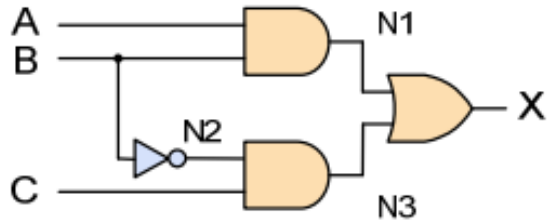
IEE 2463
Programmable Electronic Systems

VHDL Programming Language

Electrical Engineering Department
Pontificia Universidad Católica de Chile
peclab.ing.uc.cl

Introduction
VHDL Programming Language

This section provides a brief overview of the programming language and compilers.




```
architecture simple of example is begin
   Y <= (A and B) or (not B and C);
end simple;

architecture simple of example is begin
   Y <= (A and B) or (not B and C) after 3ns;
end simple;

architecture gates of example is
  signal N1, N2, N3 : std_logic;
begin
  N1 <= (A and B) after 2ns;
  N2 <= not B after 1ns;
  N3 <= (N2 and C) after 2ns;
  X  <= (N1 or N3) after 3ns;
end gates;
```

Introduction

Hardware Description Lenguaje

- A hardware description language is not a program, but a **code**, which describes the behavior of a digital circuit.
- **VHDL** stands for Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (HDL)
- It is used to build digital system/circuit using Programmable Logic Device like CPLD (Complex Programmable Logic Device) or FPGA (Field Programmable Gate Array)
- VHDL program (code) is used to implement digital circuit inside CPLD / FPGA, or it can be used to fabricate ASIC (Application Specific Integrated Circuit)
- In 1980s US Department of Defense (DoD) initiate the VHSIC program to standardize HDL from different companies. This standardized the HDL.
- In 1985 the first version of VHDL was created by IBM, Texas Instruments and Intermatrix under contract of DoD.
- In 1987 the IEEE 1076 standard was published. Then standard **IEEE 1164**, was added to introduce multi-valued logic system.
- Three common HDLs are Verilog (IEEE 1364), VHDL and SystemC.

Advantages of VHDL

- VHDL is vendor independent, portable (a simple component can be exported) and reusable.
- It is designed to work modularly based on “blocks”. Thereby a complex big system can be traced down to small components.
- All statements in VHDL are executed concurrently (unless otherwise is desired by the programmer)
- It has IEEE and ANSI standard.
- Verilog and VHDL are similar. Both are useful as HDL tool. However:
  - VHDL is strongly typed. This makes it harder to make mistakes as a beginner because the compiler will not allow you to write code that is in valid. Verilog is weakly typed. It allows you to write code that is wrong, but more concise.
  - Verilog looks closer to a software language like C. This makes it easier for someone who knows C well to read and understand what Verilog is doing.
  - VHDL requires a lot of typing. Verilog generally requires less code to do the same thing.
  - VHDL is very deterministic, where as Verilog is non-deterministic under certain circumstances.

Introduction

VHDL vs Verilog…There is any better?->NO

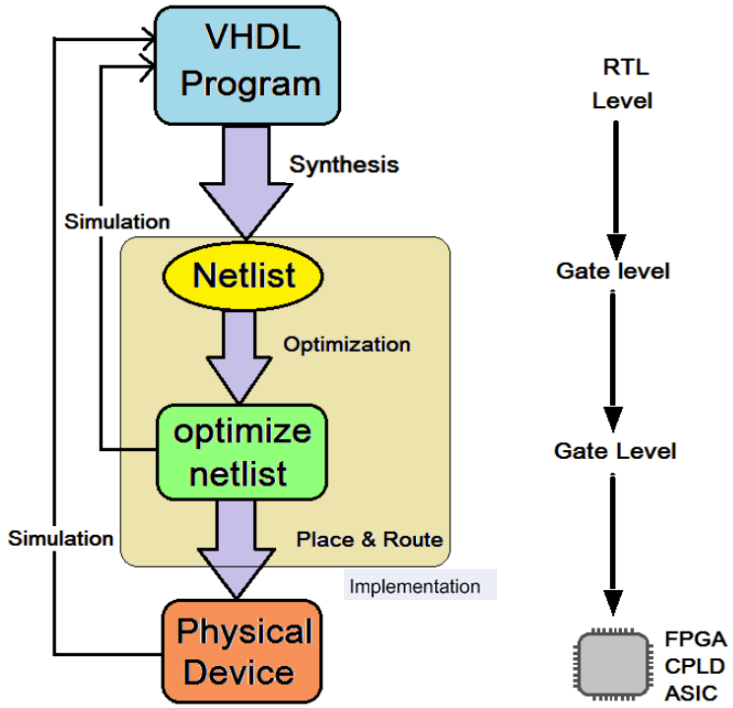
VHDL	Verilog
Strongly typed	Weakly typed
Easier to understand	Less code to write
More natural in use	More of a hardware modeling language
Wordy	Succinct
Non-C-like syntax	Similarities to the C language
Variables must be described by data type	A lower level of programming constructs
Widely used for FPGAs and military	A better grasp on hardware modeling
More difficult to learn	Simpler to learn

Finally, the decision of which is better depends on what suits better for you.

More about this:
Here, Here and Here

Introduction

Design Flow



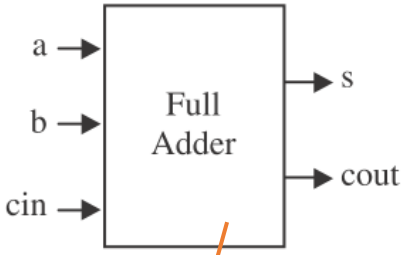
1.- First we write our VHDL code on a .vhd file. This code describe the circuit at Register Transfer Level (RTL). (abstraction level for creating circuits).
2.- Then synthesis is executed. Here VHDL code is transformed into a netlist at gate level.
3.- A netlist optimization process is executed to reduce delay signals and/or space required by the circuit. After that simulation can be done:

- Simulation can be including time delays or not (to test the logic).
- If simulation is not as expected, VHDL code must be modified.

4.- Finally the designed circuit is either: i) “printed” in a programmable device such as FPGA/CPLD or ii) transformed into a MASK to create an ASIC (Application Specific Integrated Circuit) by *placing and routing the software* (fitter). At this stage, the final device can be simulated and verified again.

Electronic Design Automation (**EDA**): Are the sofwares provided by companies to do simulation and synthesis. Like Quartus from Altera and Vitis/Vivado from Xilinx.

Introduction
How it works



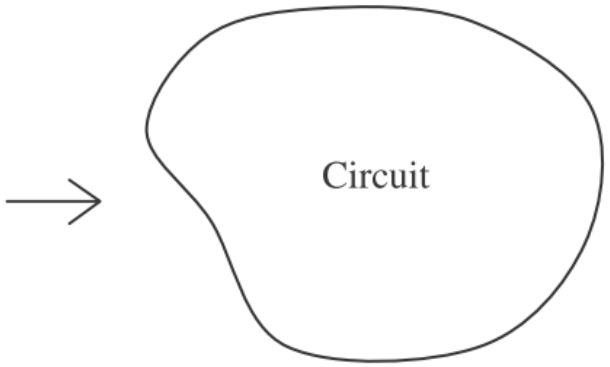
a b cin	s cout
0 0 0	0 0
0 1 0	1 0
1 0 0	1 0
1 1 0	0 1
0 0 1	1 0
0 1 1	0 1
1 0 1	0 1
1 1 1	1 1

s and count are computed as: $\text{cout} = \text{a.b} + \text{a.cin} + \text{b.cin}.$

$\text{s} = \text{a} \oplus \text{b} \oplus \text{cin}$

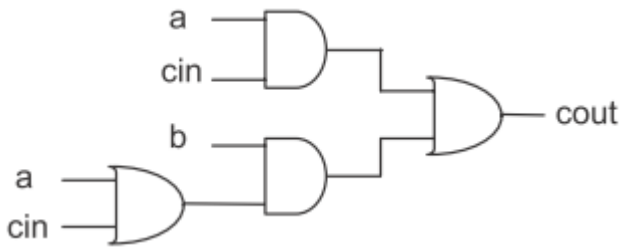
We generate an "**ENTITY**"(circuit structure) which has 3 input **ports** (a,b,cin) and 2 output ports (s, count). Besides that, its **architecture** (circuit functioning) describes how to assign values to s and count.

