

Chapter 2

Synthesizable VHDL for FPGA-Based Devices

Abstract This chapter presents a concise introduction to synthesizable VHDL that is sufficient for the design methods and examples given in subsequent chapters to be understood without much background knowledge. The main objective of this chapter is to explain the basis of VHDL modules and their specification capabilities without going into detail. There are many excellent books dedicated to VHDL that may be used to complement this book. Our primary target is the synthesis and optimization of FPGA-based circuits and systems and VHDL is just an instrument that is used in the book to describe the desired functionalities and structures. Thus this chapter only provides the minimum necessary to allow subsequent chapters to be read without additional material, and to enable all the proposed examples to be understood and tested with the FPGA-based prototyping boards.

2.1 Introduction to VHDL

VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL) was created as a result of USA government sponsored program in 1980s [1]. The language has been standardized in 1987 (with revisions done in 1993, 2002, and 2008) and is widely adopted by designers.

The target of this section is to provide a brief introduction to VHDL through simple examples. The main objective is to describe such constructions that will be used for FPGA projects in the book. VHDL is a complex language with wide-ranging specifications not all of which are synthesizable. Subsequent sections of this chapter will present just a basis for using VHDL in FPGA design. For deeper study of the language the books [1, 2] are recommended.

A specification of a digital circuit in VHDL includes two major parts: an *entity declaration* which is a definition of the circuit interface (where all the external circuit connections are declared), and an *architecture body* where a description of internal functionality is given. There are three types of architecture: *structural*, *behavioral*, and *mixed*.

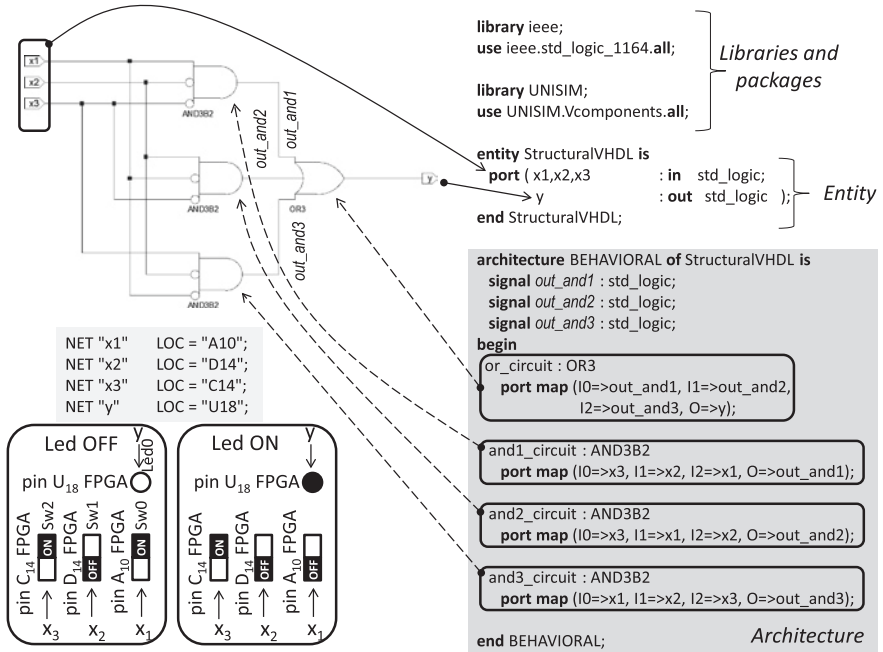


Fig. 2.1 Structural VHDL for the circuit shown in Fig. 1.2

Structural architecture provides all necessary internal connections between the circuit components that are either library primitives or previously developed circuits. Figure 2.1 demonstrates a structural VHDL description of the circuit firstly shown as a schematic entry in Fig. 1.2.

The first two lines of VHDL code identify a standard library, IEEE, and a package, `std_logic_1164`, which contains important definitions needed for our specification. In particular, we would like to use the type `std_logic` and the associated operations defined in that package. The type `std_logic` includes 9 values ('U'—uninitialized, 'X'—unknown, '0'—0, '1'—1, 'Z'—high impedance, 'W'—weak unknown, 'L'—weak 0, 'H'—weak 1, '-'—don't care) that allow signals to be modeled with strong, weak and high-impedance strengths. For now, from these 9 values we need just two: '0' and '1' (logic values are enclosed in quotation marks to distinguish them from the numbers 0 and 1). VHDL is not case sensitive language. That is why we can use the name `STD_LOGIC` instead of `std_logic`.

The second two lines of VHDL code identify a library, `UNISIM.vcomponents`, (with the package `vcomponents`) which contains the component declarations for the Xilinx primitives and defines models needed for simulation.

As you can see from Fig. 2.1 there are three sections in VHDL code:

1. Specification of libraries and packages that are intended to be used.
2. Specification of interface (entity).
3. Specification of architecture.

The components OR3 and AND3B2 are Xilinx library primitives and they correspond to the relevant schematic symbols in Fig. 1.2. The declared internal signals `out_and1`, `out_and2`, and `out_and3` are needed to describe internal connections between the library primitives (there are totally 3 instances `and1_circuit`, `and2_circuit` and `and3_circuit` of the primitive AND3B2 and one instance `or_circuit` of the primitive OR3). Connections are shown by comma delimited lines in parenthesis after the **port map** keywords, for example, **port map** (`l0=>x3`, `l1=>x2`, `l2=>x1`, `O=>out_and1`). The component AND3B2 is defined in the UNISIM library (the file *unisim_VCOMP.vhd*) as follows:

```
component AND3B3
  port (O      : out   std_ulogic;  -- std_ulogic is unresolved type [1] similar to std_logic
        l0, l1, l2 : in   std_ulogic);
end component;
```

The VHDL keyword **signal** permits signals to be declared in the declarative part of an **architecture** (between the head of the **architecture** and the keyword **begin**). Signals in VHDL are similar to wires in hardware circuits.

Keywords (reserved words) here and later in the book are shown in **bold** font. In VHDL two successive hyphens (–) denote a single-line comment and they are shown in such font. Each port is given a name (e.g. O, l0, l1, l2) and is either an input (**in**) or an output (**out**). Other types (namely **inout** and **buffer**) are also allowed and they are described in Appendix A. For every port we specify the associated type which states the range of values that can be used on that port. In the example above each port is of type `std_ulogic`. Please note that the specification of each port is followed by a semicolon except for the last port. A signal of the type `std_ulogic` is similar to `std_logic` but it does not contain predefined resolution functions (the details can be found in [1, 3]). The names O, l0, l1, l2 of the interface signals in the component declaration above appear in the mapping line: **port map** (`l0=>x3`, `l1=>x2`, `l2=>x1`, `O=>out_and1`). The latter involves a named association where each component port l0, l1, l2, O (see the component AND3B3 above) is associated with x3, x2, x1 and `out_and1` signals from the entity where the component is used (see the StructuralVHDL entity in Fig. 2.1). Internal signals (used for connections just within the entity StructuralVHDL) are explicitly declared as (Fig. 2.1):

```
signal out_and1 : std_logic;  -- signal and component declarations appear in the declarative
signal out_and2 : std_logic;  -- part of architecture which is between the keywords
signal out_and3 : std_logic;  -- architecture...of...is and begin (see example in Fig. 2.1)
```

Besides of the named association a positional association can be used, which will be considered in another example of structural specification below and is also described in Appendix A (see *Aggregate*).

Behavioral architecture represents the desired *functionality* of a circuit in an abstract way similar to general-purpose programming languages. However, VHDL statements differ in many aspects mainly because of inherent to hardware

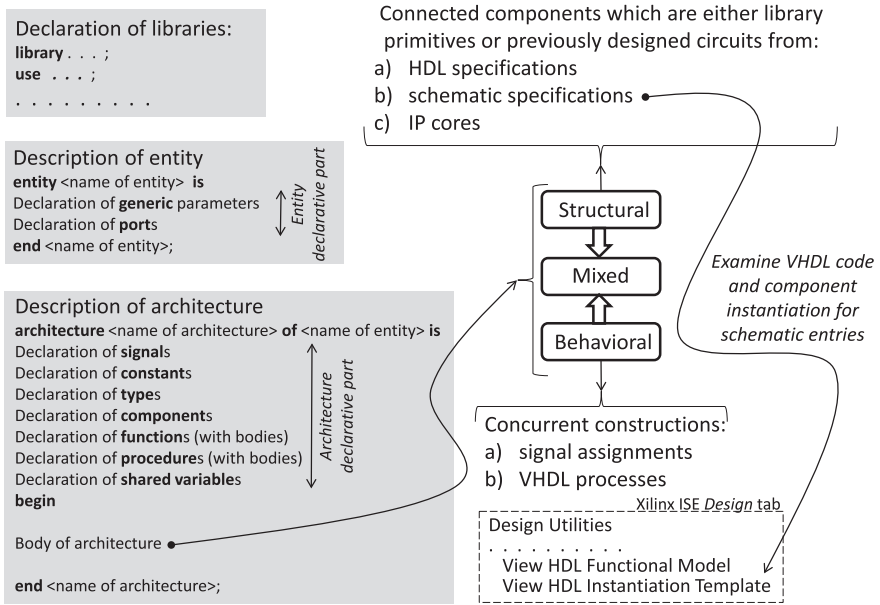


Fig. 2.2 A simplified structure of elements for a VHDL module

description languages concurrency and advanced operations manipulating individual bits and sets of bits.

For the considered above structural architecture an equivalent behavioral specification can be done as follows:

```
library ieee;                -- note that the UNISIM library is not needed now
use ieee.std_logic_1164.all;

entity BehavioralVHDL is      -- the entity name (such as BehavioralVHDL) is chosen by the designer
  port (x1, x2, x3 : in  std_logic;
        y          : out std_logic);
end BehavioralVHDL;

architecture behavioral of BehavioralVHDL is
begin -- and/not/or are VHDL logical operators for AND/NOT/OR logical operations
y <= (x1 and not x2 and not x3) or (not x1 and x2 and not x3) or
    (not x1 and not x2 and x3);    -- <= is VHDL signal assignment operator
end behavioral;
```

Functionality of the synthesized circuit is exactly the same. *Structural* and *behavioral* specifications complement each other and may have different effectiveness for different projects. Thus, it is reasonable to combine them within a *mixed architecture*, which is composed of both *behavioral* and *structural* specifications. For complex projects such *mixed architecture* can often be seen as the most frequently used.

Figure 2.2 gives a simplified structure of elements for a VHDL module (design entry in VHDL) which nevertheless is sufficient for an introductory level.

Up to now we have not described yet many keywords shown in Fig. 2.2:

- **generic** enables compact scalable and parameterizable designs to be described (see Sect. 2.5 for details and Appendix A);
- **constant** permits objects with unchangeable values to be declared (see Sect. 2.2 for details and Appendix A);
- **type** is used to declare new types including arrays and enumerations (see Sect. 2.2 for details and Appendix A);
- **function** and **procedure** (subprograms) allow pieces of code to be used multiple times in a design (see Sect. 2.4 for details and Appendix A);
- shared variable is an extension of **variable**, allowing inter-process communication. Note that variable cannot be declared directly in architecture and it is declared in a process or in a subprogram (function or procedure). Variable is assigned using the `:=` operator.
- **process** is a concurrent statement with such behavior that is described by sequential statements (see also Sect. 2.3 and Appendix A).

Subsequent sections of this chapter will present details about indicated above and other VHDL keywords (reserved words). A summary about the use of different reserved words is given in Appendix A.

A code below demonstrates a *behavioral* VHDL specification for a half-adder discussed in Sect. 1.5. The external interface and the truth table of the half-adder are shown in Fig. 2.3.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
  port (A      : in std_logic;
        B      : in std_logic;
        carry_out : out std_logic;
        sum     : out std_logic); -- there is no semicolon following the specification
end half_adder;                  -- of the last port

architecture half_adder_behavior of half_adder is
begin
  sum      <= A xor B;           -- xor is a VHDL keyword for XOR logical operation
  carry_out <= A and B;          -- and is a VHDL keyword for AND logical operation
end half_adder_behavior;
```

Each port of the half-adder is given a name (A, B, carry_out, sum). The architecture is entitled `half_adder_behavior` and is associated with the `half_adder` entity. These names can be chosen arbitrary but have to respect VHDL syntax rules, i.e. a user identifier can only include alphanumerical symbols and the underline character `_` must start with a letter, may not include two consecutive underline characters, and may not have an underline character at the end.

The next example presents the complete *mixed* VHDL specification of a full adder composed of two structural components (half-adders) and a behavioral description of a two-input OR gate: `carry_out <= s2 or s3`; (see also Fig. 1.19b).

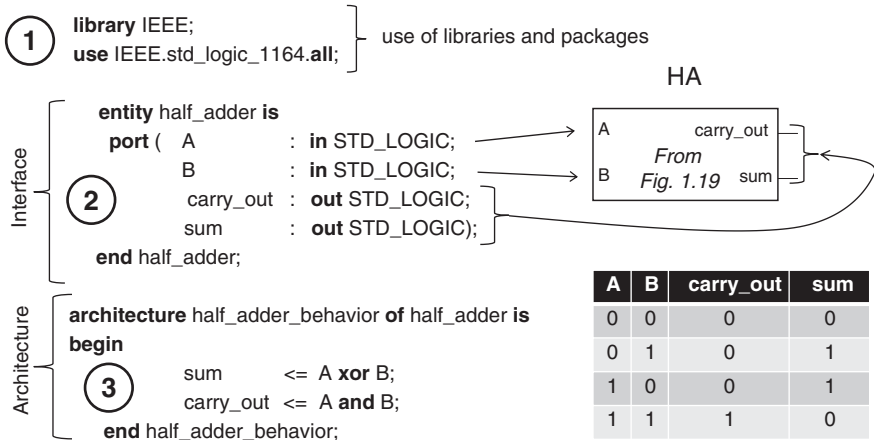


Fig. 2.3 Specification in VHDL and the truth table of a half-adder

