Lab 5
Introduction to Verilog - 2

Instructor: Chia-Chi, Tsai

Speaker: Johnson Liu

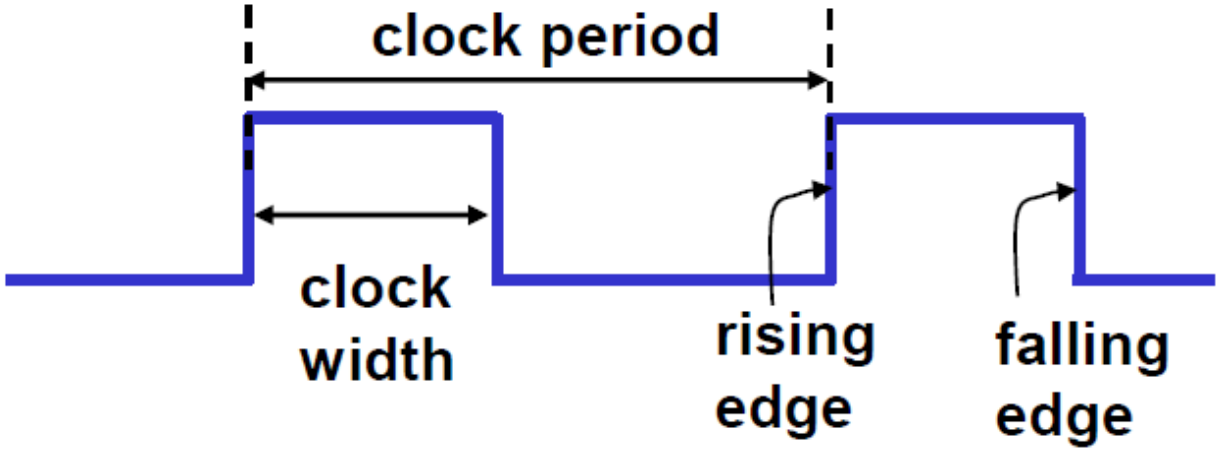
Outline

1. Latch & Flip-Flop & Register
2. Combinational vs Sequential
3. Blocking vs Non-Blocking
4. Finite-State Machine(FSM)
5. Choosing Correct Data Type
6. Reference

Latch & Flip-Flop & Register

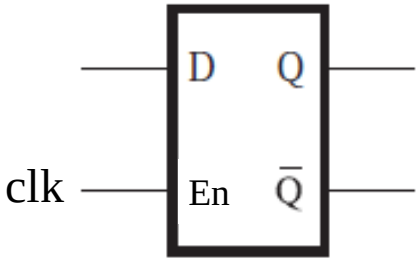
Clock

- Clock period: the time between successive transitions in the same direction. (second/cycle)
- Clock frequency: the reciprocal of clock period. (cycle/second)
- Clock width: the time interval during which clock is equal to 1.
- Duty cycle: the ratio of the clock width and clock period
- Active high: the circuit changes occur at the rising edge or during the logic is 1.
- Active low: the circuit changes occur at the falling edge or during the logic is 0.

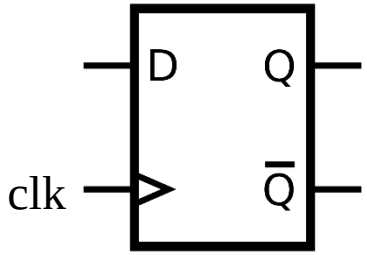


D-Latch & D-Flip-Flop

- The state of a latch or flip-flop is switched by a change of the control input like clock.
- Flip-flops and latches are used as data storage elements.
- D: Data/Delay



D-Latch



D-FF

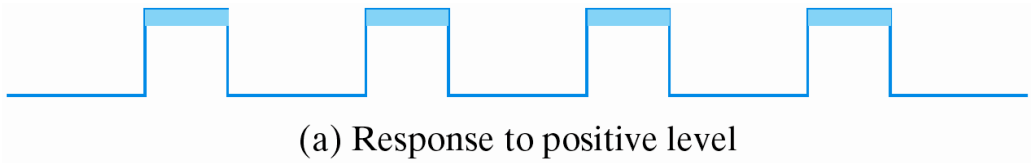
Enable

En	D	Q	$\overline{Q}$	Comment
0	X	Q_{prev}	$\overline{Q}_{prev}$	No change
1	0	0	1	Reset
1	1	1	0	Set

Clock	D	Q
Rising edge	0	0
Rising edge	1	1
Non-rising	X	Q_{prev}

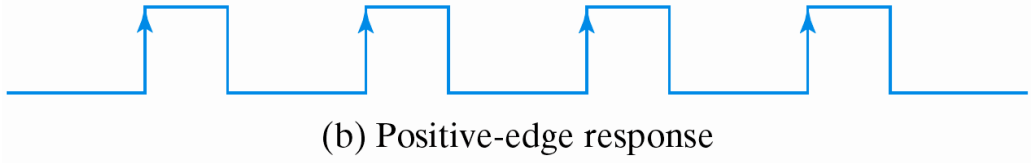
Clock as Control Input

- Clock(clk) can be divided into 2 type
  1. Level triggered => Latches
  2. Edge triggered => Flip-Flop

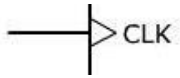


(a) Response to positive level

Warning! multiple transitions might happen during logic-1 level.

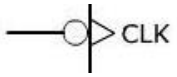


(b) Positive-edge response



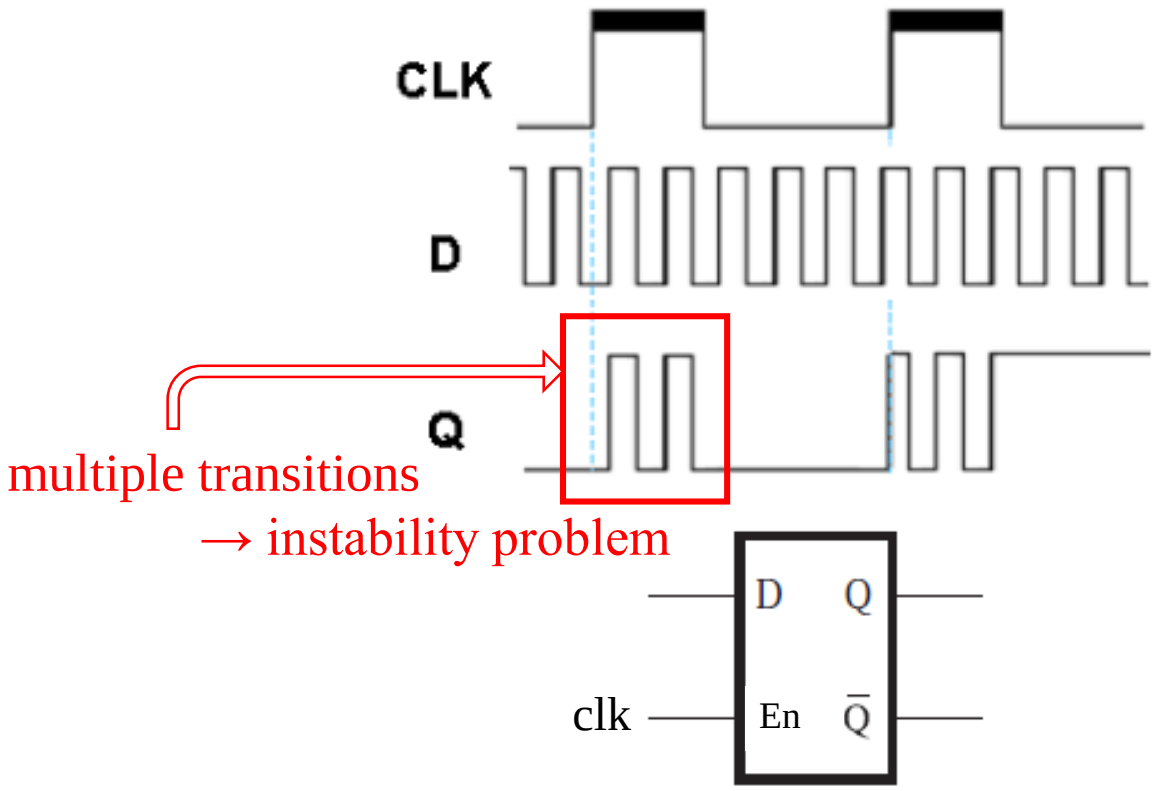

```
always @(posedge clk)
```

(c) Negative-edge response

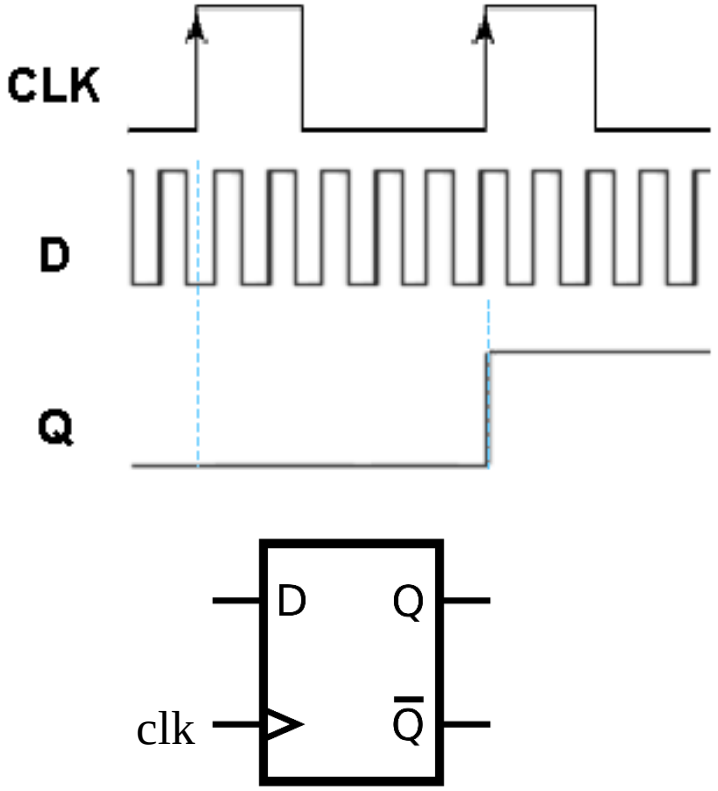



```
always @(negedge clk)
```

D-Latch vs D-Flip-Flop in Waveform



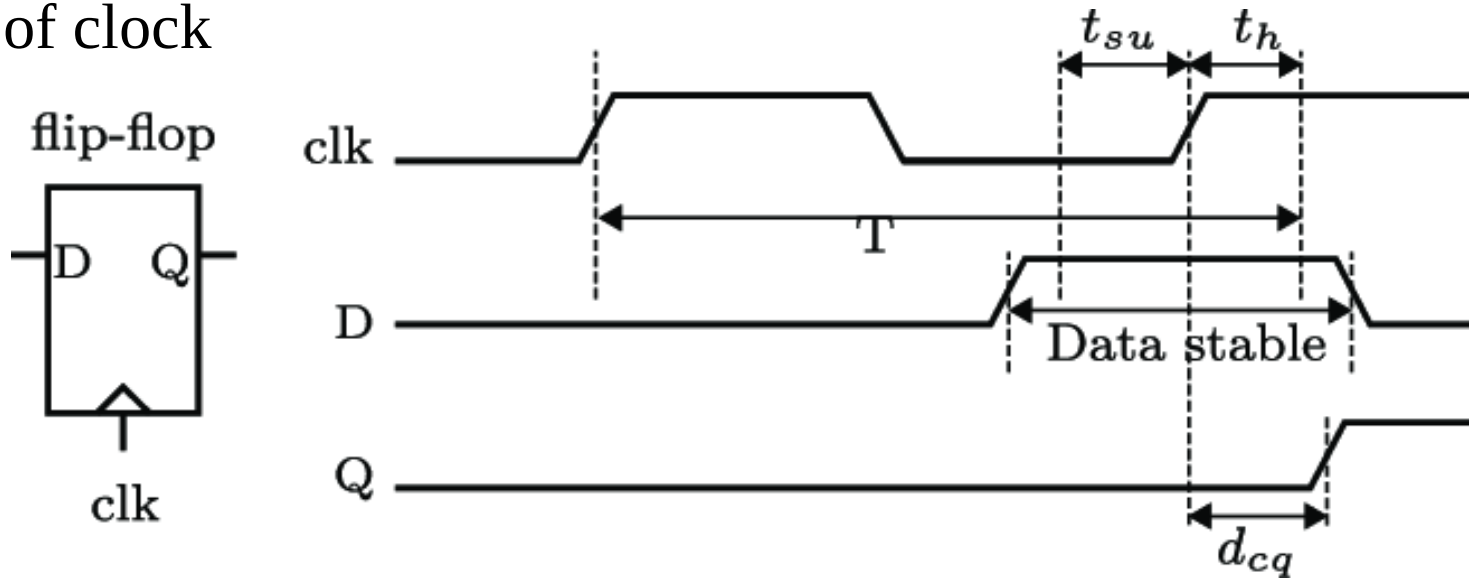
D-Latch



D-FF

Setup Time & Hold Time

- Setup Time
  The amount of time the data at the synchronous input (D) must be stable before the active edge of clock.
- Hold Time
  The amount of time the data at the synchronous input (D) must be stable after the active edge of clock



