Module 5

HDL and Data Flow Management

Structure

5.1 Objective

- The programming language will reduce the size compared to building up the circuit
- Different types of programming used for represent the digital circuits
- Usage of different programming language based on requirement.
- To learn and apply VHDL and HDL code for Digital Circuits.

5.2Introduction to VHDL:

VHDL stands for **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. In the mid-1980's the U.S. Department of Defense and the IEEE sponsored the development of this hardware description language with the goal to develop very high-speed integrated circuit. It has become now one of industry's standard languages used to describe digital systems.

The other widely used hardware description language is Verilog. Both are powerful languages that allow you to describe and simulate complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language) which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry

5.3 VHDL versus conventional programming languages

(1) A hardware description language is inherently parallel, i.e. commands, which correspond to logic gates, are executed (computed) in parallel, as soon as a new input arrives.

(2) A HDL program mimics the behavior of a physical, usually digital, system.

(3) It also allows incorporation of timing specifications (gate delays) as well as to describe a system as an interconnection of different components.

Levels of representation and abstraction

A digital system can be represented at different levels of abstraction [1]. This keeps the description and design of complex systems manageable. Figure 1 shows different levels of abstraction.

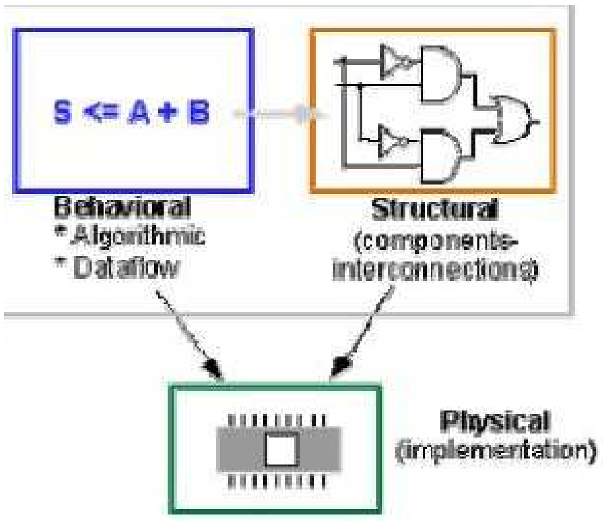


Figure 1: Levels of abstraction: Behavioral, Structural and Physical

The highest level of abstraction is the **behavioral** level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.

As an **example,** let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure 2 below.

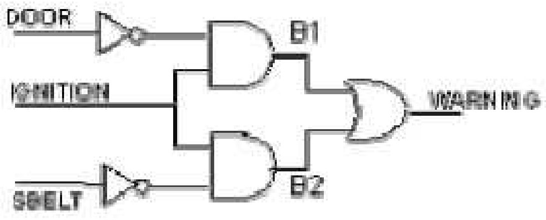


Figure 2: Structural representation of a "buzzer" circuit.

VHDL allows to describe a digital system at the **structural or the behavioral** level. The behavioral level can be further divided into two kinds of styles: **Data flow** and **Sequential**. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow betwee rnegisters (Register Transfer level).

The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input. On the other hand, **sequential statements** are executed in the sequence that they are specified. VHDL allows both **concurrent** and **sequential** signal assignments that will determine the manner in which they are executed.

Mixed level design consists both behavioral and structural design in one block diagram.

5.4 Basic Structure of a VHDL file

(a) Entity

A digital system in VHDL consistosf a design **entity** that can contain other entities that

are then considered components of the top-level entity. Each entity is modeled by an *entity declaration* and an *architecture body*. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure 3 below. In a typical design there will be many such entities connected together to perform the desired function.

A VHDL entity consisting of an interface (entity declaration) and a body (architectural description).

a. Entity Declaration

The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

entity NAME_OF_ENTITY **is** [**generic** *generic_declarations*);]
port (*signal_names*: **mode** *type*;
signal_names: : **mode** *type*;
signal_names: **mode** *type*);
end [NAME_OF_ENTITY] ;

An entity always starts with the keyword **entity**, followed by its name and the keyword **is**. Next are the port declarations using the keyword **port**. An entity declaration always ends with the keyword **end**, optionally [] followed by the name of the entity.

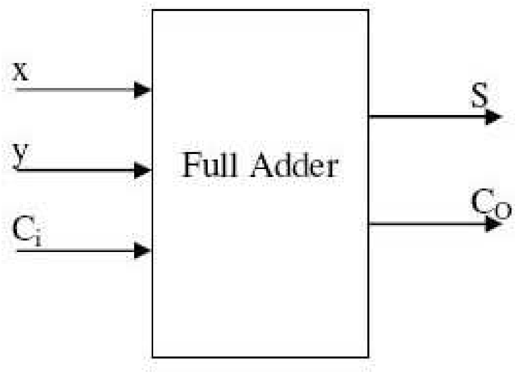


Figure 3: Block diagram of Full Adder

Example 1:

```
entity FULLADDER
       is
-- (After a double minus sign (-) the rest of
-- the line is treated as a comment)
--
-- Interface description of FULLADDER
port ( x, y, Ci: in bit;
S, CO: out bit);


end FULLADDER;
```

The module FULLADDER has five interface ports. Three of them are the input ports **x, y** and **Ci**indicated by the VHDL keyword **in**. The remaining two are the output ports **S** and
COindicated by **out**. The signals going through these ports are chosen to be of the type **bit**. The type **bit** consists of the two characters '0' and '1' and represents the binary logic values of the signals.

∉ The NAME_OF_ENTITY is a user-selected identifier

signal_names consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.

∉ **mode**: is one of the reserved words to indicate the signal direction:

o **in** – indicates that the signal is an input

o **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.

o **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity's architecture

o **inout** – the signal can be an input or an output.

∉ ***type***: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and stc_ulogic.

o *bit* – can have the value 0 and 1

o *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)

o *std_logic*, *std_ulogic*, *std_logic_vector*, *std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.

- *boolean* – can have the value TRUE and FALSE
- *integer* – can have a range of integer values
- *real* – cahave a range of real values
- *character* – any printing character
- *time* – to indicate time

∉ **generic:** generic declarations are optional

Example 2:

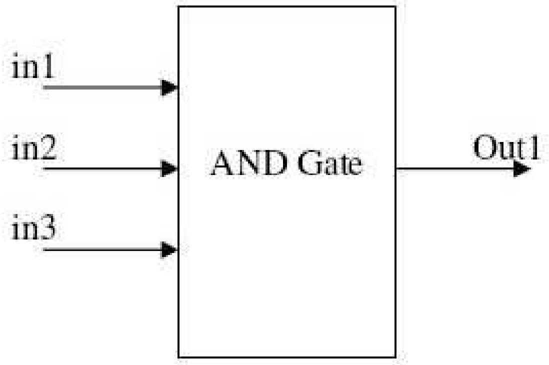


Figure 4: Block diagram of AND Gate

Example 3:

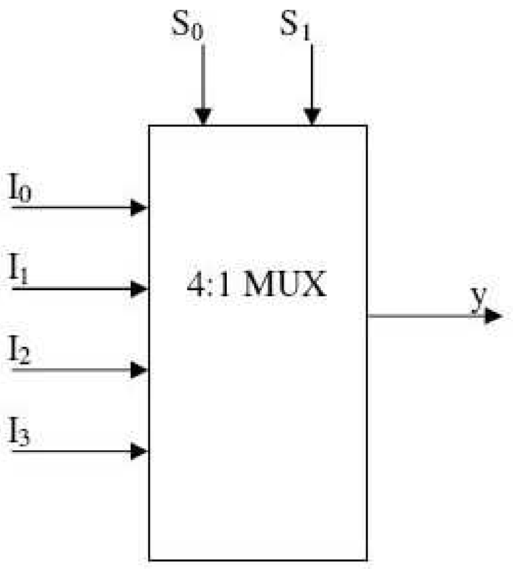


Figure 5: Block Diagram of 4:1 Multiplexer

entity mux4_to_1 **is**
port (I0,I1,I2,I3: **in** std_logic;

```
S: in      std_logic_vector(1downto          0);
```

y: **out** std_logic);
end mux4_to_1;

Example 4:

D Flip-Flop:

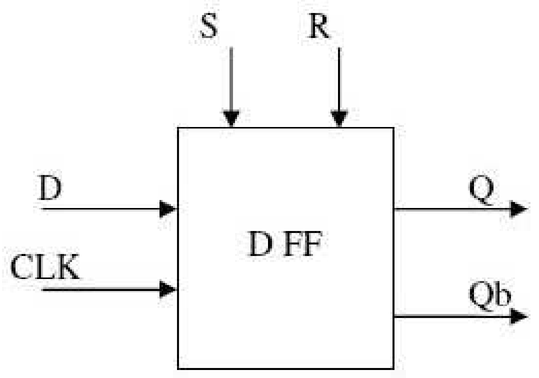


Figure 6: Block Diagram of D Flip Flop

entity dff_sr **is**
port (D,CLK,S,R: **in** std_logic;
Q,Qb: **out** std_logic);
end dff_sr;

Architecture body

The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

The architecture body looks as follows,

architecture architecture_name **of** NAME_OF_ENTITY **is**
-- Declarations
-- components declarations
-- signal declarations
-- constant declarations
-- function declarations
-- procedure declarations
-- type declarations
:
begin
-- Statements
:
end architecture_name;

The types of Architecture are:

(a) The behavioral Model
(b) Structure Model
(c) Mixed Model

(a) Behavioral model

The architecture body for the example of Figure 2, described at the behavioral level, is given below,

Example 1:

architecture behavioral **of** BUZZER **is**
begin
WARNING <= (**not** DOOR **and** IGNITION) **or** (**not** SBELT **and** IGNITION);
end behavioral;

The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword **begin** and gives the Boolean expression of the function. We will see later that a b haevioral model can be described in several other ways. The "<=" symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an **end** keyword followed by the architecture name.

Example 2:

The behavioral description of a 3 input AND gate is shown below.

entity AND3 **is**
port (in1, in2, in3: **in** std_logic;
out1: **out** std_logic);
end AND3;
architecture behavioral_2 **of** AND3**is**

```
begin
  out1  <= in1  and in2  and in3;
  end behavioral_2;
  Example 3:
  entity XNOR2 is
  port  (A,  B: in   std_logic;
  Z: out std_logic);
  end XNOR2;
  architecture    behavioral_xnor of     XNOR2 is
  -- signal     declaration     (of  internal    signals X, Y)
  signal X, Y:     std_logic;
  begin
  X <= A and B;
  Y <=  (not A)   and (not   B);
  Z <= X or Y;
  End behavioral_xnor;
  Example   4:
  SR Flip Flop:
  entity SRFF is
  port  (S,  R: in   std_logic;
  Q, Qb:   out  std_logic);
  end SRFF;
  architecture    behavioral_2 of     SRFF is
  begin
  Q <=  NOT  (S and   Qb);
  Qb <=  NOT (  R and  Q);
  end behavioral_2;
```

The statements in the body of the architecture make use of logic operators. In addition, other types of operators including relational, shift, arithmetic are allowed as well.

Concurrency

The signal assignments in the above examples are ***concurrent* statements**. This implies that the statements are executed when one or more of the signals on the right hand side change their value **(i.e. an event occurs on one of the signals).**

In general, a change of the current value of a signal is called an ***event.*** For instance, when the input S (in SR FF) changes, the first expression gets evaluated, which changes the value of Q, change in Q in turn triggers second expression and evaluates Qb. Thus Q and Qb are updated concurrently.

There may be a propagation delay associated with this change. **Digital systems are basically data-driven and an event which occurs on one signal will lead to an event on another signal, etc. Hence, the execution of the statements is determined by the flow of signal values. As a result, the order in which these statements are given does not matter** (i.e., moving the statement for the output Z ahead of that for X and Y does not change the outcome). This is in contrast to conventional, software programs that execute the statements in a sequential or procedural manner.

Example 5

```
architecture    CONCURR    of FULLADDER is
begin           ENT
S  <   x xor y  xor  Ci after   5 ns;
   =
CO    (x and y)    or (y   and  Ci)  or (x   and  Ci)  after    3 ns;
<=
```

Example2:

```
architecture CONCURRENT_VERSION2 of FULLADDER is
signal PROD1, PROD2, PROD3 : bit;
begin
SUM <= A xor B xor C; -- statement 1
CARRY <= PROD1 or PROD2 or PROD3; -- statement 2
PROD1 <= A and B; -- statement 3
PROD2 <= B and C; -- statement 4
PROD3 <= A and C; -- statement 5
end CONCURRENT_VERSION2;
```

(a)Concurrent statement: In VHDL With select and When else statements are called as concurrent statements and they do not require Process statement

Example 1: VHDL code for 4:1 multiplexor

```
library ieee;
use ieee.std_logic_1164.all;
entity Mux is
port( I: in std_logic_vector(3 downto 0);
S: in std_logic_vector(1 downto 0);
y: out std_logic);
end Mux;
-- architecture using logic expression
architecture behv1 of Mux is
begin
y<= (not(s(0)) and not(s(1)) and I(0)) or(s(0) and not(s(1))
and I(1)) or (not(s(0)) and s(1) and I(2))or (s(0) and s(1) and
I(3));
end behv1;
-- Architecture using when..else:
architecture behv2 of Mux is
begin
y <= I(0) when S="00" else
I(1) when S="01" else
I(2) when S="10" else
I(3) when S="11" else
'Z' ;
```

```
end behv2;
-- architecture using with select statement
architecture behv3 of Mux is
begin
with s select
y<=i(0) when "00",
i(1) when "01",
i(2) when "10",
i(3) when "11",
'Z' when others;
end behv3;
```

Note: 'Z' high impedence state should be entered in capital Z

Example 2: SR flipflop using when else statement

```
entity SRFF is
port ( S, R: in bit;
Q, QB: inout bit);
end RSFF;
architecture beh of RSFF is
begin
Q <= Q when S= '0' and R = '0' else
'0' when S = '0' and R = '1' else
'1' when S = '1' and R = '0' else
'Z';
QB <= not(Q);
end beh;
```

The statement **WHEN…..ELSE** conditions are executed one at a time in sequential order until the conditions of a statement are met. The first statement that matches the conditions required assigns the value to the target signal. The target signal for this example is the local signal **Q.** Depending on the values of signals **S** and **R,** the values Q,1,0 and Z are assigned to **Q.**

If more than one statements conditions match, the first statement that matches does the assign, and the other matching state.

In **with …select** statement all the alternatives arte checked simultaneously to find a matching pattern. Therefore the **with … select** must cover all possible values of the selector

Structural Descriptions

A description style where different components of an architecture and their interconnections are specified is known as a VHDL structural description. Initially, these components are declared and then components' instances are generated or instantiated. At the same time, signals are mapped to the components' ports in order to connect them like wires in hardware. VHDL simulator handles component instantiations as concurrent assignments.

Syntax:

component declaration:

component *component_name*
[**generic** (*generic_list*: *type_name* [:= *expression*] {;
generic_list: *type_name* [:= *expression*]});]
[**port** (*signal_list*: **in|out|inout|buffer** *type_name* {;
signal_list: **in|out|inout|buffer** *type_name*});]
end component;

Component instantiation:
component_label: *component_name* **port map** (*signal_mapping*);
The mapping of ports to the connecting signals during the instantiation can be done through the positional notation. Alternatively, it may be done by using the named notation.
If one of the ports has no signal connected to it (this happens, for example, when there are unused outputs), a reserved word open may be used.
Example 1:
signal_mapping: *declaration_name* => *signal_name*.
entity RSFF **is**
port (SET, RESET: **in** bit;
Q, QBAR: **inout** bit);
end RSFF;
architecture NETLIST **of** RSFF **is**
component NAND2
port (A, B: **in** bit; C: **out** bit);
end component;
begin
U1: NAND2 **port map** (SET, QBAR, Q);
U2: NAND2 **port map** (Q, RESET, QBAR);
end NETLIST;
--- named notation instantiation: ---
U1: NAND2 **port map** (A => SET, C => Q, B => QBAR);

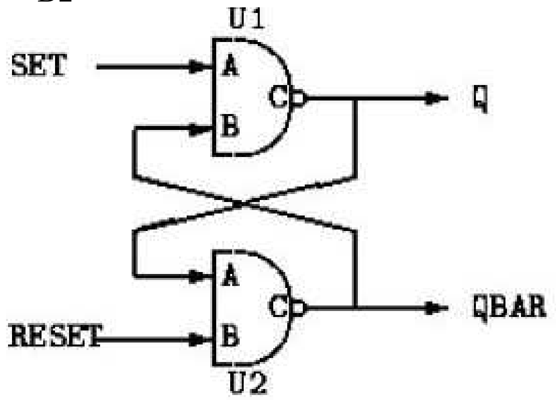


Figure 1: Schematic of SR FF using NAND Gate

The lines between the first and the keyword *begin* are a *component declaration*. It describes the interface of the entity *nand_gate* that we would like to use as a component in (or part of) this design. Between the *begin* and *end* keywords, the statements define *component instances*.

