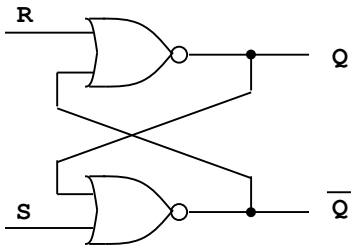
Synchronous Sequential Circuits

Synchronous Components

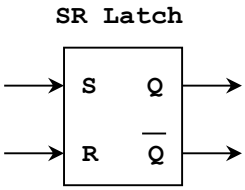
Asynchronous Circuits: Latches

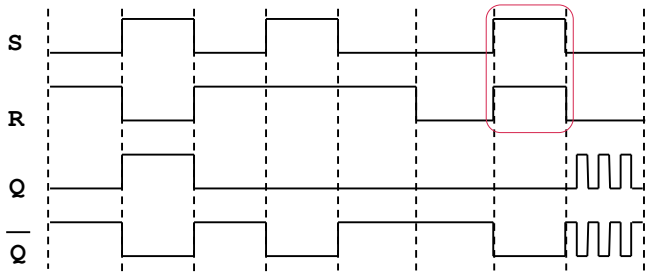
SR LATCH:



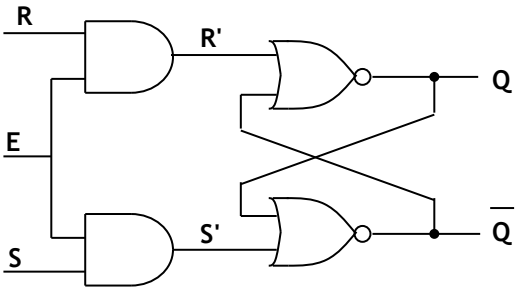
S	R	Q_{t+1}	$\overline{Q}_{t+1}$
0	0	Q_t	$\overline{Q}_t$
0	1	0	1
1	0	1	0
1	1	0	0

restricted





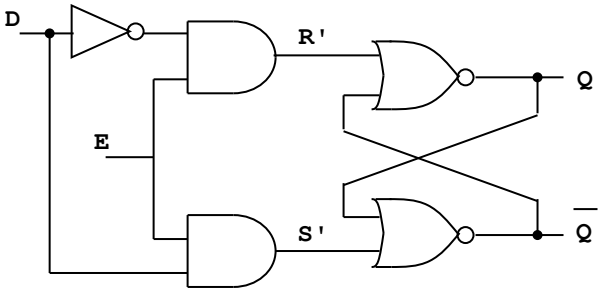
SR LATCH WITH ENABLE:



E	S	R	Q_{t+1}	$\overline{Q}_{t+1}$
0	x	x	Q_t	$\overline{Q}_t$
1	0	0	Q_t	$\overline{Q}_t$
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

D LATCH WITH ENABLE:

- This is essentially an SR Latch, where $R = not(D),\ S = D$

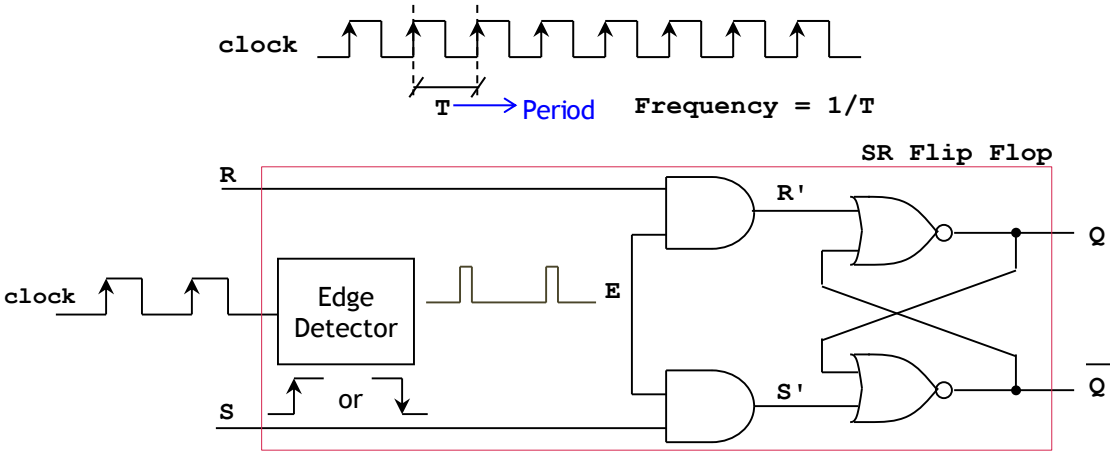


E	D	Q_{t+1}
0	x	Q_t
1	0	0
1	1	1

SYNCHRONOUS CIRCUITS: FLIP FLOPS

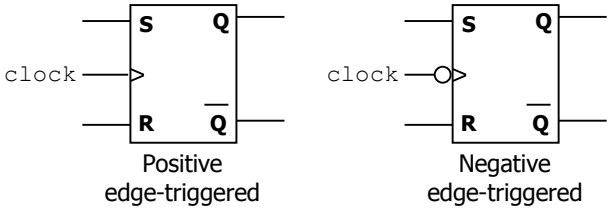
- Flip flops are made out of:
  - A Latch with an enable input.
  - An Edge detector circuit.

- The figure depicts an SR Latch, where the enable is connected to the output of an *Edge Detector* Circuit. The input to the Edge Detector is a signal called '***clock***'. A clock signal is a square wave with a fixed frequency.



- The edge detector circuit generates short-duration pulses during rising (or falling) edges. These pulses act as short-time enables of the Latch.
- The behavior of the flip flops can be described as that of a Latch that is only enabled during rising (or falling edges).

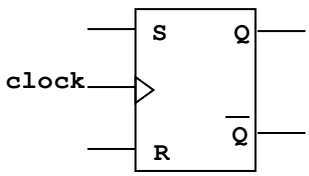
- Depending on what type of edge we are detecting, flip flops can be classified as:
  - Positive-edge triggered flip flop: The edge detector circuit generates pulses during rising edges.
  - Negative-edge triggered flip flop: The edge detector circuit generates pulses during falling edges.
  - Dual-edge triggered flip flop: The edge detector circuit generates pulses during both rising and falling edges. Current FPGA technology does not support dual-edge triggered flip flops.



FLIP FLOP TYPES

SR Flip Flop

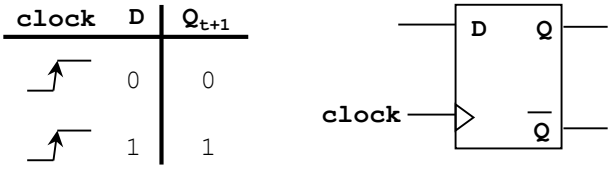
clock	S	R	Q_{t+1}	$\overline{Q_{t+1}}$
↑	0	0	Q_t	$\overline{Q_t}$
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	0	0

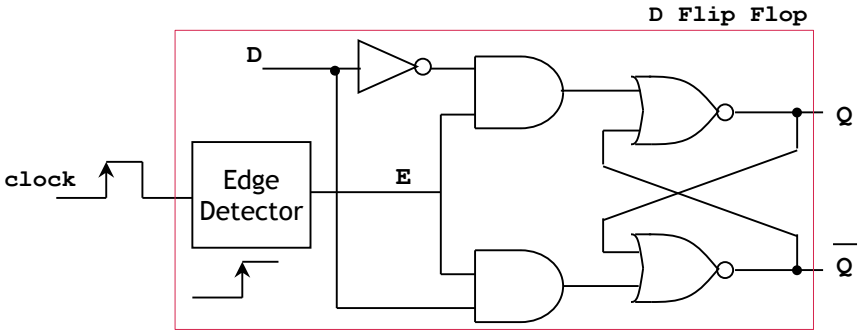


Excitation Equation:

$$Q_{t+1} = S\bar{R} + Q_t\bar{S}\bar{R} = \bar{R}(S + Q_t\bar{S}) = \bar{R}(S + \bar{S})(S + Q_t) = \bar{R}S + \bar{R}Q_t \text{ (on the edge)}$$

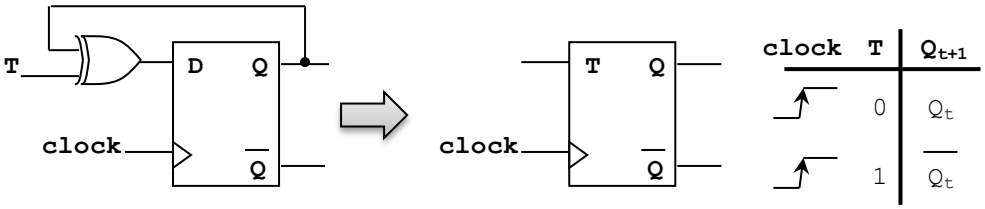
D Flip Flop





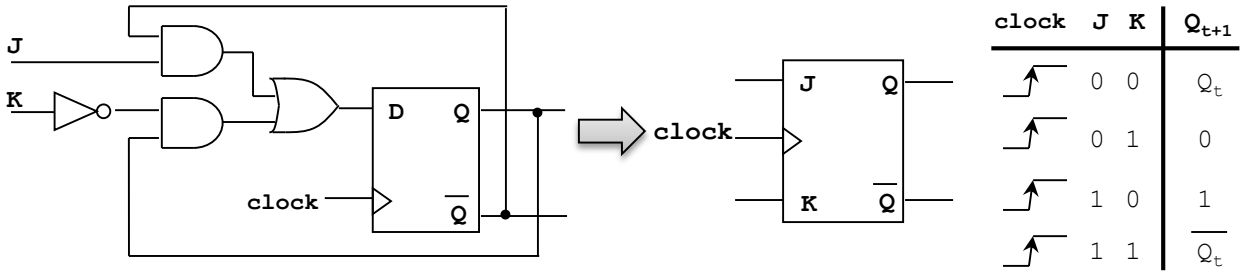
Excitation Equation: $Q_{t+1} = D$ (on the edge)

T Flip Flop



Excitation Equation: $Q_{t+1} = T \oplus Q_t$ (on the edge)

JK Flip Flop



Excitation Equation: $Q_{t+1} = J\overline{Q_t} + \overline{K}Q_t$ (on the edge)

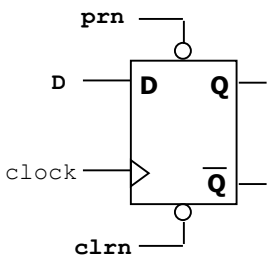
SYNCHRONOUS AND ASYNCHRONOUS INPUTS

Synchronous Inputs

- Typically, flip flops only change their outputs on the rising (or falling edge). Usually, a change on the inputs forces a change on the outputs. These inputs are known as *synchronous inputs*, as the inputs' state is only checked on the rising (or falling) edges. Example: Input D of a D flip flop, Inputs J, K of a JK flip flop.

