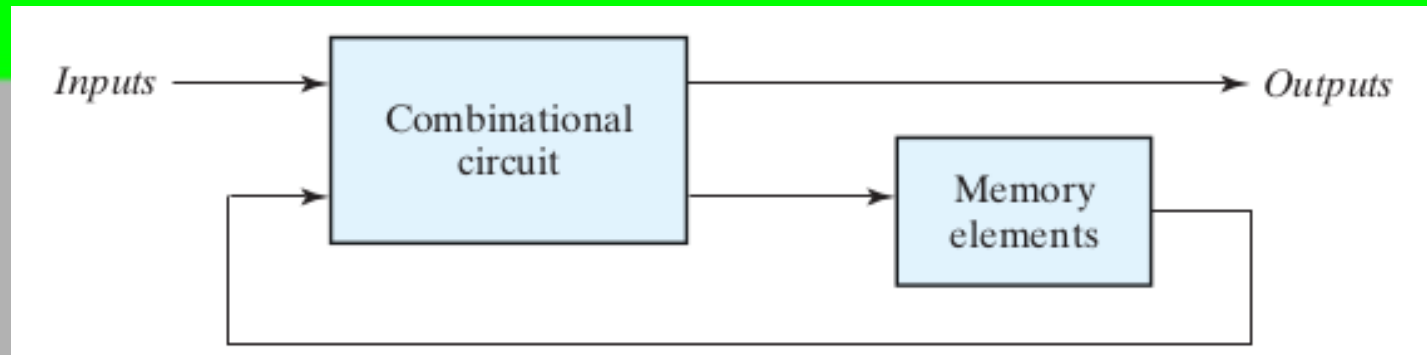
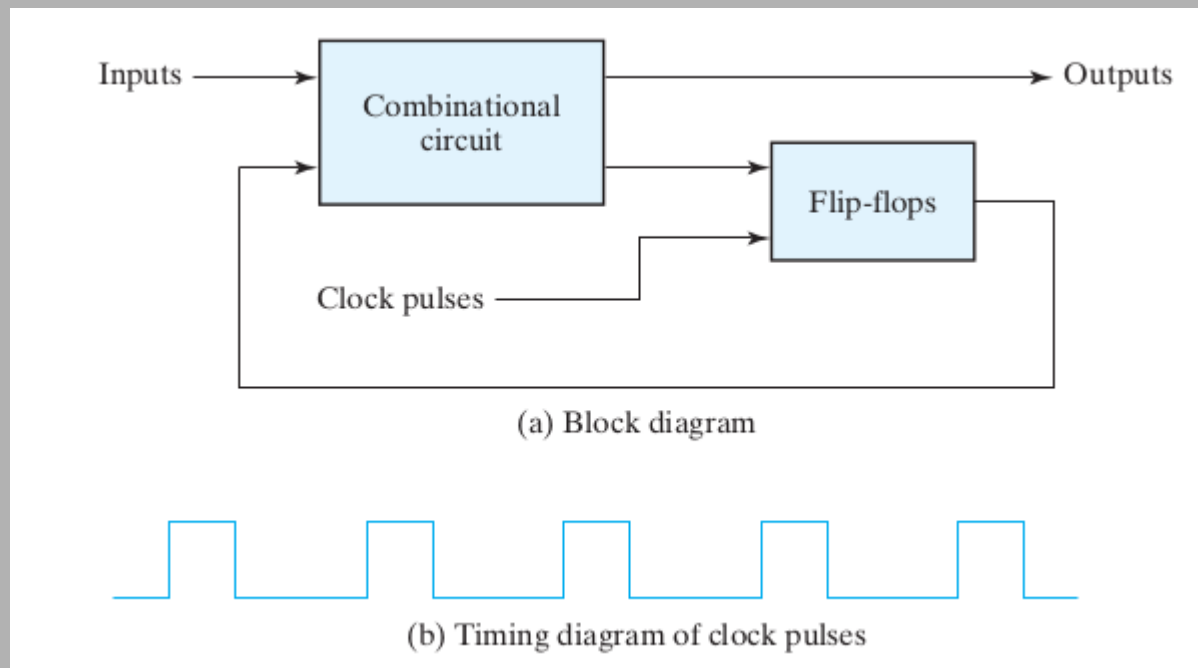


A sequential circuit is specified by a time sequence of inputs, outputs, and internal states



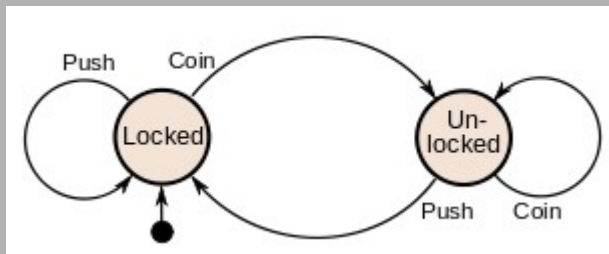
Synchronous clocked sequential circuit



Finite-state machine

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, 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.

Examples are vending machines, which dispense products when the proper combination of coins is deposited, elevators, whose sequence of stops is determined by the floors requested by riders, traffic lights, which change sequence when cars are waiting, and combination locks, which require the input of combination numbers in the proper order.