Synchronous & Asynchronous Reset

1. Synchronous Reset

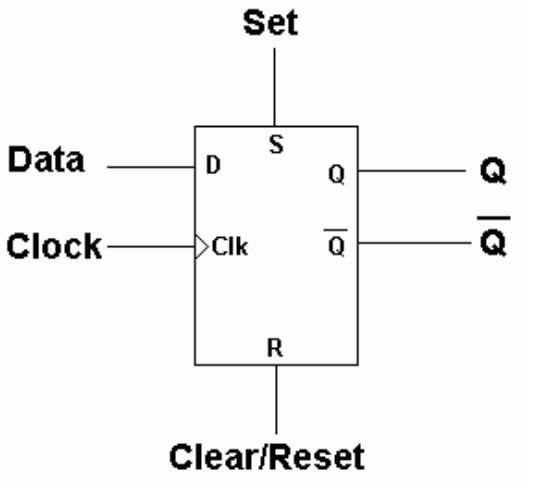
```
module DFF (clk, rst, D, Q);
    input clk, rst;
    input D;
    output Q;
    reg Q;

    always @ (posedge clk)
    begin
        if (rst)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

2. Asynchronous Reset

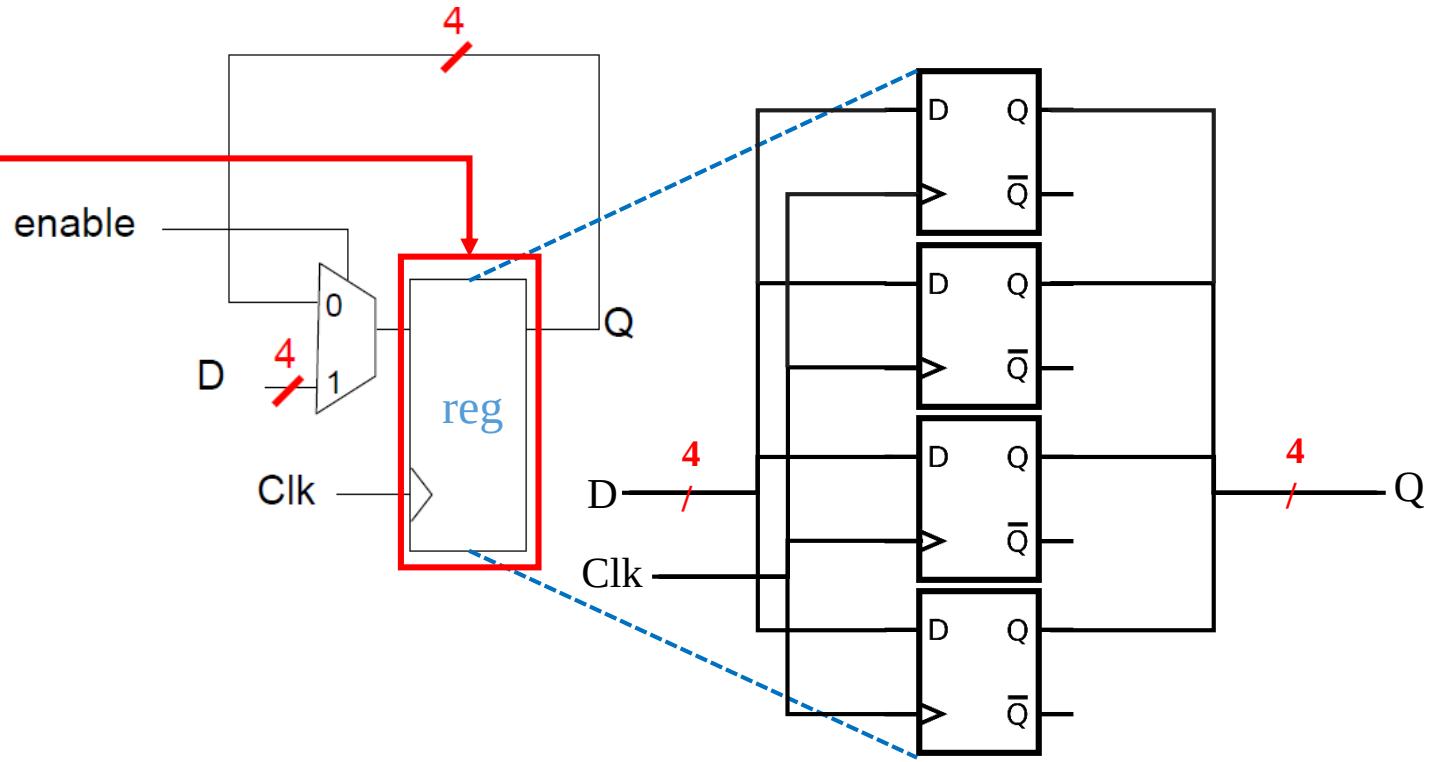
```
module DFF (clk, rst, D, Q);
    input clk, rst;
    input D;
    output Q;
    reg Q;

    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```



Register

```
module DFF_REG (clk, enable, D, Q);
    input clk, enable;
    input [3:0] D;
    output [3:0] Q;
    reg [3:0] Q;
    always @(posedge clk)
    begin
        if (enable)
        begin
            Q <= D;
        end
        else
        begin
            Q <= Q;
        end
    end
endmodule
```



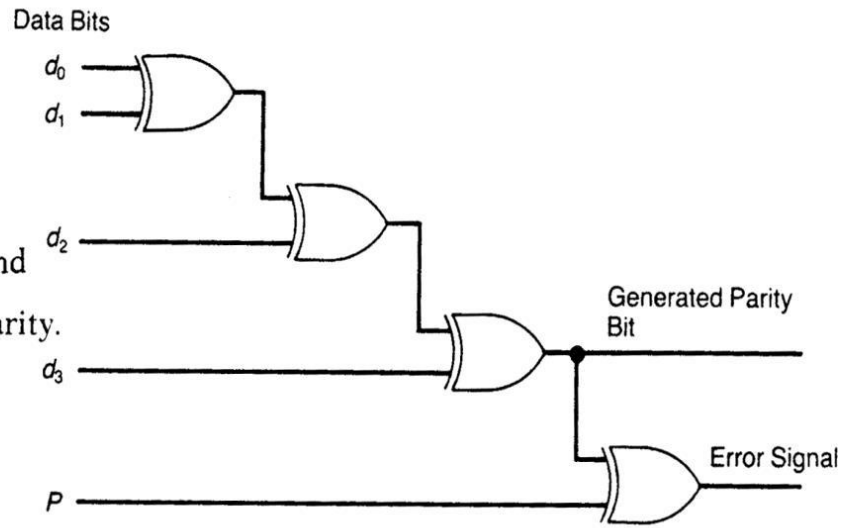
Combinational

vs

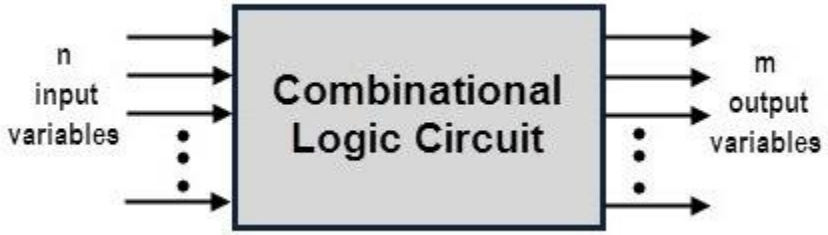
Sequential

Combinational Circuit

A combinational circuit consists of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.

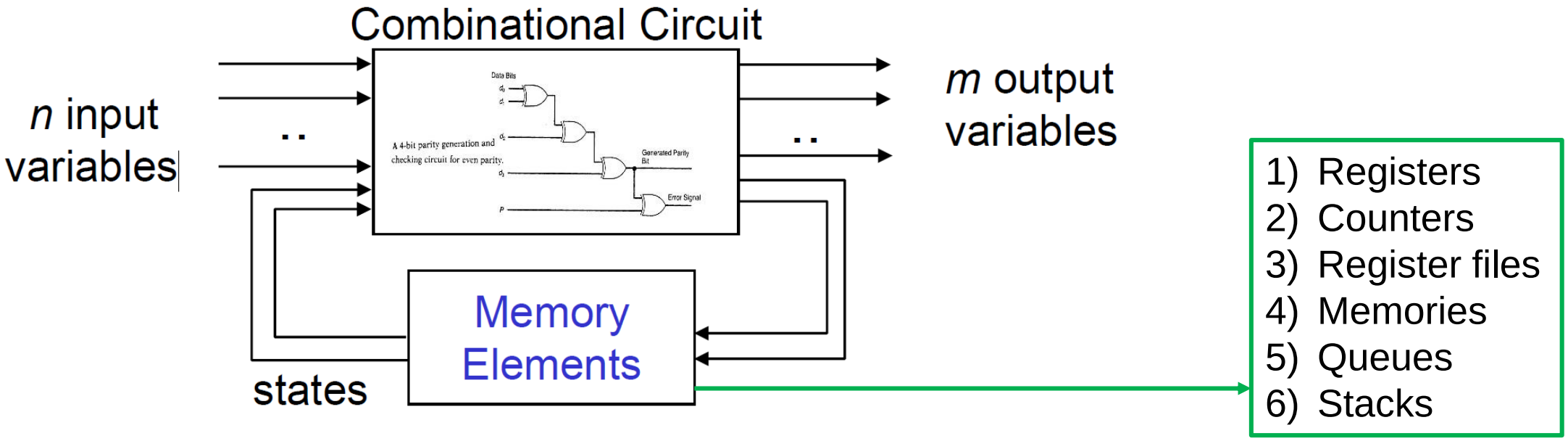


A 4-bit parity generation and checking circuit for even parity.



Sequential Circuit

A sequential circuit is a system whose outputs at any time are determined from the present combination of inputs and the previous inputs or outputs, so sequential components contain memory elements.



Blocking

vs

Non-Blocking

Different in Assignment

- Blocking Assignment (=) are order sensitive
- Non-Blocking Assignment (<=) are order independent Blocking

1. Blocking

```
initial
begin
    a = #12 1;
    b = #3 0;
    c = #2 3;
end
```

2. Non-Blocking

```
initial
begin
    d <= #12 1;
    e <= #3 0;
    f <= #2 3;
end
```

- initial: Simulation start at 0.
- #n: Delay of n time units.

