will output 1, otherwise it will output 0.

ID_branch.v

ID_branch module outputs the result of and operation, whose input is the branch signal from Control.v and the result of ID_zero module.

HazardDetectionUnit.v

HazardDetectionUnit module reads Memory Read signal from EX stage, and RS1, RS2 and RD as inputs. By default, the NoOP signal and stall signal is set to 0, and PC write signal is first set to 1. If a hazard caused by load instruction is detected, i.e., Memory Read Signal is 1 and RD equals to RS1 or RS2, it will set NoOp signal and stall signal to 1 and set PC write signal to 0.
For convenience, it also ouputs NoOp signal to ID_EX module. (This is different from the datapath provided by TAs).

Control.v

Control module reads Opcode as input, and outputs Register Write, Memory to Register, Memory Read, Memory Write, ALU Opcode and ALU Source. It will first use Opcode to check the type of the instruction. Following are the mapping of Opcodes and the types of instruction:

- 0000000 (in binary): NoOp
- 0110011 (in binary): R-type
- 0010011 (in binary): I-type
- 0000011 (in binary): load
- 0100011 (in binary): store
- 1100011 (in binary): beq
  Register Write, Memory to Register, Memory Read, Memory Write, ALU Opcode and ALU Source will be set to their corresponding value.

Sign_Extend.v

Sign_Extend Module takes the whole instruction as input, and combines funct3 and opcode part to decide the instruction. For addi instruction and lw instruction, it takes Instruction[31:20] as immediate, for srai instruction, it takes Instruction[24:20] as immediate, for sw instruction, it takes the combination of Instruction[31:25] and Instruction[11:7] as immediate, and for beq instruction, it takes the combination of Instruction[31], Instruction[7], Instruction[30:25] and Instruction[11:8] as immediate. All of them will then be extended to 32 bits according to the highest bit, and this result is the output of the module.

ID_EX.v

ID_EX module takes control signals, the two data from register file, immediate after sign extension, and parts of instructions as input, and it will assign the value to the corresponding output when the clock is on edge. We also takes NoOp signal as input, and assign all the value to zero if NoOp signal is set.

MUX32_Prev.v

MUX32_Prev module is the MUX before ALU and ALU MUX. It is used twice for the decision of the both input of ALU. It will use the forwarding signal to decide which input to use. If the signal equals to 00, it will take the original value from register file, if the signal equals to 01, it will use the forwarding data from WB stage, and if the signal equals to 10, it will use the forwarding data from MEM stage.

MUX32.v

MUX32 Module is used at two different places. First is to decide the second input value of ALU.v. It reads two data sources and a select value as inputs, and outputs the value for ALU.v. If the value of select equals to 0, then it will take the first source of input (which is the data from the second MUX32_Prev mentioned above) as the output. On the other hand, if the value of select equals to 1, then it will take the second source of input (which is the immediate value) as the output. The other one is at the WB stage. It reads two data sources and a Memory to Register signal as inputs, and outputs the value to write data in Register module. If the signal equals to zero, it will take the data of ALU output, and if the signal equals to one, it will take read data in Data Memory.

ALU.v

ALU Module takes two data sources and ALU Control as inputs, and outputs a zero signal and the value after doing operation on the two input. It uses ALU Control to determine which operation is going to be done. The mapping of ALU Control value and operations can be described as follows:

- 000 (in binary): and
- 001 (in binary): xor
- 011 (in binary): sll
- 010 (in binary): add
- 110 (in binary): sub
- 111 (in binary): mul
- 101 (in binary): addi
- 100 (in binary): srai

ALU_Control.v

ALU_Control Module takes the combination of funct7 and funct3, ALU Opcode as inputs and output ALU Control to tell ALU which operation to do. It will first use ALU Opcode to determine the instruction type and use the combination of funct7 and funct3 to assign each instruction a ALU Control value (the value of ALU Control and the corresponding operation is described in ALU module).

Forwarding_Unit.v

Forwarding_Unit Module takes the two data from EX stage, the destination and register write signal from both MEM stage and WB stage as input, and use the if else statement to decide which stage to forward, then it will output two forwarding signals to tell the two MUX32_Prev which data to use.

EX_MEM.v

EX_MEM module takes control signals (Memory Read and Memory Write), ALU result and write data as inputs, and it will assign the value to the corresponding output when the clock is changing.

MEM_WB.v

MEM_WB module takes control signals (Register Write and Memory to Register), the data from data memory and ALU result as input, and it will assign the value to the corresponding output when the clock is changing.

CPU.v

CPU Module connects modules with wires, which means that some of the output will be the input of other modules.

testbench.v

In addition to what has been written by TAs, we also added the initialization of all registers in the 4 pipeline latches, i.e., we set all of their values to 0.

Members and Teamwork

- B07902031 黃永雯: codes of all modules of EX stage, EX/MEM stage, MEM stage, MEM/WB stage, WB stage, and CPU.v module, report writing, debugging and error fixing.
- B07902043 許喬茵: codes of all modules of IF stage, IF/ID stage, ID stage ID/EX stage and CPU.v module, debugging and error fixing.
- B07902069 李哲宇: codes in CPU.v module, go through all the modules, debugging and error fixing.

Difficulties Encountered and Solutions in This Project

- The first problem we met is that in MEM/WB module, we find out that the input is correct, but the output will be wrong. We first check for the correctness in wire connecting and the logic in module but still get wrong output. We add posedge rst_i in pipeline register modules. If rst_i equals to one, then set all the wires in that module to

zero. We also add the initialization of all registers in the 4 pipeline latches and fix the error.

- The second problem we met is that for store instruction, we will be one stage earlier than it supposed to be. We changed all the assignment in pipeline registers to nonblocking ones and fixed the problem.

Development Environment

The OS we use include MacOS and Windows, and the compiler we use is iverilog.