VHDL for FPGA Development Notes

Fundamentals of FPGA:

→FPGA - Field Programmable Gate Array

→ It can be reprogrammed multiple times

→ Gate array means collection of gate primitives

→ Modern FPGAs have gates, registers, configurable routing, memory, PLLs (for clock frequency scaling), Arithmetic blocks, DSP, Hard IP Blocks for hard peripherals.

→ FPGAs can be programmed via VHDL

→ General FPGA architecture involves LUT, MUX and D-Registers in a configurable logic block (ALM) and many such ALMs make up the FPGA

→LUT can bypass the register optionally

→ CLBs sit along PLLs, RAM and IO cells, IO cells connect IO pins and there are interconnects between all these blocks

→Why FPGA? if we require data acquisition system from high-speed ADCs, which may also require filtering, which can be used for Real TIme calculations such as running a FFT algorithm. Microcontrollers have limited amount of peripherals such as counters, timers, UARTs, SPI but an FPGA can have as many peripherals as you like, by creating your own peripheral modules like the UART peripheral, one can also make use of ready made IP blocks

→FPGAs are very fast as they process in parallel as opposed to a processor which does sequential execution, making it good for data intensive activities. We can set IO drive strengths for IO pins too. The low clock rate does not affect much

→The bandwidth is also configurable, as the data path bus can be varied unlike a processors whose data paths are fixed

→Avoid FPGAs when carrying out sequential tasks like look ups/searching/sorting algorithms and when speed and latency are not important

→HDLs are used to program the FPGA by describing the hardware individually

→Synthesisable code means code that can run on the hardware in terms of RTL code

→ RTL code describes hardware in terms of registers and combinational logic. There cannot be any delays given explicitly.

→ Behavioural code is only for simulations and can have delays and used to build verification mode.

→ Timing constraint have to be written too

→ How do we program it? we load the image into an external configuration flash memory via a microcontroller or via the JTAG programmer device into the JTAG port, then after a power cycling, it loads with the image.

Signals and Data Types :

→ VHDL is not case sensitive, has reserved keywords, involves identifiers to name objects but cannot be a reserved word

→ signal signal_name : data_type :=initial_value; or signal signal_name : data_type;

→ signal assignment operator : <=

→ Different data types is via the package - Std_Logic_1164, declare this like a header file

→ 1/0 or ’z’-tristated, ‘H/L’ - pull up/pull down values, ‘W’ - weak unknown, ‘U’ - uninitialized, ‘X‘- unknown, ‘-’ - don’t care

→ You can only tristate signals that directly drive the IO pins of the FPGA because only they have tristate buffers, whereas other internal signals don’t

→ H/L depends on how strong the signal is being driven

→ Example of declaration and assignment

signal A : std_logic;

signal B : std_logic := ‘0’;

A <= B;

→ an extension is std_logic_vector, which is an array of bits, it is also declared by the same package name, used for counters, multibit registers, address busses

→ Example

signal A : std_logic_vector(7 downto 0);

signal B : std_logic_vector(7 downto 0) := x”A1”;

So signal B is 1 0 1 0 0 0 0 1, we can do sliced assignment as follows

A(7 downto 6) < = B(3 downto 2);

→array attributes for signal A are

A’left - MSB

A’right - LSB

A’length - length of array

A’range - length of downto

So during assignment, we can do B < = A(A’right); as well as B < = A(A’right-2) which means A(7) and A(5) respectively.

→package - numeric_std is for signed and unsigned operations for arithmetic operations, used for counters also

→Example declaration

signal B : signed(7 downto 0) := x”A1”;

and during assignments both sides must be same type like signed signed or unsigned unsigned and of same size as well

→datatype - integer, for 32-bits, no need of a package

→Example declaration

signal A : integer range -128 to 127 := 35; (best way to initialize if at all)

→signal A : boolean := true;

→Enumerated types

type TrafficLight is (red,amber,green);

signal my_signal : TrafficLight;

my_signal < = red;

very similar to struct or enums from C/C++

it will take two FF to encode 3 states as 00,01,10,11, where 11 will go unused

→ type myarraytype is array (0 to 3) of integer;

myarraytype(0) < = 12;

this above syntax makes an array of size 4 for integers along with a sample assignment

→ type ABC is array (integer range <>) of integer;

signal Sig1: ABC(0 to 7);

signal Sig2: ABC(1 to 16);

this above syntax makes an unconstrained array of unknown size

→ We can also make our own subtypes

subtype uint8_t is INTEGER range 0 to 255;

subtype MyVector8 is std_logic_vector(7 downto 0);

signal RxChar : uint8_t;

signal DataBus : MyVector8;

→Special assignments and declarations on arrays are

type _ is array (NATURAL range <>) of _;

type string is array (NATURAL range <>) of character;

where _ is character, boolean, bit, integer and first placeholder is a name given,

signal mychararray : string(1 to 10);

→ Some special enum style arrays

type TLightType is (RED,AMBER,GREEN);

type TLightType is array (NATURAL range<>) of TLightType;

signal MyTrafficLight : TLightArray(1 to 10);

MyTrafficLight(1) < = red;

→ Consider a RAM, where we have 5 signals

signal WriteData : std_logic_vector(15 downto 0);

signal ReadData : std_logic_vector(15 downto 0);

signal WriteAddr : std_logic_vector(7 downto 0);

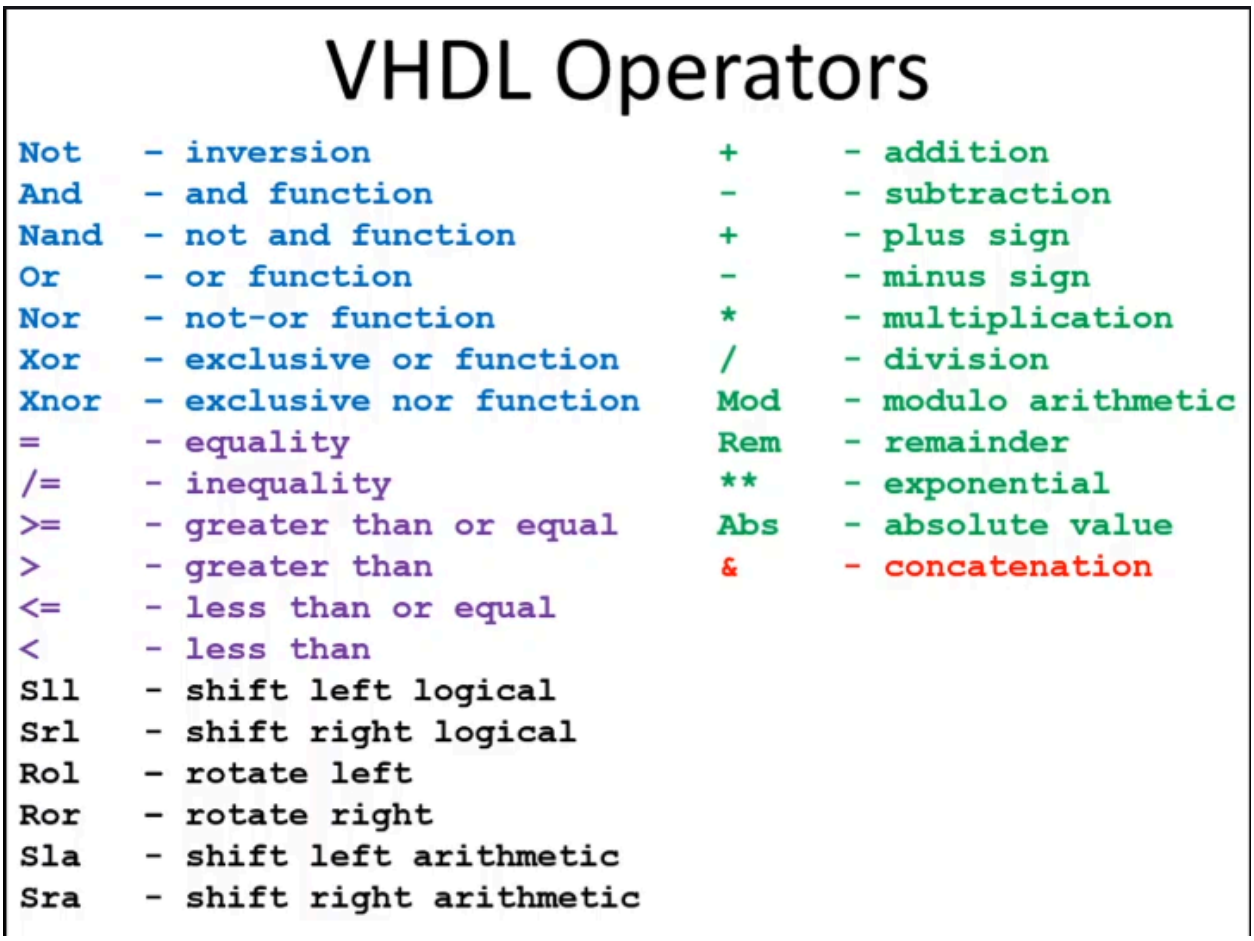
signal ReadAddr : std_logic_vector(7 downto 0);

signal WriteEnable : std_logic;

this can be grouped as a record,

```
type MyRAMType is record
    signal WriteData : std_logic_vector(15 downto 0);
    signal ReadData : std_logic_vector(15 downto 0);
    signal WriteAddr : std_logic_vector(7 downto 0);
    signal ReadAddr : std_logic_vector(7 downto 0);
    signal WriteEnable : std_logic;
end record;
signal RAM_IO : MyRAMType;
RAM_IO.WriteData < = x”8000”;
```

VHDL Operators :



Blue - boolean operators like not,and,nand,... done via std_logic_1164, can be bit, boolean, array, example → C < = A and B; (A,B,C can be a vector too)

if sig1 = “1101” or sig2 = true then

—code to execute;

end if;

Purple - comparison operators, done via std_logic_1164, returns a boolean type value after comparison

Black - shift operators, done via std_logic_1164, fills 0 for new spaces, Y < = X sll 1;

Y < = X rol 1;

Green - Arithmetic operators, done via numeric_std, C < = A + B; , check for overflows!

Red - Concatenation operator, done via std_logic_1164, its to combine two busses

signal A : std_logic_vector(3 down to 0) : = “0011”;

signal B : std_logic_vector(3 down to 0) : = “1111”;

signal C : std_logic_vector(4 down to 0);

signal D : std_logic_vector(7 down to 0);

C < = A & ‘0’; c = 00110

C < = ‘1’ & A; c = 10011

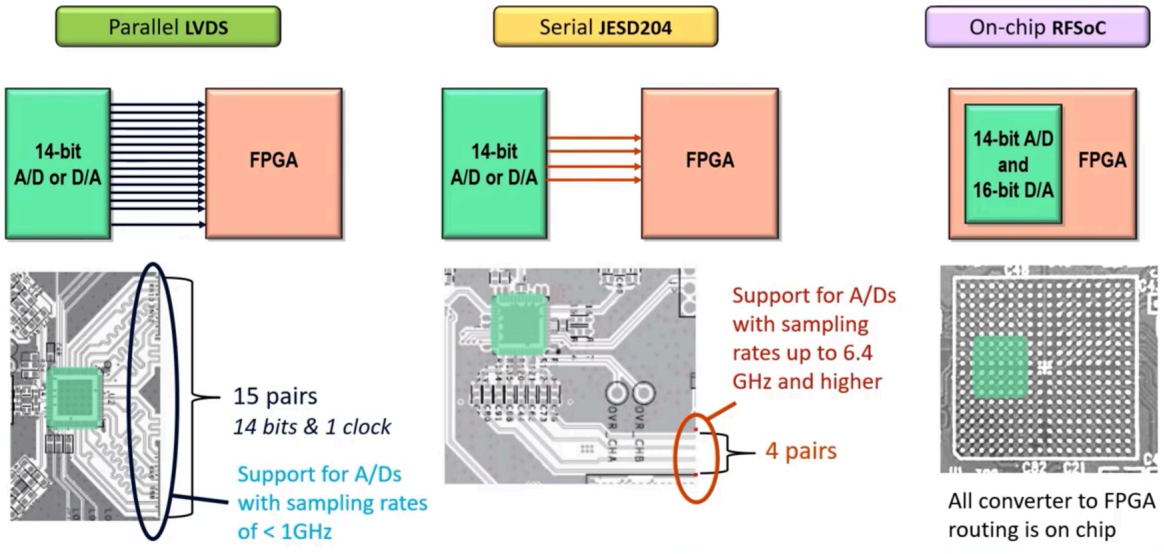
D < = A & B; D = 00111111

ensure sizes are matched!

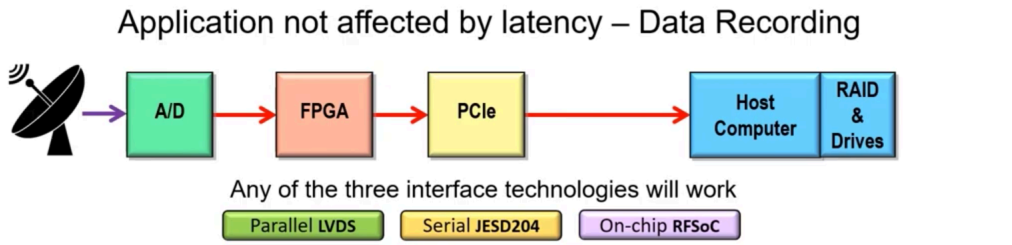
Operator Precedence

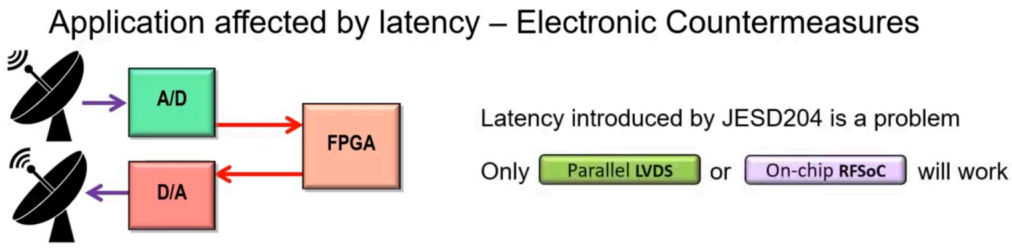
Operator
Abs, not, **
Mod, rem, *, /
+, - (signs)
+, - , &
Sll, srl, rol, ror
And, or, nand, nor, xor, xnor

PCB Routing Comparison



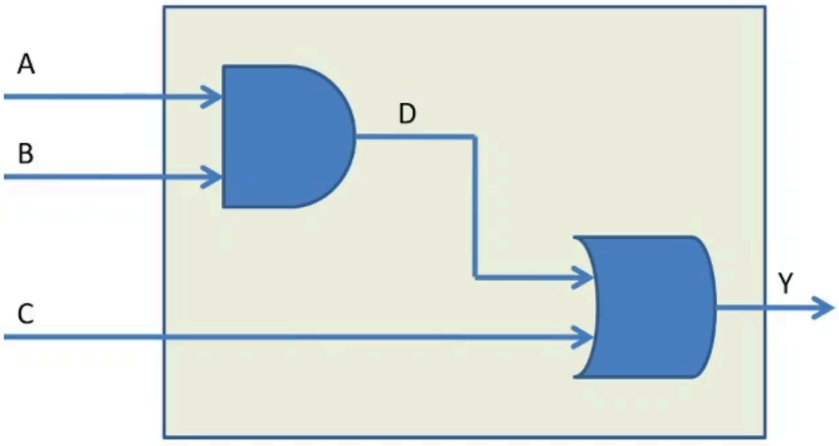
Applications and Latency



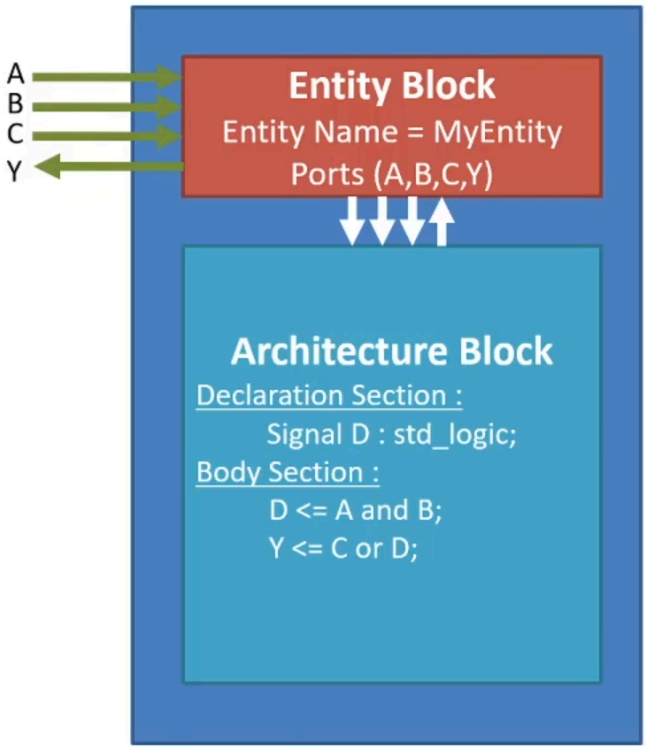


VHDL File Structures

VHDL File Structure



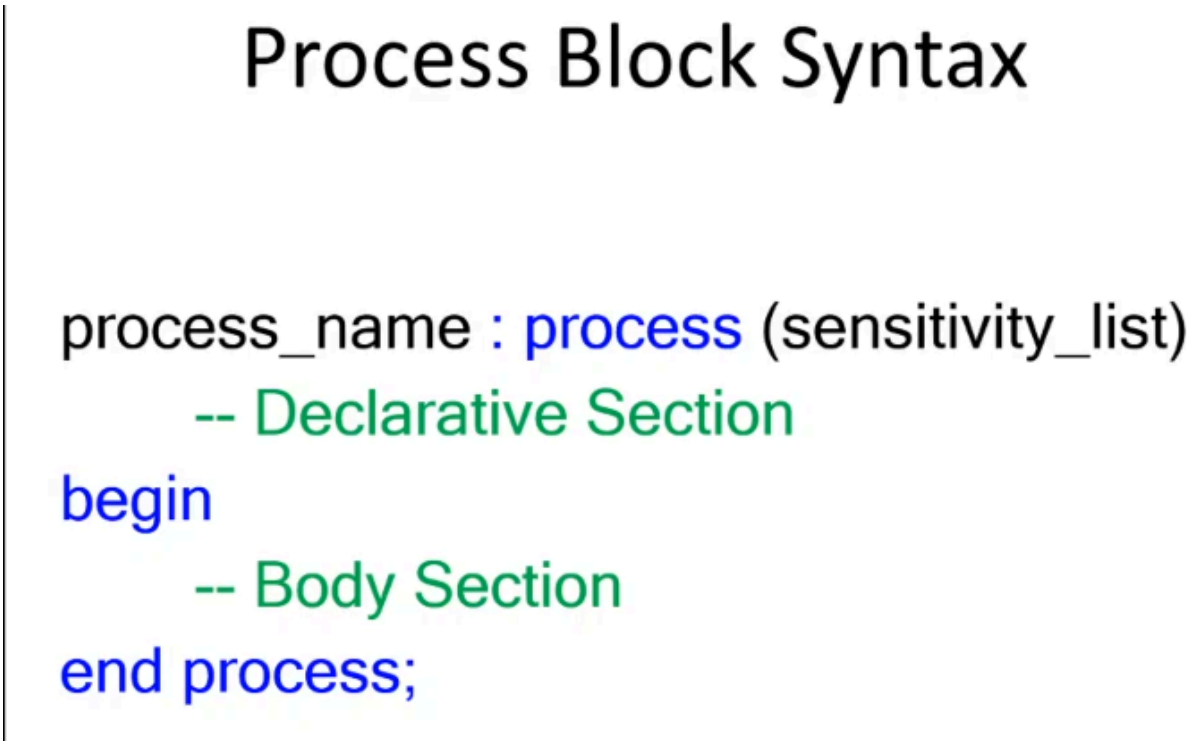
VHDL File Structure



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

entity MyEntity is
    port (
          A : in std_logic;
               B : in std_logic;
               C : in std_logic;
               Y : out std_logic
    );
end entity;

