Parallel and Reconfigurable VLSI Computing (5)

RTL Design Introduction

Hiroki Nakahara

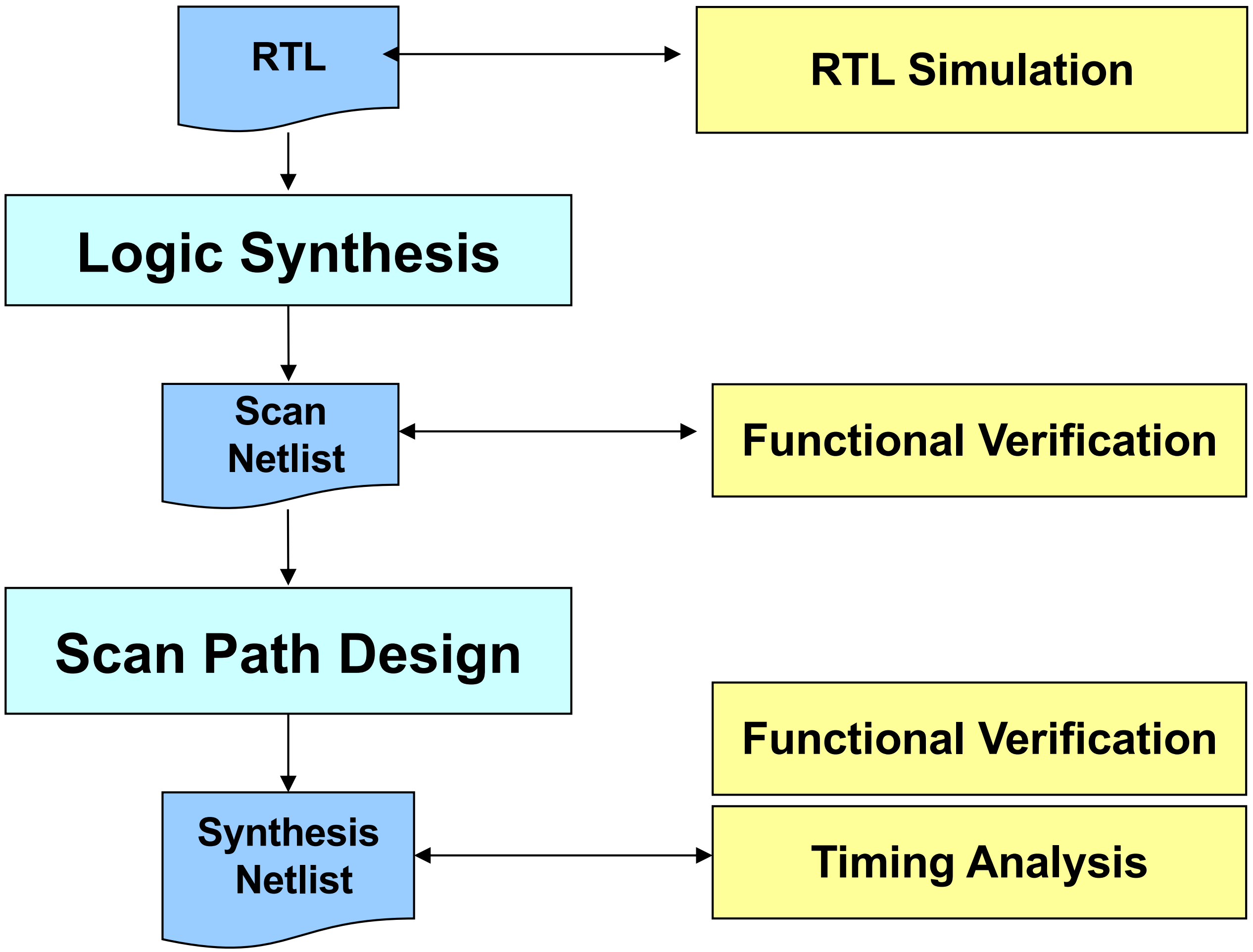
Tokyo Institute of Technology

Outline

- Verilog HDL (Hardware Description Language) Introduction
    - Language Rule
- Example of RTL Design
- RTL Design of Sequential Circuit
- FSM-based Sequential Logic Design
    - Ternary Counter
    - Security Key Machine

Verilog HDL (Hardware Description Language) Introduction

Typical Logic Design



Hardware Description Language (HDL)

- Describe a digital system, e.g., a computer and/or a component of a computer
- Several description level

Behavior Level	This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.
RTL Level (Structural Level)	Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".
Gate Level	Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (\`0', \`1', \`X', \`Z\`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). *Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.*

VHDL vs. Verilog HDL

- VHDL ("V" short for Very High Speed Integrated Circuits)
  - Designed for and sponsored by US Department of Defense
  - Syntax based on Ada programming language
- Verilog HDL
  - Introduced in 1985 by Gateway Design System Copr.
  - Syntax based on C-programming language

Register Transfer Level (RTL) Description

- Describe the data flow between registers in the circuit explicitly
- Circuit functions should not be written directly in the gate but should be written with a high degree of abstraction
- data:
  - Describe data transfer between registers as combinational circuit
- register:
  - From the functional description to the logic synthesis (To be described later)
- Combinational circuit:
  - A combinational circuit uses a function statement or the like to separate and describe a sequential circuit

Basic Design Strategy

- It is known that a circuit that reliably operates by logic synthesis can be obtained by using a synchronous circuit
- Do not write description of feedback in combination circuit
- Do not generate clock distribution circuit with logic circuit
- The description level is RTL (description separating register and combinational circuit)
- We describe the circuit structure after logic synthesis by image
- Simulation initialization by delay specification (#) or initial sentence is not used (no logic synthesis)
- Basically it does not describe fancy

Your First HDL Description

```
module hello_hw_top(
input [3:0]btn,
output [3:0]led
    );

    assign led[0] = btn[0] & btn[1];
    assign led[1] = btn[0] | btn[1];
    assign led[2] = ~btn[0];
    assign led[3] = btn[1] & (btn[2] | btn[3]);
endmodule
```

Port List and Declaration

- The portion representing the input / output interface of the module is called a port list
- The port list is defined by enclosing it with () after the module name
- Port declaration is done for each port specified in the port list
- In the port list, describe only the signal name and write the signal width in the port declaration

Module design by Instantiation

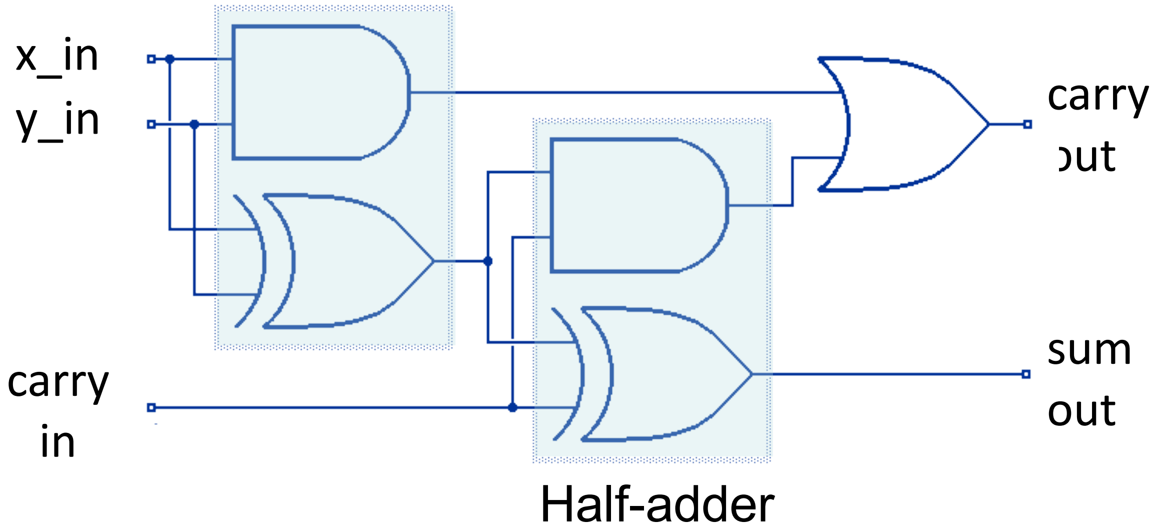
- Hierarchical design: Calls lower modules from upper module

  module_name Instance_name (terminal list);
- Instantiation: Instantiation, calling the defined module
- Instance: module called as part
- Connection with lower module
- Specify port name of subordinate module for each port in terminal list

  .***port_name***(***signal_name***)

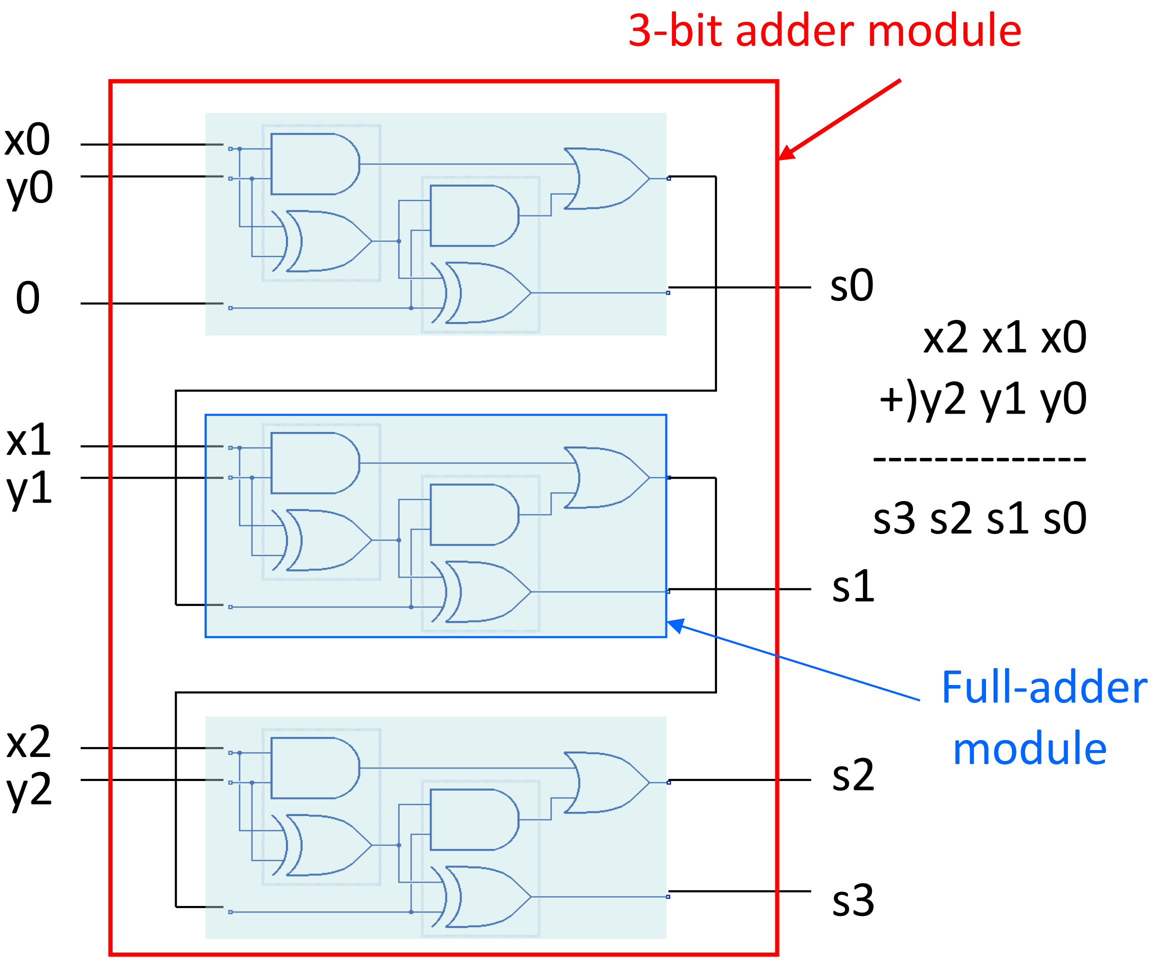
dot   port name for a sub-module   signal name for a top module

Example: Full-adder



x_in	y_in	carry in	sum out	carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Example of Hierarchy




```
full_adder fadr_inst_1(
    .x_in( SW[0]),
    .y_in( SW[3]),
    .carry_in( 0),
    .sum_out( LEDG[0]),
    .carry_out( w[0])
);

full_adder fadr_inst_2(
    .x_in( SW[1]),
    .y_in( SW[4]),
    .carry_in( w[0]),
    .sum_out( LEDG[1]),
    .carry_out( w[1])
);

full_adder fadr_inst_3(
    .x_in( SW[2]),
    .y_in( SW[5]),
    .carry_in( w[1]),
    .sum_out( LEDG[2]),
    .carry_out( LEDG[3])
);
```

