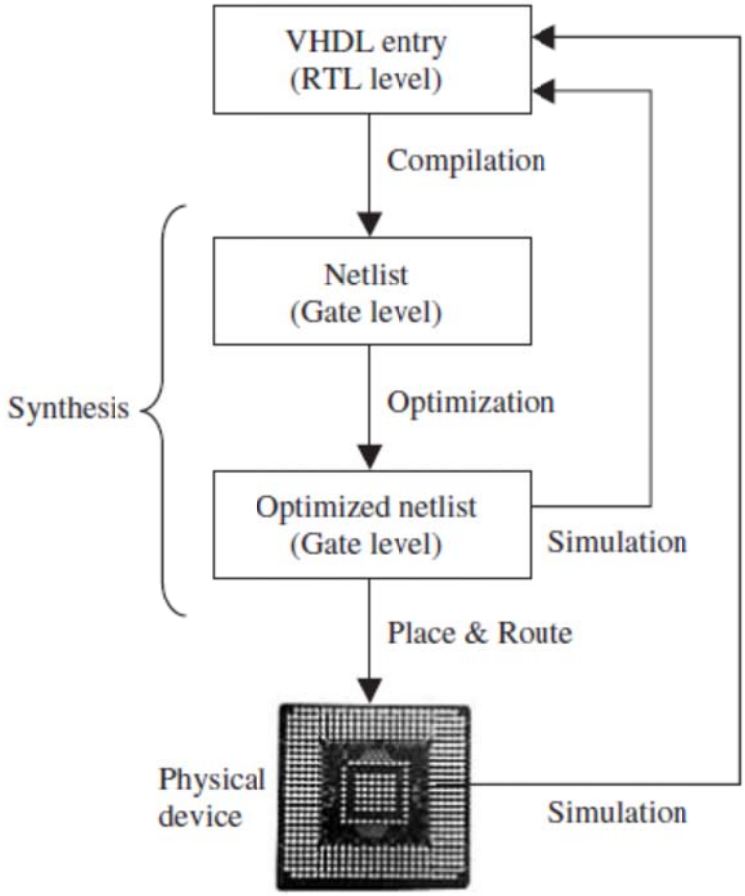
VHDL is a hardware description language. The code describes the behavior or structure of an electronic circuit.

Its main applications include synthesis of digital circuits onto CPLD/FPGA (Complex Programmable Logic Device/Field Programmable Gate Array) chips and layout/mask generation for ASIC (Application-Specific Integrated Circuit) fabrication.

VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language, and resulted from an initiative funded by the U.S. Department of Defense in the 1980s.

Its first version was VHDL 87, later upgraded by VHDL 93, then VHDL 2002, and finally VHDL 2008.

It was the first hardware description language standardized by the IEEE, through the 1076 and 1164 standards. VHDL is technology/vendor independent, so VHDL codes are portable and reusable.



EDA Tools

There are several EDA (Electronic Design Automation) tools available for circuit synthesis and simulation using VHDL. Some tools are offered by CPLD/FPGA companies (Altera, Xilinx, etc.), while others are offered by third-party software companies (Mentor Graphics, Synopsys, Cadence, etc.). Some examples are listed below.

* From Altera: Quartus II (for synthesis and graphical simulation)

* From Xilinx: ISE (XST for synthesis, ISE Simulator for simulation)

* From Mentor Graphics: Precision RTL and Leonardo Spectrum (synthesis), ModelSim (simulation)

* From Synopsys/Synplicity: Design Compiler Ultra and Synplify Pro/Premier (synthesis), VCS (simulation)

* From Cadence: NC-Sim (simulation)

* From Aldec: Active-HDL (simulation).

