Computer Architecture Lab

Introduction

The VHSIC Hardware Description Language is an industry standard language used to describe hardware from the abstract to the concrete level. VHDL resulted from work done in the ’70s and early ’80s by the U.S. Department of Defense. Its roots are in the ADA language, as will be seen by the overall structure of VHDL as well as other VHDL statements. VHDL usage has risen rapidly since its inception and is used by literally tens of thousands of engineers around the globe to create sophisticated electronic products. VHDL is a powerful language with numerous language constructs that are capable of describing very complex behavior.

Design Flow

A simplified view of the design flow is presented in figure 1. We assume that the designer already has a set of specifications for which a compliant circuit should be generated. The first step is to write a VHDL code that fulfills such specifications. The code must be saved in a text file with the extension .vhd and the same name as that of its main entity. Next, the code is compiled using a synthesis tool. Several files are generated during the compilation process. The synthesizer breaks down the code into the hardware structures that are available within the chosen device, so during fitting (place and route) each structure inferred by the synthesizer is assigned a specific place inside the device. This positional information is important because it greatly influences the resulting circuit’s timing behavior. With the timing information generated by the fitting process, the software allows the circuit to be fully simulated. Once the specifications have been met, the designer can proceed to the final step (implementation), during which a programming file for the device (when using a CPLD or FPGA) or for the masks (for ASICs) is generated. In the case of CPLDs/FPGAs, the design is concluded by downloading the programming file from the computer to the target device.

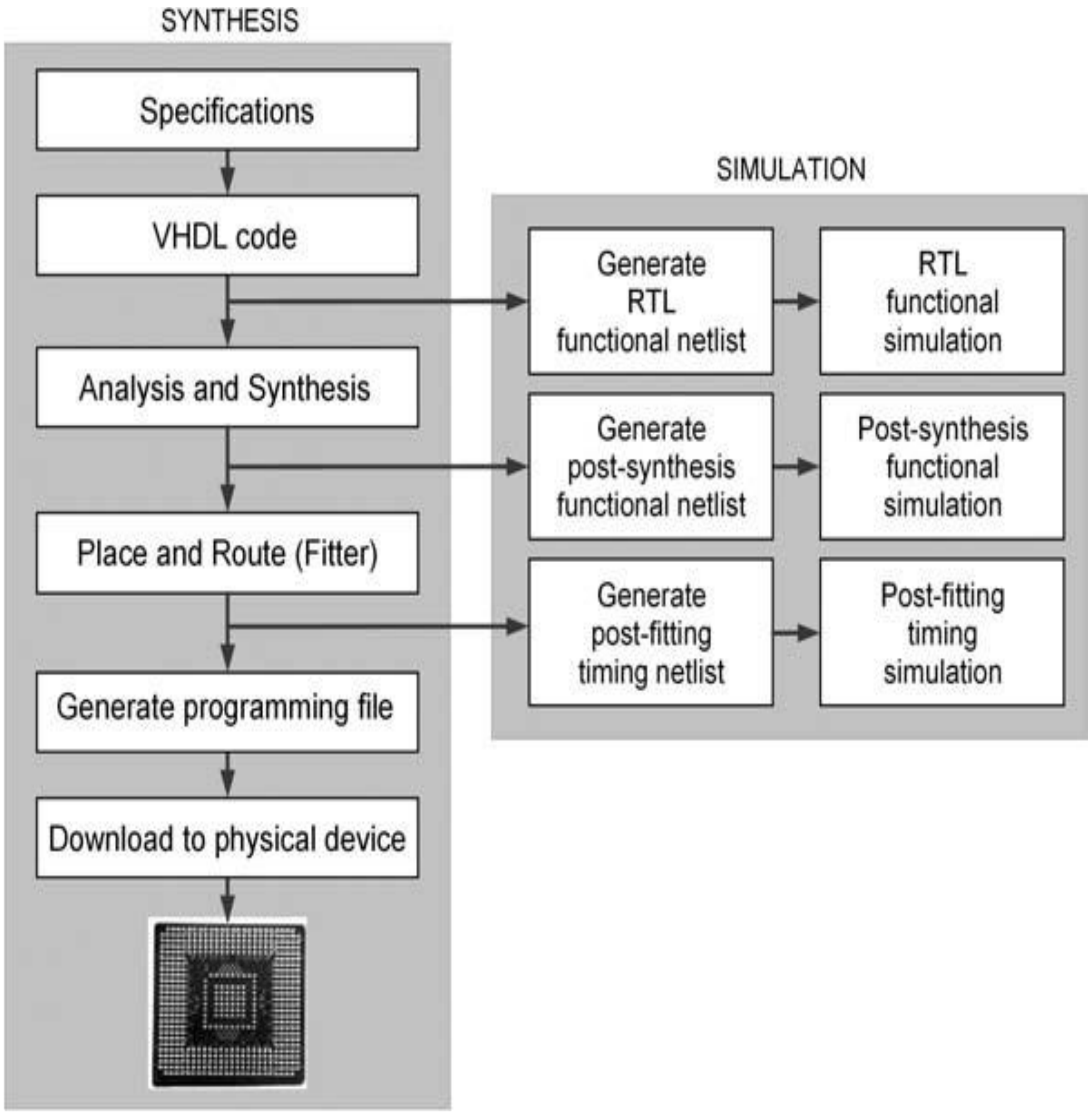
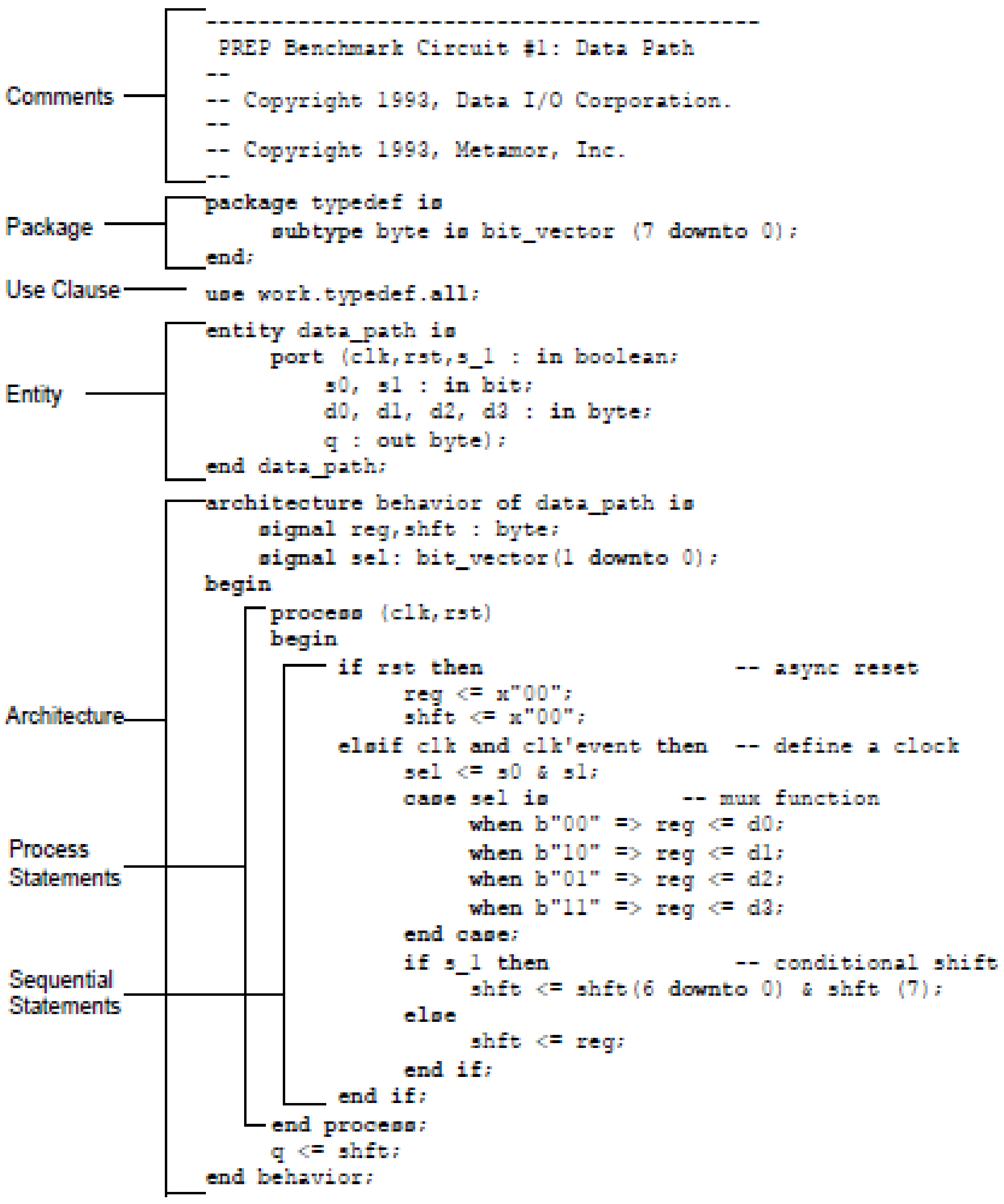

SYNTHESIS
Specifications
VHDL code
Analysis and Synthesis
Place and Route (Fitter)
Generate programming file
Download to physical device
SIMULATION
Generate RTL functional netlist
RTL functional simulation
Generate post-synthesis functional netlist
Post-synthesis functional simulation
Generate post-fitting timing netlist
Post-fitting timing simulation

Figure 1, Simplified VHDL design flow.

The Structure of a VHDL Design Description

The basic organization of a VHDL Design Description is shown in figure 2. The sample file shown includes an entity- architecture pair and a package.



```
----------------------------------------
 PREP Benchmark Circuit #1: Data Path
--
-- Copyright 1993, Data I/O Corporation.
--
-- Copyright 1993, Metamor, Inc.
--
package typedef is
     subtype byte is bit_vector (7 downto 0);
end;
use work.typedef.all;
entity data_path is
     port (clk,rst,s_l : in boolean;
          s0, s1 : in bit;
          d0, d1, d2, d3 : in byte;
          q : out byte);
end data_path;
architecture behavior of data_path is
    signal reg,shft : byte;
    signal sel: bit_vector(1 downto 0);
begin
     process (clk,rst)
     begin
          if rst then                    -- async reset
               reg <= x"00";
               shft <= x"00";
          elsif clk and clk'event then  -- define a clock
               sel <= s0 & s1;
               case sel is          -- mux function
                    when b"00" => reg <= d0;
                    when b"10" => reg <= d1;
                    when b"01" => reg <= d2;
                    when b"11" => reg <= d3;
               end case;
               if s_l then               -- conditional shift
                    shft <= shft(6 downto 0) & shft (7);
               else
                    shft <= reg;
               end if;
          end if;
     end process;
     q <= shft;
end behavior;
```

Figure 2, the Structure of a VHDL Design Description

Describing Hardware in VHDL

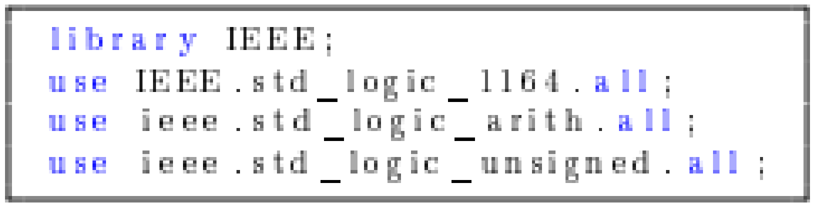
VHDL Descriptions consist of primary design units and secondary design units. The primary design units are the Entity and the Package. The secondary design units are the Architecture and the Package Body. Secondary design units are always related to a primary design unit. Libraries are collections of primary and secondary design units. A typical design usually contains one or more libraries of design units.

Basic Elements of VHDL

A digital system is basically described by its inputs and its outputs, as well as how these outputs are obtained from the inputs.

