Registers and Counters

VOLKAN KURSUN

Some material from McGraw Hill

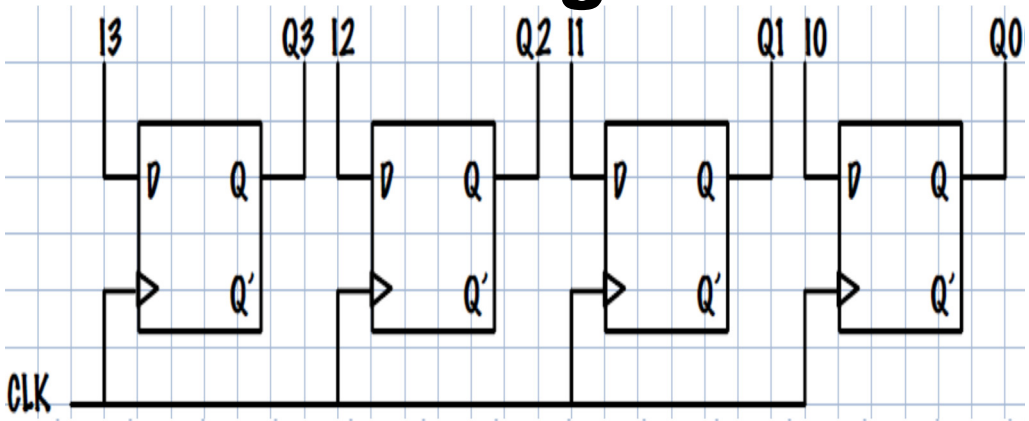
Outline

- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Registers

- Flip-flop: stores one bit of information
- N-bit register: a set of n flip-flops to store n bits of information (an n-bit structure consisting of n flip-flops)
- Number of input combinations: 2^n
- Number of output combinations: 2^n
- Outputs represent the state
- Number of states: 2^n

4-Bit Register



input: I = (I3, I2, I1, I0)
output: Q = (Q3,Q2,Q1,Q0)

How many states? 16
Number of input combinations? 16
Number of output combinations? 16

4-Bit Register VHDL Code

```
entity reg4 is
    port(I: in std_logic_vector(3 downto 0);
         CLK: in std_logic;
          Q: out std_logic_vector(3 downto 0);
end reg4;

architecture reg4arch of reg4 is
begin
    process(CLK) begin
        if rising_edge(CLK) then
            Q<=I;
        end if;
    end process;
end reg4arch;
```

The code does not specify what Q should be assigned when the condition for the if statement is not satisfied: implies that the Q should maintain its value. **IMPLIES MEMORY**

8-Bit Register VHDL Code

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "00000000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Asynchronous reset

N-Bit Register VHDL Code

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

A parameter N is declared using the keyword GENERIC

Clear every bit of Q to 0: OTHERS permit generic code that can be used for any value of N

N-Bit Register VHDL Code

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

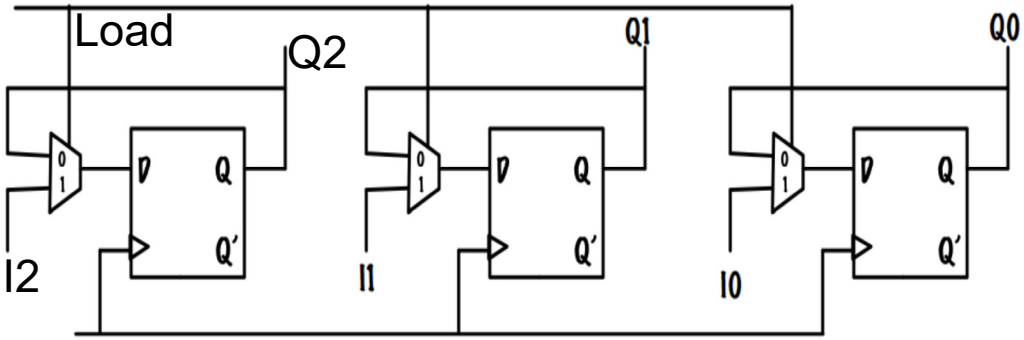
Warning: error if no actual is specified for a given formal generic parameter and no default expression is present in the corresponding interface element.

```
Use GENERIC MAP () if the
default value of N needs to be
changed during instantiation:
reg1: regn GENERIC MAP (64);
Or use named associations
reg1: regn GENERIC MAP (N =>
64);
```

Both named and positional associations are allowed in a generic association list.

3-Bit Register with Load

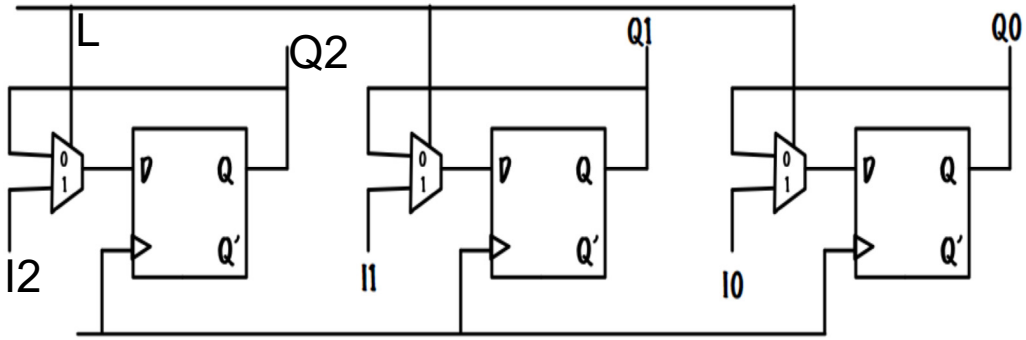
- With the registers in the previous slides, unless there is a reset signal, outputs are updated every clock cycle with the positive edges of the clock signal
- **An additional load control signal** would permit controlling when (at which positive edges of the clock signal) the register outputs would be updated
- **Load = 0: maintain state**, **Load = 1: load new data**



3-Bit Register with Load VHDL

```
process(CLK) begin
    if rising_edge(CLK) then
        if L='1' then
            Q<=I;
        end if;
    end if;
end process;
```

The code does not specify what Q should be assigned when the condition for the if statement is not satisfied: implies that the Q should maintain its value. **IMPLIES MEMORY**



Outline

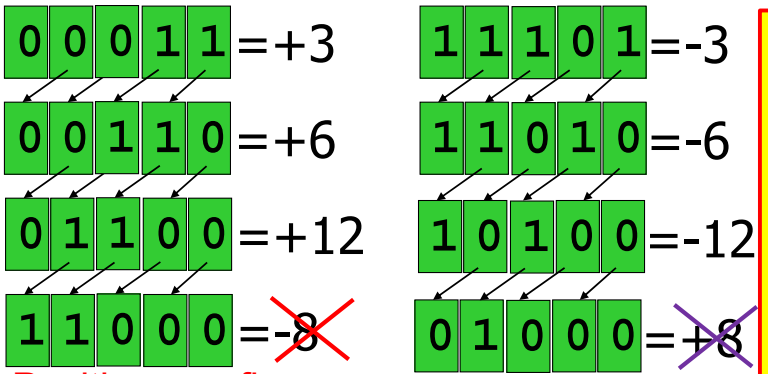
- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Shift Register

- A register that can shift its contents
- **Shift left**: shift all bits by one bit position to the left. 0 is inserted as new least significant bit and the most significant bit is lost
- **Shift** an n-bit signed binary number to the **left**: **multiply by 2**
- **Shift right**: shift all bits by one bit position to the right. A serial input is inserted as the new most significant bit and the least significant bit is lost
- **Shift** an n-bit **unsigned binary number to the right** with **0 inserted from left**: **divide by 2**

Shift Left for Multiplication

Shift left by i bits (x << i) is equivalent to multiplying a signed number by 2^i: works **ONLY for signed numbers**

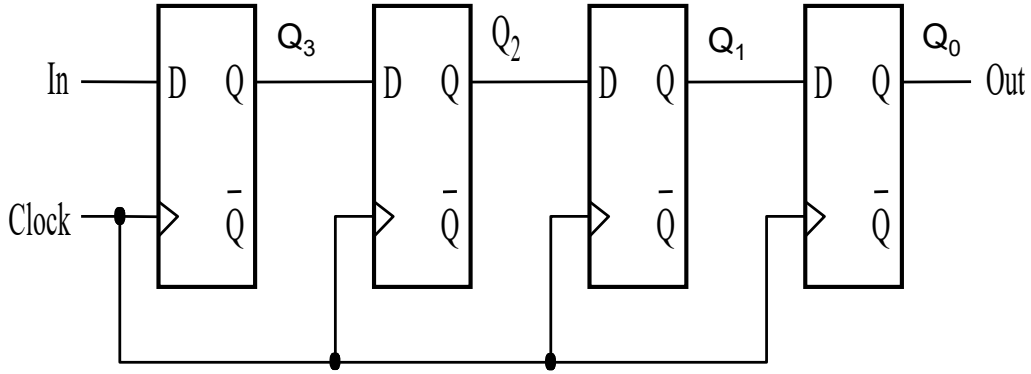


Note: **any unsigned number** that starts with a 1 is **guaranteed to overflow when shifted left**: **result > $+2^n-1$**

