Chapter 6: Synchronous Sequential Circuits

Trương Ngọc Sơn, Ph.D

Synchronous sequential circuit

- Combinational logic circuits: The outputs are determined fully by the present values of inputs
- Flip-flop: The output depends on the state of the flip-flop rather than the value of its inputs at any given time; the inputs cause changes in the state
- Sequential circuit: The outputs depend on the past behavior of the circuit, as well as on the present values of inputs
- Synchronous sequential circuit: clock signal is used to control the operation of a sequential circuit
- The alternative, in which no clock signal is used, is called an asynchronous sequential circuit

Synchronous sequential circuit

- A sequential circuit is a circuit with memory, which forms the internal state of the circuit.
- Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The synchronous design methodology is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal

Review of Verilog assignment and procedure

Continuous Assignments
review

- Continuously assigns right side of expression to left side.
- Limited to basic Boolean and ? operators. For example a 2:1 mux:
  - ? operator

    assign D = (A= =1) ? B : C; // if A then D = B else D = C;

  - Boolean operators

    assign D = (B & A) | (C & ~A); // if A then D = B else D = C;

Procedural Assignments

- Executes a procedure allowing for more powerful constructs such as if-then-else and case statement.
- For example 2:1 mux:
  - if-else

```
if (A) D = B else D = C;
```

  - case

```
case(A)
    1'b1 : D = B;
    1'b0 : D = C;
    endcase
```

This is obviously much easier to implement and read then Boolean expressions!!

Always Block

- An always block is an example of a procedure.
- The procedure executes a set of assignments when a defined set of inputs **change**.

2:1 mux Always Block

```
Module mux_2_1(a, b, out, sel);
        input a, b, sel;
        output out;

        reg out;
        always @ (a or b or sel)
        begin
                if (sel) out = a;
                else out = b;
        end
endmodule
```

```
wire out;
assign out =(sel==1)?a:b;
```

Declare Module and IO as before.

All data types in always blocks must be declared as a ‘reg’ type.

This is required even if the data type is for combinational logic.

The always block ‘executes’ whenever signals named in the sensitivity list change.

Literally: always execute at a or b or sel.

Sensitivity list should include conditional (sel) and right side (a, b) assignment variables.

As Easier Way to Implement the Sensitivity List

- Recent versions of Verilog provides a means to implement the sensitivity list without explicitly listing each potential variable.
- Instead of listing variables as in the previous example

```
always @ (a or b or sel)
```

Simply use

```
always @*
```

The * operator will automatically identify all sensitive variables.

Blocking vs Non-Blocking Assignments

- Blocking (=) and non-blocking (<=) assignments are provided to control the execution order within an always block.

- Blocking assignments **literally block** the execution of the next statement until the current statement is executed.
  - **Consequently, blocking assignments result in ordered statement execution.**

  For example:

```
assume a = b = 0 initially;
a = 1;      //executed first
b = a;      //executed second
then a = 1, b = 1 after ordered execution
```

Blocking vs Non-Blocking Cont

- Non-blocking assignments **literally do not block** the execution of the next statements. The right side of all statements are determined first, then the left sides are assigned together.
  - **Consequently, non-blocking assignments result in simultaneous or parallel statement execution**.

For example:

assume a = b = 0 initially;

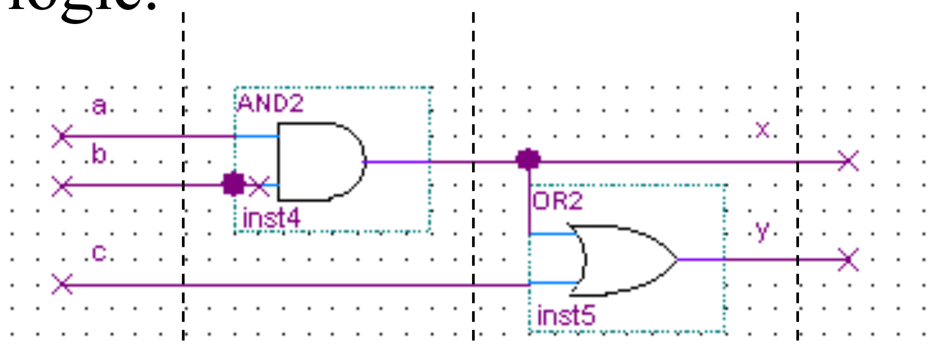
$$\left.\begin{array}{l} a <= 1; \\ b <= a; \end{array}\right\} \text{Execute together (in parallel)}$$

then a = 1, b = 0 after parallel execution

Result is different from ordered exec!!! Does not preserve logic flow

To Block or Not to Block ?