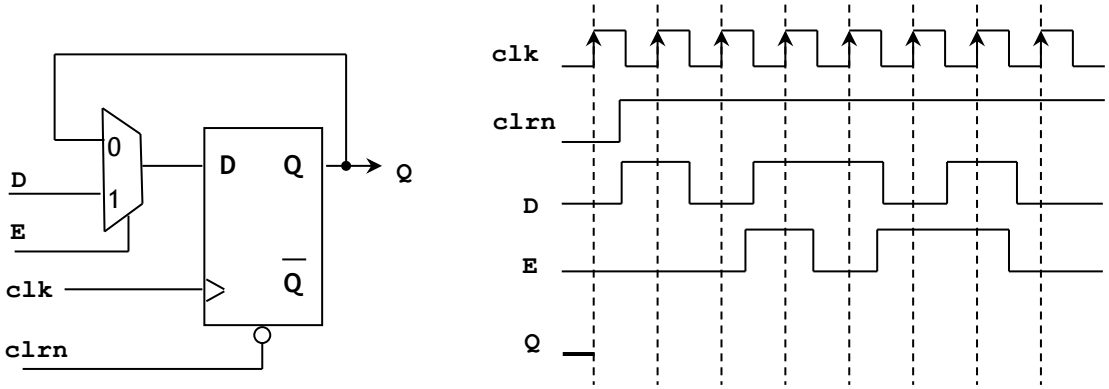
Asynchronous Inputs

- However, in many instances, it is useful to have inputs that force the outputs to a value immediately, disregarding the rising (or falling edges). These inputs are known as *asynchronous inputs*. Common asynchronous inputs are prn and $clrn$ (they can be active-low or active high)
- The figure depicts a D Flip Flop with two asynchronous inputs:
  - prn: Preset (active low). When $prn = 0 \rightarrow Q = 1$.
  - $clrn$ (sometimes called $resetn$): Clear (active low). When $clrn = 0 \rightarrow Q = 0$.
  - If prn and $clrn$ are both 0, usually $clrn$ is given priority.
- A Flip flop can have more than one asynchronous inputs, or none.



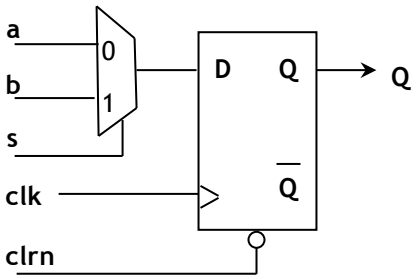
PRACTICE EXERCISES

1. Complete the timing diagram of the circuit shown below:

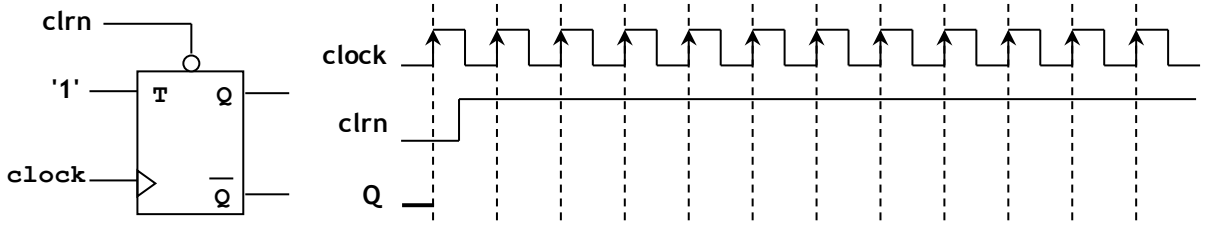


2. Complete the VHDL description of the circuit shown below:

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

entity circ is
  port ( a, b, s, clk, clrn: in std_logic;
         q: out std_logic);
end circ;

architecture a of circ is

begin
  -- ???

end a;
```

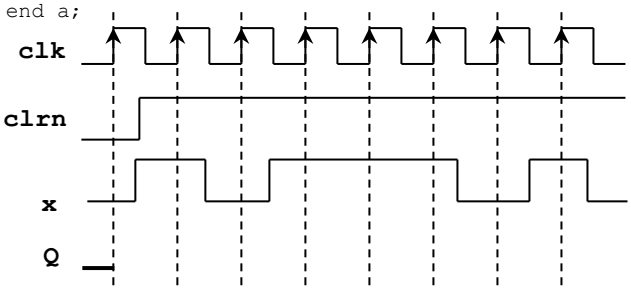


3. Complete the timing diagram of the circuit shown below. If the frequency of the signal clock is 25 MHz, what is the frequency (in MHz) of the signal Q?

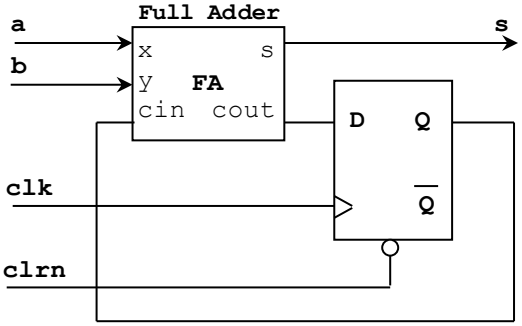


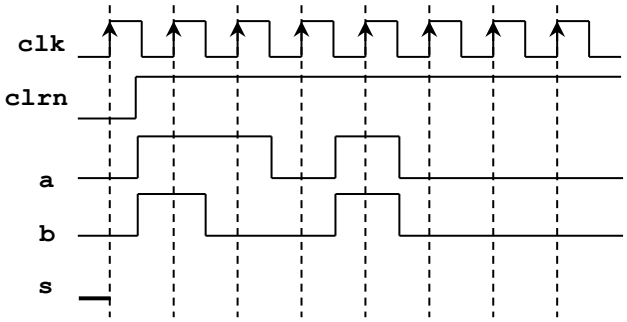
4. Complete the timing diagram of the circuit whose VHDL description is shown below:

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

entity circ is
  port ( clrn, x, clk: in std_logic;
         q: out std_logic);
end circ;

architecture a of circ is
   signal qt: std_logic;
begin
   process (clrn, clk, x)
   begin
     if clrn = '0' then
         qt <= '0';
      elsif (clk'event and clk = '1') then
         if x = '1' then
             qt <= not (qt);
         end if;
      end if;
   end process;
   q <= qt
end a;
```



5. Complete the timing diagram of the circuit shown below:



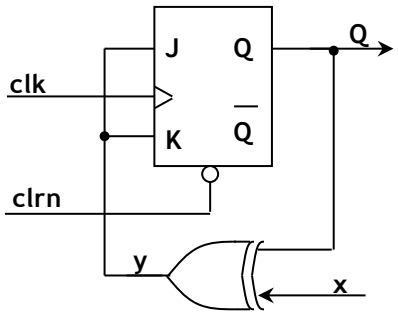


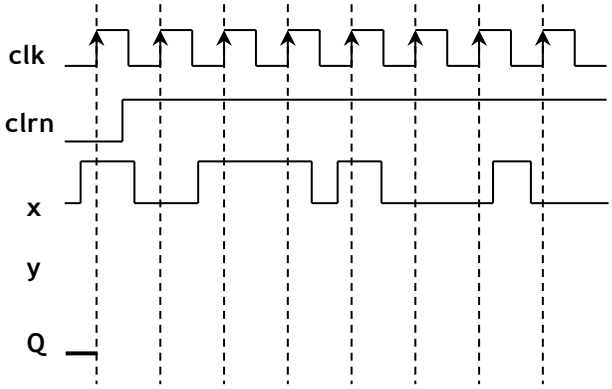
6. Complete the VHDL description of the synchronous sequential circuit whose truth table is shown below:

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

entity circ is
  port ( A, B, C: in std_logic;
         clrn, clk: in std_logic;
         q: out std_logic);
end circ;

architecture a of circ is
begin
  -- ???

