ECE 385

Fall 2015

Experiment #6

Simple Computer SLC-3.2 in Systemverilog

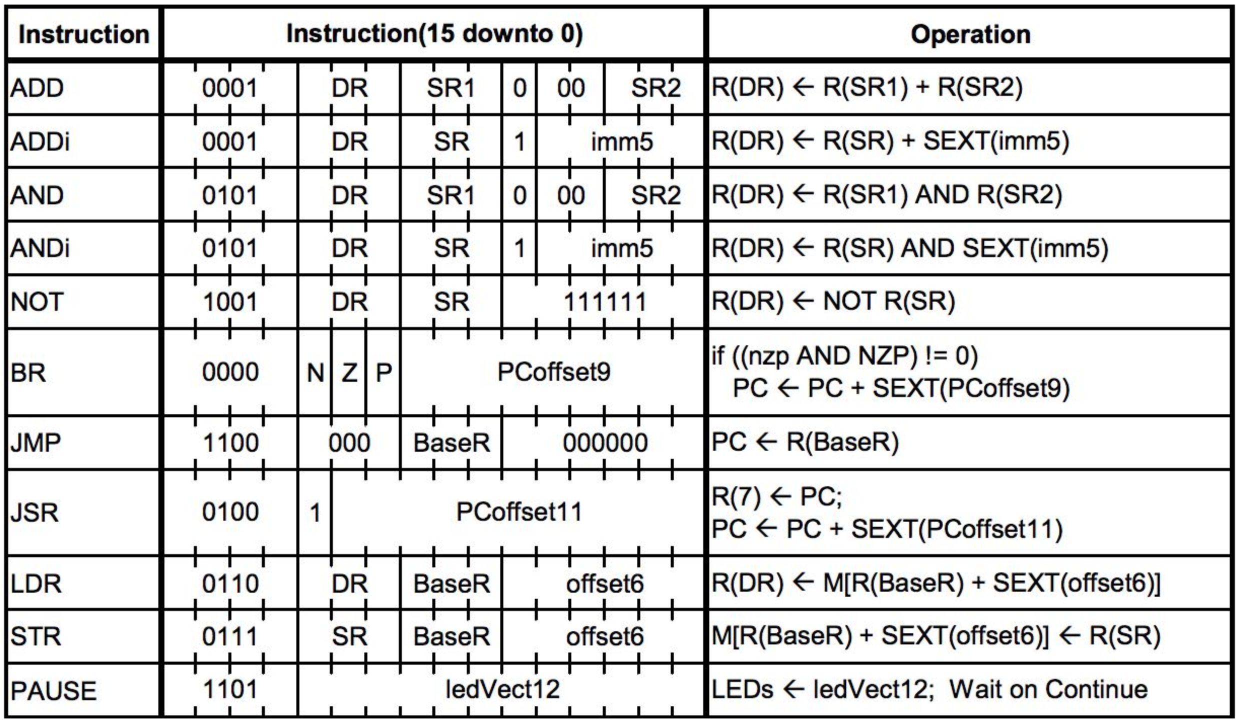
Michael Goldstein (mjgolds2), Noah Prince (nprince2)

AB8, Wednesday 9am

Jong Lim

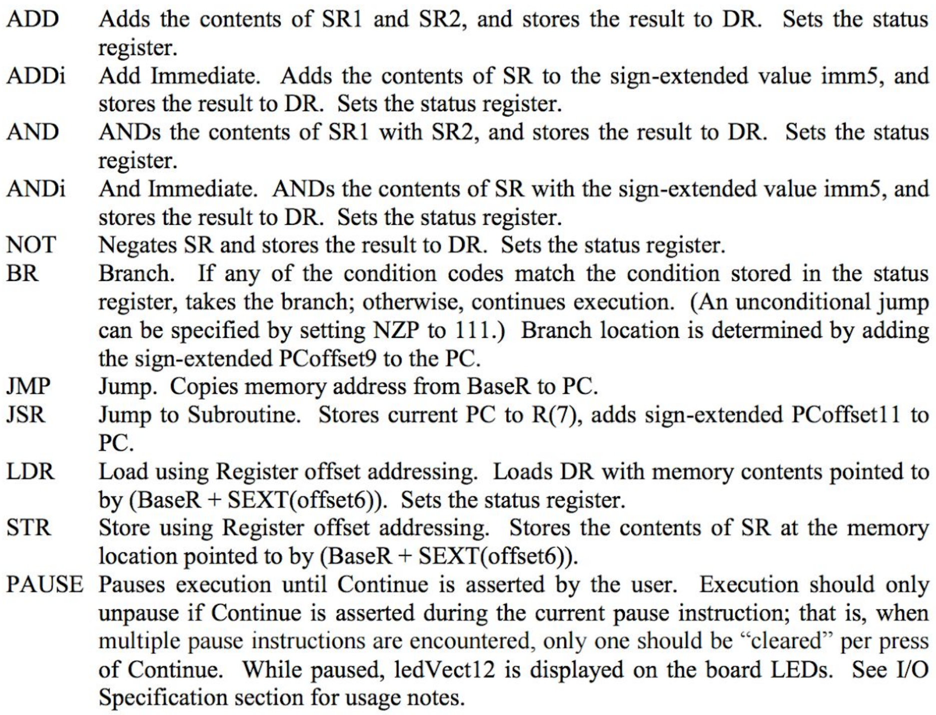
Introduction

The purpose of this lab was to build a simplified LC-3 (Little Computer-3) computer. This 16bit computer includes a register unit with seven 16-bit registers and memory access using the Altera Cyclone's built in SRAM. Here is the ISA for the SLC-3:



Instruction	Instruction(15 downto 0)	Operation
ADD	0001 DR SR1 0 00 SR2	R(DR) ← R(SR1) + R(SR2)
ADDi	0001 DR SR 1 imm5	R(DR) ← R(SR) + SEXT(imm5)
AND	0101 DR SR1 0 00 SR2	R(DR) ← R(SR1) AND R(SR2)
ANDi	0101 DR SR 1 imm5	R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001 DR SR 111111	R(DR) ← NOT R(SR)
BR	0000 N Z P PCoffset9	if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100 000 BaseR 000000	PC ← R(BaseR)
JSR	0100 1 PCoffset11	R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110 DR BaseR offset6	R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111 SR BaseR offset6	M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101 ledVect12	LEDs ← ledVect12; Wait on Continue

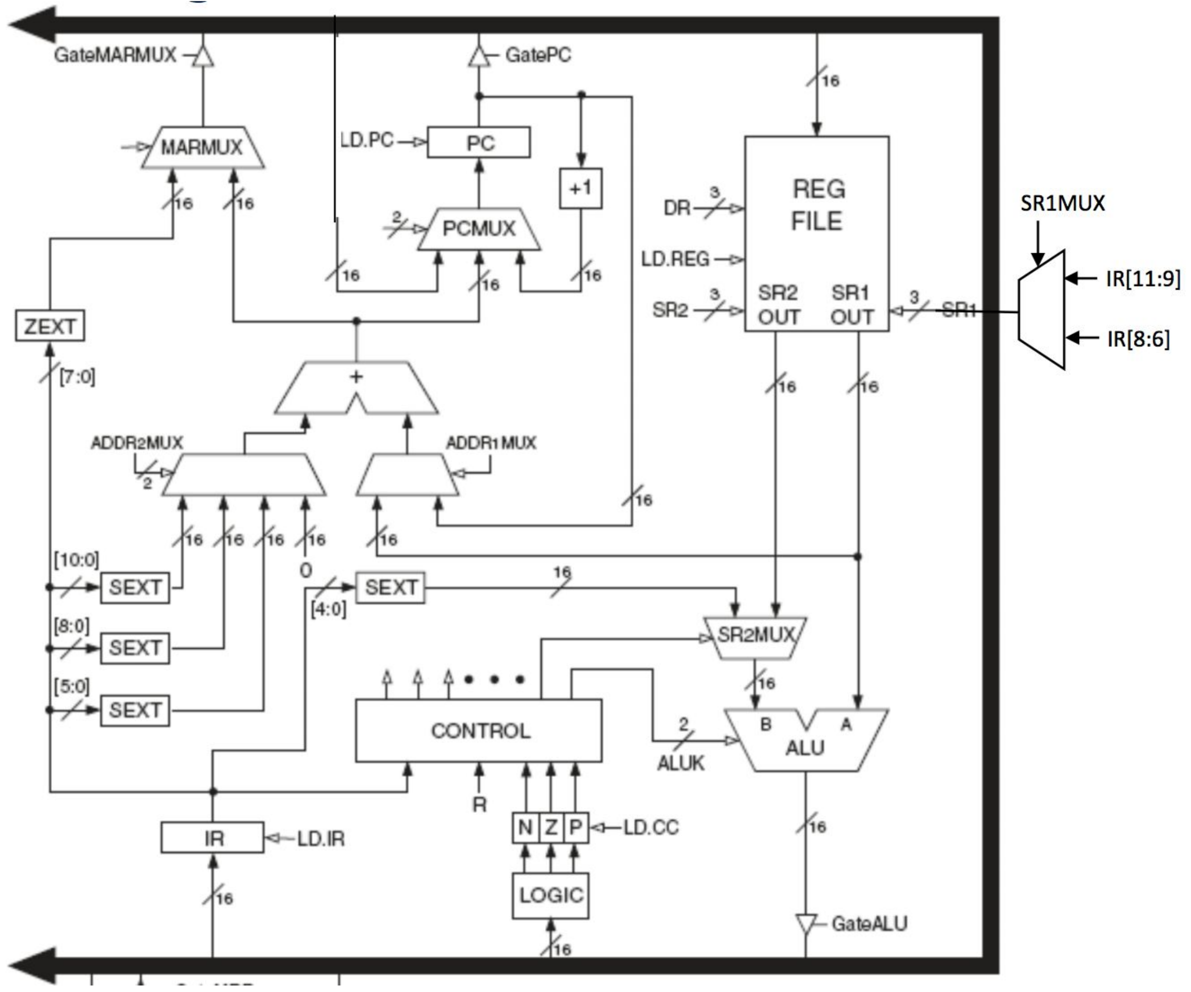
The description of the instructions are as follows:



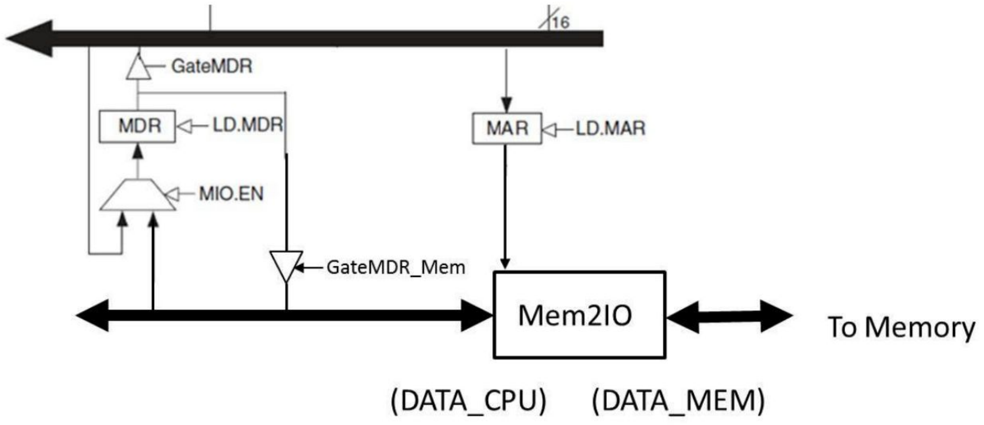
ADD	Adds the contents of SR1 and SR2, and stores the result to DR. Sets the status register.
ADDi	Add Immediate. Adds the contents of SR to the sign-extended value imm5, and stores the result to DR. Sets the status register.
AND	ANDs the contents of SR1 with SR2, and stores the result to DR. Sets the status register.
ANDi	And Immediate. ANDs the contents of SR with the sign-extended value imm5, and stores the result to DR. Sets the status register.
NOT	Negates SR and stores the result to DR. Sets the status register.
BR	Branch. If any of the condition codes match the condition stored in the status register, takes the branch; otherwise, continues execution. (An unconditional jump can be specified by setting NZP to 111.) Branch location is determined by adding the sign-extended PCoffset9 to the PC.
JMP	Jump. Copies memory address from BaseR to PC.
JSR	Jump to Subroutine. Stores current PC to R(7), adds sign-extended PCoffset11 to PC.
LDR	Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT(offset6)). Sets the status register.
STR	Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
PAUSE	Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be "cleared" per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.

Written Description of Circuit

The circuit consists of several modules; namely the high level SLC-3, the Register Unit/File, the datapath, the IDSU, ALU, several 16bit registers (PC, IR, MDR, MAR), and several MUX's and tristate buffers. This is the high level block diagram of the circuit:



And the bottom containing the memory interaction:



The combination of all of these elements provide a framework for all of the necessary commands to be completed. The challenge is propagating the signals in the correct locations to perform the required operation.

Written Description of Components

There are several major components used around the Control Unit inside the datapath.

ALU
inputs:
[15:0] A, B,
[1:0] ALUK

outputs:
[15:0] result

This unit takes the input A and B and puts the result of an operation in the output according to this truth table of ALUK values:

ALUK	**Operation**
00	(add) result = A + B
01	(and) result = A AND B
10	(not) result = NOT(A)
11	(pass A) result = A

The implementation of this unit is a combinational logic case/switch statement on ALUK, with each operation (A + B, A & B, ~A, A).

Adder
Inputs:
[15:0] ADD_in0, ADD_in1

output:
[15:0] ADD_out

This is implemented directly in the datapath, not as a module, but it will be referred to in the rest of the report as the Adder. The purpose of this small circuit is to add the values coming from ADDRMUXs. This circuit makes use of the SystemVerilog ‘+’ operator, and performs the operation in a combinational logic block

Register File
Inputs:
Clk, LD_REG,
[15:0] value,
[2:0] SR1, SR2, DR

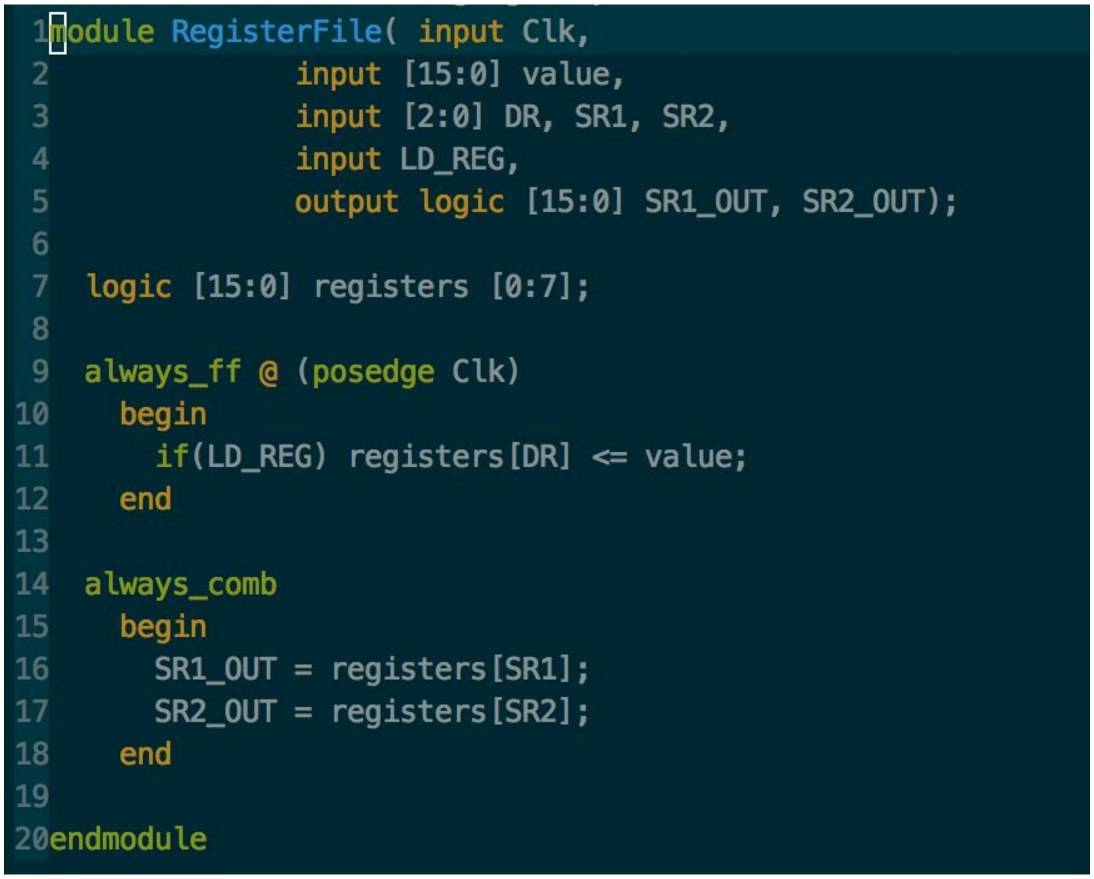
Outputs:
[15:0] SR1_OUT, SR2_OUT

This module is meant to retain the values of eight 16-bit registers. SR1, SR2, and DR are 3 bit values representing an integer between 0-7. These integers represent an index into the registers, I.E. 010 would represent Register 2.

The LD_REG signal controls loading into the registers; namely if LD_REG is high on the rising edge of Clk, the register indexed by DR will be loaded with ‘value’.

