# Computer Architecture Lab1 Report

## Modules Explanation

- Adder
  - Add 2 inputs and assign to output.
  - usage
    - deal with program counter
    - deal with branch address
- ALU_Control
  - Take ALUOp, funct3, and funct7 from **ID/EX** as input, and ALUCtrl (3-bit) as output. It refers to ALUop and map funct7+funct3 (10-bit) to ALUCtrl (3-bit) as below.
  - | **funct** | **ALU_ctrl** | **operation** |
    |---|---|---|
    | 0000000111 | 000 | and |
    | 0000000100 | 111 | xor |
    | 0000000001 | 011 | sll |
    | 0000000000 | 010 | add |
    | 0100000000 | 110 | sub |
    | 0000001000 | 100 | mul |
    | 0100000101 | 001 | srai |
    | xxxxxxx000 | 010 | add |
    | xxxxxxx010 | 010 | add |
- ALU
  - Take the outputs of 3-to-1 multiplexer and 2-to-1 multiplexer as inputs, **ALUCtrl** as select, and output the result. It maps ALUCtrl to its job like what I mentioned above in ALU Control module. The result of ALU's computation is assigned to data_o.
- Branch_Unit
  - If rs1 data is equal to rs2 data and current instruction is Branch, then we flush the pipeline registers and pc uses the branch address.
- Control
  - decide the value of wires in the table below by the opcode decoded from **IF/ID.inst**
  - | **instruciton/type** | **ALUOp** | **ALUSrc** | **RegWrite** | **MemtoReg** | **MemRead** | **MemWrite** | **Branch** |
    |---|---|---|---|---|---|---|---|
    | R-type | 10 | 0 | 1 | 0 | 0 | 0 | 0 |
    | I-type(except lw) | 11 | 1 | 1 | 0 | 0 | 0 | 0 |
    | lw | 11 | 1 | 1 | 1 | 1 | 0 | 0 |

| instruciton/type | ALUOp | ALUSrc | RegWrite | MemtoReg | MemRead | MemWrite | Branch |
|---|---|---|---|---|---|---|---|
| sw | 11 | 1 | 0 | 0 | 0 | 1 | 0 |
| beq | 11 | 0 | 0 | 0 | 0 | 0 | 1 |
| flush or default | 00 | 0 | 0 | 0 | 0 | 0 | 0 |

- Sign_Extend
  - Take the **immediate** field of the instruction as input, data_o as output. It copys the msb (sign bit) to left 20 bits, and then assign to data_o.
- CPU
  - Declare wires for each units and pipeline Registers. Decode the fields as variables like funct3, funct7, rs1, rs2, etc no matter the type of the instruction. Also, we need pc and pc_next to fetch the instruciotn from memory. After declaring these variables, we put them to the modules above if needed.
- Forwarding_Unit
  - Detect EX Hazard and MEM Hazard by checking the hazard condition listing on the spec.
- Hazard_Detection
  - Detect control Hazard by checking the hazard condition listing on the lecture slide.
- IF_ID
  - 2 registers to store the following values from IF stage.
    - Instruction;
    - PC;
  - If stall is 1, then we don't modify PC and instruction.
  - If flush is 1, then we modify PC and instruction as 0.
- ID_EX
  - 13 registers to store the following values from ID stage.
    - RegWrite;
    - MemtoReg;
    - MemRead;
    - MemWrite;
    - ALUOp;
    - ALUSrc;
    - rs1_data;
    - rs2_data;
    - immediate_32;
    - rs1;
    - rs2;
    - rd;
    - funct (funct7 + funct3);
- EX_MEM
  - 7 registers to store the following values from EX stage.

        - RegWrite;
        - MemtoReg;
        - MemRead;
        - MemWrite;
        - ALU_result;
        - MUX_B_out;
        - rd;

- MEM_WB
    - 5 registers to store the following values from MEM stage.
        - RegWrite;
        - MemtoReg;
        - read_data;
        - ALU_result;
        - rd;

- MUX32
    - 2-to-1 multiplexer
        - Mux write back
            - select between read data and ALU result.
            - selected by MemtoReg.
        - Mux PC
            - select between PC+4 and Branch address.
            - selected by the result of Branch Unit.
        - Mux ALU source
            - select between 4-to-1 mux b's output and immediate.
            - selected by ALUSrc.
    - 4-to-1 multiplexer
        - select among WB.write, MEM.ALUresult, and rs1(or rs2) data.
        - selected by forwarding unit.
        - only 3 inputs.

## Difficulties Encountered and Solutions in This Lab

- At the beginning, it was hard to decide the whole architecture including the declaring of modules, and I had to modify the files from previous homework as needing to support new instructions and pipeline. Secondly, I was not able to understand how to declare pipeline registers, at the meanwhile initialize them in the init block in testbench. The worst case is I forget to provide a default value in Control for the beginning conidition that all values are empty or zero. Therefore, the wires are messed up without assigned values. Fortunately, my roommate knows verilog very well. I discussed with him and finally solved these difficulties.

## Development Environment

- Os: MacOs 12.4
- compiler: iverlog
- IDE: vscode