Timestamp	0	2	3	12	15	17
a	x	x	x	1	1	1
b	x	x	x	x	0	0
c	x	x	x	x	x	3
d	x	x	x	1	1	1
e	x	x	0	0	0	0
f	x	3	3	3	3	3

Test Yourself

1. Blocking

```
initial
begin
    ...
    A = 1;
    B = 0;
    ...
    A = B;
    B = A;
```

B = ? is used
A = ? is used

```
initial
begin
    ...
    A = 1;
    B = 0;
    ...
    B = A;
    A = B;
```

A = ? is used
B = ? is used

2. Non-Blocking

```
initial
begin
    ...
    A <= 1;
    B <= 0;
    ...
    A <= B;
    B <= A;
```

B = ? is used
A = ? is used

```
initial
begin
    ...
    A <= 1;
    B <= 0;
    ...
    B <= A;
    A <= B;
```

A = ? is used
B = ? is used

Test Yourself

1. Blocking

```
initial
begin
    ...
    A = 1;
    B = 0;
    ...
    A = B;
    B = A;
```

B = 0 is used
A = 0 is used

```
initial
begin
    ...
    A = 1;
    B = 0;
    ...
    B = A;
    A = B;
```

A = 1 is used
B = 1 is used

2. Non-Blocking

```
initial
begin
    ...
    A <= 1;
    B <= 0;
    ...
    A <= B;
    B <= A;
```

B = 0 is used
A = 1 is used

```
initial
begin
    ...
    A <= 1;
    B <= 0;
    ...
    B <= A;
    A <= B;
```

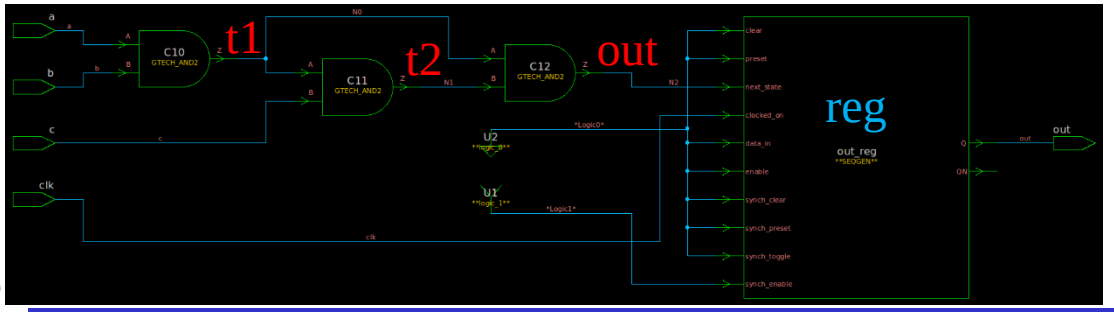
A = 1 is used
B = 0 is used

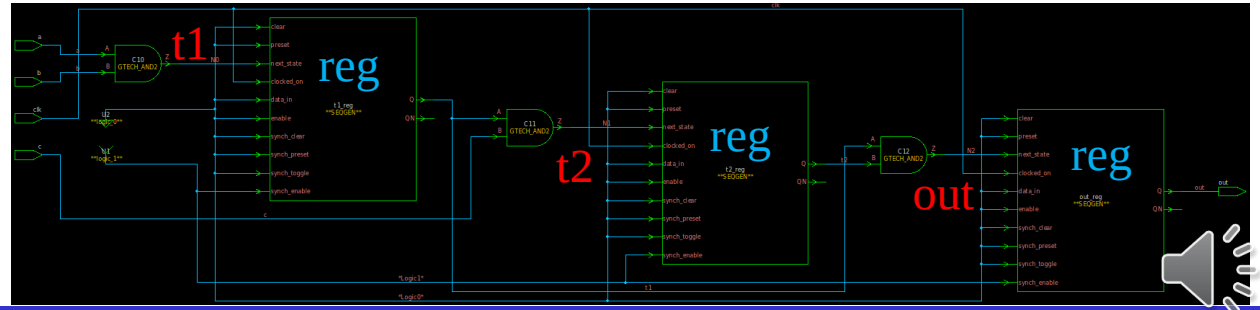
！ ！ ！Left side value in initial or always statements should declare as “reg”. ！ ！ ！

Different in Sequential Circuit

```
module test(clk, a, b, c, out);
    input clk, a, b, c;
    output out;
    reg t1, t2;
    reg out;
    always @(posedge clk)
    begin
        t1 = a & b;        ①
        t2 = t1 & c;       ②
        out = t1 & t2;     ③
    end
endmodule
```

```
module test(clk, a, b, c, out);
    input clk, a, b, c;
    output out;
    reg t1, t2;
    reg out;
    always @(posedge clk)
    begin
        t1 <= a & b;       ①
        t2 <= t1 & c;      ① old t1 is used
        out <= t1 & t2;    ① older t1 & old t2 is used
    end
endmodule
```

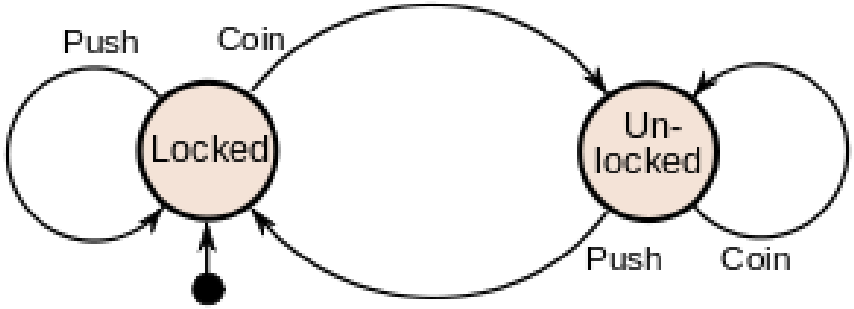




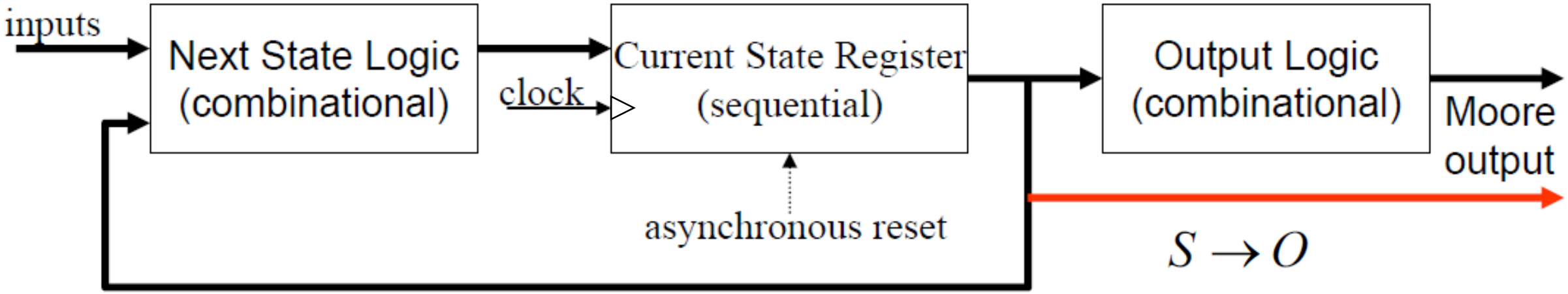
Finite-State Machine(FSM)

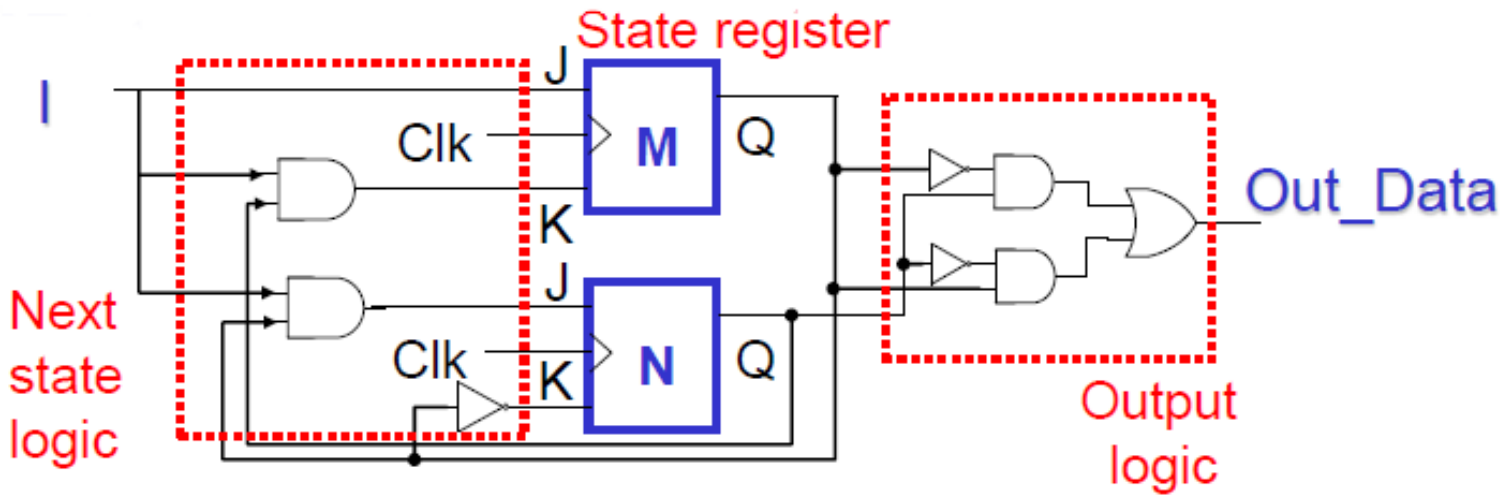
Finite-State Machine(FSM)

- A finite-State Machine (FSM) or finite-state automaton (FSA) is a mathematical model of computation.
- It is an abstract machine that can be in exactly one of a finite number of states at any given time.
- It can change from one state to another in response to some inputs; the change from one state to another is called a transition
- Type of FSM:
  1. Moore machine
  2. Mealy machine



Moore Machine in Circuit

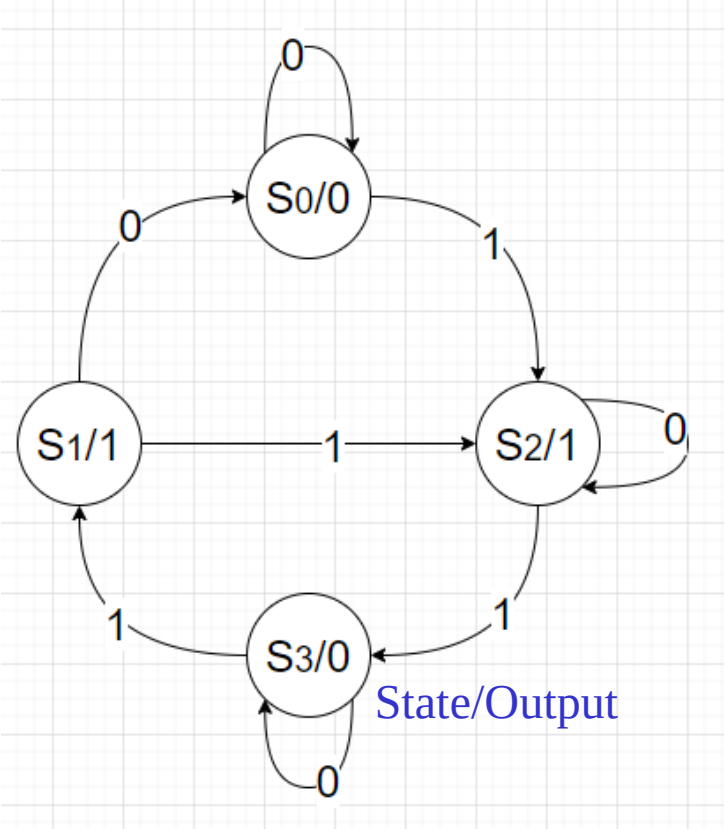




Moore Machine

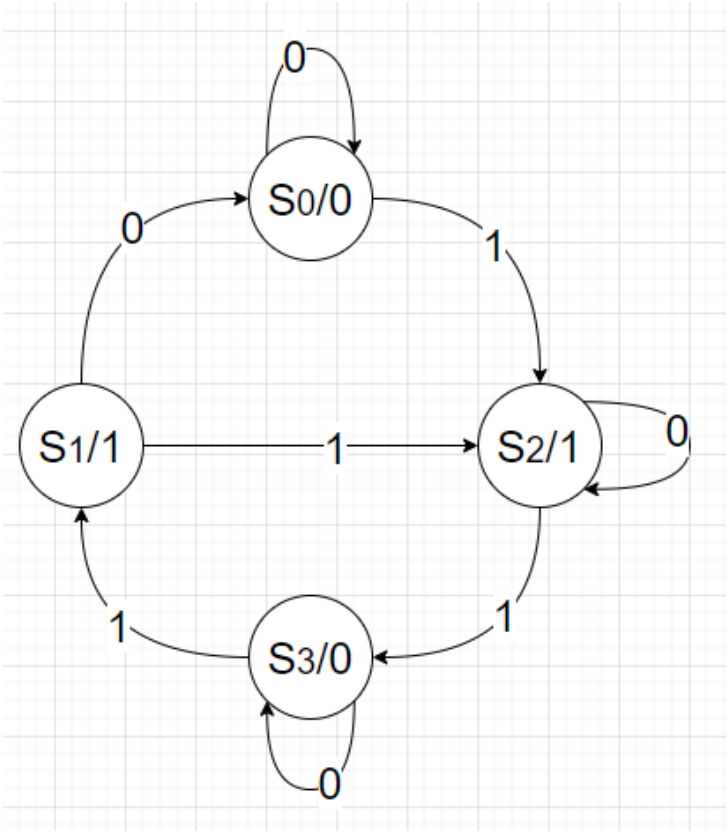
- Moore machine is a finite-state machine whose output values are determined only by its current state.
- Moore machine: S → O (S : state, O: output)

