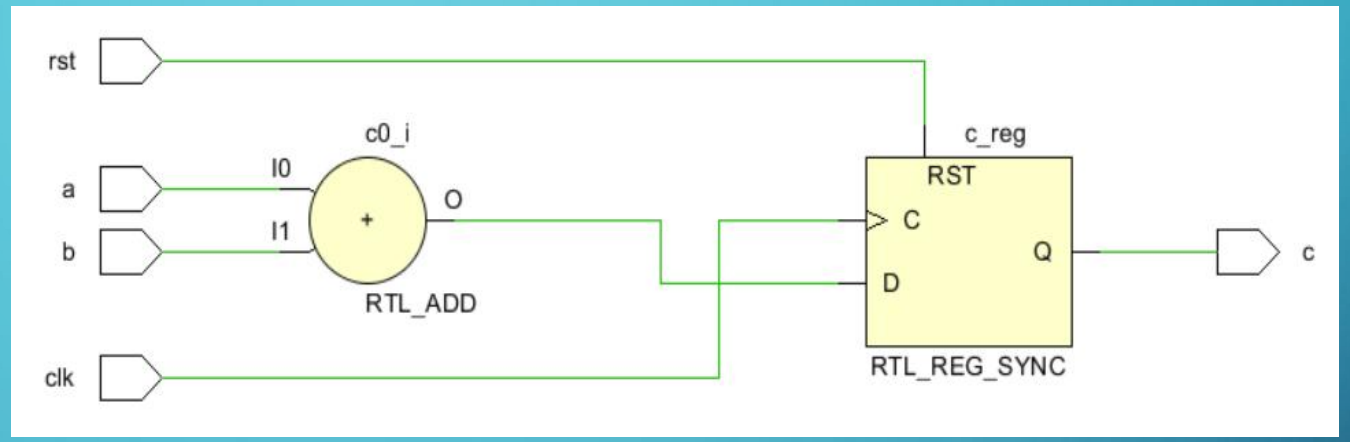
DIGITAL DESIGN

LAB12 FSM MOORE MEALY

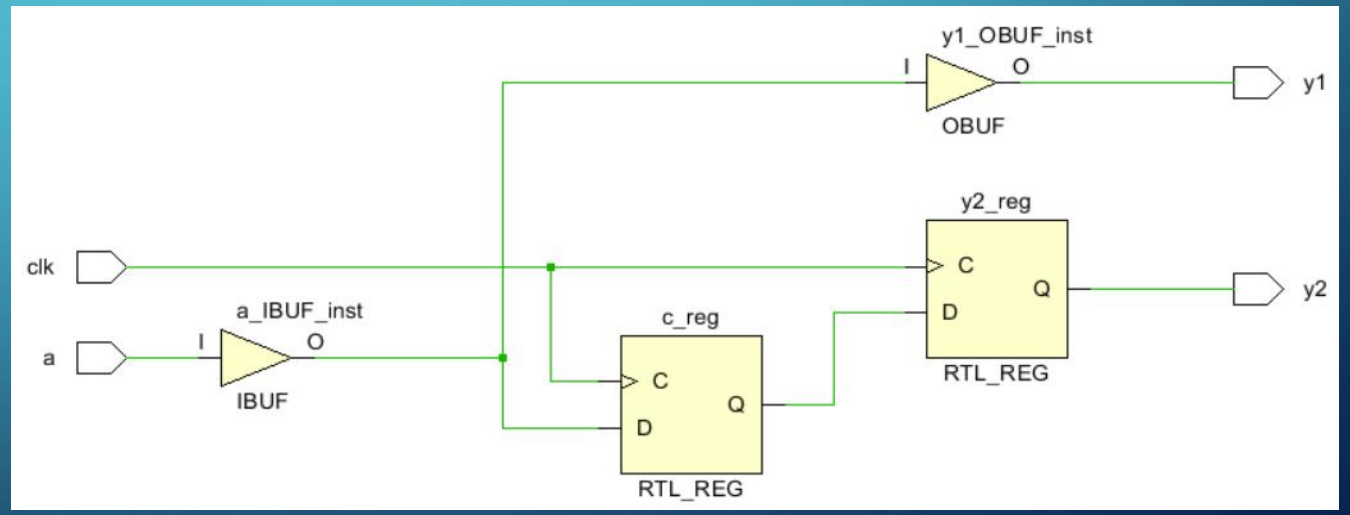
2022 SUMMER TERM

COMBINATORIAL VS SEQUENTIAL

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module s_adder(input clk,rst,a,b, output c );
reg c;
always @(posedge clk)
begin
    if(rst)
        c<=0;
    else
        c<=a+b;
end
endmodule
```




