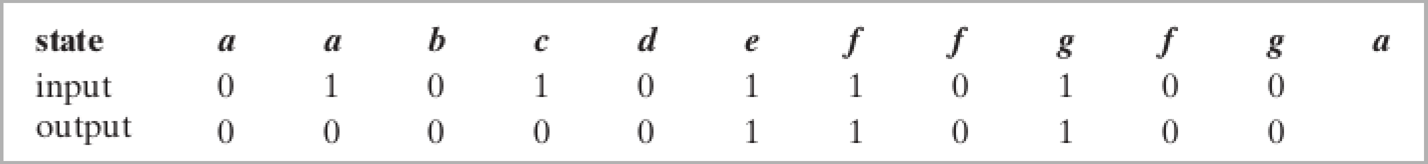
State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the state-reduction problem.

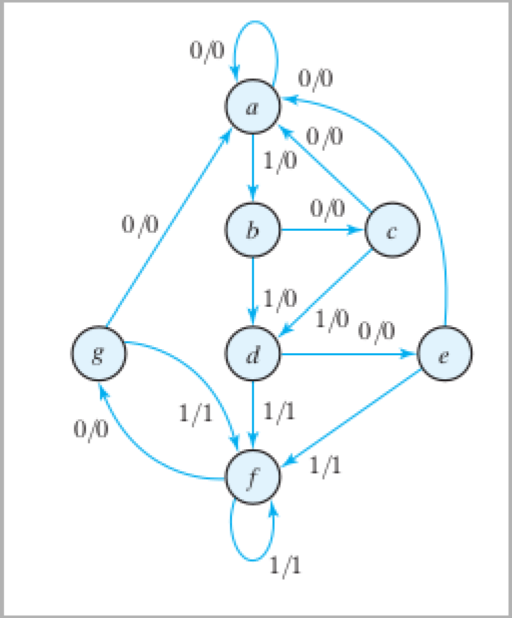
State-reduction algorithms are concerned with procedures for
reducing the number of states in a state table, while keeping the external input–output requirements unchanged.

Since m flip-flops produce 2^m states, a reduction in the number
of states may (or may not) result in a reduction in the number of flip-flops

An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.



state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	



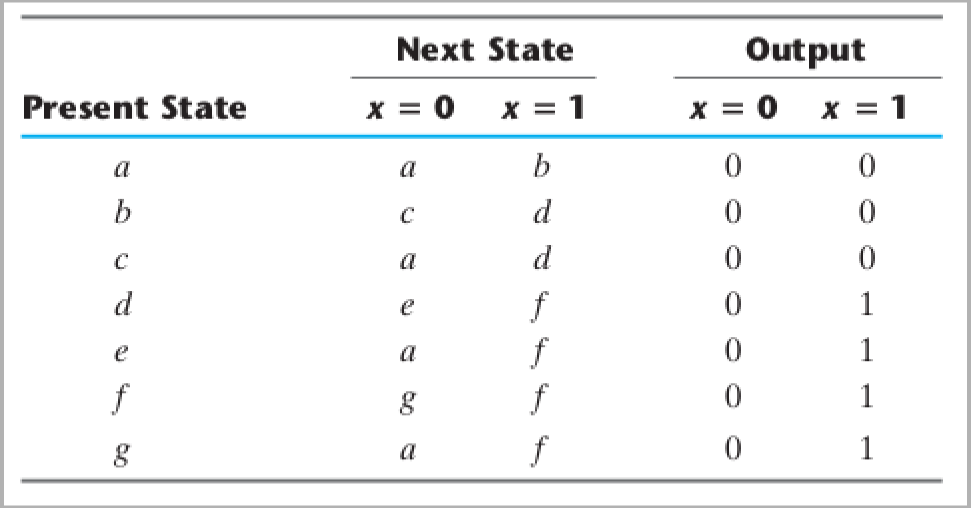
In each column, we have the present state, input value, and output value.

The next state is written on top of the next column.

It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.

When two states are equivalent, one of them can be removed without altering the input–output relationships.

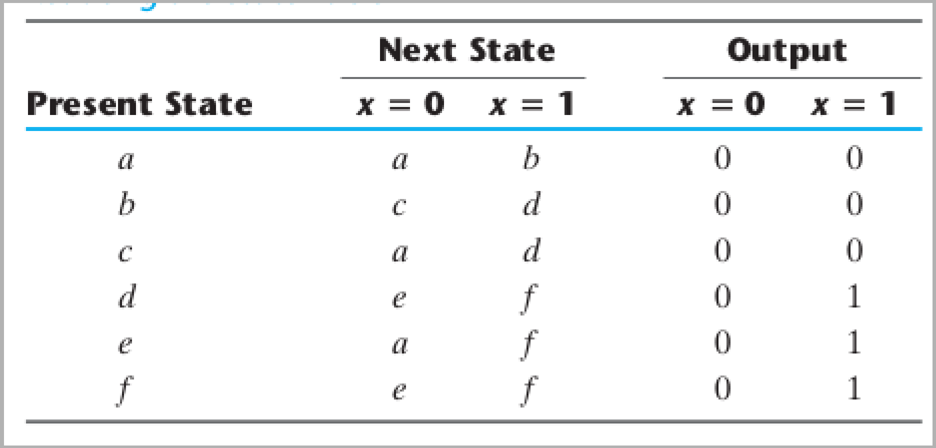


	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

look for two present states that go to the same next state and have the same output for both input combinations.
States e and g are two such states: They both go to states a and f and have outputs of 0 and 1 for x = 0 and x = 1, respectively.

