Making an Eight-Bit Computer:

Using Verilog and Assembly Coding Styles

Courtney Johnson
New Mexico Tech
Electrical Engineering Dept.
Socorro, NM
cjohnson@nmt.edu

***Abstract*—Using Verilog code, an eight-bit computer was created with its own unique assembly language to produce specific light patterns. To do this, computer components such as an arithmetic logic unit, a control unit, and multiple registers were written and connected using a block diagram file within Altera's computer program Quartus II. Each computer component's logic was navigated using a finite state machine (FSM) and fed through a series of registers and commands to create an output. The output was the successful interaction of all the computer components and was measured with the use of LED lights on a Field-Programmable Gate Array (FPGA).**

Keywords— assembly language, arithmetic logic unit, control unit, field-programmable gate array, finite state machine, registers

I. Introduction

The microcomputer designed in this final project uses eight-bit logic and operation codes (op codes) to perform functions. To test the functionality of the computer, assembly language was written to implement a code which would cycle through LED's on an FPGA. This was done using a block diagram, as seen in Figure 7.

The computer was built using a control unit, an ALU, registers, a clock divider, a multiplexer, and a memory component, as seen in Figure 1. The control unit was the source of instructions via an FSM. Two case statements were used to implement the fetch, reset, execution 1, and execution 2 stages. The control unit also controls different load signals within the computer to allow for complete command of registers. When an instruction is fed into the control unit, it uses the case statement to determine which command is being fed in and uses that feedback to switch into another case statement and begin the executables. Once in either execution state, a series of signals are either turned on or off to prepare the ALU and the other registers. The ALU performs the logic for the instruction that is being sent to it. Its output is then fed to the appropriate registers. There are five eight-bit registers and they are all used to navigate the current instruction, as well as store and access memory. The Program Counter (PC) register stores the current location in the program memory. The Memory Addressing Register (MAR) collects data from arbitrary memory without losing the current memory address spot pointed to by the PC register. The Accumulator A register (ACCA) serves as temporary data storage for arithmetic operations. The Instruction Register (IR or INST) stores instructions for the executables in the FSM. The C and Z registers are one bit registers used to store the carry and zero flags, respectively, from the ALU operations. A clock divider is used to slow the cycle of the FPGA from 50 MHz to approximately 100 Hz to make the output display slow enough to be seen. The multiplexer is used to select where data is going currently and allows for storage into memory. The memory block component is a series of addresses which hold op codes for the different instructions. The memory unit is the basis for the light program because, as the pc register shifts to a new memory address and reads it, the memory component tells the computer which command it must use to continue the program.

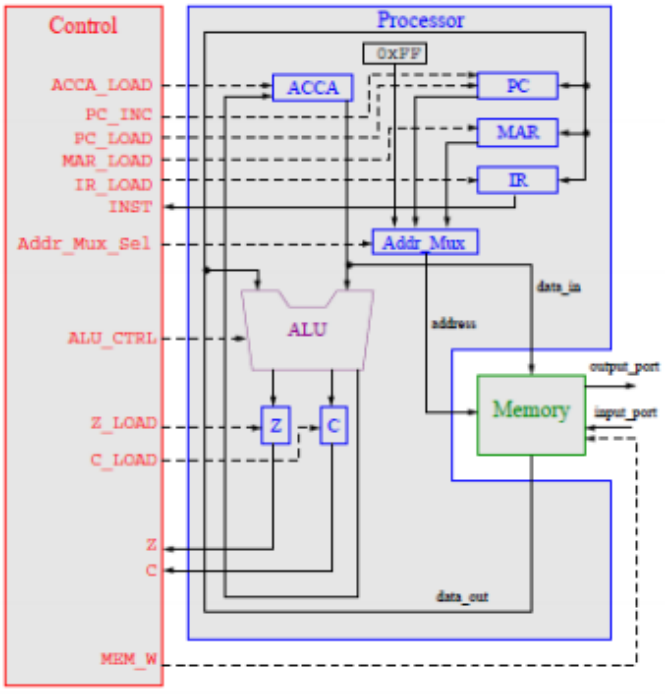

Figure 1: Simple Computer

To prove that the different components work together, two methods of observation were implemented. On the computer, a waveform was generated to show the values of all component inputs and outputs. The purpose of this is to allow for a step-by-step logic check of each function. After the waveform was checked for accuracy, the program was

downloaded and run on an FPGA. With successful programming and pin assignment, the onboard LEDs were illuminated in the desired pattern. This result can be interpreted as a successfully connected eight-bit computer. The code for all portions of the computer can be found in the Appendix, Figures 10-19.

II. Background

A. Assembly Language

Assembly is one of the most basic forms of computer programming which allows the user to breakdown processes to very basic steps. In assembly language, a programmer is able to directly manipulate how the processor stores information. This is done with the use of very simple digital logic in the forms individual commands interacting with the system's memory and registers.

B. Digital Logic

Digital logic is the backbone of all electric devices. 1's (high or on) and 0's (low or off) are used to signify the most basic form of logic. Using 1's and 0's in the form of logic represents patterns and signals which are used to communicate with other electronic devices. Circuits and logic gates are created using these simple strings of 1's and 0's which are the fundamentals of the electronic world.