end a;
```

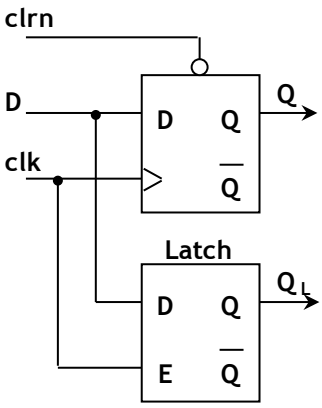
clrn	clk	A	B	Q_{t+1}
1	↑	0	0	1
1	↑	0	1	C
1	↑	1	0	$\overline{Q_t}$
1	↑	1	1	Q_t
0	X	X	X	0

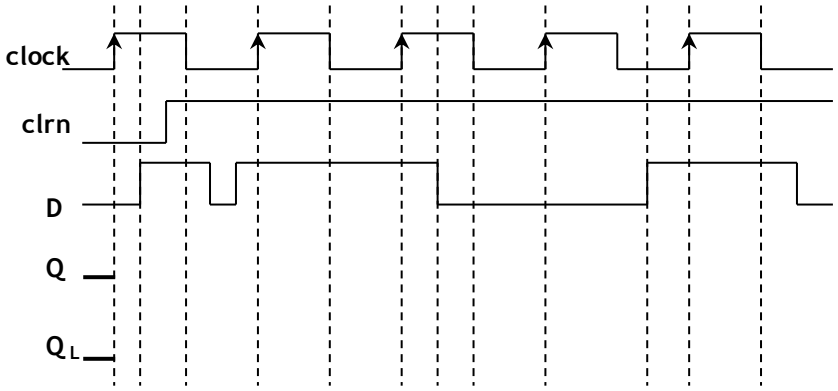
7. Complete the timing diagram of the circuit shown below:





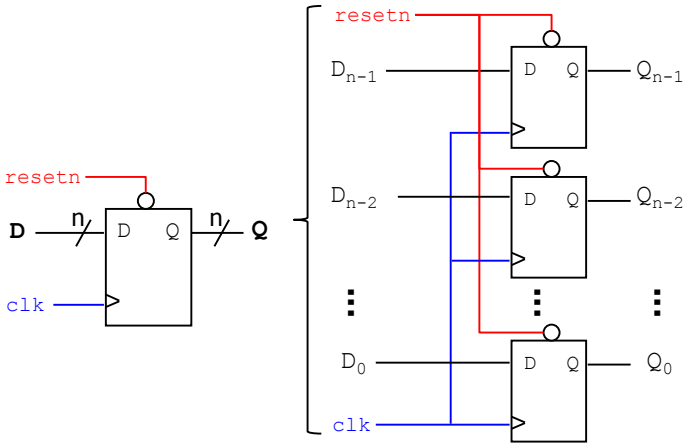
8. Complete the timing diagram of the circuit shown below:





SYNCHRONOUS CIRCUITS: REGISTERS

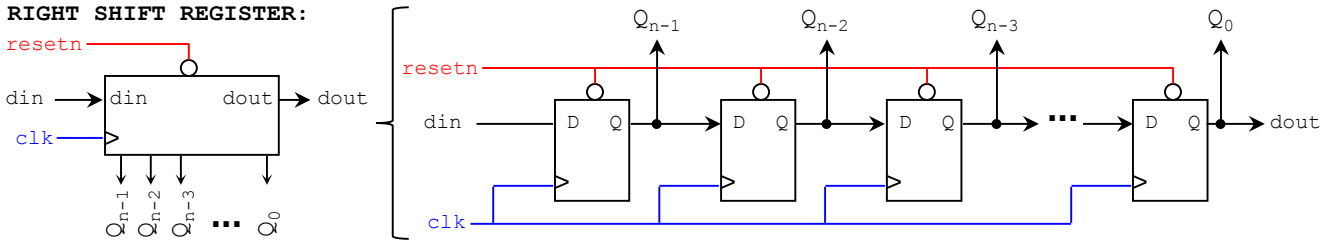
n-bit Register: This is a collection of 'n' D-type flip flops, where each flip flop independently stores one bit. The flip flops are connected in parallel. They also share the same `resetn` and `clock` signals.



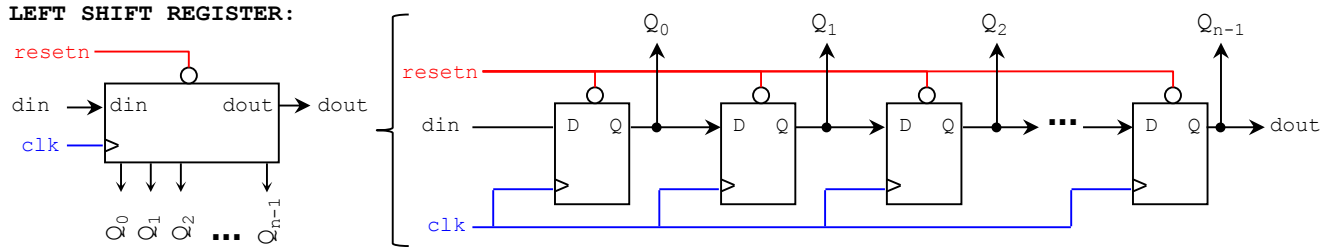
n-bit shift registers: This is a collection of 'n' D-type flip flops, connected serially. The flip flops share the same `resetn` and `clock` signals. The serial input is called 'din', and the serial output is called 'dout'. The flip flop outputs (also called the parallel output) are called $Q = Q_{n-1}Q_{n-2}\cdots Q_0$. Depending on how we label the bits, we can have:

- **Right shift register:** The input bit moves from the MSB to the LSB, and
- **Left shift register:** The input bit moves from the LSB to the MSB.

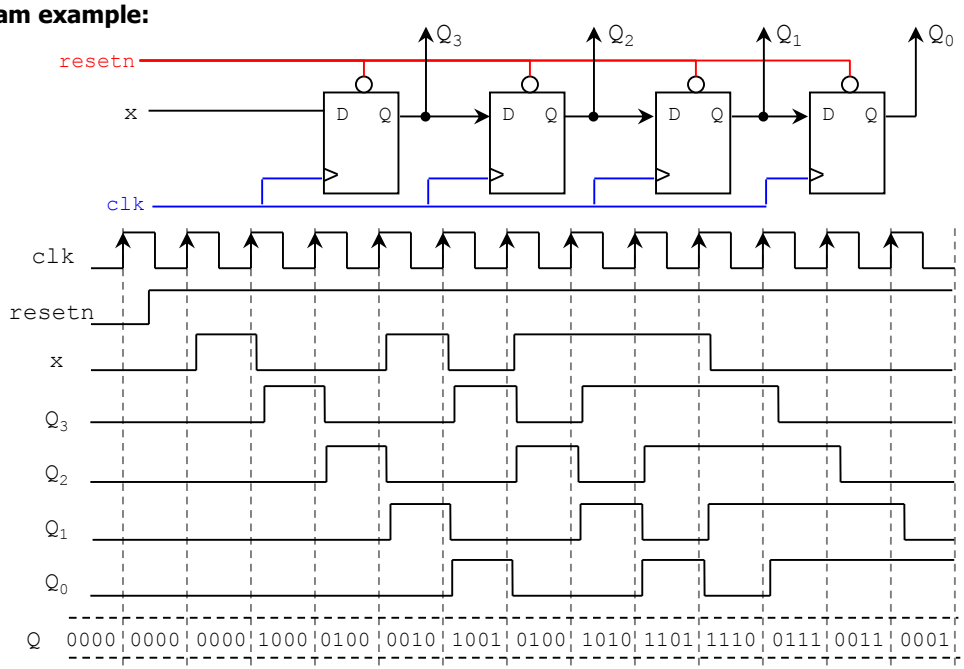
RIGHT SHIFT REGISTER:



LEFT SHIFT REGISTER:

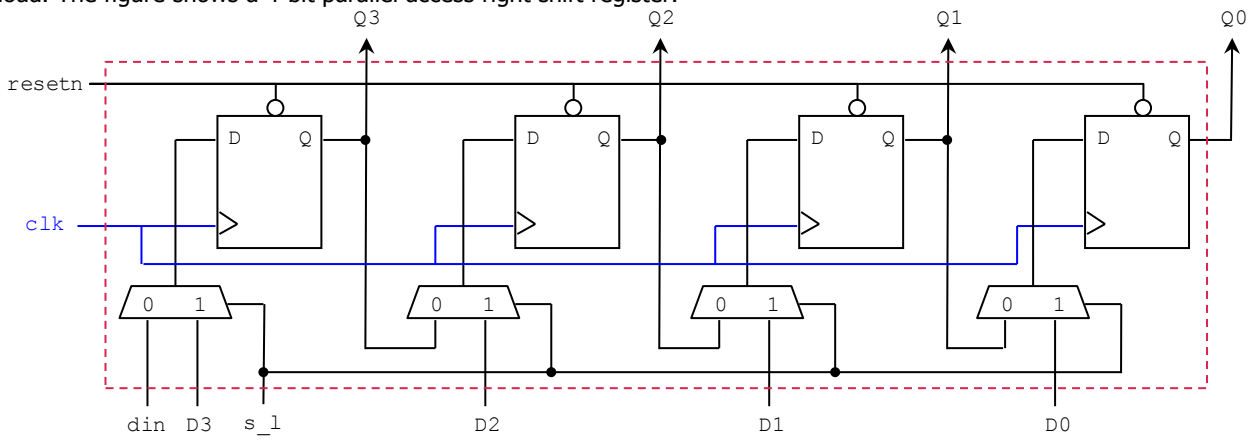


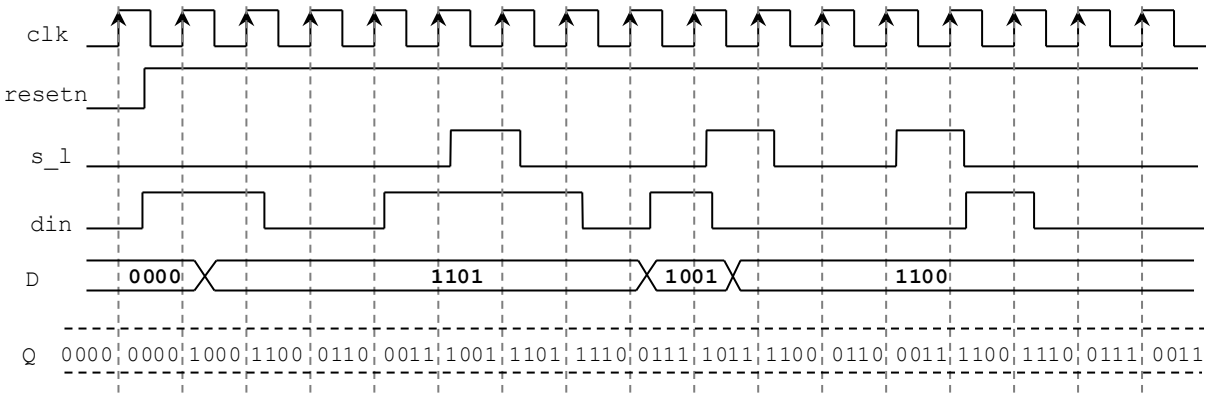
Timing Diagram example:



PARALLEL ACCESS SHIFT REGISTER

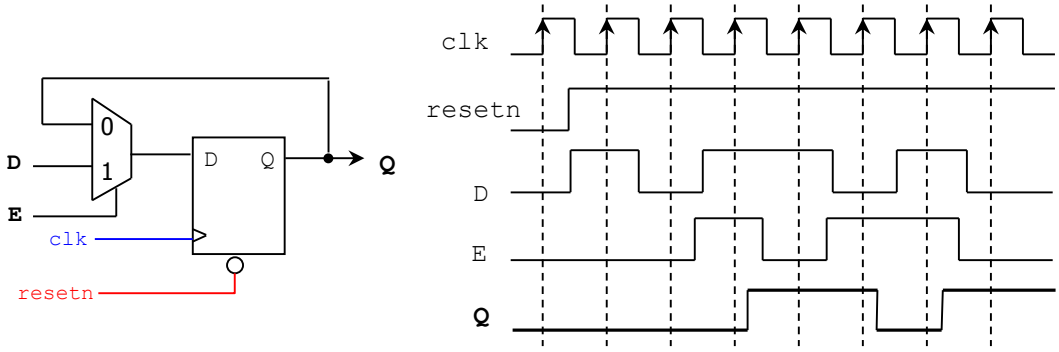
- This is a shift register in which we can write data on the flip flops in parallel. $s\_l = 0 \rightarrow$ shifting operation, $s\_l = 1 \rightarrow$ parallel load. The figure shows a 4-bit parallel access right shift register.



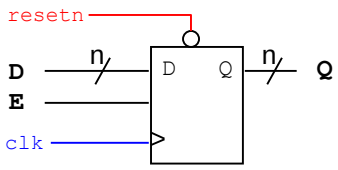


ADDING AN ENABLE INPUT TO FLIP FLOPS

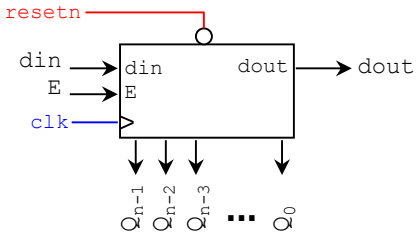
- In many instances, it is very useful to have a signal that controls whether the flip flop works. The following circuit represents a D flip flop with synchronous enable. When E = '0', the flip flop does not work, i.e., the flip flop keeps its value. When E = '1', the flip flop works, i.e., Q=D on the rising edge.
- We can thus create n-bit registers and n-bit shift registers with enable. Here, all the flip flops share the same enable input.



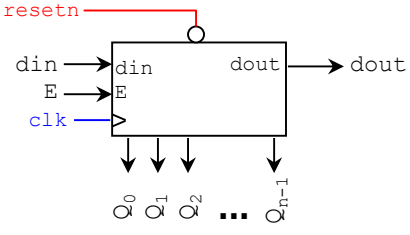
REGISTER:



RIGHT SHIFT REGISTER:

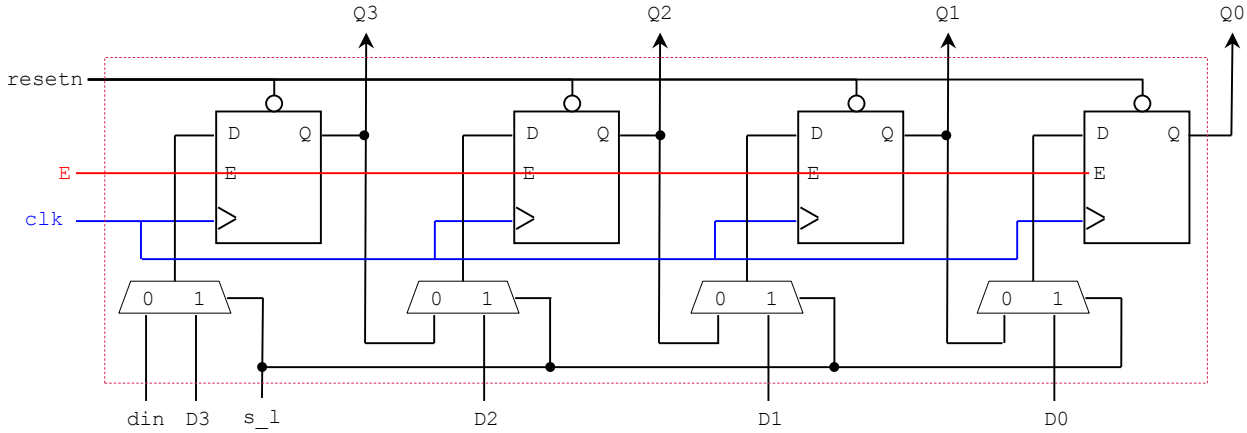


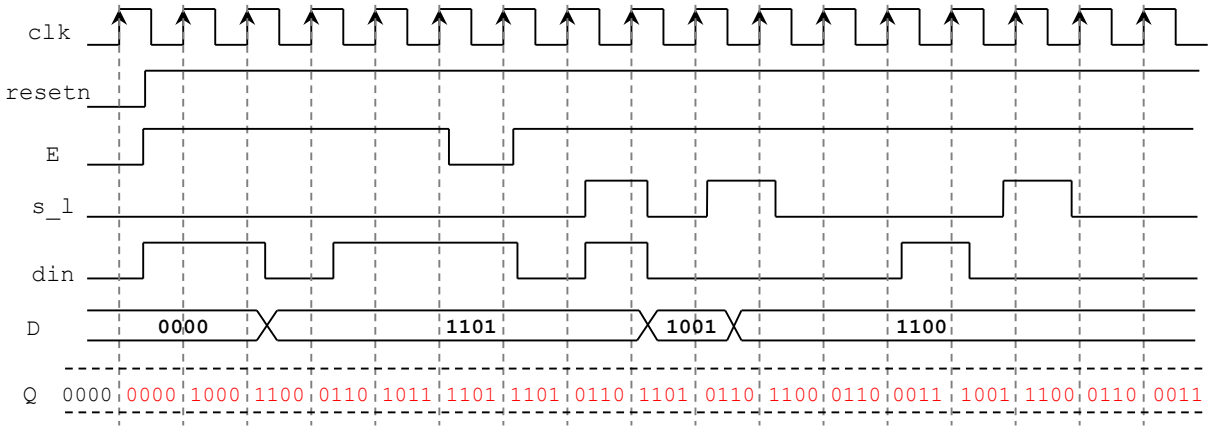
LEFT SHIFT REGISTER:



Parallel access shift register with enable

- All the flip flops share the same enable input.

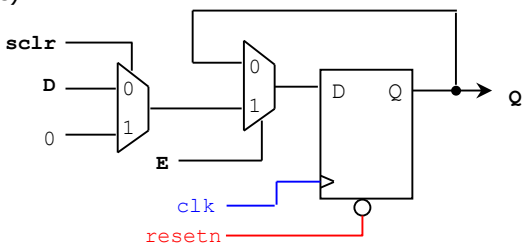




ADDING A SYNCHRONOUS CLEAR INPUT TO FLIP FLOPS

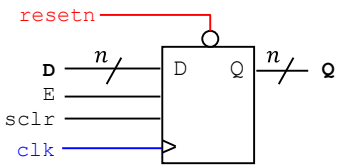
- In many instances, it is very useful to have a signal that can clear the value of the flip flop but only on the rising (or falling edges): synchronous clear ($sclr$). Typically, all synchronous signals are validated by enable. For example, for a D flip flop, the table show how the output state Q changes (on the rising edge):

E	$sclr$	Q (output state)
0	X	$Q \leftarrow Q$
1	0	$Q \leftarrow D$
1	1	$Q \leftarrow 0$

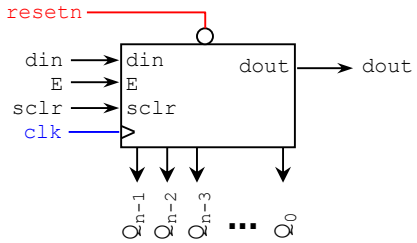


- We can thus create n-bit registers and n-bit shift registers with enable and $sclr$. Here, all the flip flops share the same enable and $sclr$ inputs.

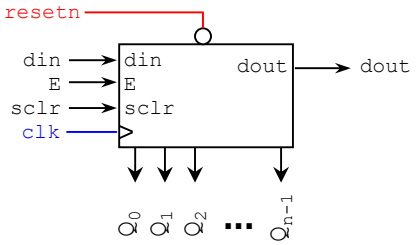
REGISTER:



RIGHT SHIFT REGISTER:



LEFT SHIFT REGISTER:

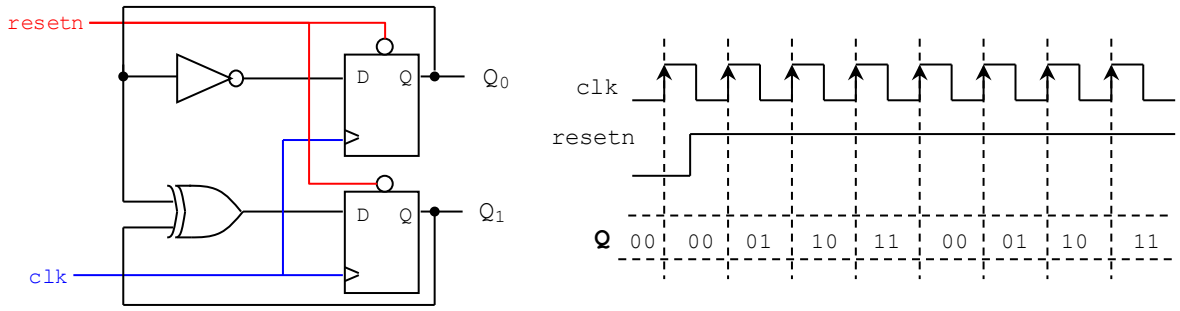


SYNCHRONOUS COUNTERS

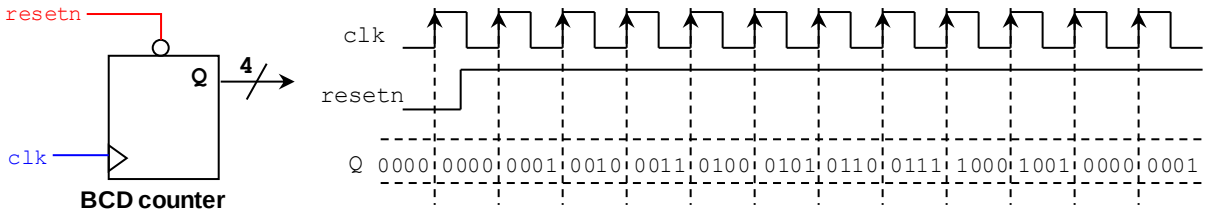
- Counters are useful for: counting the number of occurrences of a certain event, generate time intervals for task control, track elapsed time between two events, etc. Counters are made of flip flops and combinatorial logic. Common design strategies include using the Finite State Machines (FSM) method or a synchronous accumulator.
- Synchronous counters change their output on the clock edge (rising or falling). Each flip flop shares the same clock input signal. If the initial count is zero, each flip flop shares the $resetn$ input signal.

COUNTER CLASSIFICATION:

a) **Binary counter**: An $n-bit$ counter counts from 0 to $2^n - 1$. The figure depicts a 2-bit counter.



b) **Modulus counter**: A counter $modulo-N$ counts from 0 to N-1. Special case: BCD (or decade) counter: Counts from 0 to 9.

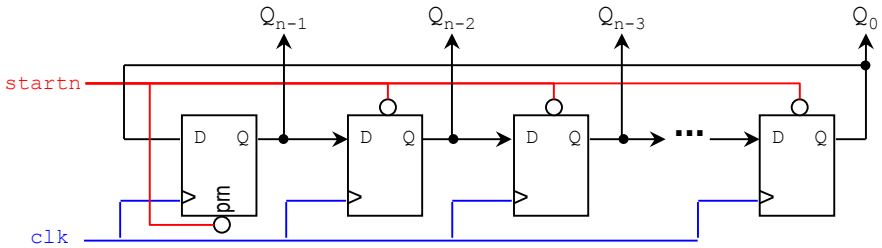


c) **Up/down counter**: Counts both up and down, under command of a control input.

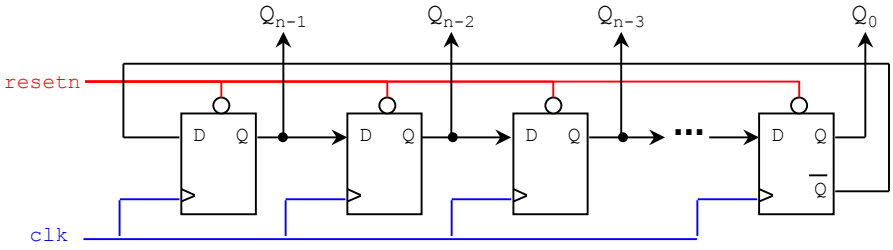
d) **Parallel load counter**: The count can be given an arbitrary value.

e) **Counter with enable**: If enable = 0, the count stops. If enable = 1, the counter counts. This is usually done by connecting the enable inputs of the flip flops to a single enable.

f) **Ring counter**: Also called one-hot counter (only one bit is 1 at a time). It can be constructed using a shift register. The output of the last stage is fed back to the input to the first stage, which creates a ring-like structure. The asynchronous signal `startn` sets the initial count to 100…000 (first bit set to 1). Example (4-bits): 1000, 0100, 0010, 0001, 1000, … The figure below depicts an $n-bit$ ring counter.

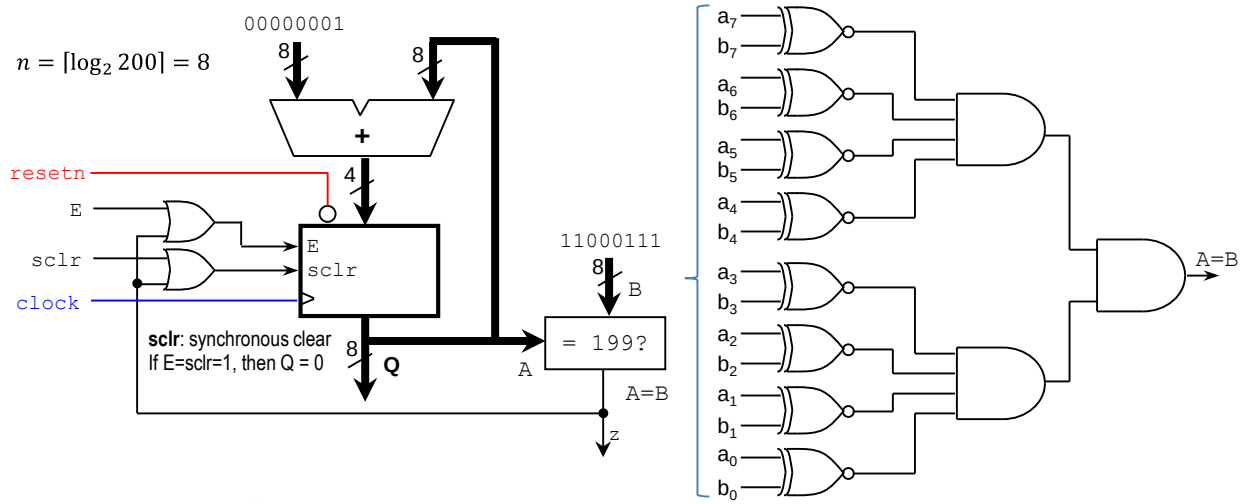


g) **Johnson counter**: Also called twisted ring counter. It can be constructed using a shift register, where the $\bar{Q}$ output of the last flip flop is fed back to the first stage. The result is a counter where only a single bit has a different value for two consecutive counts. All the flip flops share the asynchronous signal 'resetn', which sets the initial count to 000…000. Example (4 bits): `0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000,` … The figure below depicts an $n-bit$ Johnson counter.



COUNTER DESIGN USING AN ACCUMULATOR:

- The figure shows a counter modulo-200 that uses a register, and adder, a comparator, and logic gates.
- Q: count. z: output signal asserted only when the maximum count (199) is reached.

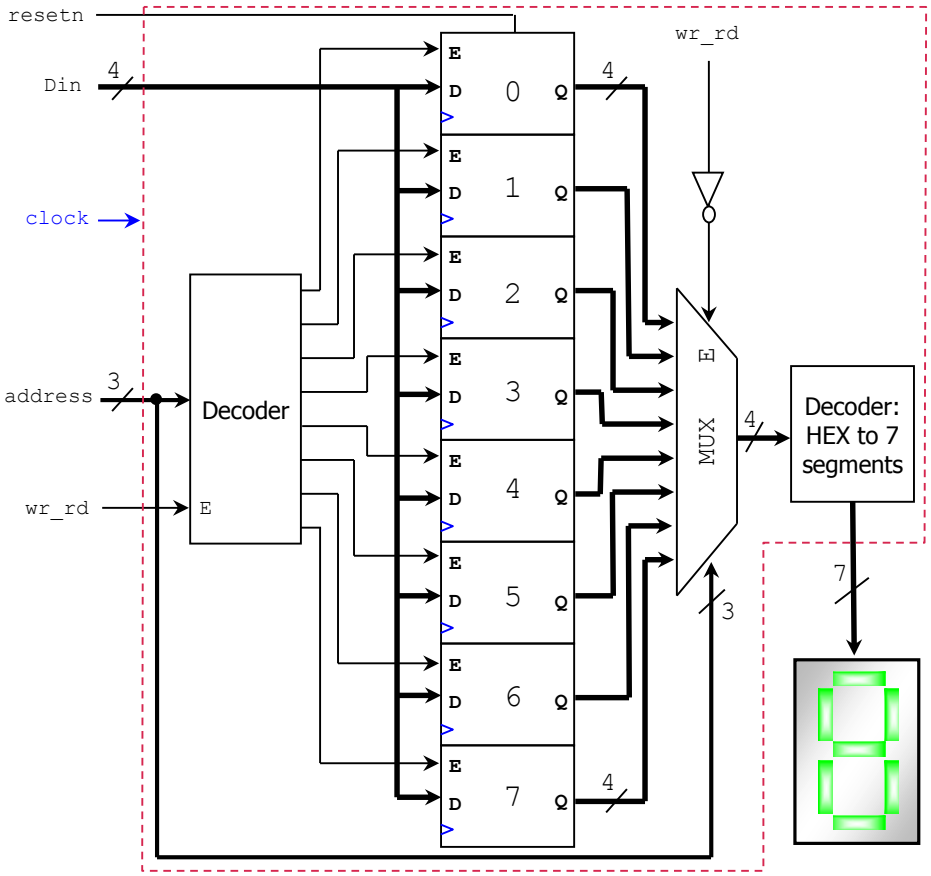


RANDOM ACCESS MEMORY EMULATOR

- The following sequential circuit represents a memory with 8 addresses, where each address holds a 4-bit data. The memory positions are implemented by 4-bit registers. The reset and clock signals are shared by all the registers. Data is written or read onto/from one of the registers (selected by the signal $address$).

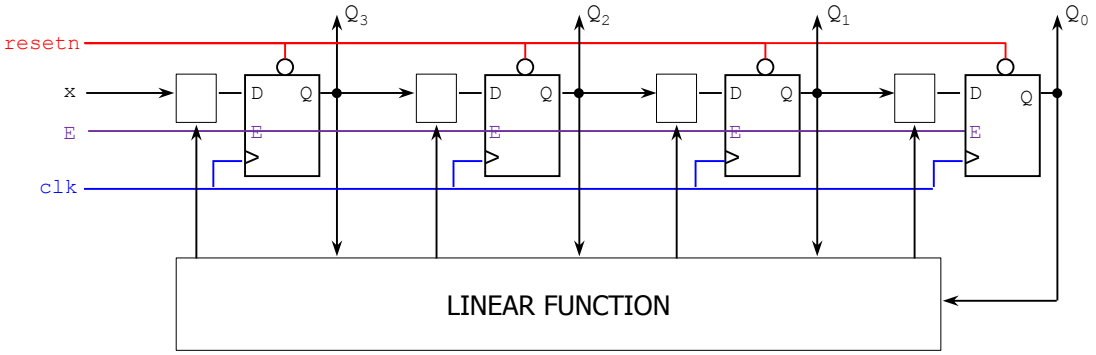
- **Writing onto memory** ($wr\_rd = 1$): The 4-bit input data (D_in) is written into one of the 8 registers. The address signal selects which register is to be written. Here, the 7-segment display must show 0. For example: if address = "101", then D_in is written into register 5.

- **Reading from memory** ($wr\_rd = 0$): The MUX output appears on the 7-segment display (hexadecimal value). The address signal selects the register from which data is read.