The VHDL code of any circuit is divided into two separate parts: On the one hand, the entity specifies the input and output ports of the circuit. On the other hand, the architecture describes the behavior of that circuit. An architecture must be associated with an entity. It is also possible to associate several architectures to the same entity, so the programmer can select one of the available ones

The IEEE library and the following three packets (whose meaning is explained below) appear by default in any source VHDL code created



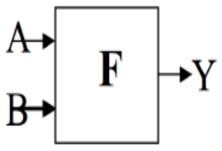
```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

Entity

An entity is an abstraction of a circuit, either from a complex electronic system or a single logic gate. An entity externally describes the I/O interface of the circuit.

The ports of an entity can be inputs (in), outputs (out), input-outputs (in out) or buffer. The input ports can only be read, and they cannot be modified inside the architecture. On the other hand, the output ports can only be written, but not read. In case an output port needs to be read (for instance, to make a decision about its value) or an input port needs to be written, they must be instantiated as an inout or a buffer port. However, in this course we will try to avoid these situations, so the utilization of in out and buffer ports are beyond the learning outcomes of this course.

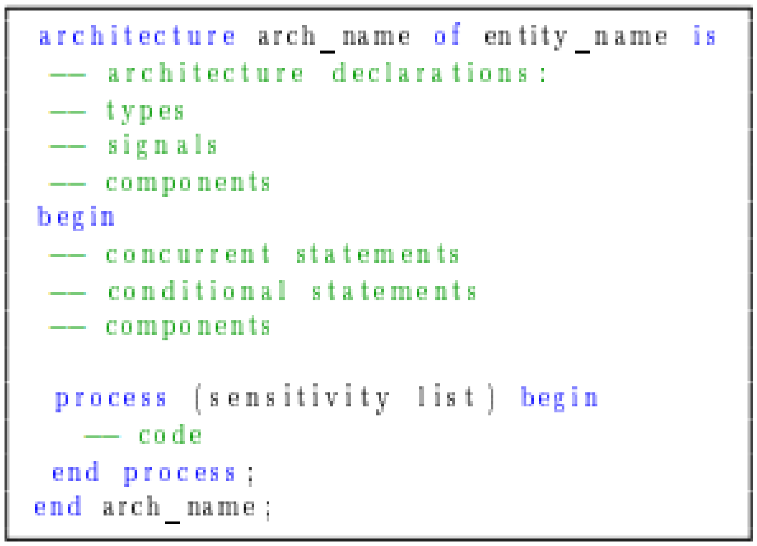
The interface described by an entity may also include a set of generic values that are used to declare properties and constants of the circuits, independently of its architecture. Generics can have multiple uses: On the one hand, they can be used to dene delays in signals and clock cycles (these definitions will not be taken into account at the synthesis level, as explained later throughout this manual). On the other hand, generics can also be used as constants that will be used inside the architecture. These constants help to make the code more understandable, portable and maintainable. For instance, the length of a register (in number of bits) can be defined by means of a generic parameter. This means that another VHDL code can instantiate this register several times, even if this code instantiates registers with different number of bits. Generic parameters are not necessary. Hence, a circuit that does not need them, it simply does not instantiate any generic statement in the entity declaration. The example below shows a description of the entity of a circuit. This circuit has two N-bit inputs (A and B) and a single output (Y). Thus, in this case the entity description includes a generic statement defining a parameter named N whose value is set to 8. This parameter is also used in the declaration of the circuit inputs.



```
entity F is
 generic (N: natural := 8);
 port (A, B: in bit_vector (N-1 downto 0); Y: out bit);
end F;
```

Architecture

The pair entity-architecture are used in VHDL to completely describe the operation of a circuit. An architecture defines how the circuit operates, by including a set of inner signals, functions, procedures, functions... and its description can be either structural or behavioral. The code below shows an example of an architecture. The association between this architecture and the entity it refers to is made in the first line (architecture arch_name of entity_name is). Next, the code must include the signals, customized types, and components (whose I/O is known) that will be used inside the architecture.



```
architecture arch_name of entity_name is
 -- architecture declarations:
 -- types
 -- signals
 -- components
begin
 -- concurrent statements
 -- conditional statements
 -- components

  process (sensitivity list) begin
    -- code
  end process;
end arch_name;
```

The begin and the end reserved words mark the boundaries of the VHDL code that will actually describe the operation of the circuit. As shown in the example, this code may include: concurrent and conditional statements, components and processes.

GENERIC

GENERIC declarations allow the specification of generic parameters (that is, generic constants, which can be easily modified or adapted to different applications). Their purpose is to parameterize a design, conferring the code more flexibility and reusability.
As seen in the syntax for ENTITY, GENERIC is the only declaration allowed before the PORT clause, which causes such constants to be truly global because they can be used even in the PORT specifications. A simplified syntax for GENERIC declarations is shown below.

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

Example The GENERIC declaration in the entity below specifies two parameters, called m and n. The first is of type INTEGER and has value 8, while the second is of type BIT_VECTOR and has value "0101". Therefore, whenever m and n are encountered in the code (including in the ENTITY itself), the values 8 and "0101" are automatically assigned to them.

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

GENERIC MAP: If a COMPONENT containing a GENERIC declaration (like the one above) is instantiated in another design, the values of the generic constants that appear in the component being instantiated can be overwritten by the main design. This is done with a GENERIC MAP declaration. In VHDL, besides the traditional generic constants, generic types and generic subprograms are also supported. A generic constant can be used in the specification of other generic constants in the same generic list. The where generics can be declared were also expanded; besides ENTITY and BLOCK places headers, it can also be done in PACKAGE.

VHDL objects

- **Variable**: Objects that take a single value that can change during the simulation/execution by means of an assignment statement. Variables are usually used as indexes, mainly in loops, or to take values that allow to model other components. Variables DO NOT represent physical connections or memory elements. They can be declared before the `begin` of an `architecture`, and/or before the `begin` of a `process`. A variable declaration MAY or MAY NOT assign a value to it.

```
variable identifier: type [:= value];
```

The assignment of a value to a variable is done by means of the operator `:=`

```
name_variable := value;
i := 10;
```

- **Signal**: Objects that represent memory elements or connections between subcircuits. Contrarily to constants and variables, signals can be synthesized. In other words, a signal in a VHDL source code can be physically translated into a memory element (flip-flop, register...) in the final circuit. They must be declared before the `begin` of the `architecture`. The ports of an `entity` are implicitly declared as signals upon declaration, since they represent physical connections in the circuit.

```
signal identifier: type;
```

The assignment of a value to a signal is done by means of the operator `<=`

```
name_signal <= value;
A <= 10;
```

VHDL types

In the previous definitions, as well as in the definition of the entity ports, it is necessary to define the type of the object. VHDL allows to use predefined types, as well as other user-defined ones.

Predefined types

The most commonly used predefined types are the following ones:

- **bit:** It only admits the values 0 and 1. In order to make an assignment between the object and its value, the latter must be written between single quotes ('0' or '1').
- **bit_vector (range):** The range, always written between brackets, indicates the number of bits of the bit_vector, which is an array of 0's and 1's. For an n-bit bit_vector, its range must be written in the format N-1 downto 0. The bit located to the far left is the most significant one (Most Significant Bit, or MSB), whereas the bit located to the far right is the least significant one (Least Significant Bit or LSB). In order to make an assignment between the object and its value, the latter must be written between quotation marks (i.e., “0011”).
- **Boolean:** It only can take the values true or false.
- **Character:** It can take any ASCII value.
- **String:** Any chain consisting of ASCII characters.
- **Integer range:** Any integer number within the range, which in this case is not written between brackets. For instance, 0 to MAX. The range is optional.
- **Natural range:** Any natural number within the range. The range is optional.
- **Positive range**: Any positive number within the range. The range is optional.
- **Real range:** Any real number within the range. The range is optional.
- **std_logic:** Type predefined in the IEEE 1164 standard. This type represents a multivalued logic comprising 9 different possible values. The most commonly used ones are: '0', '1', 'Z' (for high impedance), 'X' (for

uninitialized) and 'U' (for undefined), among others. In order to make an assignment between the object and its value, the latter must be written between single quotes ('0', '1', 'X', ...).

- **std_logic_vector (range):** It represents a vector of elements of type std_logic. Its assignment and denition rules are the same ones as the std_logic ones.

User-defined types

An enumerated type is a data type that comprises a number of user-defined values. Enumerated types are used mainly for the definition of finite state machines.

```
type name is (value1 , value2 , ...);
```

Assuming that A has been defined as an enumerated type, the assignment will be as follows: A <= valuei; where valuei must be one of the enumerated values in the type definition. Enumerated types are sorted according to their values. Typically, synthesis tools automatically code the enumerated values in such a way that they can be further synthesized. For that purpose, they usually select an ascending sequence or a coding that minimizes the circuit or that maximizes its operating frequency. It may also be possible to directly type the coding by means of ad-hoc directives. A composed type is a data type comprised by elements of other data types. Composed types can be either arrays or records.

- An array is a data object that comprises a set of elements of the same type.

```
type name is array (range) of type;
```

The assignment of a value on a position of the array is done by means of integer numbers.

Once defined the composed and/or enumerated data type, any signal in the design can be declared of belonging to this new type and this will be done by using the operator defined for signals <=.

Examples

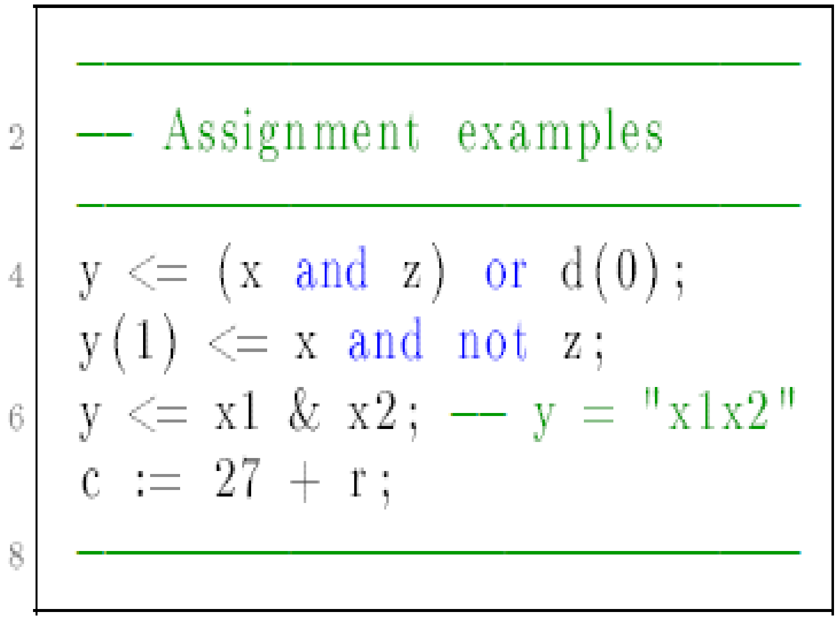
This subsection presents some examples showing how to define and to assign values to signals and variables.

```vhdl
--------------------------------------------------------------------
-- Two dashes are used to introduce comments in the VHDL code
-- Examples of definitions and assignments
--------------------------------------------------------------------
constant DATA_WIDTH: integer := 8;
signal CTRL: bit_vector(7 downto 0);
variable SIG1, SIG2: integer range 0 to 15;
--------------------------------------------------------------------
type color is (red, yellow, blue);
signal BMP: color;
BMP <= red;
--------------------------------------------------------------------
type word is array (0 to 15) of std_logic_vector (7 downto 0);
signal w: word;
-- w(integer/natural) <= vector of bits;
w(0) <= "00111110";
w(1) <= "00011010";
...
w(15) <= "11111110";
--------------------------------------------------------------------
type matrix is array (0 to 15)(7 downto 0) of std_logic;
signal m: matrix;
m(2)(5) <='1';
--------------------------------------------------------------------
type set is record
 word: std_logic_vector (0 to 15);
 value: integer range -256 to 256;
end record;
signal data: set;
data.value <= 176;
```

Operators

Operators can be used to build a wide variety of expressions that allow to calculate data and/or to assign them to signals or variables.

- +, -, *, /, mod, rem: Arithmetic operations.
- +, -: Sign change.
- &: Concatenation.
- and, or, nand, nor, xor: Logical operations.
- :=: Value assignment to constants and variables.
- <=: Value assignment to signals



```
------------------------------------------
-- Assignment examples
------------------------------------------
y <= (x and z) or d(0);
y(1) <= x and not z;
y <= x1 & x2; -- y = "x1x2"
c := 27 + r;
------------------------------------------
```

Basic Structure of a Source File in VHDL

As previously pointed out, the VHDL code modeling a digital circuit is composed of two parts: an entity and one or several architectures. The latter contains the statements describing the behavior of the circuit.

```vhdl
architecture circuit of name is
  -- signals
begin
  -- concurrent statements (assignment statements to signals)
  process (sensitivity list) begin
   -- conditional statements (assignment statements to
   variables)
  end process;
end architecture circuit;
```

Inside the architecture, we can find:

- Types and intermediate signals needed to describe its behavior.
- Assignment statements to signals, as well as other concurrent statements.
- Processes, which may contain conditional and/or assignment statements to variables

Concurrent statements

Concurrent statements are a kind of assignment statements to signals whose operation depends on a set of conditions. Two kinds of concurrent statements exist:

WHEN-ELSE

```vhdl
signal_to_modify <= value_1 when condition_1 else
                    value_2 when condition_2 else
                    ...
                    value_n when condition_n else
                    default_value;
```

This statement modifies the value of a given signal depending on a set of conditions, being the assigned values and the conditions independent among each other. The order in which the conditions are sorted determines their preference with respect to the others. In other words, in the previous definition, if condition_i is true, then value_i will be assigned to signal_to_modify, even if any other condition_j is also true (j>i).

```
--------------------------------------------
-- Examples WHEN-ELSE
--------------------------------------------
C <= "00" when A=B else
     "01" when A < B else
     "10";
--------------------------------------------
C <= "00" when A=B else
     "01" when D = "00" else
     "10";
--------------------------------------------
```

WITH-SELECT-WHEN

```
with signal_condition select
signal_to_modify <=  value_1 when value_1_signal_condition,
                     value_2 when value_2_signal_condition,
                     ...
                     value_n when value_n_signal_condition,
                     default_value when others;
```

This statement is less general than when-else one. It modifies the value of a signal, depending on the values that signal_condition may have.

