Lab 12: Pipeline Fetch and Decode

Jiasen Zhou, Jon Johnston

April 27, 2019

1 Executive Summary

The goal of this lab was to pipeline the Fetch and Decode stages of the finished datapath. The Execute, Memory, and Writeback stages are commented out to isolate the Fetch and Decode stages of the datapath. The pipeline buffers between the stages are added so that the datapath stil operated correctly. The lab is successful, and the Fetch and Decode stages of the datapath are now pipelined.

2 Test Report

To verify operation of these modules, this lab required one test bench.

1. Pipeline Test Bench

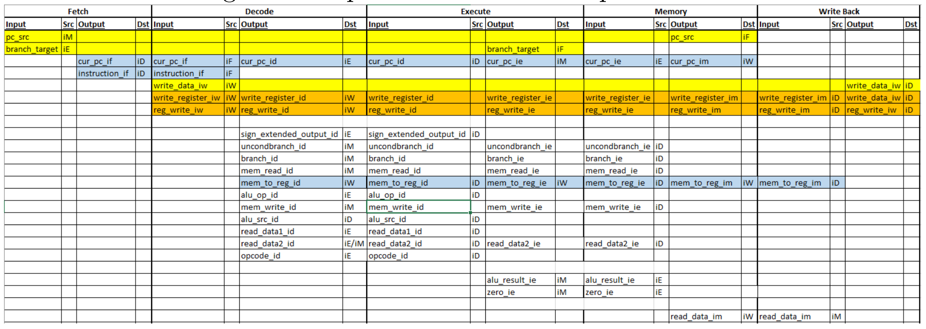
2.1 Pipeline Test Bench

The Pipeline Test Bench contains:

1. Inputs
   (a) branch_target - branch address
   (b) pc_src - the control of branch mux
   (c) reset - set the current pc to be 0
2. Outputs
   (a) uncond_branch_id, branch_id, mem_read_id, mem_to_reg_id, mem_write_id, ALU_src_id, reg_write_id, ALU_op_id, opcode_id, read_data1_id, read_data2_id, sign_extended_output_id, cur_pc_id - control signals that stored in buffer

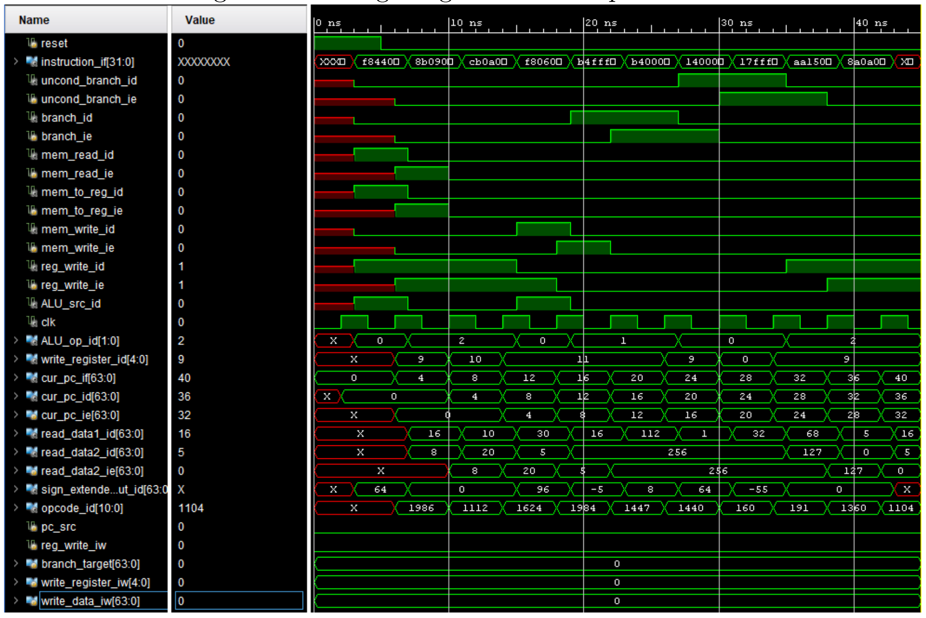
   The pipelined datapath takes the standard ten instructions and runs them through the Fetch and Decode stages to ensure that the correct values are being operated on in each stage. Operation of the testbench is verified by comparing the Simulation Results with the Expected Results Table. The pipelined Fetch and Decode stages operate as expected.

Figure 1: Expected table of the Pipeline test.



Fetch				Decode				Execute				Memory				Write Back			
Input	Src	Output	Dst	Input	Src	Output	Dst	Input	Src	Output	Dst	Input	Src	Output	Dst	Input	Src	Output	Dst
pc_src	iM													pc_src	iF				
branch_target	iE									branch_target	iF								
		cur_pc_if	iD	cur_pc_if	iF	cur_pc_id	iE	cur_pc_id	iD	cur_pc_ie	iM	cur_pc_ie	iE	cur_pc_im	iW				
		instruction_if	iD	instruction_if	iF														
				write_data_iw	iW													write_data_iw	iD
				write_register_iw	iW	write_register_id	iW	write_register_id		write_register_ie		write_register_ie		write_register_im		write_register_im	iD	write_data_iw	iD
				reg_write_iw	iW	reg_write_id	iW	reg_write_id		reg_write_ie		reg_write_ie		reg_write_im		reg_write_im	iD	reg_write_iw	iD
						sign_extended_output_id	iE	sign_extended_output_id	iD										
						uncondbranch_id	iM	uncondbranch_id		uncondbranch_ie		uncondbranch_ie	iD						
						branch_id	iM	branch_id		branch_ie		branch_ie	iD						
						mem_read_id	iM	mem_read_id		mem_read_ie		mem_read_ie	iD						
						mem_to_reg_id	iW	mem_to_reg_id	iD	mem_to_reg_ie	iW	mem_to_reg_ie	iD	mem_to_reg_im	iW	mem_to_reg_im	iD		
						alu_op_id	iE	alu_op_id	iD										
						mem_write_id	iM	mem_write_id		mem_write_ie		mem_write_ie	iD						
						alu_src_id	iD	alu_src_id	iD										
						read_data1_id	iE	read_data1_id	iD										
						read_data2_id	iE/iM	read_data2_id	iD	read_data2_ie		read_data2_ie	iD						
						opcode_id	iE	opcode_id	iD										
										alu_result_ie	iM	alu_result_ie	iE						
										zero_ie	iM	zero_ie	iE						
														read_data_im	iW	read_data_im	iM		