- Ordered execution mimics the inherent logic flow of combinational logic.
- Hence blocking assignments generally work better for combinational logic.
- For example:



x = a & b   y = x | b

logic flow

To Block or Not to Block ? cont

```
Module blocking(a,b,c,x,y);
        input a,b,c;
        output x,y;
        reg x,y;
        always @*
        begin
        x = a & b;
        y = x | c;
        end
endmodule
```

Blocking behavior	a	b	c	x	y
Initial values	1	1	0	1	1
a changes→always block execs	0	1	0	1	1
x = a & b; //make assignment	0	1	0	0	1
y = x | c; //make assignment	0	1	0	0	0

```
Module nonblocking(a,b,c,x,y);
        input a,b,c;
        output x,y;
        reg x,y;
        always @*
        begin
        x <= a & b;
        y <= x | c;
        end
endmodule
```

Non-blocking behavior	a	b	c	x	y
Initial values	1	1	0	1	1
a changes→always block execs	0	1	0	1	1
x = a & b;	0	1	0	1	1
y = x | c; //x not passed from here	0	1	0	1	1
make x, y assignments	0	1	0	0	1

non-blocking behavior does not preserve logic flow!!

Synchronous sequential circuit

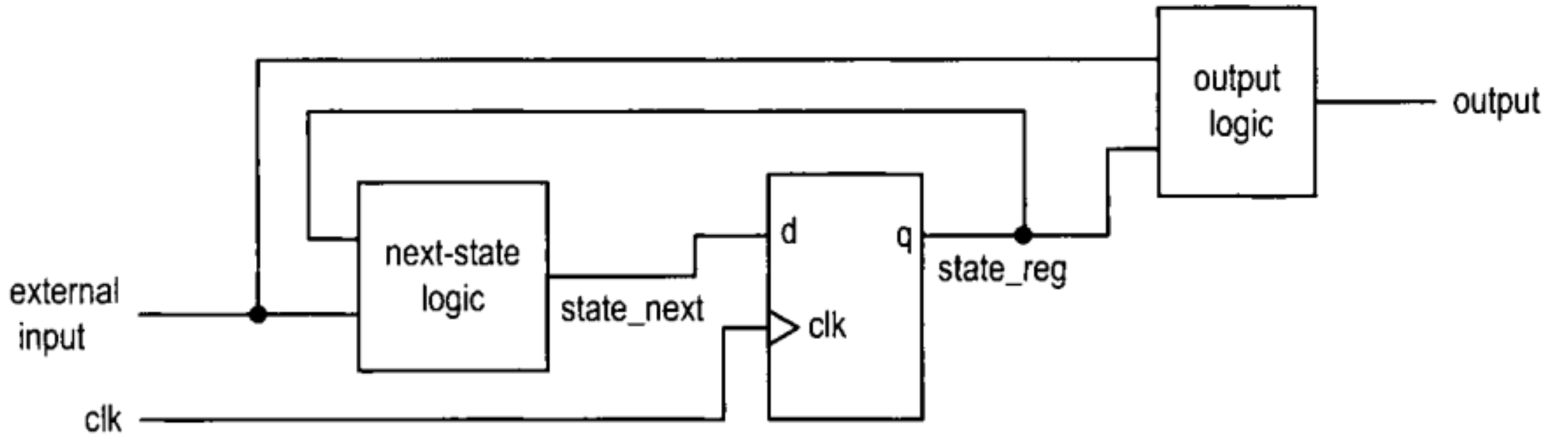
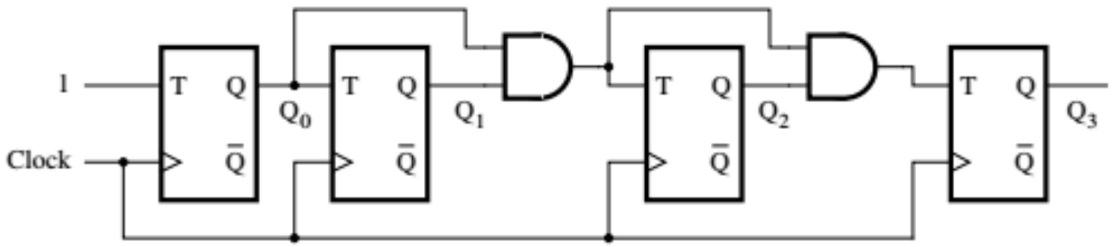


