Digital Logic
Design with FPGA

Digital Design with VHDL

Sequential Logic 1

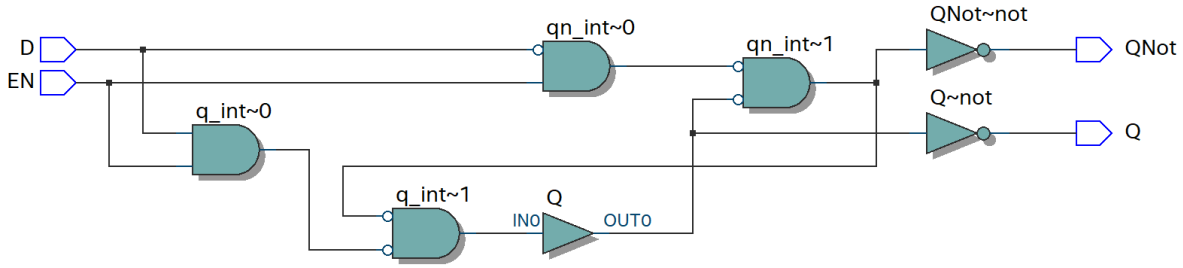
Outline

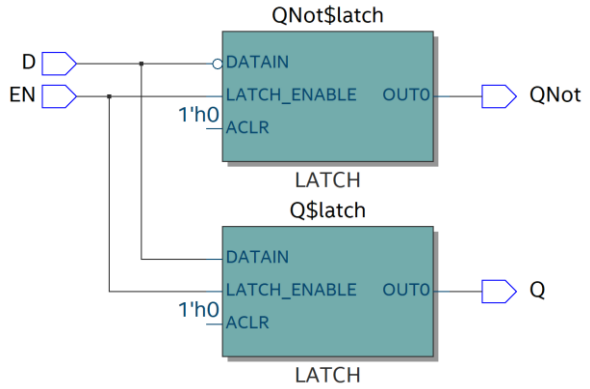
- Latches
- Flip-flops
- Flip-Flop Operating Characteristics
- Shift Register Operations
- Johnson Counter
- Ring Counter

Gated D Latch

- The latch is a type of temporary storage device that has two stable states (bistable) and is normally placed in a category separate from that of flip-flops.
- The main difference between latches and flip-flops is in the method used for changing state.
- Output Q follows the input D when EN is HIGH.

Inputs		Outputs		
D	*EN*	*Q*	$\overline{Q}$	Comments
0	1	0	1	RESET
1	1	1	0	SET
X	0	Q_0	$\overline{Q}_0$	No change





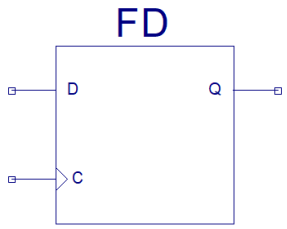

```
4   entity DLatch is
5       port (D, EN: in std_logic;
6             Q, QNot: out std_logic );
7   end entity DLatch;
```

```
9   architecture LogicOperation of DLatch is
10  signal q_int, qn_int: std_logic;
11  begin
12      q_int <= qn_int nand (D nand EN);
13      qn_int <= q_int nand (not D nand EN);
14      -- output
15      Q <= q_int;
16      QNot <= qn_int;
17  end architecture LogicOperation;
```

```
9   architecture behav of DLatch is
10  begin
11      process(EN,D) begin
12          if EN='1' then
13              Q <= D;
14              QNot <= not D;
15          end if;
16      end process;
17  end architecture;
```

D Flip-Flop

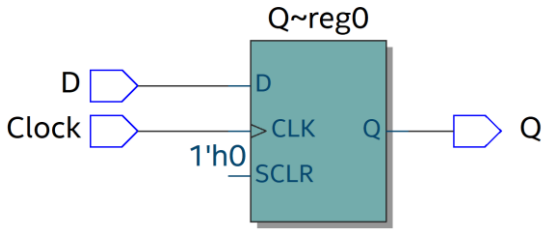
- Flip-flops are synchronous bistable devices.
- The output changes state only at a specified point (leading or trailing edge) on the triggering input called the clock (CLK)
- The D input can be changed at any time when the clock input is LOW or HIGH (except for a very short interval around the triggering transition of the clock) without affecting the output.



Truth table for a positive edge-triggered D flip-flop.

Inputs		Outputs		
D	CLK	Q	$\overline{Q}$	Comments
0	↑	0	1	RESET
1	↑	1	0	SET

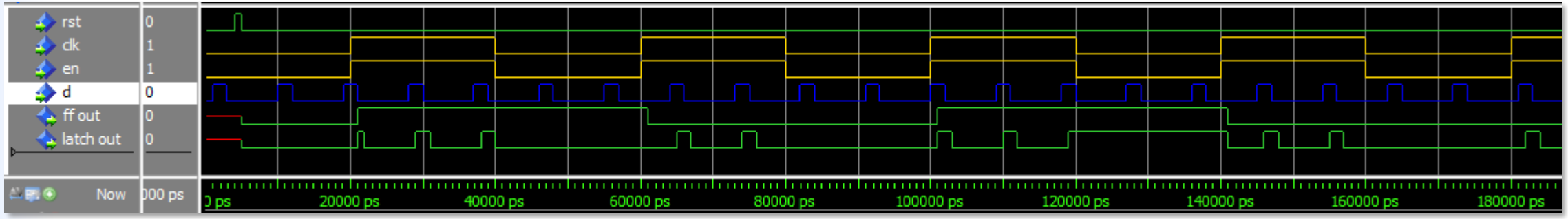
↑ = clock transition LOW to HIGH




```
4   entity dff is
5       port (D, Clock: in std_logic;
6             Q: out std_logic );
7   end entity dff;
8
9   architecture behav of dff is
10  begin
11    process(Clock) begin
12      if rising_edge(Clock) then
13          Q <= D;
14      end if;
15    end process;
16  end architecture;
```

D Flip-Flop vs D Latch

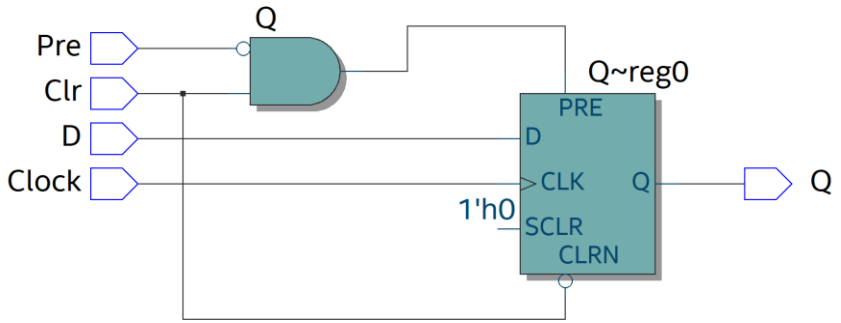
- The time waveforms show the operation of two different D-type elements: a flip-flop (ff_out signal) and a latch (latch_out signal).
- Identical forces were supplied to the inputs of both (**rst**, **clk/en**, **d**).
- The state at the flip-flop output changes only when the edge is active. The latch is transparent to the data signal d for half a clock period. The data at the latch_out output is latched when the **en** is turned off.
- Unknowingly introducing latches in the design of a synchronous system using FPGA technology will certainly cause **errors** in the operation of the device.

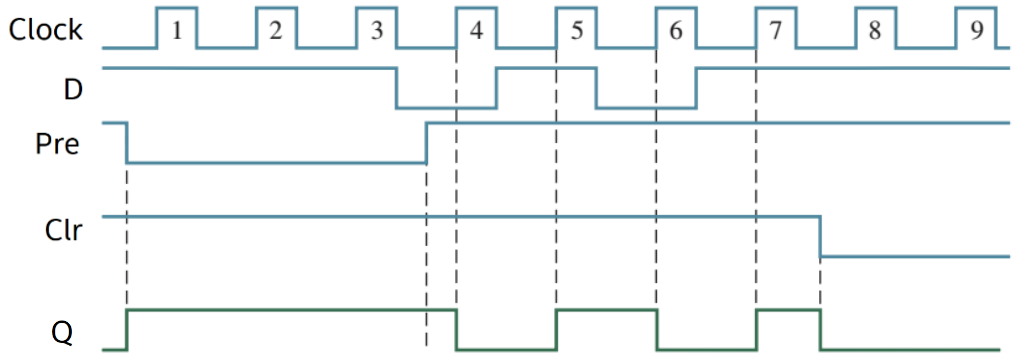


Asynchronous Inputs

- Most integrated circuit flip-flops also have asynchronous inputs.
  These are inputs that affect the state of the flip-flop independent of the clock.
- They are normally labeled preset (PRE) and clear (CLR).
- An active level on the preset input will set the flip-flop, and an active level on the clear input will reset it.

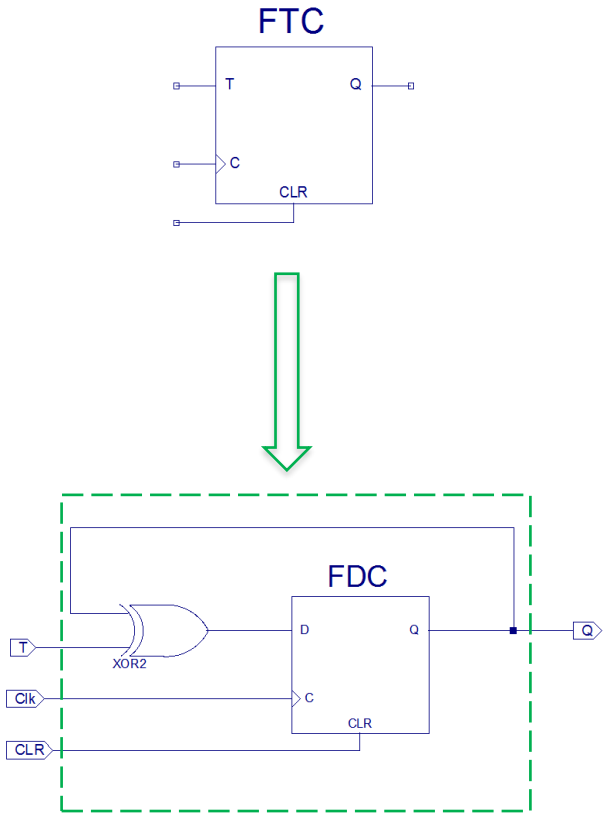
```
47  entity dff_cpa is
48      port (D, Clock: in std_logic;
49            Pre, Clr: in std_logic; -- low active
50            Q: out std_logic );
51  end entity dff_cpa;
52
53  architecture behav of dff_cpa is
54  begin
55    process(Clock,Pre,Clr) begin
56      if Clr = '0' then
57          Q <= '0';
58      elsif Pre = '0' then
59          Q <= '1';
60      elsif rising_edge(Clock) then
61          Q <= D;
62      end if;
63    end process;
64  end architecture;
```





T Flip-Flop

- The T (toggle) input of the T flip-flop is synchronous.
- When T is HIGH, the flip-flop changes state.




```
4   entity tff is
5       port (T, Clock: in std_logic;
6             Clr: in std_logic; -- active low, async
7             Q: out std_logic );
8   end entity tff;
9
10  architecture behav of tff is
11  signal q_int: std_logic:='0';
12  begin
13    process(Clock, Clr) begin
14      if Clr = '0' then
15          q_int <= '0';
16      elsif rising_edge(Clock) then
17          if T = '1' then
18              q_int <= not q_int; -- toggle
19          else
20              q_int <= q_int; -- no change
21          end if;
22      end if;
23    end process;
24    Q <= q_int;
25  end architecture;
```

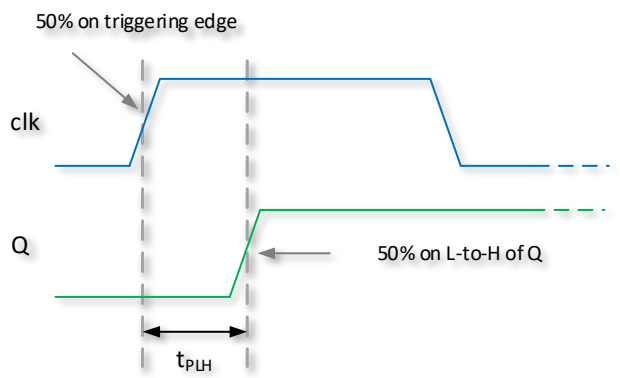
Flip-Flop operating characteristics

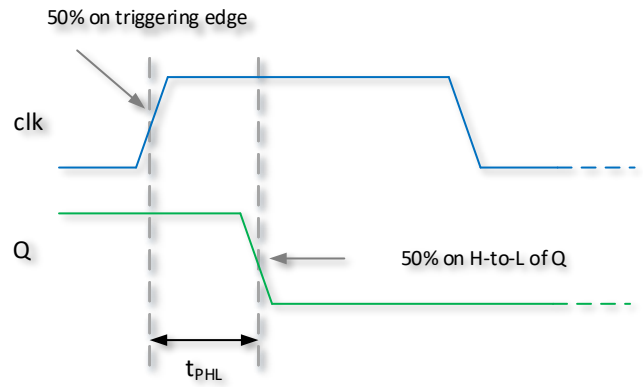
- Propagation Delay Times
- Set-up Time
- Hold Time
- Maximum Clock Frequency
- Pulse Widths
- Power Dissipation

Propagation Delay Times

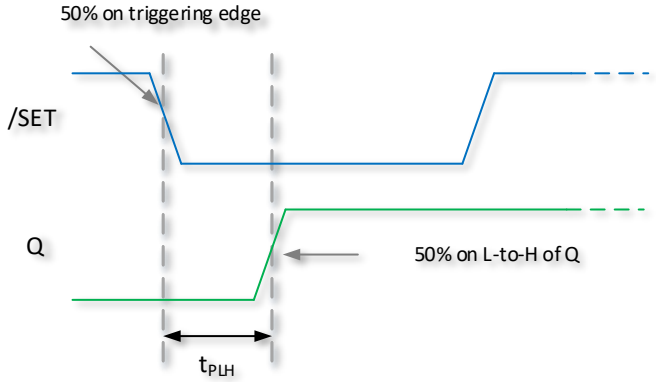
A propagation delay time is the interval of time required after an input signal has been applied for the resulting output change to occur.

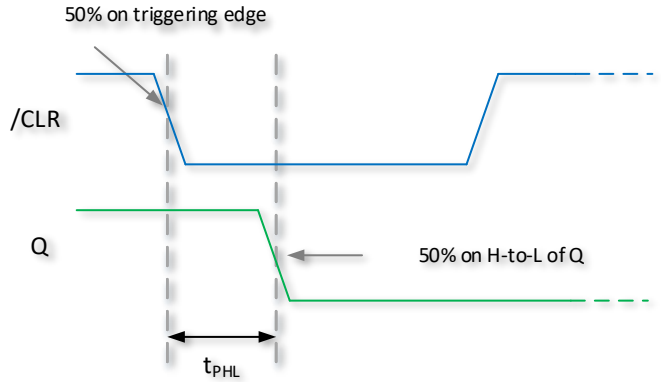
- t_{PLH} from the triggering edge of the clock pulse to the L-to-H transition of the output
- t_{PHL} from the triggering edge of the clock pulse to the H-to-L transition of the output





- t_{PLH} from the leading edge of the preset input to the L-to-H transition of the output
- t_{PHL} from the leading edge of the clear input to the H-to-L transition of the output

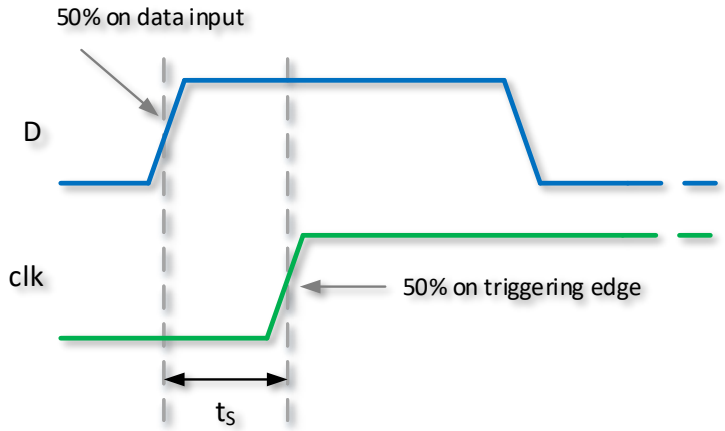




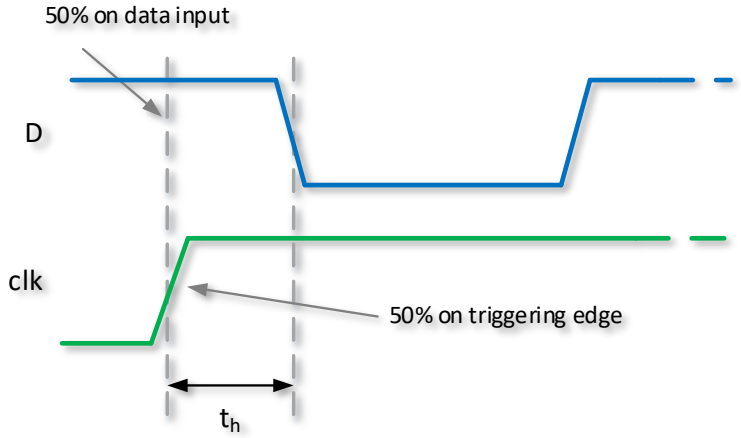
source: Thomas Floyd: Digital Fundamentals

Set-up & Hold times

- The set-up time (t_s) is the minimum interval required for the logic levels to be maintained constantly on the inputs (T or D) prior to the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.