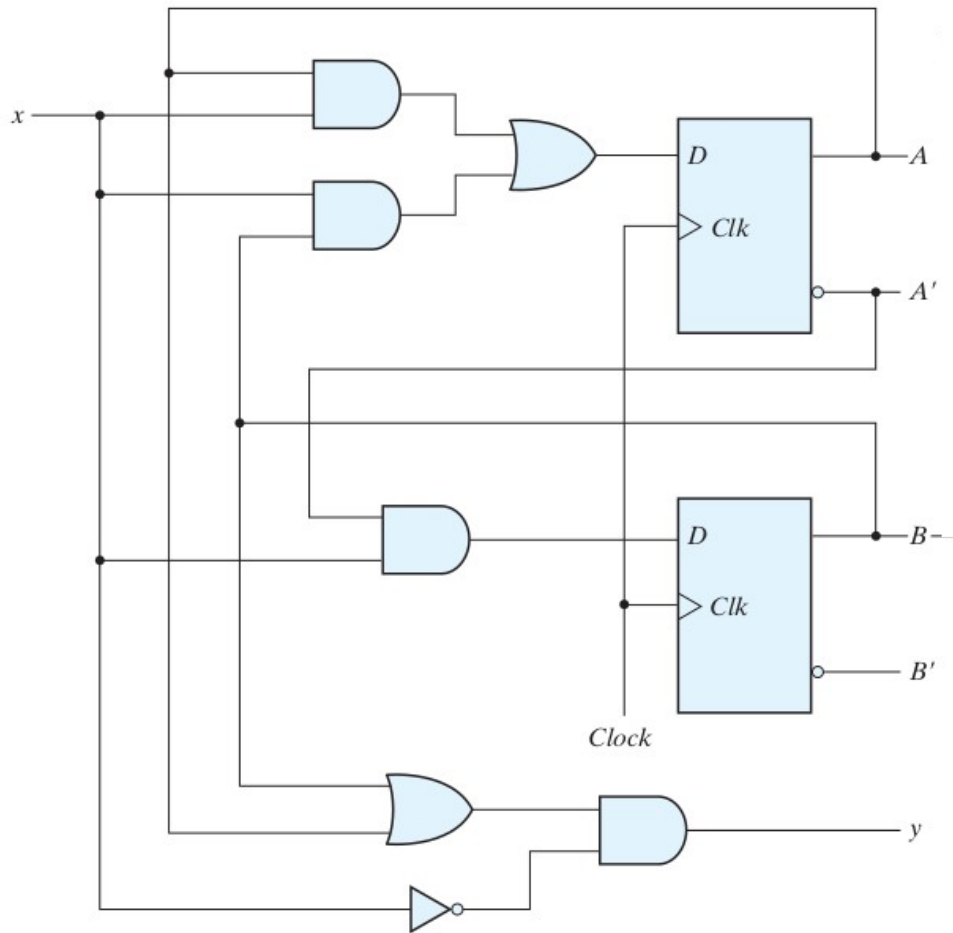
A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received.

Real computers are finite state machines: they have finite memory so there's a finite number of states the machine can be in.

Common Examples of Synchronous FSM

- Up and Down Binary Counters
- Shift Registers
- Sequence Detectors
- Controllers

State Equations



$$A(t + 1) = A(t)x(t) + B(t)x(t)$$
$$B(t + 1) = A'(t)x(t)$$

$$A(t + 1) = Ax + Bx$$
$$B(t + 1) = A'x$$

A state equation (also called a transition equation) specifies the next state as a function of the present state and inputs.

$$y = (A + B)x'$$

Present State		Input	Next State		Output
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The time sequence of inputs, outputs, and flip-flop states can be enumerated in a state table (sometimes called a transition table).

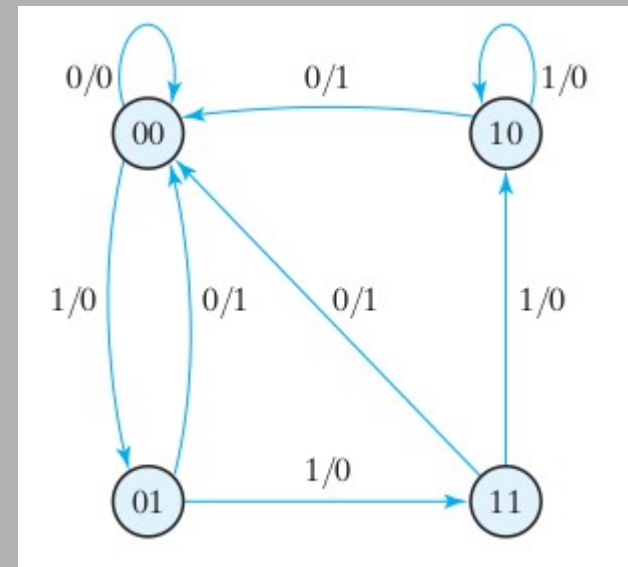
The derivation of a state table requires listing all possible binary combinations of present states and inputs. In this case, we have eight binary combinations from 000 to 111. The next-state values are then determined from the logic diagram or from the state equations.

In general, a sequential circuit with m flip-flops and n inputs needs 2^{m+n} rows in the state table. The binary numbers from 0 through $2^{m+n}-1$ are listed under the present-state and input columns. The next-state section has m columns, one for each flip-flop. The binary values for the next state are derived directly from the state equations. The output section has as many columns as there are output variables. Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

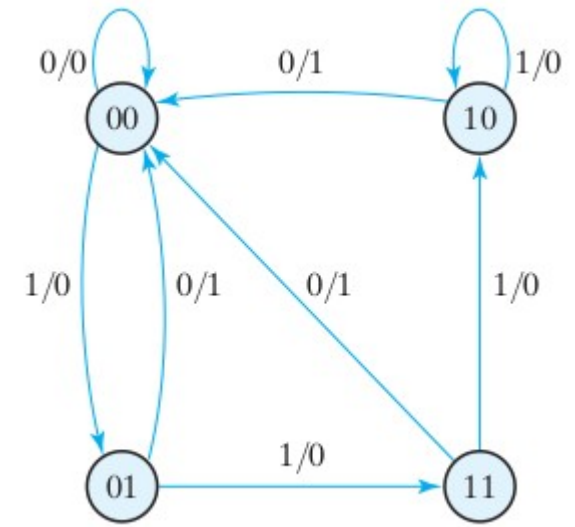
The binary number inside each circle identifies the state of the flip-flops. The directed lines are labelled with two binary numbers separated by a slash.

The input value during the present state is labelled first, and the number after the slash gives the output during the present state with the given input.

Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>y</i>	<i>y</i>
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0



The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation.



