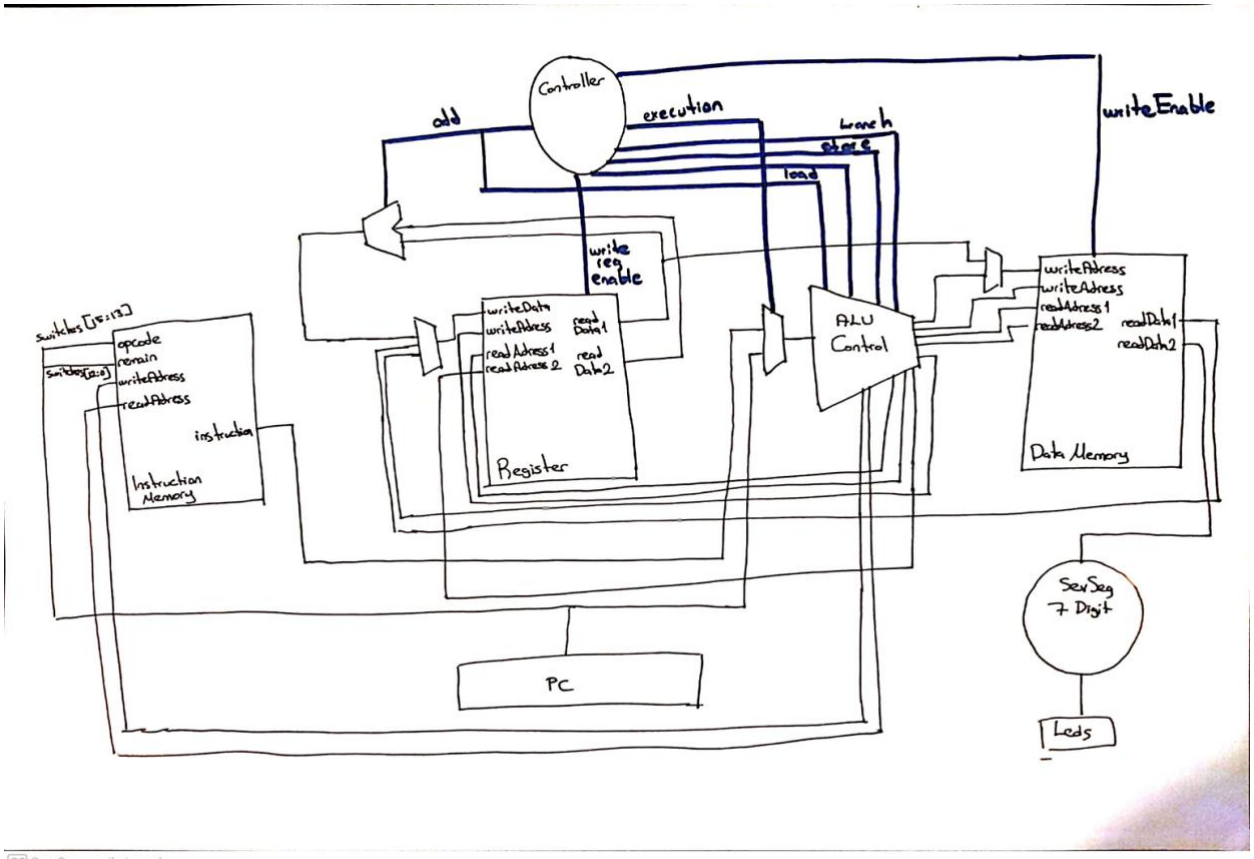
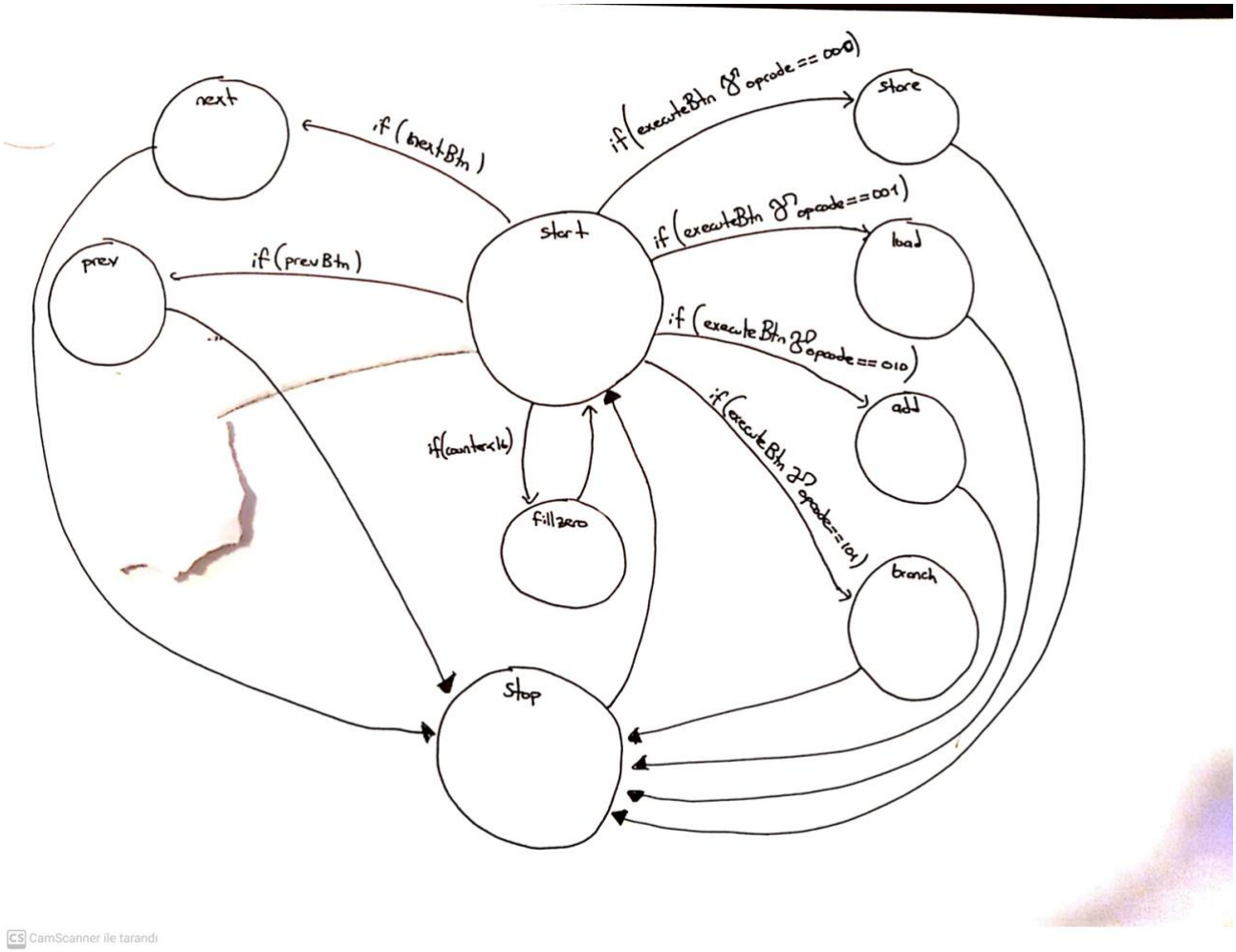
# CS 223 PROJECT

