Digital Circuit Design 2
10636321

Dr. Ashraf Armoush

VHDL

Outline

- Levels of Representation and Abstraction
- Hardware Description Languages (HDLs)
- VHDL Constructs
- Libraries and Packages
- Process in VHDL
- Sequential Statements
- Hierarchical Model Layout - Component
- Variables vs. Signals
- Subprograms
- Combinational vs. Sequential Process.
- Clock Edge Detection
- Synchronous vs. Asynchronous Set/Reset
- ASM.

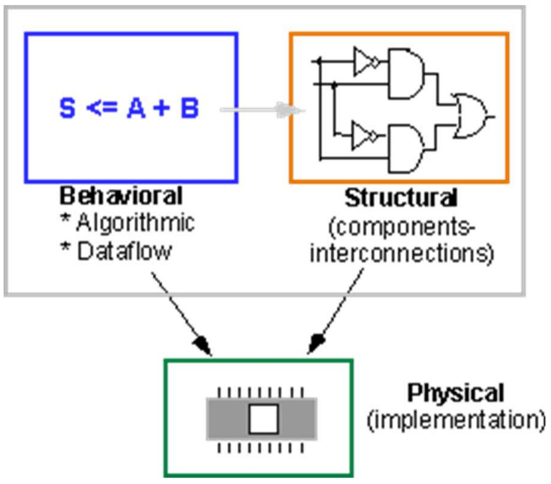
Levels of Representation and Abstraction

- A digital system can be represented at different levels of abstraction in order to keep the description and design of complex systems manageable.

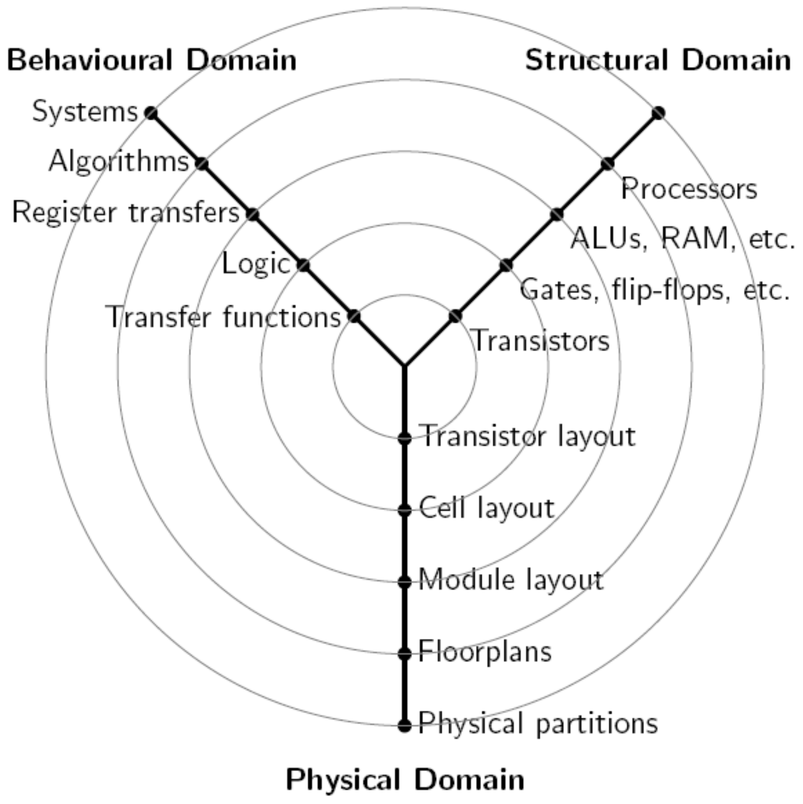
➢ **Behavioral:** describes a system in terms of what it does (or ***how it behaves***) rather than in terms of its components and interconnection between them. (*e.g. RTL , Boolean Expression, Algorithmic Level*)

➢ **Structural:** describes a system as **a collection of gates and components** that are interconnected to perform a desired function.

➢ **Physical:** describes the physical implementation of the system and the final result of the design process.



Levels of Representation and Abstraction (cont.)

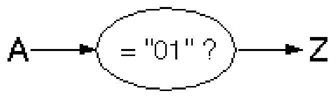


Hardware Description Languages (HDLs)

- Hardware description languages (HDLs) are used to describe the structure and behavior of digital electronic systems.
- HDLs were developed to deal with increasingly complex designs.
- Advantages (from previous slide):
  1. **Verification and simulation**: you can verify design functionality early in the design process and immediately simulate a design written as an HDL description.
  2. **Synthesis an Optimization**: you can automatically convert a HDL description to a gate-level implementation in a given technology.
  3. **Documentation**: HDL descriptions supply technology-independent documentation of a design and its functionality.
  4. **Type checking**: like most high-level software languages, provides strong type checking.
  5. **Re-use** of Intellectual Property.

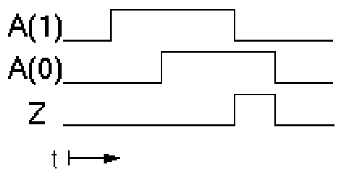
Why do we describe systems (using a description language)?

- **<u>Design Specification</u>**:
  - unambiguous definition of components and interfaces in a large design

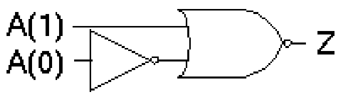


```
Z <= '1' when A="01"
    else '0' ;
```

- **<u>Design Simulation</u>**:
  - verify system/subsystem/chip performance prior to design Implementation.



- **<u>Design Synthesis</u>**:
  - *automated generation of a hardware design*



VHDL and the other hardware description languages allow one to describe a digital system at the structural or the behavioral level.

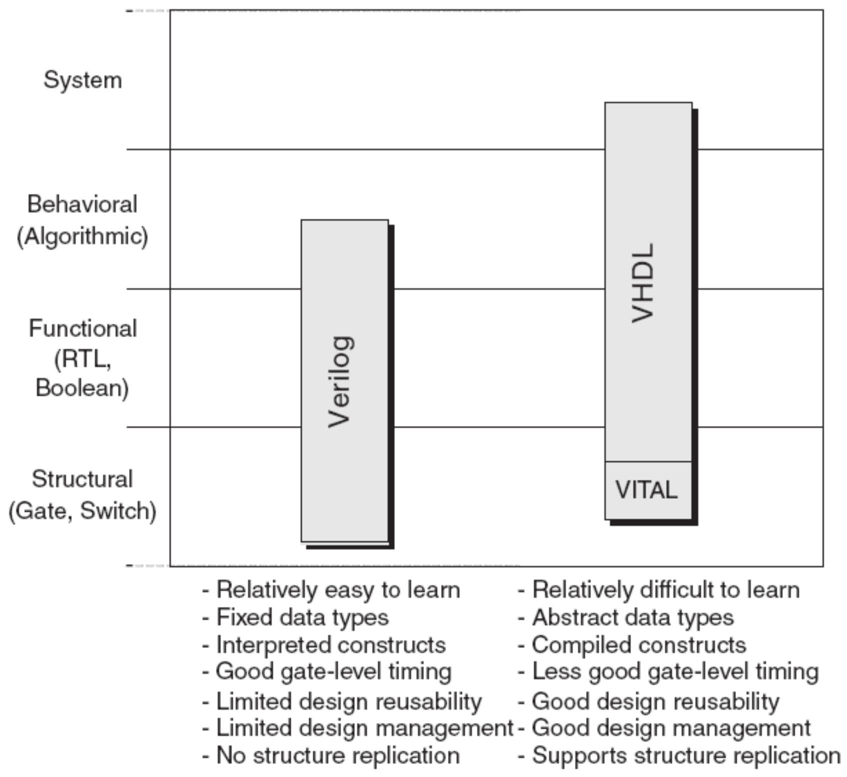
Design HW as if it is SW

Software Design (Microcontroller)	Hardware Design (CPLD/FPGA)
➢Specification	➢Specification
➢Implementation (e.g.: C or assembly)	➢Implementation in HDL ▪Structural description ▪Behavioral description
➢Compilation to machine code	➢Automatic transformation in Gate Level Description (Synthesis)
➢Load code in program memory of target	➢Load configuration in target
➢Functionality is realized by execution of code by CPU (CPU can use certain peripherals as timers)	➢Functionality is implemented in hardware

HDLs (Examples)

- **Verilog**
  - Invented in 1984
  - Syntax similar to C
- **VHDL**
  - Was originally developed for the US Department of Defense (DoD)
  - Syntax similar to ADA
  - VHDL is more complex than Verilog.
- **System C**
  - C++ library for hardware specific constructs
  - Good for Simulation, but synthesis still problematic.

Verilog vs. VHDL



- Relatively easy to learn	- Relatively difficult to learn
- Fixed data types	- Abstract data types
- Interpreted constructs	- Compiled constructs
- Good gate-level timing	- Less good gate-level timing
- Limited design reusability	- Good design reusability
- Limited design management	- Good design management
- No structure replication	- Supports structure replication

Levels of abstraction (Verilog versus VHDL).

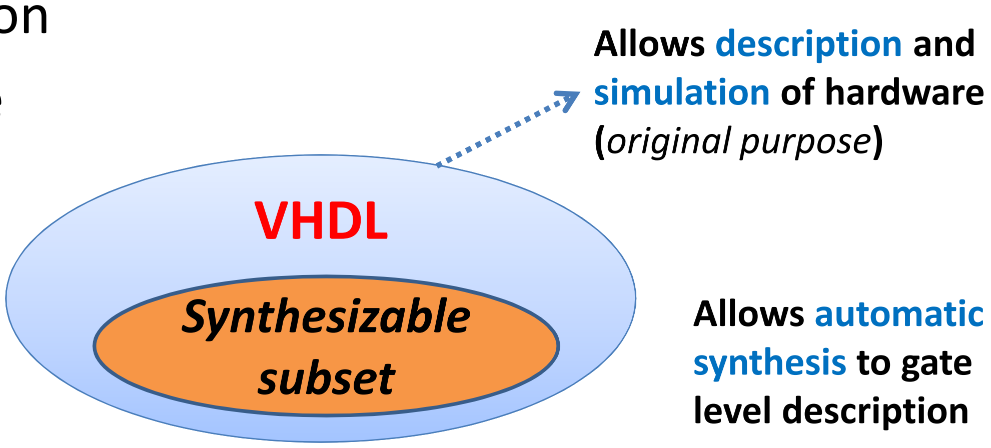
VHDL

Very High Speed Integrated Circuit (VHSIC)

Hardware

Description

Language



Allows description and simulation of hardware (original purpose)
VHDL
Synthesizable subset
Allows automatic synthesis to gate level description

- Not all constructs in VHDL are suitable for synthesis (e.g. *wait for 10 ns*).
- This VHDL subset is not standardized. (why?)

