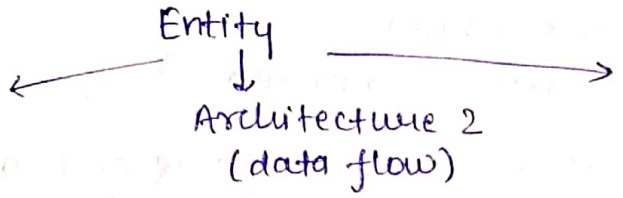
DSD NOTES

- HDL → Hardware description language → VHDL / Verilog
- Verilog → syntax based on C, systemc based on C++
- VHDL (~~Very~~ VHSIC HDL)
  ↳ Very high speed integrated circuit hardware description language

Netlist: In electronics design a netlist is the description of the connectivity on an electronic circuit. It consist of a list of the electronics component in a ckt & a list of nodes they are connected to

→ Features of VHDL
1. Support for concurrent statement
2. Library support
3. Sequential statement

→ VHDL Design Process

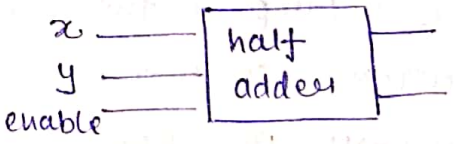


To describe an entity, there can be 4 types of architecture
i) structural ii) data flow iii) behavioural iv) Mixed

To describe an entity, VHDL provides 5 different types of primary construct called design units. They ~~can~~ are
i) Entity declaration iii) Configuration declaration
ii) Architecture body iv) Package body v) Package declaration

- Entity Declaration

```
ENTITY half adder IS
 PORT (x,y; enable :IN bit;
        carry ; result : out bit);
END half adder;
```




```
* entity < entity_name > is
  port ( < port_name > ; <mode>
        < type > ; ---
         );
  end < entity_name > ;
```

→ Full Adder

```
ENTITY FA IS
PORT (x,y,z ; IN bit;
      carry, sum ; OUT bit);
END FA;
```

- Port Declaration : Three parts of port declaration
  i) Name ii) Mode iii) Data type

* Port Mode : The port mode of the interface describes the direction of the data flow wrt the component.

→ 5 type of data flow are

IN : data flow in this port and can only be read
OUT : data flow out in this port & can be written only.
Buffer : similar to out, but it allows for internal feedback
INOUT : data flow can be in either dirn with any no. of sources allowed (implies a bus)
Linkage : data flow dirn is unknown

- Data Object : A data object holds the value of a specified type. Every data object belongs to work of 4 classes.
  i) Constant ii) Signals iii) Variables iv) Files
  Syntax : class object name : data type

→ Variables :

Syn: VARIABLE <variable_name> : <datatype_name> [:= <value>]

Eg. 1. VARIABLE opcode : BIT_VECTOR (3 DOWNTO 0) := "0000";
2. VARIABLE freq. : INTEGER ;

→ Constants : (improve the readability of code)

Syn : CONSTANT < constant_name > : <type_name > : = <value>;

Eg. CONSTANT PI : REAL : = 3.14

→ Signals :

Syn : SIGNAL <signalname > : <type_name > : [:= <value >] ;
for assigning

Eg. SIGNAL enable : BIT ;
SIGNAL output : bit_vector (3 downto 0)

→ Files : (sequence of value called file)

Syn : file file-name : file-type-name [[open mode] is string expression]
↳ class

The mode specifies whether the files is to be used as read only or write only or in the append mode.

Eg. type POSITIVE_FILE is file of POSITIVE ; -- declare a file of +ve no.
type integer File is File of INTEGER ;

Q Declare a data object of variable of type integer

Ans. VARIABLE AB : INTEGER

Data Type

Every data object in VHDL can hold a value that belongs to a set of values. This data type can be classified as

i) scalar type ii) composite type iii) File type iv) Access type v) Sub type

↳ Sub Type : It is a type with constants

type ~~number~~ NUMBER is ('0','1',---'9');
sub type MD is NUMBER range '3' to '7'.

- MD is the subtype whose base type is number and it has values 3,4,5,6,7. Sometime a constraint might not be defined. In that case it gives another name to an already existing type.

type NUMBER is ('0','1',... '9');
Sub type MD is NUMBER ;

↳ Scalar Type : In this values are ordered i.e., relational operators can be used.

There are 4 type of scalar type

1. Enumeration 2. Integer 3. Physical 4. Floating point
↓

Enumeration is a type which has a set of user defined values consisting of identifiers & literal characters.

type MVL is ('0','U','I','Z');
type OP is (LOAD, STORE, ADD, SOB, MUL, DIV);
Sub type ARITH_OP is OP range ADD to DIV;

Q Define a data object of type MVL

Ans. Signal SS : MVL ;

- STORE DIV : This statement is true because a value is always less than a value that appears to its right.

Values of an enumeration type are referred to as enumeration which is predefined Enumeration type.

i) character ii) Bit iii) Boolean iv) severity-level
v) FILE-open-kind vi) FILE-close-status

* Values belongings to the type character constitutes the 128 characters of ASCII character set

→ These values are called character literals and are always written b/w two singal quotes.

→ Bit has literals 0 and 1

→ Boolean has literal true & False

→ Type severity-level has the values NOTE, WARNING, ERROR and FAILURE.

→ File-open-kind has the values Read, write & append mode.

→ File-open-status has the values open, STATUS-error, NAME-error, Mode-Error.

STD_ULOGIC Type - an enumerated type

```
type STD_ULOGIC is (
    'U' → Uninital        'L' → weak 0
    'X' → forcing unknown 'H' → weak 1
    '0' → forcing 0       '-');→ don't care
    '1' → forcing 1
    'Z' → high impedence
    'W' → weak unknown.
```

x ——— x ——— x ——— x ——— x ——— x ——— x ——— x

- Integer data type : In which the set of values fall within a specified integer range.

Declare a integer type with range from 0 to 31

```
type value is range 0 to 31;
  constant V1 : value
```

Sub-type subtype MD IS value;

The Range is implementation dependent but it must atleast cover the values from $-2^{31}-1$ to $2^{31}-1$

↳ Values belonging to an integer type are - refered to as integer literals

- Floating Point Type : It has a set of values in a given range of real numbers.

type value is -2.3 to 2.3
variable V2 : values

subtype subtype M

↳ integer and floating point ~~has~~ numbers can also be written in a ways other than 10. The base value can be vary from 2 to 16.

base # basevalue #
$(1001)_2$ → 2 # 1001 #

↳ the only predefined floating type is REAL

- Physical Type : ↳ It contains values that represents measurement of some physical quantity, like time, length, voltage & current.
↳ Values of this type are expressed an integer multiple of base values
↳ Values of this type are called physical literals. Physical literals are written as an integer literal followed by the unit name.

Eg. type current is range 0 to 1E9
nA (base unit), uA = 1000 nA , mA = 1000 uA

Amp = 1000 mA
end units

SUBTYPE FILTER$ CURRENT IS CURRENT range 10uA to 5uA.

* Predefined Physical type :
↳ The only predefined physical type is TIME
↳ Its range is implementation dependent, must atleast be $(-2^{31}-1)$ to $(2^{31}-1)$.
↳ The declaration of type TIME appears in package STADARD

* composite Type : It represent collection of values
there are two composite type

i) array type	ii) record type
It represent a collection of values all belongings to a single type.	It represents collection of values that may belongs to same or different type

- **Array Type** : An object of an array type consist of elements that have the same type.