# TARIK BUĞRA KARALİ
# ID: 21703937
# SECTION :1




**Datapath-controller**

The picture given below is demostrates that datapath- controller of single cycle processor. The controller determines the conditions by signals. Execution signal determines the instruction which will be used for operations takes value from swithces or instruction memory.Write enable is controls the data written or not on data memory and write reg enable controls the data written or not on register. Also the Alu control takes the instruction and signals from the controller and according the signals condition, instruction is divided properly to send data memory or register or both of them as inputs.Also the add signal controls the sum operation from the register's data at the sepecified adresses. Seven segment display shows the data memory on board. Instruction memory holds the instruction written by user.




**State Diagram**

The state diagram given below demonstrates the top down designs work logic. Start condition goes to fillzero in order to ensure the data memory have 8'b00000000 elements on initial condition or after the reset.Then the pulse taken from the buttons determines the next state. If the execute command is delivered (middle button) , value of the opp code is determines the next state and in the strore,load,add ,branch states apply the operation according to instruction.After the operation done ,that states goes the stop and then goes to start state. It means that machine get ready to new command. Also if the next command is delivered (right button) , the next state is increment the memory address;if the previous command is delivered (left button) , the next state is deccrement the memory address and the operations seen in the seven segment display.If the down button is pressed , instruction is taken from instruction memory in the start state and again after the determining opcode , appropriate operation will be executed.The upper button is represent the reset and if the reset is pushed, next state will be start and counter takes the zero value so the data memory, register and instruction memory turns the initial condition.