There is an important distinction between an entity, a component, and a component instance in VHDL.

The entity describes a design interface, the component describes the interface of an entity that will be used as an instance (or a sub-block), and the component instance is a distinct copy of the component that has been connected to other parts and signals.

In this example the component *nand_gate* has two inputs (*A* and *B*) and an output ©. There are two instances of the *nand_gate* compone nt in this architectureo rresponding to the two nand symbols in the schematic. The first instance refers to the top nand gate in

the schematic and the statement is called the **component instantiation statement.** The first word of the component instantiation statement (u1:nand2) gives instance a name, *u1*, and specifies that it is an instance of the component *nand_gate*. The next words describes how the component is connected to the set of the design using the **port map** clause.

The **port map clause** specifies what signals of the design should be connected to the interface of the component in the same order as they are listed in the component declaration. The interface is specified in order as *A*, *B* and then *C*, so this instance connects ***set to A, QBAR*** **to** ***B*** and ***Q*** **to C.** This corresponds to the way the top gate in the schematic is connected. The second instance, named *n2*, connects ***RESET*** **to** ***A, Q*** **to A,** and ***QBAR*** **to C** of a different instance of the same *nand_gate* component in the same manner as shown in the schematic.

The structural description of a design is simply a textual description of a schematic. A list of components and there connections in any language is also called a netlist. The structural description of a design in VHDL is one of many means of specifying netlists

.

Example 2: Four Bit Adder – Illustrating a structural VHDL model:

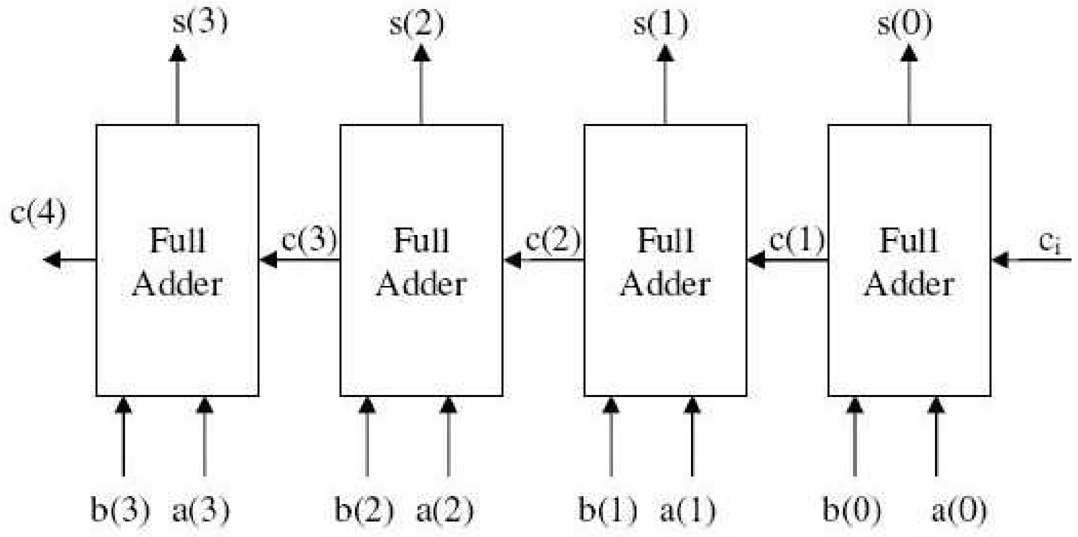


Figure 2: 4-bit Adder using four Full Adders.

```
-- Example of a four bit adder
library ieee;
use ieee.std_logic_1164.all;
-- definition of a full adder
entity FULLADDER is
port (x, y, ci: in std_logic;
s, co: out std_logic);
end FULLADDER;
architecture fulladder_behav of FULLADDER is
begin
s <= x xor y xor ci ;
co <= (x and y) or (x and ci)or(y and ci));
end fulladder_behav;
```

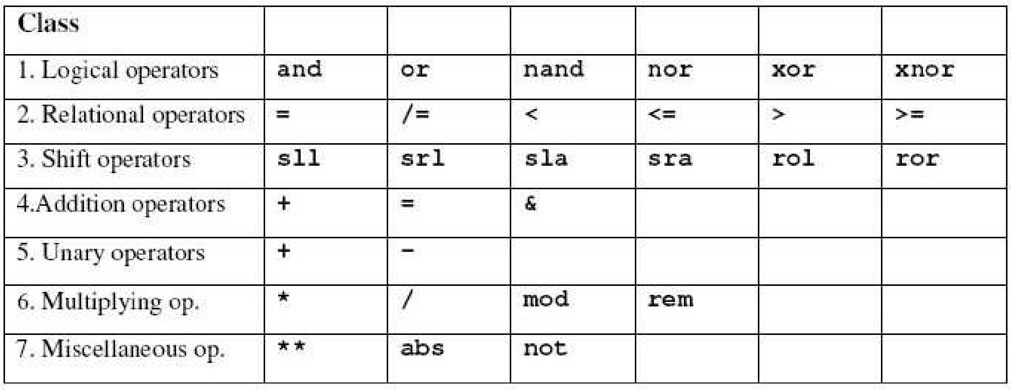
```
-- 4-bit adder
library ieee;
use ieee.std_logic_1164.all;
entity FOURBITADD is
port (a, b: in std_logic_vector(3 downto 0);
Cin : in std_logic;
sum: out std_logic_vector (3 downto 0);
Cout: out std_logic);
end FOURBITADD;
architecture fouradder_structure of FOURBITADD is
signal c: std_logic_vector (4 downto 0);
component FULLADDER
port(x, y, ci: in std_logic;
s, co: out std_logic);
end component;
begin
FA0: FULLADDER
port map (a(0), b(0), Cin, sum(0), c(1));
FA1: FULLADDER
port map (a(1), b(1), C(1), sum(1), c(2));
FA2: FULLADDER
port map (a(2), b(2), C(2), sum(2), c(3));
FA3: FULLADDER
port map (a(3), b(3), C(3), sum(3), c(4));
Cout <= c(4);
end fouradder_structure;
```

We needed to define the internal signals c (4 downto 0) to indicate the nets that connect the output carry to the input carry of the next full adder. For the first input we used the input signal Cin. For the last carry we defined c (4) as an internal signal. We could not use the output signal Cout since VHDL does not allow the use of outputs as internal signals! For this reason we had to define the internal carry c(4) and assign c(4) to the output carry signal Cout.

5.5 Operators

(a) VHDL Operators

VHDL supports different classes of operators that operate on signals, variables and constants. The different classes of operators are summarized below.



Class						
1. Logical operators	and	or	nand	nor	xor	xnor
2. Relational operators	=	/=	<	<=	>	>=
3. Shift operators	sll	srl	sla	sra	rol	ror
4.Addition operators	+	=	&			
5. Unary operators	+	-				
6. Multiplying op.	*	/	mod	rem		
7. Miscellaneous op.	**	abs	not			

The order of precedence is the highest for the operators of class 7, followed by class 6 with the lowest prec dence for class 1. Unless parentheses are used, the operators with the highest precedence are applied first. Operators of the same class have the same precedence and are applied from left to right in an expression. As an example, consider the following std_ulogic_vectors, X (=’010’), Y(=’10’), and Z (‘10101’). The expression **not** X & Y **xor** Z **rol** 1
is equivalent to ((**not** X) & Y) **xor** (Z **rol** 1) = ((101) & 10) **xor** (01011) =(10110) **xor** (01011) = 11101. The xor is executed on a bit-per-bit basis.

1. Logic operators

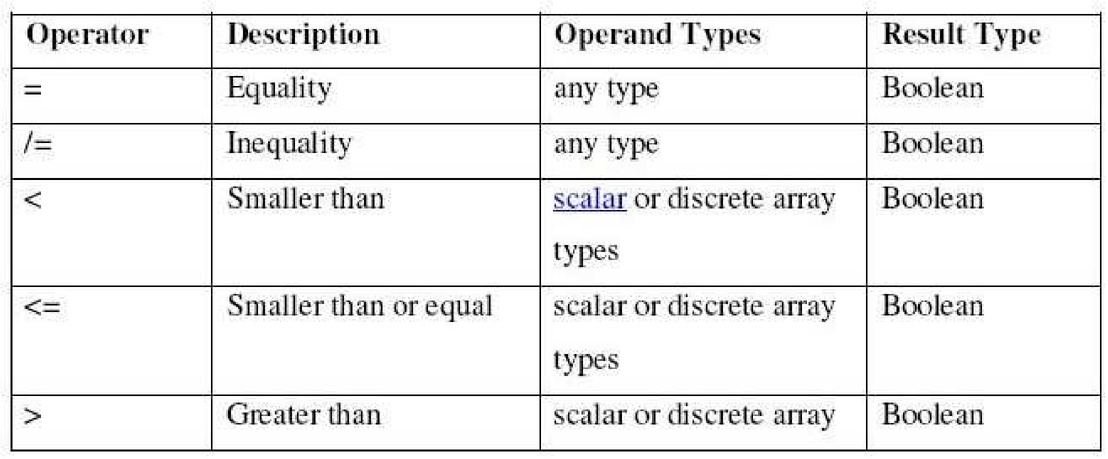
The logic operators (and, or, nand, nor, xor and xnor) are defined for the “bit”, “boolean”, “std_logic” and “std_ulogic” types and their vectors. They are used to define Boolean logic expression or to perform bit-per-bit operations on arrays of bits. They give a result of the same type as the operand (Bit or Boolean). These operators can be applied to signals, variables and constants.

