Lab 1: The Instruction Fetch Stage

Yousef Jarrar, Nicholas Chiodini

April 15, 2019

# Introduction

The objective of this lab is to implement and test the Instruction Fetch (IF) pipeline stage of the MIPS five stage pipeline. There are five modules that needs to be implemented which are: the program counter, the incrementer, the MUX, the memory, and the instruction fetch ID latch. All of these modules will make up the Instruction Fetch stage.

# Interface

The program counter module takes in a 32-bit number that represents the New Program Counter (NPC), updates the current PC to the NPC value, and outputs a 32-bit number to the rest of the dependent modules.

Table 1: pc\_mod Inputs

|  |  |
| --- | --- |
| Name | Function |
| npc | The current 32-bit PC value |

Table 2: pc\_mod Outputs

|  |  |
| --- | --- |
| Name | Function |
| pc\_out | The updated 32-bit PC value |

The incrementer module takes in a 32-bit constant 1 and a 32-bit PC value. Then the two inputs are added together, and outputs the result to another 32-bit wire.

Table 3: incrementer Inputs

|  |  |
| --- | --- |
| Name | Function |
| pc\_in | The 32-bit number that is the current PC value |

Table 4: incrementer Outputs

|  |  |
| --- | --- |
| Name | Function |
| pc\_out | The updated 32-bit number that is the result of incrementing pc\_in by 1. |

The MUX module takes in the next sequential PC value and the PC value that is calculated in the Execute stage depending on the value of the select line set in the Memory stage.

Table 5: mux Inputs

|  |  |
| --- | --- |
| Name | Function |
| a | The 32-bit number that holds the next sequential PC value |
| b | The 32-bit number that holds the PC value that is calculated in the Execute stage |
| sel | The line that determines if inputs a or b will be selected to be the output |

Table 6: mux Onputs

|  |  |
| --- | --- |
| Name | Function |
| y | The 32-bit wire that holds the value that is specified via the select line |

The instruction memory module takes in the current 32-bit PC value as its input and outputs the 32-bit instruction which that PC value is currently pointing at.

Table 7: mem Inputs

|  |  |
| --- | --- |
| Name | Function |
| addr | The current 32-bit PC value |

Table 8: mem Outputs

|  |  |
| --- | --- |
| Name | Function |
| data | The 32-bit register which the PC value is currently pointing at |

The Instruction Fetch-Decode latch (IF\_ID\_Latch) module takes in the next sequential PC value and address and outputs them to the next stage.

Table 9: IF\_ID Inputs

|  |  |
| --- | --- |
| Name | Function |
| instruction\_in | The 32-bit wire from the Fetch stage |
| npc\_in | The 32-bit wire from the Fetch stage |

Table 10: IF\_ID outputs

|  |  |
| --- | --- |
| Name | Function |
| instruction\_out | The updated 32-bit PC value from instruction\_in |
| npc\_out | The updated 32-bit PC value from npc\_in |

The Instruction Fetch module for the Instruction Fetch stage is where all the individual modules are connected to each other. It takes in a PC value and the sequential PC value, and outputs the IF\_ID latch instructions and IF\_ID latch PC value.

Table 11: ifetch Inputs

|  |  |
| --- | --- |
| Name | Function |
| EX\_MEM\_PCSrc | The PC value from the instruction fetch-decode latch stage |
| EX\_MEM\_NPC | The 32-bit sequential PC value from the instruction fetch-decode latch stage |

Table 12: ifetch Outputs

|  |  |
| --- | --- |
| Name | Function |
| IF\_ID\_INSTR | The instructions from the instruction fetch-decode latch stage |
| IF\_ID\_NPC | The PC value from the instructions fetch-decode latch stage |

# Design

The design of the program counter takes in a 32-bit New Program Counter (NPC), updates the current PC value to the NPC value, and outputs the results to all the dependent modules.

Figure 1: Program Counter design



The design of the incrementer takes in the current PC value and outputs the PC value incremented by 1.

Figure 2: Incrementer design



The design of the MUX module takes in a s0 and s1 wire in which a select line determines the next PC value depending if the line goes high (1) or low (1).

Figure 3: MUX design



The design for the instruction memory module takes in a 32-bit PC value, and outputs the 32-bit instruction depending on where the PC value is currently pointing at.