**Debounce**: This module detect the button pressed and turns them to a pulse.

```
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
```

**Instruction Mememory:** This module holds the instruction and capable of carry  32 instruction.

```
module InstructionMem( input logic [2:0]opcode,
                       input logic [12:0]remain,
                       input logic [4:0] writeaddress,
                       input logic [4:0] address,
                       input  logic reset,
                       input  logic clock,
                       output logic    [15:0]   instruction
   );

   logic [15:0] IM [31:0];

   always_ff @ (posedge clock, posedge reset)
   begin
     if(reset)
```

```
        for( int j = 0; j <= 16; j++)
        begin
            IM[j] = 16'b000_0_0000_0000_0001;
        end

      else
        IM[writeaddress] [15:13] = opcode;
        IM[writeaddress] [12:0] = remain;
        instruction <= IM[address] ;
      end
endmodule
```

**Register:**

```
module register( input logic clock,
 input logic [3:0] writeAddress,
 input logic [7:0] writeData,
 input logic writeEnable,
 input   logic reset,
 input logic [3:0] readAddress1, readAddress2,
 output logic [7:0] readData1, readData2
 );
 logic [7:0] mem[15:0]  ;


 logic enable;
 logic [7:0] data;
 logic [3:0] address;
 logic [3:0] readAdd1, readAdd2;
 logic [7:0] readDat1, readDat2;

 assign enable = writeEnable;
 assign data = writeData;
 assign address = writeAddress;
 assign readAdd1 = readAddress1;
 assign readAdd2 = readAddress2;

 //always_comb
 always_ff @ (posedge clock)
 begin
  if(reset)
   for( int j = 0; j <= 16; j++)
      begin
            mem[j] = 8'b00000000;
       end
 if( writeEnable ) mem[writeAddress] <= writeData;


 readData1 <= mem[readAddress1];
 readData2 <= mem[readAddress2];
 end
 endmodule
```

**Data Memory:**

```
module memory( input logic clock,
```

```
 input logic [3:0] writeAddress,
 input logic [7:0] writeData,
 input logic writeEnable,
 input   logic reset,
 input logic [3:0] readAddress1, readAddress2,
 output logic [7:0] readData1, readData2
 );
 logic [7:0] mem[15:0]  ;


 logic enable;
 logic [7:0] data;
 logic [3:0] address;
 logic [3:0] readAdd1, readAdd2;
 logic [7:0] readDat1, readDat2;

 assign enable = writeEnable;
 assign data = writeData;
 assign address = writeAddress;
 assign readAdd1 = readAddress1;
 assign readAdd2 = readAddress2;

 //always_comb
 always_ff @ (posedge clock)
 begin
  if(reset)
   for( int j = 0; j <= 16; j++)
      begin
            mem[j] = 8'b00000000;
       end
 if( writeEnable ) mem[writeAddress] <= writeData;


 readData1 <= mem[readAddress1];
 readData2 <= mem[readAddress2];
 end
 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
```

**ALU:**

```
module ALU(    input logic clk,
               input logic [3:0] opcode,
               input logic [12:0] instruction,
               input logic pressed,
               output  logic [7:0] write_reg_data,
               output logic [3:0] write_reg_address,
               output logic [3:0] read_reg_address,
               output logic [7:0] read_reg_data,
               output logic [7:0] write_data,
               output logic [3:0] write_address,
               output logic [3:0] read_address1,
               output logic [3:0] read_address2,
               output logic [7:0] read_data1,
               output logic [7:0] read_data2
    );


always @(*)
begin
 case(opcode)
 3'b000:  // Store0
   begin
     if(pressed)
     begin
        if(instruction[12] == 0 ) //Immediate bit
        begin
              read_reg_address = instruction [7:4];
              write_data =  read_reg_data;
              write_address = instruction [3:0];
        end

        if(instruction[12] == 1 ) //Immediate bit
```

```
      begin
            write_address = instruction [11:8];
            write_data = instruction [7:0];
      end
    end
  end
 3'b001:  // data_processing
 begin
  if(pressed)
  begin
        if(instruction[12] == 0 ) //Immediate bit
        begin
            read_address1 = instruction [7:4];
            write_reg_address = instruction [7:4];
            write_reg_data = read_data1;
        end

        if(instruction[12] == 1 ) //Immediate bit
        begin
            write_reg_data = instruction [7:0];
            write_reg_address = instruction [11:8];
        end
  end
 end
 default: begin
            read_reg_address = instruction [7:4];
            write_data =  read_reg_data;
            write_address = instruction [3:0];
  end
 endcase
 end

endmodule
```