Therefore, states g and e are equivalent, and one of these states can be removed.

The row with present state g is removed, and state g is replaced by state e each time it occurs in the columns headed “Next State.”

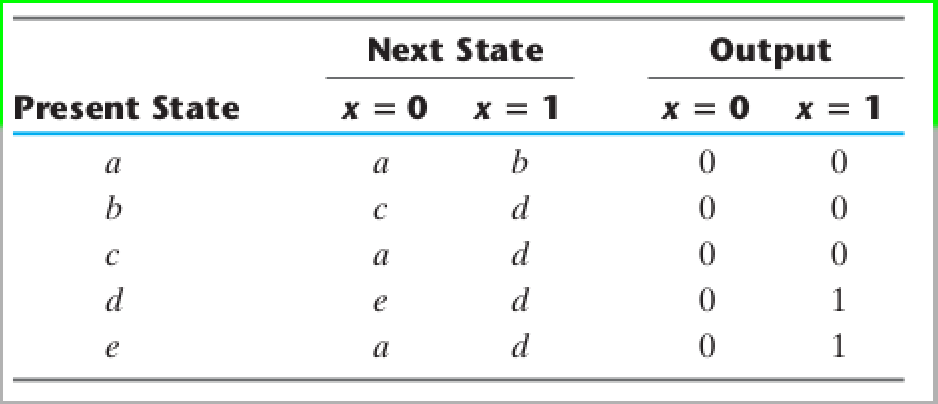


Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

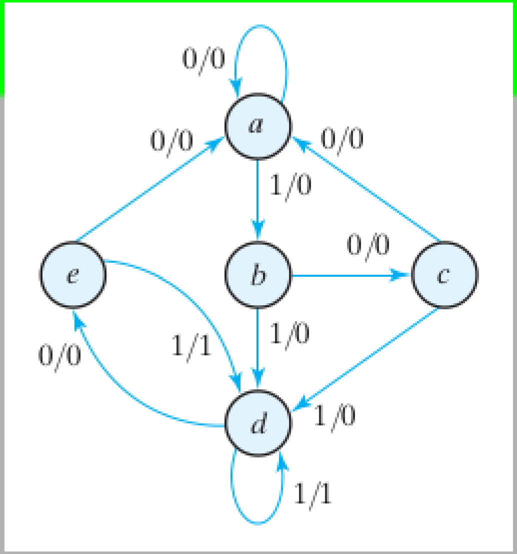
Present state f now has next states e and f and outputs 0 and 1 for x = 0 and x = 1, respectively.

The same next states and outputs appear in the row with present state d

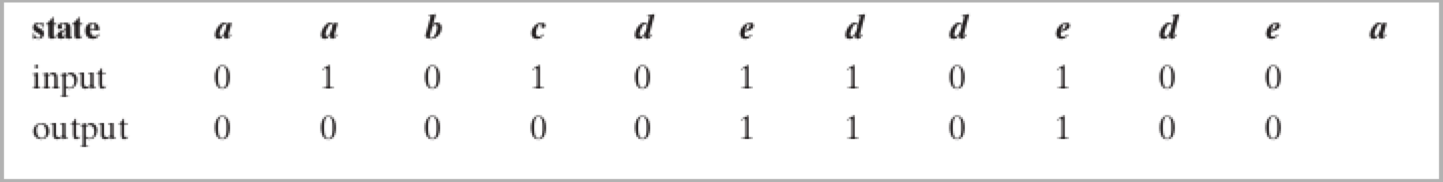
Therefore, states f and d are equivalent, and state f can be removed and replaced by d



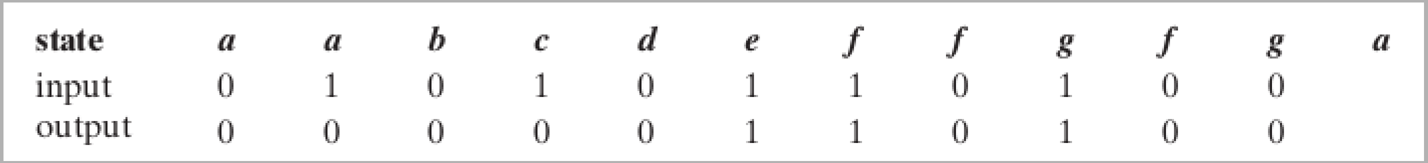
Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1



This state diagram satisfies the original input–output specifications and will produce the required output sequence for any given input sequence.