- The hold time (t_h) is the minimum interval required for the logic levels to remain on the inputs after the triggering edge of the clock pulse in order for the levels to be reliably clocked into the flip-flop.



source: Thomas Floyd: Digital Fundamentals

Flip-Flop operating characteristics

- The maximum clock frequency (f_{max}) is the highest rate at which a flip-flop can be reliably triggered.
  At clock frequencies above the maximum, the flip-flop would be unable to respond quickly enough.
- Minimum pulse widths (t_W) for reliable operation are usually specified by the manufacturer for the clock, preset, and clear inputs.
  Typically, the clock is specified by its minimum HIGH time and its minimum LOW time.
- The power dissipation of any digital circuit is the total power consumption of the device.

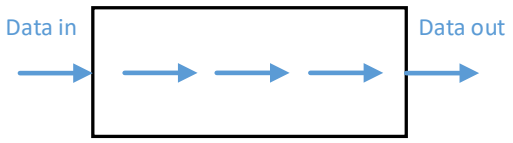
Comparison of operating parameters for IC families of flip-flops at 25°C.

	CMOS		Bipolar (TTL)	
Parameter	**74HC74A**	**74AHC74**	**74LS74A**	**74F74**
t_{PHL} (CLK to Q)	17 ns	4.6 ns	40 ns	6.8 ns
t_{PLH} (CLK to Q)	17 ns	4.6 ns	25 ns	8.0 ns
t_{PHL}($\overline{CLR}$ to Q)	18 ns	4.8 ns	40 ns	9.0 ns
t_{PLH}($\overline{PRE}$ to Q)	18 ns	4.8 ns	25 ns	6.1 ns
t_s (set-up time)	14 ns	5.0 ns	20 ns	2.0 ns
t_h (hold time)	3.0 ns	0.5 ns	5 ns	1.0 ns
t_W (CLK HIGH)	10 ns	5.0 ns	25 ns	4.0 ns
t_W (CLK LOW)	10 ns	5.0 ns	25 ns	5.0 ns
t_W($\overline{CLR}$/$\overline{PRE}$)	10 ns	5.0 ns	25 ns	4.0 ns
f_{max}	35 MHz	170 MHz	25 MHz	100 MHz
Power, quiescent	0.012 mW	1.1 mW		
Power, 50% duty cycle			44 mW	88 mW

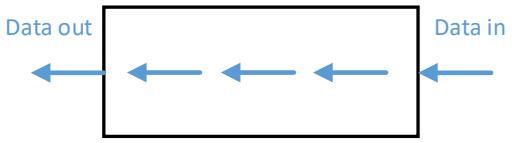
source: Thomas Floyd: Digital Fundamentals

Shift Register Operations

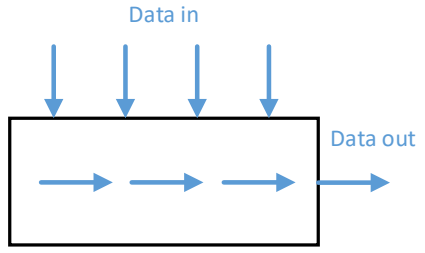
- A register is a digital circuit with two basic functions: data storage and data movement.
  The storage capability of a register makes it an important type of memory device.
- The storage capacity of a register is the total number of bits (1s and 0s) of digital data it can retain. Each stage (flip-flop) in a shift register represents one bit of storage capacity; therefore, the number of stages in a register determines its storage capacity.
- The shift capability of a register permits the movement of data from stage to stage within the register or into or out of the register upon application of clock pulses.



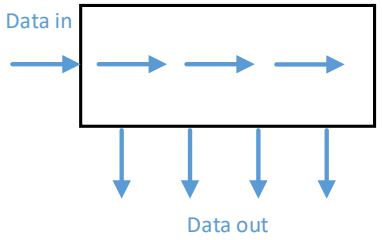
(a) Serial in/shift right/serial out



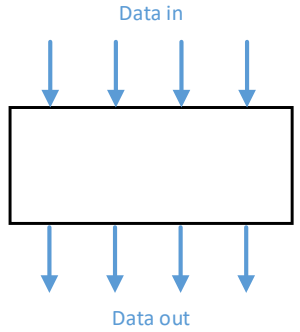
(b) Serial in/shift left/serial out



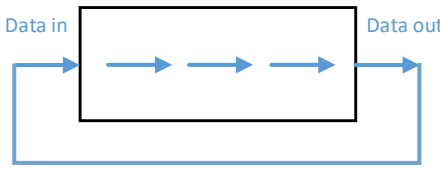
(c) Parallel in/serial out



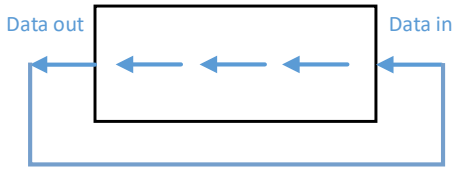
(d) Serial in/parallel out



(e) Parallel in/parallel out



(f) Rotate right

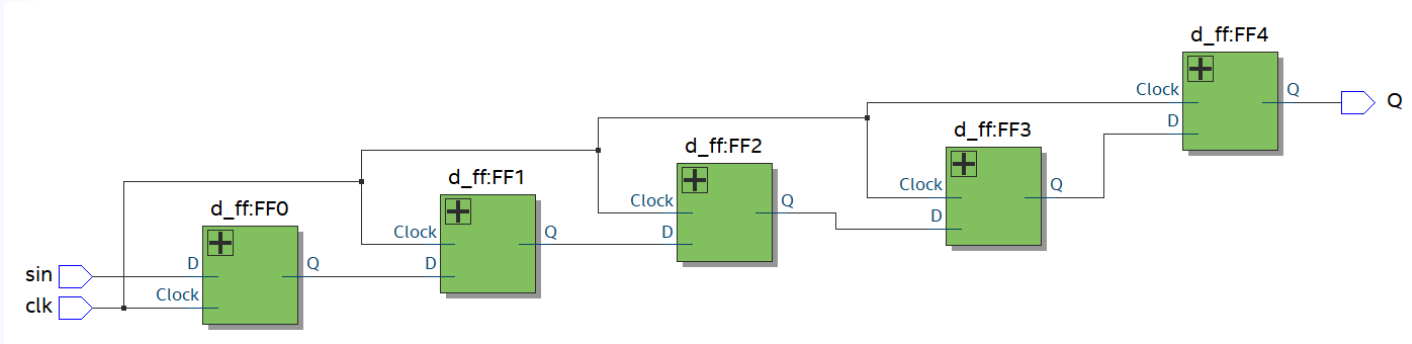


(g) Rotate left

source: Thomas Floyd: Digital Fundamentals

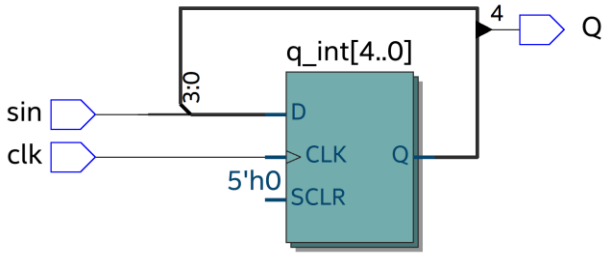
Serial In/Serial Out

The serial in/serial out shift register accepts data serially - one bit at a time on a single line.
It produces the stored information on its output also in serial form.




```
4   entity SRG5 is
5       port (sin, clk: in std_logic;
6             Q: out std_logic );
7   end entity;
```

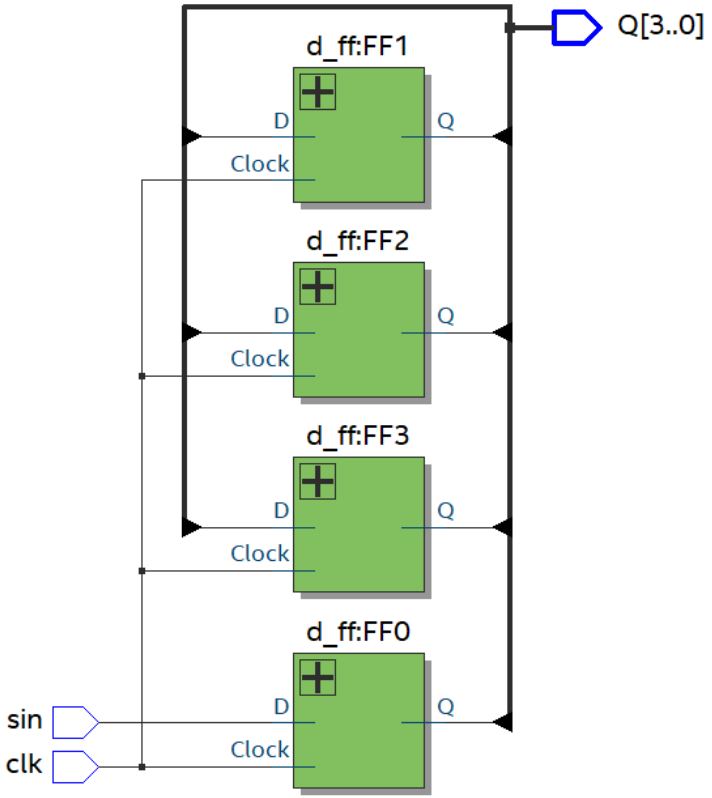
```
9   architecture struct of SRG5 is
10  signal q_int: std_logic_vector(4 downto 0);
11  begin
12  FF0: dff port map (clk=>clk, d=>sin, q=>q_int(0));
13  FF1: dff port map (clk=>clk, d=>q_int(0), q=>q_int(1));
14  FF2: dff port map (clk=>clk, d=>q_int(1), q=>q_int(2));
15  FF3: dff port map (clk=>clk, d=>q_int(2), q=>q_int(3));
16  FF4: dff port map (clk=>clk, d=>q_int(3), q=>q_int(4));
17  Q <= q_int(4);
18  end architecture;
```




```
20  architecture behav of SRG5 is
21  signal q_int: std_logic_vector(4 downto 0);
22  begin
23    process(clk) begin
24      if rising_edge(clk) then
25          q_int <= q_int(3 downto 0) & sin;
26      end if;
27    end process;
28    Q <= q_int(4);
29  end architecture;
```

Serial In/Parallel Out

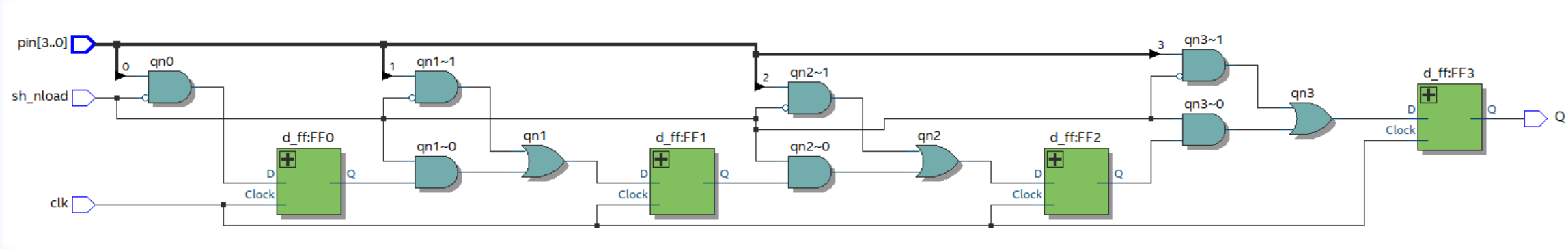
- Data bits are entered serially (least-significant bit first) into a serial in/parallel out shift register in the same manner as in serial in/serial out registers.
- In the parallel output register, the output of each stage is available. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously.

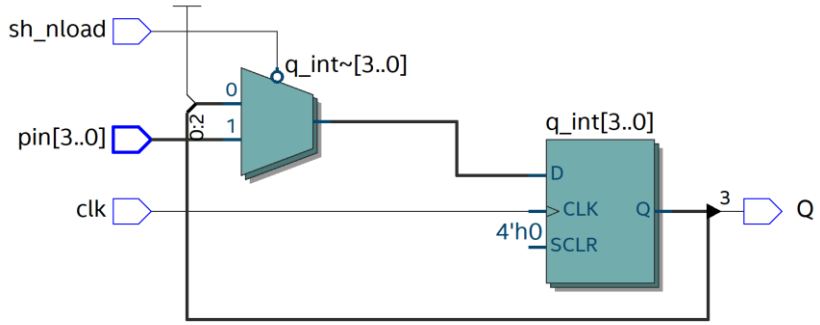



```
4   entity SRG4 is
5       port (sin, clk: in std_logic;
6             Q: out std_logic_vector(3 downto 0) );
7   end entity;
8   --
9   architecture behav of SRG4 is
10  signal q_int: std_logic_vector(3 downto 0);
11  begin
12    process(clk) begin
13      if rising_edge(clk) then
14          q_int <= q_int(2 downto 0) & sin;
15      end if;
16    end process;
17    Q <= q_int;
18  end architecture;
```

Parallel In/Serial Out

- For parallel in data, multiple bits are transferred at one time.
- SHIFT/LOAD input allows all bits of data to load in parallel into the register.

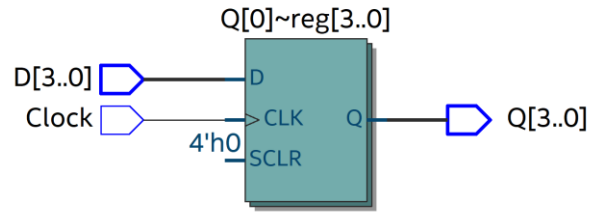




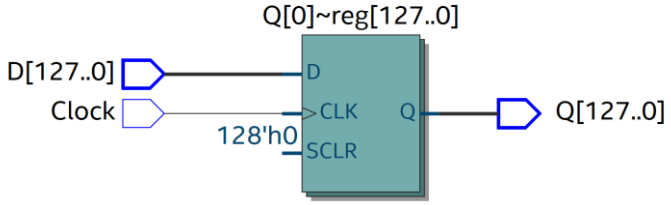

```
entity SRGL4 is
    port (clk: in std_logic;
          pin: in std_logic_vector(3 downto 0);
          sh_nload: in std_logic;
          Q: out std_logic );
end entity;
--
architecture behav of SRGL4 is
signal q_int: std_logic_vector(3 downto 0);
begin
  process(clk) begin
    if rising_edge(clk) then
        if sh_nload = '0' then
            q_int <= pin;
        else
            q_int <= q_int(2 downto 0) & '1';
        end if;
    end if;
  end process;
  Q <= q_int(3);
end architecture;
```

Parallel In/Parallel Out

- Immediately following the simultaneous entry of all data bits, the bits appear on the parallel outputs.



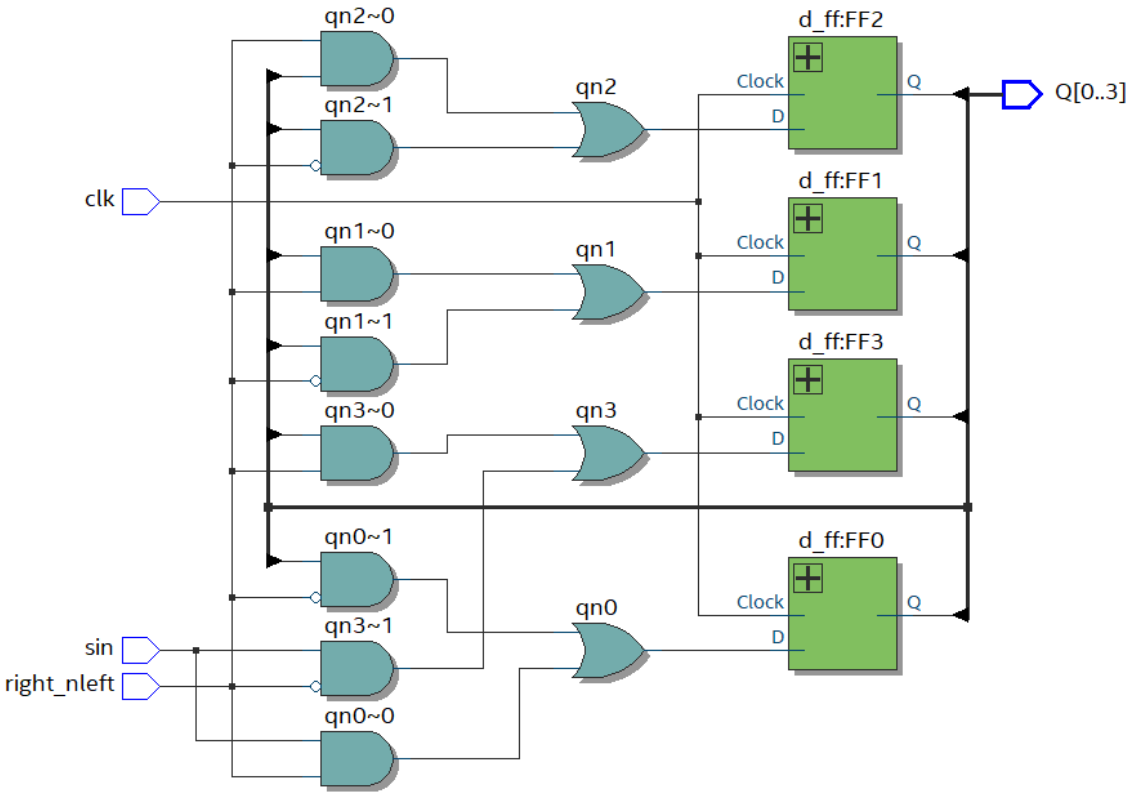

```
30  entity SRD4 is
31   port (Clock: in std_logic;
32         D: in std_logic_vector(3 downto 0);
33         Q: out std_logic_vector(3 downto 0) );
34  end entity SRD4;
35
36  architecture behav of SRD4 is
37  begin
38    process(Clock) begin
39      if rising_edge(Clock) then
40          Q <= D;
41      end if;
42    end process;
43  end architecture;
```

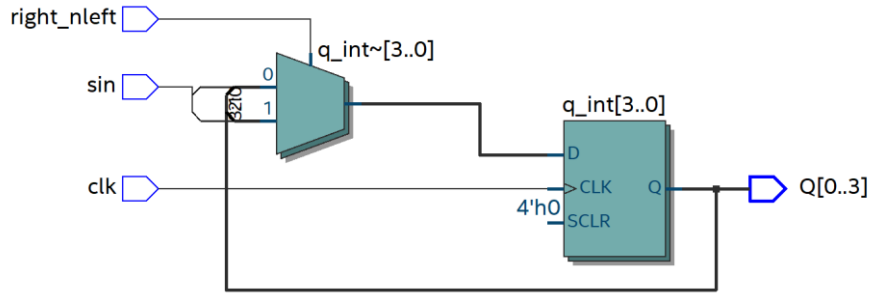



```
46  entity SRDN is
47   generic(N: positive:=128);
48   port (Clock: in std_logic;
49         D: in std_logic_vector(N-1 downto 0);
50         Q: out std_logic_vector(N-1 downto 0) );
51  end entity SRDN;
52
53  architecture behav of SRDN is
54  begin
55    process(Clock) begin
56      if rising_edge(Clock) then
57          Q <= D;
58      end if;
59    end process;
60  end architecture;
```

Bidirectional Shift Registers

- A bidirectional shift register is one in which the data can be shifted either left or right.
- It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left, depending on the level of a control line.