```
------------------------------------
-- Example WITH-SELECT-WHEN
------------------------------------
with input select
output <=   "00" when "0001",
            "01" when "0010",
            "10" when "0100",
            "11" when others;
------------------------------------
```

From the point of view of the hardware, these two statements give as a result pure combinatorial hardware; in other words, logic gates, multiplexers, decoders...

Conditional statements

These statements are assignment statements to variables that may or may not be based on a condition. As previously pointed out, they MUST be placed inside a process. The following conditional statements exist in VHDL:

IF-THEN-ELSE

```
process (sensitivity list)
begin
if condition_1 then
 -- assignments
elsif condition_2 then
 -- assignments
else
 -- assignments
end if;
end process;
```

```
---------------------------------------
-- Example IF-THEN-ELSE
---------------------------------------
process (control, A, B)
begin
if control = "00" then
 output <= A + B;
elsif control = "11" then
 output <= A - B;
else
 output <= A;
end if;
end process;
---------------------------------------
```

It is possible to chain as many if-then-else statements as desired, as in software description languages, such as Pascal, C, Java...

CASE-WHEN

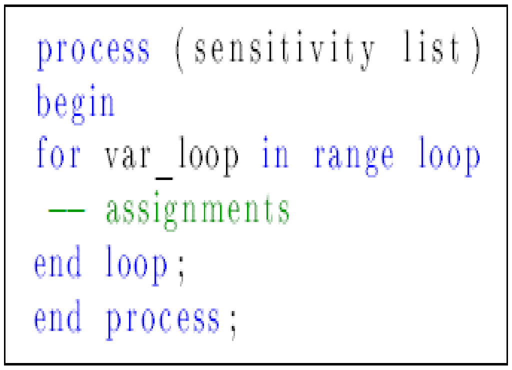
```
process (sensitivity list)
begin
case signal_condition is
 when value_condition_1 => -- assignments
 ...
 when value_condition_n => -- assignments
 when others => -- assignments
end case;
end process;
```

In this case, assignments may also be if-then-else statements. The when others clause must appear in the statement, but it is not necessary to write any assignment associated to it.

```
--------------------------------------------------
-- Example CASE-WHEN
--------------------------------------------------
process (control, A, B)
begin
   case control is
    when "00" => result <= A+B;
    when "11" => result <= A-B;
    when others => result <= A;
 end case;
end process;
--------------------------------------------------
```

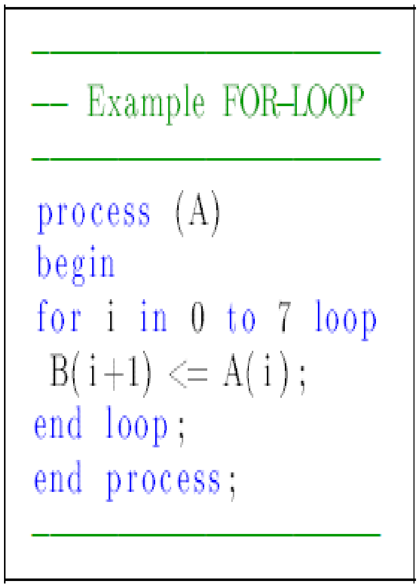
As in other software programming languages, several types of loops are possible:

FOR-LOOP



```
process (sensitivity list)
begin
for var_loop in range loop
 -- assignments
end loop;
end process;
```

The range can be defined as 0 to N or as N downto 0.



```
--------------------------------
-- Example FOR-LOOP
--------------------------------
process (A)
begin
for i in 0 to 7 loop
 B(i+1) <= A(i);
end loop;
end process;
--------------------------------
```

WHILE-LOOP

```
process (sensitivity list)
begin
 while condition loop
   -- assignments
 end loop;
end process;
```

```
----------------------------------------
-- Example WHILE-LOOP
----------------------------------------
process (A)
 variable i: natural := 0;
begin
while i < 7 loop
 B(i+1) <= A(i);
 i := i+1;
end loop;
end process;
----------------------------------------
```

PROCESS

PROCESS is a sequential section of VHDL code, located in the statements part of an architecture. Inside it, only sequential statements (IF, WAIT, LOOP, CASE) are allowed. A simplified syntax is shown below.

```
[label:] PROCESS [(sensitivity_list)] [IS]
   [declarative_part]
BEGIN
  sequential_statements_part
END PROCESS [label];
```

As shown in the syntax, the label, whose purpose is to improve readability in long codes, is optional. The sensitivity list is mandatory (but is forbidden when WAIT is used), and causes the process to be run every time a signal in the list changes (or the condition associated with WAIT is fulfilled).

The declarative part of PROCESS can contain the following: subprogram declaration, subprogram body, type declaration, subtype declaration, constant declaration, variable declaration, file declaration, alias declaration, attribute declaration, attribute specification, use clause, group template declaration, and group declaration. Signal declaration is not allowed, while variable is by far the most common declaration.

In the statements part of PROCESS, only sequential statements are allowed (besides operators, because these can go in any kind of code). Example the (partial) process below is executed whenever clk or rst changes. It contains three variable declarations *(*a; b; c*)*, the first two specified as INTEGER, the last one as BIT_VECTOR. Only for c a default value (optional) was entered.

```
PROCESS (clk, rst)
    VARIABLE a, b: INTEGER RANGE 0 TO 255;
    VARIABLE c: BIT_VECTOR(7 DOWNTO 0) := "00001111";
BEGIN
...
END PROCESS;
```

In VHDL, the following is allowed in the declarative part of PROCESS besides the items already listed above: subprogram instantiation declaration, package declaration, package body, and package instantiation declaration. Additionally, the keyword ALL was introduced for the sensitivity list (to reduce errors when implementing combinational circuits with sequential code).

Structural Description

This description is used to create an architecture that instantiates other entities that have already been defined elsewhere. This makes possible to build hierarchical descriptions of circuits, which improves their reusability and scalability. In order to do this, such an architecture must declare the entities that will be instantiated as components, and add as many instances of these components as needed in the body of the architecture, as the following code illustrates. Structural descriptions are very useful in bottom-up hierarchical designs.

```
architecture circuit of name is
 component subcircuit
   port (...);
 end component;

 -- signals
 ...
begin
 -- "chip_i" is the name of the instance declared in this code
 -- "subcircuit" is the name of the component that is used
 chip_i: subcircuit port map (...);
 -- This can be combined with behavioral descriptions
end circuit;
```

The architecture may add as many instances of the same component as needed. The only restriction that VHDL imposes is that each one of the component instances must be given a different name in the body of the architecture.

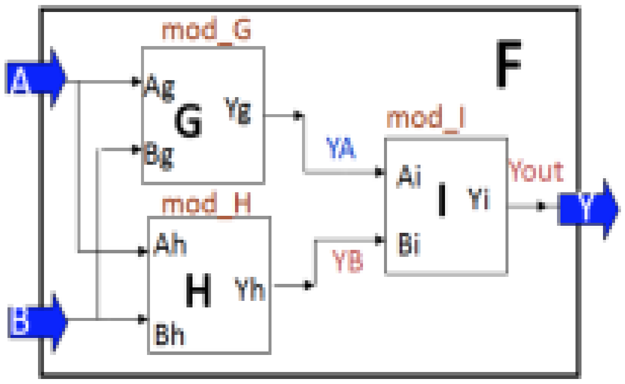
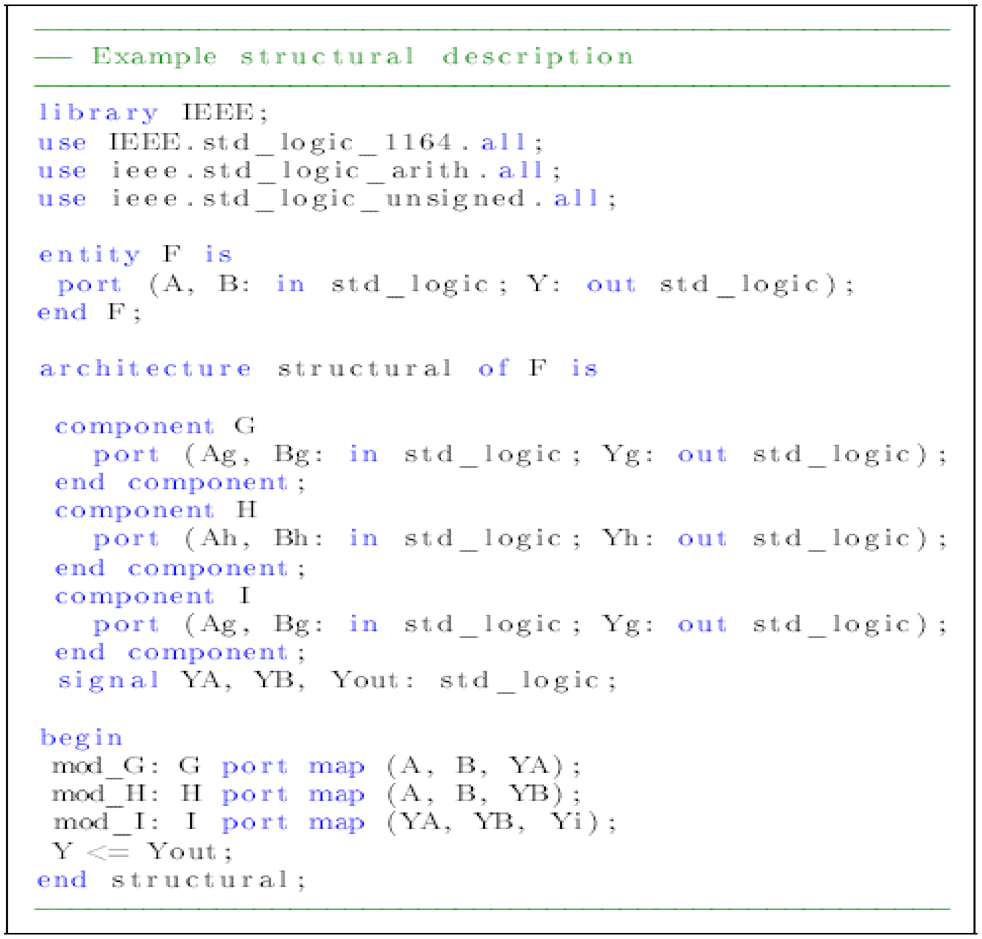


Fig 3: Example of a structural description of an entity

The code below is an example of a structural description of the circuit depicted in Figure 3. Note that, in order to make the interconnections needed between the output of a component and the input of another one, intermediate signals are needed.



```
----------------------------------------------------------------
-- Example structural description
----------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity F is
 port (A, B: in std_logic; Y: out std_logic);
end F;

architecture structural of F is

 component G
   port (Ag, Bg: in std_logic; Yg: out std_logic);
 end component;
 component H
   port (Ah, Bh: in std_logic; Yh: out std_logic);
 end component;
 component I
   port (Ag, Bg: in std_logic; Yg: out std_logic);
 end component;
 signal YA, YB, Yout: std_logic;

begin
 mod_G: G port map (A, B, YA);
 mod_H: H port map (A, B, YB);
 mod_I: I port map (YA, YB, Yi);
 Y <= Yout;
end structural;
----------------------------------------------------------------
```

In this example, note that the Y intermediate signals are needed, whereas no intermediate signal is needed in order to connect the inputs of the entity F (A and B) and the inputs of components mod_G and mod_H.

Structural descriptions of circuits can also be made by means of generate statements. These statements are used to automatically create an array of instances of the same component and/or other concurrent statements. The syntax of the generate statement is as follows:

```
for index in range generate
 -- range can be 0 to N or N downto 0; N being a constant
 -- concurrent statements
 -- component instances
end generate;
```

The two following examples show how generate instances are instantiated and used in a VHDL code:

```
-----------------------------------------------
-- Example GENERATE 1
-----------------------------------------------
signal a, b: std_logic_vector(0 to 7)
...
gen1: for i in 0 to 7 generate
   a(i) <= not b(i);
end generate gen1;
-----------------------------------------------
```

```
----------------------------------------------------------
-- Example GENERATE 2
----------------------------------------------------------
component subcircuit
 port( x: in std_logic; y: out std_logic);
end component comp
...
signal a, b: std_logic_vector(0 to 7);
...
gen2: for i in 0 to 7 generate
 u: subcircuit port map(a(i), b(i));
end generate gen2;
----------------------------------------------------------
```

Component instantiations in generate statements can also include conditions, as long as they are referred to the index of the for in the generate statement. The following code shows an example of this:

```
----------------------------------------------------------
-- Example GENERATE 3
----------------------------------------------------------
signal a, b: std_logic_vector(0 to 7)
...
loop_1: for i in 0 to 7 generate
 condition: if i > 0 generate
   a(i) <= b(i-1);
 end generate condition;
end generate loop_1;
----------------------------------------------------------
```

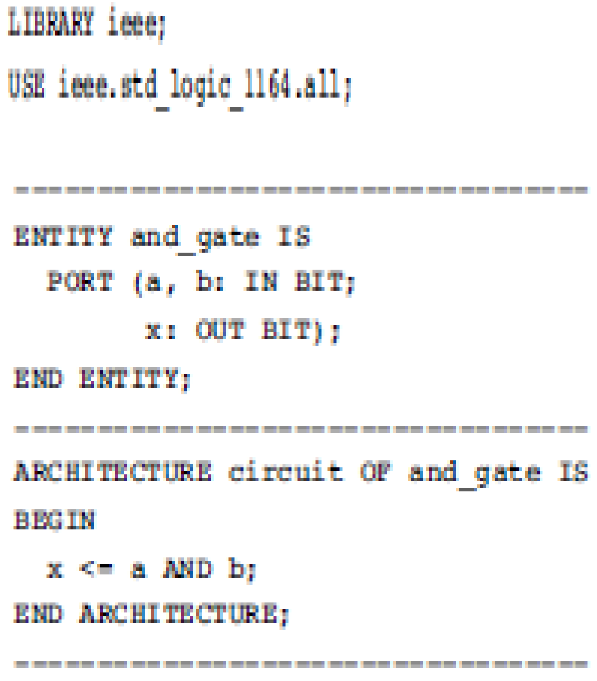
This example assigns the values of the vector b to the vector a, by left-shifting them 1 position. However, it does not assign any value to a (0), since in that case, the condition in the generate statement is not met. However, the following code (which is NOT correct), the generated hardware depends on the value of b(i), which is not known at design time. Since the actual value of the b vector depends on the execution of the circuit at any point of time, it is not possible to generate any hardware with this generate statement.

