Lab 3 - Fetch Stage

Shaun Gordon and Sam Sahli

February 6, 2017

1 Introduction

The purpose of this lab was to build and test a fetch unit. To accomplish this an instruction memory was initialized and tested. Then, to implement the fetch module, all of the previous modules were connected into a instruction fetch unit. This lab introduced the concept of a fetch module, and helped develop the knowledge of how one is used in the operation of a MIPS computer.

2 Interface

The first module designed in this lab was the instruction memory for the computer. This module was constructed with two inputs and a single output. The first input was a one bit clock and the second was a 32 bit wire. This wire is responsible for delivering the program count to the instruction memory. This wire was titled pc. The output of the memory was a 32 bit register titled instruction. The instruction register is responsible for carrying the data that was stored at a specific location in memory. So the program counter input tells the memory the address of a specific instruction, and then the output carries the data from that address out of the module. All of the data mentioned is sent in binary.

The other module designed in this lab was the fetch module. As stated earlier, this module required the connection of the four previous modules. The program counter, adder, multiplexer, and instruction memory are all included in the fetch module. The fetch module itself has four inputs and two outputs. There is a one bit clock input as well as a one bit reset input. The clock input is controlled by the system clock and the reset input reinitializes the program counter to zero. There is also a one bit program counter source input that is responsible for controlling the multiplexer. The final input is a 32 bit branch destination wire that holds the data that controls where the program counter will go next. If the PC source wire is high, the multiplexer will return the value on the branch destination wire and set the program counter to whatever value was on it. The two outputs from the fetch module are the instruction and next program counter wires. They are both 32 bits but are used very differently. The instruction wire sends the data from a memory address out of the module, and the next program counter wire carries the next value that will be sent to the program counter when there is no branch.

3 Design

The best way to describe the design of the fetch module is to characterize all the modules that are contained inside of it. First there is the multiplexer, which is responsible for determining the value that will be sent to the program counter. The multiplexer has one input that is incremented by the adder and another input that contains a possible branch destination. If the control wire of the multiplexer goes high, the possible branch destination will be sent to the program counter. The program counter accepts a value from the multiplexer every clock cycle, and then, because it is a register, holds that value until it is changed. It sends whatever value it holds to both the adder and the instruction memory. The adder increments the value from the program counter and by way of the multiplexer, sends this new value back into the program counter register. The instruction memory takes the value from the program counter and locates a memory address based on that value. It then outputs the data that was held in that memory address. So as a whole, the fetch module is designed to operate on a normally incrementing program counter until a branch is requested. When this happens it should deliver data from the instruction memory based upon the branch value that is passed through the multiplexer. This is accomplished with the modules described above.

4 Implementation

The Verilog code for the instruction memory module is displayed below in Listing 1. The module is built using a parameter, SIZE, that is used to determine the number of memory locations. This is done so any time the module is implemented, the size can be manipulated without any change to the original code. The output instruction is also initialized to be the 32 bit number zero. This makes sense because when the computer is booted up the most logical value for the instruction module output is zero. Next, an array of 1024 32 bit memory locations is created and named imem. The array is created with parameter SIZE described above. The Verilog code then specifies that at the positive edge of every clock cycle, the 32 bit data that is held in a memory location of imem is carried onto the instruction output. Finally, the imem array is initialized with a text file that contains 1024 32 bit numbers. Essentially this file is used as a sample memory and will operate as the data that is held in the instruction memory during testing.

The Verilog code for the Fetch module is displayed below in Listing 2. The first thing done in the code, is the initialization of three internal wires required for proper function. There is a wire, titled nextPC, that carries the value from the adder output to the A input of the multiplexer . There is a wire, titled newPC, that carries the output of the multiplexer into the program counter register . Finally, there is a wire, title PC, that carries the value of the program counter to the instruction memory and the A input of the adder. Next, there is an assign statement that puts the value of the nextPC wire onto the output wire nPC. This is done because the value of nextPC is used as an input to an internal module, and therefore can't be set as an output.The body of the code consists of function calls to the four modules that are required in a fetch unit.

Listing 1: Verilog code for implementing an instruction memory.

```
`include "definitions.vh"

module instr_mem#(
    parameter SIZE=1024)(
    input clk,
    input [`WORD - 1:0] pc,
    output reg [`WORD - 1:0] instruction=`WORD'b0
    );

    reg[`WORD - 1:0] imem [SIZE-1:0];

    always @(posedge(clk))
        instruction<= imem[pc];

    initial
        $readmemb(`IMEMFILE, imem);

endmodule
```

Listing 2: Verilog code for implementing a Fetch Module.

```
`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;

    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)
    );
endmodule
```

5 Test Bench Design

The test bench for the instruction memory is displayed below in Listing 3. This test bench was designed to test that the value on the program counter was used to find and address in the instruction memory, and then put the data in that address onto the instruction output. To do this specific address locations needed to be put on the program counter, and if the instruction output matched what was known to be in that memory location, the operation of the memory module would be confirmed. The test bench in Listing 3 does this by testing memory locations 0 through 8, a middle address, and the largest address location. The value in each address location was known and matched up with what the module produced.

