計算機結構 Lab2 Report

許博翔

November 27, 2023

1 Modules Explanation

1.1 `Add` Module

`Add` module reads two signed 32-bits integers as input, and outputs a signed 32-bit integer, which is the sum of the two input integers.

1.2 `ALU_Control` Module

`ALU_Control` module reads the 12..14, 25..31-th bits of the instruction, `ALUOp` as input, and outputs `ALUOp3`, which will be used by `ALU` later. The module uses `ALUOp` to determine if the instruction is addi, srai, noop, beq, or others, and then uses those 10 bits of the instruction to determine what kind of instruction it is. The value of `ALUOp3` is as follow:
`0:and 1:xor 2:sll 3:add 4:sub 5:mul 6:srai`.

1.3 `ALU` Module

`ALU` module reads `a`, `b`, `op` as input, and outputs `c`. The module uses `ALUOp3` to determine which operation should be done to `a`, `b`, and then calculates the result as `c`.

1.4 `Control` Module

`Control` module reads the 0..6-th of the instruction, `noop` as input, and outputs the six control signals: `RegWrite`, `MemtoReg`, `MemRead`, `Write`, `ALUOp`, `ALUSrc`, which are stored in `ctrl` in this order, and the signal `Branch_o`. If the instruction is all 0 or `noop` is true, then it outputs all control signals in 0. Otherwise, it outputs the control signals in what is written in the slide.

1.5 `ForwardingUnit` Module

`ForwardingUnit` module reads `rs1` and `rs2` from `EX` stage, `RegWrite` and `Rd` from `MEM` and `WB` stages as input, and outputs `fa` (`Forwarding A`), `fb` (`Forwarding B`) using the rule in the spec.

1.6 `HazardDetectionUnit` Module

`HazardDetectionUnit` module reads `rs1` and `rs2` from `ID` stage, `rd` and `MemRead` from `EX` stage as input, and outputs `PCWrite`, `Stall_o`, and `noop`. The hazard happens when `MemRead` is set to be true, and the register to be written is one of `rs1`, `rs2`. Therefore, `Stall_o` and `noop` is set to be true when the above happens. Besides, the program counter needs to be changed when the operation is not `noop`, and therefore `PCWrite` is set to be true when `noop` is not true.

1.7 `ImmGen` Module

`ImmGen` module reads the instruction as input, and outputs the immediate storing in the instruction (which is `res`). It determines which instruction it is using `funct3` and `opcode`, and then outputs the correspond (signed extend) value of immediate using the form given in the spec.

1.8 `MUX2` Module

`MUX2` module reads 32-bits a_0, a_1 and 1-bit sel as input, and outputs a_{sel} as b.

1.9 `MUX4` Module

`MUX4` module reads 32-bits a_0, a_1, a_2, a_3 and 2-bits *sel* as input, and outputs a_{sel} as b.

1.10 `IFID` Module

`IFID` module reads clock signals, reset bit, next cycle PC, inst, `flush`, `stall` as input, and outputs the PC of the current cycle. This module changes its internal register `pc`, `inst` at the positive edge of the clock signal. When the reset bit is set, `pc`, `inst` are reset to 0. Besides, if `flush` is set, it means that it must jump to another stage, which should flush its current inst. And if `stall` is set, then its internal register `pc`, `inst` won't be changed at the positive edge of the clock signal.

1.11 `IDEX` Module

`IDEX` module reads clock signals, reset bit, next cycle control signals (stored in `ctrli` in the order same as that in `Control` module), rs1, rs2, immediate, inst as input, and outputs the control signals, rs1, rs2, immediate, inst of the current cycle. Note that on the figure given in the spec, it only deals with `inst[7..11, 15..19, 20..24]`, but in my implementation, I deal with `inst[0..31]`. This module changes its internal register `ctrl`, `r1` (rs1), `r2` (rs2), `imm` (immediate), `inst` at the positive edge of the clock signal. When the reset bit is set, those registers are reset to 0.