VHDL – History

- **1981**: The development of ***VHDL*** was initiated by ***US DoD*** in order to document the behavior of ***ASIC***s that were included in the supplied equipments.
- **1983**: a team of ***Intermetrics***, ***IBM*** and ***Texas Instruments*** were awarded a contract to develop ***VHDL***
- **1985:** the official release of the final version of the language was released.
- **1987:** (***VHDL-87***) the first ***IEEE*** standardized version was adopted (*IEEE -1076*)
- **1993**: (***VHDL 93***) is the first ***IEEE*** revision. It is still the most widely supported version of ***VHDL***.
- **2000:** (***VHDL 2000***): added the idea of protected types (similar to the concept of class in ***C++)***
- **2002**: (***VHDL-2002***) is a minor revision of ***VHDL 2000*** Edition
- **2008**: (**VHDL-2008**) *addressed more than 90 issues discovered during the trial period for the previous version*)
- **2019**: (**VHDL-2019**) The VHDL standard *IEEE 1076-2019 was approved*

VHDL Constructs

- **Entity**: specifies inputs and outputs of each module
- **Architecture**: specifies the structure or the behavior of a module
- **Process**: can be used for description of the behavior
- **Signal**: can be understood as physical connections
- **Variable**: can be understood as memory cell

- VHDL language constructs are divided into three categories by their level of abstraction:
  - ***Behavioral:*** (functional aspects)
  - ***Dataflow:*** (the data flow from input to output)
  - ***Structural:*** (a model where the components of a design are interconnected)

VHDL Basic Constructs

There are 2 basic constructs that are required by every VHDL file:

- **Entity** declaration :
  - Serves as an interface to other designs
  - Defines the inputs and the outputs ports of a design
  - A design can contain more than one entity.
  - Each entity has its own architecture statement.
- **Architecture** body:
  - Defines the relationship between the inputs and the outputs
  - Determines the implementation of an entity

Entity

- *The syntax follows*.

```
entity entity_name is  [ generic (generic_declarations );]

            [ port ( port_declarations ) ;]

end [ entity_name ] ;
```

[] : optional

- **entity_name** is the name of the entity.
- **generic_declarations** determine local constants used for sizing or timing the entity.
- **port_declarations** determine the number and type of input and output ports.

Entity ***Port*** *Specifications*

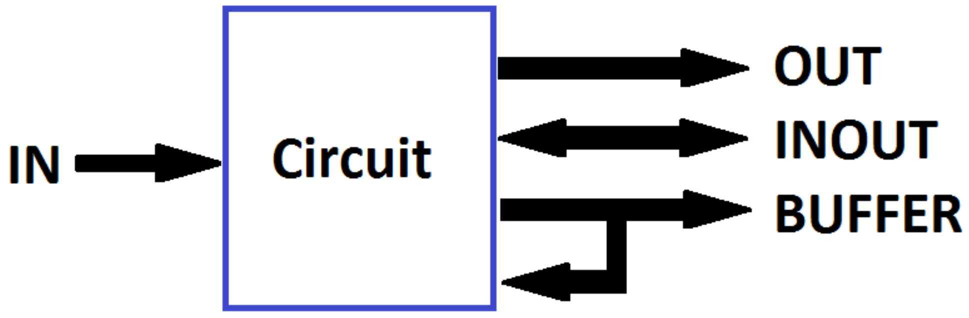
- A **port** in VHDL is a connection from a VHDL design entity to the outside world
- Port specifications define the number and type of ports in the entity.

```
port(
port_name : mode port_type
{ ; port_name : mode port_type}
);
```

1. **port_name**: is the name of the port.
2. **mode :** specifies the direction of information transfer
3. **Port_type:** defines the range of values that the port can have.

VHDL Port Modes

- **in**: can only be read
- **out**: can only be assigned a value
- **inout**: can be read and assigned a value
- **buffer**: is similar to out but can be read. The value read is the assigned value.



Data Type

- Determines the values that an object (e.g. constant, signal, variable, function, and parameter) can have.
- **BIT** , **BIT_VECTOR**, and **INTEGER** are the most common used types.
- **BIT**: can have two values '1' and '0'
  Ex: ***X : in bit;***
- **BIT_VECTOR**: is a ***one dimensional array***

ex: ***D : in bit_vector(3 downto 0)***

D(3)	D(2)	D(1)	D(0)

D : in bit_vector(0 to 3)

D(0)	D(1)	D(2)	D(3)

❖ *Data Types in VHDL will be covered later in detail*

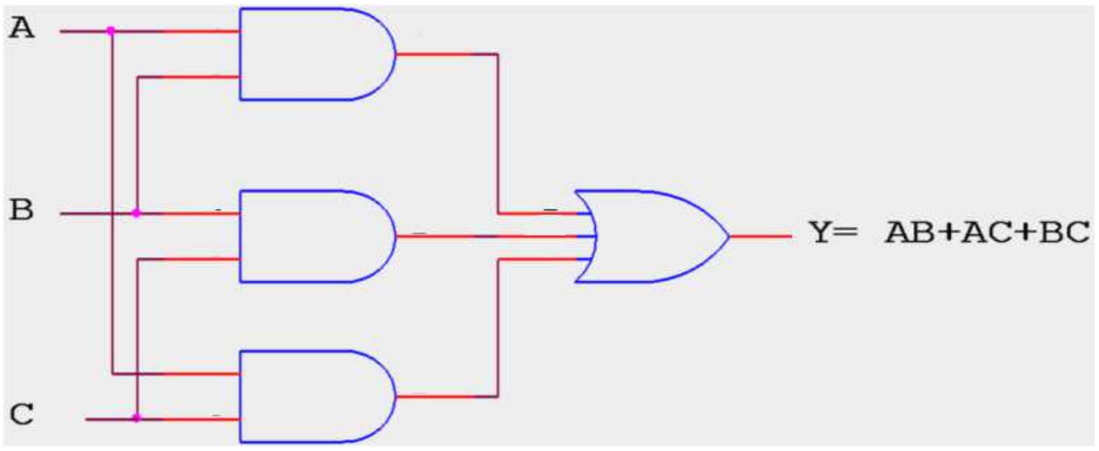
Architecture

- *The syntax follows.*

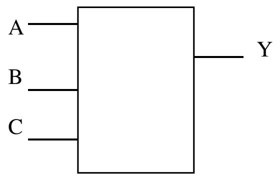
```
architecture architecture_name of entity_name is
{ block_declarative_item }
begin
{ concurrent_statement }
end [ architecture_name ] ;
```

- **architecture_name** is the name of the architecture.
- **entity_name** is the name of the entity being implemented.
- **block_declarative_item** is any of the following statements: *Subprogram Declarations, Subprogram Body, Type Declarations, Subtype Declarations, Constant Declarations, Signal Declarations, Concurrent Statements, etc.*

Ex1: Majority Vote



Logic Circuit for majority vote



Symbol for VHDL majority vote

Ex1: (cont.)

Entity name

```
entity Majority_Vote is

          port(         A, B, C :in   bit;
                              Y     :out          bit);
end Majority_Vote;
```

Entity Declaration

```
architecture Maj_Vote  of  Majority_Vote is
begin
                    Y <= (A and B) or (B and C) or (A and C);
end Maj_Vote;
```

Architecture body

Architecture name

Notes

- Statements in VHDL are terminated by semicolons.
- This operator (**<=**) is called "assignment operator".
- Comments in VHDL start with two adjacent hyphens (‘**--**’)
- The Boolean operators (**AND, OR, NOT, NAND, NOR, XOR,** and **XNOR**) have an equal order of precedence.
- The VHDL statements are case **insensitive**.
- The VHDL file name must be the same as the top-level entity name.
- A valid name in VHDL consists of a letter followed by any number of letters or numbers, without spaces. An underscore can be used within a name, but can not begin or end the name.
- You can define a **literal constant** to be used within an entity with the **generic** declaration, which is placed before the port declaration within the entity block

generic (name : type := value);

Signals Assignment (<=)

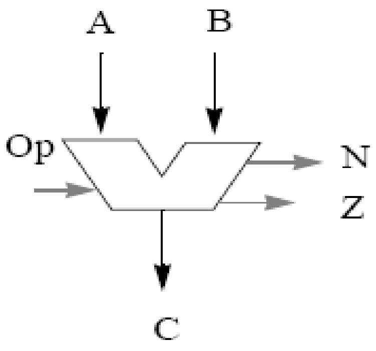
- **A : out bit;**

```
A <= '0';
A <= '1';
```

- **D : out bit_vector (3 downto 0);**

D(3) <= '0';
D(2) <= '1';
D(1) <= '0';
D(0) <= '1';
	←→	d <= "0101";

Ex2:



```
entity ALU32 is
          port( A, B: in    bit_vector (31 downto 0);
                    C : out bit_vector (31 downto 0);
                   Op: in    bit_vector ( 5  downto 0);
                 N, Z: out  bit);
end ALU32;
```

Ex3: Implement the following function:

$Y = D_3`D_2`D_1D_0 + D_3`D_2D_1D_0` + D_3D_2`D_1`D_0 + D_3D_2D_1`D_0`$

- This SOP can be represented by the following truth table.

D3	D2	D1	D0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Ex3: (cont.) : with/select

- The truth table can be implemented using " **Selected Signal Assignment Statement"**

```
entity select_example is
          port (        d  :in   bit_vector(3 downto 0);
                         y :out  bit);
end select_example;
```

```
architecture cct  of select_example is
begin
              with d select
              y <= '1'   when "0011",
                    '1'   when "0110",
                    '1'   when "1001",
                    '1'   when "1100",
                    '0'   when others;
end cct;
```

Select the value of y based on the value of d

Libraries and Packages

- ***Package**: is a collection of declarations of commonly used objects, data types, component declarations, signal, procedures and functions that can be shared among different VHDL designs.*

- ***Library**: can be considered as a place where the compiler stores information about a design project and packages.*

  – Much of the power of VHDL comes from the use of predefined libraries and packages.

Libraries and Packages (cont.)