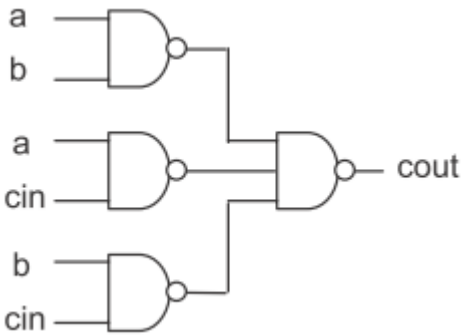
```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
-------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```



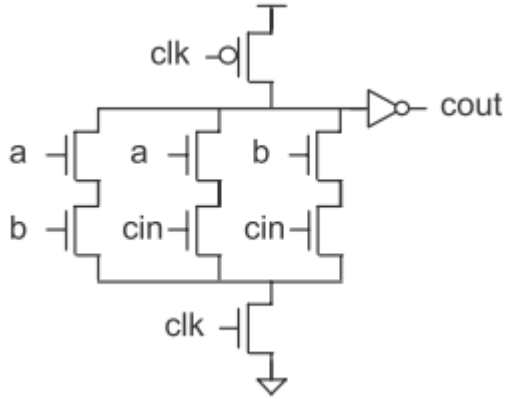
Several possible implementations

Equtions of ARCHITECTURE described before can be implemented in several ways. The results depends on the **compiler/optimizer** and more important, the **target technology.**



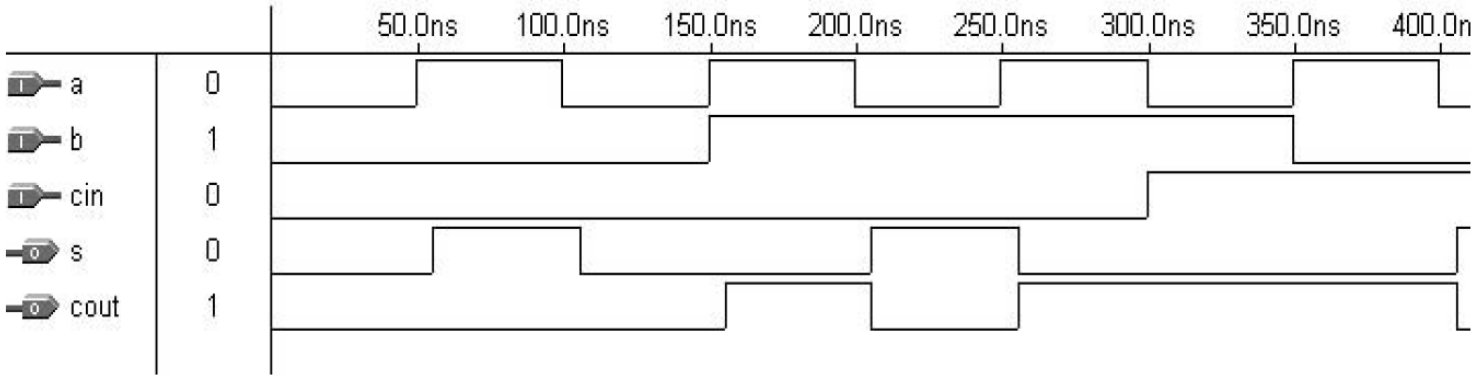


These are two possible implementations if the target technology is a FPGA or CPLD.



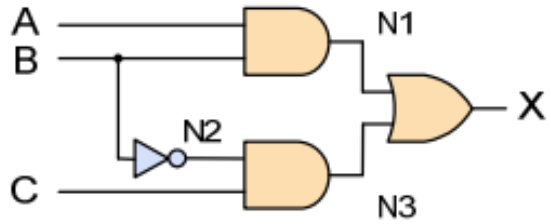
This shows a possible implementation in CMOS for an ASIC at transistor level.

Simulation results from VHDL design: Where is the error? Check it!



END-Introduction
VHDL Programming Language

This section provides a brief overview of the programming language and compilers.




```
architecture simple of example is begin
   Y <= (A and B) or (not B and C);
end simple;

architecture simple of example is begin
   Y <= (A and B) or (not B and C) after 3ns;
end simple;

architecture gates of example is
  signal N1, N2, N3 : std_logic;
begin
  N1 <= (A and B) after 2ns;
  N2 <= not B after 1ns;
  N3 <= (N2 and C) after 2ns;
  X  <= (N1 or N3) after 3ns;
end gates;
```

Code Structure
Library, Entity, Architecture

Definition, operators, functions, procedures components, constans, types.

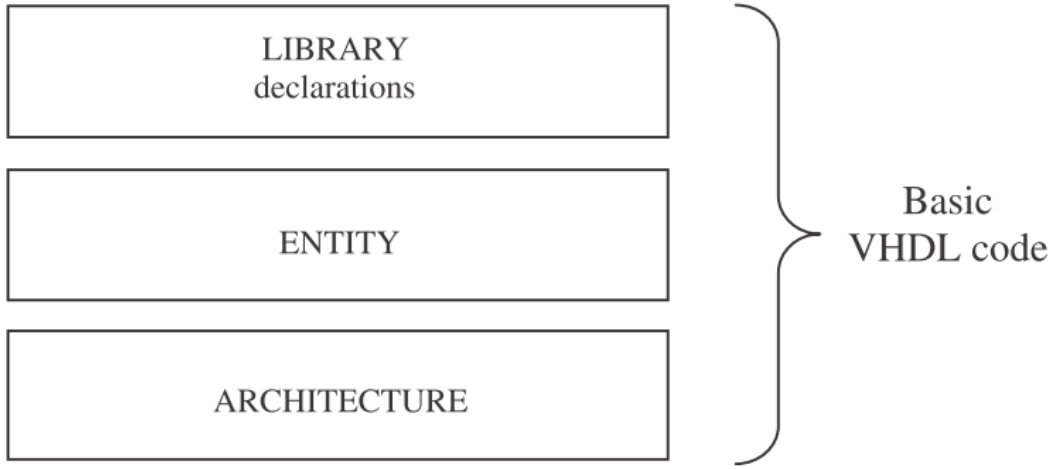
VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Code Structure

Design Flow

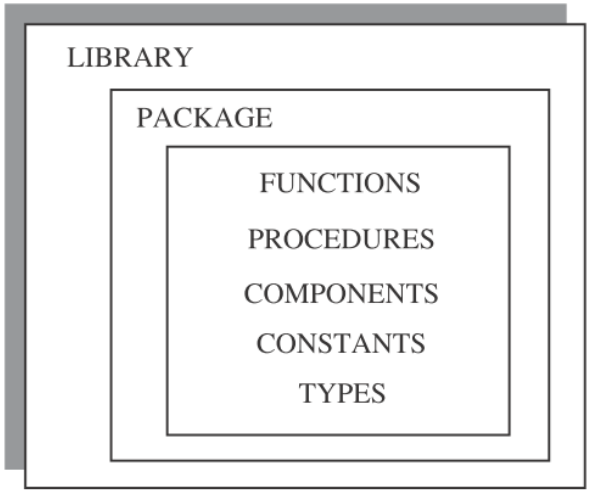
```
-------------------------------------
 LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 -------------------------------------
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
s, cout: OUT BIT);
END full_adder;
-------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
s <= a XOR b XOR cin;
cout <= (a AND b) OR (a AND cin) OR
       (b AND cin);
END dataflow;
```

- A minimum standalone code of VHDL is composed of **Library**, **Entity** and **Architecture**.
  - **LIBRARY** declarations: Contains a list of all libraries to be used in the design.
  - **ENTITY**: Specifies the I/O pins of the circuit.
  - **ARCHITECTURE:** Contains the VHDL code proper, which describes how the circuit should behave (function).



Code Structure

Library



A **Library** is structured based on Functions, procedures or components which are placed inside ***PACKAGES*** and then compiled into the destination library.

LIBRARY declarations: Contains a list of all libraries to be used in the design.

Syntax:

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

Libraries usually required:

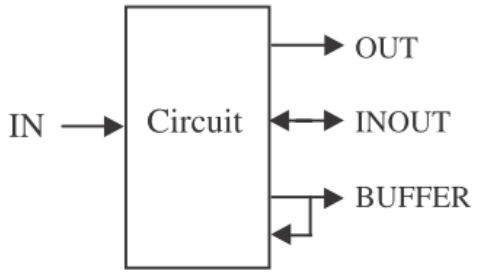
```
LIBRARY ieee;                       -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;        -- the end of a statement or