- **Positive overflow** if product > $+2^{n-1} - 1$
- **Negative overflow** if product < -2^{n-1}

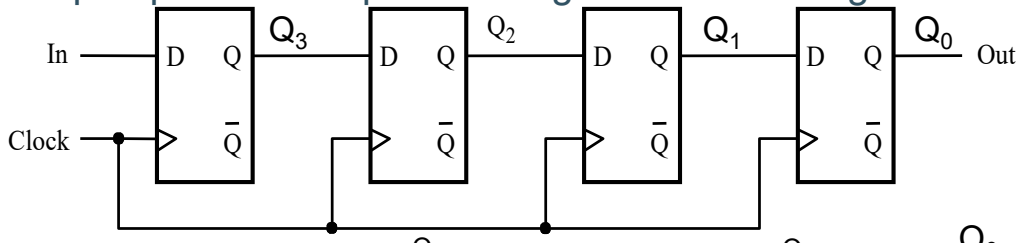
4-Bit Shift-Right Register

- **Flip-flops are needed** to design a shift register: **latches are not suitable** since the input could propagate through multiple latches when the clock is high or low with positive or negative latches, respectively



4-Bit Shift-Right Register

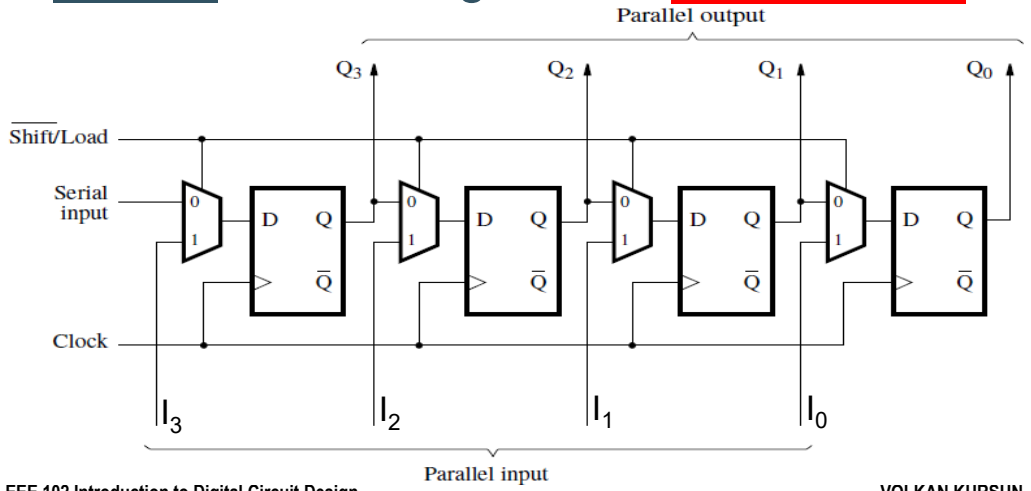
- **Data bits are loaded into the shift register in serial fashion with the Input from left**
- The content of each flip-flop is transferred to the next flip-flop with each positive edge of the clock signal



	In	Q_3	Q_2	Q_1	Q_0
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

Parallel-Access Shift Register

- **Shift/Load = 0**, operates as **shift register**
- **Shift/Load = 1**, **parallel input data** are **loaded** into the register for **initialization**



Notes About VHDL Signal Modes

- The signal mode determines how the entity reads and writes to each signal. There are **four different signal modes**:

1) IN: Data flows into the entity, and the **entity can NOT write** to these signals. The IN mode is used for clock inputs, control inputs, and other **unidirectional data inputs to entity**.

2) OUT: Data flows out of the entity (**unidirectional data output**), and the **entity can NOT read** these signals. The OUT mode is only used when the **output signal is not read by the entity**.

3) BUFFER: Data flows out of the entity (**unidirectional data output**), but the **entity can read** the signal (**allows internal feedback**). However, the signal **cannot be driven from outside the entity**, so it **can NOT be used for data input to the entity**.

4) INOUT: Data can flow both in and out of the entity, and the signal can be driven from outside the entity (for instance, a bidirectional data bus). **Bidirectional data flow**

4-Bit Parallel-Access Shift Register VHDL

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( R      : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
5              Clock  : IN     STD_LOGIC ;
6              L, w   : IN     STD_LOGIC ;
7              Q      : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
8   END shift4 ;

9   ARCHITECTURE Behavior OF shift4 IS
10  BEGIN
11      PROCESS
12      BEGIN
13          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
14          IF L = '1' THEN
15              Q <= R ;
16          ELSE
17              Q(0) <= Q(1) ;
18              Q(1) <= Q(2);
19              Q(2) <= Q(3) ;
20              Q(3) <= w ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

BUFFER signal mode: Data flows out of the entity and the **entity can read** the signal (allowing for internal feedback)

Since Q is declared as BUFFER: the **entity can directly read** and write to the Q port

No sensitivity list: the VHDL code inside the process block will run continuously (the program loops back to the start of the block after executing the last line).

Concurrent assignments: the order of assignments do not matter. All four outputs are updated with the same positive edge of the clock

w: serial data input from left

N-Bit Parallel-Access Shift Register VHDL

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shiftn IS
4       GENERIC ( N : INTEGER := 8 ) ;
5       PORT ( R      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
6              Clock  : IN     STD_LOGIC ;
7              L, w   : IN     STD_LOGIC ;
8              Q      : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
9   END shiftn ;

10  ARCHITECTURE Behavior OF shiftn IS
11  BEGIN
12      PROCESS
13      BEGIN
14          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
15          IF L = '1' THEN
16              Q <= R ;
17          ELSE
                Genbits: FOR i IN 0 TO N-2 LOOP
                    Q(i) <= Q(i+1) ;
                END LOOP ;
20              Q(N-1) <= w ;
21          END IF ;
22      END PROCESS ;
23  END Behavior ;
```

A parameter N is declared using the keyword GENERIC

Shift right

Synchronous load

MSB assigned w

FOR LOOP is used to generate a set of concurrent assignments: Q(0) <= Q(1), Q(1) <= Q(2), Q(2) <= Q(3), Q(3) <= Q(4)…

Notes on For Loops in VHDL

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shiftn IS
4       GENERIC ( N : INTEGER := 8 ) ;
5       PORT ( R      : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
6              Clock  : IN     STD_LOGIC ;
7              L, w   : IN     STD_LOGIC ;
8              Q      : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
9   END shiftn ;

10  ARCHITECTURE Behavior OF shiftn IS
11  BEGIN
12      PROCESS
13      BEGIN
14          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
15          IF L = '1' THEN
16              Q <= R ;
17          ELSE
18              Genbits: FOR i IN 0 TO N-2 LOOP
19                  Q(i) <= Q(i+1) ;
20              END LOOP ;
21              Q(N-1) <= w ;
22          END IF ;
23      END PROCESS ;
24  END Behavior ;
```

Loop labels demarcate the beginning and end points of loops

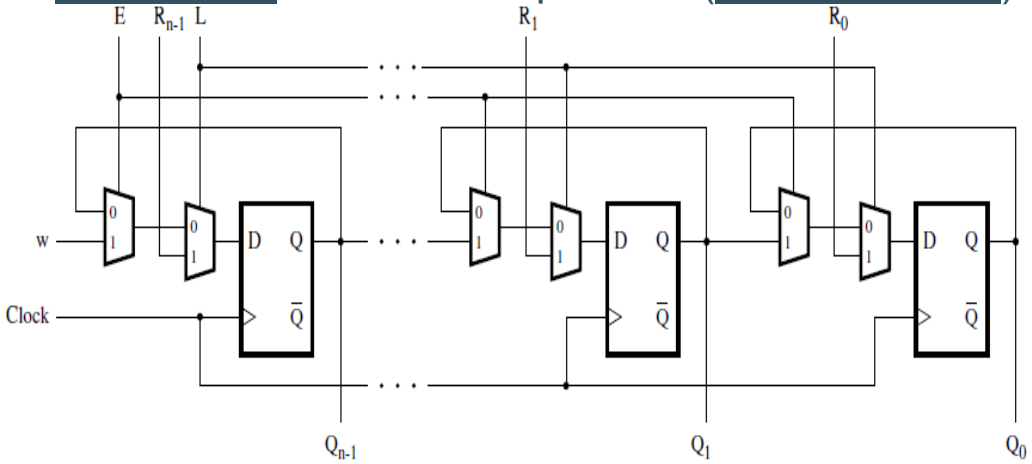
Loop labels may be used to enhance readability, especially when loops are nested or the code block executed within the loop is long.