1.12 `EXMEM` Module

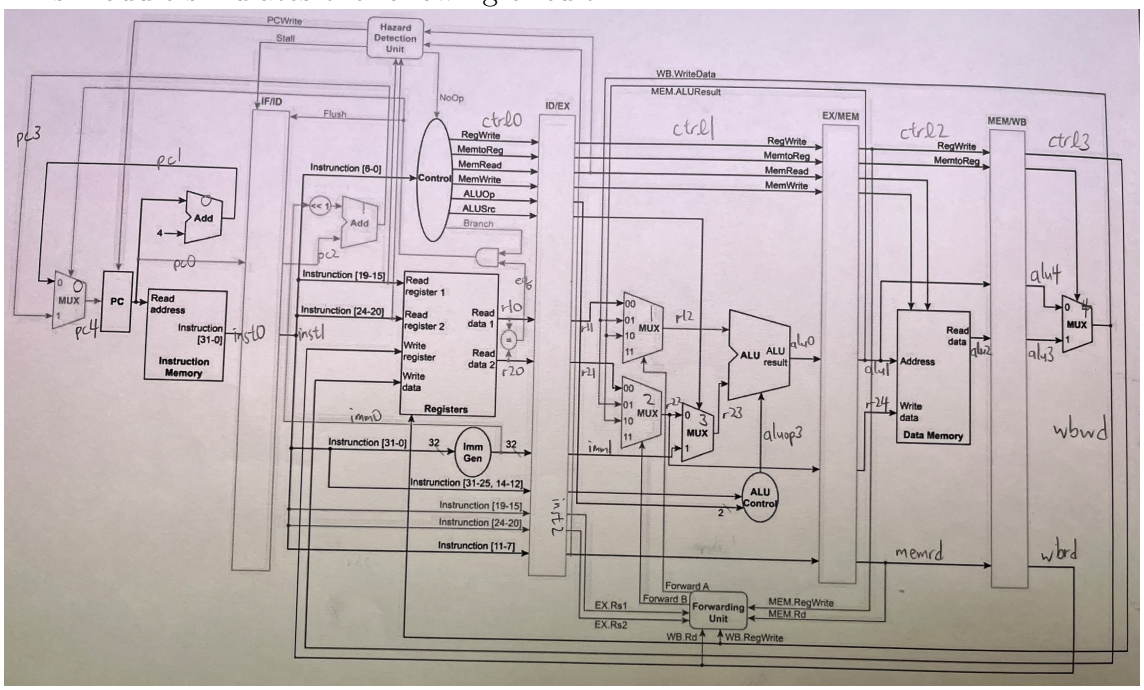
`IDEX` module reads clock signals, reset bit, next cycle control signals (stored in `ctrli` in the order `RegWrite`, `MemtoReg`, `MemRead`, `MemWrite`), ALU result, forward value of rs2, rd as input, and outputs the control signals, ALU result, forward value of rs2, rd of the current cycle. This module changes its internal register `ctrl`, `alu` (ALU result), `r2` (forward value of rs2), `inst` at the positive edge of the clock signal. When the reset bit is set, those registers are reset to 0.

1.13 `MEMWB` Module

`MEMWB` module reads clock signals, reset bit, next cycle control signals (stored in `ctrli` in the order `RegWrite`, `MemtoReg`), ALU result, data read in memory, rd as input, and outputs the control signals, ALU result, data read in memory, rd of the current cycle. This module changes its internal register `ctrl`, `alu1` (ALU result), `alu2` (data read in memory), `rd` at the positive edge of the clock signal. When the reset bit is set, those registers are reset to 0.

1.14 `CPU` Module

This module simulates the following circuit:



where the wires' names are marked above. Note that the wires `ctrl0`, `ctrl1`, `ctrl2`, `ctrl3` are wires of some control signals (see `Control, IDEX, EXMEM, MEMWB` modules described above for more detials), the wires `inst[31..25, 14..12], inst[19..15], inst[24..20], inst[11..7]` at the bottom of the ID stage is replaced by a single wire `inst1`, and the wires `inst[31..25, 14..12], inst[19..15], inst[24..20], inst[11..7]` at the bottom of the EX stage is replaced by a single wire `inst2`.

2 Difficulties Encountered and Solutions

When implementing instruction `beq`, the next instruction after `beq` shouldn't be done if `beq` jumped another branch; however, I found that my implementation did since I didn't deal with `flush` well in `IFID` module.

I used to wonder why `Control` module needs the `noop` signal; after reviewing the powerpoints in class again, I found that's because the instruction being done in the ID stage should become bubble.

I used to implement all control signals in a single wire, but I found that `textbench.v` required the wire `Branch` having the name `Branch_o`. Therefore, I changed a lot of modules' implementations including `Control`, `IDEX`, `CPU` modules.

3 Development Environment

- OS: Ubuntu 22.04.2 LTS
- Compiler: iverilog