- To make a package visible to the design, two declarations are needed:
  – One for the library where the package is located.
  – The other a ***use*** clause pointing to the specific package.

```
LIBRARY  LIBRARY_NAME;
USE LIBRARY_NAME.PACKAGE_NAME.ALL;
```

```
library IEEE;
use IEEE.std_logic_1164.ALL ;   -- Defines the standard data types
```

- There are two standard libraries:
  – ***IEEE***
  – ***STD***

Library IEEE

- Standard Packages
  - use IEEE.**std_logic_1164**

    *Defines the 9-value data types **STD_ULOGIC** and **STD_LOGIC***
  - use IEEE.**numeric_std**

    *Introduces the type **SIGNED** and **UNSIGNED** and the corresponding operators, having **STD_LOGIC** as the base type.*
  - use IEEE.**numeric_bit**

    *Introduces the type **SIGNED** and **UNSIGNED** and the corresponding operators, having **BIT** as the base type.*
  - use IEEE.**numeric_std_unsigned**
  - use IEEE.**numeric_bit_unsigned**

  .

  .

Library IEEE (cont.)

- <u>Nonstandard</u> Packages
  - use IEEE.**std_logic_arith**

    *Defines the type **SIGNED** and **UNSIGNED** and the corresponding operators. This package is partially equivalent to* IEEE.numeric_std
  - use IEEE.**std_logic_signed**

    Introduces functions that allow arithmetic, and some shift operations with signals of type STD_LOGIC_VECTOR operating as signed numbers.
  - use IEEE.**std_logic_unsigned**

    Same as above , but operating as unsigned numbers

The last two packages can be considered as complements to the package std_logic_1164, because the latter does not contain arithmetic and comparison operators for the type *STD_LOGIC_VECTOR*

STD_LOGIC and STD_LOGIC_VECTOR (IEEE.std_logic_1164 Multi-valued LOGIC)

Any port, signal, or variable of type **STD_LOGIC** or **STD_LOGIC_VECTOR** can have any of the following 9 values:

'U'	Uninitialized (default initial value)
'X'	Forcing Unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	**High Impedance**
'W'	Weak Unknown
'L'	Weak 0 (pull down to 0)
'H'	Weak 1 (pull up to 1)
'-'	**Don't care** (for synthesis purposes only)

Ex4: (2-to-4 Decoder)

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

entity decoder is
   Port ( A : in  STD_LOGIC_VECTOR (1 downto 0);
              Y : out  STD_LOGIC_VECTOR (3 downto 0));
end decoder;
```

```
architecture Behavioral of decoder is
begin
                        with A SELECT
                        Y <=      "0001" when "00",
                                  "0010" when "01",
                                  "0100" when "10",
                                  "1000" when "11",
                                  "0000" when others;
end Behavioral;
```

Circuits with Don’t Care Outputs

- Conditional signal assignment (when / else)

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

entity circuit is
    Port (  X : in  STD_LOGIC_VECTOR (1 downto 0);
            Y : out  STD_LOGIC_VECTOR (1 downto 0));
end circuit;
```

X1	X0	Y1	Y0
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**0**	**1**
1	1	**-**	**-**

```
architecture Behavioral of circuit is
begin
       Y <=    "00"   when X="00"    else
               "01"   when X="10"    else
               "10"   when X="01"    else
               "--";
end Behavioral;
```

Signals

- A signal is like an internal wire connecting two or more points inside an architecture body.
- It is declared before the BEGIN statement of an *architecture body*

signal *signal_name:* ***signal_type*** *[****:=default_value****];*

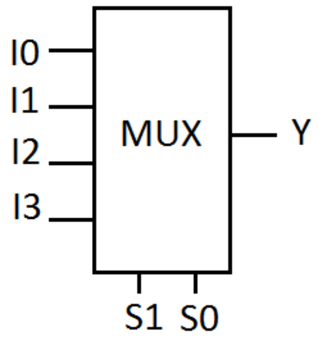
Ex:

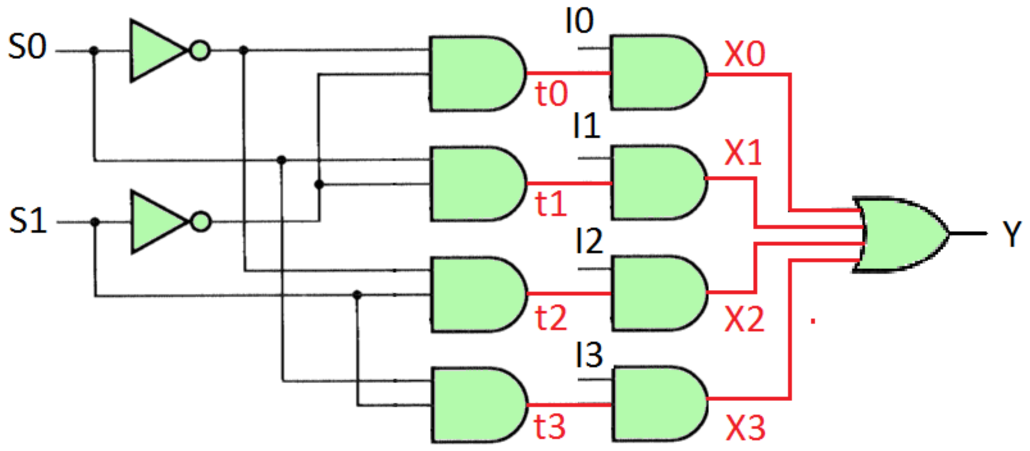
signal *flag : STD_LOGIC :=* **'0'** **;**

- It is global to the architecture.
- Its value is assigned with the **<=** operator.
- Statements within architecture blocks, to this point, are executed **concurrently** (at the same time). So they are called concurrent signals.

__signal <= __expression;

Ex5: (Multiplexer)





```
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 mux_arch of mux is
                signal  t : std_logic_vector (3 downto 0);
                 signal  X : std_logic_vector (3 downto 0);
begin
              t(3) <= s(1) and s(0);
              t(2) <= s(1) and not s(0);
              t(1) <= not s(1) and s(0);
              t(0) <= not s(1) and not s(0);
              X <= I and t;
               Y <= X(0) or X(1) or X(2) or X(3);
end mux_arch;
```

Keyword OTHERS

- **others** is a useful keyword for making assignments.
- It represents all index values that were left unspecified.

```
signal  c : STD_LOGIC_VECTOR (1 to 8) := "00000000" ;
or
signal  c : STD_LOGIC_VECTOR (1 to 8) := (others=> '0') ;
```

```
signal  d : STD_LOGIC_VECTOR (1 to 8) := "01100000" ;
or
signal  d : STD_LOGIC_VECTOR (1 to 8) := (2|3 => ‘1' , others=> '0') ;
```

```
signal  d : STD_LOGIC_VECTOR (1 to 16) := "1111111100000000" ;
or
signal  d : STD_LOGIC_VECTOR (1 to 16) := (1 to 8 => ‘1' , others=> '0') ;
```

Standard Data Types

- Include the synthesizable data Types
  - **BIT**
  - **BIT_VECTOR**
  - **BOOLEAN**
  - BOOELAN_VECTOR
  - INTEGER
  - **NATURAL**
  - POSITIVE
  - INTEGER_VECTOR
  - CHARACTER
  - STRING

Standard Data Types (cont.)

- **BIT:**
  - Two values: *'0'* and *'1'*
  - It supports logical and comparison operations
- **BIT_VECTOR**
  - The vector form of BIT
  - It supports logical, comparison , shift , and concatenation operations

```
signal  a, b : BIT_VECTOR (7 downto 0) ;
signal  x, y : BIT_VECTOR (7 downto 0) ;
signal  v : BIT_VECTOR (15 downto 0) ;
signal  w : BIT;
x <= "11110000" ;
y <= a XOR b;
b <= a SLL 2;
w <= '1'  when a>b  else '0‘ ;
v <=  a & b;  -- concatenation
```

Standard Data Types (cont.)

- **BOOLEAN:**
  - Two values: *FALSE* and *TRUE*
  - It supports logical and comparison operations
- **BOOLEAN_VECTOR**
  - The vector form of BOOLEAN
- **INTEGER**
  - The maximum range of a VHDL integer type is $-(2^{31}-1)$ to $2^{31}-1$
  - The Actual bounds are referred to as INTEGER'HIGH and INTEGER'LOW
  - In VHDL code for synthesize, it is important to always specify the range.
  - It supports arithmetic and comparison operations

```
signal  a : INTEGER range 0 to 15;     -- 4 bits
signal  b : INTEGER range -15 to 15;  -- 5 bits
signal  x : INTEGER range -31 to 31;   -- 6 bits
signal  y : BIT;
x <= a + b;
y <=  '1'  when a>b  else '0' ;
```

Standard Data Types (cont.)

- NATURAL:
  - NON-negative integer
  - It is a subtype of INTEGER (supports the same operations)

```
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGHT
```

- POSITIVE:
  - POSITIVE integer
  - It is a subtype of INTEGER (supports the same operations)

```
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGHT
```

- INTEGER_VECTOR:
  - The vector form of INTEGER
  - Introduced in VHDL 2008 is a subtype
- CHARACTER:
  - A 256-symbols enumerated type;
  - It supports only comparison operation.

```
signal char1 : CHARACTER := 'A';
```

Standard Data Types (cont.)

- STRING:
  - The vector form of CHARACTER
  - It supports comparison and concatenation operations
- REAL
  - Floating-point numbers..
  - It supports arithmetic and comparison operations
- TIME
  - Represented by integers with the same range as INTEGER
  - It supports arithmetic and comparison operations
- TIME_VECTOR:
  - The vector form of TIME
  - Introduced in VHDL 2008

Arithmetic Operations on STD_LOGIC_VECTOR

- Type **STD_LOGIC_VECTOR** is not defined as a numeric representation.
- No arithmetic operators are defined for it in package ***IEEE.std_logic_1164***
- There are two industry packages (Nonstandard) that can be used to interpret **STD_LOGIC_VECTOR as signed or unsigned numbers.**
  - ❑ IEEE.**std_logic_signed**
  - ❑ IEEE.**std_logic_unsigned**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity multiplier is
        port(a, b:  in  STD_LOGIC_VECTOR (3 downto 0);
                    y: out STD_LOGIC_VECTOR (7 downto 0));
end multiplier;
architecture behavioral of multiplier IS
begin
      y <=  a  * b;   -- they are implicitly considered as unsigned numbers
end  behavioral ;
```

Unsigned and Signed Data Types

- UNSIGNED and SIGNED data types are defined in two packages
  - ***IEEE.numeric_std*** *(Standard Package)*
  - ***IEEE.std_logic_arith*** *(Nonstandard Package)*
- Each type is a one-dimensional unconstrained array of std_logic.
- To make use of these types, one of the packages above must be declared in the code.