Number and Character Representations in VHDL

Integers:

For Integers default range in VHDL is from $-(2^{31}-1)$ to $(2^{31}-1)$

Base 10: 5, 32, 3250, 3_250, 3E2 ($3x10^2$)

Other Bases (From 2 to 16):

Base-2: 2#0111# $\rightarrow 0\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0=7$

Base-16: 16#9f# $\rightarrow 9\cdot 16^1+15\cdot 16^0=159$

Binary Values

Regular Binary Form

$'0'(=0),\quad "0111"(=7),\quad b"0111"(=7),\quad B"11110000"(=240)$

Octal and Hexadecimal Form

O"54", o"0", X"C2F", x"D"

Unsigned Values

For N bits from 0 to 2^{N-1}

Signed Values

For N bits from -2^{N-1} to $2^{N-1}-1$

The usual representation for negative numbers is two's complement

"110000" =-16

Characters

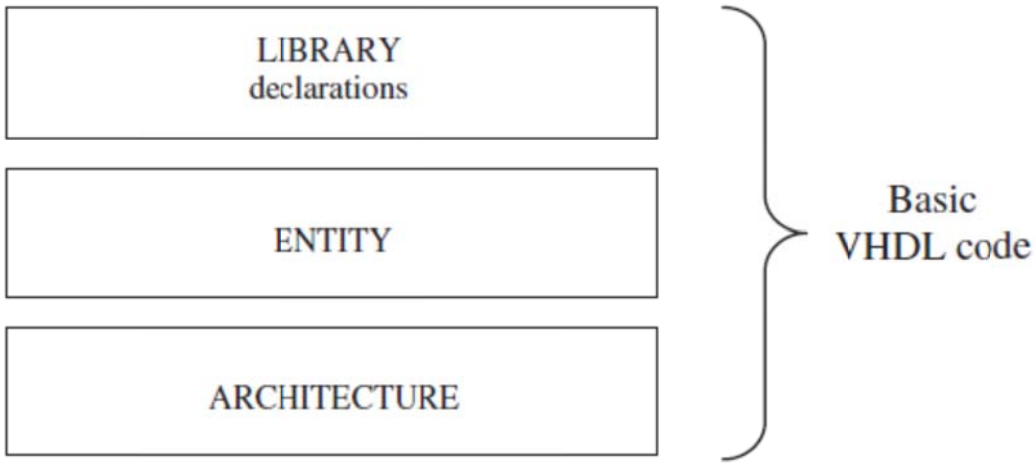
'A', 'a', '$', "VHDL", "mp3"

VHDL Code Structure

Library/package declarations: Contains a list of all libraries and respective packages needed in the design. The most commonly used libraries are ieee, std, and work

ENTITY: Specifies mainly the circuit's I/O ports, plus (optional) generic constants.

ARCHITECTURE: Contains the VHDL code proper, which describes how the circuit should function, from which a compliant hardware is inferred.



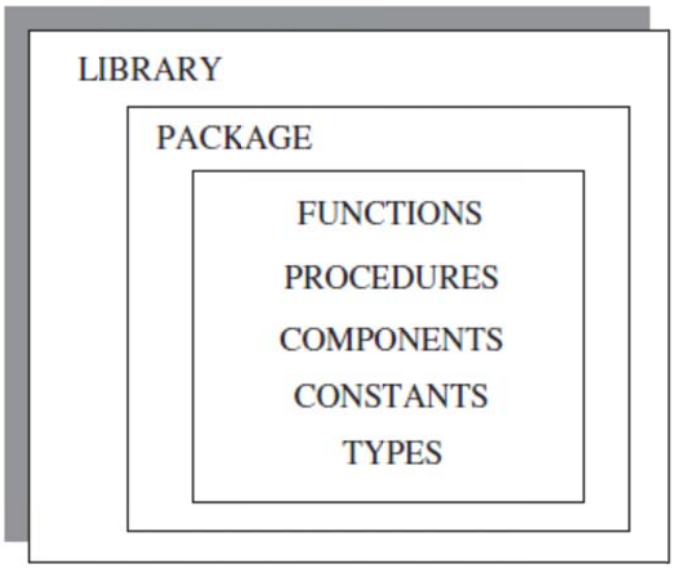


Figure 2.2
Fundamental parts of a LIBRARY.