```
------------------------------------------
-- Example GENERATE 4
------------------------------------------
signal a, b: std_logic_vector(0 to 7)
...
loop_1: for i in 0 to 7 generate
 condition: if b(i) = '0' generate
   a(i) <= b(i-1);
 end generate condition;
end generate bucle;
------------------------------------------
```

<u>Example 1, Simple Gate—Concurrent Assignment</u>

a) Creating a VHDL file that *describes* an And Gate.

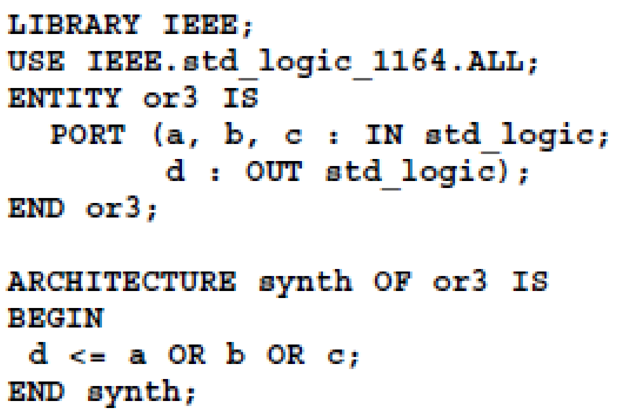
As a refresher, a simple And Gate has two inputs and one output. The output is equal to 1 only when both of the inputs are equal to 1.



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

-----------------------------------
ENTITY and_gate IS
  PORT (a, b: IN BIT;
        x: OUT BIT);
END ENTITY;
-----------------------------------
ARCHITECTURE circuit OF and_gate IS
BEGIN
  x <= a AND b;
END ARCHITECTURE;
-----------------------------------
```

b) A simple description for a 3-input OR gate



```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY or3 IS
  PORT (a, b, c : IN std_logic;
        d : OUT std_logic);
END or3;

ARCHITECTURE synth OF or3 IS
BEGIN
 d <= a OR b OR c;
END synth;
```

c) A simple description for a 3-input NAND gate

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

-- Entity
Entity my_nand3 is
        port ( A, B, C : in std_logic;
                       F : out std_logic);
End my_nand3;

-- Architecture
Architecture exa_nand3 of my_nand3 is
Begin
   F <= NOT (A AND B AND C);
 End exa_nand3;

-- Another architecture

Architecture exb_nand3 of my_nand3 is
Begin
      F <= A NAND B NAND C;

  End exb_nand3;
```

===

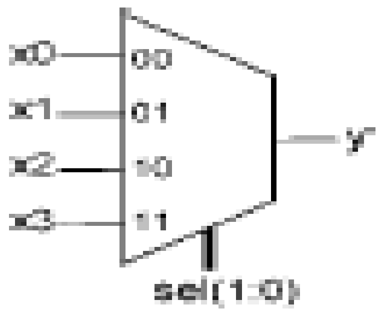
Examples 2: Multiplexer

a) Implemented with Operators

Implement the 4 *x* 1 (four inputs of one bit each) multiplexer using only logical operators.

Solution This multiplexer's logical equation (Pedroni 2008) is $y = sel_1' \cdot sel_0' \cdot x_0 + sel_1' \cdot sel_0 \cdot x_1 + sel_1 \cdot sel_0' \cdot x_2 + sel_1 \cdot sel_0 \cdot x_3$, which employs only AND, OR, and NOT operators. Its implementation is in lines 13–16 of the code below.

```
 1 -------------------------------------------
 2 LIBRARY ieee;
 3 USE ieee.std_logic_1164.all;
 4 ---------------------------------------------
 5 ENTITY mux IS
 6    PORT (x0, x1, x2, x3: IN STD_LOGIC;
 7          sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
 8          y: OUT STD_LOGIC);
 9 END mux;
10 ---------------------------------------------
11 ARCHITECTURE operators_only OF mux IS
12 BEGIN
13    y <= (NOT sel(1) AND NOT sel(0) AND x0) OR
14         (NOT sel(1) AND sel(0) AND x1) OR
15         (sel(1) AND NOT sel(0) AND x2) OR
16         (sel(1) AND sel(0) AND x3);
17 END operators_only;
19 ---------------------------------------------
```



<u>b) Implemented with WHEN and SELECT</u>

Implement the same multiplexer of example 2, but now with N-bit inputs instead of single bit, as shown in figure 4 . Specify N using GENERIC. A VHDL code (with two architectures) for this circuit is presented below, under the title mux (line 5). N is entered as a generic parameter (line 6), which is used in lines 7 and 9 to establish the size of the input-output buses. Only STD_LOGIC_VECTOR ports (industry standard) are employed in the code. In the first architecture (called with_WHEN), the WHEN statement is employed, while in the second architecture (called with_SELECT), the SELECT statement is used instead. Note that in both cases all possible input values are covered.

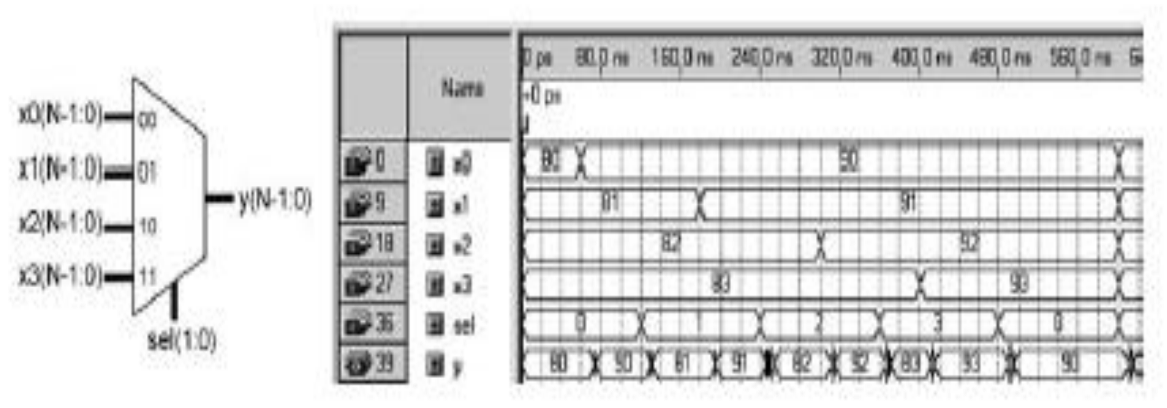
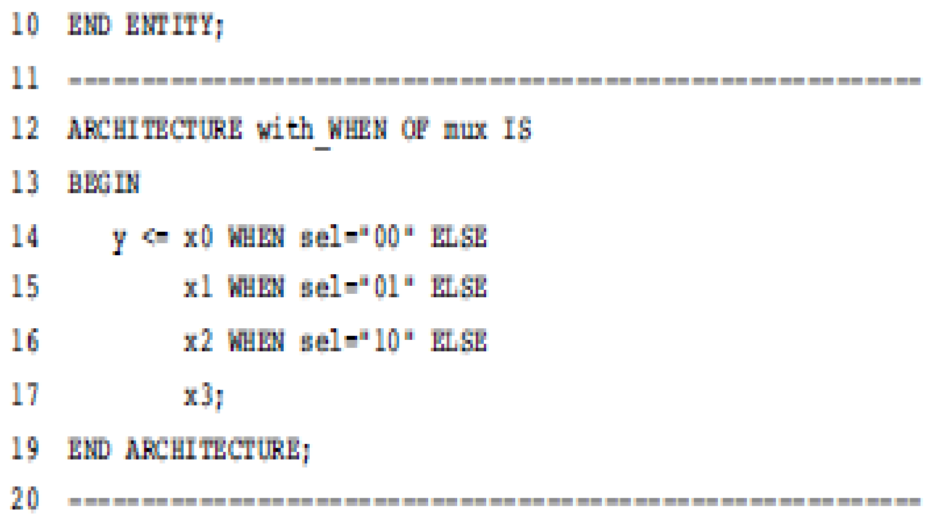
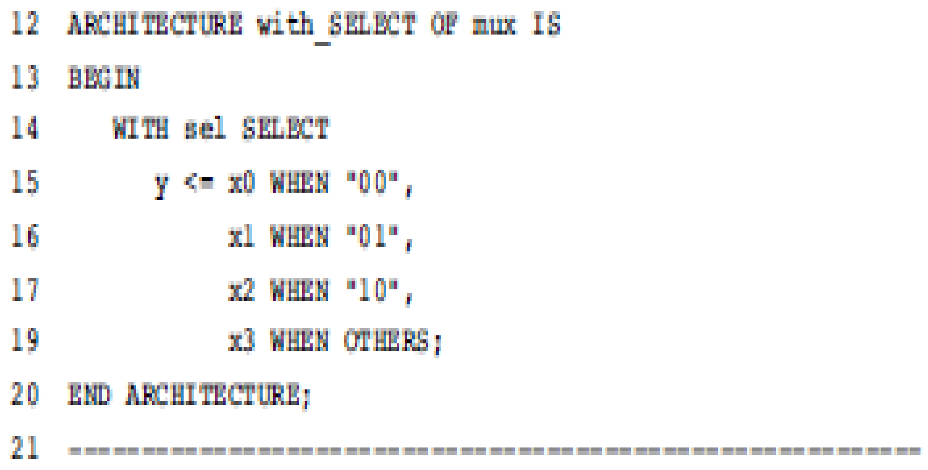


Figure 4, 4 _ N multiplexer of example and respective simulation results.

```
1  ---------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------------
5  ENTITY mux IS
6     GENERIC (N: INTEGER := 8);
7     PORT (x0, x1, x2, x3: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8           sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
9           y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
```



```
END ENTITY;
-----------------------------------------------------------
ARCHITECTURE with_WHEN OF mux IS
BEGIN
   y <= x0 WHEN sel="00" ELSE
        x1 WHEN sel="01" ELSE
        x2 WHEN sel="10" ELSE
        x3;
END ARCHITECTURE;
-----------------------------------------------------------
```



```
ARCHITECTURE with_SELECT OF mux IS
BEGIN
   WITH sel SELECT
      y <= x0 WHEN "00",
           x1 WHEN "01",
           x2 WHEN "10",
           x3 WHEN OTHERS;
END ARCHITECTURE;
-----------------------------------------------------------
```

Simulation results (for N = 8), confirming the correct circuit operation, are included in figure 4. Recall that the apparent glitches in the waveform for y are expected because multiple signals are considered at once, whose actual values neither change instantaneously nor change all exactly at the same time.

==

Example 3: Compare-Add Circuit

On the left of figure 5, a two-block circuit is shown. 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 com- pare a to b, producing a '1' when a > b or '0' otherwise. The lower part must add a and b, producing sum.

A VHDL code for this circuit is shown below. Note that dashed lines (lines 1, 4, 10, 16) were used to better organize the code (separating it into the three fundamental sections mentioned earlier). A library declaration appears in lines 2–3. The entity, named comp_add, is in lines 5–9. Finally, the architecture, called circuit, appears in lines 11–15.

