Josh Monson
620 Verilog and System Verilog Assignment #1
Report

Test Strategy:

(Datapath Validation)

My test strategy was two-fold. First, I attempted to validate the datapath using a Questasim .do file (included in this document). The general idea here was to validate the register, multiplexors, functional units, and bus by using Questasim to drive inputs and enable signal and determine if the functional unit was operating correctly. Using this method I was able to find the four bugs listed below.

*** DataPath TestBench: Found Missing Register Always Blocks (MDR, MAR)
*** DataPath TestBench: Found Incorrect Driver (MDR)
*** DataPath TestBench: Found Missing Signals in Sensitivity List
*** DataPath TestBench: Found Swapped Inputs to SR2MUX

This verification method was time-consuming and did not yield many bugs. Additionally, it was unnecessary because the next test method covered the whole area this test covered and more. As I found during my control validation this method was fooled by an error in the testing code. This error was not corrected because the space covered by the datapath test was also covered by the control validation test.

(Control Validation)

The second validation step was to test the control state machine of the LC3. For this step, a new test bench was built that tracked expected register and memory updates and then monitored to ensure those updates were completed. Additionally, the test reported a failure it a register was updated when it was not expected to be. Instructions were randomly generated and fed into the LC3. The following registers were monitored and checked for correctness:

PC, MAR, MDR, IR, NZP, REGFILE, MEMORY

Using this method I was able to find the following 8 bugs:

*** Verilog TestBench: Found I wasn't asserting ldPC during Fetch to Increment PC
*** Verilog TestBench: Found Bad Control Signal to ALU in STORE0
*** Verilog TestBench: Found Bad Control Signal to MDRMUX in STORE0
*** Verilog TestBench: Found Bad NextState Signal to MDRMUX in STORE1
*** Verilog TestBench: Found Missing Connection From RA to ADDR1MUX (RB was Connected instead RB held value that fooled my test code)
*** Verilog TestBench: Found Undefined State in Control State Machine (state that didn't have a case in nextstate logic)
*** Verilog TestBench: Forgot to enable MDR to BUSS during LOAD
*** Verilog TestBench: Forgot to apply SR1 during STORE0

Overall, I injected 10,000 instructions into the LC3 implementation. This number could easily be increased if necessary. This mean that each instruction was executed roughly 600 times. For convenience, I have attached the test output report showing the number of times each instruction was executed.

Test Vectors Complete! Executed 10000 instructions!

BR: 654
ADD: 625
LD: 647
ST: 586
JSR: 612
AND: 628
LDR: 619

```
# STR:   616
# RTI:   616
# NOT:   605
# LDI:   585
# STI:   673
# JMP:   603
# RES:   638
# LEA:   663
# TRAP:   630
```

Console Print out of Successfully Verified Instruction (from Control test)

```
# ****Starting Fetch of Instr:  9999 @ time ****            960730
# Checking Register      MAR @ time:            960770
#    Success:      MAR was updated as expected
# Checking Register      MDR @ time:            960790
#    Success:      MDR was updated as expected
# Checking Register       PC @ time:            960790
#    Success:       PC was updated as expected
# Checking Register       IR @ time:            960810
#    Success:       IR was updated as expected
# Fetch of Instr  9999 Complete
# ****Starting Execution of Instr:  9999 @ time:            960810 ****
# OPCODE: 35087
#  Instr  9999: RTI
# executeUpdateState_i:    0
# 0:Execution of Instr  9999 Complete @ time:            960810
```

Discussion of System Verilog Features Used:

1. Change to wire and reg to logic type. All changed but BUSS which had to be a wire to permit multiple drivers.
2. Added a Package to share State Machine and Opcode ENUMs with testbench.
3. Used typedef to Make State Machine values their own type
4. Added always_ff and always_comb to make my intent more clear.

5. Added Unique Case to allow case statements to execute in parallel.
6. default_nettype none

7. Attempted to add Class Data Structures to Simplify Testbench

SystemVerilog changes 1,3, and 4 were made to more clearly describe the intent of Verilog code. In addition, the always_comb block offered some piece of mind as well as it remove the requirement that I identify all the signals needed for the sensitivity list.

Change 2 and 7 allowed me to use a programming style that was closer to a high-level programming language. For example, the SystemVerilog packages allowed me to centrally locate the OPCODE enum so that the testbench could use it as well. I started trying to make my testbench more readable to implementing it using classes. The classes simplified a lot of code by allowing me to centrally locate and operate on a lot of test bench state that before was individually declared and operated on; bloating the size of my testbench and making it more difficult to handle. Unfortunately, I ran out of time to completely implement the classes but I have included the system Verilog testbench file to show what I accomplished. The other system Verilog changes were completed and correctly verify the testbench. I have included this file as well.

The unique case (5) is supported to improve the final circuit by removing priority encoded muxes. All case statements in the control and datapath were changed to implement these case statements. The default_nettypte none was used to prevent the undeclared signals from causing any problems.

Code Listings:

Contents:

1. Verilog Implementation of LC3
2. Verilog Testbench
3. Test Datapath “do” file
4. SystemVerilog revised code and Testbench
5. Partial SystemVerilog testbench with Classes

lc3.v **********

```
`default_nettype none
module lc3(clk,
           rst,
           memory_dout,
           memory_addr,
           memory_din,
          memWE);

input wire clk;
input wire rst;

input wire [15:0] memory_dout;

output wire [15:0] memory_addr;
output wire [15:0] memory_din;
output wire memWE;

wire [15:0] IR;
wire N;
wire Z;
wire P;
wire [1:0] aluControl;
wire enaALU;
wire [2:0] SR1;
wire [2:0] SR2;
wire [2:0] DR;
wire regWE;
wire [1:0] selPC;
wire enaMARM;
wire selMAR;
wire selEAB1;
wire [1:0] selEAB2;
wire enaPC;
wire ldPC;
wire ldIR;
wire ldMAR;
wire ldMDR;
wire selMDR;

wire flagWE;
wire enaMDR;

lc3_datapath DATAPATH( clk, rst,
                    IR, N, Z, P,
                    aluControl, enaALU, SR1, SR2,
                    DR, regWE, selPC, enaMARM, selMAR,
                 selEAB1, selEAB2, enaPC, ldPC, ldIR,
                   ldMAR, ldMDR, selMDR, flagWE, enaMDR,
                    memory_din, memory_dout, memory_addr);
lc3_control CONTROL( clk, rst, IR, N, Z, P,
                    aluControl, enaALU, SR1, SR2,
                    DR, regWE, selPC, enaMARM, selMAR,
                 selEAB1, selEAB2, enaPC, ldPC, ldIR,
                   ldMAR, ldMDR, selMDR, memWE, flagWE, enaMDR);
endmodule
```

lc3_datapath.v **********

```
`default_nettype none
module lc3_datapath ( clk, rst,
                      IR_OUT, N_OUT, Z_OUT, P_OUT,
                      aluControl, enaALU, SR1, SR2,
                      DR, regWE, selPC, enaMARM, selMAR,
                  selEAB1, selEAB2, enaPC, ldPC, ldIR,
                    ldMAR, ldMDR, selMDR, flagWE, enaMDR,
                      memory_din, memory_dout, memory_addr);
input wire clk;
input wire rst;

input wire [1:0] aluControl;
input wire enaALU;
input wire [2:0] SR1;
input wire [2:0] SR2;
input wire [2:0] DR;
input wire regWE;
input wire [1:0] selPC;
input wire enaMARM;
input wire selMAR;
input wire selEAB1;
input wire [1:0] selEAB2;
input wire enaPC;
input wire ldPC;
input wire ldIR;
input wire ldMAR;
input wire ldMDR;
input wire selMDR;
input wire flagWE;
input wire enaMDR;

input wire [15:0] memory_dout;

output wire [15:0] IR_OUT;
output wire N_OUT;
output wire Z_OUT;
output wire P_OUT;

output wire [15:0] memory_din;
output wire [15:0] memory_addr;


//Datapath Registers
reg [15:0] PC;
reg [15:0] IR;
reg [15:0] MAR;
reg [15:0] MDR;
reg N, Z, P;
reg [15:0] REGFILE [0:7];

//reg [15:0] MEMORY [0:255];

wire [15:0] BUSS;

//Multiplexors
reg [15:0] PCMUX;
wire [15:0] MARMUX;
wire [15:0] MDRMUX;
wire [15:0] ADDR1MUX;
reg [15:0] ADDR2MUX;
wire [15:0] SR2MUX;

//Arithmetic Units
wire [15:0] ADDER;
wire [15:0] PCINCR;
```

```
reg [15:0] ALU;
wire [15:0] ZERO16;
reg [15:0] i;
//Register File Outputs
wire [15:0] RA;
wire [15:0] RB;

//IR Sign Extension
wire [15:0] SEXT4;
wire [15:0] SEXT5;
wire [15:0] SEXT8;
wire [15:0] SEXT10;
wire [15:0] ZEXT;

wire [15:0] memOut;

assign IR_OUT = IR;
assign N_OUT = N;
assign Z_OUT = Z;
assign P_OUT = P;

assign memOut = memory_dout;
assign memory_din = MDR;
assign memory_addr = MAR;

/***********************************
 Program Counter
***********************************/
always @ (posedge clk) begin
  if (rst == 1'b1) begin
    PC <= 16'd0;
  end else if(ldPC) begin
    PC = PCMUX;
  end
end

/***********************************
 Program Counter MUX
***********************************/

assign PCINCR = PC + 16'd1;

always @ (selPC or PCINCR or ADDER or BUSS)
  case(selPC)
    2'b00: PCMUX = PCINCR;
    2'b01: PCMUX = ADDER;
    2'b10: PCMUX = BUSS;
    default: $display("PCMUX ERROR: Illegal Select Signal ");
  endcase


/***********************************
 Memory
***********************************/

//always @(posedge clk)
//begin
//  if (memWE)
//    MEMORY[MAR] <= MDR;
//end

//assign memOut = MEMORY[MAR];

/***********************************
 MAR
***********************************/
```

```
always @ (posedge clk or rst) begin
 if(rst) begin
   MAR <= 16'd0;
 end else if(ldMAR == 1'b1) begin
   MAR <= BUSS;
 end
end
/***********************************
 MDR
***********************************/

always @ (posedge clk or rst) begin
 if(rst) begin
   MDR <= 16'd0;
 end else if(ldMDR == 1'b1) begin
   MDR <= MDRMUX;
 end
end

/***********************************
 Instruction Register
***********************************/

always @ (posedge clk or rst) begin
  if(rst) begin
    IR <= 16'd0;
  end else if(ldIR == 1'b1) begin
    IR <= BUSS;
  end
end

/***********************************
 Instruction Register Sign Extend
***********************************/

assign SEXT4 = { {11{IR[4]}}, IR[4:0] };
assign SEXT5 = { {10{IR[5]}}, IR[5:0] };
assign SEXT8 = { {7{IR[8]}}, IR[8:0] };
assign SEXT10 = { {5{IR[10]}}, IR[10:0] };
assign ZEXT =   { 8'b0, IR[7:0] };

/***********************************
 MARMUX
***********************************/

assign MARMUX = (selMAR) ? ZEXT : ADDER;

/***********************************
 ADDER
***********************************/

assign ADDER = ADDR2MUX + ADDR1MUX;

/***********************************
 ADDR1MUX
***********************************/

assign ADDR1MUX = (selEAB1) ? RA : PC;

/***********************************
 ADDR2MUX
***********************************/
assign ZERO16 = 16'h0000;

always @ (selEAB2 or ZERO16 or SEXT5 or SEXT8 or SEXT10)
  case (selEAB2)
    2'b00: ADDR2MUX = ZERO16;
```

```
    2'b01: ADDR2MUX = SEXT5;
    2'b10: ADDR2MUX = SEXT8;
    2'b11: ADDR2MUX = SEXT10;
  endcase

/***********************************
 SR2MUX
***********************************/
assign SR2MUX = (IR[5]) ? SEXT4 : RB;
/***********************************
 MDRMUX
***********************************/

assign MDRMUX = (selMDR) ? memOut : BUSS;

/***********************************
 ALU
***********************************/

always @ (aluControl, SR2MUX, RA)
  case(aluControl)
    2'b00: ALU = RA;
    2'b01: ALU = RA + SR2MUX;
    2'b10: ALU = RA & SR2MUX;
    2'b11: ALU = ~RA;
  endcase

/***********************************
 NZP Logic
***********************************/

always @ (posedge clk) begin
  if(BUSS[15] == 1'b1) begin
    N <= 1'b1; Z <= 1'b0; P <= 1'b0;
  end else if(BUSS == 16'h0000) begin
    N <= 1'b0; Z <= 1'b1; P <= 1'b0;
  end else begin
    N <= 1'b0; Z <= 1'b0; P <= 1'b1;
  end
end

/***********************************
 Register File
***********************************/

always @ (posedge clk or posedge rst) begin
  if(rst) begin
    for(i=0; i<16; i = i + 1) begin
      REGFILE[i] <= 16'd0;
    end
  end else if(regWE) begin
    REGFILE[DR] <= BUSS;
  end
end
assign RA = REGFILE[SR1];
assign RB = REGFILE[SR2];

/***********************************
 BUSS
***********************************/

assign BUSS = (enaMARM) ? MARMUX : 16'hZZZZ;
assign BUSS = (enaPC) ? PC : 16'hZZZZ;
assign BUSS = (enaALU) ? ALU : 16'hZZZZ;
assign BUSS = (enaMDR) ? MDR : 16'hZZZZ;

endmodule
```

