Budapest University of Technology and Economics
Department of Electron Devices

Combinational and Sequential logic circuits

Digital Lab.3

Osama Ali

2024

osamaalisalman.khafajy@edu.bme.hu

Combinational and Sequential logic circuits

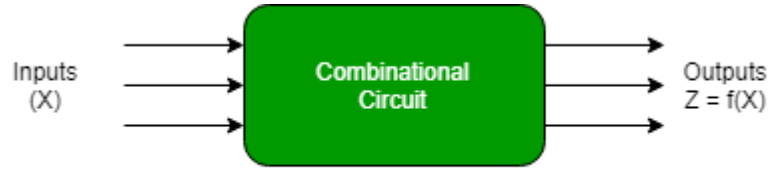


Figure: Combinational Circuits

- **Combinational circuits** are defined as the time independent circuits which do not depends upon previous inputs to generate any output are termed as combinational circuits
  1. Output depends only upon present input.
  2. Fast.
  3. easy to design.
  4. No feedback between input and output.
  5. Time independent.
  6. Elementary building blocks: Logic gates
  7. Used for **arithmetic** as well as **boolean operations**.
  8. Don't have capability to store any state.
  9. As combinational circuits don't have clock, they don't require triggering.
  10. These circuits do not have any memory element.
  11. It is easy to use and handle.

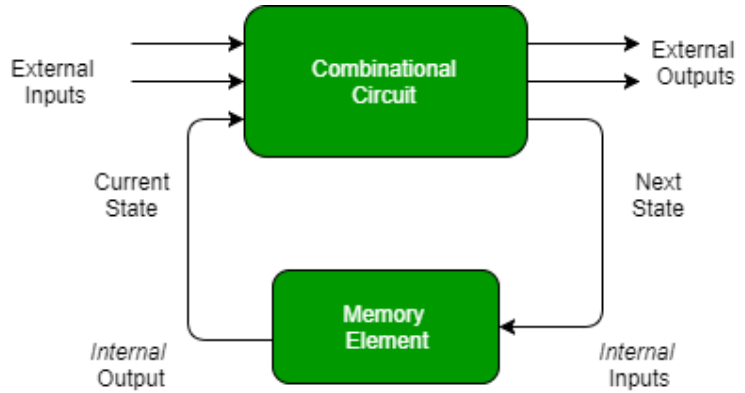
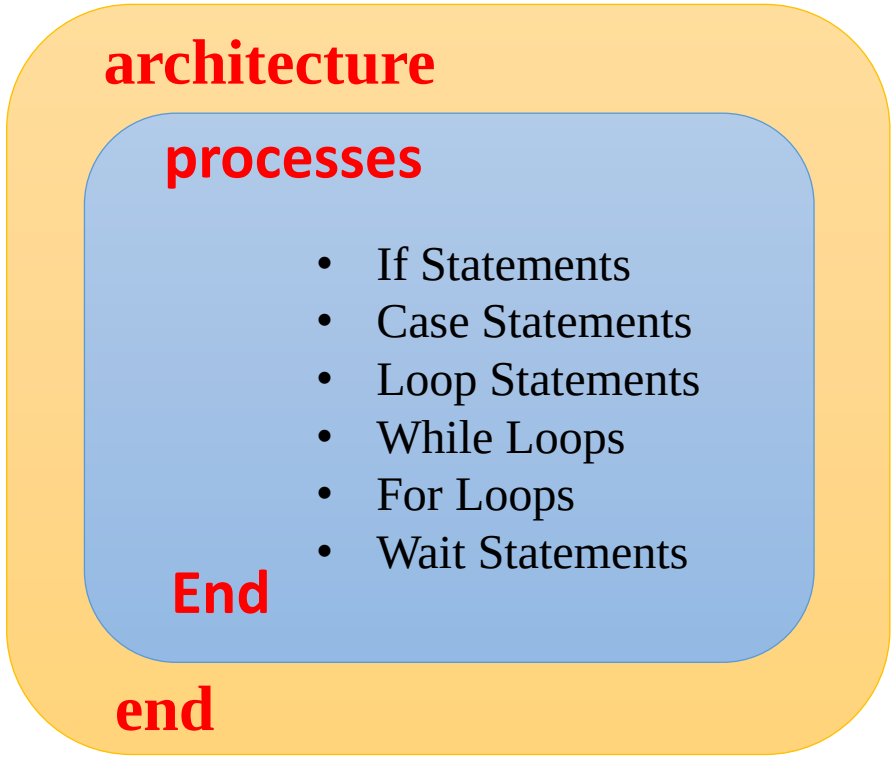