The loop label is **optional**

END LOOP Genbits;

Shift Register with Parallel-Load and Enable

- L = 1: new data loaded into the shift-register
- L = 0, E = 1: shift right with the positive edges of the clock signal
- L = 0, E = 0: inhibit shift operation (maintain state)



Shift Register with Parallel-Load and Enable

Shift right

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

-- left-to-right shift register with parallel load and enable
ENTITY shiftrne IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT( R          : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          L, E, w    : IN     STD_LOGIC ;
          Clock      : IN     STD_LOGIC ;
          Q          : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftrne ;

ARCHITECTURE Behavior OF shiftrne IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        ELSIF E = '1' THEN
            Q(N-1) <= w ;
            Genbits: FOR i IN N-2 DOWNTO 0 LOOP
                Q(i) <= Q(i+1) ;
            END LOOP ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

BUFFER signal mode: Data flows out of the entity and the **entity can read** the signal (allowing for internal feedback)

IF L = 0, E = 1, FOR LOOP: Q(N-2) <= Q(N-1), Q(N-3) <= Q(N-2),…Q(1) <= Q(2), Q(0) <= Q(1)

Since Q is declared as BUFFER: the **entity can directly read** and write to the Q port

Outline

- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Counters

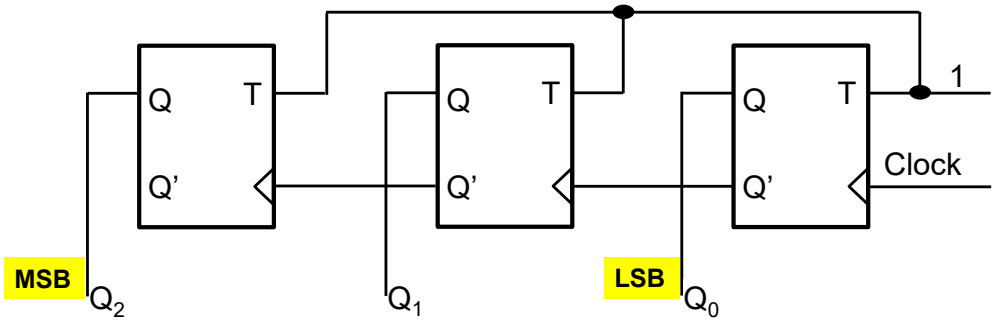
- Can be considered to be special purpose adders or subtractors which are used for the purpose of counting up or down: increment or decrement count by 1
- Used for counting the occurrences of certain events, generating timing intervals for control of various tasks, and keeping track of the elapsed time between specific events in digital systems
- T and D flip-flops are commonly used to implement counters

Asynchronous Counters

- Asynchronous clocking (cascaded clocking schemes): flip-flops at different bit positions are clocked at different times
- Simple design but not very fast: counter delays tend to be long in cascaded clocking schemes
- Toggle feature of T flip-flops is naturally suited for the implementation of counters

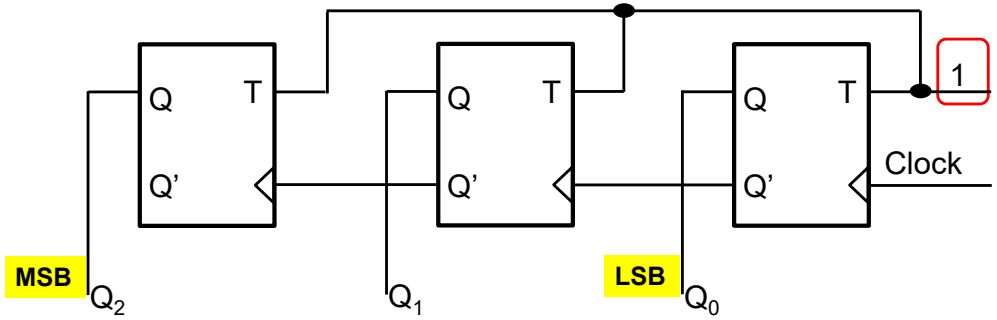
Asynchronous Up Counter

- Modulo operator (%): returns the remainder after a division operation
- 13 % 8 = 5, 7 % 8 = 7, 8 % 8 = 0
- Example: a 3-bit up counter that can count from 0 to 7 (modulo-8 up counter)



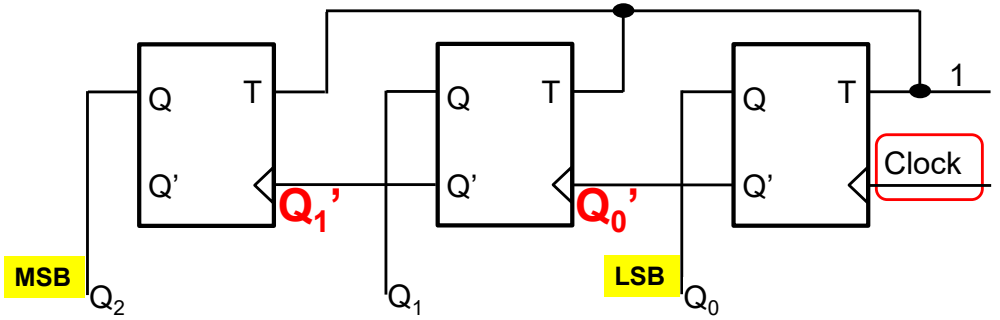
Asynchronous Up Counter

- Example: a 3-bit up counter that can count from 0 to 7 (modulo-8 counter)
- T inputs of 3 flip-flops are connected to 1: the state of a flip-flop will be reversed (toggled) at each positive edge of its clock



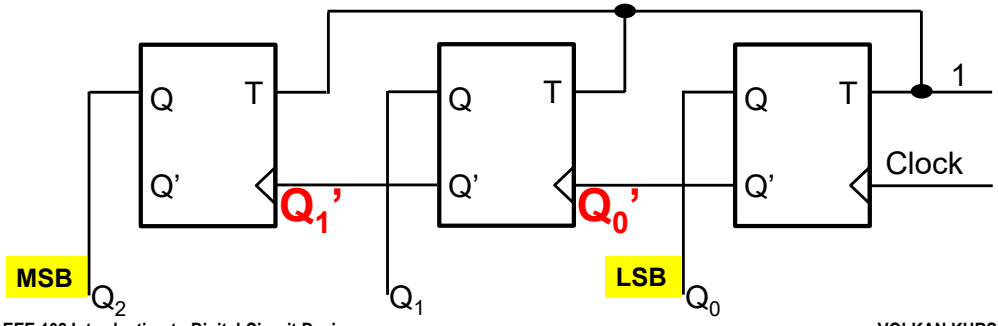
Asynchronous Up Counter

- ONLY the first flip-flop is connected to the actual Clock signal: Q_0 toggles once every clock cycle
- The clock inputs of the remaining T flip-flops are connected in cascade
- **Second flip-flop is clocked by Q_0'**
- **Third flip-flop (MSB) is clocked by Q_1'**

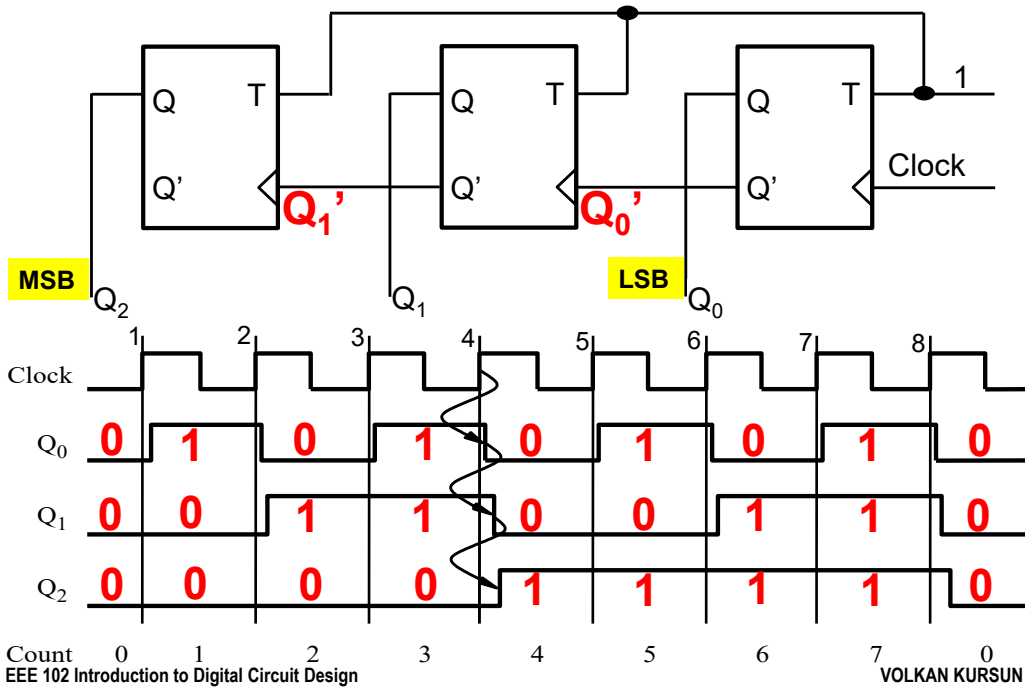


Asynchronous Up Counter

- Q_0 toggles once every clock cycle: the circuit **counts the number of external clock pulses that occur on the primary input (applied to the Q_0 flip-flop)**
- Second flip-flop is clocked by Q_0': value of Q_1 changes (toggles) **after the negative edge of the Q_0 signal**
- Third flip-flop is clocked by Q_1': value of Q_2 changes (toggles) **after the negative edge of the Q_1 signal**