architecture MyArchitecture of MyEntity is
        signal D  : std_logic;
begin
        D <= A and B;
        Y <= C or D;
end architecture;
```

Process blocks :

→ It’s written inside the architecture body and can have multiple of these



→Executes only iff any one signal changes in the sensitivity list, but it runs once at time T=0, but we want to avoid this, so we force a non empty sensitivity list

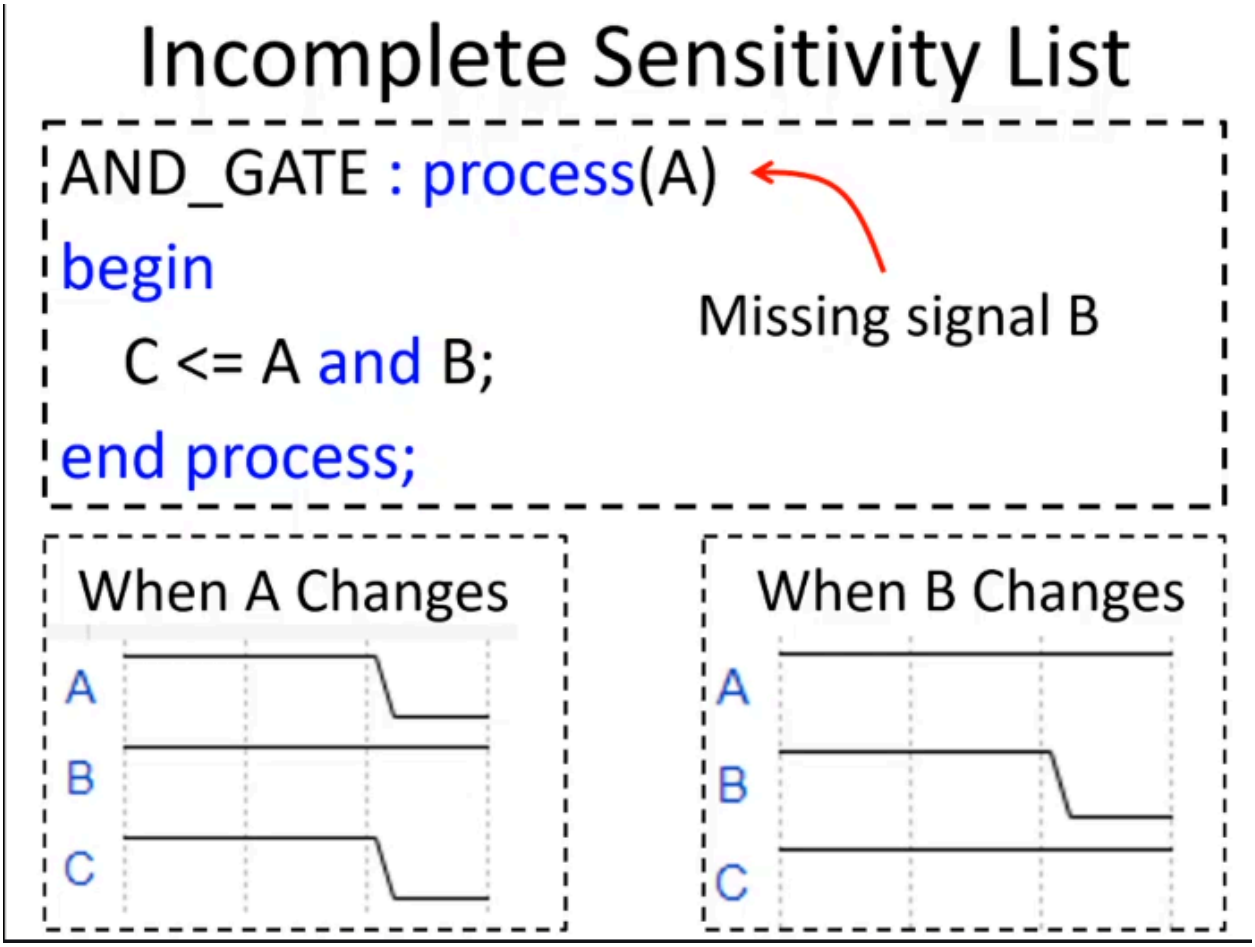
A whole VHDL code would look like

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

    entity switchdrivesled is
    port
    (
    sw1 : in std_logic;
    led1 : out std_logic
    );
    end entity;

        architecture rtl of switchdrivesled is

            *some process would be inserted here*
            begin
```

```
led1 <= sw1;
end rtl;
```

Process - Example

```
AND_GATE : process(A, B)
begin
   C <= A and B;
end process;
```



→Registered process is for flip flops (always) and must have clock signal and optionally a reset, and clock is edge trigger default

→ Btw all these still lie in the std_logic_1164 package

→ if else elseif all these can be written only inside the always block and not outside

→Registered process only has clock and reset in the sensitivity list

→Empty sensitivity lists require wait statements

→ Do not write to the same signal in two different process but read is allowed

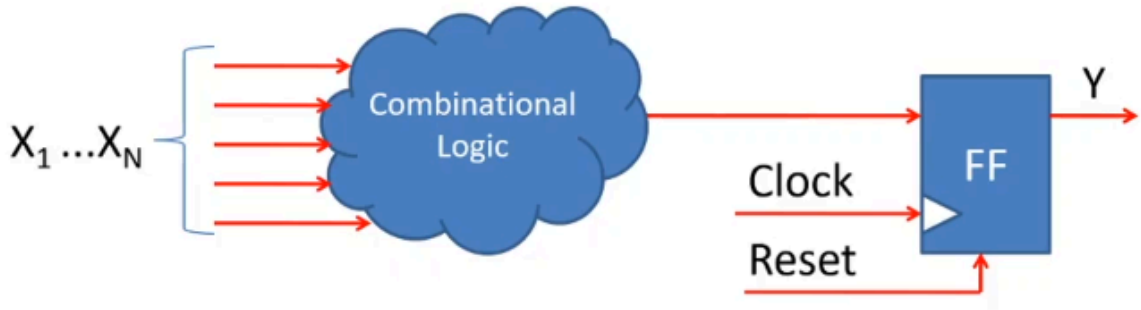
example for D register

TestProcess : process(Rst, Clk)

```
begin
    if Rst = '1' then
        Q < = '0';
    elseif rising_edge(Clk) then
        Q < = D;
    end if
end process;
```

More on Registered Processes

```
TestProcessGeneral : process(Reset, Clock)
begin
    if Reset = '1' then
        Y <= '0';
elsif rising_edge(Clock) then
        Y <= Function(X1, X2, X3,... XN);
    end if;
end process;
```

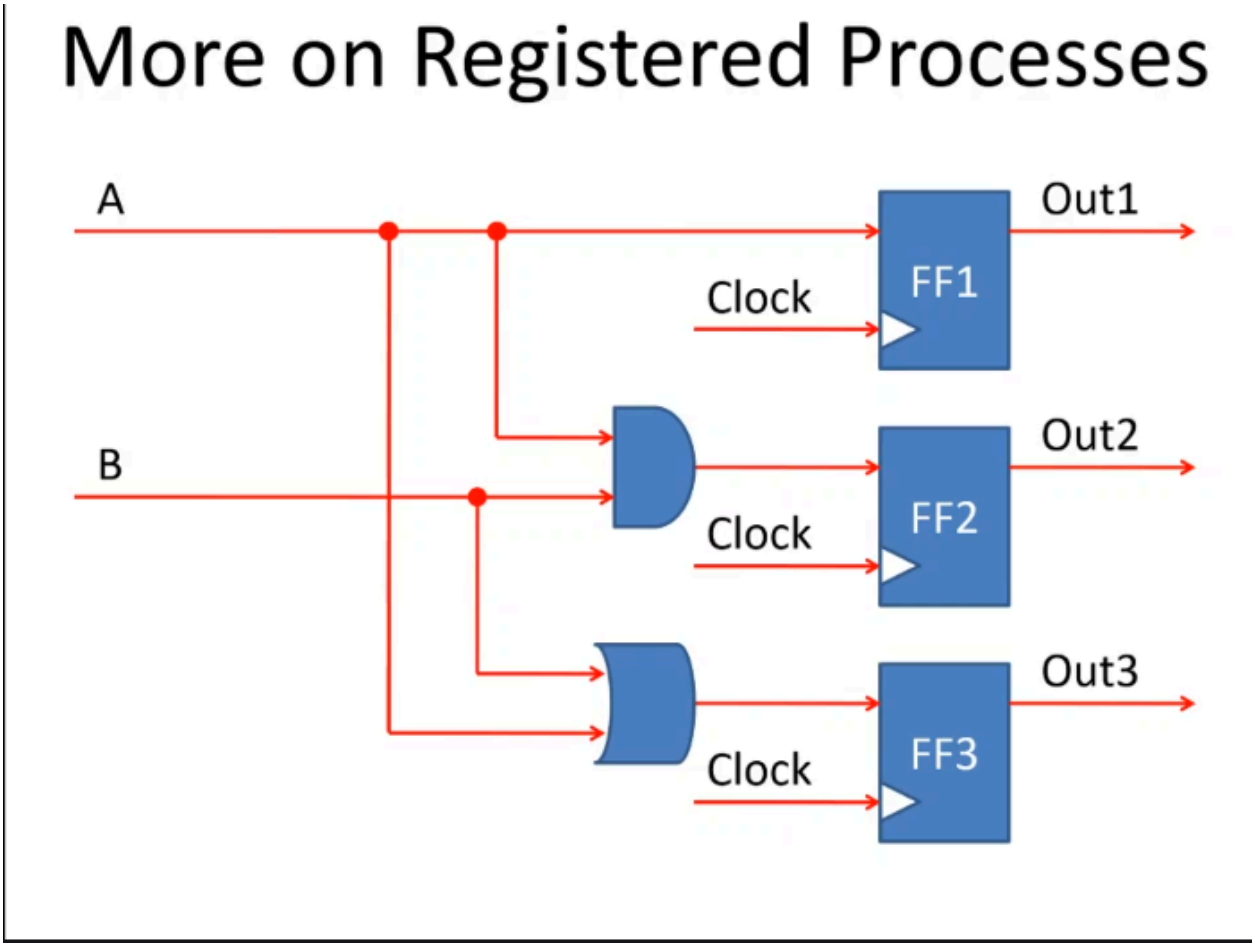


More on Registered Processes

- Signals driven by a registered process are always outputs of flip flops.

```
TestProcess2 : process(Reset, Clock)
begin
    if Reset = '1' then
       Out1 <= '0';
       Out2 <= '0';
       Out3 <= '0';
   elsif rising_edge(Clock) then
       Out1 <= A;
       Out2 <= A and B;
       Out3 <= A or B;
    end if;
end process;
```

Clocked Section
A signal assigned here will create a flip flop.



Entity blocks define the I/O port class. But how do you put an entity inside another larger entity, you do it by creating an instance, which is also called a component of the smaller entity. You can multiple copies of the small entity.

VHDL File Template

```
-- Declare your VHDL libraries & packages up here.
entity EntityName is
    port
    (
        -- Declare your modules's IO ports here!
    );
end entity;

architecture ArchitectureName of EntityName is
    -- Declare your internal signals & constants here
begin
    -- Define your module's behaviour here!
end architecture;
```

consider this adder which is a smaller entity

VHDL Entity A

```
entity EntityA is
    port
    (
        A : in  integer range 0 to 255;
        B : in  integer range 0 to 255;
        C : out integer range 0 to 511
    );
end entity;

architecture rtl of EntityA is
begin
        C <= A + B; -- Perform the addition operation.
end architecture;
```

```
entity TopLevelEntity is
  port
  (
    X : in  integer range 0 to 255;
    Y : out integer range 0 to 255
  );
end entity;

architecture rtl of TopLevelEntity is

      signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
      signal Sig3, Sig6 : integer range 0 to 511;

       component EntityA is
       port
       (
            A : in  integer range 0 to 255;
            B : in  integer range 0 to 255;
            C : out integer range 0 to 511
       );
       end component ;

begin
       Instance1 : EntityA
       port map
       (
            A => Sig1,
            B => Sig2,
            C => Sig3
       );

       Instance2 : EntityA
       port map
       (
            A => Sig4,
            B => Sig5,
            C => Sig6
       );

end architecture;
```

what is phase of a non periodic signal? not defined but phase is **the position of a wave at a point in time (instant) on a waveform cycle**

Concurrent VHDL statements:

Concurrent VHDL Statements

- Process Block
- Component Instantiation
- Concurrent Signal Assignments
- Conditional Signal Assignment (When – Else)
- Selected Signal Assignment (With – Select)
- Generate

When-Else

```
architecture RTL of MyEntity is

    A <= "1000" when B = "00" else
         "0100" when B = "01" else
         "0010" when B = "10" else
         "0001";

end architecture;
```

use when else and with select only in arch and not in process

---

With-Select

```
architecture RTL of MyEntity is

        with B select
    A <= "1000"  when  "00",
         "0100"  when  "01",
         "0010" when  "10",
         "0001" when others;

end architecture;
```

generate is to duplicate components

→you generate using for or if

For Generate Syntax

```
Label: for parameter in range generate
        -- Hardware to Generate
end generate;
```

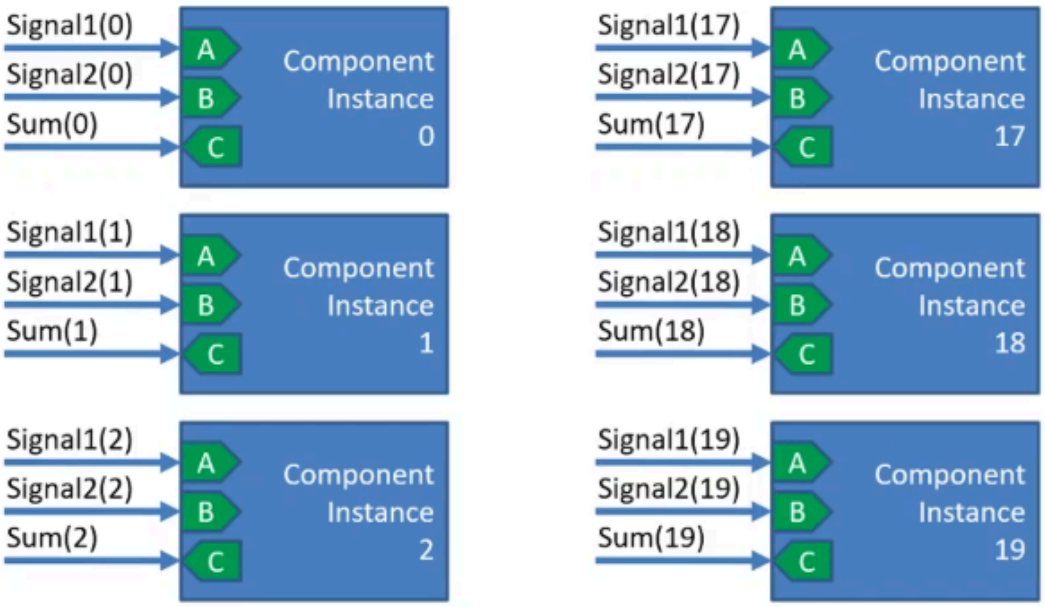
For Generate Example 1

```
Signal InputSignal   : std_logic_vector(15 downto 0);
Signal EvenBits      : std_logic_vector(7 downto 0);
Signal OddBits       : std_logic_vector(7 downto 0);

Example1: for  count  in 0 to 7 generate
        EvenBits (count) <= InputSignal(2*count);
        OddBits (count) <= InputSignal(2*count+1);
end generate;
```

For Generate Example 2

```
Example2: for n in 0 to 19 generate
   MyComponent : EntityA
   port map
   (
        A => Signal1(n),
        B => Signal2(n),
        C => Sum(n)
   );
end generate;
```

For Generate Example 2



If Generate Syntax

```
Label : if condition1 generate
-- Hardware to generate on condition 1.
elsif condition2 generate
-- Hardware to generate on condition 2.
else generate
-- Hardware to generate if all conditions false.
end generate;
```

If Generate Example

```
Example  :  if   ASSIGN_TO_Y  generate
        Y <= X;
else generate
        Z <= X;
end generate;
```

If Generate Example

```
Example : if ENABLE_REGISTER generate
    Reg : process (clk)
    begin
        if rising_edge(clk) then
            Y <= X;
        end if;
    end process;
else generate
    Y <= X;
end generate;
```

This section of code will be implemented if ENABLE_REGISTER is TRUE.

This section of code will be implemented if ENABLE_REGISTER is false.

Sequential statements :

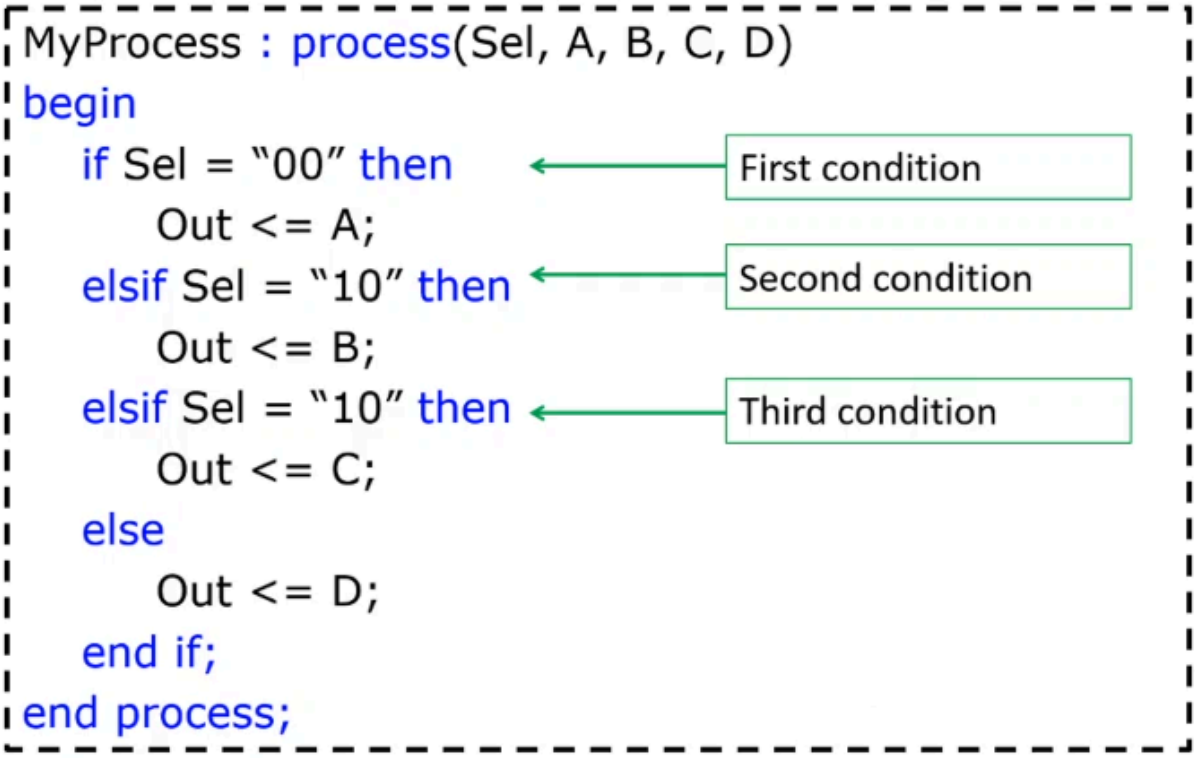
Sequential Statements

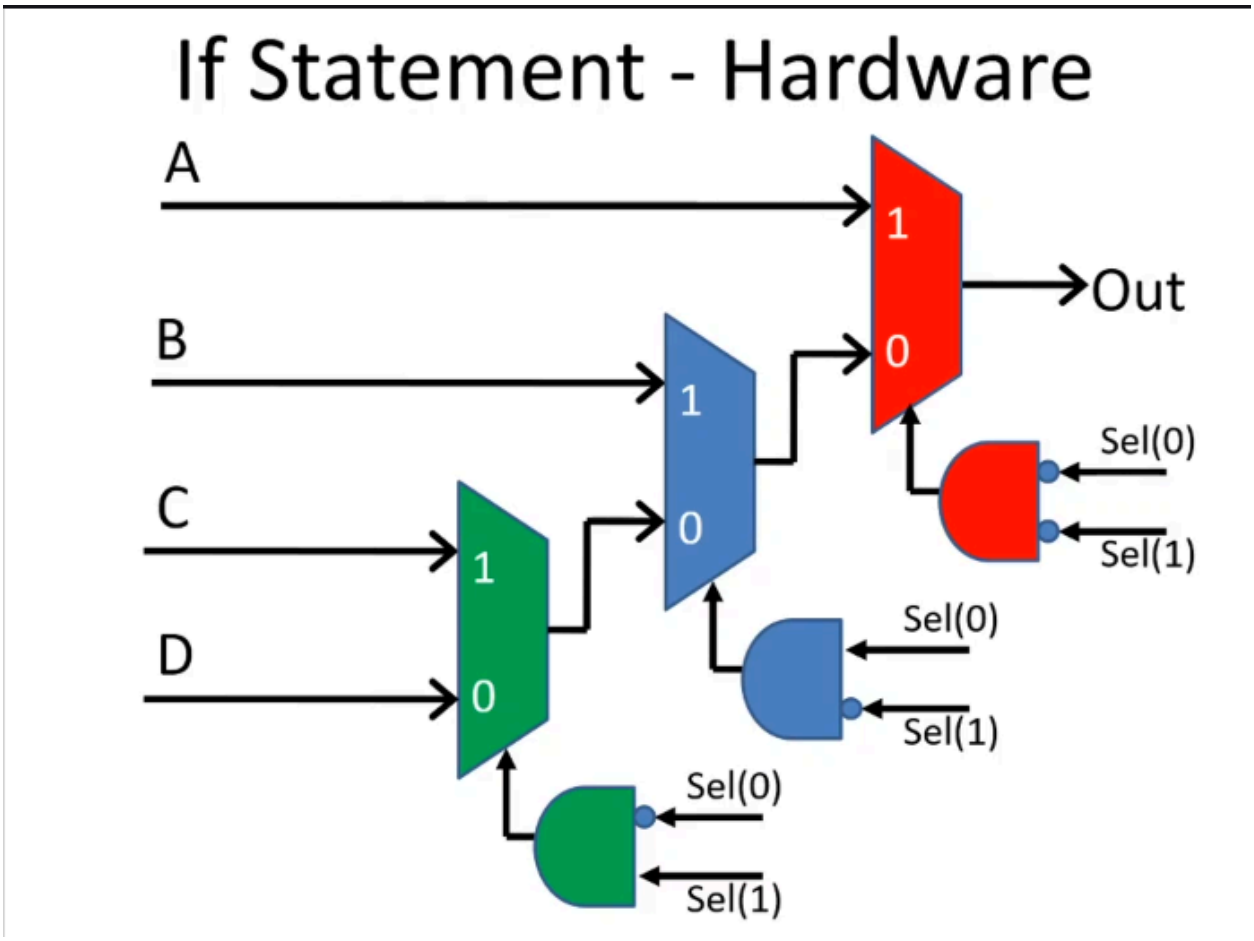
- Can only be used inside **Process Blocks**.
- Sequential Assignment
- If statements
- Case statements
- For loops
- Wait