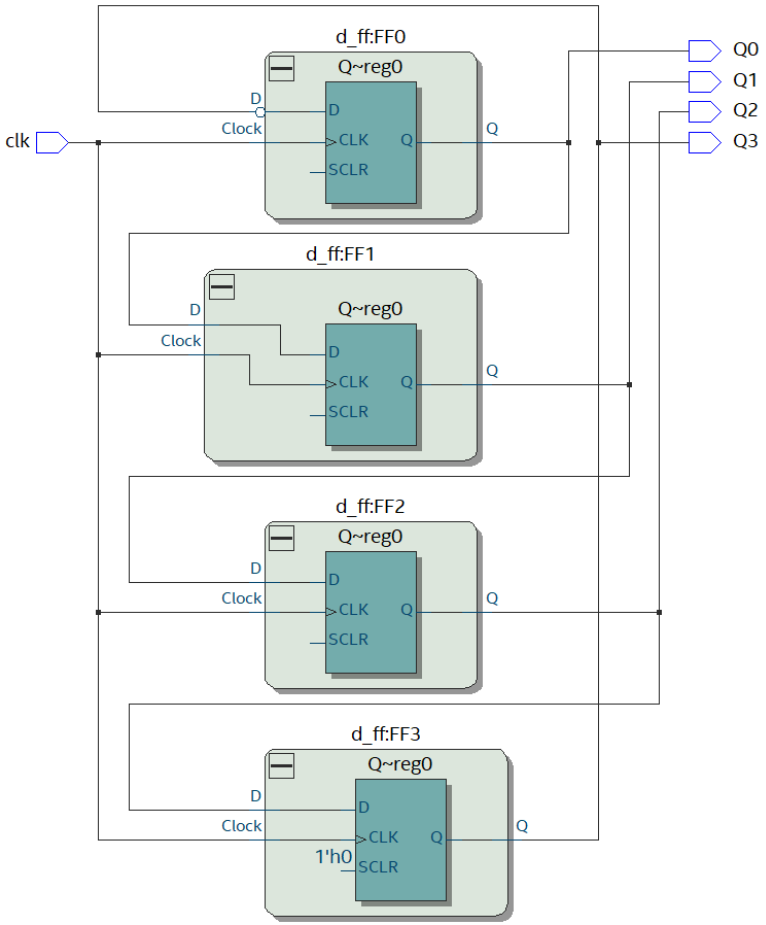





```
4   entity SRG4_BI is
5     port (clk: in std_logic;
6       sin: in std_logic;
7       right_nleft: in std_logic;
8       Q: out std_logic_vector(0 to 3) );
9   end entity;
10    --
11  architecture behav of SRG4_BI is
12  signal q_int: std_logic_vector(Q'range);
13  begin
14    process(clk) begin
15      if rising_edge(clk) then
16        if right_nleft = '1' then
17          q_int <= sin & q_int(0 to 2);
18        else
19          q_int <= q_int(1 to 3) & sin;
20        end if;
21      end if;
22    end process;
23    Q <= q_int;
24  end architecture;
```

Johnson Counter

- In a Johnson counter the complement of the output of the last flip-flop is connected back to the D input of the first flip-flop.
- The 4-bit sequence has a total of eight states, or bit patterns. In general, a Johnson counter will produce a modulus of 2n, where n is the number of stages in the counter.



Four-bit Johnson sequence.

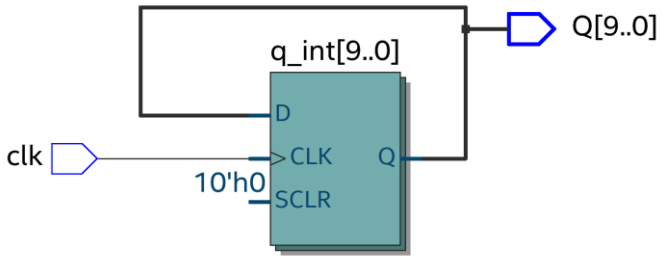
Clock Pulse	Q_0	Q_1	Q_2	Q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

```
5  entity SR4_JOHN is
6   port (clk: in std_logic;
7         Q: out std_logic_vector(3 downto 0)
8         );
9  end entity;
10 --
11 architecture behav of SR4_JOHN is
12 signal q_int: std_logic_vector(3 downto 0):=x"0";
13 begin
14   process(clk) begin
15     if rising_edge(clk) then
16         q_int <= q_int(2 downto 0) & not(q_int(3));
17     end if;
18   end process;
19   Q <= q_int;
20 end architecture;
```


source: Thomas Floyd: Digital Fundamentals

Ring Counter

A ring counter utilizes one flip-flop for each state in its sequence. It has the advantage that decoding gates are not required. In the case of a 10-bit ring counter, there is a unique output for each decimal digit.



Ten-bit ring counter sequence.

Clock Pulse	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

```
27  entity SR10_RING is
28   port (
29         clk: in std_logic;
30         Q: out std_logic_vector(9 downto 0)
31         );
32  end entity;
33
34
```

```
37  architecture behav of SR10_RING is
38   signal q_int: std_logic_vector(9 downto 0):=(9=>'1',others=>'0');
39  begin
40    process(clk) begin
41      if rising_edge(clk) then q_int <= q_int(8 downto 0) & q_int(9); end if;
42    end process;
43    Q <= q_int;
44  end architecture;
```

source: Thomas Floyd: Digital Fundamentals

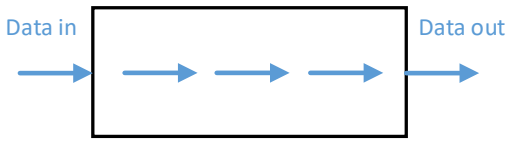
Digital Logic
Design with FPGA

Digital Design with VHDL

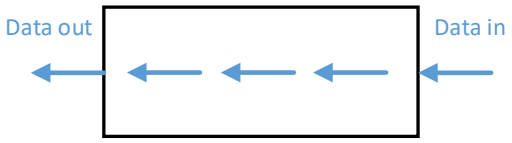
Sequential Logic 2

Shift Register Operations

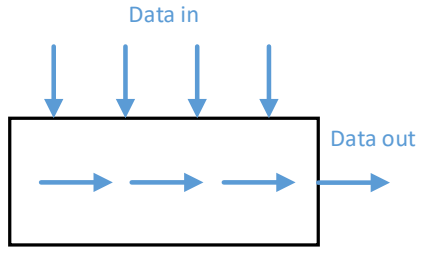
- A register is a digital circuit with two basic functions: data storage and data movement.
  The storage capability of a register makes it an important type of memory device.
- The storage capacity of a register is the total number of bits (1s and 0s) of digital data it can retain. Each stage (flip-flop) in a shift register represents one bit of storage capacity; therefore, the number of stages in a register determines its storage capacity.
- The shift capability of a register permits the movement of data from stage to stage within the register or into or out of the register upon application of clock pulses.



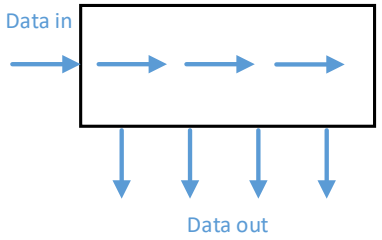
(a) Serial in/shift right/serial out



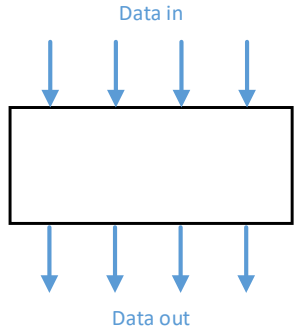
(b) Serial in/shift left/serial out



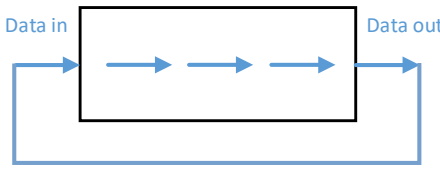
(c) Parallel in/serial out



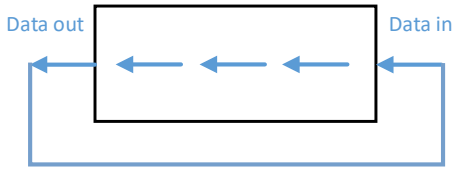
(d) Serial in/parallel out



(e) Parallel in/parallel out



(f) Rotate right

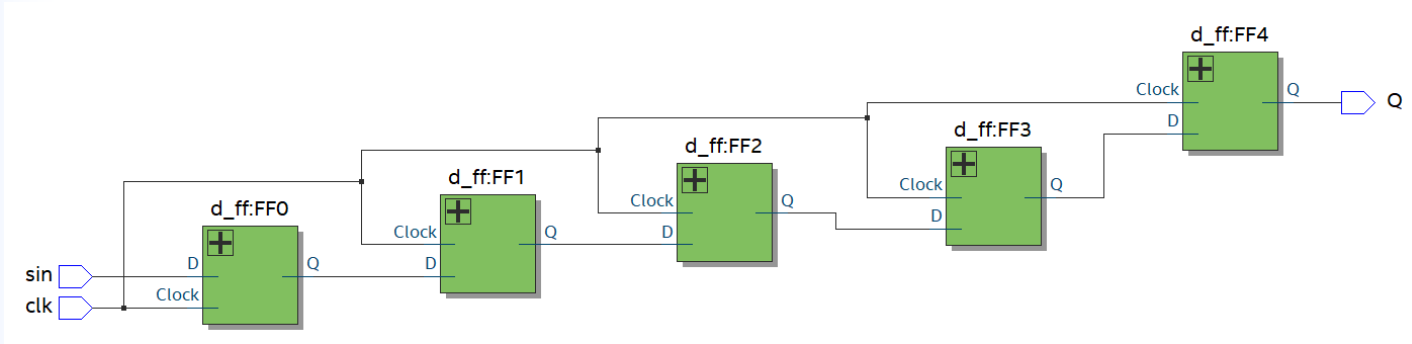


(g) Rotate left

source: Thomas Floyd: Digital Fundamentals

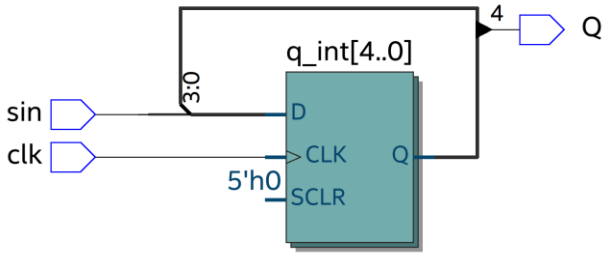
Serial In/Serial Out

The serial in/serial out shift register accepts data serially - one bit at a time on a single line.
It produces the stored information on its output also in serial form.




```
4   entity SRG5 is
5       port (sin, clk: in std_logic;
6             Q: out std_logic );
7   end entity;
```

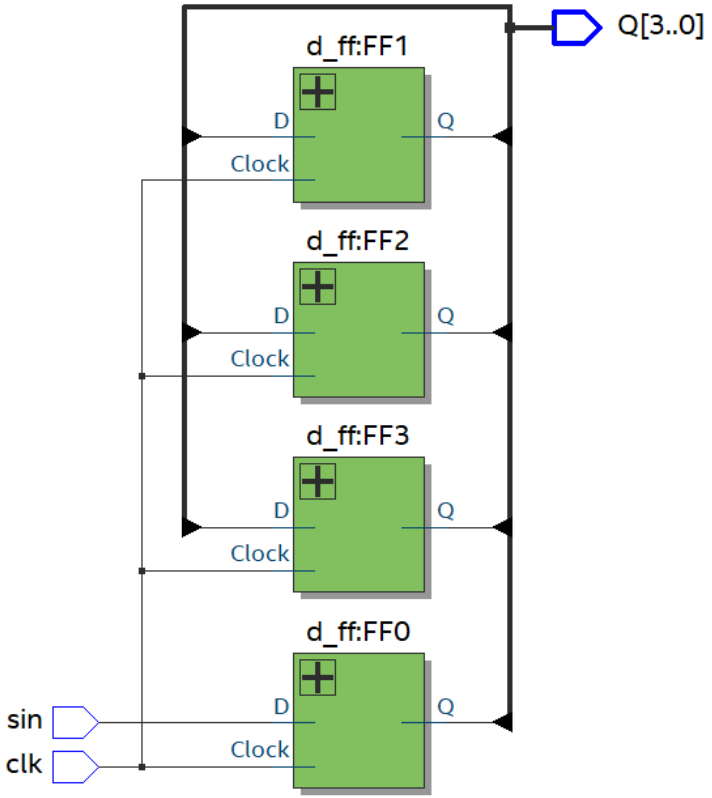
```
9   architecture struct of SRG5 is
10  signal q_int: std_logic_vector(4 downto 0);
11  begin
12  FF0: dff port map (clk=>clk, d=>sin, q=>q_int(0));
13  FF1: dff port map (clk=>clk, d=>q_int(0), q=>q_int(1));
14  FF2: dff port map (clk=>clk, d=>q_int(1), q=>q_int(2));
15  FF3: dff port map (clk=>clk, d=>q_int(2), q=>q_int(3));
16  FF4: dff port map (clk=>clk, d=>q_int(3), q=>q_int(4));
17  Q <= q_int(4);
18  end architecture;
```




```
20  architecture behav of SRG5 is
21  signal q_int: std_logic_vector(4 downto 0);
22  begin
23    process(clk) begin
24      if rising_edge(clk) then
25          q_int <= q_int(3 downto 0) & sin;
26      end if;
27    end process;
28    Q <= q_int(4);
29  end architecture;
```

Serial In/Parallel Out

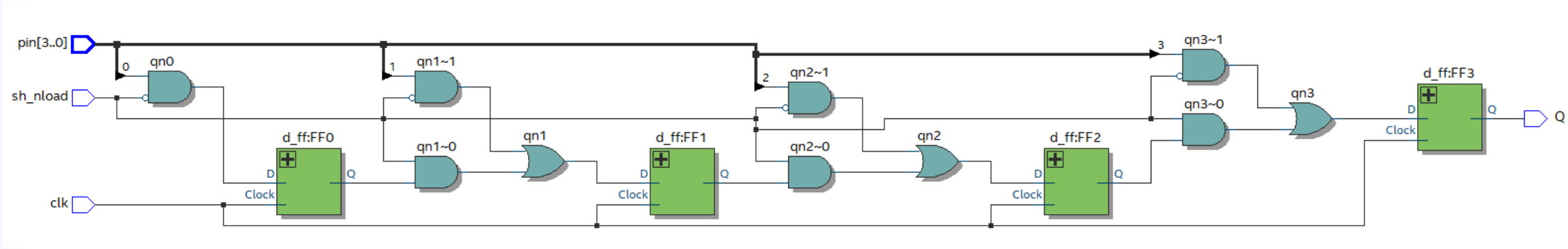
- Data bits are entered serially (least-significant bit first) into a serial in/parallel out shift register in the same manner as in serial in/serial out registers.
- In the parallel output register, the output of each stage is available. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously.