C. Field-Programmable Gate Arrey (FPGA)

An FPGA is a kind of Programmable Logic Device or PLD. PLD's consist of a series of programmable switches which allow user interface to control the function of a device. These switches are the internal circuitry of such devices as PLG's and FPGA's. In general, FPGA's contain a large number of small logic circuit elements which can be connected and controlled with the use of the onboard logical switches. The setup of these devices makes them widely useful in a variety of tailored situations. The specific FPGA used in this experiment is the De0-Nano from Terasic. The layout can be seen in Figure 2.

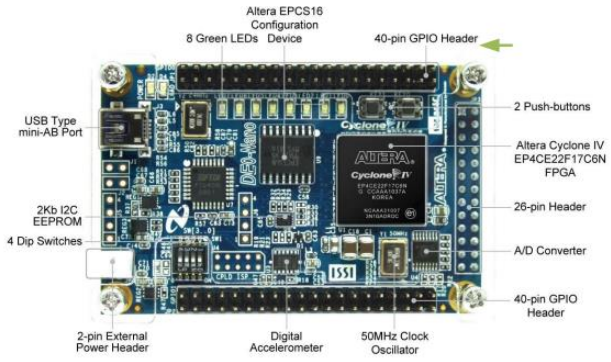

Figure 2: De0-Nano Layout

D. Arithmetic Logic Unit (ALU)

An ALU is a kind of logic circuit which performs various Boolean and arithmetic operations. This is useful because it allows for a user to create different functions that the ALU can perform such as addition and subtraction, logical and bitwise operations, such as AND and OR, and data shifts to name a few. As seen in Figure 3, the ALU takes in two different data sets and an instruction. It performs the given task and outputs the result, c, and z flags based upon the arithmetic operation.

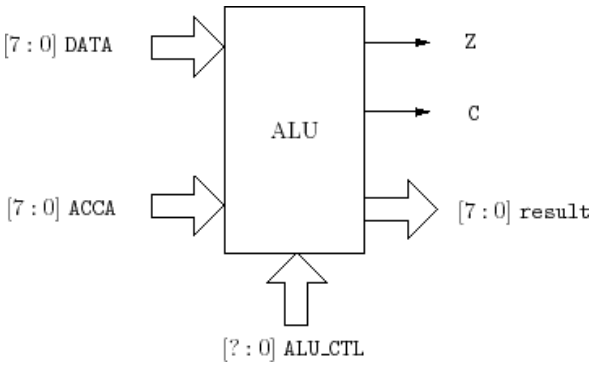

Figure 3: Arithmetic Logic Unit

E. Finite State Machine (FSM)

A FSM is a sequential circuit which is generally depicted in a state diagram. State diagrams visually show how the logic of the circuit works. If an input is a 1, then it may go to one state, but if it is a 0, then it may go to another or cycle through its present state. FSM's generally have fetch, reset, and executable states, as seen in Figure 4. The fetch state awaits instruction to determine which function is desired. After the instruction is chosen, the fetch state prepares for its execution state(s). For each op code the control unit must change signals within the machine to properly parse through memory and perform the task at hand. The final option within this system is a reset which sets all applicable one-bit inputs to 1 (as they are active-low) and changes the current memory address to 0xFF so that when the program begins, it begins at 0x80, the location of the start of the program.

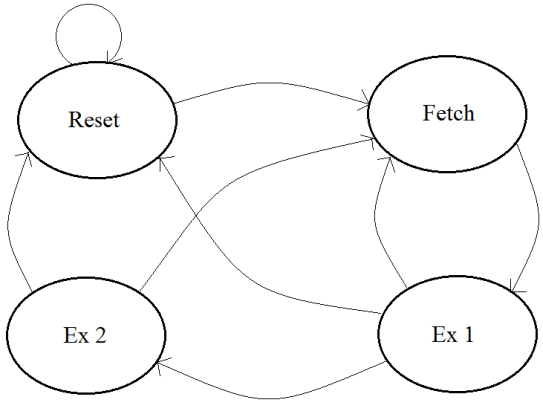

Figure 4: Finite State Machine

F. Latches and Registers

A latch is a memory element built with NOR or NAND gates which use set and reset signals to change the state of the circuit. A basic latch, shown in Figure 5, is a circuit in which

two NOR gates are connected. The output of a latch can be saved as memory. Gated latches are basic latches which are controlled by a clock upon either a rising or falling edge. A flip-flop is another name for a gated latch. A register is a flip-flop which stores one bit of information. Registers can be used in conjunction with other registers to allow for larger amounts of bitwise storage.

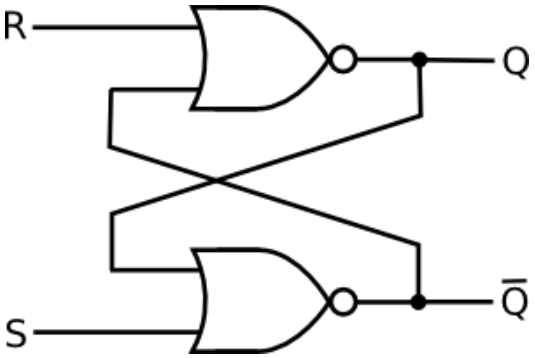

Figure 5: Basic SR Latch

G. Multiplexers

A multiplexer is implemented when there are a number of input possibilities and only one output value allowed. When a multiplexer is present, a computer is able to decide exactly which of the different options it wants to use to access data. It works by using input signals to generate an output signal based upon the state of one of the inputs. For example, Figure 6 shows a four-to-one multiplexer. When a certain two bit value is passed in through the select line, one of the four possible inputs is selected and passed through to the output.