Notice that the nand and nor operators are not associative. One should use parentheses in a sequence of nand or nor operators to prevent a syntax error:

X **nand** Y **nand** Z will give a syntax error and should be written as (X **nand** Y) **nand** Z.

2. Relational operators

The relational operators test the relative values of two scalar types and give as result a Boolean output of “TRUE” or “FALSE”.



Operator	Description	Operand Types	Result Type
=	Equality	any type	Boolean
/=	Inequality	any type	Boolean
<	Smaller than	scalar or discrete array types	Boolean
<=	Smaller than or equal	scalar or discrete array types	Boolean
>	Greater than	scalar or discrete array	Boolean

Notice that symbol of the operator “<=” (smaller or equal to) is the same one as the assignment operator used to assign a value to a signal or variable. In the following examples the first “<=” symbol is the assignment operator. Some examples of relational operations are:

variable STS : Boolean;
constant A : integer :=24;
constant B_COUNT : integer :=32;
constant C : integer :=14;
STS <= (A < B_COUNT) ; -- will assign the value “TRUE” to STS
STS <= ((A >= B_COUNT) **or** (A > C)); -- will result in “TRUE”
STS <= (std_logic (‘1’, ‘0’, ‘1’) < std_logic(‘0’, ‘1’,’1’));--makes STS “FALSE”
type new_std_logic **is** (‘0’, ‘1’, ‘Z’, ‘-‘);
variable A1: **new_std_logic** :=’1’;
variable A2: **new_std_logic** :=’Z’;
STS <= (A1 < A2); will result in “TRUE” since ‘1’ occurs to the left of ‘Z’.

For discrete array types, the comparison is done on an element-per-element basis, starting from the left towards the right, as illustrated by the last two examples.

3. Shift operators

These operators perform a bit-wise shift or rotate operation on a one-dimensional array of elements of the type bit (or std_logic) or Boolean.

Operator	Description	Operand Type	Result Type
`sll`	Shift left logical (fill right vacated bits with the 0)	Left: Any one-dimensional array type with elements of type bit or Boolean; Right: integer	Same as left type
`srl`	Shift right logical (fill left vacated bits with 0)	same as above	Same as left type
`sla`	Shift left arithmetic (fill right vacated bits with rightmost bit)	same as above	Same as left type
`sra`	Shift right arithmetic (fill left vacated bits with leftmost bit)	same as above	Same as left type
`rol`	Rotate left (circular)	same as above	Same as left type
`ror`	Rotate right (circular)	same as above	Same as left type

The operand is on the left of the operator and the number (integer) of shifts is on the right side of the operator. As an example,
variable NUM1 :bit_vector := “10010110”;
NUM1 **srl** 2;
will result in the number “00100101”.
When a negative integer is given, the opposite action occurs, i.e. a shift to the left will be a shift to the right. As an example
NUM1 **srl** –2 would be equivalent to NUM1 **sll** 2 and give the result “01011000”.
Other examples of shift operations are for the bit_vector A = “101001”
variable A: bit_vector :=”101001”;

A sll 2 results in "100100"

A srl 2 results in "001010"

A sla 2 results in "100111"

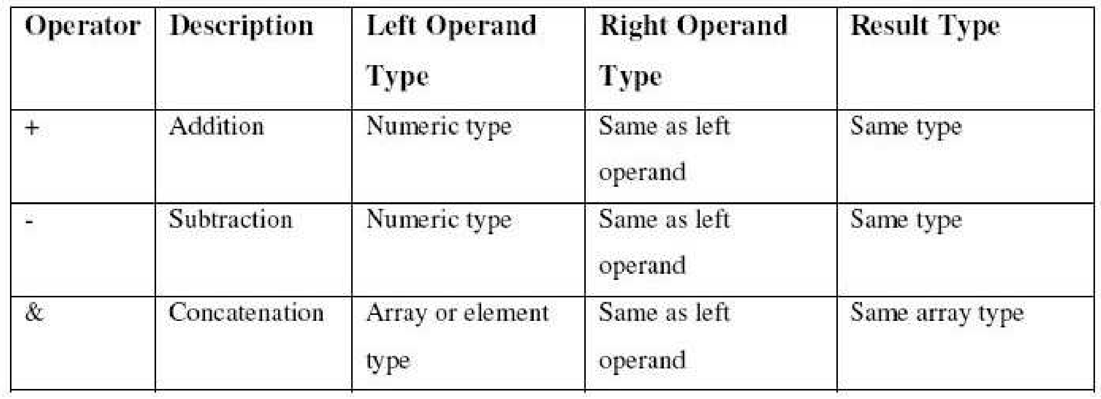
A sra 2 results in "111010"

A rol 2 results in "100110"

A ror 2 results in "011010"

4. Addition operators

The addition operators are used to perform arithmetic operation (addition and subtraction) on operands of any numeric type. The concatenation (&) operator is used to concatenate two vectors together to make a longer one. In order to use these operators one has to specify the ieee.std_logic_unsigned.all or std_logic_arith package package in addition to the ieee.std_logic_1164 package.



Operator	Description	Left Operand Type	Right Operand Type	Result Type
+	Addition	Numeric type	Same as left operand	Same type
-	Subtraction	Numeric type	Same as left operand	Same type
&	Concatenation	Array or element type	Same as left operand	Same array type

An example of concatenation is the grouping of signals into a single bus [4].
signal MYBUS :std_log ic_vectro (15 **downto** 0);
signal STATUS :std_logic_vector (2 **downto** 0);
signal RW, CS1, CS2 :std_logic;
signal MDATA :std_logic_vector (0 **to** 9);
MYBUS <= STATUS & RW & CS1 & CS2 & MDATA;
Other examples are
MYARRAY (15 **downto** 0) <= "1111_1111" & MDATA (2 **to** 9);
NEWWORD <= "VHDL" & "93";
The first example results in filling up the first 8 leftmost bits of MYARRAY with 1's and the rest with the 8 rightmost bits of MDATA. The last example results in an array of characters "VHDL93".
Example:

```
Signal a: std_logic_vector (3 downto 0);
Signal b: std_logic_vector (3 downto 0);
Signal y:std_logic_vector (7 downto 0);
Y<=a & b;
```

5. Unary operators

The unary operators “+” and “-“ are used to specify the sign of a numeric type.

Operator	Description	Operand Type	Result Type
+	Identity	Any numeric type	Same type
-	Negation	Any numeric type	Same type

6. Multiplying operators

The multiplying operators are used to perform mathematical functions on numeric types (integer or floating point).

Operator	Description	Left Operand Type	Right Operand Type	Result Type
*	Multiplication	Any integer or floating point	Same type	Same type
		Any physical type	Integer or real type	Same as left
		Any integer or real type	Any physical type	Same as right
/	Division	Any integer or floating point	Any integer or floating point	Same type
		Any physical type	Any integer or real t ype	Same as left
		Any physical type	Same type	Integer
`mod`	Modulus	Any integer type		Same type
`rem`	Remainder	Any integer type		Same type

The multiplication operator is also defined when one of the operands is a physical type and the other an integer or real type.

