Appendix A

VHDL Constructions Used in the Book and Additional Support Materials

Abstract We present here concise information about used in the book synthesizable constructions and keywords (reserved words) of VHDL that are listed alphabetically and complemented with a brief informal description. We provide also a few useful tables (e.g. ASCII) with supplementary data needed for different chapters. Words written in *ITALIC SMALL CAPS* need to be replaced with user code.

**Absolute Value—abs**

This is a unary operator that is predefined for any numeric type and returns an absolute value of the operand. See examples in Sect. 2.2.

**Aggregate**

is a grouping of values to form an array or record expression. In *positional association*, the values are associated with elements from left to right. *Named association* indicates explicitly each value. Note that positional association cannot follow named association. Example:

```vhdl
-- Aggregates below are used in order to assign a record (positional association):
record_data <= ('0', '1', "01"); -- record_data is a signal of type my_packet (see below)
type my_packet is record
  first_bit, second_bit : std_logic;
data : std_logic_vector (1 downto 0);
end record;
```
Elements can be grouped by named association, where the keyword `others` indicates the remaining elements:

\[
\begin{align*}
0=>\text{bit}3, & \ 2=>\text{bit}2, & \ 1=>\text{bit}1, & \ 3=>\text{bit}0 \\
\text{A}(7 \ \text{downto} \ 0) <= (7=>'0', & \ 5 \ \text{downto} \ 4 => '0', & \ \text{others} => '1'); & \ -- \ \text{an \ example \ of \ named \ association}
\end{align*}
\]

### Aggregates and Arrays

are shown on examples below. Two-dimensional arrays may be declared as follows (arrays and indices can be signals or variables):

- `type my_array is array (3 downto 0) of std_logic;` -- type for a one-dimensional array
- `type my_packet is array (0 to 9) of my_array;` -- type for a two-dimensional array
- `signal my_data : my_packet;` -- my_data is a two-dimensional array
- `type array2vect is array (0 to 1) of std_logic_vector(1 downto 0);` -- type for a one-dimensional array
- `type array4vect is array (0 to 3) of array2vect;` -- type for a two-dimensional array
- `signal table : array4vect;` -- table is a three-dimensional array
- `signal table_line : array2vect;` -- table_line is a two-dimensional array
- `signal table_data : std_logic_vector(1 downto 0);` -- table_data is a one-dimensional array

The following statement assigns all the LEDs the value ‘1’ (the LEDs are ON):

```
(0=>bit3, 2=>bit2, 1=>bit1, 3=>bit0)
```

The next statements give results shown in the comments:

```
A(7 downto 0) <= (7=>’0’, 5 downto 4 => ’0’, others => ’1’); -- an example of named association
```

Suppose the following declarations are done in the architecture.

```
type my_array is array (3 downto 0) of std_logic; -- type for a one-dimensional array
type my_packet is array (0 to 9) of my_array; -- type for a two-dimensional array
signal my_data : my_packet; -- my_data is a two-dimensional array
```

Different results can be tested in the architecture body using onboard LEDs:

```
led <= table(0)(0) & table(0)(1) & table(1)(0) & table(1)(1) & table(2)(0) & table(2)(1) & table(3)(0) & table(3)(1); -- there are 16 individual LEDs: led(15 downto 0)
```

The following statement assigns all the LEDs the value ‘1’ (the LEDs are ON):

```
table <= (others=>(others=>(others=>’1’)));
```

The next statements give results shown in the comments:

```
-- the result below is: led(15 downto 0) = 00 01 10 11 11 00 10 01
table <= (“00”,”01”), (“10”,”11”), (“11”,”00”), (“10”,”01”);
```

-- `sw(15 downto 0)` are connected to `led(15 downto 0)` with the same indices
```
table <= ((sw(15 downto 14), sw(13 downto 12)), (sw(11 downto 10), sw(9 downto 8)), -- #
(sw(7 downto 6), sw(5 downto 4)), (sw(3 downto 2), sw(1 downto 0))); -- #
```