Present State	Next State		Output	
	I=0	I=1	I=0	I=1
S_0	S_0	S_2	0	0
S_1	S_0	S_2	1	1
S_2	S_2	S_3	1	1
S_3	S_3	S_1	0	0



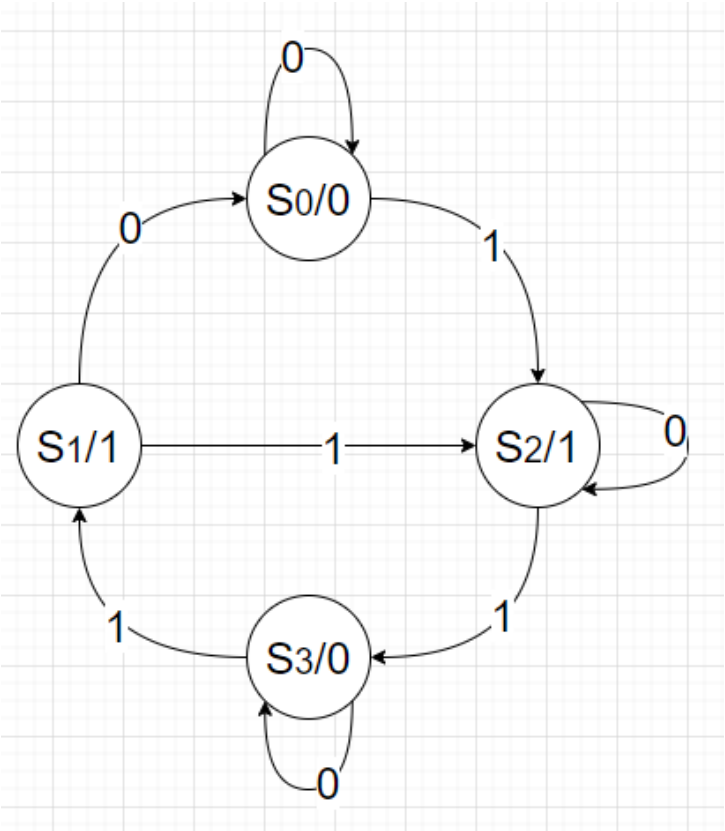
Moore - State Register(Sequential)

```
module moore(clk, rst, In_Data, Out_Data);
    input clk, rst, In_Data;
    output [1:0] Out_Data;
    reg [1:0] Out_Data;
    reg [1:0] State, NextState;
    parameter S0 = 2'b00, S1 = 2'b01,
              S2 = 2'b10, S3 = 2'b11;
    // State Register (Flip-Flops)
    always @(posedge clk or posedge rst)
    begin
        if(rst)
            State <= S0;
        else
            State <= NextState;
    end
```



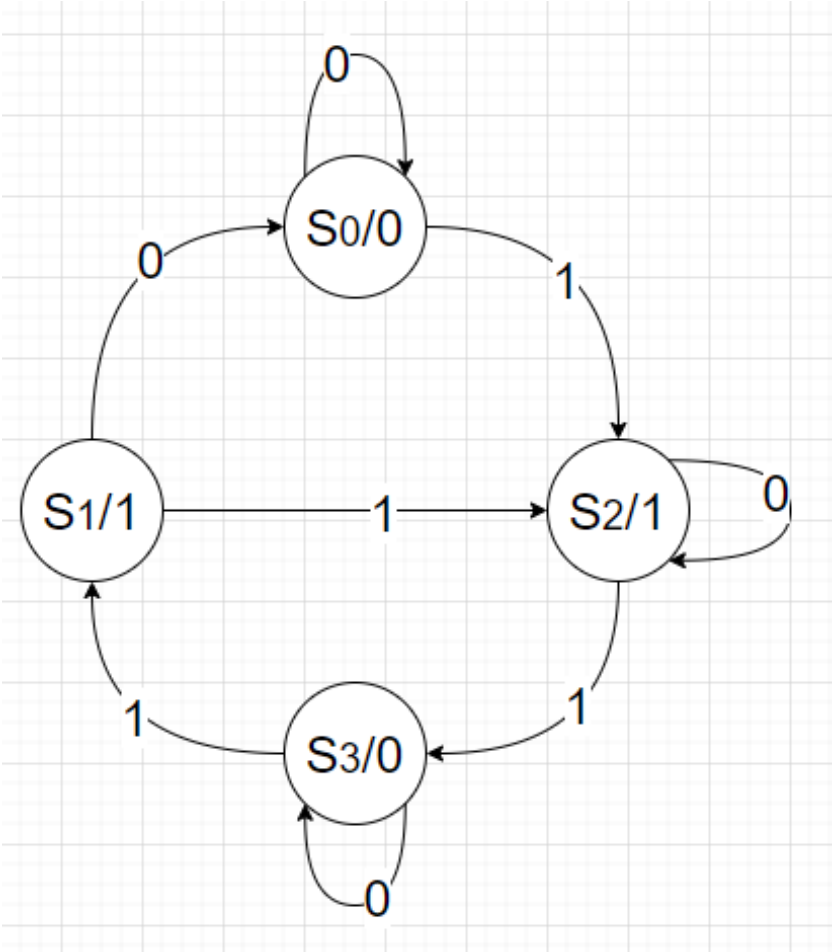
Moore - Next State Logic(Combinational)

```
// Next State Logic
always @(In_Data or State)
begin
    case(State)
        S0:
        begin
            if(In_Data == 1)
                NextState = S2;
            else
                NextState = S0;
        end
        S1:
        begin
            if(In_Data == 1)
                NextState = S2;
            else
                NextState = S0;
        end
        S2:
        begin
            if(In_Data == 1)
                NextState = S3;
            else
                NextState = S2;
        end
        S3:
        begin
            if(In_Data == 1)
                NextState = S1;
            else
                NextState = S3;
        end
    endcase
```



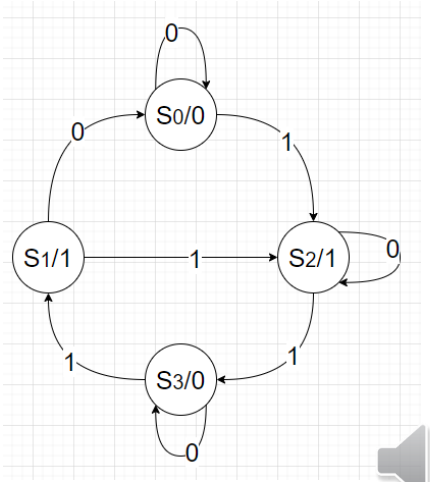
Moore - Output Logic(Combinational)

```
    // Output Logic
    always @(State)
    begin
        case(State)
            S0:Out_Data = 0;
            S1:Out_Data = 1;
            S2:Out_Data = 1;
            S3:Out_Data = 0;
        endcase
    end
endmodule
```

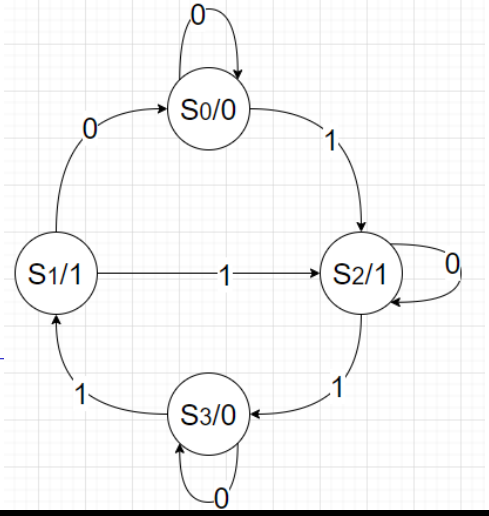


Moore - Bad Example(All Sequential)

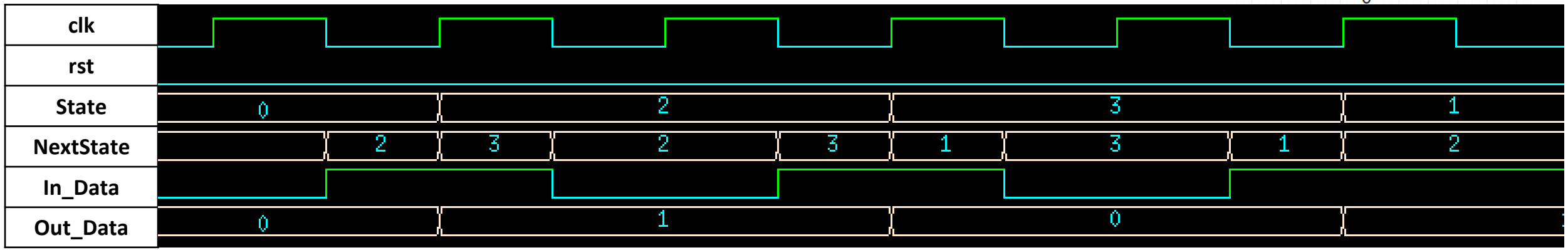
```
module moore_bad(clk, rst, In_Data, Out_Data);
    input clk, rst, In_Data;
    output [1:0] Out_Data;
    reg [1:0] Out_Data;
    reg [1:0] State;
    parameter S0 = 2'b00, S1 = 2'b01,
              S2 = 2'b10, S3 = 2'b11;
    always @(posedge clk)
    begin
        if(rst)
        begin
            State <= S0;
        end
        else
        begin
            case(State)
                S0:
                begin
                    Out_Data <= 0;
                    if(In_Data == 1)
                        State <= S2;
                    else
                        State <= S0;
                    end
                end
                S1:
                begin
                    Out_Data <= 1;
                    if(In_Data == 1)
                        State <= S2;
                    else
                        State <= S0;
                end
                S2:
                begin
                    Out_Data <= 1;
                    if(In_Data == 1)
                        State <= S3;
                    else
                        State <= S2;
                end
                S3:
                begin
                    Out_Data<= 0;
                    if(In_Data == 1)
                        State <= S1;
                    else
                        State <= S3;
                end
```



