# Lab 3 - Fetch Stage

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.

```
`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 Fetch Stage.

```
`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

The adder test bench creates regs for a and b as well as a wire for the add_out output of the adder module. All signals are 64-bits. An adder module, UUT, is instantiated, and then a series of a and b inputs are set in the initial section. Regs a and b are set using non-blocking procedural assignments, then a cycle delay is initiated. As defined in definitions.vh, a cycle is 10ns. Then a and b are changed several times, with various amounts of delay. The goal is to test a variety of inputs and outputs at various intervals. The results of the test are verified by evaluating the simulation results and verifying that add_out equals a + b at all times. View the adder_test code in Listing 3 on page 4.

Listing 3: Verilog code for testing the Instruction Memory.

```
`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
```

The mux test bench creates regs for a, b, and control as well as a wire for the mux_out output of the mux module. All signals are 64-bits. An adder module, UUT, is instantiated with a size of 64 by using the syntax of mux#(64) UUT. Then a series of a and b inputs are set in the initial section, and the control line is set to 0. a and b are varied with various amounts of delay. After several cycles, control is changed to 1, causing the value of b to be sent to mux_out. And then control is set back to 0 to verify that the mux can switch back and forth effectively. The results of the test are verified by evaluating the simulation results and verifying that:

1. mux_out always equals a when control is 0
2. mux_out always equals b when control is 1

View the mux_test code in Listing 4 on page 5.

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;
```

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

    oscillator clk_gen(clk);

 initial begin
 pc_s <= 0;
 rst <= 0;
 #(4*`CYCLE);
 pc_s <= 1;
 branch <= 4;
 #200;
 pc_s <=0;



end
endmodule
```

# 6 Simulation

The Simulation Results of the adder test confirm that the adder works as expected, with add_out always producing the sum of a and b. The timing diagram can be viewed in Figure 1 on page 7.

The Simulation Results of the mux test confirm that the mux works as expected, with mux_out always matching the conditions listed in the Test Bench Design section. The timing diagram can be viewed in Figure 2 on page 7.

# 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,

Figure 1: Timing diagram for the instruction memory test (First Half).

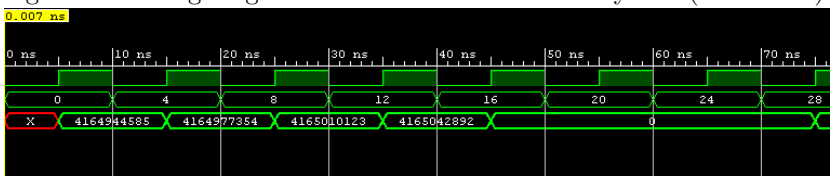
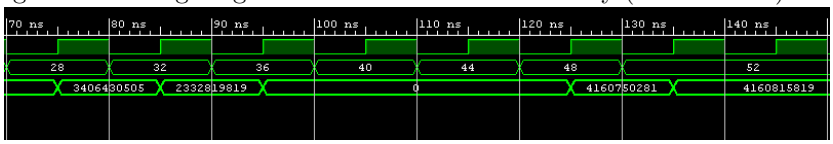



Figure 2: Timing diagram for the instruction memory (Second Half) test.




these modules form the first stage for the ARM-64 that will be further processes in the decoding stage.