Figure 4: Memory module design



# Implementation

Listing 1: Implementation for pc\_mod

|  |
| --- |
| `timescale 1ns / 1ps  module pc\_mod( output reg [31:0] pc\_out, input wire [31:0] npc );  initial  begin  pc\_out <= 0;  end  always @ ( npc )  begin  #1 pc\_out <= npc;  end  endmodule |

Listing 2: Implementation for incrementer

|  |
| --- |
| `timescale 1ns / 1ps  module incrementer ( input wire [31:0] pc\_in, output wire [31:0] pc\_out );  assign pc\_out = pc\_in + 1; // Increment PC by 1  endmodule |

Listing 3: Implementation for mem

|  |
| --- |
| module mem(  output reg [31:0] data,  input wire [31:0] addr  );  // Regsiter Declarations  reg [31:0] MEM[0:127];  // Initialize Registers  initial begin  MEM[0] <= 'hA00000AA;  MEM[1] <= 'h10000011;  MEM[2] <= 'h20000022;  MEM[3] <= 'h30000033;  MEM[4] <= 'h40000044;  MEM[5] <= 'h50000055;  MEM[6] <= 'h60000066;  MEM[7] <= 'h70000077;  MEM[8] <= 'h80000088;  MEM[9] <= 'h90000099;  end  always @ ( addr )  begin  data <= MEM[addr];  end  endmodule |

Listing 4: Implementation for if\_id module

|  |
| --- |
| `timescale 1ns / 1ps  module if\_id ( output reg [31:0] instructions\_out, npc\_out,  input wire [31:0] instructions\_in, npc\_in  );  initial begin  instructions\_out <= 0;  npc\_out <= 0;  end  always @ \* begin  #1  instructions\_out <= instructions\_in;  npc\_out <= npc\_in;  end  endmodule |

Listing 5: Implementation for the top-level Instruction Fetch module

|  |
| --- |
| module I\_FETCH (  input EX\_MEM\_PCSrc,  input wire [31:0] EX\_MEM\_NPC,  output wire [31:0] IF\_ID\_INSTR, IF\_ID\_NPC  );  // signals  wire [31:0] PC;  wire [31:0] data\_out;  wire [31:0] npc, npc\_mux;  // instantiations  mux mux1 ( .y(npc\_mux),  .a(EX\_MEM\_NPC),  .b(npc),  .sel(EX\_MEM\_PCSrc));  pc\_mod pc\_mod1 ( .pc\_out(PC),  .npc(npc\_mux));  mem mem1 ( .data(data\_out),  .addr(PC));  if\_id if\_id1 ( .instructions\_out(IF\_ID\_INSTR),  .npc\_out(IF\_ID\_NPC),  .instructions\_in(data\_out),  .npc\_in(npc));  incrementer incrementer1 ( .pc\_out(npc),  .pc\_in(PC));  endmodule |

# Test Bench Design

This test bench is designed to simulate instruction fetch routines over a *20ns* simulated run. The PC value starts at 0 initially and jumps to 1 after 9 cycles, and then jumps the PC value to 5 in which it outputs all the instruction fetch routines that were in the instruction memory module (Listing 3).

Listing 6: Instruction Fetch Test Bench

|  |
| --- |
| `timescale 1ns / 1ps  module pipeline();  // Inputs  reg EX\_MEM\_PCSrc;  reg [31:0] EX\_MEM\_NPC;  // Outputs  wire [31:0] IF\_ID\_INSTR;  wire [31:0] IF\_ID\_NPC;  // Instantiate the Unit Under Test (UUT)  I\_FETCH uut (  .EX\_MEM\_PCSrc(EX\_MEM\_PCSrc),  .EX\_MEM\_NPC(EX\_MEM\_NPC),  .IF\_ID\_INSTR(IF\_ID\_INSTR),  .IF\_ID\_NPC(IF\_ID\_NPC)  );  initial  begin  EX\_MEM\_NPC = 0;  EX\_MEM\_PCSrc = 0;  #9  EX\_MEM\_PCSrc = 1;  EX\_MEM\_NPC = 5;  #1  EX\_MEM\_PCSrc = 0;  #10;  $stop;  end  endmodule |

