PART I

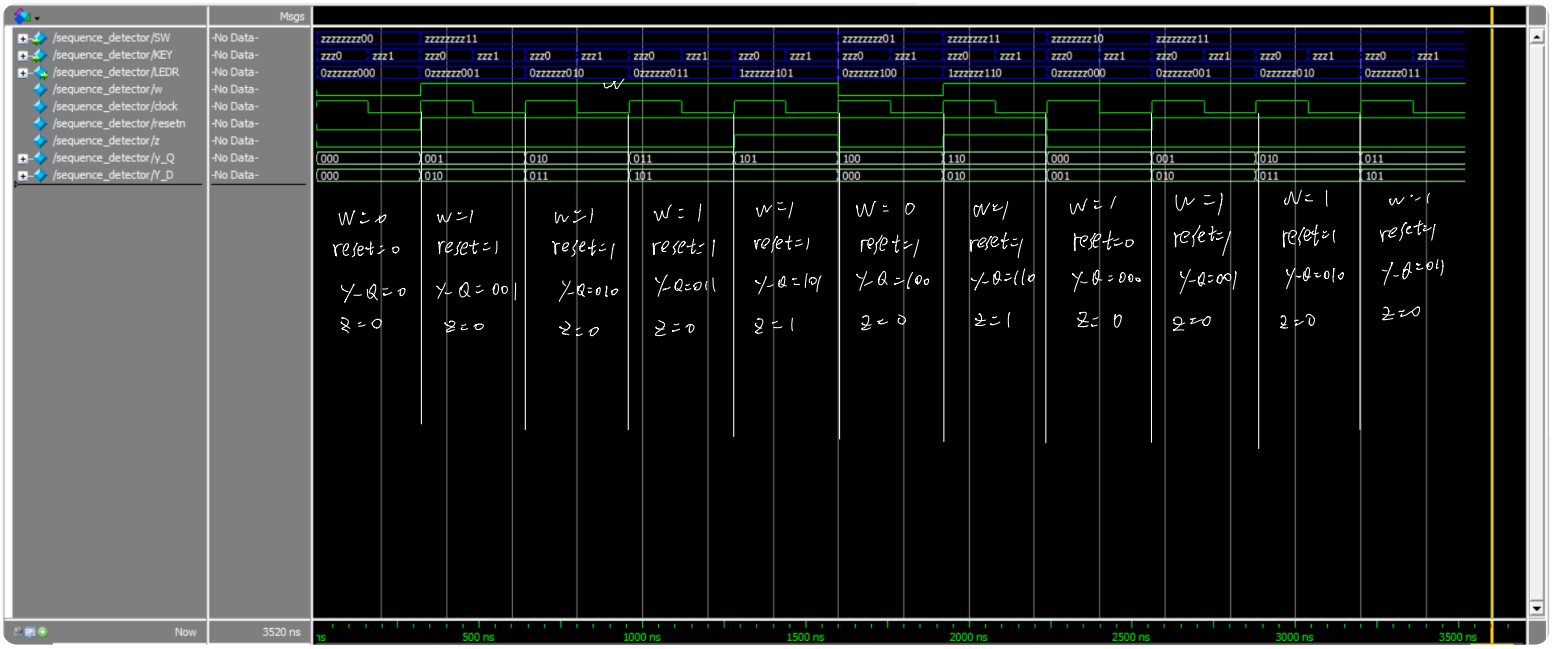
1. Synchronous;
   Active low;
   Keep sw[0] low in a block cycle

2. 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));
endmodule
```

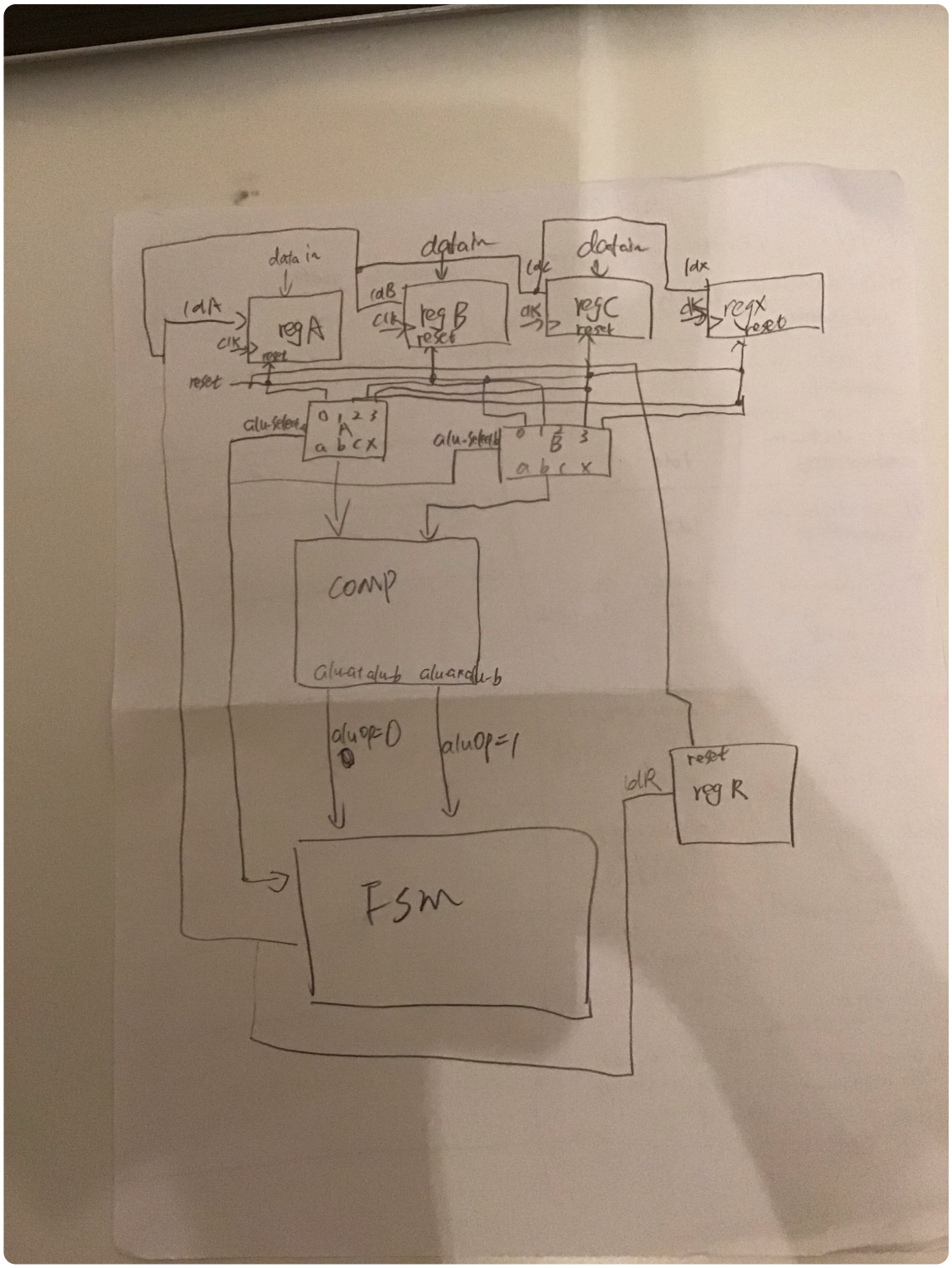
1101

3. Simulation



PART II

2. datapath



Table

ld_a, ld_b, ld_c, ld_x, ld_alu_out, alu_select_a, alu_select_b, alu_op, ldr

RTN	control signals activated	control vector
S_0: a = data_in	ld_a	1,0,0,0,0,0,0,0,0
S_1: b = data_in	ld_b	0,1,0,0,0,0,0,0,0
S_2: c = data_in	ld_c	0,0,1,0,0,0,0,0,0
S_3: x = data_in	ld_x	0,0,0,1,0,0,0,0,0

S_4: a ← alu_out
alu_a ← a
alu_b ← b
alu_out ← alu_a * alu_b
	ld_alu_out : ld_a;
	alu_select_a(00);
	alu_select_b(11);