If Statement - Template

```
if Condition1 then
        -- Code Section 1
elsif Condition2 then
        -- Code Section 2
elsif Condition3 then
        -- Code Section 3
else
        -- Code Section 4
end if;
```

If Statement - Combinational




If Statement - Hardware
A
B
C
D
1
0
1
0
1
0
Out
Sel(0)
Sel(1)
Sel(0)
Sel(1)
Sel(0)
Sel(1)

Case Statement - Template

```
case (Expression) is
    when Choice1=>
        -- Code Section 1
    when Choice2=>
        -- Code Section 2
    when others =>
        -- Code Section 3
end case;
```

Case Statement – Combinational Process

```
MyProcess : process(Sel, A, B, C, D)
begin
    case (Sel) is
        when "00" => Out <= A;
        when "01" => Out <= B;
        when "10" => Out <= C;
        when others => Out <= D;
    end case;
end process;
```

For Loop - Template

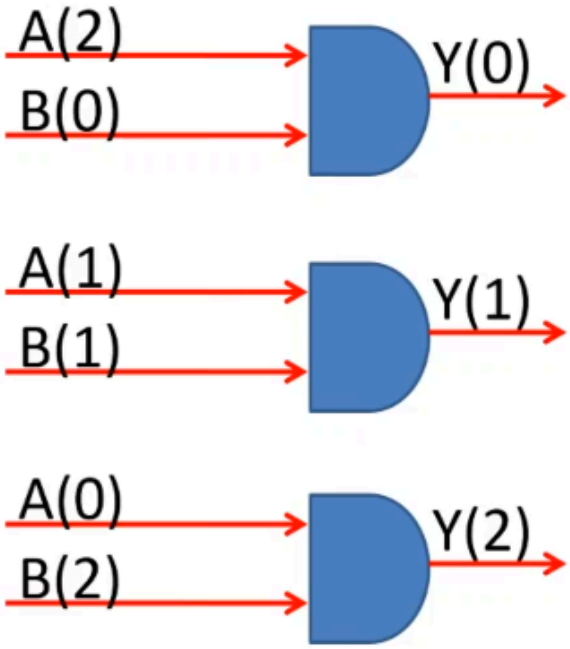
```
for LoopVariableName in Range loop
  -- Code Section That Needs Repeating
end loop;
```

For Loop – Combinational Process

```
MyProcess : process(A, B)
Begin
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
end process;
```

```
MyProcess : process(A, B)
Begin
        Y(0) <= A(2) and B(0);
        Y(1) <= A(1) and B(1);
        Y(2) <= A(0) and B(2);
end process;
```

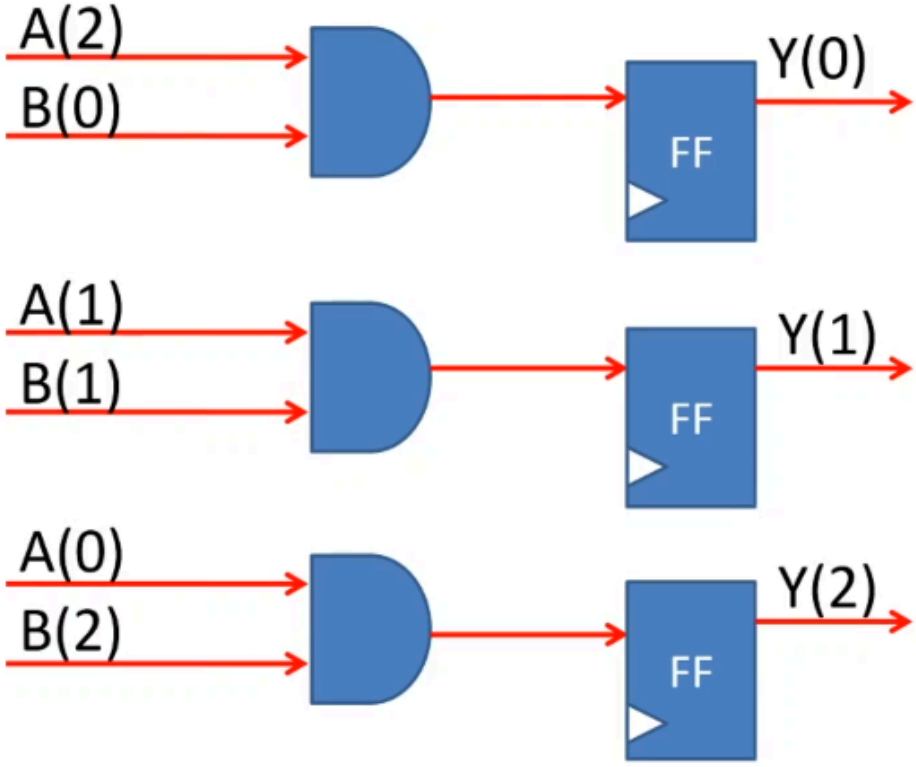
For Loop – Combinational Process



For Loop – Registered Process

```
MyProcess : process(Reset, Clock)
Begin
   if Reset = '1' then
       Y <= "000";
   elsif rising_edge(Clock) then
      for i in 0 to 2 loop
            Y(i) <= A(2-i) and B(i);
      end loop;
   end if;
end process;
```

For Loop – Registered Process



For Loop Notes

```
-- A is a std_logic_vector of length 8
for i in 0 to 7 loop
    A(i) <= '1';
    if i = 3 then
        exit;
    end if;
end loop;
```

Can use "exit" to immediately exit the loop.

```
for i in 0 to 7 loop
    A(i) <= '1';
    exit when i = 3;
end loop;
```

Wait Statement

- NOT Synthesisable!
- Used only in Test-Benches (or Models)
- Used to pause a simulation or model.
- Process must not have a sensitivity list
- Allows us to pause the simulation :
  - for a fixed period of time
  - until some event occurs

Wait

```
Test : process
begin
        Y <= X;
        wait;
        Z <= Y;
end process;
```

Wait for

```
Test : process
begin
        Y <= X;
        wait for 100ns;
        Z <= Y;
end process;
```

Clock Signal Using Wait for

```
Clk_Generator: process
begin
        Clock <= '0';
        wait for 10 ns;
        Clock <= '1';
        wait for 10 ns;
end process;
```

Wait On

```
Test : process
begin
        Y <= X;
        wait on A;
        Z <= Y;
end process;
```

Wait Until

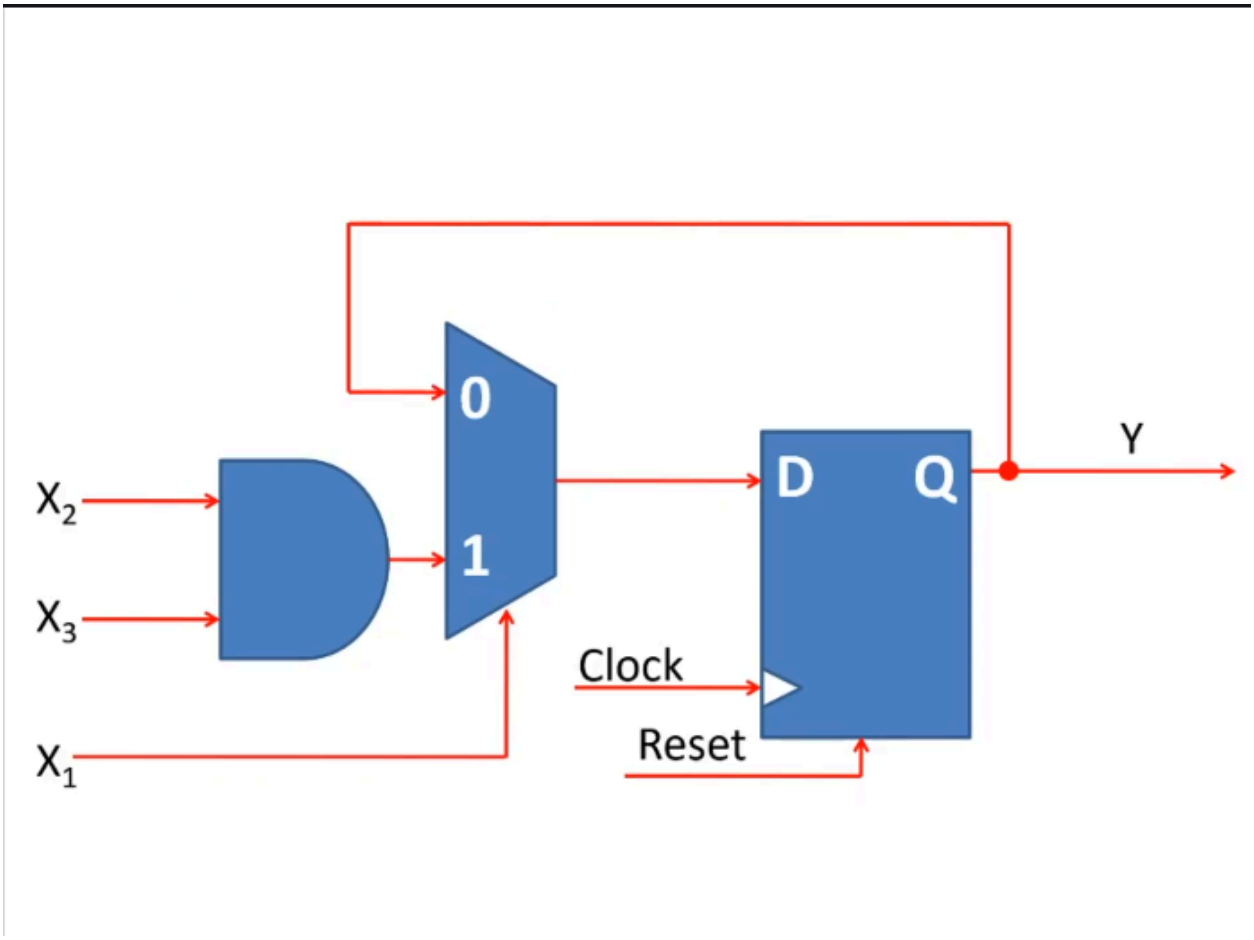
```
Test : process
begin
        Y <= X;
        wait until A = '0';
        Z <= Y;
end process;
```

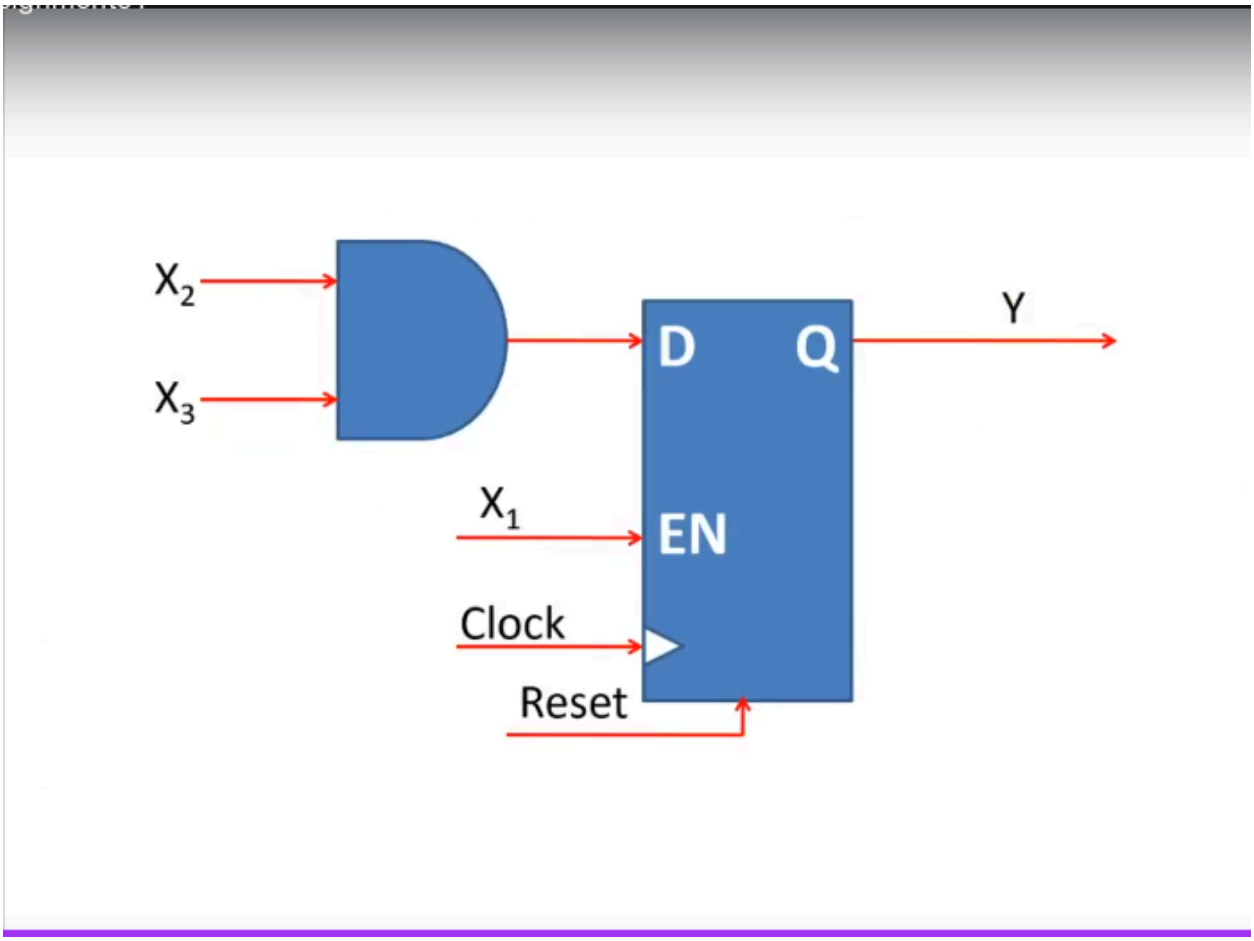
Wait Until

```
Test : process
begin
        Y <= X;
        wait until rising_edge(A);
        Z <= Y;
end process;
```

if you try assigning different logic to same output it will pick the last signal assignment and ignore the rest.

```
TestProcess : process (Reset, Clock)
Begin
        if Reset = '1' then
            Y <= '0';
        elsif rising_edge(Clock)  then
             if X1 = '1' then
                  Y <= X2 and X3;
             end if;
        end if;
end process;
```


0
1
D
Q
Y
X_2
X_3
X_1
Clock
Reset



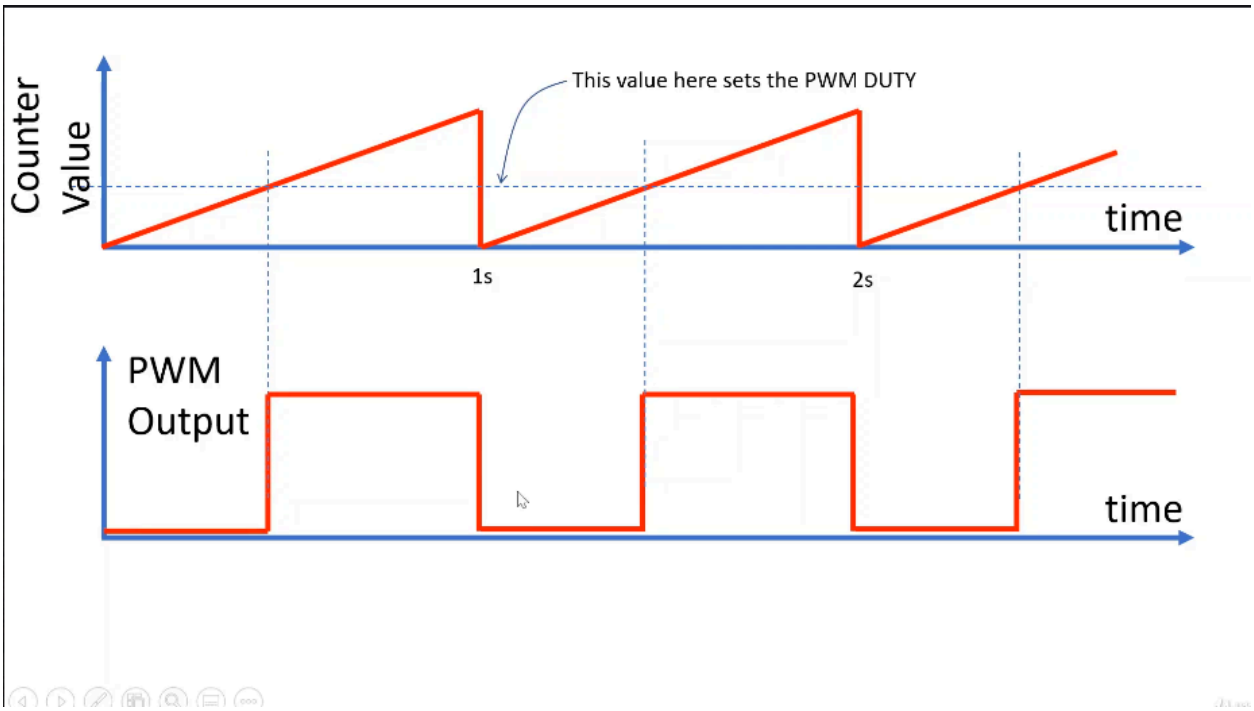
We can implement PWM using counters,

f=50mhz, t=20ns

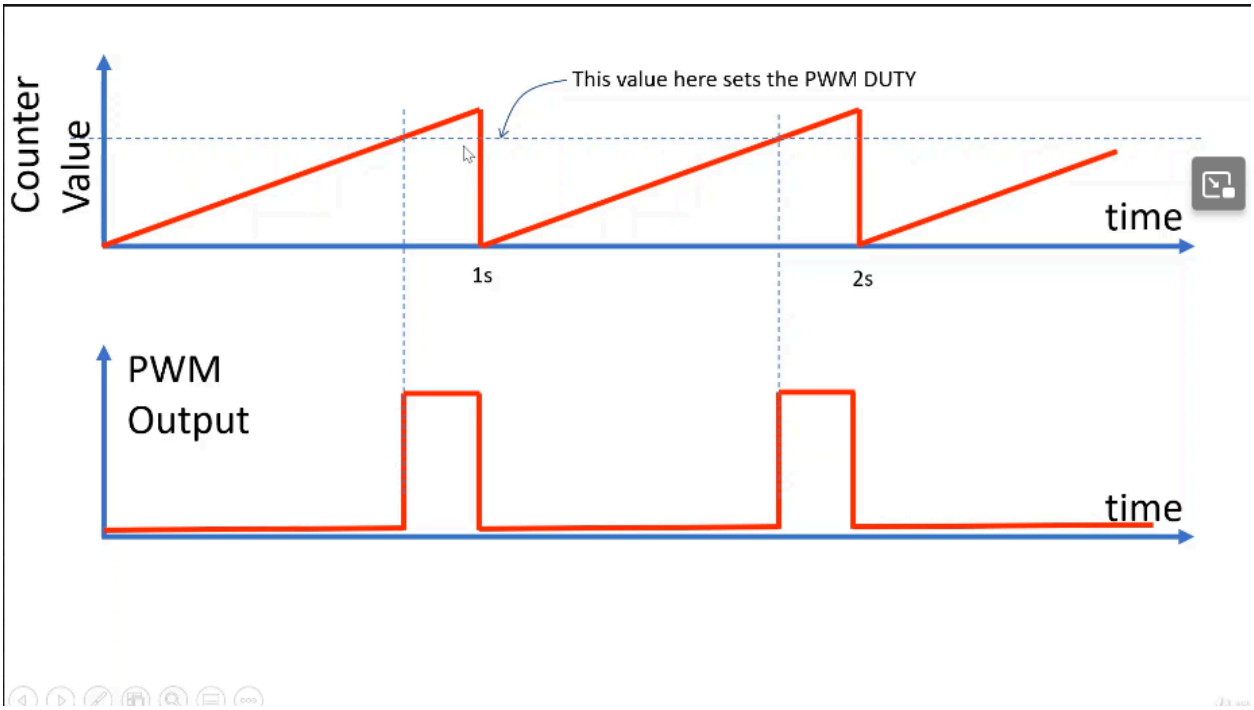
we need 20ns to implement counter by 1, how many times does the counter increments to get a 1 second interval 1/20ns = 50,000,000 increments.