Eg. type ADDRESS_WORD is array (0 to 63) of BIT
type DATA_WORD is array (-1 downto 0) of STD ULOGIC.
signal ADDRESS BUS : ADDRESS_WORD ;

→ To access elements of array
ADDRESS_BUS (26) - refers to 27th element of ADDRESS-BUS

→ There are 2 predefined 1D-array type in the language
~~&R~~ STRING

* STRING is an array of characteristic while BIT_VECTOR is an array of bits

VARIABLE MESSAGE : STRING (1 to 17) := "Hello, VHDL World";
CONSTANT ADD_CODE : BIT_VECTOR : ('0','1','0','1','0');

- **RECORD Type** : An object of record type is composed of elements of same or diff. types.

```
type PIN_TYPE is range 0 to 10;
type MODULE is
        record
                SIZE : INTEGER RANGE 20 to 200;
                CRITICAL_DLY : TIME
                NO_INPUTS : PIN_TYPE
                NO_OUTPUTS : PIN_TYPE
        end record;
```

values can be assigned to a record type object

Variable NAND_COMP : MODULE :
NAND COMP is an object of record type MODULE
NAND_COMP : (50,20,ns,3,2);

- **Data types defined in standard package**

Type, bit, bit vector, boolean, character, file open kind, file open status, integer, natural, positive, real, severity level, string, time

- Operators

1. Assignment operator 2. Logical operator
3. Adding operator 4. Multiplying operator 5. Miscellaneous oper.

→ Assignment Operator :

1. Used to assign values to signals, variables and constant
   '<=' used to assign a value to a signal
   ':=' used to assign a value to variable, constant or must be the same size & type.
   '=>' used to assign value to individual vector element.

→ Logical Operators : The seven logical operators are AND, OR, NAND, are not associative ; NOR are not associative ; XOR, XNOR, NOR & highest precedence over the other.

→ Relational operators : =, /= inequality, <, <=, >, >=
The result type for all relational operation is always BOOLEAN(T or F)

Eg. BIT_VECTOR ('0','1','1') < BIT_VECTOR ('1','1','1')

type MVL is ('U','0','I','Z')
MVL ('U') < MVL ('Z')

→ Shift Operators

SLL → shift left logic (fill value is '0')
SRL → " right " [" " " '0')
SLA → " left arithmatic (" " is right hand bit)
SRA → " right " [" " is left " ")
rol → rotate left
ror → rotate right

Eg. 1100 SLL 1 yield 1000
1100 srl 2 " 0110
1100 sla 1 " 1000
1100 sra 2 " 1111
1100 rol 1 " 1001
1100 ror 2 " 0011

→ Adding Operators :

+ addition , − subtraction , & → concatenation

→ Multiplying Operator :

↳ * → multiplication

↳ mod → (modulus) − sign of 2nd operand

A mod B = A − B * N (for some integers)

↳ rem → (remainder) − sign of 1st operand

A rem B = A − [A/B] * B

→ Miscellaneous operators

↳ * → abs absolute value

↳ ** → exponentiation

↳ not → logical negative

left operand = integer or floating pt.

right operand = integer only.

	Operator class	Operator
Highest precedence ↑	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, −
	Adding	+, −, &
	Shift	SLL, SRL, SLA, SRA ROL, ROR
Lowest precedence	Relational	=, /=, <, <=, >, >=
	Logical	AND, OR, NAND, NOR, XOR, XNOR

• Evaluation Rules

1. Operators evaluated in order of precedence highest are evaluated first.
2. Operators of equal precedence are evaluated from left to right.
3. Deepest nested parenthesis are evaluated first.

∘ Behavioural Modeling

1. In this modelling style, the behaviour of the entity is expressed using sequentially executed, procedural type code like C or pascal.
2. A process statement which is concurrent statement is the primary mechanism used to describe the functionality of an entity in this modelling style.
3. A process is first entered at the start of simulation (actually during the initialization phase of simulation) at which time it is executed until it suspends because of a wait statement (wait statement are described later in this chapter) as a sensitivity list.

∘ Process statement

A process statement contains sequential statements that describes the functionality of a portion of an entity in sequential term.

[Process label] Process [(sensitivity list)]
[Process item declaration]
begin
sequential statement these are ⇒

Variable assignment statement
signal " "
wait " "

if statement
case statement
loop "
null "
exit "
next "
assertion "
procedure call statement
return statement
end process [process label];

→ A set of signal that the process is sensitive to is defined by the sensitivity list.

→ Each time an event occurs on any of the signal ~~for~~ in the sensitivity list the seq. statement within the process are executed in a sequential order.

The process then suspend after executing the last sequential statement & waits for another event to occur on a signal in the sensitivity list.

```
architecture AOI_SEQUENCIAL of AOI is
begin
process (A,B,C,D)
  variable TEMP1, TEMP2 : BIT ;
  begin
    TEMP 1 : = A and B ;  -- statement 1
    TEMP 2 : = C and D ;  -- statement 2
    TEMP 1 : = TEMP 1 or TEMP 2 ;  -- statement 3
    Z <= not TEMP 1 ;  -- statement 4
  end process ;
  end AOI_SEQUENTIAL ;
```

Variable assignment

signal assignment

* Variable Assignment statement

→ Variable can be declared & used inside a process statement. A variable is assigned a value using variable assignment statement that typical has the form.

→ Variable can also be declared outside the process or a sub-programme. These type of variable are refer to as shared variables. They can be read & updated by more than 1 process.

* Signal Assignment statement

→ signals are assigned value using a signal statement. The simplest form of a signal assignment statement is

signal_object <= expression [after delay value]

→ When a single assignment is executed the value of the expression is computed & this value is scheduled to be assigned to the signal after the specified delay.

→ If no after clause is specified, the delay is assumed to be a default delta delay.

Eg.
```
process (a,b)  -- if event occurs at a or b at threshold.
  T
variable V1, V2, V3 : integer
begin
V1 := b + V3 ;  -- value is assigned at T
Z <= V2 + V1 ;  -- signal Z is scheduled at T
but old value of Z is used
```

end process ; -- when simulation time advances to T + ΔT, the signal Z gets its new value.

- **Delta Delay**

→ a very small delay (infinitesinally small)
→ it doesn't respond to actual delay
→ The actual simulation time doesn't advance
→ This delay models hardware where a minimal amt of time is required for a change to occur at simulation time during simu.
→ The event always occurs at simulation time
+
an integral multiple of Δ (delta) delay.

- **Variable assignment V/s signal assignment**

```
signal A,Z : INTEGER
P2 : process (A)      -- P2 is label for process
variable V1,V2 : INTEGER;
 begin
 V1 := A - V2;              -- statement 1
  Z <= -V1;                 --    "     2
 V2 := Z + V1 * 2;          --    "     3
 end process P2;
```

If an event occurs on signal A at time T execution of statement 1 causes V1 to get a value at time t. signal Z is then scheduled to get a value at time T + Δ so statement 3 would be using old value of initial.

- **Wait Statement**

The wait statement provides an alternate way to suspend the execution of process.

→ 3 basic form of wait statement
  - wait on sensitivity list ;
  - wait until boolean expression ;
  - wait for time - expression ;

- combined form in a single wait statement

→ wait on sensitivity list until boolean expression for time expression

Eg. wait on A, B, C ; -- statement 1
wait until (A = B) ; -- " 2
wait for 10 ns ; -- " 3
wait on CLOCK for 20ns ; -- " 4

↳ In statement 1, the execution of wait statement causes the process to suspend and then it wait for an event to occur on signal A, B, C.

↳ The execution of statement 4 causes the process to suspend and then it waits for an event to occur on clock signal for 20 ns.

wait until wait until CLK = '1' ;

when the wait statement is executed the process must first suspend then wait for an event to occur on clock & then check the value of boolean expression. It doesn't matter if the boolean is true. At the time the wait state is executed. If the CLK is already 1 when the statement execute, the boolean expression is not checked until another event occurs on signal clock.

wait until TRUE ;

- Process with NO sensitivity list

```
process          -- no sensitivity list
variable TEMP1, TEMP2 : BIT;
begin
TEMP1 : = A and B ;
TEMP 2 : = C and D;
TEMP 1 : = TEMP 1 or TEMP 2;
Z < = not TEMP 1;
wait on A, B, C, D ;  -- Replace the sensitivity list
end
process.
```

→ If statement

↳ An If statement selects a seq. of statement of statement for execution based on the value of a condition.

↳ The condition can be any expression that evaluates to a boolean value.

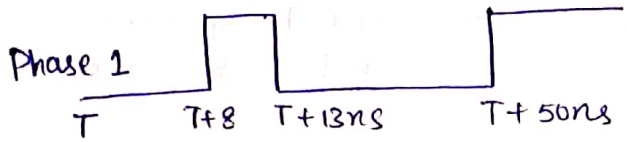
Syntax : The general form of an if statement is

```
if boolean expression then
    sequential statement
[else if boolean expression then
    seq - statement ]
[else - else clause
    seq. statement ]
end if
```

- **Creating signal waveform**

→ Phase 1 <= '0', '1' after 13 ns, '1' after 50 ns.