Data Type and Number Representation

- Data type:
  - Net declaration: wire
  - Register declaration: reg
  - Integer declaration: integer
- Other type:
  - Parameter declaration: parameter
  - Definition: define
- Number representation:
  - Boolean value: 0, 1, x(don't care), z (high impedance)
  - Number: <bit width>'<radix><number>
    - Radix: b,B (Binary), d,D (Decimal), h,H (Hexadecimal)
  - e.g., 4'b0000, 8'h7F

Operator

Higher Priority

Operator	**Symbols**
Concatenate	{} {{}}
Unary	! ~ & ^ ^~ |
Arithmetic	+ - * / %
Logical shift	<< >>
Relational	< <= > >=
Equality	== !=
Binary bit-wise	& ^ ^~ |
Binary logical	&& ||
Conditional	?:

Lower

Example

- Arithmetic operator

```
wire [3:0] a, b, c, d, e;
assign c = 4'b1101 + 4'b0110; // constant
assign d = a + 4'h1; // incrementor
assign e = a + b; // 4-bit adder
```

- Reduction operator

```
wire a, b;
wire [3:0]c, d;
assign a = &c; // c[0] & c[1] & c[2] & c[3];
assign b = ^d; // d[0] ^ d[1] ^ d[2] ^ d[3];
```

- Concatenate operator

```
wire a, b, c, d;

wire [3:0] e;

wire [15:0] f;

assign e = {a,b,c,d}; // e[3]=a, e[2]=b, e[1]=c, e[0]=d;

assign f = {1'b1, {15{1'b0}}}; // f = 16'b1000_0000_0000_0000 or 16'h8000
```

Control Statement

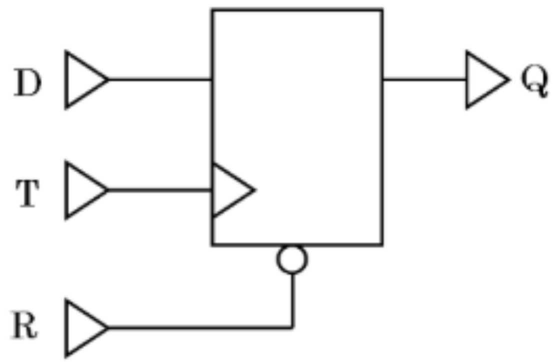
- Block statement: begin - end
  - Combine two or more sentences into one sentence
  - Execute sentences sandwiched between ***begin*** and ***end*** sequentially
  - ***begin*** and ***end*** can be nested
- Control statement:

  if - else -, if - else if - else -

  case - endcase (casex, casez)
  - Alternatively, function or assign with "?:" operator can be used

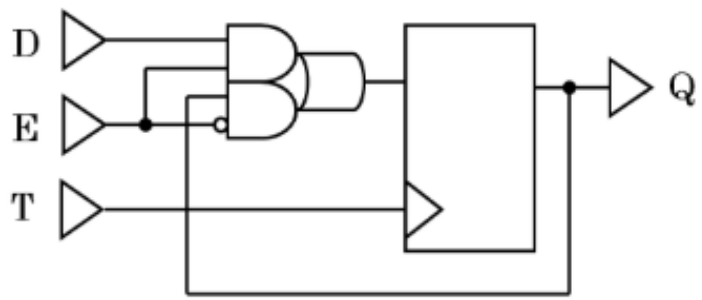
Examples of RTL Design

Flip-Flop (FF) with asynchronized reset



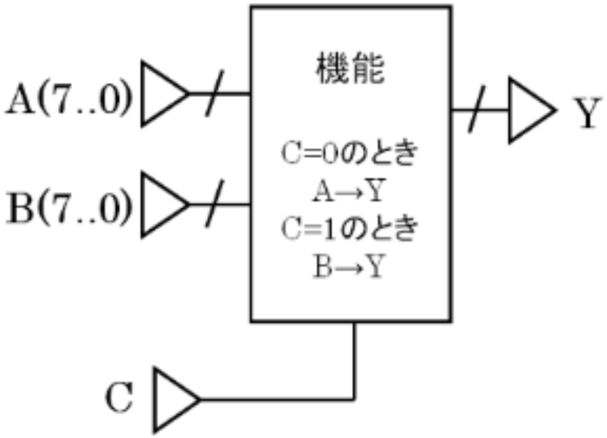

```
module  rdff ( Q, D, R, T );
output Q;
input D, R, T;
reg Q;
always @( posedge T or
negedge R )
begin
   if ( !R)
         Q <= 1'b0;
   else
         Q <= D;
end
endmodule
```

FF with an enable-signal



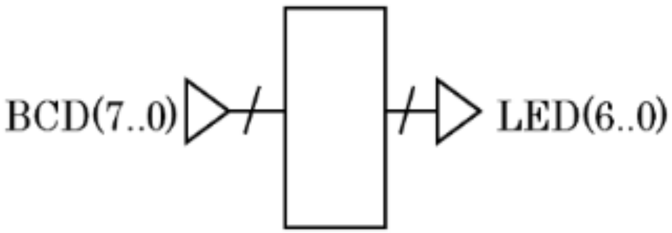

```
module  edff ( Q, D, E, T );
output Q;
input D, E, T;
reg Q;
always @( posedge T )
begin
    if ( E )
        Q <= D;
    else
        Q <= Q;
end
endmodule
```

Multiplexer



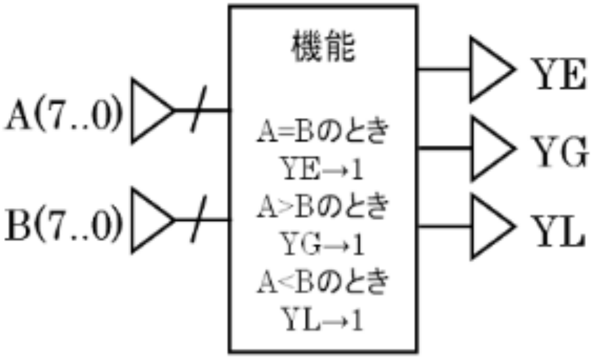

```
module  mux ( Y, A, B, C );
output  [7:0] Y;
input [7:0] A, B, C;
  assign Y = func_mux ( A, B, C );
  function [7:0] func_mux;
  input [7:0] A, B;
  input C;
    if ( C == 1'b0 )
       func_mux = A;
    else if ( C == 1'b1 )
       func_mux = B;
    else
       func_mux = 8'bxxxxxxxx;
  endfunction
endmodule
```

Decoder



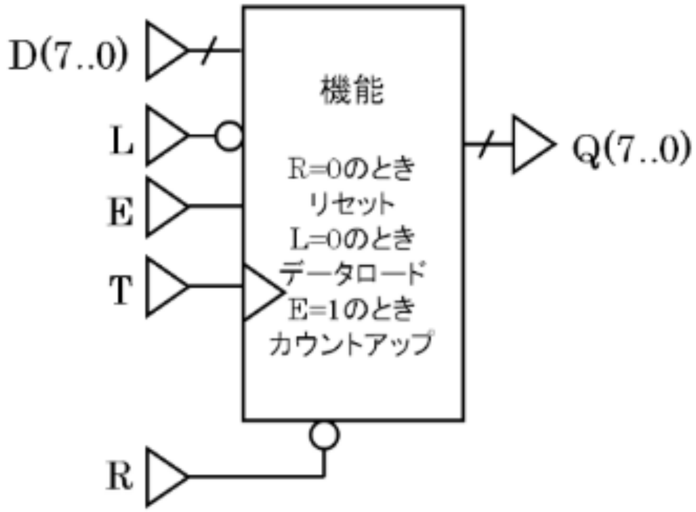

```
module  dec ( LED, BCD );
output [6:0] LED;
input  [3:0] BCD;
  assign LED = func_dec ( BCD );
  function [6:0] func_dec;
  input [3:0] BCD;
    case ( BCD )
       4'b0000 : func_dec = 7'b1111110;
       4'b0001 : func_dec = 7'b1100000;
       4'b0010 : func_dec = 7'b1011011;
       4'b0011 : func_dec = 7'b1110011;
       4'b0100 : func_dec = 7'b1100101;
       4'b0101 : func_dec = 7'b0110111;
       4'b0110 : func_dec = 7'b1011111;
       4'b0111 : func_dec = 7'b1110000;
       4'b1000 : func_dec = 7'b1111111;
       4'b1001 : func_dec = 7'b1110111;
       default  : func_dec = 7'bxxxxxxx;
    endcase
  endfunction
endmodule
```

Comparator



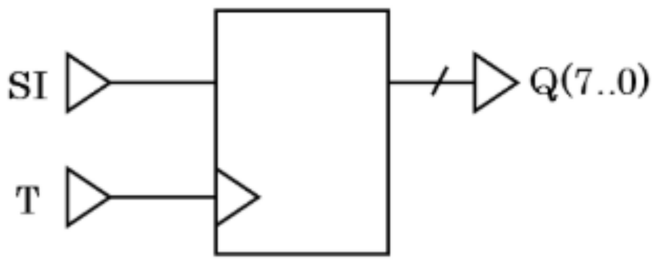

```
module  cmp ( YE, YG, YL, A, B );
output YE, YG, YL;
input  [7:0] A, B;
  assign { YE, YG, YL } = func_cmp ( A, B );
  function  [2:0] func_cmp;
  input [7:0] A, B;
  begin
    if ( A == B )
       func_cmp[2] = 1'b1;
    else
       func_cmp[2] = 1'b0;
    if ( A > B )
       func_cmp[1] = 1'b1;
    else
       func_cmp[1] = 1'b0;
    if ( A < B )
       func_cmp[0] = 1'b1;
    else
       func_cmp[0] = 1'b0;
  endfunction
endmodule
```

Counter

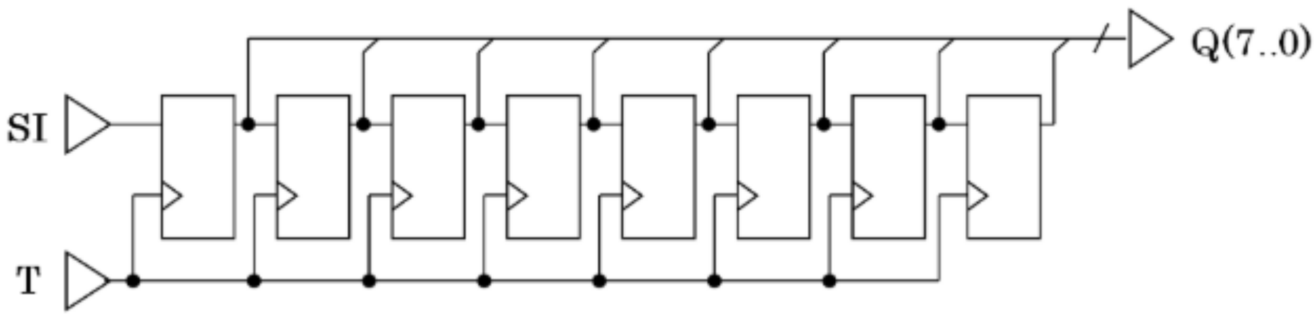



```
module  count ( Q, D, E, L, R, T );
output [7:0] Q;
input [7:0] D;
input E, L, R, T;
reg [7:0] Q;
  always @ ( posedge T or negedge R)
  begin
     if ( !R )
       Q <= 8'h00;
     else if ( !L )
       Q <= D;
     else if ( !E )
       ;
     else if ( Q == 8'hff )
       Q <= 8'h00;
     else
       Q <= Q + 8'h01;
  end
endmodule
```

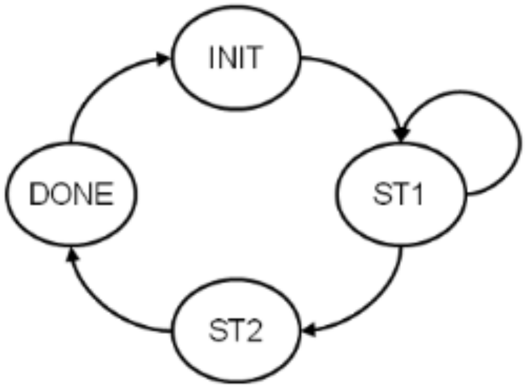
Shift Register

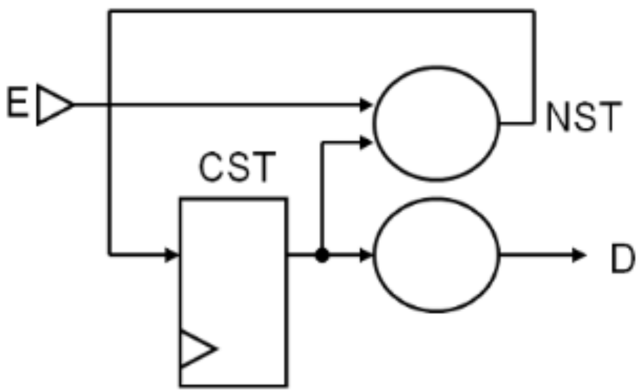



```
module  sht ( Q, SI, T );
output [7:0] Q;
input SI, T;
reg [7:0] Q;
  always @( posedge T )
  begin
     Q <= { Q[6:0], SI };
  end
endmodule
```



