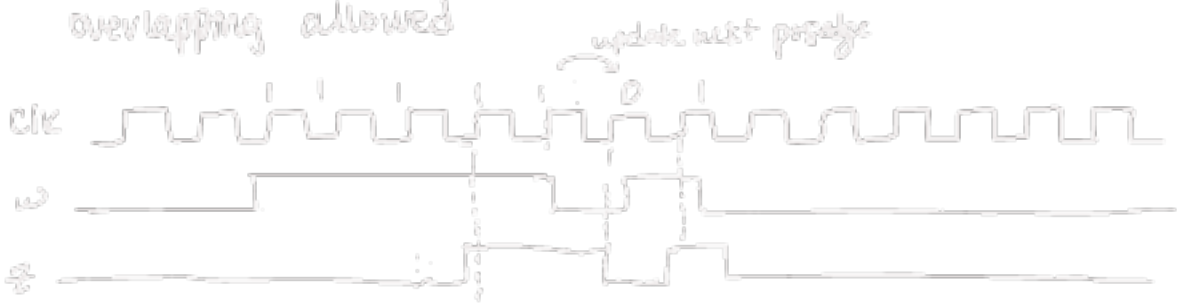
Part 1

1111 or 1101

input w → output z (1 if 1111 or 1101 pulses)

overlapping allowed



1) resetn is active low. It is synchronous, because it is inside the "posedge" clock cycle code, so changes w/ the clock cycle. ↳ b/c if(!resetn)

Simulation to reset FSM to starting state: set resetn = 0

Part 2

FSM to control a datapath. $Cx^3 + Bx + A$ ↳ 8 bit unsigned

E_x	E_a	E_b	E_c
1	1	1	1
x	A	B	C

reset is not clock, use Cyclone-50 50MHz clock

1) Currently $A^2 + C$

2) 1. load data into 4 registers C, X, A, B

2. compute Bx, load into B

3. compute Bx + A, load into B

4. compute x^2, load into A

5. compute Cx^2, load into Ax

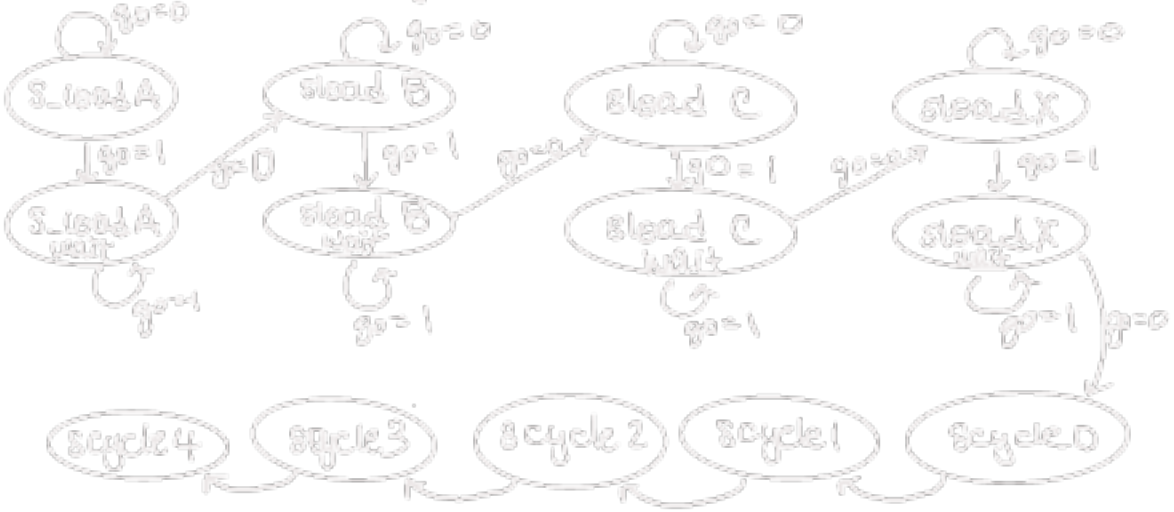
6. compute $ax^2 + Bx + A$, load into out register R

A=0, B=1, C=2, X=3

Steps	Register State RA RB RC RX	Control Signals
• load C into RC	C	• ld_C = 1
• load X into RX	X	• ld_x = 1
• load A into RA	A	• ld_a = 1, ld_alu_out = 0 Iselect signal
• load B into RB	B	• ld_b = 1, ld_alu_out = 0
• multiply RB & Rx, store in RB	BX	• alu-select_a = 01 alu-select_b = 11 alu_op = 1 ld_alu_out = 1 ld_b = 1
• add RB & RA, store in RB	BXA	• alu-select_a = 01 alu-select_b = 00 alu_op = 0 ld_alu_out = 1 ld_b = 1
• multiply Rx & Rx, store in RA	X^2	• alu-select_a = 11 alu-select_b = 11 alu_op = 1 ld_alu_out = 1 ld_a = 1
• multiply RA & RC store in RA	CX^2	• alu-select_a = 00 alu-select_b = 10 alu_op = 1 ld_alu_out = 1 ld_a = 1
• add RA & RB store in R		• alu-select_a = 00 alu-select_b = 01 alu_op = 0 ld_r = 1

