Introduction to Digital Logic Design Lab EECS 31L

Lab #4: Pipelining

Christiano Gravina

#83469296

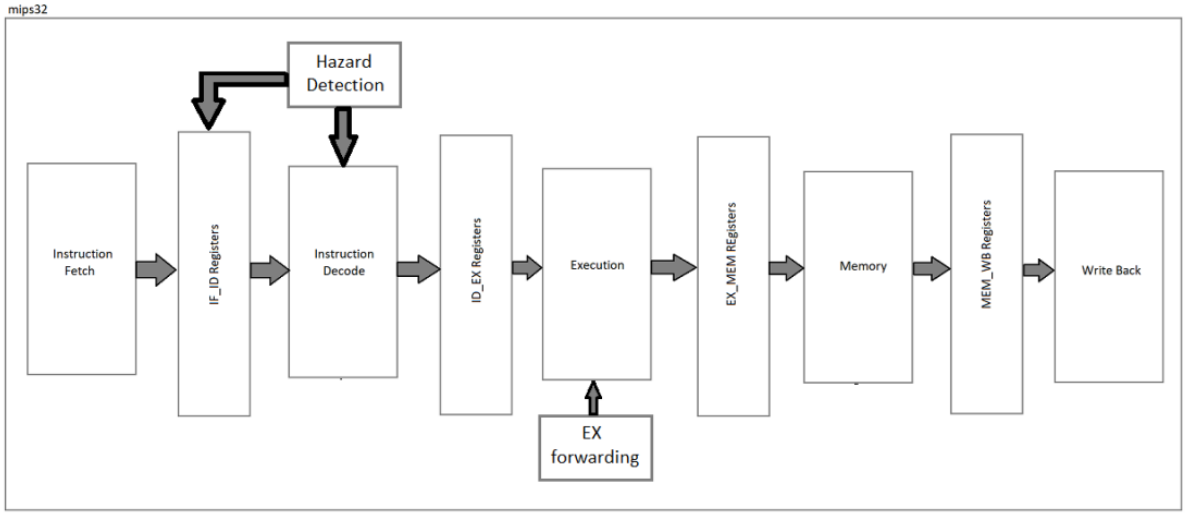
11/23/2022

---

1 Objective

The objective of this lab was inserting pipelining registers into our processor design. Pipelining means our processor is always busy, or at least as much as possible, in order to increase performance. To do this, we inserted registers to transfer data between each stage of data processing. Each pipeline register is a sequence of latches that will keep the data through clock cycles so we can access them afterwards

Figure 1: Pipeline processor



2 Procedure

Programming this in Verilog was quite easy, the problem was debugging. I say that programming was easy because the concept is simple: instantiate the components and wire them together. So, I started by the "hardest" part, which was outlining the processor in the mips32.v file by doing exactly that: instantiating every pipeline stage then declaring or connecting every wire that came in or out of them.

For example, in the Instruction Fetch stage, I declared all the wires you can see outside of the main square. Later, I connected the outputs into the IF_ID Registers and the inputs from where they come from (because they come from different places). Then, the last step was to connect all outputs from the IF_ID registers to the Instruction Decoding pipeline stage.
As you can see in the images below, "pc_plus4" and "instr"go into IF_ID registers and come out as IF_ID_pc_plus4 and IF_ID_instr respectively. Then, these latter two are connected to the Instruction Decode stage.

Figure 2: Instruction Fetch.

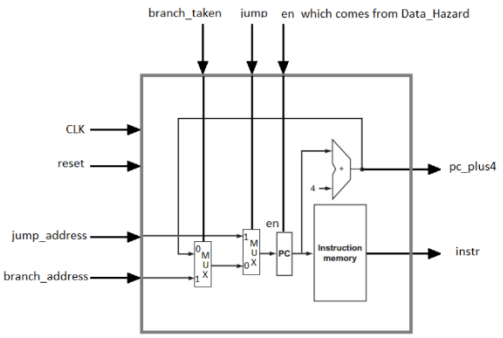


Figure 3: IF/ID Registers.