lc3_control.v **********

```
`default_nettype none
module lc3_control ( clk, rst,
                     IR, N, Z, P,
                     aluControl, enaALU, SR1, SR2,
                     DR, regWE, selPC, enaMARM, selMAR,
                  selEAB1, selEAB2, enaPC, ldPC, ldIR,
                    ldMAR, ldMDR, selMDR, memWE, flagWE, enaMDR);

input wire clk;
input wire rst;

input wire [15:0] IR;
input wire N;
input wire Z;
input wire P;

//Output
output reg [1:0] aluControl = 2'b00;
output reg [2:0] SR1 = 3'b000;
output reg [2:0] SR2 = 3'b000;
output reg [2:0] DR = 3'b000;

output reg enaALU = 1'b0;
output reg enaPC = 1'b0;
output reg enaMDR = 1'b0;
output reg enaMARM = 1'b0;

output reg [1:0] selPC = 2'b00;
output reg selMAR = 1'b0;
output reg selEAB1 = 1'b0;
output reg [1:0] selEAB2 = 2'b00;
output reg selMDR = 1'b0;

output reg ldPC = 1'b0;
output reg ldIR = 1'b0;
output reg ldMAR = 1'b0;
output reg ldMDR = 1'b0;

output reg memWE = 1'b0;
output reg flagWE = 1'b0;
output reg regWE = 1'b0;


reg [4:0] CurrentState;
reg [4:0] NextState;
wire branch_enable;

parameter FETCH0=5'd0,
          FETCH1=5'd1,
          FETCH2=5'd2,
          DECODE=5'd3,
          BRANCH0=5'd4,
          ADD0=5'd5,
          STORE0=5'd7,
          STORE1=5'd8,
          STORE2=5'd9,
          JSR0=5'd10,
          JSR1=5'd11,
          AND0=5'd12,
          NOT0=5'd13,
          JMP0=5'd14,
          LD0=5'd15,
          LD1=5'd16,
          LD2=5'd17;
```

```
parameter BR=4'b0000, ADD=4'b0001, LD=4'b0010, ST=4'b0011,
          JSR=4'b0100, AND=4'b0101, LDR=4'b0110, STR=4'b0111,
          RTI=4'b1000, NOT=4'b1001, LDI=4'b1010, STI=4'b1011,
          JMP=4'b1100, RES=4'b1101, LEA=4'b1110, TRAP=4'b1111;


assign  branch_enable = ((N == IR[11]) || (Z == IR[10]) || (P == IR[9])) ? 1'b1 : 1'b0;

always @ (posedge clk or posedge rst) begin
 if(rst)
  CurrentState <= FETCH0;
 else
  CurrentState <= NextState;
end

always @ (CurrentState or IR or
          N or Z or P or branch_enable) begin
  //Tristate Signals
  enaALU <= 1'b0; enaMARM <= 1'b0;
  enaPC <= 1'b0; enaMDR <= 1'b0;

  //Register Load Signals
  ldPC <= 1'b0; ldIR <= 1'b0;
  ldMAR <= 1'b0; ldMDR <= 1'b0;

  //MUX Select Signal
  selPC <= 2'b00; selMAR <= 1'b0;
  selEAB1 <= 1'b0; selEAB2 <= 2'b00;
  selMDR <= 1'b0;

  //Write Enable Signals
  flagWE <= 1'b0;
  memWE <= 1'b0;
  regWE <= 1'b0;

  //Control Signals
  aluControl <= 2'b00;
  SR1 <= 3'b000;
  SR2 <= 3'b000;
  DR <= 3'b000;

  case (CurrentState)
    FETCH0: begin
      NextState <= FETCH1;
      //Load PC ADDRESS
      enaPC <= 1'b1; ldMAR <= 1'b1;
    end
    FETCH1: begin
      NextState <= FETCH2;
      //READ Instruction From Memory
      selMDR<=1'b1; ldMDR<=1'b1;
      //Incremente Program Counter
      selPC<=2'b00;
      ldPC<=1'b1;
    end
    FETCH2: begin
      NextState <= DECODE;
      //Load Instruction Register
      enaMDR <= 1'b1; ldIR <= 1'b1;
    end
    DECODE: begin
      case (IR[15:12])  //AND, ADD, NOT, JSR, BR, LD, ST, JMP.
        BR:  NextState <= BRANCH0;//**//
        ADD: NextState <= ADD0;   //**//
        LD:  NextState <= LD0;  //**//
        ST:  NextState <= STORE0; //**//
        JSR: NextState <= JSR0;   //**//
```

```
    AND: NextState <= AND0;   //**//
    LDR: NextState <= FETCH0;
    STR: NextState <= FETCH0;
    RTI: NextState <= FETCH0;
    NOT: NextState <= NOT0;   //**//
    LDI: NextState <= FETCH0;
    STI: NextState <= FETCH0;
    JMP: NextState <= JMP0;   //
    RES: NextState <=  FETCH0;
    LEA:  NextState <= FETCH0;
    TRAP: NextState <= FETCH0;
   endcase
end
BRANCH0:  begin
 //Select ADDER inputs
 selEAB1 <= 1'b0;
 selEAB2 <= 2'b10;
 //Load the New PC Value (if Branch Condition Met)
 ldPC <= branch_enable;
 selPC <= 2'b01;
 NextState <= FETCH0;
end
LD0: begin
 //Load MAR
 selEAB2 <= 2'b10;
 selEAB1 <= 1'b0;
 selMAR <= 1'b0;
 enaMARM <= 1'b1;
 ldMAR <= 1'b1;
 NextState <= LD1;
end
LD1: begin
 //Load MDR
 selMDR <= 1'b1;
 ldMDR <= 1'b1;

 NextState <= LD2;
end
LD2: begin
 //Write to Register File
 DR <= IR[11:9];
 regWE <= 1'b1;
 enaMDR <= 1'b1;
 NextState <= FETCH0;
end
NOT0: begin
 aluControl <= 2'b11;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
 DR <= IR[11:9];
 regWE <= 1'b1;
 NextState <= FETCH0;
end
ADD0: begin
 aluControl <= 2'b01;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
 SR2 <= IR[2:0];
 DR <= IR[11:9];
 regWE <= 1'b1;
 flagWE <= 1'b1;
 NextState <= FETCH0;
end
AND0: begin
 aluControl <= 2'b10;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
```

```
     SR2 <= IR[2:0];
     DR <= IR[11:9];
     regWE <= 1'b1;
     NextState <= FETCH0;
    end
    STORE0: begin
     //Load the MDR
     SR1 <= IR[11:9];
     aluControl <= 2'b00;
     enaALU <= 1'b1;
     selMDR <= 1'b0;
     ldMDR <= 1'b1;
     NextState <= STORE1;
    end
    STORE1: begin
     //Load the MAR
     selEAB1 <= 1'b0;
     selEAB2 <= 2'b10;
     selMAR <= 1'b0;
     enaMARM <= 1'b1;
     ldMAR <= 1'b1;
     NextState <= STORE2;
    end
    STORE2: begin
     memWE <= 1'b1;
     NextState <= FETCH0;
   end
   JSR0: begin
     DR <= 3'b111;
     enaPC <= 1'b1;
     regWE <= 1'b1;
     NextState <= JSR1;
   end
   JSR1: begin
     selEAB1 <= 1'b0;
     selEAB2 <= 2'b11;
     selPC <= 2'b01;
     ldPC <= 1'b1;
     NextState <= FETCH0;
   end
   JMP0: begin
     SR1 <= IR[8:6];
     selEAB1 <= 1'b1;
     selEAB2 <= 2'b00;
     selPC <= 2'b01;
     ldPC <= 1'b1;
     NextState <= FETCH0;
   end
  endcase
end


endmodule
```