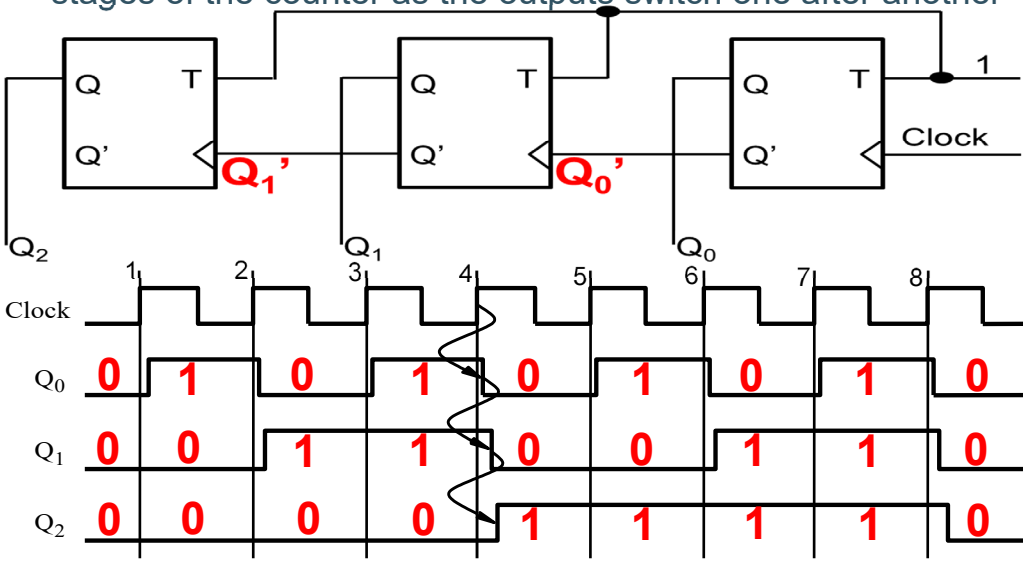


Asynchronous Up Counter



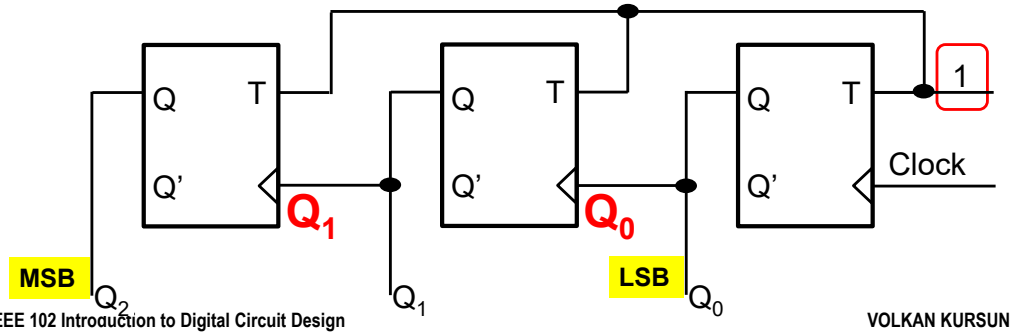
Asynchronous Up Counter

- AKA **ripple counter**: triggering signals ripple through the stages of the counter as the outputs switch one after another

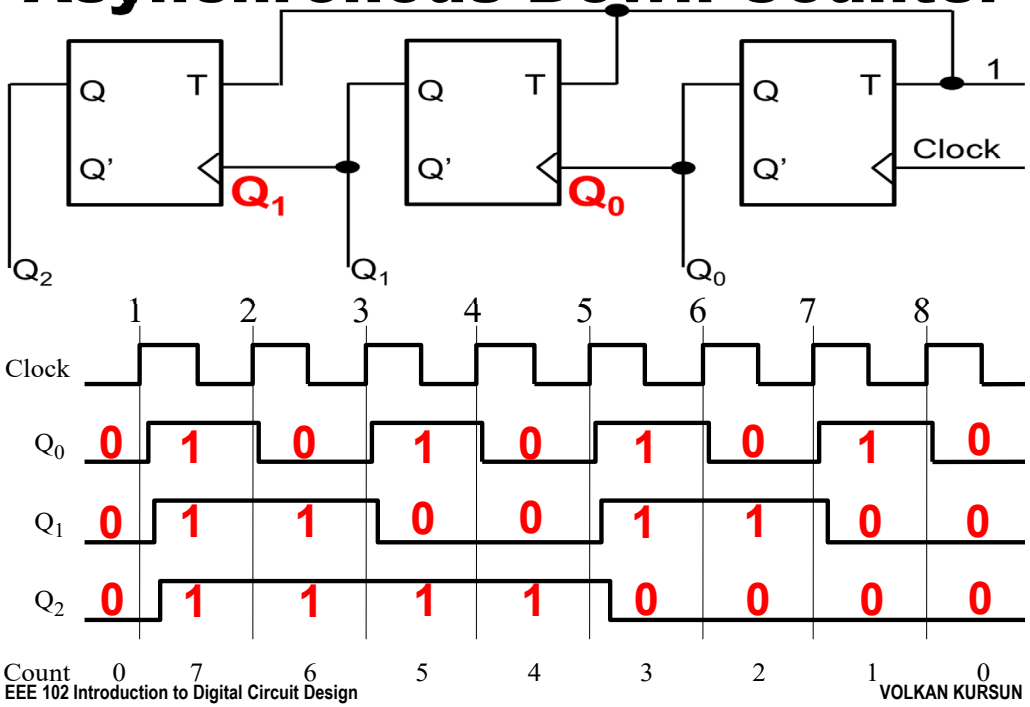


Asynchronous Down Counter

- Example: a 3-bit down counter that can **count from 7 down to 0** (asynchronous modulo-8 down counter)
- **ONLY the first flip-flop (LSB position) is connected to the actual Clock** signal: Q_0 toggles once every clock cycle
- The clock inputs of the remaining T flip-flops are connected in cascade
- **Second flip-flop is clocked by Q_0: Q_1 toggles whenever Q_0 switches from 0 to 1**
- **Third flip-flop (MSB) is clocked by Q_1: Q_2 toggles whenever Q_1 switches from 0 to 1**



Asynchronous Down Counter



Outline

- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

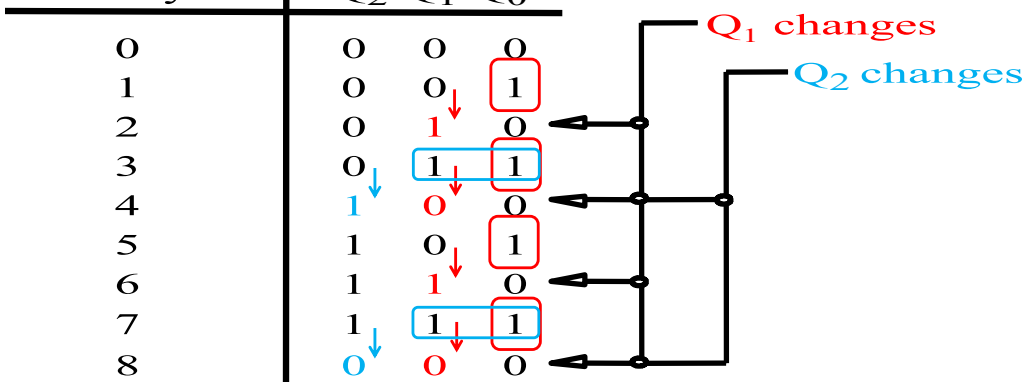
Synchronous Counters

- **Asynchronous** clocking (**cascaded clocking** schemes): simple design but **not very fast**
- **Accumulating delays** across multiple cascaded clocking stages become a concern for speed if the number of bits is high (**low clock frequency** problem)
- **Synchronous** counters: **all flip-flops are clocked at the same time** and they operate faster (higher frequency)

Synchronous Counter with T

- Example: synchronous 3-bit up counter

Clock cycle	Q_2	Q_1	Q_0
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
8	0	0	0



- Q_0 changes on each clock cycle
- Q_1 changes only when $Q_0 = 1$
- Q_2 changes only when $Q_1 = Q_0 = 1$

Synchronous Counter with T

- Example: synchronous 3-bit up counter

Clock cycle	Q_2 Q_1 Q_0
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
8	0 0 0

Q_1 changes

Q_2 changes

- In an n-bit synchronous up counter, a flip-flop changes state only when the flip-flops in previous bit positions are all in set state Q = 1

Synchronous Counter with T

Clock cycle	Q_2 Q_1 Q_0
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
8	0 0 0