```

library IEEE;
use IEEE.std_logic_1164.all;

entity FULLADD is
port ( A, B, carry_in : in std_logic;
      sum, carry_out  : out std_logic );
end FULLADD;

architecture STRUCT of FULLADD is
signal s1, s2, s3 : std_logic;

component half_adder
port(A,B : in std_logic;
     carry_out, sum : out std_logic);
end component;

begin
u1: half_adder port map(A, B, s2, s1);
u2: half_adder port map(s1, carry_in, s3, sum);
carry_out <= s2 or s3;
end STRUCT;

```

The component `half_adder` is described explicitly using the VHDL keyword **component**. If we comment the lines:

```

component half_adder
port( A,B : in std_logic;
     carry_out, sum : out std_logic);
end component;

```

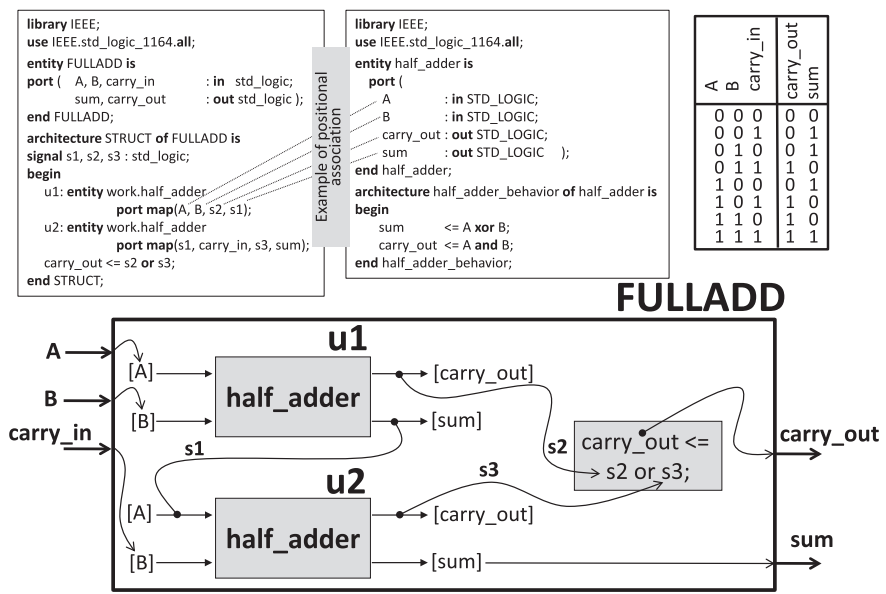


Fig. 2.4 Structural VHDL description of a full adder

the following error appears: *<half_adder> is not declared*. However, since all VHDL modules are compiled (by default) to a library with the name *work* we can use the half-adder component directly from the library as follows:

```
architecture STRUCT of FULLADD is
  signal s1, s2, s3 : std_logic;
begin
  -- getting the half_adder from the library work in the construction: entity work.half_adder
  u1: entity work.half_adder      port map(A, B, s2, s1);
  u2: entity work.half_adder      port map(s1, carry_in, s3, sum);
  carry_out <= s2 or s3;
end STRUCT;
```

Now the code does not have errors and the resulting circuit works exactly the same as the circuit in Fig. 1.19b. The connections of the components are done through the respective external (A, B, carry_in, sum, carry_out) and internal (s1, s2, s3) signals that are associated with components' ports by positions (i.e. a positional association has been used). For example, the half adder has 4 ports A, B, carry_out, sum. In the component *u1* they are connected with external signals A, B and internal signals s2, s1, accordingly. In the component *u2* they are connected with s1 (internal), carry_in (external), s3 (internal), and sum (external) signals. All other details should be understandable from Fig. 2.4 (see also Appendix A).

The examples above illustrate the general organization of *structural*, *behavioral*, and *mixed* VHDL specifications. In the next sections of this chapter we

will present more details about different VHDL constructions paying the main attention to comprehensive examples that can be directly synthesized, implemented and tested in FPGA-based circuits.

There are two appendices A and B in this book. Appendix A explains informally a variety of synthesizable constructions and VHDL keywords listed alphabetically. Appendix B includes some coding examples for frequently needed modules.

To conclude this section we would like to explicitly indicate that the book is not about VHDL and only a subset of this language is used to describe functionality of the considered FPGA-targeted circuits and systems. There are some limitations assumed in the book and they are listed below:

1. Only two values ‘0’ and ‘1’ from the allowed values of `std_logic` type are used.
2. For the majority of examples unsigned vectors with element values ‘0’ and ‘1’ are used and their type is declared as `std_logic_vector`. There are just a few examples with the types signed and unsigned (see the next section and Appendices).
3. Taking into account the assumptions 1 and 2, in many examples below the type `std_logic_vector` is used in the same way as an unsigned type although the latter might be more correct, for instance, for such operations as comparison, arithmetical, and some others. This way does not give rise to any problem for the resulting (synthesized and implemented) circuits that are presented in the book and it permits the number of conversion functions to be minimized. This is done because we would like to pay the primary attention to the design methods and the described circuits but not to supplementary constructions, which often make the code more difficult to analyze and understand.
4. Many design methods described in the book are equally applicable to signed vectors and if required the necessary (minimal) changes can easily be done assuming that the given examples are firstly well understood and tested.

2.2 Data Types, Objects and Operators

We consider the following VHDL basic data types: (1) *enumerated* (including pre-defined and user-defined); (2) *bit vector*; (3) *integer*; and (4) *record*.

Pre-defined enumerated types are: (1) `bit` (with possible values ‘0’ and ‘1’); `boolean` (with possible values `false` and `true`); and (3) `std_logic` defined in the *IEEE std_logic_1164* package (with 9 possible values ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’ described in the previous section).

User-defined enumerated types are frequently introduced for naming states of finite state machines, for example:

```
type FSM_states is (begin, run, end); -- begin, run, end are user-defined names of FSM states
```

Bit vector is (1) a standard `bit_vector` type with elements of the type `bit`, and (2) defined in the *IEEE std_logic_1164* package `std_logic_vector` with elements of type

`std_logic`. `Std_logic` and `std_logic_vector` are the most frequently used types in the book. Two examples are given below:

```
signal sw      :      std_logic_vector(3 downto 0);
signal my_bit  :      bit_vector(2 to 3);
```

The first example declares a vector `sw` with 4 elements: `sw(3)`, `sw(2)`, `sw(1)`, `sw(0)`. If, for example, `sw <= "1100"` then `sw(3)` is '1', `sw(2)` = '1', `sw(1)` = '0', `sw(0)` = '0'. If for the second example `my_bit <= "01"` then `my_bit(2)` is '0', and `my_bit(3)` is '1'. Single-bit values are written in between single quotes while multi-bit values are specified with double quotes.

Integer type enables an integer to be declared. The range of the `integer` values can explicitly be defined, for example:

```
signal my_int : integer range 3 to 8; -- allowed values now are only 3, 4, 5, 6, 7, and 8
```

Record type permits a set of data with different types to be combined in a named structure, for example:

```
type user_defined_record is record -- the name of the structure is user_defined_record
  data1 : std_logic_vector(7 downto 0);    -- record fields
  data2 : integer range 0 to 7;           -- a field can also be of type record
end record;
```

Data types can form an array. Although any number of dimensions can be chosen it is frequently recommended to limit them, for example, by 3 in [3]. The following type declares an array named `my_array` of 16 integers with possible values 0, 1, 2, 3, 4:

```
type my_array is array (0 to 15) of integer range 0 to 4;
```

The following line declares a two dimensional array containing 4 sets of integers:

```
type my_table is array (3 downto 0) of my_array; -- the type my_array is declared above
```

We consider here three VHDL objects that are *signals*, *variables* and *constants*.

Signals are declared in the declarative part of architecture (shown in Fig. 2.2 between the lines **architecture...** and **begin**) with the keyword **signal** and used within that architecture.

Variables are declared in the declarative part of a process or a sub-program (function or procedure) with the keyword **variable** and used within that process or sub-program. We will discuss processes and sub-programs a bit later in this section.

Constants are declared in the declarative part of architecture, process, or sub-program (function or procedure) with the keyword **constant**. Declarative part of a process, a function or a procedure is placed between the lines **process...** / **function...** / **procedure...** and **begin**).

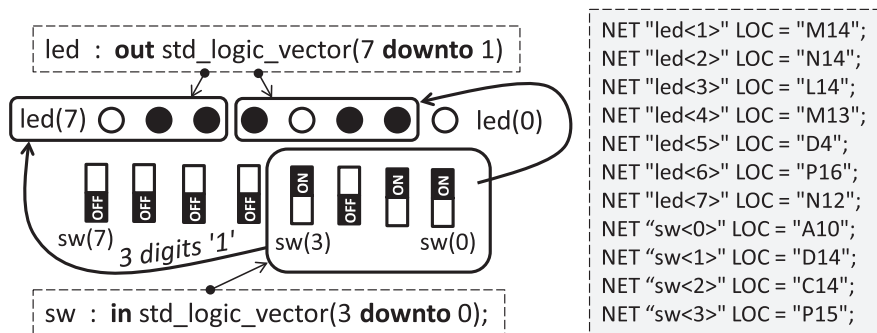


Fig. 2.5 UCF and functionality of the project with the entity types_and_objects for the Atlys board

Let us consider a complete example:

```
library IEEE; -- in future VHDL modules we will assume including these libraries
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all; -- see also appendix A and section 2.6
use IEEE.STD_LOGIC_UNSIGNED.all; -- for conversion functions

entity types_and_objects is -- sw and led are signals from switches and to LEDs
  port (sw : in std_logic_vector(3 downto 0);
        led : out std_logic_vector(7 downto 1));
end types_and_objects;

architecture Behavioral of types_and_objects is
  type my_array is array (0 to 15) of integer range 0 to 4;
  constant Hamming_weight : my_array := (0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4);
  signal index : integer range 0 to 15;
begin
  led(4 downto 1) <= sw;
  index <= conv_integer(sw(3 downto 0));
  led(7 downto 5) <= conv_std_logic_vector(Hamming_weight(index), 3);
end Behavioral;
```

Here, `conv_integer` (casting `std_logic_vector` type to integer type) and `conv_std_logic_vector` (casting integer type to `std_logic_vector` type of size `n` where `n` is the second argument) are conversion functions for which we need to include additional packages indicated in the code above. The line:

```
constant Hamming_weight : my_array := (0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4);
```

declares and initializes a constant `Hamming_weight` which is a one-dimensional array of integers. Each integer with index i_0 is a Hamming weight of i_2 , i.e. the number of values '1' in the binary vector i_2 . Indeed, if $i_0 = 5$, then $my_array(5) = 2$, and $i_2 = "0101"$ contains 2 digits '1'. The one-dimensional array is a new type `my_array` declared in the line: `type my_array is array (0 to 15) of integer range 0 to 4;`. Figure 2.5 demonstrates the user constraints file (UCF) for our project and the project functionality.

In subsequent VHDL modules we will also use the following derived data types (they are also described in Appendix A):

- **natural** that declares integers with nonnegative values (0,1,2,...);
- **positive** which is the same as **natural** without the value 0 (1,2,...);
- **unsigned** declares unsigned vectors based on **std_logic** type and is defined, for example, in the VHDL package **std_logic_arith** (see also [Sect. 2.6](#));
- **signed** declares signed vectors based on **std_logic** type and is defined, for example, in the VHDL package **std_logic_arith** (see also [Sect. 2.6](#));
- **character** is a 7-bit ASCII code;
- **string(positive)** is an array of characters.

The following two lines give declaration examples for a character and a string:

```
signal my_string : string(1 to 3); -- declaration of signal my_string of type string(1 to 3)
signal my_char  : character;      -- declaration of signal my_char of type character
```

The following lines give examples of assignments which can be done in an architecture body:

```
my_char <= '3';                -- my_char receives the ASCII code of digit 3
my_string(1) <= '5';           -- my_string(1) receives the ASCII code of digit 5
my_string(2) <= my_char;       -- my_string(2) receives the value of my_char
my_string(3) <= '9';           -- my_string(3) receives the ASCII code of digit 9
led <= std_logic_vector(conv_unsigned(character'pos(my_char), 8));
```

The last line finds position of **my_char** in ASCII table (**character'pos(my_char)**), then converts the position to an 8-element unsigned vector of **std_logic** (**conv_unsigned(<position>,8)**) and finally converts the unsigned vector to **std_logic_vector** (**std_logic_vector(<unsigned vector>)**) which is assumed to be displayed on eight onboard LEDs.

The following operators will be used in examples of this book:

1. **Arithmetical**: **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division). Often, division is supported only if the right operand is a power of 2 [[3](#)].
2. **Assignment**: **<=** (for signals) and **:=** (for variables).
3. **Concatenation**: **&**.
4. **Logical**: **and**, **nand**, **nor**, **not**, **or**, **xor**, **xnor** (see appendix A for details).
5. **Relation**: **=** (equal to), **/=** (not equal to), **<** (less than), **<=** (less than or equal to), **>** (greater than), **>=** (greater than or equal to).
6. **Shift**: **sll** (logic shift left), **srl** (logic shift right), **sla** (arithmetic shift left), **sra** (arithmetic shift right), **rol** (rotate left), **ror** (rotate right). Examples and additional explanations are given in Appendix A. We would prefer to use logically equivalent operators (see *Shift operators* in Appendix A).
7. **Others**: **abs** (absolute value), **rem** (remainder), **mod** (modulo), ****** (power if the left operand is 2). Frequently, the operations **rem** and **mod** are supported only if the right operand is a constant power of 2 [[3](#)].

Using the majority of the operators is clear. So, we will consider below just a part of them. The first VHDL module is given below:

```

entity abs_rem_mod is  -- the project was tested in the ISE 14.7 and Atlys board
port ( sw                : in std_logic_vector(7 downto 0);
      led                : out std_logic_vector(7 downto 0);
      BTNU, BTNC, BTND, BTNL, BTNR : in std_logic); -- onboard buttons in the Atlys
end abs_rem_mod;

architecture Behavioral of abs_rem_mod is
  signal result : integer range 0 to 16;
  signal but    : std_logic_vector(4 downto 0);
begin
  but <= BTNU & BTNC & BTND & BTNL & BTNR; -- concatenation of five signals
  result <= 16 when conv_integer(sw(3 downto 0)) = 0 else -- 16 indicates "divide by 0"
    conv_integer(sw(7 downto 4)) mod conv_integer(sw(3 downto 0))
    when but = "00001" else -- only BTNR is pressed
    conv_integer(sw(7 downto 4)) rem conv_integer(sw(3 downto 0))
    when but = "00010" else -- only BTNL is pressed
    conv_integer(sw(7 downto 4)) / conv_integer(sw(3 downto 0))
    when but = "00100" else -- only BTND is pressed
    abs(-10) when but = "01000" else -- abs(-10) = 10 (only BTNC is pressed)
    abs(5) when but = "10000" else 0; -- abs(5) = 5 (only BTNU is pressed)
  led <= conv_std_logic_vector(result, 8);
end Behavioral;

```

We introduced here **when ... else** conditional signal assignment which allows more operators to be described in a compact code. The conditional assignment has the following general form:

```

<name> <= <expression> when <condition> else <expression>;

```

which can be repeated any number of times. For example **mod** operator will be applied if and only if but = "00001", i.e. only one BTNR button is pressed. Indeed, the signal but is a concatenation (&) of 5 signals from the onboard buttons (BTNU & BTNC & BTND & BTNL & BTNR). Some operations are explained in comments above and some others are shown in Table 2.1. For example, using a modulo ($A \bmod B$) operator permits the result to be changed from $A = 0$ up to the value $B-1$ and then again from 0 to the value $B-1$ until the final allowed value A is reached (exact definition of the described above operators is given in [1]). As you can see from Table 2.1 division (/) and remainder (**rem**) give correct results in the Xilinx ISE 14.7 for any integer operands (the document [3] indicates that the respective operations are only supported if the second operand is a power of 2 or both operands are constants). The operator with an asterisk in Table 2.1 (**mod***, **rem***, and /) are applied to the first positive and to the second negative arguments:

```

conv_integer(sw(7 downto 4)) mod (-conv_integer(sw(3 downto 0)))
conv_integer(sw(7 downto 4)) rem (-conv_integer(sw(3 downto 0)))
conv_integer(sw(7 downto 4)) / (-conv_integer(sw(3 downto 0)))

```

Table 2.1 The results of **mod**, **rem** and division (/) operations

| A = sw(7:4) | B = sw(3:0) | Mod | mod* | rem (rem*) | / (/*) |
|---------------------------------------|--|--------------------------------------|--|--------------------------------------|--------|
| 0000 ₂ (0 ₁₀) | mod, rem, / : | (0000 ₂) 0 ₁₀ | (00000 ₂) 0 ₁₀ | (0000 ₂) 0 ₁₀ | 0 (0) |
| 0001 ₂ (1 ₁₀) | 0101 ₂ (5 ₁₀) | (0001 ₂) 1 ₁₀ | (11100 ₂) -4 ₁₀ | (0001 ₂) 1 ₁₀ | 0 (0) |
| 0010 ₂ (2 ₁₀) | | (0010 ₂) 2 ₁₀ | (11101 ₂) -3 ₁₀ | (0010 ₂) 2 ₁₀ | 0 (0) |
| 0011 ₂ (3 ₁₀) | mod*, rem*, /* | (0011 ₂) 3 ₁₀ | (11110 ₂) -2 ₁₀ | (0011 ₂) 3 ₁₀ | 0 (0) |
| 0100 ₂ (4 ₁₀) | (-5 ₁₀), | (0100 ₂) 4 ₁₀ | (11111 ₂) -1 ₁₀ | (0100 ₂) 4 ₁₀ | 0 (0) |
| 0101 ₂ (5 ₁₀) | i.e. the sign is forced to be changed | (0000 ₂) 0 ₁₀ | (00000 ₂) 0 ₁₀ | (0000 ₂) 0 ₁₀ | 1 (-1) |
| 0110 ₂ (6 ₁₀) | | (0001 ₂) 1 ₁₀ | (11100 ₂) -4 ₁₀ | (0001 ₂) 1 ₁₀ | 1 (-1) |
| 0111 ₂ (7 ₁₀) | | (0010 ₂) 2 ₁₀ | (11101 ₂) -3 ₁₀ | (0010 ₂) 2 ₁₀ | 1 (-1) |
| 1000 ₂ (8 ₁₀) | | (0011 ₂) 3 ₁₀ | (11110 ₂) -2 ₁₀ | (0011 ₂) 3 ₁₀ | 1 (-1) |
| 1001 ₂ (9 ₁₀) | | (0100 ₂) 4 ₁₀ | (11111 ₂) -1 ₁₀ | (0100 ₂) 4 ₁₀ | 1 (-1) |
| 1010 ₂ (10 ₁₀) | | (0000 ₂) 0 ₁₀ | (00000 ₂) 0 ₁₀ | (0000 ₂) 0 ₁₀ | 2 (-2) |
| 1011 ₂ (11 ₁₀) | | (0001 ₂) 1 ₁₀ | (11100 ₂) -4 ₁₀ | (0001 ₂) 1 ₁₀ | 2 (-2) |
| 1100 ₂ (12 ₁₀) | | (0010 ₂) 2 ₁₀ | (11101 ₂) -3 ₁₀ | (0010 ₂) 2 ₁₀ | 2 (-2) |
| 1101 ₂ (13 ₁₀) | | (0011 ₂) 3 ₁₀ | (11110 ₂) -2 ₁₀ | (0011 ₂) 3 ₁₀ | 2 (-2) |
| 1110 ₂ (14 ₁₀) | | (0100 ₂) 4 ₁₀ | (11111 ₂) -1 ₁₀ | (0100 ₂) 4 ₁₀ | 2 (-2) |
| 1111 ₂ (15 ₁₀) | | (0000 ₂) 0 ₁₀ | (00000 ₂) 0 ₁₀ | (0000 ₂) 0 ₁₀ | 3 (-3) |

Since the result of $(A \bmod B)$ has the same sign as B and $\text{abs}(\text{result}) < \text{abs}(B)$, the result of $(A \bmod B)$ is different from $(A \bmod (-B))$. The result of $(A \text{ rem } B)$ has the same sign as A and, thus, $(A \text{ rem } B) = (A \text{ rem } (-B))$. Clearly $(A/B) \neq (A/(-B))$. Table 2.1 (where two's complement codes are used for negative numbers and for positive numbers just the absolute values are given) presents various examples for different values of the first operand (A) and $B = 5_{10}$ with different signs for the latter one (positive: 5_{10} and negative: -5_{10}). The column $/$ ($/^*$) contains only decimal values. Additional details are given in Appendix A.

2.3 Combinational and Sequential Processes

VHDL process is a concurrent statement which is described by sequential statements. Almost always in this book we consider processes with a sensitivity list that appears within parentheses after the **process** keyword (it is recommended for greater flexibility, in particular, by the document [3]). A few examples of processes without a sensitivity list are given in Appendix A (see *on* and *until*). A process is activated if any of sensitivity list signals is changed (i.e. in case of event on these signals). For simulation purposes (see Sect. 2.7) processes with **wait** statement without a sensitivity list will also be used (it is not allowed to include both a sensitivity list and a **wait** statement). Additional details can be found in Appendix A.

2.3.1 Combinational Processes

A process is a *combinational* when all signals/variables assigned in the process explicitly receive new values every time the process is executed [3]. Thus, the sensitivity list must contain: (1) all signals in conditional statements, and (2) all signals on the right-hand side of assignment operators (\leq or \neq). If any value needs to be stored from the previous execution of the process the latter cannot be *combinational*.

There are a number of VHDL constructions that can be used in a process. Some of them (primarily needed for this book) will be described on examples below. The following combinational process tests if the value of an input vector `sw` is between the given low and high bounds (**if** (`sw > low`) **and** (`sw < high`) **then** `led <= sw;`) or less than the low bound (**elsif** `sw < low` **then** `led <= not sw;`):

```
entity TestCombProc is -- simplified syntax rules for processes are given in appendix A
port ( sw      : in  std_logic_vector(7 downto 0);      -- onboard switches
      led      : out std_logic_vector(7 downto 0));      -- onboard LEDs
end TestCombProc;

architecture Behavioral of TestCombProc is
  constant low  : integer := 5;
  constant high : integer := 10;
begin
  cp1: process(sw) -- cp1 (combinational process 1) is an optional label
  begin -- A simplified syntax rule for if...elsif...else...end if statement is given in appendix A
    if (sw > low) and (sw < high) then led <= sw;
    elsif sw < low then led <= not sw;
    else led <= (others => '0');
    end if;
  end process cp1; -- cp1 (combinational process 1) is an optional label

end Behavioral;
```

If the value of `sw` is greater than `low` and less than `high` then this value is displayed on the onboard LEDs. If `sw < low` then the values of all `sw` elements are inverted (applying the **not** operator) and displayed on the LEDs. Otherwise all LEDs are OFF. The statement `led <= (others => '0');` assigns to zero all elements of the signal `led` (corresponding to all LEDs OFF). The following conditional assignments (either the first or the second) directly used in the architecture body instead of the `cp1` process execute exactly the same operations:

```
led <= sw when (sw > low) and (sw < high) else -- the first conditional assignment
      not sw when sw < low else (others => '0'); -- see also Appendix A
with conv_integer(sw) select -- the second (alternative) conditional assignment
  led <= sw      when low+1 to high-1,
  not sw        when low-1 downto 0,
  (others => '0') when others; -- see also Appendix A
```

If statement can be replaced with **case** statement in the following process cp2 below which implements similar to the process cp1 functionality:

```
cp2: process(sw) -- A simplified syntax rule for case statement is given in Appendix A
begin
  case conv_integer(sw) is
    when low+1 to high-1 => led <= sw;
    when low-1 downto 0 => led <= not sw;
    when others          => led <= (others => '0');
  end case;
end process cp2;
```

The next combinational process cp3 can be used to find out the Hamming weight—HW (i.e. the number of ones) in the sw.

```
cp3: process(sw) -- numerous examples with for statement are given in appendix A
variable HammingWeightCount : integer range 0 to 8;
begin
  HammingWeightCount := 0;
  for i in sw'range loop -- HW for sw(7), sw(6), ... , sw(0)
    if sw(i) = '1' then HammingWeightCount := HammingWeightCount+1;
    end if;
  end loop;
  led <= conv_std_logic_vector(HammingWeightCount,8);
end process cp3;
```

The line **for i in sw'range loop** begins a loop that is implemented combinatorially and causes replication of the logic described in the loop body. Index *i* does not need to be declared and it is incremented in a range of the vector *sw* (i.e. 7 **downto** 0 in the order: 7,6,5,4,3,2,1,0). Besides of range we will use some other VHDL attributes shown in Appendix A (see *Attribute*). Let us consider some examples:

| | |
|--|--|
| for i in sw'left downto sw'right+4 loop | -- HW for sw(7 downto 4): i.e. for <i>i</i> values 7,6,5,4 |
| for i in sw'reverse_range loop | -- the order of <i>i</i> values is: 0,1,2,3,4,5,6,7 |
| for i in sw'length-4 downto 0 loop | -- HW for sw(4 downto 0), because the length is 8 |
| for i in 5 downto 3 loop | -- the order of <i>i</i> values is: 5,4,3 |

The following combinational process cp4 demonstrates using the **exit** statement that allows the subsequent index values in the loop to be skipped:

```
cp4: process(sw)
  variable left_1, right_1 : integer range 0 to 8;
begin
  left_1 := 8; right_1 := 8; -- the value 8 is chosen to indicate all zeros in the sw
  for i in sw'range loop -- exit as soon as the first '1' from the left is encountered
    if sw(i) = '1' then left_1 := i; exit;
    end if;
```

```

end loop;
for i in sw'reverse_range loop -- exit as soon as the first '1' from the right is found
    if sw(i) = '1' then right_1 := i; exit;    -- see also exit in Appendix A
    end if;
end loop;
led(7 downto 4) <= conv_std_logic_vector(left_1, 4);
led(3 downto 0) <= conv_std_logic_vector(right_1, 4);
end process cp4;

```

The keyword **next** permits to terminate the loop with the current index value and to continue the loop with the next index value. Note that any iteration with a particular index value is not a cycle in a sequential circuit. Each iteration replicates the logic in the loop body described between the **loop** and **end loop** lines. The loop **while** (also available in VHDL) can be used similarly to the loop **for**. The details are given in Appendix A.

A process may use signals and variables. There is an important difference between them. Assignments (:=) of variables are done immediately (without delays) unlike signal assignments (<=) that are done when the process suspends. The statements in the process are executed sequentially (from the top to the bottom). If there are some mutually reassigned signals in a process they are not updated immediately. For example if A, B are integer signals initialized with the values $A = 10$ and $B = 20$:

```

A <= 5;      -- initialized before with the value 10
B <= A;      -- initialized before with the value 20

```

then at the end of the process (with single invocation) $B = 10$ (but not 5) because the above assignments of A and B are done at the same time at the end of the process (i.e. when the process suspends). Thus, $B = 10$ (the initial value of A) and $A = 5$ (the assigned value in the statement $A \leq 5$ above).

In some practical applications iterative invocations of the same statement are required, for example, the statement $A \leq A + 1$ can be executed in a combinational process with a loop such as **for** or **while**. The results are obviously wrong with the signal A because of the following: (1) the signal A has to be included in the process sensitivity list (because it appears on the right-hand side in the expression above); (2) any change of A (any event on A) forces reinvocation of the same process; (3) a combinational loop is created and this is a wrong for our example. Since variables are assigned immediately, a similar process with variables does not give rise to any problem. Let us consider the following example:

```

entity TestLoops is
    port ( led_signal      : out std_logic_vector (3 downto 0);
          led_variable    : out std_logic_vector (3 downto 0);
          sw              : in std_logic_vector(7 downto 0) );
end TestLoops;

```


architecture Behavioral of TestLoops is

```

signal count_sig : integer range 0 to 15;
begin

use_of_signals: process(sw, count_sig)    -- this process gives definitely wrong results
begin    -- warnings in ISE about a combinational loop are displayed
    count_sig <= 0;
    optional_label: for i in sw'range loop    -- DO NOT USE SIGNALS IN SUCH LOOPS
        if(sw(i) = '1') then count_sig <= count_sig+1;    -- this is definitely wrong
        end if;
    end loop optional_label;
    led_signal <= conv_std_logic_vector(count_sig, 4);
end process use_of_signals;

use_of_variables: process(sw)    -- this process gives correct results
variable count_var    : integer range 0 to 15;
begin
    count_var := 0;
    optional_label: for i in sw'range loop    -- this loop is correct
        if(sw(i) = '1') then count_var := count_var+1;    -- now this line is correct
        end if;
    end loop optional_label;
    led_variable <= conv_std_logic_vector(count_var, 4);
end process use_of_variables;

end Behavioral;

```

It is easy to examine that the first process `use_of_signals` gives wrong results and the second process `use_of_variables` gives correct results.

2.3.2 Sequential Processes

A process is *sequential* if some previously assigned signals keep their previous values and, thus, are not explicitly assigned in a new process execution [3]. We mainly consider *clock-edge-triggered sequential processes* with a sensitivity list and with an eventual *synchronous* reset that can be described as follows:

```

<optional label:> process(clock)    -- clock is the name of the clock signal
< optional declarative part>
begin
    if rising_edge(clock) then    -- the same as: if clock'event and clock = '1' then
        <sequential (possibly conditional) statements>
    end if;
end process <optional label>;

```

The `rising_edge` statement can be replaced with a `falling_edge` statement:

```

if falling_edge(clock) then    -- the same as: if clock'event and clock = '0' then

```

The following example demonstrates communication between several sequential processes. The first process sp1 together with a conditional assignment (marked with `--**`) describe a circuit that reduces the frequency of the clock (clk):

```
sp1: process(clk)
begin
    if rising_edge(clk) then internal_clock <= internal_clock+1; end if;
end process sp1;           -- sw is a 3-bit vector (2 downto 0)
divided_clk <= internal_clock(internal_clock'left - conv_integer(sw)) --**
                    when falling_edge(clk);                          --**
```

The following declarations have to be done in the architecture declarative part:

```
signal internal_clock : unsigned(how_fast downto 0); -- how_fast = 30
signal positive_reset : std_logic; -- this signal will be needed in examples below
signal divided_clk : std_logic;
```

Since `internal_clock` is a 31-bit unsigned vector (`std_logic_vector` can also be used) and the signal `divided_clk` takes (`internal_clock'left - conv_integer(sw)`) bit in the vector `internal_clock`, the frequency of the clock `clk` is divided by $2^{\text{how_fast}+1-\text{conv_integer}(sw)}$. If `conv_integer(sw) = 0` then the base frequency for the Atlys board (which is 100 MHz) is divided by $2^{31} = 2,147,483,648$. Thus, the clock period of the `divided_clk` becomes ~ 21.5 s. If `conv_integer(sw) = 7` then the base frequency is divided by $2^{31-7} = 16,777,216$. Thus, the clock period becomes ~ 0.16 s. The greater the value of `sw` the higher frequency (the shorter period) of the `divided_clk` is provided.

Conditional signal assignment (marked with `--**` in the code above) can be replaced by the following lines in the `sp1` process body:

```
if falling_edge(clk) then
    divided_clk <= internal_clock(internal_clock'left - conv_integer(sw));
end if;
```

The next sequential process `sp2` describes functionality of a binary counter:

```
sp2: process (divided_clk) -- signal count keeps the result of the counter
begin
    if rising_edge(divided_clk) then -- using divided_clk enables the results to be observed visually
        if positive_reset = '1' then count <= (others=>'0'); -- synchronous reset of the counter
        else
            if count_enable = '1' then -- increment/decrement of the counter
                if increment='1' then count <= count + 1;
                else count <= count - 1;
                end if;
            end if;
        end if;
    end if;
end process sp2;
```

Here, `count_enable` is the enable signal for the counter and increment permits either the counter increment (`increment = '1'`) or decrement (`increment = '0'`) to be selected.

The last sequential process sp3 describes functionality of a shift register:

```

sp3: process (divided_clk)          -- signal shift keeps the result of the register
begin                               -- the size of shift is chosen to be (6 downto 0)
    if rising_edge(divided_clk) then -- using divided_clk enables the results to be observed visually
        if positive_reset = '1' then shift <= (others=>'0'); -- reset of the register
        else
            if load_enable = '1' then shift <= count; -- loading the register
            elsif right = '1' then -- shift right/left of the register
                shift <= shift(0) & shift(5 downto 1);
            else
                shift <= shift(4 downto 0) & shift(5);
            end if;
        end if;
    end if;
end process sp3;

```

Here, load_enable is the enable signal for the register (allowing the value of the count from the counter to be loaded) and the signal right permits either the shift right (right = '1') or the shift left (right = '0') to be selected.

The code below includes all the processes described above:

```

entity sequential_processes is      -- pins are given below for the Atlys board
    generic (how_fast: integer := 30 ); -- generic how_fast constant with the default value 30
port ( clk                : in std_logic;          -- clock 100 MHz   -- pin L15
       load_enable         : in std_logic;          -- signal from sw(6) -- pin T5
       count_enable        : in std_logic;          -- signal from sw(7) -- pin E4
       increment           : in std_logic;          -- signal from sw(3) -- pin P15
       right               : in std_logic;          -- signal from sw(4) -- pin P12
       count_shift         : in std_logic;          -- signal from sw(5) -- pin R5
       sw                 : in std_logic_vector(2 downto 0); -- pins C14, D14, A10
       rst                : in std_logic;          -- RESET button   -- pin T15
       led                : out std_logic_vector(7 downto 0)); -- see pins in Fig. 2.5 above

end sequential_processes;

architecture Behavioral of sequential_processes is
    signal internal_clock : unsigned(how_fast downto 0);
    signal positive_reset : std_logic;
    signal divided_clk    : std_logic;
    signal shift, count   : std_logic_vector(5 downto 0);
begin
    positive_reset <= not rst; -- the onboard RESET button for the Atlys produces 0 when pressed
    -- the described above sp1 process
    -- the described above sp2 process
    -- the described above sp3 process
    led(7 downto 2) <= count when count_shift = '1' else shift; -- the results of count or shift
    led(1) <= '0'; -- LED1 is set to OFF
    led(0) <= divided_clk; -- divided_clk with the selected by sw frequency
    divided_clk <= internal_clock(internal_clock'left-conv_integer(sw))
        when falling_edge(clk);
end Behavioral;

```

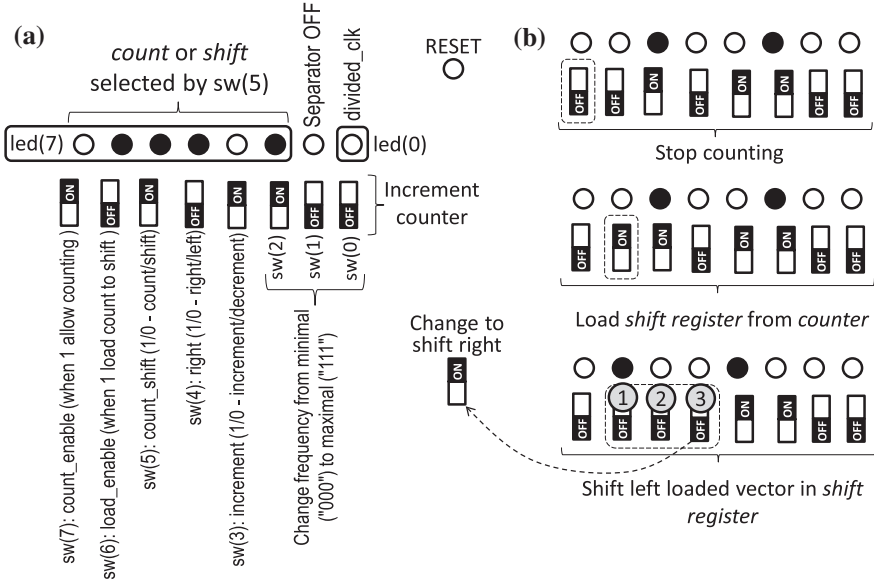


Fig. 2.6 Test of the project with sequential processes: Links with the board components **(a)** and the results of the test **(b)**

Figure 2.6 demonstrates how the results of the project above can be tested.

We already mentioned in the previous section that a process may use signals and variables and that there is an important difference between them. Figure 2.7 gives an additional example of a sequential process in which the block marked with 1 is executed just once. There are two signals A and B in the process test_assign. These signals are updated only when the process suspends. Thus, in the **if** statement within the process test_assign the signals led(1) and led(2) are assigned the previous values of A and B, which is perhaps not the result that you might expect.

```

if B = '1' then      A <= B;      B <= A;
                    led(1) <= A;  led(0) <= B;
end if;

```

If variables would be used instead of signals they would be assigned immediately and, thus, led(1) would receive the updated value of A and led(2) would receive the updated value of B.

In conclusion let us consider a complete example with two processes: test_variable with a variable vA; and test_signal (looking similarly) with a signal sA.

```

entity TestProc is
port ( clk      : in std_logic;
       sw       : in std_logic_vector(3 downto 0);
       led      : out std_logic_vector(7 downto 0));
end TestProc;

```

```

architecture Behavioral of TestProc is
  signal sA          : std_logic_vector(3 downto 0) := (others => '0');
  signal divided_clk : std_logic;
begin -- the lines of the test_variable process are similar to the lines of the test_signal process
  test_variable: process(divided_clk)
    variable vA : std_logic_vector(3 downto 0) := (others => '0');
  begin -- the functionality of the test_variable and the test_signal processes is not the same
    if rising_edge(divided_clk) then
      vA := sw(3 downto 0);      -- a new value is assigned without delay
      led(7 downto 4) <= vA;    -- the new value is displayed
    end if;
  end process test_variable;

  test_signal: process(divided_clk)
  begin
    if rising_edge(divided_clk) then
      sA <= sw(3 downto 0);      -- a new value is assigned
      led(3 downto 0) <= sA;    -- the new value is delayed until the next activation
    end if;                      -- of the test_signal process
  end process test_signal;

  low_freq: entity work.clock_divider
    port map (clk, divided_clk);

end Behavioral;

```

If values of the switches sw3, sw2, sw1, sw0 are changed then these changes first appear on LEDs 7,6,5,4 and only after one period of the clock signal divided_clk — on LEDs 3,2,1,0. Such functionality can easily be examined because the clock frequency is divided (by the clock_divider) up to a visual scale (1 Hz or so).

As follows from the previous examples and explanations, using signals in loops might give problems. For example, if the variable HammingWeightCount is replaced with a signal in the combinational process sp3 in Sect. 2.3 then the functionality will be different from what we might expect (and eventually wrong). Many potential problems of such kind in combinational processes are recognized by synthesis tools which produce warnings about combinatorial (combinational) loops. Thus, the designers are informed. For sequential processes (like shown above and in Fig. 2.7) there is no reason for warnings but in many cases the functionality is different from what we might expect.

2.4 Functions, Procedures, and Blocks

Functions and procedures are used for blocks of codes that need to be invoked multiple times in the design. They permit such functionality to be described that is similar to combinational processes. A *function* is always terminated with a **return** statement and enables a single value to be computed and returned. Simplified syntax rules for functions and procedures are given in Appendix A. Note, that input

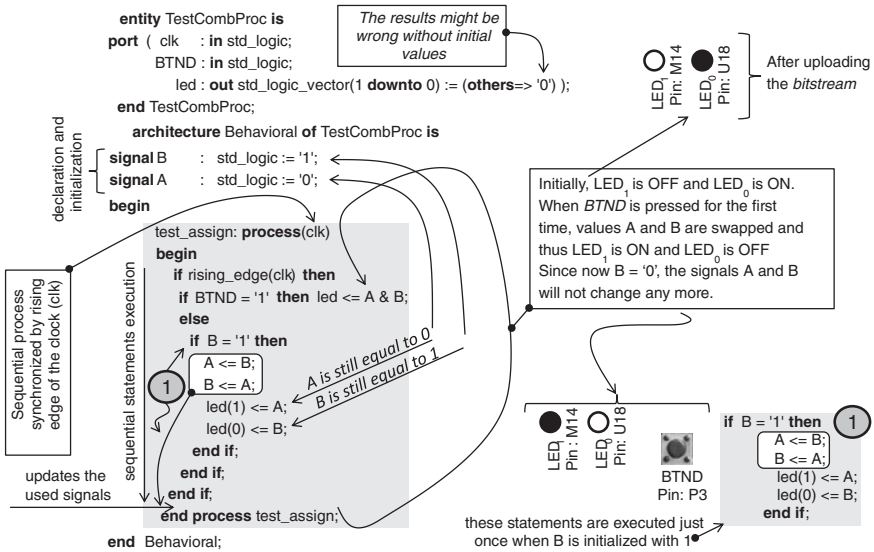


Fig. 2.7 An example demonstrating how a process test_assign is executed

parameters can be unconstrained, i.e. they do not have bounds. Let us describe a function HammingWeight that implements operations of the process sp3 in Sect. 2.3:

```
function HammingWeight(input: std_logic_vector) return integer is  

  variable HammingWeightCount : integer range 0 to input'length;  

begin      -- the "input" parameter is unconstrained above because bounds are not declared  

  HammingWeightCount := 0;  

  for i in input'range loop  

    if input(i) = '1' then HammingWeightCount := HammingWeightCount+1;  

    end if;  

  end loop;  

  return HammingWeightCount;  

end HammingWeight;
```

The code of the function (such as that is shown above) needs to be defined in the declarative part of architecture.

A function can have more than one argument and may activate another function. For example, the following function HammingWeightComparator has three arguments and calls the first function HammingWeight:

```
function HammingWeightComparator(input: std_logic_vector;  

  thresholdLow: integer; thresholdHigh: integer) return Boolean is  

begin  

  if HammingWeight(input) < thresholdLow then return false;  

  elsif HammingWeight(input) > thresholdHigh then return false;  

  else return true;  

  end if;  

end HammingWeightComparator;
```

The code below presents a complete description of a module that invokes the functions `HammingWeight` and `HammingWeightComparator`.