lc3_testbench.v **********

```
`default_nettype none
`timescale 1ns/100ps
`define TBSTATE_FETCH 1'b0
`define TBSTATE_EXECUTE 1'b1
`define INSTRUCTIONS_TO_EXECUTE 16'd10000
module lc3_testbench ();

  event found_error;
  event evaluate_executeUpdateStatus;
  function [16:0] ValidateRegisterUpdate;
   input [63:0] regName;
   input [15:0] currentValue, expectedValue;
   input update;
   reg [16:0] ret;
   begin
     $display("Checking Register %s @ time: %d", regName, $time);
     if(update == 1'b0) begin
        $display("   Error: %s is not scheduled for Update!", regName);
        -> found_error;
        ret = 1;
     end else if(currentValue == expectedValue) begin
        //Register Update Correctly
        $display("   Success: %s was updated as expected", regName);
        ret = 0;
     end else begin
        $display("   Error: %s was %d expected %d", regName, currentValue, expectedValue);
        ->found_error;
        ret = 1;
     end
     ValidateRegisterUpdate = ret;
   end
  endfunction

  function [2:0] calc_nzp;
    input reg [15:0] buss;
    reg [2:0] nzp;
    begin

      if(buss[15] == 1'b1)
        nzp = 3'b100;
      else if (buss == 15'd0)
        nzp = 3'b010;
      else
        nzp = 3'b001;
      $display("CALC NZP: %d", nzp);
      calc_nzp = nzp;
    end
  endfunction

  parameter BR=4'b0000, ADD=4'b0001, LD=4'b0010, ST=4'b0011,
          JSR=4'b0100, AND=4'b0101, LDR=4'b0110, STR=4'b0111,
          RTI=4'b1000, NOT=4'b1001, LDI=4'b1010, STI=4'b1011,
          JMP=4'b1100, RES=4'b1101, LEA=4'b1110, TRAP=4'b1111;

  reg [15:0] regNamePC = "PC";

  reg clk, rst;

  reg [15:0] memory_dout;
  wire [15:0] memory_addr;
  wire [15:0] memory_din;
  wire memWE;

  event start_fetch;
  event start_execute;
```

```
reg rst_done = 1'b0;
reg tbState = `TBSTATE_FETCH;

reg [15:0] NEXT_MEMORY_OUTPUT;

reg UPDATE_MAR = 1'b0;
reg [15:0] UPDATE_MAR_VALUE;

reg UPDATE_MDR = 1'b0;
reg [15:0] UPDATE_MDR_VALUE;

reg UPDATE_PC = 1'b0;
reg [15:0] UPDATE_PC_VALUE;

reg UPDATE_IR = 1'b0;
reg [15:0] UPDATE_IR_VALUE;

reg UPDATE_MEMORY = 1'b0;
reg [15:0] UPDATE_MEMORY_VALUE;

reg UPDATE_NZP = 1'b0;
reg [15:0] UPDATE_NZP_VALUE;

reg [15:0] InstrCount [15:0];
reg [15:0] i;

reg UPDATE_REG0 = 1'b0;
reg UPDATE_REG1 = 1'b0;
reg UPDATE_REG2 = 1'b0;
reg UPDATE_REG3 = 1'b0;
reg UPDATE_REG4 = 1'b0;
reg UPDATE_REG5 = 1'b0;
reg UPDATE_REG6 = 1'b0;
reg UPDATE_REG7 = 1'b0;

reg [15:0] UPDATE_REG0_VALUE;
reg [15:0] UPDATE_REG1_VALUE;
reg [15:0] UPDATE_REG2_VALUE;
reg [15:0] UPDATE_REG3_VALUE;
reg [15:0] UPDATE_REG4_VALUE;
reg [15:0] UPDATE_REG5_VALUE;
reg [15:0] UPDATE_REG6_VALUE;
reg [15:0] UPDATE_REG7_VALUE;

wire [3:0] fetchUpdateStatus;
wire [11:0] executeUpdateStatus;
reg [11:0] executeUpdateStatus_i;
reg [15:0] instructionCount;

reg [15:0] SEXT10, SEXT8, SEXT4, STOREVAL;
reg [15:0] ADDER;
reg [15:0] OP1, OP2;
reg [2:0] DST, SRC, SRC1, SRC2;
reg [2:0] DSTREG;
reg [15:0] NZPTMP;
task setDstRegUpdate;
  input [2:0] dst;
  input [15:0] val;
  begin
    $display("Setting Reg %d with value %d for expected update", dst, val);
    case (dst)
      3'd0: begin
        UPDATE_REG0 = 1'b1;
        UPDATE_REG0_VALUE = val;
      end
      3'd1: begin
```

```
        UPDATE_REG1 = 1'b1;
        UPDATE_REG1_VALUE = val;
      end
      3'd2: begin
        UPDATE_REG2 = 1'b1;
        UPDATE_REG2_VALUE = val;
      end
      3'd3: begin
        UPDATE_REG3 = 1'b1;
        UPDATE_REG3_VALUE = val;
      end
      3'd4: begin
        UPDATE_REG4 = 1'b1;
        UPDATE_REG4_VALUE = val;
      end
      3'd5: begin
        UPDATE_REG5 = 1'b1;
        UPDATE_REG5_VALUE = val;
      end
      3'd6: begin
        UPDATE_REG6 = 1'b1;
        UPDATE_REG6_VALUE = val;
      end
      3'd7: begin
        UPDATE_REG7 = 1'b1;
        UPDATE_REG7_VALUE = val;
      end
    endcase
  end
endtask

assign fetchUpdateStatus = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_IR };
assign executeUpdateStatus = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_MEMORY,
                               UPDATE_REG0, UPDATE_REG1, UPDATE_REG2, UPDATE_REG3,
                               UPDATE_REG4, UPDATE_REG5, UPDATE_REG6, UPDATE_REG7 };
always begin
  #10 clk = !clk;
end

initial begin
  clk = 0;
  rst = 0;
  for (i=0; i<16; i = i + 1) begin
    InstrCount[i] = 16'd0;
  end
  instructionCount = 0;
  NEXT_MEMORY_OUTPUT = 16'd0;
  @ (negedge clk)
  rst = 1;
  @ (negedge clk)
  @ (negedge clk)
  @ (negedge clk)
  rst = 0;
  rst_done = 1'b1;
  ->start_fetch;
end

initial begin
  @ (found_error)
  $display("Simulation Terminated Due to Error");
  $finish();
end
initial begin
  forever begin
    @ (start_fetch)
    tbState <= `TBSTATE_FETCH;
    $display("****Starting Fetch of Instr: %d @ time ****", instructionCount, $time);
```

```
    //Schdule Register Updates
    UPDATE_MAR = 1'b1;
    UPDATE_MAR_VALUE = LC3.DATAPATH.PC;
    UPDATE_PC = 1'b1;
    UPDATE_PC_VALUE = LC3.DATAPATH.PC + 1;
    UPDATE_IR = 1'b1;
    UPDATE_IR_VALUE = $random;
    //$display("FET:IR_VALUE: %d", UPDATE_IR_VALUE);
    NEXT_MEMORY_OUTPUT = UPDATE_IR_VALUE;
    //$display("FET:NEXT_MEM: %d", NEXT_MEMORY_OUTPUT);
    UPDATE_MDR = 1'b1;
    UPDATE_MDR_VALUE = NEXT_MEMORY_OUTPUT;
  end
end

initial begin
  forever begin
    @ (start_execute);
    tbState = `TBSTATE_EXECUTE;
    $display("****Starting Execution of Instr: %d @ time: %d ****", instructionCount, $time);
    DST = UPDATE_IR_VALUE[11:9];
    SRC = UPDATE_IR_VALUE[11:9];
    SRC1 = UPDATE_IR_VALUE[8:6];
    SRC2 = UPDATE_IR_VALUE[2:0];
    SEXT4 = { {11{UPDATE_IR_VALUE[4]}}, UPDATE_IR_VALUE[4:0] };
    SEXT8 = { {7{UPDATE_IR_VALUE[8]}}, UPDATE_IR_VALUE[8:0] };
    SEXT10 = { {5{UPDATE_IR_VALUE[10]}}, UPDATE_IR_VALUE[10:0] };
    STOREVAL = LC3.DATAPATH.REGFILE[SRC];
    OP1 =  LC3.DATAPATH.REGFILE[SRC1];
    OP2 =  LC3.DATAPATH.REGFILE[SRC2];
    InstrCount[UPDATE_IR_VALUE[15:12]] = InstrCount[UPDATE_IR_VALUE[15:12]] + 1;
    //Schedule any Register Updates
    $display("OPCODE: %d", UPDATE_IR_VALUE);
    case (UPDATE_IR_VALUE[15:12])  //AND, ADD, NOT, JSR, BR, LD, ST, JMP.
      BR: begin //**//
       $display(" Instr %d: BR", instructionCount);
       if(UPDATE_IR_VALUE[11] == LC3.DATAPATH.N ||
          UPDATE_IR_VALUE[10] == LC3.DATAPATH.Z ||
          UPDATE_IR_VALUE[9]  == LC3.DATAPATH.P) begin
          //Update Program Counter
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = UPDATE_PC_VALUE + SEXT8;
       end
      end
      ADD: begin    //**//
         $display(" Instr %d: ADD", instructionCount);
         if(UPDATE_IR_VALUE[5] == 1'b1) begin
           OP2 = SEXT4;
         end
         ADDER = OP1 + OP2;
         setDstRegUpdate(DST, ADDER);
         UPDATE_NZP = 1'b1;
         UPDATE_NZP_VALUE = { 12'd0, calc_nzp(ADDER)};
         $display("UPDATE_NZP: %d", UPDATE_NZP_VALUE);
      end
      LD: begin //**//
        $display(" Instr %d: LD", instructionCount);
        UPDATE_MAR = 1'b1;
        UPDATE_MAR_VALUE = UPDATE_PC_VALUE + SEXT8;
        UPDATE_MDR = 1'b1;
        NEXT_MEMORY_OUTPUT = $random;
        UPDATE_MDR_VALUE =  NEXT_MEMORY_OUTPUT;
        setDstRegUpdate(DST, UPDATE_MDR_VALUE);
        UPDATE_NZP = 1'b1;