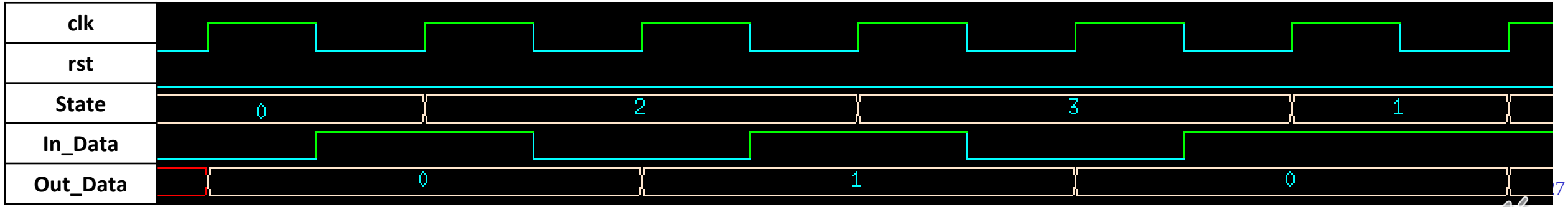
Moore Waveform



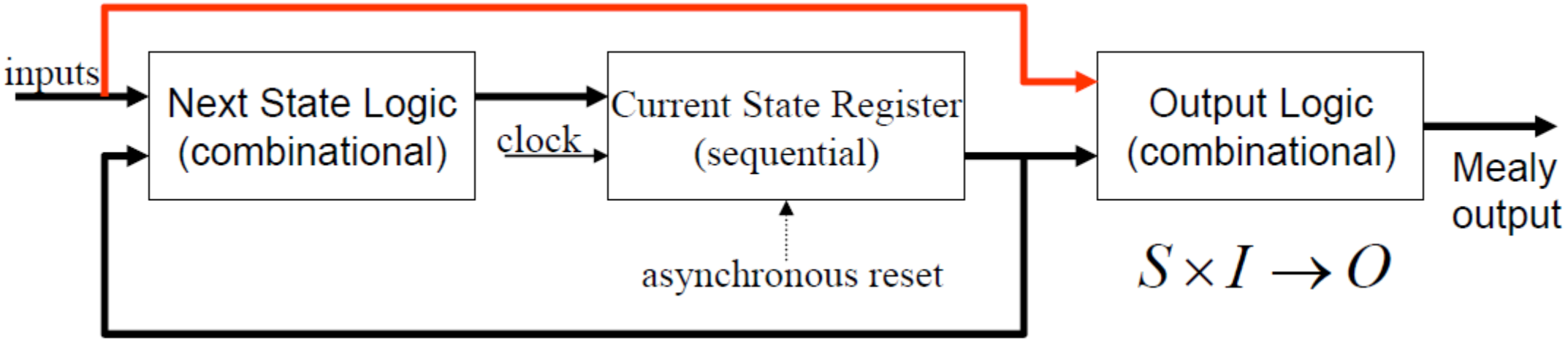
- Good Moore



- Bad Moore

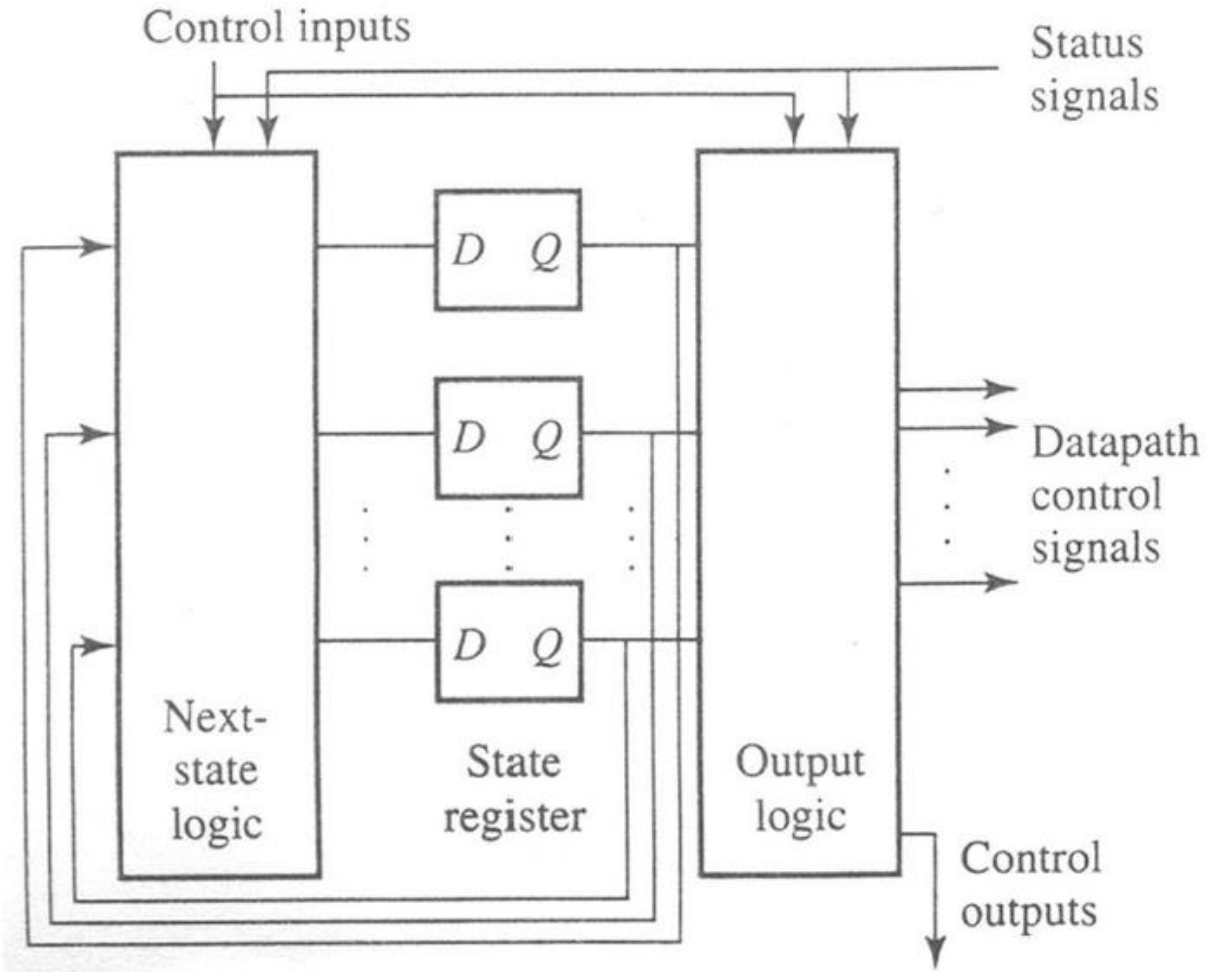


Mealy Machine in Circuit



$$S \times I \rightarrow O$$

Mealy Machine in Circuit

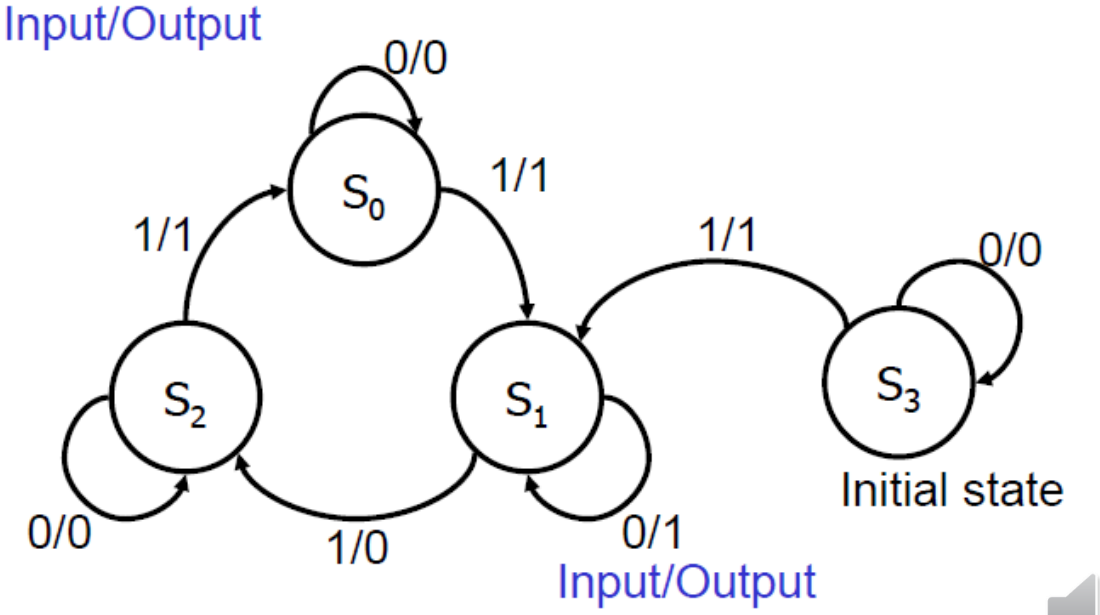


Mealy Machine

- Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs.
- Mealy machine: $S \times I \rightarrow O$ (S : state, I: Input, O: output)

Present State	Next State		Output	
	I=0	I=1	I=0	I=1
S_0	S_0	S_1	0	1
S_1	S_1	S_2	1	0
S_2	S_2	S_0	0	1
S_3	S_3	S_1	0	1

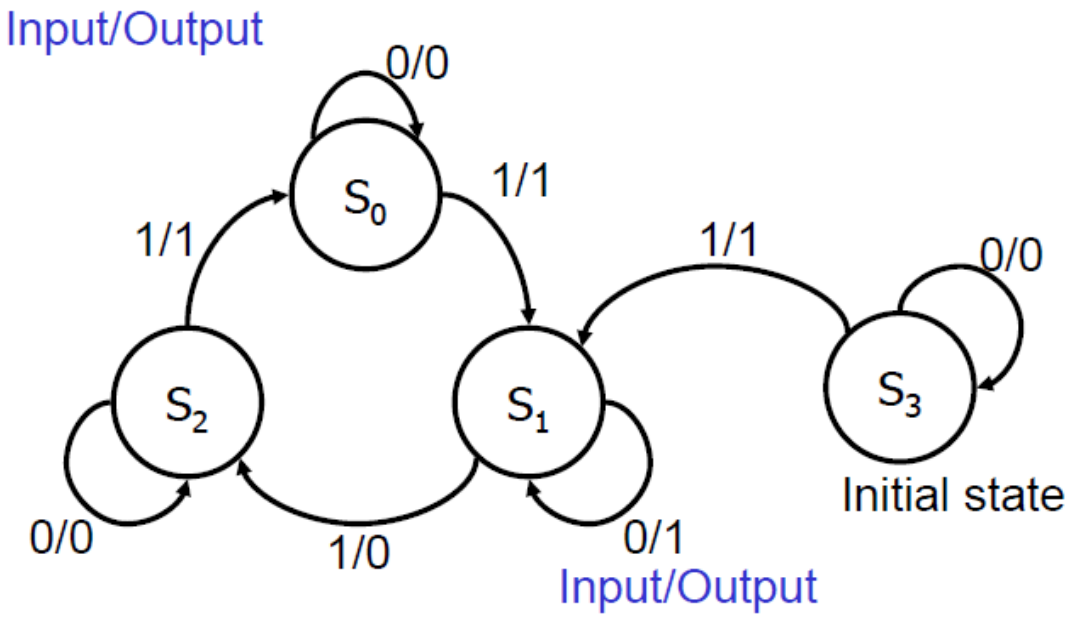
Four states: S_0, S_1, S_2, S_3



Mealy - State Register(Sequential)

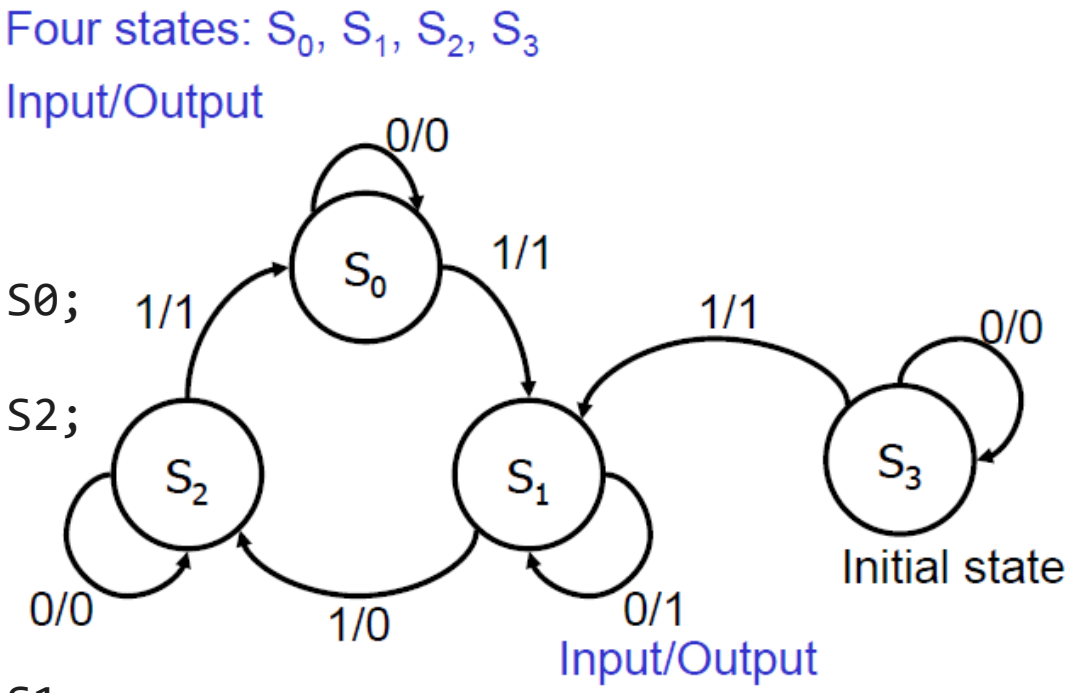
```
module mealy(clk, rst, In_Data, Out_Data);
    input clk, rst, In_Data;
    output [1:0] Out_Data;
    reg [1:0] Out_Data;
    reg [1:0] State, NextState;
    parameter S0 = 2'b00, S1 = 2'b01,
              S2 = 2'b10, S3 = 2'b11;
    // State Register (Flip-Flops)
    always @(posedge clk or posedge rst)
    begin
        if(rst)
            State <= S3;
        else
            State <= NextState;
    end
```