Library/Package Declarations

LIBRARY library_name;
USE library_name.package_name.all;

Example:

LIBRARY std;
USE std.standard.all;

LIBRARY work;
USE work.all;

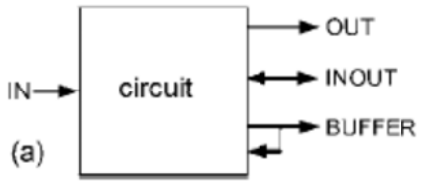
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;

ENTITY
The main part of an ENTITY is PORT, which is a list with specifications of all input and output ports (pins) of the circuit

```
ENTITY entity_name IS
      PORT (
               port_name: port_mode signal_type;
               port_name: port_mode signal_type;
        . );
END [ENTITY] [entity_name];
```

PORT mode can be IN, OUT, INOUT, or BUFFER.
BUFFER is employed when a signal is sent out but it must also be used (read) internally.

SIGNAL Type can be BIT, INTEGER, STD_LOGIC, and so on.



Example: NAND GATE

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

An entity can contain three other fields, which are

a GENERIC declarations section (before PORT),
a general declarative part (after PORT),
and finally a section with passive calls or processes (also after PORT)

```
ENTITY entity_name IS
    [GENERIC (
        const_name: const_type const_value;
        ...);]
    [PORT (
        signal_name: mode signal_type;
        ...);]
    [entity_declarative_part]
[BEGIN
    entity_statement_part]
END [ENTITY] [entity_name];
```

Example: The ENTITY below contains the first three of the four sections mentioned above.

```
ENTITY controller IS

    GENERIC (N: INTEGER := 8);

    PORT (a, b: IN INTEGER RANGE 0 TO 2**N-1;
        x: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));

    TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
    CONSTANT mask: byte "00001111";

END ENTITY;
```

ARCHITECTURE

ARCHITECTURE contains a description of how the circuit should function, from which the actual circuit is inferred.

```
ARCHITECTURE architecture_name OF entity_name IS
    [architecture_declarative_part]
BEGIN
    architecture_statements_part

END [ARCHITECTURE] [architecture_name];
```

Example:

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

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

GENERIC

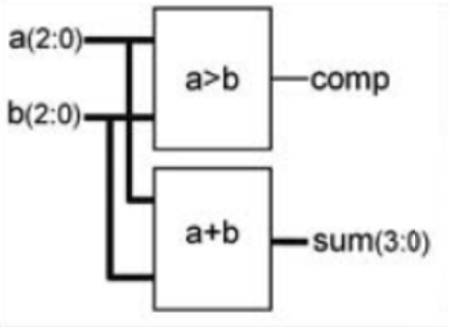
GENERIC declarations allow the specification of generic parameters (that is, generic constants, which can be easily modified or adapted to different applications).

```
GENERIC (constant_name: constant_type := constant_value;
        constant_name: constant_type := constant_value;
... );
```

Example:

```
ENTITY my_entity IS
        GENERIC (m: INTEGER := 8;
        n: BIT_VECTOR(3 DOWNTO 0) := "0101");
        PORT (...);
END my_entity;
```

Example: Compare-Add Circuit



The inputs are two unsigned 3-bit values (a and b, ranging from 0 to 7), while the outputs are comp (single bit) and sum (to avoid overflow, 4 bits are needed, hence ranging from 0 to 15). The upper part must compare a to b, producing a '1' when a > b or '0' otherwise. The lower part must add a and b, producing sum.

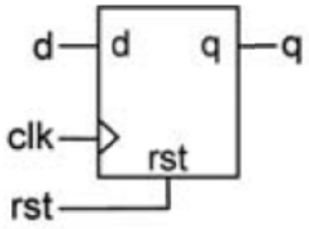
```
1    ----------------------------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    ----------------------------------------
5    ENTITY comp_add IS
6       PORT (a, b: IN INTEGER RANGE 0 TO 7;
7       comp: OUT STD_LOGIC;
8       sum: OUT INTEGER RANGE 0 TO 15);
9    END ENTITY;
10   ----------------------------------------
11   ARCHITECTURE circuit OF comp_add IS
12   BEGIN
13      comp <= '1' WHEN a>b ELSE '0';
14      sum <= a + b;
15   END ARCHITECTURE;
16 ----------------------------------------
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY comp_add IS
    PORT (
    a, b: IN INTEGER RANGE 0 TO 7;
    comp: OUT STD_LOGIC;
    sum: OUT INTEGER RANGE 0 TO 15
    );
END ENTITY;
```

```
ARCHITECTURE circuit OF comp_add IS
BEGIN
      comp <= '1' WHEN a>b ELSE '0';
      sum <= a + b;
END ARCHITECTURE;
```