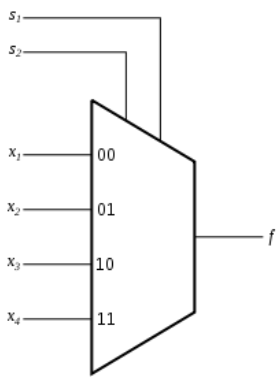

Figure 6: Four to One Multiplexer

H. Control Unit

A control unit takes in simple inputs and determines the desired functions which must be used to output a desired result. To make this happen, an FSM is used to navigate sequential logical circuits and pathways, using fetch, reset, ex1 and ex2. The control unit can be seen in Figure 1, on the left side of the image. It shows how the different signals interact with the rest of the computer.

III. RESULTS

An eight-bit computer was designed to use assembly language and op codes to implement logical programs. This was proven by the demonstration of a running light program in which an illuminated LED began on one side of the eight on-board LEDs and moved through to the other side. As the program continued, the previous LED was turned off and the next was turned on sequentially, moving from the first to last LED and back to the first once more, as seen in Figure 9 in the Appendix. The pattern continued until disrupted by the reset signal, assigned to one of the push buttons on the FPGA. By using op codes stored in memory, the computer worked with its components to send information throughout the computer to output the correct sequence. When the program began, the PC register pointed to the first memory address. Within that address was the instruction LDAA_IMM which immediately loads the next value. For this to happen, the computer must pass the relevant information through the control unit to allow it to select from one of its potential functions (which can be found in Table 1 in the Appendix).To illuminate an LED, a value, in this case $(01)_{16}$, was placed in the next accessed memory address. The following memory address contained the op code to store ACCA, which stores the previously loaded value into memory. The last of this basic sequence is a left shift of the data, which pushes all of the numbers to the left, filling a 0 where there is no data. This is done with the command LSLA. After the value is stored in ACCA, the next memory address contains the op code to output to the LEDs at value 8'h00 (as designated in the memory file). After the output is seen, memory continues to cycle through storing the 1, left shifting with a zero pad, and outputting the shifted value. Once the program reaches the eighth repetition, it must use the JMP operation to jump back to the beginning of the program to allow it to run continuously.

To prove that is output is logically correct, a waveform file was created. While reset is high, the program can run. This happens because of the active low nature of reset. In looking at output, the binary values correspond to the state of the lights in which one light is on and seven are off. In the individual output values, each high value corresponds to when that LED is illuminated. ACCA shows the values at each stage of the pattern, as the 1 in an eight-bit number shifts throughout the different places. Data and inst both show how the computer moves through the instructions and the different stages of the FSM (fetch, ex1, and ex2). Pc_out shows the different places in memory that the program accesses as it runs through the entire program and jumps back to the start.

IV. CONCLUSION

Over the course of one semester, an eight-bit computer was constructed and interconnected in the style of Verilog coding. With the use of assembly language and op codes, the computer successfully accessed memory, an arithmetic logic unit, a multiplexer, and a control unit to perform a given program. A running light program was designed for this project as a demonstration of success. This computer worked successfully

and displayed a running light program on the onboard LED lights on the FPGA.