Four states: S_0, S_1, S_2, S_3



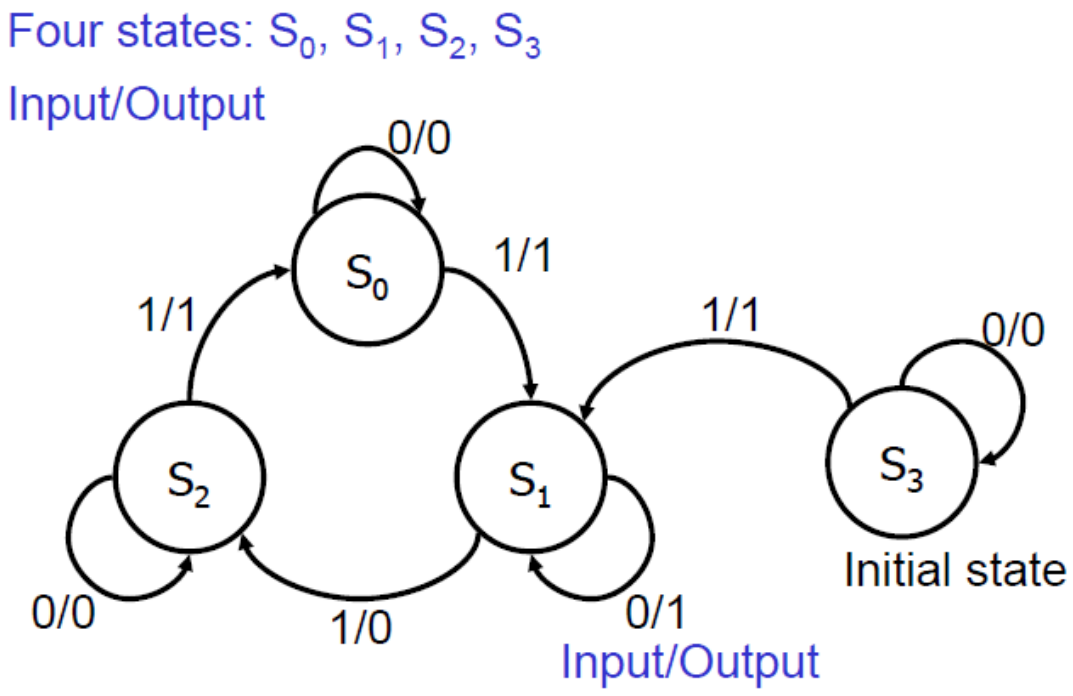
Mealy - Next State Logic(Combinational)

```
// Next State Logic
always @(In_Data or State)
begin
    case(State)
        S0:
        begin
            if(In_Data == 1)
                NextState = S1;
            else
                NextState = S0;
        end
        S1:
        begin
            if(In_Data == 1)
                NextState = S2;
            else
                NextState = S1;
        end
        S2:
        begin
            if(In_Data == 1)
                NextState = S0;
            else
                NextState = S2;
        end
        S3:
        begin
            if(In_Data == 1)
                NextState = S1;
            else
                NextState = S3;
        end
    endcase
```

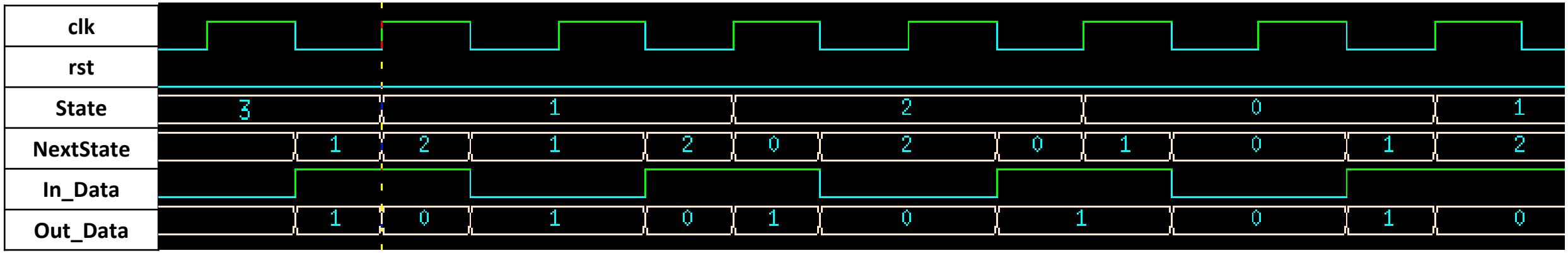


Mealy - Output Logic(Combinational)

```
// Output Logic
always @(In_Data or State)
begin
    case(State)
        S0:
        begin
            if(In_Data == 1)
                Out_Data = 1;
            else
                Out_Data = 0;
        end
        S1:
        begin
            if(In_Data == 1)
                Out_Data = 0;
            else
                Out_Data = 1;
        end
        S2:
        begin
            if(In_Data == 1)
                Out_Data = 1;
            else
                Out_Data = 0;
        end
        S3:
        begin
            if(In_Data == 1)
                Out_Data = 1;
            else
                Out_Data = 0;
        end
    endcase
```

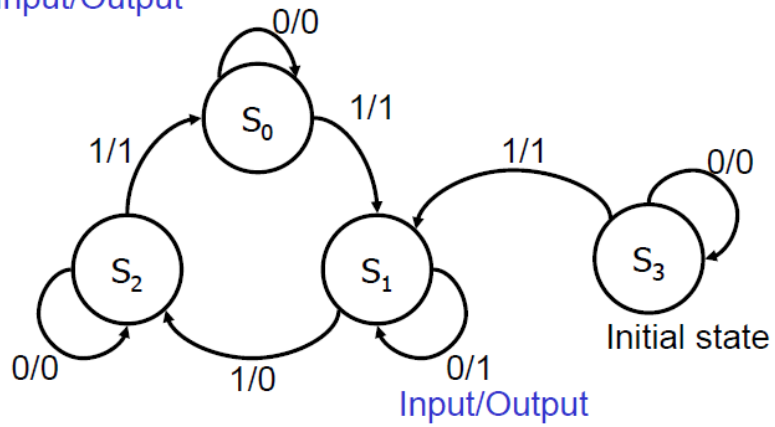


Mealy Waveform

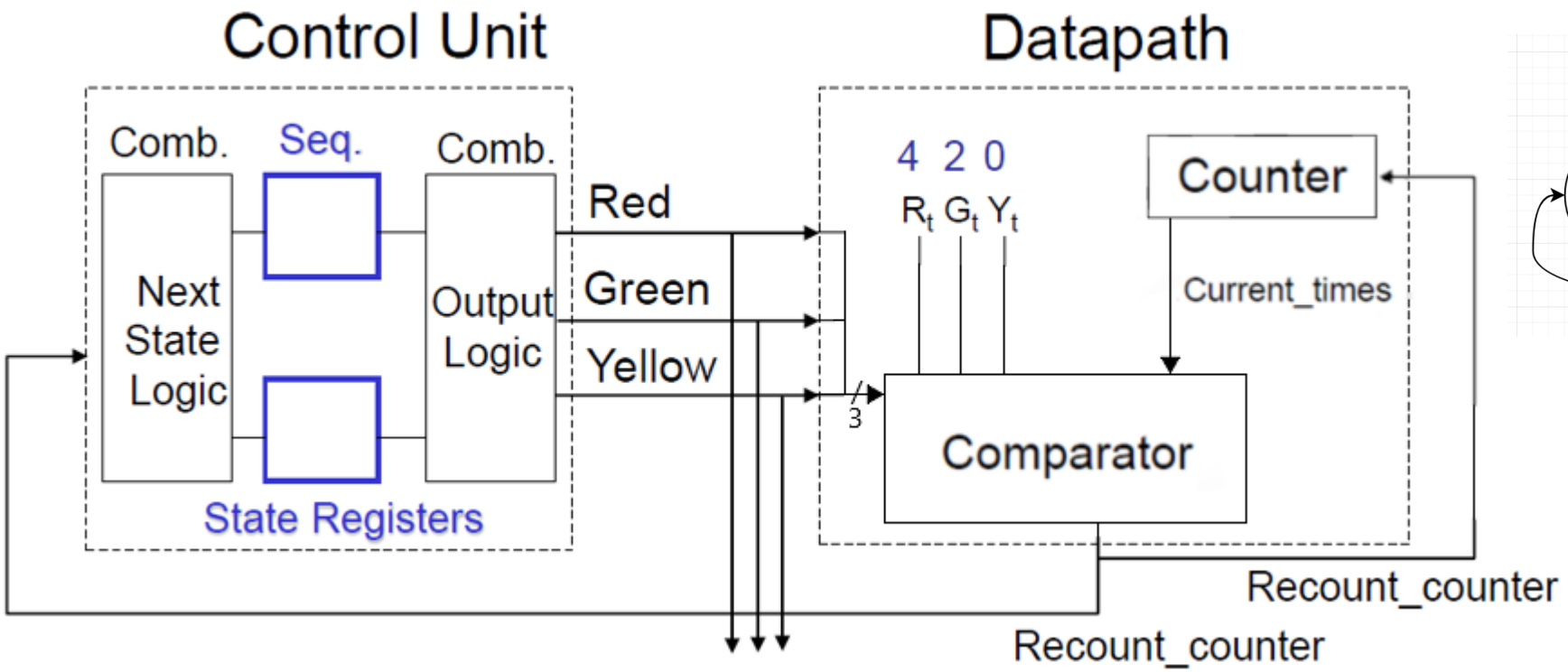


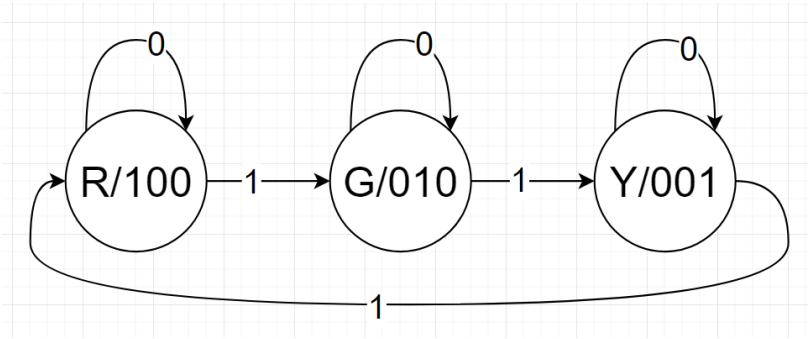
Four states: S_0, S_1, S_2, S_3

Input/Output



Example - Traffic Light Controller

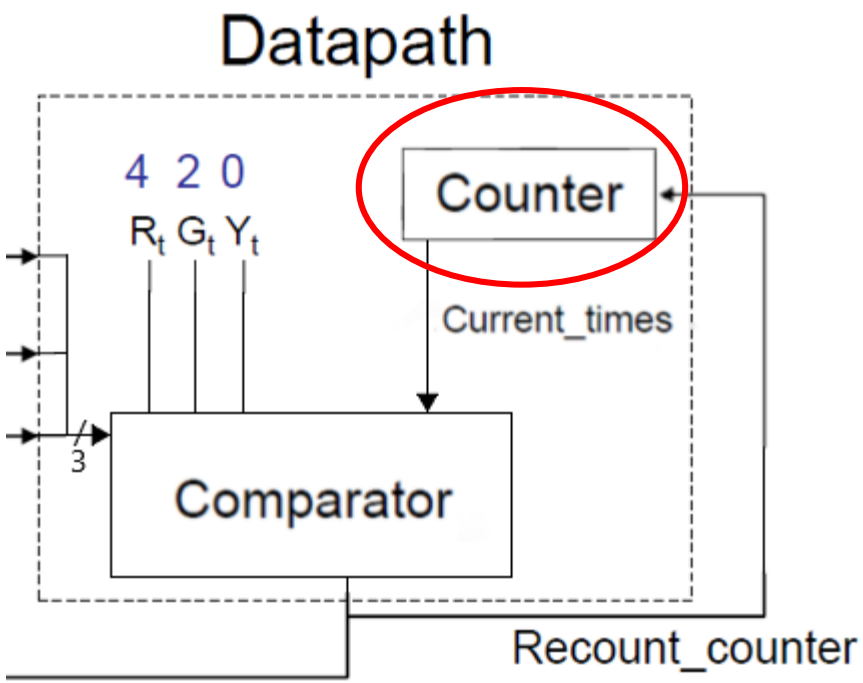




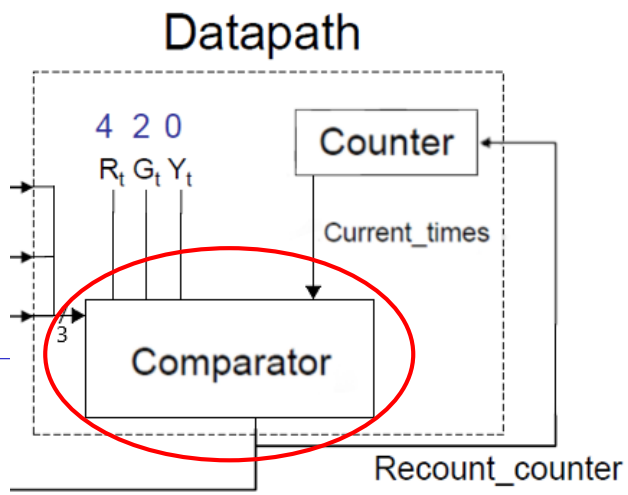
R_time: 4+1=5 cycles
G_time: 2+1=3 cycles
Y_time: 0+1=1 cycles