Q_1 changes

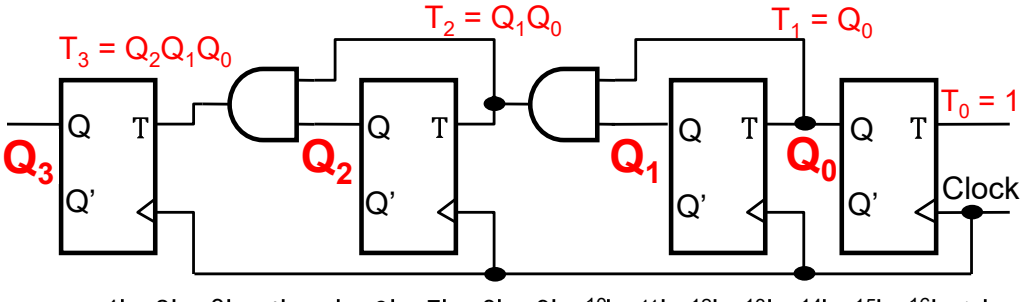
Q_2 changes

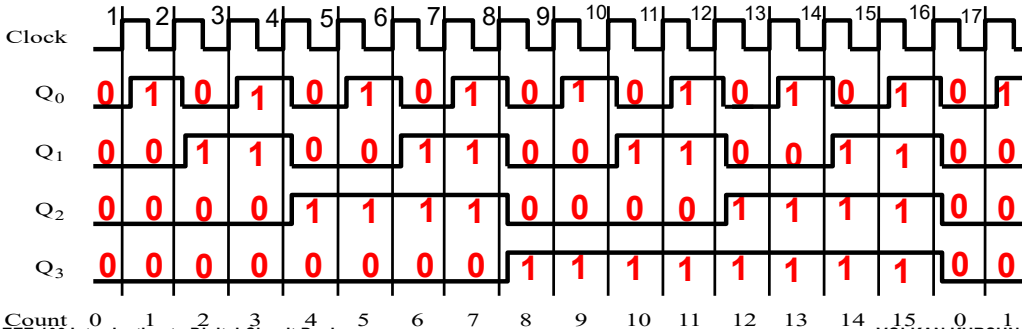
- T inputs of the flip-flops must satisfy:

$T_0 = 1$, $T_1 = Q_0$, $T_2 = Q_1Q_0$, $T_3 = Q_2Q_1Q_0$, …, $T_n = Q_{n-1}Q_{n-2}\ldots Q_1Q_0$

Synchronous Counter with T

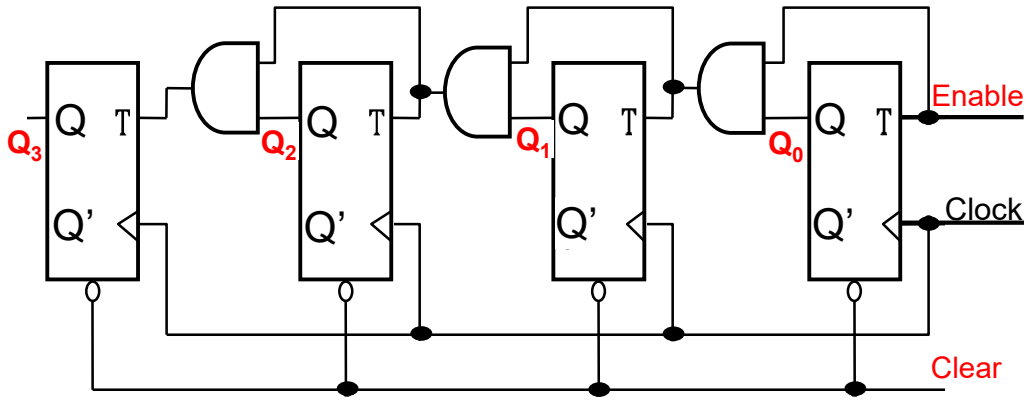
Example: 4-bit synchronous modulo-16 up counter





Enable and Clear Capability

- **Enable: control when to count**
- **Enable = 0**: inhibit counting (**maintain state**)
- Clear: start with a count of 0 (initialize)
- Clear = 0: clear all outputs to 0 (initialize)

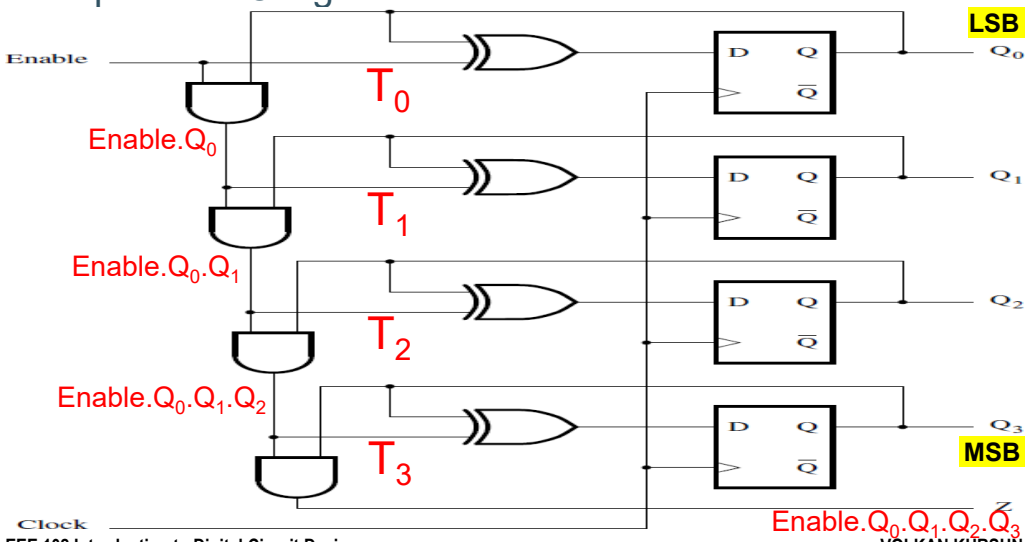


Synchronous Counter with D

- A T flip-flop can be implemented with a D flip-flop and an XOR gate by applying T XOR Q to the D input of the flip-flop: $D = T \oplus Q$
- We can redraw the 4-bit synchronous up counter in previous slide using D flip-flops and XOR gates

Synchronous Counter with D

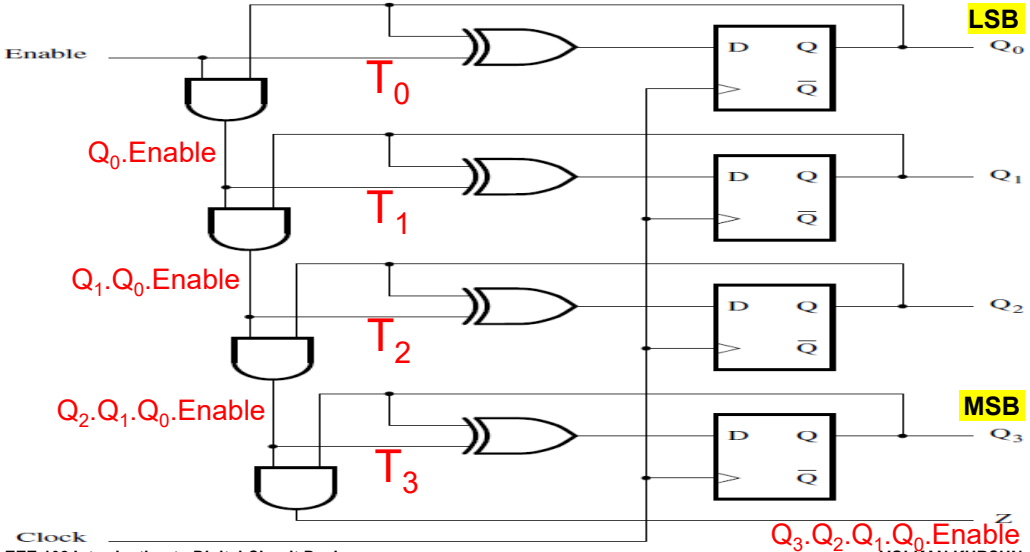
- Apply T XOR Q to the D input of the flip-flop: $D = T \oplus Q$
- Redraw the 4-bit synchronous up counter using D flip-flops and XOR gates