Different results can be tested in the architecture body using onboard LEDs:

```
led <= (1 to 2 =>(1=>(others=>’1’), 0=>”01”), others=>(others=>(others=>’0’))); -- the result below is: led(15 downto 0) = 00 00 01 11 01 11 00 00
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>”11”, 1=>”00”),
2 =>(0=>”00”, 1=>”01”), 3 =>(0=>”10”, 1=>”11”);
```

-- `sw(1 downto 0)` in the process below are connected to the `leds(15 downto 0)` with the same indices
```
table <= (1 =>(0=>sw(1 downto 0), 1=>sw(3 downto 2)),
1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>”11”, 1=>”00”),
2 =>(0=>”00”, 1=>”01”), 3 =>(0=>”10”, 1=>”11”);
```

-- `sw(3 downto 2)` control `led(13 downto 12)`, etc.
```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

-- `sw(1 downto 0)` control `led(15 downto 14)`.
```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

```
table <= (0=>”01”, 1=>”10”), 1 =>(0=>sw(5 downto 4), 1=>sw(7 downto 6)),
2 =>(0=>sw(9 downto 8), 1=>sw(11 downto 10)),
3 =>(0=>sw(13 downto 12), 1=>sw(15 downto 14))); -- sw(15 downto 0)
```

---
Appendix A: VHDL Constructions Used in the Book

Alias declaration permits an alternative name to be defined for an object.Alias declarations may be done in declarative parts. Alias declaration is done like the following:

```
alias <NEW NAME> is <EXISTING IDENTIFIER>;
```

All

identifies all declarations within the package or library, for example:

```
use ieee.std_logic_1164.all;
```

Architecture

is demonstrated on a general template below:

```
architecture <NAME OF ARCHITECTURE> of <NAME OF ENTITY> is
  -- declarative part
  -- declarations (of signals, components, functions, procedures)
  -- definitions (of types)
begin
  -- architecture body
end <NAME OF ARCHITECTURE>;
```

Array

is declared in the following general form:

```
type <NAME OF TYPE> is array <RANGE OF ARRAY> of <TYPE OF ELEMENTS>;
```
One-dimensional, two-dimensional, and three-dimensional arrays are shown above in *aggregates and arrays*.

**ASCII Table**

provides an encoding for 128 characters. It is given in Table A.1 for 33 special characters (codes 0, \ldots, 31, 127) and in Table A.2 for the remaining 95 printable characters (codes 32, \ldots, 126).

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nul</td>
</tr>
<tr>
<td>1</td>
<td>soh (start of heading)</td>
</tr>
<tr>
<td>2</td>
<td>stx (start of text)</td>
</tr>
<tr>
<td>3</td>
<td>etx (end of text)</td>
</tr>
<tr>
<td>4</td>
<td>eot (end of transmission)</td>
</tr>
<tr>
<td>5</td>
<td>enq (enquiry)</td>
</tr>
<tr>
<td>6</td>
<td>ack (acknowledge)</td>
</tr>
<tr>
<td>7</td>
<td>bel (bell)</td>
</tr>
<tr>
<td>8</td>
<td>bs (back space)</td>
</tr>
<tr>
<td>9</td>
<td>ht (horizontal tab)</td>
</tr>
<tr>
<td>10</td>
<td>lf (line feed)</td>
</tr>
<tr>
<td>11</td>
<td>vt (vert. tab)</td>
</tr>
<tr>
<td>12</td>
<td>ff (form feed)</td>
</tr>
<tr>
<td>13</td>
<td>cr (carriage return)</td>
</tr>
<tr>
<td>14</td>
<td>so (shift out)</td>
</tr>
<tr>
<td>15</td>
<td>si (shift in)</td>
</tr>
<tr>
<td>16</td>
<td>dle (data link escape)</td>
</tr>
<tr>
<td>17</td>
<td>dc1 (device control 1)</td>
</tr>
<tr>
<td>18</td>
<td>dc2 (device control 2)</td>
</tr>
<tr>
<td>19</td>
<td>dc3 (device control 3)</td>
</tr>
<tr>
<td>20</td>
<td>dc4 (device control 4)</td>
</tr>
<tr>
<td>21</td>
<td>nak (negative acknowledge)</td>
</tr>
<tr>
<td>22</td>
<td>syn (synchronous idle)</td>
</tr>
<tr>
<td>23</td>
<td>etb (end of transmitted block)</td>
</tr>
<tr>
<td>24</td>
<td>can (cancel)</td>
</tr>
<tr>
<td>25</td>
<td>em (end of medium)</td>
</tr>
<tr>
<td>26</td>
<td>sub (substitute)</td>
</tr>
<tr>
<td>27</td>
<td>esc (escape)</td>
</tr>
<tr>
<td>28</td>
<td>fsp (file separator)</td>
</tr>
<tr>
<td>29</td>
<td>gsp (group separator)</td>
</tr>
<tr>
<td>30</td>
<td>rsp (record separator)</td>
</tr>
<tr>
<td>31</td>
<td>usp (unit separator)</td>
</tr>
<tr>
<td>127</td>
<td>del (delete)</td>
</tr>
</tbody>
</table>
**Assert**

