DVA494

Programming of Reliable Embedded Systems

Obed Mogaka | Hamid Mousavi | Masoud Daneshtalab

IDD, Malardalens University

January 22, 2026

Today's Agenda

- What is HDL
- VHDL Primer
  - Introduction to VHDL
  - Code Structure and Composition
  - Data Types
  - Operators and Attributes
- Concurrent Statements

Hardware Description Languages

A hardware description language (HDL) is a specialized computer language used to describe the structure, behavior, and timing of electronic circuits, such as digital logic circuits, ASICs, and FPGAs.

- **Popular HDLs:** VHDL, Verilog, SystemVerilog, SystemC
- **Many More:** ADA, COLAMO, Confluence, CoWareC, CUPL, ELLA, ESys.net, Handel-C, Hardcaml, HHDL, HJJ, HML, Hydra, Impulse C, ISPS, ParC, JHDL, KARL, Lava, Lola, M, MyHDL, PALASM, PyMTL, ROCCC, RHDL, SpinalHDL

VHDL

- VHDL (VHSIC-HDL) stands for **Very High-Speed Integrated Circuit Hardware Description Language.**
- It was developed as part of the Very High-Speed Integrated Circuits (VHSIC) initiative funded by the U.S. Department of Defense in the 1980s.
- Was then transferred to the IEEE (Institute of Electrical and Electronics Engineers). Was ratified as IEEE standard 1076 in 1987 - referred to as **VHDL-87**.
- IEEE revised the VHDL standard in 1993, which is referred to as **VHDL-93**, and made minor modifications and bug fixes in 2001, which is referred to as **VHDL-2001.**

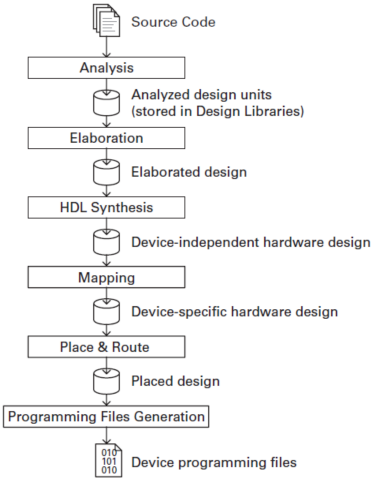
HDL Tools

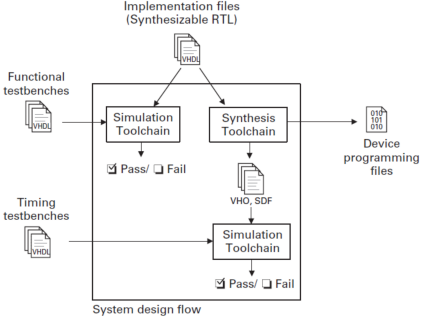
- Some tools are offered by FPGA companies (like Altera-now Intel FPGA-and Xilinx - now AMD), while others are offered by third-party software companies.

- Some of them include:
  - From Xilinx (AMD): Vivado (synthesis/implementation and simulation).
  - From Altera (Intel): Quartus Prime (synthesis/implementation and some simulation).
  - From Mentor Graphics: ModelSim (simulation), Questa (simulation), Precision RTL (synthesis).
  - ModelSim is indeed the tool that implements the simulation part in Quartus Prime and, optionally, also in Vivado.
  - From Synopsys: Synplify Pro (synthesis).
  - From Aldec: Active-HDL (simulation), Riviera-Pro (simulation).

Design Flow

Translation of human-readable source code into machine-readable code.





VHDL Primer

Lexical Elements of VHDL

These are the most basic building blocks of VHDL code

Comments

- Versions of VHDL prior to VHDL-2008 allowed only a single-line comment
- VHDL-2008 introduced delimited comments, similar to those existing in C++

Literals

- a value written directly in the code.
- Several kinds: Numeric, character, string, bit-string

Delimiters

- Used as operators, as punctuation symbols,
- and to define the general shape of VHDL constructs.

Assignment Symbols

- used to assign a value to a signal
- various delimiters are used.

```
1  -- Single line comment
2  /* delimited comment */
3  /*
4    Multi-line comment
5  */
6
7  -- Examples of integer literals:
8  33, 1e6, 2#1111_0000# , 16#dead_beef#
9
10 -- Examples of real literals:
11 33.0, 1.0e6, 2#1111_0000.0000# , 16#dead.beef#
12
13 -- Examples of character literals:
14 'a', 'b', ' ', 'X', '0', '1', '@', '''
15
16 -- Examples of string literals:
17 "abc", "a", "123", "1", "2", "11110000"
18
19 -- Examples of bit-string literals:
20 b"10100101", b"1010_0101", x"f0", 8 d"165"
21
22 -- Single-character delimiters
23 & ' ( ) * + , - . / : ; < = >  | [ ] ?
24
25 -- Compound delimiters
26 => ** := /= >= <= <> ?? ?= ?/= ?< ?<= ?>
27 ?>= << >>
```

Lexical Elements of VHDL

Identifiers

- Identifiers are used to name VHDL items, like signals, variables, entity declarations, architecture bodies, package declarations
- It can contain only letters (a to z, A to Z), decimal digits (0 to 9), and the underline character(_)
- It must start with a letter;
- It must not have two adjacent underline characters or end with an underline character;
- It must be different from any of the **reserved** VHDL words
- VHDL is not case sensitive

```
1
2   -- Examples of legal names
3   clk, clk_5MHZ, data_reday,
4   NUMBER_OF_BITS, CLK_FREQ
5
6   -- Illegal names
7   !rst, ena_, 4input_nand,
8   return, abs, and, architecture
9
10  -- VHDL is case insensitive
11  reset, RESET, /* same meaning*/
```

Choosing Good Names for Your Design

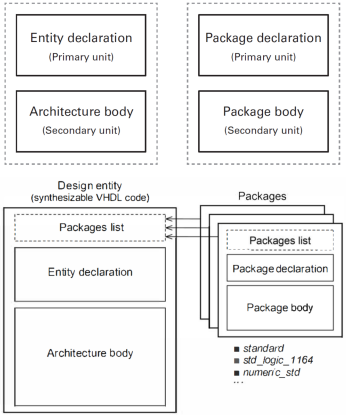
- **Naming an Entity Declaration ("The Design"), Functions and Procedures:** a good choice is a name that makes the purpose of the design very clear.
- **Naming an Architecture Body:** can be the same as the entity's name.
  A common option is to use a name that indicates the abstraction level of the hardware description, like behavioral, dataflow, rtl, structural
- **Naming Constants:** By convention, only capital letters will be used for constant names.
- **Naming Signals and Variables:** Well-established abbreviations from digital circuits analysis or the entire original names (when they are short) are both good name choices
- **Naming Files:** Save synthesizable VHDL code in a file with the same name as the code's entity declaration, with the extension .vhd.
  If a corresponding simulation (testbench) file is also produced, save it with the same name as the synthesizable file plus the _tb suffix.

VHDL Design Units

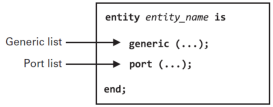
Structurally, a VHDL design is composed of a series of modules organized hierarchically.

There are four main kinds of design units in VHDL:

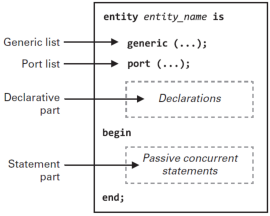
- An **entity declaration** describes a system, subsystem, or component from an external point of view. It specifies a module's interface, including its name, inputs, and outputs.
- An **architecture bod**y describes the internal operation of a module. It specifies the module's behavior or internal structure.
- A **package declaration** groups a collection of declarations that can be reused in other design units at the source code level. The package declaration only defines the interface to a package and specifies no actual behavior.
- A **package body** contains the implementation of a package, providing the actual behavior and operations described in the package declaration.



Entity Declaration

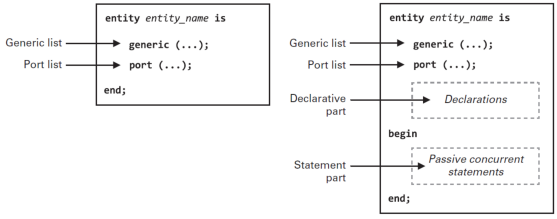


(a) Commonly used regions.