LIBRARY std;                        -- declaration, while a double
USE std.standard.all;               -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

Note that libraries *std* and *work* are available by default and therefore there is no need to declare them.

Code Structure

Entity

NAND Gate




```
ENTITY nand_gate IS
   PORT (a, b : IN BIT;
         x : OUT BIT);
END nand_gate;
```

ENTITY is a list with specifications of all input and output pins (PORTS) of the circuit.
Syntax:

```
ENTITY entity_name IS
PORT (
port_name : signal_mode signal_type;
port_name : signal_mode signal_type;
...);
END entity_name;
```

Signal Mode can be IN, OUT, INOUT (bidirectional) or BUFFER (for output signals that are used (read) internally).

Signal Type can be BIT, STD_LOGIC, INTEGER, etc. (we see this later).

Entity Name can be any name, except VHDL. It is the name of your entity **and .vhd file**

Code Structure
Architecture

NAND Gate

ARCHITECTURE is the description of the functionality of the circuit.
Syntax (it has two parts, declarations and code)

ARCHITECTURE architecture_name OF entity_name IS
[declarations]
BEGIN
(code)
END architecture_name;

- The declarative part (optional) declares signals and constants (among others).
- The code part define the function of the code.

```
ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;
END myarch;
```

´<=´Assign the result of “a AND b” to x.

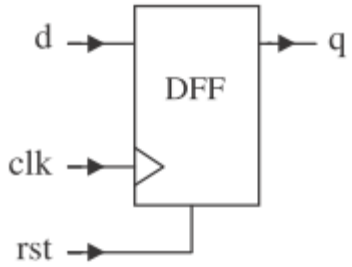
There are three different modelling styles for architecture body:
Data flow : The circuit is described using concurrent statements
Behavioral : The circuit is described using sequential statements
Structural : The circuit is described using different interconnected components

Mixed style is also allowed, using two or the three styles.

Code Structure

Exercises –FlipFlop Source Code

Implement a D-type flip-flop (DFF) and simulate its behavior to validate its code.



- The flip-flop is triggered at rising Edge of the clock signal clk
- It possesses an asynchronous reset
- If reset is high, output must be low regardless of clk.
- If reset is low, the input d is copied to the output q at the clock raising

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Class_VHDL_Exercise_1 is
    Port ( d, clk, rst : in STD_LOGIC;
           q : out STD_LOGIC);
end Class_VHDL_Exercise_1;

architecture Behavioral of Class_VHDL_Exercise_1 is

begin
    PROCESS (rst, clk)
    BEGIN
        IF(rst='1') THEN
            q <= '0';
        ELSIF (clk'EVENT AND clk='1') THEN
            q <= d;
        END IF;
    END PROCESS;

end Behavioral;
```

END-Code Structure
Library, Entity, Architecture

Definition, operators, functions, procedures components, constans, types.

VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Data Types
Vector, bits, etc

This chapter discusses different data types availables in VHDL

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

VHDL Pre-Defined Data Types

- Package standard of library std, defines:

  BIT, **BOOLEAN**, **INTEGER**, and **REAL** data types.

- Package std_logic_1164 of library ieee, defines:

  STD_LOGIC and **STD_ULOGIC** data types.

- Package numeric (formal std_logic_arith by synopsis) of library ieee, defines:

  SIGNED and **UNSIGNED** data types, plus several data conversion functions,
  like conv_integer(p), conv_unsigned(p, b), conv_signed(p, b), and conv_std_logic_vector(p, b).

- Packages std_logic_signed and std_logic_unsigned of library ieee: Contain functions that allow operations with **STD_LOGIC_VECTOR** data to be performed as if the data were of type **SIGNED** or **UNSIGNED**, respectively

VHDL Pre-Defined Data Types

- **BIT** : It can only have the value 0 or 1. When assigning a value of 0 or 1 to a BIT in VHDL code, the 0 or 1 must be enclosed in single quotes: '0' or ‘1’.

Example:

```
SIGNAL x: BIT;          -- x is declared as a one-digit signal of type BIT.
x <= '1’;               -- x is a single-bit signal (as specified above), whose value is
```

- **BIT_VECTOR:** The BIT_VECTOR data type is the vector version of the BIT type consisting of two or more bits. Each bit in a BIT_VECTOR can only have the value 0 or 1. When assigning a value to a BIT_VECTOR, the value must be enclosed in double quotes, e.g. "1011" and the number of bits in the value must match the size of the BIT_VECTOR.

Example:

```
SIGNAL y: BIT_VECTOR (3 DOWNTO 0); -- y is a 4-bit vector, with the leftmost bit being the MSB.
y <= "0111";                       -- y is a 4-bit signal (as specified above), whose value is "0111"
                                   -- (MSB='0'). Notice that double quotes (" ") are used for vectors.
```

VHDL Pre-Defined Data Types

- **STD_LOGIC** : Data type can have 8 logic values. When assigning the value, it must be in enclosed in single quotes.

  - ‘X’ Forcing Unknown (synthesizable unknown)
  - ‘0’ Forcing Low (synthesizable logic ‘1’)
  - ‘1’ Forcing High (synthesizable logic ‘0’)
  - ‘Z’ High impedance (synthesizable tri-state buffer)
  - ‘W’ Weak unknown
  - ‘L’ Weak low
  - ‘H’ Weak high
  - ‘–’ Don’t care

Only four logic values are synthesizable, i.e. usable for programming FPGA or CPLD. The rest can be used for simulation.

- **STD_LOGIC_VECTOR:** Vector version of STD_LOGIC. Each bit can also take same 8 logic values. Value must be enclosed in double quotes