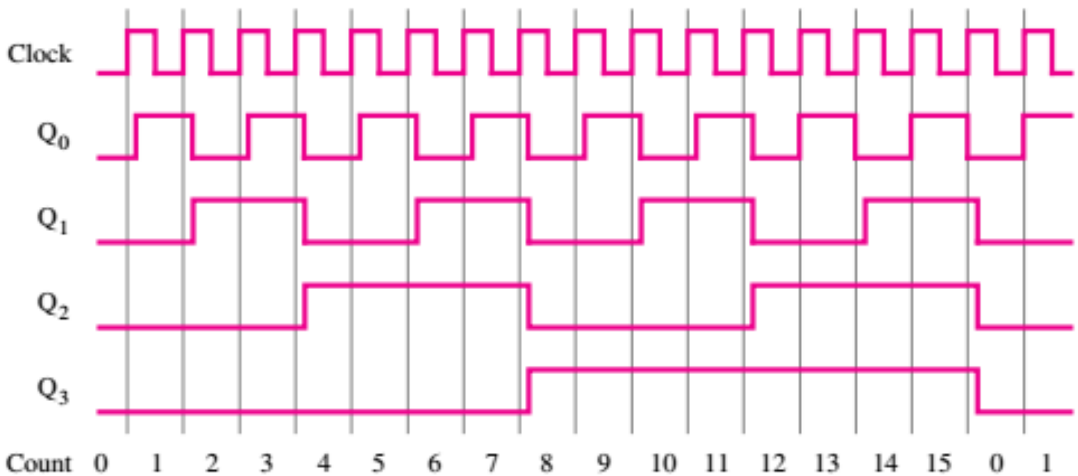
Figure 4.2 Block diagram of a synchronous system.

State register: a collection of D FFs controlled by the same clock signal
Next-state logic: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register
Output logic: combinational logic that generates the output signal

Design of synchronous counter



(a) Circuit



(b) Timing diagram

Figure 5.21 A four-bit synchronous up-counter.

Design of synchronous counter

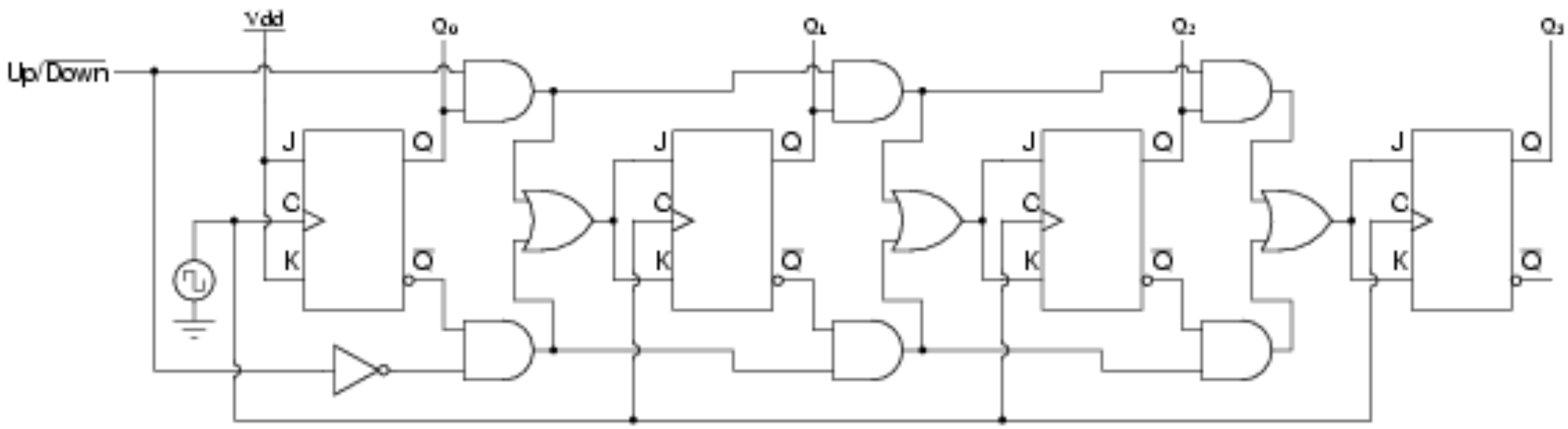
- Sample code

```
module Counter
   #(parameter N= 8)
            ( input wire clk, reset,
             output wire          [N-1:0] q );
            // signal declaration
            reg [N-1:0] r_reg;
            wire [N-1:0] r_next;
            // body, register
            always @(posedge clk, posedge reset)
            if (reset)
                        r_reg <= 0;
            else
                        r_reg<=r_next;    // <= is non-blocking statement
            // next state logic
            assign r_next = r_reg + 1;
            // output logic
            assign q=r_reg;
endmodule
```