Finite State Machine (FSM)





ムーア型ステートマシン

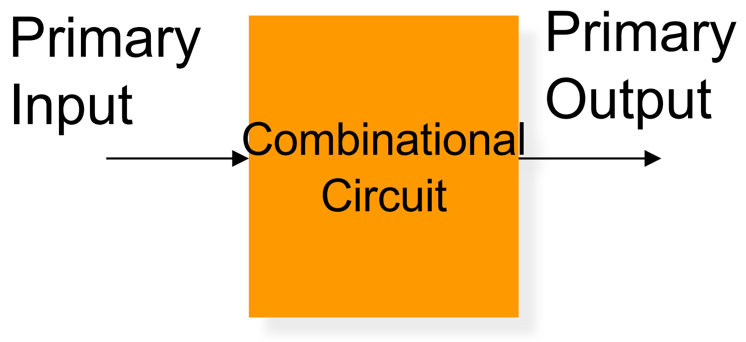
```
module  count ( D, E, RSTL, CLK );
output D;
input E, RSTL, CLK;
`define INIT 2'b00
`define ST1 2'b01
`define ST2 2'b10
`define DONE 2'b11
reg [1:0] CST, NST;
wire D;
assign D = ( CST == `DONE );
always @ ( posedge CLK or negedge RSTL )
    if( !RSTL )  CST <= `INIT;
    else             CST <= NST;
always @ ( E or CST )
  case( CST )
      `INIT : NST = `ST1;
      `ST1 : if ( E == 1'b1 ) NST = `ST2;
                else NST = `ST1;
      `ST2 : NST = `DONE;
      `DONE : NST = `INIT;
      default : NST = `INIT;
  endcase
endmodule
```

RTL Design of Sequential Circuit

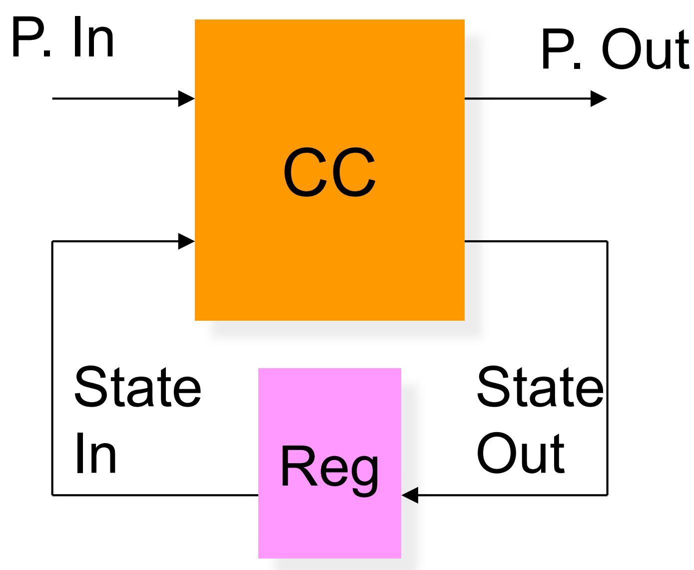
Sequential Circuit

- The output of a combinational circuit (CC) depends on a present input
- That of a sequential circuit depends on both a present and past ones

- Register stores a past input
- It is described by "always" statement
  - With clock and reset variables

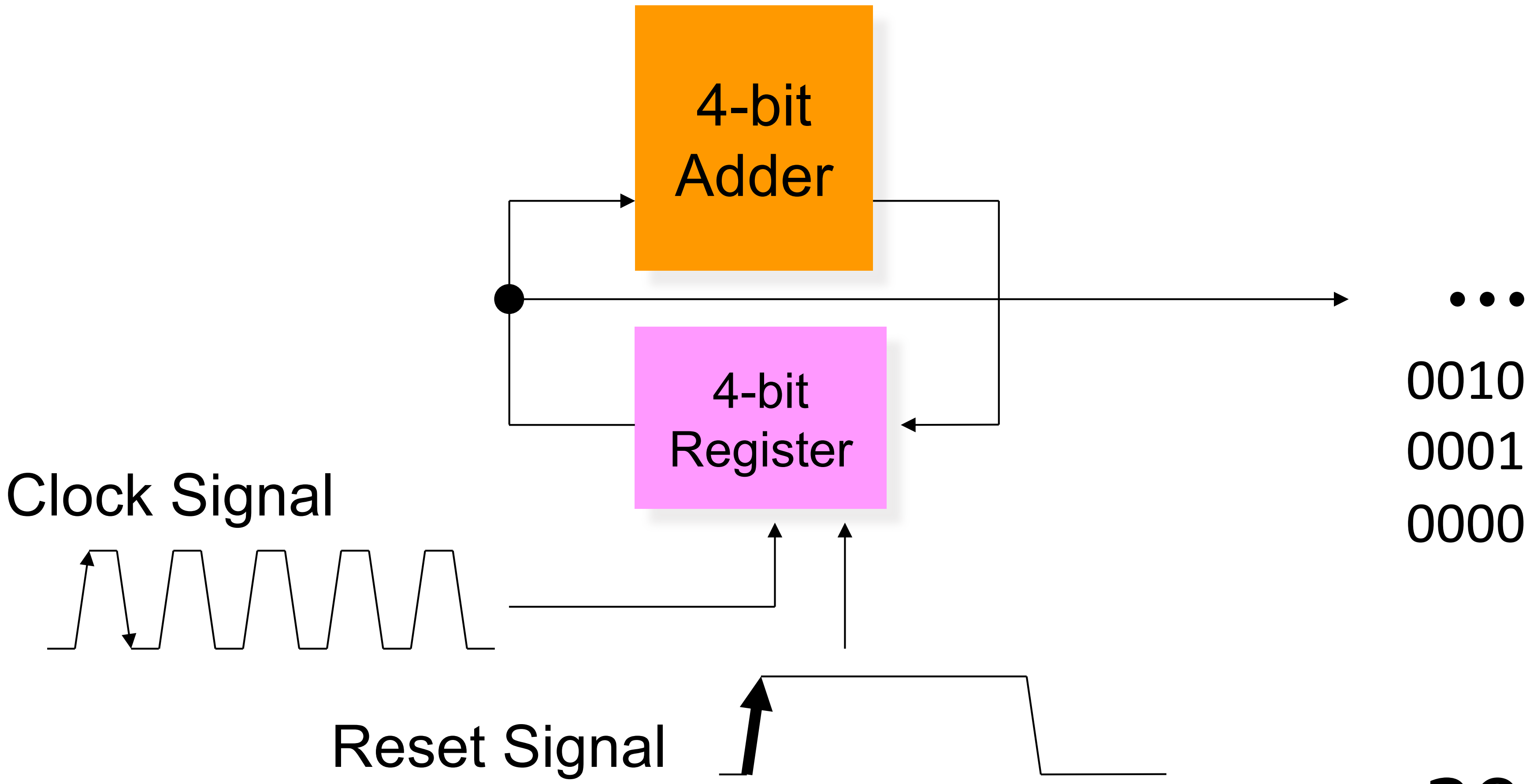


Combinational Circuit



Sequential Circuit

Case Study: 4-bit Counter



Create Project

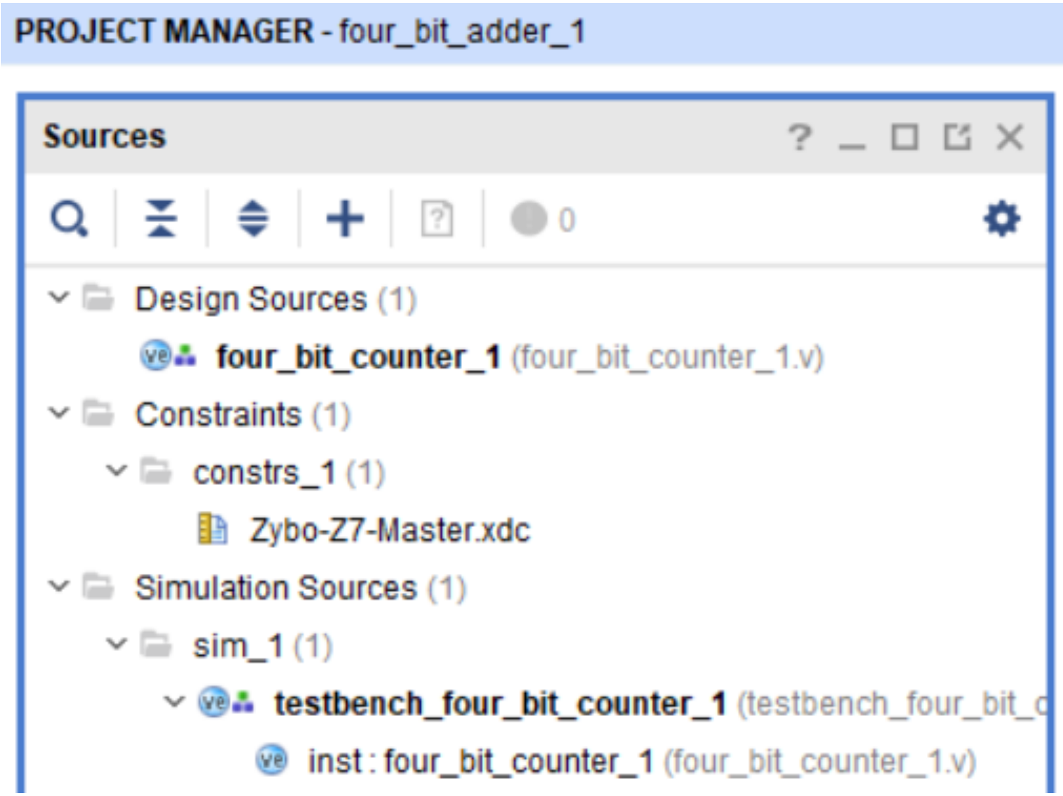
Project location: C:¥FPGA¥lect5_1¥four_bit_counter_1

Target FPGA: Zybo-Z7-10 or (Z7-20)