→ When this signal assignment statement is executed say at time T. It causes four values to be scheduled for signal phase 1. The value '0' is scheduled to be assigned at time T + A . 1 at ~~the~~ T + 8ns, '0' at T + 13ns & '1' at T + 50ns.



signal object <= waveform elements

- **Signal Drivers**

Every signal assignment in a process element creates a driver for that particular signal. The driver of a signal holds its current value & all its feature value as a sequence of 1 or more transaction where each transaction identifies the values to appear on the signal along with the time at which the value is to be appeared.

→ All the transaction on drivers are ordered in increasing order of time.

What if there is more than 1 assignment to the same signal within a process?

```
process
begin
..
RESET <= 3 after 5ns, 21 after 10ns, 14 after 17ns
end process;
```

RESET ← | Current | — | 3 @ T+5ns | 21 @ T+10ns | 14 @ T+17ns |

* When simulation time advances to T+5ns, the first transactions is deleted from the driver & reset gets the value of 3.

* Effect of Transport Delay on signal driver

```
signal Rx-DATA : NATURAL;
...
process
begin
Rx-DATA <= transport 11 after 10ns;
Rx-DATA <= transport 20 after 22ns;
Rx-DATA <= transport 35 after 18ns;
end process;
```

Rx_data ← | Current | 11 @T+10ns | 20 @T+22ns |

- After the 2nd signal assignment is executed, the transection 20 @T+22ns is appended to the driver, since the delay of this transaction is larger than the delay of the appending transection of deriver.
- When the 3rd signal assignment statement is executed, the new transection causes the 20 @ T+22ns to be deleted & the new transection is appended to the driver.

Rx-DATA — | Current | 11 @T+10ns | 35@T+18ns |

- In general, a new transection will ~~be~~ delete all transaction on a driver that are to occur later on that of new transection.

Rule 1. If the delay time of the new transection is greater than those of all the transaction already present on the driver, then the new transactions is added at the end of the driver.

2. If the delay time of the new transaction is earlier than or equal to one or more transection on the driver, then these transection are deleted from the driver & the new transaction is added at the end of the driver.

* Effect of Inertial Delay on signal Drivers

```
Process
begin
Tx-DATA <= 11 after 10ns;
Tx-DATA <= 22 after 20ns;
Tx-DATA <= 33 after 35ns
wait; -- suspends indefinitely
end process;
```

→ when inertial delays are used both the signal value being assigned and the delay value affect the deletion and addition of transaction.

Rules 1. If the delay of new trans. is earlier than an existing tran. the later is deleted ← new one is added at the end of the driver regardless of the signal values of true transaction.

2. If the delay of new tran. is greater than an already existing trans. the signal values of the two transaction are compared if they are same they are added at the end of driver. If not, the existing one is deleted before adding the new transaction.

- Deletion occurs for every existing trans. with a signal value that is different from the new trans.

Tx-DATA ← | current | 22 @ 20ns |

Tx- DATA ← | current | 33 @ 15ns |

- The transaction 11 @ T+10ns, first get added to the driver, the 2nd tranc. 22 @ 20ns causes the existing transaction to be deleted with this the signal value i.e., 22 of new transc. is assigned. The execution of 3rd signal assignment cause the transc. to be deleted from the driver since the delay of new transc. is less than the existing transaction.

* Summary of rules for adding a new transaction when internal delay is used is

1. All transactions on a driver that are scheduled to occur at or after the delay of the new transaction are deleted (as in the transport class)
2. If the value of new transaction is same as the value of the transc. on the driver, the new transc. is added to the driver. If the value of new transc is different from the value of one or more transc. on the driver these transc. are deleted from the driver.

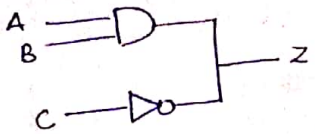
* Data Flow Model

→ A dataflow model specifies the functionality of the entity without explicity specifying its structure

→ data flow model for a 2-input or gate

```
entity OR2 is
port(signal A,B: IN Bit ; signal Z: out bit);
end OR2;
architecture OR2 of OR2 is
begin
Z <= A or B after 9ns
end OR2;
```

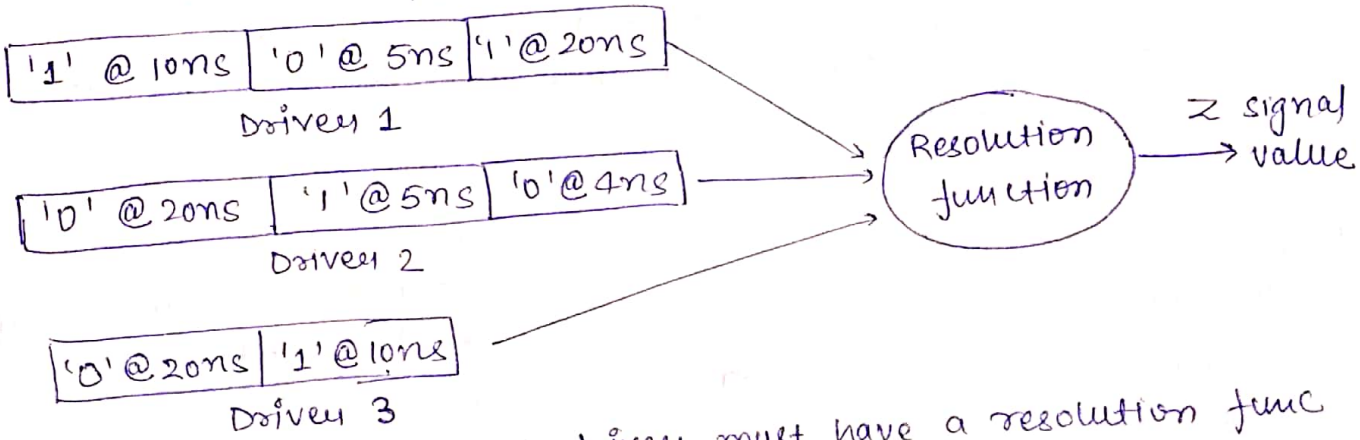
- **Multiple Drivers**




```
entity TWO_OR_EXAMPLE is
port (A,B,C : in Bit ; Z : out Bit);
end TWO_OR_Example ;
architecture TWO_OR_EX_BEH of TWO_OR_EXAMPLE IS
begin
    Z <= A and B after 10ns;
    Z <= not C after 5ns;
    end ;
```

The value of Z would be determined by using a user defined resolution func that considers the values of both the drivers of Z & determines the effective value

```
begin
    Z <= '1' after 2ns, '0' after 5ns, '1' after 10ns;
    Z <= '0' after 4ns, '1' after 5ns, '0' after 20ns;
    Z <= '1' after 10ns, '0' after 20ns;
    end NO_entity;
```



A signal with more than 1 drivers must have a resolution func associated with it otherwise it is in error such a signal is refers to as resolved signal.

A resolution func is associated with a signal by specifying its name in the signal declaration

- **Resolution Function**

→ VHDL uses a resolution func to determine the actual O/P.
→ For a multiple driven signal, value of all drivers are resolved together to create a single value for the signal.
→ This is known as "Resolution Func"
Examines the value of all the drivers.

The TWO-OR-EXAMPLE entity previously described is

```
port (A,B,C : in bit ; out wired-OR bit);
```

Eg

```
function WIRED-OR( INPUTS : BIT-VECTOR ) return BIT is
begin
    for J in INPUTS RANGE loop
    if INPUTS(J) = '1' then
    return '1';
    end if;
    end loop;
    return '0';
    end WIRED-OR;
```

- Block Statements

It represent a way of partioning the code
It is a concurrent statement, it can be used for 3 purposes.

i) to disable signal drivers by using guards
ii) to limit scope of signals
iii) to represent certain part of the design

There are two kinds of block statement

i) Simple

Syntax:
```
Label : BLOCK
[declarative part]
BEGIN
(concurrent statements)
END BLOCK label ;
```

ii) Guarded

Syntax:
```
BLOCK(guarded expressing)
[declarative part]
BEGIN
(concurrent guarded & unguarded stat.)
END BLOCK label ;
```

→ It includes additional expn called guarded expression.
(i.e., like if else statement)

* If a guard expression appears in a block statement there is an implicit ~~called~~ expression called guard of type boolean declared within the block.