Up/ down counter

- Design 8-bit synchronous up/down counter

A four-bit synchronous "up/down" counter

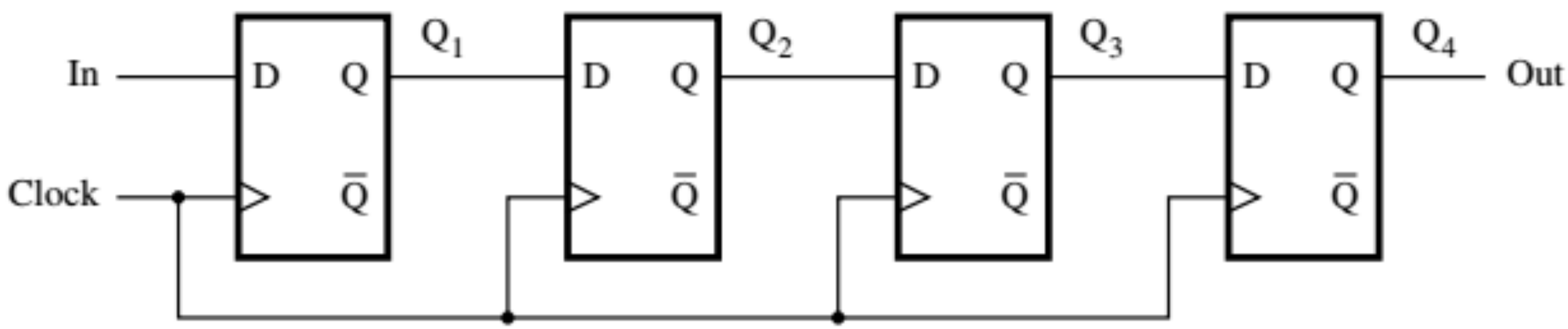


8-bit up/down counter

```
module CounterUD
        ( input wire clk,reset,ud,
         output wire          [7:0] q  );
        // signal declaration
        reg [7:0] r_reg;
        wire [7:0] r_next;
        // body, register
        always @(posedge clk, posedge reset)
         if (reset)
         r_reg<=0;
         else
         r_reg<=r_next;
        // next state logic
        assign r_next = (ud==1)?r_reg + 1:r_reg - 1;
        // output logic
        assign q=r_reg;
endmodule
```

Register

- A register is a collection of D FFs that are controlled by the same clock and reset signals
- Serial In – Serial Out (SISO) shift register. The block diagram of 4-bit SISO shift register is shown in the following figure.



Register

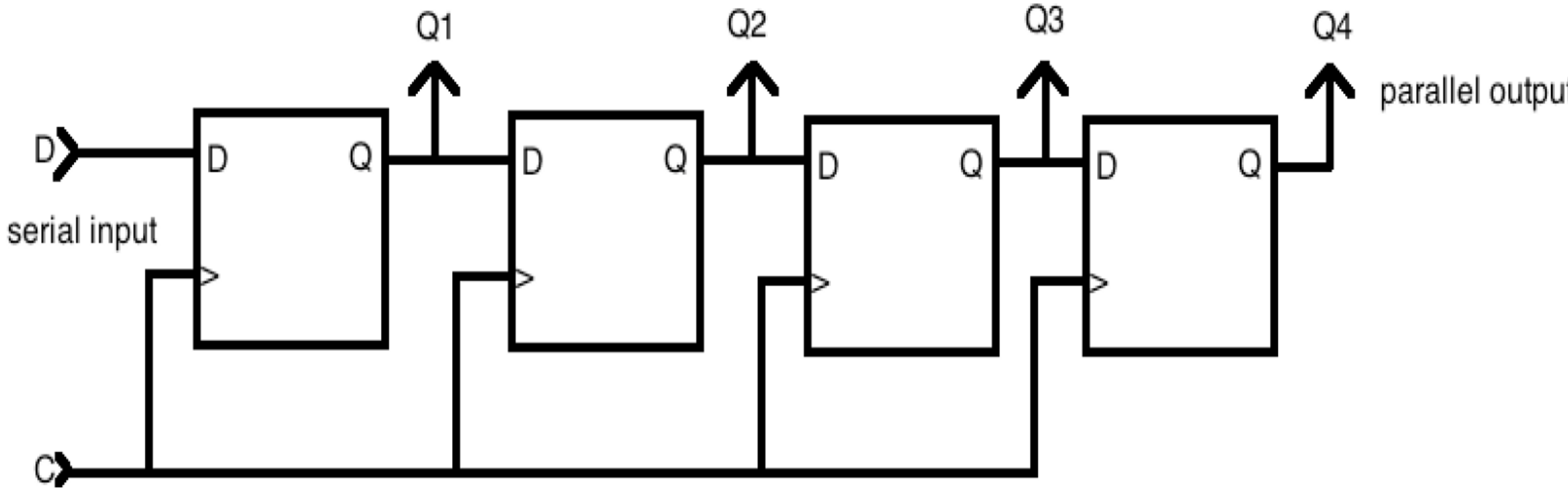
- Sample code

```
module Shift_SISO

#(parameter N= 4) // 500,000,000 for 0.1Hz
            ( input wire clk,reset,s_in,
              output wire          s_out
   );
             // signal declaration

             reg [N-1:0] r_reg;
             wire [N-1:0] r_next;
             // body, register
             always @(posedge clk, posedge reset)
            r_reg<=r_next;
            // next state logic
            assign r_next = {s_in,r_reg[N-1: 1]};
            // output logic
            assign s_out= r_reg[0];
endmodule
```

