CA Lab1 Report

Modules Explanation

> The following modules are sorted alphabetically, except for CPU.

Adder_32

This module simply assigns the sum of two input `a`, `b` to output `c`, which is rather trivial given that the addition of vector is built-in in Verilog.

ALU_Control

Inputs & outputs:

```
input [1:0] ALUOp;
input [6:0] funct7;
input [2:0] funct3;

output reg [2:0] ALUctl;
```

The `ALUOp` indicates whether the output `ALUctl` depends on the instruction. More specificly, in my implementation, I set `ALUOp = 11` when the operation type is decided by both `funct3` and `funct7` (R-type), while in the I-type case I set `ALUOp = 10`, meaning that the operation is decided only by `funct3`.

The output `ALUctl` represents the operation type of current instruction, as the following:

ALUctl	Operation	ALUOp	funct3	funct7
0	and	11	111	0
1	xor	11	100	0
2	sll	11	001	0
3	add	11	000	0
3	add	10	000	N.A.
4	sub	11	000	32
5	mul	11	000	1
6	sra	10	101	N.A.

I implemented this logic by several `case` syntax. I belive there should be some technique to transform this into some logic gates, but I have no background in logic design. 😳

ALU

Inputs & outputs:

```
input [2:0] ctl;
input signed [31:0]op1, op2;
output reg [31:0] res;
```

In this module we calculate the result by executing corresponding operation of current instruction. All the operations are built-in in Verilog. Note that we have to label `op1, op2` as `signed` to get the correct results.

Furthermore, since we get a 32-bit `op2` even when that operand is actually a 5-bits immediate (that is, the `srai` case), we take only `op2[4:0]` into account when executing **shift right** operation.

Control

Inputs & outputs:

```
input [6:0] opcode;
output reg [1:0] ALUOp;
output reg ALUSrc;
output reg RegWrite;
```

The outputs will be set according to `opcode`:

Opcode	ALUOp	ALUSrc	RegWrite
0110011 (R-type)	11	0	1
0010011 (I-type)	10	1	1

I also implement this part with `case`, for readability.

MUX32

Inputs & outputs:

```
input swt;
input [31:0] in0, in1;
output reg [31:0] res;
```

The logic in this module is very easy: `res = in0` when `swt == 0` and `res = in1` otherwise.

Sign_Extend

Take the highest bit of input. Duplicate it by 20 times. Then combine this extension and original input into the output. Done.

CPU

Finally, we are going to connect all the other modules in this main module. Basically we create a wire for every signal, and connect them as in the figure below.

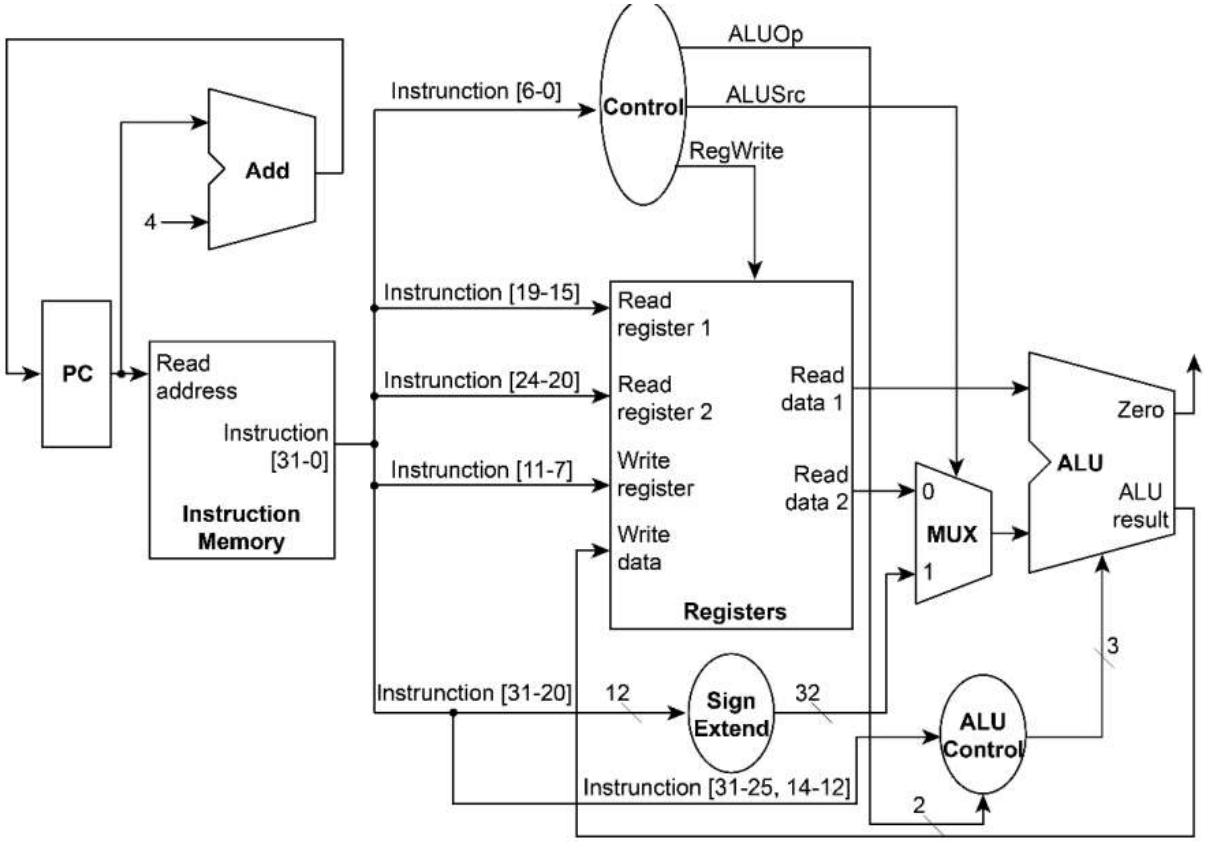

Figure 1 Data path of the CPU in this homework

The only tricky part here is that we always want to write the registers **after** the `ALU result` has been updated. (I am not quite sure if this is always necessary, but I do encounter error caused by this problem.) Thus there is a slight delay between the clock and actual instruction update, as the code here.

```
always @(posedge clk_i or negedge rst_i)
  begin
    #2
     instr_addr = pc_o;
  end
```

The `instr_addr` is the input of `Instruction_memory` . Since the `Registers` module will write the register exacly when `clk` updates, the data and address written will be of the **last** instruction, as we desired.

Development Environment

Using `iverilog` in **Win 11**.