Synchronous Counter with D

$D_0 = Q_0 \oplus Enable$ $D_1 = Q_1 \oplus (Q_0.Enable)$

$D_2 = Q_2 \oplus (Q_1.Q_0.Enable)$ $D_3 = Q_3 \oplus (Q_2.Q_1.Q_0.Enable)$

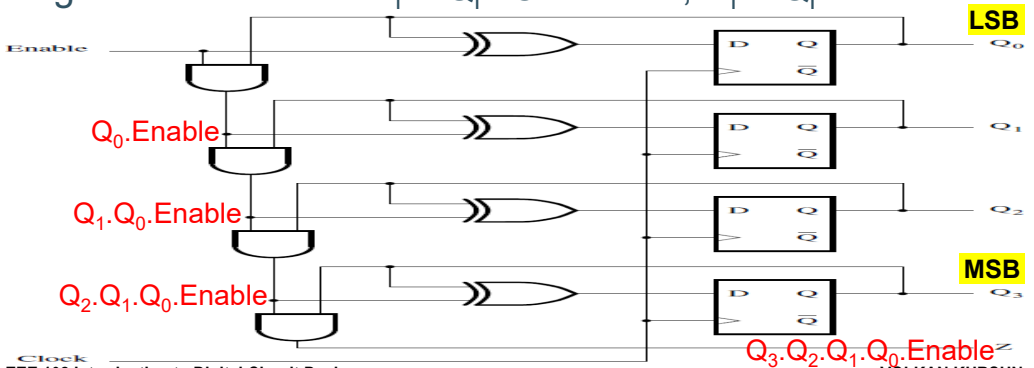


Synchronous Counter with D

$D_0 = Q_0 \oplus Enable$ $D_1 = Q_1 \oplus (Q_0.Enable)$

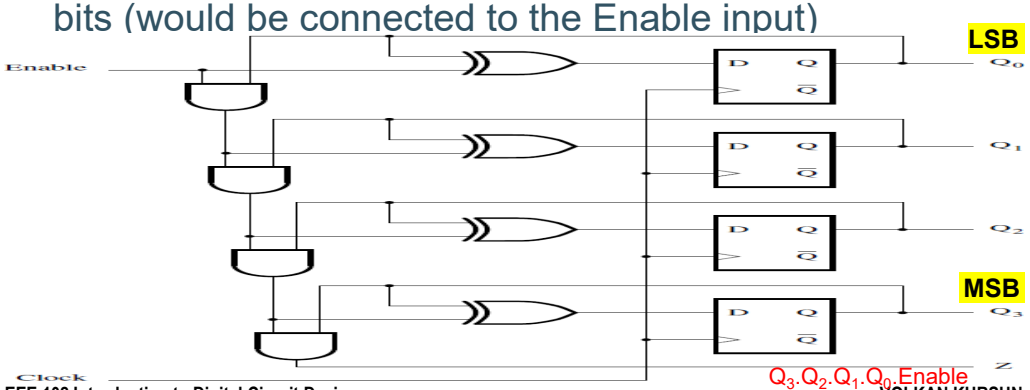
$D_2 = Q_2 \oplus (Q_1.Q_0.Enable)$ $D_3 = Q_3 \oplus (Q_2.Q_1.Q_0.Enable)$

- The state of flip-flop in stage i changes only if all preceding flip-flops are in state Q = 1: if the output of AND gate that feeds stage i is 1, the output of the XOR gate connected to $D_i = Q_i'$. Otherwise, $D_i = Q_i$



Synchronous Counter with D

- The additional **Z flag** at the final output **indicates** if the counter has reached the **maximum value** (0b1111). The counter goes to 0b0000 state (all bits cleared) with the next positive edge of the clock signal after Z is asserted
- The Z flag also allows **modular** design by concatenating two counters to implement a larger counter with more bits (would be connected to the Enable input)



$Q_3.Q_2.Q_1.Q_0.Enable$

4-Bit Up Counter VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clock, Resetn, E   : IN   STD_LOGIC ;
           Q                  : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            ELSE
                Count <= Count ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

OUT: Data flows out of the entity, and the **entity cannot read** these signals. The OUT mode is only used when the **signal is not read by the entity**.

To be able to read and write the count, an internal variable named Count is declared

Asynchronous Reset

Read and write the internal signal named Count

Optional assignment: would work without it too due to IMPLIED MEMORY

Assign the value of the internal signal named Count to the OUTPUT Q

4-Bit Up Counter VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clock, Resetn, E   : IN   STD_LOGIC ;
           Q                  : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            ELSE
                Count <= Count ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Can you write Q instead of Count on the left of the assignment statement to eliminate the final Q <= Count assignment?

Q <= Count + 1, would not update the Count. Therefore, Count would be stuck at 0 after reset and Q would be stuck at 1! You can NOT write Q instead of Count on the left of the assignment statement.

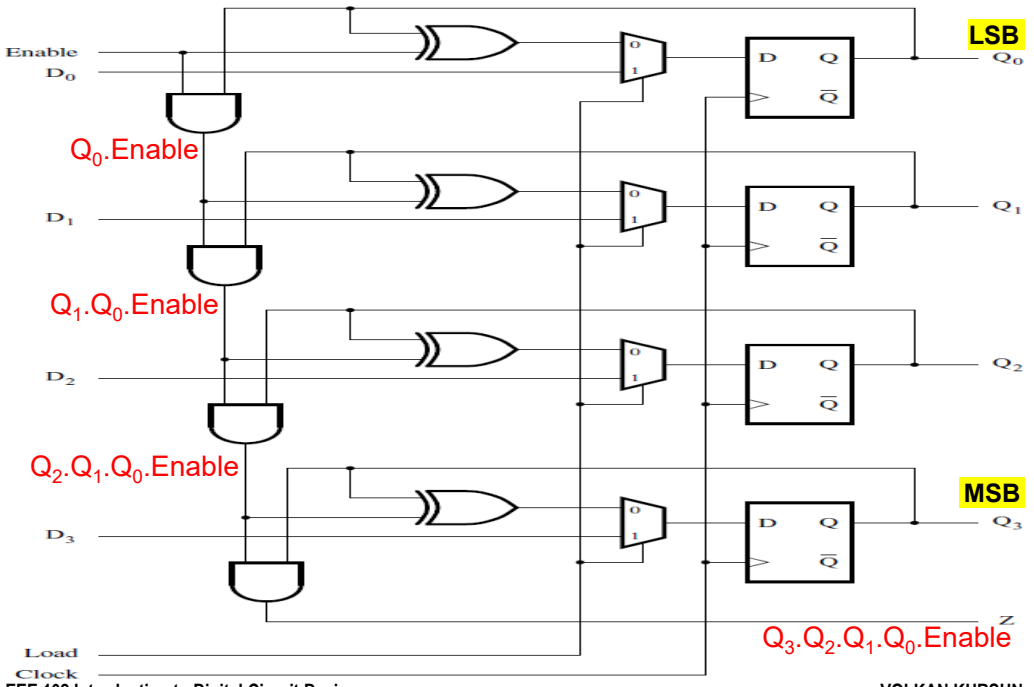
Outline

- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Counter with Parallel Load

- Sometimes it is desirable to **start counting from any arbitrary number** (not necessarily zero): a **counter with parallel load** capability is needed **to initialize** the counter
- Use 2-to-1 multiplexers for each input to choose either the data bit to be directly loaded or the bit needed to continue counting
- Enable = 1 and Load = 0: count
- Load = 1 (Enable = x): load new data to the counter (initialize the counter with $D_3D_2D_1D_0$)

4-Bit Counter with Parallel Load



4-Bit Counter with Parallel Load VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
    PORT ( R                : IN     INTEGER RANGE 0 TO 15 ;
           Clock, Resetn, L : IN     STD_LOGIC ;
           Q                : BUFFER INTEGER RANGE 0 TO 15 ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q + 1 ;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

INTEGER data object: the number of bits is not specified explicitly. Only the range is specified, from which the number of bits is deduced

BUFFER: Data flows out of the entity and the **entity can read** the signal (allowing for internal feedback)

Asychronous reset

Parallel (synchronous) load

Since Q is declared as BUFFER: the **entity can directly read** and write to the Q port

4-Bit Counter with Parallel Load VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
    PORT ( R                : IN     INTEGER RANGE 0 TO 15 ;
           Clock, Resetn, L : IN     STD_LOGIC ;
           Q                : BUFFER INTEGER RANGE 0 TO 15 ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF L = '1' THEN
                Q <= R ;
            ELSE
                Q <= Q + 1 ;
            END IF;
        END IF;
    END PROCESS;
END Behavior;
```

Package STANDARD predefines a number of types, subtypes, and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit.

The INTEGER data type is in the "standard types" category which is defined in the "standard" package from "std" library: don't need to explicitly make the "standard" package and the "std" library visible to the design.

Down Counter with Parallel Load VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY downcnt IS
    GENERIC ( modulus : INTEGER := 8 ) ;
    PORT ( Clock, L, E : IN  STD_LOGIC ;
           Q           : OUT INTEGER RANGE 0 TO modulus-1 ) ;
END downcnt ;

ARCHITECTURE Behavior OF downcnt IS
    SIGNAL Count : INTEGER RANGE 0 TO modulus-1 ;
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
        IF L = '1' THEN
            Count <= modulus-1 ;
        ELSE
            IF E = '1' THEN
                Count <= Count-1 ;
            END IF ;
        END IF ;
    END PROCESS;
    Q <= Count ;
END Behavior ;
```

GENERIC parameter named modulus: the starting count and the size of the counter can be changed during instantiation

OUT: Data flows out of the entity and the **entity cannot read** these signals

To be able to read and write the count, an internal variable named Count is declared

No sensitivity list: the VHDL code inside the process block will run continuously (the program loops back to the start of the block after executing the last line).