```
module test_assign(input a,clk,output y1,y2 );
reg b,c,y1,y2;
always @ *
begin
    b=a;
    y1=b;
end

always @(posedge clk)
begin
    c<=a;
    y2<=c;
end
endmodule
```



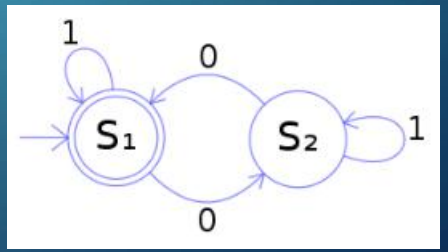
Strongly recommend： using blocking-assignment in combinatorial circuit while using non-blocking-assignment in sequential circuit

FSM : FINITE STATE MACHINES

An FSM is defined by a list of its states, its initial state, and the conditions for each transition. Finite state machines are of two types – deterministic finite state machines and non-deterministic finite state machines.[1] A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

When describing a FSM, the key is to clearly describe several elements of the state machine :

- how to make state transition
- the conditions of state transition
- what is the output of each state.



Generally speaking, the state transition part is a synchronous sequential circuit after the state machine is implemented, and the judgment of the state transition condition is combinational logic.

https://en.wikipedia.org/wiki/Finite-state_machine

FSM - STATE CODE

- **Binary code**
- **Gray code**
- **One hot code**

.....

State	S0	S1	S2	S3
Binary code	00	01	10	11
Gray code	00	01	11	10
One hot code	0001	0010	0100	1000

FSM - STATE CODE CONTINUED

- In Verilog design, states of FSM must be explicitly assigned
- Using code number directly

  ... current_state <= 2'h2; ...

- Using parameters

  parameter state1 = 2'h1, state2 = 2'h2;

  ... current_state <= state2; ...

- Using define
  - 'define state1 2'h1
  - 'define state2 2'h2
  - ... current_state <= 'state2; ...

FSM - SYNCHRONOUS SET & RESET

- Positive edge trigger @(posedge clk)
- Negtive edge trigger @(negedge clk)
- Example

```
always @(posedge clk) begin
        if(reset) begin
                // set output as 0, set state as S0
        end
        else
                if(set) begin
                        // set output as 1
                end
                else begin
                        //logical function transformation
                end
end
```

FSM - ASYNCHRONOUS SET & RESET

- Positive edge trigger & high level effective set/reset @(posedge clk or posedge set/reset)
- Positive edge trigger & low level effective set/reset @(posedge clk or negedge set/reset)
- Example

