Lab 3

Matthew Carrano and Breana Leal

January 30, 2018

1 Introduction

The goal of this lab is to create an instruction memory module, and then combine said module with previous modules to form the fetch stage module. The instruction memory module outputs data determined by the input address. The fetch stage module uses the instruction memory module in conjunction with an adder, mux and program counter (a register module). In the fetch stage, by using the select of the mux, the program counter will either give an address to the instruction memory that is sequential or a branch value. Then the respective data will be outputted.

2 Interface

The inputs of the instruction memory module are clk and pc. The clk is simply a pulse with a period of 10ns. With each positive pulse edge, the pc will increment, which will affect the output. This will be explained further in implementation.

The inputs of the iFetch are clk, reset, pc_src, and branch_target. The outputs of the iFetch are incremented_pc, instruction, and cur_pc. Its inputs and outputs serve as connecting pieces for the previously discussed modules: mux, program counter, adder, and instruction memory.

The modules in iFetch are connected as follows. incremented_pc is the output of the adder, and thus the incremented value of the current program counter address (cur_pc). incremented_pc is also an input of the mux. The other input of the mux is branch_target. pc_src is the select signal of the mux. The output from the mux is the input of the program counter (this is a wire called new_pc) along with inputs clk and reset. As mentioned, the output of the program counter is cur_pc. cur_pc becomes an input for the adder module.The adder's other input is STEP, a parameter defined as 'WORD'd4; this equals 4 in decimal. cur_pc and clk are inputs of the instruction memory. They lead to the output called instruction.

3 Design

The first module to build is the instruction memory module. The instruction memory module is designed to take one 64-bit input value and produce a 32-bit output value from a data file.

The second module to build is the iFetch module. The iFetch module needs to be the instruction memory module with the added functionality of automatically increasing the program counter value by 4, and the option to branch to a specified address.

4 Implementation

The instruction memory module is implemented with a 32-bit data reg called imem. in The initial block seen in Listing 1, imem is set equal to the data in Figure 1. Creating a vector of length 18. The output, instruction, is equal to the value of imem at the index of the pc input divided by 4. For example, if the pc input is 8, the instruction will output the binary code at line 2 from the data file.

Listing 1: Verilog code for implementing the instruction memory module.

```
`include "definitions.vh"

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

    reg[`INSTR_LEN - 1:0] imem [SIZE-1:0];
    integer i;

    //handle output
    always @(posedge(clk))
        instruction<= imem[pc/4];

    //initialize memory from file
    initial
        $readmemb(`IMEMFILE, imem);

endmodule
```

The iFetch module is implemented by instantiating the mux, register, adder, and instruction memory modules, and then connecting all the inputs and outputs as described in Interface. The mux is altered to have a WORD parameter and instruction memory is altered to have SIZE parameter. cur_pc does not have to be an output, but is set as one, so that we can see it in the timing diagram. The new_pc wire is not set as an output because we do not need to see its value.