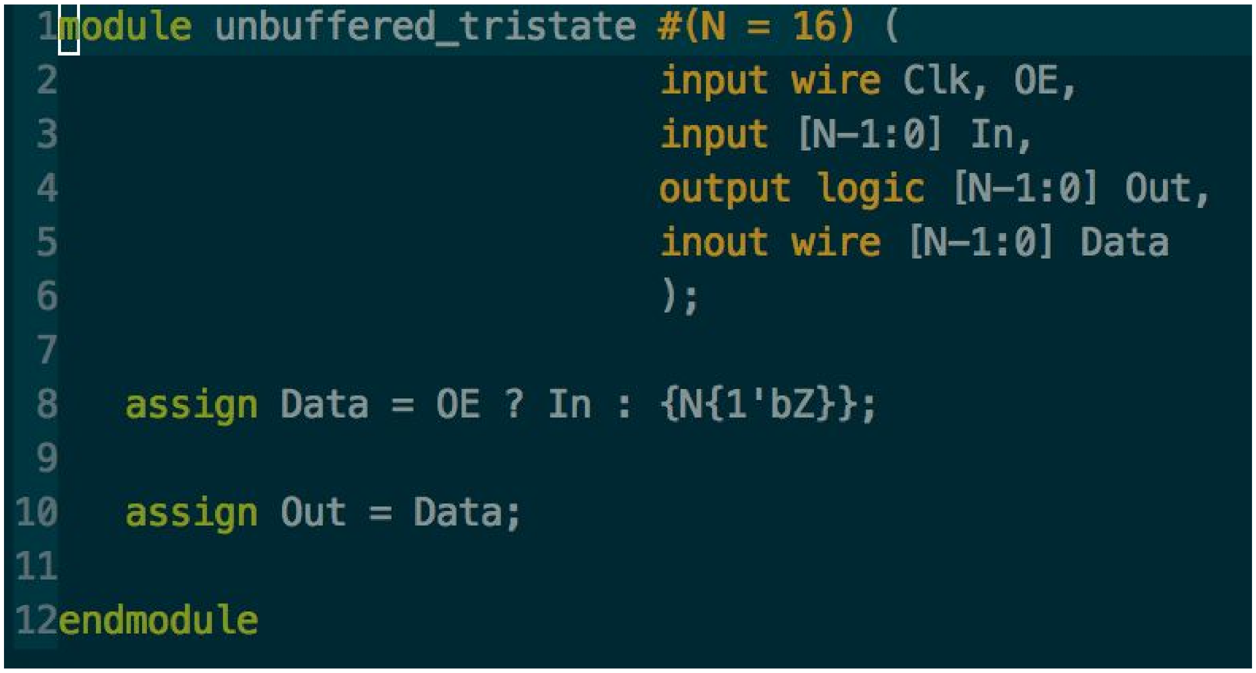
SR1_OUT and SR2_OUT output the value of the registers indexed by SR1 and SR2, respectively.

The implementation was done using SystemVerilog arrays. The implementation is as follows:



```
module RegisterFile( input Clk,
                     input [15:0] value,
                     input [2:0] DR, SR1, SR2,
                     input LD_REG,
                     output logic [15:0] SR1_OUT, SR2_OUT);

  logic [15:0] registers [0:7];

  always_ff @ (posedge Clk)
    begin
      if(LD_REG) registers[DR] <= value;
    end

  always_comb
    begin
      SR1_OUT = registers[SR1];
      SR2_OUT = registers[SR2];
    end

endmodule
```

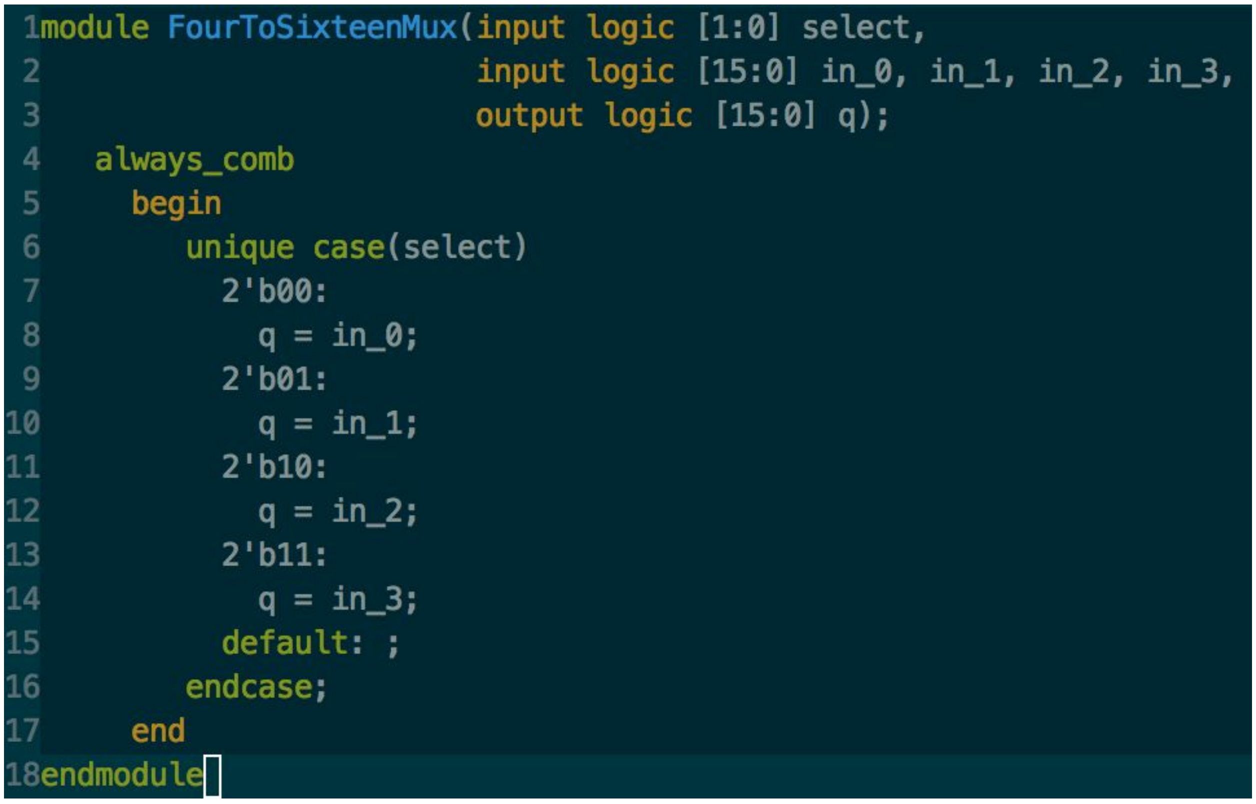
Tristate Buffers:
These are implemented using the code presented in lecture. They write high impedance value to the output when OE is low, and the value of their input when OE is high.



```
module unbuffered_tristate #(N = 16) (
                             input wire Clk, OE,
                             input [N-1:0] In,
                             output logic [N-1:0] Out,
                             inout wire [N-1:0] Data
                             );

  assign Data = OE ? In : {N{1'bZ}};

  assign Out = Data;

endmodule
```

Multiplexers

Two separate kinds of multiplexers are required to implement the SLC-3 block diagram as shown: One that takes two 16-bit inputs and outputs a single 16-bit line using a single bit select; and one that takes four 16-bit inputs and outputs a single 16-bit output using a two-bit select. In the code, these are called the TwoToSixteenMux and the FourToSixteenMux (the naming isn't perfect, as it doesn't specify that the four inputs are also 16 bits). These are implemented using a case/switch statement on the select input, and routing the relevant outputs. This is the code for the larger mux:



```
module FourToSixteenMux(input logic [1:0] select,
                        input logic [15:0] in_0, in_1, in_2, in_3,
                        output logic [15:0] q);
  always_comb
    begin
      unique case(select)
        2'b00:
          q = in_0;
        2'b01:
          q = in_1;
        2'b10:
          q = in_2;
        2'b11:
          q = in_3;
        default: ;
      endcase;
    end
endmodule
```

Datapath

The datapath combines all of these components as described in the block diagram, using the signals from the IDSU to perform all operations. Its inputs and outputs are as follows:

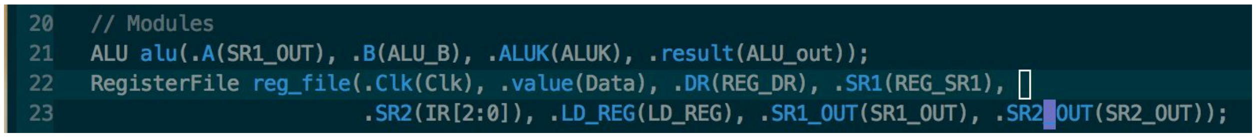
Inputs:
Load Signals (From IDSU): LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED,
Tristate Signals (From IDSU): GatePC, GateMDR, GateALU, GateMARMUX,
1 bit Mux Signals (From IDSU): SR2MUX, ADDR1MUX, MARMUX,
2 bit Mux Signals (From IDSU): [1:0] PCMUX, DRMUX, SR1MUX, ADDR2MUX,
ALU control (From IDSU): ALUK,
MIO_EN, BEN,
Input from Memory: [15:0] MDR_In,
[15:0] Data_Mem_In, Data_Mem_Out,
Reset, Clk

Outputs:
[15:0] MAR, MDR, IR, PC
[2:0] nzp

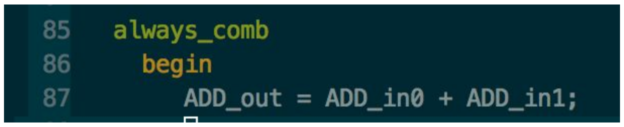
Inouts:
[15:0] Data

The inout wire Data represents the Data Bus seen in the block diagram. This unit outputs the values of MAR, MDR, IR, nzp, and PC as they are needed for Mem2IO reads/writes and control unit functionality.