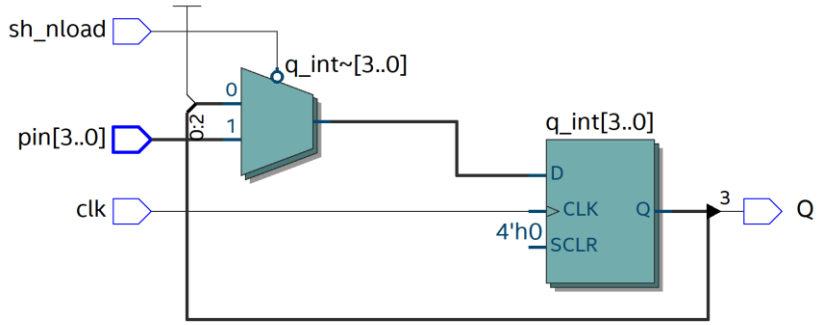



```
4   entity SRG4 is
5       port (sin, clk: in std_logic;
6             Q: out std_logic_vector(3 downto 0) );
7   end entity;
8   --
9   architecture behav of SRG4 is
10  signal q_int: std_logic_vector(3 downto 0);
11  begin
12    process(clk) begin
13      if rising_edge(clk) then
14          q_int <= q_int(2 downto 0) & sin;
15      end if;
16    end process;
17    Q <= q_int;
18  end architecture;
```

Parallel In/Serial Out

- For parallel in data, multiple bits are transferred at one time.
- SHIFT/LOAD input allows all bits of data to load in parallel into the register.

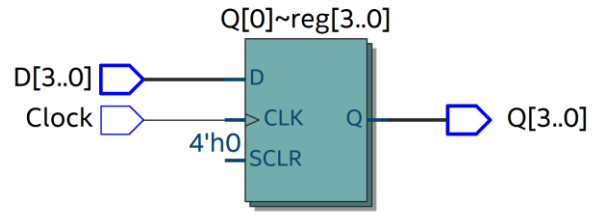




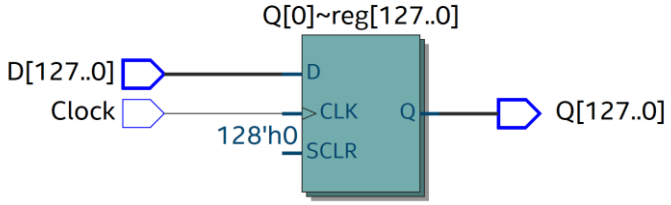

```
entity SRGL4 is
    port (clk: in std_logic;
          pin: in std_logic_vector(3 downto 0);
          sh_nload: in std_logic;
          Q: out std_logic );
end entity;
--
architecture behav of SRGL4 is
signal q_int: std_logic_vector(3 downto 0);
begin
  process(clk) begin
    if rising_edge(clk) then
        if sh_nload = '0' then
            q_int <= pin;
        else
            q_int <= q_int(2 downto 0) & '1';
        end if;
    end if;
  end process;
  Q <= q_int(3);
end architecture;
```

Parallel In/Parallel Out

- Immediately following the simultaneous entry of all data bits, the bits appear on the parallel outputs.



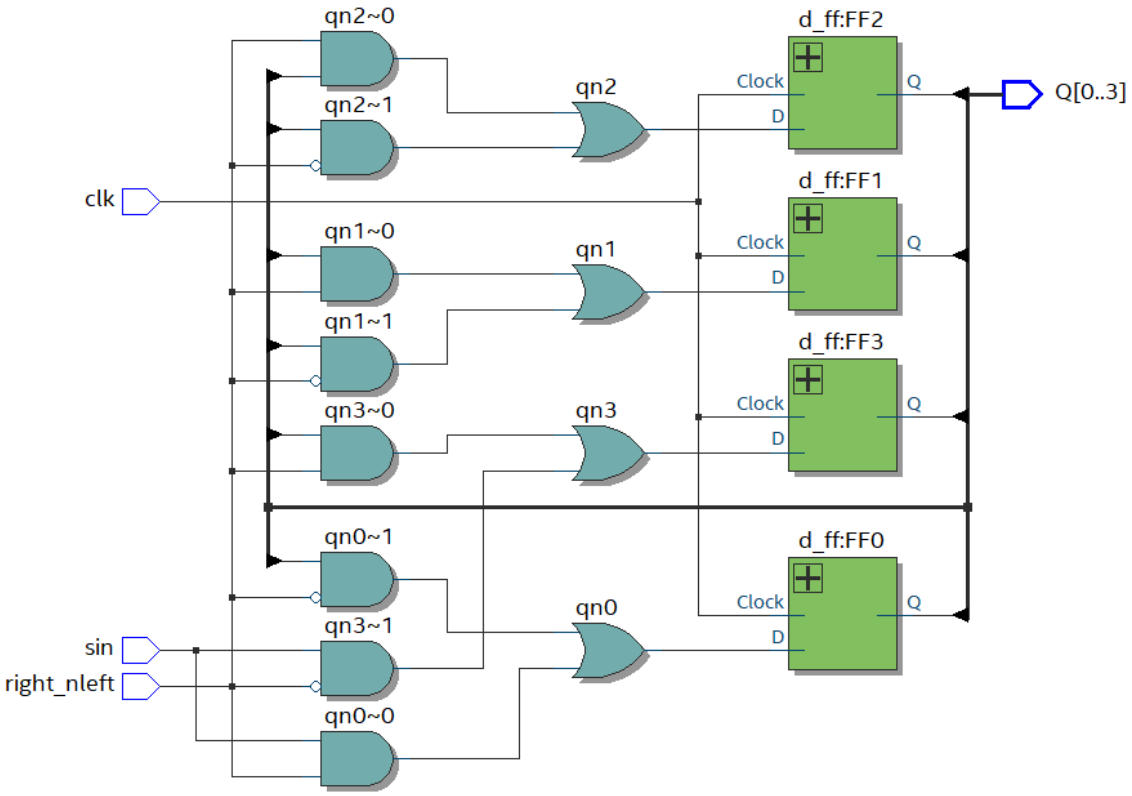

```
30  entity SRD4 is
31   port (Clock: in std_logic;
32         D: in std_logic_vector(3 downto 0);
33         Q: out std_logic_vector(3 downto 0) );
34  end entity SRD4;
35
36  architecture behav of SRD4 is
37  begin
38    process(Clock) begin
39      if rising_edge(Clock) then
40          Q <= D;
41      end if;
42    end process;
43  end architecture;
```

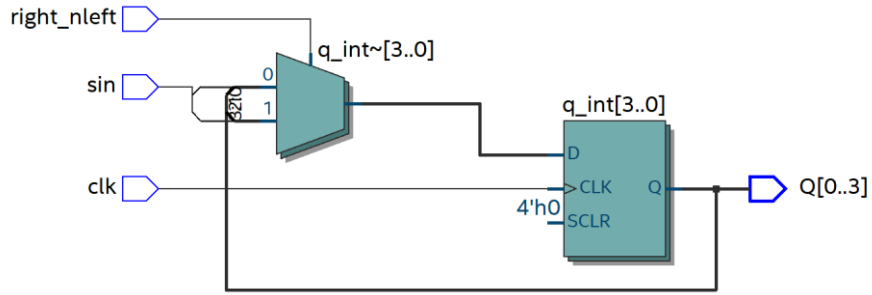



```
46  entity SRDN is
47   generic(N: positive:=128);
48   port (Clock: in std_logic;
49         D: in std_logic_vector(N-1 downto 0);
50         Q: out std_logic_vector(N-1 downto 0) );
51  end entity SRDN;
52
53  architecture behav of SRDN is
54  begin
55    process(Clock) begin
56      if rising_edge(Clock) then
57          Q <= D;
58      end if;
59    end process;
60  end architecture;
```

Bidirectional Shift Registers

- A bidirectional shift register is one in which the data can be shifted either left or right.
- It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left, depending on the level of a control line.

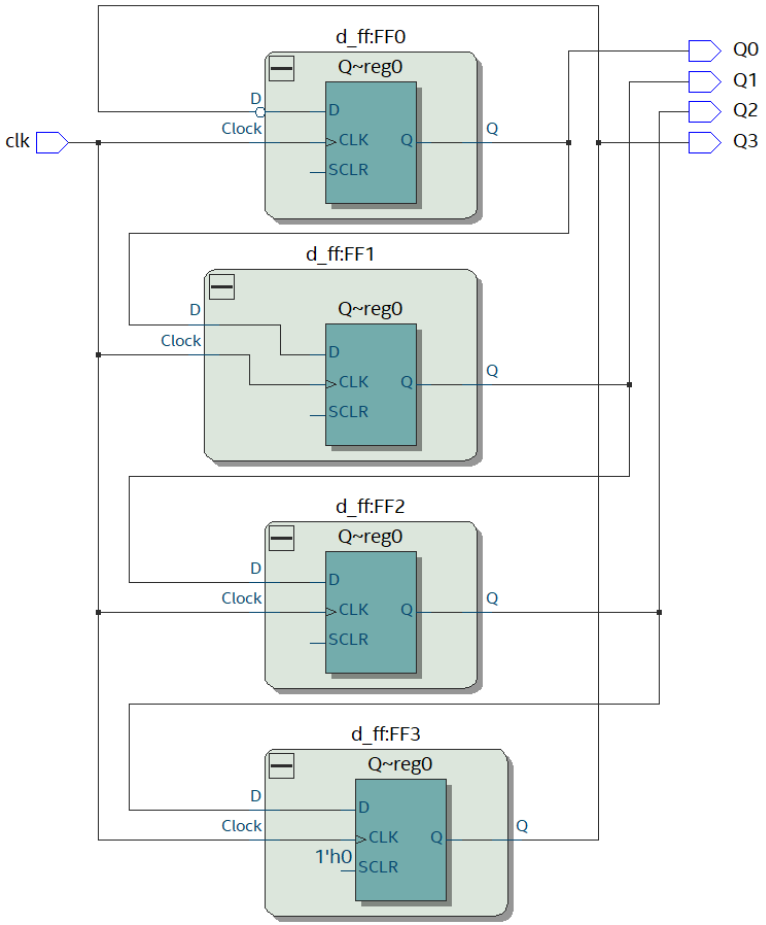





```
4   entity SRG4_BI is
5     port (clk: in std_logic;
6       sin: in std_logic;
7       right_nleft: in std_logic;
8       Q: out std_logic_vector(0 to 3) );
9   end entity;
10    --
11  architecture behav of SRG4_BI is
12  signal q_int: std_logic_vector(Q'range);
13  begin
14    process(clk) begin
15      if rising_edge(clk) then
16        if right_nleft = '1' then
17          q_int <= sin & q_int(0 to 2);
18        else
19          q_int <= q_int(1 to 3) & sin;
20        end if;
21      end if;
22    end process;
23    Q <= q_int;
24  end architecture;
```

Johnson Counter

- In a Johnson counter the complement of the output of the last flip-flop is connected back to the D input of the first flip-flop.
- The 4-bit sequence has a total of eight states, or bit patterns. In general, a Johnson counter will produce a modulus of 2n, where n is the number of stages in the counter.



Four-bit Johnson sequence.

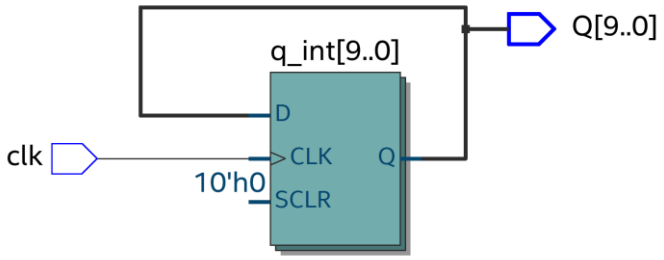
Clock Pulse	Q_0	Q_1	Q_2	Q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

```
5  entity SR4_JOHN is
6   port (clk: in std_logic;
7         Q: out std_logic_vector(3 downto 0)
8         );
9  end entity;
10 --
11 architecture behav of SR4_JOHN is
12 signal q_int: std_logic_vector(3 downto 0):=x"0";
13 begin
14   process(clk) begin
15     if rising_edge(clk) then
16         q_int <= q_int(2 downto 0) & not(q_int(3));
17     end if;
18   end process;
19   Q <= q_int;
20 end architecture;
```


source: Thomas Floyd: Digital Fundamentals

Ring Counter

A ring counter utilizes one flip-flop for each state in its sequence. It has the advantage that decoding gates are not required. In the case of a 10-bit ring counter, there is a unique output for each decimal digit.



Ten-bit ring counter sequence.

Clock Pulse	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

```
27  entity SR10_RING is
28   port (
29         clk: in std_logic;
30         Q: out std_logic_vector(9 downto 0)
31         );
32  end entity;
33
34
```

```
37  architecture behav of SR10_RING is
38   signal q_int: std_logic_vector(9 downto 0):=(9=>'1',others=>'0');
39  begin
40    process(clk) begin
41      if rising_edge(clk) then q_int <= q_int(8 downto 0) & q_int(9); end if;
42    end process;
43    Q <= q_int;
44  end architecture;
```

source: Thomas Floyd: Digital Fundamentals

Digital Logic
Design with FPGA

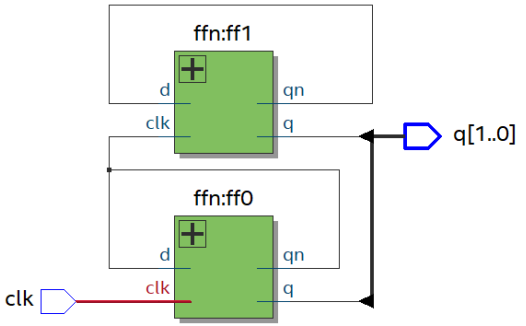
Digital Design with VHDL

Sequential Logic 3

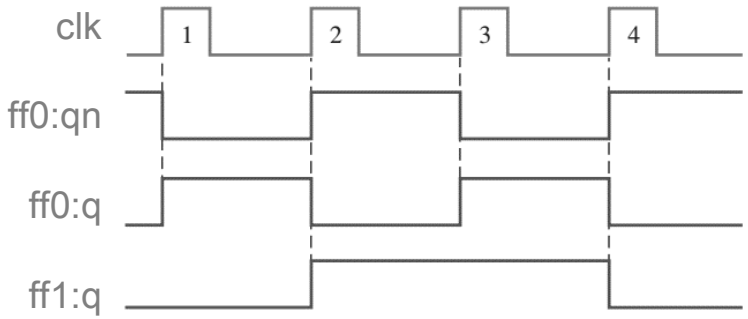
Outline

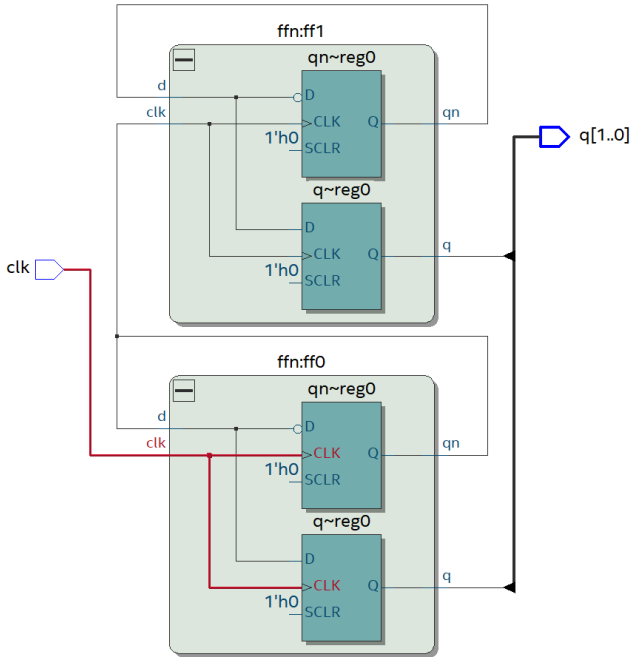
- Asynchronous Counters
- Finite State Machines (FSMs)
- Synchronous Counters
- Up/Down Synchronous Counters
- Cascaded Counters
- Counter Applications

Asynchronous Counters



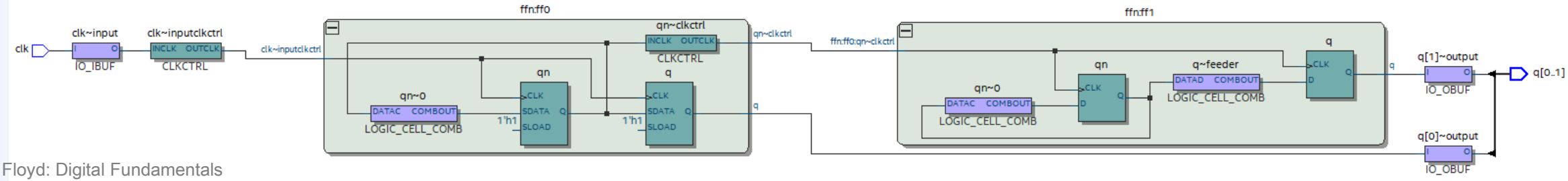
- The term asynchronous refers to events that do not have a fixed time relationship with each other and do not occur at the same time.
- An asynchronous counter is one in which the flip-flops within the counter do not change states at exactly the same time because they do not have a common clock pulse.