```
library IEEE;
use IEEE.numeric_std.all;
entity multiplier is
        port(a, b: in unsigned (3 downto 0);
                    y: out unisgned (7 downto 0));
end multiplier;
architecture behavioral of multiplier IS
begin
      y <=  a  * b;
end  behavioral ;
```

Conversion Between STD_LOGIC_VECTOR, UNSIGNED, and SIGNED [TYPE CASTING]

- Although types STD_LOGIC_VECTOR, UNSIGNED, and SIGNED all have elements that are type std_logic, they are different types. (Nevertheless , they are closely related)
- A value of one type **cannot be assigned** to one of the others.
- Type conversion between closely related types is accomplished by simply using the name of the target type as if it were a function.

```
signal  x : std_logic_vector (3 downto 0);
signal  y : unsigned (3 downto 0);

x <= y ;    --  illegal assignment, type conflict
y <= x ;    --  illegal assignment, type conflict

x <= std_logic_vector (y) ;    --  valid assignment
y <= unsigned (x) ;            --  valid assignment
```

Type Conversion Functions

- The VHDL packages provide type conversion functions:

e.g.: Conversion to and from INTEGER

From	***To***	***Function***	***Package of origin***
INTEGER	STD_LOGIC_VECTOR	conv_std_logic_vector( )	std_logic_arith

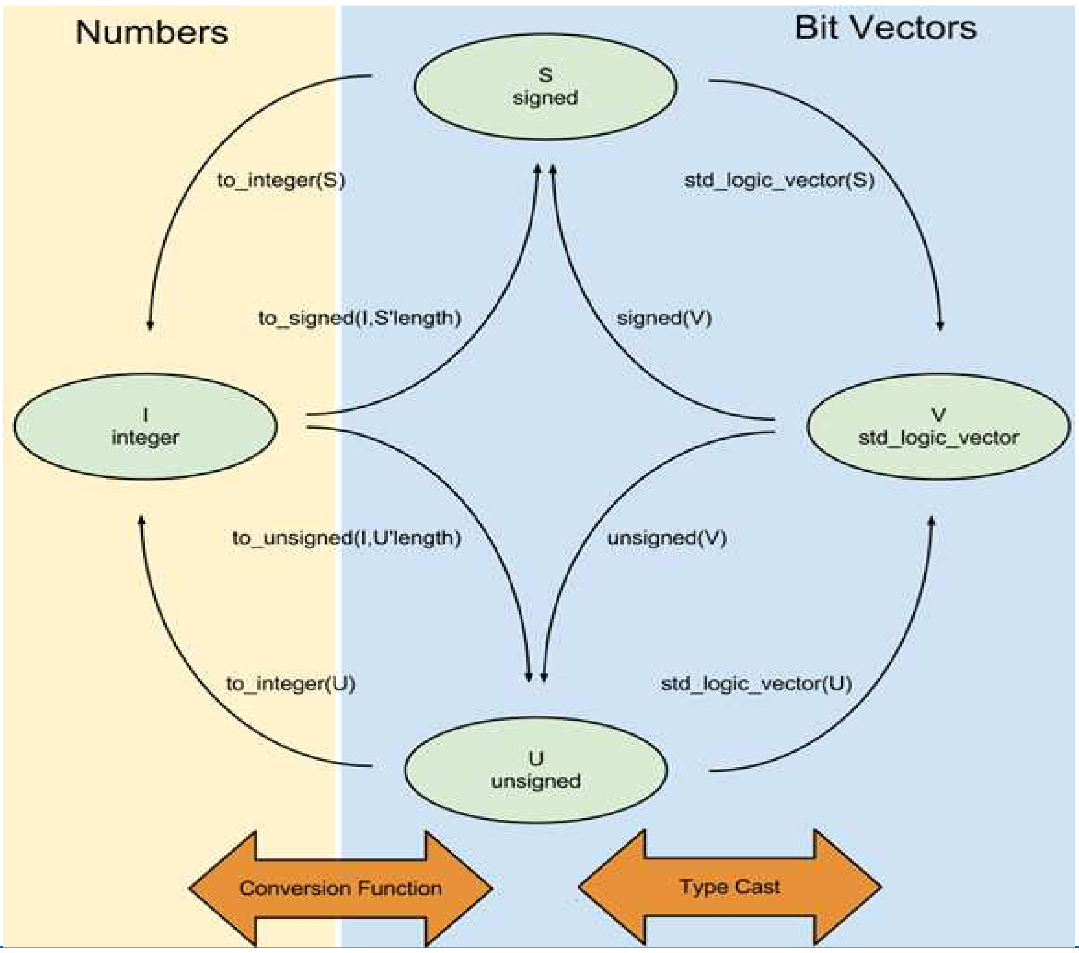
	UNSIGNED	to_unsigned( )
		conv_unsigned( )
			numeric_std
			std_logic_arith

	SIGNED	to_signed( )
		conv_signed( )
			numeric_std
			std_logic_arith

STD_LOGIC_VECTOR	INTEGER	conv_integer( )
		conv_integer()
		to_integer()
			std_logic_signed
			std_logic_unsigned
			numeric_std_unsigned

UNSIGNED	INTEGER	to_integer( )
		conv_integer( )
			numeric_std
			std_logic_arith

SIGNED	INTEGER	to_integer( )
		conv_integer( )
			numeric_std
			std_logic_arith



Array Types

- An array is an object that is a collection of elements of the same type.
- VHDL supports N-dimensional arrays, but Foundation Express supports only one-dimensional arrays.

```
type BYTE is array (7 downto 0) of BIT;
```

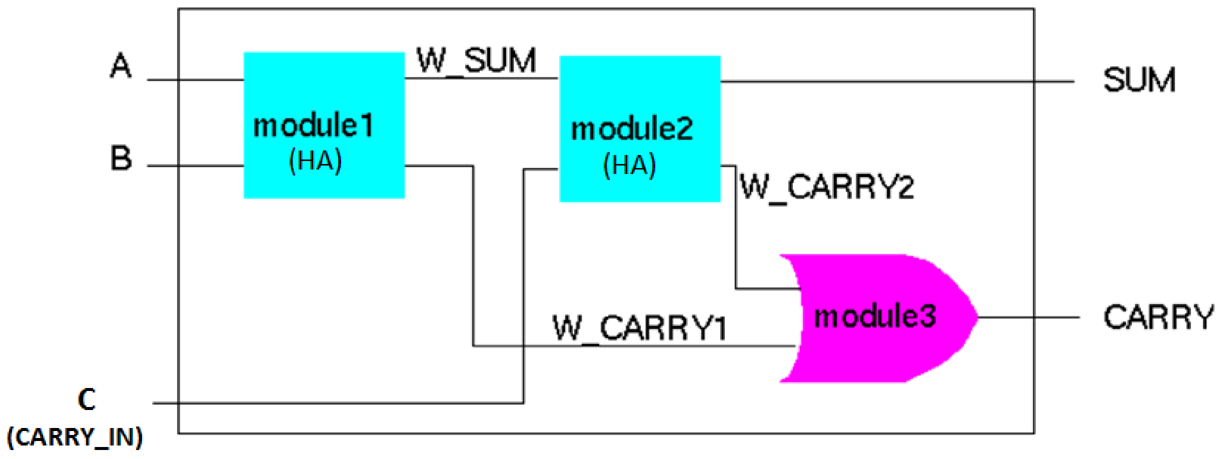
Hierarchical Model Layout

1. **Component:** a complete VHDL design entity that can be used as a part of a higher-level file in a hierarchical design.

2. **Component Declaration Statement:** a statement that defines the input and output port names of a component used in a VHDL design entity

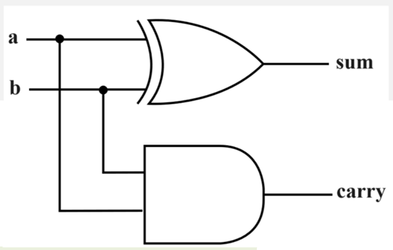
3. **Component Instantiation Statement**: a statement that maps port names of a VHDL component to the port names, internal signals, or variables of a higher level VHDL design entity.

Ex:



Full adder: 2 halfadders + 1 OR-gate

Half Adder:



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity HALFADDER is
        port (   a, b              : in    std_logic;
                 sum, carry        : out   std_logic);
end  HALFADDER;
```

```
architecture Halfadder_Arch  of HALFADDER  is
begin
        sum <= a XOR b;
        carry <=  a AND b;
end Halfadder_Arch;
```

Component Declaration

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FULLADDER is
        port (   a, b, c           : in    std_logic;
                 sum, carry        : out   std_logic);
end  FULLADDER;
```

```
architecture Fulladder_Arch  of FULLADDER  is

    signal W_SUM, W_CARRY1, W_CARRY2 : std_logic ;

    component  HALFADDER
      port (  a, b            : in   std_logic;
              sum, carry      : out std_logic);
     end component;

Begin
-
-
-
end Fulladder_Arch;
```

- Every component declaration in has to correspond to an entity.
- Components declared in an architecture are local to that architecture.

Component Declaration come from:

- – The same VHDL source file
- – A different VHDL source file

Component Instantiation (method 1)

```
architecture Fulladder_Arch  of FULLADDER  is
            -
            -
Begin

MODULE1: HALFADDER
            port map (    a       => a,
                          sum     => W_SUM,
                          b       => b,
                          carry   => W_CARRY1 );