Serial input – parallel output shift register

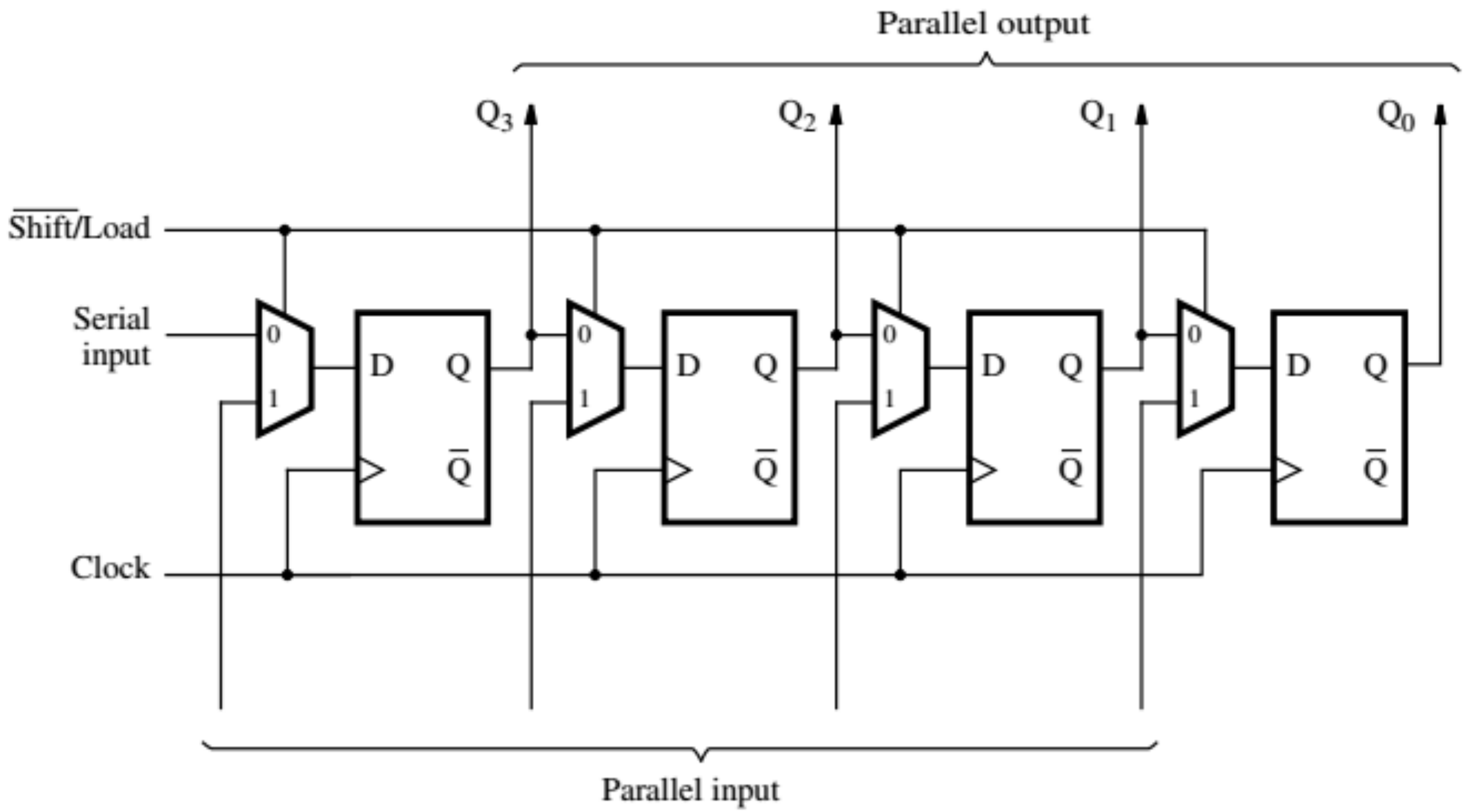


Register

- Sample code

```
module Shift_SIPO
          (
           input wire clk,s_in,
           output wire          [7:0] q_out );
          // signal declaration
          reg [7:0] r_reg;
          wire [7:0] r_next;
          // body, register
          always@(negedge clk)
          r_reg<=r_next;
          // next state logic
          assign r_next = {s_in,r_reg[7:1]};
          // output logic
          assign q_out= r_reg;
```

Serial input – parallel output shift register



Synchronous sequential circuit Finite state machine (FSM)

