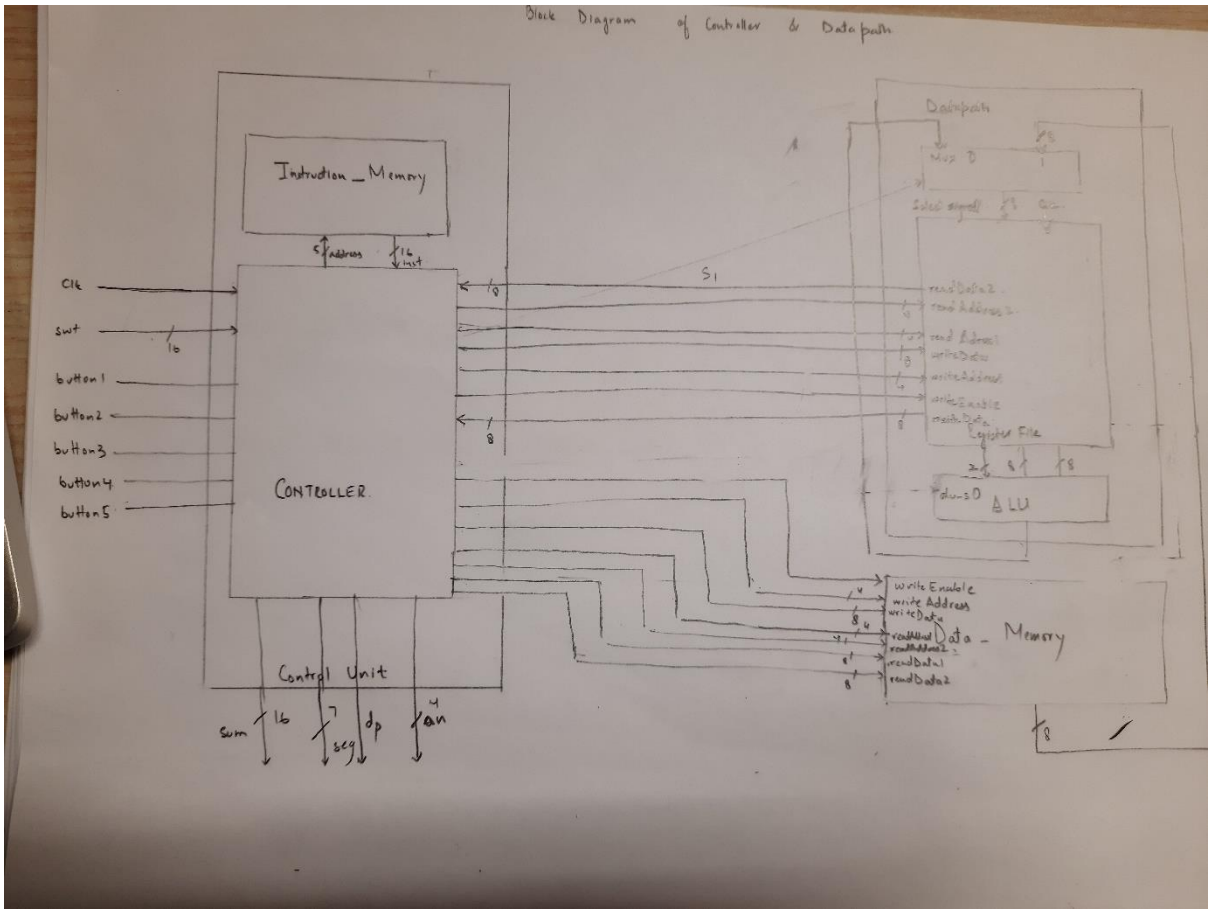
Ammaar Iftikhar

21901257

Section 3

Project Report

Block Diagram of the six instruction processor



Part 1.) The main components of the processor are:

1.) ALU: The Arithmetic Logic Unit performs the required action on the inputs which are selected by the select signals sent to the processor. The operations are to either select one input or to add two inputs and then output them.