```
entity TestFunctions is
  port ( BTND      : in std_logic;           -- signals from the onboard BTND
        sw        : in std_logic_vector(7 downto 0); -- signals from the onboard switches
        led       : out std_logic_vector(7 downto 0)); -- signals to the onboard LEDs
end TestFunctions;

architecture Behavioral of TestFunctions is
  -- the code of the function HammingWeight given above
  -- the code of the function HammingWeightComparator given above
begin -- invocations of the functions are shown below on simple examples
  led(6 downto 0) <= conv_std_logic_vector(HammingWeight(sw), 7) when BTND='0'
    else conv_std_logic_vector(HammingWeight(not sw(7 downto 4)), 7);
  led(7) <= '1' when HammingWeightComparator(sw, 3, 6) = true else '0';
end Behavioral;
```

It is allowed for a function to use signals that do not appear in the list of the function arguments. However, in such case the function has to be declared as *impure* (all functions are *pure* by default). Let us remove the first argument from the function `HammingWeightComparator` and examine the following code:

```
impure function HammingWeightComparator -- error without the use of the impure keyword
(thresholdLow: integer; thresholdHigh: integer) return Boolean is
begin
  -- the lines from the function HammingWeightComparator given above
end HammingWeightComparator;
```

The line for `led(7)` in the `TestFunctions` entity above has to be also changed (because now there are just 2 arguments) as follows: `led(7) <= '1' when HammingWeightComparator(3,6) = true else '0';`. Now the functionality is exactly the same as before.

The keyword **impure** is an option for a function that extends the scope of variables and signals declared outside of the function that become available in the function. Thus, an *impure function* (in contrast to a *pure function*) may return different values for the same arguments (much like it is shown in the example above).

A function can receive and return values with user-defined types. Let us consider the following example:

```
entity FunctionSort is -- this function was tested for the Nexys-4 board
  port ( sw      : in std_logic_vector(15 downto 0); -- the onboard switches
        led     : out std_logic_vector(15 downto 0)); -- the onboard LEDs
end FunctionSort;

architecture Behavioral of FunctionSort is
  type array4vect is array (0 to 3) of std_logic_vector(3 downto 0); -- user-defined type
  signal my_array : array4vect;
```

```

function sort (Data_in : in array4vect) return array4vect is
  variable data_l1      : array4vect;
  variable data_l2      : array4vect;
  variable Data_out     : array4vect;

begin
  for i in 0 to 1 loop
    if data_in(i*2) <= data_in(i*2+1) then
      Data_l1(i*2) := data_in(i*2+1);      Data_l1(i*2+1) := data_in(i*2);
    else      Data_l1(i*2) := data_in(i*2);      Data_l1(i*2+1) := data_in(i*2+1);
    end if;
  end loop;
  for i in 0 to 1 loop
    if data_l1(i) <= data_l1(i+2) then
      Data_l2(i) := data_l1(i+2);      Data_l2(i+2) := data_l1(i);
    else      Data_l2(i) := data_l1(i);      Data_l2(i+2) := data_l1(i+2);
    end if;
    Data_out(i*3) := data_l2(i*3);
  end loop;
  if data_l2(1) > data_l2(2) then
    Data_out(1) := data_l2(1);      Data_out(2) := data_l2(2);
  else      Data_out(1) := data_l2(2);      Data_out(2) := data_l2(1);
  end if;
  return Data_out;
end sort;

begin
my_array <= (sw(15 downto 12), sw(11 downto 8), sw(7 downto 4), sw(3 downto 0));
(led(15 downto 12), led(11 downto 8), led(7 downto 4), led(3 downto 0)) <=
  sort(my_array);

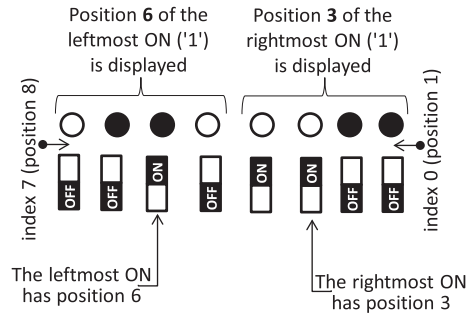
end Behavioral;

```

The function implements a combinational even–odd merge sorting network for four 4-bit data items. It is not important now how the even–odd merge sorting network is coded in the function. Such networks will be described in [Sect. 3.4.1](#). We would only like to demonstrate how to use input and return parameters of user-defined type (*e.g.* array4vect type in the code above). The presented example is ready to be tested in the Nexys-4 board with 16 onboard switches and 16 onboard LEDs. Data items are taken from groups of 4 switches as it is shown above in the assignment to my_array. The results are displayed on LEDs divided in similar groups (4 LEDs in each group shown in the statement above where the function sort is called. Data items are displayed in descending order (the maximum value on led(15 **downto** 12) and the minimum value on led(3 **downto** 0)).

Procedures differ from functions because they permit more than one object to be produced. The following example demonstrates the use of a procedure left1_right1 which finds the first and the last position ‘1’ in the supplied vector (sw). The number of each position is indicated relatively to the right-hand switch starting with 1 (*i.e.* the right-hand switch is assumed to be 1 and not 0 to avoid all zeros on the LEDs when this switch is ON) (see [Fig. 2.8](#)).

Fig. 2.8 An example demonstrating how to test the procedure



```

entity TestProcedure is                                -- see Fig. 2.8 for additional explanations
port ( sw      : in  std_logic_vector(7 downto 0);    -- the onboard switches
      led      : out std_logic_vector(7 downto 0));    -- the onboard LEDs
end TestProcedure;

architecture Behavioral of TestProcedure is

procedure left1_right1
  (signal sw      : in std_logic_vector;
   -- sw is an input vector (all parameters are unconstrained; see appendix A)
   signal f_left  : out std_logic_vector; -- f_left is the first result (the leftmost value 1 in the sw)
   signal f_right : out std_logic_vector) is
   -- f_right is the second result (the rightmost value 1 in the sw)
   variable first_left, first_right : integer range 0 to 8;
begin                                -- initially the leftmost and the rightmost positions of '1' are assigned to be 0
   first_right := 0; first_left := 0;

   for i in sw range loop -- the first loop finds the leftmost position of '1' (from N-1 downto 0)
     if sw(i) = '1' then first_left := i+1; exit; -- the range of first_left is from N downto 1
     end if;
   end loop; -- f_left below receives the value of the leftmost '1' in the given vector
   f_left <= conv_std_logic_vector(first_left, 4);

   for i in sw reverse_range loop -- the second loop finds the rightmost '1' (from 0 to N-1)
     if sw(i) = '1' then first_right := i+1; exit; -- the range of first_right is from 1 to N
     end if;
   end loop; -- f_right below receives the value of the rightmost '1' in the given vector
   f_right <= conv_std_logic_vector(first_right, 4);

end left1_right1;                                -- end of the procedure

   signal first_left, first_right : std_logic_vector(3 downto 0);

begin
  left1_right1(sw, first_left, first_right); -- use of the procedure left1_right1
  led(7 downto 4) <= first_left; -- in this example the vector is taken from 8 switches and the
  led(3 downto 0) <= first_right; -- results are displayed on groups of LEDs (7,6,5,4) and (3,2,1,0)
end Behavioral;

```

If we declare the procedure like the following:

```
procedure left1_right1 ( sw : in std_logic_vector;
    -- sw is an input vector (all parameters are unconstrained; see appendix A)
    f_left: out std_logic_vector;
    -- f_left is the first result (the leftmost value 1 in the sw))
    f_right: out std_logic_vector) is
    -- f_right is the second result (the rightmost value 1 in the sw)
```

then the synthesis tools will report an error saying that the output arguments must be variables whereas the parameters supplied to the procedure sw, first_left and first_right were declared as signals in the entity TestProcedure above. However, the procedure may be called in a process for the parameters first_left and first_right declared as variables like the following:

```
process (sw) -- note that the signal sw does not appear on the left-hand side of assignments in the
    -- procedure left1_right1 and the signal declaration does not give rise to any problem
    variable first_left, first_right : std_logic_vector(3 downto 0);
begin -- pay attention to the correct use of operators <= and := in the procedure left1_right1
    left1_right1(sw, first_left, first_right);
    led(7 downto 4) <= first_left;
    led(3 downto 0) <= first_right;
end process;
```

Let us consider another example in which a procedure finds the minimum and the maximum values in a set of data items used for the function FunctionSort above:

```
entity ProcMaxMin is -- this function was tested for the Nexys-4 board
port ( sw : in std_logic_vector(15 downto 0); -- the onboard switches
    led : out std_logic_vector(7 downto 0)); -- the onboard LEDs
end ProcMaxMin;

architecture Behavioral of ProcMaxMin is
    type array4vect is array (0 to 3) of std_logic_vector(3 downto 0);
    signal my_array : array4vect;
    procedure max_min ( signal Data_in : in array4vect;
        signal max_v : out std_logic_vector;
        signal min_v : out std_logic_vector) is
        variable data_out : array4vect;
    begin
        for i in 0 to 1 loop
            if data_in(i*2) <= data_in(i*2+1) then
                Data_out(i*2) := data_in(i*2+1); Data_out(i*2+1) := data_in(i*2);
            else
                Data_out(i*2) := data_in(i*2); Data_out(i*2+1) := data_in(i*2+1);
            end if;
        end loop;
        if Data_out(0) > Data_out(2) then max_v <= Data_out(0);
        else max_v <= Data_out(2);
        end if;
```

```

        if Data_out(3) < Data_out(1) then      min_v <= Data_out(3);
        else                                  min_v <= Data_out(1);
        end if;
    end max_min;
begin
my_array <= (sw(15 downto 12), sw(11 downto 8), sw(7 downto 4), sw(3 downto 0));
max_min(my_array, led(7 downto 4), led(3 downto 0));
end Behavioral;

```

The method used to find the maximum and the minimum values in a combinational circuit is described in [Sect. 3.6](#) (see [Fig. 3.16](#)). We would only like to demonstrate here how to use different types of procedures. The presented example is ready to be tested in prototyping boards with 16 onboard switches and 8 onboard LEDs. Data items are taken similarly to the function `FunctionSort` above. The results are displayed on LEDs divided in groups: `led(7 downto 4)` for the maximum value and `led(3 downto 0)` for the minimum value.

Blocks are concurrent statements that enable designs to be partitioned. They are intended to clarify hierarchical structure of VHDL modules and (although are not widely used) may be helpful for some projects. A simplified syntax rule for block statements is given in [appendix A](#). We will not use blocks in the subsequent chapters and only minimum details about them are given below. Let us partition the described above module with two functions `HammingWeight` and `HammingWeightComparator` in two blocks labeled `block_with_one_function` and `block_with_another_function`.