```
always @(posedge clk or posedge set or posedge reset) begin
        if(reset) begin
                // set output as 0, set state as S0
        end
        else
                if(set) begin
                        // set output as 1
                end
                else begin
                        //logical function transformation
                end
end
```

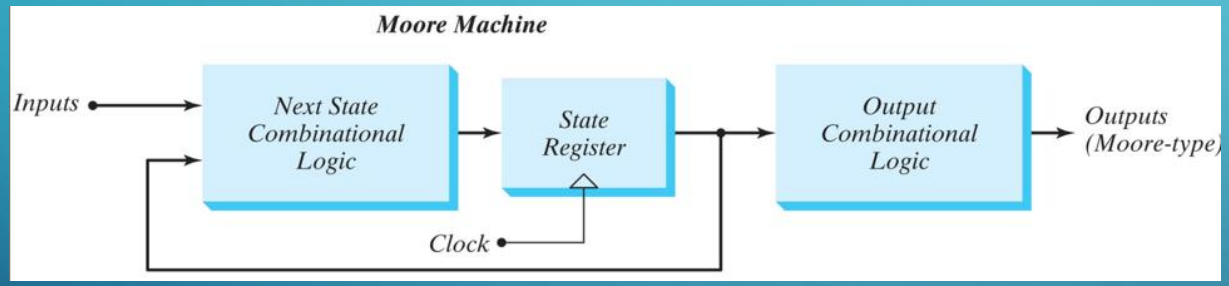
FSM - WAYS ON FSM

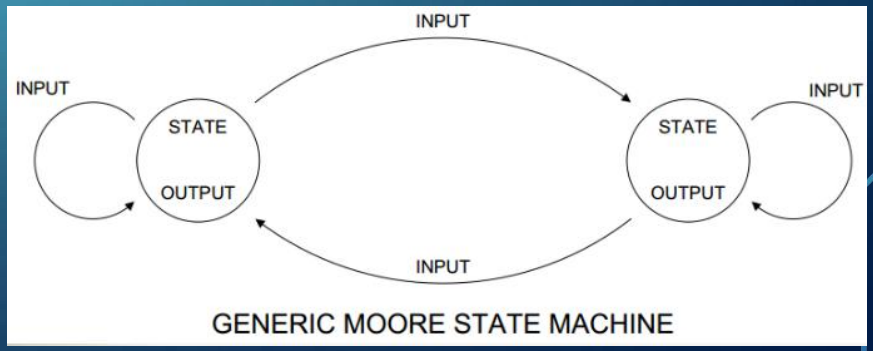
- (1) **One-stage:** The whole FSM is written into one always block, which describes not only the state transition, but also the input and output of the state.（NOT suggested）
- (2) **Two-stages:** two always blocks are used to describe the state machine, one of which uses synchronous time series to describe the state transition; the other uses combinational logic to judge the condition of state transition, to describe the law of state transition and output;