Design Sources: four_bit_counter_1.v

Constraints: Zybo-Z7-Master.xdc

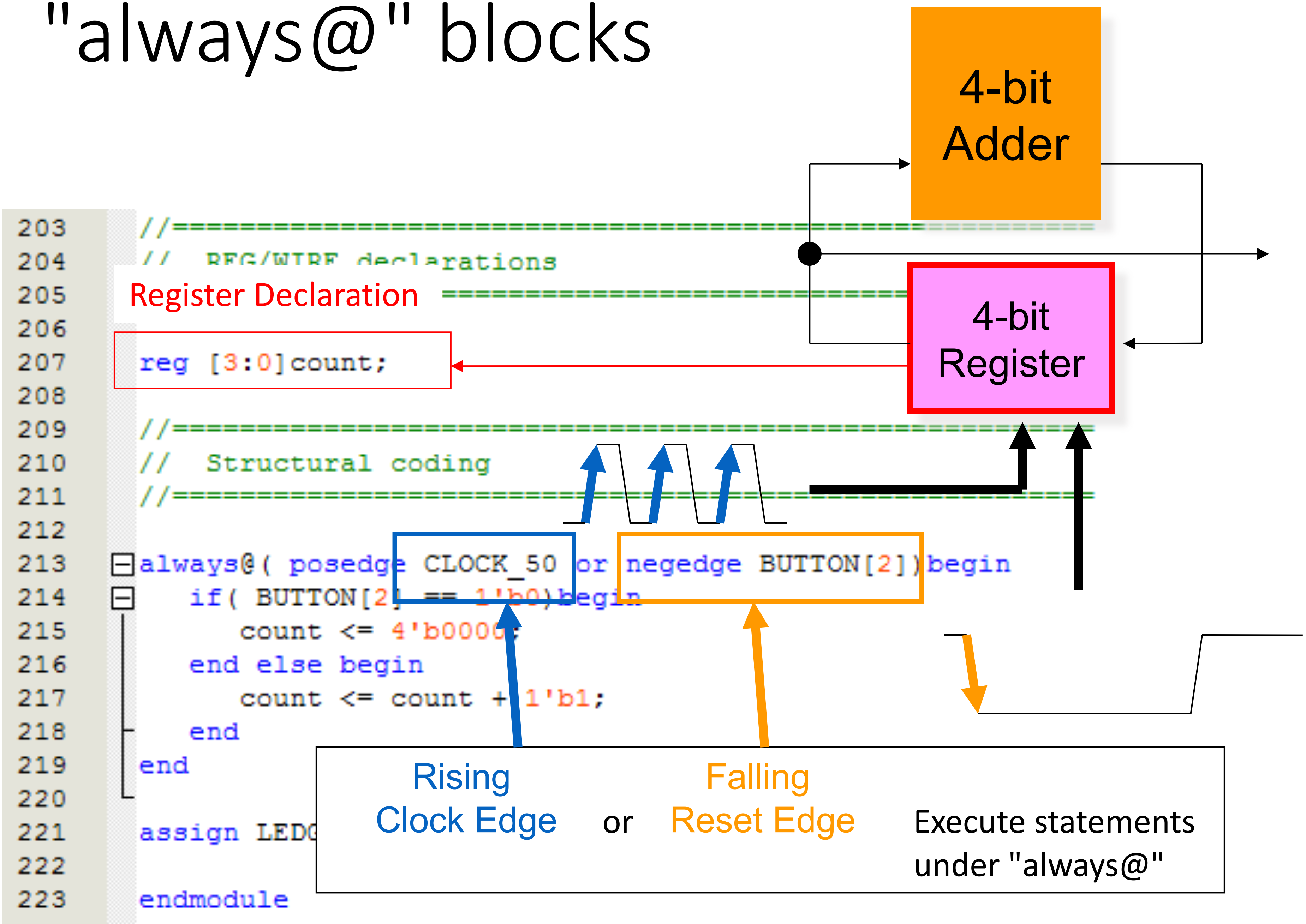
Simulation Sources: testbench_four_bit_counter_1.v



Example: 4-bit Counter

```
//=======================================================
//  REG/WIRE declarations
//=======================================================

reg [3:0]count;

//=======================================================
//  Structural coding
//=======================================================

always@( posedge CLOCK_50 or negedge BUTTON[2])begin
    if( BUTTON[2] == 1'b0)begin
        count <= 4'b0000;
    end else begin
        count <= count + 1'b1;
    end
end

assign LEDG[3:0] = count[3:0];

endmodule
```

"always@" blocks



Cont'd

always@() should be followed by a template

```
always@( event for clock or reset signals)
begin
        if( clock signal )begin
                (initialize registers);
        end else begin
                (update of registers);
        end
end
```

```
//==============================
//  REG/WIRE declarations
//==============================

reg [3:0]count;

//==============================
//  Structural coding
//=====================================================================

always@( posedge CLOCK_50 or negedge BUTTON[2])begin
    if( BUTTON[2] == 1'b0)begin
        count <= 4'b0000;
    end else begin
        count <= count + 1'b1;
    end
end

assign LEDG[3:0] = count[3:0];

endmodule
```

Use a non-blocking assignment "<="
for updating a register

Left value must be a register variable

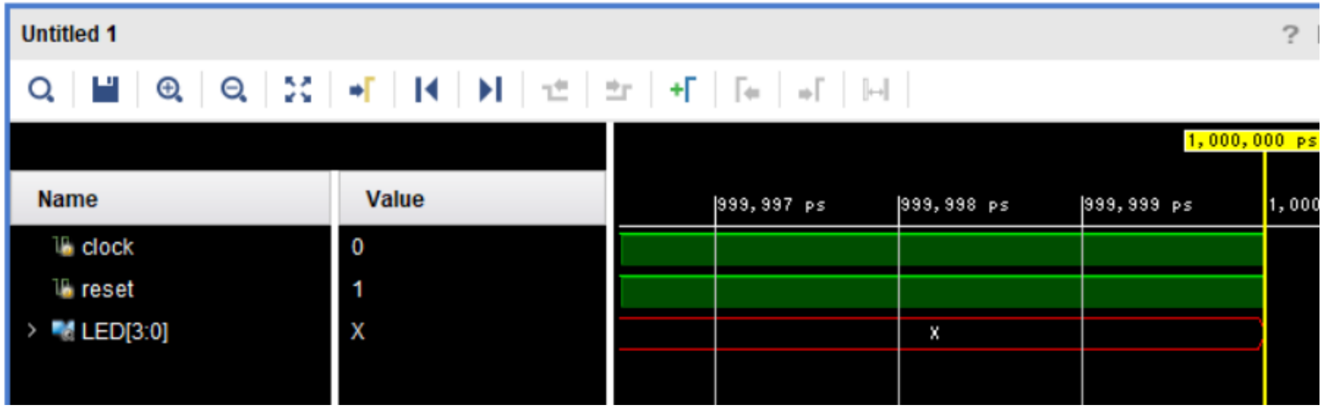
Verilog-HDL Code

```
module four_bit_counter_1(
    input clock,
    input reset,
    output [3:0]LED
    );

    reg [3:0]count;

    always@( posedge clock or negedge reset)begin
        if( reset == 1'b0)begin
            count <= 4'b0000;
        end else begin
            count <= count + 1'b1;
        end
    end

    assign LED = count;
endmodule
```

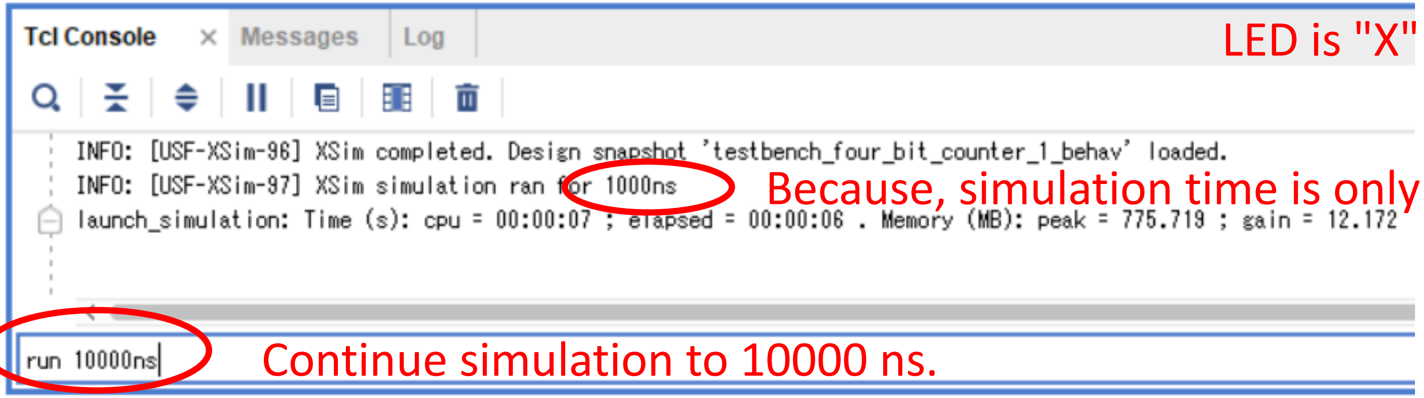
Simulation Code

```
module testbench_four_bit_counter_1(
    );

    reg clock, reset;
    wire [3:0]LED;

    initial begin
    clock = 0;
    reset = 1;

    #5000
    reset = 0;
    #3000
    reset = 1;
    end

    initial begin
        forever begin
            #500 clock = ~clock;
        end
    end

    four_bit_counter_1 inst(
        .clock(clock),
        .reset(reset),
        .LED(LED)
    );
endmodule
```

Behavior Simulation



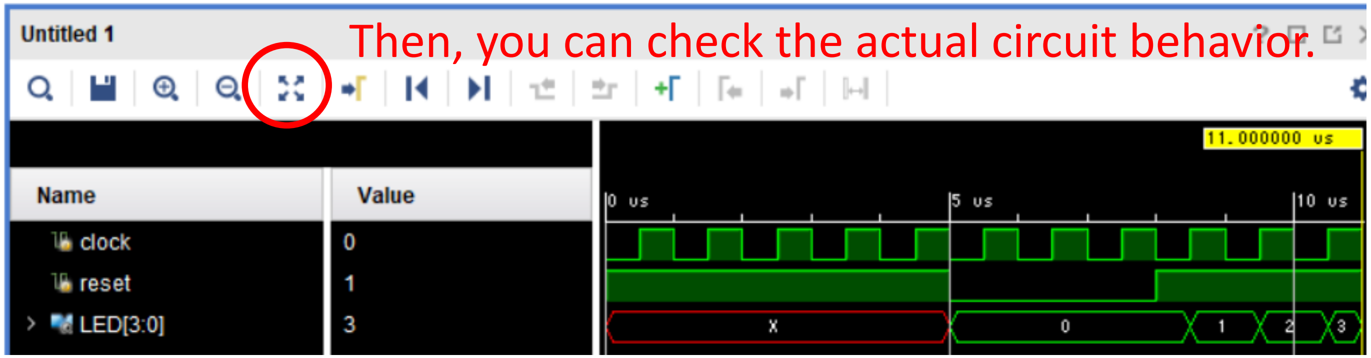
LED is "X" (unknown)??



Because, simulation time is only 1000nsec.

Continue simulation to 10000 ns.

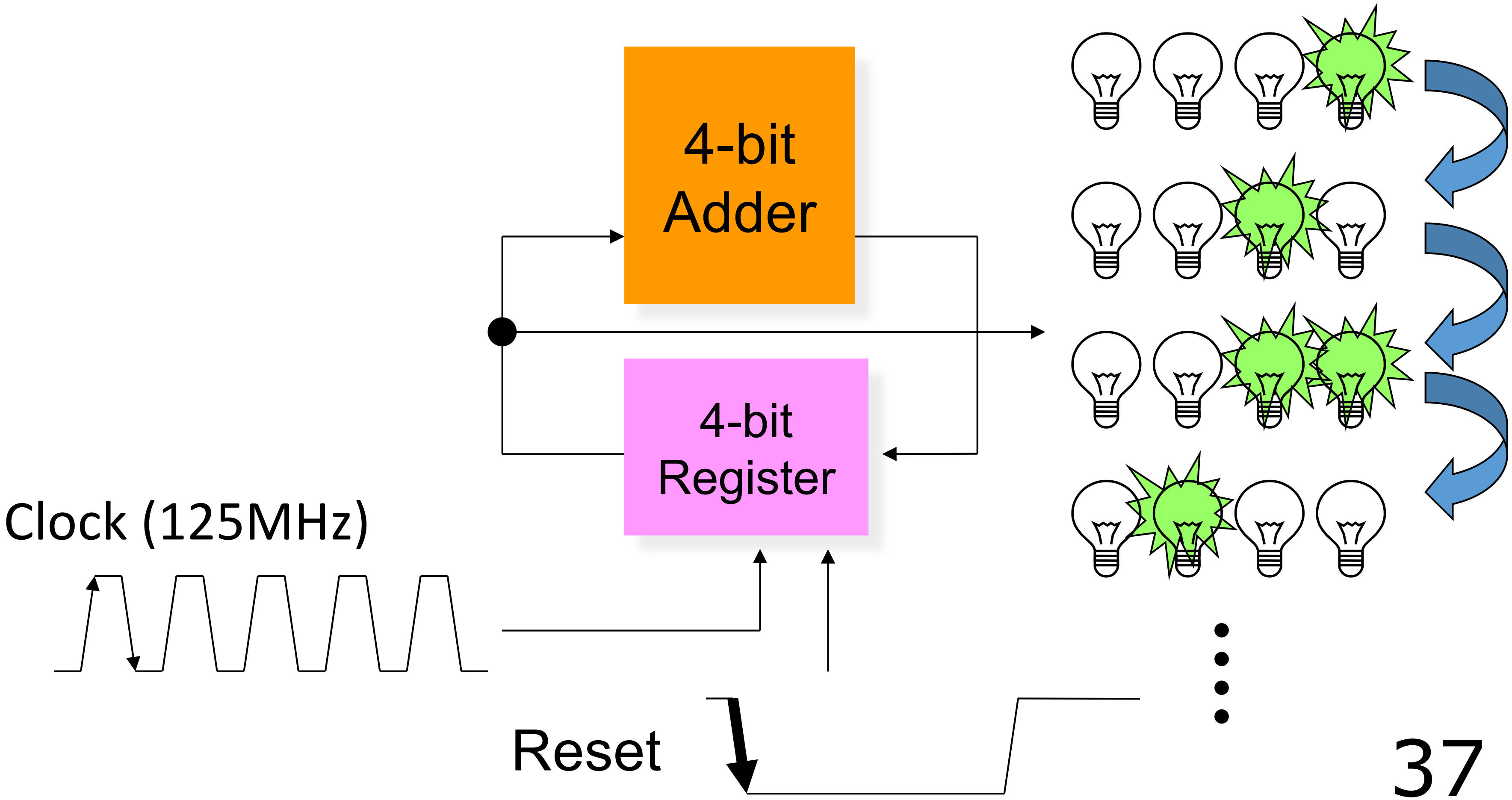
Then, you can check the actual circuit behavior.