V. Appendix

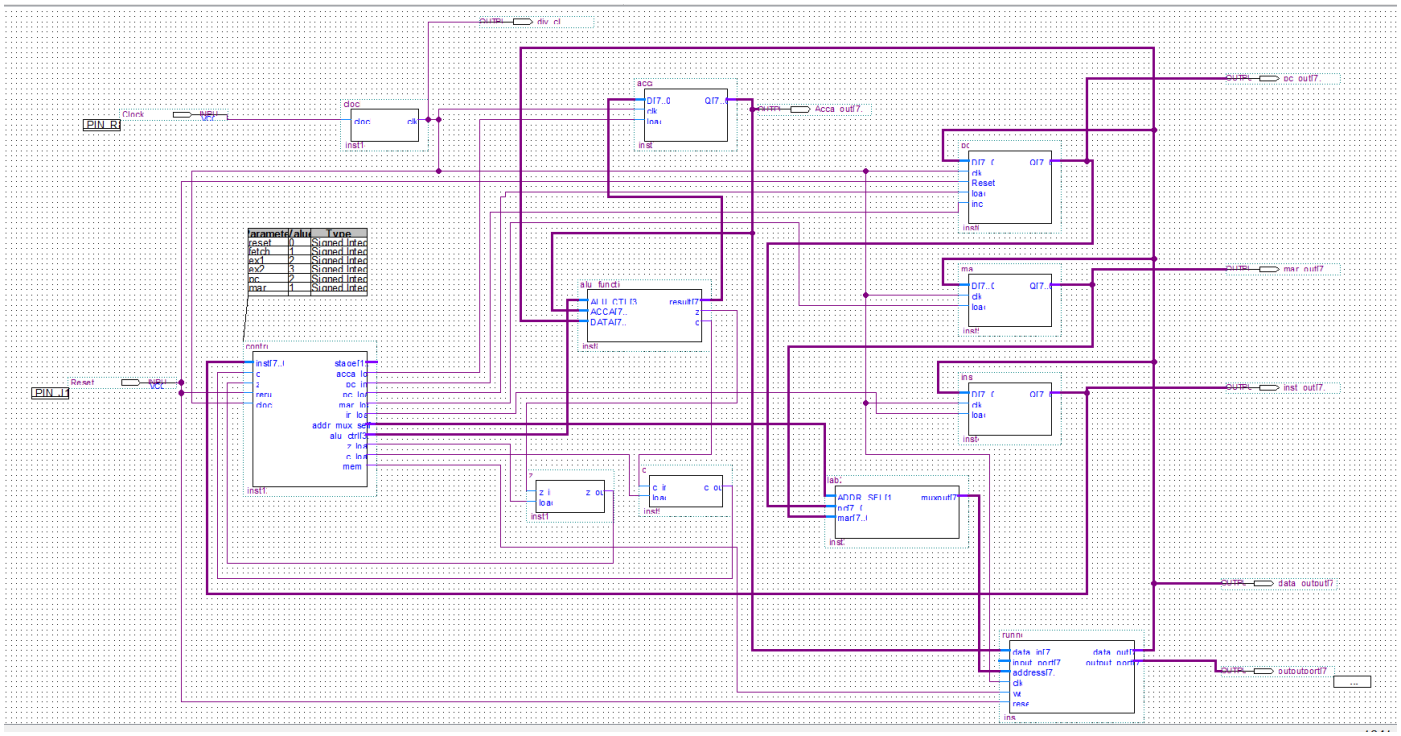

Figure 7: Block Diagram

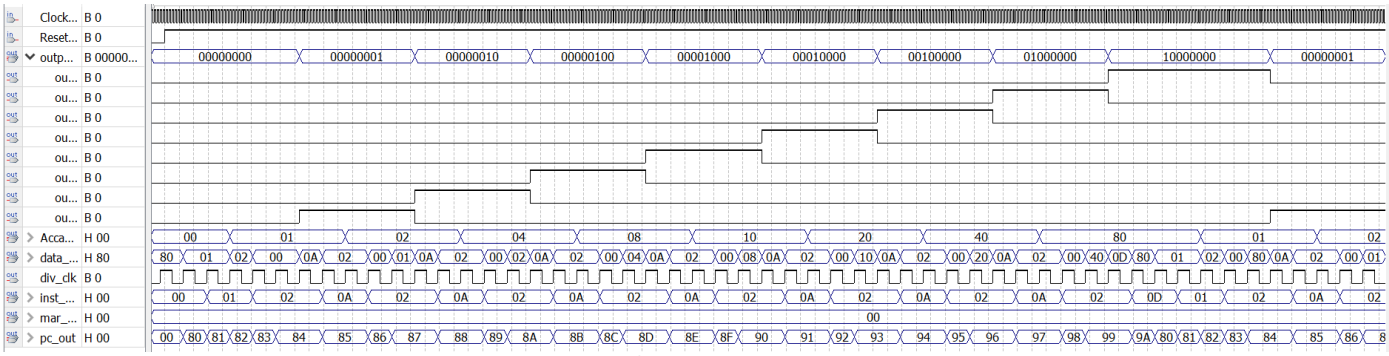

Figure 8: Waveform File For Running Lights

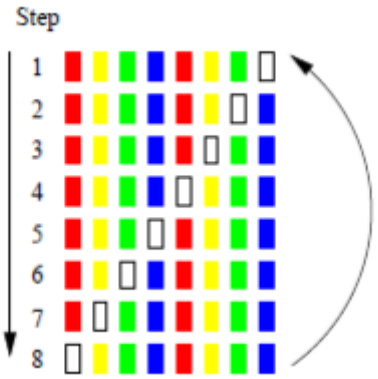

Figure 9: Visual Representation of Running Light Program

```
//One bit register to hold value for zero flag
//Courtney Johnson
//October 6, 2015

module z(z_in,load,z_out);

   input z_in,load;
   output reg z_out;

   always @ (*)
   begin
      if (~load)
      begin
         if (z_in == 0)          //if value is zero, flag is turned on
            z_out = 1;
         else
            z_out = 0;
      end
   end
endmodule
```

Figure 10: Z Register Code

```
//One bit register to hold value for carry flag
//Courtney Johnson
//October 6, 2015

module c(c_in,load,c_out);

   input c_in,load;
   output reg c_out;

   always @ (*)             //holds last value passed in
      begin
      if(~load)
         c_out <= c_in;
      end
endmodule
```