- (3) **Three-stages:** One always module uses synchronous timing to describe state transition, One always uses combination logic to judge state transition conditions and describe state transition rules, and the Other always block describes state output (which can either be output of combination circuit or the output of sequential circuit).

Generally speaking, the recommended FSM description method is the latter two. This is because: FSM, like other designs, is best designed in a synchronous sequential manner to improve the stability of the design and eliminate burrs.

MOORE MODE

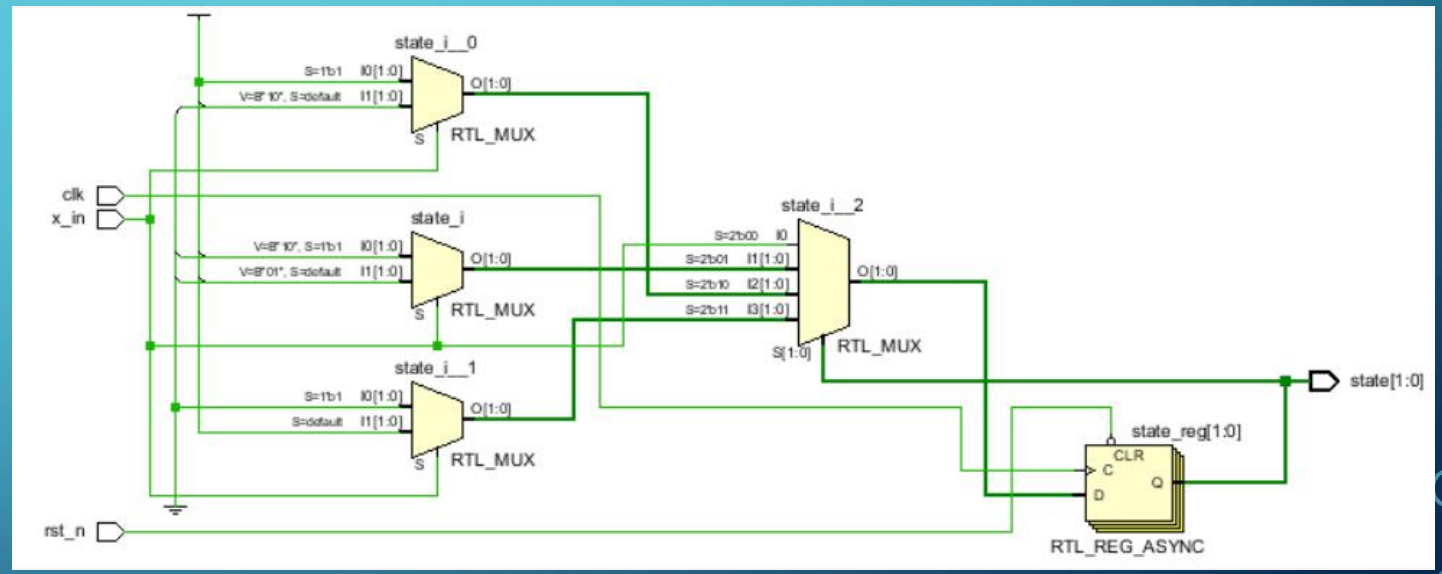
- Outputs are functions of present state only
- Outputs are synchronized with clock





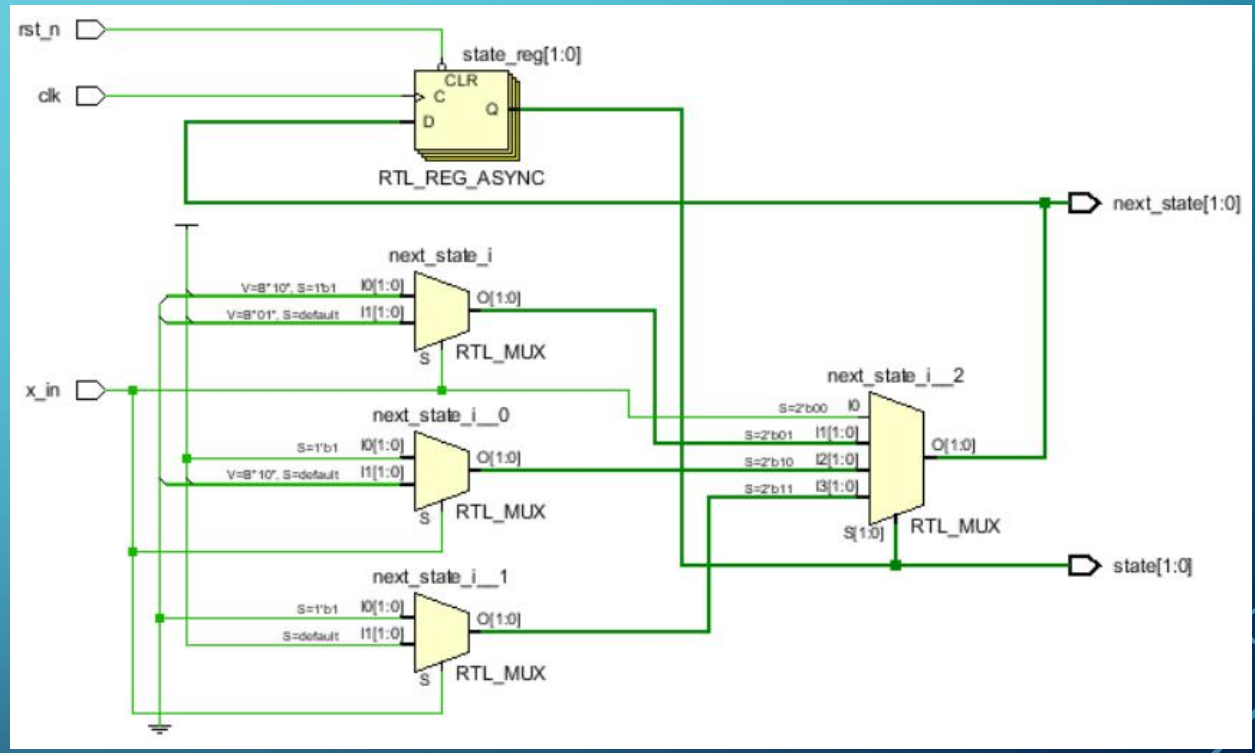
MOORE MODE WITH 1 STAGE

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module moore_1b(input clk,rst_n,x_in,output[1:0] state);
reg [1:0] state;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin
    if(~rst_n)
        state <= S0;
    else
    case(state)
    S0: if(x_in) state <= S1; else state <= S0;
    S1: if(x_in) state <= S2; else state <= S1;
    S2: if(x_in) state <= S3; else state <= S2;
    S3: if(x_in) state <= S0; else state <= S3;
    endcase
end
endmodule
```