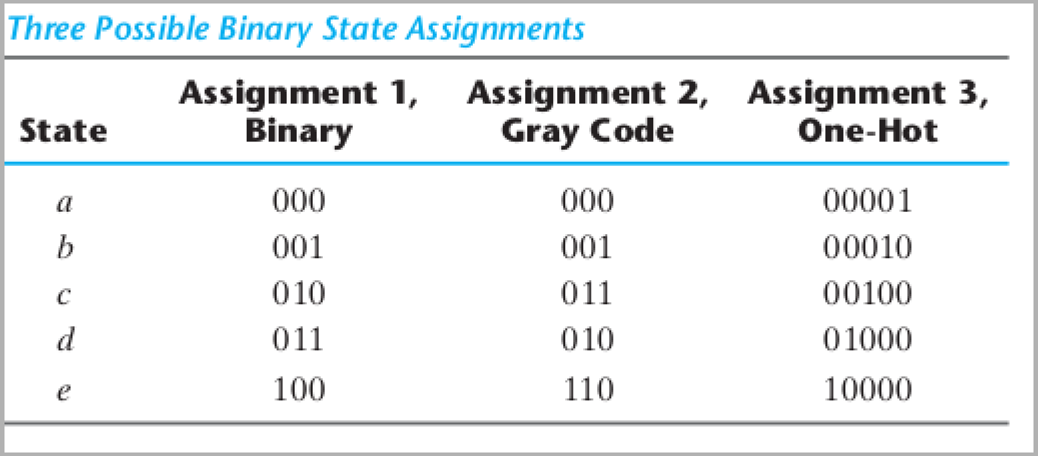
state	*a*	*a*	*b*	*c*	*d*	*e*	*d*	*d*	*e*	*d*	*e*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	



state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Note that the same output sequence results, although the state sequence is different

State Assignment



Three Possible Binary State Assignments

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

The simplest way to code five states is to use the first five integers in binary counting order
Another similar assignment is the Gray code shown in assignment 2. Here, only one bit in the code group changes when going from one number to the next.

This code makes it easier for the Boolean functions to be placed in the map for simplification.

Another possible assignment often used in the design of state machines to control data-path units is the one-hot assignment.

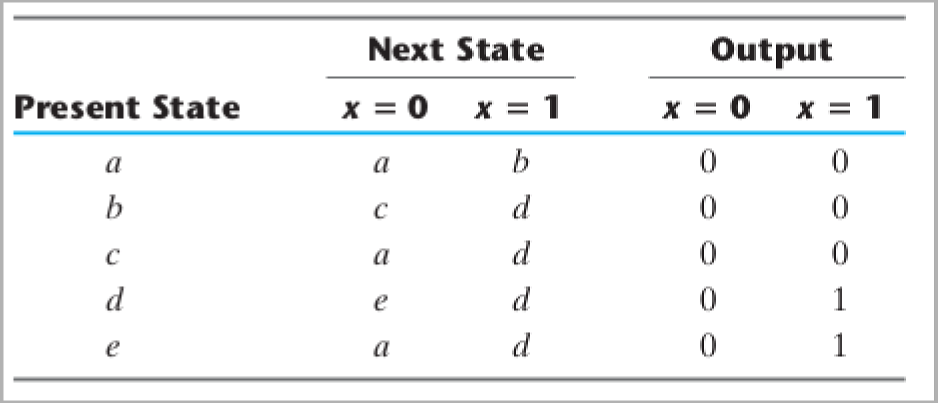
This configuration uses as many bits as there are states in the circuit.

At any given time, only one bit is equal to 1 while all others are kept at 0.

This type of assignment uses one flip-flop per state

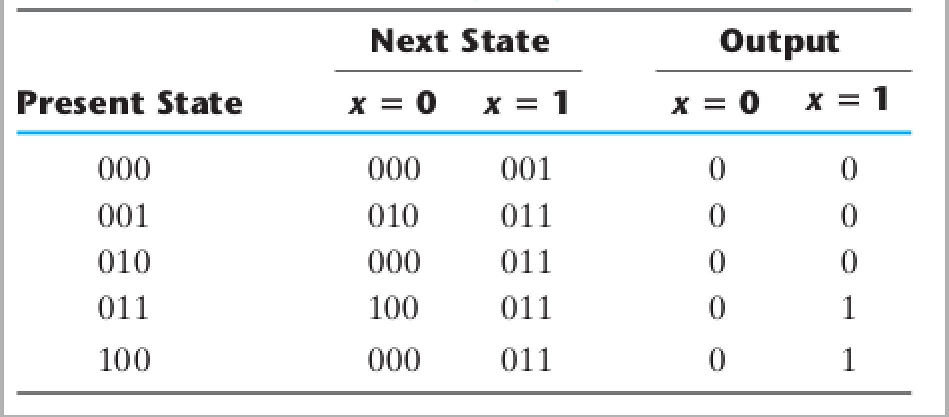
One-hot encoding usually leads to simpler decoding logic for the next state
and output.

One-hot machines can be faster than machines with sequential binary encoding, and the silicon area required by the extra flip-flops can be offset by the area saved by using simpler decoding logic



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

Reduced State Table with Binary Assignment 1



Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

Design procedure

The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained.

In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification.

The first step in the design of sequential circuits is to obtain a state table or an equivalent representation, such as a state diagram.

1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.