Eg

```
ARCHITECTURE latch of latch IS
BEGIN
    b1 : BLOCK ( clk = '1');
    BEGIN
      q <= GUARDED d;
    END BLOCK b1;
    END latch;
```

In eg. clk = 1 is guarded expression while q = guarded in a guarded statement ∴ q = d will only occur if clk = 1

- std_logic is a type of unsolved logic, that mean a signal can be driven by 2 inputs.
- std_ulogic is a type of unsolved logic, that mean a signal cannot be driven by 2 inputs.

Q1. Write VHDL code for half adder using data flow modelling?

```
ENTITY HA IS
PORT (x, y : IN bit;
      sum, carry : out bit);
END HA IS;
ARCHITECTURE HA of HA
begin
    SUM <= x XOR y;
    carry <= x and y;
end HA;
```

↳ Full Adder

```
ENTITY FA IS
PORT (a,b,c : IN bit;
      S, carry : OUT bit);
END FA;
ARCHITECTURE FF of FA
BEGIN
    S<= a XOR b XOR c;
    carry <= (a and b) OR (b and c) OR (c and a);
END FF;
```

↳ VHDL code for Half subtractor

```
ENTITY HS IS;
PORT ( a,b : IN bit;
       d, borrow : out bit);

END HS;
ARCHITECTURE HSS OF HS IS
BEGIN
    d <= a XOR b;
    borrow <= (not a) and b;
END HSS;
```

- **Structure Modelling** : An entity is modeled as a set of components connected by ~~using~~ signals that is as a netlist.

→ **Component Declaration**

i) A component instantiated in a structural description must first be declared using a component declaration.

ii) A component declaration declares the name & the interface of a component.

iii) Must be declared in the declarative part of the architecture.

The syntax of a simple form of component declaration is

```
component component_name ;
port ( list_of_interface_ports );
end components ;
```

- **Component Instantiation**

The instance of a component in the entity is described as follow

```
< instance_name > : <comp-name>
 port map ( < association list >);
```

Eg. 3 i/p, AND gate by instantiation of 2 i/p AND gate

```
entity and3gate is
 port ( o : out std_logic ;
        i1 : "   "   " ;
        i2 : "   "   " ;
        i3 : "   "   "
      );
end and3gate ;
architecture arc_and3gate of and3gate is
 component andgate is
      port ( c : out std_logic;
             a : IN   " ;
             b : IN   " );
end component ;
signal temp1 : std_logic ;
begin
     U1 : andgate
     portmap ( temp1, i1, i2);
     U2 : andgate
     portmap ( o , temp1, i3 );
end arc_and3gate ;
```

- **components** : The connections in the association list may be declared in 2 ways
  i) Positional Association ii) Named Associati

↳ **Positional Association**

i) The name of the signals in the higher level entity are written in the port map statement & each signal is mapped with port in the component declaration in the order in which they are declared.

ii) The order in which the signals are written has to match with the order of the parts of the components.

↳ **Named Association**

The signals of the higher level entity are connected to the part of components by explicity stating the connections.

eg. The 3 i/p AND gate may be written as

```
architecture arc_and 3 gate of and 3gate is
component andgate is
 port ( c : out std_logic;
        a : IN  std_logic;
        b : IN  std_logic);
 end component;
signal temp 1 : std_logic;
 begin
    U1 : andgate
      portmap ( c => temp1;
                b => i2;
                a => i1);
     42 : and gate
      portmap ( a => temp 1;
                b => i3;
                c => o);
     end arc_and 3gate;
```

Hence the order in which the code, the signal are written is not important.

* VHDL code for half adder using structural modelling

```
Component andgate is
    port ( x,y: in std_logic;
           z,w : out std_logic);
end hal
```

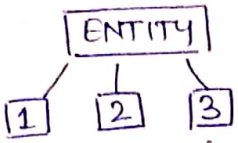
* Configuration : A configuration is used to bind (i) an architecture with its entity declaration

(ii) to bind a component with an entity

→ The VHDL language provides 2 ways of binding

i) by using a configuration specification

ii) by using a configuration ~~specification~~ declaration



syntax

```
for list_of_comp_label : component name
use binding indication;
```

```
entity entity-name [(ciruit-name)]
  [generic map(generic-association)]
  [portmap (port association)]
```

A confi. specification is used to bind components intantisation to specific entity that ~~eeeee~~ are stored in desired library. The specification appears in the declaration part of the archit. or the block in which the component are intantiated.

Q VHDL code for priority encoder using behavioural modelling

```
ENTITY Priority_Encoder IS
  PORT ( w : IN std_logic_vector (3 downto 0);
         y : out  "      "  (1 downto 0);
         z : out  std_logic );
END Priority_Encoder
Architecture Behaviour of priority is
BEGIN
    process (w)
    BEGIN
      if w(3) = '1', Then
            y <= '11';
      else w(2) = '1', then
            y <= '10';
      else w(1) = '1', then
            y <= '01';
      else  y <= '00';
      END if;
      END Process;
  z <= '0' when w = 0000
           Else '1';
END Behaviour;
```

Q1. VHDL code for full subtractor using data flow model

→
```
ENTITY FS,IS ;
Port ( A,B,C : IN BIT ;
       diff , borrow : out BIT);
END FS;
Architecture FSS of FS IS
begin
      diff <= A XOR B XOR C ;
      borrow <= (not A ) and B OR ( b and C ) OR ( C and (not A));
END FSS;
```

Q2. VHDL code for binary to gray code converter using data flow model.

→
```
ENTITY GC IS :
port ( b0,b1,b2,b3 : IN bit ;
       g0,g1,g2,g3 : OUT bit);
END GC;
Architecture BG of GC
begin
      g3 <= b3
      g2 <= b3 XOR b2
      g1 <= b2 XOR b1
      g0 <= b1 XOR b0
END BG;
```

o For Full Adder

```
Library HS-Lib , CMOS-Lib
entity FULL-Adder IS
    port( A,B,CIN : In Bit ; SUM,cout : out Bit);
end FULL-Adder ;
Architecture FA-str of FULL-Adder is
  for X1 , X2 : XOR2
     use entity work.XOR(XOR_2BEH); -- Binding the entity with
                                       more than one instantiation
                                       of a component
  for A3 : AND2
     use entity HS-Lib AND2HS (AND2STR)
     port map ( HS-B => A1 , HS-Z => Z , HS-A => A0); -- Binding the
                                       entity with a single instantiation
                                       of a component
```

```
for all : OR2
    use entity CMOS_Lib.OR2CMOS(OR2STR) -- Binding the entity with all
                                           instantiation of OR2 component.
for others : AND2
    use entity WORK.A_Gate(A_Gate_Body)
  portmap(A0, A1, Z); -- Binding the entity will all unbound instantiation
                         of AND 2 component.
signal S1,S2,S4,S5 : BIT;
begin
        X1 : XOR 2  port map (A,B,S1);
        X2 : XOR 2  port map (S1,CIN,SUM);
        A1 : AND 2  port map (S2,A,B);
        A2 : AND 2  port map (S3,B,CIN);
        A3 : AND2   portmap (S4,A,B,CIN);
        O1 : OR2    portmap (S1,S3,S5);
        O2 : OR2    portmap (S4,S5,COUT);
end FA_STR;
```

- First specification statement indicates that instances X_1 & X_2 of component XOR or formed to the entity, represented by the entity architecture pair. ~~XOR2 & XORBEH~~
- The second specification binds the end to component label A3 to the entity represented by entity architecture.

- Configuration Declaration

  used to select one of the many architecture that an entity may have.