describes a condition that has to be evaluated and it is normally used to report warning and error messages (see Sect. 2.5 for details).

**Attribute**

is a named characteristic of certain objects and it permits constraints to be described directly in the code. In this book only predefined attributes (existing for types, arrays, and signals) have been used. They are defined in the form: type/array/signal<NAME OF ATTRIBUTE>. An example of attribute event for signal clk is: if clk'event and clk = '1' then … (clock is changing from 0 to 1, i.e. the same as in if rising_edge(clk) then …) or if clk'event and clk = '0' then … (clock is changing from 1 to 0, i.e. the same as in if falling_edge(clk) then …). Examples of other useful attributes are:

- Test for an ascending range, for example: led_flash <= divided_clk when not led'ascending else '1'; If the LED is a descending range then the led_flash gets the divided clock value.
- The highest value of a type: integer'high.
- Test for the last value lv just before the last event on lv. For example: last_led <= lv(3)'last_value;
- Left-bound index signal'left. Let us consider the following example: internal_clock(internal_clock'left) where the type of internal_clock is std_logic_vector.
- Length of a dimension array'length. For example: my_RAM(i)'length.
- Position within a type: type'pos(...). For example: character'pos('A') returns the position of ‘A’ in the ASCII table, which is 65.
- Left downto/to right in an array array'range. For example: for i in input'range loop.
• Right downto/to left in an array array\textquotesingle{}reversed_range. For example: for i in input\textquotesingle{}reversed_range loop.
• Right-bound index signal\textquotesingle{}right For example: internal_clock(internal_clock\textquotesingle{}right) where the type of internal Clock is std_logic_vector.

The following lines give examples of user-defined attributes:

\begin{verbatim}
attribute LOC : string; -- specifying location constraints
attribute LOC of led: signal is "P2"; -- the led signal is assigned to the pin P2
attribute IOSTANDARD : string; -- specifying input/output standard
attribute IOSTANDARD of led: signal is "LVCMOS33"; -- see the user constraints file for Nexys-4
\end{verbatim}

**Begin**

marks the beginning of process/function/procedure statements or architecture body and the end of the respective declarative part in the process/function/procedure/architecture. It is also used in some other constructions such as in blocks and to describe multiple instances in generate statements.

**Block**

is a concurrent statement simplifying partition of a design and declared using the following basic format (examples are given in Sect. 2.4):

\begin{verbatim}
<OPTIONAL LABEL>: block (<OPTIONAL BOOLEAN GUARD EXPRESSION>) is
begin
  -- concurrent statements
end block <OPTIONAL LABEL>;
\end{verbatim}

**Body**

is used with the package reserved word (see package).