  For example: If address = "010", then data in register 2 must appear on the 7-segment display. If data in register 2 is '1010', then the symbol 'A' appears on the 7-segment display.



LINEAR FEEDBACK SHIFT REGISTERS (LFSRS)

- LFSRs have a simple and fairly regular structure. Typical components: D-type flip flops, and logic gates.
- Advantages: very little hardware and high speed operation.
- Inputs to flip flops are linear functions of the previous state.
- Despite the simple appearance, LFSRs are based on rather complex mathematical theory and have interesting applications in the area of digital system testing, cryptography, and fault-tolerant computing.
- Typical applications: Error correction/detection, pseudo random number generators, fast counters.
- A generic 4-bit LFSR is depicted below:

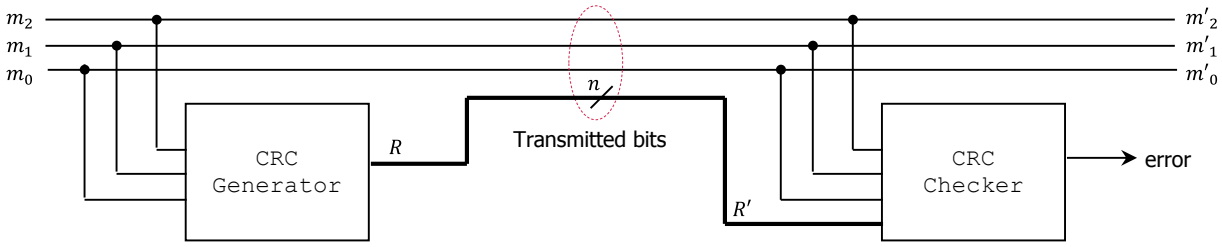


APPLICATION: CYCLIC REDUNDANCY CHECK (CRC)

- This error-detection code is used in digital communications (e.g.: Ethernet, CAN), and storage devices (e.g.: RAMs).

Communication system

- k-bit message: $M = m_{k-1}m_{k-2} \dots m_1 m_0$.
- CRC code: $R = r_{n-1}r_{n-2} \dots r_1 r_0$.
- Stream sent: $m_{k-1}m_{k-2} \dots m_1 m_0 r_{n-1}r_{n-2} \dots r_1 r_0$



- Associated polynomials:
  - ✓ $M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x^1 + m_0x^0$. Order: $k-1$
  - ✓ $R(x) = r_{n-1}x^{n-1} + r_{n-2}x^{n-2} + \cdots + r_1x^1 + r_0x^0$. Order: $n-1$
- The CRC code is calculated as a remainder: $R(x) = remainder\left(\frac{x^n M(x)}{G(x)}\right)$
- $G(x) = g_n x^n + g_{n-1}x^{n-1} + \cdots + g_1x^1 + g_0x^0$. This is the generating polynomial of order n. $G = g_n g_{n-1} \dots g_1 g_0$

- Here, when dealing with polynomial operations, we use modulo-2 polynomial arithmetic (Galois field of two elements: GF(2)). The coefficients of the polynomials can only be 0 or 1. The multiplication operation can be implemented as a simple AND gate, and the addition (or subtraction) operation can be implemented by a XOR gate: $a \pm b = a \oplus b, a \in \{0,1\}$.
  - ✓ Example: $x^n + x^n = x^n - x^n = 0$.

- **CRC generator**: It computes $R(x)$. Its input is $x^n M(x) \equiv m_{k-1}m_{k-2} \dots m_1 m_0 00 \dots 0$.

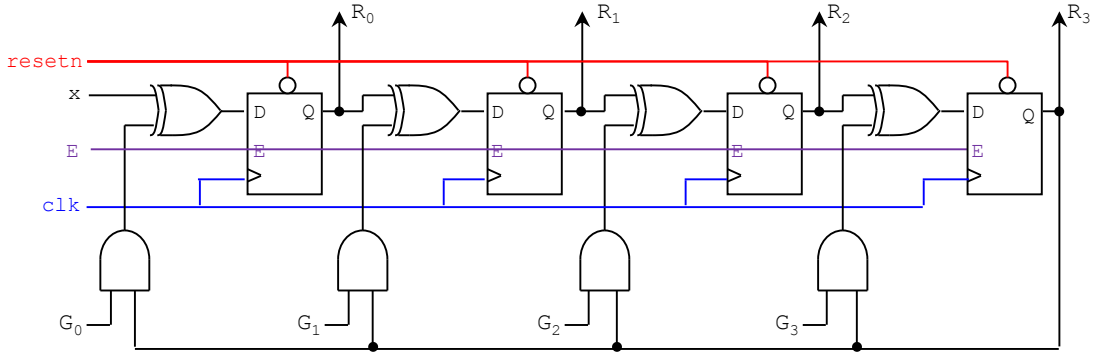
- **CRC checker**: On the receiver side, both the message and CRC code might be corrupted by the channel: $M'(x)$ and $R'(x)$.
  - ✓ So, we must check if $remainder\left(\frac{x^n M'(x)}{G(x)}\right) = R'(x)$. If so, we say that the system passed the CRC check.
  - ✓ Note that this implies: $x^n M'(x) = Q'(x)G(x) + R'(x) \rightarrow x^n M'(x) - R'(x) = x^n M'(x) + R'(x) = Q'(x)G(x)$. In modulo-2 arithmetic, $R'(x) = -R'(x)$. It follows that $x^n M'(x) + R'(x)$ should be divisible by $G(x)$.
  - ✓ The CRC checker tests if $remainder\left(\frac{x^n M'(x) + R'(x)}{G(x)}\right) = 0$. If the remainder is zero, we say that it passed the CRC check. It is still possible for $R'(x)$ and $M'(x)$ to make the remainder equal to zero but this is far less likely.
  - ✓ The input to the CRC checker is $x^n M'(x) + R'(x) \equiv m'_{k-1}m'_{k-2} \dots m'_0 r'_{n-1}r'_{n-2} \dots r'_0$.

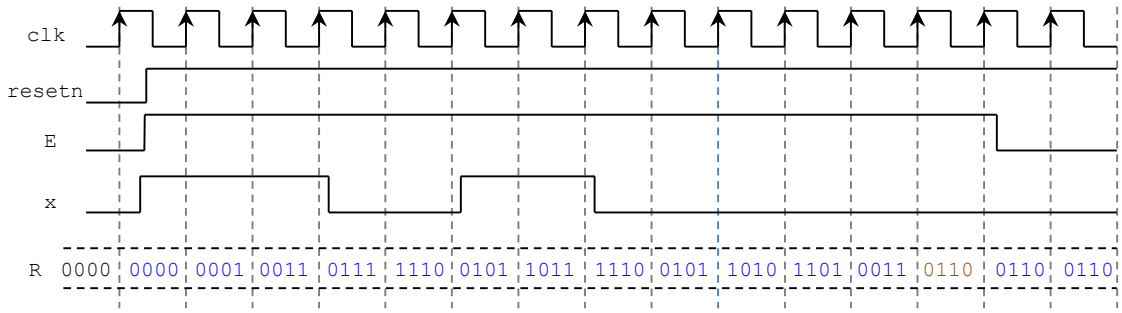
- **Example**: CRC-4 ($n = 4$): $M = 11100110$, $G = 11001$
  $M(x) = x^7 + x^6 + x^5 + x^2 + x$
  $\rightarrow x^n M(x) = x^4(x^7 + x^6 + x^5 + x^2 + x) = x^{11} + x^{10} + x^9 + x^6 + x^5$
  $G(x) = x^4 + x^3 + 1$
  $R(x) = x^2 + x$

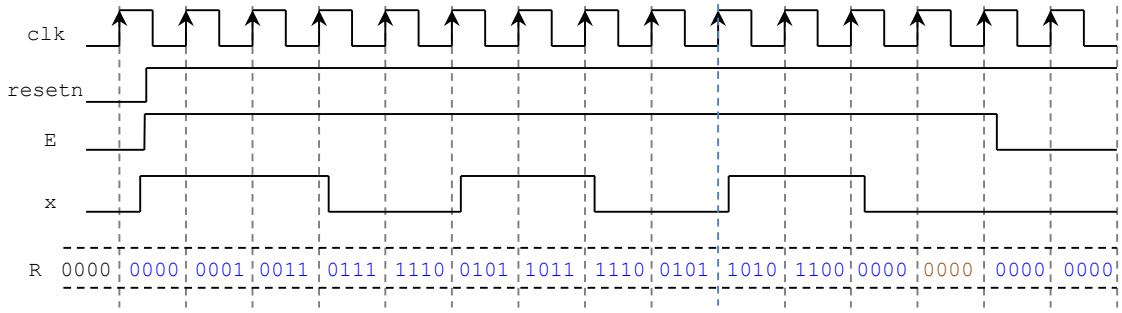