```
SIGNAL x: STD_LOGIC;                          -- x is declared as a one-digit (scalar) signal of type STD_LOGIC.
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";   -- y is declared as a 4-bit vector, with the leftmost bit being MSB.
                                              -- The initial value (optional) of y is "0001“ (use ":=“)
```

VHDL Pre-Defined Data Types Examples

- x0 <= '0’; -- bit, std_logic, or std_ulogic value '0'
- x1 <= "00011111"; -- bit_vector, std_logic_vector,
  -- std_ulogic_vector, signed, or unsigned
- x2 <= "0001_1111"; -- underscore allowed to ease visualization

LEGAL and ILEGAL Assignment

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5); -- legal (same scalar type: BIT)
b(0) <= a; -- legal (same scalar type: BIT)
c <= d(5); -- legal (same scalar type: STD_LOGIC)
d(0) <= c; -- legal (same scalar type: STD_LOGIC)
a <= c; -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d; -- illegal (type mismatch: BIT_VECTOR x STD_LOGIC_VECTOR)
e <= b; -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d; -- illegal (type mismatch: INTEGER x STD_LOGIC_VECTOR)
```

VHDL Pre-Defined Data Types

- **BOOLEAN**: True, False.
- **INTEGER**: 32-bit integers (from -2,147,483,647 to +2,147,483,647).
- **NATURAL**: Non-negative integers (from 0 to +2,147,483,647).
- **REAL**: Real numbers ranging from 1.0E38 to þ1.0E38. Not synthesizable.
- **Physical literals**: Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.
- **Character literals**: Single ASCII character or a string of such characters. Notsynthesizable.
- **SIGNED** and **UNSIGNED**: data types defined in the std_logic_arith package of the ieee library. They have the appearance of STD_LOGIC_VECTOR, but accept arithmetic operations, which are typical of INTEGER data types

VHDL User Define Data Types

- VHDL Allows user to define their own type of variables.

User-defined *integer* types:

```
TYPE my_integer IS RANGE -32 TO 32; -- A user-defined subset of integers.
TYPE student_grade IS RANGE 0 TO 100; -- A user-defined subset of integers or naturals.
```

User-defined *enumerated* types (very useful for state machines):

```
TYPE bit IS ('0', '1');                                   -- This is indeed the pre-defined type BIT
TYPE my_logic IS ('0', '1', 'Z');                         -- A user-defined subset of std_logic.
TYPE state IS (idle, forward, backward, stop);            -- An enumerated data type, typical of finite state machines.
TYPE color IS (red, green, blue, white);                  -- Another enumerated data type.
```

- Generally, encoding is assigned sequentially and automatically, i.e. *00 for red, 01 for green, 10 for blue and 011 for white.*

```
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
                                                          -- This is indeed the pre-defined type BIT_VECTOR.
                                                          -- RANGE <> is used to indicate that the range is unconstrained.
                                                          -- NATURAL RANGE <>, on the other hand, indicates that the only
                                                          -- restriction is that the range must fall within the NATURAL range.
```

Signed and Unsigned Data Types

- These data type are defined in *std_logic_arith* of the package ieee.
- *Syntax is like STD_LOCIG_VECTOR (not like integer):*
  - *SIGNAL x: SIGNED (7 DOWNTO 0);*
  - *SIGNAL y: UNSIGNED (0 TO 3);*
- *Unsigned uses all bits for number representation. Signed type uses "two complement" format.*
- *SIGNED and UNSIGNED are intended for arithmetic operations. (contrary to STD_LOGIC, which do not accept arithmetic operations). However,* ***logic operations are not accepted.*** *There are no restrictions regarding relational (comparison) operations.*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;    -- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b;      -- legal (arithmetic operation OK)
w <= a AND b;    -- illegal (logical operation not OK)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;    -- no extra package required
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;      -- illegal (arithmetic operation not OK)
w <= a AND b;    -- legal (logical operation OK)
```

Data Conversion

- *VHDL does not allow direct operations between data of different type.*
- *To convert data, it can be done manually based on a FUNCTION from a pre-defined PACKAGE.*
- *OR use the **std_logic_1164** from ieee library which provides **straightforward conversion when data are closely related.***

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5;          -- error, type mismatch
y <= long(2*x + 5);    -- OK, result converted into type long
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an
-- 8-bit STD_LOGIC_VECTOR value, then assigned to y.
```

There are several data conversión functions in the library.

END-Data Types
Vector, bits, etc

This chapter discusses different data types availables in VHDL

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

Operators and Attributes
Logical, arithmetic, relational, etc

Operators and attributes are very important to develop a VHDL code, which allows us to work with data.

VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Operators

- This is a very simple, straightforward, but **necessary** chapter. It defines the set of tools that you must work with data. We will start with operators.
- Operators are the key for managing data and making all the logic within or code.

- VHDL provide the following pre-defined operators:
  1. **Assignment** operators
  2. **Logical** Operators
  3. **Arithmetic** Operators
  4. **Relational** Operators
  5. **Shift** Operators
  6. **Concatenation** Operators

Operators and Attributes

Operators

- **Assignment operators** are used to assign values to signals, variables and constants

<= Used to assign a value to a SIGNAL.

:= Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Used also for establishing initial values.

=> Used to assign values to individual vector elements or with OTHERS.

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- Leftmost bit is MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7);          -- Rightmost bit is
                                             -- MSB

x <= '1';        -- '1' is assigned to SIGNAL x using "<="
y := "0000";     -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000";                  -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0');     -- LSB is '1', the others are '0'
```

Operators

- **Logical operators** are used to perform logical operations. Data must be **BIT** or **STD_LOGIC** or **STD_ULOGIC** (and its vector forms), these are:

  - NOT (it has precedence over the others)
  - OR
  - NOR
  - XOR
  - XNOR
  - AND
  - NAND

```
y <= NOT a AND b;       -- (a'.b)
y <= NOT (a AND b);     -- (a.b)'
y <= a NAND b;          -- (a.b)'
```

Operators

- **Arithmetic Operators** are used to perform arithmetic operations.

Data must be **INTEGER, UNSIGNED or REAL** (this is not synthesized directly).

If *std_logic_signed* or *std_logic_unsigned* is used, STD_LOGIC_VECTOR can be also used directly for addition and substraction.

+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	-> Only power of two dividers are allows (shift operation).
**	Exponentiation	-> Only static values of base and exponent are allowed.
MOD	Modulus	-> a MOD b returns the remainder of a/b with the sign of b
REM	Remainder	-> a REM b returns the remainder of a/b with the sign of a
ABS	Absolute value.	-> Return the absolute value

Note that MOD , REM and ABS are usually not synthesized.

Operators

- **Comparison Operators** are used to perform comparisons. Data can be **INTEGER, UNSIGNED or REAL** (this is not synthesized directly), **STD_LOGIC** and its vector versions.
  - = Equal to
  - /= Not Equal to
  - < Less than
  - > Greater than
  - <= Less than or equal to
  - >= Greater than or equal to

- **Shift Operators** uses the syntax: <left operand><shift operation><right operand>.

Left operand must be BIT_VECTOR
Right operand must be INTEGER (+ or – in front of it can be used)

- sll Shift Left Logic -> positions on the right are filled with 0s
- srl Shift Right Logic -> positions on the left are filled with 0s

Examples:
“1100” sll 1 yields “1000”
“1100” srl 2 yields “0011”

Operators and Attributes

Signal Attributes (VHDL Prededined Attributes)

- **Signal Attributes** helps us to check the state of signals. Suppose a signal **x**, then (full list):
  - x'EVENT — Returns true when an event (a change) occurs in x
  - x'STABLE — Returns true when no event occurs in x
  - x'ACTIVE — Returns true if x='1'
  - x'QUIET<time> — Returns true if no event has occurred during "time"
  - x'LAST_EVENT — Return the time since the last event
  - x'LAST_ACTIVE — Return the time since the last event x=´1´
  - x'LAST_VALUE — Return the value of x before the event.
  - Others see full list here

```
IF (clk'EVENT AND clk='1')...          -- EVENT attribute used
                                       -- with IF
IF (NOT clk'STABLE AND clk='1')...     -- STABLE attribute used
                                       -- with IF
WAIT UNTIL (clk'EVENT AND clk='1');    -- EVENT attribute used
                                       -- with WAIT
IF RISING_EDGE(clk)...                 -- call to a function
```

All these conditions are true when a rising clk occurs. Note that the latter instructions call a function. (we see that later)

Operators and Attributes

USER Defined Operations

- Like attributes, operations can be also user defined. To define a new operation, only operators symbols or combination of those can be used. See IEEE1076Std (you must be connected to UC network to be able to download this file) to check all available operator symbols. Creating a new operation, based on available symbols is known as **“operator overloading”.**
- Suppose you want to use the symbol **+** to add an integer to a binary 1-bit number.