How do we set the duty?

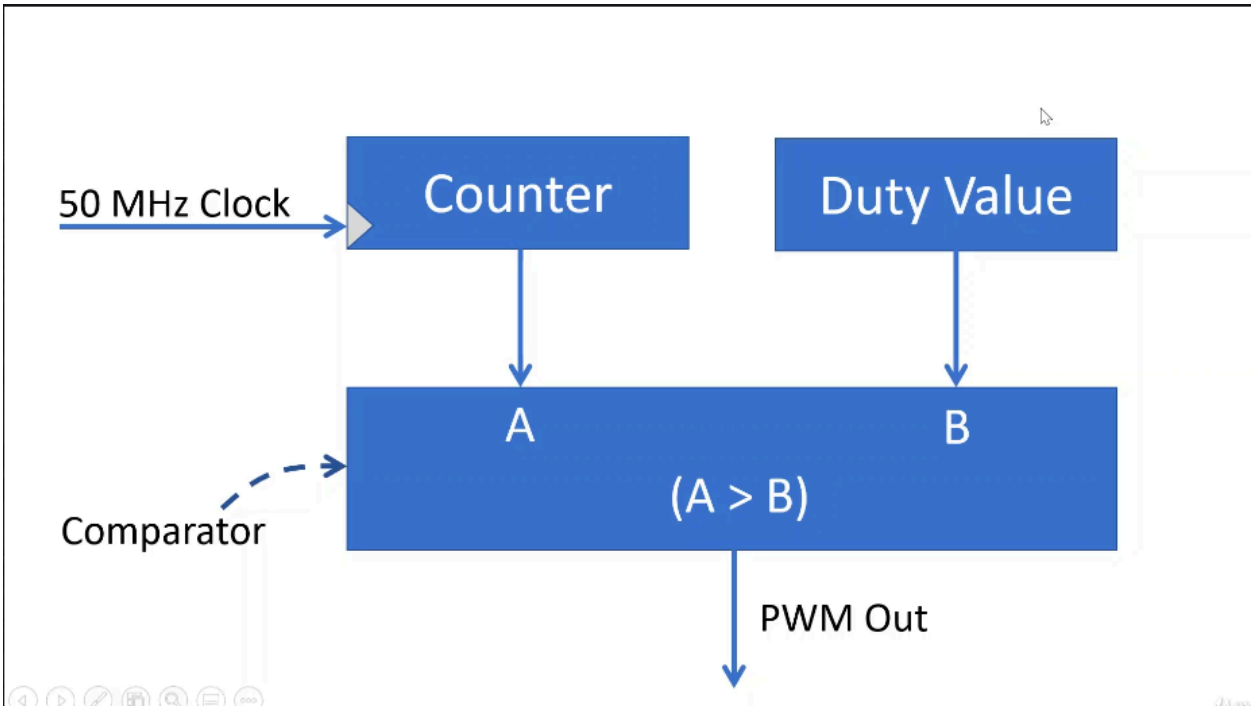
pick a number between 0 and 49 million, like 20 million, it will be used to set the duty



Now to vary the duty, increasing the count to 30 million



if we decreased the count to 10 million it will increase the duty cycle



for 50% duty we will do the duty value at 25 million

What Are Variables?

- Stores intermediate values within a **process**, **function** or **procedure**.
- Cannot be used outside of process, function or procedure.
- Variable declaration is similar to signal declaration.


```
TestProcess : process(Reset, Clock)
     variable Temp: integer; -- Declaration
begin
   if Reset = '1' then
      Temp := 0;
      Sum <= 0;
   elsif rising_edge(Clock) then
      Temp := 0;
      for i in 0 to 7 loop
         Temp := Temp + A(i);
      end loop;
      Sum <= Temp;
    end if;
end process;
```

Functions & Procedures

- Similar to functions in software programming languages.
- Used to perform commonly repeated operations or tasks.
- Called from VHDL code to perform tasks.
- Cannot use registers. Only logic gates.
- Can be declared in architecture, processes & packages.

Functions

- Similar to functions in software programming languages.
- Used to perform regularly used algorithms.
- Can take zero or more inputs.
  - Functions cannot change input values.
- Must return an output value.
  - Cannot return void or omit a return value.
- Cannot contain wait statements.

Procedure Example

```
procedure FullAdder
( signal A, B, C       : in  std_logic;
  signal Sum, Carry : out std_logic
) is
begin
   Sum <= A xor B xor C;
   Carry <= (A and B) or (A and C) or (B and C);
end;

FullAdder (P, Q, R, SO, CO); -- Function Usage
```

Writing A Basic Package

MyPkg.vhd

```
package MyPackage is

    type TLightType is (RED, AMBER, GREEN);

end MyPackage;
```

Example Using Packages

VHDL File 1

```
use work.MyPackage.all;
Architecture Arch1 of Module1 is
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
use work.MyPackage.all;
Architecture Arch1 of Module2 is
    Signal TrafficLight : TLightType;
Begin
```

Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.
- By default, the intermediate forms are stored in the Library you are **currently working in**.
- Most simulators and synthesis tools **automatically create** this default Library for you.
- The library you are working in will have a name of its own.
- You can use the keyword Work to refer to (or point to) the library you are currently working in.

Libraries

- Every library has a name.
- To use a library, you must declare it at the top of your file.
- This makes all the design units in the library available for use in the current design.

Example:

```
Library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
```

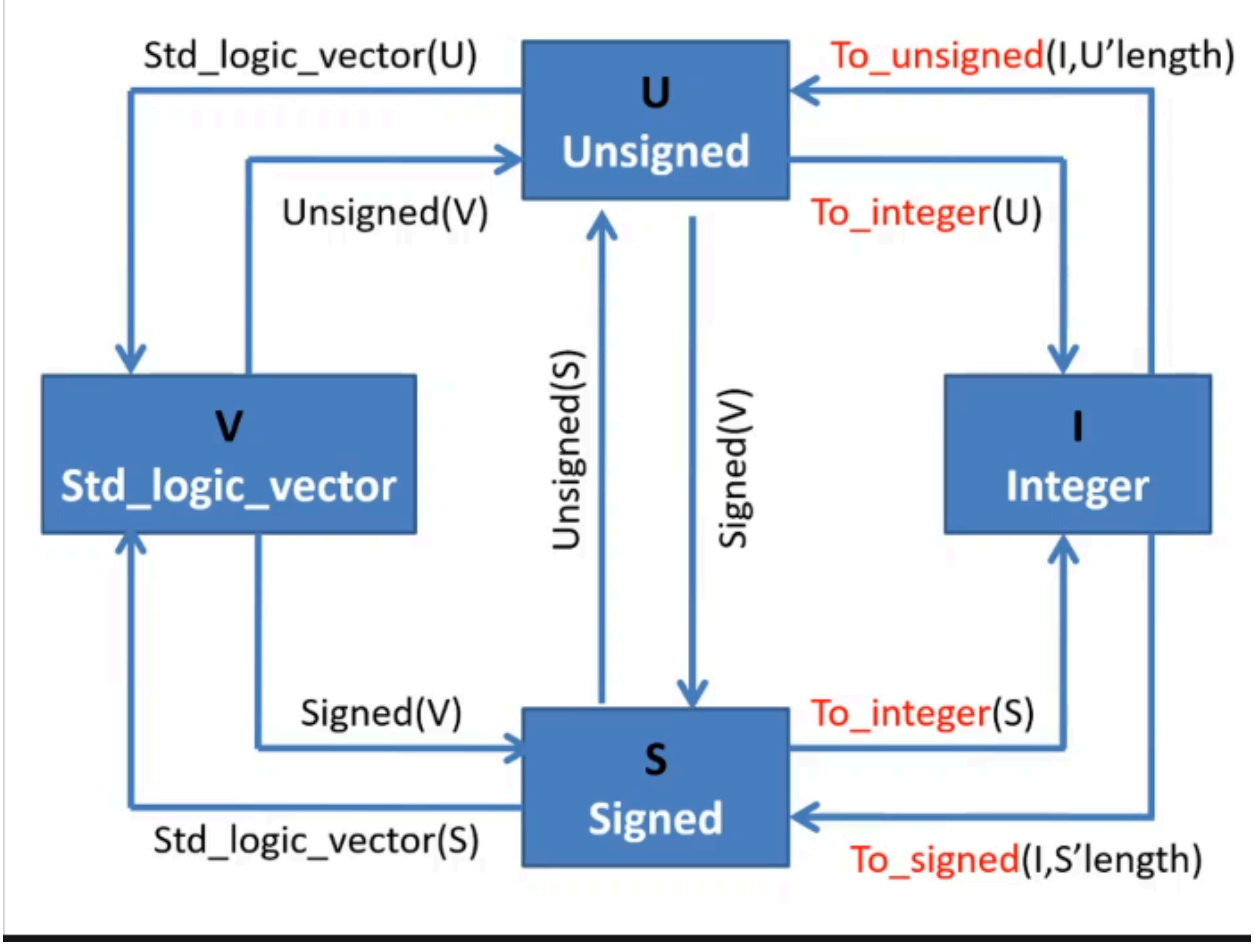
Generics

- Generics are a special type of constant.
- Constant is declared in:
  - The declarative part of the architecture (for module scope).
  - VHDL packages (for global scope)
  - Value of a constant is the same across whole design.
- Generic is declared in:
  - The entity declaration.
  - Value of generic unique to each module instance.

to use 16 or 8 bit adders both

```
entity Adder is
  generic (
      N : integer := 8
  );
   port
   (
      A   : in   unsigned(N-1 downto 0);
      B   : in   unsigned(N-1 downto 0);
      C   : out unsigned(N-1 downto 0)
    );
end entity;

architecture rtl of Adder is
begin
      C <= A + B;
end architecture;
```

```
Adder12Bit : Adder
generic map (
    N => 12
)
port
(
    ...
);

Adder16Bit : Adder
generic map (
    N => 16
)
port
(
    ...
);
```

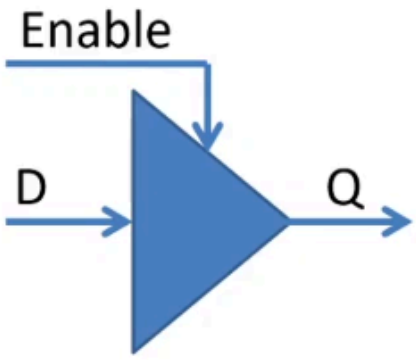

Std_logic_vector(U)
U
Unsigned
To_unsigned(I,U'length)
Unsigned(V)
To_integer(U)
V
Std_logic_vector
Unsigned(S)
Signed(V)
I
Integer
Signed(V)
To_integer(S)
S
Signed
Std_logic_vector(S)
To_signed(I,S'length)

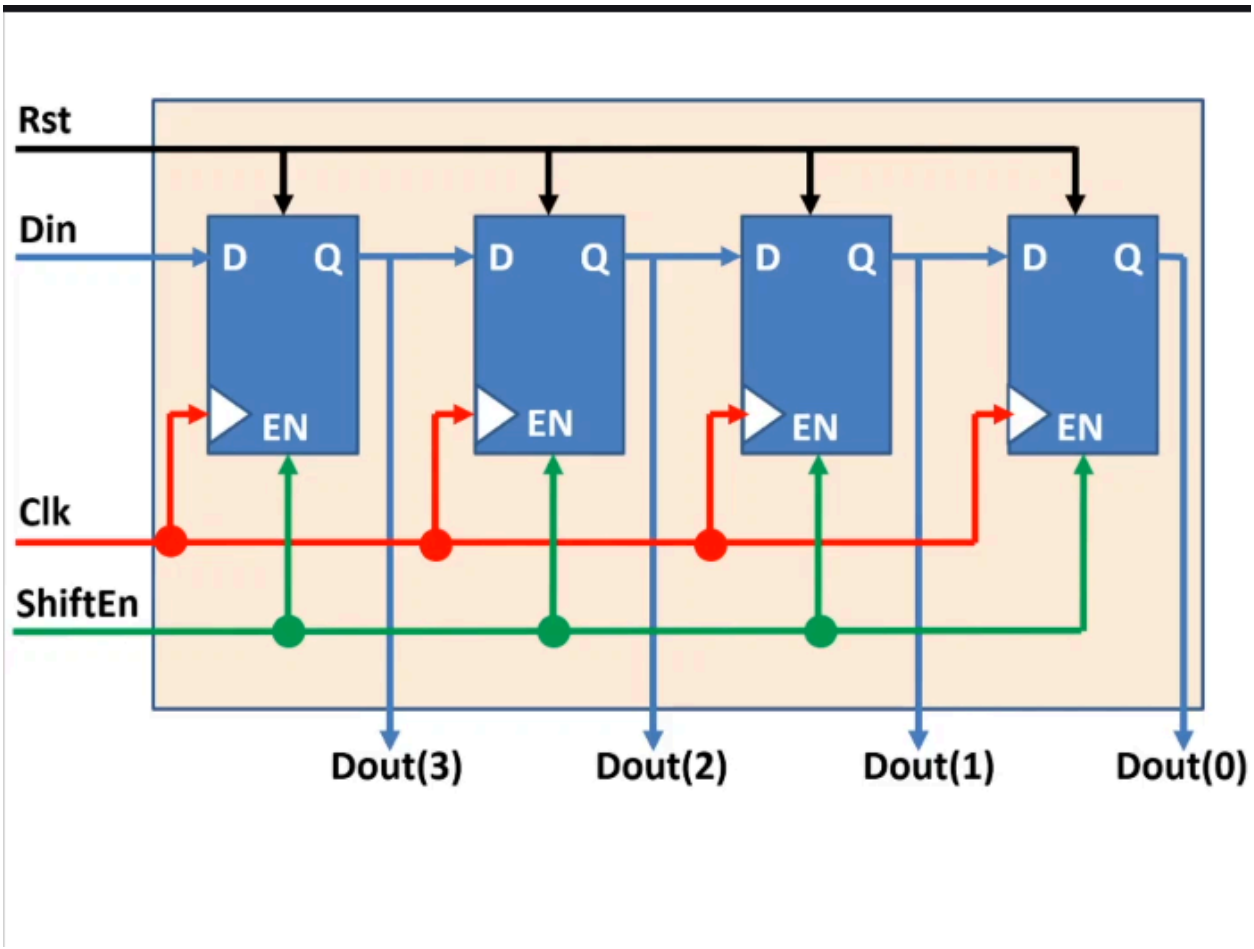
Tri-State Drivers

- Are only available on IO pins of FPGA.
- Cannot have Tri-state signals internal to the FPGA.

