1 Instruction Fetch

The first stage of the MIPS processor is the instruction fetch stage (IF). This is the stage where the instructions are fetched from memory. The objective of this lab is to fully implement and understand the first stage of the MIPS processor data path.

1.1 Components

List of Stage 1 Components
Figure 1: The PC register holds the current value of the Program Counter (PC). It takes in as its input a 32-bit number which represents the New Program Counter (NPC), updates the current PC to the NPC value, and outputs the results to all dependent modules. See listing 1 for corresponding Verilog code block.

Figure 2: The Incrementer takes in the current PC value and outputs the next PC value. It takes in as its input a 32-bit constant 1 and the 32-bit PC. It adds the two inputs together and outputs the result. See listing 2 for corresponding Verilog code block.
Figure 3: The MUX takes in the next sequential PC value, the PC value calculated in Stage Three: Execute, and outputs value specified via the select line from Stage Four: Memory. When the select line from the MEM stage is low (0), it will output the next sequential PC Value. When the line goes high (1) it will output the value calculated in the Execute stage. See listing 3 for corresponding Verilog code block.
Figure 4: 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. See listing 4 for corresponding Verilog code block.

1.2 Code

Listing 1: Code for the Program Counter Module

```
`timescale 1ns / 1ps

// Program Counter Module

module pc(input [31:0] npc,
          output reg [31:0] pc_out
          );

initial
begin

/* Initial Thread which will set pc_out to zero on the first run */

pc_out <= 0;
end

always @ (npc)
begin

#1  /* update value of PC to New Program Counter */
end

endmodule
```
Figure 5: The Instruction Fetch-Decode latch (IF_ID_Latch). This latch takes in the next sequential PC value and address and outputs them to the next stage. See listing 5 for corresponding Verilog code block.

Listing 2: Code for the Incrementer Module

```verilog
//timescale 1ns / 1ps
// Incrementer Module which accepts the Program Counter and adds 1 to it.
module incrementer(
```

5
Listing 3: Code for the MUX Module

```verilog
input wire [31:0] pc_in,
output wire [31:0] pc_out
);

assign pc_out = /* add 1 to pc_in */;
endmodule
```

Listing 4: Code for the Instruction Memory Module

```verilog
// /////////////////////////////////////////////////////
// Memory module which holds all of the instructions
// needed by the MIPS processor
// /////////////////////////////////////////////////////
module mem(
    input [31:0] addr,
    output reg [31:0] data
);
reg [31:0] MEM[0:127];

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 5: Code for the IF_ID Latch

*timescale 1ns / 1ps

// Latch witch passes data from the Fetch Stage to the Decode Stage.
module IF_ID(
    input wire [31:0] instruction_in, npc_in,
    output reg [31:0] instruction_out, npc_out
);

initial begin
    instruction_out <= 0;
    npc_out <= 0;
end

always @ (*
begin

Listing 6: Code for the top level Instruction Fetch Module

```verilog
`timescale 1ns / 1ps

// The top level module for the Instruction Fetch Stage.
// This is how all of the individual modules are connected to each other.

module IFETCH(
    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] dataout;
wire [31:0] npc, npc_mux;

// instantiations
mux mux_1 (.npc(npc_mux),
            .s1(EX_MEM_NPC),
            .s0(npc),
            .select(EX_MEM_PCSrc));

pc pc_1 (.pc_out(PC),
         .npc(npc_mux));
mem mem_1 (.data(dataout),
            .addr(PC));

IF_ID IF_ID_1 (.instruction_out(IF_ID_instr),
                .npc_out(IF_ID_npc),
                .instruction_in(dataout),
                .npc_in(npc));
```

#1

/* update instruction_out and npc_out values with new instruction and npc values from the Fetch stage. */

end
endmodule

incrementer incrementer1 (.pc_out(npc),
    .pc_in(PC));

endmodule

1.3 Testbenches & Timing Diagrams

Listing 7: Instruction Fetch Test Bench

```vhdl
'timescale 1ns / 1ps

///////////////////////////////////////////////////////////////////////
// Stage 1 Test Bench
///////////////////////////////////////////////////////////////////////

module IFETCH_TEST;

    // 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)
    IFETCH uut (EX_MEM_PCSrc,
        .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;
```


endmodule

Figure 6: Instruction Fetch Stage Simulation.

1.4 What to turn in

Fully implement all of the components listed in section 1.1. You may use the code provided in section 1.2. Run a full 20 ns simulation and demonstrate successful instruction fetch routines. For this test show your Program Counter incrementing sequentially for the first 10 ns then emulate out of order PC jumps. Compare your results to Figure 6. Turn in a lab report as outlined by your lab instructor and be ready to demonstrate your code if asked.