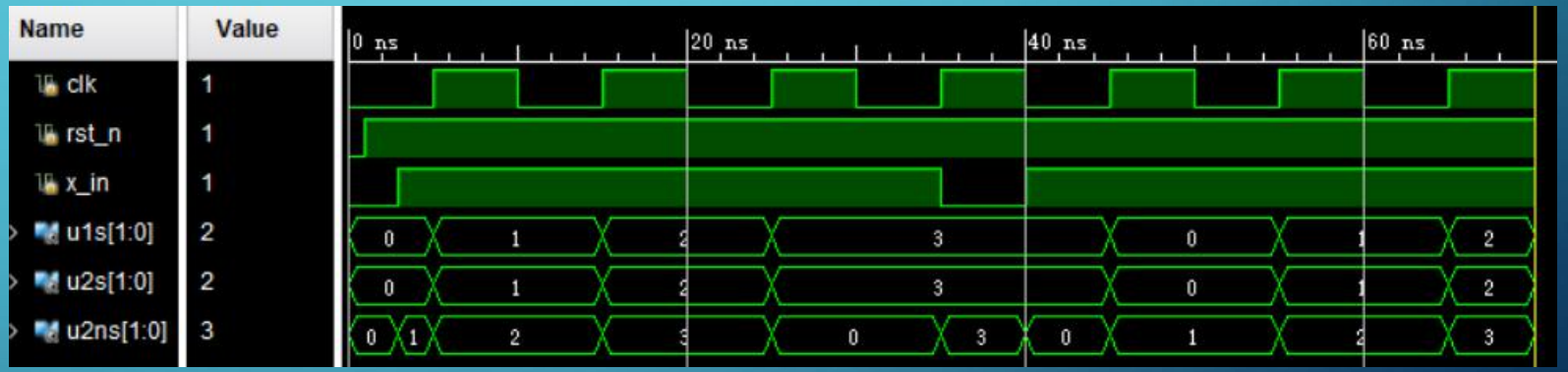
MOORE MODE WITH 2-STAGES

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module moore_2b(input clk,rst_n,x_in,output[1:0] state,next_state);
reg [1:0] state,next_state;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin
    if(~rst_n)
        state <= S0;
    else
        state <= next_state;
end
always @(state,x_in) begin
    case(state)
    S0: if(x_in) next_state = S1; else next_state = S0;
    S1: if(x_in) next_state = S2; else next_state = S1;
    S2: if(x_in) next_state = S3; else next_state = S2;
    S3: if(x_in) next_state = S0; else next_state = S3;
    endcase
end
endmodule
```



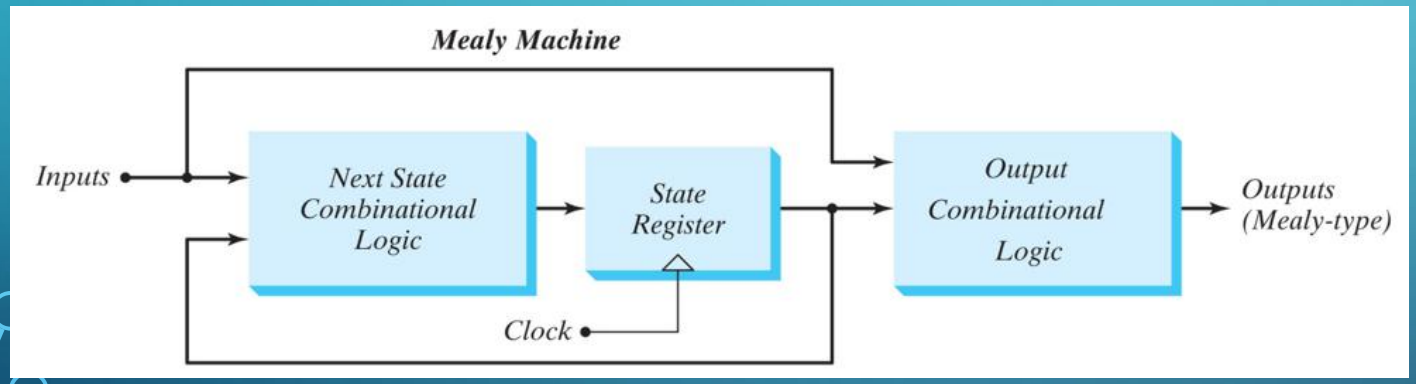
SIMULATION ON ONE & TWO STAGE OF MOORE

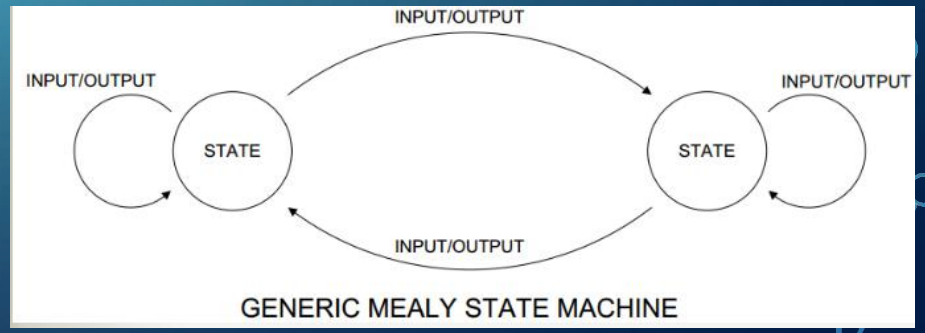
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module sim_moore_12();
reg clk,rst_n,x_in;
wire [1:0] u1s,u2s,u2ns;
moore_1b u1(clk,rst_n,x_in,u1s);
moore_2b u2(clk,rst_n,x_in,u2s,u2ns);
initial #70 $finish;
initial begin
clk = 1'b0;
rst_n=1'b0;
forever #5 clk=~clk;
end

initial fork
    x_in=1'b0;
    #1 rst_n = 1'b1;
    #3 x_in = 1'b1;
    #35 x_in = 1'b0;
    #40 x_in = 1'b1;
join
endmodule
```



MEALY MODE

- Outputs are functions of both present state and inputs
- Outputs may change if inputs change





MEALY MODE WITH 1-STAGE

