Lab 4: iFetch Buffer

Madeline Stephens and Michael Hu

February 20, 2017

1 Introduction

Building upon the previous lab, Lab 4 elaborates upon the first stage of the pipeline of the MIPS computer – Fetch. In Lab 3, it was discovered that the output of the instruction memory was lagged in comparison the program counter. This was due to the fact that both the program counter and Fetch were programmed to operate on the same edge of the clock cycle. To remedy this error, Lab 4 adds a delay by implementing a buffer.

2 Interface

A simple buffer was created to rectify the unintentional lag between Fetch and the program counter. In order to do this, the buffer took in four inputs: one from the program counter, one from instruction memory, clock, and reset. Once completed, the buffer was instantiated in iFetch. Interestingly, the two outputs from the buffer, nPC and IR, make up all of the outputs from the Fetch module as a whole. The buffer simply ensures that the two outputs are constant through the entire clock cycle so that they may be correctly used to calculate new values. This task is fulfilled by a delay function that is triggered by the negative edge of the clock cycle. It must be noted that all inputs into the buffer are taken in as binary. Similarly, the outputs of the buffer are also represented in binary.

3 Design

This lab focused on testing the buffer and adding it to the iFetch module. When properly hooked up in Fetch, the buffer takes in the outputs of the program counter and instruction memory and adds a delay on the negative clock edge so that both units are no longer programmed on the same clock cycle. By doing so, it is ensured that the two outputs are constant. This allows the next stage of the program to operate off of a finalized value.

4 Implementation

The buffer was created using an always block in Verilog. The code for the buffer was provided by Dr. Schubert and can be referenced in Listing 1 on page 2.The sensitivity list of the always block included the negative edge of the clock cycle and the positive edge of reset. It is important to note the negative edge of the clock cycle due to the fact that this is what allows the buffer to eliminate the lag between the program counter and instruction memory. This can be attributed to the fact that the lag can ultimately be traced to the fact that both outputs are time of the same edge of the clock cycle. Additionally, the positive edge of the rest i used in order to ensure that the reset occurs where intended. The buffer utilizes the delay function defined in the Verilog header file "definitions.vh" provided to the class. After the delay is called, an if-else statement is used to enable reset functionality. As can be seen in Listing 1, of reset is HIGH, both outputs are set to zero. Otherwise, the inputs are connected to the outputs. Similar to an op-amp buffer, the output is connected to the input.

Listing 1: Verilog code for building a buffer.

```
`include "definitions.vh"

module buffer_ifid#(parameter DELAY=0)(
    input clk,
    input reset,
    input [`WORD-1:0] nPC_if,
    input [`WORD-1:0] IR_if,
    output reg[`WORD-1:0] nPC_id=`ZERO,
    output reg[`WORD-1:0] IR_id=`ZERO
    );

    always @(negedge(clk), posedge(reset))
    begin
      #(`DELAY);
      if (reset)
      begin
        nPC_id<=`ZERO;
        IR_id<=`ZERO;
      end
      else
      begin
        nPC_id<=nPC_if;
        IR_id<=IR_if;
      end
    end
endmodule
```

After the buffer was created and tested, it was then instantiated inside the

iFetch module. The code for this can be found in Listing 2. While hooking up the buffer inside of iFetch was a largely simple task, it must be noted that it was necessary to create two new internal signals: IR_wire and PC_out. These two wires were needed due to the cyclical nature of a buffer. Simply, when reset was low, these two wires were used to connect the inputs to the outputs. All other signals hooked up to the buffer were predefined in the iFetch module.

Listing 2: Verilog code for iFetch.

```
`include "definitions.vh"

module iFetch#(parameter STEP=32'd1, SIZE=1024)(
    input clk,
    input reset,
    input PCSrc,
    input [`WORD-1:0] BrDest,
    output [`WORD-1:0] nPC,
    output [`WORD-1:0] IR
    );
    wire [`WORD-1:0] PC;
    wire [`WORD-1:0] new_PC;
    wire[`WORD-1:0] nextPC;


    wire [`WORD-1:0] IR_wire;
    wire [`WORD-1:0] PC_out;

    assign nPC=nextPC;

    mux#(`WORD) PCsel(
    .Ain(nextPC),
    .Bin(BrDest),
    .control(PCSrc),
    .mux_out(new_PC)
    );

    register myPC(
    .clk(clk),
    .reset(reset),
    .D(new_PC),
    .Q(PC)
    );

    adder incrementer(
    .Ain(PC),
    .Bin(STEP),
    .add_out(nextPC)
```

```
    );

    instr_mem#(SIZE) iMemory(
    .clk(clk),
    .pc(PC),
    .instruction(IR_wire)
    );

     buffer_ifid buffer(
    .clk(clk),
    .reset(reset),
    .nPC_if(nextPC),
    .IR_if(IR_wire),
    .nPC_id(PC_out),
    .IR_id(IR)
    );

endmodule
```

5 Test Bench Design

To verify that the above code functioned properly, test benches were created and simulated. Various cases were used to test each module and were chosen according to the operation of the unit under test. Typically, the test cases were chosen to ensure that the code operated as intended, even where common errors could likely occur.
The code for the buffer test bench can be found in Listing 3 on page 4. The test cases for the buffer test bench were chosen to test and illustrate the functionality of the buffer. The initial test case was a basic test to ensure that the outputs would be the same as the inputs if reset was LOW. Next, the value of nPC_if was changed to see if the buffer would properly handle a changing value. The third test case put the functionality of the reset signal under scrutiny. Reset was set to HIGH in order to make sure that the outputs would be set to zero as expected. Finally, reset was set to zero to verify the reset could be changed and that the outputs would return to their values set when reset was low.