The first step to a fully functioning datapath is to instantiate all modules on the path. This was done as follows:

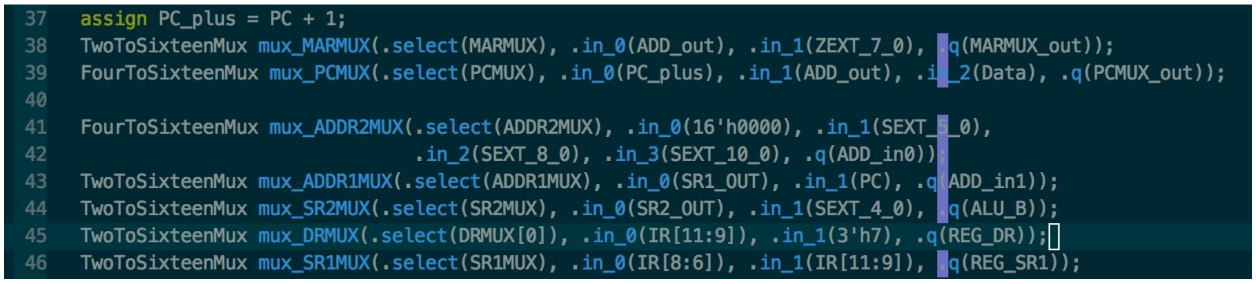

```
// Modules
ALU alu(.A(SR1_OUT), .B(ALU_B), .ALUK(ALUK), .result(ALU_out));
RegisterFile reg_file(.Clk(Clk), .value(Data), .DR(REG_DR), .SR1(REG_SR1), 
                      .SR2(IR[2:0]), .LD_REG(LD_REG), .SR1_OUT(SR1_OUT), .SR2_OUT(SR2_OUT));
```

And the adder:


```
always_comb
  begin
     ADD_out = ADD_in0 + ADD_in1;
```

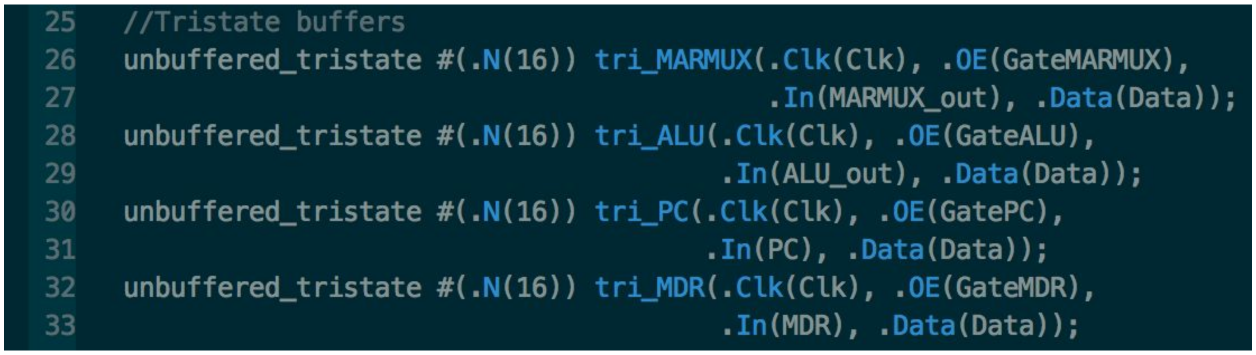
These modules are controlled by ALUK and LD_REG from the IDSU, and also by the registers indices specified by the IR.

The next step is to instantiate and wire up all of the MUXs, using the IDSU MUX signals to control them:



```verilog
assign PC_plus = PC + 1;
TwoToSixteenMux mux_MARMUX(.select(MARMUX), .in_0(ADD_out), .in_1(ZEXT_7_0), .q(MARMUX_out));
FourToSixteenMux mux_PCMUX(.select(PCMUX), .in_0(PC_plus), .in_1(ADD_out), .in_2(Data), .q(PCMUX_out));

FourToSixteenMux mux_ADDR2MUX(.select(ADDR2MUX), .in_0(16'h0000), .in_1(SEXT_5_0),
                              .in_2(SEXT_8_0), .in_3(SEXT_10_0), .q(ADD_in0));
TwoToSixteenMux mux_ADDR1MUX(.select(ADDR1MUX), .in_0(SR1_OUT), .in_1(PC), .q(ADD_in1));
TwoToSixteenMux mux_SR2MUX(.select(SR2MUX), .in_0(SR2_OUT), .in_1(SEXT_4_0), .q(ALU_B));
TwoToSixteenMux mux_DRMUX(.select(DRMUX[0]), .in_0(IR[11:9]), .in_1(3'h7), .q(REG_DR));
TwoToSixteenMux mux_SR1MUX(.select(SR1MUX), .in_0(IR[8:6]), .in_1(IR[11:9]), .q(REG_SR1));
```

Next, to keep multiple outputs from driving the data bus, add the tristate buffers, controlled by the gate signals from the IDSU:

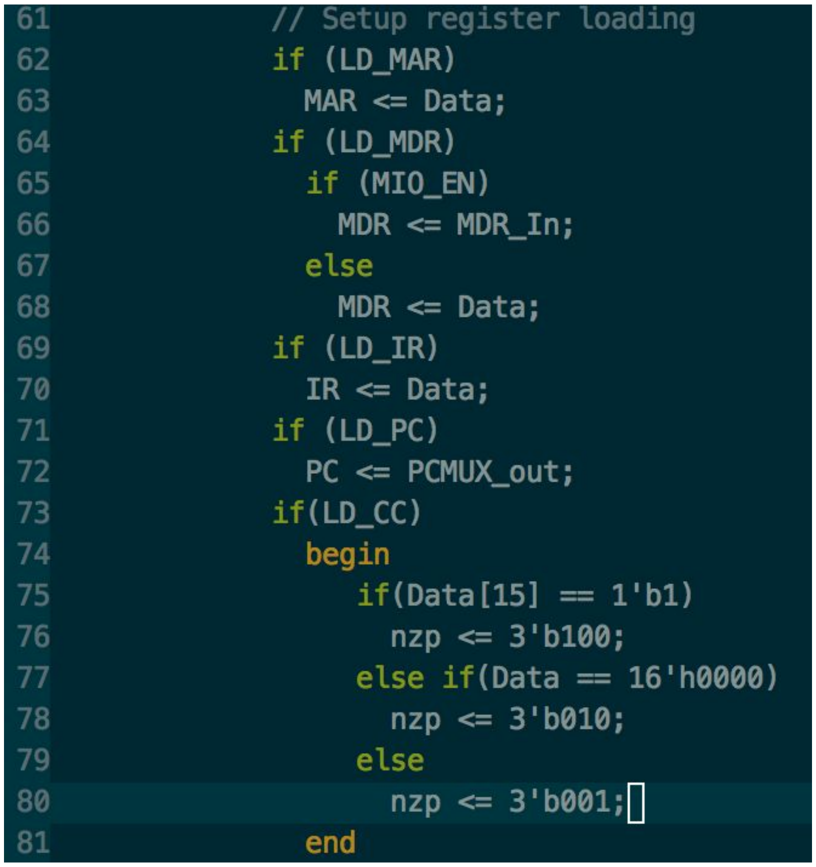


```verilog
//Tristate buffers
unbuffered_tristate #(.N(16)) tri_MARMUX(.Clk(Clk), .OE(GateMARMUX),
                                         .In(MARMUX_out), .Data(Data));
unbuffered_tristate #(.N(16)) tri_ALU(.Clk(Clk), .OE(GateALU),
                                      .In(ALU_out), .Data(Data));
unbuffered_tristate #(.N(16)) tri_PC(.Clk(Clk), .OE(GatePC),
                                     .In(PC), .Data(Data));
unbuffered_tristate #(.N(16)) tri_MDR(.Clk(Clk), .OE(GateMDR),
                                      .In(MDR), .Data(Data));
```

Next, the load signals from the IDSU are attached. These are attached using ‘if’ statements on the rising edge of the clock; only setting the register values when the load signal is high.

The nzp register is slightly different in that when LD_CC is high, it stores nzp based on the Data Bus value. Differentiating a negative value involves checking the final(sign) bit on the bus. A zero value means checking that all bits on the bus are zero. A positive value happens when the value is neither negative or zero.

This can all be seen here:

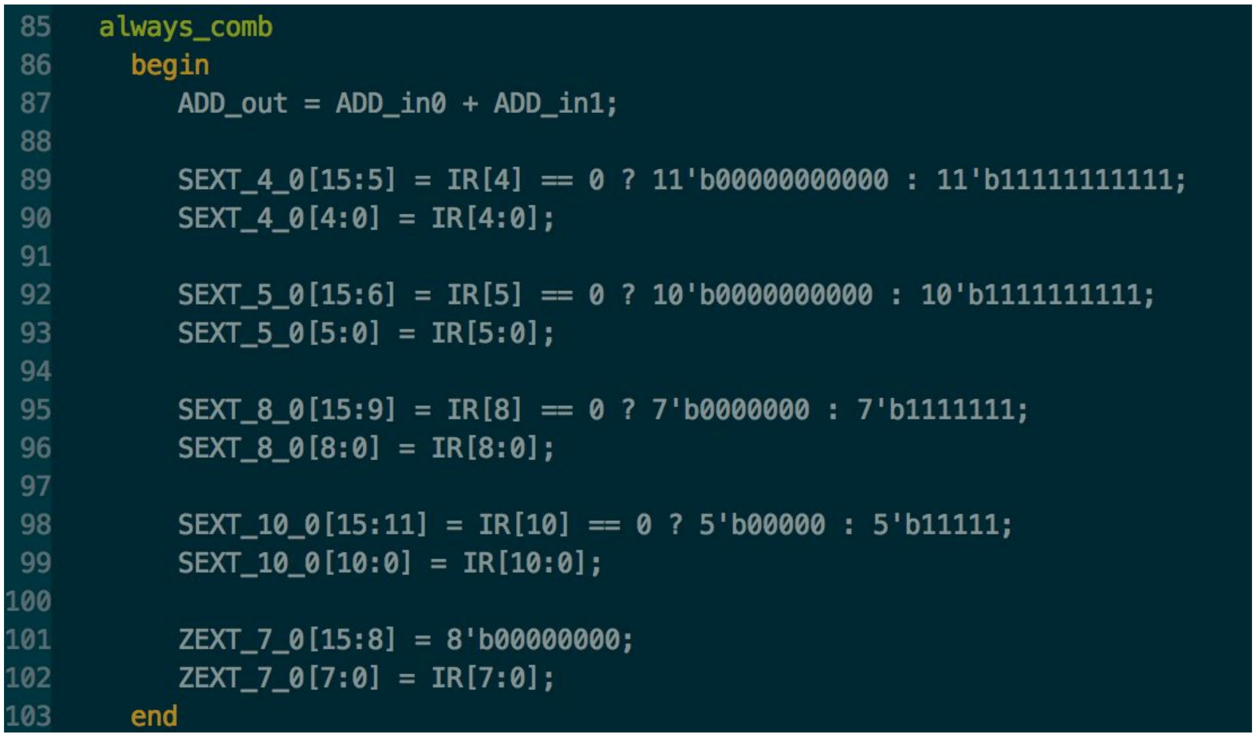


```
            // Setup register loading
            if (LD_MAR)
              MAR <= Data;
            if (LD_MDR)
              if (MIO_EN)
                MDR <= MDR_In;
              else
                MDR <= Data;
            if (LD_IR)
              IR <= Data;
            if (LD_PC)
              PC <= PCMUX_out;
            if(LD_CC)
              begin
                 if(Data[15] == 1'b1)
                   nzp <= 3'b100;
                 else if(Data == 16'h0000)
                   nzp <= 3'b010;
                 else
                   nzp <= 3'b001;
              end
```

The only wires that remain undefined from the block diagram are the SEXT and ZEXT values. Those must be implemented in an always_comb block.