It is important to remember that the bit value listed for the output along the directed line occurs during the present state and with the indicated input, and has nothing to do with the transition to the next state.

For example, the directed line from state 00 to 01 is labelled 1/0, meaning that if the sequential circuit is in the present state 00 and the input is 1, the output is 0. After the next clock edge, the circuit goes to the next state, 01. If the input changed to 0, then the output becomes 1, but if the input remained at 1, the output stays at 0. This information is obtained from the state diagram along the two directed lines emanating from the circle with state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

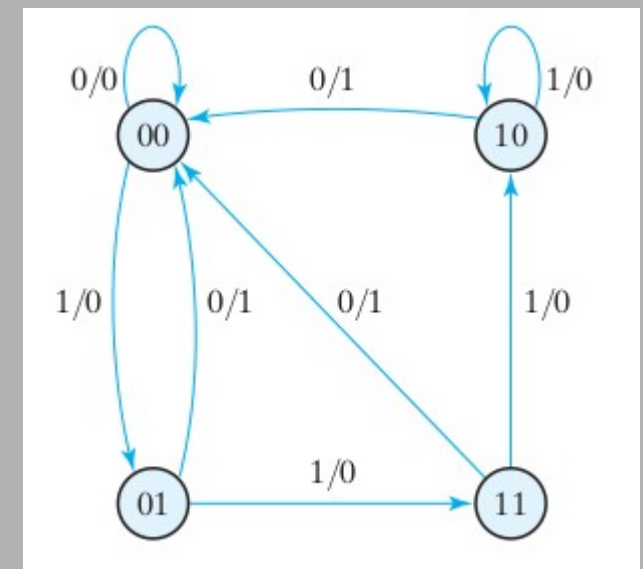
The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation.

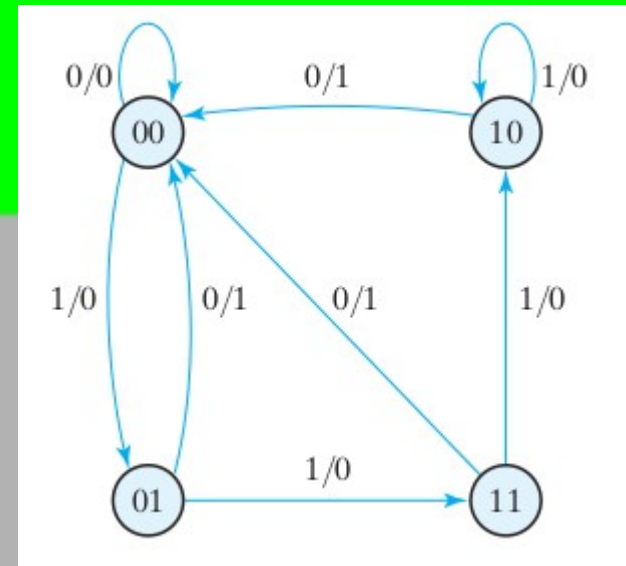
The state diagram of clearly shows that, starting from state 00, the output is 0 as long as the input stays at 1.

The first 0 input after a string of 1's gives an output of 1 and transfers the circuit back to the initial state, 00.

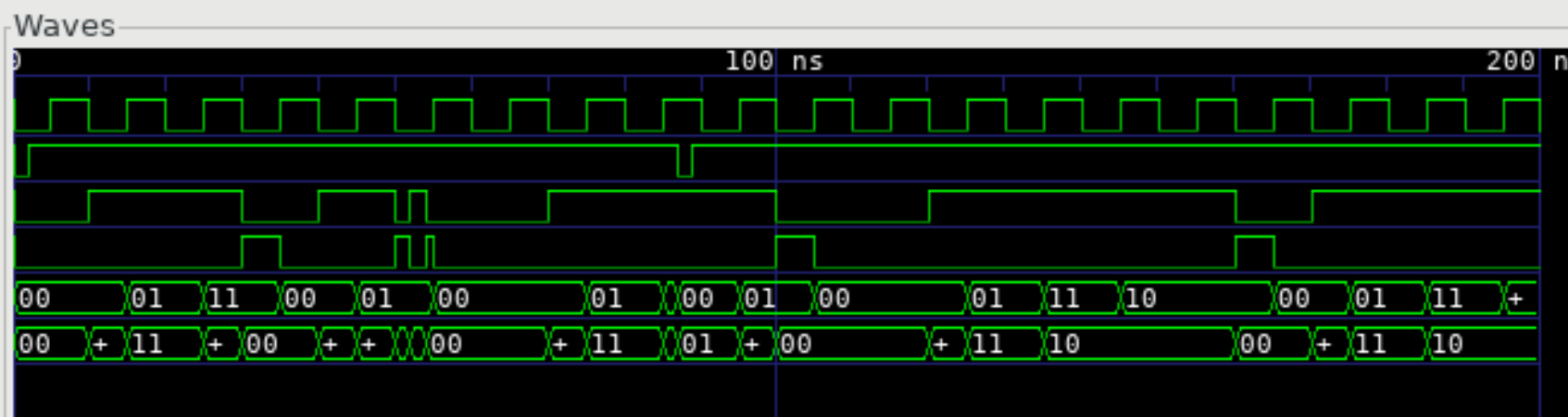
The machine represented by this state diagram acts to detect a zero in the bit stream of data.