(b) All available regions.

Entity Declaration



(a) Commonly used regions. (b) All available regions.

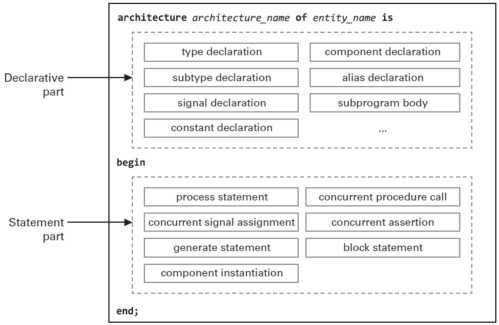
- **Generic list:** used for declaring global constants so parameterized codes can be easily written
  Important for code maintenance, readability, and reusability.

- **Port list:** specifies the type, direction, and name of each port.
  The direction of a port (in , out , inout , or buffer) is called its mode .

```
1  entity fifo is
2      generic (
3          DEPTH: natural := 8 ;
4          WIDTH: natural := 32
5      );
6      port (
7          input_data: in std_logic_vector (WIDTH- 1 downto 0);
8          ...
9      );
10 end ;
```

Architecture Body

An architecture body specifies the operation of a module, describing its internal behavior or structure.



- **Declarative part:** can contain declarations of subprogram, package, type, subtype, constant, signal, shared variable, file, alias, component, attribute, and group.
  The most popular declarations are of types, signals, and constants.

- **Statement part:** where the VHDL statements (the code proper) are placed.

Object Classes

An object is a named item that has a value of a type.

There are four object classes in VHDL: constant, signal, variable, and file

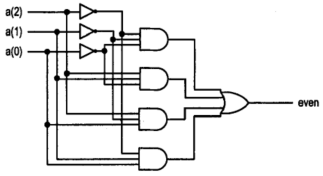
- **Constant**: Members of the constant class have a static value. A constant can be explicitly declared constant or it can also be declared without a value (a deferred constant).
- **Signal**: Members of the signal class are capable of passing values in and out of the circuit as well as between its internal parts. In other words, signals represent **wires**.
- **Variable**: Contrary to signals, they do not represent wires (they can only do that indirectly, via signals).
  They are used solely in sequential code, where they are updated immediately, thus recommended for implementing particularly circuits that involve counters and loops.

```vhdl
-- constants
constant constant_name: constant_type := constant_value;
constant constant_name : constant_type;
constant NUM_BITS: natural := 16;
constant DEPTH: natural := 2**NUM_BITS;
constant MASK: std_logic_vector(7 downto B) := "BBBB1111";

-- signals
signal signal_name: signal_type [:= default_value];

signal clk, rst, data_in: std_logic;
signal address: natural range a to 255;

-- Variables
variable variable_name : variable_type [:= initial_value];

variable count: natural range 0 to 2**NUM_BITS-1;
if rising_edge(clk) then
    count := count + 1;
end if;
```

VHDL Module Example

Even-parity detector based on a sum-of-products expression

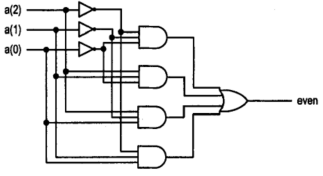
a(2)	a(1)	a(0)	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



VHDL Module Example

Even-parity detector based on a sum-of-products expression

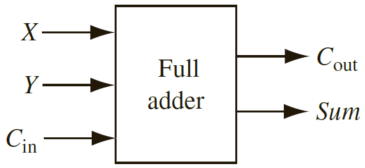
a(2)	a(1)	a(0)	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



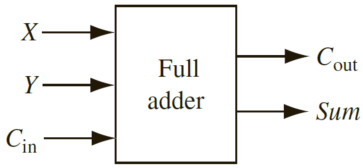

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity even_detector is
    port(
        a: in std_logic_vector(2 downto 0);
        even: out std_logic;
        );
end even_detector;

architecture sop_arch of even_detector is
    signal p1, p2, p3, p4 : std_logic;
begin
    even <= (p1 or p2) or (p3 or p4) after 20 ns;
    p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
    p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
    p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
    p4 <= a(2) and a(1) and (not a(0)) after 12 ns;

end sop_arch;
```

- The signal assignment statements containing after delay create what is called an inertial delay model.
- The result is available after a specific amount of propagation delay, (δ — *delay*)

Full Adder Module

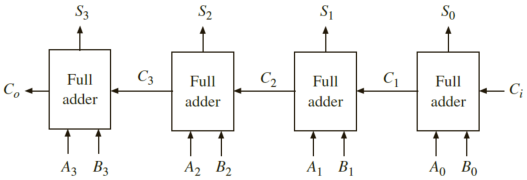


Full Adder Module




```
1  entity FullAdder is
2      port(
3          X, Y, Cin: in bit;      --Inputs
4           Cout, Sum: out bit     --Outputs
5      );
6
7  end FullAdder;
8
9  architecture Equations of FullAdder is
10 begin
11     Sum <= X xor Y xor Cin;
12     Cout <= (X and Y) or (X and Cin) or (Y and Cin);
13
14 end Equations;
```

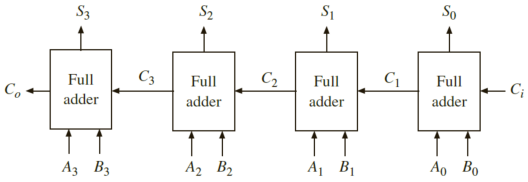
Four-Bit Full Adder



- We can use the FullAdder module as a component in a system, which consists of four full adders connected to form a 4-bit binary adder
- In CAD jargon, we instantiate four copies of the FullAdder

VHDL Module Examples

Four-Bit Full Adder



- We can use the FullAdder module as a component in a system, which consists of four full adders connected to form a 4-bit binary adder
- In CAD jargon, we instantiate four copies of the FullAdder

```
1  entity Adder4 is
2      port(A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
3      S: out bit_vector(3 downto 0); Co: out bit);       -- Outputs
4  end Adder4;
5
6  architecture Structure of Adder4 is
7      -- component declaration
8      component FullAdder
9          port (X, Y, Cin: in bit;    -- Inputs
10                Cout, Sum: out bit); -- Outputs
11     end component;
12
13     signal C: bit_vector(3 downto 1); -- C is an internal signal
14
15 begin --instantiate four copies of the FullAdder
16     -- label: component-name port map (list-of-actual-signals);
17     FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
18     FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
19     FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
20     FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
21 end Structure;
```

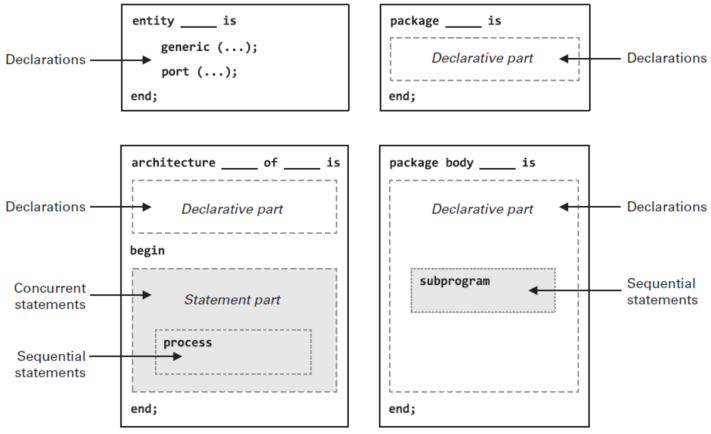
Concurrent versus Sequential Statements

- **Concurrency** implies that the individual statements can be written in any order.
- A common mistake is to assign a value to a signal somewhere in the code and then assign another value to it later, believing that the compiler will simply consider the last value as the valid one.
- The only exceptions in VHDL are processes and subprograms, inside which the code is interpreted **sequentially**.