Practically Design

- Input clock (125MHz) is too fast
  - Nobody can see a LED changing!

Blink (on-off) period for 125 MHz is ____[nsec]



Hierarchy Description for "Watchable" 4-bit Counter

- Comment out "sysclk", "btn[0]", "led"s at the XDC file
- Add a "top_four_bit_counter_1"

  to "Design Sources"
- Then, write the following HDL code:

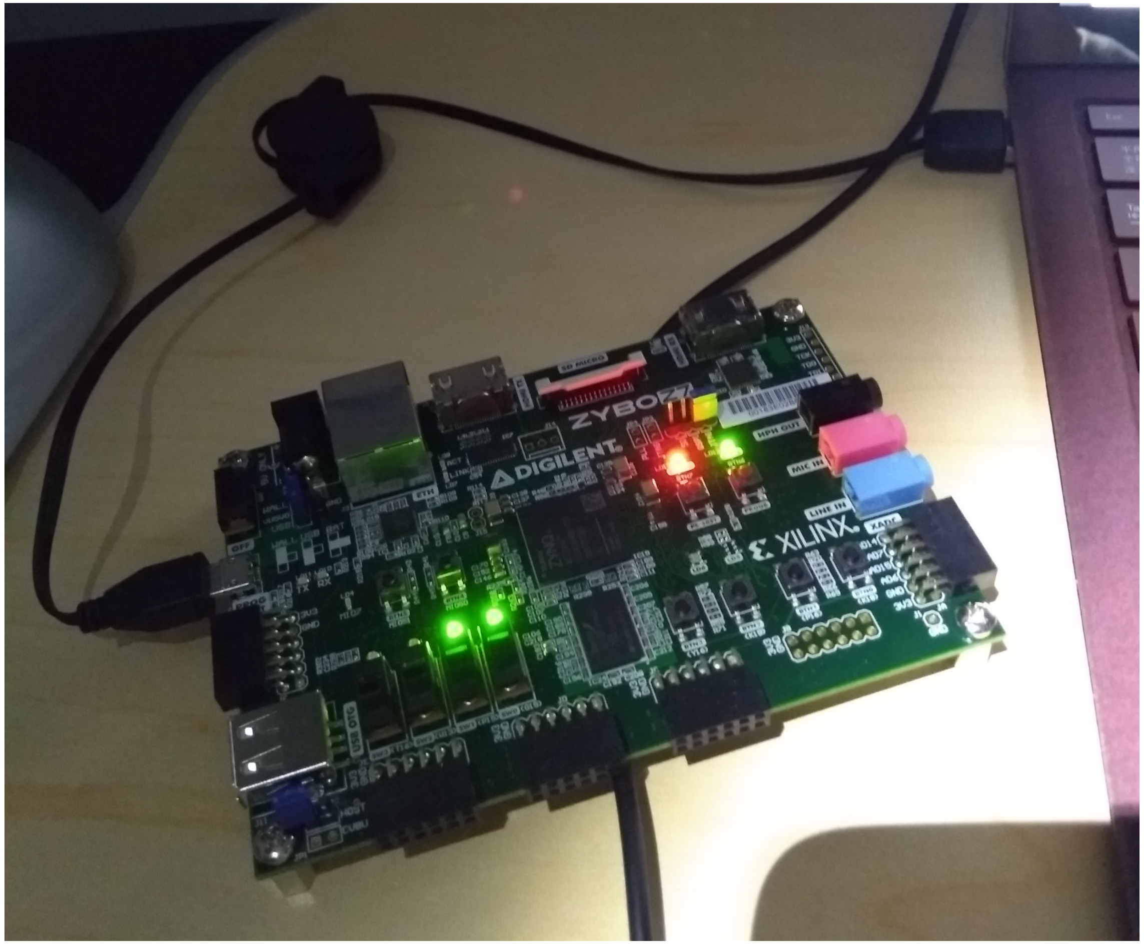
```
module top_four_bit_counter_1(
    input sysclk,
    input [0:0]btn,
    output [3:0]led
    );

    reg [26:0]count; // 1sec ~ 8ns(125MHz) * 125,000,000

    always@( posedge sysclk or posedge btn[0])begin
        if( btn[0] == 1'b1) begin
            count <= 27'b0;
        end else begin
            count <= count + 1'b1;
        end
    end

    four_bit_counter_1 inst_four_bit_counter(
        .clock( count[26]),
        .reset( ~btn[0]),
        .LED( led)
        );

endmodule
```

1Hz Generator

Rising Edge to Falling Edge →
(to Negative logic)

Implementation on a Zybo-Z7

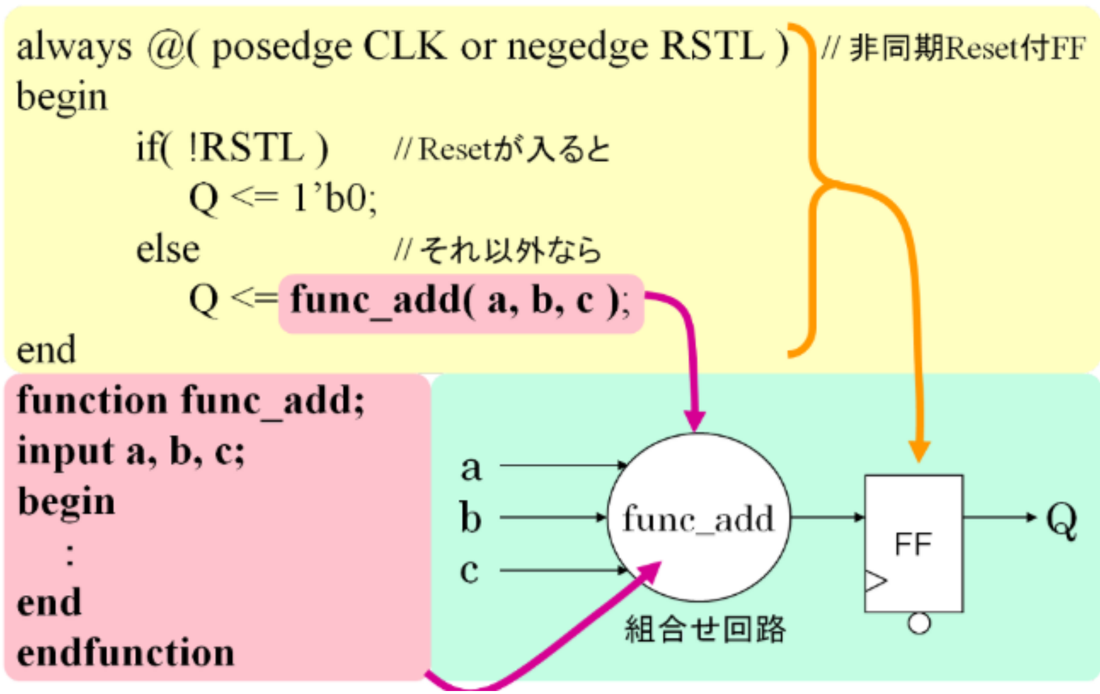
- Click "Generate Bitstream", then configure your Zybo



FSM-based RTL Design

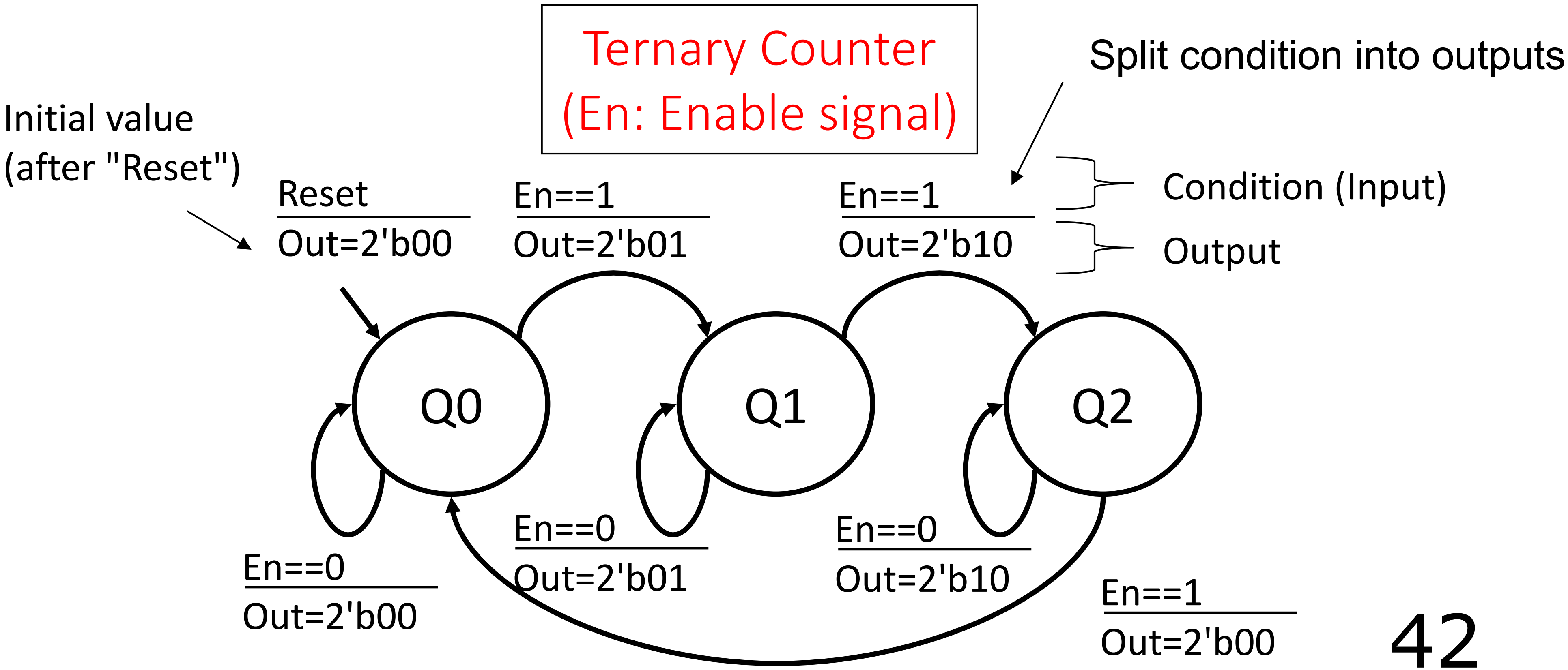
Basic FSM-based RTL Design

- Combination of the always sentence that generates the register and the function statement that describes the combinational circuit as shown below
- It can reliably perform logic synthesis (Non simulation description)



Finite State Machine (FSM)

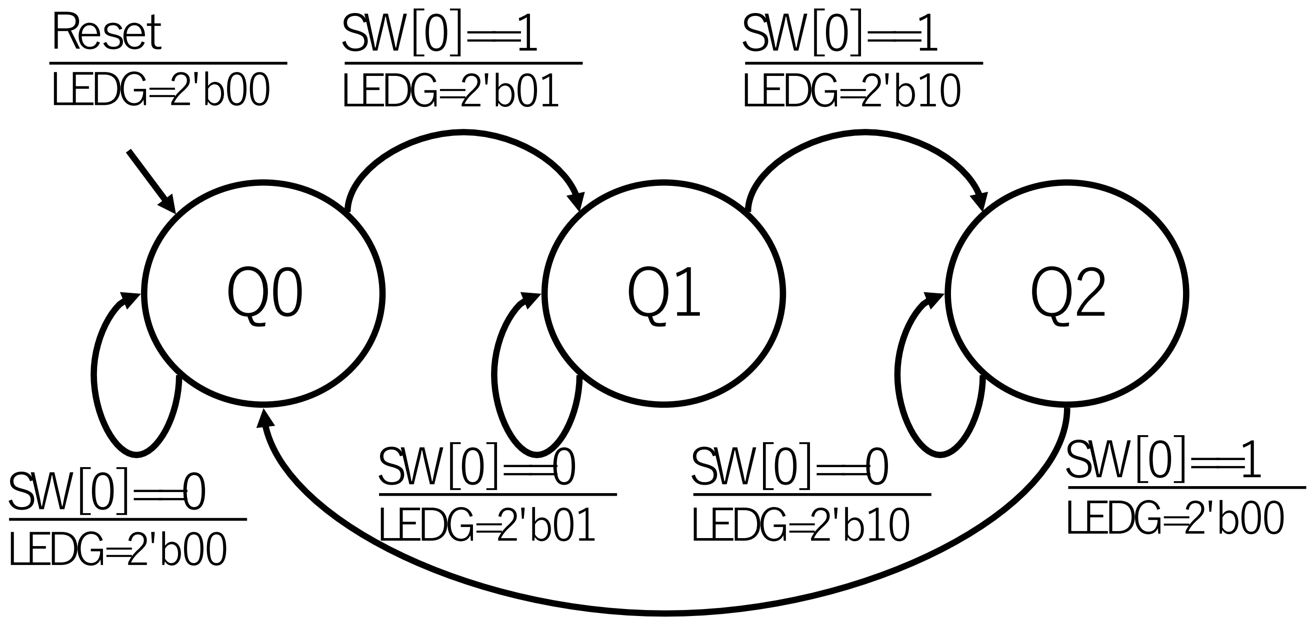
- It is a figure which shows an operation of a sequential circuit
  - Circle denotes a statement
  - Edge denotes a transition state
  - Edge values consist of condition and output