**Top Design:**

```
module topDesign( input logic clk, reset,
                  input logic [2:0] opcode,
                  input logic [12:0] instruction,
                  input   logic   nextDisplay, prevDisplay, executDisplay,show,
                  output logic [15:0] led,
                  output [6:0] seg, logic dp,
                  output [3:0] an
                  );

    logic nextBtn;
    logic prevBtn;
    logic executeBtn;
    logic resetBtn;
    logic showBtn;

    debounce buttonNextDebouncer( clk, nextDisplay, nextBtn);
    debounce buttonPreviousDebouncer( clk, prevDisplay, prevBtn);
    debounce buttonMiddleDebouncer( clk,executDisplay, executeBtn );
```

```
    debounce buttonResetDebouncer( clk, reset, resetBtn );
    debounce buttonDownDebouncer( clk, show, showBtn );

    //IM variables
    logic [4:0] current_IM_address;
    logic [4:0] write_im_address;
    logic [4:0] readIMAddress;
    logic [15:0] readDataIM;

    //DataMem variables
    logic [3:0] write_address;
    logic [7:0] write_data;
    logic [7:0] readData1;
    logic [7:0] readData2;
    logic writeEnable;
    logic [3:0] read_address1;
    logic [3:0] read_address2;

    //Reg variables
    logic writeRegEnable;
    logic [3:0] read_reg_address1;
    logic [3:0] read_reg_address2;
    logic [3:0] write_reg_address;
    logic [7:0] read_reg_data1;
    logic [7:0] read_reg_data2;
    logic [7:0] write_reg_data;
   //control
    logic  executeSignal;

    assign led = readDataIM;

    InstructionMem im( opcode,instruction, write_im_address,current_IM_address, reset,clk, readDataIM);
    memory dataMem( clk,write_address, write_data, writeEnable, reset, read_address1, read_address2, readData1, readData2 );
    register rf( clk,write_reg_address, write_reg_data, writeRegEnable , reset, read_reg_address1, read_reg_address2, read_reg_data1,read_reg_data2 );
    SevSeg_4digit sev( clk, read_address1, 0, readData1[7:4], readData1[3:0], seg, dp, an);


    typedef enum logic [3:0] { start,fillzero, store, load, add, branch, stop, next, prev } statetype;
    statetype [3:0] state, nextstate;

    logic [4:0] counter = 0;

    always_ff @ ( posedge clk)
    begin
        if( resetBtn )
        begin
            state <= start;
            current_IM_address <= 0;
        end
```

```
else
   state <= nextstate;

case( state)
   start:
   begin
      writeEnable = 0;
      writeRegEnable = 0;
      write_im_address = 0;
      readIMAddress = current_IM_address;
      if (counter < 16) nextstate = fillzero;
      else
      begin
        if( executeBtn )
        begin
          executeSignal = 1;
          write_im_address =(write_im_address + 5'd1);

          if( opcode == 3'b000 )nextstate = store;
          else if( opcode == 3'b001 )nextstate = load;
          else if( opcode == 3'b010)nextstate = add;
          else if( opcode == 3'b101)nextstate = branch;
          else if( opcode == 3'b111 )nextstate = stop;
        end

        else if( nextBtn ) nextstate = next;
        else if( prevBtn ) nextstate = prev;
        else if( showBtn )
        begin
          if( readDataIM[15:13] == 3'b000)nextstate = store;
          else if( readDataIM[15:13] == 3'b001) nextstate = load;
          else if( readDataIM[15:13] == 3'b010)nextstate = add;
          else if( readDataIM[15:13] == 3'b101)nextstate = branch;
        end

        else
        begin
          nextstate = start;
        end
      end
   end

 fillzero:
      begin
      writeEnable = 1;
      write_address <= counter;
      writeRegEnable = 1;
      write_reg_address <= counter;
      write_reg_data <= 0;
      counter = counter + 1;
      nextstate = start;
```

```
    end

store:
begin
    writeEnable = 1;
    if( executeSignal == 1)
    begin
        if( instruction[12] == 1 )
        begin
            write_address = instruction [11:8];
            write_data = instruction[7:0];

            read_address1 = write_address;
        end

        else if ( instruction[12] == 0 )
        begin

            read_reg_address1 = instruction[3:0];
            write_address = instruction[7:4];