```
1  -- all three pieces of code would lead to the same result
2
3  total <= a1 + a2*a3;
4  flag <= '1' when total > LIMIT else ' 0 ' ;
5
6  flag <= '1' when total > LIMIT else ' 0 ' ;
7  total <= a1 + a2*a3;
8
9  total <= a1 + a2*a3;
10 flag <= '1' when a1 + a2*a3 > LIMIT else '0';
11
12 -- Not synthesizable
13 -- two assignments to sum
14 sum <= B;
15 ...
16 sum <= a + b;
```

```
1  -- sequential example
2  process (all)
3  begin
4      sum <= 0;
5
6      ...
7
8      sum <= a + b;
9  end process;
```

Concurrent versus Sequential Statements



Levels of Abstraction

There are different levels of abstraction at which we can describe of a digital design:

- **General description:** from boolean algebra
  e.g. sum-of-products expression using concurrent statements.
- **Structural description**: using the concept of component instantiation.
  The component is **declared** and then can be **instantiated** (actually used) as needed.
- **Register Transfer Level (RTL) description**: Describes the clocked behavior of the design.
  Expressly describes data transfers between storage elements in sequential logic.
- **Abstract behavioral description**: encapsulated in in a special construct known as a process.
  Provide language constructs that resemble the sequential semantics, including the use of variables and sequential execution.

Data Types and Operators

- VHDL is said to be a **strongly typed language** - an object can only be assigned a value , and only the operations defined with the data type can be performed on the object.
- If a value of a different type has to be assigned to an object, the value must be converted to the proper data type by a type conversion function or type casting.

- The motivation behind a strongly typed language is to catch errors in the early stage.

- The subject of data types in VHDL, including both **predefined** and **user-defined** types, is relatively complex, but **it is indispensable to understand it well to write code efficiently.**

Predefined VHDL Types

- The types defined in the package `standard` are considered the true predefined types of VHDL because they are built into the language.
- The list of predefined types includes `integer`, `boolean`, `bit`, `bit_vector`, among others.

- Examples of standard types include `std_logic`, `signed`, `unsigned`, and `float`. They are declared in the following packages:
  - **Standard mathematical packages:** math_real and math_complex;
  - **Standard multivalue logic package:** std_logic_1164 and std_logic_textio ;
  - **Standard synthesis packages:** numeric_bit, numeric_std, numeric_bit_unsigned, and numeric_std_unsigned;
  - **Fixed-point and floating-point packages:** fixed_float_types, fixed_generic_pkg, fixed_pkg, float_generic_pkg, and float_pkg.

Predefined VHDL Types

Types	Values	Defined in
`boolean`	`false`, `true`	package *standard* (library *std*)
`bit`	`'0'`, `'1'`	package *standard* (library *std*)
`std_ulogic`	`'U'`, `'X'`, `'0'`, `'1'`, `'Z'`, `'W'`, `'L'`, `'H'`, `'-'`	package *std_logic_1164* (library *ieee*)
`std_ulogic_vector`	Array of `std_ulogic`	package *std_logic_1164* (library *ieee*)
`std_logic`	Same as `std_ulogic`	package *std_logic_1164* (library *ieee*)
`std_logic_vector`	Same as `std_ulogic_vector`	package *std_logic_1164* (library *ieee*)
`integer`	At least -2,147,483,647 to +2,147,483,647	package *standard* (library *std*)
`unsigned, signed`	Array of `std_ulogic`, or array of `bit`	packages *numeric_std* or *numeric_bit* (library *ieee*)
`ufixed, sfixed`	Array of `std_ulogic`, with decimal point between the indices "0" and "-1"	packages *fixed_pkg* and *fixed_generic_pkg* (library *ieee*)
`float, float32, float64, float128`	Array of `std_ulogic`, with meaning given by IEEE Std 754 and IEEE Std 854	packages *float_pkg* and *float_generic_pkg* (library *ieee*)
`character`	`NUL`, ..., `'0'`, `'1'`, `'2'`, ..., `'A'`, `'B'`, `'C'`, ..., `'a'`, `'b'`, `'c'`, ..., `'ÿ'`	package *standard* (library *std*)
`string`	Array of `character`	package *standard* (library *std*)
`real` (non-synthesizable)	At least -1.80×10^{308} to $+1.80\times10^{308}$ with approximately 15 decimal digits	package *standard* (library *std*)
`time` (non-synthesizable)	At least -2,147,483,647 fs to +2,147,483,647 fs	package *standard* (library *std*)

Types Classes

There are five official type classes in VHDL, called **scalar**, **composite**, **file**, **access**, and **protected** types.

Type class	Subclass	Main predefined types (and subtypes) in each class
Scalar types	Integer types	`integer (natural, positive)`
	Enumeration types	`bit`
		`boolean`
		`character`
		`std_ulogic (std_logic)`
	Float. point types	`real`
	Physical types	`time`
Composite types	Array types	`bit_vector`
		`boolean_vector`
		`integer_vector`
		`real_vector`
		`time_vector`
		`string`
		`std_ulogic_vector (std_logic_vector)`
		`unsigned`
		`signed`
		`ufixed`
		`sfixed`
		`float`
	Record types	Collection of named elements of same or different types
Access types	Used for creating pointers in simulation models that change dynamically or with structures that are not known in advance. (Pointers should be avoided as much as possible.)	
File types	For dealing with files (opening, reading, etc.) during simulation.	
Protected types	To prevent multiple processes from simultaneously accessing the same shared variable during simulation. (Shared variables should be avoided as much as possible.)	

Aggregation, Concatenation, and Resizing

Data Aggregation

- To build an aggregate, we write the elements to be combined within parentheses, separated by commas.
- The keyword `others`, when used, must come last

Data Concatenation

- Putting separate data pieces together to build or enlarge arrays.
- Made using the concatenation operator (&)
- keyword others is not allowed.

Resizing Data Arrays

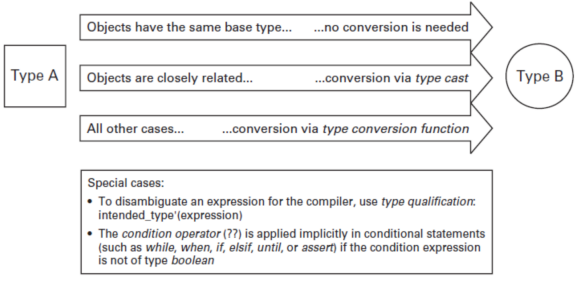
- The `resize` function can be used to extend or reduce the size of arrays.

Function	For types/subtypes	Package of origin
resize	unsigned, signed (1)	*numeric_bit*
	unsigned, signed (2)	*numeric_std*
	bit_vector	*numeric_bit_unsigned* (3)
	std_ulogic_vector, std_logic_vector	*numeric_std_unsigned* (3)
	ufixed, sfixed	*fixed_generic_pkg*
	float	*float_generic_pkg*

```
1  -- Aggregation
2  signal s1, s2, s3: std_logic_vector(5 downto 0);
3
4  s1 (= '1', '0', '0', '0', '1', '0');  --s1="100010"
5  s2 (= '1', '00', others => 'Z');      --s2="100ZZZ"
6  s3 (= '0', '-', others => s1(5));     --s3="0-1111"
7
8
9  -- Concatenation
10 constant CONST: std_logic_vector(1 to 4) := "1100";
11 signal s1, s2, s3: std_logic_vector(7 to 0);
12 signal s4, s5: signed(5 downto 0);
13
14 s1 <= "11" & "00" & CONST;
15 s2 <= '1' & s1(7) & s1(7 downto 3) & '0';
16 s3 < = CONST & ('1', '0', '1', '0');
17 s4 <= s1(7) & 51(7) & "1100";
18 s5 <= (s4(5 downto 2) & "00");
19
20
21 -- Resize
22 signal uns1: unsigned(7 downto 0); --8 bits
23 signal uns2: unsigned(5 downto 0); -- 6 bits
24
25 uns1 <= resize(uns2, 8); --result: uns1 = "00" & uns2
26 uns2 <= resize(uns1, 6); --result: uns2 = uns1(5 downto 0)
27 ...
28 signal sig1: signed(7 downto 0); --8 bits
29 signal sig2: signed(S downto 0); --6 bits
30
31 sig1 <= resize(sig2, 8); --result: sig1 = sig2(5) & sig2(5)
          & sig2
32 sig2 <= resize(sig1, 6); --result: sig2 = sig1(7) & sig1(4
        downto 0)
```

Converting between Types

- Often the same value needs to be used in more than one form.
- Strictly speaking, there are two ways to convert objects to a different type in VHDL:
  - **Type casting**:
  - **conversion function**:
- **No Conversion Is Necessary When Object Subtypes Have a Common Base Type**



Type Casting

- Some types are similar enough that conversion between them is trivial.
- Closely related types in VHDL:
  - Any abstract numeric types (i.e., integer and abstract floating-point types) are closely related.
  - Array types with the same number of dimensions whose element types are also closely related.

- To perform type casting, we write a **type mark** (the name of a type) and an expression of a closely related type in parentheses:

```
1  -- synthax
2  intended_type (value_of_original_type)
3
4  -- Examples
5  variable int_vector: integer_vector ( 1 to 3 );
6  variable real_vect: real_vector ( 1 to 3 );
7  ...
8  real_vect := ( 1.0 , 2.0 , 3.0 );
9  int_vector := integer_vector ( real_vect );
10 real_vect := real_vector ( int_vect );
11
12 signal sly: std_logic_vector(7 downto 0);
13 signal uns: unsigned(7 downto 0);
14 signal sig: signed(15 downto 0);
15
16 sly < = std_logic_vector(uns);      --type cast from UNS to SLY
17 uns <= unsigned(slv);               --type cast from SLY to UNS
18 sig(15 downto 0) <= signed(slv);    --type cast from SLY to SIG
19 sig <= signed(uns) & signed(slv);   --type cast from UNS to SIG and SLY to SIG
```

A note about terminology: the term **type cast** is borrowed from other programming languages. In VHDL, the official name for this operation is called **type conversion**

Type-Conversion Functions

- When two types are not closely related, we need a conversion function to convert between them.
- Type conversion functions may be predefined by the language, provided with a type or package, or written by the designer.

Predefined type-conversion functions for synthesizable types and subtypes

From type	To type		Type-conversion function	Package of origin
integer, natural, positive	std_ulogic_vector	1	*to_std_ulogic_vector(arg, width)*	*numeric_std_unsigned*
	std_logic_vector	2	*to_std_logic_vector(arg, width)*	*numeric_std_unsigned*
	unsigned	3	*to_unsigned(arg, width)*	*numeric_std*
	signed	4	*to_signed(arg, width)*	*numeric_std*
	ufixed	5	*to_ufixed(arg, L, R)*	*fixed_generic_pkg*
	sfixed	6	*to_sfixed(arg, L, R)*	*fixed_generic_pkg*
	float	7	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*
	real	8	Type cast: *real(arg)*	*math_real*
bit	std_ulogic, std_logic	9	*to_stdulogic(arg)*	*std_logic_1164*
bit_vector	std_ulogic_vector	10	*to_stdulogicvector(arg)*	*std_logic_1164*
	std_logic_vector	11	*to_stdlogicvector(arg)*	*std_logic_1164*
std_ulogic, std_logic	bit	12	*to_bit(arg)*	*std_logic_1164*

Type-Conversion Functions

Predefined type-conversion functions for synthesizable types and subtypes

From type	To type		Type-conversion function	Package of origin
std_ulogic_vector, std_logic_vector	integer or natural	13	*to_integer(arg)*	*numeric_std_unsigned*
	bit_vector	14	*to_bitvector(arg)*	*std_logic_1164*
	std_ulogic_vector	15	*to_stdulogicvector(arg)*	*std_logic_1164*
	std_logic_vector	16	*to_stdlogicvector(arg)*	*std_logic_1164*
	unsigned	17	Type cast: *unsigned(arg)*	*numeric_std*
	signed	18	Type cast: *signed(arg)*	*numeric_std*
	ufixed	19	*to_ufixed(arg, L, R)*	*fixed_generic_pkg*
	sfixed	20	*to_sfixed(arg, L , R)*	*fixed_generic_pkg*
	float	21	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*
unsigned, signed	integer or natural	22	*to_integer(arg)*	*numeric_std*
	std_logic_vector	23	Type cast: *std_logic_vector(arg)*	*numeric_std*
	ufixed	24	*to_ufixed(arg_unsigned, L, R)*	*fixed_generic_pkg*
	sfixed	25	*to_sfixed(arg_signed, L, R)*	*fixed_generic_pkg*
	float	26	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*
ufixed, sfixed	integer or natural	27	*to_integer(arg)*	*fixed_generic_pkg*
	std_ulogic_vector	28	*to_std_ulogic_vector(arg)*	*fixed_generic_pkg*
	std_logic_vector	29	*to_std_logic_vector(arg)*	*fixed_generic_pkg*
	unsigned	30	*to_unsigned(arg_ufixed, width)*	*fixed_generic_pkg*
	signed	31	*to_signed(arg_sfixed, width)*	*fixed_generic_pkg*
	sfixed	32	*to_sfixed(arg_ufixed)*	*fixed_generic_pkg*
	float	33	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*
	real	34	*to_real(arg)*	*fixed_generic_pkg*

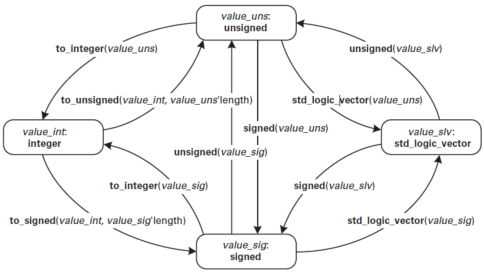
Type-Conversion Functions

Predefined type-conversion functions for synthesizable types and subtypes

From type	To type		Type-conversion function	Package of origin
float	integer	35	*to_integer(arg)*	*float_generic_pkg*
	std_ulogic_vector	36	*to_sulv(arg)*	*float_generic_pkg*
	std_logic_vector	37	*to_std_logic_vector(arg)*	*float_generic_pkg*
	unsigned	38	*to_unsigned(arg, width)*	*float_generic_pkg*
	signed	39	*to_signed(arg, width)*	*float_generic_pkg*
	ufixed	40	*to_ufixed(arg, L, R)*	*float_generic_pkg*
	sfixed	41	*to_sfixed(arg, L, R)*	*float_generic_pkg*
	real	42	*to_real(arg)*	*float_generic_pkg*
real	integer	43	Type cast: *integer(arg)*	*math_real*
	ufixed	44	*to_ufixed(arg, L, R)*	*fixed_generic_pkg*
	sfixed	45	*to_sfixed(arg, L, R)*	*fixed_generic_pkg*
	float	46	*to_float(arg, Ewidth, Fwidth)*	*float_generic_pkg*

Note: *width* is the vector width, ***L*** and ***R*** are the left and right index bounds, and ***Ewidth*** and ***Fwidth*** are the exponent and fraction widths.

Converting between Numeric Representations




```
1
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std_unsigned.all;
4
5  slv <= to_std_logic_vector(int, 8); --from INT to SLV
6  int <= to_integer(slv);             --from SLV to INT
7
8  slv <= std_logic_vector(to_signed(int, 8)); --from INT to SLV
9  int <= to_integer(signed(slv));             --from SLV to INT
```

Operators and Attributes

Operators are predefined symbols or keywords that can be used as functions in expressions but allow a more convenient notation similar to mathematical formulas.

The predefined VHDL operators are divided in six categories:

- Logical operators
- Arithmetic operators
- Comparison (relational) operators
- Shift operators
- Concatenation operator
- Condition operator

Operator Category	Operators
Miscellaneous	`**` `abs` `not`
	Unary version of `and` `or` `nand` `nor` `xor` `xnor` [1]
Multiplying operators	`*` `/` `mod` `rem`
Sign operators	`+` `-`
Adding operators	`+` `-` `&`
Shift operators	`sll` `srl` `sla` `sra` `rol` `ror`
Relational operators	`=` `/=` `<` `<=` `>` `>=`
	`?=` `?/=` `?<` `?<=` `?>` `?>=`
Logical operators	Binary version of `and` `or` `nand` `nor` `xor` `xnor`
Condition operator	`??`

[1] In VHDL-2008, the new unary logic operators (the single-operand versions of `and`, `or`, `nand`, `nor`, `xor`, and `xnor`) have the highest precedence level).

Attributes

Attributes are a mechanism to retrieve special information about named entities.
The predefined attributes are divided in **four categories**: attributes of scalar types, of array types and objects, of signals, and of named entities in general.
These attributes are always called using the tick symbol (')

- **Predefined attributes** are specified in the VHDL LRM and are available for use in any model, with the provision that not all predefined attributes are synthesizable.
- **User-defined attributes** are constants that can be associated with named entities to provide additional information about them