```

entity TestBlock is
port ( sw      : in std_logic_vector(7 downto 0);    -- onboard switches
      led      : out std_logic_vector(7 downto 0));  -- onboard LEDs
end TestBlock;

architecture Behavioral of TestBlock is
    signal HW : integer range 0 to 8;
begin
    block_with_one_function: block is                -- the first line of the first block
    -- code of the function HammingWeight given above
    begin
        led(6 downto 0) <= conv_std_logic_vector(HammingWeight(sw), 7);
        HW <= HammingWeight(sw);
    end block block_with_one_function;                -- the last line of the first block

    block_with_another_function: block is            -- the first line of the second block
    -- code of the impure function HammingWeightComparator given above
    begin -- see example available at the Internet (http://sweet.ua.pt/skl/Springer2014.html)
        led(7) <= '1' when HammingWeightComparator(3,6) = true else '0';
    end block block_with_another_function;            -- the last line of the second block
end Behavioral;

```

Functionality of the partitioned design is the same as before. New signal `HW` in the architecture declarative part is used to supply the result of the first block to the second block.

A block statement may include a *guarded signal* assignment that allows the assignment only when the guard condition in the block is true. Let us consider an example:

```

entity TestBlockGuarded is
  port ( clk           : in std_logic;
         enableBTND    : in std_logic;    -- the onboard BTND button
         BTNU          : in std_logic;    -- the onboard BTNU button
         sw            : in std_logic_vector(7 downto 0);  -- onboard switches
         led           : out std_logic_vector(7 downto 0));  -- onboard LEDs
  end TestBlockGuarded;

architecture Behavioral of TestBlockGuarded is
  signal shift_rg      : std_logic_vector(7 downto 0);
  signal divided_clk   : std_logic;
  begin
    -- the block below copies sw to LEDs when BTND=1 and shifts the copied values left
    -- when BTND=BTNU=1
    my_block: block (enableBTND='1' and rising_edge(divided_clk)) is
      begin    -- the guarded assignment below is done only if the condition above is true
        shift_rg <= guarded sw when BTNU = '0' else shift_rg(6 downto 0) & shift_rg(7);
      end block my_block;  -- the end of the block

    led <= shift_rg;      -- the value of shift_rg is displayed on the onboard LEDs

    -- the clock divider below reduces the clock frequency to observe the changes of the LEDs visually
    low_freq: entity work.clock_divider port map(clk, divided_clk);  -- see appendix B
  end Behavioral;

```

If the onboard button `BTND` is pressed the states of the onboard switches are copied to the `shift_rg`; if, in addition, the onboard button `BTNU` is pressed the copied values are shifted left on each rising edge of the `divided_clk`.

2.5 Generics and Generates

Generic statements provide support for scalable designs through supplying such parameters as sizes of vectors, ranges of values, and numbers of repetitive elements. Generics are declared with default values in the entity declarative part. The first example shows the use of different types of generics.

```

entity TestGenerics is  -- it is assumed to be used for the Atlys board
  generic( name         : string := "7954321"; -- generic parameters with default values
          position      : integer := 2;      -- indicated after the characters ":= "
          max_length    : integer := 7;
          my_char0      : character := '0';

```

```

        my_char9      : character := '9';
        MSL           : integer  := 4;
        bool_value    : Boolean := true;
    port (led         : out std_logic_vector(2*MSL-1 downto 0));
end TestGenerics;

architecture Behavioral of TestGenerics is
    signal tmp : Boolean := false;
    begin
        assert (MSL <= 4)           -- if MSL > 4 the message "wrong size for LEDs" is displayed
        report "wrong size for LEDs" -- the message indicated here is displayed if MSL > 4
        severity FAILURE;           -- severity can be NOTE, FAILURE, WARNING and ERROR
        assert position <= name'length -- check the position
        report "position is wrong"
        severity FAILURE;           -- for severity FAILURE terminates the synthesis
        assert name'length <= max_length -- check the maximal length
        report "max length is wrong"
        severity WARNING;           -- for severity WARNING the warning message "max length is wrong"
                                     -- (if activated) appears in the Design Summary/Reports
        led(2*MSL-1 downto MSL) <= std_logic_vector(conv_unsigned
            ((character'pos(name(position))-character'pos(my_char0)), MSL));
        tmp <= bool_value when character'pos(name(position)) >
            character'pos(my_char9) else not bool_value;
        led(MSL-1) <= '1' when tmp else '0';
        led(MSL-2 downto 0) <= conv_std_logic_vector(name'length,MSL-1); -- name'length = 7
    end Behavioral;

```

The result on the LEDs is the value 10010111. The first 4 digits (1001) is the difference in the positions in the ASCII table of the characters ‘9’ and ‘0’. The next bit is 0 because the position of ‘9’ is not greater than the position of ‘9’ (the second character in the string “7954321” is ‘9’ and my_char9 is ‘9’). The last 3 bits (111) represent the length of the string “7954321”.

The generic line `name: string := “7954321”;` defines a generic parameter name which is a string with the default value “7954321” (see *literal* in appendix A). The leftmost character ‘7’ in “7954321” has the position 1 and the rightmost character ‘1’ has the position 7. The part `character'pos(name(position))` in the expression above uses the `pos` attribute (see *attribute* in appendix A). For our example with the default value of the position (i.e. 2) the result of `character'pos(name(2)) = character'pos('9')` returns the position of the character ‘9’ in the ASCII table, which is $57_{10} = 39_{16}$. It can be verified in the following statement:

```

led(2*MSL-1 downto 0) <=
    std_logic_vector(conv_unsigned( (character'pos(name(2)) ), 8 ));

```

displaying on the LEDs the value “00111001” which is a binary equivalent of $57_{10} = 39_{16}$. The `conv_unsigned` and `std_logic_vector` provide the necessary conversion and casting. The similar result can also be obtained in the following statement:

```

led(2*MSL-1 downto 0) <= conv_std_logic_vector(character'pos(name(2)), 8);

```

which produces the LEDs value “00111001”.

It is clearly seen from the code above that the design is scalable. Indeed, it is sufficient to change generic parameters to customize the module for the proper needs. For example, the `tmp` signal indicates if a character in the name is below the position of the character ‘9’ in the ASCII table. If we change the default value of `my_char9` from 9 to, for instance, 5 then a character is checked relatively to the position of the character ‘5’.

The `assert` statement ensures that some constraints are satisfied. For example in the following fragment:

```
assert position <= name'length    -- check position
report "position is wrong"
severity FAILURE;
```

it is checked if the position is less than or equal to the `name'length`. If the condition (less or equal: `<=`) is not satisfied then synthesis is terminated (because of the option `severity` FAILURE;) and the message “position is wrong” is displayed. Similarly other errors and warnings may be discovered and they are shown in the comments above.

We can now use the entity `TestGenerics` as a component of a higher level entity, for instance:

```
entity NowForNexys4Board is    -- it is assumed to be used for the Nexys-4 board
generic (name : string := "FBCD"; -- the default value "7954321" was changed to "ABCD"
         new_position : integer := 3; -- the default value 2 was changed to 3
         max_length   : integer := "FBCD"length; -- the default value 7 was changed to 4
         my_char_F    : character := 'F'; -- the default value '0' was changed to 'F'
         -- the default value '9' for the my_char9 was unchanged
         MSL          : integer := 8); -- the default value '4' was changed to '8'
         -- the default value true for the bool_value was unchanged
port (led : out std_logic_vector(2 * MSL-1 downto 0));
end NowForNexys4Board;

architecture Behavioral of NowForNexys4Board is -- the code is adjusted for the Nexys-4
begin
  assert (MSL <= 8)    -- now the MSL is tested for the value 8
  report "wrong size for LEDs"
  severity FAILURE;
  assert new_position <= name'length -- the name new_position is used instead of the position
  report "position is wrong"
  severity FAILURE;
  assert name'length <= max_length
  report "max length is wrong"
  severity WARNING;
  To_test: entity work.TestGenerics -- unchanged generics my_char9 and bool_value are
                                -- not used in the generic map statement below
    generic map (name => name, position=> new_position,
               max_length => max_length, my_char0 => my_char_F, MSL => MSL)
    port map (led => led);
end Behavioral;
```

As you can see the code above is now used for the Nexys-4 board and the onboard LEDs show the following values: 1111110110000100 (the construction **generic map** permits the default generic values to be replaced with new generic values). The first eight bits 11111101 represent two's complement representation of -3_{10} that is the difference in the positions of 'C' (i.e. 67_{10}) and 'F' (i.e. 70_{10}) in the ASCII table (i.e. position of 'C' minus position of 'F'). Please note, that all the generic names that were not used in the generic map statement were left unchanged.

The second example uses generic parameters for the HammingWeight function described in the Sect. 2.4. Let us create a schematic symbol for the project shown in Fig. 1.6 in Chap. 1. At the beginning we need to add a copy of schematic source from Sect. 1.2.1 (see Fig. 1.6) to a new project, i.e. create a new project and select options *Project* → *Add Copy of Source...* in the ISE and add the file *DistTop.sch* from the previous project. At the next step let us add a new source that is a top level module. Then under the *Design Utilities* option double click on *View HDL Instantiation Template* and copy the following code to the top module:

```
UUT: DistTop port map (-- UUT is a label and we remind that VHDL is not case sensitiv
    s_in => ,
    clk1Hz => ,
    Sw => ,
    s_out => ,
    clock => ,
    BTND => );
```

Finally the top-level module TestGenerics1Top needs the following code:

```
entity TestGenerics1Top is
generic( number_of_bits : integer := 48;    -- generic parameters with default values
        max_bits       : integer := 52;
        bits_sr        : std_logic_vector(4 downto 0) := (4 downto 2 => '0', others=>'1');
        rst             : std_logic := '0');
port (   clk            : in std_logic;
        led             : out std_logic_vector(7 downto 0));
end TestGenerics1Top;

architecture Behavioral of TestGenerics1Top is
    signal Rg : std_logic_vector(number_of_bits-1 downto 0):=(others=>'0');
    signal s_in, clk1Hz, s_out : std_logic;
    signal limit : integer range 0 to max_bits + conv_integer(bits_sr) := 0;
    -- code of the function HammingWeight given above in section 2.4
    begin

    process(clk1Hz) -- the process takes bits from the output s_out of the project from Fig. 1.6
    begin          -- and pushes them to the shift register RG
        if rising_edge(clk1Hz) then
            if limit <= (max_bits + conv_integer(bits_sr)) then -- less than or equal operator <=
                limit <= limit+1;                                -- assignment operator <=
                Rg <= Rg(number_of_bits-2 downto 0) & s_out;
            else Rg <= Rg;
```



```

signal carry_out      : std_logic_vector(N-1 downto 0);
signal sum            : std_logic_vector(N-1 downto 0);
begin
carry_in(0) <= '0';      -- carry in signal for the least significant full adder is zero

generate_adder:        -- an initial line with the label generate_adder at the beginning
for i in 0 to N-1 generate -- "for" is used to generate a network from connected full adders
FA: entity work.FULLADD -- connections are provided through indexed links
    port map( Op1(i), Op2(i), carry_in(i), sum(i), carry_out(i));
    carry_in(i+1) <= carry_out(i);
end generate generate_adder;

led <= carry_out(N-1) & sum; -- the results are displayed on the onboard LEDs

end Behavioral;

```

Figure 2.9 demonstrates how the ripple adder for $N = 4$ has been generated. The figure also gives the user constraints file and shows how the adder can be tested.

Nested generates are also allowed and many examples of networks created with the aid of nested generates will be discussed in the next chapter. Any VHDL component may be generic and the default generic parameters can be replaced with new values by supplying a **generic map** construction. We have already explained such an opportunity when described the entity `NowForNexys4Board` above. For example, we can consider the following higher level component:

```

entity higher_level is
generic( New_N      : integer := 3);
    port( A          : in std_logic_vector(New_N-1 downto 0);
        B          : in std_logic_vector(New_N-1 downto 0);
        result     : out std_logic_vector(New_N downto 0));
end higher_level;

architecture Behavioral of higher_level is
begin
-- other statements
h_level: entity work.Top -- generic map permits default generics to be replaced with new generics
    generic map( N=> New_N) -- now N = New_N = 3
    port map(Op1=>A, Op2=>B, led=>result);
-- other statements
end Behavioral;

```

The construction **generic map** permits the default generic ($N = 4$ in our example for the `Top` entity) to be replaced with the new generic ($New_N = 3$ in our example).

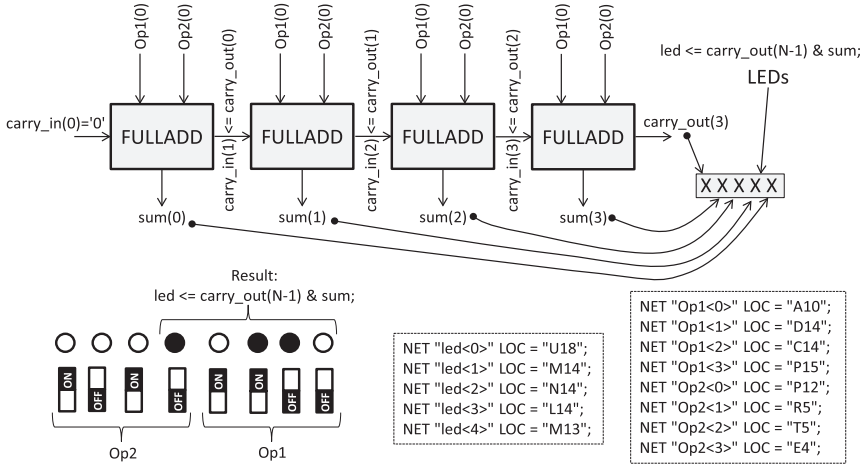


Fig. 2.9 Functionality of the ripple adder

2.6 Libraries, Packages, and Files

A *library* is a location with project's design units (entities or architectures and packages). The default library has the name *work* and contains all synthesizable source files of the project. For example, for the last project of the previous section the *work* panel displays the following five files: *Atlys.ucf*, *Full_adder.vhd*, *Half_adder.vhd*, *higher_level.vhd*, and *Top.vhd*. For the entity *TestGenerics1Top*, considered in the previous section, four files are displayed and one of them contains schematic: *Atlys.ucf*, *Clock_divider.vhd*, *DistTop.sch*, and *GenericsAndAssert.vhd*. If required, a user-defined library can be created, for example, with the name *MyLibrary*. In this case in the ISE the following steps can be done: (1) select *Project* → *New VHDL library* → < specify the name *MyLibrary* and location (directory) of the library >; (2) move necessary files to the *MyLibrary* (select the module and options *Source* → *Move to Library* → *MyLibrary*). Now the new library *MyLibrary* needs to be declared, for example:

```
library MyLibrary;      -- the default library work does not need to be declared
use MyLibrary.all;
```

and the library *work* in the line like: *h_level: entity work.Top* needs to be replaced with a new line: *h_level: entity MyLibrary.Top*.

A *package* permits functions, procedures, constants, types, and components to be described in a (shared) separate file. It provides a way of grouping a collection of related declarations that serve a common purpose [1]. We consider the following three groups: (1) predefined *standard packages*; (2) predefined *IEEE packages*; and (3) *user-defined packages*. The group (1), included by default, is defined

in the *std* and *IEEE* standard libraries and describes the basic types: *bit*, *bit_vector*, *integer*, *natural*, *real* (*real* is frequently not fully supported by synthesis tools), and *boolean*. The group (2) is defined in the *IEEE* packages (that have to be declared) and describes common data types, functions, and procedures. We consider here the following packages supported by the XST [3]: *std_logic_1164* (describing *std_logic*, *std_ulogic*, *std_logic_vector*, and *std_ulogic_vector* types and the relevant conversion functions); *std_logic_arith* (describing unsigned and signed vectors based on the *std_logic* type and the relevant arithmetic operations and functions); *std_logic_unsigned* (describing unsigned arithmetic operators for the *std_logic* and *std_logic_vector* types); *std_logic_signed* (describing signed arithmetic operators for the *std_logic* and *std_logic_vector* types); and *std_logic_textio* (providing support for text-based file input/output). Note, that another available package *numeric_std* is similar to the *std_logic_arith*. The package *std.textio* (defined in the *std* standard library) provides support for a simple text-based file input/output.

A user-defined package (group 3) enables access to shared definitions from project's modules. A simplified syntax rule is given in Appendix A. A package needs to be declared and its body needs to be defined. Let us consider an example:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package MyPackage is          -- declarative part of the package MyPackage
    constant limit           : integer := 10;
    type my_array is array (0 to limit-1) of std_logic_vector(1 downto 0);
    function HammingWeight (input: std_logic_vector) return integer;
    component clock_divider
        port( clk : in std_logic; divided_clk : out std_logic );
    end component;
end MyPackage;

package body MyPackage is     -- body of the package MyPackage
    -- code of the function HammingWeight given above in section 2.4
end MyPackage;
```

The package is created selecting a new source in the ISE (*Project* → *New Source...*) and then *VHDL Package*. Now the package can be used in other modules something like the following:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use work.MyPackage.all;      -- this line is required

entity UsesPackage is        -- we would like to use MyPackage from the work library
    port ( clk               : in std_logic;
           sw                : in std_logic_vector(7 downto 0);
           led               : out std_logic_vector(7 downto 0));
end UsesPackage;
```

```

architecture Behavioral of UsesPackage is
  signal divided_clk      : std_logic;
begin
  -- other eventual statements that might use objects declared in the MyPackage
  led <= conv_std_logic_vector(HammingWeight(sw),8) when divided_clk = '1'
    else (others => '0');

  my_divider : clock_divider port map (clk, divided_clk); -- positional association

end Behavioral;

```

Since the component `clock_divider` is declared in the `MyPackage`, an explicit library indication (such as `my_divider : entity work.clock_divider`) is now not needed.

In [Sect. 1.7](#) we described an interaction of the Atlys board with a host computer using the *IOExpansion* component from Digilent [4]. The module *IOExpansion* can be taken either from a library, for example:

```

IO_interface : entity work.IOExpansion
  port map(EppAsth, EppDsth, EppWr, EppDB, EppWait, MyLed,
           MyLBar, MySw, MyBtn, data_from_PC, data_to_PC);

```

or, alternatively, be declared in a package, for instance:

```

package InteractionWithPC is
component IOExpansion is      -- all the names have to be taken from the IOExpansion [4]
  port (EppAsth: in std_logic; EppDsth: in std_logic; EppWr : in std_logic;
        EppDB  : inout std_logic_vector(7 downto 0); EppWait: out std_logic;
        Led    : in std_logic_vector(7 downto 0);    -- 8 LEDs on the PC side
        LBar   : in std_logic_vector(23 downto 0);   -- 24 light bars on the PC side
        Sw     : out std_logic_vector(15 downto 0);  -- 16 switches on the PC side
        Btn    : out std_logic_vector(15 downto 0);  -- 16 buttons on the PC side
        dwOut  : out std_logic_vector(31 downto 0);  -- 32-bit user-data from PC side
        dwIn   : in std_logic_vector(31 downto 0) ); -- 32-bit user-data to PC side

end component;
end InteractionWithPC;

package body InteractionWithPC is      -- the package body is empty
end InteractionWithPC;

```

Let us demonstrate the same interactions as shown in [Fig. 1.27](#) (see [Sect. 1.7](#)):

```

use work.InteractionWithPC.all;

entity TestIntPC is
port ( sw      : in std_logic_vector(7 downto 0);  -- onboard switches
      led      : out std_logic_vector(7 downto 0); -- onboard LEDs
      EppAsth  : in std_logic;                      -- signals for the IOExpansion component
      EppDsth  : in std_logic;
      EppWr    : in std_logic;
      EppDB    : inout std_logic_vector(7 downto 0);

```

```

        EppWait : out std_logic);
end TestIntPC;

architecture Behavioral of TestIntPC is
    signal MyLed      : std_logic_vector(7 downto 0); -- declarations of user signals
    signal MyLBar      : std_logic_vector(23 downto 0);
    signal MySw        : std_logic_vector(15 downto 0);
    signal MyBtn       : std_logic_vector(15 downto 0);
    signal data_to_PC  : std_logic_vector(31 downto 0);
    signal data_from_PC : std_logic_vector(31 downto 0);
begin
    data_to_PC <= data_from_PC; -- data received from the host PC are sent back to the PC
    MyLed      <= sw;          -- onboard switches are displayed on virtual LEDs (PC side)
    led        <= MySw(7 downto 0); -- 8 switches (PC side) are displayed on the board LEDs
    MyLBar     <= MySw(15 downto 8) & MyBtn; -- 8 switches and MyBtn are displayed

    IO_interface : IOExpansion
        port map(EppAsth, EppDsth, EppWr, EppDB, EppWait, MyLed,
                MyLBar, MySw, MyBtn, data_from_PC, data_to_PC);
end Behavioral;

```

Alternatively the line `use work.InteractionWithPC.all` can be removed and the line `IO_interface : IOExpansion` needs to be replaced with: `IO_interface : entity work.IOExpansion`.

The XST (Xilinx Synthesis Technology) provides a limited support for working with files, which is described in [3]. We consider here only one example demonstrating how to read 8-bit words from a file *data.txt* and to record these words in an array *my_array*.

```

use std.textio.all;           -- this package has to be used
use ieee.std_logic_textio.all; -- this package has to be used

entity TestTextFile is
    -- text file data.txt has to be recorded in the same directory
    port ( clk      : in std_logic; -- ports can be initialized if required (see below)
          led      : out std_logic_vector(7 downto 0) := (others=>'0'));
end TestTextFile;

architecture Behavioral of TestTextFile is
    type my_array is array(0 to 15) of std_logic_vector(7 downto 0);
    impure function read_array (input_data : in string) return my_array is
        file my_file      : text is in input_data;
        variable line_name : line;
        variable a_name    : my_array;
    begin
        for i in my_array'range loop
            readline (my_file, line_name); -- reading a line from the file my_file
            read (line_name, a_name(i)); -- reading std_logic_vector from the line line_name
        end loop;
        return a_name;
    end function;
end Behavioral;

```

```

signal array_name : my_array:=read_array("data.txt"); -- initializing the signal array_name
signal divided_clk : std_logic;                      -- a low-frequency clock

begin

process(divided_clk) -- changes are done with a low frequency and can be appreciated visually
  variable address : integer range 0 to 15 := 0;
  begin
    if rising_edge(divided_clk) then
      led <= array_name(address); -- displaying on the LEDs lines from the file data.txt
      address := address+1;       -- incrementing the address to get the next vector
    end if;
  end process;

  divider: entity work.clock_divider port map (clk, divided_clk);

end Behavioral;

```

The file `my_file` is declared as follows: `file my_file : text is in input_data`; where `input_data` is a string with the file name (`data.txt` in our example) supplied to the function `read_array` as an argument (see: `signal array_name:my_array: = read_array("data.txt");`). Two functions `getline (text, line)` (defined in the package `std.textio`) and `read (line, std_logic_vector)` (defined in the package `std_logic_textio`) are used to get data from the file `data.txt`, where the variables `my_file` and `line_name` have the types `text` and `line`, respectively. The variable `a_name` is an array of 16 vectors of type `std_logic_vector(7 downto 0)`. Thus, firstly a line `line_name` is read: `getline (my_file, line_name)`; and then a vector `a_name(i)` of type `std_logic_vector(7 downto 0)` is taken by the function `read (line_name, a_name(i))`; and returned from the function `read_array`. A similar technique is used in [3] to initialize embedded memories from files like `data.txt`. Figure 2.10 shows how the `TestTextFile` can be tested in the Atlys prototyping board (the file `data.txt` can be prepared in any text editor and saved in the same directory with the project). Additional examples are given in Appendix A (see *file*).

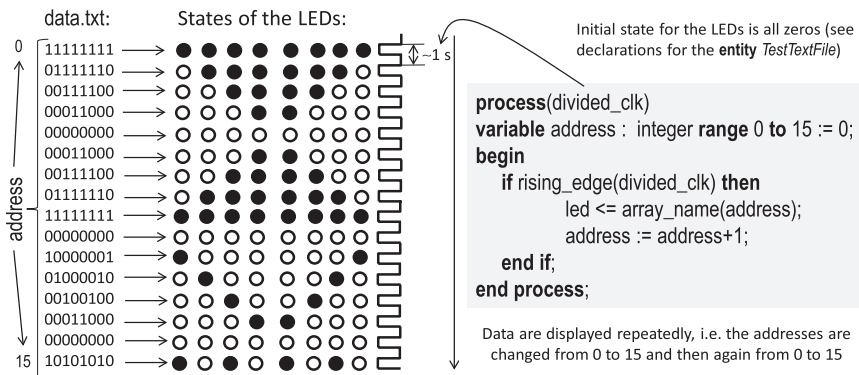


Fig. 2.10 Testing the project, which reads data from the file `data.txt`

Reading from a file can be useful to fill in a memory/array during synthesis which is similar to initialization. Writing to a file cannot be done from a working project (since it is done during synthesis). It may be used for debugging, writing specific constants or generic values [3]. Some examples can be found in [3] and one example is given in Appendix A (see *file*).

2.7 Behavioral Simulation

This section presents a brief introduction to a behavioral (functional) simulation that can be done before an implementation of the project to verify that the logic in the project modules is correct. Additional details can be found in [5, 6]. We will use the Xilinx *ISim* simulator which is automatically installed with the ISE (and selected when needed in the *Design Properties* dialog box of the ISE).

Figure 2.11 explains how a behavioral simulation is organized for which two types of files are required: (1) the *modules* which we would like to examine (VHDL or schematic for our examples); (2) a *test bench* file created for the modules. Besides, simulation libraries for environment specific components (such as libraries for Xilinx primitives and IP cores) have to be included if the primitives/cores are used in the design. A *test bench* file is created for a particular project and supplies stimulus to the modules. The creation can be done in the ISE by adding a new source (of type *VHDL Test Bench*) and associating it with the verified module.

We consider below three examples. The first one demonstrates behavioral simulation for the full adder (FULLADD) described in VHDL in Sect. 2.1, which is a *combinational circuit*. The second example illustrates simulation of a *sequential circuit* that is an up/down binary counter with clock enable and synchronous active-high reset. The counter was taken from the ISE templates available through selection *Edit* → *Language Templates...* → *VHDL* → *Synthesis Constructs* → *Coding Examples* → *Counters* → *Binary* → *Up/Down Counters*. The last example enables the behavior of the circuit created in the ISE schematic editor (see Fig. 1.6 in Chap. 1) with *Xilinx library primitives* to be tested.

All the steps (a, b, and c), needed for the first example, are shown in Fig. 2.11. At the first step (a) we create a project for simulation, i.e. we add a test bench (named TestBenchFA) and associate the test bench with the FULLADD module described in Sect. 2.1. A template for the test bench is proposed by the ISE but we will change the code as it is shown at the right-hand part of Fig. 2.11. The entity FULLADD is instantiated in the architecture and the project structure is shown in Fig. 2.11 near the label a. There is one process (stim_proc) in the architecture body which generates stimulus (inputs of the FULLADD that are changed every 50 ns until the final **wait** statement is reached). At the second step b the test bench is checked for errors. In our case there is no error and we proceed to the last step c where the *Simulate Behavioral Model* option is activated. As a result, the *ISim* window with simulation waveforms is opened. For better analysis the waveforms need to be zoomed (see Fig. 2.11). A cursor permits to check values in a particular time (after 77 ns in our example depicted in Fig. 2.11).

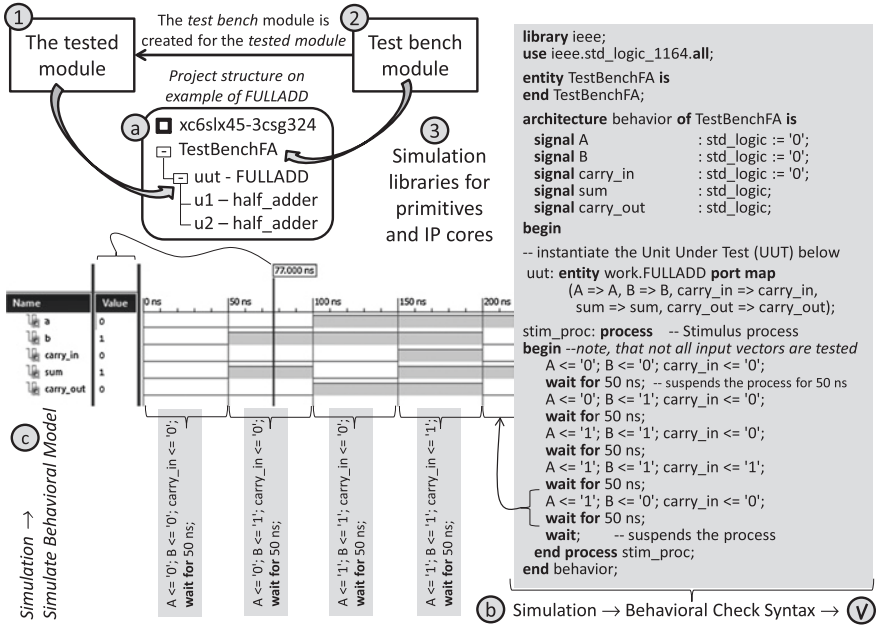


Fig. 2.11 An example of behavioral simulation for the full adder

The following module will be simulated in the second example:

```

entity Counter is
    port (
        reset, clock      : in std_logic;
        clock_enable      : in std_logic;
        inc_dec           : in std_logic;
        outputs           : out std_logic_vector (3 downto 0));
end Counter;

architecture Behavioral of Counter is
    signal count : std_logic_vector(3 downto 0);
begin

    process (clock)
    begin
        if rising_edge(clock) then
            if reset='1' then count <= (others => '0'); -- synchronous reset
            elsif clock_enable='1' then
                if inc_dec='1' then count <= count + 1; -- if inc_dec=1 then increment the counter
                else count <= count - 1; -- if inc_dec=0 then decrement the counter
                end if;
            end if;
        end if;
    end process;

    outputs <= count;
end Behavioral;
  
```


The following test bench for `for_counter` is added and associated with the Counter:

```

entity for_counter is
end for_counter;

architecture behavior of for_counter is
    signal reset          : std_logic := '0';
    signal clock          : std_logic := '0';
    signal clock_enable   : std_logic := '0';
    signal inc_dec        : std_logic := '0';
    signal outputs        : std_logic_vector(3 downto 0);
    constant clock_period : time := 30 ns; -- clock period definitions (valid for simulation only)
begin

    uut: entity work.Counter port map          -- instantiate the unit under test (uut)
        (reset => reset, clock => clock, clock_enable => clock_enable,
         inc_dec => inc_dec, outputs => outputs );

    clock_generator : process                -- clock process definitions
    begin -- the process generates clock pulses
        clock <= '0';
        wait for clock_period/2;              -- duty cycle for the clock is 50%
        clock <= '1';
        wait for clock_period/2;
    end process clock_generator ;

    stim_proc: process                       -- stimulus process
    begin
        reset <= '1';                        -- the first line **reset<='1'**
        wait for 30 ns;                       -- set the reset signal to '1' and wait for 30 ns
        reset <= '0'; clock_enable <= '0'; inc_dec <= '1';
        wait for 20 ns;                       -- change signals as it is indicated above and wait for 20 ns
        reset <= '0'; clock_enable <= '1'; inc_dec <= '1';
        wait for 150 ns;                      -- change signals as it is indicated above and wait for 150 ns
        reset <= '0'; clock_enable <= '1'; inc_dec <= '0';
        wait for 550 ns;                      -- change signals as it is indicated above and wait for 550 ns
    end process;                             -- begin from the line **reset<='1'** after 30+20+150+550=750 ns

end behavior;

```

Since the Counter is a sequential circuit the test bench needs to supply clock signal and it is done in the `clock_generator` process. The simulation results with additional details are given in Fig. 2.12.

The last simulation is done for the circuit in Fig. 1.6 from which the `clock_divider` has been removed (see Fig. 2.13). Indeed, for simulation purposes a low frequency clock is not needed. All the required steps are exactly the same as for VHDL modules (see Fig. 2.11). The only difference is the association of the added test bench with the top-level schematic entity (`DistTop.sch` in our example).


```

clock_generation: process          -- clock process definitions
begin                             -- the process generates clock pulses
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process clock_generation;

-- a stimulus process is not needed because we would like the values
-- of sw and BTND to be permanently assigned in the line below
sw <= (4 downto 3 => '0', others=>'1'); BTND <= '0'; -- settings are the same as in Fig. 1.7
-- if required the values of sw and BTND may be changed in the relevant stimulus process, which
-- will be used instead of the line above

end behavioral;

```

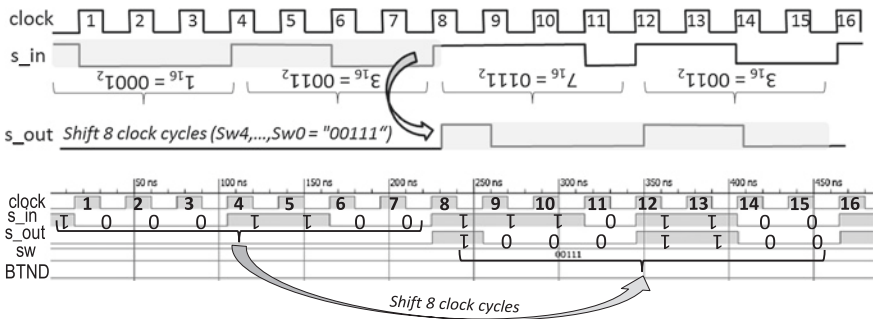


Fig. 2.14 Comparing the results of physical tests in Fig. 1.7 (*the upper part*) and behavioral simulation (*the lower part*)

The results of simulation are exactly the same as in Fig. 1.7. To show it clearer Fig. 2.14 depicts the waveforms from Fig. 1.7 and the results of behavioral simulation that uses the test bench given above.

There are many options and modes of simulation which are not described here and can be found in [5, 6].

2.8 Prototyping

The majority of the considered in this chapter examples can be implemented and tested in different prototyping boards described in Sect. 1.6. Clearly, the user constraints file (i.e. pin assignments) and the FPGA part number have to be changed properly.

The following example has been tested in the DE2-115 board (the Xilinx user constraints file has been changed to the proper Altera setting file [7]):

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity AlteraProject is
    -- all names (except clock and reset) are the same as in [7]
generic( size      : integer := 18; -- the size of vectors for the HammingWeight function
         n_LEDs     : integer := 5; -- the number of the used LEDs

port ( clock      : in std_logic; -- PIN_Y2
      reset      : in std_logic; -- PIN_M23 for key0
      sw         : in std_logic_vector(size-1 downto 0);
      ledr       : out std_logic_vector(n_LEDs-1 downto 0);
      ledg       : out std_logic_vector(n_LEDs-1 downto 0));
end AlteraProject;

architecture Behavioral of AlteraProject is
    -- code of the function HammingWeight from section 2.4 without any change
    signal count      : integer range 0 to size-1;

    signal divided_clk : std_logic;
begin

    process (divided_clk)
    begin
        if rising_edge(divided_clk) then
            if not reset = '1' then count <= 0; -- when the key0 is pressed then count is zero
            else -- if HammingWeight(sw)>0 then count is changed from 1 to HammingWeight(sw)
                count <= (count mod HammingWeight(sw))+1; -- mod is the VHDL modulo operator
            end if;
        end if;
    end process;

    ledr <= conv_std_logic_vector(count, n_LEDs);

    ledg <= conv_std_logic_vector(HammingWeight(sw), n_LEDs);

    divider: entity work.clock_divider
        port map (clock, divided_clk);

end Behavioral;

```

If one or more switches are ON ($\text{HammingWeight}(\text{sw}) > 0$) then the count is changed cyclically from 1 to the $\text{HammingWeight}(\text{sw})$. If reset is active (*key0* button is pressed) then $\text{count} = 0$. The value of the $\text{HammingWeight}(\text{sw})$ is displayed on green LEDs (*ledg*) and the value of count is displayed on red LEDs (*ledr*). The reset signal is active *low* (that is why the **not** operation is applied to this signal).

Some projects of the book use vendor-specific libraries and technology-dependent components. The following VHDL code gives an example:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
library UNISIM;

-- Xilinx LUT-based computation of the Hamming weight (see
-- the simplest Hamming weight counter in section 3.9)
-- Xilinx library UNISIM for LUT primitives that are used below

```

```

use UNISIM.VComponents.all;

entity LUT_6to3 is
    port ( SixBitInput      : in std_logic_vector (5 downto 0); -- 6-bit input vector
          ThreeBitOutput   : out std_logic_vector (2 downto 0)); -- 3-bit Hamming weight
end LUT_6to3;

architecture Behavioral of LUT_6to3 is -- Xilinx LUTs below are configured in such a way that
begin -- permits the Hamming weight of 6-bit input vector to be produced in a combinational circuit

LUT6_inst1 : LUT6 -- Xilinx LUT primitive LUT6
    generic map (INIT => X"fee8e880e8808000") -- LUT Contents
    port map (ThreeBitOutput(2), SixBitInput(0), SixBitInput(1), SixBitInput(2),
              SixBitInput(3), SixBitInput(4), SixBitInput(5));
LUT6_inst2 : LUT6 -- Xilinx LUT primitive LUT6
    generic map (INIT => X"8117177e177e7ee8") -- LUT Contents
    port map (ThreeBitOutput(1), SixBitInput(0), SixBitInput(1), SixBitInput(2),
              SixBitInput(3), SixBitInput(4), SixBitInput(5));

LUT6_inst3 : LUT6 -- Xilinx LUT primitive LUT6
    generic map (INIT => X"6996966996696996") -- LUT Contents
    port map (ThreeBitOutput(0), SixBitInput(0), SixBitInput(1), SixBitInput(2),
              SixBitInput(3), SixBitInput(4), SixBitInput(5));

end Behavioral;

```

The code above cannot be synthesized in the Quartus environment for Altera FPGAs. However, an alternative code below that uses constants instead of the Xilinx LUT6 primitive can be synthesized and works fine for both Altera and Xilinx FPGAs:

```

library IEEE; -- the code below is tested in the DE2-115 board
use IEEE.STD_LOGIC_1164.all; -- with the Altera Cyclone-IVF FPGA
use IEEE.STD_LOGIC_UNSIGNED.all; -- this package is needed for type conversions below

entity LUT_6to3 is
    port ( SixBitInput      : in std_logic_vector (5 downto 0);
          ThreeBitOutput   : out std_logic_vector (2 downto 0));
end LUT_6to3;

architecture Behavioral of LUT_6to3 is

type LUT is array (2 downto 0) of std_logic_vector(63 downto 0);
-- array below contains the same constants as used in the INIT statements in the code with LUTs above
constant conf_LUT : LUT := ( X"fee8e880e8808000", -- array of constants
                              X"8117177e177e7ee8", -- is used here
                              X"6996966996696996");

begin -- Hamming weight is found in the statements below

ThreeBitOutput <= conf_LUT(2)(conv_integer(SixBitInput)) &
                  conf_LUT(1)(conv_integer(SixBitInput)) &
                  conf_LUT(0)(conv_integer(SixBitInput));

-- alternatively the following generate statement can be used:
-- gen: for i in conf_LUT'range generate
--     ThreeBitOutput(i) <= conf_LUT(i)(conv_integer(SixBitInput));
-- end generate gen;

end Behavioral; -- the same code can be used for Xilinx FPGAs without any change

```

The two given above VHDL codes describe similar functionalities and permit the Hamming weight of 6-bit input vectors to be calculated in combinational circuits. The first code explicitly configures the Xilinx LUTs and the second code implicitly configures actually the same LUTs but without the need for vendor-specific libraries. The circuit built by Altera Quartus occupies 8 logic elements and the circuit built by Xilinx ISE for the Nexys-4 board occupies 3 LUTs. Such way enables the projects of the book to be also implemented and tested in FPGAs of other companies.

Similarly the majority of other modules described in this chapter have been tested in the DE2-115 board.

Many additional examples can be found in [8, 9].

References

1. Ashenden PJ (2008) The designer's guide to VHDL, 3rd edn. Morgan Kaufmann
2. Ashenden PJ (2008) Digital design: an embedded systems approach using VHDL. Morgan Kaufmann
3. Xilinx Inc (2013) XST user guide for Virtex-6, Spartan-6, and 7 series devices. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/xst_v6s6.pdf. Accessed 17 Nov 2013
4. Digilent Inc (2009) Adept I/O expansion reference design. <http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,828&Prod=ADEPT2>. Accessed 9 Nov 2013
5. Xilinx Inc (2011) ISE In-Depth Tutorial. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/ise_tutorial_ug695.pdf. Accessed 17 Nov 2013
6. Xilinx Inc (2009) Synthesis and simulation design guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf. Accessed 17 Nov 2013
7. Altera Inc (2013) Quartus II setting file with pin assignments for DE2-115. <http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html>. Accessed 17 Nov 2013
8. Skliarova I, Sklyarov V, Sudnitson A (2012) Design of FPGA-based circuits using hierarchical finite state machines. TUT Press, Tallinn
9. Sklyarov V, Skliarova I (2013) Parallel processing in FPGA-based digital circuits and systems. TUT Press, Tallinn

Synthesis and Optimization of FPGA-Based Systems

Sklyarov, V.; Skliarova, I.; Barkalov, A.; Titarenko, L.

2014, XIX, 432 p. 235 illus., Hardcover

ISBN: 978-3-319-04707-2