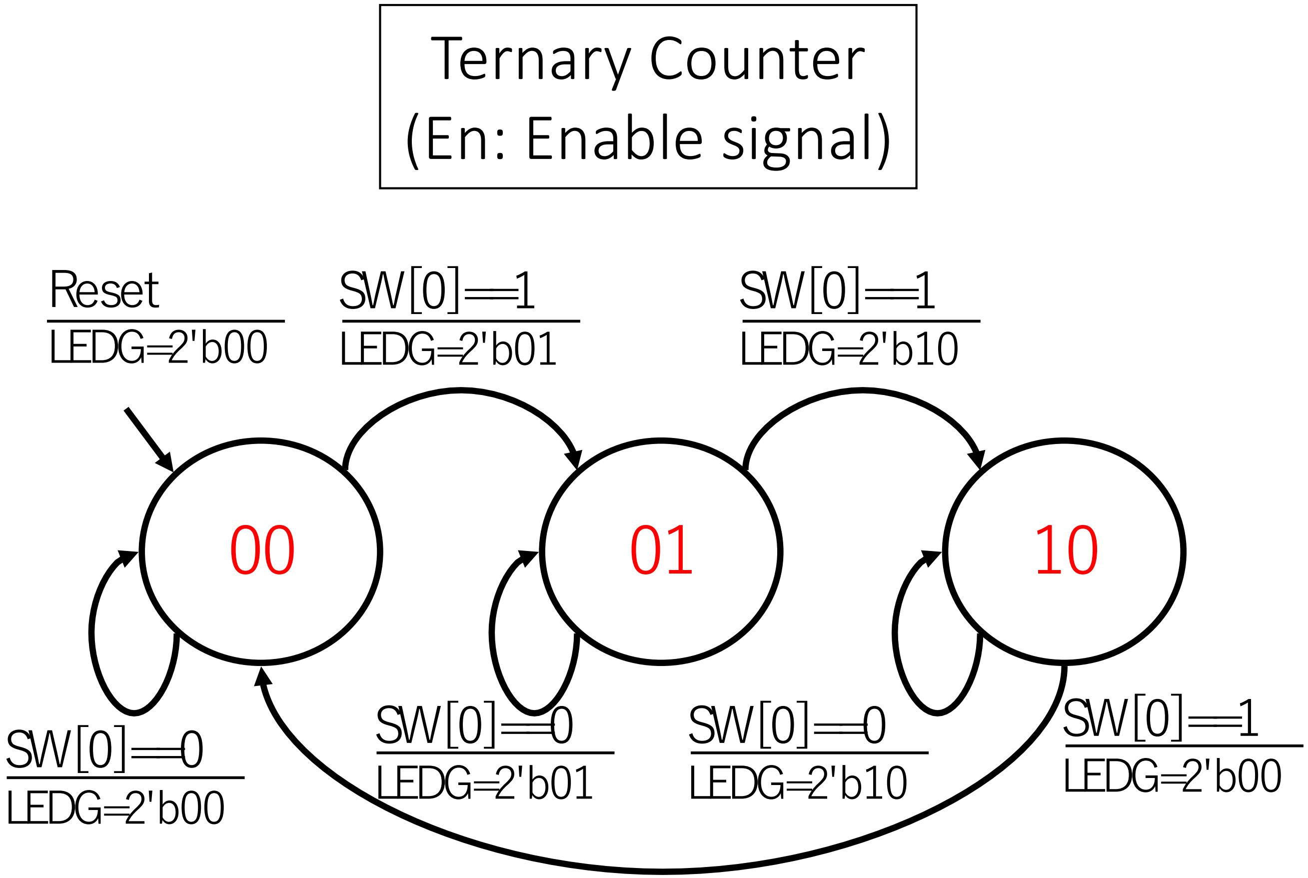
Rewrite Signals for the Zybo-Z7 board

- Reset -> BUTTON[0]: Negative value (Push: 0, Release: 1)
- Out -> LEDG[1:0]: Positive (Turn-on: 1, off: 0)
- En -> SW[0]: Positive (Slide to up: 1, to down:0)

Ternary Counter
(En: Enable signal)

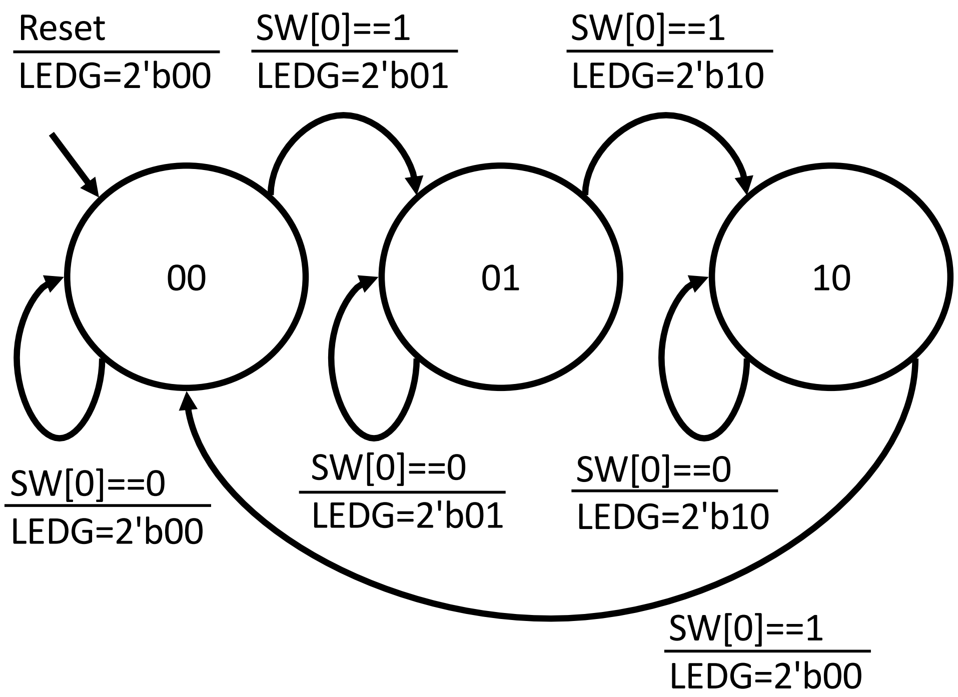


Assign Binary Code



Declare Port-list

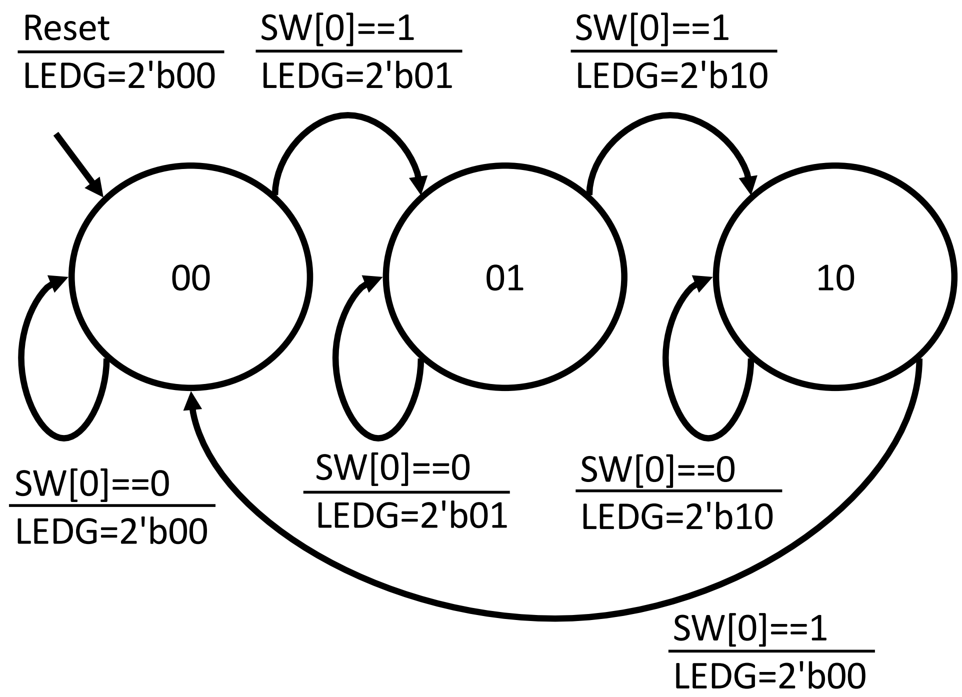
- Describe module
  - Note that you must include clock and reset signals



module threebit_counter(clk, reset, En, out);	endmodule

Declare in/out

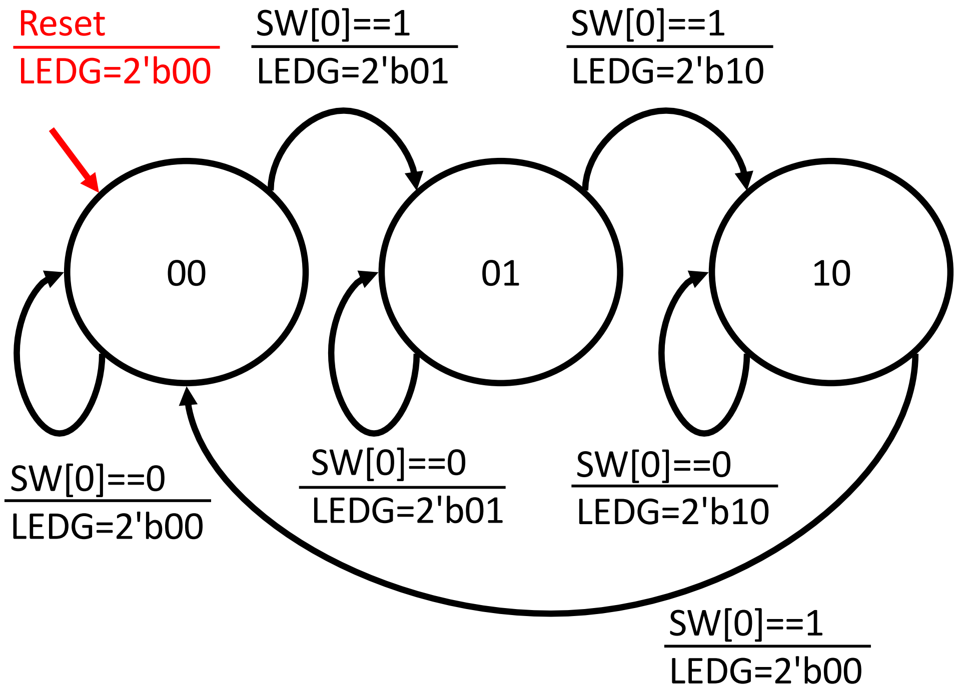
- Also, output and state variables are assigned by "reg" declaration




```
module threebit_counter(
clk, reset, En, out );
input clk, reset, En;
output [1:0]out;
reg [1:0]out;
reg [1:0]state;

endmodule
```

Set Initial Values

- After "reset"