        UPDATE_NZP_VALUE = { 12'd0, calc_nzp(NEXT_MEMORY_OUTPUT)};
      end
      ST:  begin //**//
```

```
          $display(" Instr %d: ST SRC: %d ", instructionCount, SRC);
          UPDATE_MAR = 1'b1;
          UPDATE_MAR_VALUE = UPDATE_PC_VALUE + SEXT8;
          UPDATE_MDR = 1'b1;
          UPDATE_MDR_VALUE = STOREVAL;
          UPDATE_MEMORY = 1'b1;
          UPDATE_MEMORY_VALUE = STOREVAL;
        end
        JSR: begin   //**//
          $display(" Instr %d: JSR", instructionCount);
          setDstRegUpdate(3'd7, UPDATE_PC_VALUE);
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = UPDATE_PC_VALUE + SEXT10;
        end
        AND: begin    //**//
          $display(" Instr %d: AND DST: %d SRC1: %d SRC2: %d", instructionCount, DST, SRC1,
SRC2);
           if(UPDATE_IR_VALUE[5] == 1'b1) begin
             $display(" Instr %d: AND DST: %d SRC1: %d IMMED: %d", instructionCount, DST, SRC1,
SEXT4);
             OP2 = SEXT4;
           end
           ADDER = OP1 & OP2;
           setDstRegUpdate(DST, ADDER);
           UPDATE_NZP = 1'b1;
           UPDATE_NZP_VALUE = { 12'd0, calc_nzp(ADDER)};
        end
        LDR: $display(" Instr %d: LDR", instructionCount);
        STR: $display(" Instr %d: STR", instructionCount);
        RTI: $display(" Instr %d: RTI", instructionCount);
        NOT: begin  //**//
           $display(" Instr %d: NOT", instructionCount);
           ADDER = ~OP1;
           setDstRegUpdate(DST, ADDER);
           UPDATE_NZP = 1'b1;
           UPDATE_NZP_VALUE = { 12'd0, calc_nzp(ADDER)};
        end
        LDI:  $display(" Instr %d: LDI", instructionCount);
        STI:  $display(" Instr %d: STI", instructionCount);
        JMP:  begin //
          $display(" Instr %d: JMP BASE: %d", instructionCount, SRC1);
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = OP1;
        end
        RES:  $display(" Instr %d: RES", instructionCount);
        LEA:  $display(" Instr %d: LEA", instructionCount);
        TRAP:  $display(" Instr %d: TRAP", instructionCount);
      endcase

      executeUpdateStatus_i = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_MEMORY,
                                UPDATE_REG0, UPDATE_REG1, UPDATE_REG2, UPDATE_REG3,
                                UPDATE_REG4, UPDATE_REG5, UPDATE_REG6, UPDATE_REG7 };
      $display("executeUpdateState_i: %d", executeUpdateStatus_i);
      if(executeUpdateStatus_i == 12'd0) begin
        $display("0:Execution of Instr %d Complete @ time: %d", instructionCount, $time);
        instructionCount = instructionCount + 1;
        -> start_fetch;
      end
    end
  end

  //Are we done with FETCH?
  always @ (fetchUpdateStatus) begin
     //$display("fetchUpdateStatus: %x", fetchUpdateStatus);
     if ( fetchUpdateStatus == 4'b0000 && tbState == `TBSTATE_FETCH && rst_done) begin
       $display("Fetch of Instr %d Complete", instructionCount);
       -> start_execute;
```

```
    end
  end

  //Are we done with FETCH?
  always @ (executeUpdateStatus) begin
     //$display("executeUpdateStatus: %x", executeUpdateStatus);
     if ( executeUpdateStatus == 12'd0 && tbState == `TBSTATE_EXECUTE && rst_done) begin
       $display("1: Execution of Instr %d Complete", instructionCount);
       instructionCount = instructionCount + 1;
       -> start_fetch;
     end
  end

  /*************************************/
  /*  Processes to Monitor State Changes  */
  /*************************************/

  always @ (negedge LC3.DATAPATH.ldPC) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_PC <= ValidateRegisterUpdate("PC", LC3.DATAPATH.PC, UPDATE_PC_VALUE, UPDATE_PC);
   end
  end

  always @ (negedge LC3.DATAPATH.ldMAR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_MAR <= ValidateRegisterUpdate("MAR", LC3.DATAPATH.MAR, UPDATE_MAR_VALUE, UPDATE_MAR);
   end
  end

  always @ (negedge LC3.DATAPATH.ldMDR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_MDR <= ValidateRegisterUpdate("MDR", LC3.DATAPATH.MDR, UPDATE_MDR_VALUE, UPDATE_MDR);
   end
  end

  always @ (negedge LC3.DATAPATH.ldIR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_IR <= ValidateRegisterUpdate("IR", LC3.DATAPATH.IR, UPDATE_IR_VALUE, UPDATE_IR);
   end
  end

  always @ (negedge LC3.DATAPATH.flagWE) begin
   if(rst == 1'b0 && rst_done) begin
    NZPTMP = { 12'd0, LC3.DATAPATH.N, LC3.DATAPATH.Z, LC3.DATAPATH.P};
    UPDATE_NZP <= ValidateRegisterUpdate("NZP", NZPTMP, UPDATE_NZP_VALUE, UPDATE_NZP);
   end
  end

  always @ (posedge LC3.DATAPATH.regWE) begin
   DSTREG = LC3.DATAPATH.DR;
   @ ( negedge LC3.DATAPATH.regWE )
   if(rst == 1'b0 && rst_done ) begin
    case (DSTREG)
      3'd0: UPDATE_REG0 <= ValidateRegisterUpdate("REG0", LC3.DATAPATH.REGFILE[0],
UPDATE_REG0_VALUE, UPDATE_REG0);
      3'd1: UPDATE_REG1 <= ValidateRegisterUpdate("REG1", LC3.DATAPATH.REGFILE[1],
UPDATE_REG1_VALUE, UPDATE_REG1);
      3'd2: UPDATE_REG2 <= ValidateRegisterUpdate("REG2", LC3.DATAPATH.REGFILE[2],
UPDATE_REG2_VALUE, UPDATE_REG2);
      3'd3: UPDATE_REG3 <= ValidateRegisterUpdate("REG3", LC3.DATAPATH.REGFILE[3],
UPDATE_REG3_VALUE, UPDATE_REG3);
      3'd4: UPDATE_REG4 <= ValidateRegisterUpdate("REG4", LC3.DATAPATH.REGFILE[4],
UPDATE_REG4_VALUE, UPDATE_REG4);
      3'd5: UPDATE_REG5 <= ValidateRegisterUpdate("REG5", LC3.DATAPATH.REGFILE[5],
UPDATE_REG5_VALUE, UPDATE_REG5);
      3'd6: UPDATE_REG6 <= ValidateRegisterUpdate("REG6", LC3.DATAPATH.REGFILE[6],
UPDATE_REG6_VALUE, UPDATE_REG6);
```

```
      3'd7: UPDATE_REG7 <= ValidateRegisterUpdate("REG7", LC3.DATAPATH.REGFILE[7],
UPDATE_REG7_VALUE, UPDATE_REG7);
    endcase
   end
  end

  always @ (negedge LC3.DATAPATH.ldMAR) begin
    memory_dout <= NEXT_MEMORY_OUTPUT;
  end

  always @ (negedge memWE) begin
     if(rst == 1'b0 && rst_done) begin
    UPDATE_MEMORY <= ValidateRegisterUpdate("MEMORY", memory_din, UPDATE_MEMORY_VALUE,
UPDATE_MEMORY);
   end
  end

  always @ (instructionCount) begin
    if (instructionCount >= `INSTRUCTIONS_TO_EXECUTE) begin
      $display("Test Vectors Complete! Executed %d instructions!", instructionCount);
      //parameter BR=4'b0000, ADD=4'b0001, LD=4'b0010, ST=4'b0011,
      //    JSR=4'b0100, AND=4'b0101, LDR=4'b0110, STR=4'b0111,
      //    RTI=4'b1000, NOT=4'b1001, LDI=4'b1010, STI=4'b1011,
      //    JMP=4'b1100, RES=4'b1101, LEA=4'b1110, TRAP=4'b1111;
      $display("BR: %d", InstrCount[BR]);
      $display("ADD: %d", InstrCount[ADD]);
      $display("LD: %d", InstrCount[LD]);
      $display("ST: %d", InstrCount[ST]);
      $display("JSR: %d", InstrCount[JSR]);
      $display("AND: %d", InstrCount[AND]);
      $display("LDR: %d", InstrCount[LDR]);
      $display("STR: %d", InstrCount[STR]);
      $display("RTI: %d", InstrCount[RTI]);
      $display("NOT: %d", InstrCount[NOT]);
      $display("LDI: %d", InstrCount[LDI]);
      $display("STI: %d", InstrCount[STI]);
      $display("JMP: %d", InstrCount[JMP]);
      $display("RES: %d", InstrCount[RES]);
      $display("LEA: %d", InstrCount[LEA]);
      $display("TRAP: %d", InstrCount[TRAP]);
      $finish();
    end
  end

  lc3 LC3(clk,
          rst,
          memory_dout,
          memory_addr,
          memory_din,
          memWE);


endmodule
```