2-Bit Asynchronous Binary Counter

- clk is applied to the clock input (clk) of only the first flip-flop, ff0, which is always the least significant bit (LSB). The second flip-flop, ff1, is triggered by the qn (inverted) output of ff0.
- Because of the inherent propagation delay time through a flip-flop, a transition of the input clock pulse (clk) and a transition of the qn output of ff0 can never occur at exactly the same time.



source: Thomas Floyd: Digital Fundamentals

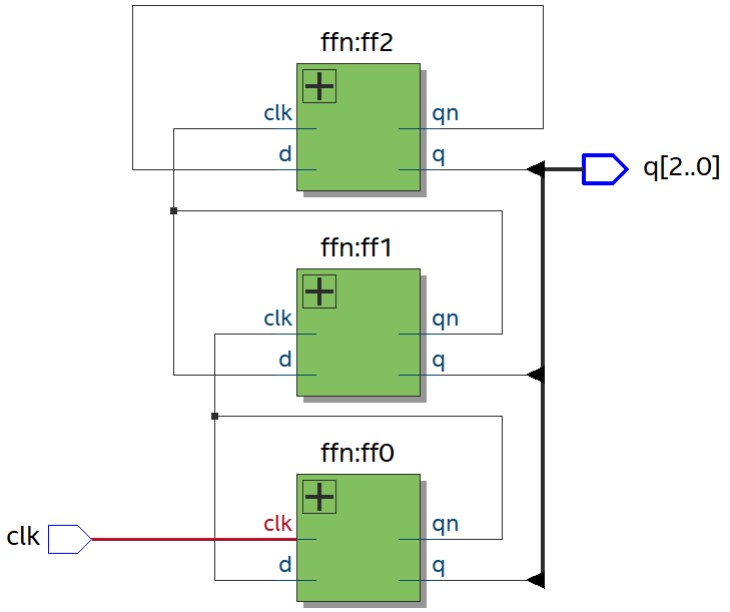
Asynchronous Counters

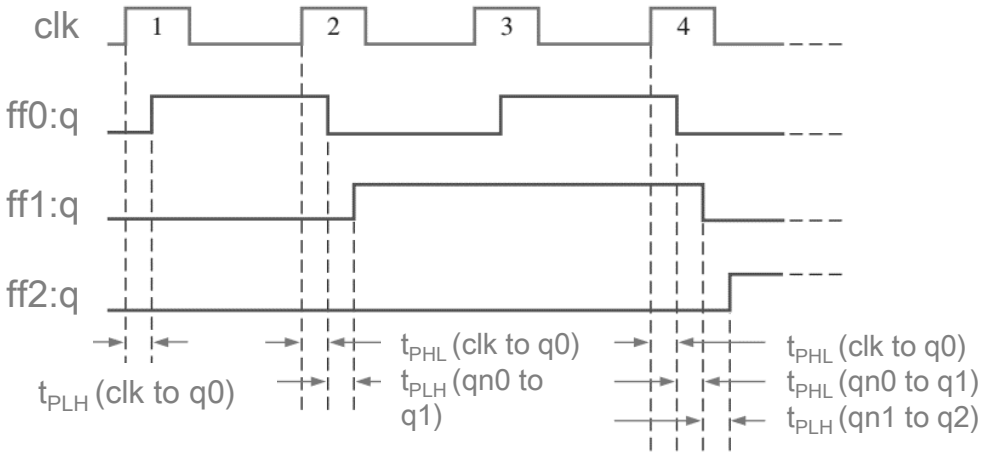
3-Bit Asynchronous Binary Counter

- This counter can be easily expanded for higher count, by connecting additional toggle flip-flops.

- Asynchronous counters are commonly referred to as ripple counters for the following reason: the effect of the input clock pulse is first “felt” by ff0. This effect cannot get to ff1 immediately because of the propagation delay through ff0. Then there is the propagation delay through ff1 before ff2 can be triggered. Thus, the effect of an input clock pulse “ripples” through the counter, due to propagation delays, to reach the last flip-flop.

- Cumulative delay of an asynchronous counter is a major disadvantage because it limits the rate at which the counter can be clocked and creates decoding problems.
- The maximum cumulative delay in a counter must be less than the period of the clock waveform.





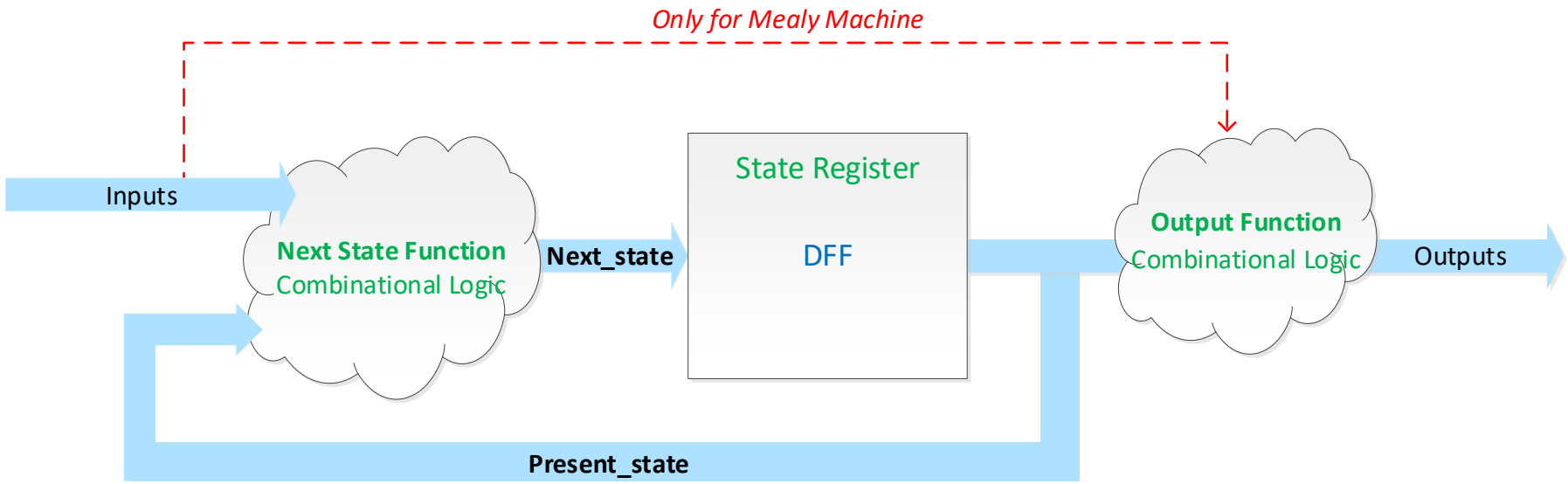
source: Thomas Floyd: Digital Fundamentals

FSMs

- A state machine is a sequential circuit having a limited (finite) number of states occuring in a prescribed order.
- A counter is an example of a state machine; the number of states is called the modulus.

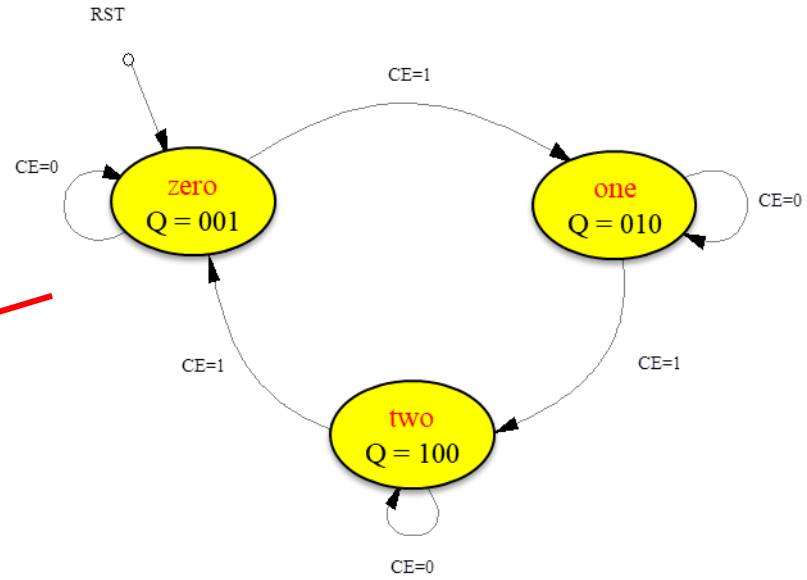
Two basic types of state machines are the Moore and the Mealy.

- The Moore state machine is one where the outputs depend only on the internal present state.
- The Mealy state machine is one where the outputs depend on both the internal present state and on the inputs.



source: Thomas Floyd: Digital Fundamentals

FSM Synthesis



- Next-state table:

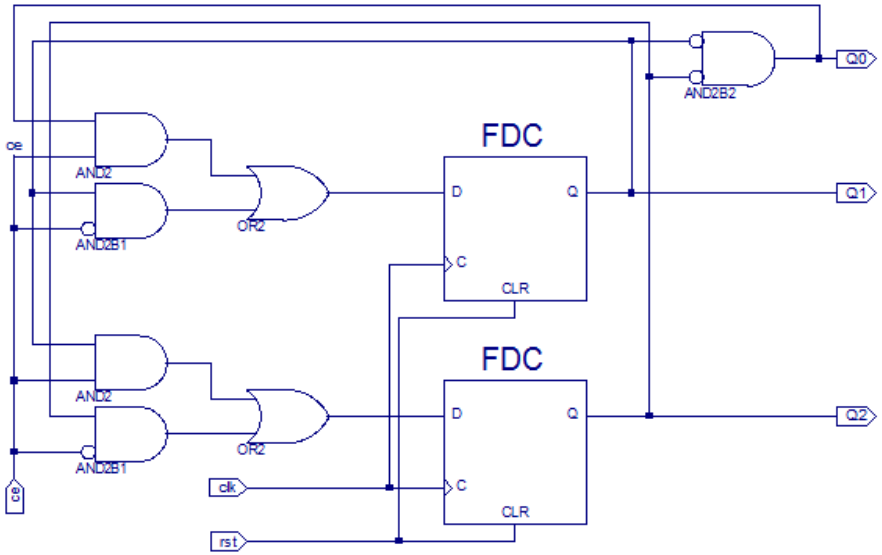
Present state	Next state		Output
	ce		
	0	1	
zero	zero	one	001
one	one	two	010
two	two	zero	100

- The simplest coding example: zero = 00, one = 01, two = 10.

Present state y1 y0	Next state y1p y0p		Output Q2 Q1 Q0
	ce		
	0	1	
0 0	0 0	0 1	001
0 1	0 1	1 0	010
1 0	1 0	0 0	100

- Next state and output equations:

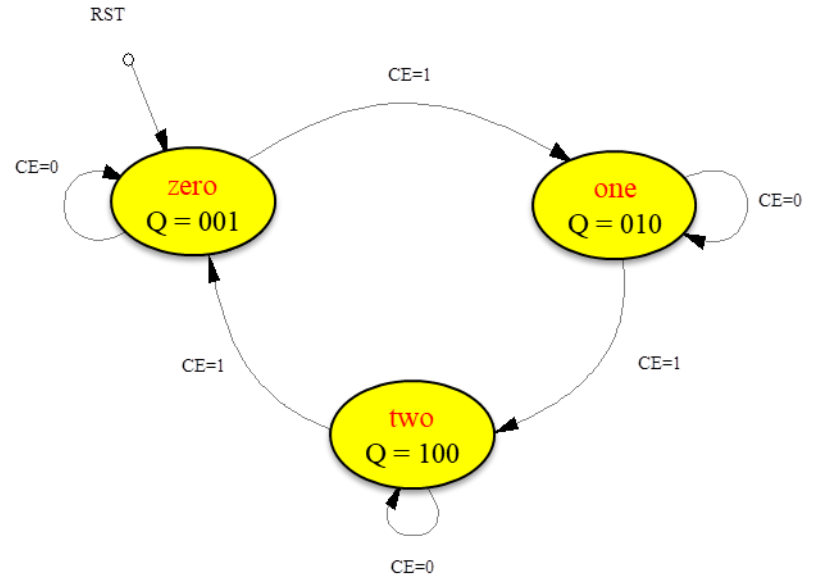
$$y0p = \overline{ce}\ y0 + ce\ \overline{y1}\ \overline{y0}$$
$$y1p = \overline{ce}\ y1 + ce\ y0$$
$$Q0 = \overline{y1}\ \overline{y0}$$
$$Q1 = y0$$
$$Q2 = y1$$

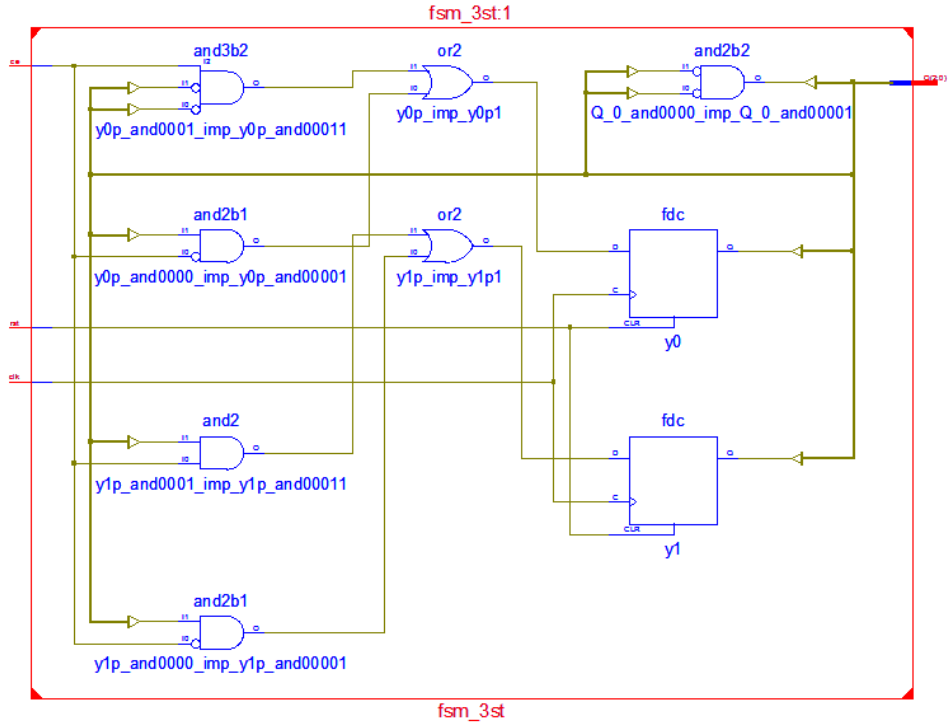


FSM Synthesis

$$y0p = \overline{ce}\ y0 + ce\ \overline{y1}\ \overline{y0}$$
$$y1p = \overline{ce}\ y1 + ce\ y0$$
$$Q0 = \overline{y1}\ \overline{y0}$$
$$Q1 = y0$$
$$Q2 = y1$$

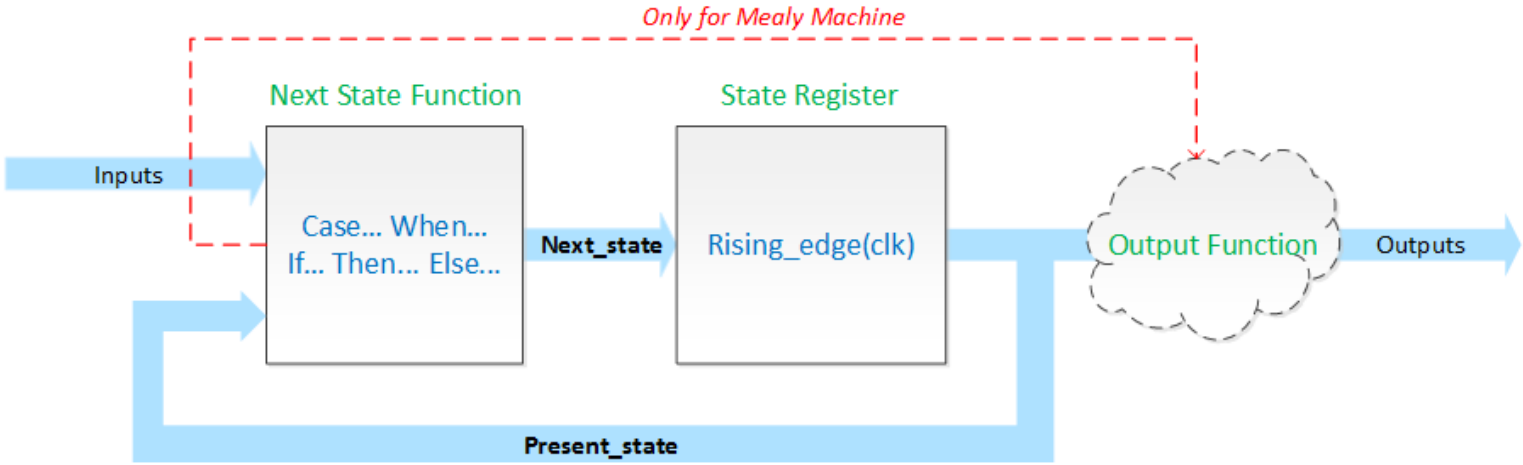
```
4   entity fsm_3st is
5    port( clk, rst, ce: in std_logic;
6          Q: out std_logic_vector(2 downto 0) );
7   end;
8
9   architecture struct of fsm_3st is
10   signal y0p, y1p, y0, y1: std_logic;
11  begin
12   -- next state logic
13   y0p <= (not ce and y0) or
14          (ce and not y0 and not y1);
15   y1p <= (not ce and y1) or
16          (ce and y0);
17
18   -- sate register
19  state_mem: process(clk,rst) begin
20    if rst='1' then
21       y0<='0'; y1<='0';
22    elsif (clk'event and clk='1') then
23       y0<=y0p; y1<=y1p;
24    end if;
25  end process;
26
27   -- output logic
28   Q(0)<=(not y0 and not y1);
29   Q(1)<=y0;
30   Q(2)<=y1;
31  end architecture;
```





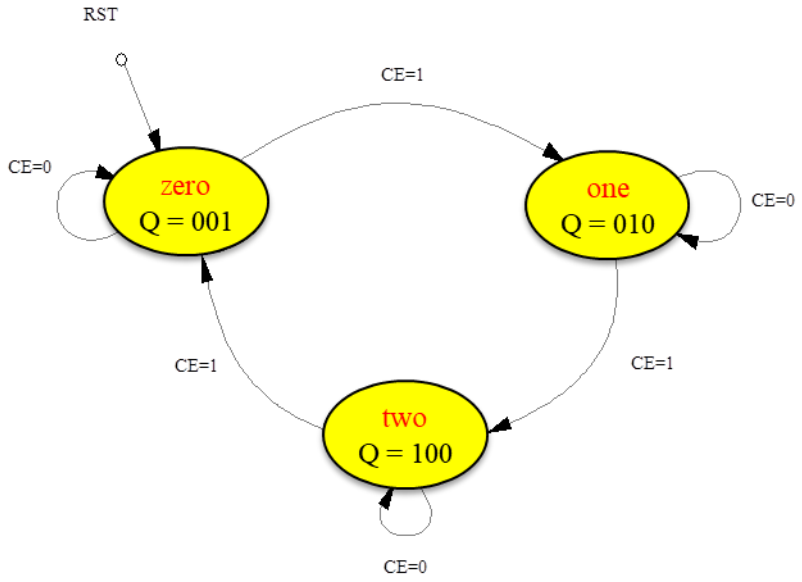
FSMs – VHDL design guidelines

- For HDL, process is the best way to describe FSM components
- Next state equations can be described directly in the sequential process or in a distinct combinatorial process. The simplest coding example is based on a case statement.
- A state register can be a different type (such as: integer, bit_vector, std_logic_vector). It is common and convenient to define an enumerated type containing all possible state values and to declare state register with that type.
- Non-registered outputs are described either in the combinatorial process or in concurrent assignments.
- Registered outputs must be assigned within the sequential process.
- Registered inputs are described using internal signals, which are assigned in the sequential process.