```
 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  -----------------------------------------
```

Note that the entity contains all I/O ports. The inputs are a and b (mode IN, line 6), both of type INTEGER and ranging from 0 to 7 (3-bit unsigned values). The outputs are comp (line 7) and sum (line 8), the former of type STD_LOGIC (single bit), the latter of type INTEGER, ranging from 0 to 15 (4-bit unsigned value).
The architecture contains only two statements, with the first (line 13) making the comparison (by means of the WHEN statement), while the second (line 14) computes the sum (by means of the "+" operator). In this example, there are no declarations in the architecture's declarative part.

Simulation results are included in figure 6. Note that any input signal is preceded by an arrow with an "I" written inside, while each output shows an arrow with an "O" inside. A fixed value (5) was assigned to a, while b varies over the whole 3-bit range (0 to 7). The results are comp = '1' when a > b and sum = a + b (without overflow). Observe that it is a timing simulation because internal propagation delays were taken into consideration.

Note the glitch that occurs on comp when b changes from 3 to 4. It is because in this transition all bits of b change ("011" → "100"), so since the bits do not all change at exactly the same time, and moreover the actual transitions are not instantaneous (it is rather like a ramp instead of a vertical step), for a brief moment b ≥ a might occur, so this type of glitch is absolutely normal.

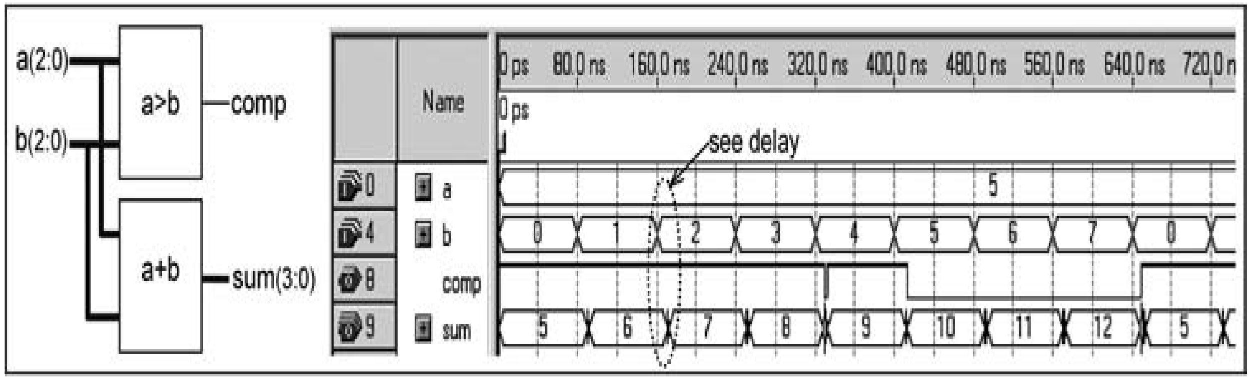


Figure 5, Circuit of example 3 and respective simulation results.

==============================

Example 4: D-type Flip-Flop (DFF)

Figure 6 shows a DFF (Pedroni 2008), which is one of the most fundamental storage circuits (there are thousands of them in FPGAs). Its inputs are d (data), clk (clock), and rst (reset), while q (stored data) is its output. In this case, the DFF is triggered at the positive (upward) clock transition, but the opposite is also possible. The output copies the input (q <= d) at the moment when clk changes from '0' to '1', remaining so until a new upward clock edge happens. Reset is asynchronous (that is, it does not depend on clk), so the output is immediately zeroed if rst = '1' occurs.

There are several ways of implementing a DFF, one being the solution presented below. 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, as shown below.

```
 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  -----------------------------------------
```

Comments about the code above follow.

Lines 2–3: First part (library declarations) of the code. Recall that this type of declaration consists of a library name followed by a library use clause. Because the data type STD_LOGIC is employed in this design, the package std_logic_1164 must be included. The other two indispensable libraries (std and work) are made visible by default. 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. Simulation results from this code are included in figure 6 (note that it is again a timing simulation). The reader is invited to check them to confirm the DFF functionality. Arrows were included in the clock waveform to highlight the (only) points where the circuit is ''transparent'' (that is, when the output copies the input). Observe also that rst is indeed asynchronous.

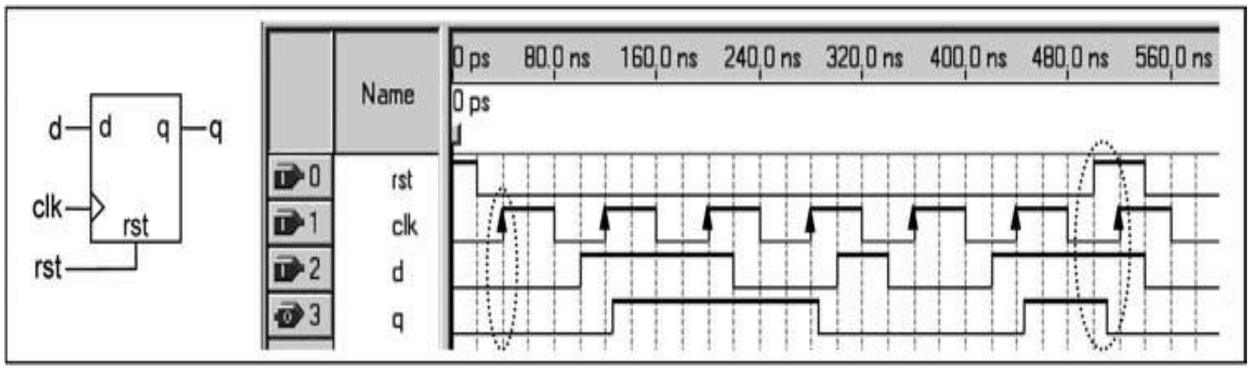


Figure 6, Circuit of example 4 and respective simulation results.

Example 5: Registered Comp-Add Circuit

Figure 7 shows a circuit that combines those seen in the previous two examples; that is, DFFs are added at the outputs of the comp_add circuit in order to ‘‘register’’ (store) comp and sum (then called reg_comp and reg_sum).

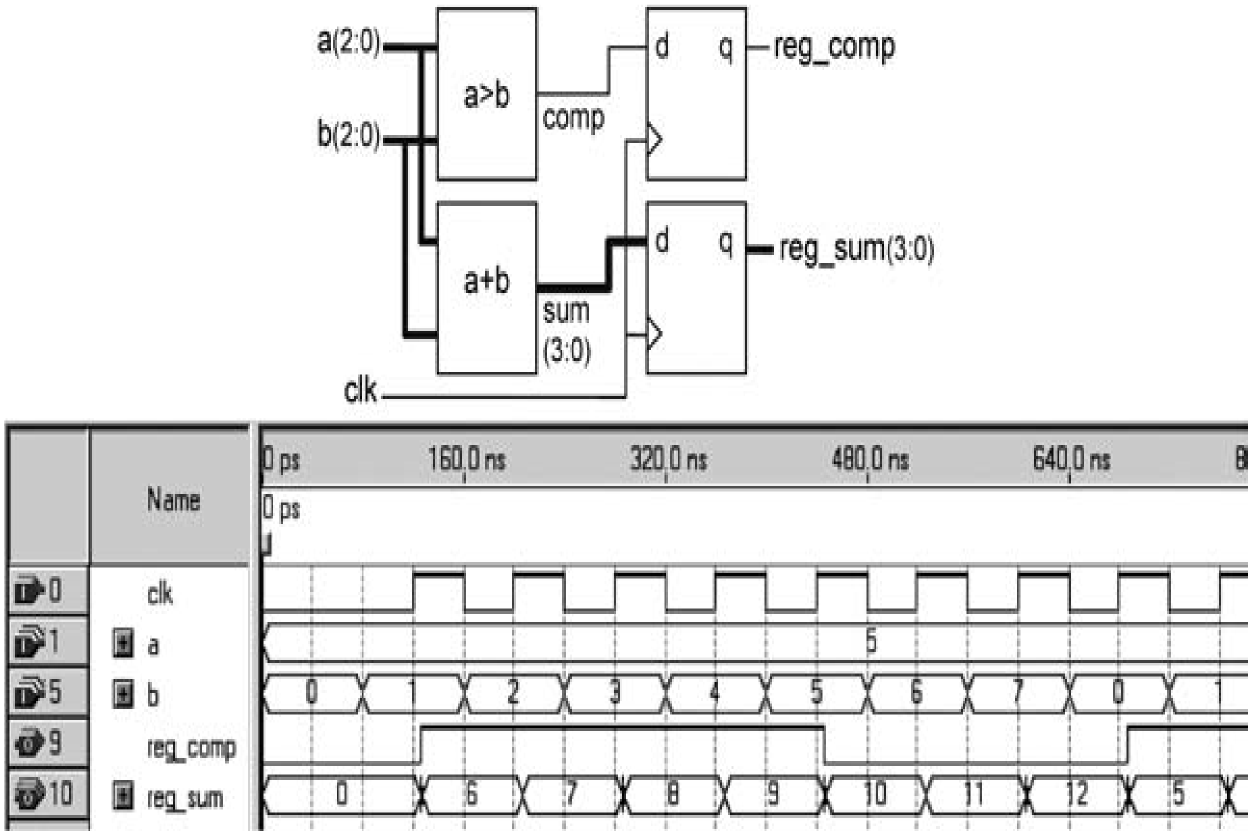


Figure 7, Circuit of example 5 and respective simulation results.

A VHDL code for this circuit is presented below. The original signals (comp, sum) are computed in the initial part of the architecture (lines 16–17). A process (lines 18–24) then follows, needed for flip-flop inference (sequential circuit). Note that a total of five DFFs are needed. Observe also that because comp and sum are now internal signals, they were specified in the declarative part of the architecture (lines 13–14).

```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY registered_comp_add IS
   PORT (clk: IN STD_LOGIC;
         a, b: IN INTEGER RANGE 0 TO 7;
         reg_comp: OUT STD_LOGIC;
         reg_sum: OUT INTEGER RANGE 0 TO 15);
END ENTITY;
---------------------------------------------
ARCHITECTURE circuit OF registered_comp_add IS
   SIGNAL comp: STD_LOGIC;
   SIGNAL sum: INTEGER RANGE 0 TO 15;
BEGIN
   comp <= '1' WHEN a>b ELSE '0';
   sum <= a + b;
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         reg_comp <= comp;
         reg_sum <= sum;
      END IF;
   END PROCESS;
END ARCHITECTURE;
---------------------------------------------
```

Simulation results are included in figure 7. Observe that now, contrary to example 3, the outputs are only updated when positive clock edges occur.

==

Example 6: Generic Address Decoder

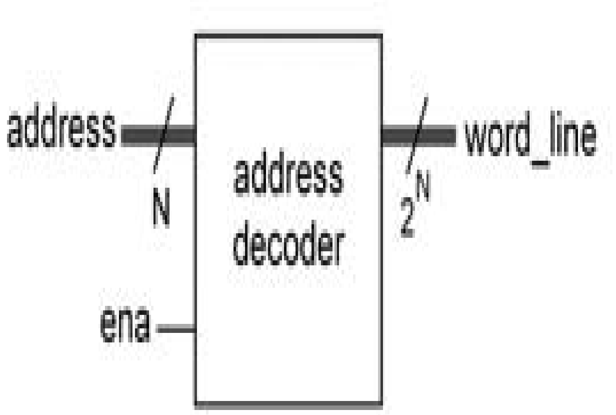
A top-level diagram for a generic N-bit address decoder is depicted in figure 8. The circuit has two inputs, called address (N bits) and ena (enable, one bit), and one output, called word_line (2^N bits). As shown in the truth table (for N = 2), the output

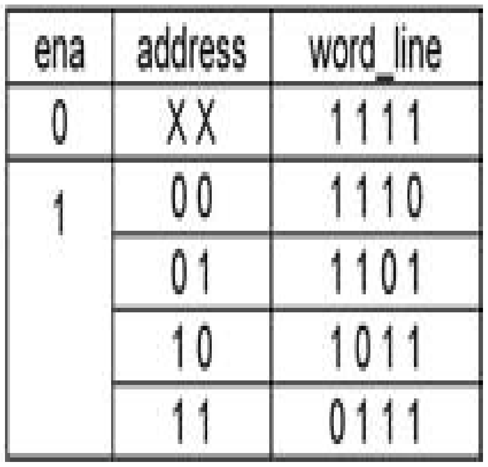
has only one bit dissimilar from all the others, located in the position determined by the input value. Note that when ena = '0' all output bits must be high. Below is a VHDL code for this circuit. Library declarations are not needed because only data types from the package standard (visible by default) are employed in this example.
The ENTITY is in lines 2–7, containing GENERIC and PORT declarations. N is entered as a generic parameter (line 3), so the code can be easily adapted to any address decoder size. The input and output signals (PORT declarations, lines 4–6) are from figure 8. The ARCHITECTURE is in lines 9–16. It is totally generic because no changes are required when the size (N) of the circuit is modified (the only change needed is in line 3). The GENERATE statement is used to create a loop, which causes all output bits to be '1' when ena = '0', or produces just one bit equal to '0' (whose position coincides with the value represented by address) when ena = '1'.