Testbench.do **********

```
#Test Datapath

#Figure out how to compile
source testLib.do

vsim -novopt lc3_datapath

#Initialize the Circuit

#Set the Clock
force -freeze sim:/lc3_datapath/clk 1 0, 0 {10ns} -r 20ns
add wave sim:/lc3_datapath/clk
#Reset the Circuit

force -drive sim:/lc3_datapath/rst 1 0
add wave sim:/lc3_datapath/rst

#Set Control Signals low
force -drive sim:/lc3_datapath/aluControl 00 0
force -drive sim:/lc3_datapath/enaALU 0 0
force -drive sim:/lc3_datapath/SR1 000 0
force -drive sim:/lc3_datapath/SR2 000 0
force -drive sim:/lc3_datapath/DR 000 0
force -drive sim:/lc3_datapath/regWE 0 0
force -drive sim:/lc3_datapath/selPC 00 0
force -drive sim:/lc3_datapath/enaMARM 0 0
force -drive sim:/lc3_datapath/selMAR 0 0
force -drive sim:/lc3_datapath/selEAB1 0 0
force -drive sim:/lc3_datapath/selEAB2 00 0
force -drive sim:/lc3_datapath/enaPC 0 0
force -drive sim:/lc3_datapath/ldPC 0 0
force -drive sim:/lc3_datapath/ldIR 0 0
force -drive sim:/lc3_datapath/ldMAR 0 0
force -drive sim:/lc3_datapath/ldMDR 0 0
force -drive sim:/lc3_datapath/selMDR 0 0
#force -drive sim:/lc3_datapath/memWE 0 0
force -drive sim:/lc3_datapath/flagWE 0 0
force -drive sim:/lc3_datapath/enaMDR 0 0
force -drive sim:/lc3_datapath/rst 1 0

#run 10 Clock Cycles
runClk 10

#Bring Circuit out of Reset
force -drive sim:/lc3_datapath/rst 0 0

runClk 20

echo "Ensure that PC was reset correctly"
checkValue "PC" hex 0000

###################### TEST REGISTERS ########################

############################
## TEST PROGRAM COUNTER ##
############################

add wave -radix hex PC
add wave -radix hex PCMUX
add wave ldPC
runClk 2

testRegister "PC" "PCMUX" "ldPC" 16

############################
```

```
##       TEST MAR       ##
#########################

add wave -radix hex MAR
add wave -radix hex BUSS
add wave ldMAR
runClk 2

testRegister "MAR" "BUSS" "ldMAR" 16

#########################
##       TEST MDR       ##
#########################

add wave -radix hex MDR
add wave -radix hex MDRMUX
add wave ldMDR
runClk 2

testRegister "MDR" "MDRMUX" "ldMDR" 16

#########################
## INSTRUCTION REGISTER ##
#########################

add wave -radix hex IR
add wave -radix hex BUSS
add wave ldIR
runClk 2

testRegister "IR" "BUSS" "ldIR" 16

#########################
##  TEST REGISTER FILE  ##
#########################

add wave -radix hex REGFILE
add wave -radix hex BUSS
add wave regWE
runClk 2

for { set i 0 } {$i < 8} {incr i} {
  force -drive DR 10#$i
  testRegister "REGFILE\[$i\]" "BUSS" "regWE" 16
}

#########################
##     TEST MEMORY      ##
#########################

#add wave -radix hex MEMORY
#add wave -radix hex MAR
#add wave -radix hex MDR
#add wave memWE
#runClk 2

#for { set i 0 } {$i < 8} {incr i} {
#  noforce MAR
#  force -freeze MAR 10#$i
#  testRegister "MEMORY\[$i\]" "MDR" "memWE" 16
#}

###################### TEST MULTIPLEXORS ######################

#########################
##     TEST MARMUX      ##
#########################
```

```
add wave -radix hex MARMUX
add wave -radix hex selMAR
add wave -radix hex ADDER
add wave -radix hex ZEXT

textMultiplexor "MARMUX" [list "ADDER" "ZEXT"] "selMAR" 16

##########################
##     TEST PCMUX      ##
##########################

add wave -radix hex PCMUX
add wave -radix hex selPC
add wave -radix hex PCINCR
add wave -radix hex ADDER
add wave -radix hex BUSS

textMultiplexor "PCMUX" [list "PCINCR" "ADDER" "BUSS" ] "selPC" 16

##########################
##    TEST ADDR1MUX     ##
##########################

add wave -radix hex ADDR1MUX
add wave -radix hex selEAB1
add wave -radix hex PC
add wave -radix hex RA

textMultiplexor "ADDR1MUX" [list "PC" "RA" ] "selEAB1" 16

##########################
##    TEST ADDR2MUX     ##
##########################

add wave -radix hex ADDR2MUX
add wave -radix hex selEAB2
add wave -radix hex ZERO16
add wave -radix hex SEXT5
add wave -radix hex SEXT8
add wave -radix hex SEXT10

textMultiplexor "ADDR2MUX" [list "ZERO16" "SEXT5" "SEXT8" "SEXT10" ] "selEAB2" 16

##########################
##    TEST SR2MUX     ##
##########################

add wave -radix hex SR2MUX
add wave -radix hex IR\[5\]
add wave -radix hex RB
add wave -radix hex SEXT4

textMultiplexor "SR2MUX" [list "RB" "SEXT4" ] "IR\[5\] " 16

##########################
##    TEST MDR2MUX     ##
##########################

add wave -radix hex MDRMUX
add wave -radix hex selMDR
add wave -radix hex BUSS
add wave -radix hex memOut

textMultiplexor "MDRMUX" [list "BUSS" "memOut" ] "selMDR" 16
```

```
#########################
##   TEST SEXT, ZEXT    ##
#########################

echo "Checking Sign Extension, Zero Extension"

add wave -radix hex IR
add wave -radix hex SEXT4
add wave -radix hex SEXT5
add wave -radix hex SEXT8
add wave -radix hex SEXT10
add wave -radix hex ZEXT

#noforce SEXT4
#noforce SEXT5
#noforce SEXT8
#noforce SEXT10
#noforce ZEXT

echo "Check ZERO Extension Correct"

force -freeze IR 16#0000
run 10ps
simtime
checkValue "SEXT4" hex 0000
checkValue "SEXT5" hex 0000
checkValue "SEXT8" hex 0000
checkValue "SEXT10" hex 0000
checkValue "ZEXT" hex 0000
runClk 1

echo "Test IR Bit 4"
simtime
noforce IR
force -freeze IR 16#0010
run 10ps
checkValue "SEXT4" hex fff0
checkValue "SEXT5" hex 0010
checkValue "SEXT8" hex 0010
checkValue "SEXT10" hex 0010
checkValue "ZEXT" hex 0010
runClk 1

echo "Test IR Bit 5"
simtime
noforce IR
force -freeze IR 16#0020
run 10ps
checkValue "SEXT4" hex 0000
checkValue "SEXT5" hex ffe0
checkValue "SEXT8" hex 0020
checkValue "SEXT10" hex 0020
checkValue "ZEXT" hex 0020
runClk 1

echo "Test IR Bit 8"
simtime
noforce IR
force -freeze IR 16#0100
run 10ps
checkValue "SEXT4" hex 0000
checkValue "SEXT5" hex 0000
checkValue "SEXT8" hex ff00
checkValue "SEXT10" hex 0100
checkValue "ZEXT" hex 0000
runClk 1
```

```
echo "Test IR Bit 10"
simtime
noforce IR
force -freeze IR 16#0400
run 10ps
checkValue "SEXT4" hex 0000
checkValue "SEXT5" hex 0000
checkValue "SEXT8" hex 0000
checkValue "SEXT10" hex fc00
checkValue "ZEXT" hex 0000
runClk 1

echo "Test IR Bit 7, ZEXT"
simtime
noforce IR
force -freeze IR 16#00FF
run 10ps
checkValue "SEXT4" hex ffff
checkValue "SEXT5" hex ffff
checkValue "SEXT8" hex 00ff
checkValue "SEXT10" hex 00ff
checkValue "ZEXT" hex 00ff
runClk 1

noforce IR

##########################
##      TEST ADDER      ##
##########################

echo "Testing Adder"

add wave -radix hex ADDER
add wave -radix hex ADDR1MUX
add wave -radix hex ADDR2MUX

runClk 1
noforce ADDER
noforce ADDR1MUX
noforce ADDR2MUX
force -freeze ADDR1MUX 10#0
force -freeze ADDR2MUX 10#0
run 10ps
checkValue "ADDER" hex 0000

runClk 1
noforce ADDR1MUX
noforce ADDR2MUX
force -freeze ADDR1MUX 10#1
force -freeze ADDR2MUX 10#1
run 10ps
checkValue "ADDER" hex 0002

runClk 1
noforce ADDR1MUX
noforce ADDR2MUX
force -freeze ADDR1MUX 10#32767
force -freeze ADDR2MUX 10#1
run 10ps
checkValue "ADDER" unsigned 32768
runClk 1

##########################
##      TEST ALU        ##
##########################

add wave -radix hex ALU
```

```
add wave -radix hex aluControl
add wave -radix hex RA
add wave -radix hex SR2MUX

echo "Testing ALU"

noforce RA
noforce SR2MUX
echo "Test PASS A"
force -drive aluControl 00
force -freeze RA 16#0000
force -freeze SR2MUX 16#FFFF
run 10ps
checkValue "ALU" hex 0000
runClk 1

noforce RA
noforce SR2MUX
force -freeze RA 16#ffff
force -freeze SR2MUX 16#0000
run 10ps
checkValue "ALU" hex ffff
runClk 1

echo "Testing ADD"
noforce RA
noforce SR2MUX
force -drive aluControl 01
force -freeze RA 16#0001
force -freeze SR2MUX 16#0001
run 10ps
checkValue "ALU" hex 0002
runClk 1


echo "Testing AND"
noforce RA
noforce SR2MUX
force -drive aluControl 10
force -freeze RA 16#AAAA
force -freeze SR2MUX 16#FFFF
run 10ps
checkValue "ALU" hex aaaa
runClk 1

echo "Testing NOT"

force -drive aluControl 11
run 10ps
checkValue "ALU" hex 5555
runClk 1
noforce RA
noforce SR2MUX

############################
##      TEST BUS          ##
############################

echo "Testing BUSS Drivers"

add wave -radix hex BUSS
add wave enaALU
add wave enaMARM
add wave enaPC
add wave enaMDR

noforce MARMUX
```

```
noforce PC
noforce ALU
noforce MDR
force -freeze MARMUX 16#ffff
force -freeze PC 16#ffff
force -freeze ALU 16#ffff
force -freeze MDR 16#ffff

force -drive enaALU 0
force -drive enaMARM 0
force -drive enaPC 0
force -drive enaMDR 0

run 10ps

checkValue BUSS hex zzzz

runClk 1

force -drive enaALU 1
force -drive enaMARM 0
force -drive enaPC 0
force -drive enaMDR 0

run 10ps

checkValue BUSS hex ffff

runClk 1

force -drive enaALU 0
force -drive enaMARM 1
force -drive enaPC 0
force -drive enaMDR 0

run 10ps

checkValue BUSS hex ffff

runClk 1

force -drive enaALU 0
force -drive enaMARM 0
force -drive enaPC 1
force -drive enaMDR 0

run 10ps

checkValue BUSS hex ffff

runClk 1

force -drive enaALU 0
force -drive enaMARM 0
force -drive enaPC 0
force -drive enaMDR 1

run 10ps

checkValue BUSS hex ffff

runClk 1

echo "done"
#quit –sim
```