	alu_op = 1
		1,0,0,0,1,00,11,1,0

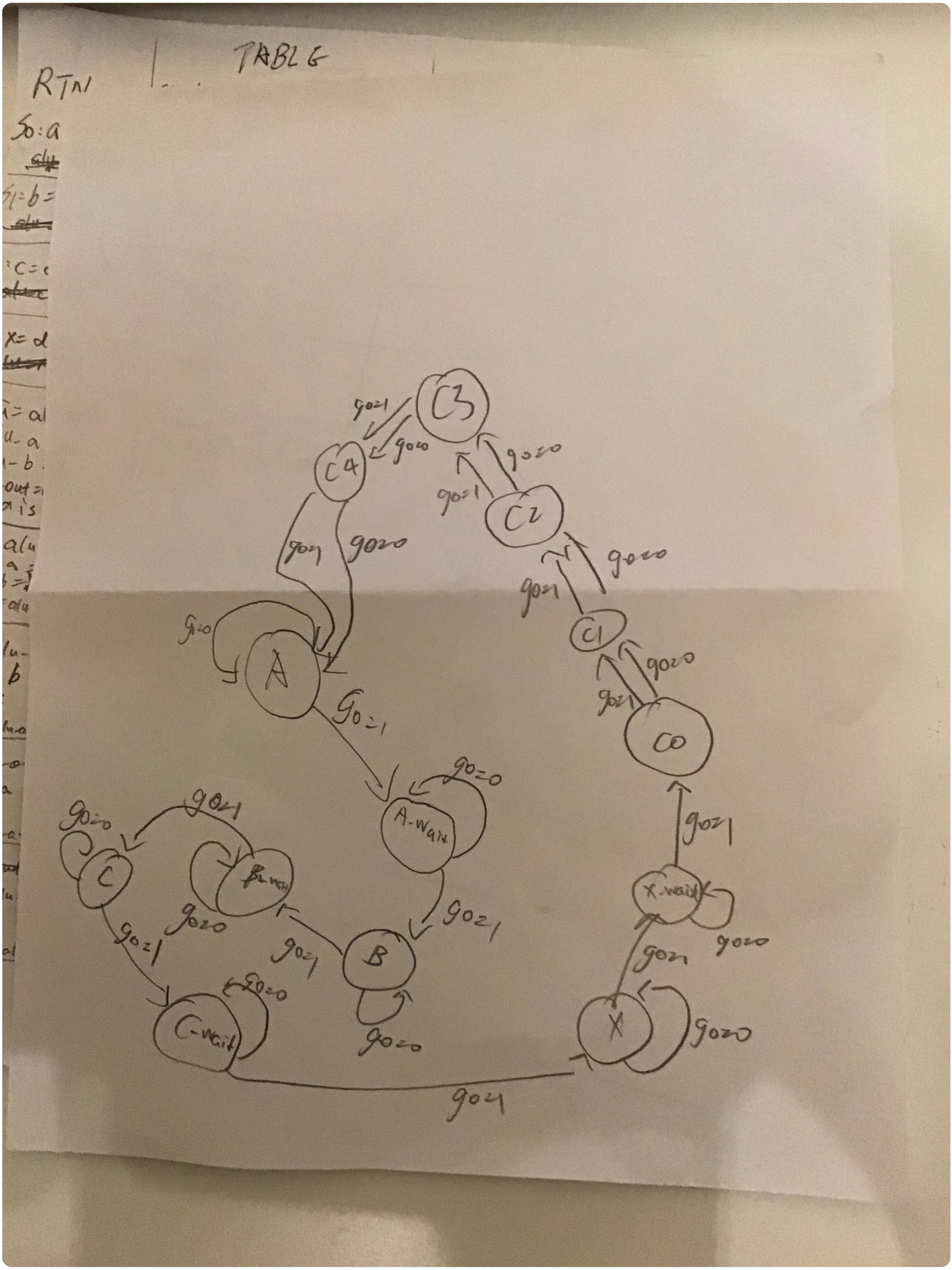
S_5: a ← alu_out alu_a ← a alu_b ← x alu_out ← alu_a * alu_b	ld_alu_out : ld_a; alu_select_a(00); alu_select_b(11); alu_op = 1	1,0,0,0,1,00,11,1,0

S_6: b ← alu_out
alu_a ← b
alu_b ← x
alu_out ← alu_a * alu_b
	ld_alu_out ; ld_b;
	alu_select_a(01);
	alu_select_b(11);
	alu_op : 1
		0,1,0,0,1,01,11,1,0

S_7: a ← alu_out
alu_a ← a
alu_b ← b
alu_out ← alu_a + alu_b
	ld_alu_out ; ld_a
	alu_select_a(00);
	alu_select_b(01);
	alu_op(0)
		1,0,0,0,1,00,01,0,0

S_8: data_result ← alu_out alu_a ← a alu_b ← b alu_out ← alu_a + alu_b	ld_r; alu_select_a(00) alu_select_b(10) alu_op = 0	0,0,0,0,0,00,10,0,1

3. State diagram