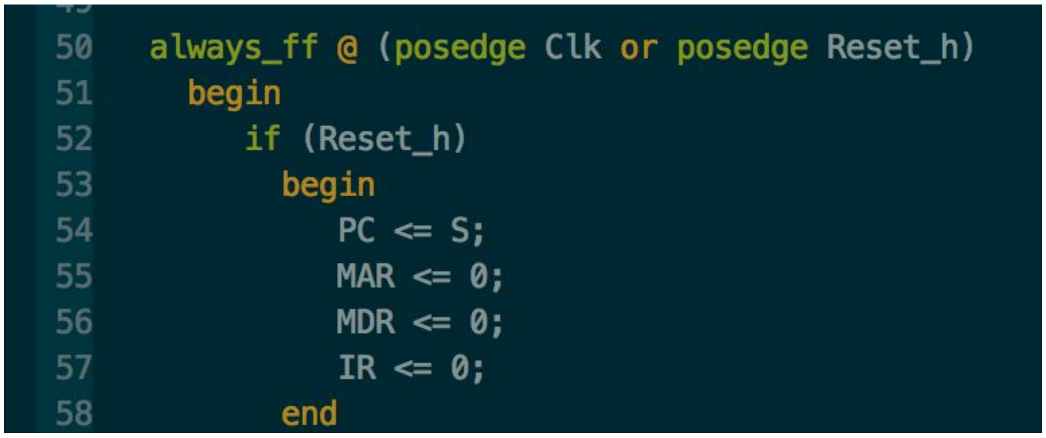
A ZEXT just means adding zeroes to the end of the bits.

The SEXT requires logic that adds 1's when the final bit is a 1, and zeros when it is a zero. This preserves the value of two's complement numbers while extending them out to sixteen bits. This can be seen here:



```
always_comb
  begin
     ADD_out = ADD_in0 + ADD_in1;

     SEXT_4_0[15:5] = IR[4] == 0 ? 11'b00000000000 : 11'b11111111111;
     SEXT_4_0[4:0] = IR[4:0];

     SEXT_5_0[15:6] = IR[5] == 0 ? 10'b0000000000 : 10'b1111111111;
     SEXT_5_0[5:0] = IR[5:0];

     SEXT_8_0[15:9] = IR[8] == 0 ? 7'b0000000 : 7'b1111111;
     SEXT_8_0[8:0] = IR[8:0];

     SEXT_10_0[15:11] = IR[10] == 0 ? 5'b00000 : 5'b11111;
     SEXT_10_0[10:0] = IR[10:0];

     ZEXT_7_0[15:8] = 8'b00000000;
     ZEXT_7_0[7:0] = IR[7:0];
  end
```

The next step is to define the action when the 'Reset' button is pressed. As defined by the documentation, a reset sets the PC to the input switches, and clears all other registers. This is implemented here:



```
always_ff @ (posedge Clk or posedge Reset_h)
  begin
     if (Reset_h)
        begin
           PC <= S;
           MAR <= 0;
           MDR <= 0;
           IR <= 0;
        end
```

With these components properly attached, all that is left is to drive this circuit using the IDSU.

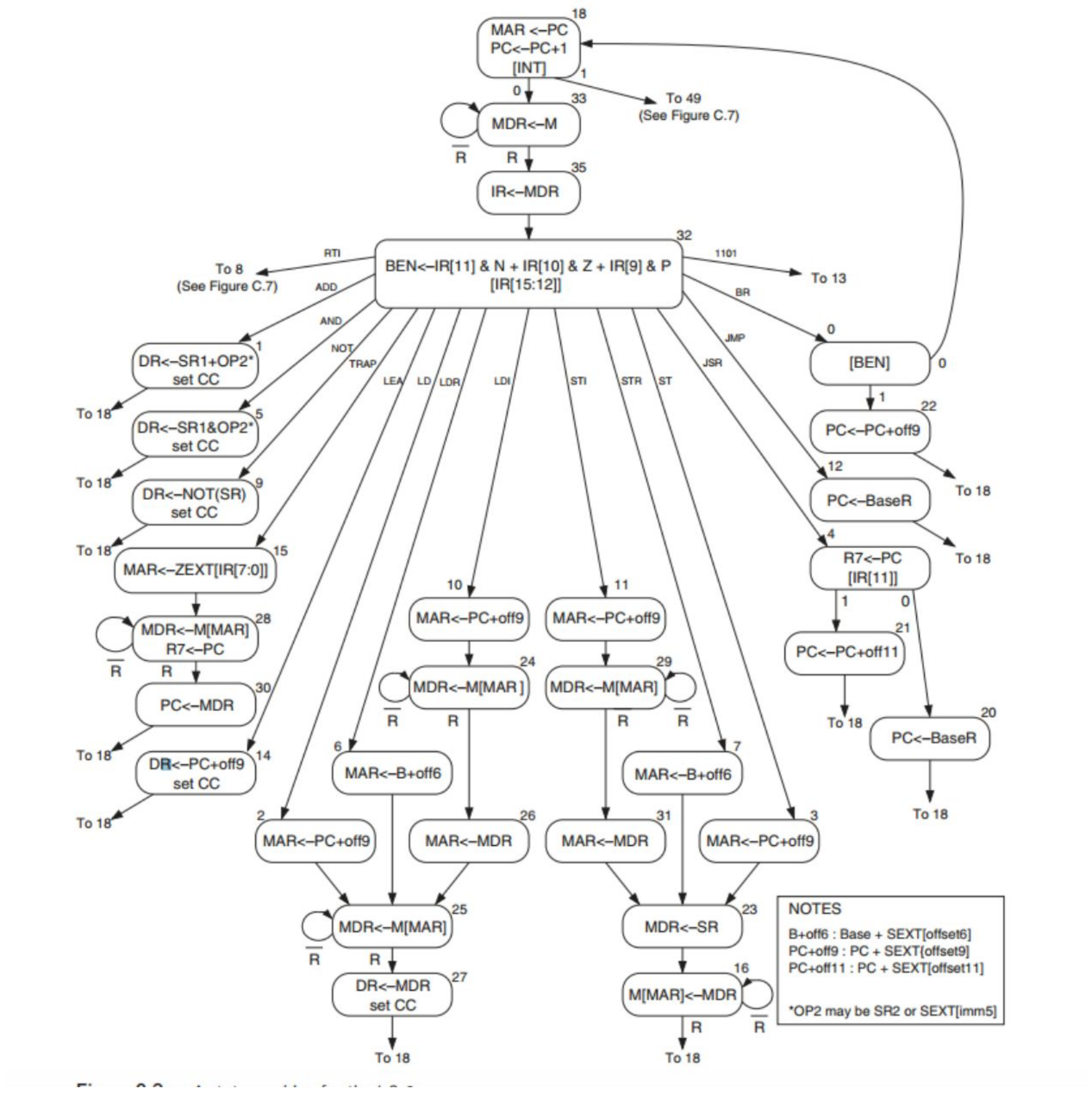
IDSU
Inputs:
Clk, Reset, Run, Continue,
[3:0] Opcode,
[15:0] IR,

Outputs:
Register load signals: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_PC
Register Unit load signal: LD_REG,
Tristate buffer gate signals: GatePC, GateMDR, GateALU, GateMARMUX,
Mux signals: PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, MARMUX
ALU input: ALUK,
Memory control signals: Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE
For displaying pause state: LED

Each command has an opcode; the IDSU is used to control all MUX's, tristate buffers, ALU, Adder, Register File, and Load Signals required to execute that command, based on the opcode. The combination of these signals routes outputs to the data bus and into the desired location for the operation.

The PAUSE state is an adaptation from the given IDSU code for PAUSEIR. It functions the same way, staying in the PAUSE1 state while continue is low, continuing to PAUSE2 when Continue is high; staying in the PAUSE2 state while continue is high, and continuing to the instruction fetch phase when it goes low. This essentially requires a full cycle of the Continue button.

The other states function according to the following state diagram:



First, to make setting signals easier, the IDSU has a default value for all signals. These default values keep anything from interfering with or receiving data from the data bus.