FSMs - VHDL

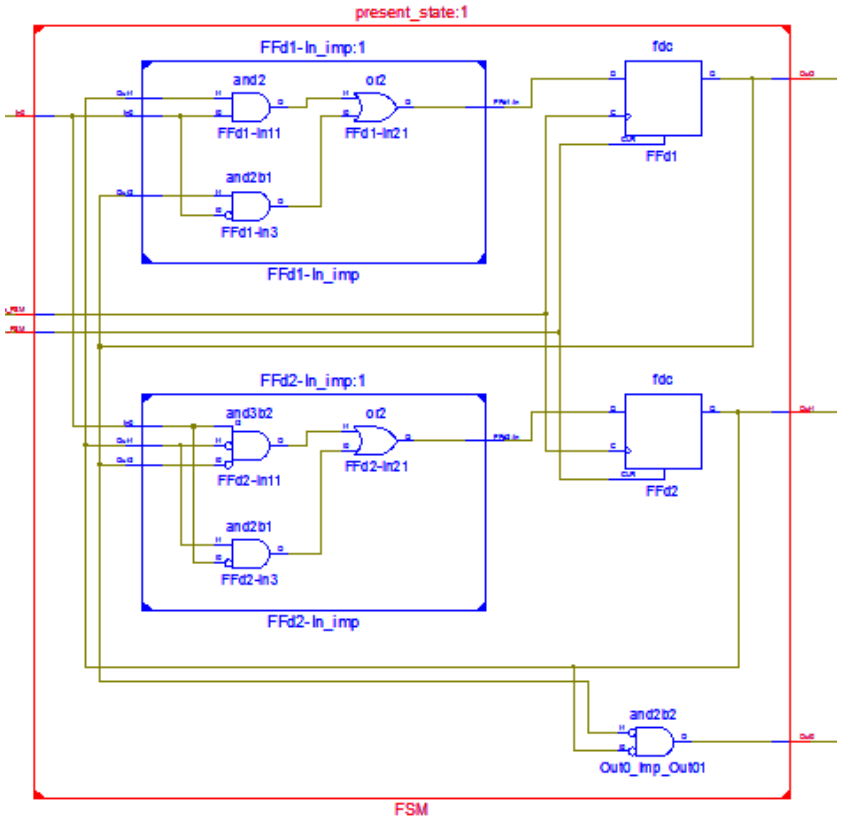
```vhdl
entity fsm_3st is
 port( clk, rst, ce: in std_logic;
       Q: out std_logic_vector(2 downto 0) );
end;

architecture behav of fsm_3st is
 type state is (zero,one,two);
 signal present_state, next_state: state;
begin
```



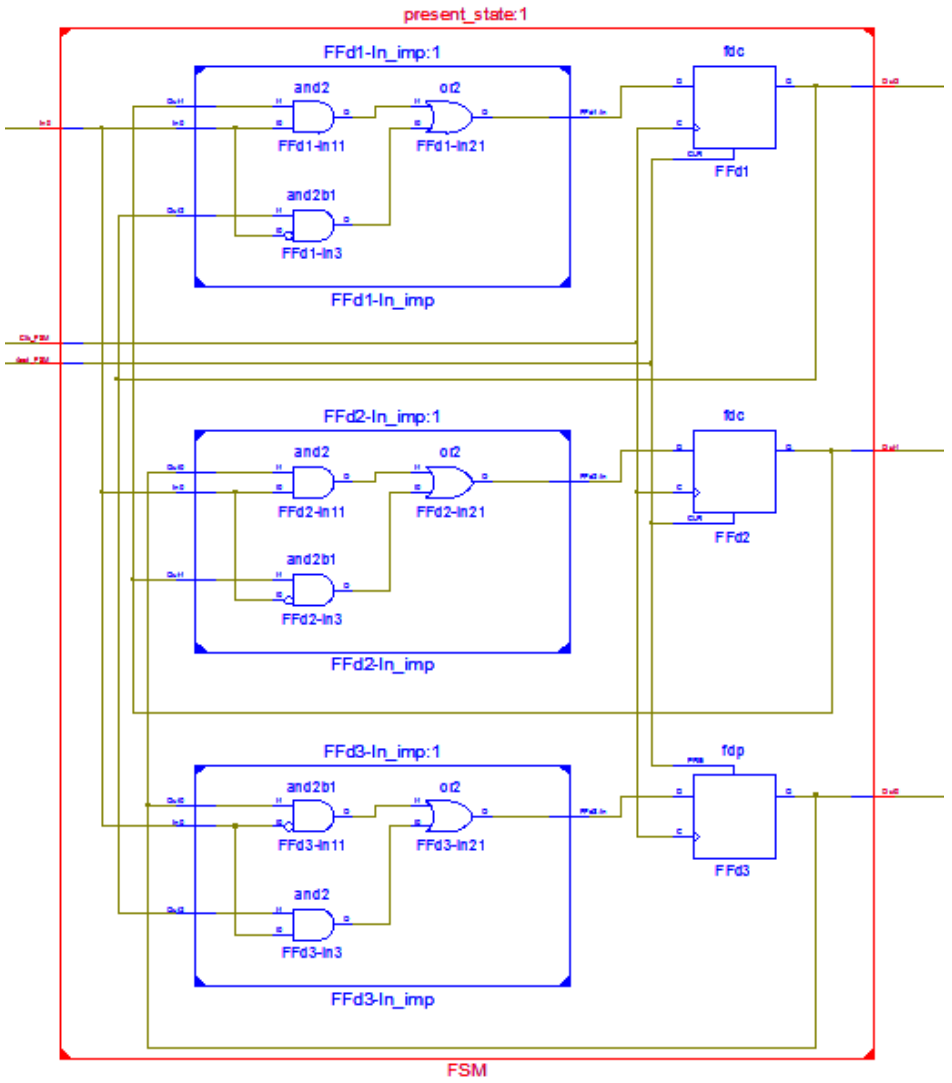

```vhdl
fsm: process(present_state,ce) begin
  case present_state is
    when zero =>
      Q <= "001";
      if ce='1' then
        next_state <= one;
      else
        next_state <= zero;
      end if;
    when one =>
      Q <= "010";
      if ce='1' then
        next_state <= two;
      else
        next_state <= one;
      end if;
    when two =>
      Q <= "100";
      if ce='1' then
        next_state <= zero;
      else
        next_state <= two;
      end if;
  end case;
end process;
state_mem: process(clk,rst) begin
  if rst='1' then
     present_state <= zero;
  elsif (clk'event and clk='1') then
     present_state <= next_state;
  end if;
end process;
end architecture;
```

FSMs - VHDL

Binary encoding



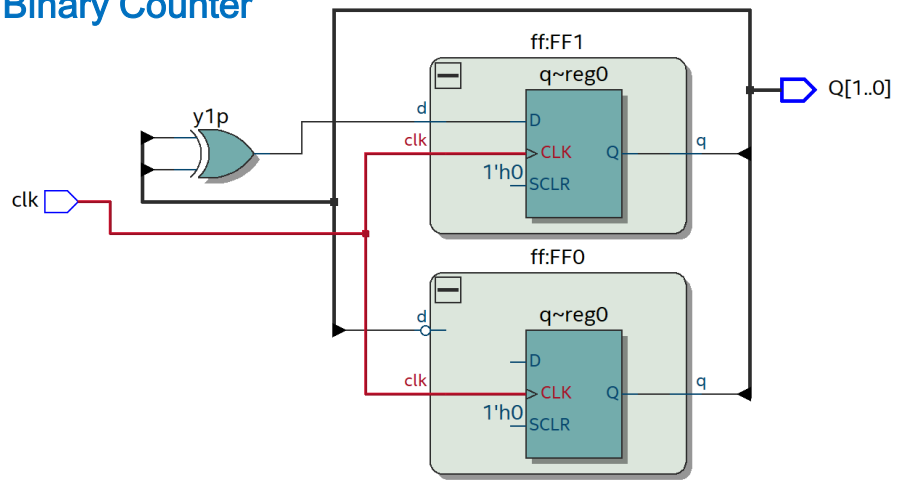
One-hot encoding



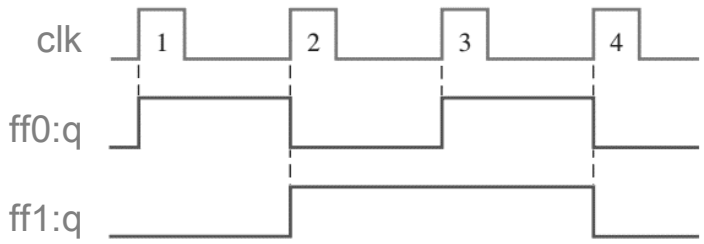
Synchronous Counters

- The term synchronous refers to events that have a fixed time relationship with each other.
- A synchronous counter is one in which all the flip-flops in the counter are clocked at the same time by a common clock pulse.

2-Bit Synchronous Binary Counter




```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity cntr2b is
 port(clk: in std_logic;
      Q: out std_logic_vector(1 downto 0));
end;
architecture behav of cntr2b is
 signal y0,y0p,y1,y1p: std_logic;
begin
FF0: entity work.dff
       port map(clk=>clk, d=>y0p, q=>y0);
FF1: entity work.dff
       port map(clk=>clk, d=>y1p, q=>y1);
y0p <= not y0;
y1p <= y1 xor y0;
Q(0) <= y0; Q(1) <= y1;
end architecture;
```

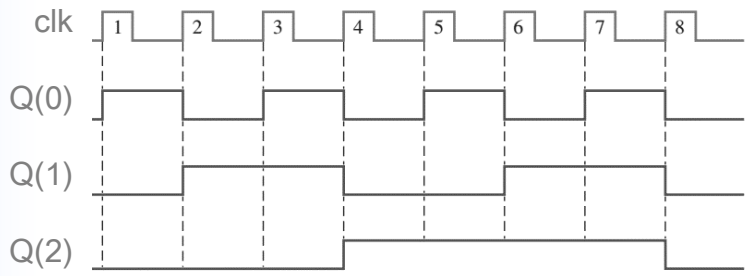



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

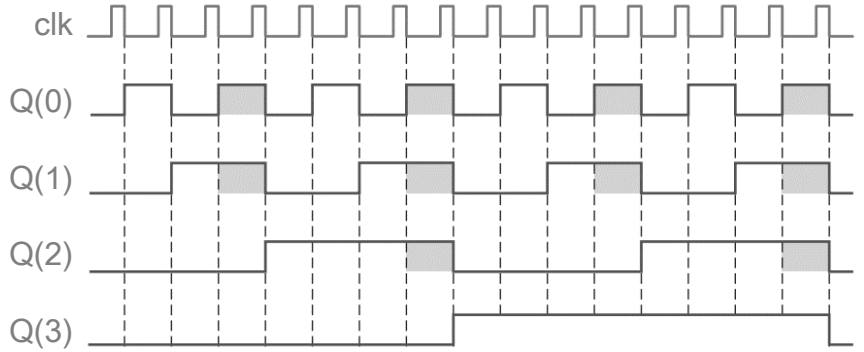
```vhdl
architecture behav of cntr2b is
 signal cntr: std_logic_vector(Q'range):="00";
begin
cntr_proc: process(clk) begin
  if rising_edge(clk) then
     cntr <= cntr +1;
  end if;
end process;
Q <= cntr;
end architecture;
```

Synchronous Counters

3-Bit Synchronous Binary Counter



4-Bit Synchronous Binary Counter



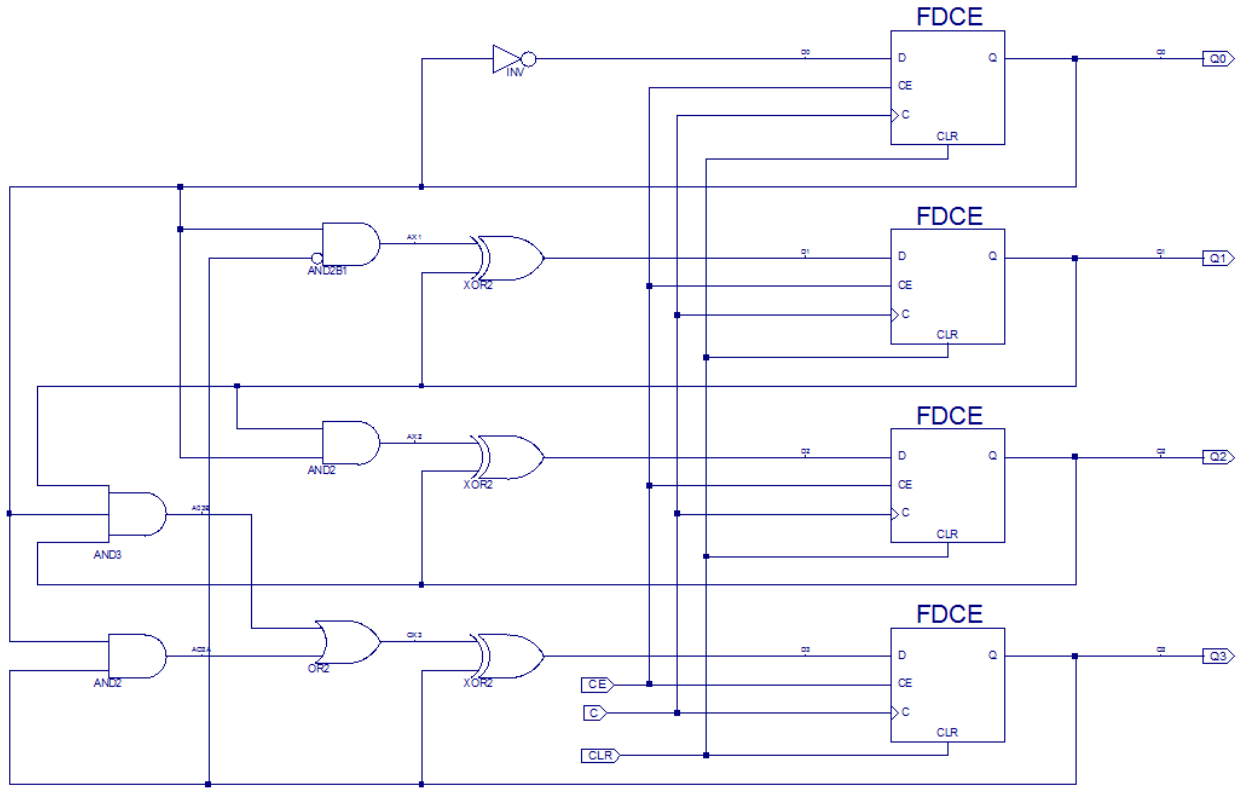

```
6  entity cntrNb_re is
7    generic(N: positive:=3);
8    port(clk: in std_logic;
9         Q: out std_logic_vector(N-1 downto 0));
10 end;
11 architecture behav of cntrNb_re is
12   signal cntr: std_logic_vector(Q'range):=(others=>'0');
13 begin
14 cntr_proc: process(clk) begin
15   if rising_edge(clk) then
16      cntr <= cntr +1;
17   end if;
18 end process;
19 Q <= cntr;
20 end architecture;
```

```
6  entity cntrNb_fe is
7    generic(N: positive:=4);
8    port(clk: in std_logic;
9         Q: out std_logic_vector(N-1 downto 0));
10 end;
11 architecture behav of cntrNb_fe is
12   signal cntr: std_logic_vector(Q'range):=(others=>'0');
13 begin
14 cntr_proc: process(clk) begin
15   if falling_edge(clk) then
16      cntr <= cntr +1;
17   end if;
18 end process;
19 Q <= cntr;
20 end architecture;
```


source: Thomas Floyd: Digital Fundamentals

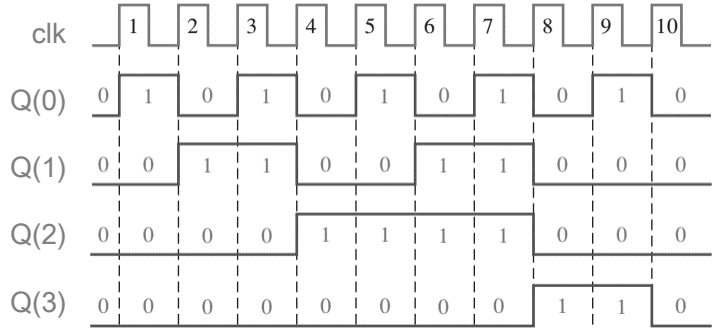
Synchronous Counters

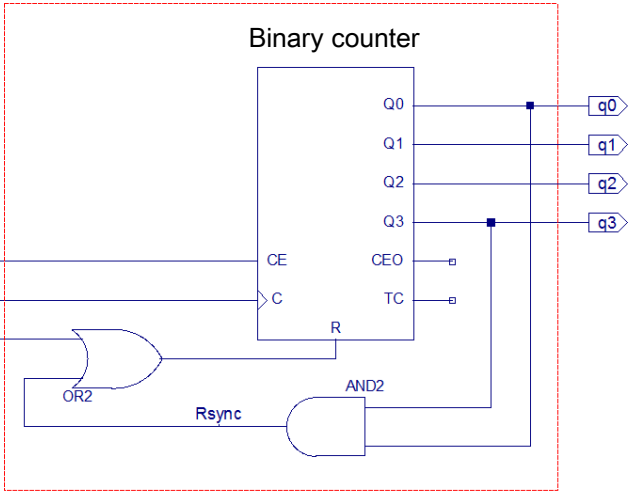
4-Bit Synchronous Decade Counter



States of a BCD decade counter.