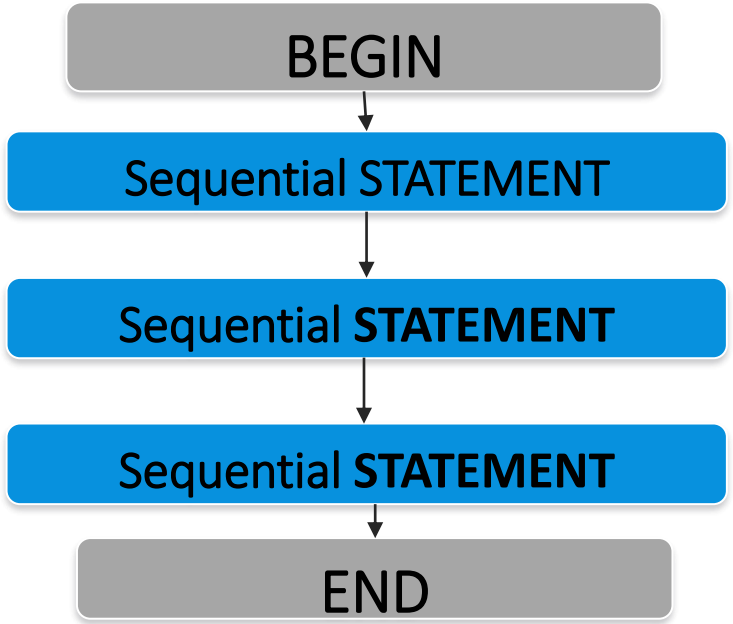
Figure: Sequential Circuit

- **Sequential circuits** are those which are dependent on clock cycles and depends on present as well as past inputs to generate any output
  1. Output depends upon present as well as past input.
  2. Slow.
  3. It is designed tough as compared to combinational circuits.
  4. Feedback path between input and output.
  5. Time dependent.
  6. Elementary building blocks: Flip-flops
  7. Mainly used for **storing data**.
  8. Have capability to store any state or to retain earlier state.
  9. As sequential circuits are clock dependent they need triggering.
  10. These circuits have memory element.
  11. It is not easy to use and handle.

Sequential Statements

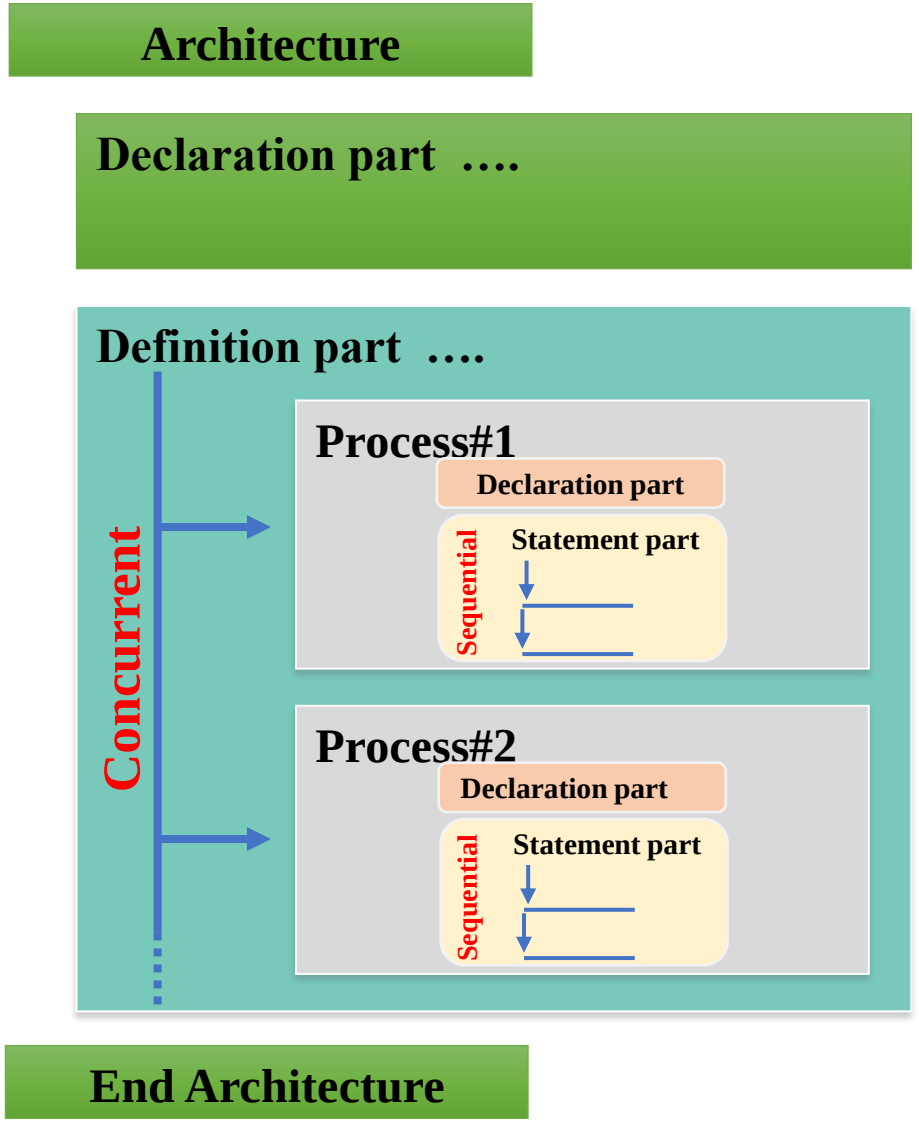
- In VHDL, sequential statements are used to describe the behavior of a digital system over time. Sequential statements are executed one after another, in a specific order, and the order of execution is determined by the control flow of the program





