HW3 Report

資工四 b08902149 徐晨祐

Development Environment

I finished this homework in **macOS**. I use this command `iverilog -o CPU.out *.v` to compile and use `vvp CPU.out` to get the final result.

Module Implementation Explanation

I implement the following modules in this homework. No `register` type variables were used and only used continuous assignment statement.

1. Control.v

This module is to read the opcode in an instruction, output the corresponding signal to ALU Control module so it can decide what control signal to produce and identify if the iinput instruction is I-type.
First we can see the difference between I-type and R-type instruction is that the opcode of I-type instruction is `0010011` and that of R-type is `0110011`.

- If the opcode is `0010011`, I will set `ALUSrc_o` to be `1` so the MUX module know it should read the input data from Sign-Extend module.
- If the opcode is `0110011`, I will set `ALUSrc_o` to be `0` so the MUX module know it should read the input data from Registers.
- If the opcode is `0010011`, I will set `ALUOp_o` to be `01` so the ALU Control module know the instruction is an I-type instruction.
- If the opcode is `0110011`, I will set `ALUOp_o` to be `00` so the ALU Control module know the instruction is an R-type instruction.

Finally, because the instructions we need to implement in this homework should write the value back to the registers, I set `RegWrite_o` to be `1`.

2. ALU_Control.v

According to the `ALUOp` signal from Control module, ALU Control module can tell which type the instruction is and output the corresponding control signal to ALU module.

- If the instruction is I-type and `funct` is equal to `0100000101`, then the control signal is `SRAI` otherwise it is `ADDI`.
- If the instruction is R-type and
    - `funct` is equal to `0000000111`, then the control signal is `AND`.
    - `funct` is equal to `0000000100`, then the control signal is `XOR`.
    - `funct` is equal to `0000000001`, then the control signal is `SLL`.
    - `funct` is equal to `0000000000`, then the control signal is `ADD`.
    - `funct` is equal to `0100000000`, then the control signal is `SUB`
    - Otherwise the control signal is `MUL`.

I define the control signals to be the following constants. The definition is also followed in the implementaion of ALU module.

```
AND = 000
XOR = 001
SLL = 010
ADD = 011
SUB = 100
MUL = 101
ADDI = 110
SRAI = 111
```

3. Sign_Extend.v

This module will read the immediate field in the instruction and make 20 copies of `data_i[11]` and concatenate `data_i` to assign this 32-bits value to `data_o`.

4. ALU.v

According to the ALU control signal output from ALU Control module, the ALU module can perform the corresponding operation on the two input data.

- If the control signal is `AND`, set the `data_o` to be `data1_i & data2_i`.
- If the control signal is `XOR`, set the `data_o` to be `data1_i ^ data2_i`.
- If the control signal is `SLL`, set the `data_o` to be `data1_i << data2_i`.
- If the control signal is `ADD`, set the `data_o` to be `data1_i + data2_i`.
- If the control signal is `SUB`, set the `data_o` to be `data1_i - data2_i`.
- If the control signal is `MUL`, set the `data_o` to be `data1_i * data2_i`.
- If the control signal is `ADDI`, set the `data_o` to be `data1_i + data2_i`.
- If the control signal is `SRAI`, set the `data_o` to be `data1_i >>> data2_i[4:0]`. (By the definition of this instruction, only the last 5 bits in `data2_i` should be used)

5. Adder.v

This module is to add two 32-bits input values and assign the result to `data_o`.

6. MUX32.v

This module decide set the output to be `data1_i` or `data2_i` according the `ALUsrc` signal from the Control module.

7. CPU.v

I defined more wires in this module, connected all these wires according to the diagram provided in the homework spec as well as the input port and output port of the implemented modules. This part is relatively simple because you only need to follow the diagrams provided in the spec.