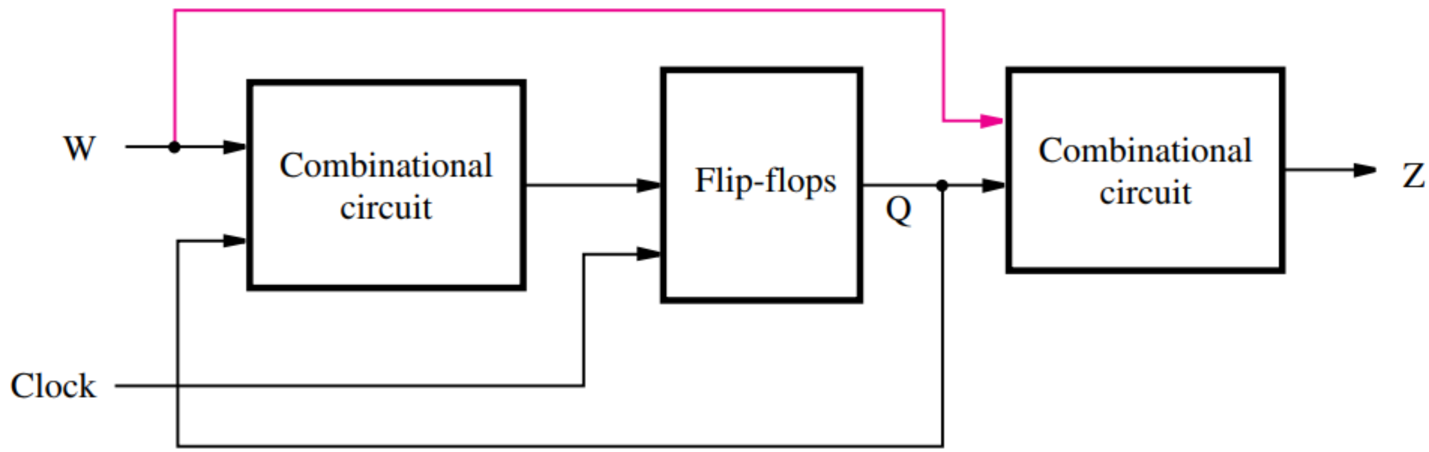


Figure 6.1 The general form of a sequential circuit.

- Synchronous sequential circuits are realized using combinational logic and one or more flip-flops.
- The circuit has a set of primary inputs, W , and produces a set of outputs, Z. The stored values in the flip-flops are referred to as the state, Q, of the circuit
- Under control of the clock signal, the flip-flops change their state as determined by the combinational logic that feeds the inputs of these flip-flops. the circuit moves from one state to another

Moore and Mealy type of FSM

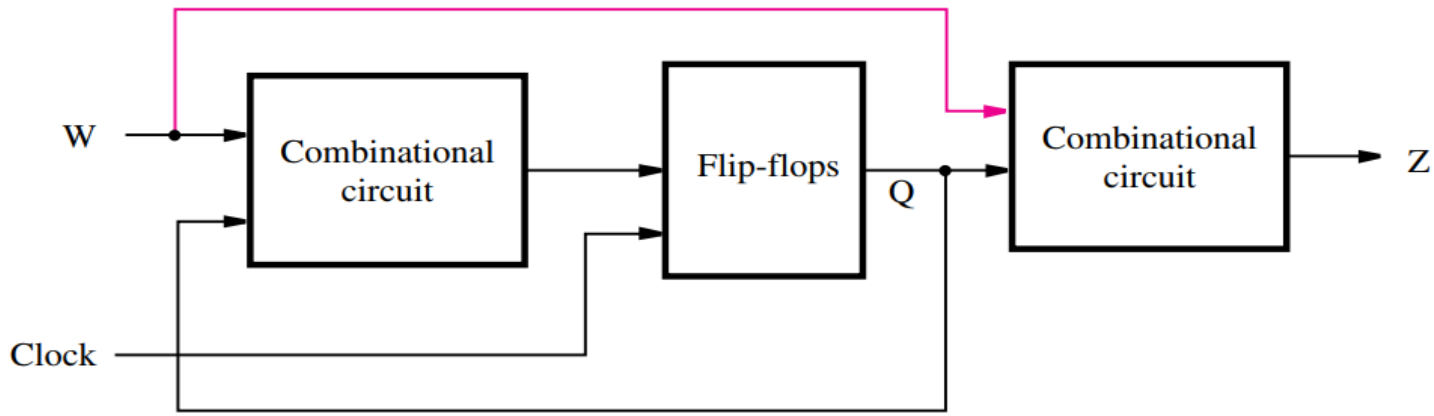


Figure 6.1 The general form of a sequential circuit.

- Mealy type: The outputs are a function of the present state of the flip-flops and of the primary inputs
- Moore type: The outputs always depend on the present state, they do not necessarily have to depend directly on the primary inputs
- that sequential circuits whose outputs depend only on the state of the circuit are of **Moore** type, while those whose outputs depend on both the state and the primary inputs are of **Mealy** type
- Sequential circuits are also called finite state machines (FSMs)

State Machine

- The first step in designing a finite state machine is to determine how many states are needed and which transitions are possible from one state to another