Example: D-type Flip-Flop (DFF)



One must remember, however, that VHDL code is inherently concurrent (contrary to regular computer programs, which are sequential).

So to implement any clocked circuit (flip-flops, for example) we have to ‘‘force’’ VHDL to be sequential, which can be done with a PROCESS.

```
1   ---------------------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ---------------------------------------
5   ENTITY flip_flop IS
6       PORT (d, clk, rst: IN STD_LOGIC;
7       q: OUT STD_LOGIC);
8   END ENTITY;
9   ---------------------------------------
10   ARCHITECTURE flip_flop OF flip_flop IS
11   BEGIN
12      PROCESS (clk, rst)
13      BEGIN
14              IF (rst='1') THEN
15                      q <= '0';
16              ELSIF (clk'EVENT AND clk='1') THEN
17                      q <= d;
18              END IF;
```

```
19     END PROCESS;
20   END ARCHITECTURE;
21   ---------------------------------------
```

Lines 2–3: Because the data typeSTD_LOGIC is employed in this design, the package std_logic_1164 must be included.

Lines 5–8: Second part (ENTITY) of the code, in this example named flip-flop.

Lines 10–20: Third part (ARCHITECTURE) of the code, here with the same name as the entity.

Line 6: Input ports, all of type STD_LOGIC.
Line 7: Output port, also of type STD_LOGIC.

Lines 12–19: Code part of the architecture (starts after the word BEGIN). In this case, the code contains just a PROCESS, needed because we want to implement a sequential (clocked) circuit (code inside a process is executed sequentially).

Line 12: Note that two signals (clk, rst) are included in the process's sensitivity list (the process is run whenever any of these signals change).

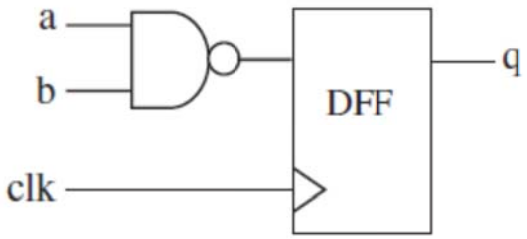
Lines 14–15: If rst goes to '1', the flip-flop is reset, regardless of clk.

Lines 16–17: If rst is not active, plus clk has changed (an EVENT occurred on clk), and such an event was a rising edge (clk = '1'), then the input signal (d) is stored into the flipflop (q <= d).

Lines 15 and 17: The operator "<=" is used to assign a value to a SIGNAL (all ports are signals by default). In contrast, ":=" would be used for a VARIABLE.

Lines 1, 4, 9, and 21: Employed to better organize the code.

Example: DFF plus NAND Gate




```
ENTITY example IS
        PORT ( a, b, clk: IN BIT;
        q: OUT BIT);
END example;

ARCHITECTURE example OF example IS
        SIGNAL temp : BIT;
BEGIN
        temp <= a NAND b;
        PROCESS (clk)
        BEGIN
                IF (clk'EVENT AND clk='1') THEN q<=temp;
                END IF;
        END PROCESS;
END example;
```

Data Types

CONSTANT

As the name says, it is an object whose value cannot be changed

CONSTANT constant_name: constant_type := constant_value;

Example:

```
CONSTANT bits: INTEGER := 16;
CONSTANT words: INTEGER := 2**bits;
CONSTANT flag: BIT := '1';
CONSTANT mask: BIT_VECTOR(1 TO 8) := "00001111";
```

CONSTANT can be declared in the declarative part of ENTITY, ARCHITECTURE, PACKAGE, PACKAGE BODY, BLOCK, GENERATE, PROCESS, FUNCTION, and PROCEDURE (the last two are called subprograms).