Lc3.sv **********

```
`default_nettype none
module lc3(clk,
           rst,
           memory_dout,
           memory_addr,
           memory_din,
          memWE);

input logic clk;
input logic rst;

input logic [15:0] memory_dout;

output logic [15:0] memory_addr;
output logic [15:0] memory_din;
output logic memWE;

logic [15:0] IR;
logic N;
logic Z;
logic P;
logic [1:0] aluControl;
logic enaALU;
logic [2:0] SR1;
logic [2:0] SR2;
logic [2:0] DR;
logic logicWE;
logic [1:0] selPC;
logic enaMARM;
logic selMAR;
logic selEAB1;
logic [1:0] selEAB2;
logic enaPC;
logic ldPC;
logic ldIR;
logic ldMAR;
logic ldMDR;
logic selMDR;

logic flagWE;
logic enaMDR;

lc3_datapath DATAPATH( clk, rst,
                    IR, N, Z, P,
                    aluControl, enaALU, SR1, SR2,
                    DR, logicWE, selPC, enaMARM, selMAR,
                selEAB1, selEAB2, enaPC, ldPC, ldIR,
                  ldMAR, ldMDR, selMDR, flagWE, enaMDR,
                    memory_din, memory_dout, memory_addr);

lc3_control CONTROL( clk, rst,
                    IR, N, Z, P,
                    aluControl, enaALU, SR1, SR2,
                    DR, logicWE, selPC, enaMARM, selMAR,
                selEAB1, selEAB2, enaPC, ldPC, ldIR,
                  ldMAR, ldMDR, selMDR, memWE, flagWE, enaMDR);

endmodule
```

```
lc3_datapath.sv ************************************************************************************

`default_nettype none
module lc3_datapath ( clk, rst,
                      IR_OUT, N_OUT, Z_OUT, P_OUT,
                      aluControl, enaALU, SR1, SR2,
                      DR, logicWE, selPC, enaMARM, selMAR,
                  selEAB1, selEAB2, enaPC, ldPC, ldIR,
                    ldMAR, ldMDR, selMDR, flagWE, enaMDR,
                      memory_din, memory_dout, memory_addr);
input logic clk;
input logic rst;

input logic [1:0] aluControl;
input logic enaALU;
input logic [2:0] SR1;
input logic [2:0] SR2;
input logic [2:0] DR;
input logic logicWE;
input logic [1:0] selPC;
input logic enaMARM;
input logic selMAR;
input logic selEAB1;
input logic [1:0] selEAB2;
input logic enaPC;
input logic ldPC;
input logic ldIR;
input logic ldMAR;
input logic ldMDR;
input logic selMDR;
input logic flagWE;
input logic enaMDR;

input logic [15:0] memory_dout;

output logic [15:0] IR_OUT;
output logic N_OUT;
output logic Z_OUT;
output logic P_OUT;

output logic [15:0] memory_din;
output logic [15:0] memory_addr;


//Datapath Registers
logic [15:0] PC;
logic [15:0] IR;
logic [15:0] MAR;
logic [15:0] MDR;
logic N, Z, P;
logic [15:0] REGFILE [0:7];

//logic [15:0] MEMORY [0:255];

wire [15:0] BUSS;

//Multiplexors
logic [15:0] PCMUX;
logic [15:0] MARMUX;
logic [15:0] MDRMUX;
logic [15:0] ADDR1MUX;
logic [15:0] ADDR2MUX;
logic [15:0] SR2MUX;

//Arithmetic Units
logic [15:0] ADDER;
logic [15:0] PCINCR;
```

```
logic [15:0] ALU;
logic [15:0] ZERO16;
logic [15:0] i;
//Register File Outputs
logic [15:0] RA;
logic [15:0] RB;

//IR Sign Extension
logic [15:0] SEXT4;
logic [15:0] SEXT5;
logic [15:0] SEXT8;
logic [15:0] SEXT10;
logic [15:0] ZEXT;

logic [15:0] memOut;

assign IR_OUT = IR;
assign N_OUT = N;
assign Z_OUT = Z;
assign P_OUT = P;

assign memOut = memory_dout;
assign memory_din = MDR;
assign memory_addr = MAR;

/***********************************
 Program Counter
***********************************/

always_ff @ (posedge clk iff rst == 0 or posedge rst) begin
  if (rst == 1'b1) begin
    PC <= 16'd0;
  end else if(ldPC) begin
    PC = PCMUX;
  end
end

/***********************************
 Program Counter MUX
***********************************/

assign PCINCR = PC + 16'd1;

always_comb begin
  unique case(selPC)
    2'b00: PCMUX = PCINCR;
    2'b01: PCMUX = ADDER;
    2'b10: PCMUX = BUSS;
    default: $display("PCMUX ERROR: Illegal Select Signal ");
  endcase
end

/***********************************
 Memory
***********************************/

//always @(posedge clk)
//begin
//  if (memWE)
//    MEMORY[MAR] <= MDR;
//end

//assign memOut = MEMORY[MAR];

/***********************************
 MAR
***********************************/
```

```
always_ff @ (posedge clk iff rst==0 or posedge rst) begin
 if(rst) begin
   MAR <= 16'd0;
 end else if(ldMAR == 1'b1) begin
   MAR <= BUSS;
 end
end
/***********************************
 MDR
***********************************/

always_ff @ (posedge clk iff rst == 0 or posedge rst) begin
 if(rst) begin
   MDR <= 16'd0;
 end else if(ldMDR == 1'b1) begin
   MDR <= MDRMUX;
 end
end

/***********************************
 Instruction Register
***********************************/

always_ff @ (posedge clk iff rst == 0 or posedge rst) begin
  if(rst) begin
    IR <= 16'd0;
  end else if(ldIR == 1'b1) begin
    IR <= BUSS;
  end
end

/***********************************
 Instruction Register Sign Extend
***********************************/

assign SEXT4 = { {11{IR[4]}}, IR[4:0] };
assign SEXT5 = { {10{IR[5]}}, IR[5:0] };
assign SEXT8 = { {7{IR[8]}}, IR[8:0] };
assign SEXT10 = { {5{IR[10]}}, IR[10:0] };
assign ZEXT =   { 8'b0, IR[7:0] };

/***********************************
 MARMUX
***********************************/

assign MARMUX = (selMAR) ? ZEXT : ADDER;

/***********************************
 ADDER
***********************************/

assign ADDER = ADDR2MUX + ADDR1MUX;

/***********************************
 ADDR1MUX
***********************************/

assign ADDR1MUX = (selEAB1) ? RA : PC;

/***********************************
 ADDR2MUX
***********************************/
assign ZERO16 = 16'h0000;

always_comb begin
  unique case (selEAB2)
```

```
    2'b00: ADDR2MUX = ZERO16;
    2'b01: ADDR2MUX = SEXT5;
    2'b10: ADDR2MUX = SEXT8;
    2'b11: ADDR2MUX = SEXT10;
  endcase
end
/***********************************
 SR2MUX
***********************************/

assign SR2MUX = (IR[5]) ? SEXT4 : RB;

/***********************************
 MDRMUX
***********************************/

assign MDRMUX = (selMDR) ? memOut : BUSS;

/***********************************
 ALU
***********************************/

always_comb
  unique case(aluControl)
    2'b00: ALU = RA;
    2'b01: ALU = RA + SR2MUX;
    2'b10: ALU = RA & SR2MUX;
    2'b11: ALU = ~RA;
  endcase

/***********************************
 NZP Logic
***********************************/

always_ff @ (posedge clk) begin
  if(BUSS == 16'h0000) begin
    N <= 1'b0; Z <= 1'b1; P <= 1'b0;
  end
  if(BUSS[15] == 1'b1) begin
    N <= 1'b1; Z <= 1'b0; P <= 1'b0;
  end
  if( | BUSS[14:0] == 1'b1) begin
    N <= 1'b0; Z <= 1'b0; P <= 1'b1;
  end
end
/***********************************
 Register File
***********************************/
always_ff @ (posedge clk iff rst == 0 or posedge rst) begin
  if(rst) begin
    for(i=0; i<16; i = i + 1) begin
      REGFILE[i] <= 16'd0;
    end
  end else if(logicWE) begin
    REGFILE[DR] <= BUSS;
  end
end
assign RA = REGFILE[SR1];
assign RB = REGFILE[SR2];
/***********************************
 BUSS
***********************************/
assign BUSS = (enaMARM) ? MARMUX : 16'hZZZZ;
assign BUSS = (enaPC) ? PC : 16'hZZZZ;
assign BUSS = (enaALU) ? ALU : 16'hZZZZ;
assign BUSS = (enaMDR) ? MDR : 16'hZZZZ;
endmodule
```

```
lc3_control.sv ********************************************************************************

`default_nettype none
//`include "lc3Pkg.sv"

import lc3Pkg::*;

module lc3_control ( clk, rst,
                     IR, N, Z, P,
                     aluControl, enaALU, SR1, SR2,
                     DR, logicWE, selPC, enaMARM, selMAR,
                             selEAB1, selEAB2, enaPC, ldPC, ldIR,
                         ldMAR, ldMDR, selMDR, memWE, flagWE, enaMDR);

input logic clk;
input logic rst;

input logic [15:0] IR;
input logic N;
input logic Z;
input logic P;

//Output
output logic [1:0] aluControl = 2'b00;
output logic [2:0] SR1 = 3'b000;
output logic [2:0] SR2 = 3'b000;
output logic [2:0] DR = 3'b000;

output logic enaALU = 1'b0;
output logic enaPC = 1'b0;
output logic enaMDR = 1'b0;
output logic enaMARM = 1'b0;

output logic [1:0] selPC = 2'b00;
output logic selMAR = 1'b0;
output logic selEAB1 = 1'b0;
output logic [1:0] selEAB2 = 2'b00;
output logic selMDR = 1'b0;

output logic ldPC = 1'b0;
output logic ldIR = 1'b0;
output logic ldMAR = 1'b0;
output logic ldMDR = 1'b0;

output logic memWE = 1'b0;
output logic flagWE = 1'b0;
output logic logicWE = 1'b0;


ControlStates CurrentState;
ControlStates NextState;
logic branch_enable;


assign  branch_enable = ((N == IR[11]) || (Z == IR[10]) || (P == IR[9])) ? 1'b1 : 1'b0;

always_ff @ (posedge clk iff rst == 0 or posedge rst) begin
 if(rst)
  CurrentState <= FETCH0;
 else
  CurrentState <= NextState;
end

always_comb begin
  //Tristate Signals
  enaALU <= 1'b0; enaMARM <= 1'b0;
  enaPC <= 1'b0; enaMDR <= 1'b0;
```

```
//Register Load Signals
ldPC <= 1'b0; ldIR <= 1'b0;
ldMAR <= 1'b0; ldMDR <= 1'b0;

//MUX Select Signal
selPC <= 2'b00; selMAR <= 1'b0;
selEAB1 <= 1'b0; selEAB2 <= 2'b00;
selMDR <= 1'b0;

//Write Enable Signals
flagWE <= 1'b0;
memWE <= 1'b0;
logicWE <= 1'b0;

//Control Signals
aluControl <= 2'b00;
SR1 <= 3'b000;
SR2 <= 3'b000;
DR <= 3'b000;

unique case (CurrentState)
  FETCH0: begin
    NextState <= FETCH1;
    //Load PC ADDRESS
    enaPC <= 1'b1; ldMAR <= 1'b1;
  end
  FETCH1: begin
    NextState <= FETCH2;
    //READ Instruction From Memory
    selMDR<=1'b1; ldMDR<=1'b1;
    //Incremente Program Counter
    selPC<=2'b00;
    ldPC<=1'b1;
  end
  FETCH2: begin
    NextState <= DECODE;
    //Load Instruction Register
    enaMDR <= 1'b1; ldIR <= 1'b1;
  end
  DECODE: begin
    unique case (IR[15:12])  //AND, ADD, NOT, JSR, BR, LD, ST, JMP.
      BR:  NextState <= BRANCH0;//**//
      ADD: NextState <= ADD0;   //**//
      LD:  NextState <= LD0;  //**//
      ST:  NextState <= STORE0; //**//
      JSR: NextState <= JSR0;   //**//
      AND: NextState <= AND0;   //**//
      LDR: NextState <= FETCH0;
      STR: NextState <= FETCH0;
      RTI: NextState <= FETCH0;
      NOT: NextState <= NOT0;   //**//
      LDI: NextState <= FETCH0;
      STI: NextState <= FETCH0;
      JMP: NextState <= JMP0;   //
      RES: NextState <=  FETCH0;
      LEA:  NextState <= FETCH0;
      TRAP: NextState <= FETCH0;
     endcase
  end
  BRANCH0:  begin
   //Select ADDER inputs
   selEAB1 <= 1'b0;
   selEAB2 <= 2'b10;
   //Load the New PC Value (if Branch Condition Met)
   ldPC <= branch_enable;
   selPC <= 2'b01;
```

```
 NextState <= FETCH0;
end
LD0: begin
 //Load MAR
 selEAB2 <= 2'b10;
 selEAB1 <= 1'b0;
 selMAR <= 1'b0;
 enaMARM <= 1'b1;
 ldMAR <= 1'b1;
 NextState <= LD1;
end
LD1: begin
 //Load MDR
 selMDR <= 1'b1;
 ldMDR <= 1'b1;

 NextState <= LD2;
end
LD2: begin
 //Write to Register File
 DR <= IR[11:9];
 logicWE <= 1'b1;
 enaMDR <= 1'b1;
 NextState <= FETCH0;
end
NOT0: begin
 aluControl <= 2'b11;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
 DR <= IR[11:9];
 logicWE <= 1'b1;
 NextState <= FETCH0;
end
ADD0: begin
 aluControl <= 2'b01;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
 SR2 <= IR[2:0];
 DR <= IR[11:9];
 logicWE <= 1'b1;
 flagWE <= 1'b1;
 NextState <= FETCH0;
end
AND0: begin
 aluControl <= 2'b10;
 enaALU <= 1'b1;
 SR1 <= IR[8:6];
 SR2 <= IR[2:0];
 DR <= IR[11:9];
 logicWE <= 1'b1;
 NextState <= FETCH0;
end
STORE0: begin
 //Load the MDR
 SR1 <= IR[11:9];
 aluControl <= 2'b00;
 enaALU <= 1'b1;
 selMDR <= 1'b0;
 ldMDR <= 1'b1;
 NextState <= STORE1;
end
STORE1: begin
 //Load the MAR
 selEAB1 <= 1'b0;
 selEAB2 <= 2'b10;
 selMAR <= 1'b0;
 enaMARM <= 1'b1;
```

```
      ldMAR <= 1'b1;
      NextState <= STORE2;
     end
     STORE2: begin
      memWE <= 1'b1;
      NextState <= FETCH0;
    end
    JSR0: begin
      DR <= 3'b111;
      enaPC <= 1'b1;
      logicWE <= 1'b1;
      NextState <= JSR1;
    end
    JSR1: begin
      selEAB1 <= 1'b0;
      selEAB2 <= 2'b11;
      selPC <= 2'b01;
      ldPC <= 1'b1;
      NextState <= FETCH0;
    end
    JMP0: begin
      SR1 <= IR[8:6];
      selEAB1 <= 1'b1;
      selEAB2 <= 2'b00;
      selPC <= 2'b01;
      ldPC <= 1'b1;
      NextState <= FETCH0;
    end
   endcase
end


endmodule
```