**Buffer**

is used for a port and enables the relevant signal to be read and written. Such port may have not more than one source and can be connected only to another buffer or linked to a signal with no more than one source. As distinct from inout ports, buffer ports cannot be connected to tri-state buses (see inout) and they allow output signals declared as ports in a module to be also read in the module.
Appendix A: VHDL Constructions Used in the Book

Case

statement has the following general form:

case < EXPRESSION > is
  when <VALUE OF THE EXPRESSION> => <STATEMENTS>;
  -- continue for other values: when <value of the expression> => <statements>;
  when others => <STATEMENTS>;
end case;

Case statements are used in processes, functions, and procedures and cannot be used directly in architectures (if required when...else can be applied instead in the architecture body). The following simple example demonstrates the use of the case statement:

process(A, B, C) -- A,B,C are integers: signal A,B,C: integer range 0 to 7;
begin
  case (A+B+C) is
    when 1 to 3 | 5 | 10 => led <= '1'; -- when A+B+C = 1 or 2 or 3 or 5 or 10
    when others => led <= '0';
  end case;
  case (A+B+C)>12 is
    when true => led1 <= '1'; -- when A+B+C is greater than 12
    when others => led1 <= '0';
  end case;
end process;

Component

is a declaration in a higher-level entity enabling a lower-level entity to be instantiated. The following two templates can be used for component declaration and instantiation.

component <NAME OF COMPONENT>
generic (  
  <NAME OF GENERIC> : <type> := <DEFAULT VALUE OF "NAME OF GENERIC">;
  <other generics...> );
port (  
  <NAME OF PORT> : <mode of port such as in, out, inout, buffer> <type>;
  <other ports...> );
end component;

generic map (  < POSITIONAL OR NAMED ASSOCIATIONS > )
port map (  < POSITIONAL OR NAMED ASSOCIATIONS > );

Note that positional associations cannot follow named associations.
Component entities can be included in a library. The library named work is available by default. The following line demonstrates the use of this library without explicit component declarations:
A library with a different name can also be created (see Sect 2.6). The majority of examples in the book assume the use of the default library work without explicit component declarations.

**Constant**

can be declared in any declarative region. Constant values cannot be changed.

```
constant <NAME OF CONSTANT> : <type> := <USER VALUE>;
```

Examples:

```
constant line_with_equal_sign : string(1 to 3) := " = "; -- the symbol = is placed in between two spaces
constant ternary_vector : std_logic_vector(5 downto 0) := "01-1-0";
constant my_integer : integer := 7;
constant line1 : string(7 downto 1):="Index:" & CR; -- CR is a non-printing character
constant binary_constant : std_logic_vector(5 downto 0) := "011100";
```

The following VHDL entity (test_const) gives an additional demonstration.

```
entity test_const is
  port ( sw : in std_logic_vector (2 downto 0);
         led : out std_logic_vector (6 downto 0));
end test_const;
architecture Behavioral of test_const is
  constant binary : std_logic_vector(6 downto 0) := "0101010";
  constant octal : std_logic_vector(6 downto 0) := o"12" & '1';
  constant hexadecimal : std_logic_vector(6 downto 0) := x"a" & o"5";
  constant decimal : integer := 63;
  type rom is array (0 to 3) of std_logic_vector (6 downto 0);
  constant ex : rom :=(x"6" & o"3", x"8" & "101", '1' & o"45", o"3" & '0' & o"2");
begin
  led <= binary when sw = "001" else octal when sw = "010" else hexadecimal when sw = "011" else ex(0) when sw = "100" else ex(1) when sw = "101" else ex(2) when sw = "110" else ex(3) when sw = "111" else conv_std_logic_vector(decimal,7);
end Behavioral;
```

**Conversion Functions**

cert types (see *type conversions* and Sect. 2.2).
**Downto**

declares a direction in a range, for example, A(7 downto 0).

**End**

concludes descriptions (statements) in a process/function/procedure/architecture. It is also used in some other constructions such as block and generate statements.