```
 1 ---------------------------------------------------
 2 ENTITY address_decoder IS
 3    GENERIC (N: NATURAL := 3);
 4    PORT (address: IN NATURAL RANGE 0 TO 2**N-1;
 5          ena: BIT;
 6          word_line: OUT BIT_VECTOR(2**N-1 DOWNTO 0));
 7 END address_decoder;
 8 ---------------------------------------------------

 9 ARCHITECTURE address_decoder OF address_decoder IS
10 BEGIN
11    gen: FOR i IN address'RANGE GENERATE
12       word_line(i) <= '1' WHEN ena='0' ELSE
13                       '0' WHEN i=address ELSE
14                       '1';
15    END GENERATE;
16 END address_decoder;
17 ---------------------------------------------------
```

Simulation results, for N = 3, are displayed in figure 9. As can be seen, all outputs are high when ena = '0'. After ena is asserted, one output bit is turned low, in the position defined by address.





ena	address	word_line
0	X X	1 1 1 1
1	0 0	1 1 1 0
	0 1	1 1 0 1
	1 0	1 0 1 1
	1 1	0 1 1 1

Figure 8, Address decoder of example 6.

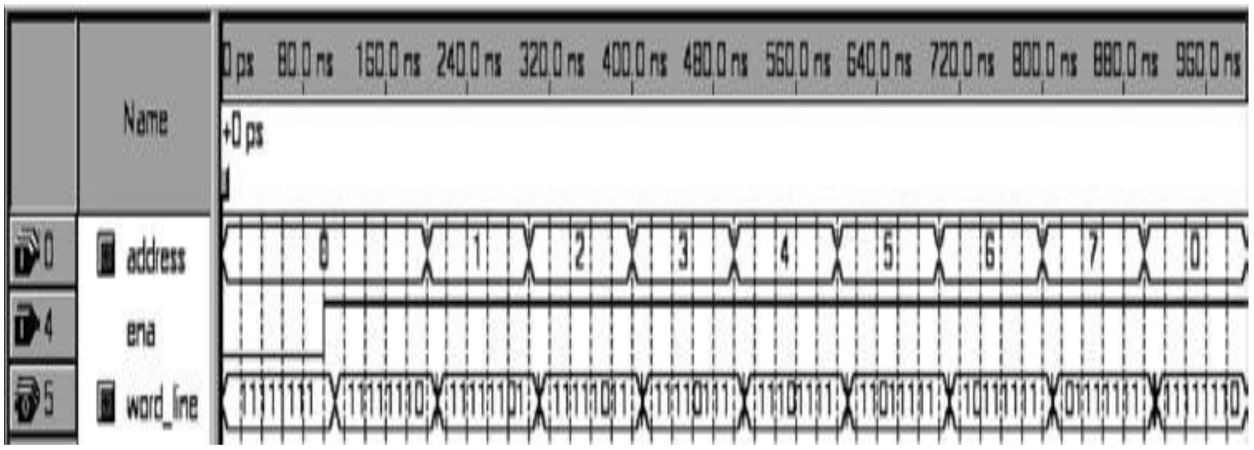
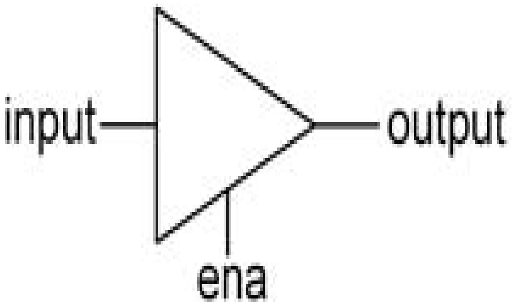


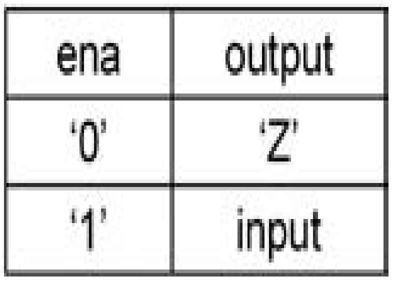
Figure 9, Simulation results from the address decoder of example 6.

====================================

Example 7: Tri-state Buffer

As already mentioned, a fundamental synthesizable value is 'Z' (high impedance), which is needed to create tri-state buffers, like that depicted in figure 10 (see its truth table). A corresponding VHDL code is shown below, with 'Z' employed in the WHEN statement of line 12. While the enable port is asserted (ena = '1'), the input is copied to the output. However, if ena = '0', the buffer is physically disconnected from the output node (high-impedance state). Note that the package std_logic_1164 (lines 2–3) is needed because it is in that package that 'Z' is defined (BIT would not do because it can only be '0' or '1').





ena	output
'0'	'Z'
'1'	input

Figure 10, Tri-state bu¤er of example 7.

```
 1  ---------------------------------------------
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.all;
 4  ---------------------------------------------
 5  ENTITY tri_state IS
 6  PORT (input, ena: IN STD_LOGIC;
 7        output: OUT STD_LOGIC);
 8  END ENTITY;
 9  ---------------------------------------------
10  ARCHITECTURE tri_state OF tri_state IS
11  BEGIN
12     output <= input WHEN ena='1' ELSE 'Z';
13  END ARCHITECTURE;
14  ---------------------------------------------
```

==============================

Example 8: Basic Counter

Figure 11 shows, on the left, a diagram for a regular binary 0-to-9 counter. A VHDL code for this counter is presented below, with input clk (line 3) and output count (line 4). A PROCESS (lines 9–19), with the IF statement playing the central role, is used to construct the circuit. In it, a VARIABLE, called temp (line 10), is employed, whose value is eventually passed to the actual output, count (line 18). Because a variable is updated immediately, the comparison in line 14 must be against 10 instead of 9 (state 10 actually never occurs because the variable never reaches line 18 with that value), so the actual range of the counter is from 0 to 9. Simulation results are included in figure 12.

```
1  ------------------------------------------
2  ENTITY counter IS
3     PORT (clk: IN BIT;
4           count: OUT INTEGER RANGE 0 TO 9);
5  END ENTITY;
6  ------------------------------------------
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9     PROCESS(clk)
10       VARIABLE temp: INTEGER RANGE 0 TO 10;
11    BEGIN
12       IF (clk'EVENT AND clk='1') THEN
13          temp := temp + 1;
14          IF (temp=10) THEN
15             temp := 0;
16          END IF;
17       END IF;
18       count <= temp;
19    END PROCESS;
20 END ARCHITECTURE;
21 ------------------------------------------
```

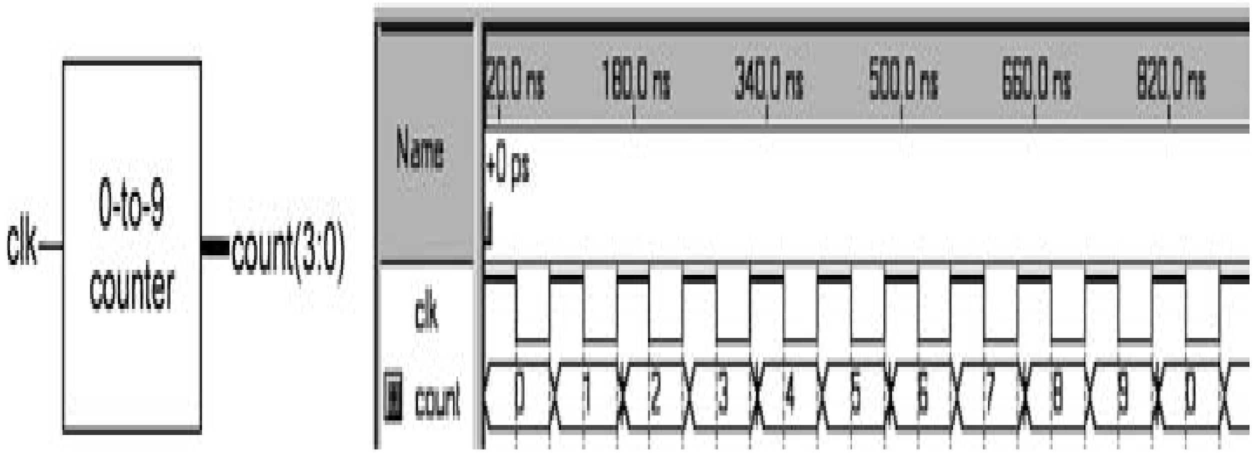


Figure 11, Counter (with simulation results) of example 8.

===

Example 9: Shift Register

Figure 12 shows a shift register, which consists of a string of serially connected DFFs. The input is din and the output is either q_0 to q_3 or just dout, depending on the application. For example, the former can be used to convert data from serial to parallel form, while the latter can be used to implement a delay line. The number of stages should be generic.

A VHDL code for this circuit is presented below. The number of stages (N) is generic (line 6). The data input and output ports are din (line 7) and dout (line 8), respectively. A process is used to implement the circuit (lines 13–22), with clk and rst in the sensitivity list (line 13). Note that the shift register is obtained by simply shifting the whole vector q one position to the right at every positive clock transition, with the rightmost value discarded and the leftmost position taken by din. Simulation results (for N = 4) are included in figure 12. As can be seen, the whole data vector does move one position to the right at every rising edge of the clock.

```vhdl
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY shift_register IS
   GENERIC (N: INTEGER := 4); --number of stages
   PORT (din, clk, rst: IN STD_LOGIC;
         dout: OUT STD_LOGIC);
END ENTITY;
-----------------------------------------------
ARCHITECTURE shift_register OF shift_register IS
BEGIN
   PROCESS (clk, rst)
      VARIABLE q: STD_LOGIC_VECTOR(0 TO N-1);
   BEGIN
      IF (rst='1') THEN
         q := (OTHERS => '0');
      ELSIF (clk'EVENT AND clk='1') THEN
         q := din & q(0 TO N-2);
      END IF;
      dout <= q(N-1);
   END PROCESS;
END ARCHITECTURE;
-----------------------------------------------
```

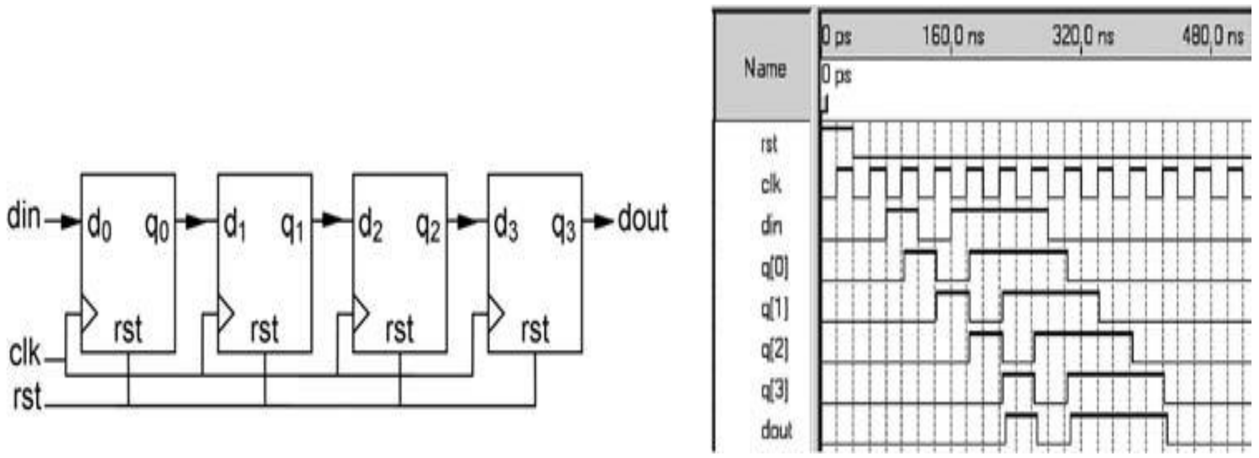


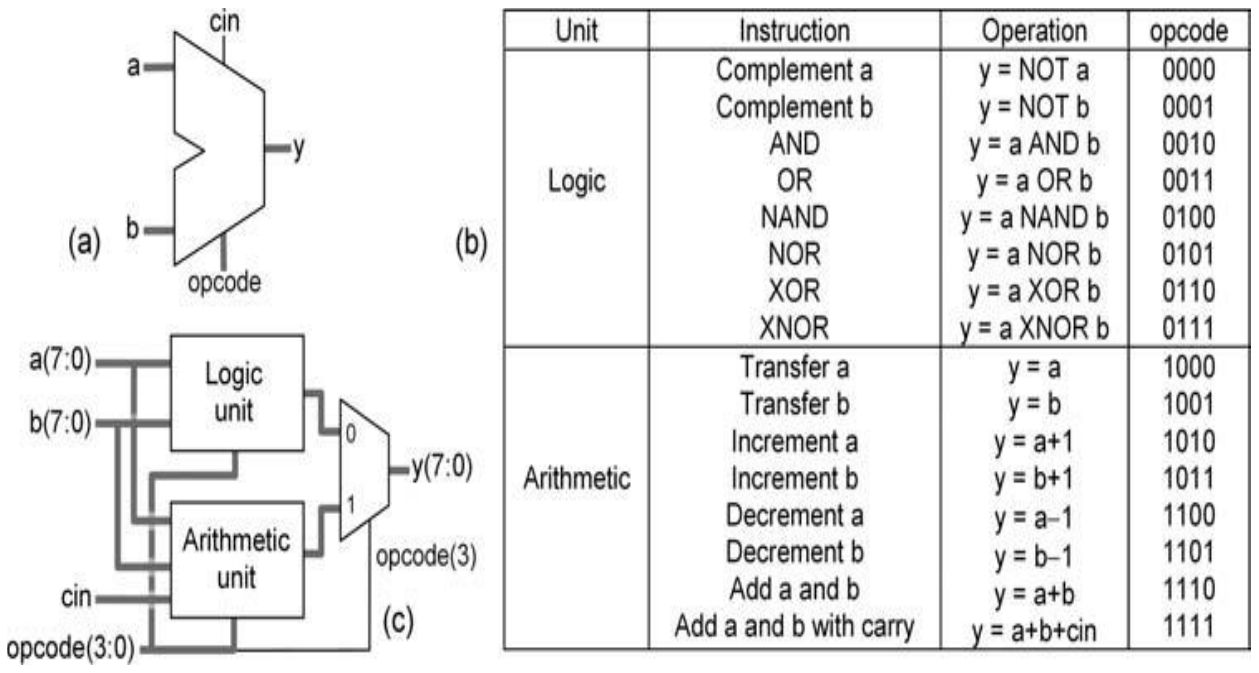
Figure 12, Shift register (with simulation results) of example 9.

Example 10: ALU

An ALU (Arithmetic Logic Unit) is shown in figure 13(a), having a, b, cin (carry in), and opcode (operation code) as inputs, and y as output. The desired functionality is expressed in the truth table of figure 13(b), where each function is selected by a different value of opcode. Note that the upper eight instructions are logical, while the lower eight are arithmetic. Design this circuit using the concurrent statement SELECT, satisfying the following conditions:

1) The arithmetic operations must be signed.
2) The number of bits for inputs a and b must be generic.
3) All ports must be of type STD_LOGIC(_VECTOR) (industry standard).
4) Simulation results must also be included in the solution.

Solution Figure 13(c) shows a possible ALU implementation (among several other options). The circuit contains two main sections, called logic and arithmetic units, each controlled by the same three LSBs of opcode. The MSB of opcode is employed to control a multiplexer, letting the logic result out when low or the arithmetic result out if high. A VHDL code for this circuit is presented below, under the title alu (line 6). The number of bits in a and b is a generic parameter (line 7), and all ports (lines 8–11) are of type STD_LOGIC (_VECTOR). Because the arithmetic operations were asked to be signed, the package numeric_std (line 4) was included in the library/package declarations. The code proper is divided according to figure 13(c)—that is, a logic unit (lines 22–30), an arithmetic unit (lines 32–43), and a multiplexer (lines 45–47).



Unit	Instruction	Operation	opcode
Logic	Complement a	y = NOT a	0000
	Complement b	y = NOT b	0001
	AND	y = a AND b	0010
	OR	y = a OR b	0011
	NAND	y = a NAND b	0100
	NOR	y = a NOR b	0101
	XOR	y = a XOR b	0110
	XNOR	y = a XNOR b	0111
Arithmetic	Transfer a	y = a	1000
	Transfer b	y = b	1001
	Increment a	y = a+1	1010
	Increment b	y = b+1	1011
	Decrement a	y = a–1	1100
	Decrement b	y = b–1	1101
	Add a and b	y = a+b	1110
	Add a and b with carry	y = a+b+cin	1111

Figure 13, ALU of example 10. (a) ALU symbol; (b) truth table; (c) a possible implementation.

Example 10, also uses the concurrent statement SELECT.

The implementation of the logic unit is straightforward. However, because the arithmetic unit must be signed, that is, the inputs are explicitly converted from STD_LOGIC_VECTOR to SIGNED (by type casting, in lines 32–33), they are then processed, and finally the result is converted back to STD_LOGIC_VECTOR (at the mux input, line 47, again by type casting). Note also that because cin is STD_LOGIC, not SIGNED, NATURAL, or INTEGER, it could not participate directly in the sum of line 43 (observe in the package numeric_std, that the overloaded operator "+" does not contain the SIGNED + STD_LOGIC option), so a small integer (lines 19 and 34) was created to allow the sum.