The next step for the IDSU is to determine the control signals necessary to perform the operations at each state. This is done by tracing out the datapath of each operation on the LC-3 block diagram. Here are the signal definitions for each operation:

MAR <- PC
PC <- PC + 1:

GatePC = 1'b1
LD_MAR = 1'b1
PCMUX = 2'b00
LD_PC = 1'b1

This ensures that the PC is output onto the data bus, and that MAR is loaded with the value from the data bus.

MDR <- M:

State 1:

MEM_OE = 1’b0

State 2:

MEM_OE = 1’b0
LD_MDR = 1’b1

There are two states for this one because memory reads take two cycles. We only want to load MDR when the memory output is correct.

IR <- MDR:

GateMDR = 1'b1
LD_IR = 1'b1

This ensures that MDR is on the data bus, and that IR is loaded from the data bus

PAUSE1 and PAUSE2:

LED = IR[11:0]

This puts the designated value from the command (see ISA) on the LEDs

State 32:

LED_BEN = 1’b1;

DR <- SR1 + OP2:

SR2MUX = IR[5];
ALUK = 2'b00;
GateALU = 1'b1;
LD_REG = 1'b1;
LD_CC = 1'b1;

This gives the ALU the SR1 and SR2 or imm5 depending on IR[5] as input. It also tells the ALU to perform an ADD operation on it’s inputs. It then places the ALU output on the data bus, which is then received by the register unit and stored in the DR (because of the LED_REG) signal.

DR <- SR1&OP2

SR2MUX = IR_5;
GateALU = 1'b1;
LD_REG = 1'b1;
ALUK = 2'b01;

```
        LD_CC = 1'b1;
```
This is similar to the ADD operation, except it tells the ALU to perform the AND operation using the signal 2’b01.

DR <- NOT(SR):
```
        SR2MUX = IR_5;
        GateALU = 1'b1;
        LD_REG = 1'b1;
        ALUK = 2'b10;
        LD_CC = 1'b1;
```
Again, this is very similar to the ADD operation, except it tells the ALU to perform the NOT operation using the signal 2’b10

PC <- PC + Off9:
```
        ADDR2MUX = 2'b10;
        ADDR1MUX = 1'b1;
        LD_PC = 1'b1;
        PCMUX = 2'b01;
```
This takes the Off9 signal from IR, runs it through the Adder and out onto the data bus. Then, because LD_PC is 1, PC receives this value from the bus.

PC <- BaseR:
```
        PCMUX = 2'b10;
        LD_PC = 1'b1;
        ALUK = 2'b11;
        GateALU = 1'b1;
```
This takes the Base Register from the Register Unit, passes it through the ALU via the ‘Pass A’ option, and routes that value to the data bus. PC is then loaded from the data bus.

R7 <- PC
```
        DRMUX = 2'b01;
        GatePC = 1'b1;
        LD_REG = 1'b1;
```
Using the DRMUX signal of 2’b01, R7 is set as the destination register. The value from PC is then loaded onto the data bus and into the register unit.

PC <- PC + Off11:
```
        ADDR2MUX = 2'b11;
        ADDR1MUX = 1'b1;
        PCMUX = 2'b01;
        LD_PC = 1'b1;
```
This takes the Off11 signal from IR, runs it through the Adder and out onto the data bus. Then, because LD_PC is 1, PC receives this value from the bus.

MAR <- BaseR + Off6:

```
ADDR2MUX=2'b01;
GateMARMUX=1'b1;
LD_MAR=1'b1;
```

This takes the base register, routes it through the ADDR1 MUX, uses the Adder to add it with SEXT(Off6), and then routes that from the MARMUX to the data bus. The value from the data bus is loaded into MAR.

MDR <- M[MAR]:

```
State 1:
        MEM_OE = 1’b0
State 2:
        MEM_OE = 1’b0
        LD_MDR = 1’b1
```

This gets a read from the on board memory and loades it into MDR.

DR <- MDR, set cc

```
LD_REG = 1'b1;
GateMDR = 1'b1;
LD_CC = 1'b1;
```

This puts MDR on the data bus, loads it into the register, and also sets the CC registers based on the data bus value.

MDR <- SR:

```
SR1MUX = 1'b1;
ALUK = 2'b11;
GateALU = 1'b1;
LD_MDR = 1'b1;
```

This passes SR through the ALU using the ‘Pass A’ option, puts it on the data bus, and loads it into MDR.

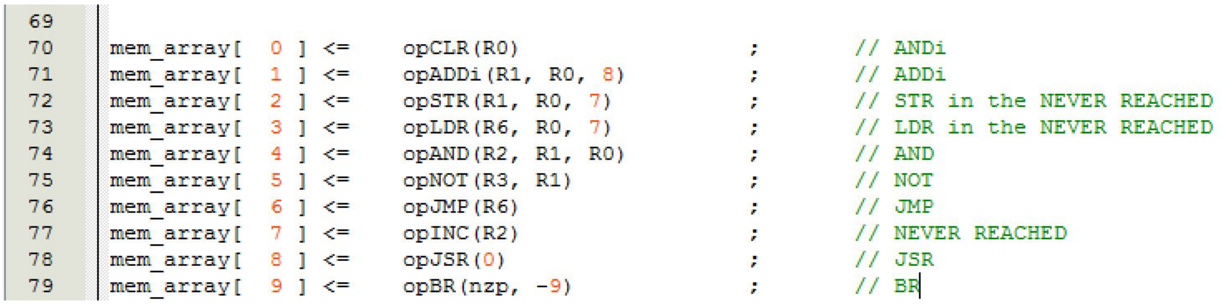
M[MAR] <- MDR (for three states)

```
GateMDR = 1'b1;
Mem_WE = 1'b0;
```

This puts MDR on the data bus, and loads the data bus value into memory.

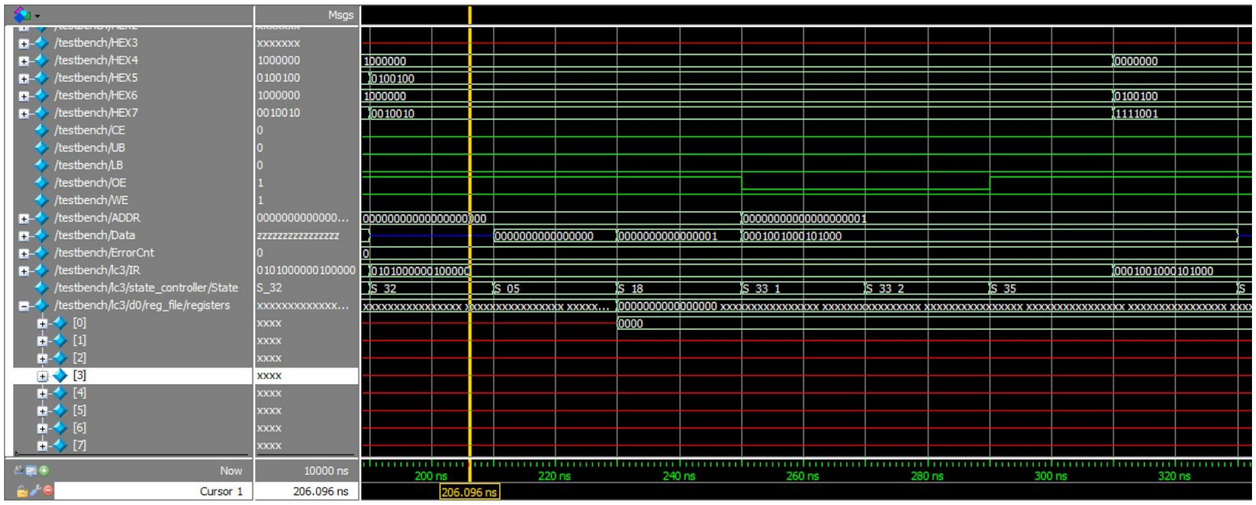
Simulations:

The following simulations of all commands were obtained using the following test program:

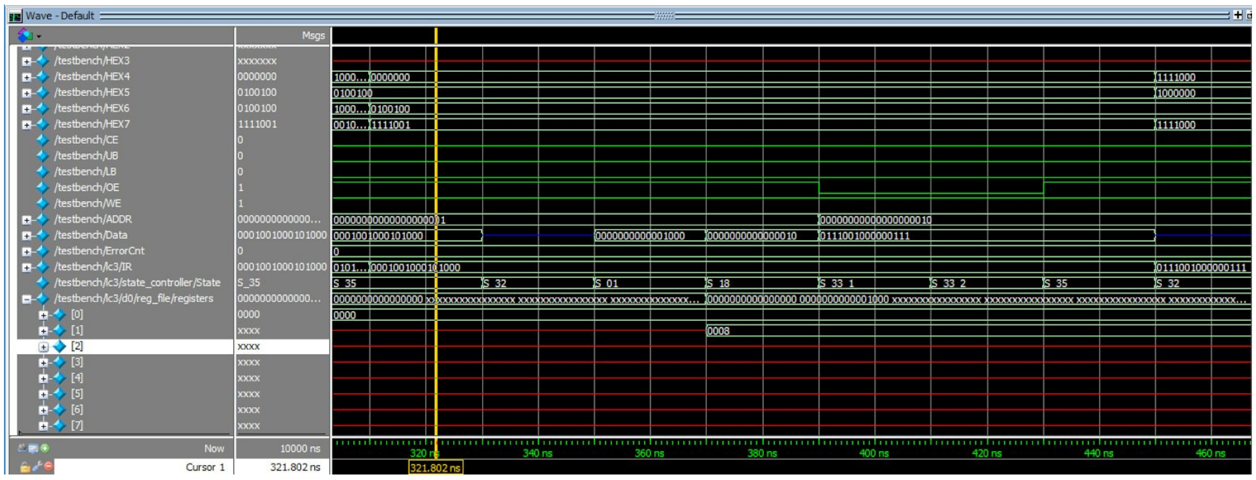


```
mem_array[  0 ] <=     opCLR(R0)                ;       // ANDi
mem_array[  1 ] <=     opADDi(R1, R0, 8)        ;       // ADDi
mem_array[  2 ] <=     opSTR(R1, R0, 7)         ;       // STR in the NEVER REACHED
mem_array[  3 ] <=     opLDR(R6, R0, 7)         ;       // LDR in the NEVER REACHED
mem_array[  4 ] <=     opAND(R2, R1, R0)        ;       // AND
mem_array[  5 ] <=     opNOT(R3, R1)            ;       // NOT
mem_array[  6 ] <=     opJMP(R6)                ;       // JMP
mem_array[  7 ] <=     opINC(R2)                ;       // NEVER REACHED
mem_array[  8 ] <=     opJSR(0)                 ;       // JSR
mem_array[  9 ] <=     opBR(nzp, -9)            ;       // BR
```