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module mealy_1b(input clk,rst_n,x_in,output[1:0] state,output y);
reg [1:0] state;
reg y;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin
    if(~rst_n)
    begin
        state <= S0;
        y <= 1'b0;
    end
    else
    case(state)
    S0: if(x_in) {state,y} <= {S1,1'b0}; else {state,y} <= {S0,1'b0};
    S1: if(x_in) {state,y} <= {S2,1'b0}; else {state,y} <= {S1,1'b0};
    S2: if(x_in) {state,y} <= {S3,1'b0}; else {state,y} <= {S2,1'b0};
    S3: if(x_in) {state,y} <= {S0,1'b1}; else {state,y} <= {S3,1'b0};
    endcase
end
endmodule
```

MEALY MODE WITH TWO-STAGES

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module mealy_2b(input clk,rst_n,x_in,output[1:0] state,next_state,output y);
reg [1:0] state,next_state;
reg y;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin
    if(~rst_n)
    begin
        state <= S0;
        y <= 1'b0;
    end
    else
        state <= next_state;
end
always @(state,x_in) begin
    case(state)
    S0: if(x_in) {next_state,y} = {S1,1'b0}; else {next_state,y} = {S0,1'b0};
    S1: if(x_in) {next_state,y} = {S2,1'b0}; else {next_state,y} = {S1,1'b0};
    S2: if(x_in) {next_state,y} = {S3,1'b0}; else {next_state,y} = {S2,1'b0};
    S3: if(x_in) {next_state,y} = {S0,1'b1}; else {next_state,y} = {S3,1'b0};
    endcase
end
endmodule
```