Figure 11: C Register Code

```
//Program works as a memory block for running lights
//Courtney Johnson
//November 9, 2015

`include "constants.v"

module runner (
     input [7:0] data_in,           //data in from other programs
     input [7:0] input_port,
     input [7:0] address,           //memory address from mux
     input clk,Clock_in,                    //adjusted clock signal
     input we,                      //memory signal from control
     input reset, Reset,                    //system reset
     output [7:0] data_out,         //data stored in current memory address
     output reg [7:0] output_port); //output to leds

reg [7:0] mem [0:127];

assign data_out = (address == 8'h00) ? output_port :
                  (address == 8'h01) ? input_port :
                  (address[7] == 1'b1) ? mem[address[6:0]]: 8'hxx;

always @ (posedge clk or negedge reset)
   if (~reset)
      begin
         //mem_init file

            //mem[7'h00] = 8'h00;   // Address 0x80
            mem[7'h00] = `INST_LDAA_IMM;   // Address 0x81-- IMM immedietly loads next value (1)
            mem[7'h01] = 8'h01;   // Address 0x82-- Sets value for acca
            mem[7'h02] = `INST_STAA;   // Address 0x83-- Stores acca in memory
            mem[7'h03] = 8'h00;   // Address 0x84-- LED output--see statement below addresses- 00000001
            mem[7'h04] = `INST_LSLA;   // Address 0x85
            mem[7'h05] = `INST_STAA;   // Address 0x86
            mem[7'h06] = 8'h00;   // Address 0x87  - 00000010
            mem[7'h07] = `INST_LSLA;   // Address 0x88
            mem[7'h08] = `INST_STAA;   // Address 0x89
            mem[7'h09] = 8'h00;   // Address 0x8a - 00000100
            mem[7'h0a] = `INST_LSLA;   // Address 0x8b
            mem[7'h0b] = `INST_STAA;   // Address 0x8c
            mem[7'h0c] = 8'h00;   // Address 0x8d - 00001000
            mem[7'h0d] = `INST_LSLA;   // Address 0x8e
            mem[7'h0e] = `INST_STAA;   // Address 0x8f

            mem[7'h0f] = 8'h00;   // Address 0x90 - 00010000
            mem[7'h10] = `INST_LSLA;   // Address 0x91
          mem[7'h11] = `INST_STAA;   // Address 0x92
          mem[7'h12] = 8'h00;   // Address 0x93 - 00100000
          mem[7'h13] = `INST_LSLA;   // Address 0x94
          mem[7'h14] = `INST_STAA;   // Address 0x95
          mem[7'h15] = 8'h00;   // Address 0x96 - 01000000
          mem[7'h16] = `INST_LSLA;   // Address 0x97
          mem[7'h17] = `INST_STAA;   // Address 0x98
          mem[7'h18] = 8'h00;   // Address 0x99 - 10000000
          mem[7'h19] = `INST_JMP;   // Address 0x9b - Jumps back to the start
          mem[7'h1a] = 8'h80;   // Address 0x99 - 10000000

          mem[7'h7f] = 8'h80;   // Address 0xff   RESET VECTOR
      end
   else begin
      if ((address == 8'h00) && (~we))
         output_port <= data_in;
      if ((address[7]) && (~we))
         mem[address[6:0]] <= data_in;
   end

endmodule
```

Figure 12: Runner Code

```
//Function stores 8 bit data and loads data if load if low on
//rising edge of clock
//Courtney Johnson
//October 6, 2015


module acca(D,clk,Q,load);

    input [7:0]D;              //8 bit register
    input clk,load;
    output reg [7:0]Q;

    always @ (posedge clk)
    begin
       if(load == 0)           //keeps data the same if load = 0
             Q <= D;
    end
endmodule
```

Figure13: ACCA Register Code

```
//Function stores 8 bit data and loads data if load if low on
//rising edge of clock
//Courtney Johnson
//October 6, 2015

module inst(D,clk,Q,load);

   input [7:0]D;
   input clk,load;
   output reg [7:0]Q;

   always @ (posedge clk)
   begin
      if(load == 0)                  //keeps data the same if load is 0
            Q <= D;
   end
endmodule
```

Figure14: INST Register Code

```
//Function stores 8 bit data and loads data if load if low on
//rising edge of clock
//Courtney Johnson
//October 6, 2015

module mar(D,clk,Q,load);

   input [7:0]D;
   input clk,load;
   output reg [7:0]Q;

   always @ (posedge clk)
   begin
      if(load == 0)                //stays the same if load is 0
            Q <= D;
   end
endmodule
```

Figure 15: MAR Register Code

```
//Program Counter register
//Courtney Johnson
//October 6, 2015

module pc(D,clk,Resetn,Q,load,incr);

   input [7:0]D;
   input clk,Resetn,load,incr;
   output reg [7:0]Q;

   always @ (negedge Resetn, posedge clk)    //negedge is falling edge, posedge is rising edge of clock
   begin
      if(Resetn == 0)
         Q <= 0;
      else
      begin
         if(load == 0)                      //New data value is loaded
            Q <= D;
         else if (incr == 0)                //Q value is incremented
            Q <= Q + 1;
         else
            Q <= Q;
      end
   end
endmodule
```