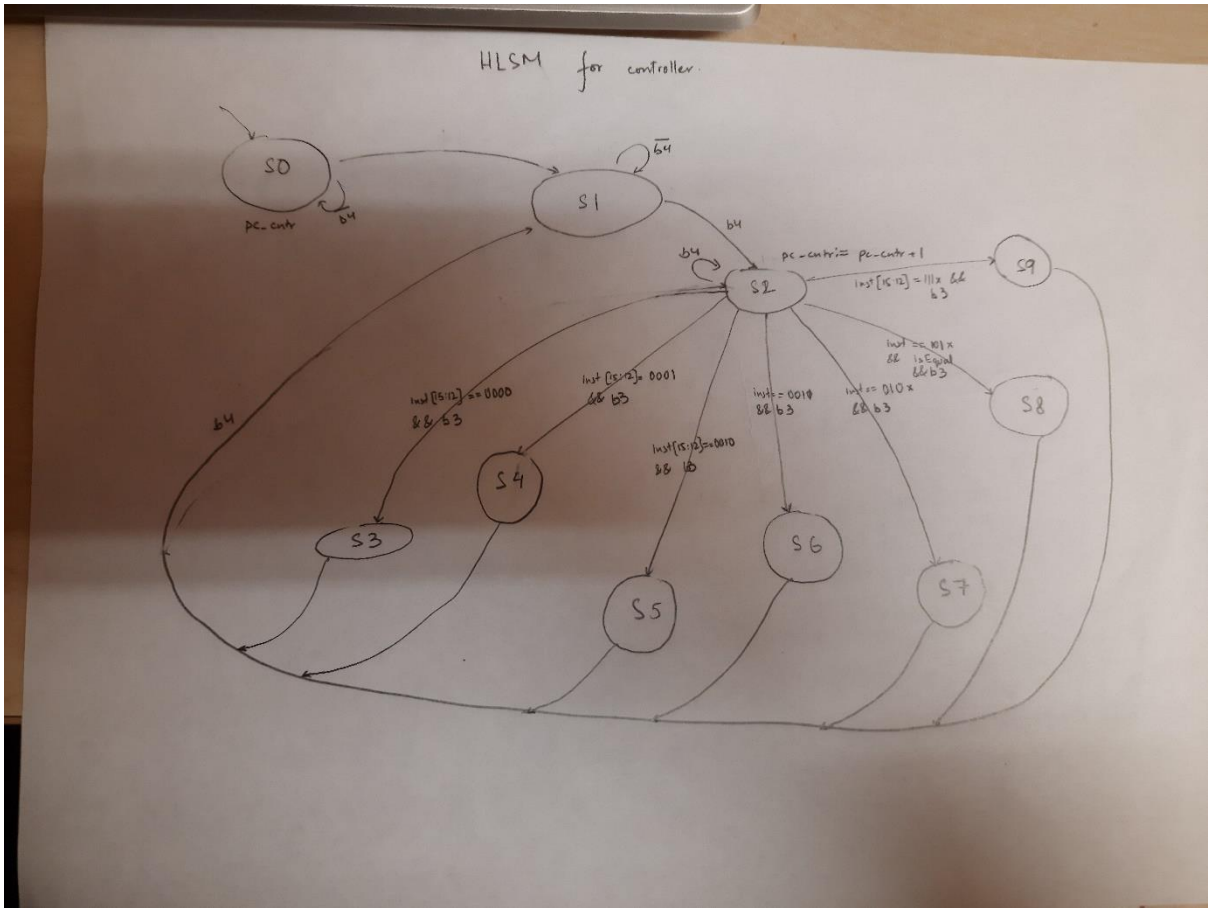
2.) MUX: The multiplexer chooses one of the two inputs to be written into the register file.

3.) Controller: The controller is responsible for implementing the HLSM design for the entire processor. It uses a finite state machine to select the operations that are to be

conducted. The controller is the center of operations in our design. The controller is connected to the Datapath, Instruction memory, and the Data memory. It is in the controller we choose and assign the input and address to be written onto the data memory, as well as the address from where we want to read data from the data memory. We also assign the value to the write enable for the data memory its desired value. In the controller we select the address where we want to write onto the register file, as well as assign the write enable signal of the register file the desired value. We also commute through the instruction memory in this module, as well as select the operation to be executed based on the input from the user on the basis board.