# Simulation

Figure 5: Timing diagram from 0ns – 8ns

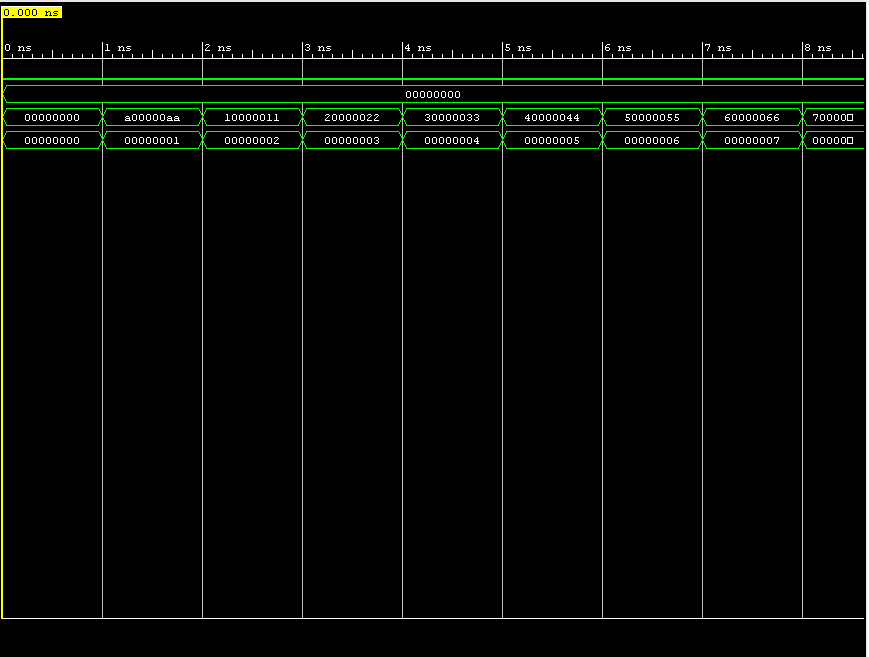


Figure 6: Timing diagram from 9ns – 17ns

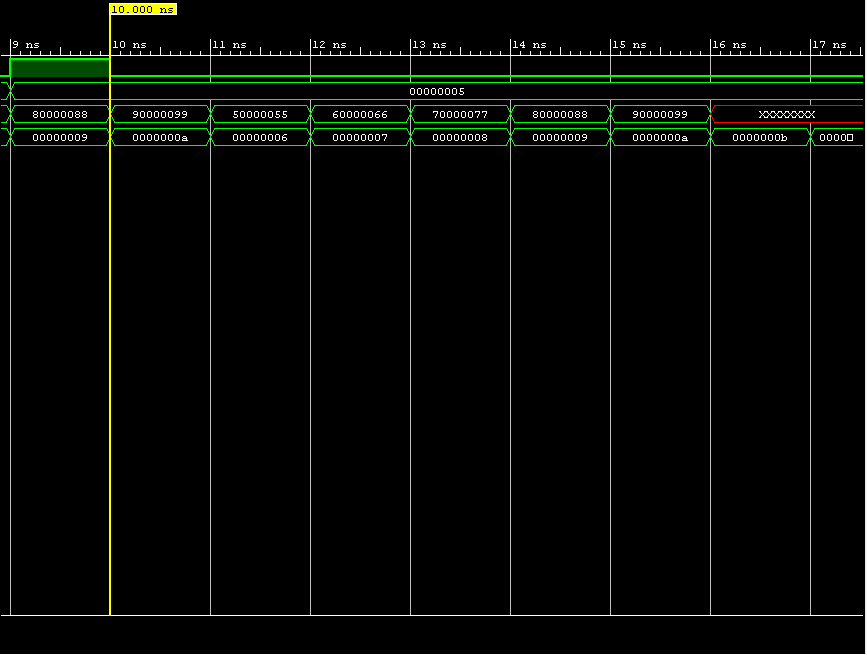
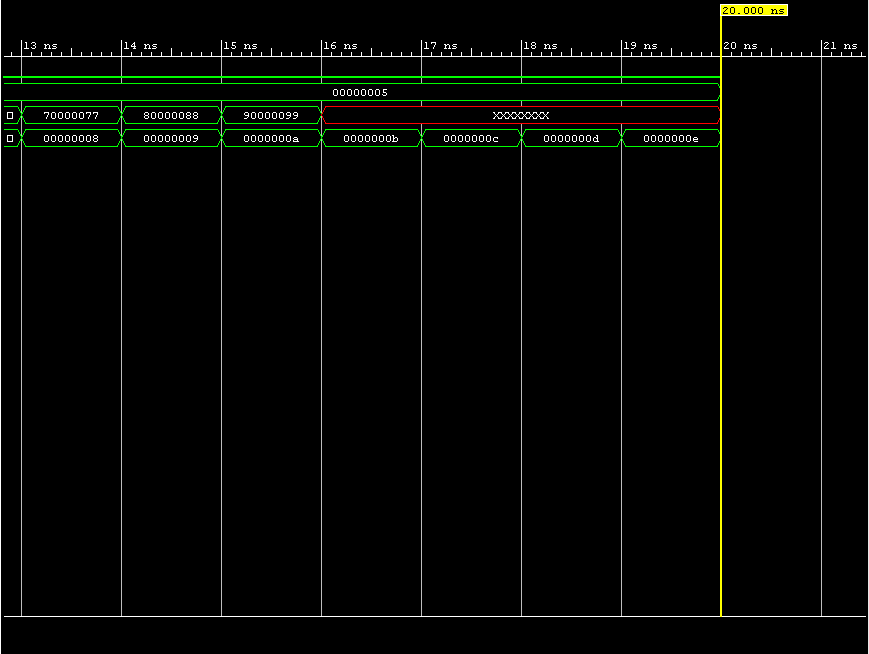


Figure 7: Timing diagram from 13ns – 20ns



# Conclusions

The instruction fetch stage for this lab was successfully implemented. The main thing I learned in doing is how the instruction fetch-decode latch is designed and implemented in Verilog code. I never thought about using a top-level module that connects all the modules together into one since I just instantiate all the modules in the test bench. What I would do differently in this lab was to get more familiar with the Vivado software beforehand to save myself time to figure out how to add source files and debug the Verilog code.