```
1  -- synthax
2  -- named_entity'attribute_identifier
3
4  -- Example: predefined
5  clock'event
6
7  type bitmap is array ( 1 to 16 , 1 to 32 ) of color ;
8
9  bitmap'length(2)
10 -- Result is 32 (length of the second array dimension):
```

```
1  -- user-defined attribute
2
3  type memory_type is array ( 0 to 1023 ) of unsigned ( 31
       downto 0 );
4  signal ram: memory_type ;
5  attribute ram_init_file: string ;
6  attribute ram_init_file of ram: signal is
       "ram_contents.rif" ;
```

Predefined Attributes

Table 1: Predefined attributes of types

Attribute	Applied to	Return Type	Return Value
T'**left**	Scalar type	*T*	Leftmost value of *T*
T'**right**	Scalar type	*T*	Rightmost value of *T*
T'**low**	Scalar type	*T*	Smallest value of *T*
T'**high**	Scalar type	*T*	Largest value of *T*
T'**ascending**	Scalar type	boolean	True if range is ascending
T'**image**(*v*)	Scalar type	string	String representation of *v*
T'**value**(*s*)	Scalar type	T	Value of *T* represented by *s*
T'**pos**(*v*)	Discrete or physical type	integer	Position of v in *T*
T'**val**(*i*)	Discrete or physical type	*T*	Value of *T* at position *i*
T'**succ**(*v*)	Discrete or physical type	*T*	Value of *T* at position *i*+1
T'**pred**(*v*)	Discrete or physical type	*T*	Value of *T* at position *i*-1
T'**leftof**(*v*)	Discrete or physical type	*T*	Value of *T* one position to the left of *v*
T'**rightof**(*v*)	Discrete or physical type	*T*	Value of *T* one position to the right of *v*
T'**base**	Any subtype	Type	The base type of *T*

Table 2: Predefined attributes of array types and array objects

Attribute	Applied to	Return Type	Return Value
A'**left**(*d*)	Array type or object	Type of dimension *d*	Left bound of dimension *d*
A'**right**(*d*)	Array type or object	Type of dimension *d*	Right bound of dimension *d*
A'**low**(*d*)	Array type or object	Type of dimension *d*	Lower bound of dimension *d*
A'**high**(*d*)	Array type or object	Type of dimension *d*	Upper bound of dimension *d*
A'**range**(*d*)	Array type or object	Range of dimension *d*	Range of dimension *d*
A'**reverse_range**(*d*)	Array type or object	Range of dimension *d*	Reverse range of dimension *d*
A'**length**(*d*)	Array type or object	integer	Size (# of elements) of dimension *d*
A'**ascending**(*d*)	Array type or object	boolean	True if range of dimension *d* is ascending
A'**element**	Array type or object	A subtype	Subtype of elements of *A*

Table 3: Predefined attributes of signals

Attribute	Applied to	Return Type	Return Value
S'**delayed**(*t*)	Any signal	Base type of *S*	A signal equivalent to *S* but delayed *t* time units
S'**stable**(*t*)	Any signal	boolean	A signal that is true if *S* had no events in the last *t* time units
S'**quiet**(*t*)	Any signal	boolean	A signal that is true if *S* had no transactions in the last *t* time units
S'**transaction**	Any signal	bit	A signal of type bit that toggles on every transaction of *S*
S'**event**	Any signal	boolean	True if *S* had an event in the current simulation cycle
S'**active**	Any signal	boolean	True if *S* had a transaction in the current simulation cycle
S'**last_event**	Any signal	time	Time interval since last event on *S*
S'**last_active**	Any signal	time	Time interval since last transaction on *S*
S'**last_value**	Any signal	Base type of *S*	Previous value of *S* just before its last event
S'**driving**	Any signal	boolean	False if driver in the surrounding process is disconnected
S'**driving_value**	Any signal	Base type of *S*	Value contributed to *S* by the surrounding process

Table 4: Predefined attributes of named entities

Attribute	Applied to	Return Type	Return Value
E'**simple_name**	Named entity	string	The simple name of the named entity *E*
E'**instance_name**	Named entity	string	Hierarchical path of *E* in the elaborated design, including the names of instantiated design entities
E'**path_name**	Named entity	string	Hierarchical path of *E* in the elaborated design, excluding the names of instantiated design entities

Synthesis Guidelines

Some general VHDL guidelines for good design and coding practices

- Use the `std_logic_vector` and `std_logic` data types instead of the `bit_vector` or `bit` data types.
- Use the `numeric_std` package and the `unsigned` and `signed` data types for synthesizing arithmetic operations
- Use only the descending range (i.e., **downto**) in the array specification of the `unsigned`, `signed` and `std_logic_vector` data types
- Use parentheses to clarify the intended order of evaluation.
- Don't use user-defined data types unless there is a compelling reason.
- Don't use immediate assignment (i.e., :=) to assign an initial value to a signal.
- Use operands with identical lengths for the relational operators.

Concurrent Statements

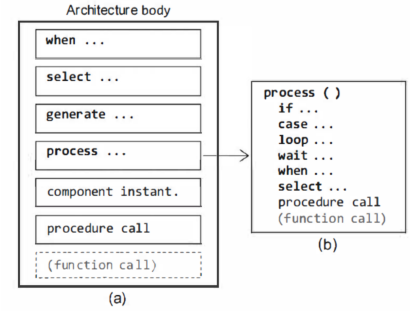
Concurrent Statements

- In VHDL, the processing elements are concurrent statements, and their communication channels are signals.
- Concurrent statements can exist only in the statement part of an architecture.
- Concurrent statements have three main characteristics:
  - they execute side by side, conceptually in parallel.
  - they execute continuously, as if in a loop.
  - they have no predefined order of execution among them.
- These three characteristics are similar to the behavior of real hardware, making concurrent statements appropriate for modeling digital circuits.
- Concurrent statements fall into one of four categories:
  - those that are a process
  - those that are a shorthand notation for a process
  - those that instantiate subcomponents
  - those that control the conditional or iterative inclusion of statements into an architecture.

Concurrent Statements

The architecture body can contain only concurrent statements
(recall that VHDL is inherently concurrent rather than sequential).

- **Concurrent statements** are the following:
  when statements; *select* statements; *generate* statements; *process* statements;
  component instantiation statements; and *procedure call* statements.
- Because these statements are concurrent, their relative positions in the code do not matter.



Main concurrent statements

Category/Subcategory		Statements
Concurrent signal assignment statements	Conditional assignment	`when`
	Selected assignment	`select`
Generate statements		`for ... generate`
		`if ... generate`
		`case ... generate`
Process statement		`process`
Component instantiation statement		`component instant.`
Concurrent procedure call statement		`procedure call`

(a) Concurrent and (b) sequential VHDL statements, all synthesizable.

Conditional Assignment

The *when* Statement

```
target <= value when condition else
          value when condition else
          value;
```

```
target <= value when condition else
          value when condition else
          value when condition;
```

Conditional Assignment

The *when* Statement

```
target <= value when condition else
          value when condition else
          value;
```

```
target <= value when condition else
          value when condition else
          value when condition;
```

- The **target** in the syntax above is a signal (VHDL-2008 allows `when` to be used also in sequential code, so there the target can be also a variable).
- Note that the `when` statement has a **priority-encoding nature** - (for any given line to be executed, the tests in all preceding lines must return false)
- hence its not tailored for entering straightforward truth tables.

