Lab 3

Matthew Carrano and Breana Leal

January 29, 2018

1 Introduction

The goal of this lab is to create an instruction memory module and combine previous modules to form the fetch stage, iFetch. The instruction memory will be used to store instructions. The instruction memory reads in data from the program counter that requires addressing executable instructions. The fetch stage is the first stage in building the ARM computer that combines the previous labs. The fetch stage will be followed by the decode stage.

2 Interface

The inputs of the instruction memory module are clk and pc. The clk is initiated to follow a positive clock edge. With each cycle, pc will increment which should give a value for its output instruction to address from a memory file. The inputs of the iFetch are clk, reset, pc_src, and branch_target. The output of the iFetch is incremented_pc, instruction, and cur_pc. Its inputs and outputs serve as connecting pieces for the previously discussed modules mux, register, adder, and instruction memory.

incremented_pc and branch_target are the two signals that are available to be selected, and pc_src is the signal that is used to select whether the value of incremented_pc or branch_target will be chosen by the mux. new_pc is a wire that contains the value that the mux selected. The output from the mux becomes an input for the register along with inputs clk and reset. The output for the register becomes cur_pc. Similarly, cur_pc becomes an input for the adder module. The adder's other input is STEP, a parameter defined as 'WORD'd4 and its output is incremented_pc. Finally, the instru_mem module has inputs clk and cur_pc. The cur_pc is the current program counter value and notifies ARM about the current instruction address. The output instruction is the final data from the provided memory file.

As noted, the fetch stage connects the previous modules in having one's input become another's output and reiterate the process to continuously update the instruction results either sequentially or branching.

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 data from a readable file as an output.

The second module to build is the iFetch module. The iFetch module is designed to process five input values with four individually instantiated modules (discussed in Interface). Each module processes its own inputs to has its own output that work under the iFetch module to evaluate the final instruction value for the fetch stage to pass onto the decode stage.

4 Implementation

The instruction memory module is implemented by having a 32-bit data reg imem. The output instruction is set to equal the program counter divided by 4 indexed data. Meaning, the instruction should equal the adjusted 32 bit data read from the file. For example, if the program counter is set to 8, the instruction will out put the binary code, line 2, from the readable file. This is done continuously and updated with each input change.

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 instantiating the mux, register, adder, and instruction memory modules. The instantiations easily verify the connections the fetch stage makes.

mux is altered to have a WORD parameter and instruction memory SIZE parameter for future reference purposes.

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 section 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 ? 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.

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.

Figure 2. Timing diagram for instruction module test.

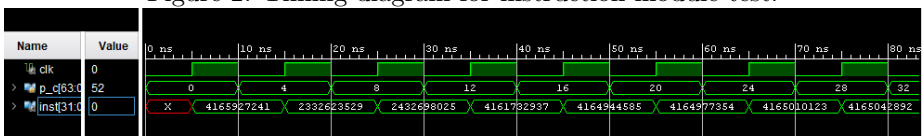
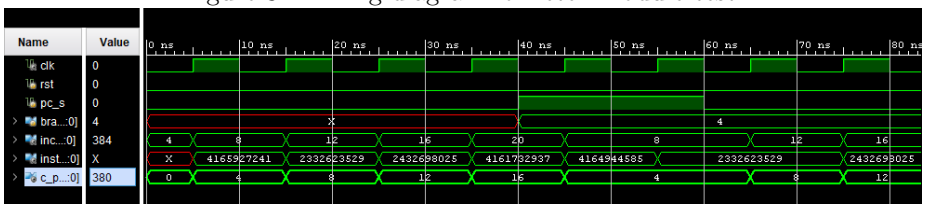


Figure 3. Timing diagram for fetch module test.



7 Conclusions

The instruction module and fetch stage were successfully established. The instruction memory module is used in conjunction with the mux, register, and adder modules. The instruction memory is established to gather data from a file based on addressed instruction values from the program counter. Combined, these modules form the first stage for the ARM-64 that will be further processes in the decoding stage.