Syntax :

```
configuration confi_name entity_name is
    for architecture_name
      for instantiation : component_name
          use library_name.entity_name(archi_name);
      end for;
    end for;
end confi_name ;
```

- It is a seperate design unit used for binding of component i.e, binding can be done after the architecture has been written.

- Generic Statement : It allows to pass ~~some~~ certain parameters into a design description from its environment.

Eg Syntax: generic (
constant_name : type [:= value]
constant_name : type [:= value]
);

Q. Entity Priority-Encoder is
signal (w0, w1, w2, w3 : IN std_logic;
y0, y1, z : OUT std_logic);

architecture Encoder of Priority_Encoder IS
begin
process (w0, w1, w2, w3)
begin
if w3 = '1' then
y <= '11';
elseif w2 = '1' then
y <= '10';
elseif w1 = '1' then
y <= '01';
else y <= '00';
end if;
end process;
z <= '0' when w = '0000' else '1';
end behaviour;

- Operator Overloading :
  - Allow us to perform arithmatic and boolean func on non built in data types.
  - Declaring ieee libraries to ".all" will work fine

Eg (This is inside the ieee library)

FUNCTION "+" (l : std_logic_vector ; r : std_logic_vector) return std_logic_vector ;
FUNCTION "+" (l : std_logic_vector ; r : integer) " " ;
FUNCTION "+" (l : " " ; r : std_logic) " " ;
FUNCTION "+" (l : integer ; r : std_logic_vector) " " ;
FUNCTION "+" (l : std_logic ; r : std_logic_vector) " " ;

- Signal Assignment : ↳ conditional
  ↳ Simple ~~Already~~ ↳ selected

* Conditional : FORMAT

```
<signal_name> <= <signal/value> when <condition 1> else
                <signal/value> when <condition 2> else
                <signal/value>;
```

* Selected : FORMAT

```
with <expression> select
   <signal_name> <= <signal/value> when <condition 1>
                    !              !            !
                    <signal/value> when others;
```

Eg.
```
with sel select
    q <= a when "00";
         b when "01";
         s when others;
```

→ 'CASE' Statement :

Consider multiple ~~format~~ statements

↳ FORMAT

```
CASE <exprn> IS
    WHEN <condition 1> =>
        {seq. statement}
    WHEN <condition 2>
        {seq. statements}
        !
    WHEN OTHERS => --{optional}
        {seq. statements}
END CASE;
```

Eg
```
PROCESS (sel, a, b, c, d)
BEGIN
   CASE sel IS
    WHEN "00" =>
       q <= a;
    WHEN "01" =>
       q <= b;
    WHEN "10" =>
       q <= c;
    WHEN OTHERS =>
       q <= d;
   END CASE;
END PROCESS;
```

- Sequential statement must go inside an explicit process.

→ Sequential Loops [NEXT and EXIT can be used in any loop type]

- Loop Statement
- loop infinitity when EXIT statement exists.

Eg
```
[loop_label] LOOP   --seq. statement
    ~~NEXT~~ NEXT loop_label WHEN --;
    EXIT loop_label WHEN --;
END LOOP;
```

↳ WHILE loop

- Conditional test to end loop
- condition 'true' → loop run number
- condition 'false' → loop stops

Eg WHILE <Condition> LOOP
-- seq. statement
END LOOP

↳ FOR LOOP

Eg. FOR <Identifier> IN <range> LOOP
-- seq. statements
END LOOP ;

- Applications of Generic
  - can be used as a static value when needed.
  - for modeling ranges of subtypes
    subtype ALUBUS is integer range top downto 0;
    -- top is a generic.

- Generic and Constants diff.

Generic	Constant
↳ Are specified in entities	• Are specified in architecture
↳ change in value of a generic affects all architecture.	• change in value of a constt. will be limited to architecture.

Syntax

-- component declaration

```
Component component_name is
    generic (list of generics);
    ports ( list_of_interface_ports );
end component;
```

-- component instantiation

```
component_label : component_name
generic map (generic_asso_list);
port map ( port_asso_list );
```

Rules

For each unbound component instance

- an entity that is visible and has the same name as that of the component is used to bind to the instance, if no such entity exists, it is an error.
- the entity's most recently analyzed architecture is used
- port and generic are matched by names.

- Sub programs

Two kinds of subprograms

i) Function ii) Procedure

1. Function : These are usually used for computing a single value. It executes in simulation time.

2. Procedure : These are used to partition large behavioural discription, Procedure can return zero or more values.

It may or may not execute in zero simulation time.

Syntax of subprogram

```
subprogram - specification is
    subprogram - item - declaration
    begin
      subprogram - statements  -- same or seq. statement
    end [subprogram_name];
```

• The subprogram specification specifies the name of a subprogram and defines its interface i.e. it defines the formal parameters name, their class, their type and mode.

- Return Statement
  1. Sequential statement
  2. Special statement that is allowed only within subprogram.

     Syntax: return {expression};
  3. Causes the program to terminate
  4. Every subprogram must have it to return to the calling program.
  5. For procedure, objects of mode out and in out return their value to the calling program.

- Function
  1. Return single value
  2. sequential algo.
  3. Value returned using return statement

Syntax : (procedure)

```
- procedure <procedure_name> (parameter list) is
      declarations;
      begin
        statements;
    end procedure procedure_name;
```

Syntax : (function)

```
- function <function_name> (parameter list) return <return_type> is
    declaration;                  | end function function_name;
    begin
      statements;                 |
```

→ Parameter list specifies

- class definition (signal, var., constt.)
- name of parameter
- mode of the parameter (in, out, inout)
- subtype indication
- optional initial value.

Eg

```
FUNCTION Largest ( TOTAL-NO : INTER ; SET : Pattern)
RETURN real is
        -- pattern is defined to be a type of 1-D array
        -- floating point value, elsewhere
    variable RETURN-VALUE : REAL := 0.0 ;
begin
    for K in 1 to Total_no loop
        if SET(K) > RETURN-VALUE then
                    RETURN-VALUE := SET (K);
        end if ;
    end loop;
return RETURN-VALUE ;
end LARGEST ;
```

→ Function specification (subprogram – specification)

syntax

```
[pure|impure] function function-name (parameter-list)
                        -- formal parameters
        return return-type ;
```

→ Impure function : 1) It return different values each time, it is called with the same set of values.

2) If neither is specified then function is by default pure func.

→ Pure function : 1) The one which return the same value each time the function is called with the same set of actual parameters

2) The mode which allowed for parameter is mode in

3) Only constants and signal objects can be passed as parameter

4) Default object is 'constant'.

→ Function call :

Syntax function-name (list_of_actual-values)

↳ Function call in an expression

Eg SUM := SUM + LARGEST (MAX_COINS , COLLECTION);

↳ Actual value may be associated by

1. Position 2. named association

↳ Named association is

LARGEST (SE => COLLECTION , TOTAL_NO => MAX-COINS)

→ Procedure 1. Allows decomposition of large behaviours into modular sections

2. Can return mor than one value

3. parameter used mode out and inout

Syntax : Procedure procedure_name (parameter_list)

4. Specifies list of formal parameters for the procedure

5. Parameter may be constt , variable or signal and then mode can be in, out or inout

6. If the object is not specified then the object is by default constant if the parameter is of mode in else if it is variable if the mode is of out or inout.

→ Procedure Body :

```
type OP_CODE is (ADD, SUB , MUL , DIV, LT, LE, LG);
procedure ARITH-UNIT (A,B : IN INTEGER;
                      OP : IN OP-CODE;
                      Z : OUT INTEGER ; ZCOMP : OUT BOOLEAN);
begin
  case op is
      when ADD => Z := A+B;
      when SUB => Z := A-B;
      when MUL => Z := A*B;
      when DIV => Z := A/B;
      when LT => Z := A< B;
      when LE => Z := A<=B;
      when LG => Z := A>B;
  end case;
end ARITH-UNIT;
```

→ Procedure call : Syntax procedure_name (list_of_actual_parameters);

1. Actual parameters specify the expression that are to be formed into the procedure and the name of objects that are to receive the completed values from the procedure.
2. Actual parameters may be possitional association or named association

```
ARITH_UNIT (D1, D2, ADD, SUM, COMP) ; — positional association
ARITH-UNIT ( Z => SUM, B => D2, A => D1
             OP => ADD, ZCOMP => COMP); -- Named association
```

3. Procedure are involved by procedure call.
4. Procedure call can either be sequential statement based on where the procedure call is present.
5. If the call is inside a process statement or inside another subprogram then it is a seq. procedure call statement, else it is a concurrent procedure call statement.