```
---------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
---------------------------------------------------
ENTITY alu IS
   GENERIC (N: INTEGER := 8); --word bits
   PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         cin: IN STD_LOGIC;
         opcode: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END ENTITY;
---------------------------------------------------
ARCHITECTURE alu OF alu IS
   SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
   SIGNAL y_sig: SIGNED(N-1 DOWNTO 0);
   SIGNAL y_unsig: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
   SIGNAL small_int: INTEGER RANGE 0 TO 1;
BEGIN
   ------Logic unit:-------------
   WITH opcode(2 DOWNTO 0) SELECT
      y_unsig <= NOT a WHEN "000",
                 NOT b WHEN "001",
                 a AND b WHEN "010",
                 a OR b WHEN "011",
                 a NAND b WHEN "100",
                 a NOR b WHEN "101",
                 a XOR b WHEN "110",
                 a XNOR b WHEN OTHERS;
   ------Arithmetic unit:---------
   a_sig <= SIGNED(a);
   b_sig <= SIGNED(b);
   small_int <= 1 WHEN cin='1' ELSE 0;
   WITH opcode(2 DOWNTO 0) SELECT
      y_sig <= a_sig WHEN "000",
               b_sig WHEN "001",
               a_sig + 1 WHEN "010",
               b_sig + 1 WHEN "011",
               a_sig - 1 WHEN "100",
               b_sig - 1 WHEN "101",
               a_sig + b_sig WHEN "110",
               a_sig + b_sig + small_int WHEN OTHERS;
   ------Mux:---------------------
   WITH opcode(3) SELECT
      y <= y_unsig WHEN '0',
           STD_LOGIC_VECTOR(y_sig) WHEN OTHERS;
END ARCHITECTURE;
-----------------------------------------------------
```

Simulation results are depicted in figure 14. The upper graph is for logic instructions, while the lower graph exhibits results from arithmetic operations. The reader is invited to examine both to check the correct circuit operation.

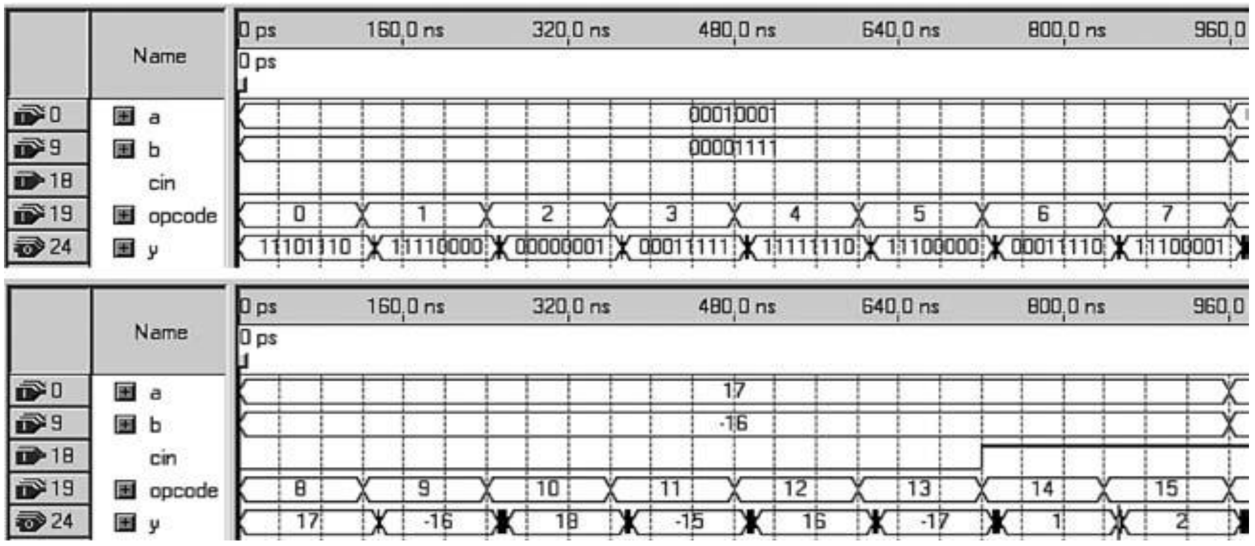


Figure 14, Simulation results from the ALU of example 10 (logic instruction in the upper graph, arithmetic instructions in the lower graph).

VHDL Design of Memory Circuits

Figure 15 shows the most basic ROM (read-only memory) and RAM (random access memory) configurations. The ROM has as input the signal address (M bits) and as output the signal data_out (N bits), with the latter exhibiting the contents in the specified memory address. Contrary to a ROM, the contents of a RAM can be modified freely, so the RAM has two additional inputs, called data_in (data to be stored in the memory) and we (write enable); the latter, when asserted, causes data_in to be stored in the specified address. This RAM arrangement is called read-on-write, because during the writing process the output reads the value that is being stored in the memory. Both circuits in figure 15, have depth = 2M (number of words) and width = N (number of bits in each word). Several variations of the architectures above exist. For example, in high-performance systems, memories are normally synchronous to allow the system clock to control their operation. This means that the memory inputs and/or outputs are registered (that is, the address and/or the data buses are stored by flip-flops).

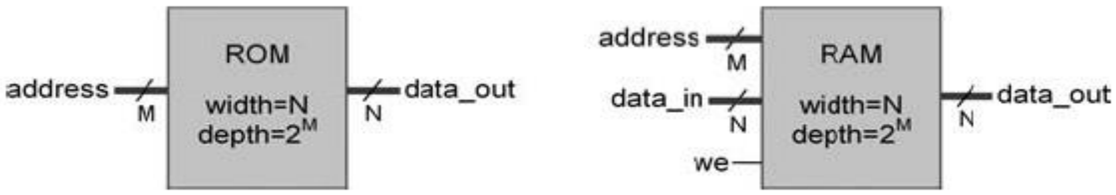


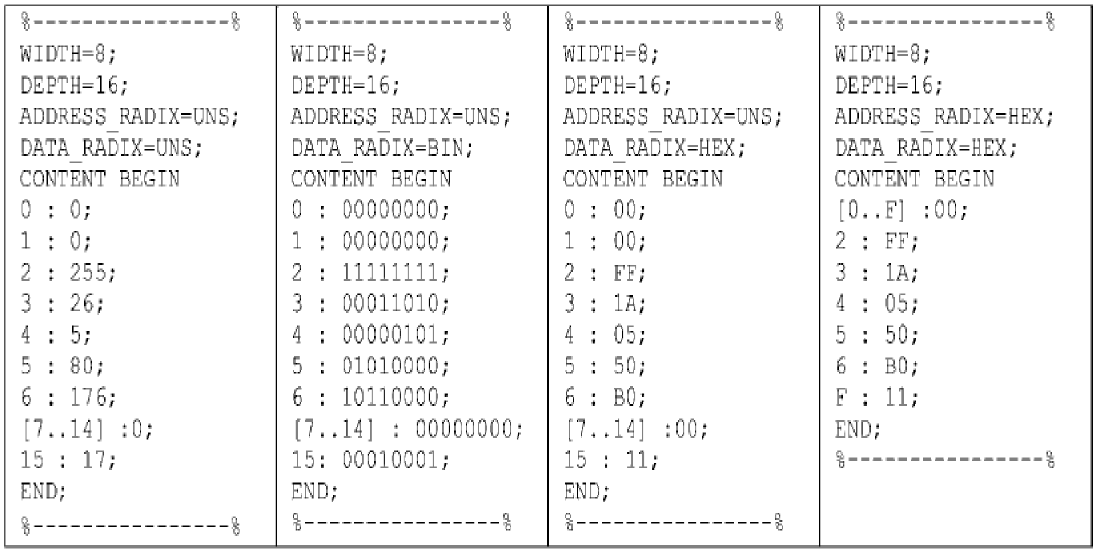
Figure 15, Basic asynchronous ROM and RAM configurations.

Memory Initialization Files

Another important topic to be examined before we start discussing memory implementations is memory initialization files.

- MIF (memory initialization file), from Altera.
- COE (derived from CGF—core generator file), from Xilinx.
- HEX (Intel hexadecimal file), with general support.

Another standard file format is RIF (RAM initialization file), used in other vendors' EDA software.



```
%----------------%
WIDTH=8;
DEPTH=16;
ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;
CONTENT BEGIN
0 : 0;
1 : 0;
2 : 255;
3 : 26;
4 : 5;
5 : 80;
6 : 176;
[7..14] :0;
15 : 17;
END;
%----------------%
```

```
%----------------%
WIDTH=8;
DEPTH=16;
ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;
CONTENT BEGIN
0 : 00000000;
1 : 00000000;
2 : 11111111;
3 : 00011010;
4 : 00000101;
5 : 01010000;
6 : 10110000;
[7..14] : 00000000;
15: 00010001;
END;
%----------------%
```

```
%----------------%
WIDTH=8;
DEPTH=16;
ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;
CONTENT BEGIN
0 : 00;
1 : 00;
2 : FF;
3 : 1A;
4 : 05;
5 : 50;
6 : B0;
[7..14] :00;
15 : 11;
END;
%----------------%
```

```
%----------------%
WIDTH=8;
DEPTH=16;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;
CONTENT BEGIN
[0..F] :00;
2 : FF;
3 : 1A;
4 : 05;
5 : 50;
6 : B0;
F : 11;
END;
%----------------%
```

Figure 16, four equivalent representations for a MIF file.

<u>MIF File</u>

MIF is a file format that can be used to initialize ROM, RAM, and CAM contents in Altera devices. Four equivalent examples are presented in figure 13.3. The file starts with a declaration regarding the width (number of bits) of the stored words, followed by the memory depth (number of words). The address radix can be binary (BIN), octal (OCT), hexadecimal (HEX), or unsigned decimal (UNS), while the memory radix can also be signed decimal (DEC). The default for both is HEX. "%" is used for comments. Note that in the last case of figure 16, the entire memory was initialized to 00h, then some of the values were overwritten.

ROM Design

Figure 17 shows, synchronous ROM architectures, with registers (D-type flip-flops (DFFs)) installed at the input and also, optionally, at the output.

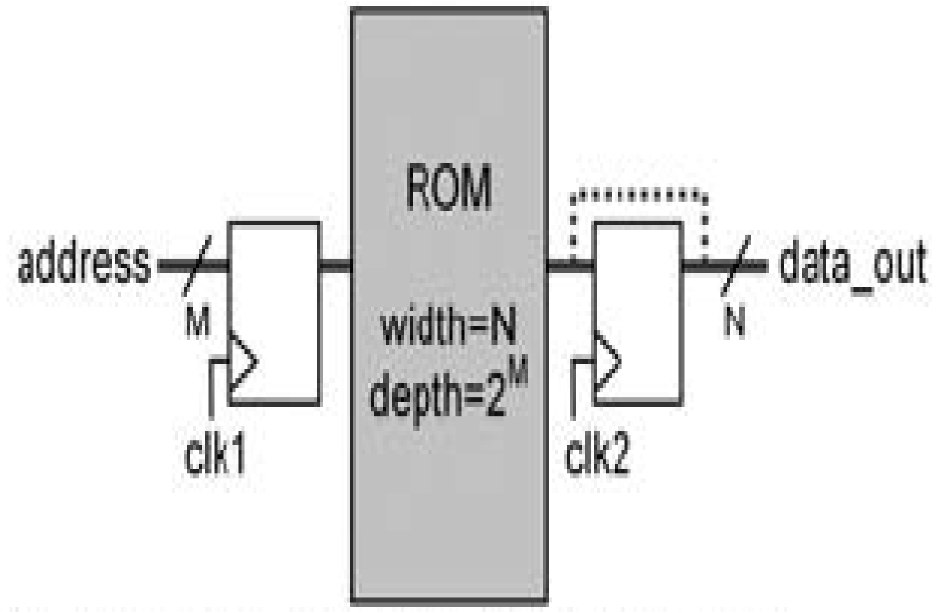


Figure 17, synchronous, single port, registered input, optionally registered output ith the following options:

- Single clock: clk1= clk2
- Dual clock: clk1 ≠ clk2

Straight VHDL code is used. There is no concern regarding particular encoding styles to help the compiler understand that memory is wanted, as there are no synthesis attributes or special functions to force the compiler to adopt, for example, on-chip SRAM memory blocks (if available). This means that the compiler might use them or not (in general, regular logic cells will be used) and memory will be created independently from the existence or not of such blocks. An example is shown below whose contents are those of figure 16, entered using CONSTANT (lines 14–21). Note that a process (lines 24–29) was used to register the address, without the output register. Simulation results are depicted in figure 18 (compare the memory contents against those in figure 16).