3) Controller FSM

- used 2 more clock cycles, 5 states in total



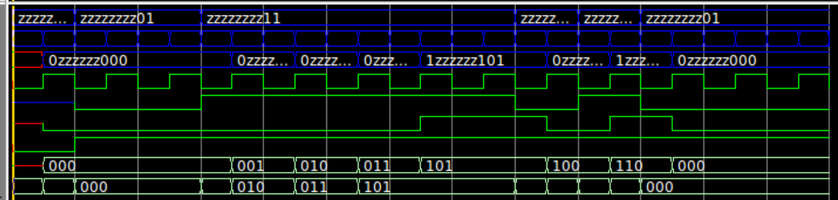
KEY[3] - reset (active low)

KEY[1] - go

```
// 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));
endmodule
```

```
vlib work
vlog -timescale 1ns/1ns sequence_detector.v
vsim sequence_detector

log {/*}
add wave {/*}


# clk signal, start at 1 then go to 0
force {KEY[0]} 1 0, 0 10 -r 20

# run 200 ns

# reset signal
force {SW[0]} 0

run 20 ns

# turn signal off, active low
force {SW[0]} 1

# specify w
force {SW[1]} 0

run 40 ns

force {SW[1]} 1
run 100 ns

force {SW[1]} 0
run 20 ns

force {SW[1]} 1
run 20 ns

force {SW[1]} 0
run 60 ns
```

Msgs	
/sequence_detector/SW	zzzzzzzz0
/sequence_detector/KEY	zzz1
/sequence_detector/LEDR	xzzzzzzxxx
/sequence_detector/clock	St0
/sequence_detector/w	HiZ
/sequence_detector/z	StX
/sequence_detector/resetn	St0
/sequence_detector/y_Q	xxx
/sequence_detector/Y_D	000