Listing 3: Verilog code for testing a buffer.

```
`timescale 1ns / 1 ps
`include "definitions.vh"

module buffer_test;

    wire clk;
    reg reset;
```

```
    reg [`WORD-1:0] nPC_if;
    reg [`WORD-1:0] IR_if;
    wire [`WORD-1:0] nPC_id;
    wire [`WORD-1:0] IR_id;

    oscillator#() clock(
    .clk(clk)
    );

    buffer_ifid#() uut(
        .clk(clk),
        .reset(reset),
        .nPC_if(nPC_if),
        .IR_if(IR_if),
        .nPC_id(nPC_id),
        .IR_id(IR_id)

    );

    initial begin

    reset = 0;
    nPC_if = 1;
    IR_if = 1;


    #5;

    nPC_if = 3;


    #12;

    reset = 1;

    #5

    reset = 0;


    end

endmodule
```

After integrating the buffer into iFetch, it was important to verify that the entire module still functioned as it should. Because iFetch could operate in two

distinct ways, two test benches were required: one for sequential operation and one for branching. The test bench for the sequential and branching operations of iFetch can be found in Listing4 and Listing 5, respectively. These test benches were created for Lab 3, and were re-used to verify the proper functionality of the fetch module with the buffer integrated.

Listing 4: Verilog code for testing a iFetch sequentially.

```
`timescale 1ns / 1ps
`include "definitions.vh"

module iFetch_sequential_test;

wire clk;
reg reset;
reg PCSrc;
reg [`WORD-1:0] BrDest;
wire [`WORD-1:0] nPC;
wire [`WORD-1:0] IR;


    oscillator#() clock(
    .clk(clk)
    );


    iFetch#() uut (
    .clk(clk),
    .reset(reset),
    .PCSrc(PCSrc),
    .BrDest(BrDest),
    .nPC(nPC),
    .IR(IR)
    );

    initial begin

    reset = 0;
    PCSrc = 0;
    BrDest = 0;

    #10;

    end
endmodule
```

Listing 5: Verilog code for testing a iFetch branching

```
`timescale 1ns / 1ps
`include "definitions.vh"

module iFetch_branch;

    wire clock;
    reg reset;
    reg PCSrc;
    reg [`WORD-1:0] BrDest;
    wire [`WORD-1:0] nPC;
    wire [`WORD-1:0] IR;

    oscillator#() clock_signal(
    .clk(clock)
    );

    iFetch#() uut(
    .clk(clock),
    .reset(reset),
    .PCSrc(PCSrc),
    .BrDest(BrDest),
    .nPC(nPC),
    .IR(IR)
    );

    initial begin

    reset = 0;
    PCSrc = 1;
    BrDest = 3;
    #15;

    BrDest = 9;
    #15;

    BrDest = 1024;
    #15;

    end

endmodule
```

Figure 1: Timing diagram for the buffer test.

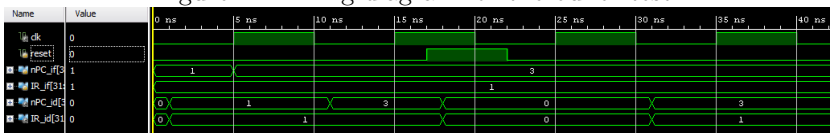
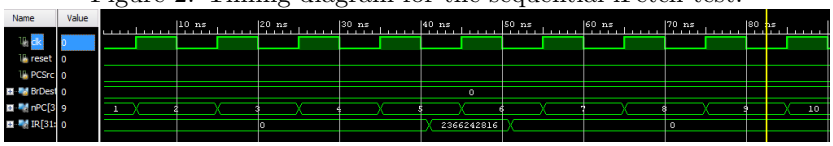


Figure 2: Timing diagram for the sequential iFetch test.



6 Simulation

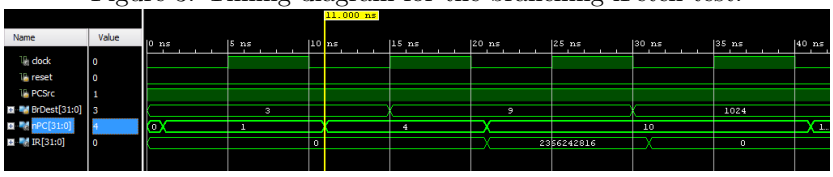
The timing diagram for the buffer test can be found in Figure 3. By studying this figure, it can be verified that the fetch module operated as intended. It can be seen that the outputs change on the negative edge of the clock cycle after a 1 ns delay. This is intentional, as the delay function defined in the header file is approximately 1 ns. Additionally, it can be observed that reset operates as intended, due to the fact the the outputs are set to zero when the reset signal is high.

The simulation diagrams for the iFetch test benches can be found in Figure 2 and Figure 3. By observing the waveforms, it can be seen that the Fetch module still operates correctly after the integration of the buffer. The most obvious indicator is the fact that the program counter increments correctly.

7 Conclusions

To build a properly functioning fetch unit, it was necessary to create and integrate a buffer. The buffer corrects lagging between outputs and allows new values to be calculated. After testing and simulating the modules, it was found that the components functioned as expected. The simulation waveforms gener-

Figure 3: Timing diagram for the branching iFetch test.



ated in the lab provide visual verification of the modules' functionality. With a bit of tweaking, the buffer was properly integrated into the fetch module and was found to function correctly. Thus, with the buffer created in Lab 4, a fully-functioning fetch unit was constructed.