             write_data = read_reg_data1;
        end
    end

    else if( executeSignal == 0 )
    begin
        readIMAddress = current_IM_address;

        if( readDataIM[12] == 1 )
        begin
            write_address = readDataIM[11:8];
            write_data = readDataIM[7:0];
        end

        else if( readDataIM[12] == 0)
        begin
            read_reg_address1 = readDataIM[3:0];
            write_address = readDataIM[7:4];
            write_data = read_reg_data1;
        end

        if( current_IM_address == 32 )
            current_IM_address = 0;
        else
            current_IM_address = current_IM_address + 1;

    end

    executeSignal = 0;
    nextstate = stop;
end
```

```
load:
begin
   writeEnable = 0;
   writeRegEnable = 1;

   if ( executeSignal == 1 )
   begin
      if( instruction[12] == 1 )
      begin
         write_reg_address = instruction[11:8];
         write_reg_data = instruction[7:0];
      end

      else if ( instruction[12] == 0 )
      begin
         read_address2 = instruction[3:0];
         write_reg_address = instruction[7:4];
         write_reg_data = readData2;
      end

   end
   else if( executeSignal == 0 )
   begin
      readIMAddress = current_IM_address;

      if( readIMAddress[12] == 1 )
      begin
         write_reg_address = readDataIM[11:8];
         write_reg_data = readDataIM[7:0];
      end

      else if( readIMAddress[12] == 0 )
      begin
        read_address2 = readDataIM[3:0];
         write_reg_address = readDataIM[7:4];
         write_reg_data = readData2;
      end

      if( current_IM_address == 9 )
         current_IM_address = 0;
      else
         current_IM_address = current_IM_address + 1;

   end

   executeSignal = 0;
   nextstate = stop;
end

add:
begin
   writeRegEnable = 1;
```

```
    if( executeSignal == 1 )
    begin
        write_reg_address = instruction[11:8];
        read_reg_address1 = instruction[7:4];
        read_reg_address2 = instruction[3:0];
        write_reg_data = read_reg_data1 + read_reg_data2;
    end

    else if ( executeSignal == 0 )
    begin
        readIMAddress = current_IM_address;
        write_reg_address = readDataIM[11:8];
        read_reg_address1 = readDataIM[7:4];
        read_reg_address2 = readDataIM[3:0];
        write_reg_data = read_reg_data1 + read_reg_data2;

        if( current_IM_address == 9 )
            current_IM_address = 0;
        else
            current_IM_address = current_IM_address + 1;

    end

    executeSignal = 0;
    nextstate = stop;

end

branch:
begin
    if( executeSignal == 1 )
    begin
        read_reg_address1 = instruction[7:4];
        read_reg_address2 = instruction[3:0];

        if( read_reg_data1 == read_reg_data2 )
            current_IM_address = instruction[12:8];
        else
        begin
            if( current_IM_address == 32 )
                current_IM_address = 0;
            else
                current_IM_address = current_IM_address + 1;
        end

    end

    else if (executeSignal == 0 )
    begin
        readIMAddress = current_IM_address;
```

```
        read_reg_address1 = readDataIM[7:4];
        read_reg_address2 = readDataIM[3:0];

        if( read_reg_data1 == read_reg_data2)
           current_IM_address = readDataIM[12:8];
        else
        begin
           if( current_IM_address == 9 )
              current_IM_address = 0;
           else
              current_IM_address = current_IM_address + 1;
        end
     end

     executeSignal = 0;
     nextstate = stop;
  end

  stop:
  begin
     nextstate = start;
  end

  next:
  begin
     if( read_address1 == 'd15 )
     begin
        read_address1 = 0;
     end

     else
     begin
        read_address1 = read_address1 + (1'b1);
     end

     nextstate = stop;
  end

  prev:
  begin
     if( read_address1 == 0 )
     begin
        read_address1 = 'd15;
     end

     else
     begin
        read_address1 = read_address1 - (1'b1);
     end

     nextstate = stop;
  end
```

```
            default:
                nextstate = state;
        endcase
    end
endmodule
```