Parallel load (initial value determined by the parameter named modulus

Read and write the internal signal named Count

Assign the value of the internal signal named Count to the OUTPUT Q

Outline

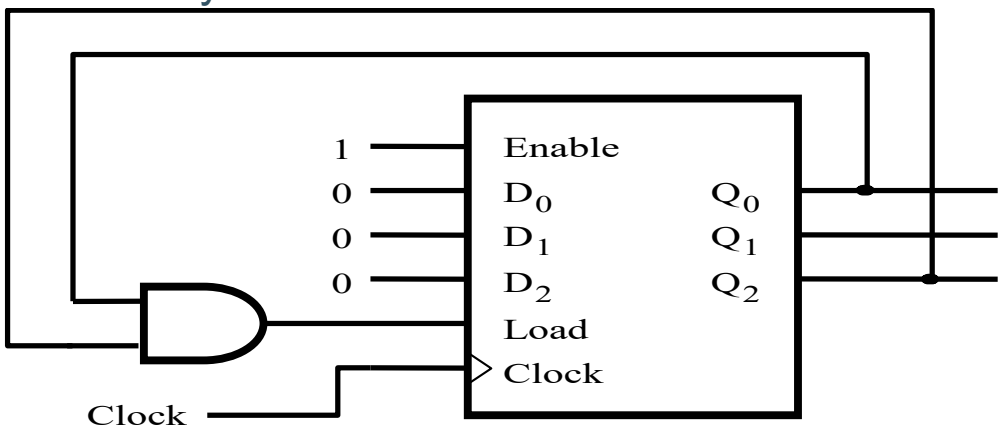
- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Modulo-x Counter

- An n-bit up-counter naturally works as a modulo-2^n counter
- We can also design a counter with an arbitrary modulo base: for modulo-x counter, the counter should reset to 0 with the first positive clock edge after the counter becomes x-1
- Recognize when the count reaches (x-1) and then reset the counter with the next positive clock edge

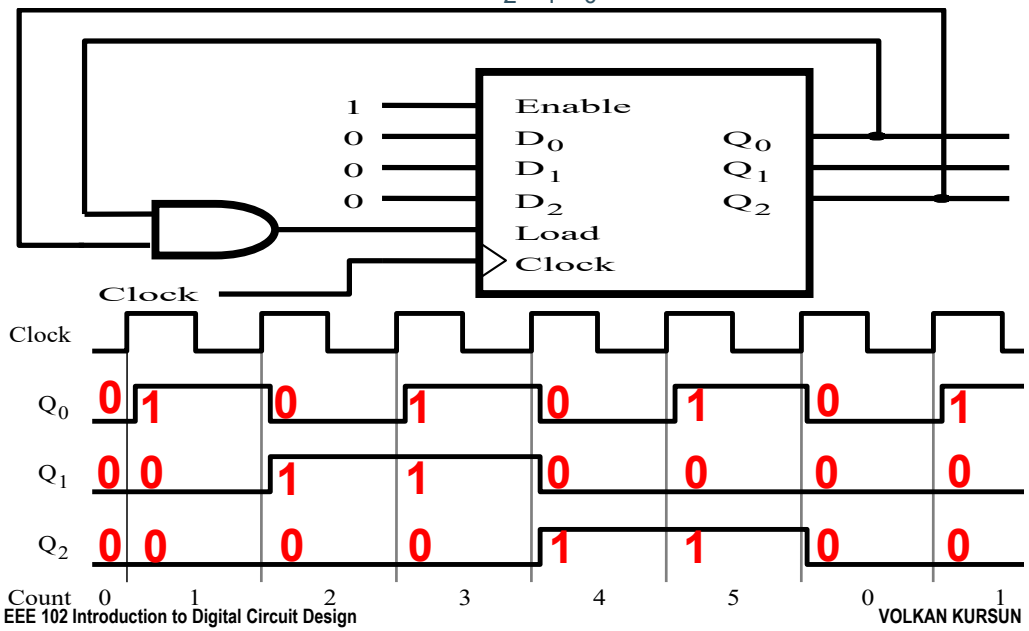
Modulo-6 Counter

- Recognize when the count reaches 5 and then reset the counter with the next positive clock edge (**synchronous reset**): use an AND gate to check if $Q_2 = Q_0 = 1$, which is true for the first time only for 5 in a 3-bit counter



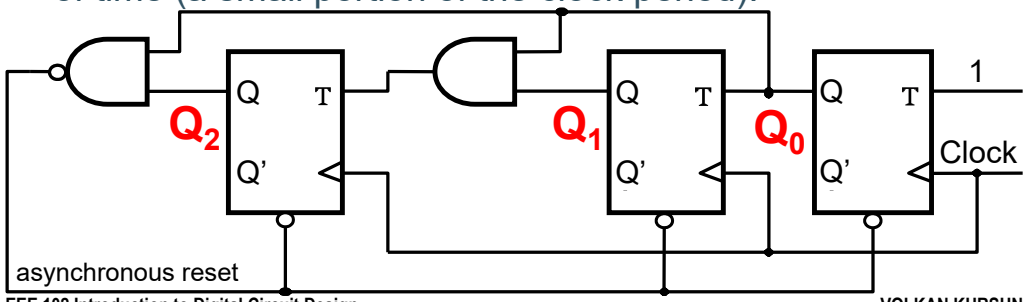
Modulo-6 Counter

- Parallel load of counter is used to reset its contents when the count reaches 5: load $D_2D_1D_0$ = 0b000 into the counter



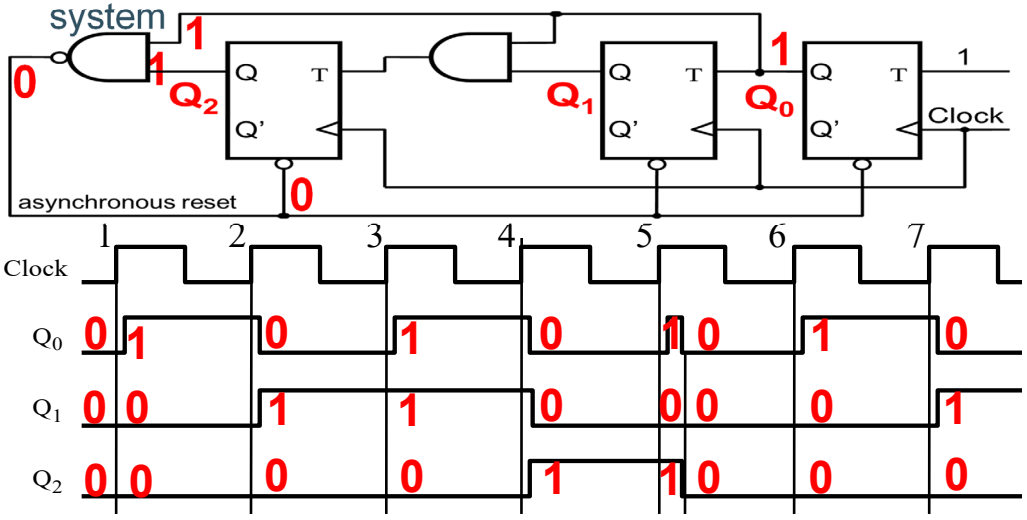
Modulo-6 with Asynchronous Reset

- Instead of the parallel load feauture, you may consider using the **asynchronous reset** inputs of the individual flip-flops when the count reaches 5: use a NAND gate to detect when count = 5 and clear all flip-flops
- **Problem with asynchronous reset**: as soon as the count reaches 5, the NAND gate triggers the resetting action. The flip-flops are cleared with some delay after the count reaches 5. The count is maintained at 5 for a short period of time (a small portion of the clock period).



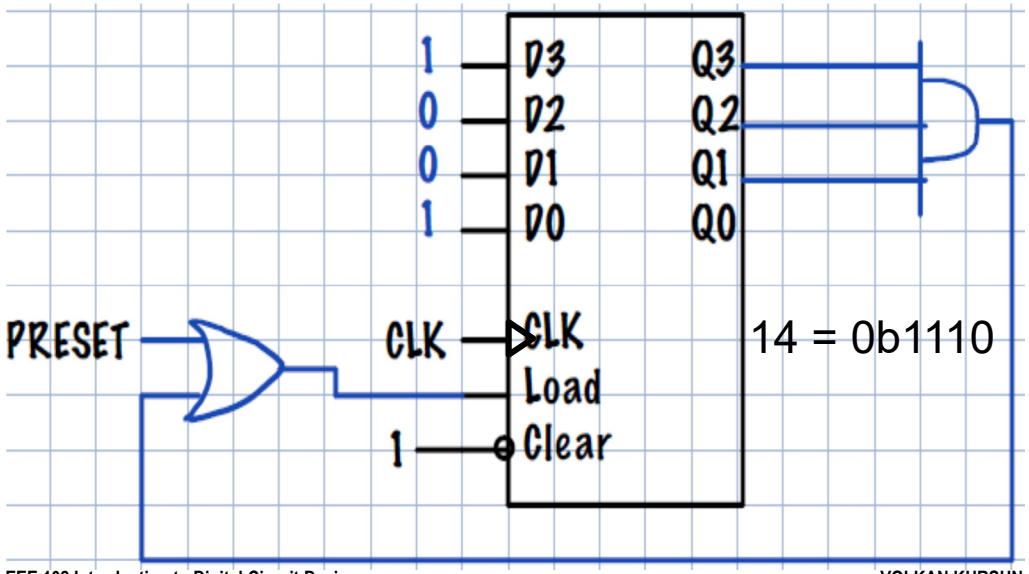
Modulo-6 with Asynchronous Reset

- **Problem with asynchronous reset**: The **narrow time when count = 5** may **NOT** be sufficient for this output to be detected and correctly used by the rest of the digital system