Lc3_testbench.sv**********

```
`default_nettype none
`timescale 1ns/100ps
`define TBSTATE_FETCH 1'b0
`define TBSTATE_EXECUTE 1'b1
`define INSTRUCTIONS_TO_EXECUTE 16'd10000
//`include "lc3Pkg.sv"

import lc3Pkg::*;

module lc3_testbench ();

  class RegisterMonitor;
    string name;
    bit UpdateScheduled;
    logic [15:0] ExpectedValue;

    //Constructor
    function new (string regName);
      begin
        this.name = regName;
      end
    endfunction

    function ScheduleForUpdate(logic newValue);
      begin
        this.UpdatedScheduled = 1;
        this.ExpectedValue = newValue;
      end
    endfunction
  endclass


  event found_error;
  event evaluate_executeUpdateStatus;
  function [16:0] ValidateRegisterUpdate;
   input [63:0] logicName;
   input [15:0] currentValue, expectedValue;
   input update;
   logic [16:0] ret;
   begin
     $display("Checking Register %s @ time: %d", logicName, $time);
     if(update == 1'b0) begin
        $display("   Error: %s is not scheduled for Update!", logicName);
        -> found_error;
        ret = 1;
     end else if(currentValue == expectedValue) begin
        //Register Update Correctly
        $display("   Success: %s was updated as expected", logicName);
        ret = 0;
     end else begin
        $display("   Error: %s was %d expected %d", logicName, currentValue, expectedValue);
        ->found_error;
        ret = 1;
     end
     ValidateRegisterUpdate = ret;
   end
  endfunction

  //parameter BR=4'b0000, ADD=4'b0001, LD=4'b0010, ST=4'b0011,
  //        JSR=4'b0100, AND=4'b0101, LDR=4'b0110, STR=4'b0111,
  //        RTI=4'b1000, NOT=4'b1001, LDI=4'b1010, STI=4'b1011,
  //        JMP=4'b1100, RES=4'b1101, LEA=4'b1110, TRAP=4'b1111;

  logic [15:0] logicNamePC = "PC";
```

```
logic clk, rst;

logic [15:0] memory_dout;
wire [15:0] memory_addr;
wire [15:0] memory_din;
wire memWE;

event start_fetch;
event start_execute;

logic rst_done = 1'b0;
logic tbState = `TBSTATE_FETCH;

logic [15:0] NEXT_MEMORY_OUTPUT;

logic UPDATE_MAR = 1'b0;
logic [15:0] UPDATE_MAR_VALUE;

logic UPDATE_MDR = 1'b0;
logic [15:0] UPDATE_MDR_VALUE;

logic UPDATE_PC = 1'b0;
logic [15:0] UPDATE_PC_VALUE;

logic UPDATE_IR = 1'b0;
logic [15:0] UPDATE_IR_VALUE;

logic UPDATE_MEMORY = 1'b0;
logic [15:0] UPDATE_MEMORY_VALUE;

logic [15:0] InstrCount [15:0];
logic [15:0] i;

logic UPDATE_REG0 = 1'b0;
logic UPDATE_REG1 = 1'b0;
logic UPDATE_REG2 = 1'b0;
logic UPDATE_REG3 = 1'b0;
logic UPDATE_REG4 = 1'b0;
logic UPDATE_REG5 = 1'b0;
logic UPDATE_REG6 = 1'b0;
logic UPDATE_REG7 = 1'b0;

logic [15:0] UPDATE_REG0_VALUE;
logic [15:0] UPDATE_REG1_VALUE;
logic [15:0] UPDATE_REG2_VALUE;
logic [15:0] UPDATE_REG3_VALUE;
logic [15:0] UPDATE_REG4_VALUE;
logic [15:0] UPDATE_REG5_VALUE;
logic [15:0] UPDATE_REG6_VALUE;
logic [15:0] UPDATE_REG7_VALUE;

wire [3:0] fetchUpdateStatus;
wire [11:0] executeUpdateStatus;
logic [11:0] executeUpdateStatus_i;
logic [15:0] instructionCount;

logic [15:0] SEXT10, SEXT8, SEXT4, STOREVAL;
logic [15:0] ADDER;
logic [15:0] OP1, OP2;
logic [2:0] DST, SRC, SRC1, SRC2;
logic [2:0] DSTREG;
task setDstRegUpdate;
  input [2:0] dst;
  input [15:0] val;
  begin
    $display("Setting Reg %d with value %d for expected update", dst, val);
    case (dst)
```

```
      3'd0: begin
        UPDATE_REG0 = 1'b1;
        UPDATE_REG0_VALUE = val;
      end
      3'd1: begin
        UPDATE_REG1 = 1'b1;
        UPDATE_REG1_VALUE = val;
      end
      3'd2: begin
        UPDATE_REG2 = 1'b1;
        UPDATE_REG2_VALUE = val;
      end
      3'd3: begin
        UPDATE_REG3 = 1'b1;
        UPDATE_REG3_VALUE = val;
      end
      3'd4: begin
        UPDATE_REG4 = 1'b1;
        UPDATE_REG4_VALUE = val;
      end
      3'd5: begin
        UPDATE_REG5 = 1'b1;
        UPDATE_REG5_VALUE = val;
      end
      3'd6: begin
        UPDATE_REG6 = 1'b1;
        UPDATE_REG6_VALUE = val;
      end
      3'd7: begin
        UPDATE_REG7 = 1'b1;
        UPDATE_REG7_VALUE = val;
      end
    endcase
  end
endtask

assign fetchUpdateStatus = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_IR };
assign executeUpdateStatus = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_MEMORY,
                               UPDATE_REG0, UPDATE_REG1, UPDATE_REG2, UPDATE_REG3,
                               UPDATE_REG4, UPDATE_REG5, UPDATE_REG6, UPDATE_REG7 };
always begin
  #10 clk = !clk;
end

initial begin
  clk = 0;
  rst = 0;
  for (i=0; i<16; i = i + 1) begin
    InstrCount[i] = 16'd0;
  end
  instructionCount = 0;
  NEXT_MEMORY_OUTPUT = 16'd0;
  @ (negedge clk)
  rst = 1;
  @ (negedge clk)
  @ (negedge clk)
  @ (negedge clk)
  rst = 0;
  rst_done = 1'b1;
  ->start_fetch;
end

initial begin
  @ (found_error)
  $display("Simulation Terminated Due to Error");
  $finish();
end
```

```
initial begin
  forever begin
    @ (start_fetch)
    tbState <= `TBSTATE_FETCH;
    $display("****Starting Fetch of Instr: %d @ time ****", instructionCount, $time);
    //Schdule Register Updates
    UPDATE_MAR = 1'b1;
    UPDATE_MAR_VALUE = LC3.DATAPATH.PC;
    UPDATE_PC = 1'b1;
    UPDATE_PC_VALUE = LC3.DATAPATH.PC + 1;
    UPDATE_IR = 1'b1;
    UPDATE_IR_VALUE = $random;
    //$display("FET:IR_VALUE: %d", UPDATE_IR_VALUE);
    NEXT_MEMORY_OUTPUT = UPDATE_IR_VALUE;
    //$display("FET:NEXT_MEM: %d", NEXT_MEMORY_OUTPUT);
    UPDATE_MDR = 1'b1;
    UPDATE_MDR_VALUE = NEXT_MEMORY_OUTPUT;
  end
end

initial begin
  forever begin
    @ (start_execute);
    tbState = `TBSTATE_EXECUTE;
    $display("****Starting Execution of Instr: %d @ time: %d ****", instructionCount, $time);
    DST = UPDATE_IR_VALUE[11:9];
    SRC = UPDATE_IR_VALUE[11:9];
    SRC1 = UPDATE_IR_VALUE[8:6];
    SRC2 = UPDATE_IR_VALUE[2:0];
    SEXT4 = { {11{UPDATE_IR_VALUE[4]}}, UPDATE_IR_VALUE[4:0] };
    SEXT8 = { {7{UPDATE_IR_VALUE[8]}}, UPDATE_IR_VALUE[8:0] };
    SEXT10 = { {5{UPDATE_IR_VALUE[10]}}, UPDATE_IR_VALUE[10:0] };
    STOREVAL = LC3.DATAPATH.REGFILE[SRC];
    OP1 =  LC3.DATAPATH.REGFILE[SRC1];
    OP2 =  LC3.DATAPATH.REGFILE[SRC2];
    InstrCount[UPDATE_IR_VALUE[15:12]] = InstrCount[UPDATE_IR_VALUE[15:12]] + 1;
    //Schedule any Register Updates
    $display("OPCODE: %d", UPDATE_IR_VALUE);
    case (UPDATE_IR_VALUE[15:12])  //AND, ADD, NOT, JSR, BR, LD, ST, JMP.
      BR: begin //**//
       $display(" Instr %d: BR", instructionCount);
       if(UPDATE_IR_VALUE[11] == LC3.DATAPATH.N ||
          UPDATE_IR_VALUE[10] == LC3.DATAPATH.Z ||
          UPDATE_IR_VALUE[9]  == LC3.DATAPATH.P) begin
          //Update Program Counter
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = UPDATE_PC_VALUE + SEXT8;
       end
      end
      ADD: begin    //**//
         $display(" Instr %d: ADD", instructionCount);
         if(UPDATE_IR_VALUE[5] == 1'b1) begin
           OP2 = SEXT4;
         end
         ADDER = OP1 + OP2;
         setDstRegUpdate(DST, ADDER);
      end
      LD: begin //**//
        $display(" Instr %d: LD", instructionCount);
        UPDATE_MAR = 1'b1;
        UPDATE_MAR_VALUE = UPDATE_PC_VALUE + SEXT8;
        UPDATE_MDR = 1'b1;
        NEXT_MEMORY_OUTPUT = $random;
        UPDATE_MDR_VALUE =  NEXT_MEMORY_OUTPUT;
        setDstRegUpdate(DST, UPDATE_MDR_VALUE);
      end
      ST:  begin //**//
```

```
          $display(" Instr %d: ST SRC: %d ", instructionCount, SRC);
          UPDATE_MAR = 1'b1;
          UPDATE_MAR_VALUE = UPDATE_PC_VALUE + SEXT8;
          UPDATE_MDR = 1'b1;
          UPDATE_MDR_VALUE = STOREVAL;
          UPDATE_MEMORY = 1'b1;
          UPDATE_MEMORY_VALUE = STOREVAL;
        end
        JSR: begin   //**//
          $display(" Instr %d: JSR", instructionCount);
          setDstRegUpdate(3'd7, UPDATE_PC_VALUE);
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = UPDATE_PC_VALUE + SEXT10;
        end
        AND: begin    //**//
          $display(" Instr %d: AND DST: %d SRC1: %d SRC2: %d", instructionCount, DST, SRC1,
SRC2);
           if(UPDATE_IR_VALUE[5] == 1'b1) begin
             $display(" Instr %d: AND DST: %d SRC1: %d IMMED: %d", instructionCount, DST, SRC1,
SEXT4);
             OP2 = SEXT4;
           end
           ADDER = OP1 & OP2;
           setDstRegUpdate(DST, ADDER);
        end
        LDR: $display(" Instr %d: LDR", instructionCount);
        STR: $display(" Instr %d: STR", instructionCount);
        RTI: $display(" Instr %d: RTI", instructionCount);
        NOT: begin  //**//
           $display(" Instr %d: NOT", instructionCount);
           ADDER = ~OP1;
           setDstRegUpdate(DST, ADDER);
        end
        LDI:  $display(" Instr %d: LDI", instructionCount);
        STI:  $display(" Instr %d: STI", instructionCount);
        JMP:  begin //
          $display(" Instr %d: JMP BASE: %d", instructionCount, SRC1);
          UPDATE_PC = 1'b1;
          UPDATE_PC_VALUE = OP1;
        end
        RES:  $display(" Instr %d: RES", instructionCount);
        LEA:  $display(" Instr %d: LEA", instructionCount);
        TRAP:  $display(" Instr %d: TRAP", instructionCount);
      endcase

      executeUpdateStatus_i = { UPDATE_MAR, UPDATE_MDR, UPDATE_PC, UPDATE_MEMORY,
                                UPDATE_REG0, UPDATE_REG1, UPDATE_REG2, UPDATE_REG3,
                                UPDATE_REG4, UPDATE_REG5, UPDATE_REG6, UPDATE_REG7 };
      $display("executeUpdateState_i: %d", executeUpdateStatus_i);
      if(executeUpdateStatus_i == 12'd0) begin
        $display("0:Execution of Instr %d Complete @ time: %d", instructionCount, $time);
        instructionCount = instructionCount + 1;
        -> start_fetch;
      end
    end
  end

  //Are we done with FETCH?
  always @ (fetchUpdateStatus) begin
     //$display("fetchUpdateStatus: %x", fetchUpdateStatus);
     if ( fetchUpdateStatus == 4'b0000 && tbState == `TBSTATE_FETCH && rst_done) begin
       $display("Fetch of Instr %d Complete", instructionCount);
       -> start_execute;
     end
  end

  //Are we done with FETCH?
```

```
  always @ (executeUpdateStatus) begin
     //$display("executeUpdateStatus: %x", executeUpdateStatus);
     if ( executeUpdateStatus == 12'd0 && tbState == `TBSTATE_EXECUTE && rst_done) begin
       $display("1: Execution of Instr %d Complete", instructionCount);
       instructionCount = instructionCount + 1;
       -> start_fetch;
     end
  end

  /*************************************/
  /*  Processes to Monitor State Changes  */
  /*************************************/

  always @ (negedge LC3.DATAPATH.ldPC) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_PC <= ValidateRegisterUpdate("PC", LC3.DATAPATH.PC, UPDATE_PC_VALUE, UPDATE_PC);
   end
  end

  always @ (negedge LC3.DATAPATH.ldMAR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_MAR <= ValidateRegisterUpdate("MAR", LC3.DATAPATH.MAR, UPDATE_MAR_VALUE, UPDATE_MAR);
   end
  end

  always @ (negedge LC3.DATAPATH.ldMDR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_MDR <= ValidateRegisterUpdate("MDR", LC3.DATAPATH.MDR, UPDATE_MDR_VALUE, UPDATE_MDR);
   end
  end

  always @ (negedge LC3.DATAPATH.ldIR) begin
   if(rst == 1'b0 && rst_done) begin
    UPDATE_IR <= ValidateRegisterUpdate("IR", LC3.DATAPATH.IR, UPDATE_IR_VALUE, UPDATE_IR);
   end
  end

  always @ (posedge LC3.DATAPATH.logicWE) begin
   DSTREG = LC3.DATAPATH.DR;
   @ ( negedge LC3.DATAPATH.logicWE )
   if(rst == 1'b0 && rst_done ) begin
    case (DSTREG)
      3'd0: UPDATE_REG0 <= ValidateRegisterUpdate("REG0", LC3.DATAPATH.REGFILE[0],
UPDATE_REG0_VALUE, UPDATE_REG0);
      3'd1: UPDATE_REG1 <= ValidateRegisterUpdate("REG1", LC3.DATAPATH.REGFILE[1],
UPDATE_REG1_VALUE, UPDATE_REG1);
      3'd2: UPDATE_REG2 <= ValidateRegisterUpdate("REG2", LC3.DATAPATH.REGFILE[2],
UPDATE_REG2_VALUE, UPDATE_REG2);
      3'd3: UPDATE_REG3 <= ValidateRegisterUpdate("REG3", LC3.DATAPATH.REGFILE[3],
UPDATE_REG3_VALUE, UPDATE_REG3);
      3'd4: UPDATE_REG4 <= ValidateRegisterUpdate("REG4", LC3.DATAPATH.REGFILE[4],
UPDATE_REG4_VALUE, UPDATE_REG4);
      3'd5: UPDATE_REG5 <= ValidateRegisterUpdate("REG5", LC3.DATAPATH.REGFILE[5],
UPDATE_REG5_VALUE, UPDATE_REG5);
      3'd6: UPDATE_REG6 <= ValidateRegisterUpdate("REG6", LC3.DATAPATH.REGFILE[6],
UPDATE_REG6_VALUE, UPDATE_REG6);
      3'd7: UPDATE_REG7 <= ValidateRegisterUpdate("REG7", LC3.DATAPATH.REGFILE[7],
UPDATE_REG7_VALUE, UPDATE_REG7);
    endcase
   end
  end

  always @ (negedge LC3.DATAPATH.ldMAR) begin
    memory_dout <= NEXT_MEMORY_OUTPUT;
  end

  always @ (negedge memWE) begin
```

```
     if(rst == 1'b0 && rst_done) begin
    UPDATE_MEMORY <= ValidateRegisterUpdate("MEMORY", memory_din, UPDATE_MEMORY_VALUE,
UPDATE_MEMORY);
   end
  end

  always @ (instructionCount) begin
    if (instructionCount >= `INSTRUCTIONS_TO_EXECUTE) begin
      $display("Test Vectors Complete! Executed %d instructions!", instructionCount);
      //parameter BR=4'b0000, ADD=4'b0001, LD=4'b0010, ST=4'b0011,
      //    JSR=4'b0100, AND=4'b0101, LDR=4'b0110, STR=4'b0111,
      //    RTI=4'b1000, NOT=4'b1001, LDI=4'b1010, STI=4'b1011,
      //    JMP=4'b1100, RES=4'b1101, LEA=4'b1110, TRAP=4'b1111;
      $display("BR: %d", InstrCount[BR]);
      $display("ADD: %d", InstrCount[ADD]);
      $display("LD: %d", InstrCount[LD]);
      $display("ST: %d", InstrCount[ST]);
      $display("JSR: %d", InstrCount[JSR]);
      $display("AND: %d", InstrCount[AND]);
      $display("LDR: %d", InstrCount[LDR]);
      $display("STR: %d", InstrCount[STR]);
      $display("RTI: %d", InstrCount[RTI]);
      $display("NOT: %d", InstrCount[NOT]);
      $display("LDI: %d", InstrCount[LDI]);
      $display("STI: %d", InstrCount[STI]);
      $display("JMP: %d", InstrCount[JMP]);
      $display("RES: %d", InstrCount[RES]);
      $display("LEA: %d", InstrCount[LEA]);
      $display("TRAP: %d", InstrCount[TRAP]);
      $finish();
    end
  end

  lc3 LC3(clk,
          rst,
          memory_dout,
          memory_addr,
          memory_din,
          memWE);


endmodule
```