- 00 zero was detected in last cycle, or reset, wait for a 1
 - if $x=0$ or 1 output is 0, if x stays at zero stay in this state
 - if x goes to 1 next state 01
- 01 last cycle was a 1 and last output a zero
 - if x is 0, output a 1, new zero detected, next 00
 - if x is 1, output a zero next state 11
- 11 two 1's were found
 - if x is 0 output a 1 next state 00
 - if x is 1 output 0 next 10
- 10 three or more ones were found
 - if x is 1 output a zero and stay here
 - if x is zero, output a 1 next state 00





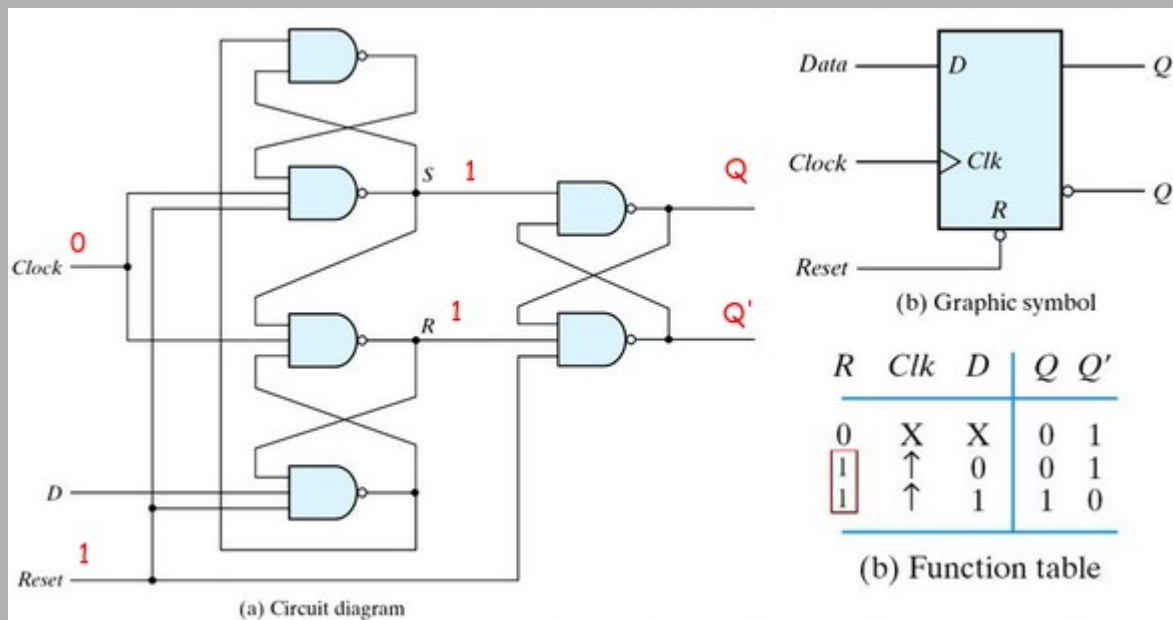
Signals	
Time	
t_clock	
t_reset	
t_x_in	
t_y_out	
state[1:0]	
next_state[1:0]	



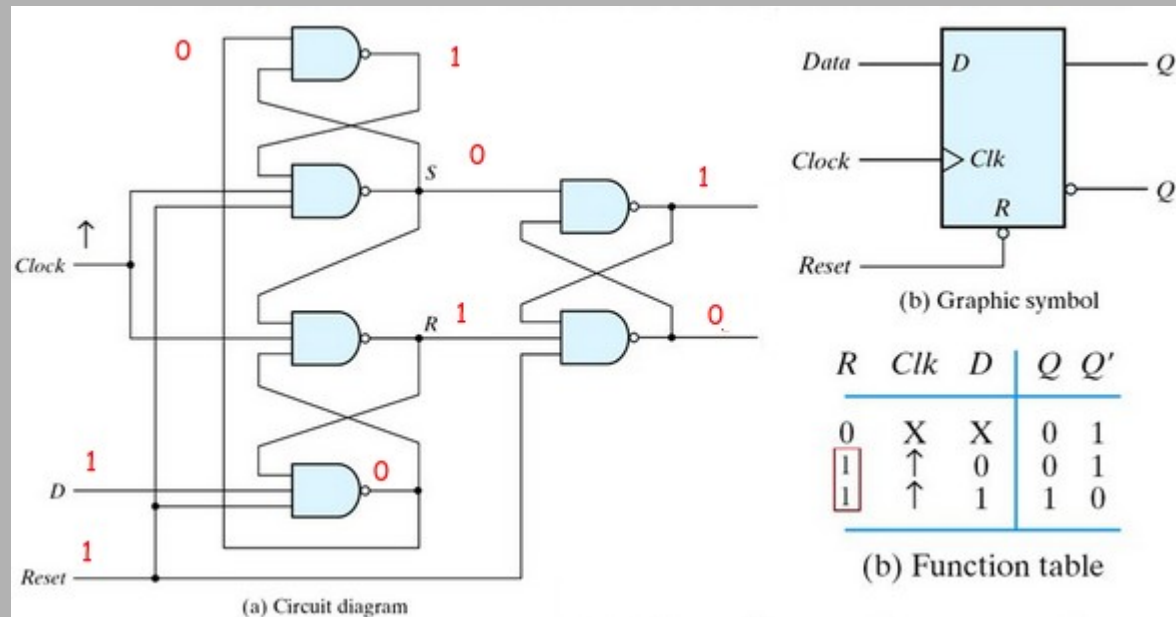
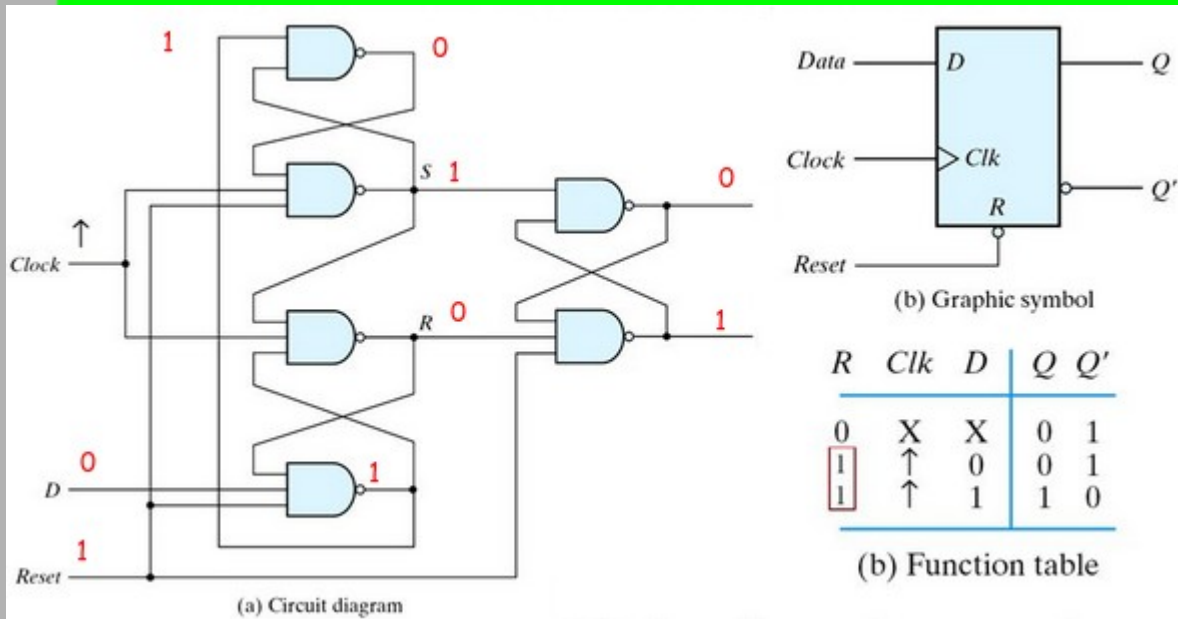
D-Type Flip-Flop with Set/Reset