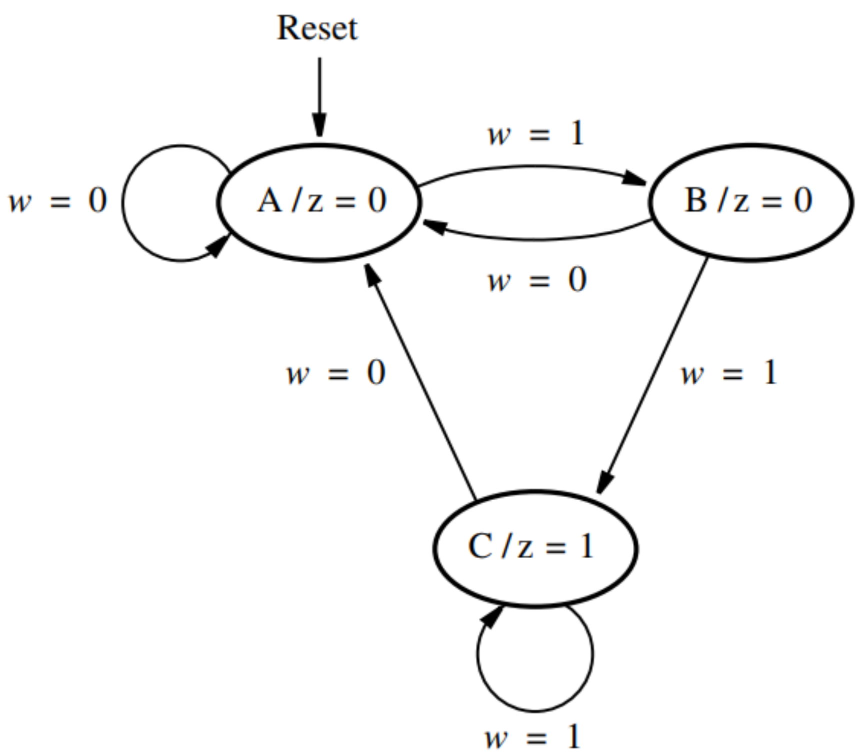


Figure 6.3 State diagram of a simple sequential circuit.

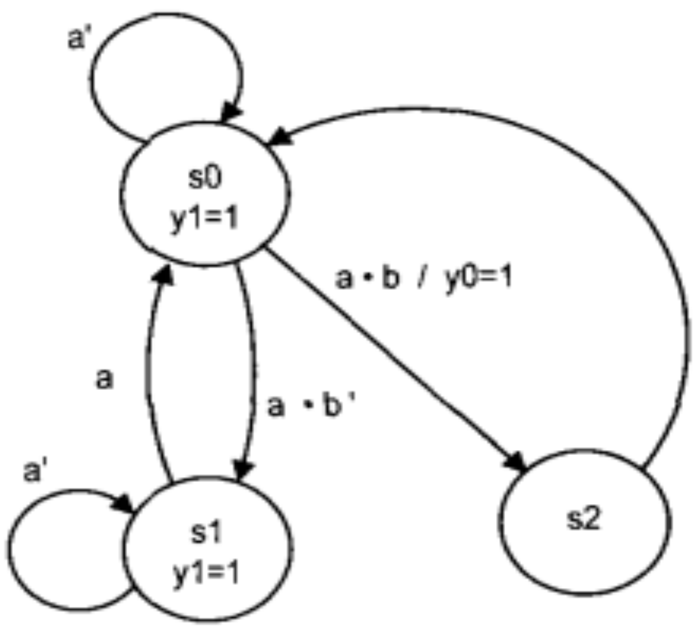
Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	A	B	0
B	A	C	0
C	A	C	1

Figure 6.4 State table corresponding to Figure 6.3.

State Machine

```
module simple (Clock, Resetn, w, z);
input Clock, Resetn, w; output z;
reg [2:1] y, Y;
parameter [2:1] A = 2’b00, B = 2’b01, C = 2’b10;
// Define the next state combinational circuit
always @(w, y)
case (y)
A: if (w) Y = B;
else Y = A;
B: if (w) Y = C;
else Y = A;
C: if (w) Y = C;
else Y = A;
default: Y = 2’bxx;
endcase
// Define the sequential block
always @(negedge Resetn, posedge Clock)
if (Resetn == 0) y < = A;
else y < = Y;
// Define output
assign z = (y == C);
Endmodule
```

FSM



FSM

```
module  fsm-eg-mult-seg
(
input   wire   clk ,   reset ,
input   wire   a ,  b ,
output   wire   yo,  y l  );
//symbolic   state   declaration
localparam   [1:0]  S0  = 2'b00;  S1  =   2'b01 ,
S2=2'b10;
//  signal  declaration
reg  [1 : 0]  state_reg,state_next ;
          // state register
always  @ (posedge  clk ,posedge  reset)
i f (reset)
state_reg<=S0;
else
state_reg<=state_next;
//next_state logic
always @*
case (state_reg)
SO: if(a)
    if(b)
     state_next=S2;
     else
     state_next=Sl;
  else
     state_next=S0;
Sl:   if(a)
     state_next=S0;
      else
     state_next=S1;
S2:  state_next=S0;
default:   state_next=S0;
endcase
//Moore outputlogic
assign yl=(state_reg==S0)||(state_reg==Sl);
//Mealy outputlogic
assign y0=(state_reg==SO)&a&b;
endmodule
```