Properties of process Statement

- Process statement contains only sequential statements.
- The process statement is itself a concurrent statement.
- So all processes in an architecture behave concurrently.
- A process statement has a declaration section and a statement part.
- In the declaration section, types, variables, constants, subprograms, and so on can be declared.



Controlling the execution of Process (Sensitivity List & Wait)

- The process must have an explicit sensitivity list or a WAIT statement.
- The sensitivity list **should** contain all input signals used in that process.
- The statements inside the process statement to execute whenever **one or more** elements of the sensitivity list change value.

```
process (A,B)
begin
    if (A='1' or B='1') then
        Z <= '1';
    else
        Z <= '0';
    end if;
end process;
```

```
process
begin
    if (A='1' or B='1') then
        Z <= '1';
    else
        Z <= '0';
    end if;
    wait on A,B;
end process;
```

Signal Assignment **Vs. Variable** Assignment

Signal

```
Architecture beh_1 of test is
    signal A,B,C: integer;
    signal Y, Z : integer;
    signal M, N : integer;
begin
    process (A,B,C,M,N)
    begin
        M <= A;
        N <= B;
        Z <= M + N;
        M <= C;
        Y <= M + N;
    end process;
end Architecture;
```

Variable

```
Architecture beh_1 of test is
    signal A,B,C: integer;
    signal Y, Z : integer;
begin
    process (A,B,C)
        variable M, N: integer;
    begin
        M := A;
        N := B;
        Z <= M + N;
        M := C;
        Y <= M + N;
    end process;
end Architecture;
```

Update Signal

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

entity DelayedSignalAssignment is
    port (
        clk : in std_logic;
        A   : in std_logic;
        B   : in std_logic;
        C   : out std_logic
    );
end entity DelayedSignalAssignment;

architecture Behavioral of DelayedSignalAssignment is
begin
    process (clk)
    begin
        if rising_edge(clk) then
            -- Delayed signal assignment
            C <= A and B;  -- This assignment is scheduled
                           -- to take effect at the end of
                           -- the current simulation cycle
            -- Other operations...
        end if;
    end process;
end architecture Behavioral;
```

- In VHDL, signals are not updated instantly within a process statement because the language is designed to model hardware behavior rather than simulate instantaneous operations. VHDL simulates the behavior of digital circuits, which have inherent delays and dependencies.

- When a signal assignment statement is encountered within a process, the assigned value is scheduled to be updated at the end of the current simulation cycle, rather than immediately. This delay mimics the propagation delay of signals in actual hardware.

If Statement

Syntax:

```
if condition_1 then
  -- statements to execute when condition_1 is true
elsif condition_2 then
  -- statements to execute when condition_2 is true
elsif condition_3 then
  -- statements to execute when condition_3 is true
else
  -- statements to execute when all conditions are false
end if;
```

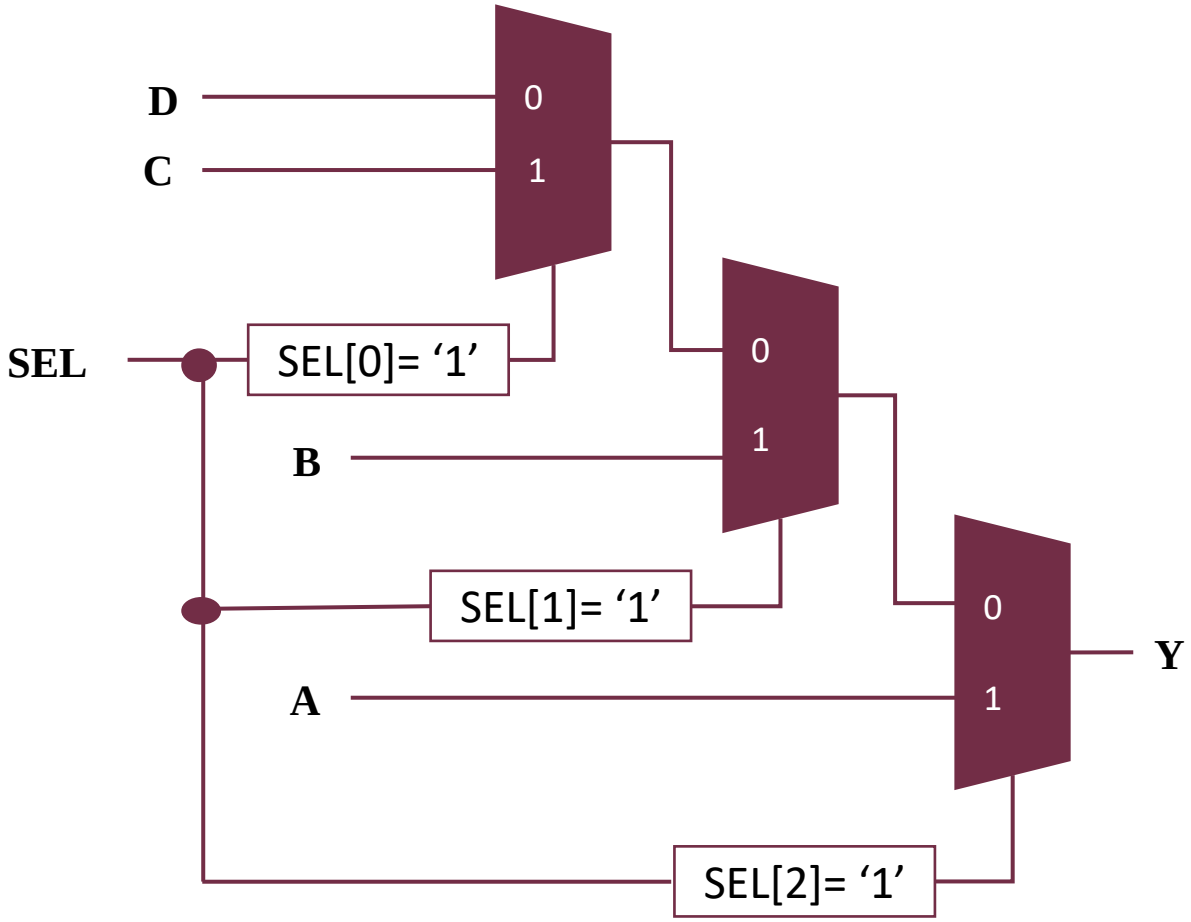
e.g.:

```
if a = '1' then
  b <= '0';
elsif c = '1' then
  b <= '1';
else
  b <= 'Z';
end if;
```

- "IF" statement evaluates each condition in order
- Statement can be nested
- Spaghetti code (Avoid using more than three levels of if…else statements)
- When defining the condition, use parentheses to differentiate levels of operations on the condition

Another example for If Statement