Now	260 ns
Cursor 1	0 ns

```
//Sw[7:0] data_in

//KEY[0] synchronous reset when pressed
//KEY[1] go signal

//LEDR displays result
//HEX0 & HEX1 also displays result

module fpga_top(SW, KEY, CLOCK_50, LEDR, HEX0, HEX1);
    input [9:0] SW;
    input [3:0] KEY;
    input CLOCK_50;
    output [9:0] LEDR;
    output [6:0] HEX0, HEX1;

    wire resetn;
    wire go;

    wire [7:0] data_result;
    assign go = ~KEY[1];
    assign resetn = KEY[0];

    part2 u0(
        .clk(CLOCK_50),
        .resetn(resetn),
        .go(go),
        .data_in(SW[7:0]),
        .data_result(data_result)
    );

    assign LEDR[9:0] = {2'b00, data_result};

    hex_decoder H0(
        .hex_digit(data_result[3:0]),
        .segments(HEX0)
        );

    hex_decoder H1(
        .hex_digit(data_result[7:4]),
        .segments(HEX1)
        );

endmodule

module part2(
    input clk,
    input resetn,
    input go,
    input [7:0] data_in,
    output [7:0] data_result
    );

    // lots of wires to connect our datapath and control
    wire ld_a, ld_b, ld_c, ld_x, ld_r;
    wire ld_alu_out;
    wire [1:0] alu_select_a, alu_select_b;
    wire alu_op;

    control C0(
        .clk(clk),
        .resetn(resetn),

        .go(go),

        .ld_alu_out(ld_alu_out),
        .ld_x(ld_x),
        .ld_a(ld_a),
        .ld_b(ld_b),
        .ld_c(ld_c),
        .ld_r(ld_r),

        .alu_select_a(alu_select_a),
        .alu_select_b(alu_select_b),
        .alu_op(alu_op)
    );

    datapath D0(
        .clk(clk),
        .resetn(resetn),

        .ld_alu_out(ld_alu_out),
        .ld_x(ld_x),
        .ld_a(ld_a),
        .ld_b(ld_b),
        .ld_c(ld_c),
        .ld_r(ld_r),

        .alu_select_a(alu_select_a),
        .alu_select_b(alu_select_b),
        .alu_op(alu_op),

        .data_in(data_in),
        .data_result(data_result)
```

```
    );

endmodule


module control(
    input clk,
    input reset,
    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
                ld_alu_out = 1'b1; ld_b = 1'b1;
                alu_select_a = 2'b01;
                alu_select_b = 2'b11;
                alu_op = 1'b1;
            end
            S_CYCLE_1: begin
                ld_alu_out = 1'b1; ld_b = 1'b1;
                alu_select_a = 2'b01;
                alu_select_b = 2'b00;
                alu_op = 1'b0;
            end
```

```
            S_CYCLE_2: begin
               ld_alu_out = 1'b1; ld_a = 1'b1;
               alu_select_a = 2'b11;
               alu_select_b = 2'bl l;
               alu_op = 1'b1;
            end
            S_CYCLE_3: begin
               ld_alu_out = 1'b1; ld_a = 1'b1;
               alu_select_a = 2'b00;
               alu_select_b = 2'bl 0;
               alu_op = 1'b1;
            end
            S_CYCLE_4: begin
               ld_r = 1'b1; // output register
               alu_select_a = 2'b00;
               alu_select_b = 2'b0 1;
               alu_op = 1'b0;
            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(!reset_n)
            current_state <= S_LOAD_A;
         else
            current_state <= next_state;
      end // state_FFS
endmodule

module datapath(
      input clk,
      input reset_n,
      input [7:0] data_in,
      input ld_alu_out,
      input ld_x, ld_a, ld_b, ld_c,
      input ld_r,
      input alu_op,
      input [1:0] alu_select_a, alu_select_b,
      output reg [7:0] data_result
      );

      // input registers
      reg [7:0] a, b, c, x;

      // output of the alu
      reg [7:0] alu_out;
      // alu input muxes
      reg [7:0] alu_a, alu_b;

      // Registers a, b, c, x with respective input logic
      always @ (posedge clk) begin
         if (!reset_n) begin
            a <= 8'd0;
            b <= 8'd0;
            c <= 8'd0;
            x <= 8'd0;
         end
         else begin
            if (ld_a)
               a <= ld_alu_out ? alu_out : data_in; // load alu_out if load_alu_out signal is high, otherwise load from data_in
            if (ld_b)
               b <= ld_alu_out ? alu_out : data_in; // load alu_out if load_alu_out signal is high, otherwise load from data_in
            if (ld_x)
               x <= data_in;

            if (ld_c)
               c <= data_in;
         end
      end

      // Output result register
      always @ (posedge clk) begin
         if (!reset_n) begin
            data_result <= 8'd0;
         end
         else
            if( ld_r)
               data_result <= alu_out;
      end

      // The ALU input multiplexers
      always @(*)
      begin
         case (alu_select_a)
            2'd0 :
               alu_a = a;
            2'd1 :
               alu_a = b;
            2'd2 :
               alu_a = c;
```

```
            2'd3 :
               alu_a = x;
            default: alu_a = 8'd0 ;
         endcase

         case (alu_select_b)
            2'd0 :
               alu_b = a;
            2'd1 :
               alu_b = b;
            2'd2 :
               alu_b = c;
            2'd3 :
               alu_b = x;
            default: alu_b = 8'd0;
         endcase
      end

      // The ALU
      always @(*)
      begin : ALU
         // alu
         case (alu_op)
            0 : begin
                  alu_out = alu_a + alu_b; //performs addition
               end
            1 : begin
                  alu_out = alu_a * alu_b; //performs multiplication
               end
            default: alu_out = 8'd0 ;
         endcase
      end

endmodule


module hex_decoder( hex_digit, segments);
   input [3:0] hex_digit;
   output reg [6:0] segments;

   always @(*)
      case (hex_digit)
         4'h0 : segments = 7'b100_0000;
         4'h1 : segments = 7'b111_1001;
         4'h2 : segments = 7'b010_0100;
         4'h3 : segments = 7'b011_0000;
         4'h4 : segments = 7'b001_1001;
         4'h5 : segments = 7'b001_0010;
         4'h6 : segments = 7'b000_0010;
         4'h7 : segments = 7'b111_1000;
         4'h8 : segments = 7'b000_0000;
         4'h9 : segments = 7'b001_1000;
         4'hA : segments = 7'b000_1000;
         4'hB: segments = 7'b000_0011;
         4'hC : segments = 7'b100_0110;
         4'hD : segments = 7'b010_0001;
         4'hE : segments = 7'b000_0110;
         4'hF: segments = 7'b000_1110;
         default: segments = 7'b111;
      endcase
endmodule
```

```
vlib work
vlog -timescale 1ns/1ns fpga_top.v
vsim fpga_top

log {/*}
add wave {/*}


# clk signal
force {CLOCK_50} 0 0, 1 2 -r 4

force {KEY[0]} 1

# gp
force {KEY[1]} 0

# load in A
force {SW[0]} 0
force {SW[1]} 1
force {SW[2]} 1
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0

run 10 ns

# gp signal
force {KEY[1]} 1

run 10 ns


# gp
force {KEY[1]} 0

# load in B
force {SW[0]} 1
force {SW[1]} 1
force {SW[2]} 1
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0

run 10 ns

# gp signal
force {KEY[1]} 1

run 10 ns

# gp
force {KEY[1]} 0

# load in C
force {SW[0]} 0
force {SW[1]} 1
force {SW[2]} 1
force {SW[3]} 1
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0

run 10 ns

# gp signal
force {KEY[1]} 1

run 10 ns

# gp
force {KEY[1]} 0

# load in X
force {SW[0]} 1
force {SW[1]} 1
force {SW[2]} 0
force {SW[3]} 0
force {SW[4]} 0
force {SW[5]} 0
force {SW[6]} 0
force {SW[7]} 0

run 10 ns

# gp signal
force {KEY[1]} 1

run 50 ns
```