```
-----------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------------------------
ENTITY rom IS
   PORT (clk: IN STD_LOGIC;
         address: IN INTEGER RANGE 0 TO 15;
         data_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END rom;
-----------------------------------------------------------------
ARCHITECTURE rom OF rom IS
   SIGNAL reg_address: INTEGER RANGE 0 TO 15;
   TYPE memory IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
   CONSTANT myrom: memory := (
      2 => "11111111", --255
      3 => "00011010", --26
      4 => "00000101", --5
      5 => "01010000", --80
      6 => "10110000", --176
      15=> "00010001", --17
      OTHERS => "00000000");
BEGIN
   --Register the address:----------
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         reg_address <= address;
      END IF;
   END PROCESS;
   --Get unregistered output:-------
   data_out <= myrom(reg_address);
END rom;
-----------------------------------------------------------------
```

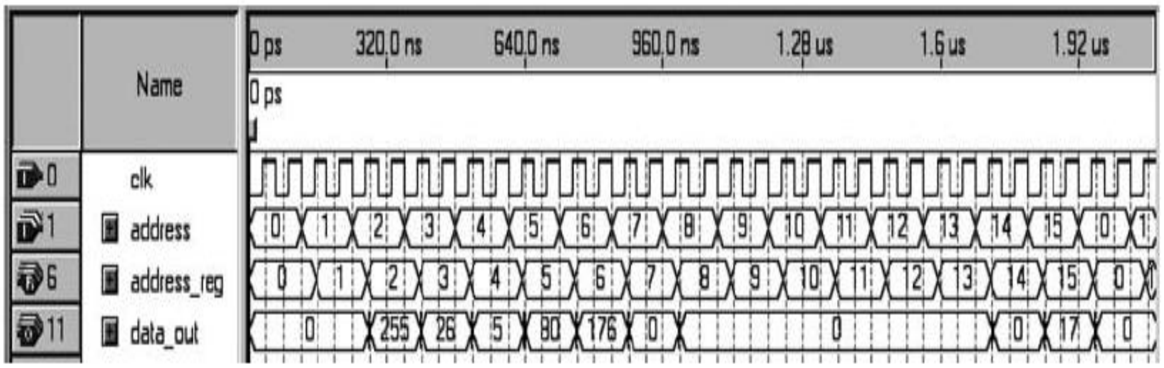


Figure 18, Simulation results from the ROM implemented with regular VHDL code

RAM Design

To better understand the construction of RAMs, a RAM model based on DFF blocks is presented in figure 19. The memory-write address (wr_address) is processed by the address decoder, which produces only one output high, corresponding to the present address value. If write-enable (we) is high, then the corresponding DFF block is enabled to store data_in at the next positive clock transition. The output is controlled by a multiplexer, which allows only the DFF block selected by the memory-read address (rd_address) to be connected to data_out.

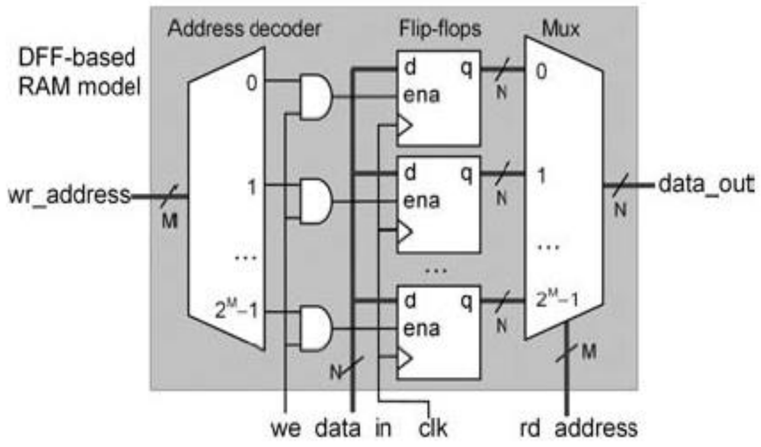


Figure 19, Flip-flop-based RAM model.

RAM Implemented with Regular VHDL Code

Straight VHDL code is used here, with no concerns regarding particular encoding styles or synthesis attributes to help the compiler understand that memory is wanted. This means that the compiler might use on-chip SRAM blocks (if available) or might not (in general, regular logic cells are used).
An example is shown below. Because regular code is used, a regular circuit (equivalent to that in figure 19) is expected, hence with 8 *x* 16 = 128 flip-flops. After compiling this code, the reader is invited to check that fact in the compilation reports. Recall also that a RAM does not need to be initialized, but if one wants to do so, the same procedure seen for ROMs implemented with an initialization file can be used, which employs the ram_init_ file attribute (this attribute is used in the code below to load a file called ram_contents.mif, containing again the data of figure 16).

```
----------------------------------------------------------------
ENTITY ram IS
   PORT (clk: IN STD_LOGIC;
         we: IN STD_LOGIC;
         address: IN INTEGER RANGE 0 TO 15;
         data_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         data_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END ram;
------------------------------------------------------------------
ARCHITECTURE ram OF ram IS
   TYPE memory IS ARRAY (0 to 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL myram: memory;
   ATTRIBUTE ram_init_file: STRING;
   ATTRIBUTE ram_init_file OF myram: SIGNAL IS "ram_contents.mif";
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         IF (we='1') THEN
            myram(address) <= data_in;
         END IF;
      END IF;
   END PROCESS;
   data_out <= myram(address);
END ram;
------------------------------------------------------------------
```

CPU Design

The example is a small, 16-bit microprocessor. A block diagram is shown in Figure 20. The processor contains a number of basic pieces. There is a register array of eight 16-bit registers, an ALU (Arithmetic Logic Unit), a shifter, a program counter, an instruction register, a comparator, an address register, and a control unit. All of these units communicate through a common, 16-bit tristate data bus.

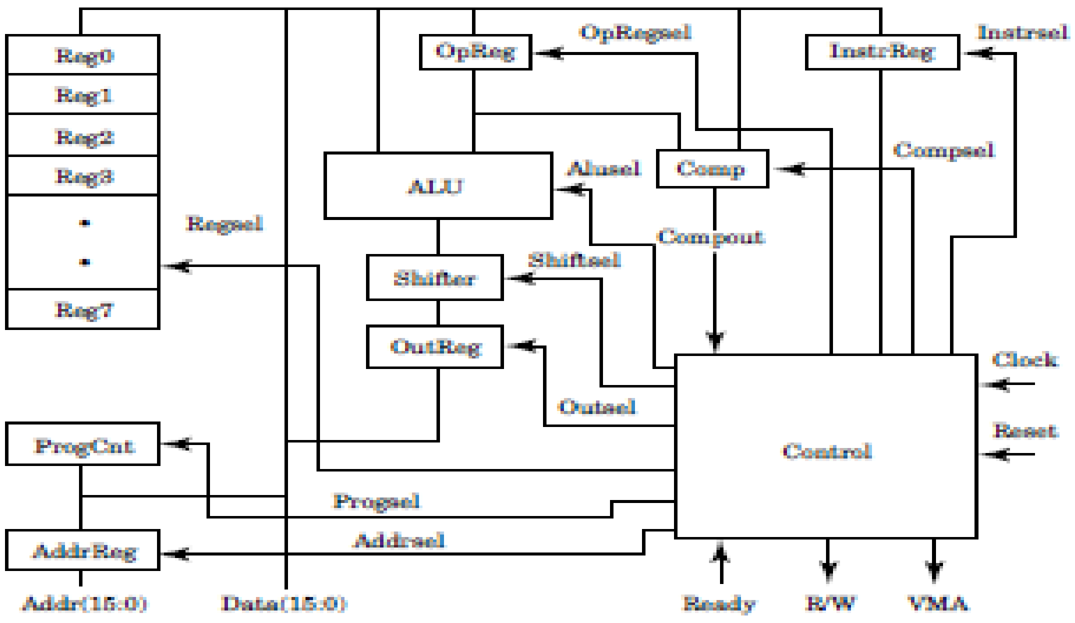


Fig 20, CPU Block Diagram.

CPU controller Design

```
library ieee;
use ieee.std_logic_1164.all;
use work.cpu_defs.all;
entity controller is
  port(clock, reset: in std_logic;
       op: in opcode;
       z_flag: in std_ulogic;  -- status signal
       ACC_bus, load_ACC, PC_bus, load_PC,
       load_IR, load_MAR, MDR_bus, load_MDR,
       ALU_ACC, ALU_add, ALU_sub, INC_PC,
       Addr_bus, CS, R_NW: out std_ulogic);
end entity controller;
```

```
architecture rtl of controller is
  type state is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9);
  signal present_state, next_state: state;
begin
  seq: process(clock, reset) is
  begin
    if reset = '1' then
      present_state <= s0;
    elsif rising_edge(clock) then
      preset_state <= next_state;
    end if;
  end process seq;
```

```
 comm: process(present_state, op, z_flag) is
 begin
   -- reset all the control signals to a default
   ACC_bus <= '0';
   load_ACC <= '0';
   PC_bus <= '0';
   load_PC <= '0';
   load_IR <= '0';
   load_MAR <= '0';
   MDR_bus <= '0';
   load_MDR <= '0';
   ALU_ACC <= '0';
   ALU_add <= '0';
   ALU_sub <= '0';
   INC_PC <= '0';
   Addr_bus <= '0';
   CS <= '0';
   R_NW <= '0';
 ...
 end process com;
```

```
comm: process(present_state, op, z_flag) is
begin
  ...
  case present_state is
    when s0 =>    -- current inst address to be loaded in
                  -- MAR, PC incremented, etc.
      PC_bus <= '1';
      load_MAR <= '1';
      INC_PC <= '1';
      load_PC <= '1';
      next_state <= s1;
    ...
  end case;
end process com;
```

```
comm: process(present_state, op, z_flag) is
begin
  ...
  case present_state is
   ...
    when s1 =>      -- memory is read, MDR contains the inst.
      CS <= '1';
      R_NW <= '1';
      next_state <= s2;
    when s2 =>      -- inst. transferred from MDR to IR
      MDR_bus <= '1';
      load_IR <= '1';
      next_state <= s3;
    when s3 =>  -- direct address to be loaded in MAR
      Addr_bus <= '1';
      load_MAR <= '1';
      if op = store then
        next_state <= s4;
      else
        next_state <= s6;
      end if;
    when s4 =>          -- store (memory to be written)
      ACC_bus <= '1';
      load_MDR <= '1';
      next_state <= s5;
    when s5 =>       -- select memory to write it with MDR
      CS <= '1';
      next_state <= s0;

    when s6 =>
      CS <= '1'; R_NW <= '1'; -- read the memory just in case
      if op = load then next_state <= s7;
      else
        if op = bne then
          if z_flag = '1' then next_state <= s9; -- taken
          else next_state <= s0; -- not taken
          end if;
        else next_state <= s8;  -- not a branch
        end if;
      end if;
      when s7 =>          -- load instruction
        MDR_bus <= '1'; -- ACC gets content of MDR
        load_ACC <= '1';
        next_state <= s0;
     when s8 =>        -- ALU operation
       MDR_bus <= '1'; -- one operand from MDR
       ALU_ACC <= '1'; -- the other from ACC
       if op = add then
         ALU_add <= '1';
       elsif op = sub then
         ALU_sub <= '1';
       end if;
       next_state <= s0;
      when s9 =>    -- completing the branch instruction
        MDR_bus <= '1';
        load_PC <= '1';
        next_state <= s0;
    end case;
  end process com;
end architecture rtl;
```