MODULE2: HALFADDER
            port map (    a       => W_SUM,
                          sum     => sum,
                          b       => c,
                          carry    => W_CARRY2 );

carry <= W_CARRY1 or W_CARRY2;

end Fulladder_Arch;
```

2 Instantiations

*- The port names from the component declaration, which are also called "**formals**", are associated with an arrow '**=>** ' with the signals and ports of the actual entity .*

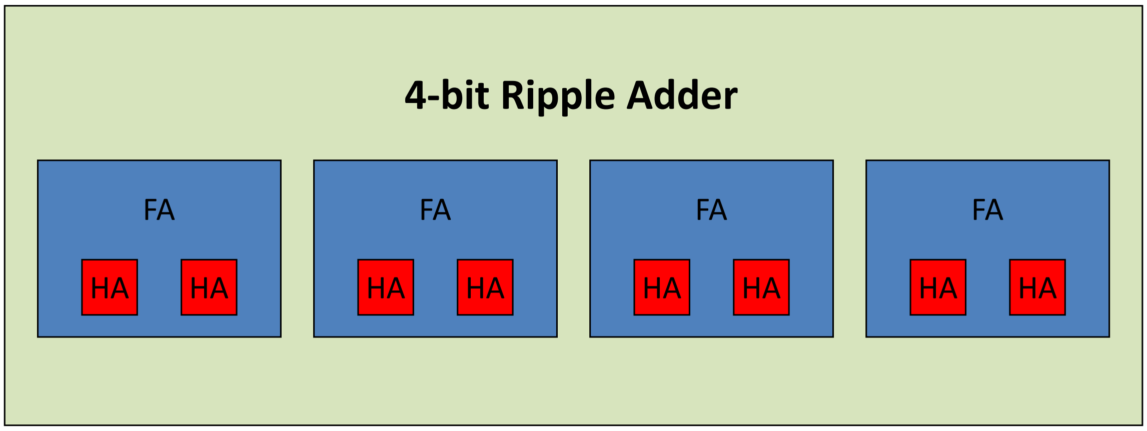
- Each component instance is given a unique name (label)

Component Instantiation (method 2)

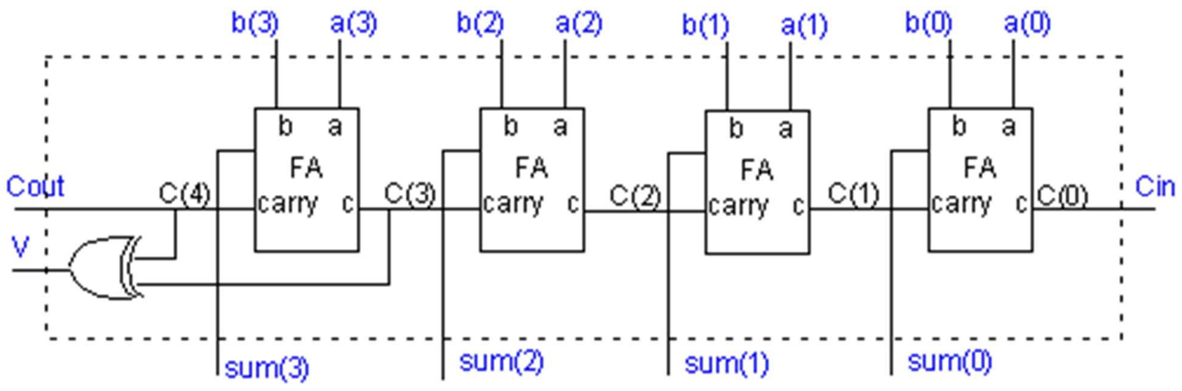
```
architecture Fulladder_Arch  of FULLADDER  is
            -
            -
Begin

MODULE1: HALFADDER
            port map (    a , b , W_SUM,  W_CARRY1 );


MODULE2: HALFADDER
            port map (    W_SUM, c , sum , W_CARRY2 );


carry <= W_CARRY1 or W_CARRY2;


end Fulladder_Arch;
```

The signal position must be in the same order as the declared component's ports

Ex: 4-bit Parallel Adder with Ripple Carry



Ex: (cont.)



- *V: (Indicate overflow condition when adding signed numbers)*

Ex: (cont.)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FOURBITADDER is
          port (    a , b       : in    std_logic_vector (3 downto 0);
                    Cin         : in    std_logic;
                    sum         : out  std_logic_vector (3 downto 0);
                    Cout , V    : out  std_logic);
end  FOURBITADDER;
```

```
architecture FourBitAdder_Arch  of FOURBITADDER  is

     signal c: std_logic_vector (4 downto 0);
     component  FULLADDER
        port (   a, b , c              : in   std_logic;
                 sum, carry            : 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));

       V <= c(3) xor c(4);

       Cout <= c(4);

end FourBitAdder_Arch;
```

Generate Statement

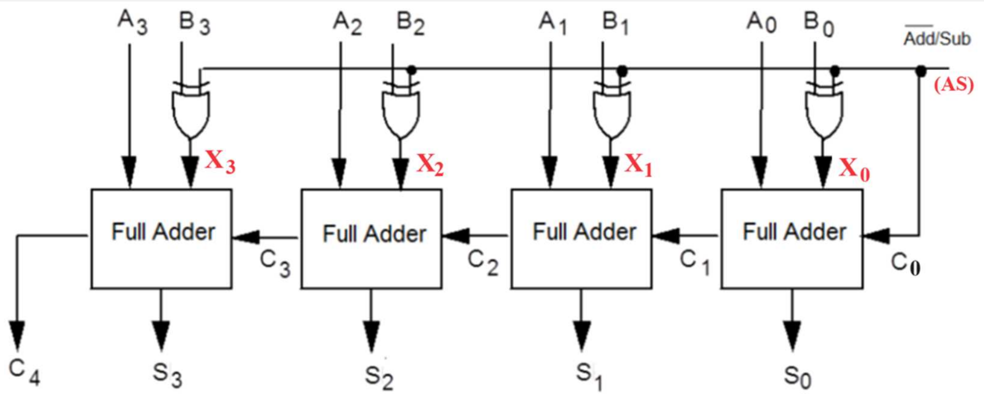
- **Generate:** is a VHDL construct that is used to create ***repetitive portions of hardware .***

```
label:
for identifier in  range generate

{ concurrent_statement }

end generate [ label: ] ;
```

Ex: 4-Bit Adder

```
architecture FourBitAdder_Arch  of FOURBITADDER  is

        signal c: std_logic_vector (4 downto 0);
        component  FULLADDER
             port (   a, b , c              : in   std_logic;
                        sum, carry          : out std_logic);
         end component;
Begin
                 c(0) <= Cin;
                adders:
                for i in 0 to 3generate
                                adder: FULLADDER
                                    port map (a(i), b(i), c(i), sum(i), c(i+1));
                end generate;

                 V <= c(3) xor c(4);
                 Cout <= c(4);
end FourBitAdder_Arch;
```

Ex: 4-Bit Adder/Subtractor



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Add_Sub_4BIT is
        port (    A , B       : in    std_logic_vector (3 downto 0);
                  AS          : in    std_logic;
                  S           : out   std_logic_vector (3 downto 0);
                  Cout        : out   std_logic);
end  Add_Sub_4BIT;
```

Ex: 4-Bit Adder/Subtractor

```
architecture Add_Sub_Arch  of Add_Sub_4BIT  is
    signal X: std_logic_vector (3 downto 0);
    signal C: std_logic_vector (4 downto 0);
    component  FULLADDER
        port (  a, b , c          : in   std_logic;
                sum, carry        : out std_logic);
    end component;
Begin
        C(0) <= AS;
        adders:
        for i in 0 to 3 generate

                X(i)  <=  AS XOR B(i) ;

                adder: FULLADDER
                    port map (A(i), X(i), C(i), S(i), C(i+1));
        end generate;
        Cout <= c(4);
end Add_Sub_Arch;
```

Entity Generic Specifications

```
generic (
            constant_name     :  type  [  := value  ]
            { ; constant_name :  type  [  := value  ] }
);
```

- **Generic** specifications are entity parameters.
- Inside an entity, a **generic** is a constant value. *(e,g for sizing or timing)*
- A **generic** can have a default value (*default value is optional*).

```
entity counter32 is
generic ( WIDTH : integer := 32);   -- default value is 32
port (      CLK, RESET, LOAD   : in      std_logic;
            DATA      : in unsigned (WIDTH-1 downto 0);
            Q         : out unsigned(WIDTH-1 downto 0));
end entity counter32;
```

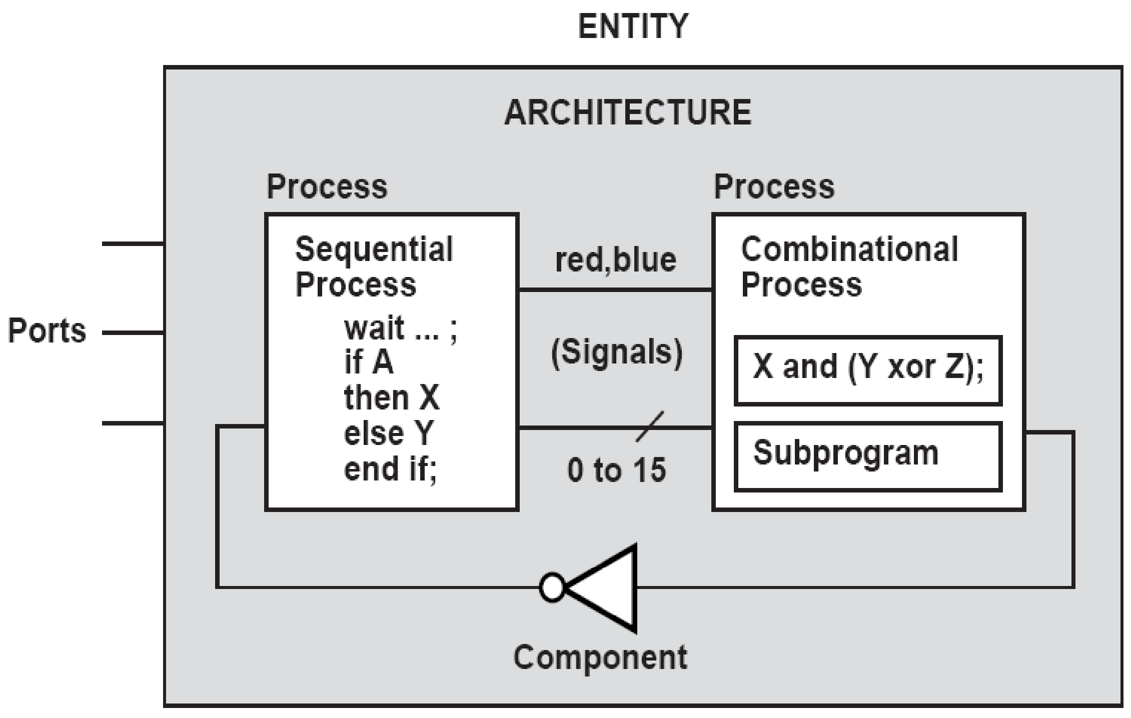
- When an entity is used as a component in a higher level design , It receives a *nondefault* value only when the entity is instantiated

Mapping Generic Values

```
architecture ...
      component  ADD
        generic (N: POSITIVE);
         port (   X, Y        : in   std_logic_vector (N-1 downto 0);
                  Z           : out std_logic_vector (N-1 downto 0);
                        C     : out std_logic);
      end component;
Begin
...
```

- When you instantiate a component with ***generics***, you can map ***generics*** to ***values***.
- A generic without a default value must be instantiated with a **generic map** value.

```
    ...
       U1:     ADD
               generic map (N => 4)
               port map (X, Y, Z, CARRY...);
```



VHDL Hardware Model

Process

- A process statement is a concurrent statement which can be used in an architecture body or block.
- It is made up of **sequentially executed statements**
- Processes are unique in that they behave like concurrent statements to the rest of the design, but they are internally sequential.
  - ➔ *Several processes run concurrently*
- A process communicates with the rest of the design by *reading values from* or *writing them to* signals or ports outside the process.