9-to-14 Counter

- **Synchronous preset** to 9 when PRESET = 1 or **synchronous load** to 9 when count = 14

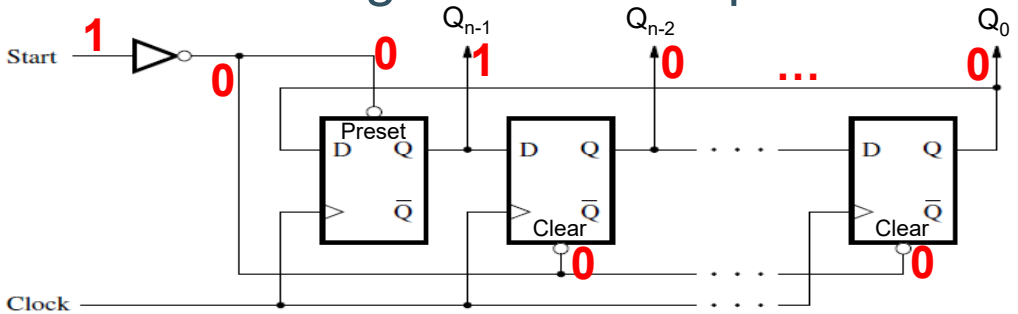


Outline

- Registers
- Shift Register
- Asynchronous Counters
- Synchronous Counters
- Counters with Parallel Load
- Modulo-x Counters (RESET)
- Ring Counters

Shift Right Ring Counter

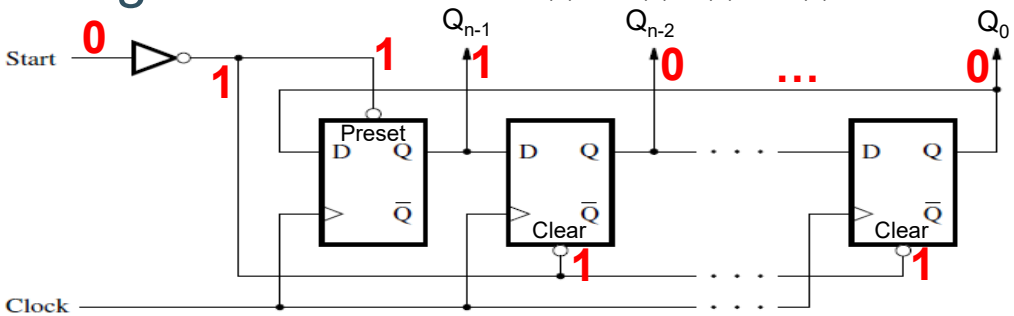
- 1000→0100→0010→0001→1000…
- Can be implemented with a shift-register with the following connections
- **Start = 1**: inject 1 into the first stage while clearing the other bit positions



Shift Right Ring Counter

- 1000→0100→0010→0001→1000…
- Can be implemented with a shift-register with the following connections
- **Start = 0**: count with the positive edges of the clock

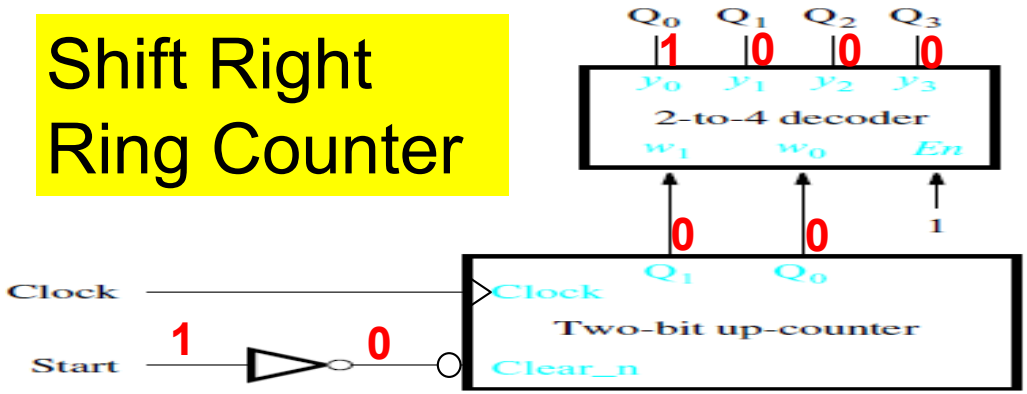
Q(N-1) <= Q(0), Q(N-2) <= Q(N-1),…,
Q(1) <= Q(2), Q(0) <= Q(1)



Ring Counter with Decoder

- Can be implemented with a 2-bit up counter and a decoder
- **Start = 1**: counter is **reset to 0b00**

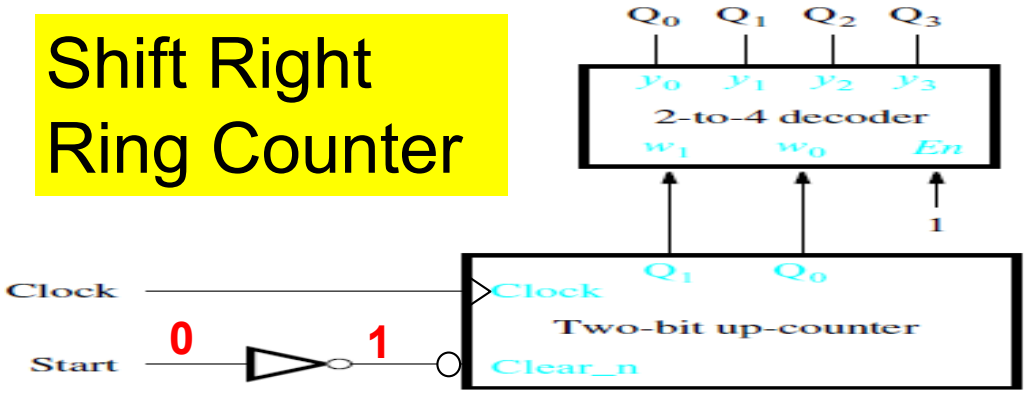
Shift Right Ring Counter



Ring Counter with Decoder

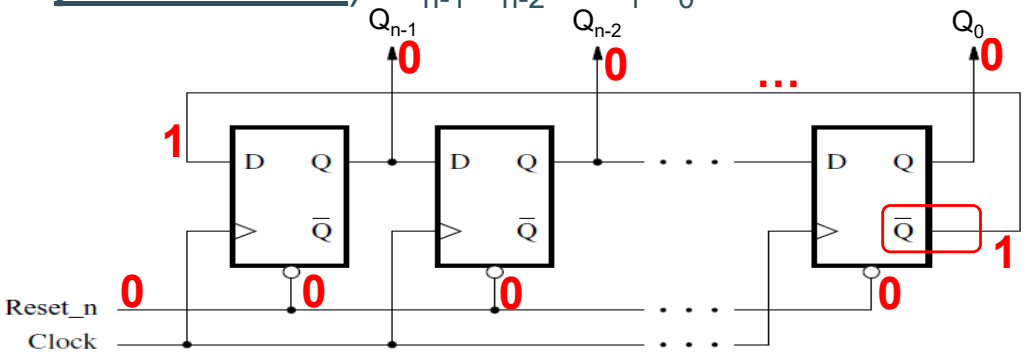
- **Start = 0**: counter starts counting with the positive edges of the clock
- For the count values **0b00**, **0b01**, **0b10**, **0b11**, **0b00**, the decoder produces **0b1000**, **0b0100**, **0b0010**, **0b0001**, **0b1000**

Shift Right Ring Counter



Shift Right Johnson Counter

- Instead of the Q output of the final stage as in a ring counter, connect the Q' output of the final stage to the D input of the first stage flip-flop in a Johnson counter
- First **initialize** with **Reset_n = 0** (**clear all bit positions to 0**): $Q_{n-1}Q_{n-2}\ldots Q_1Q_0$ = 0b00…00



Shift Right Johnson Counter

- When **Reset_n = 1**, counter **starts counting** with the positive edges of the clock signal
- 4-bit Johnson counter example:
  0000 $\xrightarrow{1}$ 1000 $\xrightarrow{2}$ 1100 $\xrightarrow{3}$ 1110 $\xrightarrow{4}$ 1111 $\xrightarrow{5}$ 0111 $\xrightarrow{6}$ 0011 $\xrightarrow{7}$ 0001 $\xrightarrow{8}$ **0000** → 1000…