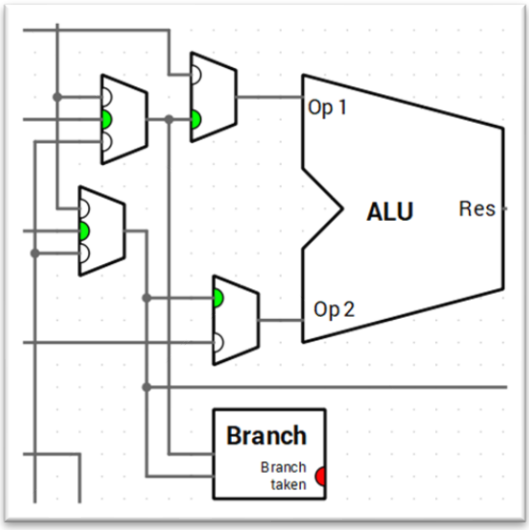
COSE222: Computer Architecture
Design Lab #5

Due: Dec 16, 2024 (Monday) 11:59pm on Blackboard

Total score: 45

In this lab you are requested to complete the 5-stage pipelined processor design. The basic idea of pipelining is simple, so we can split the processes for executing instructions into 5 stages: instruction fetch (IF), instruction decoding (ID), execution (EX), memory accessing (MEM), and write-back (WB). The data and the control signals required for executing instructions are propagated through pipeline registers between stages. The 5-stage pipeline processor manipulates the control signals for data forwarding and hazard detection also. As we discussed in the classes, data forwarding is an essential part of the pipelined processor to guarantee the performance benefits from pipelining. Without the data forwarding the pipelined processor will encounter more frequent pipeline stalls, which will degrade the performance of the pipelined processor significantly. In addition the pipelined processor will generate wrong results if the hazard detection unit is designed incorrectly. Thus you must work carefully for implementing such hardware units in the pipelined processor design. (Remember the idiom "THE DEVIL IS IN THE DETAIL".)

In the previous design lab (i.e. Design Lab #5), you were requested to design the single-cycle processor that supports the following instructions: `lw, sw, add, addi, sub, xor, xori, or, ori, and, andi, beq, bne, blt, bge`. The 5-stage pipelined processor in this lab also support the same instructions, which means your processor needs to support additional instructions along with the seven instructions supported by the processor introduced in the textbook. For this purpose, we assume the pipelined processor of this lab includes the *dedicated branch unit* (BU) as shown in the following figure. Note that ALU does not have "zero" and "sign" output ports. Actually the 5-stage pipelined processor of this lab is similar to the pipelined processor design exhibited by Ripes RISC-V simulator. (Note that the following figure is also captured from the 5-stage pipelined design in Ripes.)



As the branch unit is located in the *EX stage*, we can say a branch instruction is determined in the EX stage and the target of the branch is applied when the branch instruction is in the MEM stage. Therefore, if the result of a branch instruction is taken (resolved in the EX stage), IF/ID and ID/EX pipeline registers need to be flushed and two cycles are wasted.

1. Pipeline register design

Pipeline registers sample the input signals and update the outputs at the rising edge of a clock signal. Namely the pipeline registers are just a group of D flop-flops working with the system clock and the several control signals. In this lab you are requested to design the pipelined registers using *SystemVerilog's packed structure*. SystemVerilog supports a structure (`struct`) like C/C++. A structure can contain multiple elements of different data types. However, we don't need such C-like structures in RTL designs since most of signals and signal vectors have a logic type only. Note that pipeline register just propagate multiple data and control signals at the edge of the clock. Thus we will use "packed structure" to implement the pipeline registers. Using a packed structure, a vector signal can be divided into multiple sub-signals or sub-vectors which can be referenced as members of the structure. See the following example.

```
typedef struct packed {
    logic  [31:0]  pc;
    logic  [31:0]  inst;
} pipe_if_id;
……
pipe_if_id  id;        // IF/ID pipeline register
……
```

In this example we define the packed structure "`pipe_if_id`" that includes two member variables `pc[31:0]` and `inst[31:0]`. Note that the defined packed structure includes logic vectors. The variable called `id`, which is the IF/ID pipeline register, is declared using this packed structure. That means the variable `id` is a logic vector that includes 64 logic type variables (i.e. [63:0]). We can easily reference a part of this long logic vector using member variables such as `id[63:32]` = `id.pc` and `id[31:0]` = `id.inst`. As pipeline registers propagate lots of signals to the next stage, we can pack the required signals using a packed structure. The signals packed for the pipeline registers can be easily referenced using member variables. The following code exhibits the example of IF/ID pipeline register implementation using a packed structure in SystemVerilog. If you want to study more about the packed structure of SystemVerilog, please read this article:
https://www.chipverify.com/systemverilog/systemverilog-structure#packed-structures

```
always_ff @ (posedge clk or negedge reset_b) begin
    if (~reset_b) begin
        id <= 'b0;
    end else begin
        if (if_flush) begin
            id <= 'b0;
        end else if (~if_stall) begin
            id.pc <= pc_curr;
            id.inst <= inst;
        end
    end
end
```

In the provided incomplete pipelined processor design, the packed structures for pipeline registers are defined at the head of the design. Please implement the pipeline registers using these structures.