$$
\begin{array}{r}
x^7 + x^5 + x^4 + x^2 + x \\
x^4 + x^3 + 1 \;\overline{)\; x^{11} + x^{10} + x^9 + x^6 + x^5} \\
\underline{x^{11} + x^{10} + x^7} \\
x^9 + x^7 + x^6 + x^5 \\
\underline{x^9 + x^8 + x^5} \\
x^8 + x^7 + x^6 \\
\underline{x^8 + x^7 + x^4} \\
x^6 + x^4 \\
\underline{x^6 + x^5 + x^2} \\
x^5 + x^4 + x^2 \\
\underline{x^5 + x^4 + x} \\
x^2 + x
\end{array}
$$

CRC circuit

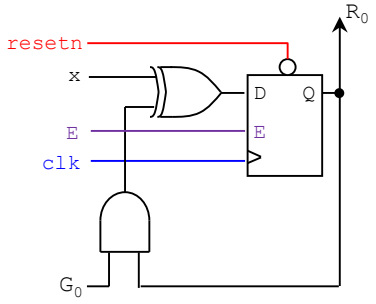
- The following LFSR implements 4-bit CRC. Components: flip flops, AND, XOR gates.
- Note that $g_4 = 1$ is not used. This is because it is common in CRC to have $g_n = 1$. This is already considered in the design of the circuit.
- Example with $M = 11100110$, $G = 11001$:
  - ✓ When integrated into the CRC generator, the input is given by: $x^n M(x) \equiv 11100110\mathbf{0000}$. The output result is $R = 0110$.
  - ✓ When integrated into the CRC checker, the input is given by: $x^n M'(x) + R'(x) \equiv 11100110\mathbf{0110}$. The result of the circuit is $remainder = 0000$, indicating a valid transmission.





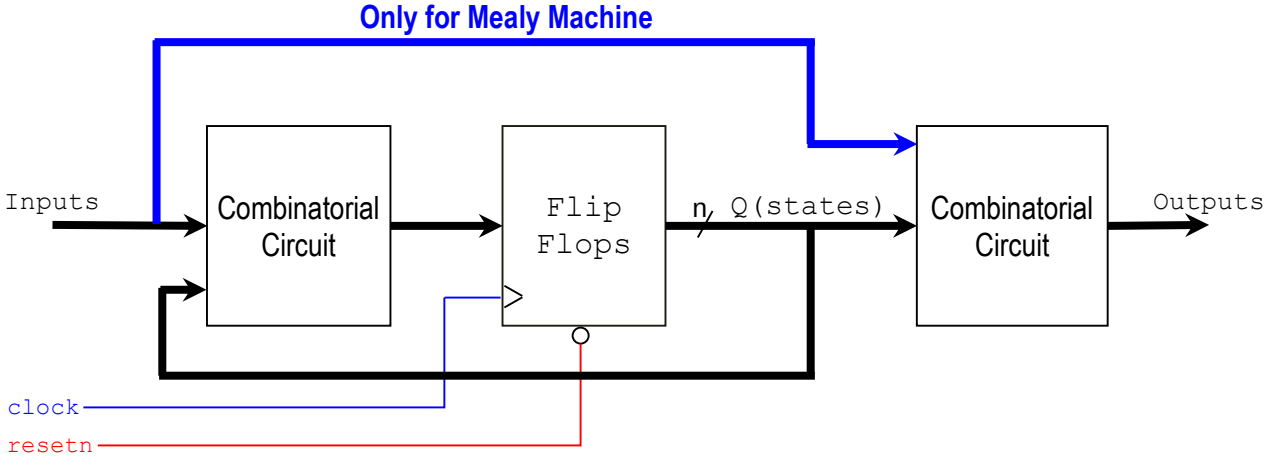


- **Applications**: The circuit above can be implemented for any number of bits n. For example: CRC-32 (Ethernet), CRC-15 (CAN), and CRC-1 (trivial even parity generator with $G(x) = x + 1$).
- This is a serial implementation, i.e., each bit is fed to the circuit at a time. Other implementation process some bits in parallel (for CRC-1, the parity generator in *Notes – Unit 1* processes all the bits at once).
- The design of the generating polynomial $G(x)$ is outside the scope of this class.
- These circuits are better implemented in hardware for high speed and low resource utilization.



FINITE STATE MACHINES:

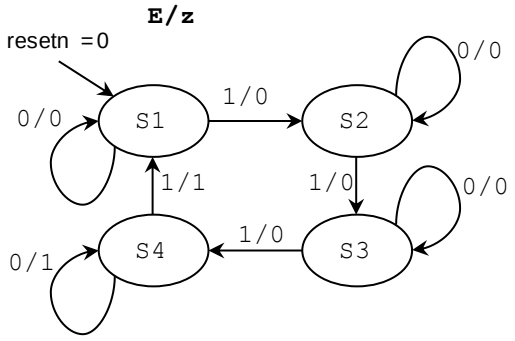
- Sequential circuits are also called Finite State Machines (FSMs), because the functional behavior of these circuits can be represented using a finite number of states (flip flop outputs).
- The signal *resetn* sets the flip flops to an initial state.
- Classification:
  - Moore machine: Outputs depend solely on the current state of the flip flops.
  - Mealy machine: Outputs depend on the current state of the flip flops as well as on the input to the circuit.



- Any general sequential circuit can be represented by the figure above (Finite State Machine model).
- A sequential circuit with certain behavior and/or specification can be formally designed using the Finite State Machine method: drawing a State Diagram and coming up the Excitation Table.
- Designing sequential circuits using Finite State Machines is a powerful method in Digital Logic Design.

Example: 2-bit gray-code counter with enable and 'z' output: 00, 01, 11, 10, 00, … The output 'z' is 1 when the present count is '10'. The count is the same as the states encoded in binary.

- *First step*: Draw the State Diagram and State Table. If we were to implement the state machine in VHDL, this is the only step we need.



E	PRESENT STATE	NEXT STATE	NEXT COUNT	z
0	S1	S1	00	0
0	S2	S2	01	0
0	S3	S3	11	0
0	S4	S4	10	1
1	S1	S2	01	0
1	S2	S3	11	0
1	S3	S4	10	0
1	S4	S1	00	1

- *Second step*: State Assignment. We assign unique flip flop states to our state labels (S1, S2, S3, S4). Notice that this is arbitrary. However, we can save resources if we assign each state to the count that we desire. Then, the output 'count' is just the flip flops' outputs.

  - ✓ S1: Q = 00
  - ✓ S2: Q = 01
