**Prelab for Lab6**

Xiaoyu Zhou, 1004081147

**Part I**

2. The **resetn** signal is a synchronous reset and it is active low. When I do the simulation, I need to begin with setting **resetn** to be 0.

3. Verilog Code:

// SW[0]: reset signal

// SW[1]: input signal (w)

// KEY[0]: clock

// LEDR[2:0]: current state

// LEDR[9]: output (z)

module sequence\_detector(SW, KEY, LEDR);

input [9:0] SW;

input [3:0] KEY;

output [9:0] LEDR;

wire w, clock, resetn, z;

reg [2:0] y\_Q, Y\_D; // y\_Q represents current state, Y\_D represents next state

localparam A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100, F = 3'b101, G = 3'b110;

// Connect inputs and outputs to internal wires

assign w = SW[1];

assign clock = ~KEY[0];

assign resetn = SW[0];

assign LEDR[9] = z;

assign LEDR[2:0] = y\_Q;

// State table

// The state table should only contain the logic for state transitions

// Do not mix in any output logic. The output logic should be handled separately.

// This will make it easier to read, modify and debug the code.

always @(\*)

begin // Start of state\_table

case (y\_Q)

A: begin

if (!w) Y\_D = A;

else Y\_D = B;

end

B: begin

if (!w) Y\_D = A;

else Y\_D = C;

end

C: begin

if (!w) Y\_D = E;

else Y\_D = D;

end

D: begin

if(!w) Y\_D = E;

else Y\_D = F;

end

E: begin

if(!w) Y\_D = A;

else Y\_D = G;

end

F: begin

if(!w) Y\_D = E;

else Y\_D = F;

end

G: begin

if(!w) Y\_D = A;

else Y\_D = C;

end

default: Y\_D = A;

endcase

end // End of state\_table

// State Register (i.e., FFs)

always @(posedge clock)

begin // Start of state\_FFs (state register)

if(resetn == 1'b0)

y\_Q <= A;

else

y\_Q <= Y\_D;

end // End of state\_FFs (state register)

// Output logic

// Set z to 1 to turn on LED when in relevant states

assign z = ((y\_Q == F) || (y\_Q == G)); // To be completed by you!

endmodule

4. Simulation.

图片包含 屏幕截图, 监视器, 墙壁, 电视

描述已自动生成

图片包含 文字

描述已自动生成**Part II**

2. Table

图片包含 文字

描述已自动生成

3. State

4. Verilog

module control(

input clk,

input resetn,

input go,

output reg ld\_a, ld\_b, ld\_c, ld\_x, ld\_r,

output reg ld\_alu\_out,

output reg [1:0] alu\_select\_a, alu\_select\_b,

output reg alu\_op

);

reg [3:0] current\_state, next\_state;

localparam S\_LOAD\_A = 4'd0,

S\_LOAD\_A\_WAIT = 4'd1,

S\_LOAD\_B = 4'd2,

S\_LOAD\_B\_WAIT = 4'd3,

S\_LOAD\_C = 4'd4,

S\_LOAD\_C\_WAIT = 4'd5,

S\_LOAD\_X = 4'd6,

S\_LOAD\_X\_WAIT = 4'd7,

S\_CYCLE\_0 = 4'd8,

S\_CYCLE\_1 = 4'd9,

S\_CYCLE\_2 = 4'd10,

S\_CYCLE\_3 = 4'd11,

S\_CYCLE\_4 = 4'd12;

// Next state logic aka our state table

always@(\*)

begin: state\_table

case (current\_state)

S\_LOAD\_A: next\_state = go ? S\_LOAD\_A\_WAIT : S\_LOAD\_A; // Loop in current state until value is input

S\_LOAD\_A\_WAIT: next\_state = go ? S\_LOAD\_A\_WAIT : S\_LOAD\_B; // Loop in current state until go signal goes low

S\_LOAD\_B: next\_state = go ? S\_LOAD\_B\_WAIT : S\_LOAD\_B; // Loop in current state until value is input

