Vijay Tripathi

Lab 3 Questions

1.

This three-stage RISC-V CPU utilizes different components manifested as different SystemVerilog modules. These components include an instruction decoder, a control unit, a register file, an ALU, a hex-display driver, and various registers and MUXs within the CPU module. The instruction decoder takes in a 32-bit wide RISC-V instruction, which comes from the instruction memory initialized with readmemh in an initial begin in the CPU module. The instruction decoder outputs various instruction fields, including the instruction type (determined by opcode), funct7, registers, immediate 12 and 20, and funct3. These fields are passed to the control unit, which determines the ALU operation, an ALU source MUX select, a register MUX select, a register write-enable, and a GPIO write-enable. Different MUXs for the design include:

- Register-source 2 uses alusrc_EX as a selector for a sign-extended imm12 field (alusrc == 1) or readdata2 (alusrc == 0).
- Register file's writedata uses regsel_WB for selecting between GPIO_in, imm20, or the ALU's result
  - regsel_WB is the result of a register with regsel_EX as an input to allow for a 1-cycle delay (from execute to write-back stage)

D flip-flops/registers are used to delay values for 1 cycle for proper implementation of 3-stages. These include:

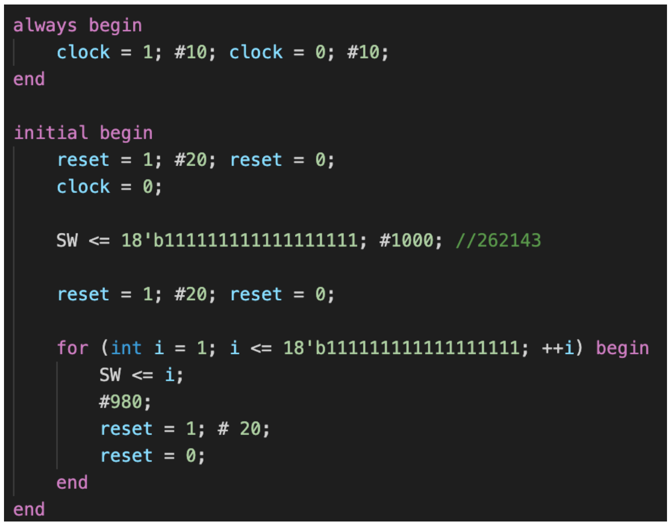
- readdata1_EX becomes gpio_out with high activation of the GPIO write-enable
- Register file's destination register
- Register file's write-enable
- All inputs for Register-Select MUX
  - gpio_in
  - imm20
  - ALU's result
  - selector (regsel)

The CPU module accepts a clock, reset, and GPIO_in (switch values) as input and outputs GPIO_out, which is used by the hex-display driver to show values on the hex-displays. The CPU instantiates an instruction decoder, control unit, register file, and ALU. Both MUXs are also in the CPU, inside an always_comb block. All registers are inside individual always_ff, positive-clock edge activated blocks.

The testbench is a simple end-to-end tester that compares the values of the switches to the resulting GPIO_out/hex-displays. Two test-cases were chosen per switch; 38 cases in total. Each case has the most-significant switch turned on, with a random combination of proceeding switches on. The testbench is contained in the simtop module. There is another testbench that compares regfile's mem[] values to the expected register value from RARS's register value table at specified program counter values. This is equivalent to "Run one step at a time" in RARS and comparing the register values table at the corresponding clock cycle. This testbench is manifested as a combinational always block in simtop.

2.

The aforementioned test benches are in always_comb blocks in simtop. Switch inputs are regulated in an initial begin as follows.



```
always begin
    clock = 1; #10; clock = 0; #10;
end

initial begin
    reset = 1; #20; reset = 0;
    clock = 0;

    SW <= 18'b111111111111111111; #1000; //262143

    reset = 1; #20; reset = 0;

    for (int i = 1; i <= 18'b111111111111111111; ++i) begin
        SW <= i;
        #980;
        reset = 1; # 20;
        reset = 0;
    end
end
```

Errors thrown in the always_comb blocks use $error and specify the switch input value, expected HEX display code, and actual HEX display code. The second test suite works, but relies on the correctness of the RARS register value table. The first test suite also works, but relies on the correctness of the expected HEX-display values.

3.

The second test suite would flag a sra-srl aluop swap error. Using test1.asm, register 14's expected value would be 0x00004400 and 15's is 0xffffc400. Running test1.asm's machine code through a CPU with inverted sra-srl alu-opcodes would result in swapped values in registers 14 and 15, triggering an error.

4.

Error debugging in this lab has followed the following algorithm.

1. Convert hexadecimal machine code into binary and manually decode instruction's fields, starting with opcode and moving up to most significant bit. Comparing with QuestaSim's instruction decoder module wave, this checks for errors in the instruction decoder.
2. Derive the outputs of the control unit by hand, referencing the block design and control unit spreadsheet. Comparing against QuestaSim's control unit module wave will reveal errors in the control unit.
3. Repeat Step 2 for the ALU, register file, and CPU MUXs and registers.

Modifying the test bench for error checking requires knowing which module the error is in. LUT-style testbenches can be written for all modules, since all expected values are already known.