  - ✓ S3: Q = 11
  - ✓ S4: Q = 10

- *Third step*: Excitation table. Here, we replace the state labels by the flip flop states:

	PRESENT STATE		NEXTSTATE		
E	**$Q_1(t)$**	**$Q_0(t)$**	**$Q_1(t+1)$**	**$Q_0(t+1)$**	**z**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	1	1	1	0
0	1	0	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	1	1	0	0
1	1	0	0	0	1

- *Fourth step*: Excitation equations and minimization. $Q_1(t+1)$ and $Q_0(t+1)$ are the next state of the flip flops, i.e. these signals are to be connected to the inputs of the flip flops.

$Q_1(t+1)$

Q_0 \ EQ_1	00	01	11	10
0	0	1	0	0
1	0	1	1	1

$Q_0(t+1)$

Q_0 \ EQ_1	00	01	11	10
0	0	0	0	1
1	1	1	0	1

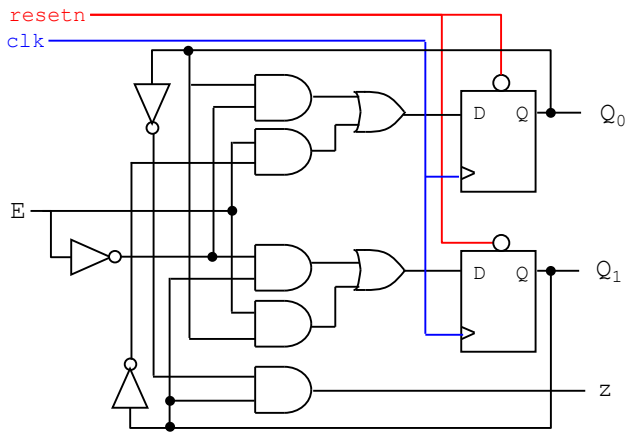
z

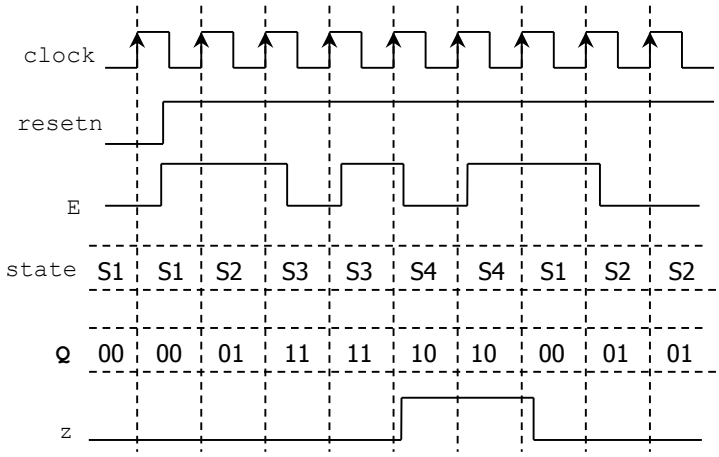
Q_0 \ EQ_1	00	01	11	10
0	0	1	1	0
1	0	0	0	0

$$Q_1(t+1) = \bar{E}Q_1 + EQ_0$$
$$Q_0(t+1) = E\overline{Q_1} + \bar{E}Q_0$$
$$z = Q_1\overline{Q_0}$$

Output z only depends on the present state. Outputs Q_1, Q_0 are the states and they only depend (in terms of the combinational output circuit) on the present state. Thus, this is a Moore FSM.

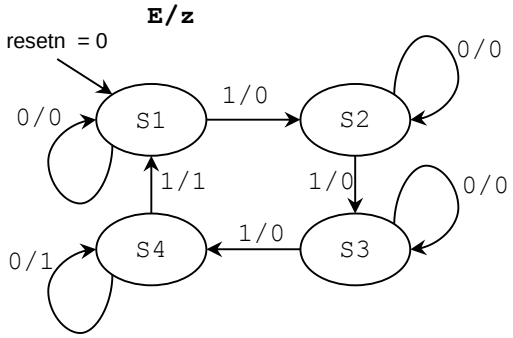
- *Fifth step:* Circuit implementation.





Example: 2-bit counter with enable and 'z' output. The output 'z' is 1 when the present count is '11'. The count is the same as the states encoded in binary.

- *First step*: Draw the State Diagram and State Table. If we were to implement the state machine in VHDL, this is the only step we need.



E	PRESENT STATE	NEXT STATE	NEXT COUNT	z
0	S1	S1	00	0
0	S2	S2	01	0
0	S3	S3	10	0
0	S4	S4	11	1
1	S1	S2	01	0
1	S2	S3	10	0
1	S3	S4	11	0
1	S4	S1	00	1

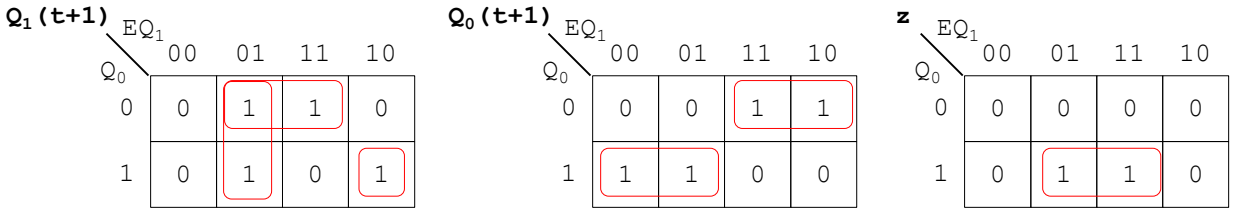
- *Second step*: State Assignment. We assign unique flip flop states to our state labels (S1, S2, S3, S4). Notice that this is arbitrary. However, we can save resources if we assign each state to the count that we desire. Then, the output 'count' is just the flip flops' outputs.

  - ✓ S1: Q = 00
  - ✓ S2: Q = 01
  - ✓ S3: Q = 10
  - ✓ S4: Q = 11

- *Third step*: Excitation table. Here, we replace the state labels by the flip flop states:

	PRESENT STATE		NEXTSTATE		
E	**$Q_1(t)$**	**$Q_0(t)$**	**$Q_1(t+1)$**	**$Q_0(t+1)$**	**z**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1

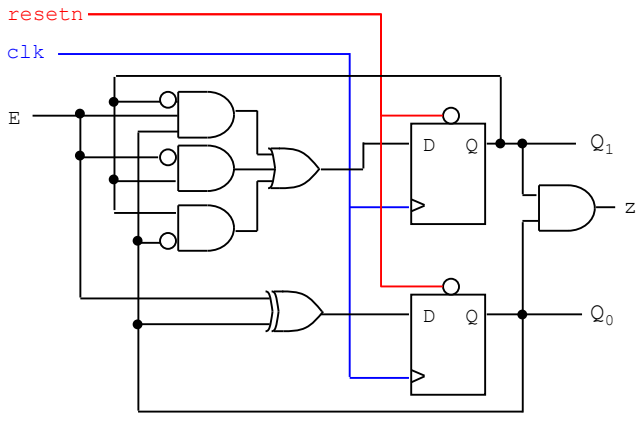
- *Fourth step*: Excitation equations and minimization. $Q_1(t+1)$ and $Q_0(t+1)$ are the next state of the flip flops, i.e. these signals are to be connected to the inputs of the flip flops.

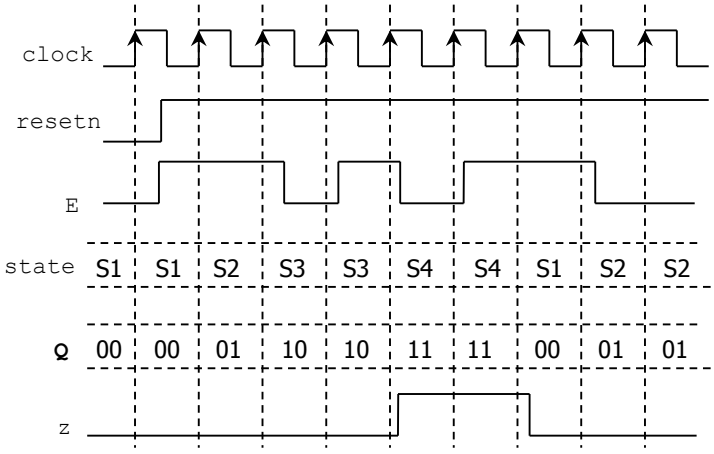


$$Q_1(t+1) = Q_1\overline{Q_0} + \bar{E}Q_1 + E\overline{Q_1}Q_0$$
$$Q_0(t+1) = E\overline{Q_0} + \bar{E}Q_0$$
$$z = Q_1Q_0$$

Output z only depends on the present state. Outputs Q_1, Q_0 are the states and they only depend (in terms of the combinational output circuit) on the present state. Thus, this is a Moore FSM.

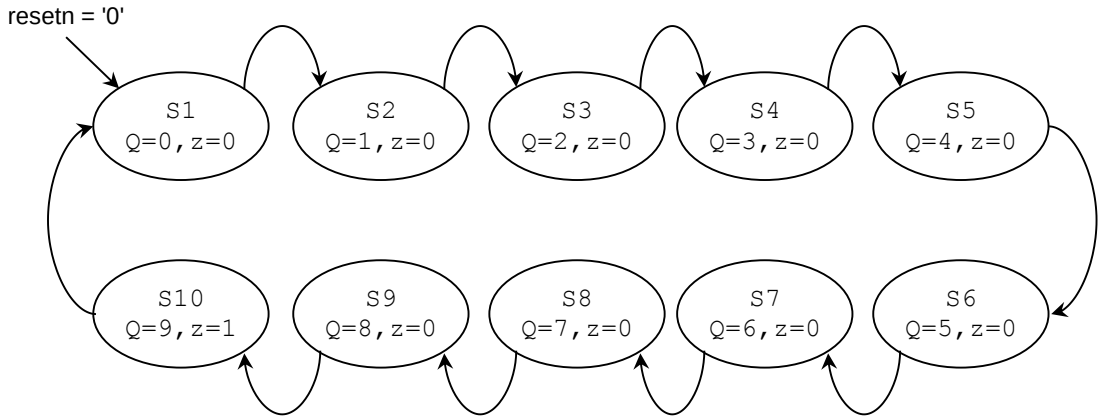
- *Fifth step:* Circuit implementation.



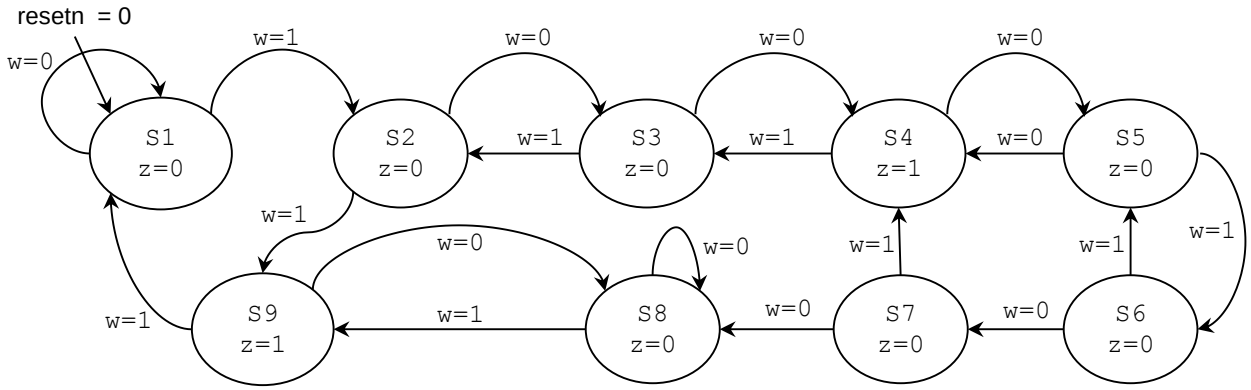


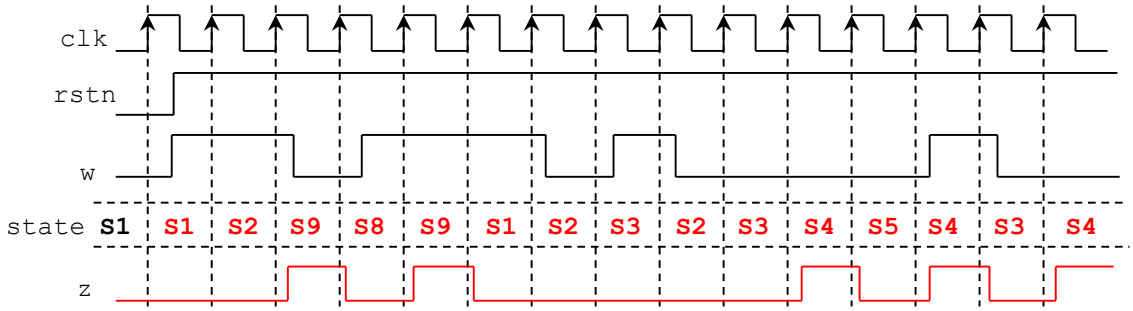
Note: In these 2-bit counters, the states are represented by the outputs of the flip flops: Q_1, Q_0. They also happen to be the outputs of the FSM. This is common in counters, as the count is usually the same as the flip flop outputs.

Example: BCD counter. Outputs: $Q(3..0)$, z. When the count reaches 1001, z becomes 1. Moore FSM



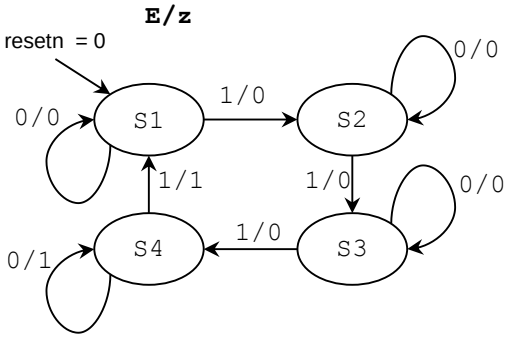
