COSE222: Computer Architecture
Design Lab #4

Due: Jun 14, 2020 (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, instruction decoding, execution, memory accessing, and write-back. 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 during the classes, data forwarding is an essential part of the pipelined processor to guarantee the performance benefits by pipelining. Without the data forwarding the pipelined processor will encounter more frequent pipeline stalls, which will degrade the performance of the pipelined processor. 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".)

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  [63:0]  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[63:0]` 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 96 logic type variables (i.e. [95:0]). We can easily reference a part of this long logic vector using member variables such as `id[95:32]` = `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 on 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. Additionally we cannot avoid the pipeline stalls by branch instructions even if the branches are not taken.

The hazard detection unit is located in the ID stage. HDU first detects the hazard conditions and then generates pipeline control signals. Note that we learned two kinds of hazard conditions: ① the data hazards caused by load instructions and ② control hazards by taken branches. For the former case, HDU compares the destination register of the preceding load instruction and the following instruction that needs the result of the load instructions. In addition to this data hazard condition, our HDU needs to detect the data hazards between the preceding instructions and the following branch instruction (`beq`). Note this data hazard condition is not described in the textbook. As the pipelined processor employs the dedicated comparison unit that directly compares two operands from the register file outputs in ID stage, branch instructions cannot benefit from the data forwarding. Note that the forwarding unit and the forwarding MUXex exist in EX stage. Hence, if the branch instruction requires the operand data generated from the preceding instructions, the data can be only forwarded internally in the register file and no other forward paths are not available. Consequently our HDU needs to detect this condition: *the branch instructions reads the register file data that is not yet written to the register file*. If the data hazard is detected, HDU stalls IF and ID stages of the pipelined processor. For the control hazard cases resulted from taken branches, HDU flushes IF/ID pipeline register and fetch the new instruction pointed by the branch target again.

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`
`add x7, x5, x4`	`00000000010000101000001110110011`
`sw x7, 24(x0)`	`00000000011100000010110000100011`
`sub x8, x5, x4`	`01000000010000101000010000110011`
`lw x6, 8(x8)`	`00000000100001000010001100000011`
`sw x8, 32(x0)`	`00000010100000000010000000100011`
`and x9, x5, x4`	`00000000010000101111010010110011`
`sw x9, 40(x0)`	`00000010100100000010010000100011`
`or x10, x5, x4`	`00000000010000101110010100110011`
`sw x10, 48(x0)`	`00000010101000000010100000100011`
`add x11, x4, x6`	`00000000011000100000010110110011`
`add x11, x11, x11`	`00000000101101011000010110110011`
`add x12, x11, x11`	`00000000101101011000011000110011`
`add x11, x4, x6`	`00000000011000100000010110110011`
`beq x6, x8, T2`	`00000000100000110000101001100011`
`T1: add x4, x4, x4`	`00000000010000100000001000110011`
`sub x5, x4, x4`	`01000000010000100000001010110011`
`or x13, x11, x0`	`00000000000001011110011010110011`
`beq x0, x5, T3`	`00000000000000101000100001100011`
`T2: lw x9, 8(x8)`	`00000000100001000010010010000011`
`lw x10, 16(x0)`	`00000001000000000010010100000011`
`beq x9, x10, T1`	`11111110101001001000010011100011`
`T3:`	

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 ModelSim.

6. What to do

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

(b) Verify your design with the testbench. The incomplete testbench file is provided. Set the clock frequency as 100 MHz and instantiate the top design file as "dut" (design under test). (Please find the provided testbench "**tb_pipeline_cpu.sv**").

(c) Create the ModelSim project "Lab4" in your workspace folder. Copy all design files and *.mem files to the project folder. The mem files include the initial memory data for the instruction memory, the data memory, and the register file. Verify your design using ModelSim. Run the testbench for 500 ns, then you will find the generated "report.txt" file, which will print the contents of the register file and the data memory.

(d) Compress all the design files (imem.sv, regfile.sv, alu.sv, dmem.sv, and pipeline_cpu.sv), the testbench (tb_pipeline_cpu.sv), 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. ☹