Figure 2: Timing Diagram of the Pipeline test.



3 Code Appendix

Listing 1: Verilog code for testing the pipelined datapath

```
`include "definitions.vh"

module pipeline;

    reg reset;
    wire [`INSTR_LEN-1:0] instruction_if;
    wire uncond_branch_id, branch_id,
         mem_read_id, mem_to_reg_id,
         mem_write_id, reg_write_id,
         ALU_src_id, clk;
         // Future
         /* zero_ie, pc_src, uncond_branch_ie,
         branch_ie, mem_read_ie, mem_to_reg_ie,
         mem_to_reg_im, mem_write_ie, reg_write_ie,
         reg_write_im, reg_write_iw,*/
    wire [1:0] ALU_op_id;
    wire [4:0] write_register_id;
         // Future
         /*write_register_ie, write_register_im,
         write_register_iw*/
    wire [`WORD-1:0]
         cur_pc_if, cur_pc_id,
         read_data1_id,
         read_data2_id,
         sign_extended_output_id;
         // Future
         /* branch_target, cur_pc_ie, cur_pc_im,
         read_data_im, read_data2_ie, write_data_iw,
         alu_result_ie*/
    wire [10:0] opcode_id;

    // Temporary Registers for Simulation
    reg pc_src, uncond_branch_ie, branch_ie,
        mem_read_ie, mem_to_reg_ie, mem_write_ie,
        reg_write_ie, reg_write_iw;
    reg [4:0] write_register_iw;
    reg [`WORD-1:0] branch_target, cur_pc_ie,
        read_data2_ie, write_data_iw;

    // Base Clock
    oscillator r_clk(.clk(clk));
    // Fetch Stage
    fetch fetch_mod(
        .clk(clk),
        .reset(reset),
        .branch_target(branch_target),
        .pc_src(pc_src),
        .instruction_if(instruction_if),
        .cur_pc_if(cur_pc_if));

    // Decode Stage
    iDecode decode_mod(
        .cur_pc_if(cur_pc_if),
        .cur_pc_id(cur_pc_id),
        .write_data_iw(write_data_iw),
        .write_register_iw(write_register_iw),
        .write_register_id(write_register_id),
        .reg_write_iw(reg_write_iw),
        .reg_write_id(reg_write_id),
        .instruction_if(instruction_if),
        .uncond_branch_id(uncond_branch_id),
        .branch_id(branch_id),
        .mem_read_id(mem_read_id),
        .mem_to_reg_id(mem_to_reg_id),
        .mem_write_id(mem_write_id),
        .ALU_src_id(ALU_src_id),
        .write_clk(clk),
        .ALU_op_id(ALU_op_id),
        .read_data1_id(read_data1_id),
        .read_data2_id(read_data2_id),
        .sign_extended_output_id(sign_extended_output_id),
        .opcode_id(opcode_id));

    // iExecute Buffer Simulation
    always @(posedge clk)
    begin
        uncond_branch_ie <= uncond_branch_id;
        branch_ie <= branch_id;
        mem_read_ie <= mem_read_id;
        mem_to_reg_ie <= mem_to_reg_id;
        mem_write_ie <= mem_write_id;
        reg_write_ie <= reg_write_id;
        cur_pc_ie <= cur_pc_id;
        read_data2_ie <= read_data2_id;
    end

//    Future Modules
//      iExecute  execute_mod(
//          .pc_in(cur_pc),
//          .read_data1(read_data1),
//          .read_data2(read_data2),
//          .sign_extend(sign_extended_output_id),
//          .opcode(opcode_id),
//          .alu_op(ALU_op),
//          .alu_src(ALU_src),
//          .alu_result(alu_result),
//          .zero(zero),
//          .branch_target(branch_target));

//      iMemory memory_mod(
//          .im_clk(clk),
//          .alu_result(alu_result),
//          .read_data2(read_data2),
//          .mem_read(mem_read),
//          .mem_write(mem_write),
//          .zero(zero),
//          .branch(branch),
//          .uncondbranch(uncond_branch),
//          .read_data(read_data),
//          .pc_src(pc_src));

//      iWrite_back  writeback_m(
//          .read_data(read_data),
//          .alu_result(alu_result),
//          .MemtoReg(mem_to_reg),
//          .write_data(write_data));

initial
    begin
        reset = 1;
        pc_src = 0;
        branch_target = 0;
        reg_write_iw = 0;
        write_register_iw = 0;
        write_data_iw = 0; #5
        reset = 0; #40
   $finish;
    end

endmodule
```