○ Diffs b/w Procedure call & Function call

1) Procedure body can have a wait statement while a func can't. Function are used to compute value that are available instantan.
2) Func can't be made to wait, eg it can't call a procedure with wait state.
3) A process that calls a procedure with a wait state. can't have a sensitivity list.

○ Declaration : 1) describes the subprogram name and list of parameters list of without describing the internal behaviour of the subprogram i.e, it describes the interface for the subprogram.

Syntax : subprogram_specification;

```
procedure ARITH_UNT (A,B : IN INTEGER);
function VRISE ( signal CLOCK_NAME : BIT ) return BOOLEAN;
```

2) A subprogram body may appear in the declaration send to the block in which call is made.
3) This is not convenient if the subprogram is to be stored by more entities
4) In such case the subprogram can be described in one place. probably packing bodies and then in the package declaration the corresponding subprogram declaration is specified.

○ Overloading :

↳ Subprogram Overloading : It is convenient to process two or more subprogram with the same name.

Eg.

```
function COUNT (ORANGES : INTEGER ) return INTEGERS;
function COUNT ( APPLES : BIT) return INTEGER;
COUNT (20)
```

- Refers to 1st function since 20 INTEGER type
  COUNT('1')
- Refers to 2nd func since '1' in BIT type
- Function bodies are written to define the behaviour if overloaded.

Eg function declaration for such function bodies

```
begin MVL is ('U','0','1','Z');
  function "and" (L,R : MVL) return MVL;
  function "or" (L,R : MVL) return MVL;
  function "not" (R : MVL) return MVL;
```

→ Operater can be called using two different types of notations

1. Standard operation notation
2. Standard func call notation

Eg
```
A <= '2' or '1';
B <= "or" ('0','2');
```

→ Literal Overloading : If we use same literal in two different enumeration type declaration then it is refered to as literal overloading.

Eg
```
type L is ('0','1','Z','U');
type L1 is ('0','Z');
variable a : L;
variable b : L1;
```

- The literal '0' & 'Z' are overloaded because these two literals are present in both the datatypes 'L' & 'L1'.
- It will refer to the first variable defined and encountered.

- Packages : • collection of commonly used subprogram, data types, constants, attributes and components.
  - Allows designer to use shared constant or function or data type.
  - 'USE' statement is used to access the package.
  - Packages are stored in libraries for convience.

Syntax
```
Package < Package_declaration > is
  { set of declaration }
  end package_name ;
```

- Can be stored by many designed units.
- set of declaration such as subprogram, type, constant, signal, variable etc.

Eg
```
Package 'SYNTM-PACK IS
  Constant L2H : TIME := 20ns;
  type ALU-OP is (ADD, SUB);
  attribute pipeline : = BOOLEAN;
  component NAND2
    port (A,B. : IN MVL ; C : out MVL);
  end component
end package ;
```

→ If this is not present here then it is known as differed constt.

- Item declared in package declaration can be used by other design units by library and use classes.

```
use WORK SYNTH-PACK all;
package PROG-PACK is
```

Whenever these 3 are present then a body must be declared →

```
constant PR-TIME : TIME;
function
procedure LOAD (signal ADD-NAME)
```

- Package Body :

```
Package body package_name is
  declaration;
  sub program body;
  end package-name;
```

- Similar as architecture
- package declaration can have only one package body with same name, unlike the entity declaration.
- Contains hidden details
- Complete constant declaration for defered constant
- It used to store private declaration that should not be visible.
- If package doesn't have function, procedure then body declaration is not required.

- VISIBILITY : * IMPLICIT

1) Architecture body implicitly inherits all declaration in the entity.
2) Package body implicitly inherits all items declared in the program declaration when the package name is same as package declaration

* EXPLICIT

1) Visibility of item declared in other design units can be achieved using contact classes.
2) Two contact classes
   - Library classes
   - Use classes

→ Design file includes
- Library
- Use

(Design file can have multiple design units)

→ Design Unit Includes
- entity
- architecture
- configuration

→ Library Classes

- Formal

  library name-of-library ;

- STD & WORK library are implicitly declared in design unit

  library < library-name > ;

  use < lib-name >.< package-name >. all ;

- Implementation dependent storage facility for furiously analyzed design unit
- STD package design 5 data types :
  ↳ bit ↳ bit-vector ↳ character ↳ time ↳ integer
- STD_logic_1164 include 4 data types :
  ↳ std_logic ↳ std_ulogic ↳ std_logic_vector ↳ std_ulogic_vector

→ Use classes

Format

use library-name . primary_unit_name . item ;

→ ATTRIBUTES

- Can be used for modeling hardware.
- provides additional information about an object (signals, arrays, types) that may not be directly wanted to the value that object carries.
- Can be broadly classified as :
  ↳ Predefined - as a part of 1076 std ↳ Under defined

↳ Predefined Attributes

- Value kind attributes (left, Right, Hight, low, length, assending)

'left : return left most value of type subtype

```
type bit_array is array (1 to 5) of bit;
variable L : integer := bit_array 'left';
    -- L taken left most bound i.e., 1.
```

'high : return upper bound of type subtype

```
type bit_array is array (-15 to 15) of bit;
variable i's : integer := bit_array 'high';
    -- it takes the value of 15
```

'length : return the length of an array

```
type bit_array is array (0 to 31) of bit;
variable len : integer := bit_array 'length';
```

'assending : return a Boolean true value of the type or subtype which is declared with an assending range

Eg. type ase-array is array (0 to 31) of bit;
type dise-array is array (36 downto 4) of bit;
variable A1 : boolean := ase-array 'assending';
variable A2 : boolean := dise-array 'assending';

• NOTE : Value kind attributes return an explicit value and can applied to a type or subtype.

→ Predefined Attributes Poz, Val, Succ, Pred, left of, Right of

- poz (a) : return the position number of 'a'
  type state-type is (init, hold, store, read,);
  variable p : integer := state-type 'pos (mod);
- val (a) : return the type of value at position 'a'
  type state-type is (init, hold, strobe, read)
  variable v : integer := state-type 'val(3)';
- succ(a) : return next value of 'a'

→ CODE for ALU

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ENTITY ALU IS
PORT ( S0,S1,S2 : IN bit;
       A,B : IN std_logic );
END ENTITY
ARCHITECTURE Arith of ALU is
BEGIN
PROCESS ( inp_a, inp_b, sel)
BEGIN
CASE sel IS
  when "000" => out_alu <= inp_a + inp_b;
  when "001" => out_alu <= inp_a - inp_b;
  when "010" => out_alu <= inp_a - 1;
  when "011" => out_alu <= inp_a + 1;
  when "100" => out_alu <= inp_a and inp_b;
  when "101" => out_alu <= inp_a or inp_b;
  when "110" => out_alu <= not inp_a;
  when "111" => out_alu <= inp_a xor inp_b;
END CASE;
END PROCESS;
END ARCHITECTURE;
```

```
PORT ( inp_a : in signed (3 downto 0);
       inp_b : in signed (3 downto 0);
       sel : in std_logic_vector (2 downto 0);
       out_alu : out signed (3 downto 0));
```

⇒ GENERATE STATEMENTS

- can be conditionally selected or replicated during the elaboration phase using the generate statement.

- Two form
  - using the for-generation scheme, concurrent statement can be replicated a predefined number of types.
  - using if-generation scheme, concurrent statement can be conditionally selected for ~~selection~~ execution.

Syntax

```
Label : for parameter in range generation
    [generate declaration begin]
    concurrent-statements
    end generator [label];
```

```
Label : if condition generator
    [generate declaration
        begin]
    concurrent-statement
    end generate [label];
```

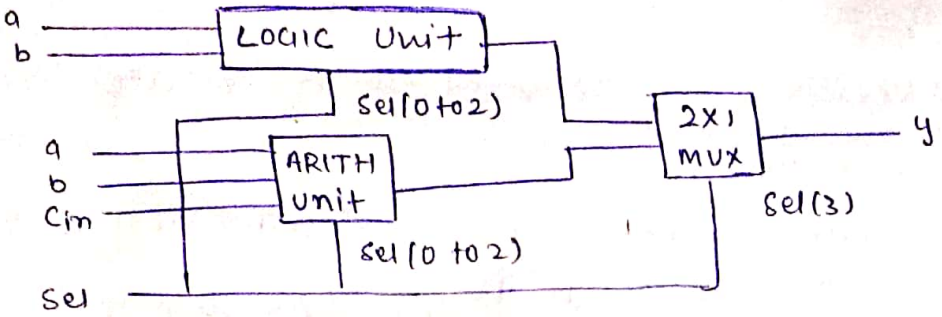
Eg.