Set/reset events are asynchronous to the clock edge.

The edge-triggered D flip-flop uses three SR latches. Two latches respond to the external D (data) and Clk (clock) inputs. The third latch provides the outputs for the flip-flop. The S and R inputs of the output latch are maintained at the logic-1 level when Clk = 0.



If $D = 0$ when Clk becomes 1, R changes to 0. This causes the flip-flop to go to the reset state, making $Q = 0$. If there is a change in the D input while $\text{Clk} = 1$, terminal R remains at 0 because Q is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input.



```

`timescale 1ns/100ps
`default_nettype none

module Mealy_Zero_Detector (
    output reg y_out,
    input x_in, clock, reset
);
reg [1: 0] state, next_state;

parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

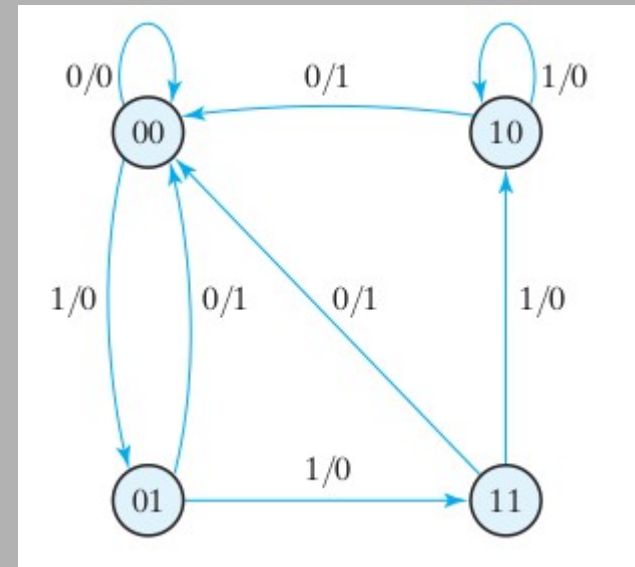
always @ ( posedge clock, negedge reset )
    if (reset == 0) state <= S0;
    else state <= next_state;

always @ (state, x_in)
// Form the next state
    case (state)
        S0: if (x_in) next_state = S1; else next_state = S0;
        S1: if (x_in) next_state = S3; else next_state = S0;
        S2: if (~x_in) next_state = S0; else next_state = S2;
        S3: if (x_in) next_state = S2; else next_state = S0;
    endcase

always @ (state, x_in)
    case (state)
        S0: y_out = 0;
        S1, S2, S3: y_out = ~x_in;
    endcase
endmodule