MEALY MODE WITH THREE-STAGES

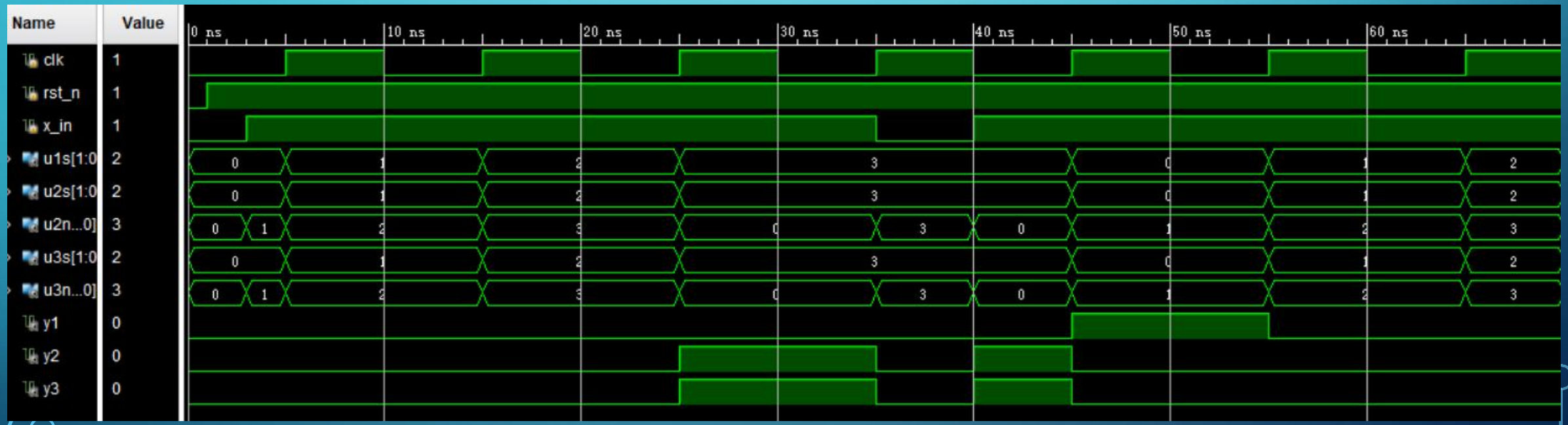
```
module mealy_3b(input clk,rst_n,x_in,output[1:0] state,next_state,output y);
reg [1:0] state,next_state;
reg y;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge clk,negedge rst_n) begin...
always @(state,x_in) begin...
always @(state,x_in) begin...
endmodule
```

```
always @(posedge clk,negedge rst_n) begin
    if(~rst_n)
    begin
        state <= S0;
        y <= 1'b0;
    end
    else
        state <= next_state;
end
```

```
always @(state,x_in) begin
    case(state)
    S0: if(x_in) next_state = S1; else next_state = S0;
    S1: if(x_in) next_state = S2; else next_state = S1;
    S2: if(x_in) next_state = S3; else next_state = S2;
    S3: if(x_in) next_state = S0; else next_state = S3;
    endcase
end
```

```
always @(state,x_in) begin
    case(state)
    S0,S1,S2: y=1'b0;
    S3: if(x_in) y=1'b1; else y=1'b0;
    endcase
end
```

SIMULATION RESULT



MOORE VS MEALY

- Moore mode :
  - output is the state of the circuit, usually not simplified the state
  - Relatively simple
- Mealy mode :
  - output is not the state of the circuit, the state can be simplified
  - Relatively complex

PRACTICE 1

A circuit has 2 inputs (x_in(5bit-width), clk) and 1 output(y_out). The circuit get the state of x_in at every posedge of clk, If the total number of received 1 is a multiple of 5, then y_out is valid, otherwise y_out is invalid.

1. Do the design by using behavior modeling in verilog. Is this a moore mode or mealy mode? Try to implement the circuit by two-stages and three-stages respectively.

2. Build testbench and verify the function on this sequential circuit.

3. Try to implement the circuit on EGO1 board