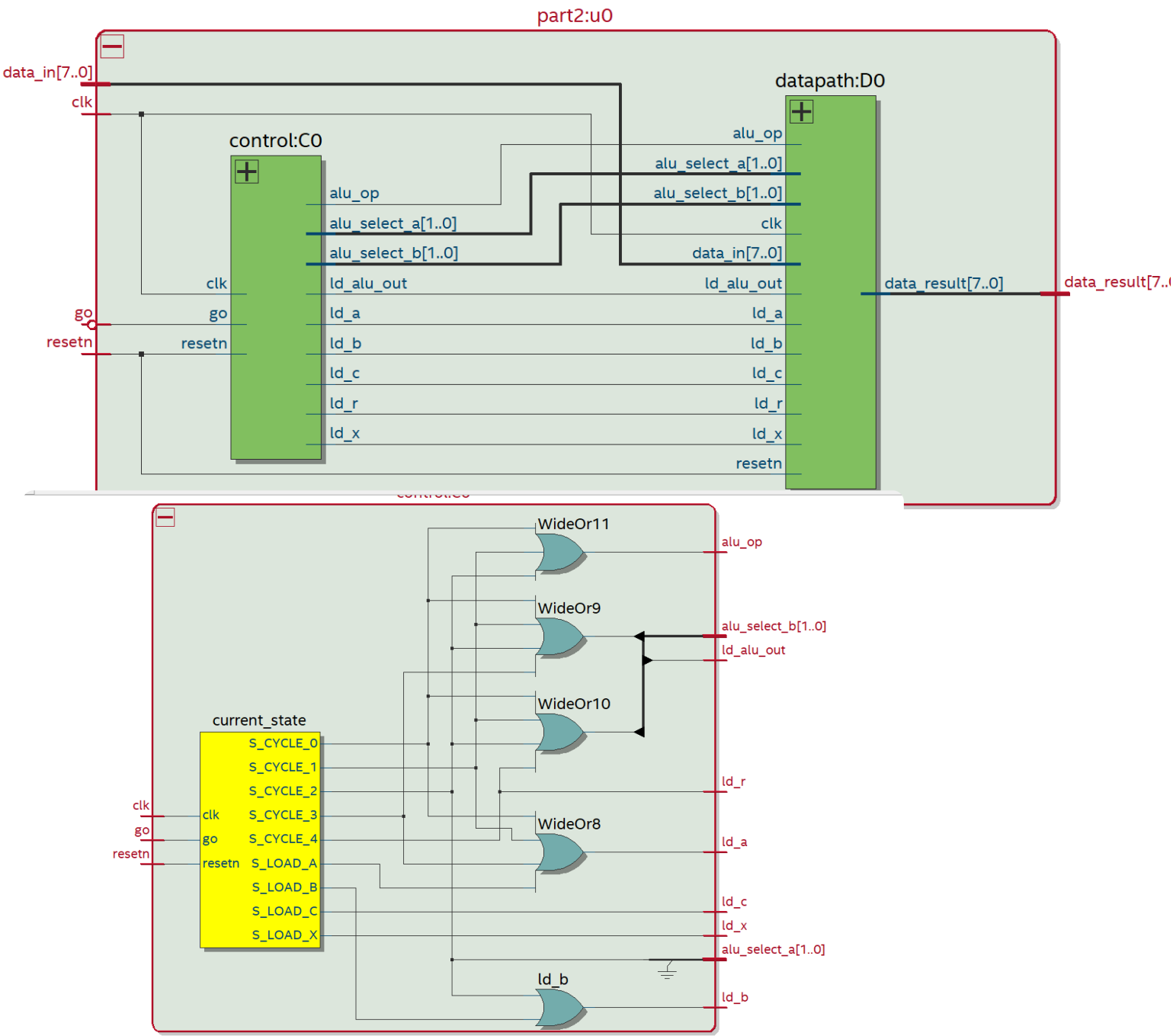


4. code

```
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; // we will be done our two operations, start over after
                S_CYCLE_2: next_state = S_CYCLE_3; // we will be done our two operations, start over after
                S_CYCLE_3: next_state = S_CYCLE_4; // we will be done our two operations, start over after
                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 <- A(A * x) * 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 A <- B * x
                ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into B
                alu_select_a = 2'b01; // Select register A
                alu_select_b = 2'b11; // Also select register A
                alu_op = 1'b1; // Do multiply operation
            end

            S_CYCLE_3: begin // Do A <- A(A * x * x) + B (B * 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'b01; // Also select register A
                alu_op = 1'b0; // Do multiply operation
            end

            S_CYCLE_4: begin // Do A <- A(A * x * x + B * x) + C
                ld_r = 1'b1; // store result in result register
                alu_select_a = 2'b00; // Select register A
                alu_select_b = 2'b10; // Also select register A
                alu_op = 1'b0; // Do multiply 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.



6. Simulation