Tri-State Drivers

```
TriState_Example : process (D , Enable)
begin
        if Enable = '1' then
                Q  <= D;
        else
                Q <= 'Z';
        end if;
end process;
```





SIPO shift register given above

VHDL for Shift Register

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ShiftRegisterChain is
    generic (
        CHAIN_LENGTH : integer
    );
    port(
        Clk       : in std_logic;
        Rst       : in std_logic;
        ShiftEn   : in std_logic;
        Din       : in std_logic;
        Dout      : out std_logic_vector(CHAIN_LENGTH-1 downto 0)
    );
end entity;
```

VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin


    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = '1' then
            Dout <= (others => '0');
        elsif rising_edge(clk) then
            Dout <= Din  & Dout (Dout'left downto 1) ;
```

VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);

Dout <= Din & Dout (3 downto 1);
```

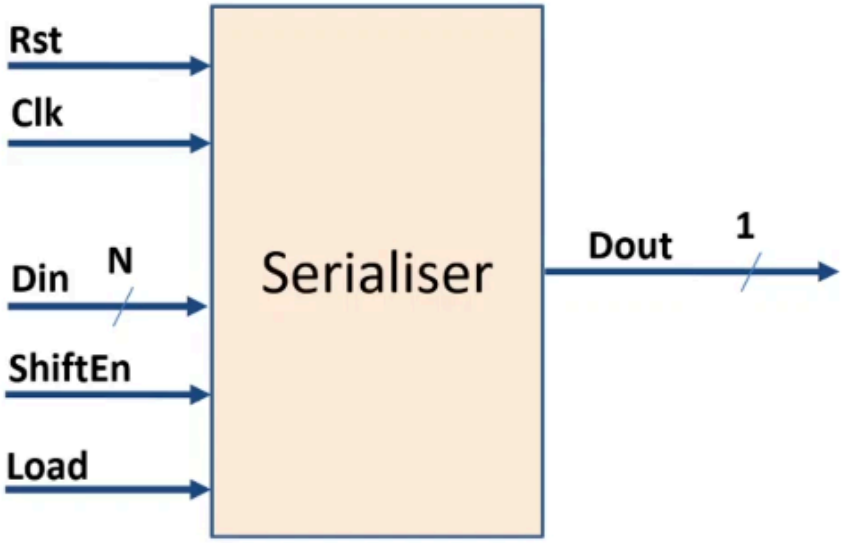
Index	3	2	1	0		Index	3	2	1	0
Value	Dout(3)	Dout(2)	Dout(1)	Dout(0)	<=	Value	Din	Dout(3)	Dout(2)	Dout(1)

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```

VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin


    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = '1' then
            Dout <= (others => '0');
        elsif rising_edge(clk) then
            if ShiftEn = '1' then
                Dout <= Din  & Dout (Dout'left downto 1) ;
            end if;
        end if;
    end process;
end rtl;
```

serializers make parallel data to serial using registers


Rst
Clk
Din
N
ShiftEn
Load
Serialiser
Dout
1

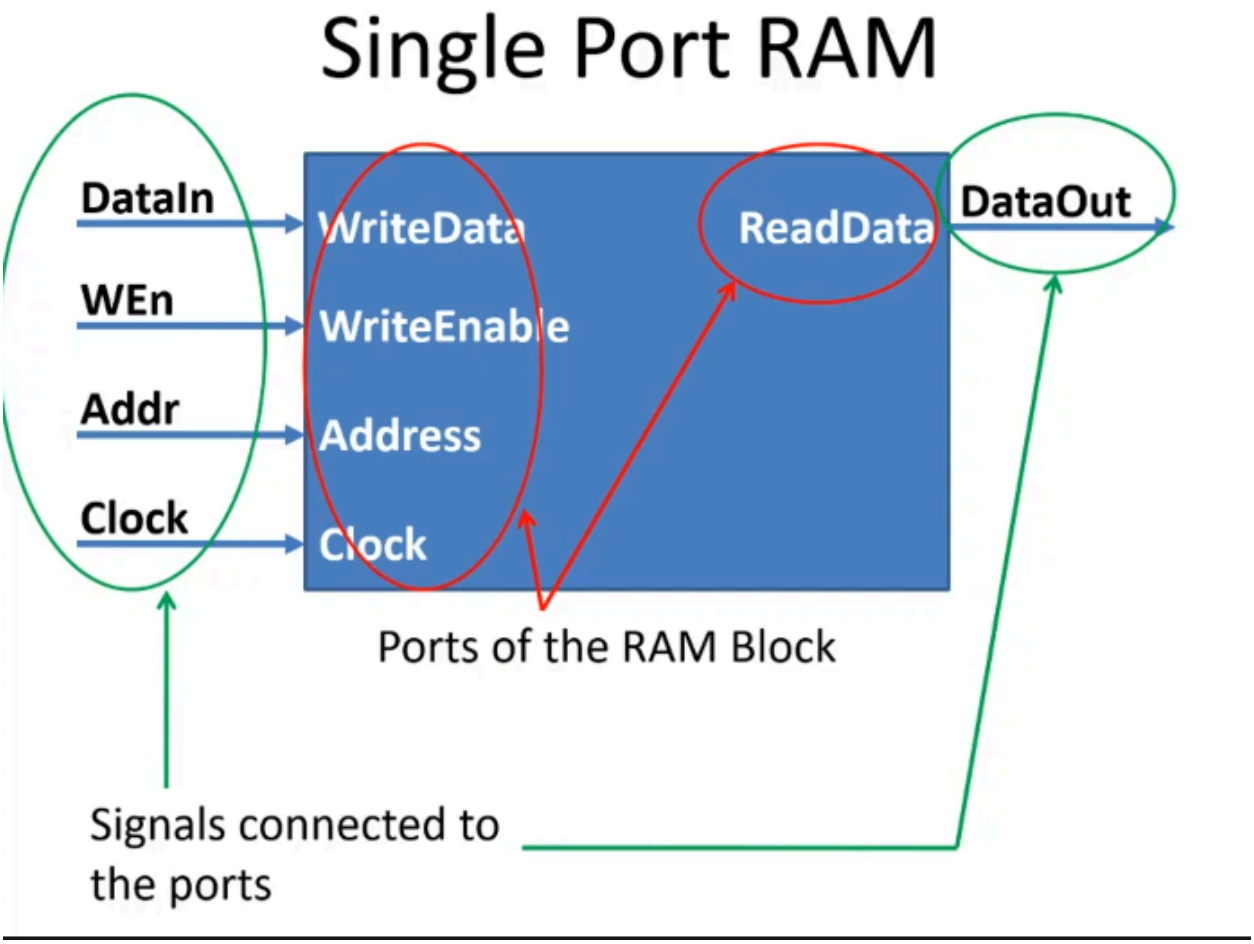
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Serialiser is
    generic (
        DataWidth : integer
    );
    port(
        Clk       : in std_logic;
        Rst       : in std_logic;
        ShiftEn   : in std_logic;
        Load      : in std_logic;
        Din       : in std_logic_vector(DataWidth -1 downto 0);
        Dout      : out std_logic
    );
end entity;
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(rst,clk)
    begin
        if rst = '1' then
                DataRegister <= (others => '0');
        elsif rising_edge(clk) then
                if Load = '1' then
                        DataRegister <= DIN;
                elsif ShiftEn = '1' then
                        DataRegister <= '0' & DataRegister(DataWidth-1 downto 1);
                end if;
        end if;
    end process;
    Dout <= DataRegister(0);
end rtl;
```

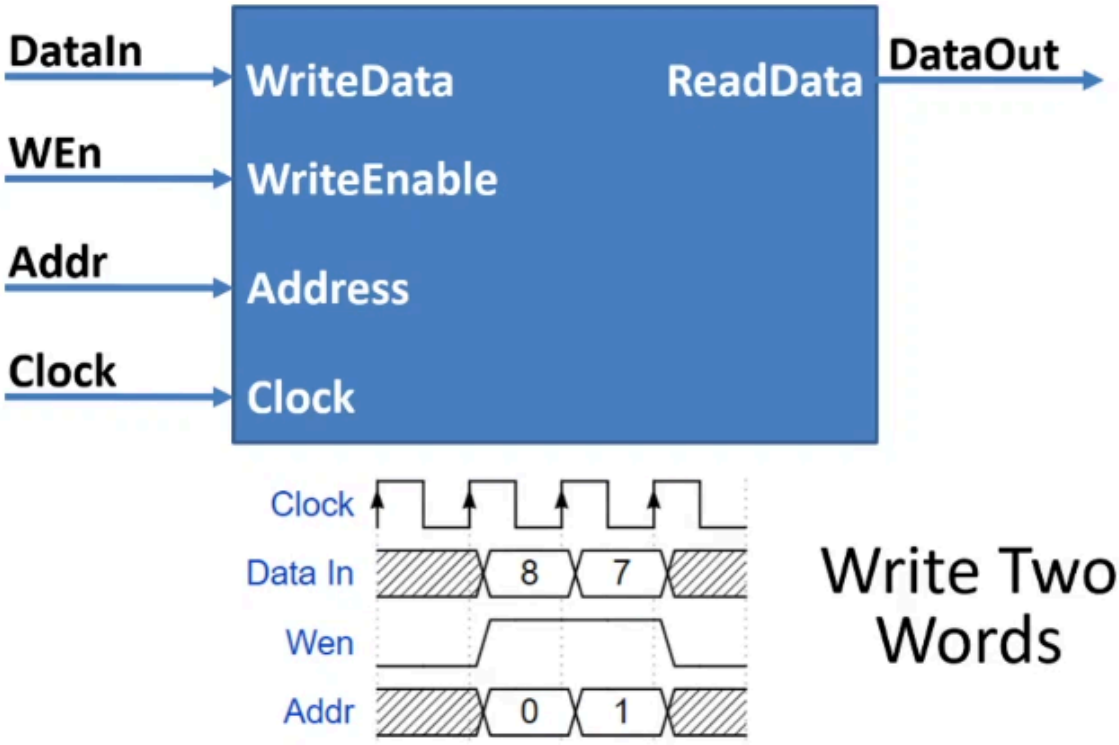
RAM Implementation

- FPGAs implement RAM using :
  1. Register Banks (for smaller memories)
  2. Block RAM (for larger memories)
- Synthesiser infers how to implement RAM by looking at your VHDL code.
- Synthesiser by default goes for Register bank implementation.
- VHDL code must follow a particular template to get the synthesiser to use Block RAM.

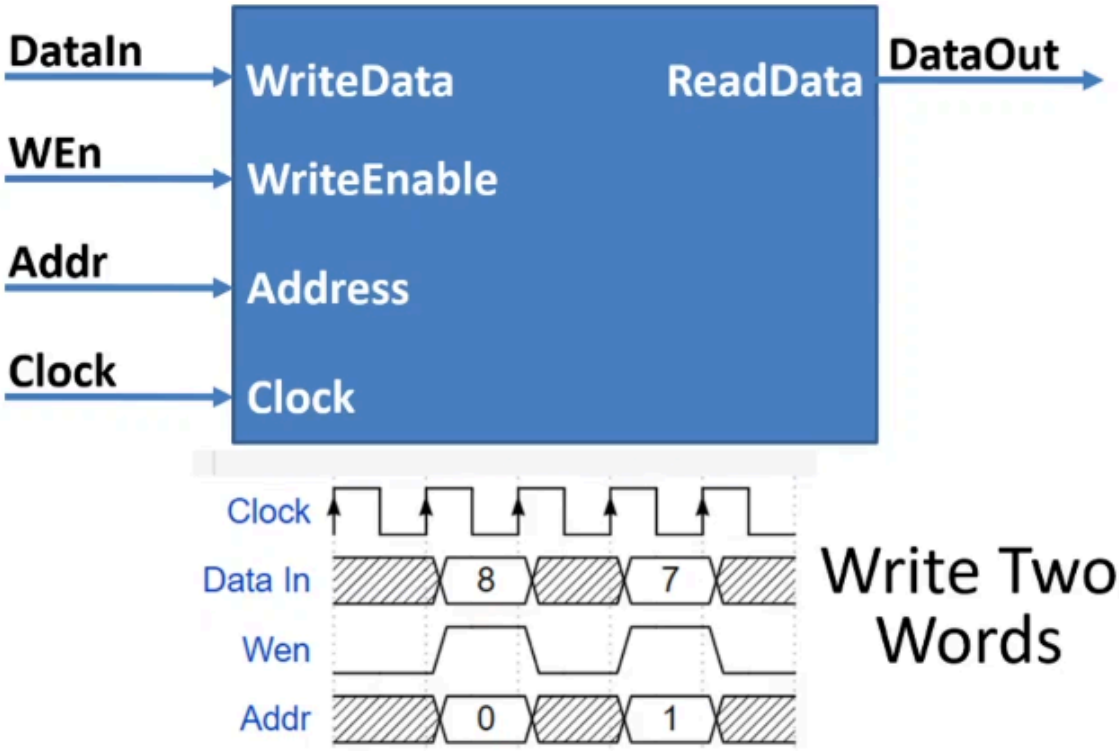
register RAMS are limited so use wisely


Single Port RAM
DataIn
WEn
Addr
Clock
WriteData
WriteEnable
Address
Clock
ReadData
DataOut
Ports of the RAM Block
Signals connected to the ports

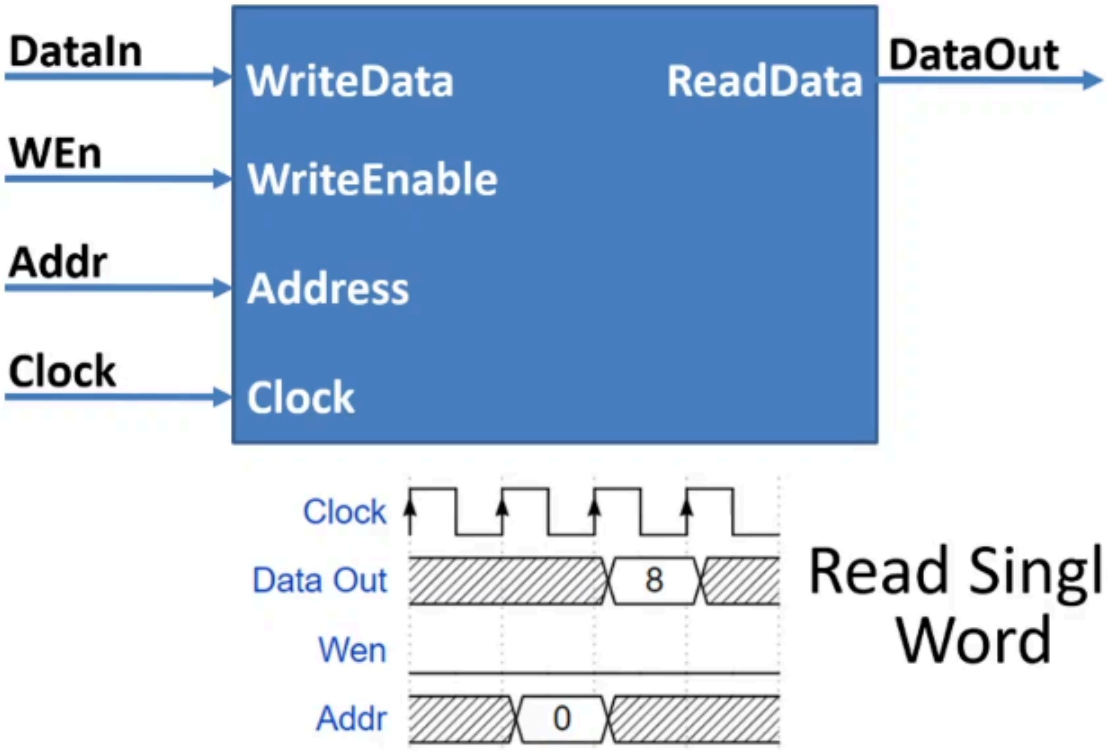
Single Port RAM



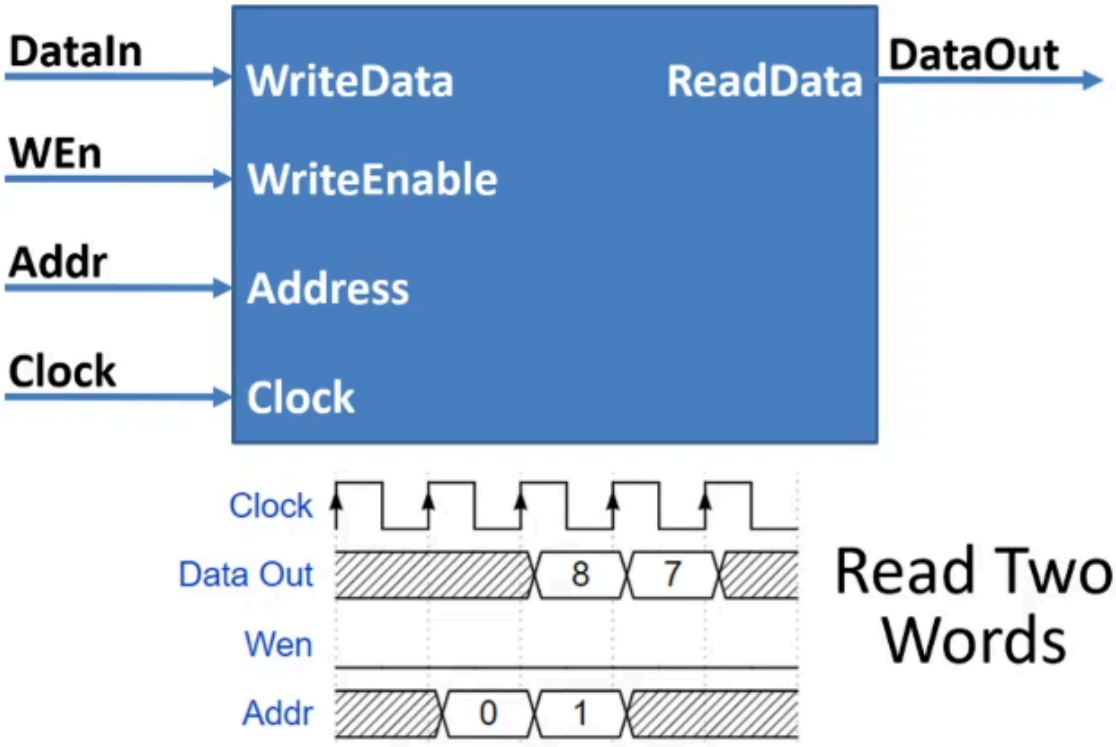
Single Port RAM



Single Port RAM



Single Port RAM



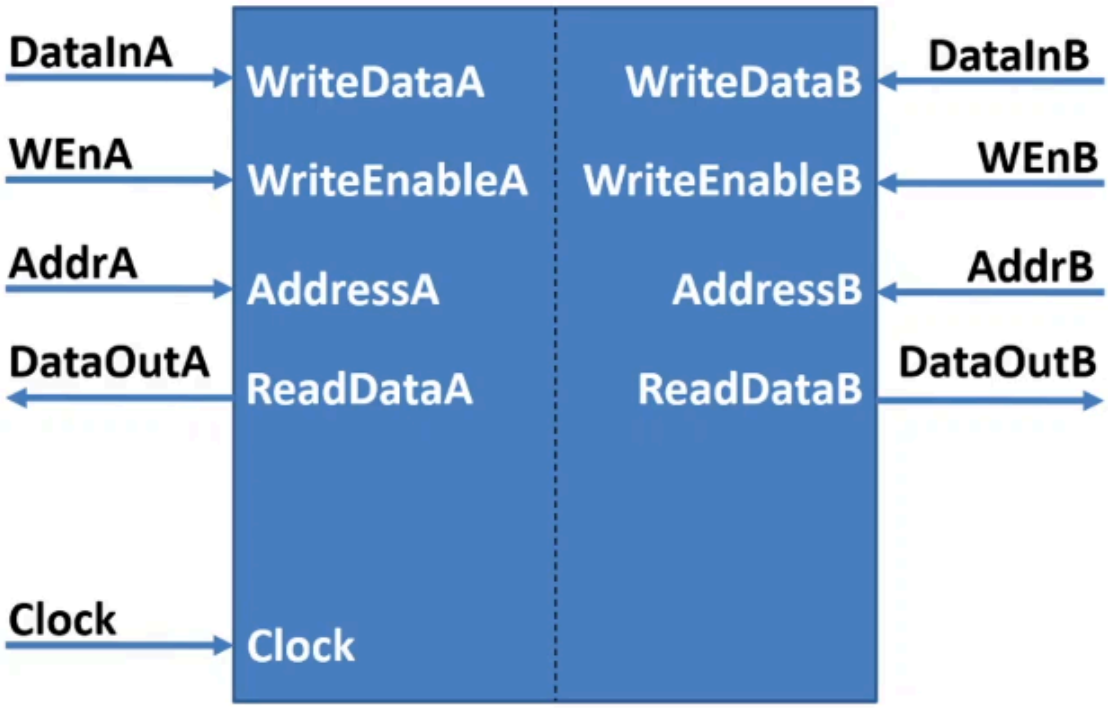
VHDL for Single Port RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
```

VHDL for Single Port RAM

```
entity SinglePortRAM is
    generic(
        Dwidth : integer;
        Awidth : integer
    );
    port(
        Clock        : in  std_logic;
        WriteData    : in  std_logic_vector(Dwidth - 1 downto 0);
        ReadData     : out std_logic_vector(Dwidth - 1 downto 0);
        Address      : in  std_logic_vector(Awidth - 1 downto 0);
        WriteEnable  : in  std_logic
    );
end SinglePortRAM ;
```

Dual Port RAM



VHDL for Dual Port RAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
```

VHDL for Dual Port RAM

```
entity DualPortRAM is
    generic(
        Dwidth : integer;
        Awidth : integer
    );
    port(
        Clock           : in  std_logic;

        WriteDataA      : in  std_logic_vector(Dwidth - 1 downto 0);
        ReadDataA       : out std_logic_vector(Dwidth - 1 downto 0);
        AddressA        : in  std_logic_vector(Awidth - 1 downto 0);
        WriteEnableA    : in  std_logic;