/fpga_top/SW	zz00000011
/fpga_top/KEY	zz11
/fpga_top/CLOC...	St1
/fpga_top/LEDR	0010011001
[9]	St0
[8]	St0
[7]	St1
[6]	St0
[5]	St0
[4]	St1
[3]	St1
[2]	St0
[1]	St0
[0]	St1
/fpga_top/HEX0	0011000
/fpga_top/HEX1	0011000
/fpga_top/resetn	St1
/fpga_top/go	St0
/fpga_top/data_...	10011001

Msgs

zz00000110 zz00000111 zz00000110 zz00000011

zz01 zz11 zz01 zz11 zz01 zz11 zz01 zz11

00xxxxxxx 0010011001

0011000

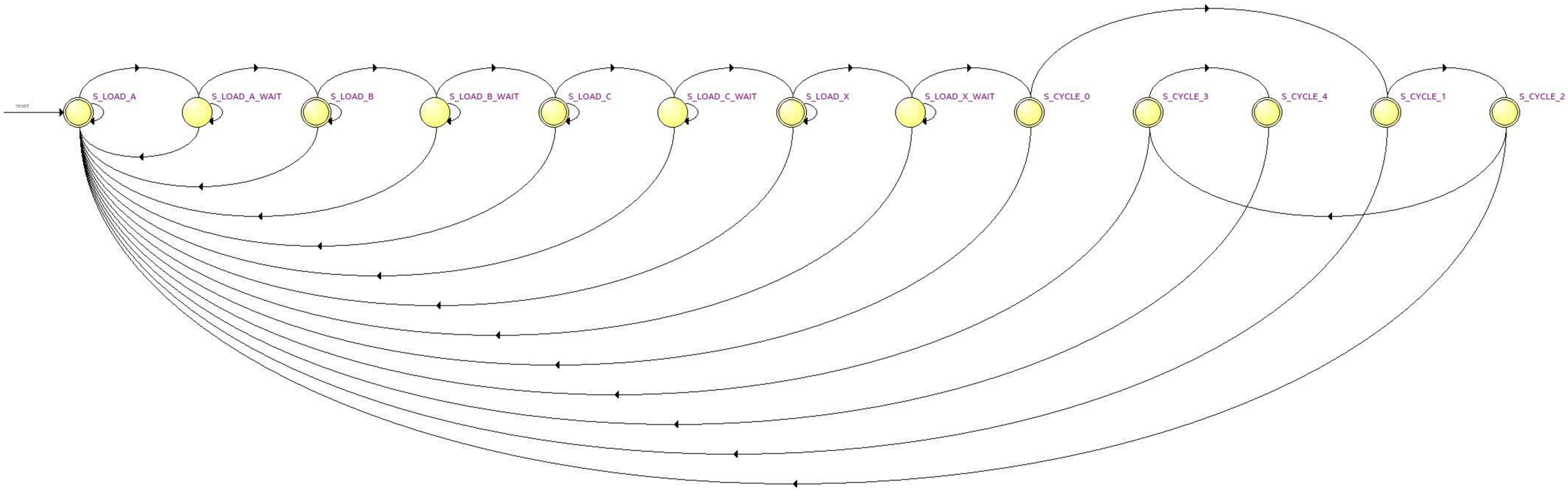
0011000

10011001

Now	120 ns
Cursor 1	115 ns

ns 20 ns 40 ns 60 ns 80 ns 100 ns 120 ns 140 ns 160 ns 180 ns 200 ns 220 ns 240 ns 260 ns 280 ns

115 ns


reset
S_LOAD_A
S_LOAD_A_WAIT
S_LOAD_B
S_LOAD_B_WAIT
S_LOAD_C
S_LOAD_C_WAIT
S_LOAD_X
S_LOAD_X_WAIT
S_CYCLE_0
S_CYCLE_3
S_CYCLE_4
S_CYCLE_1
S_CYCLE_2