Counter(Sequential)

```
module Counter(clk, rst, Recount_Counter,
                    Count_Out);
    input clk, rst, Recount_Counter;
    output [3:0] Count_Out;
    reg [3:0] Count_Out;
    always @(posedge clk)
    begin
        if(rst)
        begin
            Count_Out <= 0;
        end
        else
        begin
            if(Recount_Counter)
                Count_Out <= 0;
            else
                Count_Out <= Count_Out + 1;
        end
```



Comparator(Combinational)

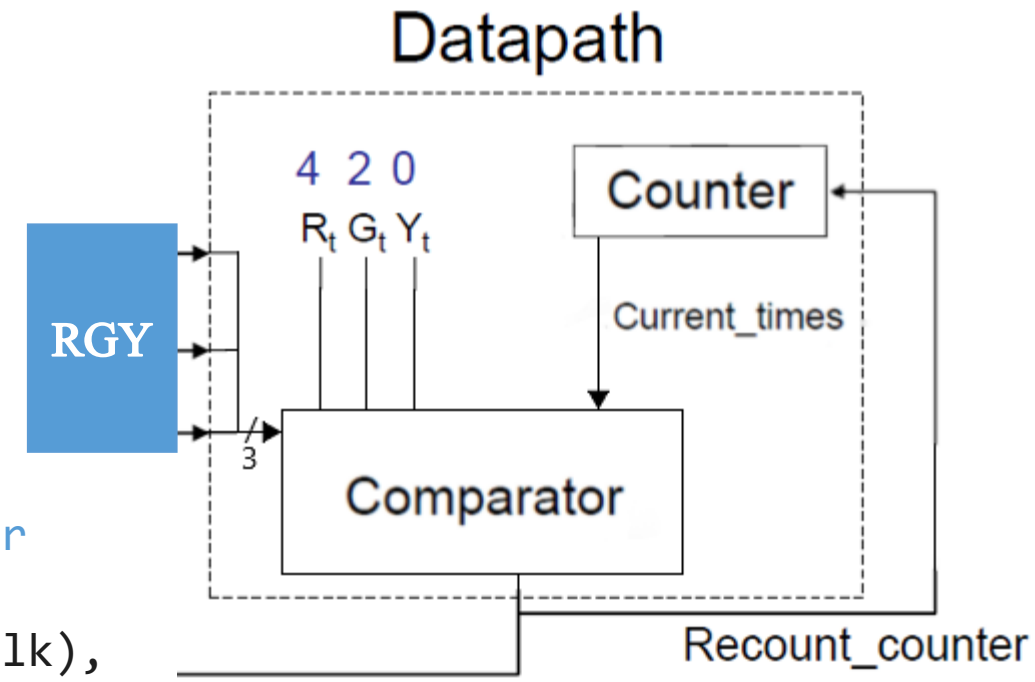



```
module Compare(current_times, RGY, Recount_counter);
    input [2:0] RGY;
    input [3:0] current_times;
    output Recount_counter;
    reg Recount_counter;
    parameter R_times = 4, G_times = 2, Y_times = 0;

    always @(*)
    begin
        case(RGY)
            3'b100:
            begin
                if(current_times == R_times)
                    Recount_counter = 1;
                else
                    Recount_counter = 0;
            end
            3'b001:
            begin
                if(current_times == Y_times)
                    Recount_counter = 1;
                else
                    Recount_counter = 0;
            end
            3'b010:
            begin
                if(current_times == G_times)
                    Recount_counter = 1;
                else
                    Recount_counter = 0;
            end
            default:
                Recount_counter = 1;
        endcase
```

Datapath

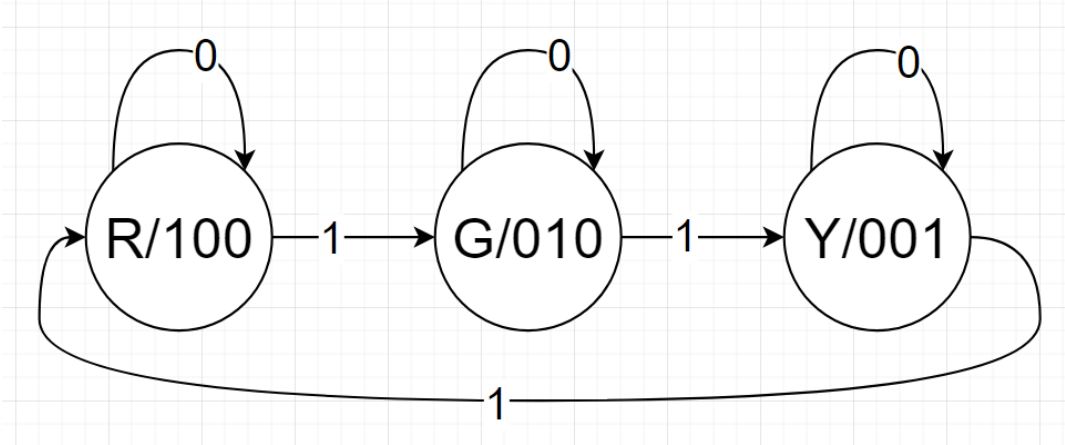
```
module Datapath(clk, rst, RGY, Recount);
    input clk, rst;
    input [2:0] RGY;
    output Recount;
    wire [3:0] Current_times;
    // Module
    Compare compare
    (
        .current_times(Current_times),
        .RGY(RGY),
        .Recount_counter(Recount)
    );
```




```
Counter counter
    (
        .clk(clk),
        .rst(rst),
        .Recount_Counter(Recount),
        .Count_Out(Current_times)
    );
endmodule
```

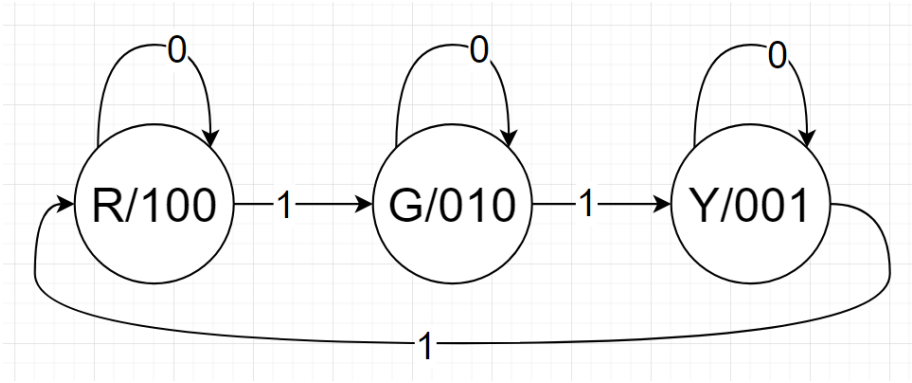
Controller - State Register(Sequential)

```
module Traffic_Control (clk, rst, Recount_Counter,
                        Red, Green, Yellow);
    input clk, rst, Recount_Counter;
    output Red, Green, Yellow;
    reg Red, Green, Yellow;
    reg [1:0] currentstate, nextstate;
    parameter [1:0] Red_Light = 0, Green_Light = 1,
                    Yellow_Light = 2;
    // State Register (Flip-Flops)
    always @(posedge clk)
    begin
        if(rst)
            currentstate <= Red_Light;
        else
            currentstate <= nextstate;
    end
```



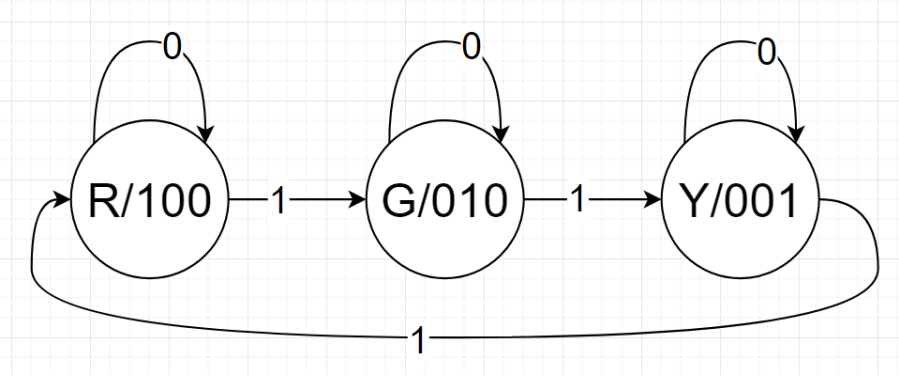
Controller - Next State Logic(Combinational)

```
// Next State Logic
always @(*)
begin
    case(currentstate)
        Red_Light:
        begin
            if(Recount_Counter)
                nextstate = Green_Light;
            else
                nextstate = Red_Light;
        end
        Green_Light:
        begin
            if(Recount_Counter)
                nextstate = Yellow_Light;
            else
                nextstate = Green_Light;
        end
        Yellow_Light:
        begin
            if(Recount_Counter)
                nextstate = Red_Light;
            else
                nextstate = Yellow_Light;
        end
        default:
            nextstate = Red_Light;
    endcase
end
```



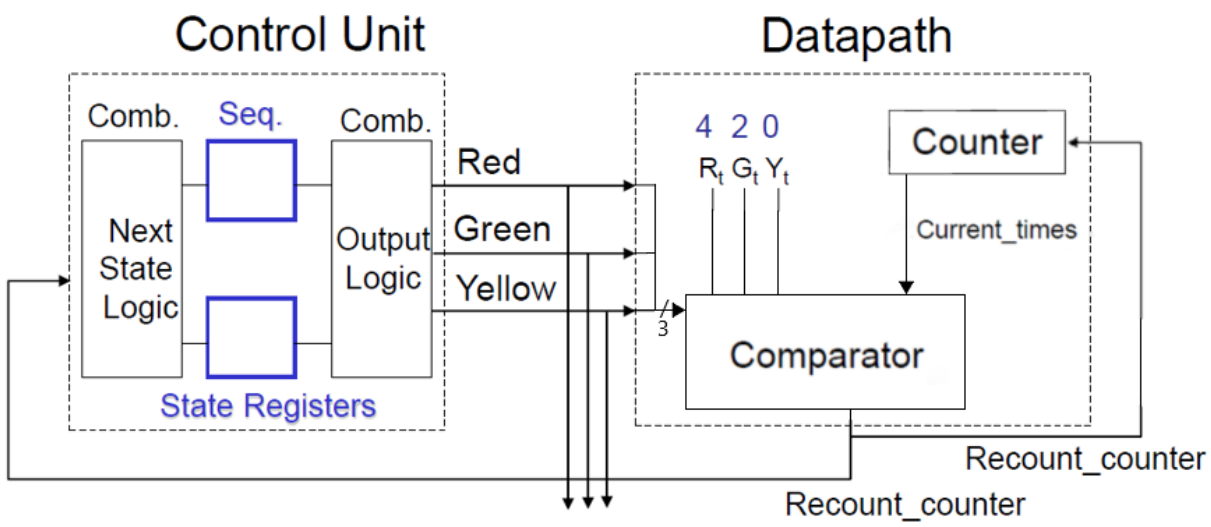
Controller - Output Logic(Combinational)

```
// Output Logic
always @(currentstate)
begin
    case(currentstate)
        Red_Light:
        begin
            Red = 1'b1;
            Green = 1'b0;
            Yellow = 1'b0;
        end
        Green_Light:
        begin
            Red = 1'b0;
            Green = 1'b1;
            Yellow = 1'b0;
        end
        Yellow_Light:
        begin
            Red = 1'b0;
            Green = 1'b0;
            Yellow = 1'b1;
        end
        default:
        begin
            Red = 1'b0;
            Green = 1'b0;
            Yellow = 1'b0;
        end
    endcase
end
```