```
-- Ending in when condition
y <= a when sel=0 else
     b when sel=1 else
     e when sel=2 else
     d when sel=3;
```

```
-- Ending in else value:
y <= a when sel=0 else
     b when sel=1 else
     e when sel=2 else
     d;
```

- The advantage of ending with "else value" is that it guarantees complete input-output mapping coverage, so the compiler will not infer latches.

Inferred latches

What's a latch and why is it bad?

The following example illustrates incomplete versus complete in-out mapping coverage.

```
1  -- Bad (infers latches):
2  outp <= "00" when rst else
3          "01" when hold else
4          "11" when run;
```

```
1  -- Fine (complete truth table coverage):
2  Qutp <= "00" when rst else
3          "01" when hold else
4          "11" when run else
5          "--";
```

- If the code on the left is employed, what should the output be?
- for example, when `rst, hold, run` are all **low**?

Inferred latches

What's a latch and why is it bad?

The following example illustrates incomplete versus complete in-out mapping coverage.

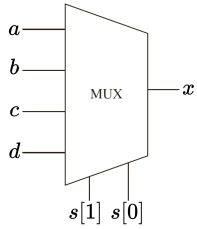
```
1  -- Bad (infers latches):
2  outp <= "00" when rst else
3          "01" when hold else
4          "11" when run;
```

```
1  -- Fine (complete truth table coverage):
2  Qutp <= "00" when rst else
3          "01" when hold else
4          "11" when run else
5          "--";
```

- If the code on the left is employed, what should the output be?
- for example, when `rst, hold, run` are all **low**?
- Since the compiler will execute anyway, it must make a decision that typically is to infer latches to hold the output's current value.
- Such latches are highly undesirable because they add delays (which are poorly predictable in FPGAs because latches are not built-in circuits), besides wasting hardware and power resources.

Example Design: Multiplexer

4-to-1 Multiplexer based on a conditional signal assignment statement




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

entity mux4 is
    port(
        a, b, c, d: in std_logic_vector(7 downto 0);
        s: in std_logic_vector(1 downto 0);
        x: out std_logic_vector(7 downto 0)
    );
end mux4;

architecture cond_arch of mux4 is
begin
    x <= a when (s="00") else
         b when (s="01") else
         c when (s="10") else
         d;
end cond_arch;
```

Selected assignment

The *select* Statement

Its main use is to enter truth tables, which select does better than when for two reasons:

- select does not posses the priority-encoding nature of when;
- full truth table coverage is checked automatically by the compiler, which is not running the compilation if the table is not completely described (hence preventing the inference of latches always).

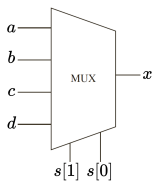
```
1  with expression select
2      target <= value when choice,
3                value when choice,
4                value when others;
```

```
1  with expression select
2      target <= value when choice,
3                value when choice,
4                value when choice;
```

- Left version employs the keyword `others` to cover all remaining cases.
- Right version describes the entire truth table explicitly, leading to a more informative code.
- However, in most cases only the [Left] is viable.

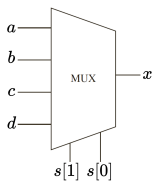
The Multiplexer Example

4-to-1 Multiplexer with both conditional signal assignment and select statements



The Multiplexer Example

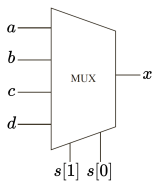
4-to-1 Multiplexer with both conditional signal assignment and select statements




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

entity mux4 is
    port(
        a, b, c, d: in stad_logic_vector(7
                downto 0);
        s: in std_logic_vector(1 downto 0);
        x: out std_logic_vector(7 downto 0)
    );
end mux4;

-- Using select statement
architecture select_arch of mux4 is
begin
    with s select
        x <= a when "00",
             b when "01",
             c when "10",
             d when others;
end select_arch;
```

The Multiplexer Example

4-to-1 Multiplexer with both conditional signal assignment and select statements




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

entity mux4 is
    port(
        a, b, c, d: in stad_logic_vector(7
                downto 0);
        s: in std_logic_vector(1 downto 0);
        x: out std_logic_vector(7 downto 0)
    );
end mux4;

-- Using select statement
architecture select_arch of mux4 is
begin
    with s select
        x <= a when "00",
             b when "01",
             c when "10",
             d when others;
end select_arch;
```

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

entity mux4 is
    port(
        a, b, c, d: in stad_logic_vector(7
                downto 0);
        s: in std_logic_vector(1 downto 0);
        x: out std_logic_vector(7 downto 0)
    );
end mux4;

-- Using conditional asignment
architecture cond_arch of mux4 is
begin
    x <= a when (s="00") else
         b when (s="01") else
         c when (s="10") else
         d;
end cond_arch;
```

The select? statement

- The matching select statement (select?) employs the matching equality comparator (?=), which, assumes the following for std_ulogic values: 'o'='L', , '1'='H', and '-'=any value.
- Any other combination returns 'U' or 'X' or 'o'.
- This statement is particularly useful when the truth table contains "don't care" values at the input.

Example: Priority encoder

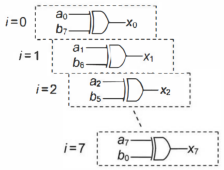
```
1  architecture sel_arch of prio_encorder42 is
2  begin
3      with r select
4          code <= "11" when "1000"|"1001"|"1010"|"1011"|
5                             "1100"|"1101"|"1110"|"1111",
6                  "10" when "0100"|"0101"|"0110"|"0111",
7                  "01" when "0010"|"0011",
8                  "00" wheb others;
9  end sel_arch;
```

```
1  architecture sel2_arch of prio_encorder42 is
2  begin
3      with r select?
4          code <= "11" when "1---",
5                  "10" when "01--",
6                  "01" when "001-",
7                  "00" wheb others;
8  end sel2_arch;
```

Generate Statements

There are three versions for this statement:

1. `for-generate` - most frequently used form.
   It acts as a loop, but because it is a concurrent statement, a piece of hardware is inferred every time the loop goes around.
2. `if-generate`, and
3. `case-generate`.

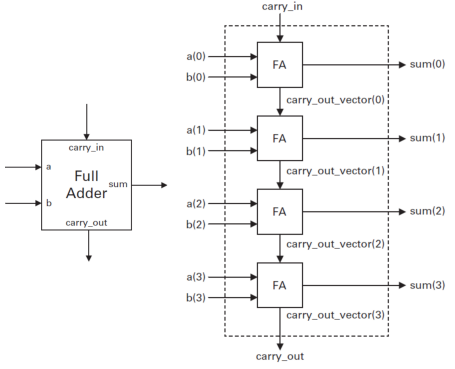



```
label : for identifier in generate_range
        generate
    [generate_declarative_part
begin]
    concurrent_statements
end generate [label] ;
```

```
label : if condition_1 generate
    concurrent_statements;
[elsif condition_n generate
    concurrent_statements;]
else generate
    concurrent_statements;]
end generate [label];
```

```
label : case expression generate
    when choices =>
        concurrent_statements;
    when others =>
        concurrent_statements;
end generate [label];
```

Generate: ripple-carry adder

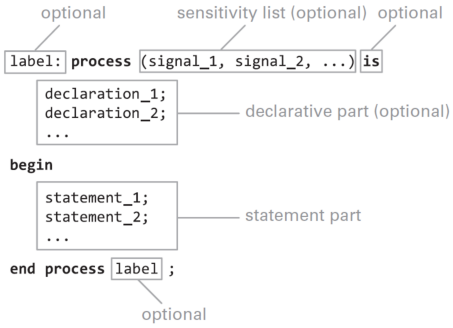