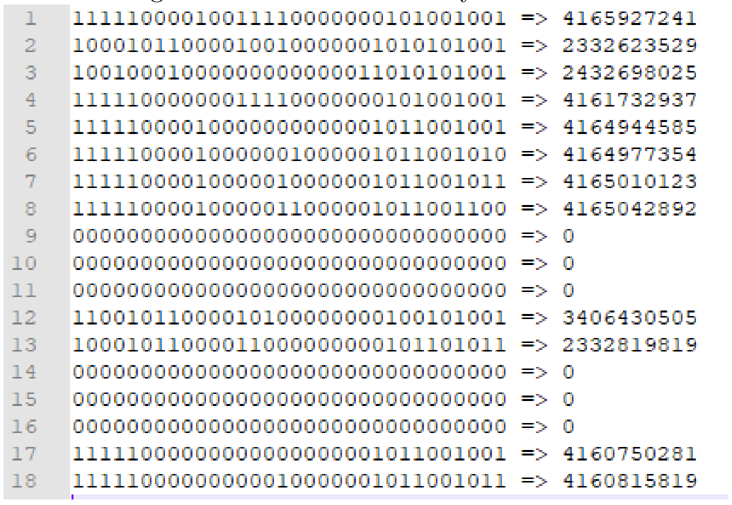
Listing 2: Verilog code for implementing the iFetch module.

```
`include "definitions.vh"

module iFetch#(parameter STEP=`WORD'd4, SIZE=1024)(
    input clk,
    input reset,
    input pc_src,
    input [`WORD-1:0] branch_target,
    output [`WORD-1:0] incremented_pc,
    output [`INSTR_LEN-1:0] instruction,
    output[`WORD-1:0] cur_pc
    );
    wire [`WORD-1:0] new_pc;

    mux#(`WORD) pc_mux(
    .a_in(incremented_pc),
    .b_in(branch_target),
    .control(pc_src),
    .mux_out(new_pc)
    );

    register pc_register(
    .clk(clk),
    .reset(reset),
    .D(new_pc),
    .Q(cur_pc)
    );

    adder incrementer(
    .a_in(cur_pc),
    .b_in(STEP),
    .add_out(incremented_pc)
    );

    instr_mem#(SIZE) instr_mem(
    .clk(clk),
    .pc(cur_pc),
    .instruction(instruction)
    );
endmodule
```

5 Test Bench Design

A test bench was made for each module. The instruction memory test bench creates a wire for the input clock (clk), and 32 bit wire for the instruction output (inst). A 64 bit reg is created for the program counter input (p_c). An oscillator module called clock_gen is instantiated. This generates a clock pulse on clk. An instruction memory module called UUT is then instantiated. In the initial block, the p_c is set from 0 to 52 with increments of 4 and delays of 1 cycle (10ns) in between each increment. This simulates the incrementation of the program counter by the adder in the fetch stage. The instruction data used for testing is shown in Figure 1, with the decimal values to the right of each binary value. Only the binary values were present at testing. The results of the test are verified if the output of the module (inst) is the value at the address specified by the p_c. For example, at p_c equal to 0, the output should be 4165927241 because this is the data at the 0 address of the test data file. All of the code used for the instruction memory testing is in Listing 3.

Figure 1: Instruction Memory Test Data.



```
11111000010011110000000101001001 => 4165927241
10001011000010010000001010101001 => 2332623529
10010001000000000000011010101001 => 2432698025
11111000000011110000000101001001 => 4161732937
11111000010000000000001011001001 => 4164944585
11111000010000001000001011001010 => 4164977354
11111000010000010000001011001011 => 4165010123
11111000010000011000001011001100 => 4165042892
00000000000000000000000000000000 => 0
00000000000000000000000000000000 => 0
00000000000000000000000000000000 => 0
11001011000010100000000100101001 => 3406430505
10001011000011000000000101101011 => 2332819819
00000000000000000000000000000000 => 0
00000000000000000000000000000000 => 0
00000000000000000000000000000000 => 0
11111000000000000000001011001001 => 4160750281
11111000000000010000001011001011 => 4160815819
```

Listing 3: Verilog code for testing the fetch stage.

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

module instr_mem_test;
wire clk;
reg[`WORD - 1:0] p_c;
wire [`INSTR_LEN-1:0] inst;

oscillator clk_gen(clk);
```

```
instr_mem UUT(
    .clk(clk),
    .pc(p_c),
    .instruction(inst)
    );

initial
begin
    p_c = 0;
    #`CYCLE;
    p_c = 4;
    #`CYCLE;
    p_c = 8;
    #`CYCLE;
    p_c = 12;
    #`CYCLE;
    p_c = 16;
    #`CYCLE;
    p_c = 20;
    #`CYCLE;
    p_c = 24;
    #`CYCLE;
    p_c = 28;
    #`CYCLE;
    p_c = 32;
    #`CYCLE;
    p_c = 36;
    #`CYCLE;
    p_c = 40;
    #`CYCLE;
    p_c = 44;
    #`CYCLE;
    p_c = 48;
    #`CYCLE;
    p_c = 52;
    #`CYCLE;