**Entity**

describes inputs and outputs of the design module. It is demonstrated on a general template below:

```vhdl
entity <NAME OF ENTITY> is
genetic ( <NAME OF GENERIC> : <type> := <USER VALUE>; 
<other generics if required...> );
port ( <NAME OF PORT> : <mode of port such as in, out, inout, buffer > <type>; 
<other ports if required...> );
end <NAME OF ENTITY>;
```

**Enumerated Type**

may be user-defined, such as that is frequently needed for listing names of states in FSMs as it is shown in the example below:

```vhdl
type state_type is (init, run_state); 
-- there are two states in the state_type: init and run_state
```

**Exit**

forces exiting from the innermost loop or from the loop with the indicated label. In the following example the variable count (declared as: variable count : integer range 0 to 4:= 0;) is always equal to 0 if the statement exit a is used and always equal to 4 when the statement without the label a (e.g. exit) is used:

```vhdl
a: for i in 0 to 3 loop
  for j in 0 to 3 loop
    if i = j then exit a;
      -- ......................
  end if;
end loop;
count := count+1;
end loop a;
```

-- a is an optional label
-- begin of the innermost loop
-- count is always equal to 0 with the label a
-- count is always equal to 4 without the label a
-- end of the innermost loop
-- a is an optional label
is a type that provides for interaction of the design with storage devices. An example with files is given in Sect. 2.6. In another example below the function `read_array` from Sect. 2.6 uses a while loop to read data from the file `data.txt`:

```vhdl
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;
    variable index  : natural;
begin
    index := 0;
    while not endfile(my_file) loop
        readline (my_file, line_name);
        read (line_name, a_name(index));
        index := index+1;
    end loop;
    return a_name;
end function;
```

The next example demonstrates writing to a file:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_arith.all;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity WriteToFile is
generic (M : integer := 32;
        N : integer := 1024);
port ( const_bit :
        out std_logic;
        bit_number :
        in integer range 0 to 14);
end WriteToFile;

architecture Behavioral of WriteToFile is
    -- opening the file MyFile.txt for writing
    file generic_and_constants : text open write_mode is "MyFile.txt";
    constant oct_const: std_logic_vector(14 downto 0) := o"37145";
begin
    process (bit_number) -- combinational process
        variable file_line : LINE;
        begin
            write(file_line, string("----------"));
            writeline(generic_and_constants, file_line);
            write(file_line, string("M = "));
            write(file_line, M);
            writeline(generic_and_constants, file_line);
            write(file_line, string("N = "));
            write(file_line, N);
            writeline(generic_and_constants, file_line);
            write(file_line, string("The maximum value of an integer: "));
            write(file_line, integer'high);
            writeline(generic_and_constants, file_line);
            write(file_line, string("Decimal value of octal constant: "));
            write(file_line, conv_integer(oct_const));
            for i in 3877 to 3879 loop
                write(file_line, conv_std_logic_vector(i,12));
            end loop;
            const_bit <= hex_const(bit_number);
        end process;
end Behavioral;
```

```
Suppose the module \texttt{WriteToFile} is a component of another module in which it is declared as follows:

\begin{verbatim}
entity Top is
generic (M: integer := 16; N: integer := 512);
port -- descriptions of ports
end Top;
architecture Behavioral of Top is
begin
test: entity work.WriteToFile
    generic map (M, N)
    port map (const_bit, bit_number);
end Behavioral;
\end{verbatim}

The file \texttt{MyFile.txt} is created during synthesis and is composed of the following lines:

\begin{verbatim}
--------------------
M = 16
N = 512
The maximum value of an integer: 2147483647
Decimal value of octal constant: 15973
111100100101
111100100110
111100100111
\end{verbatim}

It contains those generic values that have been provided by the entity \texttt{Top}. Files may be useful for such tasks as initializing arrays (ROMs), reading stimulus for simulations, etc. The predefined package \texttt{textio} in the library \texttt{std} collects useful functions, types and operations that permit to read/write files from the design. Additional details can be found in [1, 2].
For

Statement permits to replicate logic in generate and loop constructions and may also serve some other purposes [1, 2]. Iterations for-loop can be used in processes, functions, and procedures. Suppose the following signal is declared:

```vhdl
signal vector : std_logic_vector(N-1 downto 0);
```