        WriteDataB      : in  std_logic_vector(Dwidth - 1 downto 0);
        ReadDataB       : out std_logic_vector(Dwidth - 1 downto 0);
        AddressB        : in  std_logic_vector(Awidth - 1 downto 0);
        WriteEnableB    : in  std_logic
    );
end DualPortRAM ;
```

These ports implement the 1st "port" which I have called A.

These ports implement the 2nd "port" which I have called B.

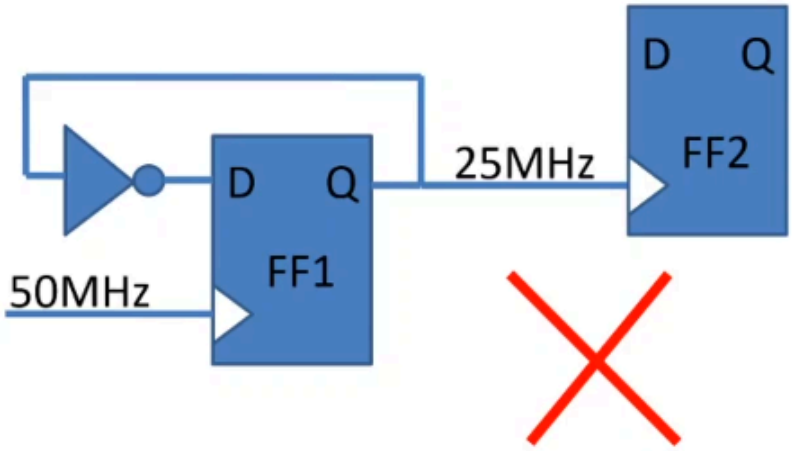
Design Practices :

Avoid Creating Latches

- Watch out for accidental latch creation inside combinational processes.
- Only a problem when there are conditional statements (if then / case).
- Make sure the outputs are defined in all possible cases.

Do Not Generate Clocks Using Logic

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.

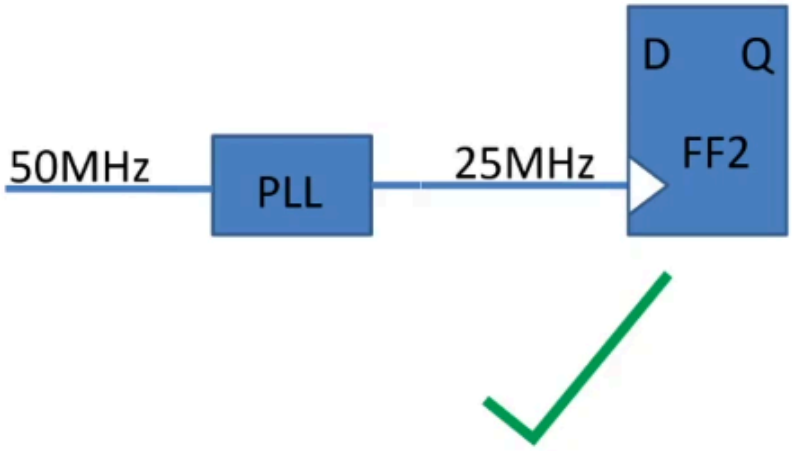


Do Not Generate Clocks Using Logic

- Using a clocks generated by FPGA logic results in slower designs.
- Because FPGA will be forced to use non-global routing resources to some degree.
- These paths have much higher skew resulting in sub-optimal timing performance.

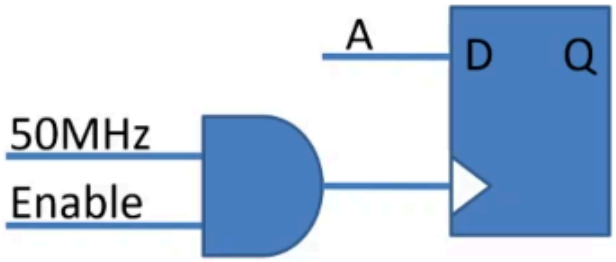
Better Way To Generate Clocks

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.



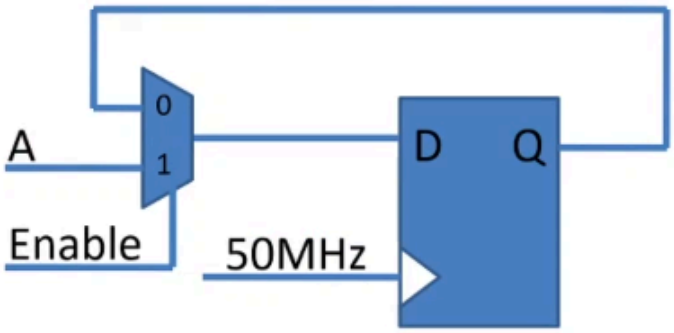
Do Not Use Clock Gating

- For the same reasons, do not use clock gating.



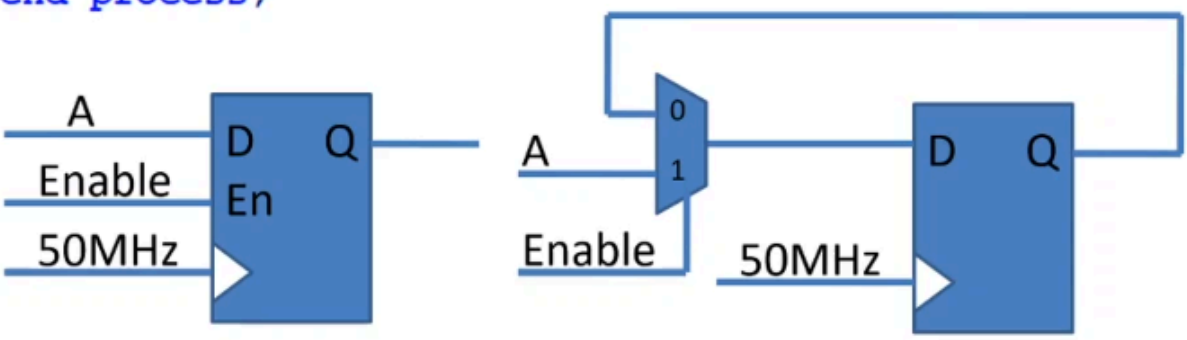
Use Data Gating Instead

- For the same reasons, do not use clock gating.



Use Data Gating Instead

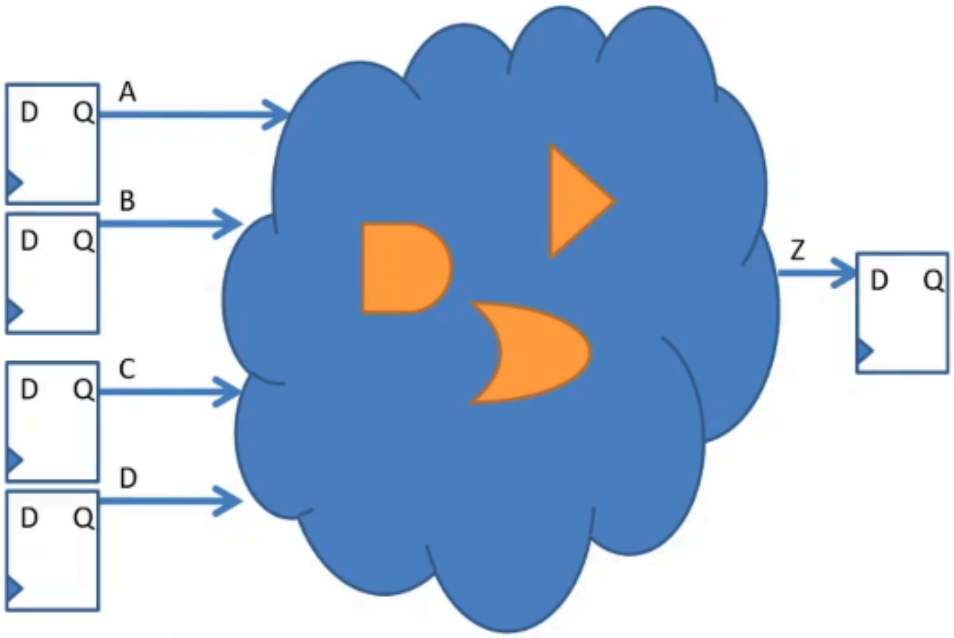
```
DataGatingExample : process(Reset, Clk_50MHz)
begin
    if Reset = '1' then
        Output <= '0';
    elsif rising_edge(Clk_50MHz) then
        if Enable = '1' then
            Output <= A;
        end if;
    end if;
end process;
```



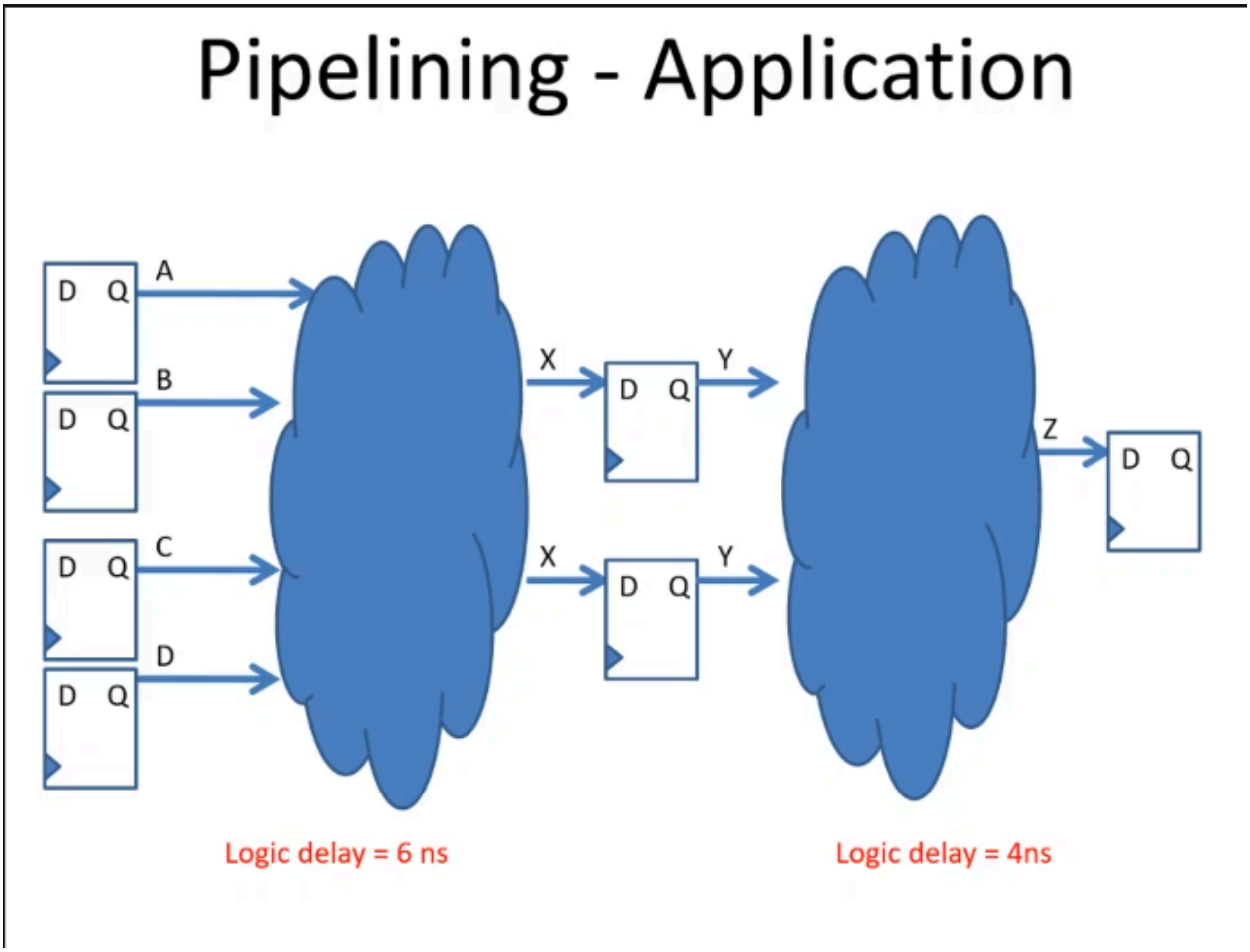
Pipelining

- Large combinational functions have big delays leading to a timing bottleneck.
- This leads to a slower FPGA design.
- Pipelining can be very effective in mitigating these delays and boost timing performance.
- Pipelining uses extra registers to break up a large combinational function.

Pipelining - Application



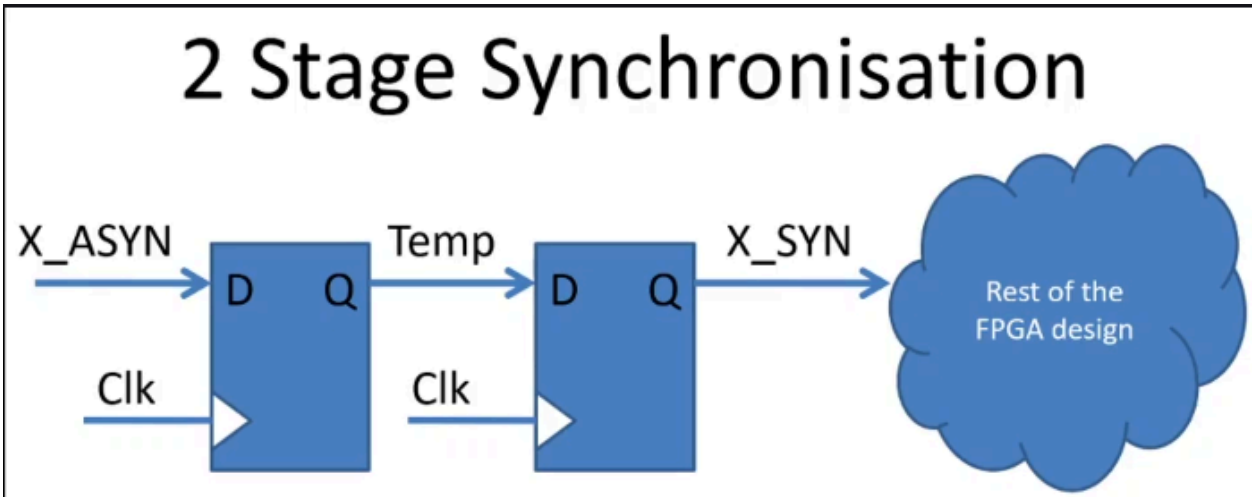
Logic delay = 10 ns



pipelining helps increase clocking rates as we have reduced delays

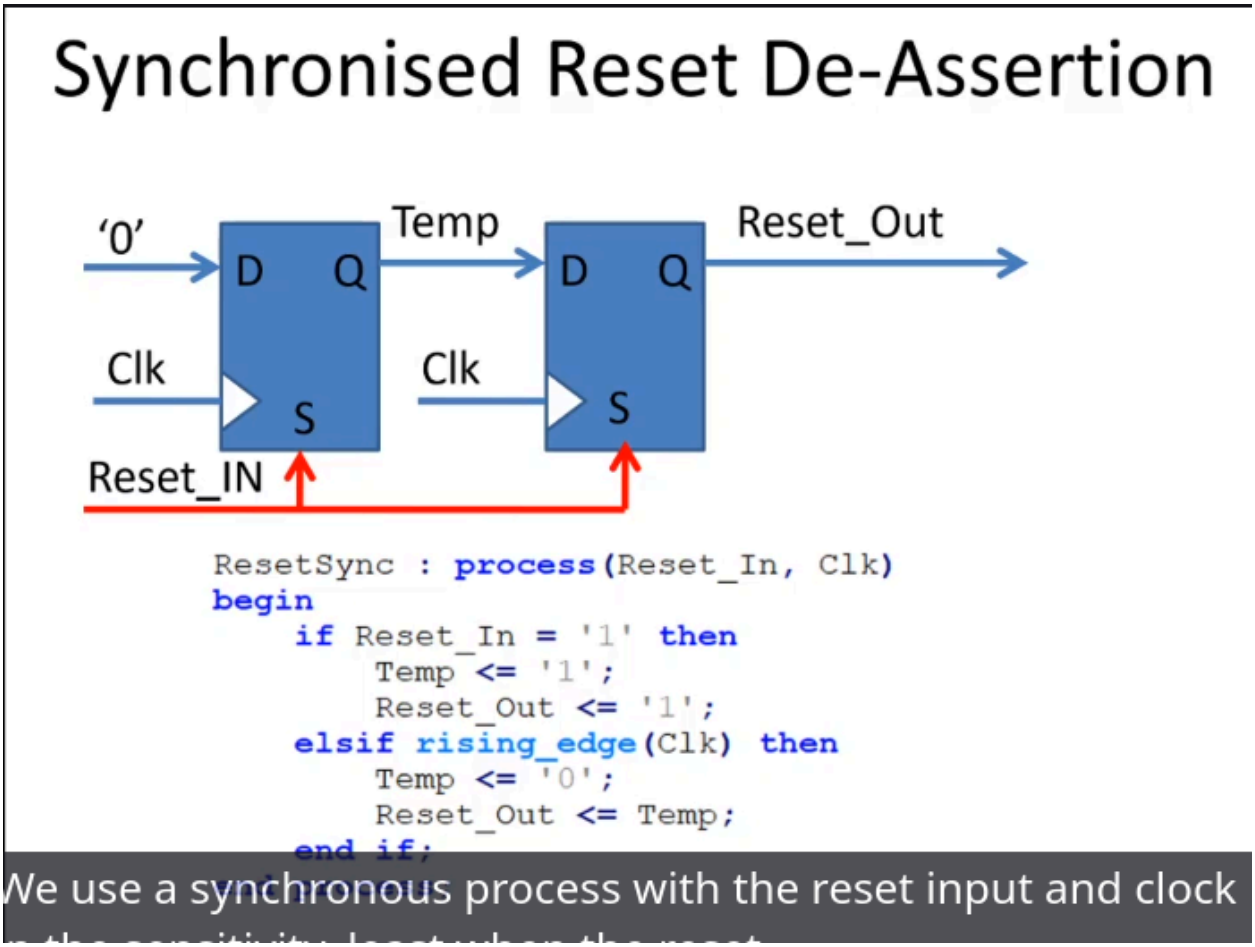
Asynchronous Inputs

- Asynchronous inputs can change state at any point in time:
    - Push button input
    - RS232 Data
- These signals are not driven from a clock.
- We must make sure to synchronise **all** asynchronous inputs before using them.