Figure16: PC Register Code

```
//This uses a four to one multiplexer to change the display on two seven segment displays at one time.
//It also allows for the use of a 8 switch dip switch to change the display individually.
//Author: Courtney Johnson
//Date: 9/15/2015

module lab2(muxout, ADDR_SEL, pc, mar);
   input [7:0]pc, mar;
   input [1:0]ADDR_SEL;          //Allocates 2 bits for ADDR_SEL
   output reg [7:0]muxout;

   always @ (ADDR_SEL)
         case (ADDR_SEL)                        //Uses the value for ADDR_SEL to select an input
            2'b11 : muxout = 8'b11111111;               //Input for dip switch
            2'b01 : muxout = mar;               //Hardcoded value for MAR 0x10
            2'b10 : muxout = pc;                //Hardcoded value for PC 0x0A
            2'b00 : muxout = 8'b11111111;       //Hardcoded valur for RST_ADDR 0xFF
         endcase
endmodule
```

Figure17: 4 to 1 Multiplexer Code

```
//This function is the logic behind an Arithmetic Logic Unit (ALU) to perform
//a variety of logic functions
//Courtney Johnson
//Septemper 29, 2015

`include "constants.v"                    //allows for easy declaration of logic functions

module alu_function(ALU_CTL,ACCA,DATA,result,z,c);
   input [3:0]ALU_CTL;                    //selects which function to perform
   input [7:0]ACCA,DATA;                  //input numbers for logic functions
   output reg [7:0]result;                //the result from the specific operand
   output reg z,c;                        //z (zero) flag and c (carry) flag

      //parameter ACCA = 8'b10110011;
      //parameter DATA = 8'b01111110;


   always @ (*)
      begin
         case(ALU_CTL)                                  //case statement switches input for ALU_
            `ALU_LOAD: result = DATA;                   //loads DATA into result
            `ALU_ADDA: {c,result} = ACCA + DATA;        //adds ACCA and DATA
            `ALU_SUBA: {c,result} = ACCA - DATA;        //subtracts DATA from ACCA
            `ALU_ANDA: result = ACCA && DATA;           //logically ands DATA and ACCA
            `ALU_ORAA: result = ACCA || DATA;           //logically ors DATA and ACCA
            `ALU_COMA: begin                            //inverts ACCA
                     result = ~ACCA;
                     c = 1;                             //denotes a carry flag
                     end
            `ALU_INCA: result = ACCA + 1;               //increments ACCA by adding 1
            `ALU_LSRA: begin                            //logically shifts ACCA to the right
                     c = ACCA[0];                       //sets carry to lsb
                     result = ACCA >> 1;
                     end
            `ALU_LSLA: begin                            //logically shifts ACCA to the left
                     c = ACCA[7];                       //sets carry to msb
                     result = ACCA << 1;
                     end
            `ALU_ASRA: begin                            //arithmetic shift to right
                     c = ACCA[0];                       //sets carry to lsb
                     result = ACCA >> 1;
                     result[7] = c;                     //wraps lsb around to msb
                     end
            default: begin                              //covers any other case
                     c = 0;
                     result = 0;
                   end
         endcase
         if (result == 0)                          //if result is 0, the z flag is 1
            begin
               z = 1;
            end
         else
            begin
               z = 0;
            end
      end

endmodule
```