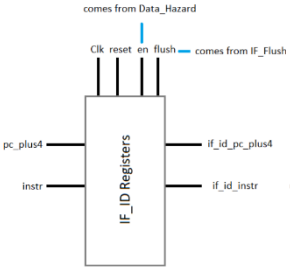
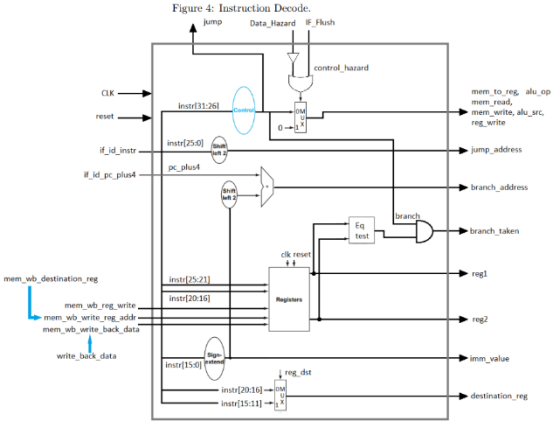


Figure 4: Instruction Decode.



The same process was repeated for each and every one of the pipeline stages and registers that had to be instantiated then connected along the processor.

The next step was developing each stage, and that was basically the same thing as before, just on a smaller, more organized scale. For example, for the ID stage, I instantiated the Register File component, a Control Unit, a sign extender and a mux. Then, it was just a matter of connecting them the right way and making sure no unwanted signals were output by placing them on temporary wires then wiring those to the actual outputs. This was done for hazard prevention, as the processor would need to stall in that case, thus no significant output was transmitted.

```
// Remember that we test if the branch is taken or not in the decode stage.
register_file INST_REG_FILE(
    .clk(clk), .reset(reset),
    .reg_write_en(mem_wb_reg_write),
    .reg_write_dest(mem_wb_write_reg_addr),
    .reg_write_data(mem_wb_write_back_data),
    .reg_read_addr_1(instr[25:21]),
    .reg_read_addr_2(instr[20:16]),
    .reg_read_data_1(reg1),
    .reg_read_data_2(reg2)
    );
mux2 #(.mux_width(5)) MUX_REG_DEST(
        .a(instr[20:16]),
        .b(instr[15:11]),
        .sel(reg_dst),
        .y(destination_reg)
        );
control INST_CONTROL(
        .reset(reset),
        .opcode(instr[31:26]),
        .reg_dst(reg_dst),
        .mem_to_reg(copy_mem_to_reg),
        .alu_op(copy_alu_op),
        .mem_read(copy_mem_read),
        .mem_write(copy_mem_write),
        .alu_src(copy_alu_src),
        .reg_write(copy_reg_write),
        .branch(branch),
        .jump(jump)
        );
sign_extend INST_SIGN_EXTEND
        (
            .sign_ex_in(instr[15:0]),
            .sign_ex_out(temp)
        );
```

Lastly, debugging was the worst part as we had to make sure that every wire was correctly spelled and connected in the correct order as that would cause major problems later on. It was the stage that took me the longest, as I stared into my code taking notes on my notebook of what was connected where. In the end, it was this simple fix:

Turning this

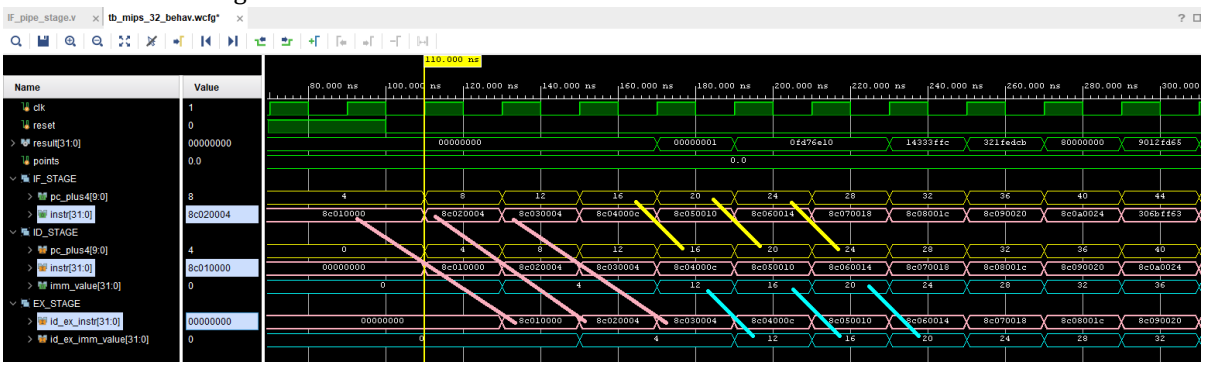
```
assign pc_plus4 = copy_pc + 10'd4;
```