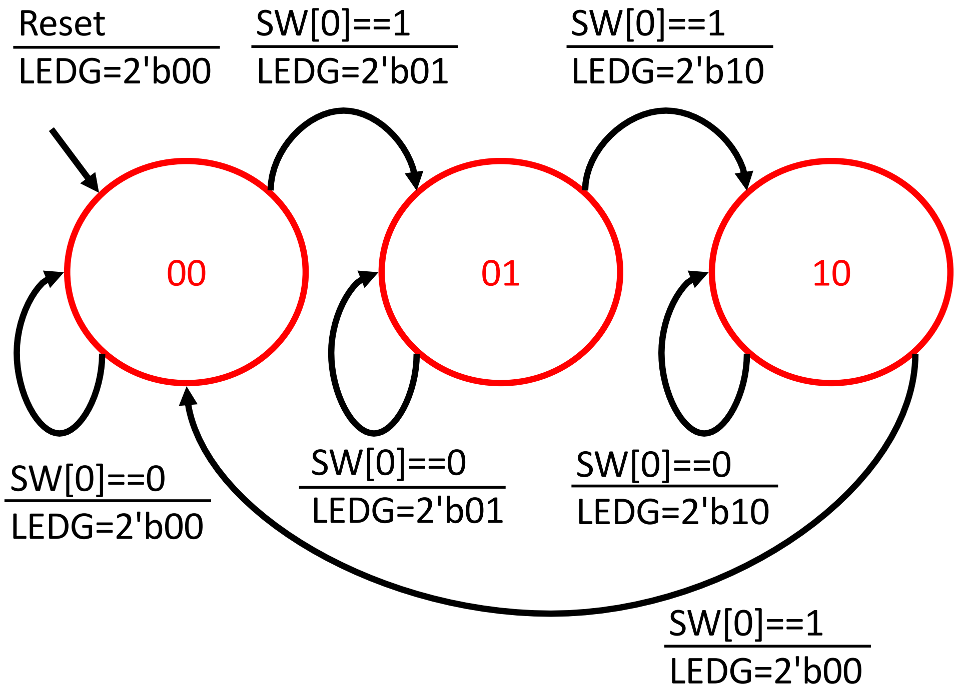



```
module threebit_counter(
clk, reset, En, out );
input clk, reset, En;
output [1:0]out;
reg [1:0]out;
reg [1:0]state;

always@( posedge clk or
posedge reset) begin
   if( reset == 1'b1)begin
      out <= 2'b00;
      state <= 2'b00;
   end else begin


   end
end
```

```
endmodule
```

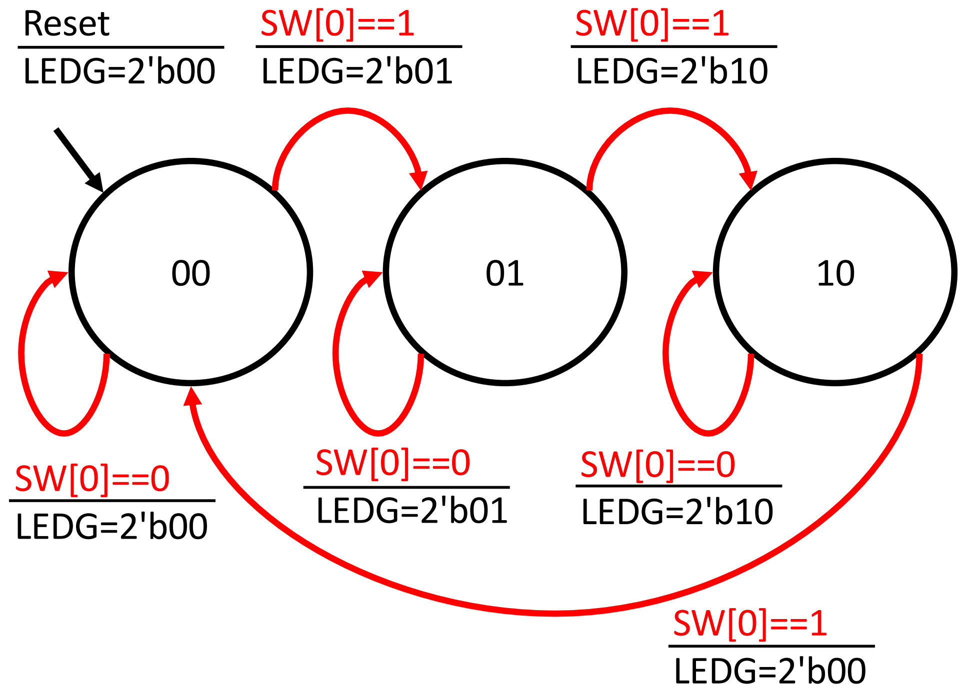
Define FSM by "case" sentence

- Use "default" for undefined state



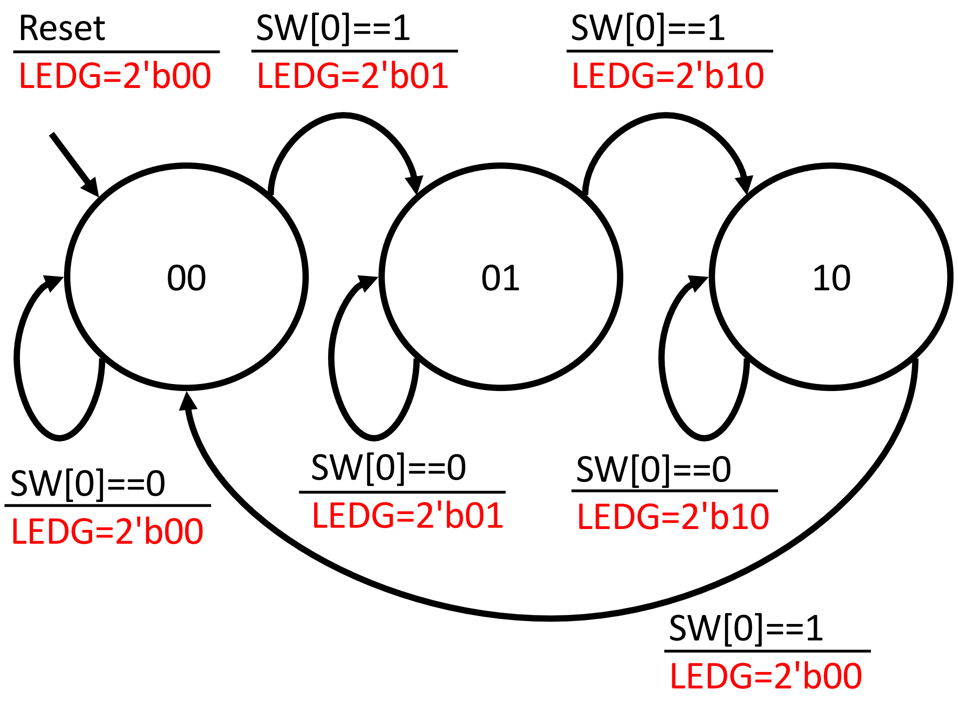

```
module threebit_counter(
clk, reset, En, out );
input clk, reset, En;
output [1:0]out;
reg [1:0]out;
reg [1:0]state;

always@( posedge clk or
posedge reset) begin
   if( reset == 1'b1)begin
      out <= 2'b00;
      state <= 2'b00;
   end else begin
   case( state)
   2'b00: begin

   end
   2'b01: begin

   end
   2'b10: begin

   end
   default: begin

   end
   endcase
   end
end
endmodule
```

Describe state transition by "if-else" statement




```
module threebit_counter(
clk, reset, En, out );
input clk, reset, En;
output [1:0]out;
reg [1:0]out;
reg [1:0]state;

always@( posedge clk or
posedge reset) begin
   if( reset == 1'b1)begin
      out <= 2'b00;
      state <= 2'b00;
   end else begin
   case( state)
   2'b00: begin
      if( En == 1'b1)begin
         state <= 2'b01;

      end else begin
         state <= state;

      end
   end
   2'b01: begin
      if( En == 1'b1)begin
         state <= 2'b10;

      end else begin
         state <= state;

      end
   end
   2'b10: begin
      if( En == 1'b1)begin
         state <= 2'b00;

      end else begin
         state <= state;

      end
   end
   default: begin
      state <= 2'b00;

   end
   endcase
   end
end
endmodule
```

Set Output Value




```
module threebit_counter(
clk, reset, En, out );
input clk, reset, En;
output [1:0]out;
reg [1:0]out;
reg [1:0]state;

always@( posedge clk or
posedge reset) begin
   if( reset == 1'b1)begin
      out <= 2'b00;
      state <= 2'b00;
   end else begin
   case( state)
   2'b00: begin
      if( En == 1'b1)begin
         state <= 2'b01;
         out   <= 2'b01;
      end else begin
         state <= state;
         out  <= 2'b00;
      end
   end
    2'b01: begin
      if( En == 1'b1)begin
         state <= 2'b10;
         out   <= 2'b10;
      end else begin
         state <= state;
         out   <= 2'b01;
      end
   end
   2'b10: begin
      if( En == 1'b1)begin
         state <= 2'b00;
         out   <= 2'b00;
      end else begin
         state <= state;
         out   <= 2'b10;
      end
   end
   default: begin
      state <= 2'b00;
      out    <= 2'b00;
   end
   endcase
   end
end
endmodule
```

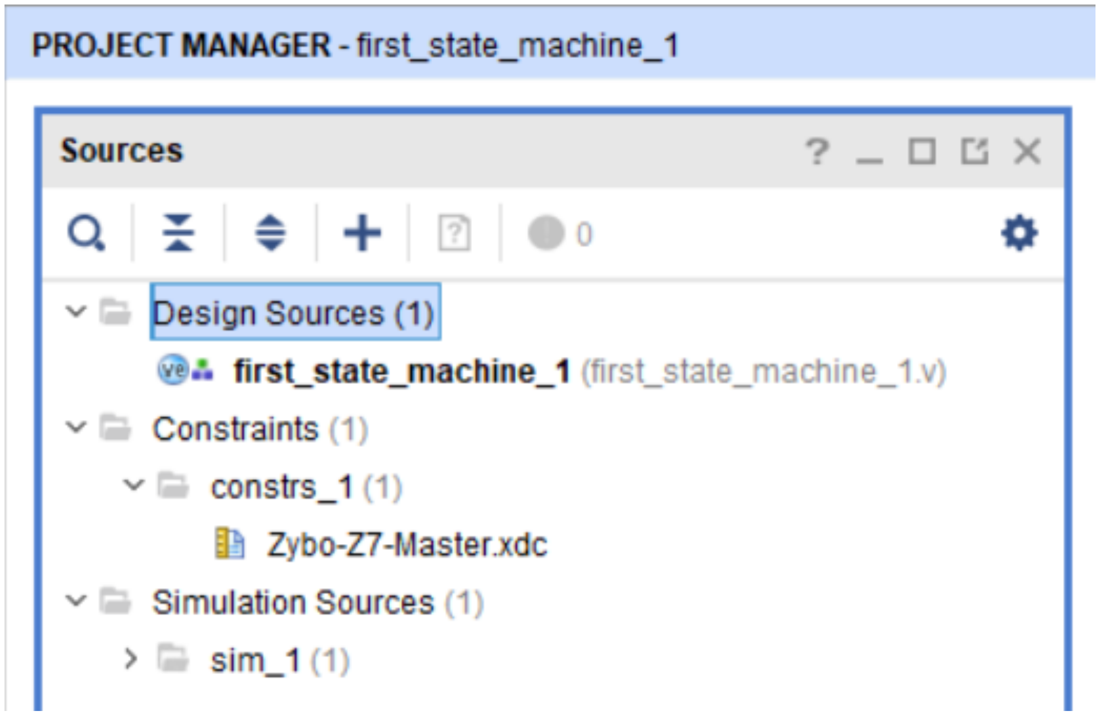
Create Project

Project location: C:¥FPGA¥lect5_2¥first_state_machine_1

Target FPGA: Zybo-Z7-10 or (Z7-20)

Design Sources: first_state_machine_1.v

Constraints: Zybo-Z7-Master.xdc



Verilog-HDL code

```
module first_state_machine_1(
    input clk, reset, En,
    output [1:0]out
    );

    reg [1:0]out, state;

    always@( posedge clk or posedge reset)begin
        if( reset == 1'b1)begin
            out <= 2'b00;
            state <= 2'b00;
        end else begin
            case( state)
            2'b00: begin
                if( En == 1'b1)begin
                    state <= 2'b01;
                    out   <= 2'b01;
                end else begin
                    state <= state;
                    out   <= 2'b00;
                end
            end
            2'b01: begin
                if( En == 1'b1)begin
                    state <= 2'b10;
                    out   <= 2'b10;
                end else begin
                    state <= state;
                    out   <= 2'b01;
                end
            end
            2'b10: begin
                if( En == 1'b1)begin
                    state <= 2'b00;
                    out   <= 2'b00;
                end else begin
                    state <= state;
                    out   <= 2'b10;
                end
            end
            default: begin
                state <= 2'b00;
                out <= 2'b00;
            end
            endcase
        end
    end
endmodule
```