The following fragments demonstrate examples of for statements:

```vhdl
<OPTIONAL LABEL>: for i in vector'range loop -- 1) elements of the vector from N-1 to 0 will be used
-- statements that specify logic that has to be replicated
end loop <OPTIONAL LABEL>;

for i in vector'reverse_range loop -- 2) elements of the vector from 0 to N-1 will be used
-- statements that specify logic that has to be replicated
end loop;

for i in N-1 downto 0 loop -- 3) elements of the vector from N-1 to 0 will be used
-- statements that specify logic that has to be replicated
end loop;

for i in a downto b loop -- 4) elements from a to b will be used (a>=b, a<N, b>=0)
-- statements that specify logic that has to be replicated
end loop;

for i in vector'left downto vector'right loop -- 5) elements from N-1 to 0 will be used
-- statements that specify logic that has to be replicated
end loop;
```

Function

Computes a single value and always terminates with a return statement. It is declared according to the following general format:

```vhdl
function <NAME OF THE FUNCTION> (<LIST OF INPUT PARAMETERS>) return <TYPE> is
<DECLARATIVE PART>
begin
-- sequential statements (the function body)
end <NAME OF THE FUNCTION>;
```

Input parameters can be unconstrained, i.e. they may have no bounds. The body of a function is similar to the body of a combinational process. Section 2.4 is dedicated to functions with many simple examples.
Appendix A: VHDL Constructions Used in the Book

Generate construction is used to instantiate an array of components allowing concurrent statements to be replicated. The modes for and if can be applied which is illustrated below on an example (Fig. A.1 shows the output_vector for the given input_vector):

```
entity Test_generic is
generic ( N : integer := 8);
port ( input_vector : in std_logic_vector (N-1 downto 0);
      output_vector : out std_logic_vector (N-1 downto 0));
end Test_generic;
architecture Behavioral of Test_generic is
begin
example: for i in N-1 downto 0 generate
  exchange: if (i >= 2 and i <= 3) generate
    for_example: entity work.OneBitComparator
    port map( input_vector(i), input_vector(i+2), output_vector(i), output_vector(i+2));
  end generate exchange;
  copy: if (i < 2) generate
    output_vector(i) <= input_vector(i);
    output_vector(i+6) <= input_vector(i+6);
  end generate copy;
end generate example;
end Behavioral;
```

Generic supplies particular constant values to entities and components. A default value may be included which will be used if another value is not specified in the generic map. Section 2.5 is dedicated to generics with many simple examples. Generic map constructions permit default generic values in lower-level modules to be changed.
Guarded Signal

allows a concurrent assignment to be done only when the guard condition in the block with this assignment is true (see an example of the entity TestBlockGuarded in Sect. 2.4).

If

is a conditional statement which can be used in a process, function, and procedure. Many simple examples are given in Sect. 2.3. This statement is demonstrated on a general template below:

```
if <condition1> then
  <statements are executed if the condition1 is true>
elsif <condition2> then
  < statements are executed if the conditions1 is false and 2 is true>
else
  < statements are executed if both the conditions are false >
end if;
```

-- then, elsif, else, end if are reserved words

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 (see Sect. 2.4 for details).

In

is a port mode allowing the port to be read only. If no mode is declared it is assumed to be in.

Inout

is a mode for bidirectional ports (for read and write). Inout is mainly aimed at indication of a tri-state port that can be both an output and an input. It can be applied to such signal MyBus that might be assigned like the following: MyBus <= MyIntBus when (MyWr = ‘1’) else (others => ‘Z’);. Ports with inout type have to be used when bidirectional communications are actually needed. Inout type can also be used for procedures (see procedure) and permits to return values of the respective arguments to the calling module and to read/write these arguments in the procedure.
Is

links identities to definitions in different constructions, for example: 
architecture Behavioral of TestTextFile is.

Library

permits the resources of a library to be used. The following examples demonstrate declaration of libraries IEEE and UNISIM:

```vhd
library IEEE; -- "library", "use" and "all" are reserved words
use IEEE.STD_LOGIC_1164.all; -- see details in section 2.6
use IEEE.STD_LOGIC_ARITH.all; -- see details in section 2.6
use IEEE.STD_LOGIC_UNSIGNED.all; -- see details in section 2.6
library UNISIM; -- these lines have to be included if Xilinx primitives and the vendor-specific libraries are used
use UNISIM.VComponents.all;
```

The library work does not need to be declared. A user-defined library must be defined explicitly (see an example in Sect. 2.6).