```
-------------------------------------
FUNCTION "+" (a: INTEGER, b: BIT) RETURN INTEGER IS
BEGIN
   IF (b='1') THEN RETURN a+1;
   ELSE RETURN a;
   END IF;
END "+";
-------------------------------------
```

Calling the function

```
------------------------------
SIGNAL inp1, outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
   (...)
outp <= 3 + inp1 + inp2;
   (...)
------------------------------
```

Here the first “+” adds two integers, using pre-defined addition operator. The second “+” adds an integer with a BIT using our user-defined operator.

Operators and Attributes

Summary

Operators.

Operator type	Operators	Data types
Assignment	<=, :=, =>	Any
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, −, *, /, ** (mod, rem, abs)◆	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus SIGNED and UNSIGNED

Attributes.

Application	Attributes	Return value
For regular DATA	d'LOW	Lower array index
	d'HIGH	Upper array index
	d'LEFT	Leftmost array index
	d'RIGHT	Rightmost array index
	d'LENGTH	Vector size
	d'RANGE	Vector range
	d'REVERSE_RANGE	Reverse vector range
For enumerated DATA	d'VAL(pos)◆	Value in the position specified
	d'POS(value)◆	Position of the value specified
	d'LEFTOF(value)◆	Value in the position to the left of the value specified
	d'VAL(row, column)◆	Value in the position specified
For a SIGNAL	s'EVENT	True when an event occurs on s
	s'STABLE	True if no event has occurred on s
	s'ACTIVE◆	True if s is high

The black diamond indicates that the construct is not synthesizable (or it has little support).

Remember that a **non-synthesizable statement is one that can not create hardware**. It is useful for simulation, but not after synthesis.

END-Operators and Attributes
Logical, arithmetic, relational, etc

Operators and attributes are very important to develop a VHDL code, which allows us to work with data.

VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Concurrent Code
WHEN, GENERATE, BLOCK

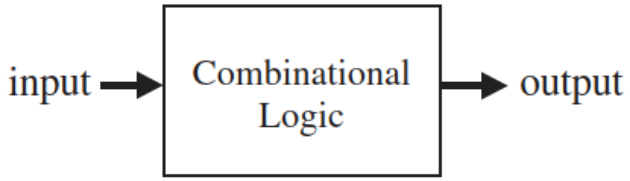
We study the staments required to create a concurrent code.

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

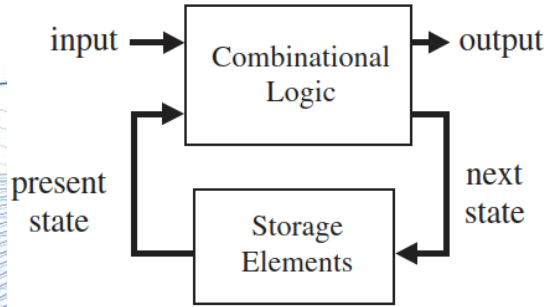
Concurrent Code

Combinational vs Sequential Logic

- We have covered the basics of VHDL. Now we start describing how to create a code properly.
- A VHDL code can be concurrent or sequential
- For concurrent CODE we use the following statements in VHDL: **WHEN** and **GENERATE**
- Also, operators can be used to create concurrent code.
- Also, a special kind of assignment, namely BLOCK can be used.
- In a **Combinational logic** circuit, the output depends only on current inputs. (System does not require stored data)

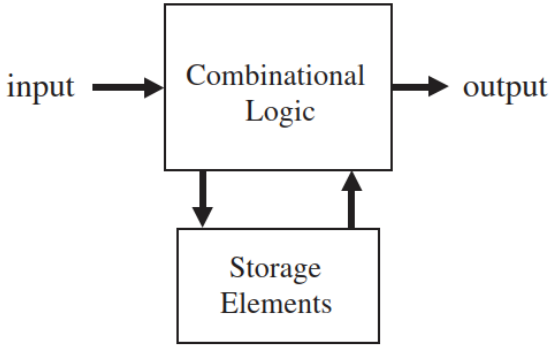


- In a **Sequential logic**, the output depends on previous and/or current inputs. The system require stored data which is feeded-back to the system.



Combinational vs Sequential Logic

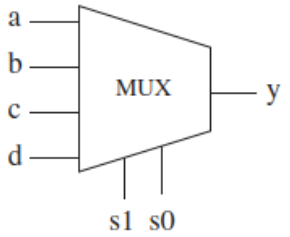
- Not any circuit that contain storage elements (flip-flops) is a sequential logic!. DO not get confused! See this example:



In a RAM, like this figure, the storge elements are in forward path instead of feedback. The memory-read operation depends only on the current address vector applied to the RAM input. No access to previous data is needed!

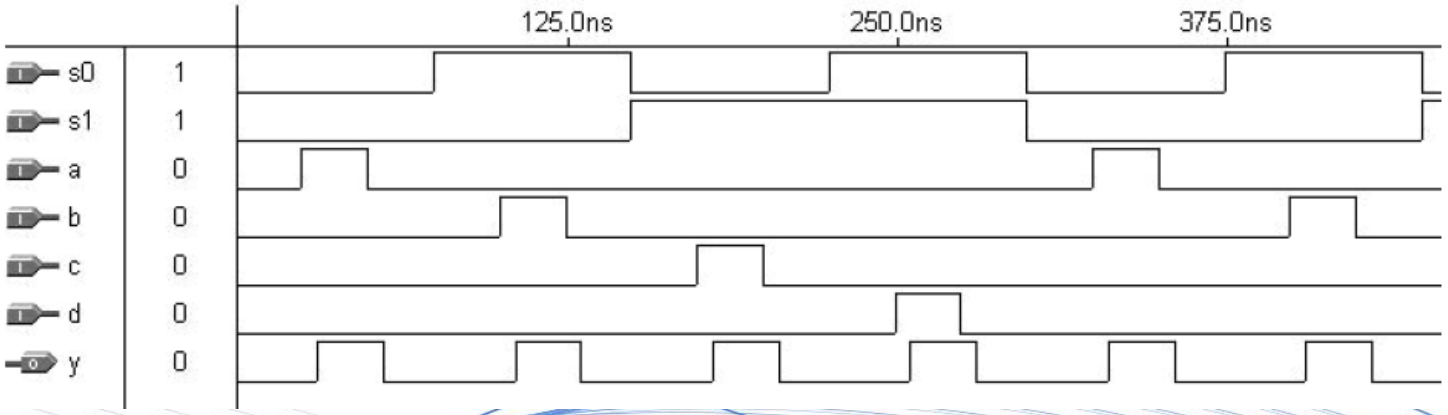
Concurrent vs Sequential Code

- Only statements placed inside PROCESS, FUNCTION or PROCEDURE are sequential. (we see this in next section).
- Concurrent code is called dataflow.
- To build combinational logic circuits, we need concurrent code:
  - Operators;
  - The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
  - The GENERATE statement;
  - The BLOCK statement



4 inputs
1 bit per input
Output is selected by s1, s0

- We have reviewed concurrent code using operators, for instance:




```
-------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
-------------------------------------
ARCHITECTURE pure_logic OF mux IS
BEGIN
   y <=  (a AND NOT s1 AND NOT s0) OR
         (b AND NOT s1 AND s0) OR
         (c AND s1 AND NOT s0) OR
         (d AND s1 AND s0);
END pure_logic;
-------------------------------------
```

Concurrent Code

WHEN (WHEN/ELSE and WITH/SELECT/WHEN)

- **WHEN/ELSE** syntax:
  - assigment WHEN condition ELSE
  - assigment WHEN condition ELSE
  - …;

- **WITH/SELECT/WHEN**
  - WITH identifier SELECT
    assigment WHEN value,
    assigment WHEN value,
    …;

```
------ With WHEN/ELSE ------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN ctl='1' ELSE
        "010";

---- With WITH/SELECT/WHEN --------------------
WITH control SELECT
   output <= "000" WHEN reset,
             "111" WHEN set,
             UNAFFECTED WHEN OTHERS;
-----------------------------------------------
```

Note that using WITH/SELECT/WHEN all combinations must be listed/tested. A nice way to avoid a long list is using keywords like UNAFFECTED and OTHERS. (full list of keywords is in next slide)

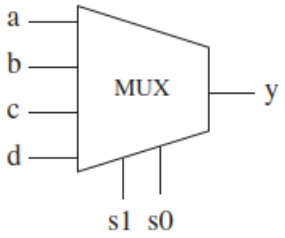
Note that WHEN can also assign:

```
WHEN value                  -- single value
WHEN value1 to value2       -- range, for enumerated data types only
WHEN value1 | value2 |...   -- value1 or value2 or ...
```

Concurrent Code

WHEN (WHEN/ELSE and WITH/SELECT/WHEN)

Implement a multiplexer



4 inputs
1 bit per input
Output is selected by s1, s0

```
------- Solution 1: with WHEN/ELSE --------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
   y <=  a WHEN sel="00" ELSE
         b WHEN sel="01" EL[No Title]
         c WHEN sel="10" ELSE
         d;
END mux1;
-----------------------------------------
```

```
--- Solution 2: with WITH/SELECT/WHEN -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE mux2 OF mux IS
BEGIN
   WITH sel SELECT
      y <=  a WHEN "00",      -- notice "," instead of ";"
            b WHEN "01",
            c WHEN "10",
            d WHEN OTHERS;    -- cannot be "d WHEN "11" "
END mux2;
-----------------------------------------
```

Concurrent Code

GENERATE

- It is another concurrent statement. It allow a section of a code to be repeated a number of times. It create several instances (examples or cases of something) of the same assignment.
- **GENERATE** is used together with **FOR** (regular form) or **IF** (irregular form).
- Syntax **FOR/GENERATE**:

  label: **FOR** identifier **IN** range **GENERATE**

  (concurrent assignments)

  END GENERATE;

- Syntax **IF/GENERATE** (ELSE is not allowed)**:**

  label1: **IF** condition **GENERATE**

  (concurrent assignments)

  END **GENERATE;**

  Nested structures are allowed (i.e., a FOR/GENERATE within IF/GENERATE or other way around).

Concurrent Code

GENERATE-Example

Generate a circuit which receives an input vector of 4 bits and generate an output vector of 8bits. The 4-bits input vector is contained in the last 4-bits of the output vector (other bits are 0). An additional input can shift the position of the vector from 0 to 4 positions to the left. For instance, if input is **1010** and shift is 2, output is 00**1010**00

```
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY shifter IS
   PORT ( inp: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
          sel: IN INTEGER RANGE 0 TO 4;
          outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END shifter;
-----------------------------------------------
ARCHITECTURE shifter OF shifter IS
   SUBTYPE vector IS STD_LOGIC_VECTOR (7 DOWNTO 0);
   TYPE matrix IS ARRAY (4 DOWNTO 0) OF vector;
   SIGNAL row: matrix;
   row(0) <= "0000" & inp;
   G1: FOR i IN 1 TO 4 GENERATE
      row(i) <= row(i-1)(6 DOWNTO 0) & '0';
   END GENERATE;
   outp <= row(sel);
END shifter;
-----------------------------------------------
```

Inputs and outpus are defined. Shifting is selected as integer, input is 4-bit vector ad output is 8-bit vector-

Within our architecture we define a signal row, which is of the matrix. Defined as an array of 5 rows of 8-bit vectors.
Each vector is defined as a subtype std_logic_vector.
This allow further assignments.

We **GENERATE** the concurrent code
(filling the matrix, each row shift the vector one to the left):

Ejemplo: input 1010.
row(0) <= “0000” & inp; (00001010)
row(1) <= row(0)(6 down to 0) & ‘0’; (00010100)
row(2) <= row(1)(6 down to 0) & ‘0’; (00101000)
row(3) <= row(2)(6 down to 0) & ‘0’; (01010000)
row(4) <= row(3)(6 down to 0) & ‘0’; (10100000)

Concurrent Code

Simple BLOCKS

- In its **simple form a BLOCK** statement allows to ***cluster*** a ***set of concurrent statements***. This makes the overall code more readable and manageable. Its syntax is:

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

- Also, a BLOCK can be nested into another BLOCK:

```
label1: BLOCK
            [declarative part of top block]
BEGIN
            [concurrent statements of top block]
label2: BLOCK
            [declarative part nested block]
BEGIN
            (concurrent statements of nested block)
END BLOCK label2;
            [more concurrent statements of top block]
END BLOCK label1;
```

```
b1: BLOCK
   SIGNAL a: STD_LOGIC;
BEGIN
   a <= input_sig   WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

Concurrent Code
Guarded BLOCKS

- A **guarded BLOCK** is a special form of BLOCK which includes a *guard* expression. The statements within the guarded BLOCK are only executed when the *guard* expression is true.

```
label: BLOCK (guard expression)
            [declarative part]
BEGIN
            (concurrent guarded and unguarded statements)
END BLOCK label;
```

```
-------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY dff IS
   PORT ( d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
-------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
   b1: BLOCK (clk'EVENT AND clk='1')
   BEGIN
      q <= GUARDED '0' WHEN rst='1' ELSE d;
   END BLOCK b1;
END dff;
```

Guard expression.
If this **expression** is true, then the **GUARDED statement** is executed.

Guarded statment assign 0 to q only if reset is 1. Otherwise, assigns the input d.

END-Concurrent Code
WHEN, GENERATE, BLOCK

We study the staments required to create a concurrent code.

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

Sequential Code
PROCESS, FUNCTIONS, PROCEDURES

We study the statements required to create a sequential code.

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

Introduction

- **Sequential** code is not limited to sequential logic
- **A Sequential** block (code) is still executed concurrently with the rest of the code (outside the sequential part of the code)
- A sequential code is writing inside a **PROCESS**, **FUNCTION** or **PROCEDURE** (we see functions and procedures in next section) using the sequential statements **IF**, **WAIT**, **CASE** and **LOOP.**
- **VARIABLES** can be used only in sequential code.
- A **VARIABLE** can not be global. Therefore, its value can not be passed out directly.

Sequential Code

PROCESS

- Initializing **PROCESS** indicates that a piece of sequential code will be written.
- Sequential code is characterized by containing IF, WAIT, CASE and/or LOOP statements and by a sensitivity list.
- A **PROCESS** is executed every time a signal in the sensitivity list changes. (or the condition of WAIT is fulfilled)
- Syntax of **PROCESS** is**:**

```
[label:] PROCESS (sensitivity list)
        [VARIABLE name type [range] [:= initial_value;]]
BEGIN
        (sequential code)
END PROCESS [label];
```

- VARIABLES are optional and its initial value is not synthesizable.
- Use of label is also optional. It is useful for code readability.