2. Internal forwarding in the register file

As described in the textbook, the register file in the pipelined processor supports internal forwarding from write data input to source register outputs. This internal forwarding in the register file is already designed in the provided register file design. Please see the following code snippet and try to understand how the internal forwarding is implemented in the existing register file design.

```
// Register file reads with supporting internal forwarding
assign rs1_dout = (reg_write & (rs1==rd)) ? rd_din: ((|rs1) ? rf_data[rs1]: 'b0);
assign rs2_Dout = (reg_write & (rs2==rd)) ? rd_din: ((|rs2) ? rf_data[rs2]: 'b0);
```

3. Forwarding unit design

In order to support data forwarding in the pipelined processor, the processor deploy the forwarding MUXes and the forwarding unit (FU). The forwarding MUXes allow the previously generated results are delivered to the inputs of ALU. The forward MUXes select the operands of ALU from ① the register file via ID/EX pipeline register, ② the destination register data from EX/MEM pipeline register, and ③ the destination register data from MEM/WB pipeline register. The forwarding unit generates the selection signals for the forwarding MUXes. You can implement the forwarding MUXes and the forwarding unit as described in the textbook. Note that ALU also receives the immediate values via ID/EX pipeline register, thus you should carefully arrange the tandem of the forwarding MUX and the MUX that selects the second ALU operand.

4. Hazard detection unit

The hardware-controlled hazard detection unit (HDU) allows the pipelined processor to detect the hazard conditions and control pipelining dynamically. Without the hardware-controlled hazard detection unit, a compiler inserts NOP (no operation) in codes if required in order to avoid incorrect operations in the pipelined processors. For supporting such compiler-based technique, the compiler needs to know the detailed anatomy of the pipelined processor.

The hazard detection unit is located in the ID stage. HDU first detects the hazard conditions and then generates pipeline control signals. The hazard detection unit of this 5-stage pipelined processor needs to detect the load-use hazard case we learned in the classes. Namely if the former instruction is a load instruction and the very next instruction consumes the result of the former load, the following instructions needs to be stalled in the ID and IF stages.

5. Instructions to be tested

The following assembly code is preloaded in the instruction memory for testing the pipelined processor design. The assembly code is designed to test the important control units such as the forwarding unit and the hazard detection unit in the pipelined processor.

Assembly code	Binary data in imem
`start: lw x4, 0(x0)`	`00000000000000000010001000000011`
`lw x5, 8(x0)`	`00000000100000000010001010000011`
`lw x6, 16(x0)`	`00000001000000000010001100000011`
`addi x7, x5, 6`	`00000000011000101000001110010011`
`T1: add x7, x5, x4`	`00000000010000101000001110110011`

	sub x8, x5, x4	01000000010000101000010000110011
	and x9, x5, x4	00000000010000101111010010110011
	or x10, x5, x4	00000000010000101110010100110011
	xor x11, x5, x4	00000000010000101100010110110011
	addi x27, x4, 1	00000000000100100000110110010011
	addi x28, x4, -7	11111111100100100000111000010011
	andi x29, x4, 15	00000000111100100111111010010011
	ori x30, x4, 16	00000001000000100110111100010011
	xori x31, x4, 15	00000000111100100100111110010011
	beq x7, x8, T1	11111100100000111000110011100011
	sw x7, 24(x0)	00000000011100000010110000100011
	lw x11, 24(x0)	00000001100000000010010110000011
	bne x7, x11, T1	11111100101100111001011011100011
T2:	lw x4, 32(x0)	00000010000000000010001000000011
	lw x5, 40(x0)	00000010100000000010001010000011
	bge x4, x5, T2	11111110010100100101110011100011
	add x4, x4, x5	00000000010100100000001000110011
	blt x5, x4, T1	11111010010000101100110011100011
	sub x4, x4, x5	01000000010100100000001000110011

You can find the more detailed instructions in the incomplete top design file (pipeline_cpu.sv) and the testbench file (tb_pipeline_cpu.sv). Please complete the design and the testbench file. Verify your design using Verilator.

6. What to do

(a) Complete the top design of the 5-stage pipelined processor (pipeline_cpu.sv).

(b) You should use "tb_pipeline_cpu.cpp" as the testbench of the pipelined processor. See the below commands for compiling and verifying the pipelined processor using Verilator. If you execute the generated binary, you will find the generated "report.txt" file. You can see the generated waveforms by reading "wave.vcd" using gtkwave.

```
$ verilator -Wall --trace --cc pipeline_cpu.sv imem.sv regfile.sv alu.sv
dmem.sv --exe tb_pipeline_cpu.cpp
$ make -C obj_dir -f Vpipeline_cpu.mk Vpipeline_cpu
$ ./obj_dir/Vpipeline_cpu
```

(c) Compress all the design files (imem.sv, regfile.sv, alu.sv, dmem.sv, and pipeline_cpu.sv), the testbench (tb_pipeline_cpu.cpp), and the report file (report.txt) in **one zip file**. You must name your zip file as "`FirstName_LastName.zip`". (e.g. `Gildong_Hong.zip` for Gildong Hong) If your submission file does not meet this rule, we will reduce 1 point from your score. 😟