2. Reduce the number of states if necessary.

3. Assign binary values to the states.

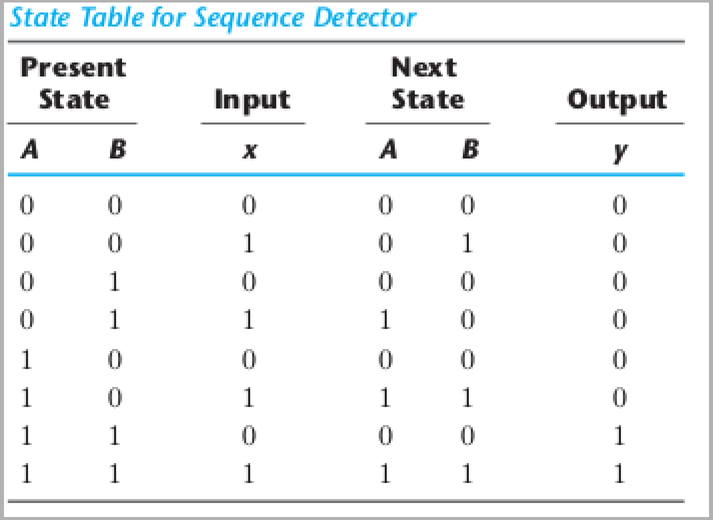
4. Obtain the binary-coded state table.

5. Choose the type of flip-flops to be used.

6. Derive the simplified flip-flop input equations and output equations.

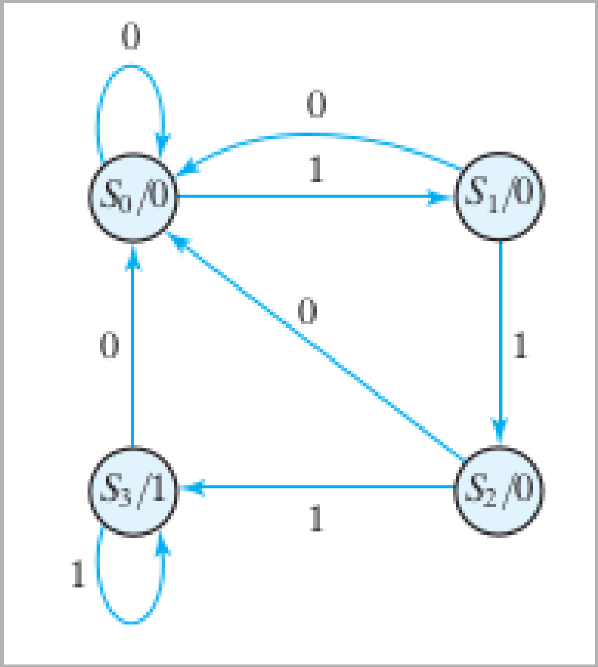
7. Draw the logic diagram.

Suppose we wish to design a circuit that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line (i.e., the input is a serial bit stream).



State Table for Sequence Detector

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

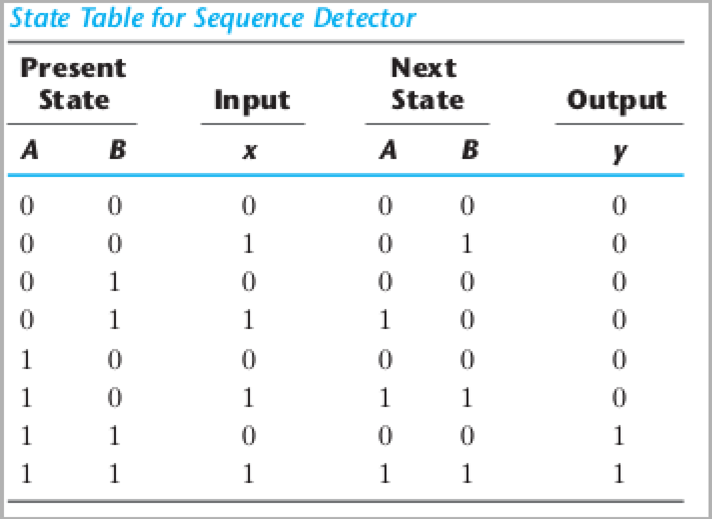


This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S3 and is 0 otherwise.

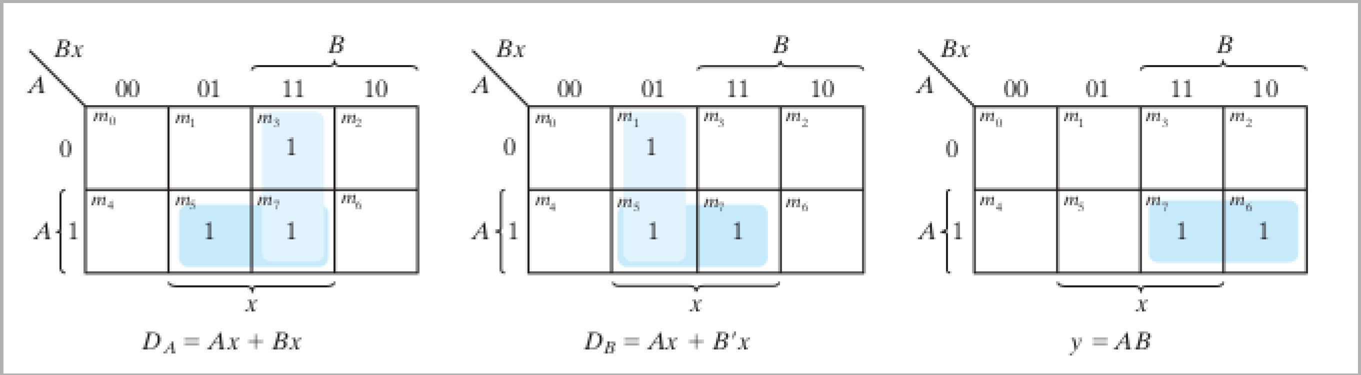
The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table.