```
[ label: ] process [ (  sensitivity_list ) ]
        { process_declarative_item }
begin
        { sequential_statement }
end process [ label ] ;
```

- The process label, which names the process, is optional.
- The execution is controlled either via
  1. sensitivity list (contains trigger signals), or
  2. wait-statements

Process (cont.)

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

entity  D_latch is
        Port  (         data_in    :   in     std_logic ;
                        enable     :  in     std_logic ;
                        data_out :   out  std_logic);
end D_latch;
```

Sensitivity list

```
architecture Behav of D_latch is
begin
      D_PROCESS: process ( data_in , enable)
      begin
            if ( enable = '1' ) then
                        data_out <= data_in;
            end if ;
      end process D_PROCESS;

end Behav;
```

Sensitivity list

- The sensitivity list is also optional and is enclosed in a '(' ')'‘
- VHDL simulator will invoke the process code whenever the value of at least one of the listed signals changes
- The synthesis tools generally ignore sensitivity lists in contrast to simulation tools

  ➔ *Missing signals in the list can produce different behavior between simulation and synthesis.*

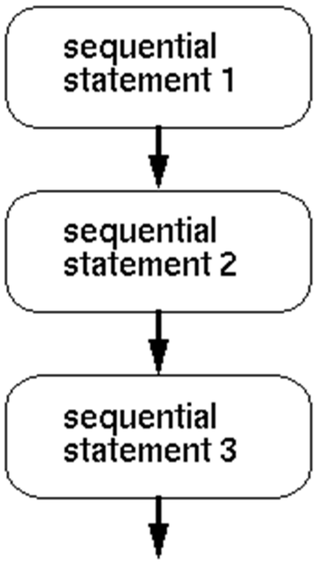
- **Follow these guidelines when developing the sensitivity list:**

➢ Synchronous processes (*processes that compute values only on clock edges*) must be sensitive to the clock signal.

➢ Asynchronous processes (*processes that compute values on clock edges and when asynchronous conditions are true*) must be sensitive to the clock signal (if any) and to inputs that affect asynchronous behavior.

Sequential Statements

- Executed according to the order in which they appear
- Permitted only within processes
- Used to describe algorithms
- All statements in processes or subprograms are processed sequentially, one after another, like in ordinary programming languages



If Statment

```
if CONDITION then
   -- sequential statements
end if;

if CONDITION then
   -- sequential statements
else
   -- sequential statements
end if;

if CONDITION then
   -- sequential statements
elsif CONDITION then
   -- sequential statements
. . .
else
   -- sequential statements
end if;
```

- **Condition is a boolean expression**
- **Optional elsif sequence**
  - **conditions may overlap**
  - **The first if block has the highest priority**
- **Optional else path**
  - **executed, if all conditions evaluate to false**

Ex: *D-Flip Flop* with asynchronous clear

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

entity  DFF is
        Port  (        D, CLK, CLEAR        :  in    std_logic ;
                       Q                     :  out  std_logic
              );
end DFF;
```

```
architecture Behav_DFF  of  DFF  is
begin
      Process (CLK, CLEAR)
      begin
            if ( CLEAR  =  '1') then
                        Q <= '0' ;
            elsif ( CLK' event and CLK = '1' ) then
                        Q <= D;
            end if ;
      end process;
end Behav_DFF;
```

CASE Statement

- All branches are equal in priority
- Choice options **must not overlap**
- The choices can be:
  1. single value
  2. value range
  3. selection of values ("|" means "or")
  4. "when others" covers all remaining choice options
- If an others choice is not present, all possible values of *EXPRESSION* must be covered by the set of choices.

```
case  EXPRESSION  is

   when  VALUE_1    =>
--      sequential statements

   when  VALUE_2 | VALUE_3 =>
--      sequential statements

   when  VALUE_4 to VALUE_N  =>
--      sequential statements

   when   others  =>
--      sequential statements

end   case;
```

Ex:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CASE_STATEMENT is
port (A, B, C, X : in     integer range 0 to 15;
                 Z      : out  integer range 0 to 15;
end CASE_STATEMENT;
```

```
Architecture Behav_Case_Stat of CASE_STATEMENT  is
Begin
        Process (A, B, C, X)  begin
            case  X  is
                when 0 =>
                                Z <= A;
                when   7  |  9 =>
                                Z <= B;
                when  1 to  5   =>
                                Z <= C;
                when others  =>
                                 Z <= 0;
            end case;
        end process;
end Behav_Case_Stat;
```

Defining Ranges

- Ranges can only be defined for data types with a fixed order

```
when 1 to 5 =>
             Z <= C ;
```

- For array types (e.g. a 'bit_vector') there is no such order

```
when  "0001" to "0101" =>
             Z <= C ;
-- wrong
```

Wait Statement

- A **wait statement** suspends (halts) the process execution.
- Different types of wait statements:

1. wait for a specific time
   ***wait for** SPECIFIC_TIME;*
2. wait for a signal event
   ***wait on** SIGNAL_LIST;*
3. **wait for a true condition (requires an event)**
   ***wait until** CONDITION;*
4. indefinite (process is never reactivated)
   wait ;

- **IEEE VHDL specifies that a process containing a wait statement must not have a sensitivity list.**
- The *Xilinx Foundation Express* has implemented only the third form of the wait statement (wait until).

Ex:

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

entity  DFF is
          Port  (        D, CLK                  :  in     std_logic ;
                         Q, Qbar                 :  out  std_logic
               );
end DFF;
```

```
architecture Arch1  of  DFF  is
Begin
Process
Begin
wait until ( CLK' event and CLK = '1' ) ;
Q <= D;
Qbar <= not D;
end process;
end Arch1;
```

```
architecture Arch2 of  DFF  is
Begin
Process
Begin
wait on  CLK;
If  ( CLK = '1' )  then
      Q <= D;
      Qbar <= not D;
end if ;
end process;
end Arch1;
```

for Loop Statements

```
[label :] for identifier in range loop

{ sequential_statement }

end loop [label];
```

- The loop parameter (identifier) is:
  - implicitly declared .
  - can not be declared externally .
  - read only access (*You cannot assign a value to a loop identifier.*)
- Loops must have a fixed range in order to be synthesized
- Range must be a computable integer range
- **The only loop supported for synthesis is the *for-loop.***

Ex:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity For_Loop is
   Port ( A : in  integer range 0 to 3;
          Z : out  STD_LOGIC_VECTOR (3 downto 0));
end For_Loop;
```

```
architecture Behavioral of For_Loop is
begin
process (A)
begin
            Z <= "0000";
            for  I  in  0 to 3  loop
                        if (A = I) then
                                    Z(I) <= '1';
                        end if;
            end loop;
end process;
end Behavioral;
```

for-Loop Statements and Arrays

- a loop statement can be used to operate on all elements of an array without explicitly depending on the size of the array.
- ‘range: is an array attribute which gives its range .

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity For_Array_Example is
   Port ( A : out     bit_vector (1 to 10);
          B : in      bit_vector (1 to 10));
end For_Array_Example;
```

```
architecture Behav of For_Array_Example is
begin
process (B)
begin
        for  I  in A'range   loop
                A(I) <= not B(I);
        end loop;
end process;
end Behav;
```

while Loop Statements

```
[label :] while condition loop

{ sequential_statement }

end loop [label];
```

- The condition is any Boolean expression
- The condition of the loop is tested before each iteration, including the first iteration. If it is false, the loop is terminated.
- ***'while' constructs usually cannot be synthesized***

Basic Loop statement

```
[label :] loop

{ sequential_statement }

end loop [label];
```

- Basic loop statement has no iteration scheme
- The enclosed statements are repeatedly executed until an exit or next statement is encountered.

next and *exit* Statment

- The next statement skips execution to the next iteration of an enclosing loop statement, called ***[label]***

```
next [label] [when condition]
```

- The **when** keyword is optional clause that executes its **next** statement when its condition evaluates to ***TRUE***.
- A **next** statement with no ***label*** terminates the current iteration of the innermost enclosing loop.
- The exit statement terminates execution of an enclosing loop statement, called ***[label]***

```
exit [label] [when condition]
```

- The **when** keyword is optional and will execute the **exit** statement when its condition evaluates to ***TRUE***.
- An **exit** statement with no ***label*** terminates the innermost enclosing loop.

null Statements

- The **null** statement states that no action will occur.
- It is often used in **case** statements because all choices must be covered

```
process (CONTROL, A)
begin
        Z <= A;
        case CONTROL is
                when 0 | 7 => -- If 0 or 7, then invert A
                                Z <= not A;
                when others =>
                                null; -- If not 0 or 7, then do nothing
        end case;
end process;
```

Variables

- A variable behaves like you would expect in a software programming language
- Variables are available within processes, only
  - named within process declarations
  - known only in this process
  - they do not have or cause events.
- When a variable is assigned a value, the assignment takes place immediately (***immediate assignment)***
- A variable keeps its assigned value until another assignment takes place (***keeps the last value***)
- Possible assignments
  - signal to variable
  - variable to signal
  - types have to match
- The ' **:=** ' operator is used for variable assignment.

```
count: process (x)
          variable cnt : integer := -1;
Begin
                -
          cnt:=cnt+1;
                -
end process;
```

Ex: Basic Counter

```
entity COUNTER is
     port (      CLK  : in     bit;
                 COUNT      : out  integer range 0 to 9);
end COUNTER;
```

```
architecture BEHAV of COUNTER  is
begin
        Process (CLK)
                variable TEMP :  integer range 0 to 9 := 0;
        Begin
                if ( CLK' event and CLK = '1' ) then
                                if (TEMP = 9) then
                                                TEMP := 0;
                                else
                                                TEMP := TEMP + 1;
                                end if ;
                end if ;
                COUNT <= TEMP;
        end process;
end BEHAV;
```

Ex: Basic Counter 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity COUNTER is
     port (      CLK  : in     STD_LOGIC;
                 COUNT      : STD_LOGIC_VECTOR (3 downto 0));
end COUNTER;
```

```
architecture BEHAV of COUNTER  is
begin
        Process (CLK)
                variable TEMP :  STD_LOGIC_VECTOR (3 downto 0) := "0000";
        Begin
                if ( CLK' event and CLK = '1' ) then
                                if (TEMP = "1001") then
                                                TEMP := "0000";
                                else
                                                TEMP := TEMP + '1';
                                end if ;
                end if ;
                COUNT <= TEMP;
        end process;
end BEHAV;
```