4.) Datapath: The Datapath is responsible for allocating the values to different elements of our processor like register file, mux, and ALU. It also sends the value that has to be allocated to the data memory back to the controller module. The Datapath is mainly for the distribution of instructions to certain parts of the processor.

Part 2.) **Controller and explanation**



In the first state (S0) of the controller HLSM, we initialize the value of pc_cntr to zero and then move to the 2nd stage (S1) irrespective of the inputs.

In the second stage (S1) of the controller HLSM, we stay in this state unless the input b4 is not 1. As soon as b4 is 1, we transition to the next state which is S2.

In the third stage (S2) of the controller HLSM, we decode the opcode provided by the instruction memory. We also increment the value pc_cntr in this state. Based on the opcode we branch into another state if the b3 is equal to 1. Otherwise, the state will come back to state S2 if b4 is 1, and, thus, again increment the value of the pc_cntr.

The seven states where S2 can branch to are:

S3, S4, S5, S6, S7, S8, and S9

S3: In this state, the write-enable of the data memory is initialized to 1 and the write address of the data memory is initialized to inst[7:4] and the value to be written onto the data memory is fetched from the register file, the address from where the data is read is inst[3:0]. After this step, if b4 is one, the state machine will go back to the S1 state.

S4: In this state, the write-enable of the data memory is initialized to 1, the write address of the data memory is initialized to [11:8], and the write data of the data memory is initialized to [7:0]. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S5: In this state, the write-enable of the register file is initialized to 1, the write address of the register file is initialized to inst[7:4], and the data to be written is read from the address = inst[3:0] of the read data address. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S5: In this state, the write-enable of the register file is initialized to 1, the write address of the resister file is registered to inst[11:8], and the write data of the register file is initialized to inst[7:0]. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S6: In this state, the write-enable of the register file is initialized to 1, the write address of the register file is initialized to inst[11:8], and the select (alu_s1) is initialized to 1. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S7: In this state, the write-enable of the register file is initialized to 1, the write address of the register file is initialized to inst[11:8], and then the two values are read from the register with the values inst[7:4] and inst[3:0], which are finally added and then written onto the register file. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S8: In this state, the write-enable of the register file is zero, two values from the register file are read and checked if they are equal. In case the two read values are equal, pc_cntr is initialized to inst[12:8]. After this step, if b4 is equal to one, the state machine will go back to the S1 state.

S9: In this state, in this state the register ceases to work, and then goes back to S0.

Part c.)