PROCESS is executed when clk or rst changes

```
-------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------
ENTITY dff IS
   PORT (d, clk, rst: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END dff;
-------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (clk, rst)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
   END IF;
   END PROCESS;
END behavior;
-------------------------------------
```

PROCESS-SIGNALS and VARIABLES

It is important to note the differences between SIGNAL and VARIABLE:

- To pass non-static values within and between circuits we can use SIGNALS or VARIABLES
- A **SIGNAL** can be declared in **PACKAGE**, **ENTITY** and **ARCHITECTURE.** It can be global.
- A **VARIABLE** can be declared only in a **PROCESS**. It is only local. (it can not go out of the PROCESS directly).
- To send out of process a VARIABLE, we need to assign its value to a SIGNAL.
- VARIABLES are immediately updated. Its new value can be used in next line of code. A signal within a PROCESS can only guarantee its updated value the next time the code enters to the PROCESS.
- Remember that assignment for SIGNAL is “<=“ . Assignment for VARIABLE is “:=“.

Sequential Code

PROCESS-IF

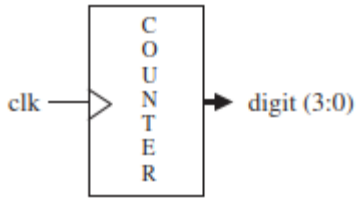
- Syntax of **IF/ELSE** is:

```
IF conditions THEN assignments;
        ELSIF conditions THEN assignments;
        ...
        ELSE assignments;
END IF;
```

- Example:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

Make a 1-digit counter (0 to 9) with 4bit output and a clk signal as input. Counter resets automatically (from 9 to 0).




```
-----------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY counter IS
   PORT (clk : IN STD_LOGIC;
         digit : OUT INTEGER RANGE 0 TO 9);
END counter;
-----------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count: PROCESS(clk)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         temp := temp + 1;
         IF (temp=10) THEN temp := 0;
         END IF;
      END IF;
      digit <= temp;
   END PROCESS count;
END counter;
-----------------------------------------
```

Note that we are comparing against a constant. This is made by a simple and cheap comparator (very good for saving FPGA resources). If a programable parameter is used instead of a constant, a full comparator is required, which uses much more resources of FPGA. Only use that when needed!

Sequential Code

PROCESS-WAIT

- Operator **WAIT** is similar to IF.
- **PROCESS** waits until its condition is fulfilled.
- When **WAIT** is used, no sensitivity list is allowed in **PROCESS**.
- There are three forms of **WAIT**, its syntaxis are:

❑ **WAIT UNTIL** signal_condition; // Only one signal can be used.

❑ **WAIT ON** signal1 [, signal2, ...]; // Accepts multiple SIGNALS. Process hold until any of the signal changes.

❑ **WAIT FOR** time; //Only for simulations (e.g. WAIT FOR 5ns;)

- WAIT UNTIL is more appropriate for Synchronous code (governed by a clock) as only one SIGNAL is accepted. WAIT ON is more appropriated for Asynchronous code, which output depends on changes in the input.
- For both cases, PROCESS is hold until signal condition is met (WAIT UNTIL) or signals change (WAIT ON).

```
PROCESS          -- no sensitivity list
BEGIN
   WAIT UNTIL (clk'EVENT AND clk='1');
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

```
PROCESS
BEGIN
   WAIT ON clk, rst;
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

Sequential Code

PROCESS-CASE

- Operator **CASE** syntax is:

```
CASE identifier IS
        WHEN value => assignments;
        WHEN value => assignments;
        ...
END CASE;
```

- CASE (sequential) might look similar to WHEN (concurrent), also all permutations must be tested. However, CASE allow multiple assignments for each test condition, WHEN allows only one.

```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

2-digit counter in 7 segment format

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY counter IS
   PORT (clk, reset : IN STD_LOGIC;
         digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
END counter;
-------------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   PROCESS(clk, reset)
      VARIABLE temp1: INTEGER RANGE 0 TO 10;
      VARIABLE temp2: INTEGER RANGE 0 TO 10;
   BEGIN
   ---- counter: ----------------------
      IF (reset='1') THEN
         temp1 := 0;
         temp2 := 0;
      ELSIF (clk'EVENT AND clk='1') THEN
         temp1 := temp1 + 1;
         IF (temp1=10) THEN
            temp1 := 0;
            temp2 := temp2 + 1;
            IF (temp2=10) THEN
               temp2 := 0;
            END IF;
         END IF;
      END IF;
      ---- BCD to SSD conversion: --------
      CASE temp1 IS
         WHEN 0 => digit1 <= "1111110";    --7E
         WHEN 1 => digit1 <= "0110000";    --30
         WHEN 2 => digit1 <= "1101101";    --6D
         WHEN 3 => digit1 <= "1111001";    --79
         WHEN 4 => digit1 <= "0110011";    --33
         WHEN 5 => digit1 <= "1011011";    --5B
         WHEN 6 => digit1 <= "1011111";    --5F
         WHEN 7 => digit1 <= "1110000";    --70
         WHEN 8 => digit1 <= "1111111";    --7F
         WHEN 9 => digit1 <= "1111011";    --7B
         WHEN OTHERS => NULL;
      END CASE;
      CASE temp2 IS
         WHEN 0 => digit2 <= "1111110";    --7E
         WHEN 1 => digit2 <= "0110000";    --30
         WHEN 2 => digit2 <= "1101101";    --6D
         WHEN 3 => digit2 <= "1111001";    --79
         WHEN 4 => digit2 <= "0110011";    --33
         WHEN 5 => digit2 <= "1011011";    --5B
         WHEN 6 => digit2 <= "1011111";    --5F
         WHEN 7 => digit2 <= "1110000";    --70
         WHEN 8 => digit2 <= "1111111";    --7F
         WHEN 9 => digit2 <= "1111011";    --7B
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS;
```

NULL is used in CASE for unaffected data.

Sequential Code
PROCESS-LOOP

- Operator **LOOP** is useful when a piece of code needs to be instantiated several times. Loop can be in different forms, as following syntaxis:

FOR/LOOP:

[label:] **FOR** identifier IN range **LOOP**

(sequential statements)

END LOOP [label];

WHILE/LOOP:

[label:] **WHILE** condition **LOOP**

(sequential statements)

END LOOP [label];

EXIT is used to end/scape of the LOOP.

[label:] EXIT [label] [WHEN condition];

NEXT is used for skipping the loop steps

[label:] NEXT [loop_label] [WHEN condition];

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

FOR range must be static

```
WHILE (i < 10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   (other statements)
END LOOP;
```

```
FOR i IN data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count:=count+1;
      WHEN OTHERS => EXIT;
   END CASE;
END LOOP;
```

It exits the loop as son as a value different to 0 is found in data vector

```
FOR i IN 0 TO 15 LOOP
   NEXT WHEN i=skip;     -- jumps to next iteration
      (...)
END LOOP;
```

If I is equal to the value of “skip”, then this loop cycle is jumped and cycle continues with the next i.