Ex: Basic Counter 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity COUNTER is
    port (     CLK  : in     STD_LOGIC;
               COUNT      : STD_LOGIC_VECTOR (3 downto 0));
end COUNTER;

architecture BEHAV of COUNTER  is
signal TEMP :  STD_LOGIC_VECTOR (3 downto 0) := "0000";
begin
        Process (CLK)
                Begin
                if ( CLK' event and CLK = '1' ) then
                                if (TEMP = "1001") then
                                                TEMP <= "0000"
                                else
                                                 TEMP <= TEMP + '1';
                                end if ;
                end if ;
      end process;
COUNT <= TEMP;
end BEHAV;
```

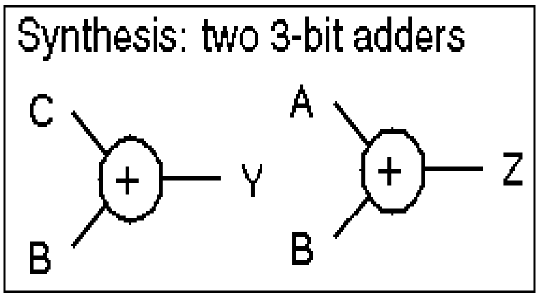
Ex: Clock Divider (f/1000)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Clock_Divider  is
    port (     CLK  : in     STD_LOGIC;
               CLK_OUT      : out STD_LOGIC);
end  Clock_Divider;
```

```
architecture BEHAV of Clock_Divider  is
signal TEMP :  STD_LOGIC := '0' ;
signal count : integer range 0 to 500 := 0;
begin
        Process (CLK)
                Begin
                if ( CLK' event and CLK = '1' ) then
                                if (count = 499) then
                                                TEMP <= not (TEMP);
                                                count <= 0;
                                else
                                                 count <=  count + 1;
                                end if ;
                end if ;
      end process;
CLK_OUT <= TEMP;
end BEHAV;
```

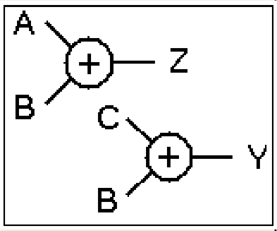
Ex:

```
architecture Behav_XYZ of XYZ is
      signal A, B, C : integer range 0 to 7;
      signal Y, Z :     integer range 0 to 7;
Begin
      -
      -
      process (A,B,C)
            variable M, N : integer range 0 to 7;
      begin
            M := A;
            N := B;
            Z <= M + N;
            M := C;
            Y <= M + N;
       end process;
       -
       -

end Behav_XYZ;
```



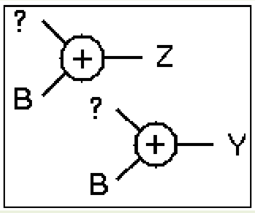
Variables vs. Signals

```
signal A, B, C, Y, Z : integer  range 0 to 7;

begin
   process (A, B, C)
       variable M, N : integer  range 0 to 7;
   begin
       M  :=  A;
       N   :=  B;
       Z <= M + N;
       M  :=  C;
       Y <= M + N;
   end process;
```



The variable assignment is carried out immediately

```
signal A, B, C, Y, Z : integer  range 0 to 7;
signal M, N          : integer  range 0 to 7;
begin
    process (A, B, C, M, N)

   begin
      M  <=  A;
      N  <=  B;
      Z  <= M + N;
      M  <=  C;
      Y  <= M + N;
   end process;
```



- The signal assignment is carried out at the end of the process

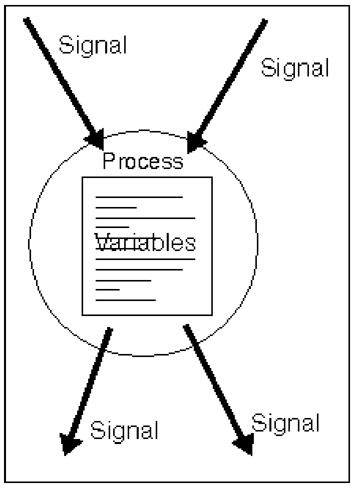
➔ M <= A is overwritten by M <= C ;
- X and Y will be set to the result of B+C.

Use of Variables

- Variables are especially suited for the implementation of algorithms
  1. signal to variable assignment
  2. execution of algorithm
  3. variable to signal assignment

➔ No access to variable values outside their process

➔ Variables store their value until the next process call.



Constants

```
constant name_of_constant: type := initial value ;
```

- ***constant*** declarations create named values of a given type.
- The value of a constant can be read but not changed.
- Constant declarations are allowed in:
  - ***Architectures***
  - ***Packages***
  - ***Entities***
  - **Blocks**
  - ***Processes***
  - ***Subprograms***

```
constant SIZE : integer := 8 ;
```

Subprograms and Packages

- Like other programming languages, VHDL provides **subprogram** facilities in the form of:
  1. **Procedures** (zero or more **in, inout**, or **out** *parameters*)
  2. **Functions** (zero or more **in** *parameters* and *one return value*)

  (Use functions when you do not need to update the parameters, and you want a single return value.)
- **Subprograms** are called by name from anywhere within a VHDL architecture or a package body.
- **Subprograms** can be called ***sequentially*** or ***concurrently.***
- In hardware terms, a subprogram call is similar to module instantiation as it becomes part of the current circuit.

Subprogram Declaration and Body

Package Declaration

```
package ExamplePack is
  procedure P  (A : in integer ;   B : inout integer);
  function INVERT (A : in bit ) return bit;
end ExamplePack;
```

Package Body

```
package body ExamplePack is
  ---
  procedure P  (A : in integer ;   B : inout integer) is
  begin
              B := A + B;
  end P;

  function INVERT (A : in bit ) return bit is
  begin
              return (not A);
  end INVERT;
  ---
 end ExamplePack;
```

Subprogram Calls

- When the subprogram is called, each formal parameter receives a value.
- Each actual parameter's value (of an appropriate type) can come from an ***expression***, a ***variable***, or a ***signal***.
- Actual parameters that use mode **out** and mode **inout** cannot be constants or expressions.
- **Procedure Calls**

```
P (X, Y);
```

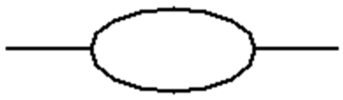
  - In the synthesized circuit, the procedure's actual inputs and outputs are wired to the procedure's internal logic.
- **Function Calls**

```
V2 := INVERT (V1) xor '1';
V3 := INVERT ('0');
```

RTL(register transfer level) -Style

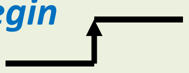
- **Combinational Process**

```
process (        )
begin
- - - - - - - - - - - - - - -
- - - - - - - - - - - - - - -
- - - - - - - - - - - - - - -
end process
```

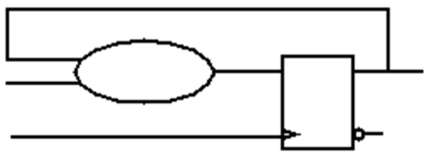


- **Sequential Process (clocked process)**

```
process (        )
begin
```



```
- - - - - - - - - - - - - - -
- - - - - - - - - - - - - - -
end process
```



Combinational Process: Sensitivity List

- The sensitivity list of a combinational process consists of all signals which will be read within the process
- The synthesis tools generally ignore sensitivity lists in contrast to simulation tools
- **During synthesis, VHDL code is simply mapped to logic elements**
- forgotten signal in the sensitivity list will most probably lead to a difference in behavior between the simulated VHDL model and the synthesized design

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

- If the signal SEL was missing, synthesis would create exactly the same result, namely a multiplexer, but simulation will show a completely different behavior.

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

- The output value Z would only change, if the input signals A or B were modified.

Combinational Process: (*Incomplete Assignments*)

```
entity  MUX is
        Port (      A, B, SELC          :  in    std_logic ;
                    Z                   :  out  std_logic );
end MUX;
```

```
architecture Arch1_ WRONG  of  MUX  is
Begin
Process (A, B, SEL)
Begin
    if SEL = '1' then
              Z <= A;
    end if;
end process;
end Arch1_ WRONG ;
```

Transparent latch

```
architecture Arch2 _OK of  MUX  is
Begin
Process (A, B, SEL)
Begin
    if SEL = '1' then
              Z <= A;
     else
               Z <= B;
    end if;
end process;
end Arch2 _OK ;
```

MUX

In the (WRONG) architecture, Synthesis would have to generate an adequate storing element, i.e. a latch instead of the required combinational circuit.

Sequential Process (clocked process)

- Clock Edge Detection:
  - sensitivity list is usually ignored by synthesis tools and wait statements are not synthesizable in general.
  - a solution to the problem of modeling storage elements has to be found.
  - Synthesis tools solved this issue by looking for certain templates in the VHDL code for event detection on the clock signal.

Clock Edge Detection:

if

clock_signal _name'EVENT and
clock_signal _name='1 '

not *clock_signal* _name'STABLE
and
clock_signal _name='1 '

RISING_EDGE (*clock_signal* _
name)

Wait until

clock_signal _name'EVENT and
clock_signal _name='1 '

not *clock_signal* _name'STABLE
and
clock_signal _name='1 '

RISING_EDGE (*clock_signal* _
name)

The first option is still the most common way of describing a rising/falling clock edge for synthesis

Ex: (clocked process)

```
entity  COUNTER is
        Port (              CLEAR : in  STD_LOGIC;
                            CLK : in  STD_LOGIC;
                            COUNT : buffer  integer range 0 to 9);
        end COUNTER ;
```

```
architecture Behavioral1 of COUNTER is
begin
      process (CLK)
      begin
            if  CLK'event and CLK ='1' then
            -- if  rising_edge ( CLK) then
               if (CLEAR = '1' or COUNT >= 9) then
                      COUNT <= 0;
               else
                      COUNT <= COUNT + 1;
               end if;
            end if;
     end process;
end Behavioral1;
```

```
architecture Behavioral2 of COUNTER is
begin
      process
      begin
         wait until  CLK'event and CLK ='1';

          if (CLEAR = '1' or COUNT >= 9) then
                        COUNT <= 0;
          else
                        COUNT <= COUNT + 1;
          end if;

     end process;
end Behavioral2;
```

Synchronous vs. Asynchronous Set/Reset

```
architecture RTL of FF_Sync_R is
begin
            process  (CLK)
            begin
            if (CLK'event and CLK = '1') then
                    if (CLR = '0') then
                           Q <= '0';
                     elsif (PR ='0') then
                           Q <= '1';
                     else
                        Q <= D;
                     end if;
             end if;
            end process;
end RTL;
```

```
architecture RTL of FF_ASync_R is
begin
            process  (CLK, CLR, PR)
            begin
            if (CLR = '0') then
                        Q <= '0';
            elsif (PR '0' then
                        Q <= '1';
            elsif (CLK'event and CLK = '1') then
                        Q <= D;


            end if;
            end process;
end RTL;
```