```

Mealy



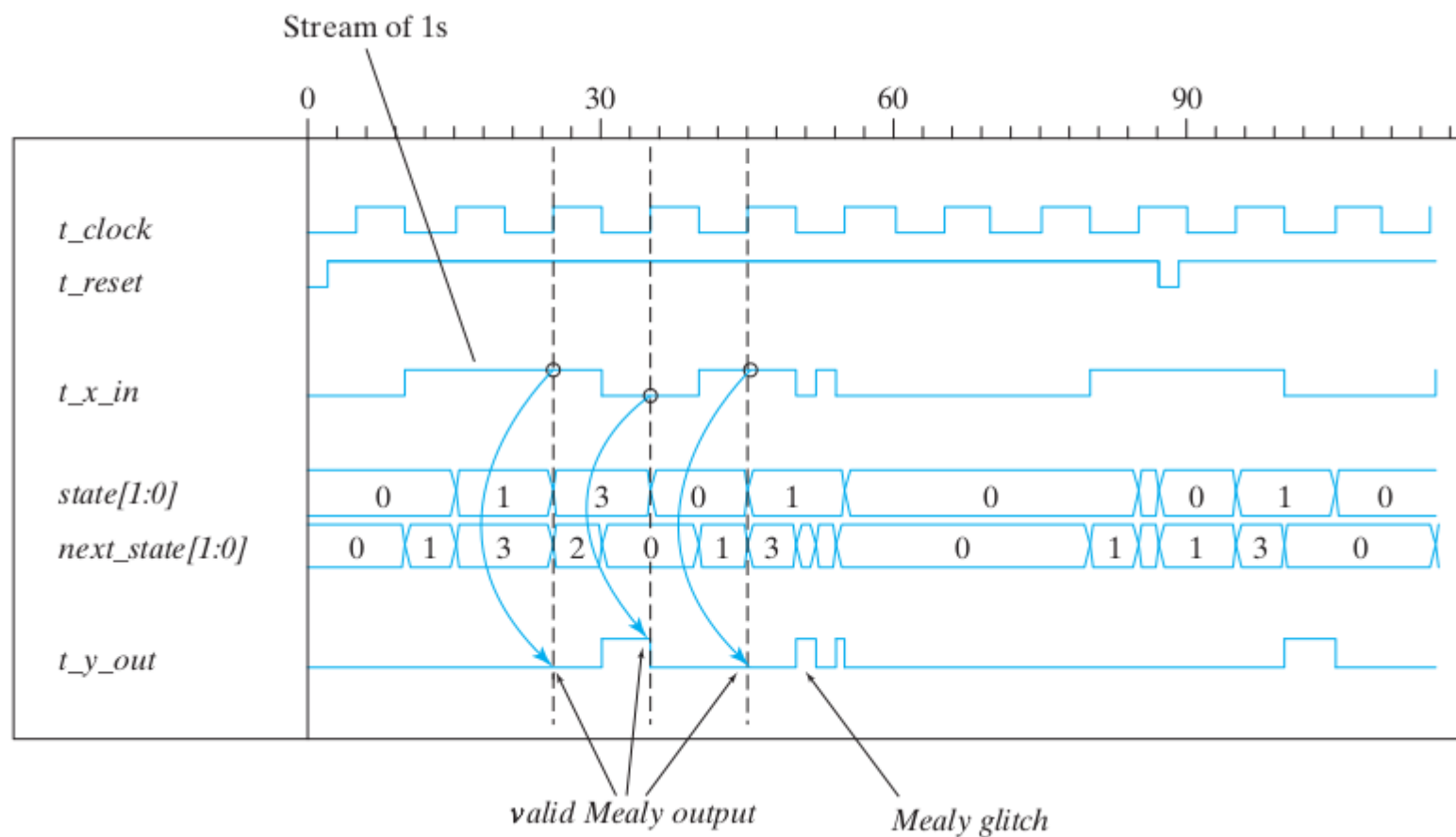
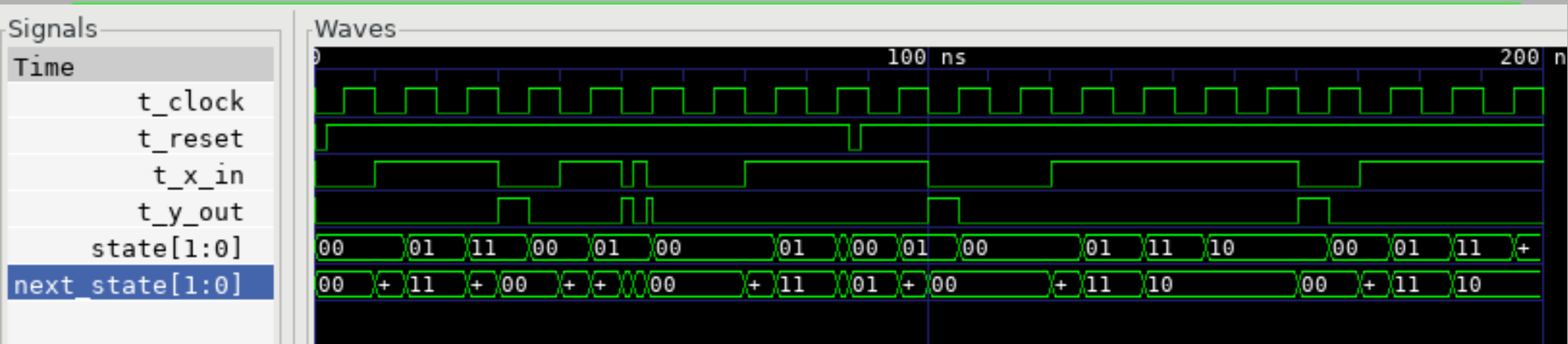
At every rising edge of clock, if reset is not asserted, the state of the machine is updated by the first always block;

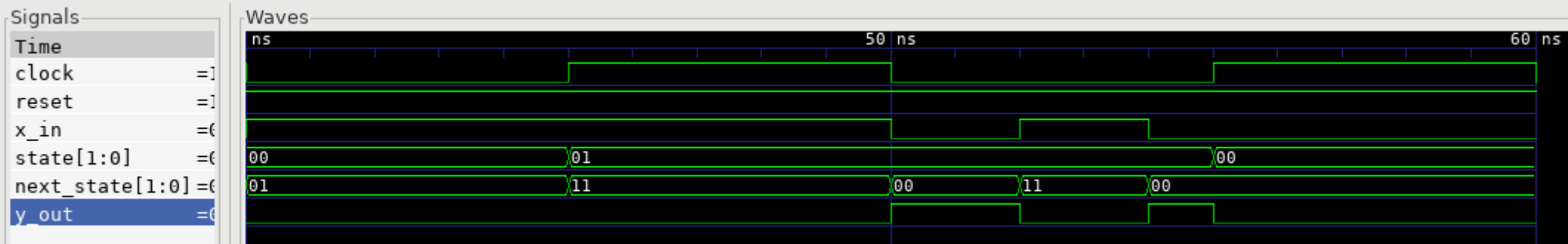
when state is updated by the first always block, the change in state is detected by the sensitivity list mechanism of the second always block; then the second always block updates the value of next_state (it will be used by the first always block at the next tick of the clock); the third always block also detects the change in state and updates the value of the output. In addition, the second and third always blocks detect changes in x_in and update next_state and y_out accordingly.



Signal transitions that are caused by input signals that change on the active edge of the clock race with the clock itself to reach the affected flip-flops, and the outcome is indeterminate (unpredictable).

Conversely, changes caused by inputs that are synchronized to the inactive edge of the clock reach stability before the active edge, with predictable outputs of the flip-flops that are affected by the inputs.