end
endmodule
```

Next, a test bench was created for the fetch module. This required the creation of a wire called clk, a rst and pc_s reg, a 64 bit branch reg, a 64 bit inc_pc wire, a 32 bit instr wire, and a 64 bit c_pc wire. Another oscillator module was instantiated, and then a iFetch module called UUT was also instantiated. To test functionality, first, the pc_s was set with a blocking statement to 0 in order to perform sequential stepping. The reset reg is set to 0 with a non-blocking statement. This set the program counter to an initial value of 0. Then a delay of 4 cycles occurs. With these three lines of code we can verify that the sequential fetching works correctly. In the simulation we should see the program counter start at 0 and increment by 4. As this is happening the respective instructions should be outputted (the same instruction data was used). After the 4 cycles are finished the pc_s is set to 1 and the branch is set to 4. A delay of 2 cycles is added. This is used to verify that the branching functionality works. We should see the program counter change to 4 at a positive clock edge, and then stay at 4 until the pc_s is set back to 0. When p_c is set back to 0, the program counter will resume stepping by 4 starting with the branch value. All of the code used for the iFetch testing is in Listing 4.

Listing 4: Verilog code for testing the fetch stage.

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

module iFetch_test;
    wire clk;
    reg rst;
    reg pc_s;
    reg [`WORD-1:0] branch;
    wire [`WORD-1:0] inc_pc;
    wire [`INSTR_LEN-1:0] instr;
    wire [`WORD-1:0] c_pc;

oscillator clk_gen(clk);

iFetch UUT(
    .clk(clk),
    .reset(rst),
    .pc_src(pc_s),
    .branch_target(branch),
    .incremented_pc(inc_pc),
    .instruction(instr),
    .cur_pc(c_pc)
    );

 initial begin
 pc_s = 0;
 rst <= 0;
 #(4*`CYCLE);
 pc_s = 1;
 branch <= 4;
 #(2*`CYCLE);
 pc_s  =0;

end
endmodule
```

6 Simulation

The timing diagrams in Figure 2 and 3 verify that both modules work as detailed in the above section. One way to easily verify functionality is to compare the outputs to the data of Figure 1.

Figure 2: Timing diagram for instruction module test.

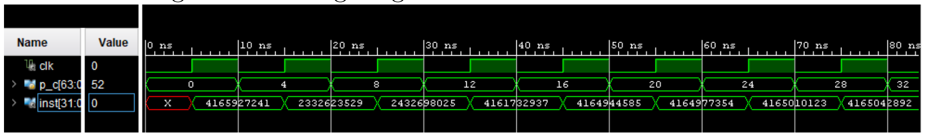
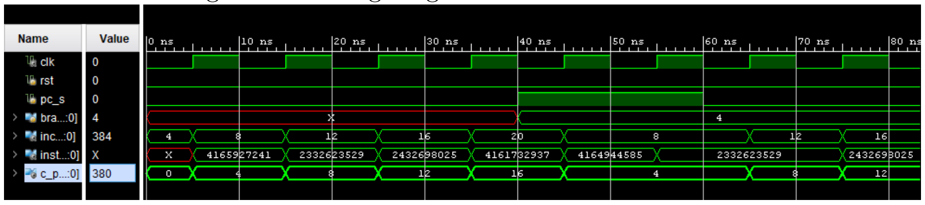


Figure 3: Timing diagram for fetch module test.



7 Conclusions

The instruction module and fetch module were successfully established. The instruction memory module is used in conjunction with the mux, register, and adder modules to create the fetch. With the fetch module, we can read and output data sequentially or through a branch.