The remainder (rem) and modulus (mod) are defined as follows:

A **rem** B = A –(A/B)*B (in which A/B in an integer)

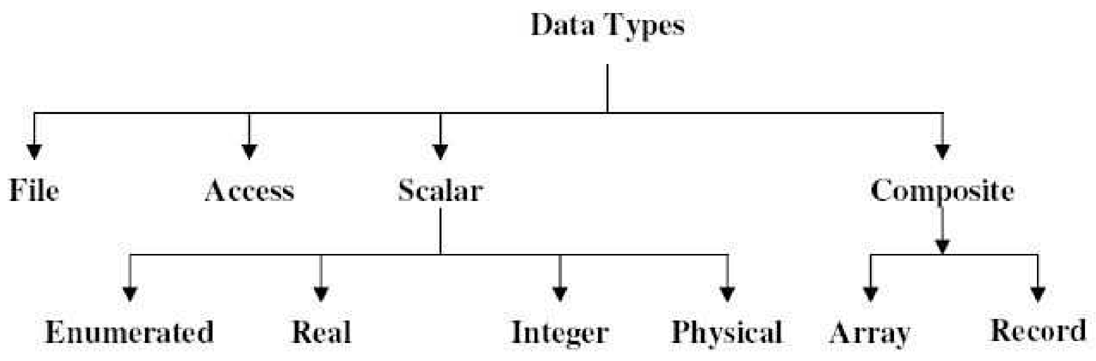
A **mod** B = A – B * N (in which N is an integer)
The result of the **rem** operator has the sign of its first operand while the result of the **mod** operators has the sign of the second operand.
Some examples of these operators are given below.
11 **rem** 4 results in 3
(-11) **rem** 4 results in -3
9 **mod** 4 results in 1
7 **mod** (-4) results in –1 (7 – 4*2 = -1).

7. Miscellaneous operators

These are the absolute value and exponentation operators that can be applied to numeric types. The logical negation (not) results in the inverse polarity but the same type.

Operator	Description	Left Operand Type	Right Operand Type	Result Type
**	Exponentiation	Integer type	Integer type	Same as left
		Floating point	Integer type	Same as left
abs	Absolute value	Any numeric type		Same type
not	Logical negation	Any bit or Boolean type		Same type

VHDL data types:
To define new type user must create a type declaration. A type declaration defines the **name of the type** and the **range of the type**.
Type declarations are allowed in
(i) Package declaration (ii) Entity Declaration (iii) Architecture Declaration
(iv)Subprogram Declaration (v) Process Declaration



Enumerated Types:

An Enumerated type is a very powerful tool for abstract modeling. All of the values of an enumerated type are user defined. These values can be identifiers or single character literals.

An identifier is like a name, for examples: day, black, x

Character literals are single characters enclosed in quotes, for example: ‘x’, ‘I’, ‘o’

```
Type    Fourval is      (‘x’,   ‘o’,   ‘I’,   ‘z’);
Type    color           is (red,   yello,   blue,   green,   orange);
Type    Instruction is    (add,   sub,  lda,   ldb,   sta,   stb,   outa,   xfr);
        s
```

Real type example:

```
Type input level is range -10.0 to +10.0
Type probability is range 0.0 to 1.0;
Type W_Day is (MON, TUE, WED, THU FRI, SAT, SUN);
type dollars is range 0 to 10;

variable day: W_Day;
variable Pkt_money:Dollars;
Case Day is
When TUE => pkt_money:=6;
When MON OR WED=> Pkt_money:=2;
When others => Pkt_money:=7;
End case;
```

Example for enumerated type - Simple Microprocessor model:

```
Package instr is
Type instruction is (add, sub, lda, ldb, sta, stb, outa, xfr);
End instr;
Use work.instr.all;
Entity mp is
PORT (instr: in Instruction;
Addr: in Integer;
Data: inout integer);
End mp;
Architecture mp of mp is
Begin
Process (instr)
type reg is array(0 to 255) of integer;
variable a,b: integer;
variable reg: reg;
begin
case instr is
when lda => a:=data;
when ldb => b:=data;
when add => a:=a+b;
when sub => a:=a-b;
when sta => reg(addr) := a;
when stb =>   reg(addr):= b;
when outa => data := a;
when xfr =>   a:=b;
end case;
end process;
end mp;
```

Physical types:
These are used to represent real world physical qualities such as length, mass, time and current.

Type______is range ______ to ______
Units identifier;
{(identifier=physical literal;)}
end units identifier;

Examples:

(1) Type resistance **is range** 0 to 1E9
units
ohms;
kohms = 1000ohms;

```
Mohms = 1000kohms;
end units;
(2) Type current is range 0 to 1E9
units
na;
ua = 1000na;
ma = 1000ua;
a = 1000ma;
end units;
```

Composite Types:

Composite types consist of array and record types.

∉ Array types are groups of elements of same type

∉ Record allow the grouping of elements of different types

∉ Arrays are used for modeling linear structures such as ROM, RAM

∉ Records are useful for modeling data packets, instruction etc.

∉ A composite type can have a value belonging to either a scalar type, composite type or an access type.

Array Type:

Array type groups are one or more elements of the same type together as a single object. Each element of the array can be accessed by one or more array indices.

```
Type data-bus is array (0to 31) of BIT;
Variable x: data-bus;
Variable y: bit;
Y := x(0);
Y := x(15);
Type address_word is array(0 to 63) of BIT;
Type data_word is array(7 downto 0) of std_logic;
Type ROM is array(0 to 255) of data_word;
```

We can declare array objects of type mentioned above as follows:

```
Variable ROM_data: ROM;
Signal Address_bus: Address_word;
Signal word: data_word;
```

Elements of an array can be accessed by specifying the index values into the array.

X<= Address_bus(25); transfers 26th element of array Address_bus to X.

Y := ROM_data(10)(5); transfers the value of 5th element in 10th row.

Multi dimentional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array variable, which is a matrix of integers with four rows and three columns:

```
Type matrix4x3 is array (1 to 4, 1 to 3) of integer;
Variable matrixA: matrix4x3 := ((1,2,3), (4,5,6), (7,8,9), (10,11,12));
Variable m:integer;
```

The viable matrixA, will be initialized to

1 2 3

4 5 6
7 8 9
10 11 12
The array element matrixA(3,2) references the element in the third row and second column, which has a value of 8.
m := matrixA(3,2); m gets the value 8

Record Type:
Record Types group objects of many types together as a single object. Each element of the record can be accessed by its field name.
Record elements can include elements of any type including arrays and records.
Elements of a record can be of the same type or different types.
Example:
Type optype **is** (add, sub, mpy, div, cmp);
Type instruction **is**
Record
Opcode : optype;
Src : integer;
Dst : integer;
End record;

Structure of Verilog module:

module module_name(signal_names)
Signal_type signal_names;
Signal_type signal_names;
Aasign statements
Assign statements
Endmodule_name

Verilog Ports:

- Input: The port is only an input port.I. In any assignment statement, the port should appear only on the right hand side of the statement
- Output: The port is an output port. The port can appear on either side of the assignment statement.
- Inout: The port can be used as both an input & output. The inout represents a bidirectional bus.

Verilog Value Set:

- *0* represents low logic level or false condition
- *1* represents high logic level or true condition
- *x* represents unknown logic level

- *z* represents high impedance logic level

Verilog Operators

Operators in Verilog are the same as operators in programming languages. They take two values and compare or operate on them to yield a new result. Nearly all the operators in Verilog are exactly the same as the ones in the C programming language.

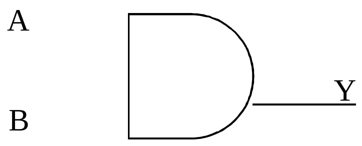
Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Addition
	-	Subtraction
	%	Modulus
	+	Unary plus
	i	Unary minus
Relational	>	Greater than
	<	Less Than
	>=	Greater than or equal to
	<=	Less than or equal to
Equality	==	Equality
	!=	Inequality
Logical	!	Logical Negation
	&&	Logical And
	||	Logical Or
Shift	>>	Right Shift
	<<	Left Shift
Conditional	?	Conditional
Reduction	~	Bitwise negation

	~&	Bitwise nand
	|	Bitwise or
	~|	Bitwise nor
	^	Bitwise xor
	^~	Bitwise xnor
	~^	Bitwise xnor
Concatenation	{}	

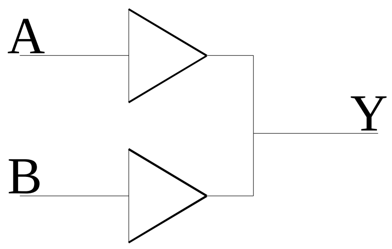
Examples:
x = y + z; //x will get the value of y added to the value of z
x = 1 >> 6;   //x will get the value of 1 shifted right by 5 positions
x = !y   //x will get the value of y inverted. If y is 1, x is 0 and vise versa

Verilog Data Types:
Nets (i)
an be thought as hardware wires driven by logic
Equal *z* when unconnected
Various types of nets
wire
wand   (wired-AND)
wor   (wired-OR)
tri   (tri-state)
In following examples: Y is evaluated, *automatically*, every time A or B changes

Nets (ii)



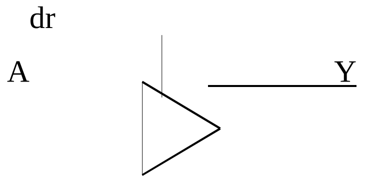
wire Y; // declaration
assign Y = A & B;



wand Y;   // declaration
assign Y = A; assign Y = B;

wor Y; // declaration

assign Y = A; assign Y = B;



tri Y; // declaration

assign Y = (dr) ? A : z;

Registers:

- Variables that store values
- Do not represent real hardware but ..
- .. real hardware can be implemented with registers
- Only one type: reg
  reg A, C; // declaration
  // assig nmentsare always done inside a procedure
  A = 1;
  C = A; // C gets the logical value 1
  A = 0; // C is still 1
  C = 0; // C is now 0
- Register values are updated explicitly!!