We choose two D flip-flops to represent the four states, and we label their outputs A and B

$$A(t+1) = D_A(A,B,x) = \Sigma(3,5,7)$$
$$B(t+1) = D_B(A,B,x) = \Sigma(1,5,7)$$
$$y(A,B,x) = \Sigma(6,7)$$

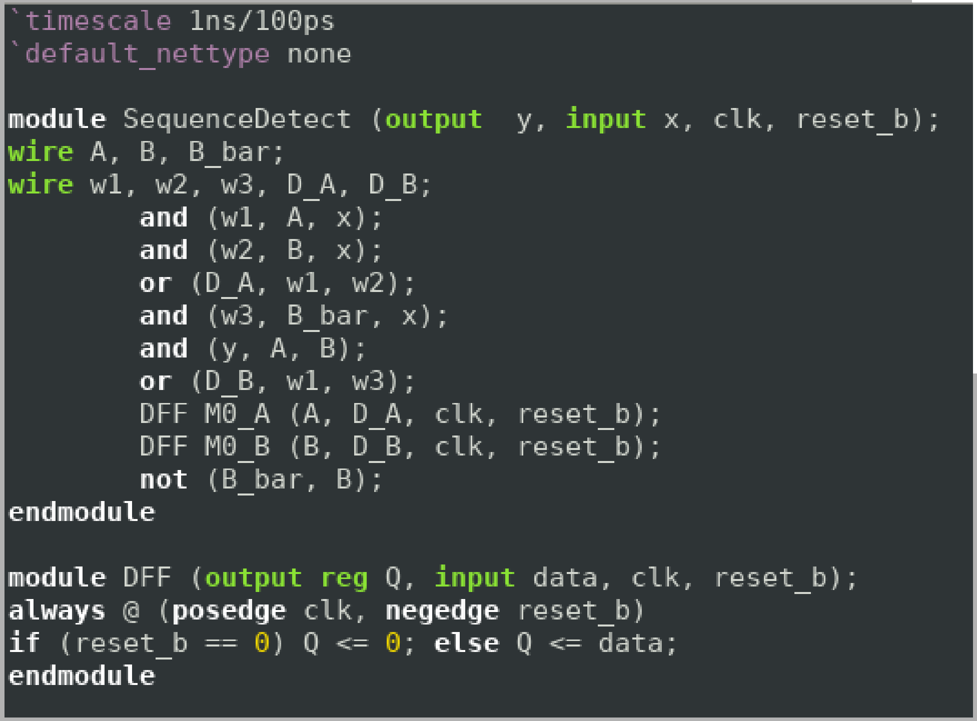

State Table for Sequence Detector

Present State		Input	Next State		Output
A	***B***	***x***	***A***	***B***	***y***
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