Sequential Code

Case and IF

- CASE and IF after optimization can (and usually do) generate same circuits. Examples for same Physical multiplexer:

```
---- With IF: --------------
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
```

```
---- With CASE: ------------
CASE sel IS
   WHEN "00" => x<=a;
   WHEN "01" => x<=b;
   WHEN "10" => x<=c;
   WHEN OTHERS => x<=d;
END CASE;
```

- CASE and WHEN might look similar, but they are intended for different type of codes:

	WHEN	CASE
Statement type	Concurrent	Sequential
Usage	Only outside PROCESSES, FUNCTIONS, or PROCEDURES	Only inside PROCESSES, FUNCTIONS, or PROCEDURES
All permutations must be tested	Yes for WITH/SELECT/WHEN	Yes
Max. # of assignments per test	1	Any
No-action keyword	UNAFFECTED	NULL

Equivalent codes,
But CASE is whitin
PROCESS

```
---- With WHEN: ---------------
WITH sel SELECT
   x <=    a WHEN "000",
           b WHEN "001",
           c WHEN "010",
           UNAFFECTED WHEN OTHERS;
```

```
---- With CASE: ---------------
CASE sel IS
   WHEN "000" => x<=a;
   WHEN "001" => x<=b;
   WHEN "010" => x<=c;
   WHEN OTHERS => NULL;
END CASE;
-------------------------------
```

Final Comments

- **Sequential code** can be used to implement **sequential** or **combinational** circuits.
- If the **sequential code** is intended for **combinational** circuit, the complete **truth-table** should be specified.
- **Registers** are necessary when making **sequential circuits** with **sequential code**.
  - Register: sequential(clocked) memory storing devices.
- Tips for writing proper sequential code for combinational circuits:
  1. Make sure **all input (read) signals used in the PROCESS appear in its sensitivity list**.
     - Otherwise, you get warning and the compiler include the signal to the PROCESS anyway. (not that serious, but good to know)
  2. Make **sure all combinations of input/output signals are defined in the code**.
     - Otherwise, compiler can create a latch for the undefined cases, which saves the last value of the signal. This create more hardware unnecessarily.

Note: Latches appear in combinatorial logic where a variable does not get assigned a value in every possible path. You find latches by checking every if, else, case, when etc. to see if at that point in the code a value has been assigned to every variable. This is not required for clocked circuits. There a variable holds its previous value if no assignment has been made.

END-Sequential Code
PROCESS, FUNCTIONS, PROCEDURES

We study the staments required to create a sequential code.

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

Signals and Variables
When and where to use them

We study the importance and differences between signals and variables

VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Introduction

- VHDL provides two objects to deal with non-static data values: **SIGNAL** and **VARIABLE**.
- VHDL provides **CONSTANT** and **GENERIC** for establishing default static values.
- **CONSTANT** and **SIGNAL** can be global (seen by the whole code) and used in sequential or concurrent code.
- **VARIABLE** is local and only used within the **sequential code (PROCESS, FUNCTION or PROCEDURE)**.

CONSTANT

- CONSTANT is used to stablish default values. Syntax:

  CONSTANT name : type := value;

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

A **CONSTANT** can be declared in a **PACKAGE** (truly global), **ENTITY** (global for the entity and all its architectures) or **ARCHITECTURE** (global for this architecture).

CONSTANT are commonly used in **PACKAGE** or **ARCHITECTURE**, less common in **ENTITY**

SIGNAL

- **SIGNAL** is the object to pass values in and out of the circuit also among internal units.
- **SIGNAL** can be seen as the wires that interconnect circuits (all PORTS of an ENTITY are signals by default).

Syntax:

SIGNAL name : type [range] [:= initial_value];

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

A **SIGNAL** can be declared in a **PACKAGE** (truly global), **ENTITY** (global for the entity and all its architectures) or **ARCHITECTURE** (global for this architecture).

A **SIGNAL** used in **sequential code** (PROCESS, PROCEDURE, FUNCTION) is **not updated immediately** (need conclusion of the sequential code first).

Signals and Variables

SIGNAL

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY count_ones IS
   PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
          ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;
---------------------------------------
ARCHITECTURE not_ok OF count_ones IS
   SIGNAL temp: INTEGER RANGE 0 TO 8;
BEGIN
   PROCESS (din)
   BEGIN
      temp <= 0;
      FOR i IN 0 TO 7 LOOP
         IF (din(i)='1') THEN
            temp <= temp + 1;
         END IF;
      END LOOP;
      ones <= temp;
   END PROCESS;
END not_ok;
---------------------------------------
```

- This circuit counts the number of 1s in the input.
- The code is not properly written as the signal temp is not updated until the PROCESS is finished. Therefore, we should not assign a “new” value to it.

- For this example, it is better to use **VARIABLE**.

- Notice that defining “ones” as **BUFFER** instead of **OUT** is also possible. In this case, “ones” can be used internally and be assigned by values. Nevertheless, use the temporal variable “temp” and “ones” as output is a better practice as “ones” is a real output of the circuit.

Signals and Variables

VARIABLE

- Unlike CONSTANT or SIGNAL, **VARIABLE** is local information. Only used in **PROCESS FUNCTION** or **PROCEDURE**.
- A VARIABLE is updated immediately. That is very good feature to work in sequential code.

  VARIABLE name : type [range] [:= init_value];

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

- Obviously, a VARIABLE can be declared only on the declaration section of PROCESS, FUNCTION and PROCEDURE.

```
-------------------------------------
ARCHITECTURE ok OF count_ones IS
BEGIN
   PROCESS (din)
      VARIABLE temp: INTEGER RANGE 0 TO 8;
   BEGIN
      temp := 0;
      FOR i IN 0 TO 7 LOOP
         IF (din(i)='1') THEN
            temp := temp + 1;
         END IF;
      END LOOP;
      ones <= temp;
   END PROCESS;
END ok;
```

Now the code is well written

Sequential Code

SIGNAL vs VARIABLE Summary

```
-- Solution 1: using a SIGNAL (not ok) --
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE not_ok OF mux IS
   SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
   PROCESS (a, b, c, d, s0, s1)
   BEGIN
      sel <= 0;
      IF (s0='1') THEN sel <= sel + 1;
      END IF;
      IF (s1='1') THEN sel <= sel + 2;
      END IF;
      CASE sel IS
         WHEN 0 => y<=a;
         WHEN 1 => y<=b;
         WHEN 2 => y<=c;
         WHEN 3 => y<=d;
      END CASE;
   END PROCESS;
END not_ok;
---------------------------------------
```

Simulate and compare results!

This is not immediately assigned.

Also, several assignments to signal are present.

Generally, only one assignment is allowed within one PROCESS. So, probably only the last assignment will be taken (sel<=sel+1) or error.

```
-- Solution 2: using a VARIABLE (ok) ----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE ok OF mux IS
BEGIN
   PROCESS (a, b, c, d, s0, s1)
      VARIABLE sel : INTEGER RANGE 0 TO 3;
   BEGIN
      sel := 0;
      IF (s0='1') THEN sel := sel + 1;
      END IF;
      IF (s1='1') THEN sel := sel + 2;
      END IF;
      CASE sel IS
         WHEN 0 => y<=a;
         WHEN 1 => y<=b;
         WHEN 2 => y<=c;
         WHEN 3 => y<=d;
      END CASE;
   END PROCESS;
END ok;
---------------------------------------
```

SIGNAL vs VARIABLE Summary

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

END-Signals and Variables
When and where to use them

We study the importance and differences between signals and variables

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

PACKAGES and COMPONENTS
SYSTEM DESIGN

It is time to integrate several circuits into a bigger system

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

FUNCTIONS and PROCEDURES
SYSTEM DESIGN

It is time to integrate several circuits into a bigger system

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

POWER & ENERGY CONVERSION LABORATORY

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

State Machines
Modeling Sequential Logic Circuits

Finite State Machines - FSM

VHDL
VHSIC HARDWARE DESCRIPTION LANGUAGE

Electrical Engineering Department
Pontificia Universidad Católica de Chile
peclab.ing.uc.cl