Figure18: Arithmetic Logic Unit Code

```
// This functions as a control unit for a computer
//Courtney Johnson
//October 30, 2015


`include "constants.v"

module control(stage,acca_load,pc_inc,pc_load,mar_load,ir_load,inst,addr_mux_sel,alu_ctrl,z_load,c_load,c,z,mem_w,rerun,clock);

//inst is the instruction register
//c and z are flags built into the programs which will be used in the computer
//rerun is the cpu reset
//clock is the input clock
//stage and addr_mux_sel keep track of where the program is and which selections have been made
//alu_ctrl is the input for the alu to run the specific program
//all of the inc and load signals hold a one bit value to turn them on or off
//mem_w chooses whether to write to memory

input [7:0]inst;
input c,z,rerun,clock;
output reg [1:0]stage,addr_mux_sel;
output reg [3:0]alu_ctrl;
output reg acca_load,pc_inc,pc_load,mar_load,ir_load,mem_w,c_load,z_load;

//parameters set values for easy access

parameter reset = 0;
parameter fetch = 1;
parameter ex1 = 2;
parameter ex2 = 3;
parameter pc = 2;
parameter mar = 1;

//This always/case statement contains info for the different instruction

always @ (posedge clock, negedge reset)
   begin
      if (rerun == 0)
         stage <= reset;
      else
      begin
         case (stage)
               reset:
                  stage <= fetch;
               fetch:
                  begin
                  stage <= ex1;
                  end
               ex1:
                  begin
                  case (inst)
                  `INST_LDAA: stage <= ex2;
                  `INST_LDAA_IMM: stage <= fetch;
                  `INST_STAA: stage <= ex2;
                  `INST_ADDA: stage <= ex2;
                  `INST_SUBA: stage <= ex2;
                  `INST_ANDA: stage <= ex2;
                  `INST_ORAA: stage <= ex2;
                  `INST_CMPA: stage <= ex2;
                  `INST_COMA: stage <= fetch;
                  `INST_INCA: stage <= fetch;
                  `INST_LSLA: stage <= fetch;
                  `INST_LSRA: stage <= fetch;
                  `INST_ASRA: stage <= fetch;
                  `INST_JMP: stage <= fetch;
                  `INST_JCS: stage <= fetch;
                  `INST_JCC: stage <= fetch;
                  `INST_JEQ: stage <= fetch;
                  default: stage <= reset;
                  endcase
                  end
               ex2:
                  stage = fetch;
         endcase
      end
   end

//This always/case statement contains info for ex1 and ex2

always @ (stage)
begin
   case(stage)
      reset:
         begin
            acca_load = 1;
            pc_inc = 1;
            pc_load = 0;
            mar_load =1;
            ir_load = 1;
            mem_w = 1;
            c_load = 1;
            z_load = 1;
            z_load = 1;
            alu_ctrl = 1;
            addr_mux_sel = reset;
         end
      fetch:
         begin
            addr_mux_sel = pc;
            pc_inc = 0;
            ir_load = 0;
            acca_load = 1;
            pc_load = 1;
            mar_load =1;
            mem_w = 1;
            c_load = 1;
            z_load = 1;
            alu_ctrl = 0;
         end
      ex1:
         begin
            case (inst)
            `INST_LDAA:
               begin
                  mar_load = 0;
                  pc_inc = 0;
                  addr_mux_sel = pc;
                  acca_load = 1;
                  pc_load = 1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 1;
                  alu_ctrl = 0;
               end
```

```
`INST_LDAA_IMM:
   begin
      acca_load = 0;
      pc_inc = 0;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 0;
      alu_ctrl = `ALU_LOAD;
      addr_mux_sel = pc;
   end
`INST_STAA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load = 0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 0;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_ADDA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load =0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_SUBA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load =0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_ANDA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load =0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_ORAA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load =0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_CMPA:
   begin
      acca_load = 1;
      pc_inc = 0;
      pc_load = 1;
      mar_load =0;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_COMA:
   begin
      acca_load = 0;
      pc_inc = 1;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 0;
      z_load = 0;
      alu_ctrl = `ALU_COMA;
      addr_mux_sel = pc;
   end
`INST_INCA:
   begin
      acca_load = 0;
      pc_inc = 1;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 0;
      alu_ctrl = `ALU_INCA;
      addr_mux_sel = pc;
   end
`INST_LSLA:
   begin
      acca_load = 0;
      pc_inc = 1;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 0;
      z_load = 0;
      alu_ctrl = `ALU_LSLA;
      addr_mux_sel = pc;
   end
```

```
`INST_LSRA:
   begin
      acca_load = 0;
      pc_inc = 1;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 0;
      z_load = 0;
      alu_ctrl = `ALU_LSRA;
      addr_mux_sel = pc;
   end
`INST_ASRA:
   begin
      acca_load = 0;
      pc_inc = 1;
      pc_load = 1;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 0;
      z_load = 0;
      alu_ctrl = `ALU_ASRA;
      addr_mux_sel = pc;
   end
`INST_JMP:
   begin
      acca_load = 1;
      pc_inc = 1;
      pc_load = 0;
      mar_load =1;
      ir_load = 1;
      mem_w = 1;
      c_load = 1;
      z_load = 1;
      alu_ctrl = 0;
      addr_mux_sel = pc;
   end
`INST_JCS:
   begin
      if ( c == 1)
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 0;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
         z_load = 1;
         alu_ctrl = 0;
         addr_mux_sel = pc;
      end
      else
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 1;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
         z_load = 1;
         alu_ctrl = 0;
         addr_mux_sel = pc;
      end
   end
`INST_JCC:
   begin
      if ( c == 0)
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 0;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
         z_load = 1;
         alu_ctrl = 0;
         addr_mux_sel = pc;
      end
      else
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 1;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
          z_load = 1;
          alu_ctrl = 0;
          addr_mux_sel = pc;
       end
    end
`INST_JEQ:
   begin
      if ( z == 1)
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 0;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
         z_load = 1;
         alu_ctrl = 0;
         addr_mux_sel = pc;
      end
      else
      begin
         acca_load = 1;
         pc_inc = 1;
         pc_load = 1;
         mar_load =1;
         ir_load = 1;
         mem_w = 1;
         c_load = 1;
         z_load = 1;
         alu_ctrl = 0;
         addr_mux_sel = pc;
      end
   end
```

```
            default:
               begin
                  acca_load = 1;
                  pc_inc = 1;
                  pc_load = 0;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 1;
                  alu_ctrl = 1;
                  addr_mux_sel = reset;
               end
            endcase
         end
      ex2:
         begin
            case (inst)
            `INST_LDAA:
               begin
                  acca_load = 0;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 0;
                  alu_ctrl = `ALU_LOAD;
                  addr_mux_sel = mar;
               end
            `INST_STAA:
               begin
                  acca_load = 1;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 0;
                  c_load = 1;
                  z_load = 1;
                  alu_ctrl = 0;
                  addr_mux_sel = mar;
               end
            `INST_ADDA:
               begin
                  acca_load = 0;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 0;
                  z_load = 0;
                  alu_ctrl = `ALU_ADDA;
                  addr_mux_sel = mar;
               end
            `INST_SUBA:
               begin
                  acca_load = 0;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 0;
                  z_load = 0;
                  alu_ctrl = `ALU_SUBA;
                  addr_mux_sel = mar;
               end
            `INST_ANDA:
               begin
                  acca_load = 0;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 0;
                  alu_ctrl = `ALU_ANDA;
                  addr_mux_sel = mar;
               end
            `INST_ORAA:
               begin
                  acca_load = 0;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 0;
                  alu_ctrl = `ALU_ORAA;
                  addr_mux_sel = mar;
               end
            `INST_CMPA:
               begin
                  acca_load = 1;
                  pc_inc = 1;
                  pc_load = 1;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 0;
                  z_load = 0;
                  alu_ctrl = `ALU_SUBA;
                  addr_mux_sel = mar;
               end
            default:
               begin
                  acca_load = 1;
                  pc_inc = 1;
                  pc_load = 0;
                  mar_load =1;
                  ir_load = 1;
                  mem_w = 1;
                  c_load = 1;
                  z_load = 1;
                  alu_ctrl = 1;
                  addr_mux_sel = reset;
               end
            endcase
         end
endcase
end
endmodule
```