The test bench for the Fetch module is displayed below in Listing 4. To ensure the operation of the fetch module, this test bench had to be designed to confirm several things. First, that when the clock was allowed to run the program counter was incremented by one for every cycle. Second, that the value of the program counter was used to find an instruction from a memory address. Third, that the branch destination could be set and used to manipulate the program counter. This therefore would provide an instruction from the memory. And finally, that the reset can be raised to return the program counter to zero and restart the incrementing process. All of this was accomplished with the values tested for the various inputs shown in the test bench in Listing 4.

Listing 3: Verilog code for testing the instruction memory.

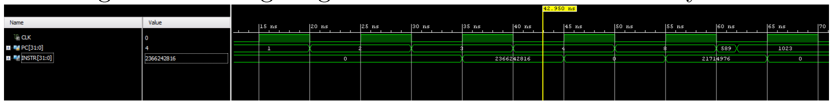
```
`include "definitions.vh"

module instr_mem_test;

        wire CLK;
        reg [`WORD-1:0] PC;
        wire [`WORD-1:0] INSTR;

        oscillator clk_gen(CLK);

        instr_mem#(1024) UUT (
                .clk(CLK),
                .pc(PC),
                .instruction(INSTR)
        );

        initial begin
            PC = 0;
            #`CYCLE;

        PC = 1;
        #`CYCLE;

        PC = 2;
        #`CYCLE;

        PC = 3;
        #`CYCLE;

        PC = 4;
        #`CYCLE;

        PC = 8;
        #`CYCLE;

        PC = 1023;
        #`CYCLE;
```

```
    end

endmodule
```

Listing 4: Verilog code for testing the Fetch module.

```
`include "definitions.vh"


module iFetch_test;

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

    oscillator clk_gen(clk);

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

        initial begin
            BrDest = 0;
            #`CYCLE;
        #`CYCLE;
        #`CYCLE;
        #`CYCLE;

        BrDest = 10;
        PCSrc = 1;
        #`CYCLE;

        BrDest = 500;
        #`CYCLE;

        BrDest = 20;
        #(`CYCLE/5);
```

Figure 1: Timing diagram for the instruction memory test.



```
          BrDest = 1023;
          #(`CYCLE*4/5);

          PCSrc = 0;
          #`CYCLE;

          reset = 1;
          #`CYCLE;

          reset = 0;
          #`CYCLE;

     end


endmodule
```

6 Simulation

The timing simulation for the instruction memory is displayed in Figure 1. This waveform shows that for the various program counter input values, there is an instruction output. This instruction output was confirmed to be correct when compared to the actual values in the test instruction memory file.

The timing simulation for the first fetch module test is displayed in Figure 2. This waveform shows that when there is no other input besides the clock signal the program counter is incremented. It also shows that when the branch destination and the multiplexer control is raised, the branch destination value is put onto the program counter. It also demonstrates that in both of these instances a memory address is found and put onto the instruction output. Finally, it shows that reset can be raised to return the program counter to its initial value.

The timing simulation for the second fetch module test is displayed in Figure 3. This waveform illustrates that when the fetch module is allowed to run continuously the program counter will continue to be incremented. This, accompanied from what was observed in Figure 2, proves that the Fetch module is operating correctly.

Figure 2: Timing diagram 1 for the Fetch module test.

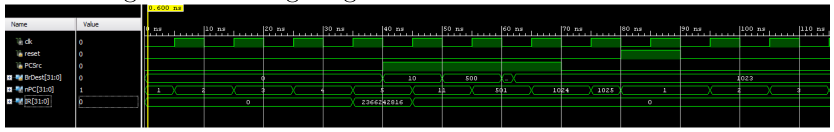
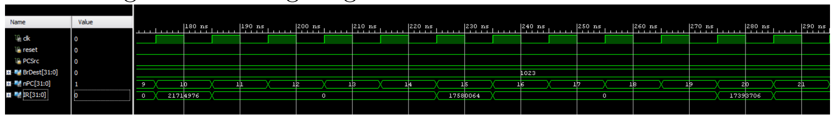


Figure 3: Timing diagram 2 for the Fetch module test.



7 Conclusions

This lab was a huge success because both the instruction memory and fetch module were designed, built, and tested to operate correctly. The most important concept to remember about this lab is how the fetch module uses its internal modules to complete a fetch function. The program counter is incremented until a branch is requested, and then the instruction on that branch address is returned. This is an oversimplification, but is a broad overview of what was accomplished in this lab.