Traffic Light

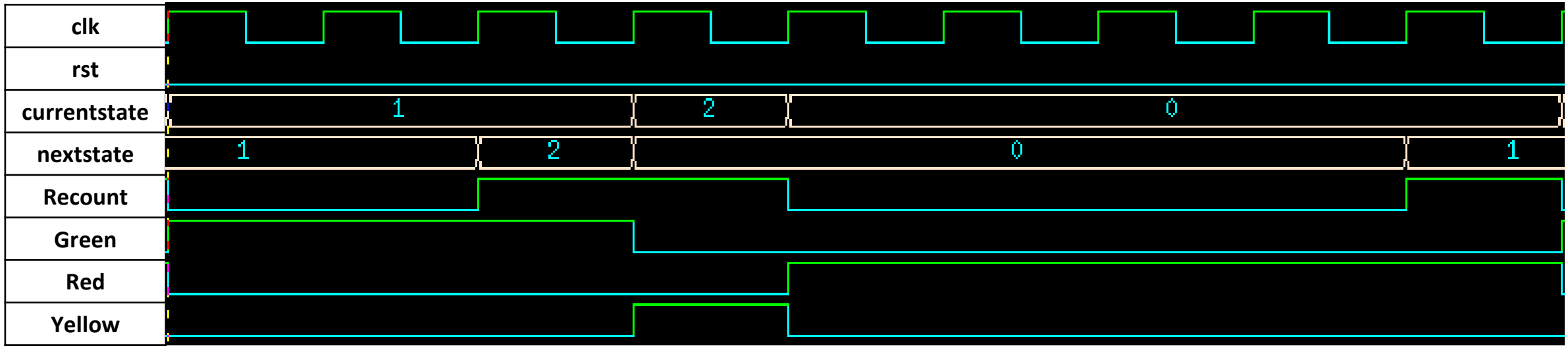
```
module traffic(clk, rst, Red, Green, Yellow);
    input clk, rst;
    output Red, Green, Yellow;
    wire Recount_counter;
    // Module
    Traffic_Control controller
    (
        .clk(clk),
        .rst(rst),
        .Recount_Counter(Recount_counter),
        .Red(Red),
        .Green(Green),
        .Yellow(Yellow)
    );
```



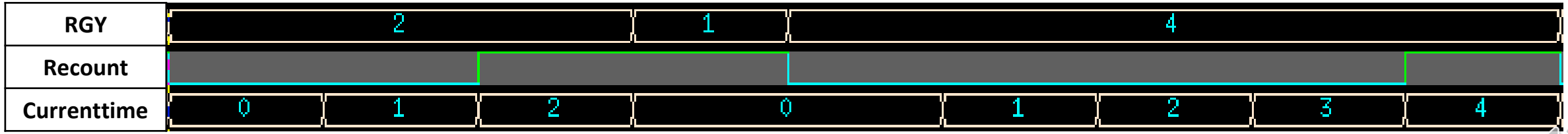

```
    Datapath datapath
    (
        .clk(clk),
        .rst(rst),
        .RGY({Red,Green,Yellow}),
        .Recount(Recount_counter)
    );
endmodule
```

Traffic Light Waveform

- Traffic Light



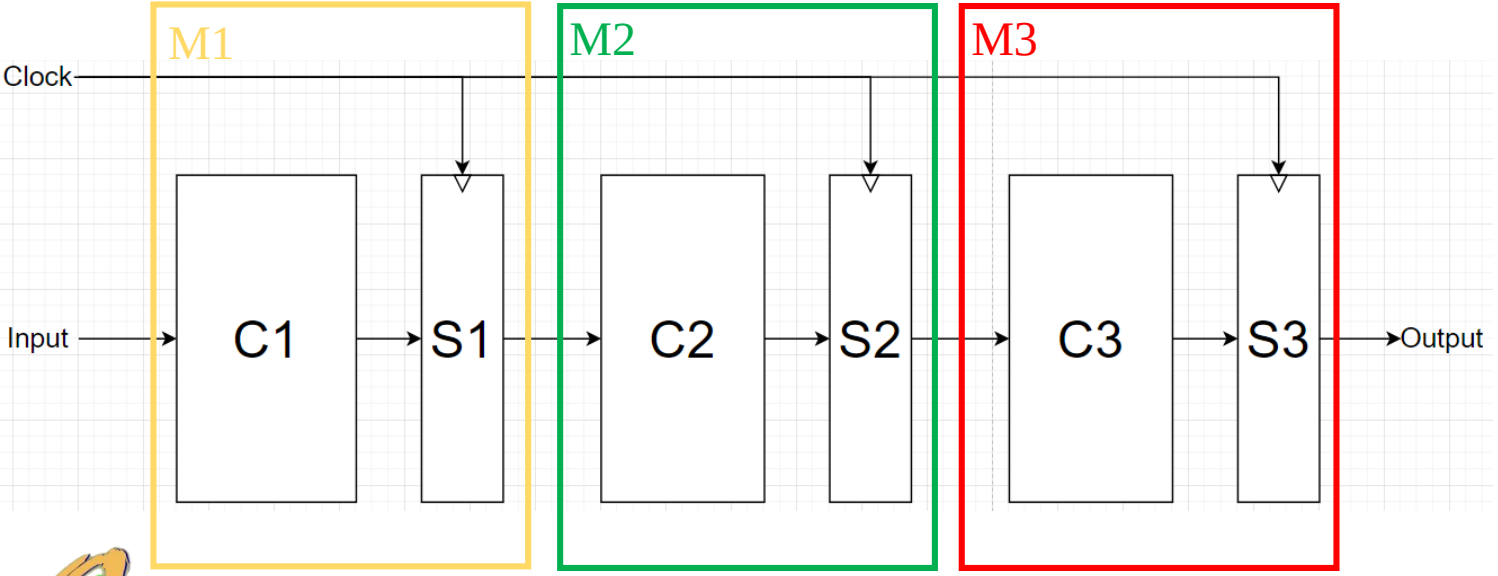
- Comparator

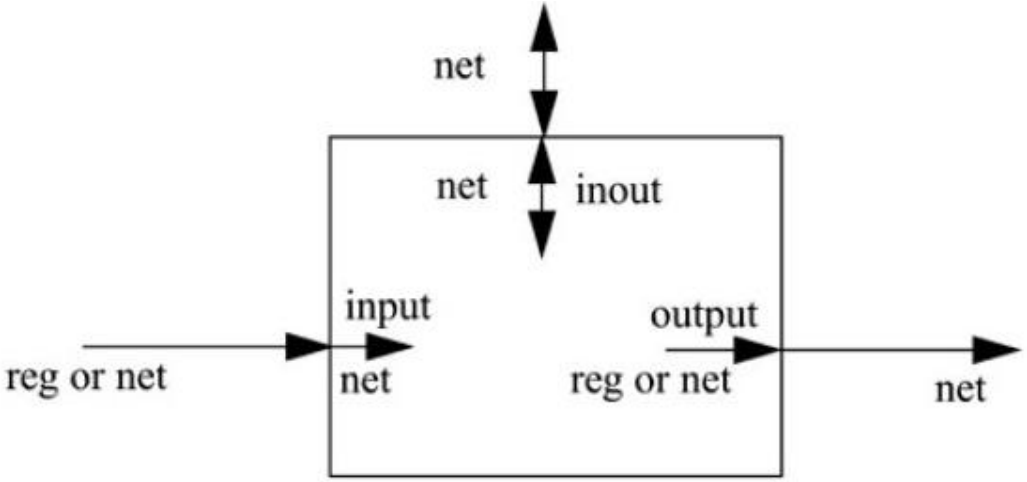


Choosing Correct Data Type

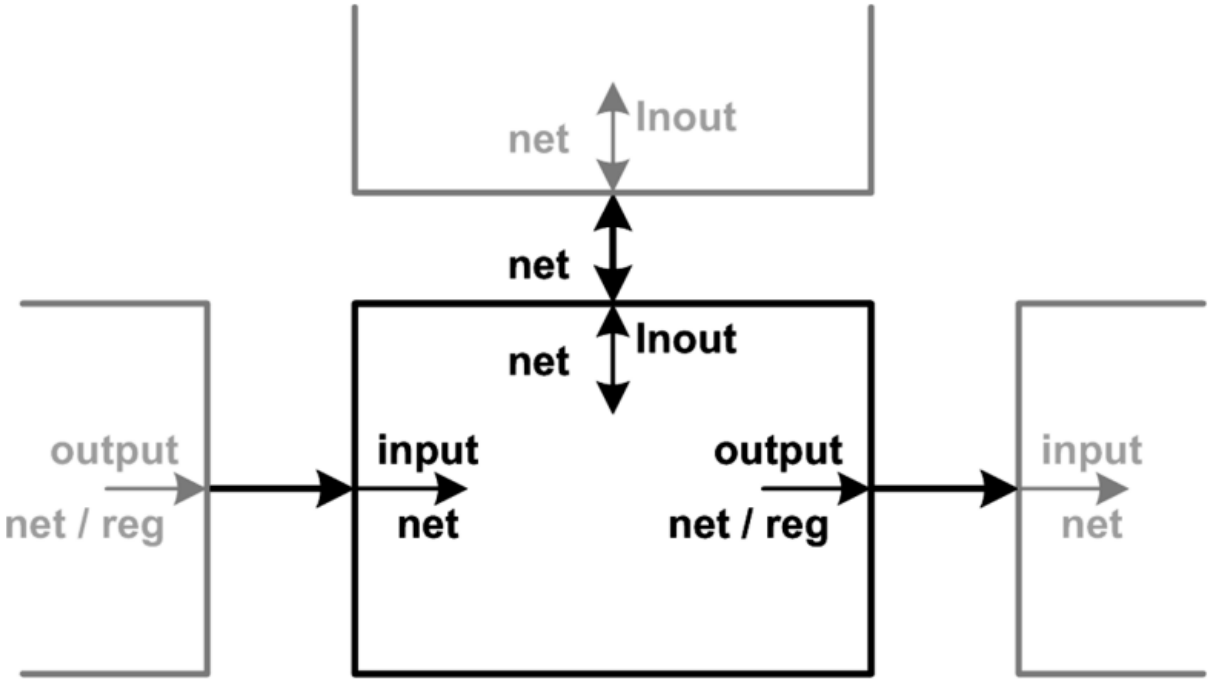
Connection Rule

- Generally, an input port is a net driven by a net or a reg.
- Generally, an output port is a net or a reg, and it drives a net.
- Generally, an inout port is a net, and it drives a net.





Inter-module Connection



Common Mistakes in Data Types

Error Messages	User Errors
Illegal Left-hand-side assignment	When a procedural assignment is made to a net or one forgets to declare a signal as a reg.
Illegal output port specification	Signal connected to an output port of another module is a register.
Gate has illegal output specification	Signal denoted an output of a primitive gate is a register.
Incompatible declaration, (signal) defined as input	An input port of a module is declared as a register.

Reference

Reference

1. Latch & Flip-Flop & Register: https://reurl.cc/em7x2m
2. Combinational vs Sequential: https://reurl.cc/OkL1l7
3. Blocking vs Non-Blocking: https://reurl.cc/GbLVMG
4. Finite-State Machine(FSM): https://reurl.cc/ye4gyM
5. Verilog HDL Design: https://pse.is/3rypr5

Thank You