Controller

```
module controller2( input logic clk,
                    input logic [15:0] swt,
                    input logic  button1, button2, button3, button4, button5,
                    output logic [15:0] led,
                    output logic [6:0] seg,
                    output logic  dp,
                    output logic [3:0] an);

    typedef enum logic [3:0]{S0, S1, S2, S3, S4, S5, S6, S7, S8, S9} state;
    state next, current;

    logic [15:0] inst;
    logic [4:0] index;
```

```
    logic [4:0] pc_cntr;
    logic [7:0] writeValueDM, readValue2, readValueDM, readDataDM2, readDataTemp, readRF1;
    logic weR, weD;
    logic b1, b2, b3, b4, b5, isEqual;
    localparam ele = 32;
    logic [1:0] alu, rf;
    logic [3:0] writeAddressDM, readAddressDM, writeAddressRF, readAddressRF1, readAddressRF2;


    // works
    instruction_memory in( pc_cntr, inst);
    datapath_final fin( clk, writeAddressRF, weR, alu[1:1], alu[0:0], rf[1:1], rf[0:0], readDataTemp, readValueDM, isEqual);
    register_file data( clk, weD, writeAddressDM, writeValueDM, readAddressDM, index, readDataDM2, readValue2);
    //datapath2 da1(clk, inst[15:0], weR, index, readDataTemp, readValueDM);


    // deboucning the buttons
    debounce db1( clk, button1, b1);
    debounce db2( clk, button2, b2);
    debounce db3( clk, button3, b3);
    debounce db4( clk, button4, b4);
    debounce db5( clk, button5, b5);


    // display on the seven segment
    SevSeg_4digit( clk, index, 4'b0, readValue2[7:4], readValue2[3:0], seg, dp, an);


    // works
    always_ff @(posedge clk)
        begin
            if( b5)  current <= S0;
```

```
        else     current <= next;
    end


always @( posedge clk)
    case(current)
        S0: begin
                next = S1;
            end
        S1: begin
                weR = 0;
                weD = 0;


                if( b4)
                    next = S2;


            end
        S2: begin
                case({inst[15:12]})
                    4'b0000: next = S3;
                    4'b0001: next = S4;
                    4'b0010: next = S5;
                    4'b0011: next = S6;
                    4'b010x: next = S7;
                    4'b10xx: next = S8;
                    4'b111x: next = S9;
                    default: next = S1;
                endcase
            end
        S3: begin
                weD = 1;
```

```
        rf[0:0] = 0;
        next = S1;
    end
S4: begin
        weD = 1;
        rf[0:0] = 0;
        next = S1;
    end
S5: begin
        weR = 1;
        alu[1:1] = 0;
        alu[0:0] = 0;
        rf[0:0]  = 0;
        writeAddressRF = inst[7:4];
        readDataTemp = readDataDM2;
        next = S1;
    end
S6: begin
        weR = 1;
        alu[1:1] = 0;
        alu[0:0] = 0;
        rf[0:0] = 0;
        writeAddressRF = inst[11:8];
        readDataTemp = inst[7:0];
        next = S1;
    end
S7: begin
        weR = 1;
        alu[1:1] = 0;
        alu[0:0] = 1;
```

```
            rf[0:0]  = 1;
            writeAddressRF = inst[11:8];
            next = S1;
        end
      S8: begin
            if( isEqual) begin
                next = S2;
            end
            else
                next = S1;
        end
      S9: begin
            weD = 0;
            weR = 0;
        end
      default: next = S0;
    endcase

always @( posedge clk)
  begin

    if( b5)
      index = 0;
    else if( b1 && index == 15)
      index = 0;
    else if( b1)
      index = index + 1;
    else if( b2 && index == 0)
      index = 15;
    else if ( b2)
```

```
            index = index - 1;
        end

    always @( posedge clk)
      begin
        readAddressDM = inst[3:0];
        led = inst;
      end

    always @( posedge clk)
      begin
        if( b5)
            pc_cntr = 0;
        else if( b3 && pc_cntr == (ele - 1))
            begin
                pc_cntr = 0;
            end
        else if( b3)
            begin
                pc_cntr = pc_cntr + 1;
            end
        else if( isEqual && current == S8)  pc_cntr = readValueDM;
      end
    always @( current)
        if( {inst[12:12]})   begin
                writeAddressDM = inst[11:8];
                writeValueDM = inst[7:0];
            end
        else  begin
              writeAddressDM = inst[7:4];
```

```
            writeValueDM   = readValueDM;
        end

endmodule
```

Datapath

```
module datapath_final(
    input logic clk,
    input logic [3:0] Rf_writeAddress,
    input logic Rf_we,
    input logic [3:0] Rf_readAddress1,
    input logic [3:0] Rf_readAddress2,
    input logic alu_s1,
    input logic alu_s0,
    input logic Rf_s1,
    input logic Rf_s0,
    input logic [7:0] R_data,
    output logic [7:0] W_data,
    output logic isEqual);

    logic [7:0] writeData, readData1, readData2, temp1;

    register_file regist( clk, Rf_we, Rf_writeAddress, writeData, Rf_readAddress1,
Rf_readAddress2, readData1, readData2);
    alu aluN( alu_s1, alu_s0, readData1, readData2, temp1);
    muxN mux( Rf_s0, temp1, R_Data);
    // to be constructed
    //alu a1( alu_s1, alu_s0, )

    assign W_data = readData1;
    assign isEqual = (readData1 == readData2);
```

```
endmodule
```

Debouncer

```
module debounce(input logic clk, input logic button, output logic pulse );

logic [24:0] timer;
typedef enum logic [1:0]{S0,S1,S2,S3} states;
states state, nextState;
logic gotInput;

always_ff@(posedge clk)
   begin
      state <= nextState;
      if(gotInput)
         timer <= 25000000;
      else
         timer <= timer - 1;
   end
always_comb
   case(state)
      S0: if(button)
         begin //startTimer
            nextState = S1;
            gotInput = 1;
         end
         else begin nextState = S0; gotInput = 0; end
      S1: begin nextState = S2; gotInput = 0; end
      S2: begin nextState = S3; gotInput = 0; end
      S3: begin if(timer == 0) nextState = S0; else nextState = S3; gotInput = 0; end
```

```
        default: begin nextState = S0; gotInput = 0; end
        endcase

assign pulse = ( state == S1 );
endmodule
```