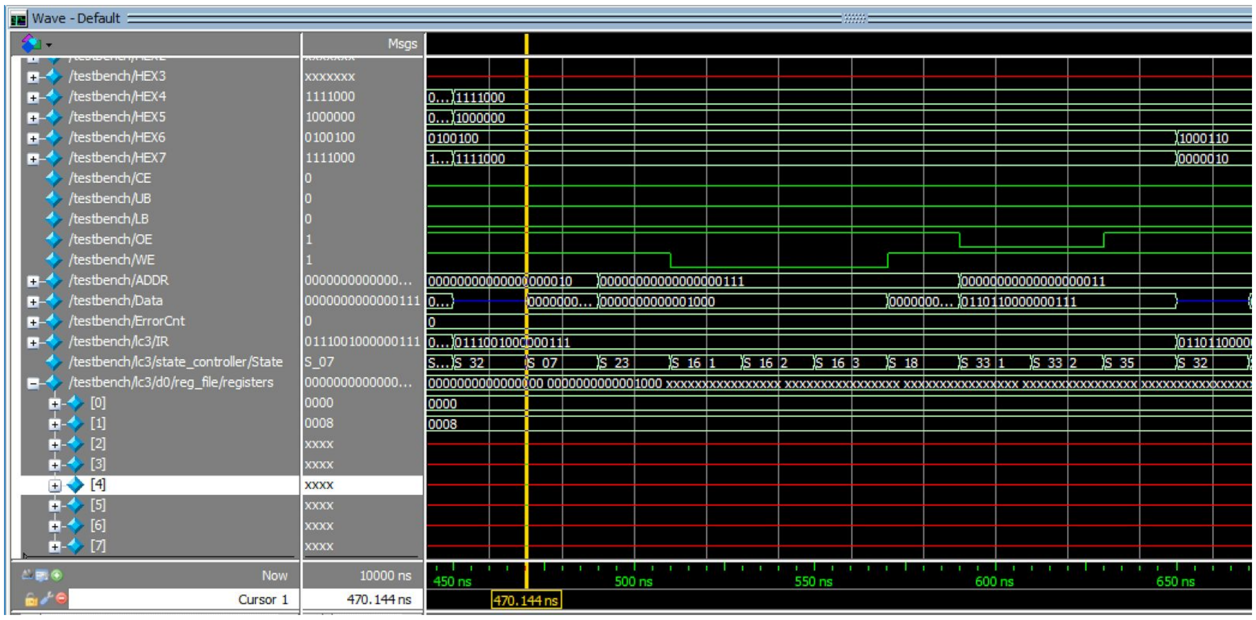
CLR/ANDi:



ADDi:

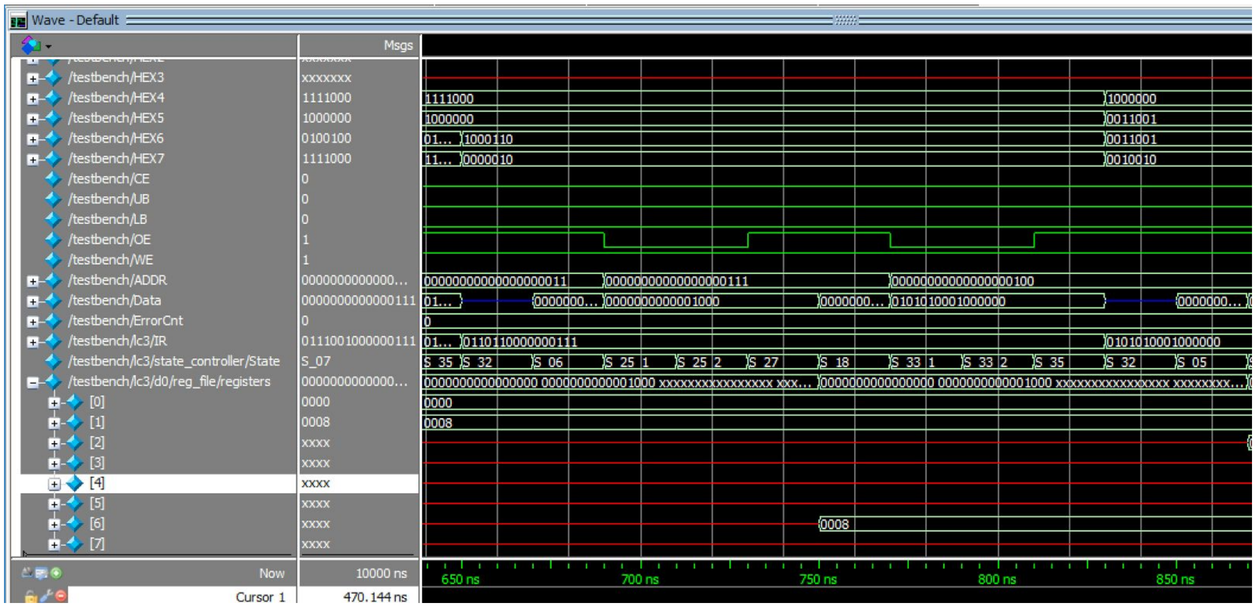


STR:



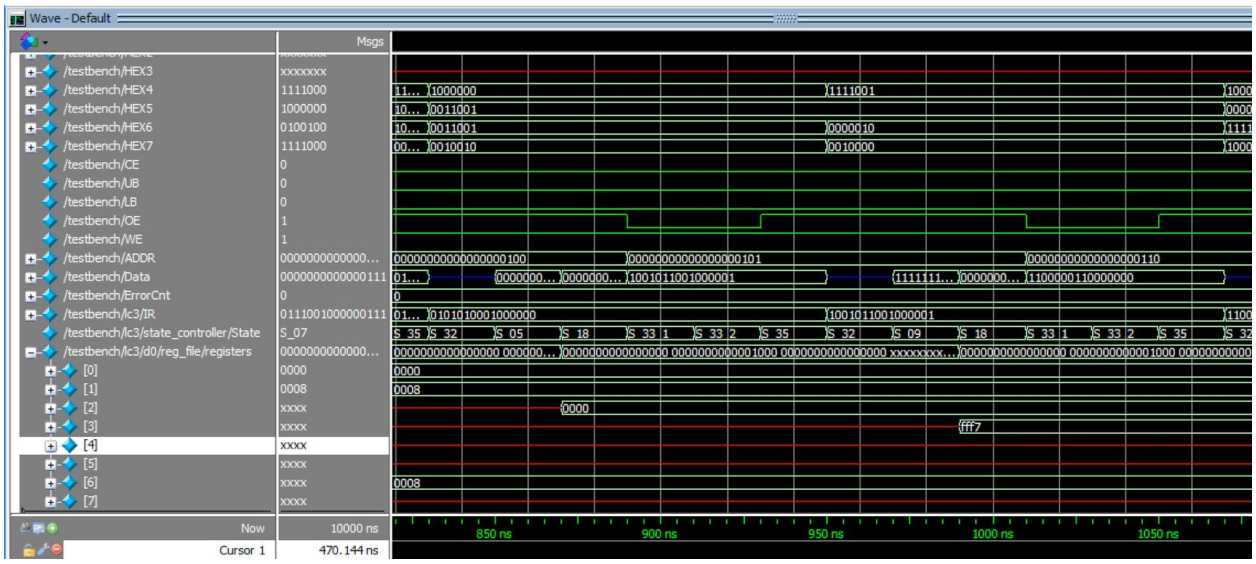
```
Wave - Default
                                       Msgs
/testbench/HEX3                        xxxxxxx
/testbench/HEX4                        1111000
/testbench/HEX5                        1000000
/testbench/HEX6                        0100100
/testbench/HEX7                        1111000
/testbench/CE                          0
/testbench/UB                          0
/testbench/LB                          0
/testbench/OE                          1
/testbench/WE                          1
/testbench/ADDR                        0000000000000...
/testbench/Data                        0000000000000111
/testbench/ErrorCnt                    0
/testbench/lc3/IR                      0111001000000111
/testbench/lc3/state_controller/State  S_07
/testbench/lc3/d0/reg_file/registers   0000000000000...
  [0]                                  0000
  [1]                                  0008
  [2]                                  xxxx
  [3]                                  xxxx
  [4]                                  xxxx
  [5]                                  xxxx
  [6]                                  xxxx
  [7]                                  xxxx
Now                                    10000 ns
Cursor 1                               470.144 ns
```