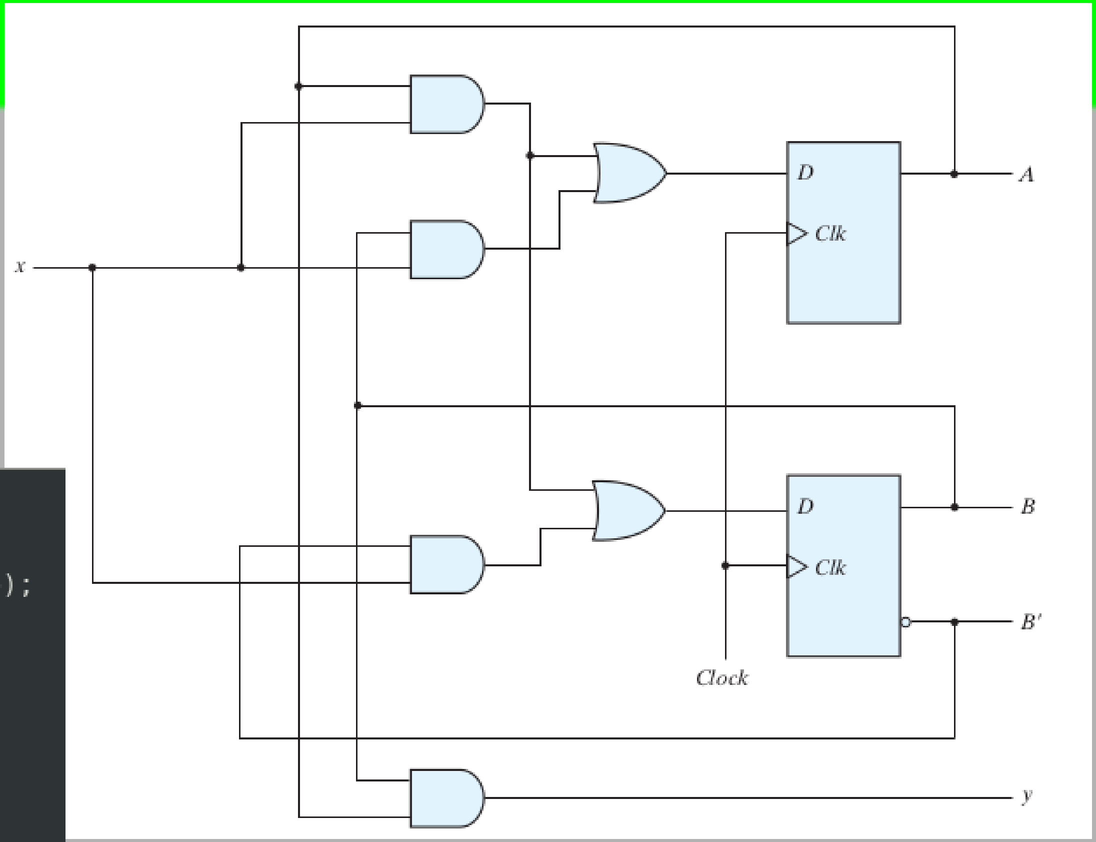


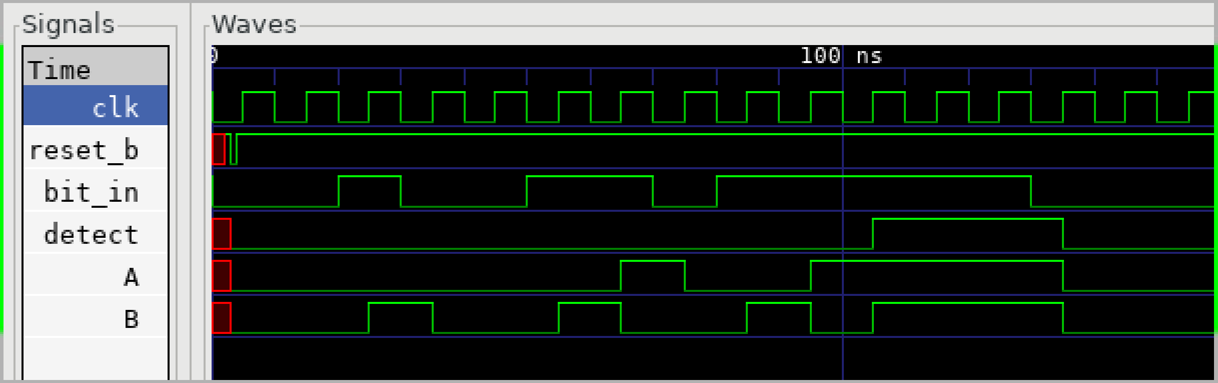
Logic diagram of a Moore-type sequence detector

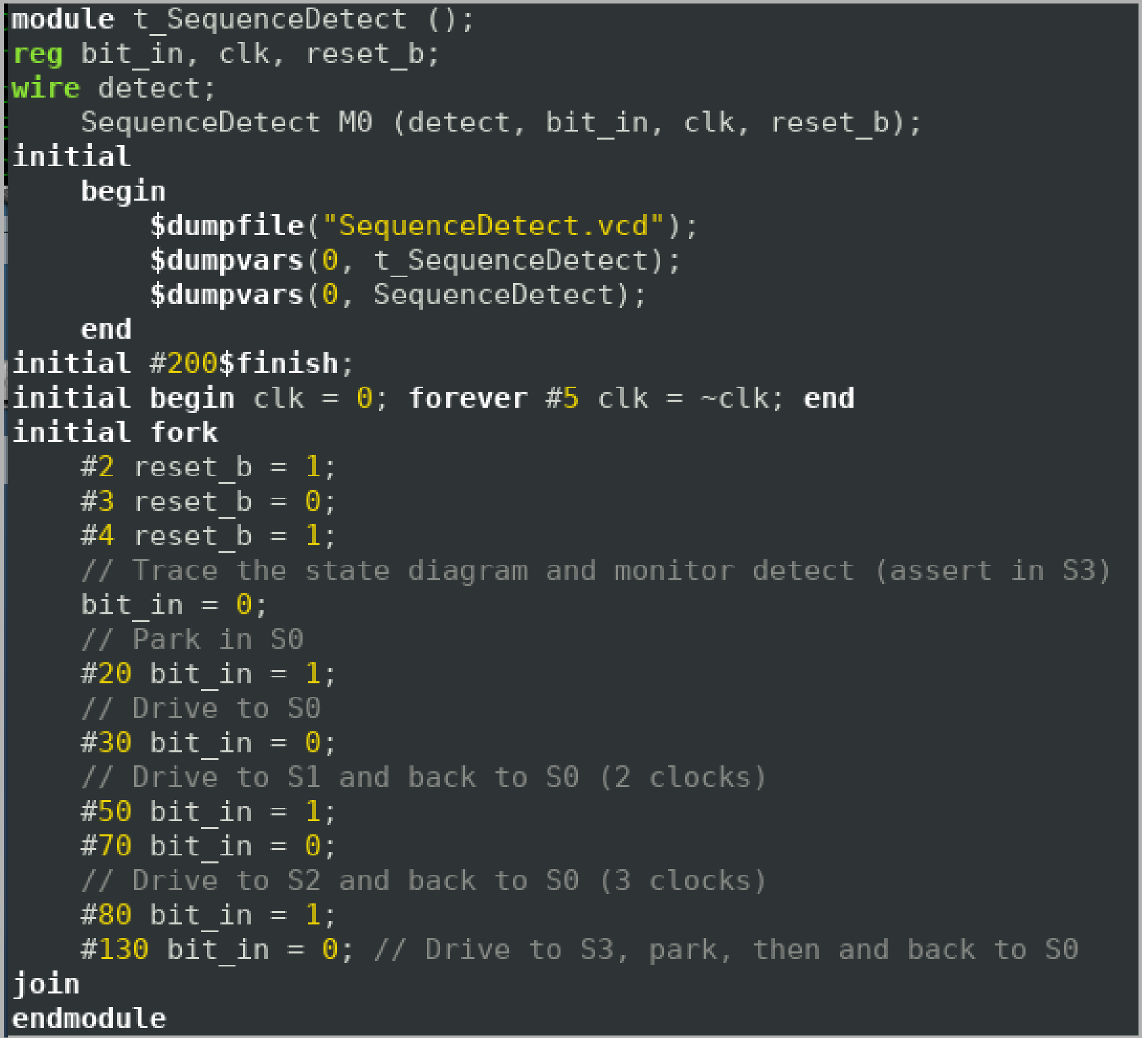
$$D_A = Ax + Bx$$
$$D_B = Ax + B'x$$
$$y = AB$$