Register File and the Data Memory use the same module but different instances

```
module register_file( input logic clk,
            input logic [3:0] writeAddress,
            input logic [7:0] writeData,
            input logic writeEnable,
            input logic [3:0] readAddress1,
            input logic [3:0] readAddress2,
            output logic [7:0] readData1,
            output logic [7:0] readData2);

    logic [7:0] mem[15:0];
    localparam up = 16;
    always @(posedge clk)
        if(writeEnable)  mem[writeAddress] <= writeData;

    initial
        begin
            for( int i = 0; i < up; i++)
                mem[i] = 8'b00000000;
        end

    assign readData1 = mem[readAddress1];
    assign readData2 = mem[readAddress2];
endmodule
```

Seven segment display

```
module SevSeg_4digit(
 input clk,
 input [3:0] in3, in2, in1, in0, //user inputs for each digit (hexadecimal value)
 output [6:0]seg, logic dp, // just connect them to FPGA pins (individual LEDs).
 output [3:0] an   // just connect them to FPGA pins (enable vector for 4 digits active low)
 );

// divide system clock (100Mhz for Basys3) by 2^N using a counter, which allows us to multiplex at lower speed
localparam N = 18;
logic [N-1:0] count = {N{1'b0}}; //initial value
always@ (posedge clk)
        count <= count + 1;


logic [4:0]digit_val; // 7-bit register to hold the current data on output
logic [3:0]digit_en;  //register for the 4 bit enable

always@ (*)
 begin
 digit_en = 4'b1111; //default
 digit_val = in0; //default

  case(count[N-1:N-2]) //using only the 2 MSB's of the counter

   2'b00 :  //select first 7Seg.
    begin
```

```
     digit_val = {1'b0, in0};
     digit_en = 4'b1110;
    end

   2'b01:  //select second 7Seg.
    begin
     digit_val = {1'b0, in1};
     digit_en = 4'b1101;
    end

   2'b10:  //select third 7Seg.
    begin
     digit_val = {1'b1, in2};
     digit_en = 4'b1011;
    end

   2'b11:  //select forth 7Seg.
    begin
     digit_val = {1'b0, in3};
     digit_en = 4'b0111;
    end
  endcase
 end
//Convert digit number to LED vector. LEDs are active low.
logic [6:0] sseg_LEDs;
always @(*)
 begin
  sseg_LEDs = 7'b1111111; //default
  case( digit_val)
   5'd0 : sseg_LEDs = 7'b1000000; //to display 0
```

```
   5'd1 : sseg_LEDs = 7'b1111001; //to display 1
   5'd2 : sseg_LEDs = 7'b0100100; //to display 2
   5'd3 : sseg_LEDs = 7'b0110000; //to display 3
   5'd4 : sseg_LEDs = 7'b0011001; //to display 4
   5'd5 : sseg_LEDs = 7'b0010010; //to display 5
   5'd6 : sseg_LEDs = 7'b0000010; //to display 6
   5'd7 : sseg_LEDs = 7'b1111000; //to display 7
   5'd8 : sseg_LEDs = 7'b0000000; //to display 8
   5'd9 : sseg_LEDs = 7'b0010000; //to display 9
   5'd10: sseg_LEDs = 7'b0001000; //to display a
   5'd11: sseg_LEDs = 7'b0000011; //to display b
   5'd12: sseg_LEDs = 7'b1000110; //to display c
   5'd13: sseg_LEDs = 7'b0100001; //to display d
   5'd14: sseg_LEDs = 7'b0000110; //to display e
   5'd15: sseg_LEDs = 7'b0001110; //to display f
   5'd16: sseg_LEDs = 7'b0110111; //to display "="
   default : sseg_LEDs = 7'b0111111; //dash
  endcase
 end
assign an = digit_en;
assign seg = sseg_LEDs;
assign dp = 1'b1; //turn dp off
endmodule
```