Into this

```
assign pc_plus4 = pc + 10'd4;
```

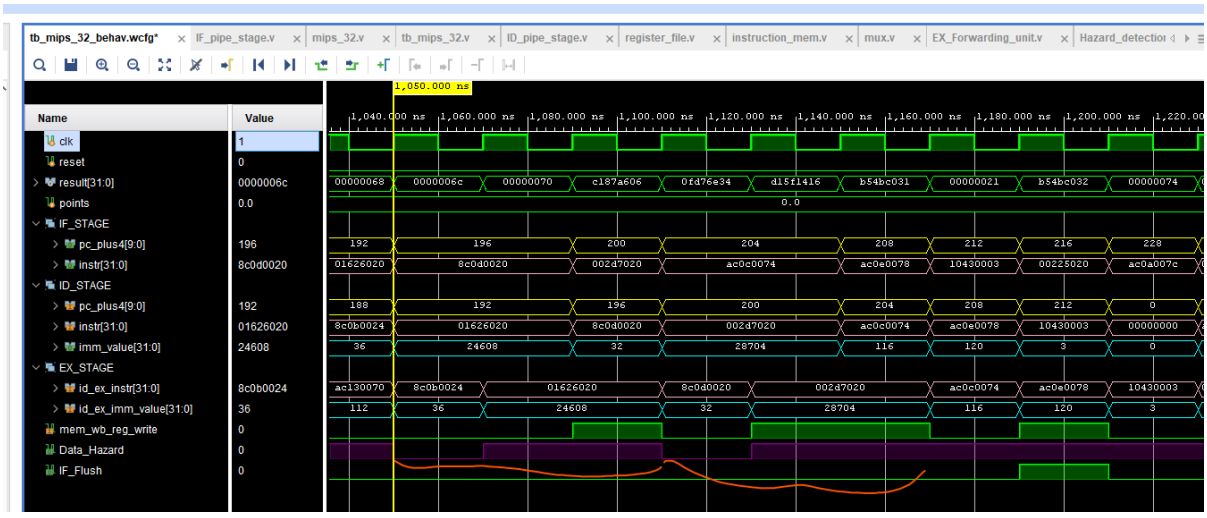
3 Simulation Results

In the image below, you can see three stages of pipelining: Instruction Fetch (IF), Instruction Decode (ID), and Execution (EX). In magenta, the instruction that is fetched in IF is passed from one stage to the other as the diagonal lines of same color point out. For example, instruction 8c010000 goes from IIF to ID then EX and the next instructions follow the same path. In yellow, the PC+4 counter is passed from IF to ID but not to EX as this stage no longer needs that information. Same thing happens when information is fetched or generated later on the course; in teal, you can see that IMM_VALUE is passed from ID to EX but non-existent in the IF stage.



Whenever there's an operation that needs information that has not been properly processed yet, we can see that that operation takes a little bit longer, as highlighted in orange in the image below. In this case, we are trying to do an operation with R13 while it hasn't been properly loaded yet so the processor stalls a cycle waiting for it to complete. Instruction 47 waits until 46 is done to continue, we can see that because Data_Hazard is 0 while 46 is executing and 47 only executes when Data_Hazard is back to one. Same thing happens in instructions 48 and 49!

```
rom[46] = 32'b10001100000010110000000000100100; // r11 = mem[36]
rom[47] = 32'b00000001011000100110000000100000; // add r12,r11,r2
rom[48] = 32'b10001100000011010000000000100000; // r13 = mem[32]
rom[49] = 32'b00000000001011010111000000100000; // add r14,r1,r13
// store the result in memory
rom[50] = 32'b10101100000011000000000000011110100; // sw mem[r0+29] <= r12
rom[51] = 32'b10101100000011100000000000011111000; // sw mem[r0+30] <= r14
```



In this last case, we are testing branch instructions. Notice that when instruction 52 (corresponding to PC_PLUS4 = 212) is decoded, we have IF_FLUSH turn on then instr, pc_plus4 and imm_value all zeroes in the following stages, all highlighted in orange. This is because IF_FLUSH means a branch is about to happen so it clears the processor and waits until all is done to then decode instruction 56 (corresponding to PC_PLUS4 = 228)

```
// Control Hazard test Branch
rom[52] = 32'b00010000010000110000000000000011; // beq r2,r3,#3                0          branch to instruction rom[56]
rom[53] = 32'b00000000001000100101000000100000; // add r10,r1,r2           0fd76e11            r10= 0fd76e11
rom[54] = 32'b00000000001001000101000000100000; // add r10,r1,r4           14333ffd            r10= 14333ffd
rom[55] = 32'b00000000001001010101000000100000; // add r10,r1,r5           321fedcc            r10= 321fedcc
// store the result in memory
rom[56] = 32'b10101100000010100000000001111100; // sw mem[r0+31] <= r10          7c                  -
```