```
process (sel, a, b, c, d)
begin
if sel(2) = '1' then
    y <= a;
elsif sel(1) = '1' then
    y <= b;
elsif sel(0) = '1' then
    y <= c;
else
    y <= d;
end if;
end process
```



Generates a priority structure.
Corresponds to “when-else” command in the concurrent part.

Case Statement

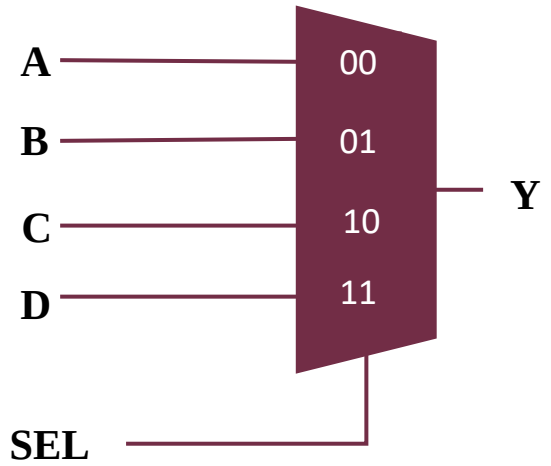
Syntax:

```
case expression is
  when value_1 =>
    -- statements to execute when expression equals value_1
  when value_2 =>
    -- statements to execute when expression equals value_2
  when value_3 =>
    -- statements to execute when expression equals value_3
  ...
  when others =>
    -- statements to execute when none of the above conditions are true
end case;
```

- **“Case” statement is a series of parallel checks to check a condition.**
- **Satements following each “when” clause is evaluated only if the choice value matches the expression value.**
- **Correspononds to “with…select” in concurrent statements**

e.g.:

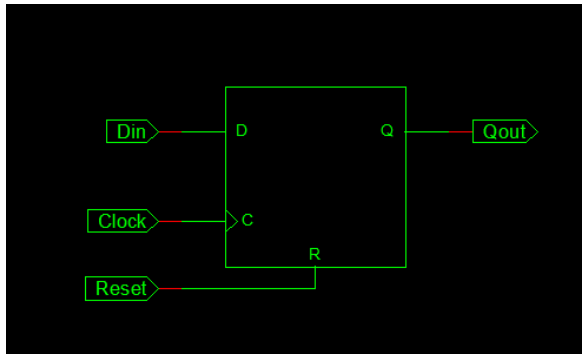
```
process (sel, a, b, c, d)
begin
case sel is
    when "00" =>   Y <=a;
    when "01" =>   Y <=b;
    when "10" =>   Y <=c;
  when others =>   Y <=d;
end case;
end process;
```



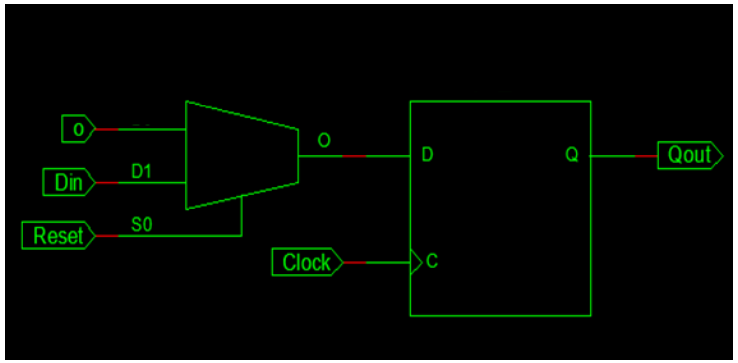
Synchronous VS. Asynchronous

- **Synchronous Reset: Flip-flops are reset on the active edge of the clock when reset is held active.**
- **<u>Asynchronous Reset</u>: Flip-flops are cleared as soon as reset is asserted.**