```
1  library ieee;
2  use ieee.std_logic_1164. all ;
3
4  entity adder is
5      generic (
6          WIDTH : integer := 4
7      );
8      port (
9          a, b: in std_logic_vector ( WIDTH - 1 downto 0 );
10         carry_in: in std_logic ;
11         sum: out std_logic_vector ( WIDTH - 1 downto 0 );
12         carry_out: out std_logic
13     );
14 end;
15
16 architecture structural of adder is
17     signal carry_out_vector: std_logic_vector ( WIDTH - 1 downto 0 );
18 begin
19     instantiate_adders: for i in 0 to WIDTH - 1 generate
20         first_bit: if i = 0 generate
21             adder_bit: entity work.full_adder
22                 port map (
23                     a = > a(i), b = > b(i), carry_in = > carry_in,
24                     sum = > sum(i), carry_out = > carry_out_vector(i)
25                 );
26         end generate;
27
28         other_bits: if i /= 0 generate
29             adder_bit: entity work.full_adder
30                 port map (
31                     a = > a(i), b = > b(i), carry_in = >
32                         carry_out_vector(i- 1 ),
33                     sum = > sum(i), carry_out = > carry_out_vector(i)
34                 );
35         end generate ;
36     end generate ;
37
38     carry_out < = carry_out_vector( WIDTH - 1 );
end ;
```

Process Statement

The process statement

- A process is a delimited region of sequential code inside an architecture body.
- The statements inside a process are sequential
- The process itself is a concurrent statement because it executes in parallel with all other processes and concurrent statements in an architecture.



Process Statement

The Ins and Outs of VHDL Processes

- Every process starts executing during the initialization phase of a simulation
- Once activated, a process keeps running until it executes a wait statement.
- Statements in a process are executed in an infinite loop.
- A processes is always in one of two states: executing or suspended.
- Signal values do not change while a process is running.
- A process can declare variables, which are local to the process.
- Variables declared in a process are initialized only once when the simulation begins.
- A process cannot declare signals.
- If a process makes multiple assignments to a signal, only the last assignment before the process suspends is effective.
- A process should contain a sensitivity list or wait statements but not both.
- Processes can model combinational or sequential logic.
- The order of execution among processes is random and should be irrelevant.
- Conceptually, all processes execute concurrently

The Sensitivity List

A sensitivity list is a list of signals to be monitored for changes while a process is suspended.

- When one of these signals changes, the process is reactivated.
- Useful for modeling both combinational and sequential logic.
- A process with a sensitivity list is equivalent to a process with a single wait on statement located immediately before the end process keywords.

```
1  -- Process with sensitivity list
2  process (a, b) begin
3      y <= a and b;
4  end process;
5
6  -- process with a wait statement
7  process begin
8      y <= a and b;
9      wait on a, b;
10 end process;
```

Sensitivity List for Combinational Logic

- In combinational circuits, a change in an input can potentially cause a change in any output depending on the implemented logic function.
- To ensure that the function is reevaluated whenever necessary, the sensitivity list must include all the inputs to a combinational logic block.

```
1  -- Manually written sensitivity list
2  next_state_logic: process (current_state, nickel_in,
       dime_in, quarter_in, coin_return)
3  begin
4      case current_state is
5          when accepting_coins =>
6              ...
7      end case;
8  end process;
```

```
1  -- With the reserved word all (VHDL-2008)
2  next_state_logic: process (all)
3  begin
4      case current_state is
5          when accepting_coins =>
6              ...
7      end case;
8  end process;
```

The Sensitivity List

Sensitivity List for Combinational Logic

- In combinational circuits, a change in an input can potentially cause a change in any output depending on the implemented logic function.
- To ensure that the function is reevaluated whenever necessary, the sensitivity list must include all the inputs to a combinational logic block.

```vhdl
-- Manually written sensitivity list
next_state_logic: process (current_state, nickel_in,
        dime_in, quarter_in, coin_return)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

```vhdl
-- With the reserved word all (VHDL-2008)
next_state_logic: process (all)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

Incomplete Sensitivity list

This causes a mismatch between simulation results and the synthesized circuit.

```vhdl
process (a) begin
    y <= (a and b) or c;
end process;
```

Sensitivity List for Sequential Logic

- A sequential logic circuit is one whose outputs depend on the internal system state, which is preserved in memory elements such as flip-flops or latches.
- The storage elements are usually controlled by special signals such as clock and reset.
- Sequential circuits are either synchronous and asynchronous.
- In synchronous circuits, state changes are allowed only at discrete times, dictated by the system synchronism pulse or clock.
- In asynchronous circuits, changes in the system state can happen at any time in response to changes on the inputs.

Sensitivity List for Sequential Logic

- A sequential logic circuit is one whose outputs depend on the internal system state, which is preserved in memory elements such as flip-flops or latches.
- The storage elements are usually controlled by special signals such as clock and reset.
- Sequential circuits are either synchronous and asynchronous.
- In synchronous circuits, state changes are allowed only at discrete times, dictated by the system synchronism pulse or clock.
- In asynchronous circuits, changes in the system state can happen at any time in response to changes on the inputs.

```vhdl
-- Synchronous counter with asynchronous reset.
process (clock, reset) begin
    if reset then
        cout <= 0;
    elsif rising_edge(clock) then
        if load then
            count <= load_value;
        elsif enable then
            count <= count + 1;
        end if;
    end if;
end process;
```

```vhdl
-- Synchronous counter with synchronous reset.
process (clock) begin
    if rising_edge(clock) then
        if reset then
            cout <= 0;
        elsif load then
            count <= load_value;
        elsif enable then
            count <= count + 1;
        end if;
    end if;
end process;
```

A Note on Sensitivity Lists

Note: One of the biggest problems with writing sensitivity lists is that as long as they contain valid signal names, they are legal VHDL code.
If we add a totally unrelated signal to a sensitivity list, or if we forget to add an important signal, then the compiler may give us a warning, but it will not generate an error.
The end result will be inconsistent behavior between simulation and synthesis, or even erroneous behavior of the compiled circuit.

Rules and recommendations for sensitivity lists of combinational or sequential logic

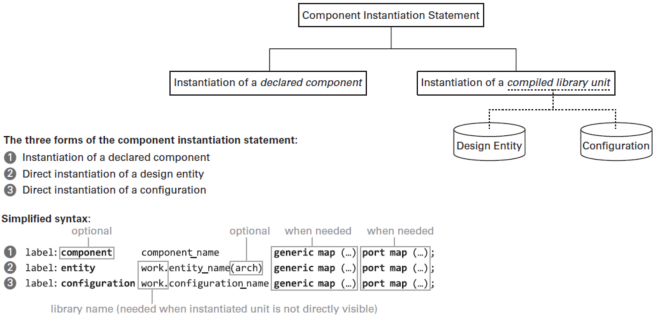
In processes modeling combinational logic, the sensitivity list ...	In processes modeling sequential logic, the sensitivity list ...
• should not include synchronism signals, such as clock and reset • should include all signals that are read within the process • is not enough to guarantee that a process is combinational	• should include the synchronism signals (clock, reset) • should include any asynchronous control signals (set, reset, enable) • should not include any other signals

In both cases, the sensitivity list ...

- should not contain unnecessary signals, for efficiency reasons
- should include all the necessary signals, or simulation results may differ from the synthesized circuit

Component Instantiation Statement

- The component instantiation statement is the basis for all structural design in VHDL, as it allows the use of subcomponents defined in other design units.
- Component instantiation statements are also concurrent statements.
- There are two basic types of component instantiation statements, defined by the kind of unit being instantiated: it is possible to instantiate a declared component or a compiled library unit. The library unit can be a design entity or a configuration.



Entity Instantiation

```vhdl
-- File adder.vhd

entity adder is
    port (
        a, b: in integer;
        sum: out integer
    );
end;

architecture rtl of adder is
begin
    sum <= a + b;
end;


--
```

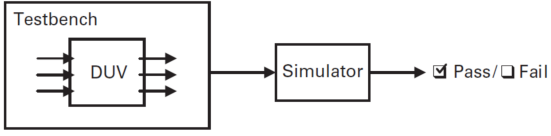
```vhdl
-- File entity_instantiation.vhd

entity entity_instantiation is
    port (
        addend, augend: in integer;
        sum: out integer
    );
end;

architecture example of entity_instantiation is
begin
    entity_inst: entity work.adder
        port map(
            a => addend,
            b => augend,
            sum => sum
        );
end;
```

Component Instantiation