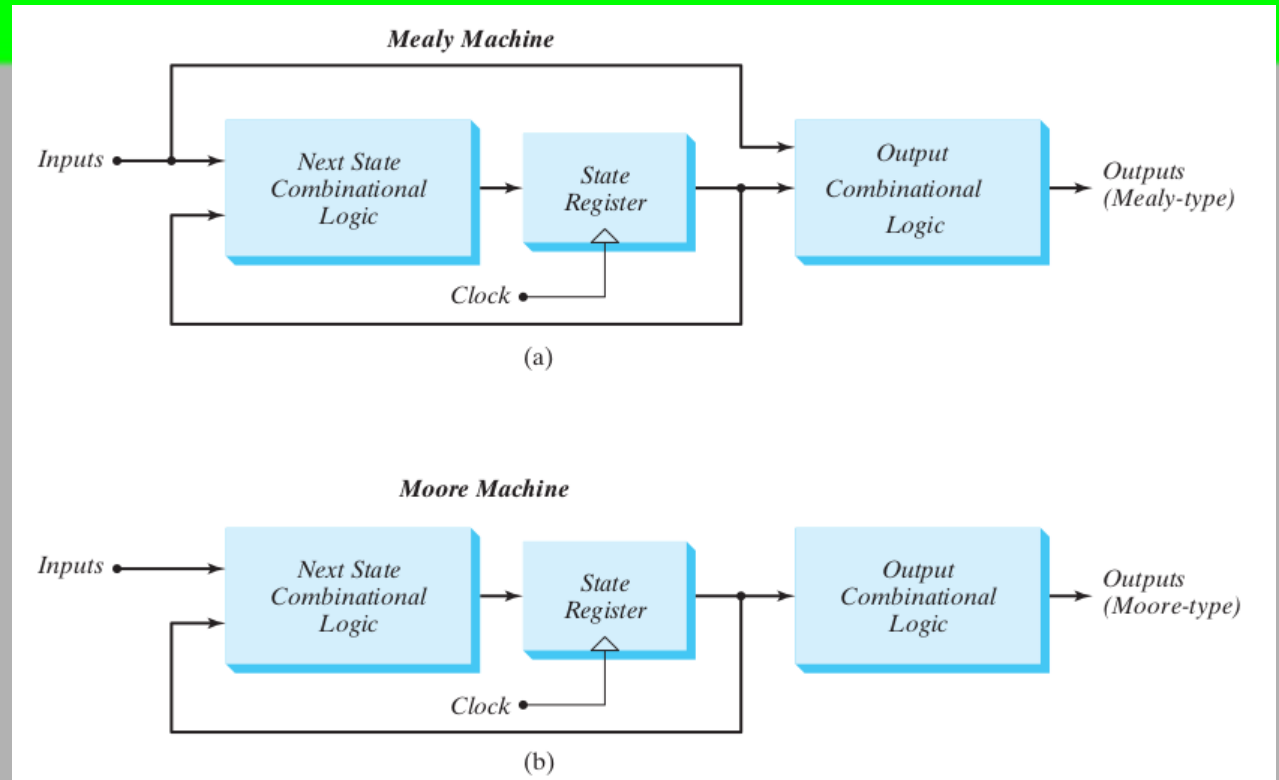




```
initial #200 $finish ;
initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

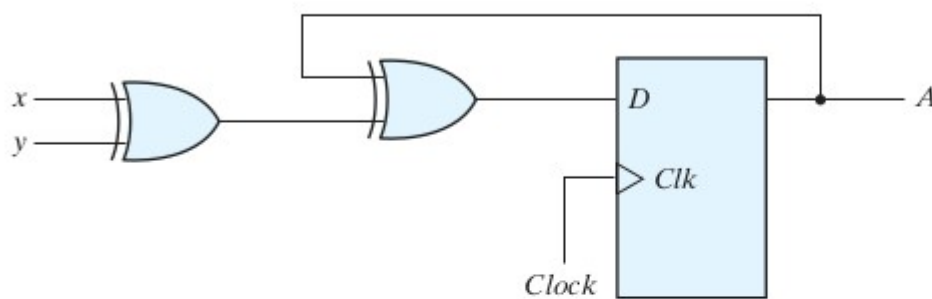
initial fork
// Statements with the fork . . . join block execute in parallel,
// so the time delays are relative to a common reference of t = 0
t_reset = 0;
t_x_in = 0;
#2 t_reset = 1;
#87 t_reset = 0;
#89 t_reset = 1;
#10 t_x_in = 1;
#30 t_x_in = 0;
#40 t_x_in = 1;
#50 t_x_in = 0;
#52 t_x_in = 1;
#54 t_x_in = 0;
#70 t_x_in = 1;
#100 t_x_in = 0;
#120 t_x_in = 1;
#160 t_x_in = 0;
#170 t_x_in = 1;
join
```

Mealy and Moore Models of Finite State Machines



In the theory of computation, a Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs. This is in contrast to a Moore machine, whose output values are determined solely by its current state.

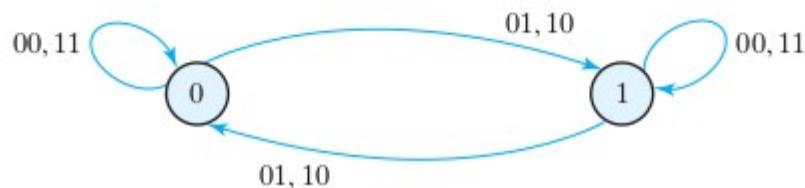
We will adopt the convention of using the flip-flop input symbol to denote the input equation variable and a subscript to designate the name of the flip-flop output. For example, $D_Q = x + y$ specifies an OR gate with inputs x and y connected to the D input of a flip-flop whose output is labelled with the symbol Q .