```
ENTITY FULL-ADD4 IS
PORT (A,B : IN BIT_VECTOR (3 downto 0);
      CIN : IN BIT;
      SUM : OUT BIT-VECTOR (3 downto 0);
      COUT : OUT BIT);
END ENTITY;
ARCHITECTURE FOR-GENERATE OF FULL-ADD4 IS
  COMPONENT FULL-ADDER
      PORT (A,B,C : IN BIT
            COUT SUM : OUT BIT);
  END COMPONENT;
  SIGNAL CAR : BIT-VECTOR (4 downto 0);
BEGIN
  CAR(0) <= CIN;
  GK : FOR K IN 3 DOWNTO 0 GENERATE
  FA : FULL-ADDER PORTMAP (CAR(K), A(K), B(K), CAR(K+1), SUM(K));
  END GENERATE GK;
  COUT <= CAR (4);
END FOR-GENERATE;
```

Eg 4-Bit Counter

```
ARCHITECTURE IF-GENERATE OF COUNTER IS
  COMPONENT D-FLIPFLOP
    port (D, CLK : IN BIT; Q : OUT BIT);
  END COMPONENT;
BEGIN
  GK : for KIN 0 to 3 generate GK0;
      if K=0 generate
      DFF : D-FLIPFLOP portmap (COUNT, clock, CLK);
    end generate GK0;
    GK1-3 : if K> generate
    DFF : D-FLIPFLOP portmap ( CLK-1, clock, CLK);
    end generate GK1-3;
  end generate GKII
END ARCHITECTURE;
```

5




```
entity ALU is
  port (a,b : IN std_logic_vector (7 downto 0);
        sel : IN std_logic_vector (3 downto 0);
        cin : IN std_logic;
        y : out std_logic_vector (7 downto 0));
end ALU;
Architecture dataflow of ALU is
signal arith, logic : std_logic_vector (7 downto 0);
begin
   with sel (2 downto 0)
      arith <= a when "000";
               a+1 when "001";
               a-1 when "010";
               b when "011";
               b+1 when "100";
               b-1 when "101";
               a+b when "110";
               a+b+cin when other;

   with sel (2 downto 0) select
    logic <= NOT a when "000";
             NOT b when "001";
             a and b when "010";
             a or b when "011";
             a nand b when "100";
             a nor b when "101";
             a xor b when "110",
             not (a xor b) when other;
   with sel(3) select
          y <= arith when '0';
               logic when '1';
   end dataflow.
```

```
entity test-bench is
end;
architecture tb of test-bench is
component entity-under-test
    port (          );
    end component;
Begin
  generate waveform using beh_constructs;
  apply to entity under-test;
  EUT : entity-under-test portmap()
        ;         ;         ;
  end tb - behaviour.
```

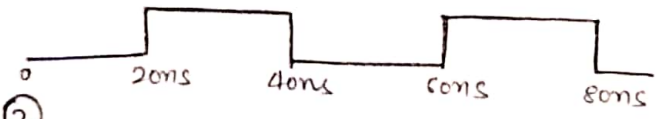
- A Test bench is a design construct which is used to verify the correctness of a hardware component. It has three main purpose
  1) TO generate stimulus for simulation
  2) To apply this stimulus to the entity under test (EUT) and collect the output response.
  3) To compare the output response with the expected value.

- Repetitive Pattern Waveform Generation

Repetitive Pattern

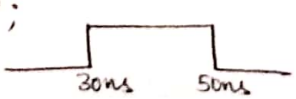
① RP with a const. ON/OFF Delay can be created using a concurrent signal assignment statement.

```
APR <= not APR after 20ns;
```



② A clock with varying ON & OFF period can be created using a process statement

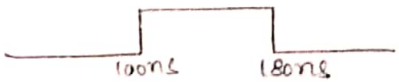
```
process
  Constant off-period : Time := 30ns;
  Constant ON-period : Time := 20ns;
  begin
    wait for off-period;
        D-Clk <= '1';
    wait for ON-period;
        D-Clk <= '0';
  end process;
```



Concurrent Assignment

① can be generated for a signal by using multiple ~~assignment~~ waveform element in a concurrent signal ass. state.

```
Reset <= '0', '1' after 100ns, '0' after 180ns
```



② The waveform can be made to repeat itself by placing a signal assignment state. inside a process statement with a wait statement

```
signal clk1, clk2 : std_logic := '0';
  !
two phase : process
 begin
  clk <= '1' after 5ns, '1' after 10ns
         '1' after 20ns, '0' after 25ns;
  clk <= '1' after 10ns, '1' after 20ns;
         '1' after 5ns, '0' after 30ns;
```

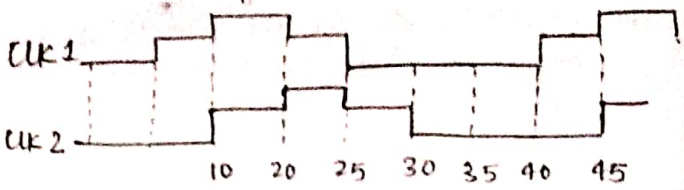
③ ↳ Conditional signal assignment statement can also be used to generate clock

D-CLK <= '1' after off-period
when D-CLK = '0'
'0' else after ON-period

④ ↳ A clock that is phase delay from another clock can be generated by using the delayed predefined attributes

Dly-D-CLK <= D-CLK'delayed (10ns)

wait for 35ns;
end process;



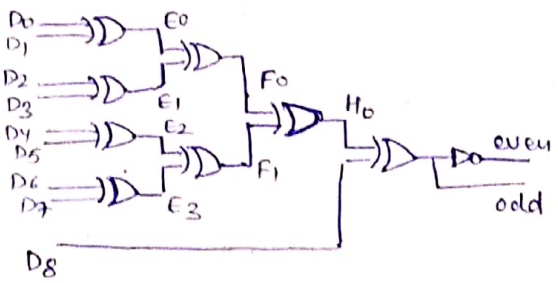
Q Test bench for 4:1 MUX

```
library ieee;
use ieee.std_logic_1164.all;
entity tb_mux is
end tb_mux;
Architecture tb of tb_mux is
component mux is
  port ( a,b,c,d : IN std_logic;
         sel : IN std_logic_vector(1 downto 0);
         y : OUT std_logic );
end component;
signal a,b,c,d,y : std_logic
signal sel : std_logic_vector(1 downto );
begin
  sel <= "00","01" after 20ns,
         "10" after 40ns,"11" after 60ns;
         "00" after 80ns;

  a<='0' after 10ns; '1' after 20ns;
  b<='0' after 30ns; '1' after 40ns;
  c<='0' after 50ns; '1' after 60ns;
  d<='0' after 70ns; '1' after 80ns;