Lc3_testbench.sv with classes**********

```
`default_nettype none
`timescale 1ns/100ps
`define TBSTATE_FETCH 1'b0
`define TBSTATE_EXECUTE 1'b1
`define INSTRUCTIONS_TO_EXECUTE 16'd10000
//`include "lc3Pkg.sv"

import lc3Pkg::*;

module lc3_testbench ();

  event found_error;
  event start_fetch;
  event start_execute;

  typedef enum { FETCH, EXECUTE } SIMULATOR_STATE;

  class RegisterMonitor;
    static integer CurrentID;
    static logic [255:0] UpdateSchedule;
    static SIMULATOR_STATE SIM_STATE;

    string name;
    integer UpdateID;
    logic [15:0] ExpectedValue;

    //Constructor
    function new (string regName);
      begin
        this.name = regName;
        this.UpdateID = CurrentID;
        CurrentID++;
      end
    endfunction

    function Update();
      begin
        Update = this.UpdateSchedule[this.UpdateID];
      end
    endfunction

    function getExpectedValue();
      begin
        getExpectedValue = this.ExpectedValue;
      end
    endfunction;

    function SetUpdate(bit val);
      begin
       this.UpdateSchedule[this.UpdateID] = val;
      end
    endfunction

    function ScheduleForUpdate(logic newValue);
      begin
        SetUpdate(1'b1);
        this.ExpectedValue = newValue;
      end
    endfunction

    static function SwitchSimState();
      begin
        if(UpdateSchedule == 256'd0) begin
          if(SIM_STATE == FETCH) begin
            SIM_STATE = EXECUTE;
```

```
            -> start_execute;
          end else begin
            SIM_STATE = FETCH;
            -> start_fetch;
          end
        end
      end
    endfunction

    function ValidateUpdate(logic currentValue);
       begin
          $display("%s: Validating Update @ time: %d", this.name, $time);
          if(this.Update() && currentValue == this.ExpectedValue) begin
             $display("%s: Updated as expected @ time: %d", this.name, $time);
             this.SetUpdate(0);
             this.SwitchSimState();
          end else if(this.Update() == 1 &&
                      currentValue != this.ExpectedValue) begin
            $display("   Error: %s was %d expected %d", this.name, currentValue,
this.ExpectedValue);
            -> found_error;
          end else begin
            $display("   Error: %s is not scheduled for Update!", this.name);
            -> found_error;
          end
       end
    endfunction;
  endclass

  event evaluate_executeUpdateStatus;

  logic clk, rst;

  logic [15:0] memory_dout;
  wire [15:0] memory_addr;
  wire [15:0] memory_din;
  wire memWE;

  logic rst_done = 1'b0;
  logic tbState = `TBSTATE_FETCH;

  logic [15:0] NEXT_MEMORY_OUTPUT;

  RegisterMonitor MAR_Monitor = new("MAR");
  RegisterMonitor MDR_Monitor = new("MDR");
  RegisterMonitor PC_Monitor = new("PC");
  RegisterMonitor IR_Monitor = new("IR");
  RegisterMonitor MEMORY_Monitor = new("MEMORY");

  RegisterMonitor REGFILE [0:7];


  logic [15:0] InstrCount [15:0];
  logic [15:0] i;

  wire [3:0] fetchUpdateStatus;
  wire [11:0] executeUpdateStatus;
  logic [11:0] executeUpdateStatus_i;
  logic [15:0] instructionCount;

  logic [15:0] SEXT10, SEXT8, SEXT4, STOREVAL;
  logic [15:0] ADDER;
  logic [15:0] OP1, OP2;
  logic [2:0] DST, SRC, SRC1, SRC2;
  logic [2:0] DSTREG;
```

```
always begin
   #10 clk = !clk;
 end

 initial begin
   clk = 0;
   rst = 0;

   for (i=0; i<16; i = i + 1) begin
     InstrCount[i] = 16'd0;
   end

   REGFILE[0] = new ("REG0");
   REGFILE[1] = new ("REG1");
   REGFILE[2] = new ("REG2");
   REGFILE[3] = new ("REG3");
   REGFILE[4] = new ("REG4");
   REGFILE[5] = new ("REG5");
   REGFILE[6] = new ("REG6");
   REGFILE[7] = new ("REG7");

   instructionCount = 0;
   NEXT_MEMORY_OUTPUT = 16'd0;
   @ (negedge clk)
   rst = 1;
   @ (negedge clk)
   @ (negedge clk)
   @ (negedge clk)
   rst = 0;
   rst_done = 1'b1;
   ->start_fetch;
 end

 initial begin
   @ (found_error)
   $display("Simulation Terminated Due to Error");
   $finish();
 end

 initial begin
   forever begin
     @ (start_fetch)
     tbState <= `TBSTATE_FETCH;
     $display("****Starting Fetch of Instr: %d @ time ****", instructionCount, $time);
     //Generate Next Instruction
     NEXT_MEMORY_OUTPUT = $random;
     //Schdule Register Updates
     MAR_Monitor.ScheduleForUpdate(LC3.DATAPATH.PC);
     PC_Monitor.ScheduleForUpdate(LC3.DATAPATH.PC + 1);
     IR_Monitor.ScheduleForUpdate(NEXT_MEMORY_OUTPUT);
     MDR_Monitor.ScheduleForUpdate(NEXT_MEMORY_OUTPUT);
   end
 end

 logic [15:0] ExpectedIR;
 logic [15:0] ExpectedPC;

 initial begin
   forever begin
     @ (start_execute);
     ExpectedIR = IR_Monitor.getExpectedValue();
     ExpectedPC = PC_Monitor.getExpectedValue();

     $display("****Starting Execution of Instr: %d @ time: %d ****", instructionCount, $time);
     DST = ExpectedIR[11:9];
     SRC = ExpectedIR[11:9];
     SRC1 = ExpectedIR[8:6];
```

```
      SRC2 = ExpectedIR[2:0];
      SEXT4 = { {11{ExpectedIR[4]}}, ExpectedIR[4:0] };
      SEXT8 = { {7{ExpectedIR[8]}}, ExpectedIR[8:0] };
      SEXT10 = { {5{ExpectedIR[10]}}, ExpectedIR[10:0] };
      STOREVAL = LC3.DATAPATH.REGFILE[SRC];
      OP1 =  LC3.DATAPATH.REGFILE[SRC1];
      OP2 =  LC3.DATAPATH.REGFILE[SRC2];
      InstrCount[ExpectedIR[15:12]] = InstrCount[ExpectedIR[15:12]] + 1;
      //Schedule any Register Updates
      $display("OPCODE: %d", ExpectedIR);
      case (ExpectedIR[15:12])  //AND, ADD, NOT, JSR, BR, LD, ST, JMP.
        BR: begin //**//
         $display(" Instr %d: BR", instructionCount);
         if(ExpectedIR[11] == LC3.DATAPATH.N ||
            ExpectedIR[10] == LC3.DATAPATH.Z ||
            ExpectedIR[9]  == LC3.DATAPATH.P) begin
            //Update Program Counter
            PC_Monitor.ScheduleForUpdate( ExpectedPC + SEXT8);
            //UPDATE_PC = 1'b1;
            //UPDATE_PC_VALUE = UPDATE_PC_VALUE + SEXT8;
         end
        end
        ADD: begin    //**//
           $display(" Instr %d: ADD", instructionCount);
           if(ExpectedIR[5] == 1'b1) begin
             OP2 = SEXT4;
           end
           ADDER = OP1 + OP2;
           REGFILE[DST].ScheduleForUpdate(ADDER);
           //setDstRegUpdate(DST, ADDER);
        end
        LD: begin //**//
          $display(" Instr %d: LD", instructionCount);
          NEXT_MEMORY_OUTPUT = $random;

          MAR_Monitor.ScheduleForUpdate(ExpectedPC + SEXT8);
          MDR_Monitor.ScheduleForUpdate(NEXT_MEMORY_OUTPUT);
          REGFILE[DST].ScheduleForUpdate(NEXT_MEMORY_OUTPUT);


        end
        ST:  begin //**//
          $display(" Instr %d: ST SRC: %d ", instructionCount, SRC);
          MAR_Monitor.ScheduleForUpdate(ExpectedPC + SEXT8);
          MDR_Monitor.ScheduleForUpdate(STOREVAL);
          MEMORY_Monitor.ScheduleForUpdate(STOREVAL);
        end
        JSR: begin   //**//
          $display(" Instr %d: JSR", instructionCount);
          REGFILE[7].ScheduleForUpdate(ExpectedPC);
          PC_Monitor.ScheduleForUpdate(ExpectedPC + SEXT10);

        end
        AND: begin    //**//
          $display(" Instr %d: AND DST: %d SRC1: %d SRC2: %d", instructionCount, DST, SRC1,
SRC2);
           if(ExpectedIR[5] == 1'b1) begin
             $display(" Instr %d: AND DST: %d SRC1: %d IMMED: %d", instructionCount, DST, SRC1,
SEXT4);
             OP2 = SEXT4;
           end
           ADDER = OP1 & OP2;
           REGFILE[DST].ScheduleForUpdate(DST);
           //setDstRegUpdate(DST, ADDER);
        end
        LDR: $display(" Instr %d: LDR", instructionCount);
        STR: $display(" Instr %d: STR", instructionCount);
```

```
      RTI: $display(" Instr %d: RTI", instructionCount);
      NOT: begin  //**//
         $display(" Instr %d: NOT", instructionCount);
         ADDER = ~OP1;
         REGFILE[DST].ScheduleForUpdate(DST);
         //setDstRegUpdate(DST, ADDER);
      end
      LDI:  $display(" Instr %d: LDI", instructionCount);
      STI:  $display(" Instr %d: STI", instructionCount);
      JMP:  begin //
        $display(" Instr %d: JMP BASE: %d", instructionCount, SRC1);
        PC_Monitor.ScheduleForUpdate(OP1);
        //UPDATE_PC = 1'b1;
        //UPDATE_PC_VALUE = OP1;
      end
      RES:  $display(" Instr %d: RES", instructionCount);
      LEA:  $display(" Instr %d: LEA", instructionCount);
      TRAP:  $display(" Instr %d: TRAP", instructionCount);
     endcase


     PC_Monitor.SwitchSimState();
     $display("executeUpdateState_i: %d", executeUpdateStatus_i);
  end
end


/*************************************/
/*  Processes to Monitor State Changes  */
/*************************************/

always @ (negedge LC3.DATAPATH.ldPC) begin
 if(rst == 1'b0 && rst_done) begin
   PC_Monitor.ValidateUpdate(LC3.DATAPATH.PC);
 end
end

always @ (negedge LC3.DATAPATH.ldMAR) begin
 if(rst == 1'b0 && rst_done) begin
  MAR_Monitor.ValidateUpdate(LC3.DATAPATH.MAR);
 end
end

always @ (negedge LC3.DATAPATH.ldMDR) begin
 if(rst == 1'b0 && rst_done) begin
  MDR_Monitor.ValidateUpdate(LC3.DATAPATH.MDR);
 end
end

always @ (negedge LC3.DATAPATH.ldIR) begin
 if(rst == 1'b0 && rst_done) begin
  IR_Monitor.ValidateUpdate(LC3.DATAPATH.IR);
 end
end

always @ (posedge LC3.DATAPATH.logicWE) begin
 DSTREG = LC3.DATAPATH.DR;
 @ ( negedge LC3.DATAPATH.logicWE )
 if(rst == 1'b0 && rst_done ) begin
   REGFILE[DSTREG].ValidateUpdate(LC3.DATAPATH.REGFILE[DSTREG]);
 end
end

always @ (negedge LC3.DATAPATH.ldMAR) begin
  memory_dout <= NEXT_MEMORY_OUTPUT;
end
```

```
  always @ (negedge memWE) begin
     if(rst == 1'b0 && rst_done) begin
       MEMORY_Monitor.ValidateUpdate(memory_din);
     // UPDATE_MEMORY <= ValidateRegisterUpdate("MEMORY", memory_din, UPDATE_MEMORY_VALUE,
UPDATE_MEMORY);
     end
  end

  always @ (instructionCount) begin
    if (instructionCount >= `INSTRUCTIONS_TO_EXECUTE) begin
      $display("Test Vectors Complete! Executed %d instructions!", instructionCount);
      $display("BR: %d", InstrCount[BR]);
      $display("ADD: %d", InstrCount[ADD]);
      $display("LD: %d", InstrCount[LD]);
      $display("ST: %d", InstrCount[ST]);
      $display("JSR: %d", InstrCount[JSR]);
      $display("AND: %d", InstrCount[AND]);
      $display("LDR: %d", InstrCount[LDR]);
      $display("STR: %d", InstrCount[STR]);
      $display("RTI: %d", InstrCount[RTI]);
      $display("NOT: %d", InstrCount[NOT]);
      $display("LDI: %d", InstrCount[LDI]);
      $display("STI: %d", InstrCount[STI]);
      $display("JMP: %d", InstrCount[JMP]);
      $display("RES: %d", InstrCount[RES]);
      $display("LEA: %d", InstrCount[LEA]);
      $display("TRAP: %d", InstrCount[TRAP]);
      $finish();
    end
  end


  /* TEST CIRCUIT */
  lc3 LC3(clk,
          rst,
          memory_dout,
          memory_addr,
          memory_din,
          memWE);


endmodule
```