Literal

is a value specified in the design that appears in expressions in form of a number, a character, a string, or a bit string:

- Numbers are represented by integer and real literals (a synthesizable object of type real cannot be defined). Examples of decimal integer literals are 45, 0, 1872. A base can be different from 10. In this case the number is enclosed in sharp characters # preceded by the base that can be any integer from 2 to 16. For example, the value 25 can be written as: 2#11_001#, or 16#19# or 5#100#. Separators (underscores _) are ignored in the literal and permit a more readable form to be used. The considered in the book conversions can be applied to numbers, for example leds(10 downto 2) <= conv_std_logic_vector(3#01_10#, 9);

- A character literal is a single character enclosed in single quotation marks, for example: ‘3’, ‘f’, ‘S’, ‘ ’ (where in the last case the character is the space). There are a number of special (non-printable) characters that can be indicated by their predefined names (such as del), for example: signal special: character: = del; (see ASCII table).

- Strings are sequences of characters enclosed in double quotation marks, for example: “this is a string”, “” (where in the last case the string is empty). Two strings can be joined by the concatenation operator (&). The indices are positive numbers and the range may be either ascending or descending, although the majority of applications use an ascending range beginning with 1 [1]. The latter is also frequently used as the default initial index in a string range.
A bit string literal represents a string of binary values. For example, a binary vector can be described in the following form: \(B^\text{“1_0_01”}\), where \(B\) (or \(b\)) is the base (binary). Other possible bases can be \(O\) or \(o\)—octal and \(X\) or \(x\)—hexadecimal. As before, separators (underscores _) are ignored. A bit string literal can be assigned to \text{std_logic_vector} with the proper size: \(\text{SLV}(2 \text{ downto } 0) \leq B^\text{“1_0_01”}\).

**Logical Operators**

are summarized in Table A.3.

**Map**

associates names in a block (port or generic) with external names. Positional and named associations can be used. Let us consider some examples:

```vhdl
divider: entity work.clock_divider -- positional association
   port map (clk, divided_clk);

entity clock_divider is
   port (  c : in std_logic; -- c corresponds to the signal clk in the upper level module
         d_c : out std_logic); -- d_c corresponds to the signal divided_clock
end clock_divider;
```

Named association may look like the following:

```vhdl
divider: entity work.clock_divider -- names in upper and lower level modules may be the same
   port map (c=>clk, d_c=>divided_clk); -- here external (clk, divided_clk)
      -- and internal (c, d_c) names are different
```

**Modulo—mod**

This is a binary operator that is predefined for integer types. The result has the sign of the second operand and absolute value less than the absolute value of the second operand. It is defined as: a \text{mod} b = a - b \times n, where \(n\) is some integer. See examples in Sect. 2.2.
Names

are used for identifiers. They may be composed of letters, digits, and underscores. Names are non-sensitive to case (i.e. AAa and aAA names are the same). Names must start with a letter, may not end with an underscore character, and may not include two successive underscore characters. VHDL reserved words cannot be used as names.

Next

terminates the current iteration (replication of logic) in loops and initiates a new iteration (replication). Much like the statement exit, next may have an optional label (e.g. next a;) which has the same interpretation as for the exit.

Null

indicates that no action is to be performed (normally used in case and if statements, for example, when others => null).

Of

links identities to names or types of elements, for example, type my_array is array(0 to 7) of std_logic_vector(15 downto 0).

Open

is used to leave the specified port unassociated. The IEEE VHDL specification does not allow unconnected input ports, but unconnected (open) output ports are permitted.