Vectors:

- Represent buses
  wire [3:0] busA;
  reg [1:4] busB;
  reg [1:0] busC;
- Left number is MS bit
- Slice management

  busC[1] = busA[2];
  busC[0] = busA[1];
- Vector assignment (*by position!!*)

  busB[1] = busA[3];
  busB[2] = busA[2];
  busB[3] = busA[1];
  busB[4] = busA[0];

Integer & Real Data Types:

- Declaration
  integer i, k;
  real r;

Use as registers (inside procedures)

```
i = 1; // assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
```

- Integers are not initialized!!
- Reals are initialized to *0.0*

Parameters:

- Parameters represents global constants.They are declared by the predefined word parameter.

```
module comp_genr(X,Y,XgtY,XltY,XeqY);
parameter N = 3;
input [ N :0] X,Y;
output XgtY,XltY,XeqY;
wire [N:0] sum,Yb;
```

Time Data Type:

- Special data type for simulation time measuring
- Declaration
  time my_time;
- Use inside procedure
  my_time = $time; // get current sim time
- Simulation runs at simulation time, not real time

Arrays (i):

Syntax

```
integer count[1:5]; // 5 integers
reg var[-15:16]; // 32 1-bit regs
reg [7:0] mem[0:1023]; // 1024 8-bit regs
```

Accessing array elements

Entire element: mem[10] = 8'b 10101010;

Element subfield (needs temp storage):

```
reg [7:0] temp;
..
temp    =    mem[10];
var[6]    =    temp[2];
```

Strings: Implemented with regs:

```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = "Hello Verilog";
string_val = "hello"; // MS Bytes are filled with 0
string_val = "I am overflowed"; // "I " is truncated
```

Escaped chars:

\n	newline
\t	tab

%% %
\\ \
\“ “

Styles(Types) of Descriptions:

- Behavioral Descriptions
- Structural Descriptions
- Switch – Level Descriptions
- Data – Flow Descriptions
- Mixed Type Descriptions

Behavioral Descriptions:

VHDL Behavioral description

```
entity half_add is
            port (I1, I2 : in bit; O1, O2 : out bit);
end half_add;
architecture behave_ex of half_add is
     --The architecture consists of a process construct
begin
     process (I1, I2)
   --The above statement is process statement                       begin
         O1 <= I1 xor I2 after 10 ns;
         O2 <= I1 and I2 after 10 ns;
      end process;
end behave_ex;
```

Verilog behavioral Description:

```
module half_add (I1, I2, O1, O2);
      input I1, I2;
      output   O1,   O2;
      reg     O1,     O2;
      always @(I1, I2)
      //The above abatement is always
      //The module consists of always construct
      begin
         #10 O1 = I1 ^ I2;
         #10 O2 = I1& I2;
      end
      endmodule
```

VHDL Structural Descriptions:

```
entity system is
      port (a, b : in bit;
            sum, cout : out bit);
end system;
architecture struct_exple of system is
 component xor2
 --The above statement is a component statement
          port(I1, I2 : in bit;
                  O1 : out bit);
 end component;
component and2
port(I1, I2 : in bit;
            O1 : out bit);
end component;
     begin
       X1 : xor2 port map (a, b, sum);
       A1 : and2 port map (a, b, cout);
end struct_exple;
```

Verilog Structural Description:

```
module system(a, b, sum, cout);
   input a, b;
   output   sum,   cout;
     xor X1(sum, a, b);
   //The above statement is EXCLUSIVE-OR gate
     and a1(cout, a, b);
   //The above statement is AND gate
   endmodule
```

Switch Level Descriptions:

VHDL Description:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Inverter is
  Port (y : out std_logic; a: in std_logic );
end Inverter;
architecture Invert_switch of Inverter is
component nmos
--nmos is one of the key words for switch-level.
port (O1: out std_logic; I1, I2 : in std_logic);
end component;
component pmos
--pmos is one of the key words for switch-level.
port (O1: out std_logic ;I1, I2 : in std_logic);
end component;
for all: pmos use entity work. mos (pmos_behavioral);
for all: nmos use entity work. mos (nmos_behavioral);
--The above two statements are referring to a package mos
--See  details  in  Chapter  5
constant vdd: std_logic := '1';
constant gnd : std_logic:= '0';
begin
p1 : pmos port map (y, vdd, a);
n1:  nmos port map (y, gnd, a);
end Invert_switch;
```

Verilog switch – Level Description:

```
module invert(y,a);
input a;
output     y;
supply1 vdd;
supply0 gnd;
pmos p1(y, vdd, a);
nmos n1(y, gnd, a);
--The above two statement are using the two primitives pmos and nmos
endmodule
```

Data – Flow Descriptions:

VHDL Data – Flow Description:

```
entity halfadder is
port (a,b: in bit;
      s,c: out bit);
end halfadder;
architecture HA_DtFl of halfadder is

begin
    s <= a xor b;
    c <= a and b;
end HA_DtFl;
```

Verilog Data – Flow Description:

```
module halfadder (a,b,s,c);
  input a;
 input   b;
  output s;
  output c;
    assign s = a ^ b;
    assign c = a & b;
endmodule
```

5.6 Comparision of VHDL & Verilog:

- Data Types

VHDL: Types are in built in or the user can create and define them.User defined types give the user a tool to write the code effectively. VHDL supports multidimensional array and physical type.

Verilog: Verilog data types are simple & easy to use. There are no user defined types.

- Ease of Learning

VHDL:Hard to learn because of its rigid type requirements.

Verilog: Easy to learn,Verilog users just write the module without worrying about what Library or package should be attached.

- Libraries and Packages

VHDL:Libraries and packages can be attached to the standard VHDL package.Packages can include procedures and functions, & the package can be made available to any module that needs to use it.

Verilog:No concept of Libraries or packages in verilog.

- ■ Operators

VHDL:An extensive set of operators is available in VHDL,but it does not have predefined unary operators.

Verilog: An extensive set of operators is also available in verilog. It also has predefined unary operators.

- ■ Procedures and Tasks

VHDL:Concurrent procedure calls are allowed. This allows a function to be written inside the procedure's body.This feature may contribute to an easier way to describe a complex system.

Verilog:Concurrent task calls are allowed.Functions are not allowed to be written in the task's body.

ASSIGNMENT QUESTIONS

1) Explain entity and architecture with an example
2) Explain structure of verilog module with an example
3) Explain VHDL operators in detail.
4) Explain verilog operators in detail.
5) Explain how data types are classified in HDL. Mention the advantages of VHDL data types over verilog.
6) Mention the types of HDL descriptions. Explain dataflow and behavioral descriptions

7) Describe different types of HDL description with suitable example.
8) Mention different styles (types) of descriptions. Explain mixed type and mixed language descriptions.
9) Compare VHDL and Verilog
10) Write the result of all shift and rotate operations inVHDL after applying them to a 7 bit vector A = 1001010
11) Explain composite and access data types with an example for each.
12) Discuss different logical operators used in HDL's

5.7 DATA FLOW DESCRIPTIONS

Data flow is one type(style) of hardware description.

Facts

∉ Data – flow descriptions simulate the system by showing how the signal flows from system inputs to outputs.

∉ Signal – assignment statements are con ucrent. At any simulation time, all signal-assignment statements that have an event are executed concurrently.

5.8 VHDL Description and structure

```
entity system is
   port (I1, I2 : in bit; O1, O2 : out bit);
end;
architecture dtfl_ex of system is
begin
   O1 <= I1 and I2; -- statement 1.
   O2 <= I1 xor I2; -- statement 2.