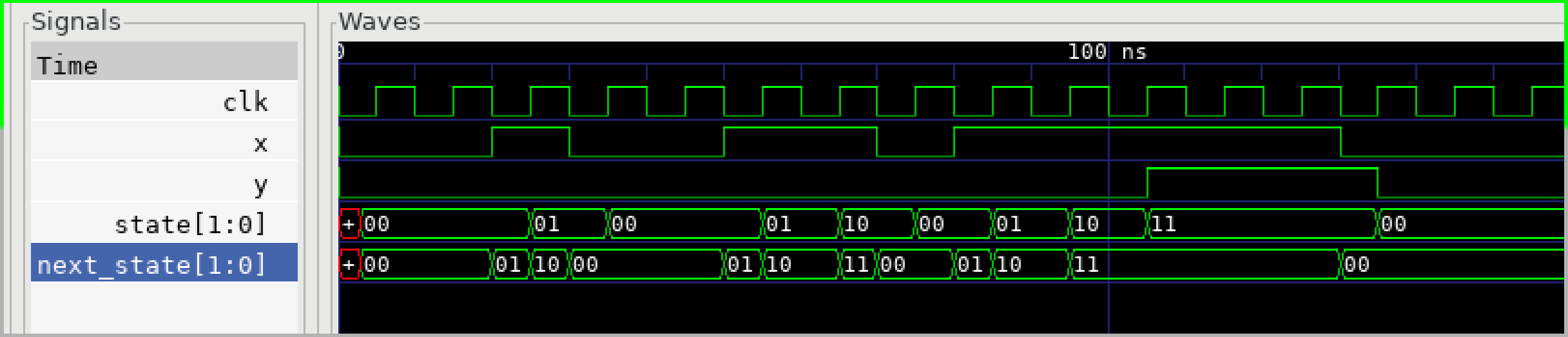
```
`timescale 1ns/100ps
`default_nettype none

module SequenceDetect (output  y, input x, clk, reset_b);
wire A, B, B_bar;
wire w1, w2, w3, D_A, D_B;
        and (w1, A, x);
        and (w2, B, x);
        or (D_A, w1, w2);
        and (w3, B_bar, x);
        and (y, A, B);
        or (D_B, w1, w3);
        DFF M0_A (A, D_A, clk, reset_b);
        DFF M0_B (B, D_B, clk, reset_b);
        not (B_bar, B);
endmodule

module DFF (output reg Q, input data, clk, reset_b);
always @ (posedge clk, negedge reset_b)
if (reset_b == 0) Q <= 0; else Q <= data;
endmodule
```