Clock Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 (recycles)	0	0	0	0



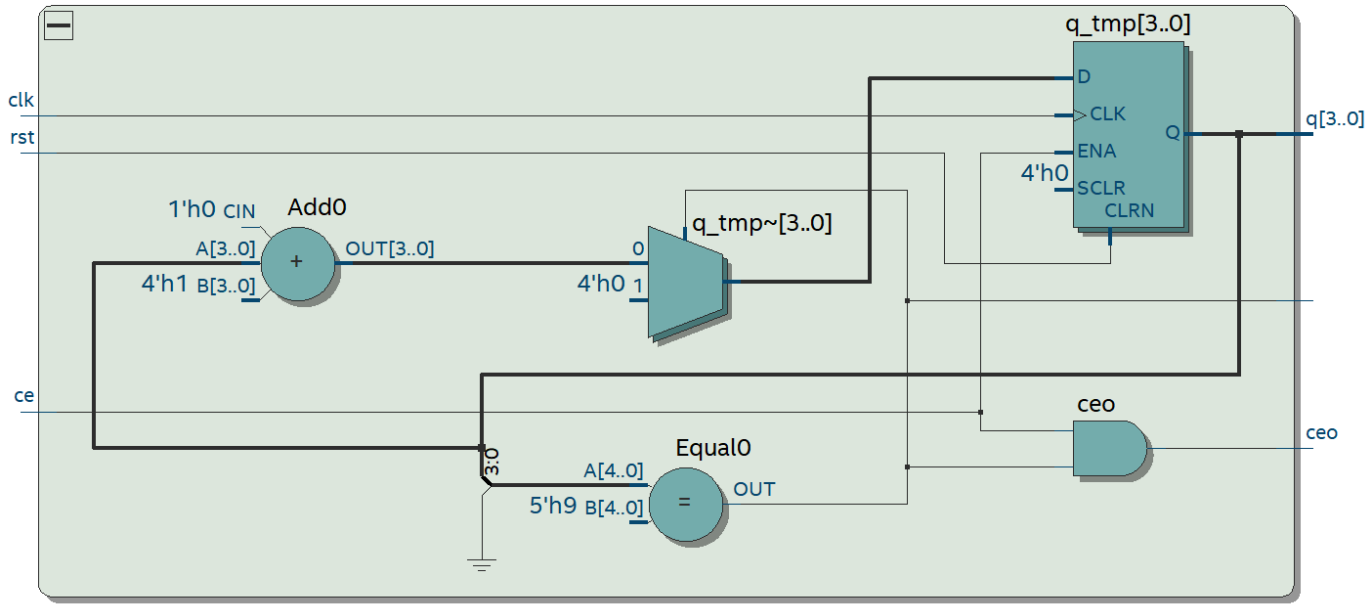


Rsync <= q3 and q0

Synchronous Counters

4-Bit Synchronous Decade Counter

```
11  entity d_cntr4Aceo is
12      Port    (clk : in std_logic;
13               rst : in std_logic;
14               ce : in std_logic;
15               tc : out std_logic;
16               ceo : out std_logic;
17               q : out std_logic_vector(3 downto 0) );
18  end entity d_cntr4Aceo;
19
20  architecture behav of d_cntr4Aceo is
21      signal q_tmp : std_logic_vector(q'range) := x"0";
22      signal tci : std_logic;
23  begin
24  process(clk,rst) begin
25      if rst='1' then
26          q_tmp <= x"0";
27      elsif rising_edge(clk) then
28          if ce='1' then
29              if tci='1' then
30                  q_tmp <= x"0";
31              else
32                  q_tmp <= q_tmp + 1;
33              end if;
34          end if;
35      end if;
36  end process;
37  -- outputs
38  tci <= '1' when (q_tmp=9) else '0';
39  ceo <= (tci and ce);
40  tc <= tci;
41  q <= q_tmp;
42  end architecture behav;
```



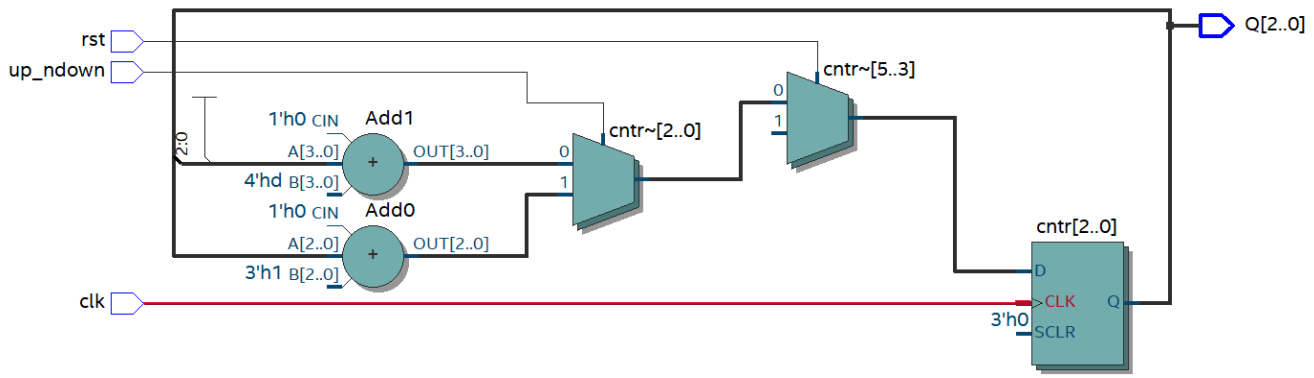
Up/Down Counter

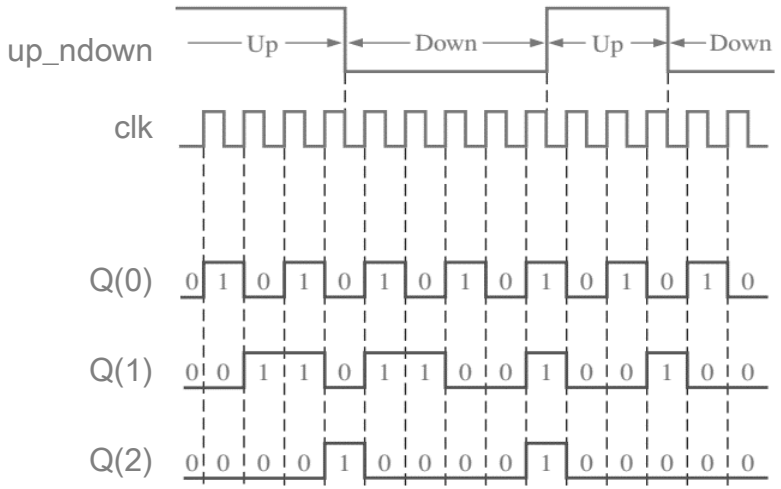
- An up/down counter is one that is capable of progressing in either direction through a certain sequence.
- An up/down counter, sometimes called a bidirectional counter, can have any specified sequence of states.
- In general, most up/down counters can be reversed at any point in their sequence.

```
6  entity cntrNbi is
7    generic(N: positive:=3);
8   port(clk,rst,up_ndown: in std_logic;
9        Q: out std_logic_vector(N-1 downto 0));
10 end;
11 architecture behav of cntrNbi is
12   signal cntr: std_logic_vector(Q'range):=(others=>'0');
13 begin
14 cntr_proc: process(clk) begin
15   if rising_edge(clk) then
16     if rst='1' then cntr <= (others=>'0');
17     elsif up_ndown='1' then cntr <= cntr +1;
18     else cntr <= cntr -1; end if;
19   end if;
20 end process;
21 Q <= cntr;
22 end architecture;
```

Up/Down sequence for a 3-bit binary counter.

Clock Pulse	Up	Q_2	Q_1	Q_0	Down
0		0	0	0	
1		0	0	1	
2		0	1	0	
3		0	1	1	
4		1	0	0	
5		1	0	1	
6		1	1	0	
7		1	1	1	



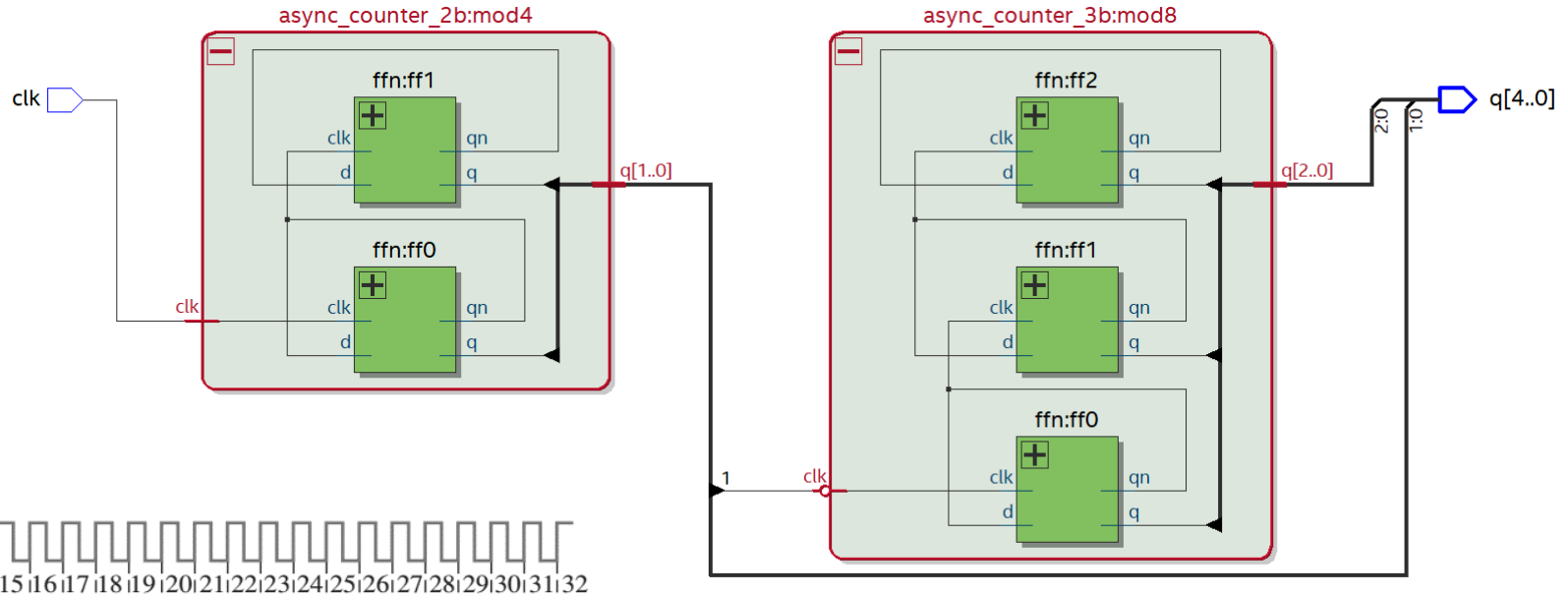


source: Thomas Floyd: Digital Fundamentals

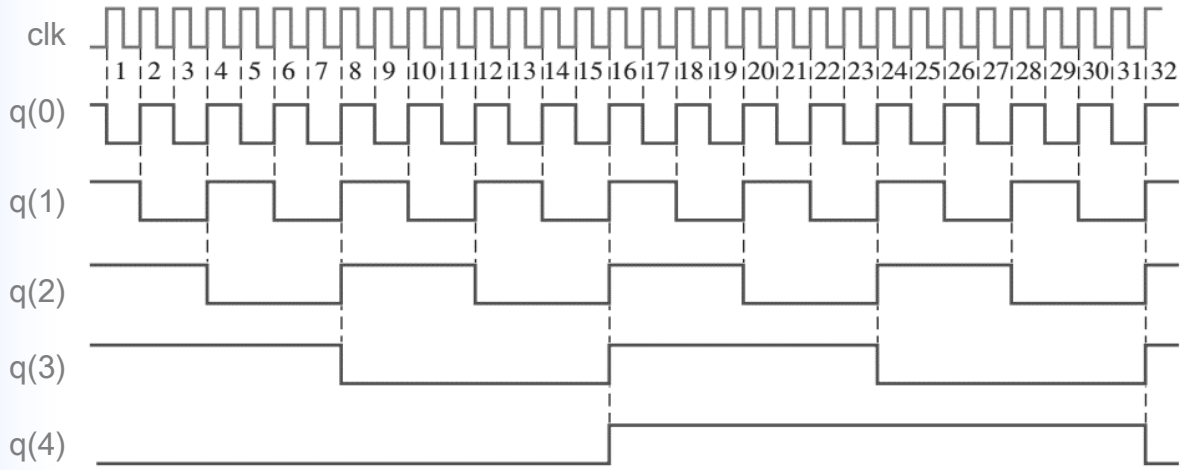
Cascaded Counter

Counters can be connected in cascade to achieve higher-modulus operation.
In essence, cascading means that the last-stage output of one counter drives the input of the next counter.

- Asynchronous Cascading



4 x 8 = 32

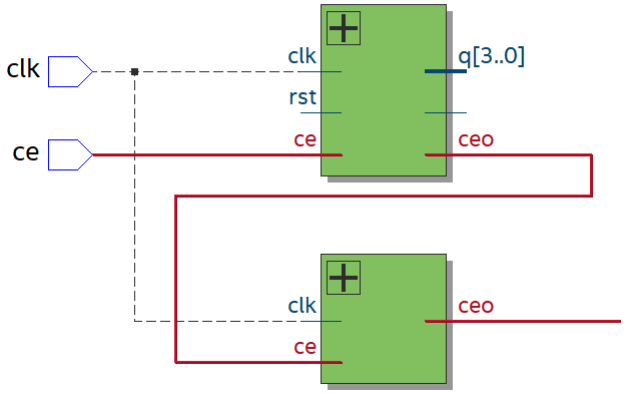


source: Thomas Floyd: Digital Fundamentals

Cascaded Counter

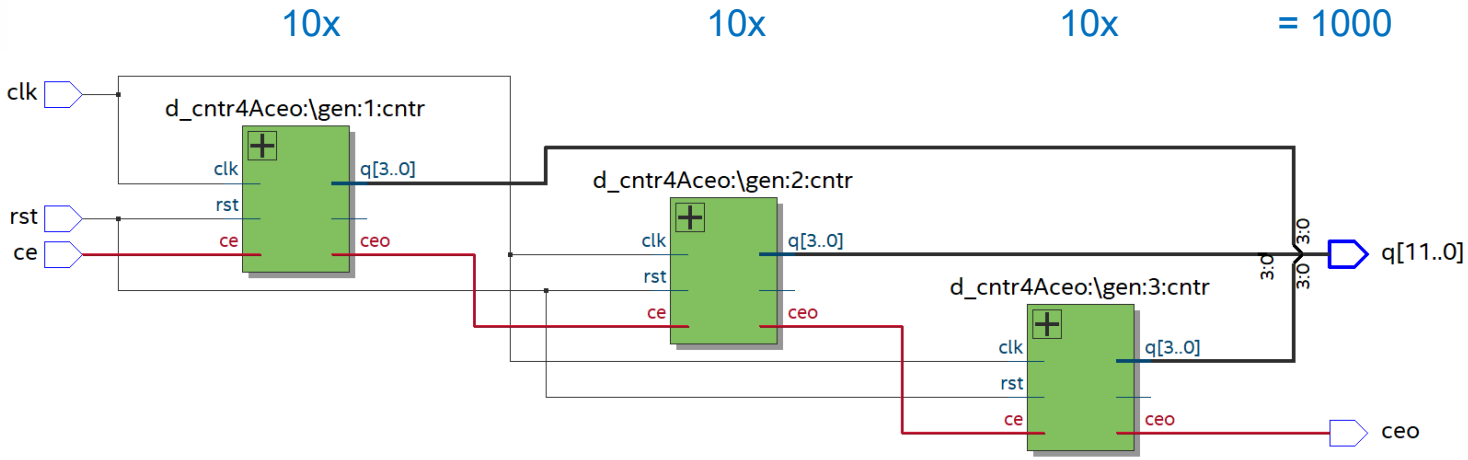
- Synchronous Cascading

  When operating synchronous counters in a cascaded configuration, it is necessary to use the clock enable (CE) and the terminal count (TC) or clock enable output (CEO) functions to achieve higher-modulus operation