(a) Circuit diagram

Present state	Inputs		Next state
A	x	y	A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(b) State table

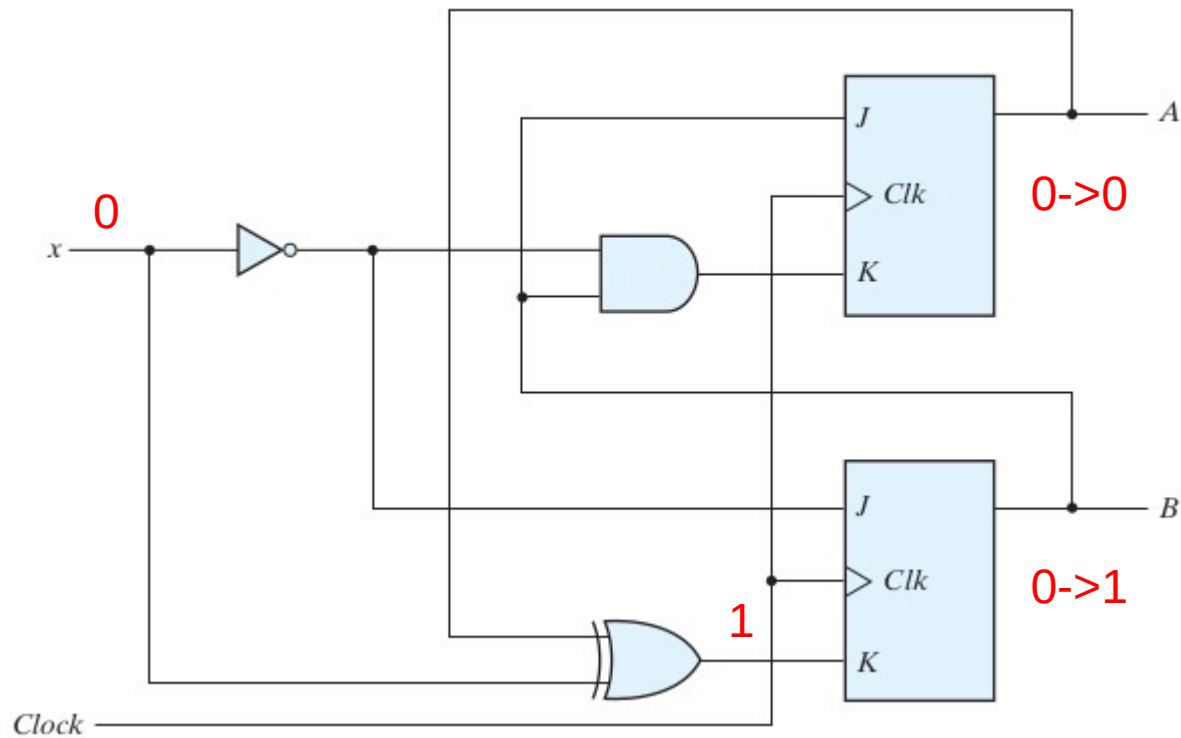


(c) State diagram

$$D_A = A \oplus x \oplus y$$

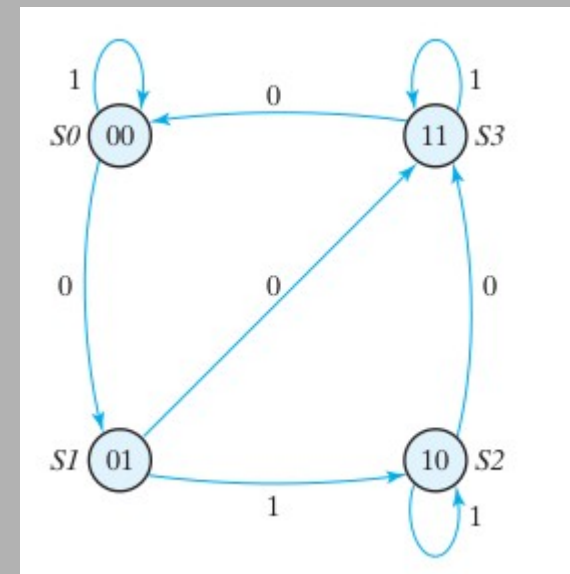
$$A(t + 1) = A \oplus x \oplus y$$

The expression specifies an odd function and is equal to 1 when only one variable is 1 or when all three variables are 1.



$$J_A = B \quad K_A = Bx'$$

$$J_B = x' \quad K_B = A'x + Ax' = A \oplus x$$



State Table for Sequential Circuit with JK Flip-Flops

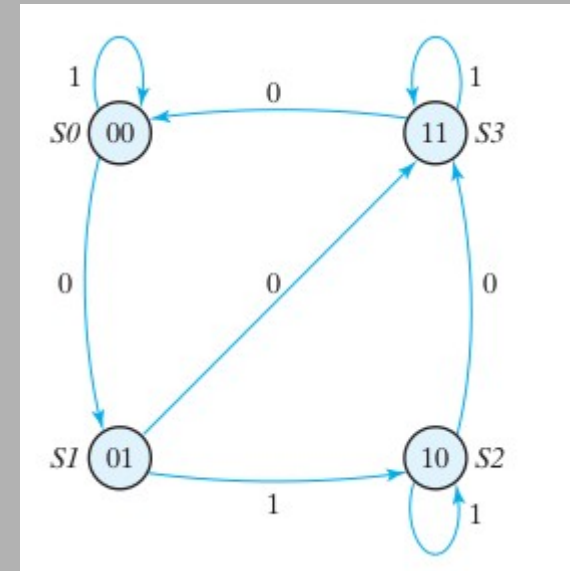
Present State		Input x	Next State		Flip-Flop Inputs			
A	B		A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

A slash on the directed lines is not needed, because there is no output from a combinational circuit.

```

reg [1: 0] state;
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
always @ ( posedge clock, negedge reset)
    if (reset == 0) state <= S0; // Initialize to state S0
    else case (state)
        S0: if (~x_in) state <= S1; else state <= S0;
        S1: if (x_in) state <= S2; else state <= S3;
        S2: if (~x_in) state <= S3; else state <= S2;
        S3: if (~x_in) state <= S0; else state <= S3;
    endcase
    assign y_out = state;
endmodule

```



- (1) the output depends on only the state,
- (2) reset “on-the-fly” forces the state of the machine back to S0 (00)

