112-1 NTU-CA Lab2 Report

`b10902138 陳德維`

---

1. Modules Explanation

- **`Adder.v`**
  - Adder module takes two 32-bit data `data1_i`, `data2_i` as input and output the bit-sum of the 2 datas (i.e. `data_o = data1_i + data2_i`). In this data path, the adder is used for PC, either add 4 to PC or branching.
- **`ALU_Control.v`**
  - ALU Control module takes an instruction segments `funct7`, `funct3` ([31:25]+[14-12]) and a 2-bit select signal `ALUOp` from Control module as input and output a 3-bit data for ALU as select signal to determine what operation should ALU do.
- **`ALU.v`**
  - ALU module takes a select signal input from ALU_Control module, two 32-bit input data from the MUX designed with the Forwarding Unit and output a ALU_result for the result after ALU finished the selected operation.
- **`Control.v`**
  - Control module takes the segment `opcode` of the instruction([6:0]) and a `NoOp` signal to output 7 signal `RegWrite`, `ALUOp`, `ALUSrc`, `MemtoReg`, `MemRead`, `MemWrite`, `Branch_o` based on the type of the instruction for other modules to use. When `NoOp` is 1, 0 will be passed down, else decided by the `opcode`.
- **`MUX32.v`**
  - MUX32 module implement:
    - A 32-bit input 2x1 multiplexer having one 1-bit select signal and two 32-bit data, and output the data which select signal indicated.
    - A 32-bit input 4x1 multiplexer having one 2-bit select signal and four 32-bit data, and output the data which select signal indicated.
- **`Imm_Gen.v`**
  - Sign_Extend module takes the 32-bit instruction from IF/ID register, determine which segment should be extracted based on the `opcode` and extend its most significant bit to a 32-bit immediate.
- **`Forwarding_Unit.v`**
  - Takes 6 inputs `EXE.Rs1`, `EXE.Rs2`, `MEM.RegWrite`, `Mem.Rd`, `WB.RegWrite`, `WB.Rd`, output the result of the forwarding signal `ForwardA`, `ForwardB` using the below rule:

    - if `EXE Hazard` (return `2'b10`):
        - `(MEM.RegWrite && MEM.Rd != 0 && MEM.Rd == EXE.Rs1)`
        - `(MEM.RegWrite && MEM.Rd != 0 && MEM.Rd == EXE.Rs2)`
    - else if `MEM Hazard` (return `2'b01`):
        - `(WB.RegWrite && WB.Rd != 0 && WB.Rd == EXE.Rs1)`
        - `(WB.RegWrite && WB.Rd != 0 && WB.Rd == EXE.Rs2)`
    - else (return `2'b00`)
- **`Hazard_Detection_Unit.v`**
    - Takes 4 inputs `EXE.MemRead`, `EXE.Rd`, `ID.Rs1`, `ID.Rs2`, and output 3 signals `PCWrite`, `Stall`, `NoOp` to stall for handling load-use data hazard by the following rules:
        - if `(EXE.MemRead && (EXE.Rd == ID.Rs1 || EXE.Rd == ID.Rs2))`
            - return 0 for `PCwrite`, 1 for `Stall` and `NoOp`
        - else opposite.
- **`Pipeline Registers`**
    - Including `IF_ID_Register.v`, `ID_EXE_Register.v`, `EXE_MEM_Register.v`, `MEM_WB_Register.v`, which collect each components value for each pipeline stage.
    - They change the register values at the positive edge of the clock signal and reset to 0 at the negative edge of the reset signal being unset.
- **`CPU.v`**
    - CPU module instantiate all the module above and uses wire to connect each module in order to construct the data path given in `Figure5 Final DataPath`.

2. Difficulties Encountered and Solutions in This Lab

- It is hard to connect all components together in `CPU.v`, since if you miss something you may need to reconstruct your current data-path. Also, since many signal passed through different stage, how to name it becomes very important. At first, I used bad naming, and therefore have trouble connecting them, spending a lot of time figuring out which signal is which. Thus, I rename all of them in a more systematical way, which helps a lot.

3. Development Environment

- OS: `Ubuntu 22.04` given by the `Dockerfile`
- Compiler: `iverilog 11.0-1.1`
- Text Editor: `VScode` + `Wavetrace`