Keyword OTHERS

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

Examples

```
-------------------------------------------------------------
The constant below is a = "000000".
CONSTANT a: BIT_VECTOR(5 DOWNTO 0) := (OTHERS=>'0');
-------------------------------------------------------------
The next constant is b = "01111111" (index 7 gets '0', the others, '1').
CONSTANT b: BIT_VECTOR(7 DOWNTO 0) := (7=>'0', OTHERS=>'1');
-------------------------------------------------------------
The signal below is c = "01100000" ("|" means "or").
SIGNAL c: STD_LOGIC_VECTOR(1 TO 8) := (2|3=>'1', OTHERS=>'0');
-------------------------------------------------------------
The variable below is d = "1111111100000000".
VARIABLE d: BIT_VECTOR(1 TO 16) := (1 TO 8=>'1', OTHERS=>'0');
```

SIGNAL

SIGNAL serves to pass values in and out of the circuit, as well as between its internal units. In other words, a signal represents circuit wires.

Signal declarations are not allowed in sequential code (i.e., PROCESS and subprograms), but signals can be used there

```
SIGNAL signal_name: signal_type [range] [:= default_value];
```

Examples:

SIGNAL enable: BIT <= '0';
SIGNAL temp: BIT_VECTOR(3 DOWNTO 0);
SIGNAL byte: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL count: NATURAL RANGE 0 TO 255;

To assign a value to a SIGNAL, the proper operator is "<=", while for CONSTANT or VARIABLE (or for default values) it is ":=". For example, "enable <= '1 ; '"

VARIABLE
Contrary to CONSTANT and SIGNAL, VARIABLE represents only local information because it can only be seen and modified inside the sequential unit (i.e., PROCESS or subprogram) where it was created.

VARIABLE variable_name: variable_type [range] [:= default_value]

VARIABLE flip: STD_LOGIC := '1';
VARIABLE address: STD_LOGIC_VECTOR(0 TO 15);
VARIABLE counter: INTEGER RANGE 0 TO 127;

Data-Type Libraries and Packages
The fundamental packages for dealing with binary logic and with integer numbers are:

Package standard (expanded in VHDL 2008)
Package std_logic_1164 (expanded in VHDL 2008)
Package numeric_bit (expanded in VHDL 2008)
Package numeric_std (expanded in VHDL 2008)
Package std_logic_arith (shareware, nonstandard)
Package std_logic_unsigned (shareware, nonstandard)
Package std_logic_signed (shareware, nonstandard)
Package textio (expanded in VHDL 2008)
Package numeric_bit_unsigned (introduced in VHDL 2008)
Package numeric_std_unsigned (introduced in VHDL 2008)

Package standard

It defines the following data types:

Bit-related (synthesizable): BIT, BIT_VECTOR, BOOLEAN
Integer-related (synthesizable): INTEGER, NATURAL, POSITIVE
Character-related (synthesizable): CHARACTER, STRING
Floating-point (limited synthesis support): REAL
Time-related (not for synthesis): TIME, DELAY_LENGTH

In VHDL 2008, the following was added to the package standard:
New types: BOOLEAN_VECTOR, INTEGER_VECTOR, REAL_VECTOR, TIME_VECTOR.

Examples:

SIGNAL x: BIT;
-- x is declared as a one-digit signal of type BIT.

SIGNAL y: BIT_VECTOR (3 DOWNTO 0);
-- y is a 4-bit vector, with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 TO 7);
-- w is an 8-bit vector, with the rightmost bit being the MSB.

Based on the signals above, the following assignments would be legal (to assign a value to a signal, the ‘‘<=’’ operator must be used):

x <= '1';
-- x is a single-bit signal (as specified above), whose value is '1'. Notice that single quotes (' ') are used for a single bit.