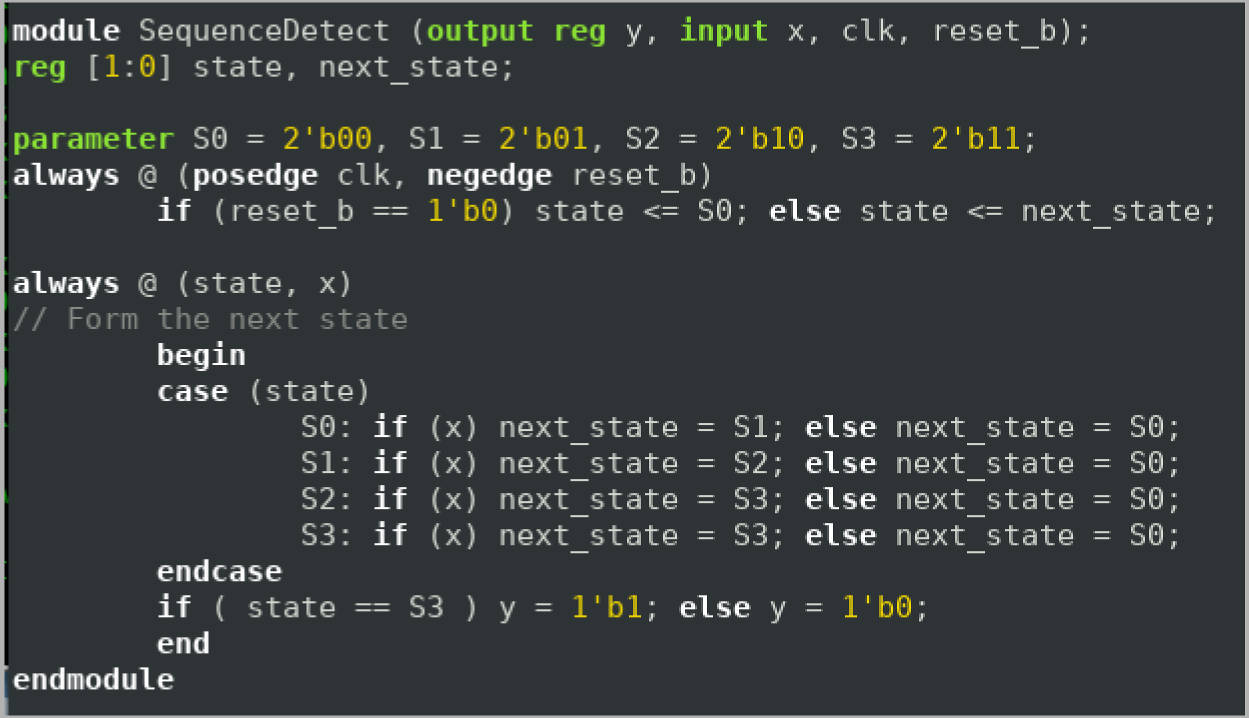




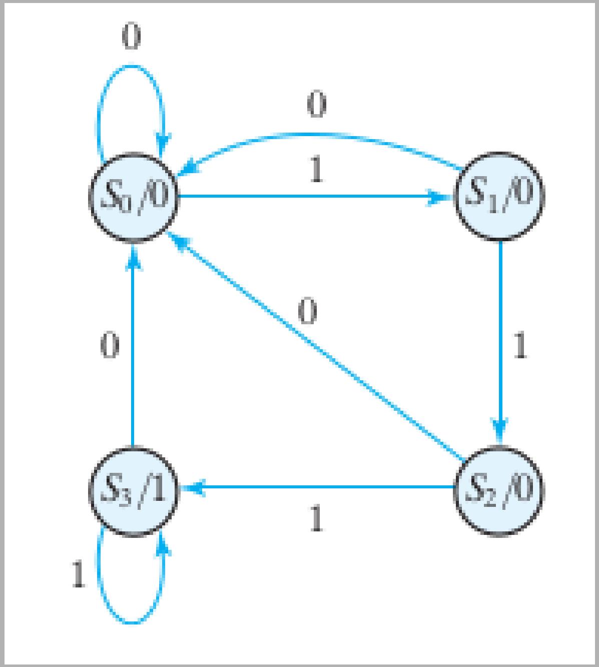


```
module t_SequenceDetect ();
reg bit_in, clk, reset_b;
wire detect;
    SequenceDetect M0 (detect, bit_in, clk, reset_b);
initial
    begin
        $dumpfile("SequenceDetect.vcd");
        $dumpvars(0, t_SequenceDetect);
        $dumpvars(0, SequenceDetect);
    end
initial #200$finish;
initial begin clk = 0; forever #5 clk = ~clk; end
initial fork
    #2 reset_b = 1;
    #3 reset_b = 0;
    #4 reset_b = 1;
    // Trace the state diagram and monitor detect (assert in S3)
    bit_in = 0;
    // Park in S0
    #20 bit_in = 1;
    // Drive to S0
    #30 bit_in = 0;
    // Drive to S1 and back to S0 (2 clocks)
    #50 bit_in = 1;
    #70 bit_in = 0;
    // Drive to S2 and back to S0 (3 clocks)
    #80 bit_in = 1;
    #130 bit_in = 0; // Drive to S3, park, then and back to S0
join
endmodule
```





```
module SequenceDetect (output reg y, input x, clk, reset_b);
reg [1:0] state, next_state;

parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
always @ (posedge clk, negedge reset_b)
        if (reset_b == 1'b0) state <= S0; else state <= next_state;

always @ (state, x)
// Form the next state
        begin
        case (state)
                S0: if (x) next_state = S1; else next_state = S0;
                S1: if (x) next_state = S2; else next_state = S0;
                S2: if (x) next_state = S3; else next_state = S0;
                S3: if (x) next_state = S3; else next_state = S0;
        endcase
        if ( state == S3 ) y = 1'b1; else y = 1'b0;
        end
endmodule
```


0
0
1
$S_0/0$
$S_1/0$
0
0
1
$S_3/1$
$S_2/0$
1
1