Example: FSM. Input: w. Output: z. This is a Moore FSM as z only depends on the present state.



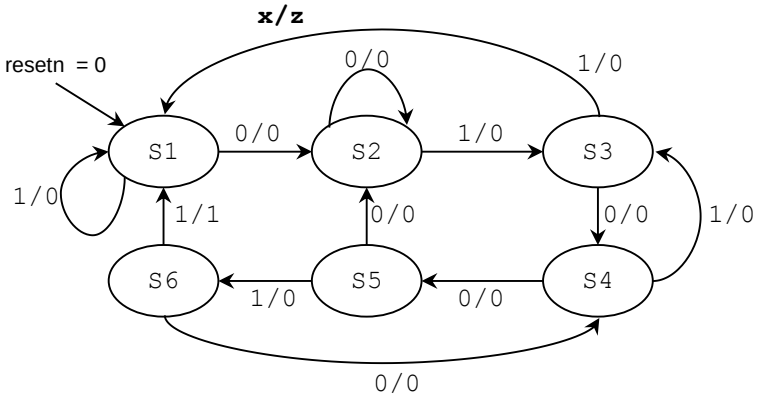


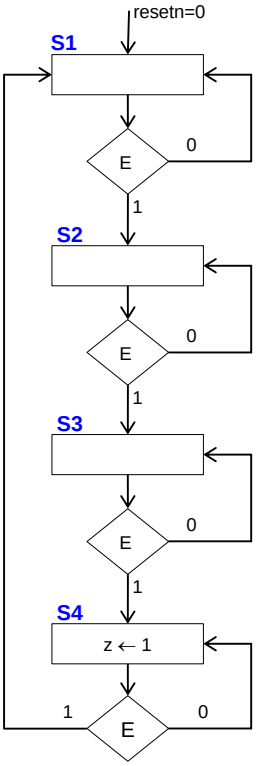
ALGORITHMIC STATE MACHINE (ASM) CHARTS:

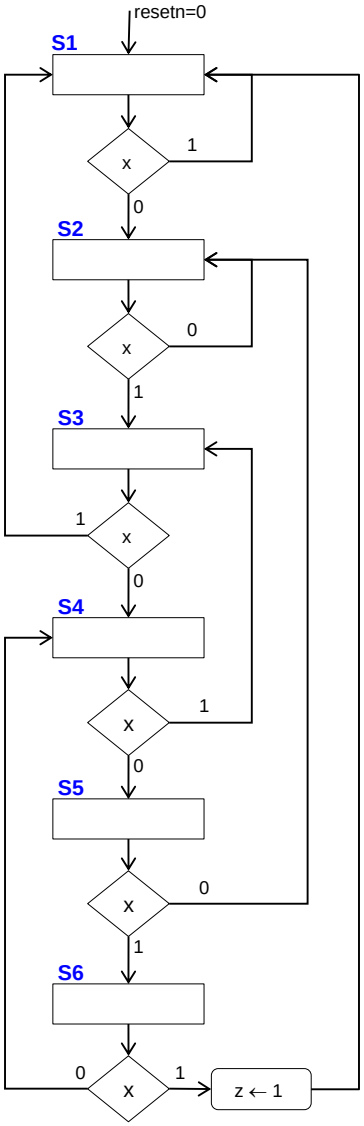
Gray counter, z=1 when Q=10



Sequence Detector (with overlap)
010011

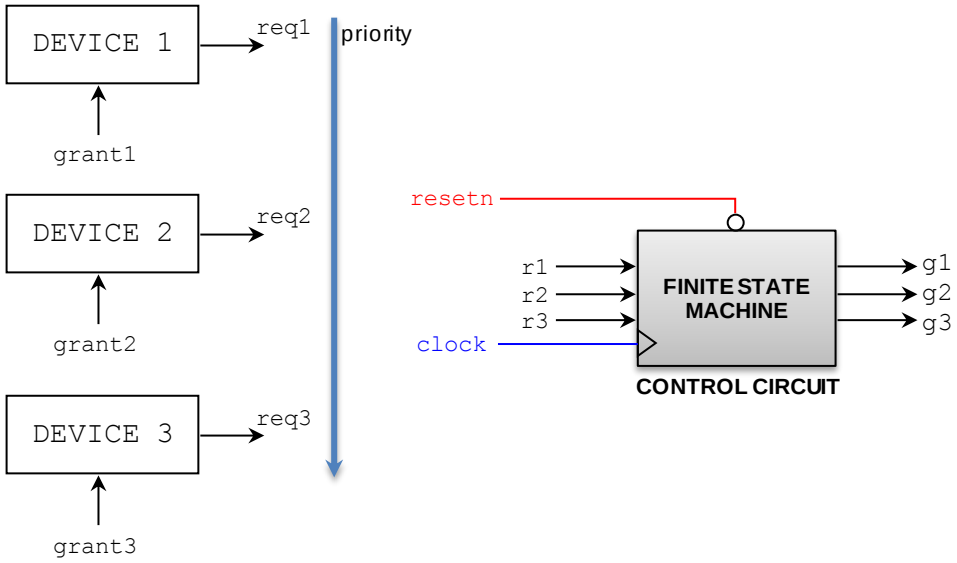




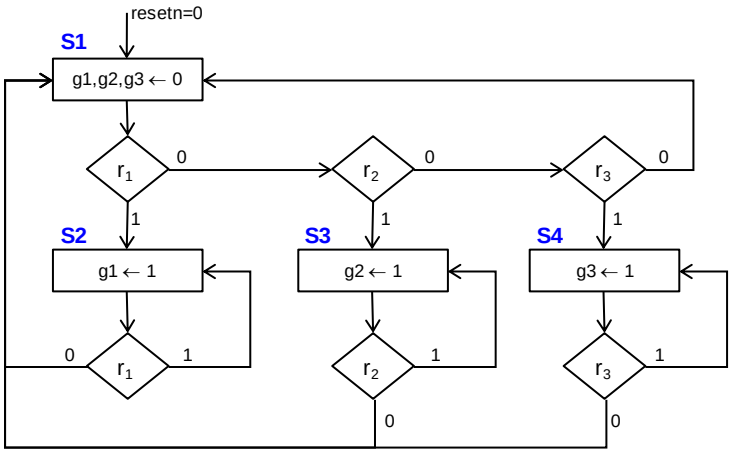


EXAMPLE: ARBITER CIRCUIT

- Three devices can request access to a certain resource at any time (example: access to a bus made of tri-state buffers, only one tri-state buffer can be enabled at a time). The FSM can only grant access to one device at a time.There should be a priority level among devices.
- If the FSM grants access to one device, one must wait until the request signal to that device is deasserted (i.e. set to zero) before granting access to a different device.

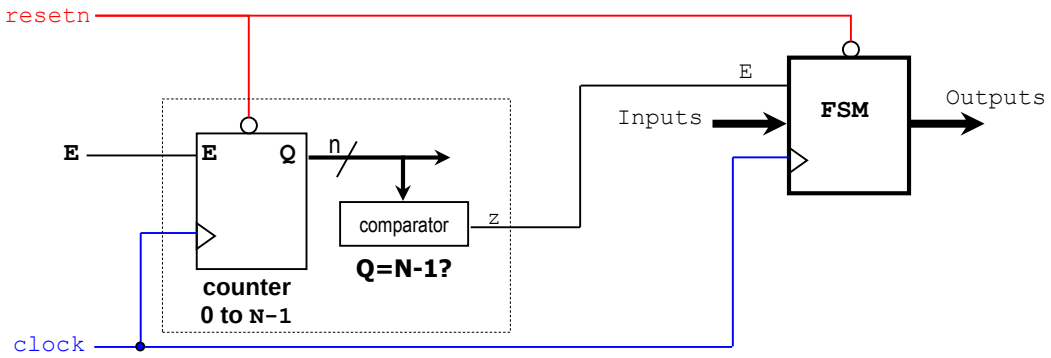


- **Algorithmic State Machine (ASM) chart:**

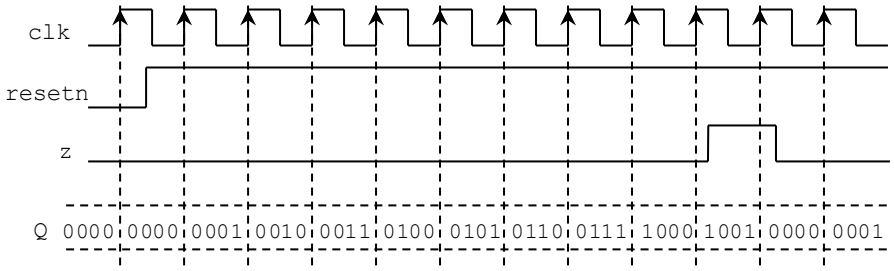


Reducing rate of change of a synchronous circuit:

- Here, we use FSM as an example of a synchronous circuit. But we can apply this technique to any synchronous circuit.
- We usually would like to reduce the rate at which FSM transitions occur. A straightforward option is to reduce the frequency of the input clock. But this is a very complicated problem when a high precision clock is required.
- Alternatively, we can reduce the rate at which FSM transitions occur by including an enable signal in our FSM: this means including an enable to every flip flop in the FSM. For any FSM transition to occur, the enable signal has to be '1'. Then we assert the enable signal only when we need it. The effect is the same as reducing the frequency of the input clock.
  - ✓ The figure below depicts a counter modulo-N (from 0 to N-1) connected to a comparator that generates a pulse (output signal 'z') of one clock period every time we hit the count 'N-1'. The number of bits the counter is given by $n = \lceil \log_2 N \rceil$. The effect is the same as reducing the frequency of the FSM to f/N, where f is the frequency of the clock.
  - ✓ A modulo-N counter is better designed using VHDL behavioral description, where the count is increased by 1 every clock cycle and 'z' is generated by comparing the count to 'N-1'. A modulo-N counter could be designed by the State Machine method, but this can be very cumbersome if N is a large number. For example, if N = 1000, we need 1000 states.



  - ✓ As an example, we provide the timing diagram of the counter from 0 to N-1, when N=10. Notice that 'z' is only activated when the count reaches "1001". This 'z' signal controls the enable of a state machine, so that the FSM transitions only occur every 10 clock cycles, thereby having the same effect as reducing the frequency by 10.



- We can apply the same technique not only to FSMs, but also to any sequential circuit. This way, we can reduce the rate of any sequential circuit (e.g. another counter) by including an enable signal of every flip flop in the circuit.

Flip flop timing parameters:

- Propagation Delay.
- Setup Time: Interval of time before the clock edge where the data must be held stable.
- Hold Time: Interval of time after the clock where the data must be held stable.
- If setup time or hold time are violated, the output may become unpredictable, or even worse it might enter into metastability.