Hierarchy Description for "Watchable" 4-bit Counter

- Comment out "sysclk", "btn[0]", "led"s at the XDC file
- Add a "top_1st_FSM_1.v"

  to "Design Sources"
- Then, write the following HDL code:

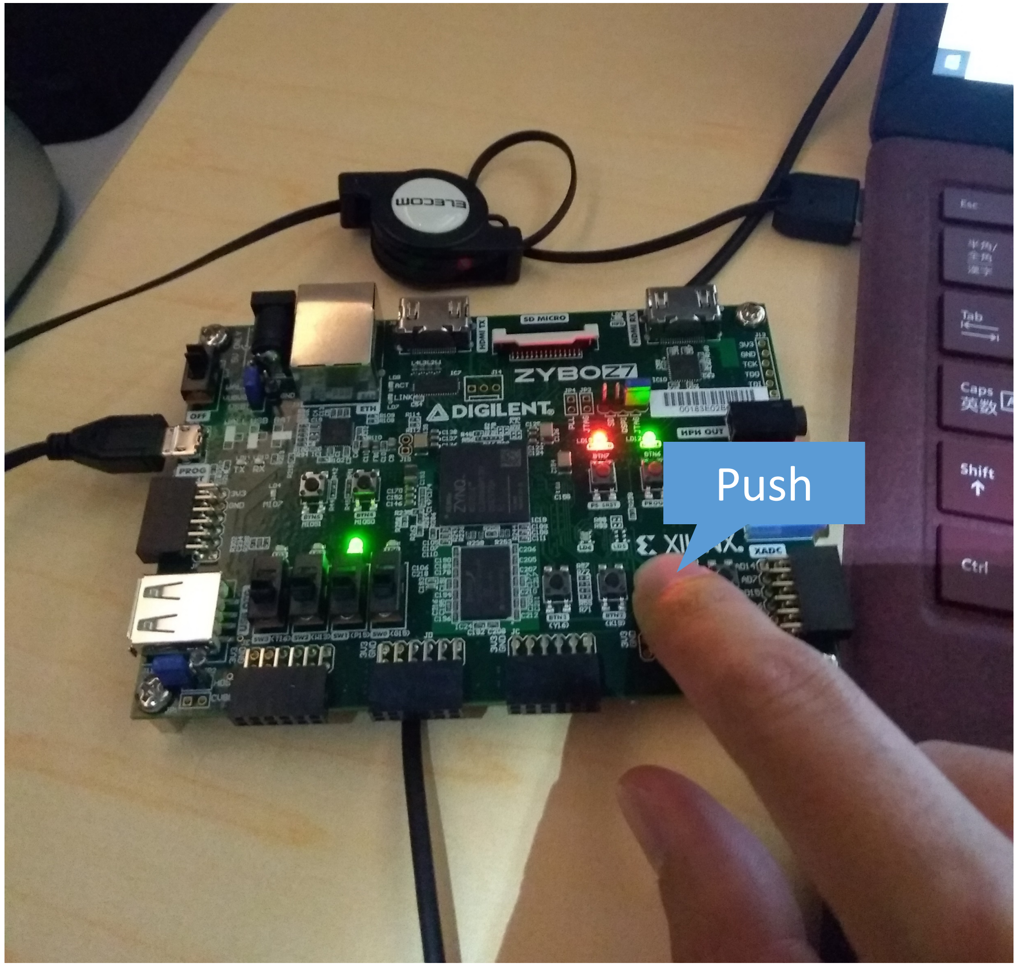
```
module top_four_bit_counter_1(
    input sysclk,
    input [0:0]btn,
    output [3:0]led
    );

    reg [26:0]count; // 1sec ~ 8ns(125MHz) * 125,000,000

    always@( posedge sysclk or posedge btn[0])begin
        if( btn[0] == 1'b1) begin
            count <= 27'b0;
        end else begin
            count <= count + 1'b1;
        end
    end

    first_state_machine_1 init_ter_counter_1(
    .clk( count[26]),
    .reset( btn[0]),
    .En( 1'b1),
    .out( led)
    );

endmodule
```

1Hz Generator

constant '1' → (always on)

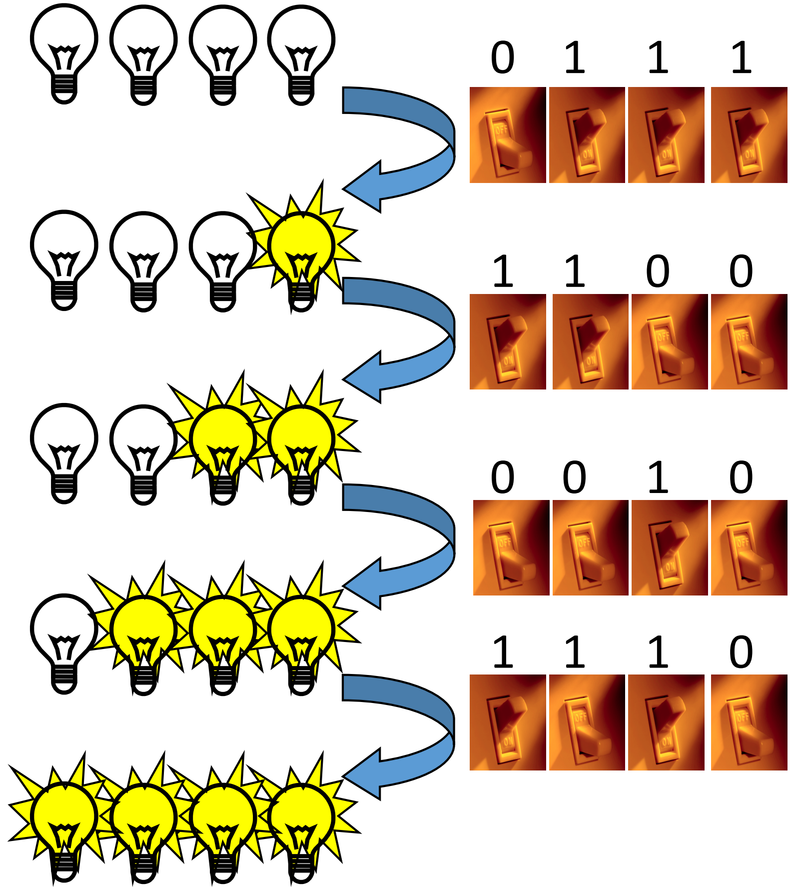
Implementation on a Zybo-Z7

- Push "btn[0]" to run your counter

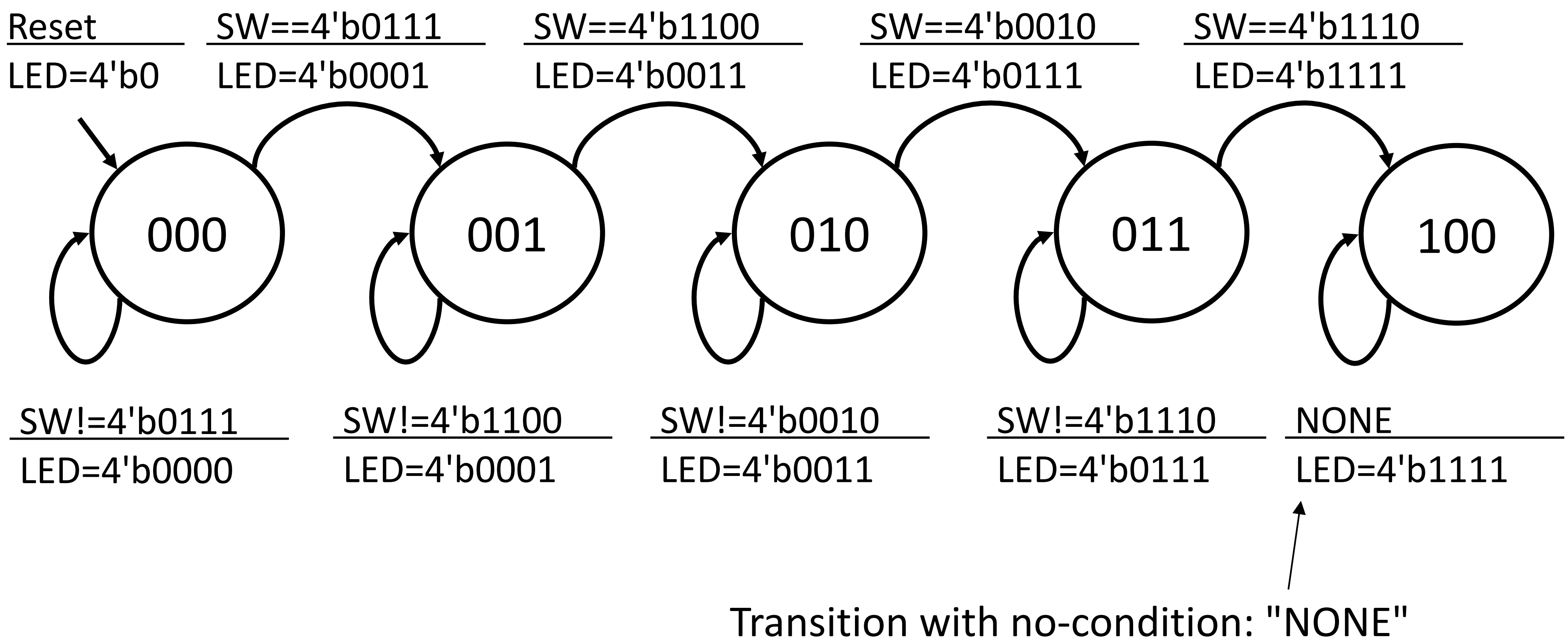


Security Key Machine

- Input: 4-bit slide switch, Output: 4 LEDs
- Specification: Correct input at 4 times, then turn-on LEDs
  - Initial (After reset)：LED
  - 1st input: only an LED
  - 2nd: two LEDs
  - 3rd: three LEDs
  - 4th: all LEDs



Write FSM from Specification



Create Project

Project location: C:¥FPGA¥lect5_3¥encrpt_1

Target FPGA: Zybo-Z7-10 or (Z7-20)

Design Sources: encrpt_1.v

Constraints: Zybo-Z7-Master.xdc

Verilog-HDL code

```
module encrpt_1(
    input [3:0]SW,
    input clock,
    input reset_p,
    output [3:0]LED
    );

    reg [2:0]state;
    reg [3:0]LED;

    always@( posedge clock or posedge reset_p)begin
        if( reset_p == 1'b1)begin
            state <= 3'b0;
            LED <= 4'b0;
        end else begin
            case( state)
            3'b000: begin
                if( SW == 4'b0110)begin
                    state <= 3'b001;
                    LED   <= 4'b0001;
                end else begin
                    // note, state <= state can be omitted
                    LED   <= 4'b0000;
                end
            end
            3'b001: begin
                if( SW == 4'b1100)begin
                    state <= 3'b010;
                    LED   <= 4'b0011;
                end else begin
                    LED   <= 4'b0001;
                end
            end
            3'b010: begin
                if( SW == 4'b0010)begin
                    state <= 3'b011;
                    LED   <= 4'b0111;
                end else begin
                    LED   <= 4'b0011;
                end
            end
            3'b011: begin
                if( SW == 4'b1110)begin
                    state <= 3'b100;
                    LED   <= 4'b1111;
                end else begin
                    LED   <= 4'b0111;
                end
            end
            3'b100: begin
                LED <= 4'b1111;
            end
            default: begin
                state <= 3'b000;
                LED   <= 4'b0000;
            end
            endcase
        end
    end
endmodule
```

Hierarchy Description for a Security Key Machine

- Comment out "sysclk", "btn[0]", "sw"s (slide switches), "led"s at the XDC file
- Add a "top_encrpt_1.v"

  to "Design Sources"
- Then, write the following HDL code:

  (It is unnecessary to a 1Hz generator!)
- Finally, run your machine

```
module top_encrpt_1(
        input [3:0]sw,
        input sysclk,
        input [0:0]btn,
        output [3:0]led
    );

    encrpt_1 inst_encrpt_1(
        .SW( sw),
        .clock( sysclk),
        .reset_p( btn[0]),
        .LED( led)
    );
endmodule
```

Exercise

- (Mandatory) Add a slide switch to the ternary counter and design a circuit that satisfies the following specifications
  - ON: Up counter (00 → 01 → 10 → 00)
  - OFF: Down counter (10 → 01 → 00 → 10)
- (Optional) For the encryption key input machine, modify it so that input is accepted after all switches are turned off
- Send a PDF file including both your Verilog-HDL source code and a photograph of the running situation via OCW-i

Deadline is 5th, July, 2019 PM13:20

(At the beginning of the lecture)