Figure19: Control Unit Code

Table 1: ALU Control Operations

	Instruction	Operation (Mnemonic)
0	`nop`	Do nothing. (No Operation)
1	`LDDA addr`	Loads `ACCA` with the value in memory at address `addr`. `C` stays the same, `Z` changes. (Load `ACCA` from memory)
2	`LDDA_IMM #num`	Loads `ACCA` with `num`, the value in memory at the address immediately following the `LDAA #num` command. `C` stays the same, `Z` changes. (Load `ACCA` with an immediate)
3	`STAA addr`	Stores the value in `ACCA` at memory address `addr`. `C` stays the same, Z changes. (Store `ACCA` in memory)
4	`ADDA addr`	Adds the value in memory location `addr` to the value in `ACCA` and saves the result in `ACCA`. `C` and `Z` change. (Add `ACCA` and value in memory)
5	`SUBA addr`	Subtracts the value in memory location `addr` from the value in `ACCA` and saves the result in `ACCA`. `C` and `Z` change. (Subtract value in memory from `ACCA`)
6	`ANDA addr`	Perform a logical `AND` of the value in memory location `addr` with the value in `ACCA`. Save the result in `ACCA`. `C` stays the same, `Z` changes. (Logical `AND` of `ACCA` and value in memory)
7	`ORAA addr`	Perform a logical `OR` of the value in memory location `addr` with the value in `ACCA`. Save the result in `ACCA`. `C` stays the same, `Z` changes. (Logical `OR` of `ACCA` and value in memory)
8	`CMPA addr`	Compare `ACCA` to value in `addr`. This is done by subtracting the value in `addr` from `ACCA`. `ACCA` does not change. `C` and `Z` change. (Compares `ACCA` to the value in `addr`)
9	`COMA`	Replace the value in `ACCA` with its one's complement. `C` is set to 1 and `Z` changes. (Compliment `ACCA`)
A	`INCA`	Increment value in `ACCA`. `C` stays the same and `Z` changes. (`INCA ACCA`)
B	`LSLA`	Logical shift left of `ACCA`. `C` and `Z` change. (Logical shift left `ACCA`)
C	`LSRA`	Logical shift right of `ACCA`. `C` and `Z` change. (Logical shift right `ACCA`)
D	`ASRA`	Arithmetic shift right of `ACCA`. `C` and `Z` change. (Arithmetic shift right `ACCA`)
E	`JMP addr`	Jumps to the instruction stored in address `addr`. The `PC` is replaced with `addr`. `C` and `Z` stay the same. (Jump)
F	`JCS addr`	Jumps to the instruction stored in address `addr` if $C = 1$. If `C` is not set, continue with next instruction. `C` and `Z` stay the same. (Jump if carry set)
10	`JCC addr`	Jumps to the instruction stored in address `addr` if $C = 0$. If `C` is set, continue with next instruction. `C` and `Z` stay the same. (Jump if carry not set)
11	`JEQ addr`	Jumps to the instruction stored in address `addr` if $Z = 1$. If `Z` is not set, continue with next instruction. `C` and `Z` stay the same. (Jump if `Z` set)

VI. REFERENCES

Content:

Brown, Stephen, and Zvonko Vranesic. *Fundamentals of Digital Logic with Verilog Design*. 3rd ed. New York: McGraw Hill, 2014. Print.

Erives, Hector. *EE 231 Digital Electronics Lab*. N.p., Aug. 2015. Web. 3 Dec. 2015. <http://www.ee.nmt.edu/~erives/231L_15/EE231L.html>.

Images:

Erives, Hector. *EE 231 Digital Electronics Lab*. N.p., Aug. 2015. Web. 3 Dec. 2015. <http://www.ee.nmt.edu/~erives/231L_15/EE231L.html>.

https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/4-to-1_multiplexer.svg/350px-4-to-1_multiplexer.svg.png

http://www.ee.nmt.edu/~elosery/fall_2009/ee231L/lab5/img1.png

https://startingelectronics.org/software/VHDL-CPLD-course/tut9-SR-latch/