--Statements 1 and 2 are signal-assignment statements

end dtfl_ex;
```

Verilog Description

```
module system (I1, I2, O1, O2);
   input I1, I2;
output O1, O2;
/*by default all the above inputs and outputs are 1-bit signals.*/
   assign O1 = I1&I2; // statement 1
   assign O2 = I1^I2; // statement 2
/*Statements 1 and 2 are continuous signal-assignment statements*/
endmodule
```

Signal Declaration and Assignment Statements:

Syntax:

signal *list_of_signal_names*: type [:= initial value];

Examples:

signal SUM, CARRY: std_logic;
signal DATA_BUS: bit_vector (0 to 7);
signal VALUE: integer range 0 to 100;

■ Signals are updated after a delta delay.

Example:

SUM <= (A xor B);

■ The result of A xor B is transferred to SUM after a delay called simulation Delta which is a infinitesimal small amount of time.

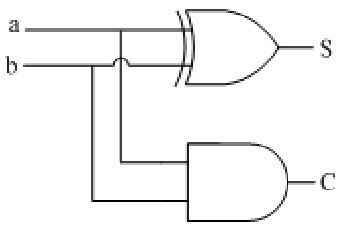
Constant:

Syntax:

constant *list_of_name_of_constant*: type [:=initial value] ;

Examples:
constant RISE_FALL_TME: time := 2 ns;
constant DELAY1: time := 4 ns;

HDL Code for Half Adder—VHDL and Verilog:



VHDL Half Adder Description

```
entity halfadder is
port (
a  :  in  bit;
b  :  in  bit;
s : out bit;
c   :   out   bit);
end halfadder;
architecture HA_DtFl of halfadder is
begin
   s <= a xor b; -- This is a signal assignment statement.
   c <= a and b; -- This is a signal assignment statement.
end HA_DtFl;
```

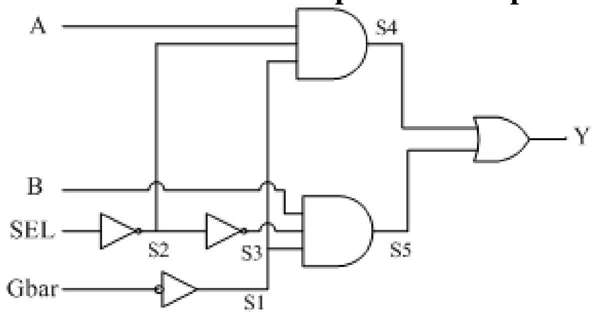
Verilog Half Adder Description

```
module  halfadder  (a,  b,  s,  c);
input a;
input   b;
output  s;
output c;
/*The default type of all inputs and outputs is a single bit. */
   assign s = a ^ b; /* This is a signal assignment statement;
                  ^is a bitwise xor logical operator. */

   assign c = a & b; /* This is a signal assignment statement
                 &   isa bitwise logical “and” operator */
endmodule
```

5.8 Data type-vectors

HDL Code of a 2x1 Multiplexer—VHDL and Verilog:

VHDL 2x1 Multiplexer Description :



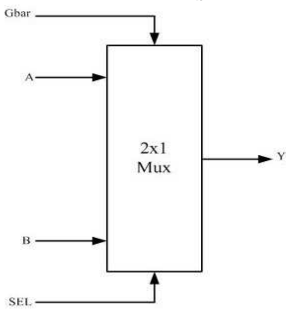


Fig: 2x1 Multiplexer (a) Logic diagram (b) Logic symbol

```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    entity mux2x1 is
    port (A, B, SEL, Gbar : in std_logic;
       Y : out std_logic);
    end mux2x1;

    architecture MUX_DF of mux2x1 is
    signal S1, S2, S3, S4, S5 : std_logic;
       Begin

    -- Assume 7 nanoseconds propagation delay
    -- for all and, or, and not.

    st1: Y <= S4 or S5 after 7 ns;
    st2: S4 <= A and S2 and S1 after 7 ns;
    st3: S5 <= B and S3 and S1 after 7 ns;
    st4: S2 <= not SEL after 7 ns;
    st5: S3 <= not S2 after 7 ns;
    st6: S1 <= not Gbar after 7 ns;
    end MUX_DF;
```

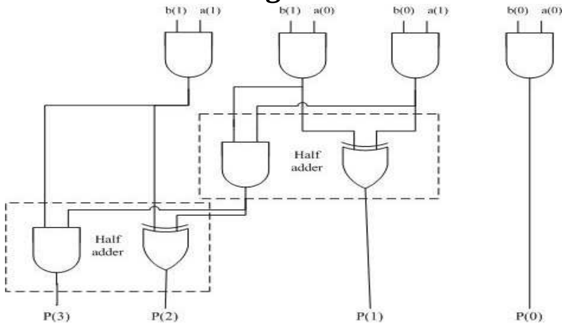
Verilog Description: 2x1 Multiplexer

```
module mux2x1 (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
```

```
output Y;
wire S1, S2, S3, S4, S5;

/* Assume 7 time units delay for all and, or, not.
In Verilog we cannot use specific time units,
such as nanoseconds. The delay here is
expressed in simulation screen units. */

assign #7 Y = S4 | S5;      //st1
assign #7 S4 = A & S2 & S1; //st2
assign #7 S5 = B & S3 & S1; //st3
assign #7 S2 = ~ SEL;      //st4
assign  #7  S3  =  ~  S2;    //st5
assign #7 S1 = ~ Gbar;      //st6
endmodule
```

HDL Code for a 2x2 Unsigned Combinational Array Multiplier—VHDL and Verilog:

VHDL 2x2 Unsigned Combinational Array Multiplier Description :



```
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    entity mult_arry is
       port (a, b : in std_logic_vector(1 downto 0);
       P : out std_logic_vector (3 downto 0));
    end mult_arry;

    architecture MULT_DF of mult_arry is
    begin
    -- For simplicity propagation delay times are not considered
    --   in   this   example.
    P(0) <= a(0) and b(0);
    P(1) <= (a(0) and b(1)) x   or (a)(1  and b(0));
    P(2) <= (a(1) and b(1)) xor ((a(0) and b(1)) and (a(1) and
             b(0)));
    P(3) <= (a(1) and b(1)) and ((a(0) and b(1)) and (a(1) and
             b(0)));
```

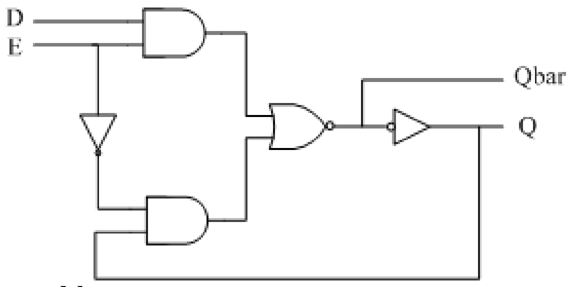
```
end MULT_DF;
```

Verilog 2x2 Unsigned Combinational Array Multiplier Description

```
module mult_arry (a, b, P);
input [1:0] a, b;
output [3:0] P;
/*For simplicity, propagation delay times are not
considered in this example.*/

assign P[0] = a[0] & b[0];
assign P[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
assign P[2] = (a[1] & b[1]) ^ ((a[0] & b[1]) & (a[1] & b[0]));
assign P[3] = (a[1] & b[1]) & ((a[0] & b[1])& (a[1] & b[0]));
endmodule
```

HDL Code for a D-Latch—VHDL and Verilog:

VHDL D-Latch Description:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is
port (D, E : in std_logic;
   Q, Qbar : buffer std_logic);
-- Q and Qbar are declared as buffer because they act as
--both input and output, they appear on the right and left
--hand side of signal assignment statements. inout or
--  linkage could have been used instead of buffer.
end D_Latch;

architecture DL_DtFl of D_Latch is
constant Delay_EorD : Time := 9 ns;
constant Delay_inv : Time := 1 ns;
begin
--Assume 9-ns propagation delay time between
--E or D and Qbar; and 1 ns between Qbar and Q.
```

```
Qbar <= (D and E) nor (not E and Q) after Delay_EorD;
Q <= not Qbar after Delay_inv;

end DL_DtFl;
```

Verilog D-Latch Description:

```
module D_latch (D, E, Q, Qbar);
input D, E;
output Q, Qbar;

/* Verilog treats the ports as internal ports,
so Q and Qbar are not considered here as
both input and output. If the port is
connected externally as bidirectional,
then we should use inout. */

time Delay_EorD = 9;
time Delay_inv = 1;
assign #Delay_EorD Qbar = ~((E & D) |
   (~E & Q));
assign #Delay_inv Q = ~ Qbar;
endmodule
```

HDL Code of a 2x2 Magnitude Comparator—VHDL and Verilog:

VHDL 2x2 Magnitude Comparator Description:

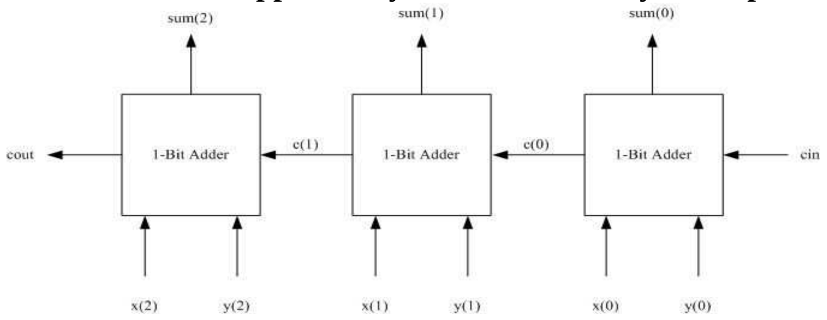
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity COMPR_2 is
port (x, y : in std_logic_vector(1 downto 0); xgty,
   xlty : buffer std_logic; xeqy : out std_logic);
end COMPR_2;

architecture COMPR_DFL of COMPR_2 is
begin
xgty <= (x(1) and not y(1)) or (x(0) and not y(1) and
        not y(0)) or
   x(0) and x(1) and not y(0));
xlty <= (y(1) and not x(1)) or ( not x(0) and y(0)
        and y(1)) or
   (not x(0) and not x(1) and y(0));
xeqy <= xgty nor xlty;
end COMPR_DFL;
```

Verilog 2x2 Magnitude Comparator Description

```
module compr_2 (x, y, xgty, xlty, xeqy);
input [1:0] x, y;
output xgty, xlty, xeqy;
assign xgty = (x[1] & ~ y[1]) | (x[0] & ~ y[1]

            & ~ y[0]) | (x[0] &  x[1] & ~ y[0]);
assign xlty = (y[1] & ~ x[1] ) | (~ x[0] & y[0] & y[1]) |
           (~ x[0] & ~ x[1] & y[0]);
assign xeqy = ~ (xgty | xlty);
endmodule
```

3-Bit Ripple-Carry Adder Case Study—VHDL and Verilog

VHDL 3-Bit Ripple-Carry Adder Case Study Description



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity adders_RL is
   port (x, y : in std_logic_vector (2 downto 0);
   cin : in std_logic;
   sum : out std_logic_vector (2 downto 0);
   cout : out std_logic);
end adders_RL;

--I. RIPPLE-CARRY ADDER

architecture RCarry_DtFl of adders_RL is
--Assume 4.0-ns propagation delay for all gates.
signal c0, c1 : std_logic;
constant delay_gt : time := 4 ns;

begin
sum(0) <= (x(0) xor y(0)) xor cin after 2*delay_gt;

--Treat the above statement as two 2-input XOR.

sum(1) <= (x(1) xor y(1)) xor c0 after 2*delay_gt;

--Treat the above statement as two 2-input XOR.
```

```
sum(2) <= (x(2) xor y(2)) xor c1 after 2*delay_gt;
--Treat the above statement as two 2-input XOR.
c0 <= (x(0) and y(0)) or (x(0) and cin) or (y(0) and cin)
   after 2*delay_gt;
c1 <= (x(1) and y(1)) or (x   (1) and0) or (y(1) and c0)
   after 2*delay_gt;
cout <= (x(2) and y(  2)) o(rx(2) and c1) or (y(2) and c1)
   after 2*delay_gt;
end RCarry_DtFl;
```

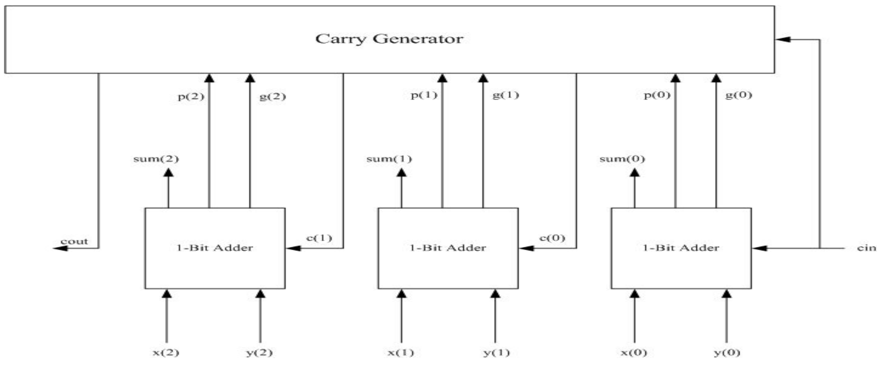
Verilog 3-Bit Ripple-Carry Adder Case Study Description

```
module adr_rcla (x, y, cin, sum, cout);
input [2:0] x, y;
input cin;
output [2:0] sum;
output cout;
// I. RIPPLE CARRY ADDER
wire c0, c1;
time delay_gt = 4;
//Assume 4.0-ns propagation delay for all gates.

assign #(2*delay_gt) sum[0] = (x[0] ^ y[0]) ^ cin;
//Treat the above statement as two 2-input XOR.

assign #(2*delay_gt) sum[1] = (x[1] ^ y[1]) ^ c0;
//Treat the above statement as two 2-input XOR.

assign #(2*delay_gt) sum[2] = (x[2] ^ y[2]) ^ c1;
//Treat the above statement as two 2-input XOR.

assign #(2*delay_gt) c0 = (x[0] & y[0]) | (x[0] & cin)
                    | (y[0] & cin);


assign #(2*delay_gt) c1 = (x[1] & y[1]) | (x[1] & c0)
                    | (y[1] & c0);

assign #(2*delay_gt) cout = (x[2] & y[2]) | (x[2] & c1)
                    | (y[2] & c1);
endmodule
```

3-Bit Carry-Lookahead Adder Case Study—VHDL and Verilog

VHDL 3-Bit Carry-Lookahead Adder Case Study Description

--II. CARRY-LOOKAHEAD ADDER



```
architecture lkh_DtFl of adders_RL is
    --Assume 4.0-ns propagation delay for all gates
    --including a 3-input xor.
    signal c0, c1 : std_logic;
    signal p, g : std_logic_vector (2 downto 0);
    constant delay_gt : time := 4 ns;
    begin

    g(0) <= x(0) and y(0) after delay_gt;
    g(1) <= x(1) and y(1) after delay_gt;
    g(2) <= x(2) and y(2) after delay_gt;
    p(0) <= x(0) or y(0) after delay_gt;
    p(1) <= x(1) or y(1) after delay_gt;
    p(2) <= x(2) or y(2) after delay_gt;
    c0 <= g(0) or (p(0) and cin) after 2*delay_gt;

    c1 <= g(1) or (p(1) and g(0)) or (p(1) and p(0)
        and cin) after 2*delay_gt;
    cout <= g(2) or (p(2) and g(1)) o   rp(2) and p(1)
        and g(0)) or
      (p(2) and p(1) and    p(0) and cin) after 2*delay_gt;

    sum(0) <= (p(0) xor g(0)) xor cin after delay_gt;
    sum(1) <= (p(1) xor g(1)) xor c0 after delay_gt;
    sum(2) <= (p(2) xor g(2)) xor c1 after delay_gt;
    end lkh_DtFl;
```

Verilog 3-Bit Carry-Lookahead Adder Case Study Description

```
// II. CARRY-LOOKAHEAD ADDER
module lkahd_adder (x, y, cin, sum, cout);
input [2:0] x, y;
input cin;
output [2:0] sum;
output cout;
/*Assume 4.0-ns propagation delay for all gates
    including a 3-input xor.*/

wire   c0,   c1;
wire [2:0] p, g;
time delay_gt = 4;
assign #delay_gt g[0] = x[0] & y[0];
assign #delay_gt g[1] = x[1] & y[1];
assign #delay_gt g[2] = x[2] & y[2];
assign #delay_gt p[0] = x[0] | y[0];
assign #delay_gt p[1] = x[1] | y[1];
assign #delay_gt p[2] = x[2] | y[2];
assign #(2*delay_gt) c0 = g[0] | (p[0] & cin);


assign #(2*delay_gt) c1 = g[1] | (p[1] & g[0]) |
   (p[1] & p[0] & cin);


assign #(2*delay_gt) cout = g[2] | (p[2] & g[1]) | (p[2] &
   p[1] & g[0]) | (p[2] & p[1] & p[0] & cin);

assign #delay_gt sum[0] = (p[0] ^ g[0]) ^ cin;
assign #delay_gt sum[1] = (p[1] ^ g[1]) ^ c0;
   assign #delay_gt sum[2] = (p[2] ^ g[2]) ^ c1;
endmodule
```