```
1  process(clk)
2  begin
3      if(clk' event and clk='1') then
4          if(rest='1') then
5              Q<='0';
6          else
7              Q<=D;
8          end if;
9      end if;
   end process;
```

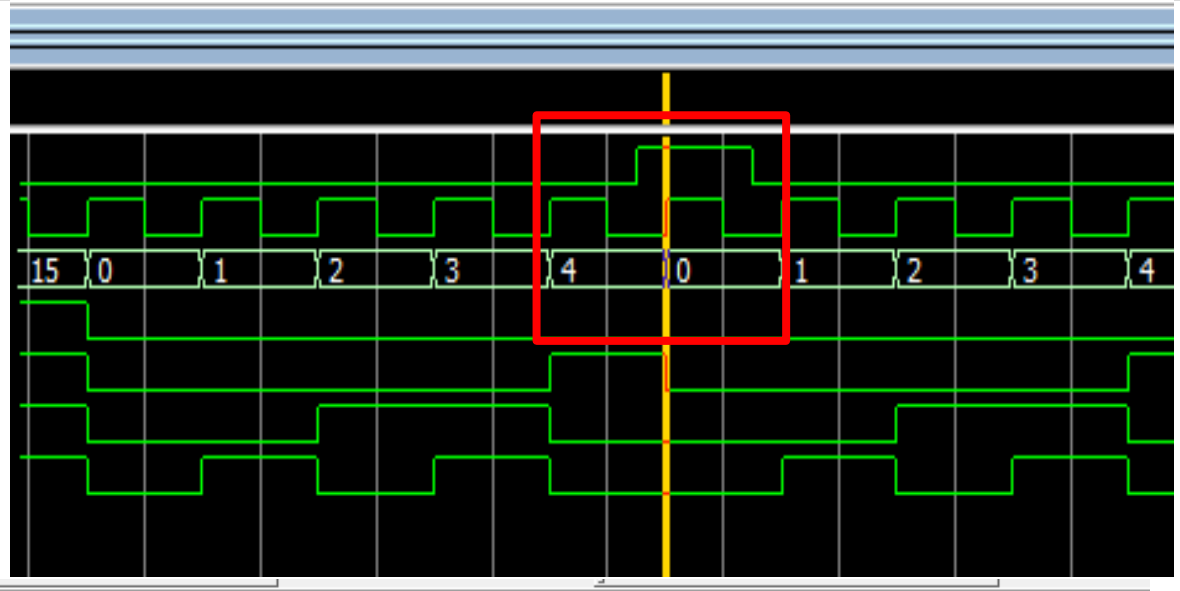