```vhdl
-- File component_instantiation.vhd

entity component_instantiation is
    port (
        addend, augend: in integer;
        sum_1: out integer
    );
end;

architecture example of component_instantiation is
    component adder is
        port (
            a, b: in integer;
            sum: out integer
        );
    end component;
begin
    component_inst: adder
        port map(
            a => addend,
            b => augend,
            sum => sum
        );
end;
```

Verification of Combinational Circuits

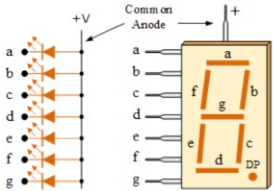
- **Functional verification** - ensuring that the design performs the operation required of it.
- In a combinational circuit we can simply verify that the circuit produces the required output for each combination of input values.
- We develop a **testbench model** that provides input values to the **design under verification (DUV)** and checks that the output values are correct.
- The **DUV** is also frequently called **a device under test (DUT)**
- The testbench model is, itself, a VHDL model that we can execute using a simulator - its purpose is to apply a sequence of values, called **test cases**, to the input connections of the DUV, and to monitor the output connections to ensure that correct values are produced.



Example: A BCD to 7-Segment Decoder

- The design should convert a 4-bit binary input representing a decimal digit to a 7-bit output appropriate for driving a 7-segment display.

Hex	Common Anode	a	b	c	d	e	f	g	Cathode(6:0)
0	high	low	low	low	low	low	low	high	0000001
1	high	high	low	low	high	high	high	high	1001111
2	high	low	low	high	low	low	high	low	0010010
3	high	low	low	low	low	high	high	low	0000110
4	high	low	low	low	high	high	low	low	1001100
5	high	low	high	low	low	high	low	low	0100100
6	high	low	high	low	low	low	low	low	0100000
7	high	low	low	low	high	high	high	high	0001111
8	high	low	low	low	low	low	low	low	0000000
9	high	low	low	low	low	high	low	low	0000100
A	high	low	low	low	low	low	high	low	0000010
B	high	high	high	low	low	low	low	low	1100000
C	high	low	high	high	low	low	low	high	0110001
D	high	high	low	low	low	low	high	low	1000010
E	high	low	high	high	low	low	low	low	0110000
F	high	low	high	high	high	low	low	low	0111000

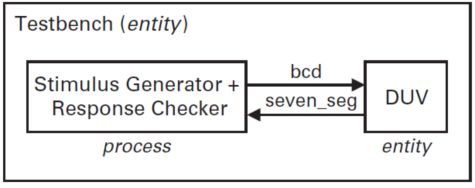


A BCD to 7-Segment Decoder

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

entity bcd_to_7seg is
  Port (
    bcd       : in std_logic_vector(3 downto 0);
    seven_seg : out std_logic_vector(6 downto 0)
  );
end bcd_to_7seg;

architecture Behavioral of bcd_to_7seg is

begin
   with bcd select
    seven_seg <= "0000001" when "0000",
                 "1001111" when "0001",
                 "0010010" when "0010",
                 "0000110" when "0011",
                 "1001100" when "0100",
                 "0100100" when "0101",
                 "0100000" when "0110",
                 "0001111" when "0111",
                 "0000000" when "1000",
                 "0000100" when "1001",
                 "0001000" when "1010",
                 "1100000" when "1011",
                 "0110001" when "1100",
                 "1000010" when "1101",
                 "0110000" when "1110",
                 "0111000" when "1111",
                 "1111111" when others; -- Default case (All segments off)
end Behavioral;
```

A Linear Testbench

- The simplest testbench instantiates the DUV in a top-level design entity and then generating the stimuli and checking the responses in a single process statement.
- This kind of testbench is called **a linear testbench** because everything happens in a single thread of execution.



Skeleton code for the linear testbench

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

entity bcd_seveg_seg_tb is
end bcd_seveg_seg_tb;

architecture linear_testbench of bcd_seveg_seg_tb is
    -- Testbench signals
    signal bcd : std_logic_vector (3 downto 0);
    signal seven_seg: std_logic_vector (6 downto 0);

begin
    -- (DUT - Device Under Test)
    DUV: entity work.bcd_to_7seg
        port map (bcd => bcd, seven_seg => seven_seg);

    -- Test Stimulus Process
    stimuli_and_checker: process
    begin
        report "Testing entity bcd_seven_seg.";

        -- TODO: Testbench code here

        report "End of testbench. All tests passed.";
        std.env.finish;
    end process;
end linear_testbench;
```

A complete code for a simple linear testbench

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

entity bcd_seveg_seg_tb is
end bcd_seveg_seg_tb;

architecture linear_testbench of bcd_seveg_seg_tb is
    -- Testbench signals
    signal bcd : std_logic_vector (3 downto 0);
    signal seven_seg: std_logic_vector (6 downto 0);

    -- Define a small delay for signals to propagate
    constant C_PROP_DELAY : time := 10 ns;

begin
    -- (DUT - Device Under Test)
    DUV: entity work.bcd_to_7seg
        port map (bcd => bcd, seven_seg => seven_seg);

    -- Test Stimulus Process
    stimuli_and_checker: process
    begin
        report "Testing entity bcd_seven_seg.";

        -- Test 0
        bcd <= "0000"; wait for C_PROP_DELAY;
        -- Test 1
        bcd <= "0001"; wait for C_PROP_DELAY;
        -- Test 2
        bcd <= "0010"; wait for C_PROP_DELAY;
        -- Test 3
        bcd <= "0011"; wait for C_PROP_DELAY;
```

```vhdl

        -- Test 4
        bcd <= "0100"; wait for C_PROP_DELAY;
        -- Test 5
        bcd <= "0101"; wait for C_PROP_DELAY;
        -- Test 6
        bcd <= "0110"; wait for C_PROP_DELAY;
        -- Test 7
        bcd <= "0111"; wait for C_PROP_DELAY;
        -- Test 8
        bcd <= "1000"; wait for C_PROP_DELAY;
        -- Test 9
        bcd <= "1001"; wait for C_PROP_DELAY;
        -- Test A (10)
        bcd <= "1010"; wait for C_PROP_DELAY;
        -- Test B (11)
        bcd <= "1011"; wait for C_PROP_DELAY;
        -- Test C (12)
        bcd <= "1100"; wait for C_PROP_DELAY;
        -- Test D (13)
        bcd <= "1101"; wait for C_PROP_DELAY;
        -- Test E (14)
        bcd <= "1110"; wait for C_PROP_DELAY;
        -- Test F (15)
        bcd <= "1111"; wait for C_PROP_DELAY;

        report "End of testbench. All tests passed.";
        std.env.finish; -- Stops the simulation
    end process;

end linear_testbench;
```

Testbench

Another version

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

entity bcd_seveg_seg_tb is
end bcd_seveg_seg_tb;

architecture linear_testbench of bcd_seveg_seg_tb is
    signal bcd : std_logic_vector (3 downto 0);
    signal seven_seg: std_logic_vector (6 downto 0);

    constant TEST_PERIOD: time := 20 ns;
begin
    DUV: entity work.bcd_to_7seg
        port map (bcd => bcd, seven_seg => seven_seg);

    stimuli_and_checker: process
        type stimulus_response_type is record
            bcd: std_logic_vector (3 downto 0);
            seven_segment: std_logic_vector (6 downto 0);
        end record;

        type stimulus_response_array_type is array (natural
              range <>) of
            stimulus_response_type;

        constant TEST_DATA: stimulus_response_array_type := (
            (x"0", "0000001"),
            (X"f", "0111000")
        );
    begin
```

```vhdl

        report "Testing entity bcd_seven_seg.";

        for i in TEST_DATA'range loop
            bcd <= TEST_DATA(i).bcd;
            wait for TEST_PERIOD;

            assert seven_seg = TEST_DATA(i).seven_segment
                report "Error";

        end loop;

        report "End of testbench. All tests passed.";
        std.env.finish;
    end process;
end linear_testbench;
```

"Talk is cheap. Show me the code."

— Linus Torvalds