S\_LOAD\_B\_WAIT: next\_state = go ? S\_LOAD\_B\_WAIT : S\_LOAD\_C; // Loop in current state until go signal goes low

S\_LOAD\_C: next\_state = go ? S\_LOAD\_C\_WAIT : S\_LOAD\_C; // Loop in current state until value is input

S\_LOAD\_C\_WAIT: next\_state = go ? S\_LOAD\_C\_WAIT : S\_LOAD\_X; // Loop in current state until go signal goes low

S\_LOAD\_X: next\_state = go ? S\_LOAD\_X\_WAIT : S\_LOAD\_X; // Loop in current state until value is input

S\_LOAD\_X\_WAIT: next\_state = go ? S\_LOAD\_X\_WAIT : S\_CYCLE\_0; // Loop in current state until go signal goes low

S\_CYCLE\_0: next\_state = S\_CYCLE\_1;

S\_CYCLE\_1: next\_state = S\_CYCLE\_2;

S\_CYCLE\_2: next\_state = S\_CYCLE\_3;

S\_CYCLE\_3: next\_state = S\_CYCLE\_4;

S\_CYCLE\_4: next\_state = S\_LOAD\_A; // we will be done our two operations, start over after

default: next\_state = S\_LOAD\_A;

endcase

end // state\_table

// Output logic aka all of our datapath control signals

always @(\*)

begin: enable\_signals

// By default make all our signals 0

ld\_alu\_out = 1'b0;

ld\_a = 1'b0;

ld\_b = 1'b0;

ld\_c = 1'b0;

ld\_x = 1'b0;

ld\_r = 1'b0;

alu\_select\_a = 2'b00;

alu\_select\_b = 2'b00;

alu\_op = 1'b0;

case (current\_state)

S\_LOAD\_A: begin

ld\_a = 1'b1;

end

S\_LOAD\_B: begin

ld\_b = 1'b1;

end

S\_LOAD\_C: begin

ld\_c = 1'b1;

end

S\_LOAD\_X: begin

ld\_x = 1'b1;

end

S\_CYCLE\_0: begin // Do A <- A \* x

ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result back into A

alu\_select\_a = 2'b00; // Select register A

alu\_select\_b = 2'b11; // Also select register A

alu\_op = 1'b1; // Do multiply operation

end

S\_CYCLE\_1: begin // Do A <- Ax \* x

ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result back into A

alu\_select\_a = 2'b00; // Select register A

alu\_select\_b = 2'b11; // Also select register A

alu\_op = 1'b1; // Do multiply operation

end

S\_CYCLE\_2: begin // Do B <- B \* x

ld\_alu\_out = 1'b1; ld\_b = 1'b1; // store result back into B

alu\_select\_a = 2'b01; // Select register B

alu\_select\_b = 2'b11; // Also select register x

alu\_op = 1'b1; // Do multiply operation

end

S\_CYCLE\_3: begin // Do A <- Axx + Bx

ld\_alu\_out = 1'b1; ld\_a = 1'b1; // store result back into A

alu\_select\_a = 2'b00; // Select register A

alu\_select\_b = 2'b01; // Also select register B

alu\_op = 1'b0; // Do multiply operation

end

S\_CYCLE\_4: begin

ld\_r = 1'b1; // store result in result register

alu\_select\_a = 2'b00; // Select register A

alu\_select\_b = 2'b10; // Select register C

alu\_op = 1'b0; // Do Add operation

end

// default: // don't need default since we already made sure all of our outputs were assigned a value at the start of the always block

endcase

end // enable\_signals

// current\_state registers

always@(posedge clk)

begin: state\_FFs

if(!resetn)

current\_state <= S\_LOAD\_A;

else

current\_state <= next\_state;

end // state\_FFS

endmodule

5. 图片包含 文字, 天空

描述已自动生成

6. Simulation

图片包含 屏幕截图

描述已自动生成