LDR:



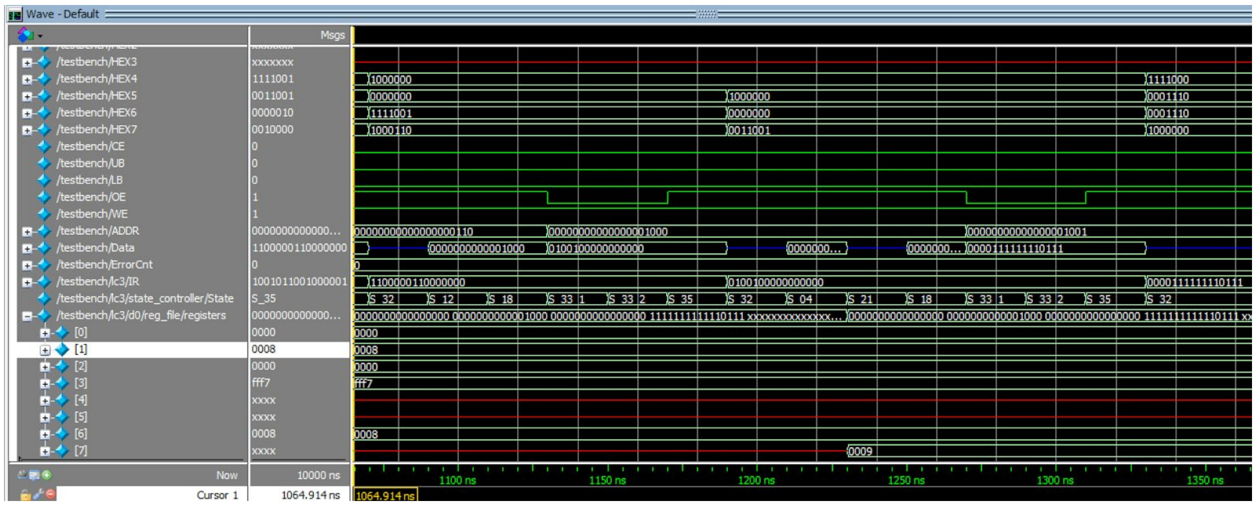
```
Wave - Default
                                       Msgs
/testbench/HEX3                        xxxxxxx
/testbench/HEX4                        1111000
/testbench/HEX5                        1000000
/testbench/HEX6                        0100100
/testbench/HEX7                        1111000
/testbench/CE                          0
/testbench/UB                          0
/testbench/LB                          0
/testbench/OE                          1
/testbench/WE                          1
/testbench/ADDR                        0000000000000...
/testbench/Data                        0000000000000111
/testbench/ErrorCnt                    0
/testbench/lc3/IR                      0111001000000111
/testbench/lc3/state_controller/State  S_07
/testbench/lc3/d0/reg_file/registers   0000000000000...
  [0]                                  0000
  [1]                                  0008
  [2]                                  xxxx
  [3]                                  xxxx
  [4]                                  xxxx
  [5]                                  xxxx
  [6]                                  xxxx
  [7]                                  xxxx
Now                                    10000 ns
Cursor 1                               470.144 ns
```

AND and NOT:



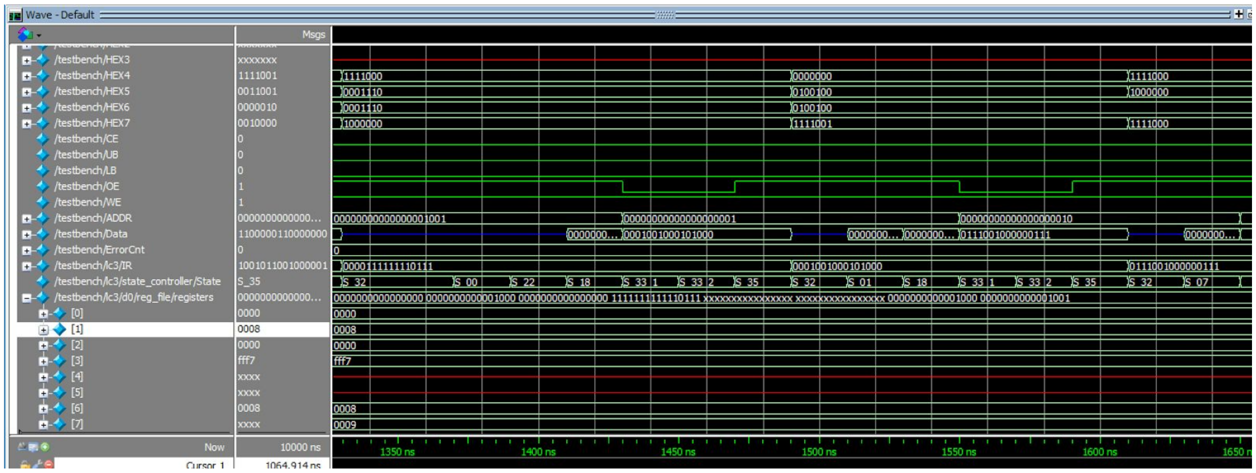
```
Wave - Default
                                             Msgs
/testbench/HEX3                              xxxxxxx
/testbench/HEX4                              1111000
/testbench/HEX5                              1000000
/testbench/HEX6                              0100100
/testbench/HEX7                              1111000
/testbench/CE                                0
/testbench/UB                                0
/testbench/LB                                0
/testbench/OE                                1
/testbench/WE                                1
/testbench/ADDR                              0000000000000...
/testbench/Data                              0000000000000111
/testbench/ErrorCnt                          0
/testbench/lc3/IR                            0111001000000111
/testbench/lc3/state_controller/State        S_07
/testbench/lc3/d0/reg_file/registers         0000000000000...
  [0]                                        0000
  [1]                                        0008
  [2]                                        xxxx
  [3]                                        xxxx
  [4]                                        xxxx
  [5]                                        xxxx
  [6]                                        xxxx
  [7]                                        xxxx
Now                                          10000 ns
Cursor 1                                     470.144 ns
```

JMP and JSR:



```
Wave - Default
                                             Msgs
/testbench/HEX3                              xxxxxxx
/testbench/HEX4                              1111001
/testbench/HEX5                              0011001
/testbench/HEX6                              0000010
/testbench/HEX7                              0010000
/testbench/CE                                0
/testbench/UB                                0
/testbench/LB                                0
/testbench/OE                                1
/testbench/WE                                1
/testbench/ADDR                              0000000000000...
/testbench/Data                              1100000110000000
/testbench/ErrorCnt                          0
/testbench/lc3/IR                            1001011001000001
/testbench/lc3/state_controller/State        S_35
/testbench/lc3/d0/reg_file/registers         0000000000000...
  [0]                                        0000
  [1]                                        0008
  [2]                                        0000
  [3]                                        fff7
  [4]                                        xxxx
  [5]                                        xxxx
  [6]                                        0008
  [7]                                        xxxx
Now                                          10000 ns
Cursor 1                                     1064.914 ns
```

JSR, BR ,and the command after BR jumps up:



```
Wave - Default
                                              Msgs
/testbench/HEX3                               xxxxxxx
/testbench/HEX4                               1111001
/testbench/HEX5                               0011001
/testbench/HEX6                               0000010
/testbench/HEX7                               0010000
/testbench/CE                                 0
/testbench/UB                                 0
/testbench/LB                                 0
/testbench/OE                                 1
/testbench/WE                                 1
/testbench/ADDR                               0000000000000...
/testbench/Data                               1100000110000000
/testbench/ErrorCnt                           0
/testbench/lc3/IR                             1001011001000001
/testbench/lc3/state_controller/State         S_35
/testbench/lc3/d0/reg_file/registers          0000000000000...
  [0]                                         0000
  [1]                                         0008
  [2]                                         0000
  [3]                                         fff7
  [4]                                         xxxx
  [5]                                         xxxx
  [6]                                         0008
  [7]                                         xxxx
Now                                           10000 ns
Cursor 1                                      1064.914 ns
```

Post Lab

1)

LUT	647
DSP	0
Memory (BSP)	0
Flip-Flop	268
Frequency	131.91 MHz
Static Power	98.64 mW
Dynamic Power	0 mW
Total Power	172.29 mW

2) Mem2IO is used to interface our SLC3 with the Altera Cyclon SRAM. It also maps hardware on the board to memory locations.

The difference between a BR and a JMP is that the BR only changes the PC register when specified NZP conditions are met (IE the last calculation was either negative, zero, or positive, respectively).

Conclusion

Week 1 went very well, we were surprised at how quickly we were able to complete the task. With the premade IDSU signals, we just needed to implement tristate buffers and multiplexers. The only difficulty was discerning how to replace Mem2IO with the test memory.

Week 2 was more difficult. We were careful to print out and trace every data path; this worked well except for some confusion around passing values through the ALU vs the Adder for Offset addition instructions.

A large difficulty we faced, and lost time to was that the memory writing program did not always work; namely that it took several (>2) writes before we could be confident that the write was successful.

Finally, the main difficulty; one that cost us several hours, was actually a failure in the documentation. This was a particularly nasty bug to track down. The documentation for the lab stated that it would only take two state cycles to perform memory read/write operations; and that it was guaranteed in that time to be finished. This is the case for reads. This is NOT the case for writes. Once we switched to three states for each write, all of our problems disappeared.

The documentation should be revised for future classes to reflect both of these findings.

Aside from these issues, the lab was very informative. Coming out of this lab, we have a complete idea of how a small processing chip functions; this knowledge is invaluable for a computer engineer.