M1 : MUX portmap (a=>a; b=>b; c=>c; d=>d; sel=>sel; y=>y);
end tb;
```

o VHDL code for 9-bit parity generator

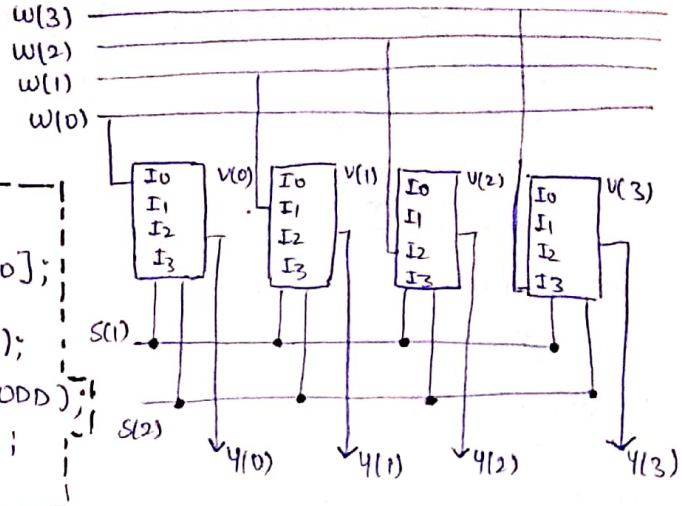



```
library ieee;
use ieee.std_logic_1164.all;
entity parity_9 is
    port ( D: in bit_vector(8 downto 0)
           even : out bit
           odd : buffer bit);
```

```
end parity-9;
architecture parity of parity-9 is
component XOR2
   port ( A,B : IN bit;
          Z : OUT bit);
end component;
component NOT2
   port ( A : IN bit
          B : OUT bit);
end component;
Signal E0, E1, E2, E3, F0, F1, H0 : bit;
begin
   XE0 : XOR2 portmap [D(0), D(1), E0];
   XH0 : XOR2 portmap ( F0, F1, H0);
   XODD : XOR2 portmap (H0, D(2), ODD);
   XEVEN : NOT2 portmap (ODD, EVEN);
end parity;
```

Barrel Shifter

It is a combination ckt which implements shifting operation without use of any seq. logic

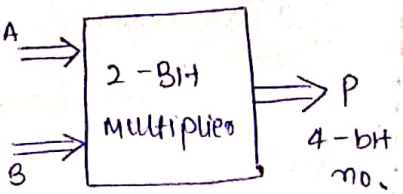



```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
Entity BS is
   port ( W : IN unsigned (3 downto 0);
          S : IN unsigned (1 downto 0);
          Y : OUT unsigned (3 downto 0));
end BS;
architecture A-BS of BS is
begin
   process (S,W)
   begin
   case S is
    when "00" => Y <= W;
    when "01" => Y <= ROR1;
    when "10" => Y <= WROR2;
    when "11" => Y <= WROR3;
    end case;
end process;
end A-BS;
```

- 2 bit Multiplier using HA.

$A = A_1 A_0$

$B = B_1 B_0$

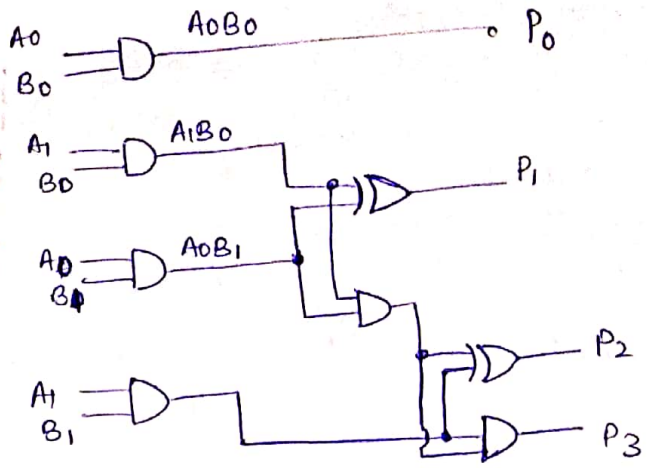


A_1B_0 A_0B_0

A_1B_1 A_0B_1 X

$A_1B_1+C_1$ $A_1B_0+A_0B_1$ A_0B_0

C_2 C_1



$P_0 = A_0B_0$

$P_1 = A_1B_0 + A_0B_1$ HA

$P_2 = A_1B_1 + C_1$

$P_3 = C_2$

- Barrel shifter
- An n-bit barrel shifter shifts or rotates the input data by the specified no. of bits in a combinational manner.

```
entity bs is
generic ( N: positive := 4)
port ( Data : in std_logic_vector ((N+1) downto 0);
       sel : in integer range 0 to (N-1);
       b_out : out std_logic_vector ((N-1) downto 0));
end bs;
architecture bs2 of bs is
begin
bout <= data ((N-1) sel) downto 0) & data ((N-1) downto N-sel);
 when sel > 0
else
     data;
end bs2;
```

Eg

```
data := 1011
let sel = 2;
bout <= data ((4-1-2) downto 0) & data ((4-1) downto (4-2));
     <= data (1 downto 0) & data (3 downto 2);
     <= 1110 → rotated by 2 bits
```

- Clock Divider

```
entity clk_divider is
generic ( N: positive);
port ( clk reset : in std_logic;
       clk_div : buffer_std_logic);
end clock_divider;
architecture behaviour of clk_divider is
begin
   process(clk, reset)
   variable COUNT: NATURAL;
```

```
begin
  if reset = '0' then
    COUNT = '0';
    clk-div <= 0;
  elsif clk-event & clk = '1' then
    COUNT := COUNT + 1;
    if COUNT = N then
      clk-div <= not clk-div;
      COUNT := '0';
    end if;
  endif;
end process;
end behaviour;
```

As long as Reset is '0', the clk-divider o/p stays at '0'. If reset goes to '1', the counter starts counting next rising edge of the clk. When N edge have been detected, clk-divider o/p is inverted & counter is reset to 0 & the cycle repeats.

- **Hamming code**

→ General algorithm for hamming code is

1. K parity bits are added to an n-bit data word, forming a code word of n + k bits (4)
3. The bits position are numbered with power of two, reserved for the parity bits and the remaining bits are the data bits.
2. The bit position are numbered in seq. from 1 to n+k
4. Parity bits are calculated by XOR operation of some combination of data bits. combination of data bits are

Bit pos. / Parity	1	2	3	4	5	6	7	8	9	10	11	12	13	...(n+k)
	P_1	P_2	D_1	P_4	D_2	D_3	D_4	P_8	D_5	D_6	D_7	D_8	D_9	
P_1	⊕		⊕		⊕		⊕		⊕		⊕		⊕	
P_2		⊕	⊕			⊕	⊕			⊕	⊕			⊕
P_4				⊕	⊕	⊕	⊕					⊕	⊕	⊕
P_8								⊕	⊕	⊕	⊕	⊕	⊕	⊕
K														

Calculation method of Parity bit

→ Calculation of Hamming code for 8-bit

If we take 8 bit data then 4 parity bit is needed because $2^0, 2^1, 2^2, 2^3$, these 4 position are reserved for parity bit.

let Data word = D1 D2 D3 D4 D5 D6 D7 D8 and parity bit = P1, P2, P4, P8

$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$

$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$

$P_4 = D_2 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$

$P_8 = D_5 \oplus D_6 \oplus D_7 \oplus D_8$

Code word : P_1 P_2 D_1 P_4 D_2 D_3 D_4 P_8 D_5 D_6 D_7 D_8

Then to check the parity bit for detecting the error by check bit

$C_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$

$C_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$

$C_3 = P_3 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_6$

$C_4 = P_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8$

If the result, $C = C_3C_4C_2C_1 = 0000$ indicates that no error has occured. However if $C \neq 0$ then 4-bit binary number is formed and gives the location of the error bit.

```
ENTITY tb_bcd_7seg IS
END tb_bcd_7seg;

ARCHITECTURE behaviour OF tb_bcd_7seg IS
 -- Component declaration for the Unit Under test (UUT)
component bcd_7segment
PORT ( BCDin : IN std_logic_vector( 3 downto 0);
       seven_segment : OUT std_logic_vector (6 downto 0));
 END COMPONENT;
-- Inputs
  signal BCDin : std_logic_vector (3 downto 0)  := (others => '0');
-- Outputs
  signal seven_segment : std_logic_vector (6 downto);
BEGIN
-- Instantiate the unit Under test (UUT)
 UUT : bcd_7segment PORTMAP( BCDin => BCDin;
                             seven_segment => seven_segment);

-- stimulus process
  stim_proc : process
  begin
      BCDin <= "0000";
      wait for 100ns;
      BCDin <= "0001";
      wait for 100 ns;
      :          :
```