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.

w <= "01110001";
-- w is an 8-bit signal, whose value is "01110001" (MSB='1').

Package std_logic_1164

The main types defined in that package are:

STD_ULOGIC, STD_ULOGIC_VECTOR
STD_LOGIC, STD_LOGIC_VECTOR

For STD_LOGIC (and STD_LOGIC_VECTOR):
8-valued logic system introduced in the IEEE 1164 standard.

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

FOR STD_ULOGIC (STD_ULOGIC_VECTOR): 9-level logic system introduced in the IEEE 1164 standard (U means unresolved)

(‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘–’).

Indeed, if any two std_logic signals are connected to the same node, then conflicting logic levels are automatically resolved according to Table below

Resolved logic system (STD_LOGIC).

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

Package **numeric_std**

The types below are defined in it:

UNSIGNED (based on STD_LOGIC)
SIGNED (also based on STD_LOGIC)

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

x3 <= "101111" -- binary representation of decimal 47

x4 <= B"101111" -- binary representation of decimal 47

x5 <= O"57" -- octal representation of decimal 47

x6 <= X"2F" -- hexadecimal representation of decimal 47

n <= 1200; -- integer

m <= 1_200; -- integer, underscore allowed

Example: Legal and illegal operations between data of di¤erent types.

```
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)

User-Defined Data Types

TYPE integer IS RANGE -2147483647 TO +2147483647;
-- This is indeed the pre-defined type INTEGER.

TYPE natural IS RANGE 0 TO +2147483647;
-- This is indeed the pre-defined type NATURAL.

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:
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 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.

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.

Subtypes

A SUBTYPE is a TYPE with a constraint. The main reason for using a subtype rather than specifying a new type is that, though operations between data of different types are not allowed, they are allowed between a subtype and its corresponding base type.

SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype (subset) of INTEGER.

SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Recall that STD_LOGIC=('X','0','1','Z','W','L','H','-').
-- Therefore, my_logic=('0','1','Z').

SUBTYPE my_color IS color RANGE red TO blue;
-- Since color=(red, green, blue, white), then
-- my_color=(red, green, blue).

SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- A subtype of INTEGER.

Example: Legal and illegal operations between types and subtypes.

```
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO '1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;
...
b <= a; --illegal (type mismatch: BIT versus STD_LOGIC)
b <= c; --legal (same "base" type: STD_LOGIC)
```

Arrays

Arrays are collections of objects of the same type. They can be one-dimensional (1D), two-dimensional (2D), or one-dimensional-by-one-dimensional (1Dx1D).

They can also be of higher dimensions, but then they are generally not synthesizable.

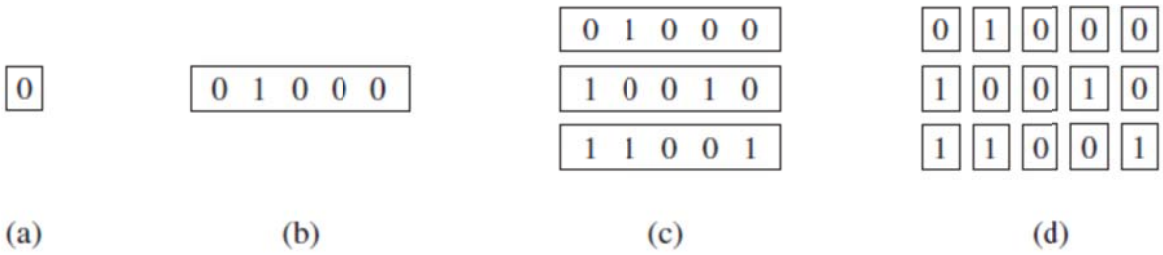


Illustration of (a) scalar, (b) 1D, (c) 1Dx1D, and (d) 2D data arrays.

TYPE type_name IS ARRAY (specification) OF data_type;

SIGNAL signal_name: type_name [:= initial_value];

Example:

TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D array

TYPE matrix IS ARRAY (0 TO 3) OF row; -- 1Dx1D array

SIGNAL x: matrix; -- 1Dx1D signal