IB Computer Design - Supervision 3

Nandor Licker <nl364@cl.cam.ac.uk>

Due noon before the day of supervision

You are given the code for a simple RISC processor, executing instructions in sequential stages. Your task is to figure out what the processor does in its current form and upgrade it to a more efficient pipelined implementation, running the same program in a fewer number of clock cycles. You should attempt to make as much progress as possible by the 3rd supervision, however the whole implementation is due by the 4th.

```
module CPU();
  reg clk, rst;

  reg[15:0] rom[0:255];
  reg[15:0] r[15:0];
  reg[15:0] pc;
  reg[1:0]  state;

  wire[15:0] IF_opc;
  reg[11:0]  ID_opc;

  reg[15:0]  ID_pc, EX_pc, WB_pc;
  reg[3:0]   ID_op, EX_op, WB_op;
  reg[3:0]   EX_dst, WB_dst;
  reg[15:0]  EX_opA, EX_opB, EX_opC, EX_imm, WB_val;
  reg        WB_wr;

  assign IF_opc = rom[pc];

  always @(posedge clk or posedge rst)
    if (rst) begin
      pc <= 0; state <= 0;
    end else begin
      $display("%d %d %d %d", r[0], r[1], r[2], r[3]);
      case (state)
        4'b00: begin
          ID_pc <= pc + 1;
          ID_opc <= IF_opc[11:0];
          ID_op <= IF_opc[15:12];
          state <= 1;
        end
        4'b01: begin
          EX_pc  <= ID_pc;
          EX_opA <= r[ID_opc[3:0]];
          EX_opB <= r[ID_opc[7:4]];
          EX_opC  <= r[ID_opc[11:8]];
          EX_imm <= { {8{ID_opc[7]}}, ID_opc[7:0] };
```

```
        EX_dst <= ID_opc[11:8];
        EX_op  <= ID_op;
        state  <= 2;
      end
      4'b10: begin
        case (EX_op)
          4'b0000: begin WB_wr <= 1; WB_pc <= EX_pc; WB_val <= EX_opB + EX_opA; end
          4'b0001: begin WB_wr <= 1; WB_pc <= EX_pc; WB_val <= EX_opB - EX_opA; end
          4'b0010: begin WB_wr <= 0; WB_pc <= EX_opC == 0 ? EX_pc + EX_imm : EX_pc; end
          4'b0011: begin WB_wr <= 0; WB_pc <= EX_opC != 0 ? EX_pc + EX_imm : EX_pc; end
          4'b0100: begin WB_wr <= 1; WB_pc <= EX_pc; WB_val <= EX_imm; end
          default: begin WB_wr <= 0; end
        endcase;
        WB_dst <= EX_dst;
        WB_op <= EX_op;
        state <= 3;
      end
      4'b11: begin
        pc <= WB_pc;
        if (WB_wr) r[WB_dst] <= WB_val;
        if (WB_op == 4'b0101) $finish;
        state <= 0;
      end
    endcase
  end

  initial begin
    rom[ 0] = { 4'h4100 };
    rom[ 1] = { 4'h4201 };
    rom[ 2] = { 4'h2007 };
    rom[ 3] = { 4'h0312 };
    rom[ 4] = { 4'h1131 };
    rom[ 5] = { 4'h4200 };
    rom[ 6] = { 4'h0232 };
    rom[ 7] = { 4'h4301 };
    rom[ 8] = { 4'h1003 };
    rom[ 9] = { 4'h30F9 };
    rom[10] = { 4'h5000 };

    r[4'h0] = 5; r[4'h1] = 0; r[4'h2] = 0; r[4'h3] = 0;
    r[4'h4] = 0; r[4'h5] = 0; r[4'h6] = 0; r[4'h7] = 0;
    r[4'h8] = 0; r[4'h9] = 0; r[4'hA] = 0; r[4'hB] = 0;
    r[4'hC] = 0; r[4'hD] = 0; r[4'hE] = 0; r[4'hF] = 0;

    clk = 0; rst = 0; #1 rst = 1; #1 rst = 0;
  end

  always #1 clk = !clk;
endmodule
```

1. The following past papers: *2013 Paper 5 Q2*, *2012 Paper 5 Q2*, and *2014 Paper 5 Q2*
2. Describe all the instructions supported by the machine, indicating their opcodes and encodings.
3. Disassemble the program encoded in the initial block of the testbench into human-readable assembly code. Explain what it does. In how many cycles does the program execute on the multi-cycle machine?
4. Create a diagram indicating how the same program would run on a pipelined processor with no branch delay slots and no forwarding, clearly indicating where stalls occur. Assume `r0` = 1. On the vertical axis, show sequence of instructions executed. Horizontally, indicate the stage each instruction is in a given cycle and show where stalls occur.
5. On a similar diagram, show how the same sequence of instructions would be executed with forwarding. Indicate with arrows which registers are forwarded to the stages that read them.
6. How many stalls does a branch instruction introduce? How could you improve this?
7. In the previous processor, all flip-flops are were set on the rising edge of the clock cycle. When do we need to use the negative edge in the pipelined processor? Consider the following program fragment:

   ```
   imm r0, #1
   add r1, r0, #1
   add r2, r0, #2
   add r3, r0, #3
   ```

   On a timing diagram with a clock pulse, show accesses to the register file.
8. Provide a pipelined implementation in Verilog or SystemVerilog. First, you need to remove the state variable an allow all blocks to run in parallel. Next, you need to stall execution until the target of branches is known. Forwarding paths are also required, from the `EX/WB` registers to the inputs of the ALU. Lastly, you might want to bring branch target calculation forward in time, reducing the number of stalls introduced by branches. The following hints might be helpful:

   (a) You might need to keep track which instruction is running in individual stages and register indices might also need to be stored, besides the values read from the register file. What additional information do you need to keep in the pipeline registers?

   (b) Where is the program counter updated? Try moving the logic out of any stage it might be in now. Think of an updated conditioned by the instruction running in the last stage.

   (c) Think carefully about stalls. In which stage are they introduced? How and when are they propagated? You should add a flag to turn each stage into a no-op in case of a stall. When a stage does not perform an operation, the next stage in the next cycle should be a no-op as well, propagating the flag. Remember that we do not stall instructions mid-flight, we only delay fetching instructions until we are certain they can be executed without conflict.

   ```
   always @(posedge clk)
     if (!rst) begin
       if (EX_bubble) begin
         ...
       end else begin
         ...
       end
     end
   ```

   (d) What needs to be forwarded from where? Instead of using `EX_opA` and `EX_opB` in the `EX` stage, introduce wires for `ALU_opA`, `ALU_opB`, defining combinatorial logic to either read from the `EX` registers or some other registers, using additional information passed on from earlier stages.

   (e) What operation needs to run on the negative edge of the clock cycle? Do you move the whole stage or only a certain portion of it?