CA-Project Report

1 Modules Explanation

1.1 Control

Control module reads Op, NoOp as input, and output ALUOp, ALUSrc, RegWrite, MemtoReg, MemRead, MemWrite, and Branch. If NoOp is true, we set all the output of the above to zero as initiation. While NoOp is false, we set the value of output according the value of Op(opcode).

1.2 Adder

Adder module reads data1_i and data2_i as input, and outputs data_o (the sum of both the inputs). This module is used in two sections:

1. **Add_PC**: to calculate the next program counter in general.

   The inputs are the current PC and 4_{10}.

2. **Add_PC_Branch**: to calculate the next program counter if branch.

   The inputs are the current PC and an immediate value which represent the address to go to if branch (that has been shifted left logically by 1).

1.3 MUX32

MUX32 module reads data1, data2, and select bit as input, then output the chosen data. That is, we choose data1 or data2 to become the output according the value of select bit. If select bit is 0, then output the value of data1, if not, then the output is the value of data2. Notice that data1, data2, and output data are all 32 bits.

1.4 MUX2

MUX2 module has four input, data0_i, data1_i, data2_i, forward_i and one output, data_o. The input forward_i decides whether we forward the data.

1. **forward_i = 00**: data_o = data0_i. (no need to forward, read data from register)
2. **forward_i = 01**: data_o = data1_i. (forward the result from MEM/WB)
3. **forward_i = 01**: data_o = data2_i. (forward the result from EX/MEM)

1.5 Imm_Gen

Imm_Gen module reads all bits of instruction as input, and output the immediate part of this instruction. According to different instruction type, we can get the immediate value of the instruction, then extend it to 32 bits as output.

1.6 ALU

ALU module reads data1, data2, ALUCtrl as input, and output the calculation answer as output. First, read the value of ALUCtrl to get the operator of instruction, then we calculate data1 and data2, and output the answer of it.

1.7 ALU_Control

ALU_Control module reads funct, ALUOp as input, and output ALUCtrl. This module read the value of funct and ALUOp, to know what operator it is. Then, it set the ALUCtrl value to the corresponding value.

1.8 Hazard_Detection

Hazard_Detection module reads data1_i, data2_i, data3_i, and MemRead_i as input, and outputs PCWrite_o, Stall_o, and NoOp_o. Hazard happens if when we have to do memory read (MemRead_i is equal to 1) and either the address of RS1 (data1_i) or RS2 (data2_i) in the ID/EX is the same as the address of RD (data3_i) in the EX/MEM stage.

If hazard is detected then sends signal not to update (PCWrite_o = 0), do stall (Stall_o = 1), and give no operation signal (NoOp_o = 1).

Else sends signal to update PC (PCWrite_o = 1), don't stall (Stall_o = 1) and don't give no operation signal (NoOp_o = 0).

1.9 Forward_Unit

Forward_Unit reads MemRegWrite_i, MemRd_i, WBRegWrite_i, WBRd_i, EXRs1_i, EXRs2_i as inputs, and outputs ForwardA_o and ForwardB_o. We determine whether there exists EX-hazard or MEM-hazard by checking the input values. The default value of ForwardA_o and ForwardB_o are two-bit zero(00). If the inputs satisfy EX-hazard's condition, the output will be 10. If the inputs satisfy MEM-hazard's condition, the output will be 01.

1.10 IFID

The module has five inputs, clk_i, Stall_i, Flush_i, PC_i, instr_i, and two outputs PC_o, instr_o. On every clock edge, if the value of Stall_i is 1, which means we have to stall for a cycle, we let PC_o equals to 32-bit zero while instr_o remain unchanged. If there is no need of stalling, PC_o equals to PC_i, while the value of instr_o depends on Flush_i. If Flush_i equals to 1, which means that we have to jump to the branching address. As a result, we let instr_o equals to 32-bit zero and wait for the next cycle. Otherwise, instr_o is equal to instr_i.

1.11 IDEX

IDEX passes down control signals, register data, immediate, and instructions. On every clock edges, we update every output by the corresponding input values.

1.12 EXMEM

EXMEM passes down some of the control signals, ALU result, memory write data address, and rd address. On every clock edges, we update every output by the corresponding input values.

1.13 MEMWB

MEMWB passes down some of the control signals, ALU result, data memory result and rd address. On every clock edges, we update every output by the corresponding input values.

1.14 If_Branch

If_Branch module reads data1_i, data2_i, and Branch_i as input, and outputs data_o. We use compare (a temporary register) to represent the value in data1_i and data2_i are the same (1 if same, 0 if not). Then calculate data_o = compare & Branch_i. This module is used to check either we have to do branch in the next cycle.

1.15 CPU

Connect all the modules.

1.16 testbench

Initialize the value of all registers used in the pipeline latch. We also changed our signal to count flush. We use CPU.If_Branch_data_o to determine if we have to do branch. If we have to do branch, then flush.

2 Members & Teamwork

- 資工三/ B07902009 / 尚沂瑾:
  MUX2, Forward_Unit, pipeline registers
- 資工三/ B07902033 / 陳郁鳳:
  Control, MUX32, Imm_Gen, ALU, ALU_Control
- 資工三/ B07902085 / 張琪:
  Adder, Hazard_Detection, If_Branch
- 共同完成:
  testbench, CPU, debug

3 Difficulties Encountered and Solutions in This Project

- Control signal initialization: At first, there are a lot of x in our register output. We spent quite a lot of time debugging that. In the end, we realised that we forgot to initialize the Control signal, which lead to the register not being able to judge which value to output.
- Stall: In IFID module, we first assign 0 to instr_o if we have to stall. However, we found that the value of instr_o should remain unchanged.

4 Development Environment

- OS: MacOS, Ubuntu 20.04
- Compiler: iverilog