Design of Counter Using Sequential Circuit

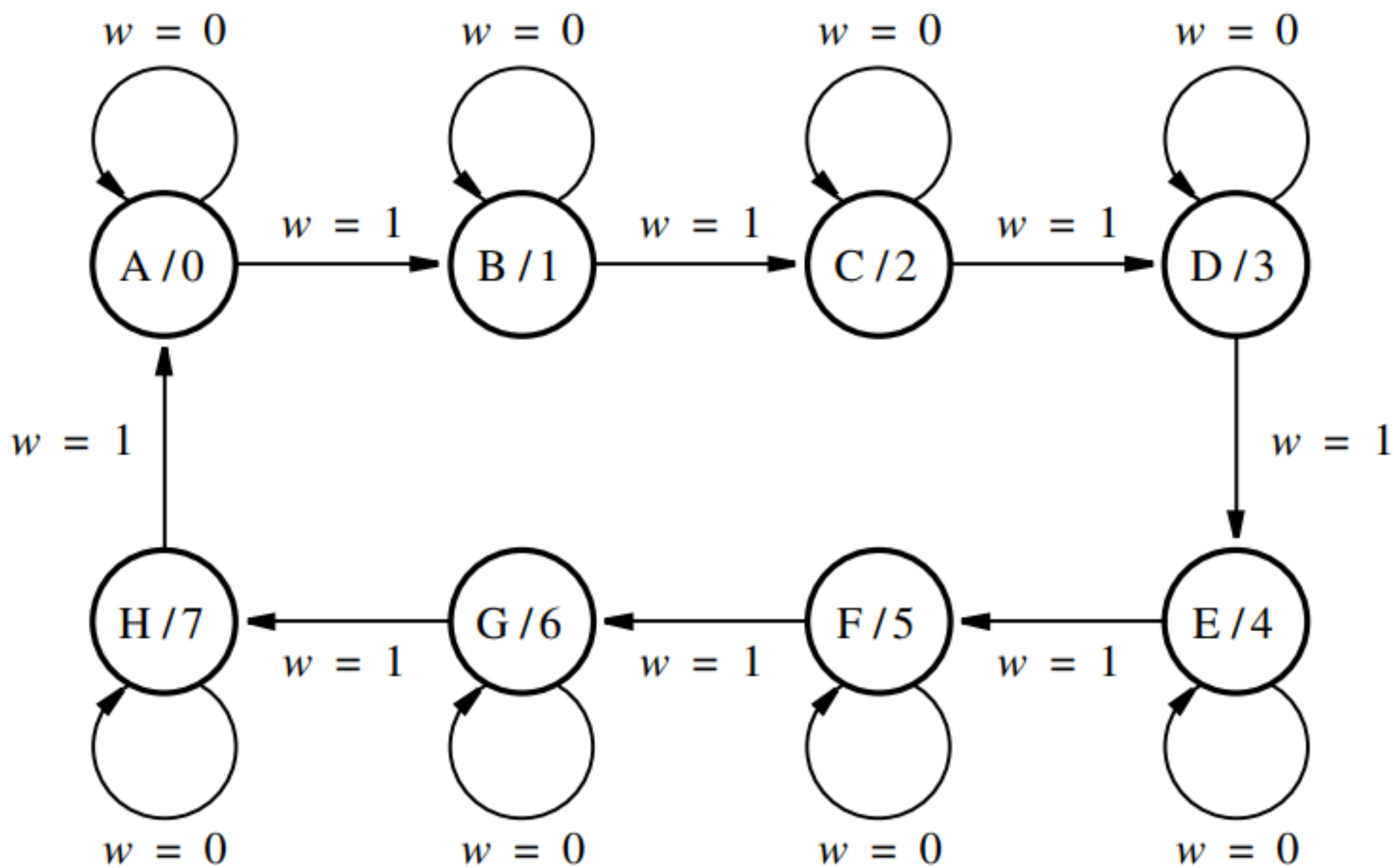


Figure 6.60 State diagram for the counter.

Design of Counter Using Sequential Circuit

Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure 6.61 State table for the counter.

	Present state $y_2y_1y_0$	Next state		Count $z_2z_1z_0$
		$w = 0$ $Y_2Y_1Y_0$	$w = 1$ $Y_2Y_1Y_0$	
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figure 6.62 State-assigned table for the counter.

Design of Counter Using Sequential Circuit

- Sample code

Homework #1

- Design the up/down counter. The input clock is 50Mhz. The circuit count up or down, with the frequency is selected by two switches (f,2*f,4*f,8*f, where f is less than f_{clk}). The block diagram is shown as follows