Enumeration Type

- **User defined** data type
- Defined within Architecture body

Ex:

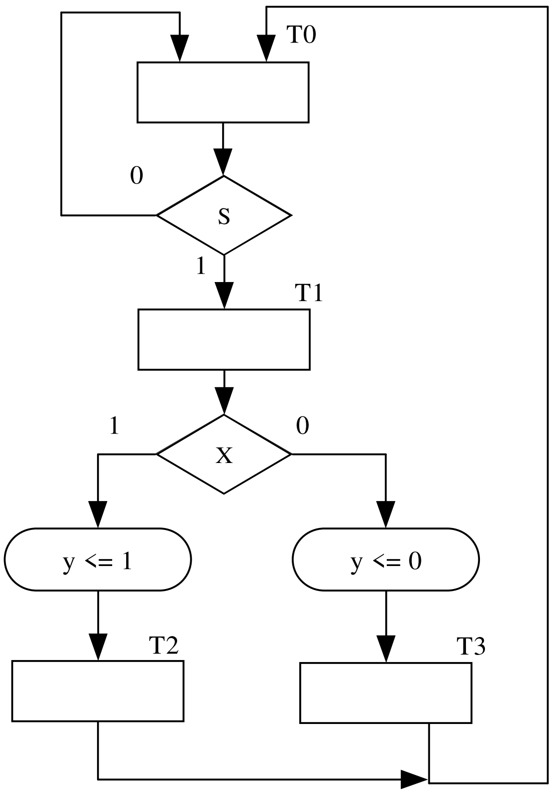
type Day is (sat, sun, mon, tue, wed, thr, fri);

- You can define any variable from this new type;

- **This type is converted internally into array of bits, that are sufficient to encode these values**

- It is normally used to implement **finite state machines**

ASM



```
entity ASM is
        port(S, X, CLOCK            : in BIT;
                 y                  : out BIT);
end ASM;
```

```
architecture BEHAVIOR of ASM is
                type STATE_TYPE is (S0, S1, S2, S3);
                signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
                process(CURRENT_STATE, X, S, CLOCK)
                begin
                                case CURRENT_STATE is
                                                when S0 =>
                                                                if S = '0' then
                                                                                NEXT_STATE <= S0;
                                                                else
                                                                                NEXT_STATE <= S1;
                                                                end if;
                                                when S1 =>
                                                                if X = '0' then
                                                                                NEXT_STATE <= S2;
                                                                else
                                                                                NEXT_STATE <= S3;
                                                                end if;
                                                when S2|S3 =>
                                                                NEXT_STATE <= S0;
                                end case;
                                if( CLOCK 'event and CLOCK = '1') then
                                                if CURRENT_STATE=S1 then
                                                   Y <= X;
                                                end if;
                                                CURRENT_STATE <= NEXT_STATE;
                                end if;
                end process;
end;
```

ASM (cont.)

You can divide the previous process into two processes:

1. The first process is used to determine the next state. (combinational process)
2. The second process is used to detect the clock edge and to transfer the next state into the current state. (Clocked process).

```
architecture BEHAVIOR of ASM is
          type STATE_TYPE is (S0, S1, S2, S3);
          signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
begin
          process( RESET , CLOCK)
          begin
                    if  RESET = '0' then
                              CURRENT_STATE <= S0;
                    elsif ( CLOCK 'event and CLOCK = '1') then
                              if CURRENT_STATE=S1 then
                                        Y <= X;
                              end if;
                              CURRENT_STATE <= NEXT_STATE;
                    end  if;
          end process

          process(CURRENT_STATE , X, S)
          begin
                    case CURRENT_STATE is
                              when S0 =>
                                        NEXT_STATE <=  - - -
                              when S1 =>
                                        - - -
                              when S2|S3 =>
                                        - - -
                    end case;
          end process;
end;
```

Second Process
(Clocked process)

First Process
(combinational process)

Operators

- Logical Operators
- Relational Operators
- Adding Operators
- Unary (Signed) Operators
- Multiplying Operators
- Miscellaneous

A VHDL operator is characterized by the following:

- *Name*
- *Computation (function)*
- *Number of operands*
- *Type of operands (such as Boolean or Character)*
- *Type of result value*

Operators (cont)

Type	Operators	Precedence
Logical	and or nand nor xor	Lowest
Relational	= / = < <= > >=	
Adding	+ - & (concatenation operator)	
Unary (sign)	+ -	
Multiplying	* / mod rem	
Miscellaneous	** abs not	Highest

Logical Operators

- Priority
  - **not** (top priority)
  - **and**, **or**, **nand**, **nor**, **xor**, **xnor** (equal priority)
- Predefined for
  - **bit**, **bit_vector**
  - **boolean**
  - **STD_LOGIC, STD_LOGIC_VECTOR**
- Data types have to match
- Logical operations with arrays require operands of the same type and the same length

```
entity LOGIC_OP is
  port (A, B, C, D : in   bit;
        Z1:          out bit;
        EQUAL :      out boolean);
end LOGIC_OP;

architecture EXAMPLE of LOGIC_OP is
begin

  Z1 <= A  and (B or (not C xor D)));

  EQUAL <= A xor B;             -- wrong

end EXAMPLE;
```

Relational (Comparison)Operators

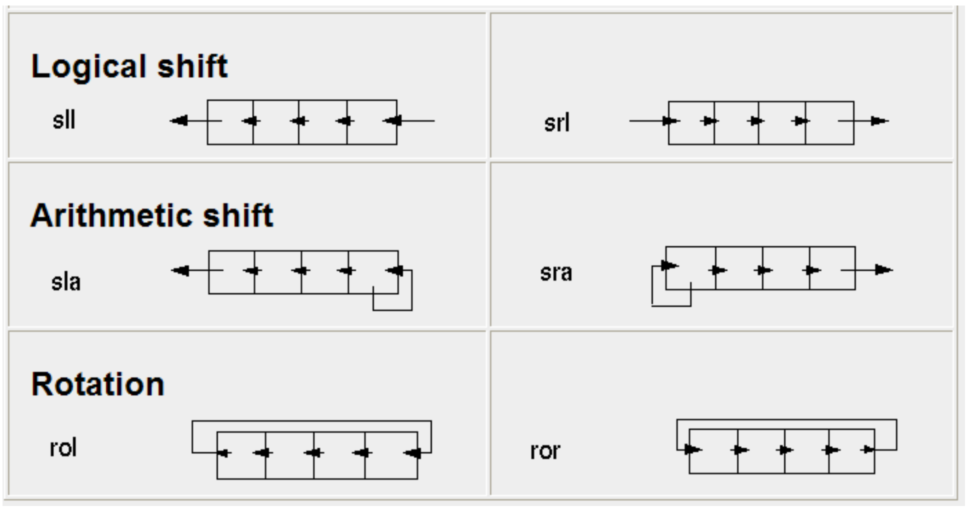
```
architecture EXAMPLE of  RELATIONAL_OP is
  signal A, B:  integer;
  signal A_EQ_B1, A_EQ_B2:  bit;
  signal A_LT_B:     boolean;

begin
  process (A, B)
  begin
    if (A = B) then
      A_EQ_B1 <= '1';
    else
      A_EQ_B1 <= '0';
    end if;
  end process;
  A_EQ_B2 <= A = B;        -- wrong
end EXAMPLE
```

<	less than
<=	less or equal
=	equal
/=	unequal
>=	greater or equal
>	greater

Shift Operators

- Defined for **BIT_VECTOR** , **BOOLEAN_VECTOR** and **STD_LOGIC_VECTOR**

```
signal  A_BUS, B_BUS, Z_BUS : std_logic_vector (3 downto 0);

Z_BUS  <=  A_BUS   SLL   2; --  shift left logical
Z_BUS  <=  B_BUS   SRA  2; --  shift right arithmetic
Z_BUS  <=  A_BUS   ROR 2; --  rotate right logical
```

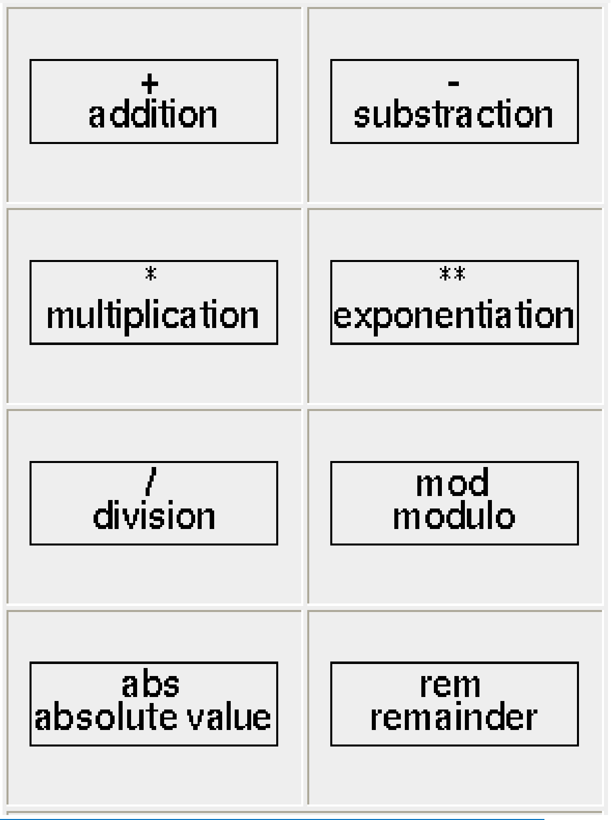


Arithmetic Operators

- **Operands of the same type**
- **Predefined for**
  1. **integer**
  2. **real (except mod and rem)**
  3. **physical types (e.g. Time)**

- **Not defined for bit_vector (undefined number format: unsigned, 2-complement, etc).**
- **Conventional mathematical meaning and priority**

```
signal  A, B, C : integer;
signal  RESULT : integer;

RESULT  <=  - A + B * C;
```



+ addition	- substraction
* multiplication	** exponentiation
/ division	mod modulo
abs absolute value	rem remainder

MOD vs. REM

- **X REM Y** : returns the remainder of X/Y, with the sign of X.

  X REM Y = X - (X/Y)*Y , where both operands are integers

  7 REM 3 = 1

  7 REM -3 = 1

  -7 REM 3 = -1

  -7 REM -3 = -1

- **x MOD y** : returns the remainder of X/Y, with the sign of Y.

  X MOD Y = X REM Y + a*Y , where **a=1** when the sign of X and Y are different or **a=0** otherwise.

  7 MOD 3 = 1

  7 MOD -3 = -2

  -7 MOD 3 = 2

  -7 MOD -3 = -1