```
1  process(clk,rest)
2  begin
3      if(rest='1') then
4          Q<='0';
5      elsif(clk' event and clk='1') then
6          Q<=D;
7      end if;
8  end process;
```

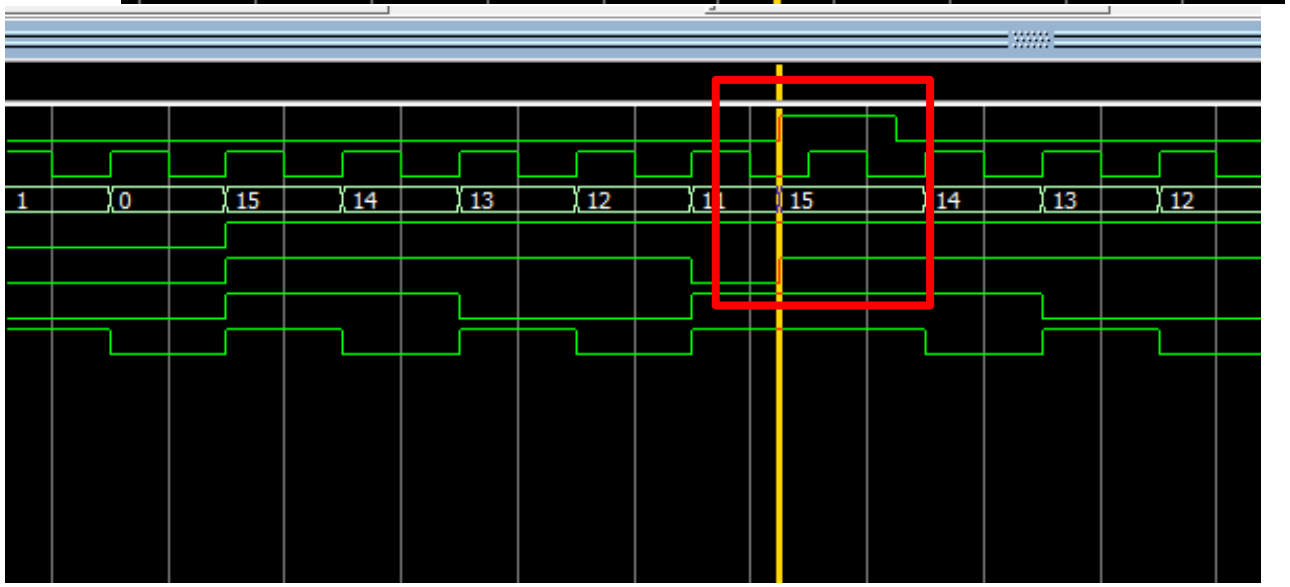


Synchronous VS. Asynchronous

Synchronous



Asynchronous

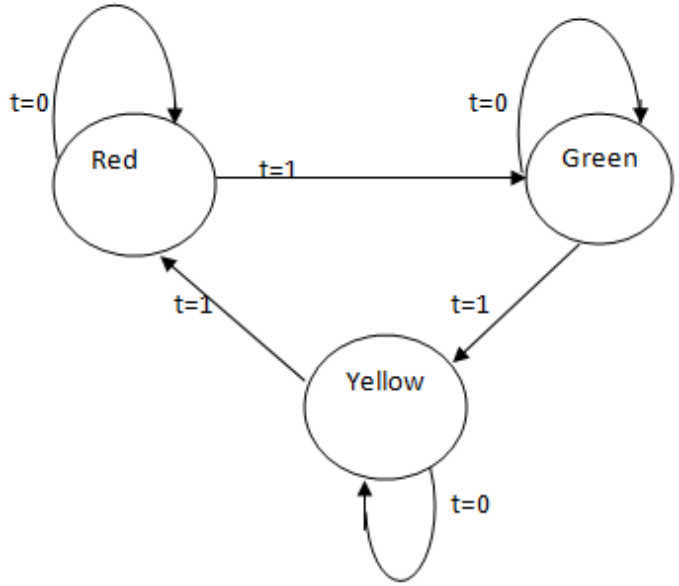


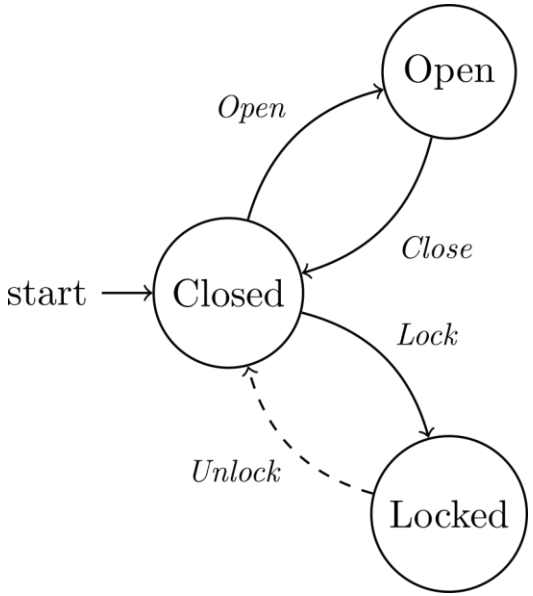
Clock generation in the TestBench

```
1   entity counter_tb is
2   end entity counter_tb;
3
4   architecture Behavioral of counter_tb is
5
6       -- def component counter
7
8       -- def signal
9
10  begin
11
12      -- counter port map
13
14      -- Clock generation
15      clk_tb<= not clk_tb after 10 ns;
16
17      -- Stimulus process
18      process
19      begin
20          rst_tb <= '1';  -- Assert reset
21          wait for 20 ns;
22          rst_tb <= '0';  -- De-assert reset
23          wait for 405 ns;
24          rst_tb <= '1';  -- Assert reset
25          wait for 20 ns;
26          rst_tb <= '0';  -- De-assert reset
27          wait;
28      end process stimulus_process;
29
30  end architecture Behavioral;
```

Finite State Machines (FSMs)

1. FSMs are essential digital models for systems with discrete states and transitions.
2. They're crucial for designing complex digital systems like traffic lights and vending machines, and protocol analyzers
3. Combinational logic circuits show how logic gates function based on current inputs.
4. Sequential logic circuits introduce memory elements for state-dependent behavior.
5. FSMs merge principles of combinational and sequential logic to model finite states and transitions.
6. Studying these circuits builds skills for analyzing and designing digital systems.
7. Understanding FSMs applies across engineering disciplines like computer science and electrical engineering.
8. Mastering these concepts enables designing efficient digital systems to meet specific requirements.





Finite State Machines (FSMs)