- Full-modulus Cascading

  An overall modulus (divide-by-factor) is the product of the individual moduli of all the cascaded counters




```
9  entity cntr_xN is
10     generic(N : natural := 3);
11     port (clk : in std_logic;
12           rst : in std_logic;
13           ce : in std_logic;
14           ceo : out std_logic;
15           q : out std_logic_vector(4*N-1 downto 0)    );
16 end entity cntr_xN;
17 architecture struct of cntr_xN is
18     signal cei : std_logic_vector(N downto 0);
19 begin
20     cei(0) <= ce; ceo <= cei(N);
21     gen: for i in 1 to N generate
22         cntr: entity work.d_cntr4Aceo
23         port map(clk,rst,cei(i-1),open,cei(i),
24                  q((i*4)-1 downto (i-1)*4));
25     end generate;
26 end architecture struct;
```

Cascaded Counter

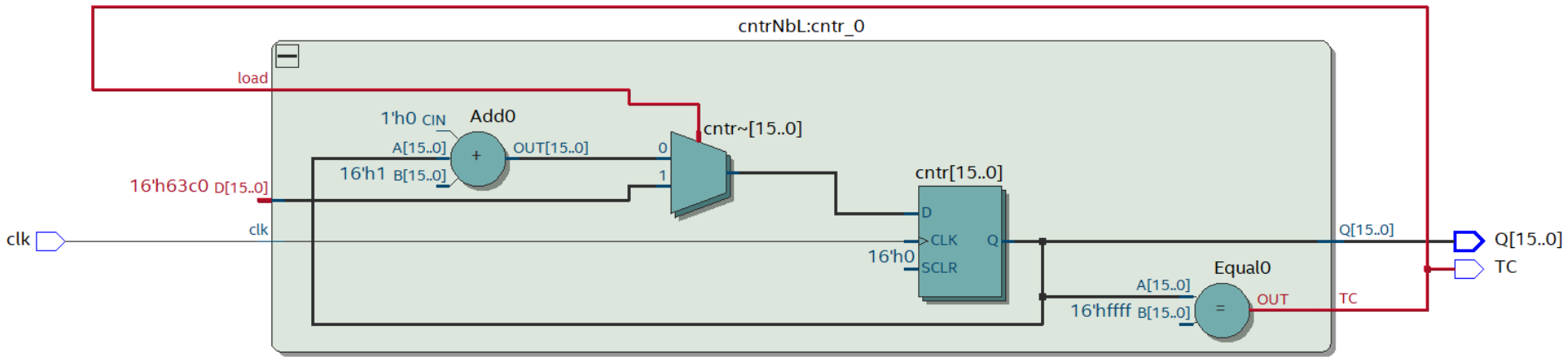
- Truncated Sequences

  Often an application requires an overall modulus that is less than that achieved by full-modulus cascading. A truncated sequence must be implemented with cascaded counters with LOAD.

  - 16-bit Loadable Synchronous Binary Counter

    Let's assume that a certain application requires a divide-by-40,000 counter (modulus 40,000). The difference between 65,536 and 40,000 is 25,536, which is the number of states that must be deleted from the full-modulus sequence. The technique used in the circuit is to preset the cascaded counter to 25,536 (63C0 in hexadecimal) each time it recycles, so that it will count from 25,536 up to 65,535 on each full cycle. Therefore, each full cycle of the counter consists of 40,000 states.

$2^{16} = 65,535$ 65,536 – 40,000 = 25,536 (x"63C0")



source: Thomas Floyd: Digital Fundamentals

Cascaded Counter

- Truncated Sequences

```
6   entity cntrNbL is
7     generic(N: positive:=16);
8     port(clk, load: in std_logic;
9          D: in std_logic_vector(N-1 downto 0);
10         TC: out std_logic;
11         Q: out std_logic_vector(N-1 downto 0));
12  end;
13  architecture behav of cntrNbL is
14    constant EOSQ: std_logic_vector(Q'range):=(others=>'1');
15    signal cntr: std_logic_vector(Q'range):=(others=>'0');
16  begin
17  cntr_proc: process(clk) begin
18    if rising_edge(clk) then
19      if load='1' then
20        cntr <= D;
21      else
22        cntr <= cntr +1;
23      end if;
24    end if;
25  end process;
26  Q <= cntr;
27  TC <= '1' when cntr=EOSQ else '0';
28  end architecture;
```

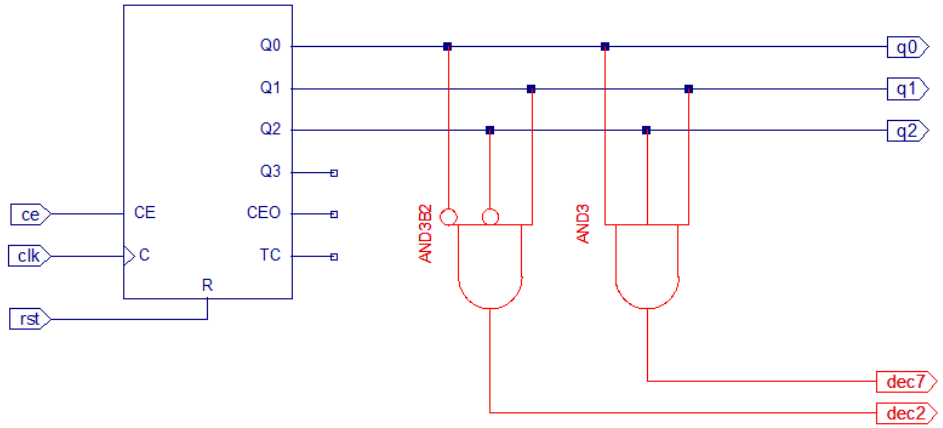
FF/LUT: 16/17

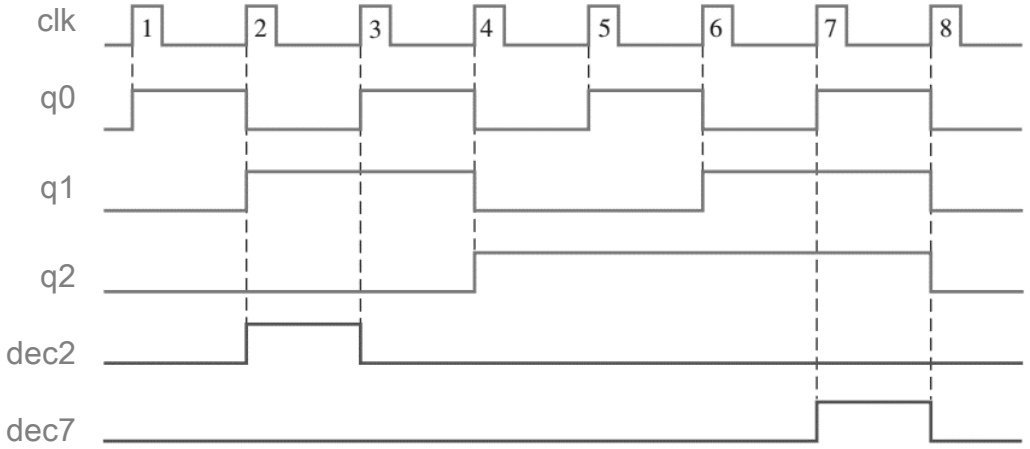
```
37  entity cntrNbDR is
38    generic(N: positive:=16);
39    port(clk, rst: in std_logic;
40         D: in std_logic_vector(N-1 downto 0);
41         TC: out std_logic;
42         Q: out std_logic_vector(N-1 downto 0));
43  end;
44  architecture behav of cntrNbDR is
45    signal cntr: std_logic_vector(Q'range):=(others=>'0');
46  begin
47  cntr_proc: process(clk) begin
48    if rising_edge(clk) then
49      if rst='1' then
50        cntr <= (others=>'0');
51      elsif cntr=D then
52        cntr <= (others=>'0');
53      else
54        cntr <= cntr +1;
55      end if;
56    end if;
57  end process;
58  Q <= cntr;
59  TC <= '1' when cntr=D else '0';
60  end architecture;
```

FF/LUT: 16/25

Counter Decoding

- In many applications, it is necessary that some or all of the counter states be decoded.
- The decoding of a counter involves using decoders or logic gates to determine when the counter is in a certain binary state in its sequence.
- For instance, the terminal count (TC) function previously discussed is a single decoded state (the last state) in the counter sequence.

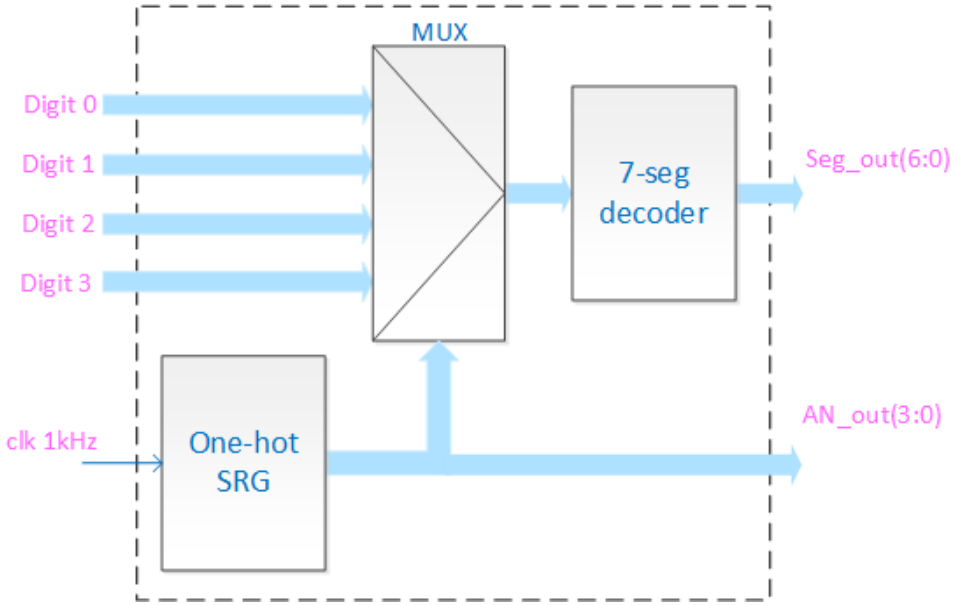


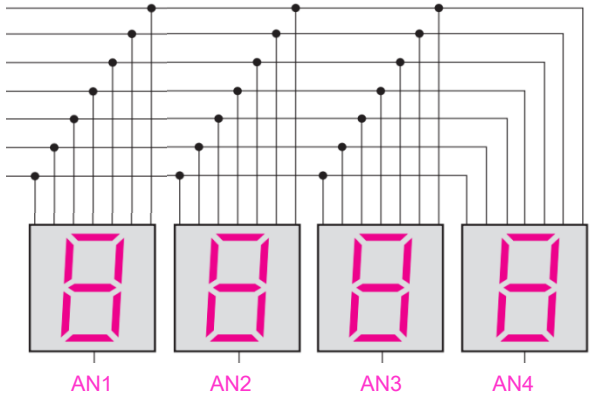


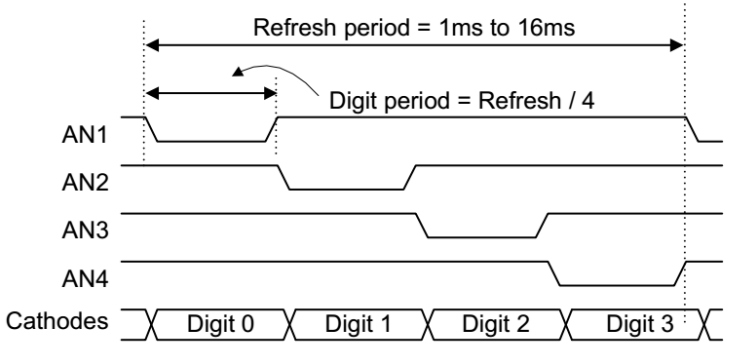
source: Thomas Floyd: Digital Fundamentals

Apps: 7Segment Display Multiplexer

- A simplified method of multiplexing BCD numbers to a multidigit 7-segment display
- 4-digit numbers are displayed on the 7-segment readout by the use of a single BCD-to-7-segment decoder





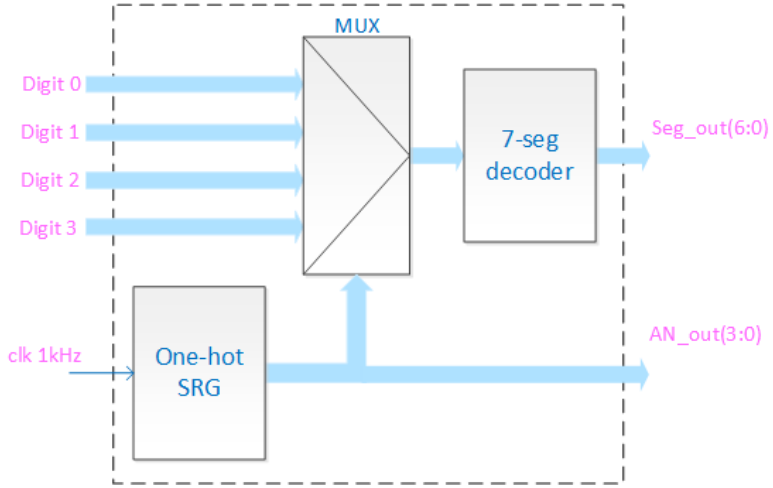


Apps: 7Segment Display Multiplexer

```vhdl
entity muxNoh_4x1 is
  Generic(N : positive:=32);
  Port(d0,d1,d2,d3: in std_logic_vector(N-1 downto 0);
       sel_oh: std_logic_vector(3 downto 0);
       y: out std_logic_vector(N-1 downto 0) );
end entity muxNoh_4x1;
```

```vhdl
entity SRN_RING is
  generic(N: positive:=10);
  port (clk: in std_logic;
        Q: out std_logic_vector(N-1 downto 0));
end entity;
```

```vhdl
entity dec7seg is
    Port ( bcd : in std_logic_vector(3 downto 0);
           seg : out std_logic_vector(6 downto 0));
end entity dec7seg;

architecture ttable of dec7seg is
begin
 with bcd select
  seg<= "1111001" when x"1",
        "0100100" when x"2",
        "0110000" when x"3",
        "0011001" when x"4",
        "0010010" when x"5",
        "0000010" when x"6",
        "1111000" when x"7",
        "0000000" when x"8",
        "0010000" when x"9",
        "1000000" when x"0",
        "0111111" when others;
end architecture ttable;
```




```vhdl
entity s7x4_drv is
 port (clk_1kHz: in std_logic;
       D0,D1,D2,D3: std_logic_vector(3 downto 0);
       Seg_out: out std_logic_vector(6 downto 0);
       AN_out: out std_logic_vector(3 downto 0)
       );
end entity;
--
architecture struct of s7x4_drv is
signal q_mux, q_srg: std_logic_vector(3 downto 0);
begin
mux_inst: entity work.muxNoh_4x1
    generic map(N=>4)
    port map(d0=>D0, d1=>D1, d2=>D2, d3=>D3,
             sel_oh=>q_srg, y=>q_mux);
onehot_inst: entity work.SRN_RING
    generic map(N=>4)
    port map(clk=>clk_1kHz, Q=>q_srg);
dec7s_inst: entity work.dec7seg
    port map(bcd=>q_mux, seg=>Seg_out);
AN_out <= q_srg;
end architecture;
```

Apps: 7Segment Display Multiplexer

```
4   entity s7x4_drv is
5    port (clk_1kHz: in std_logic;
6          D0,D1,D2,D3: std_logic_vector(3 downto 0);
7          Seg_out: out std_logic_vector(6 downto 0);
8          AN_out: out std_logic_vector(3 downto 0)
9          );
10  end entity;
11
12  architecture behav of s7x4_drv is
13   signal one_hot: std_logic_vector(3 downto 0):=x"E";
14   signal bcd: std_logic_vector(3 downto 0);
15
16  begin
17  onehot_reg: process(clk_1kHz) begin
18      if rising_edge(clk_1kHz) then
19          one_hot <= one_hot(2 downto 0) & one_hot(3);
20      end if;
21    end process;
22
23  data_mux: with one_hot select
24    bcd <= d0 when "1110",
25           d1 when "1101",
26           d2 when "1011",
27           d3 when others;
28
29  decoder: with bcd select
30   Seg_out<= "1111001" when x"1",
31             "0100100" when x"2",
32             "0110000" when x"3",
33             "0011001" when x"4",
34             "0010010" when x"5",
35             "0000010" when x"6",
36             "1111000" when x"7",
37             "0000000" when x"8",
38             "0010000" when x"9",
39             "1000000" when x"0",
40             "0111111" when others;
41  AN_out <= one_hot;
42  end architecture;
```