```
Synchronisation : process(Reset, Clk)
begin
    if Reset = '1' then
        Temp <= '0';
        X_SYN <= '0';
    elsif rising_edge(Clk) then
        Temp <= X_ASYN;
        X_SYN <= Temp;
    end if;
end process;
```

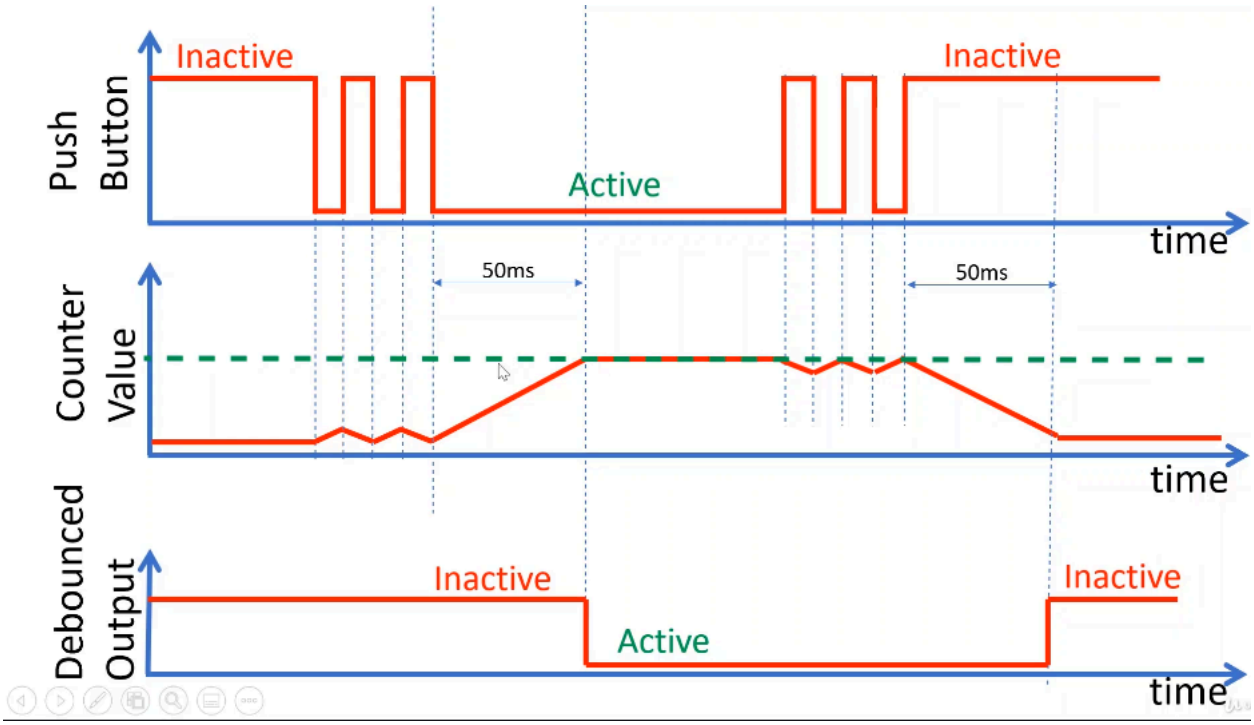
https://vlsi.pro/synchronous-asynchronous-reset/



debouncing is caused due to mechanical contact noise

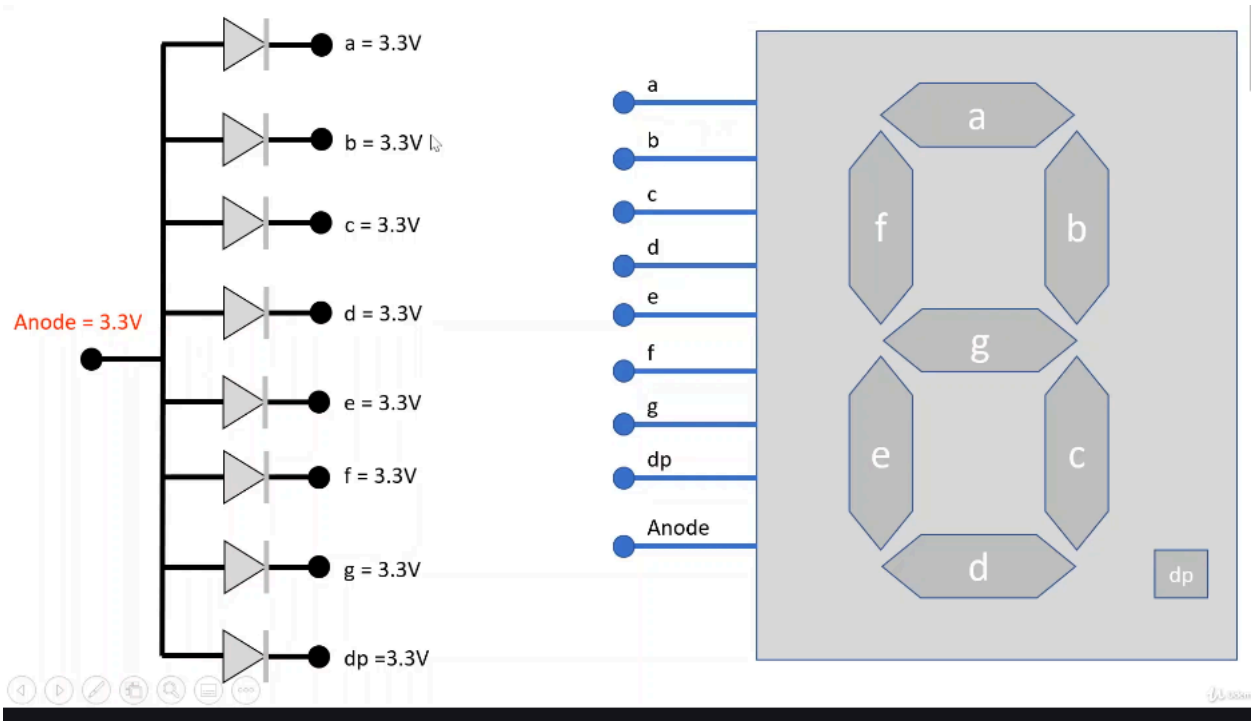
we need to debounce to clean the signal if switch is crucial, like when switch is counting for a process, as it counts per every falling edge so the extra bounces cause mismatched counts

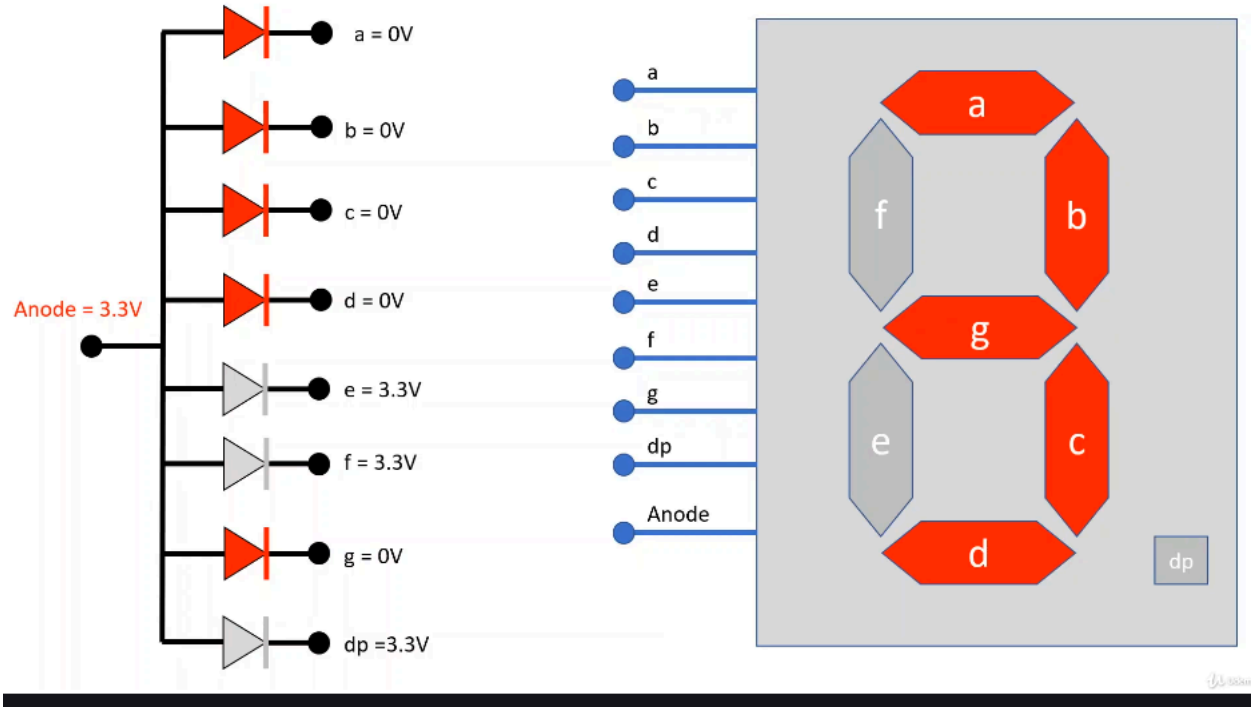
switch stays high as long as not pressed but when pressed it grounds and voltage goes to 0



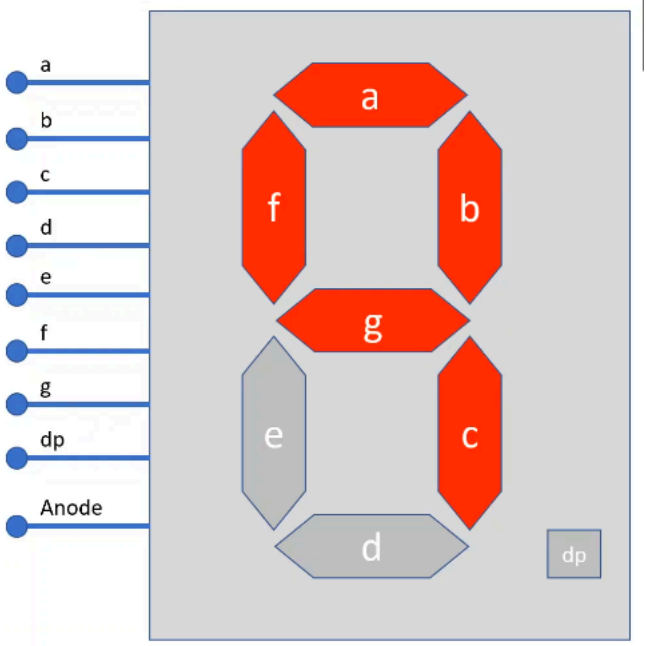
FOCUS ON TIMING ANALYSER

all off state

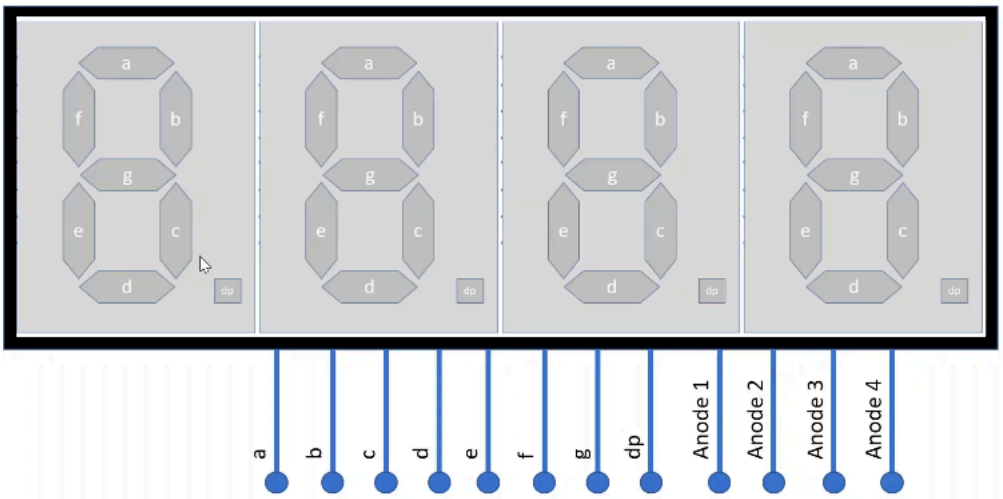

a = 3.3V
b = 3.3V
c = 3.3V
d = 3.3V
Anode = 3.3V
e = 3.3V
f = 3.3V
g = 3.3V
dp =3.3V
a
b
c
d
e
f
g
dp
Anode
a
f
b
g
e
c
d
dp

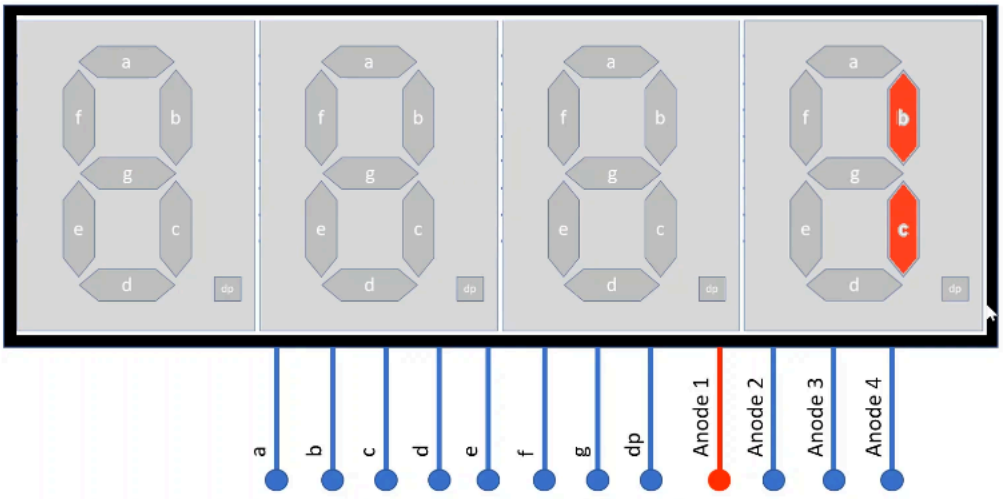

a = 0V
b = 0V
c = 0V
d = 0V
Anode = 3.3V
e = 3.3V
f = 3.3V
g = 0V
dp =3.3V
a
b
c
d
e
f
g
dp
Anode
a
f
b
g
e
c
d
dp

#	g	f	e	d	c	b	a
0	-	x	x	x	x	x	x
1	-	-	-	-	x	x	-
2	x	-	x	x	-	x	x
3	x	-	-	x	x	x	x
4	x	x	-	-	x	x	-
5	x	x	-	x	x	-	x
6	x	x	x	x	-	-	x
7	-	-	-	-	x	x	x
8	x	x	x	x	x	x	x
9	x	x	-	-	x	x	x



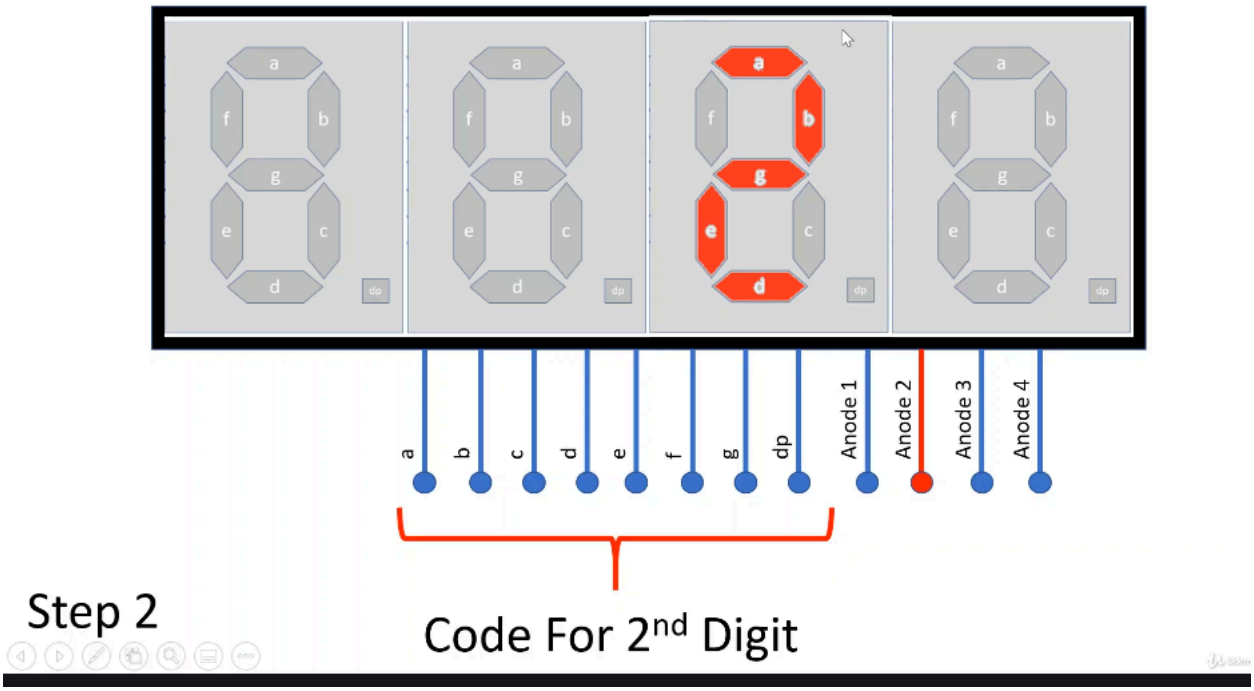
#	g	f	e	d	c	b	a	
0	-	x	x	x	x	x	x	011 1111b = 0x3F
1	-	-	-	-	x	x	-	000 0110b = 0x06
2	x	-	x	x	-	x	x	101 1011b = 0x5B
3	x	-	-	x	x	x	x	100 1111b = 0x4F
4	x	x	-	-	x	x	-	110 0110b = 0x66
5	x	x	-	x	x	-	x	110 1101b = 0x6D
6	x	x	x	x	-	-	x	111 1001b = 0x79
7	-	-	-	-	x	x	x	000 0111b = 0x07
8	x	x	x	x	x	x	x	111 1111b = 0x7F
9	x	x	-	-	x	x	x	110 0111b = 0x67

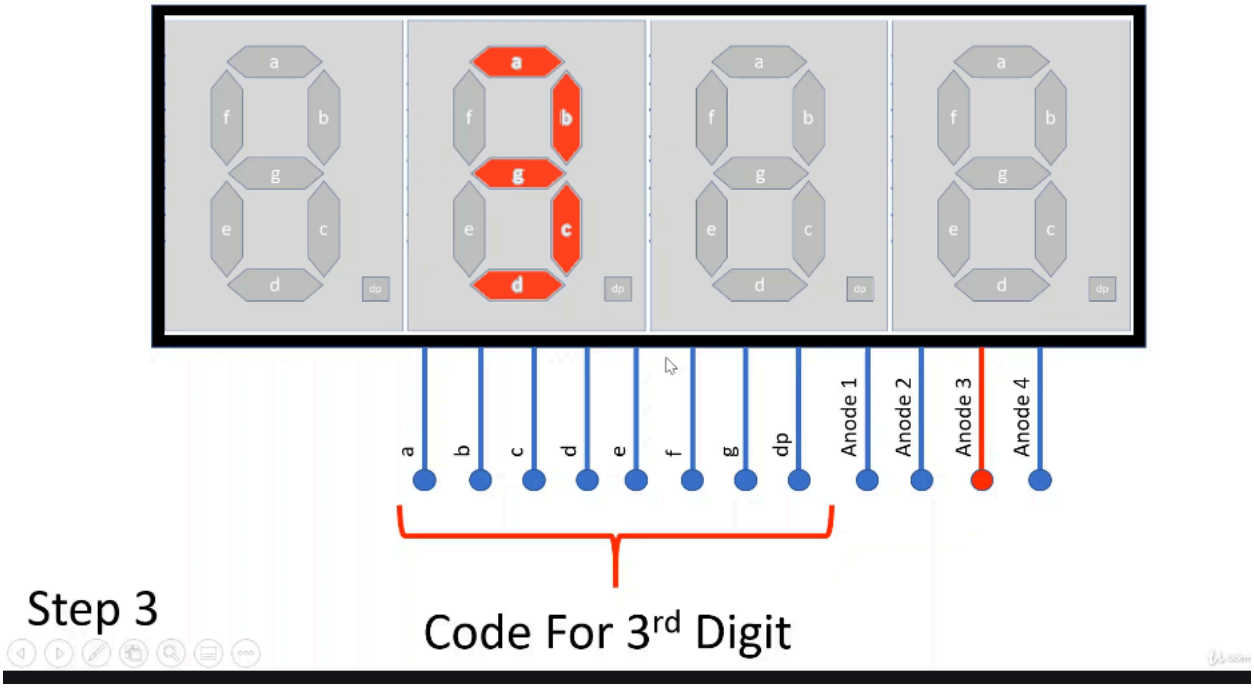


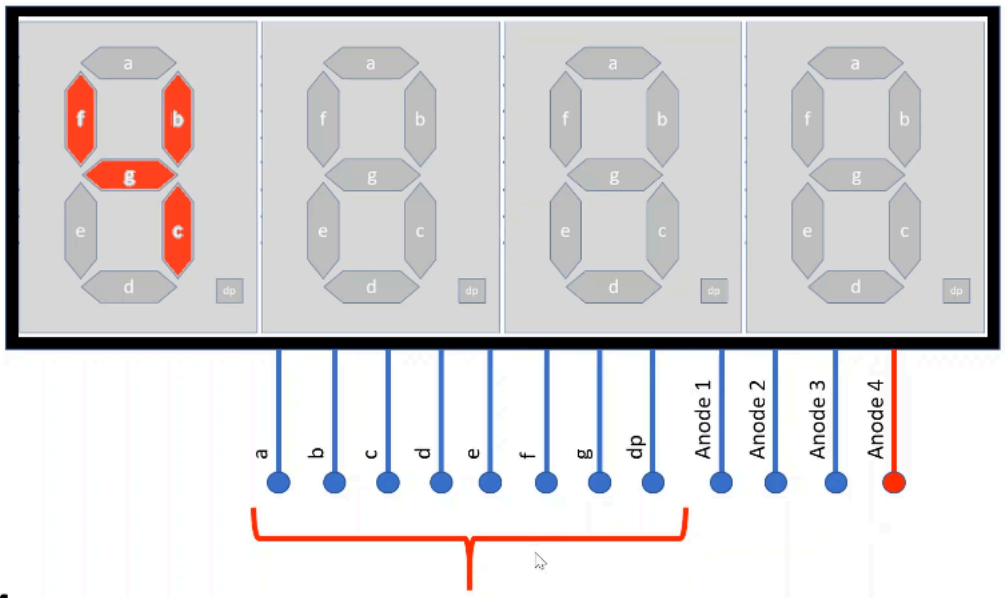


Step 1

Code For 1st Digit


a
b
c
d
e
f
g
dp
Anode 1
Anode 2
Anode 3
Anode 4
Step 2
Code For 2nd Digit


a
b
c
d
e
f
g
dp
Anode 1
Anode 2
Anode 3
Anode 4
Step 3
Code For 3rd Digit



Step 4

Code For 4th Digit

Assert Statement

- The assert statement tests a Boolean condition.
- If false, it outputs a message string to the simulator output console.

Syntax:

```
assert condition report string severity severity_level;
```

e.g.:

```
assert Overflow = '0'
report "An overflow has occurred!" severity warning;
```

Using Text Files

- Must declare **Textio** package ("use std.textio.all;")
- Allows for more complex testing.
- Input vectors can be read from text files.
- Output vectors can be written to text files.
- Data types stored in files can be anything (real numbers, strings, std_logic_vectors, integers etc).
- Basic operations with text files :
  - Declaration of a text file.
  - Opening and closing a file of a text file.
  - Reading and or writing to a file.

Declaring a Text File

Syntax :

```
file FileHandle : text;
```

e.g.:

```
file File1 : text;
```

Where to declare :

- Declarative region of the architecture
- Declarative region of a process

Opening A Text File

Syntax :

file_open(FileHandle, “FileName.txt”, OpenMode);

Open Mode :

1. read_Mode
2. write_Mode
3. append

E.g.:

file_open(File1, “InputVectors.txt”, read_mode);

file_open(File2, “OutputVectors.txt”, write_mode);

Closing A File

Syntax :

```
file_close(FileHandle);
```

E.g.:

```
file_close(File1);
```

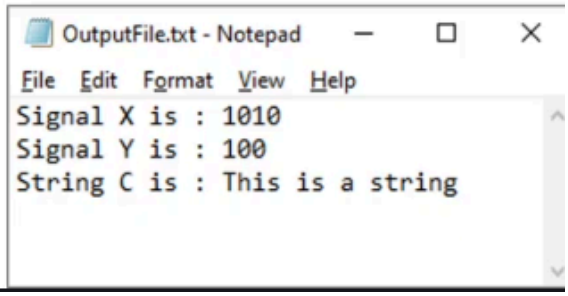
Writing To a Text File

1. Open a text file in write mode.
2. Write to a line buffer:

   write(LineBuffer, DataToWrite);
3. Write line buffer to file:

   writeline(FileHandle, LineBuffer);
4. Close file.

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

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);

        file_close(OutputFile);
        wait;
    end process;

end architecture rtl;
```



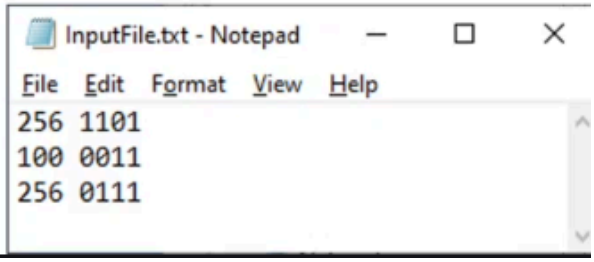
Reading From a Text File

1. Open a text file in read mode.
2. Read from a line from the text file:

   readline(FileHandle, LineBuffer);
3. Read a data value from the line buffer to a signal:

   read(LineBuffer, SignalName);
4. Repeat steps 2 and 3 until all lines of text file have been read.
5. Close file.

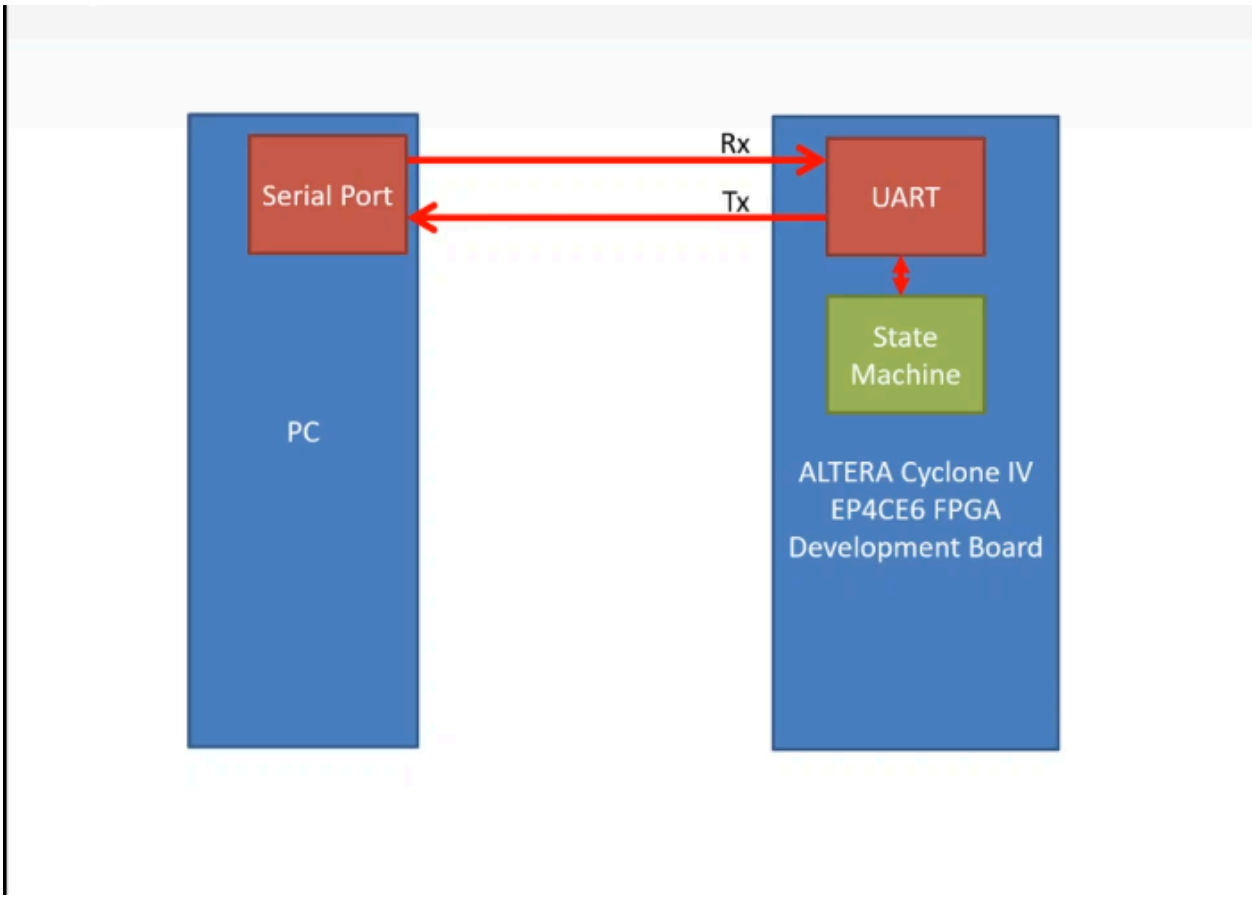
```
40      FileReadProcess:process
41          file InputFile : text;
42          variable lineBuffer : line;
43          variable Char : character;
44          variable int : integer;
45          variable vector : std_logic_vector(3 downto 0):= "0000";
46      begin
47          file_open(InputFile, "InputFile.txt",read_mode);
48
49          while not endfile(InputFile) loop
50              readline(InputFile, lineBuffer);
51              read(lineBuffer, int);
52              read(lineBuffer, Char);
53              read(lineBuffer, vector);
54          end loop;
55
56          file_close(InputFile);
57
58          wait;
59      end process;
```



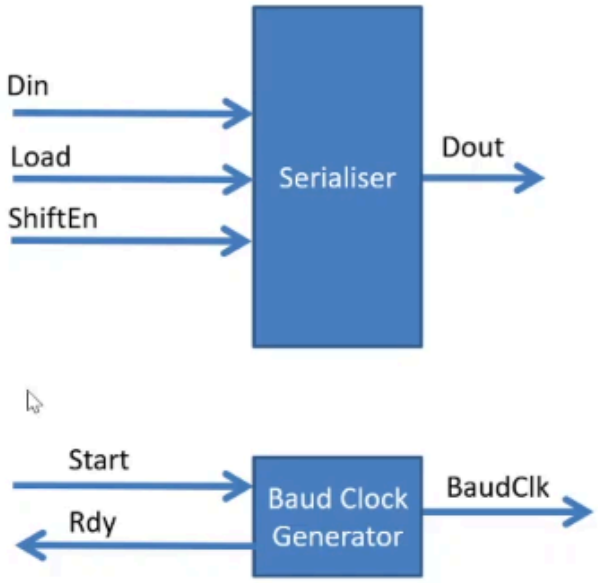
UART : IMPORTANT

see - https://www.youtube.com/results?search_query=uart+protocol+tutorial

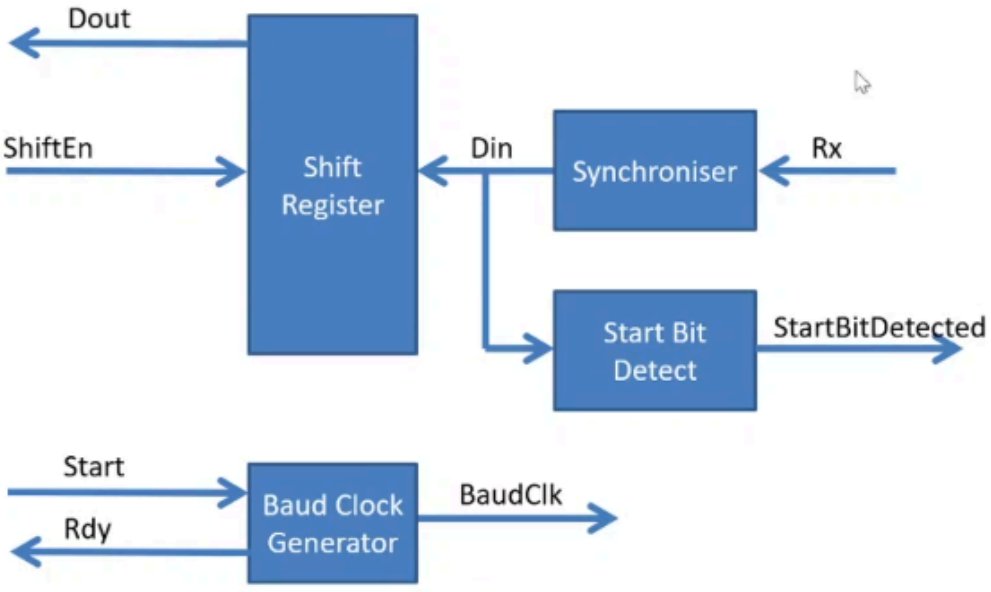
UART receives characters from a serial port of a PC



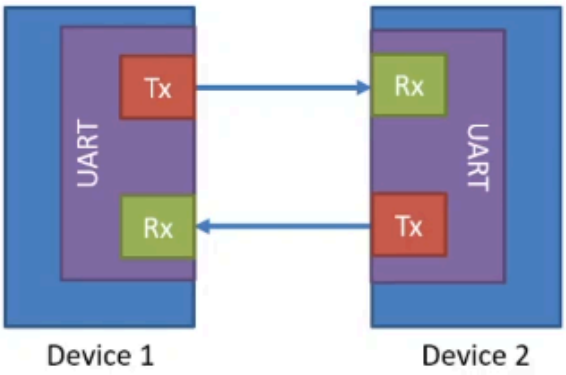
UART Transmitter



UART Receiver



RS232 Protocol

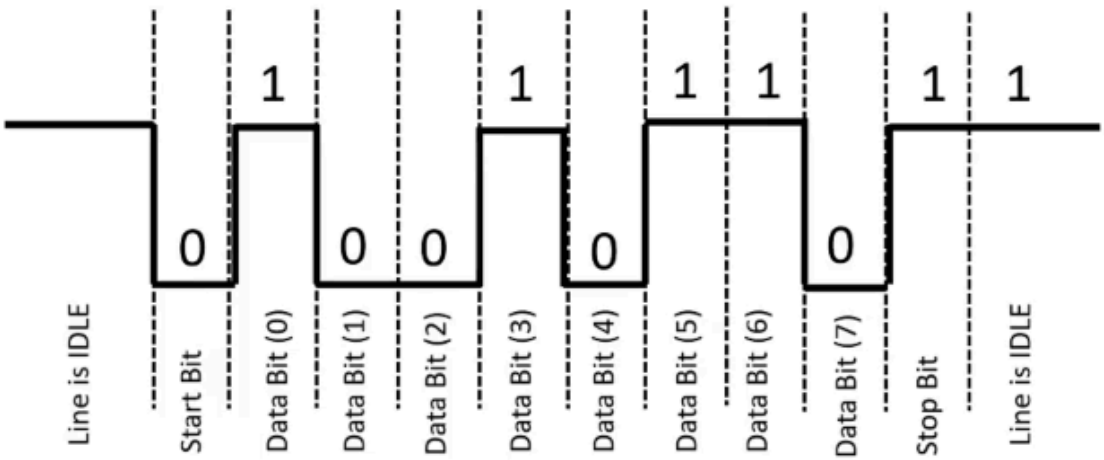


RS232 Protocol

Start Bit	Data Bits	Parity Bit	Stop Bit(s)

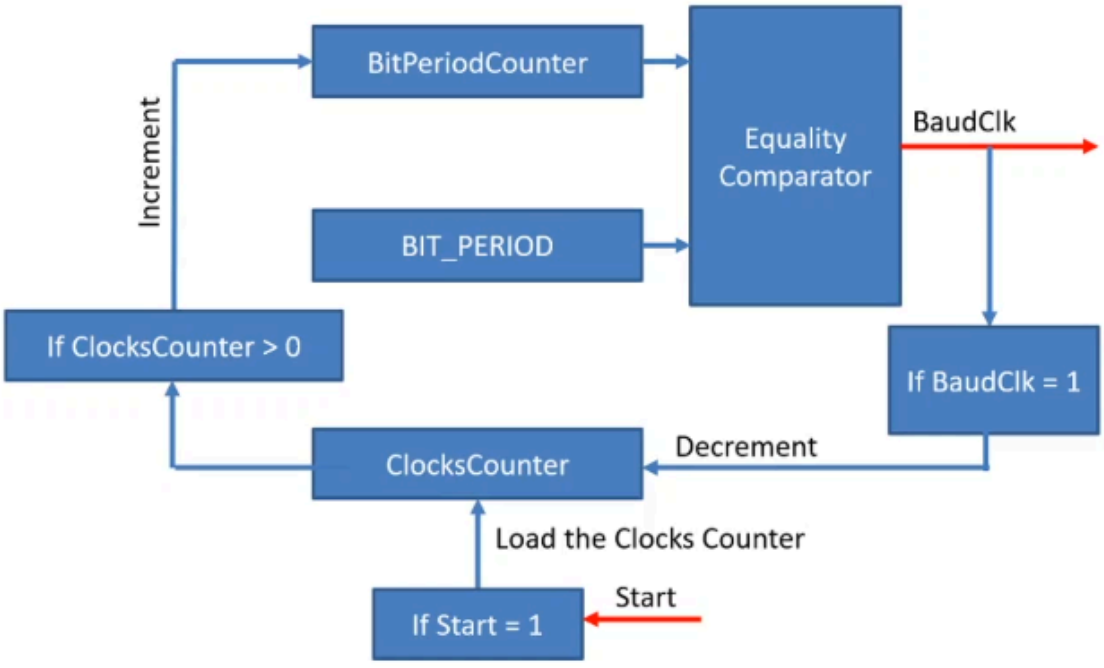
Start Bit = Supports 1 start bit. Logic 0.
Data Bits = Supports 5 to 9 data bits
Parity Bit = Optional (Even or Odd Parity)
Stop Bits = Supports 1, 1.5 or 2 stop bits. Logic 1.

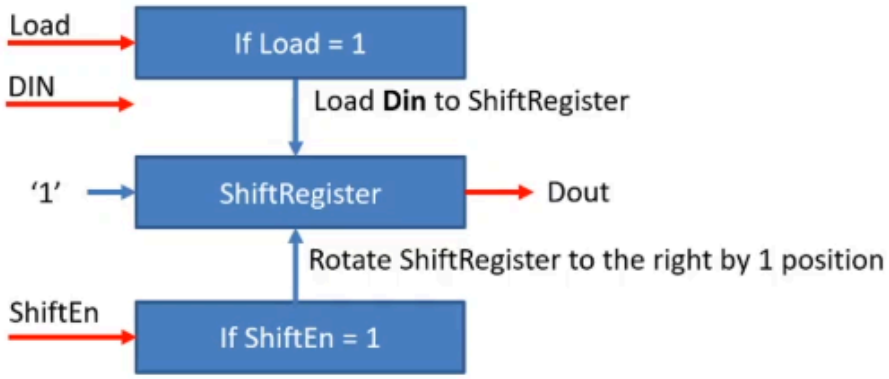
read this LSB to MSB

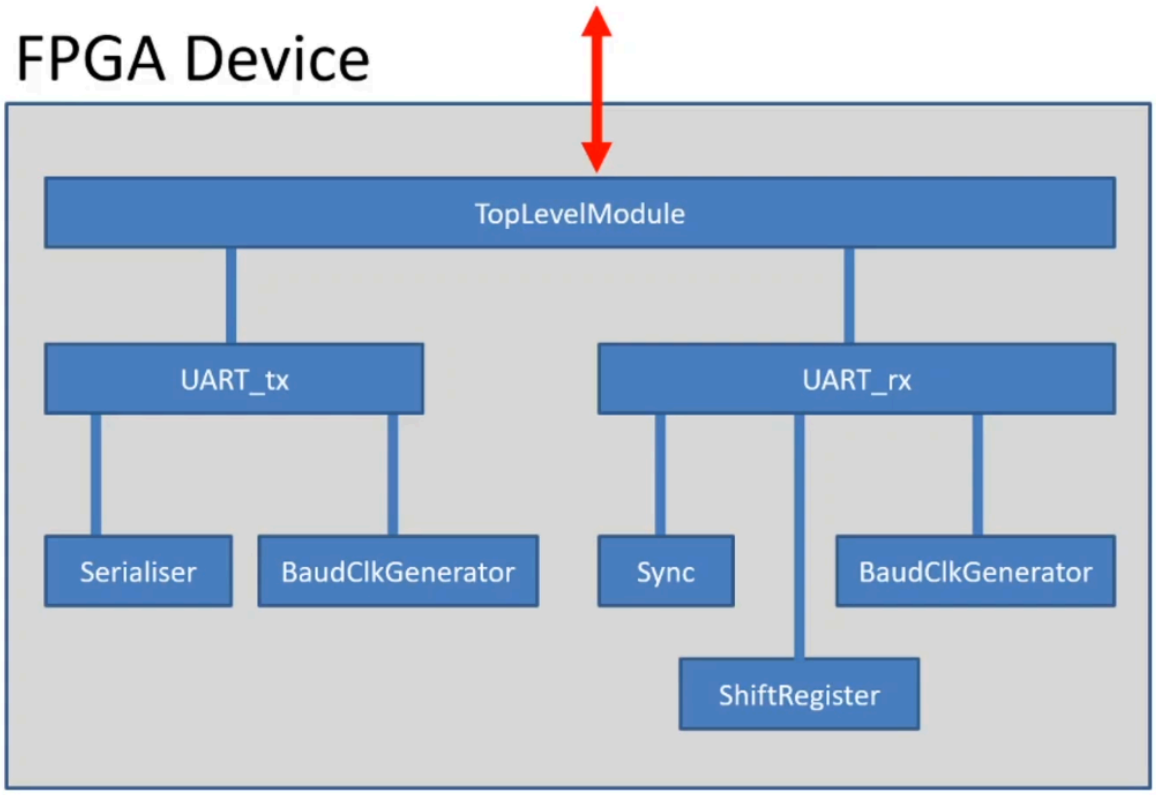


Buad Rates

- 115200 (Bit period = 1/115200 = 8.7us)
- 57600
- 38400
- 19200
- 14400
- 9600




Load
If Load = 1
DIN
Load Din to ShiftRegister
'1'
ShiftRegister
Dout
Rotate ShiftRegister to the right by 1 position
ShiftEn
If ShiftEn = 1


FPGA Device
TopLevelModule
UART_tx
UART_rx
Serialiser
BaudClkGenerator
Sync
BaudClkGenerator
ShiftRegister