Operands

that are used in the book are: array aggregates, bit string literals, enumeration literals, function calls, integers, physical literals (only for behavioral simulation), record aggregates, string literals, static expressions, type conversions (see also literals above).
Operators

are divided into arithmetic (+, −, *, /, abs, mod, rem, sign + and −, **), concatenation (&), logical (and, nand, nor, not, or, xor, xnor), relational (=, /=, <, <=, >, >=), assignment (:=, <=) and shift (rol, ror, sla, sll, sra, srl). Some operators are represented by special symbols composed of individual or pairs of characters. A pair of characters, such as <= (if they correspond to an operator) must be typed next to each other without a space in between them. Logical operators can be combined with relational operators (ex. if ((a > b) and (c /= d)) or be bitwise (e.g. a xor b where a and b are std_logic_vector signals with equal sizes). All operators shown in bold font are reserved words. Operators are organized in the following groups according to their precedence (power − **), (abs), (not), (*), (/), (mod, rem), (+ identity, − negation), (+, −), (&), (rol, ror, sll, srl), (sla, sra), (=, /=), (<, <=, >, >=), (and, nand, nor, or, xnor, xor) with the highest priority in the first and with the lowest priority in the last group. Parentheses can be used to change the order of operators and are recommended for clarity.

On

is used in wait statements to introduce the sensitivity list as follows: wait on <SENSITIVITY LIST> until <BOOLEAN EXPRESSION>. An example is given below:

```vhdl
process     -- sequential process;
begin        -- for a combinational process the line below is changed to: wait on count;
    wait on count until rising_edge(divided_clk);
    led <= count;
end process;  -- support for wait statement is often limited (see, for example, restrictions in [2])
```

Others

is used as the last branch of case statement and the right part of a signal/variable assignments to cover not specified values in the case statement and to assign values to not-assigned array elements. Some examples are given below (see also aggregate):

```vhdl
type memory is array (15 downto 0) of std_logic_vector (7 downto 0);
signal s_mem : memory := (others => (others => '0'));  -- all elements of 2-dimensional array are zeros
    -- beginning of a case statement
when others => null;
```

Out

is a port mode allowing the port to be only written.
**Package**

permits to describe functions, procedures, constants, and types in a separate file, which can be shared by different projects. See examples in Sect. 2.6. A general template for a package is shown below:

```vhdl
package <NAME OF THE PACKAGE> is
  type -- optional declaration of types
  constant -- optional declaration of constants
  function -- optional declaration of functions
  procedure -- optional declaration of procedures
end <NAME OF PACKAGE>;

package body <NAME OF THE PACKAGE> is
  -- definition of functions and procedures
end <NAME OF THE PACKAGE>;
```

---

**Port**

is a signal enabling an entity to communicate with other upper-level modules. **Port map** construction defines mapping of signals from upper-level modules to signals from lower-level modules.

**Procedure**

differs from a function because it permits more than one signal to be produced. A general template for a procedure is:

```vhdl
procedure <NAME OF PROCEDURE> (<LIST OF INPUT, OUTPUT AND INOUT PARAMETERS>) is
  -- declarative part
begin
  -- sequential statements (the procedure body)
end <NAME OF PROCEDURE>;
```

Arguments of mode `out` and `inout` in procedures return their values to the calling module. Parameters can be unconstrained, i.e. they may have no bounds. The body of a procedure is similar to the body of a combinational process. Section 2.4 is dedicated to procedures and demonstrates simple examples.

**Process**

describes a level of hierarchy in a design. Different processes are executed concurrently (in parallel with other processes and concurrent signal assignments). A general template for a process is:
Statements within a process are executed sequentially. Update of signals is done when the process suspends. Signal assignments (<=) inside processes do not take effect immediately as distinct from variables for which assignments (:=) of values are done immediately. The SENSITIVITY LIST is a set of signals written in parentheses after the word process. Any change in (any event on) these signals causes the process to be activated. The sensitivity list of a combinational process (for a combinational circuit) must contain all the input signals and the process must update all the output signals. Sequential processes include edge-triggered clocked timing. One process might change signals in a sensitivity list of another process. Processes without a sensitivity list should include a wait statement. Section 2.3 is dedicated to processes with many simple examples.

**Pure**

is an option for a function that does not allow the usage of signals or variables declared outside of the function. All functions are pure by default (see Sect. 2.4 and impure functions for details).

**Qualified Expression**

(type(expression)) permits the type of expression to be explicitly indicated, for example:

```vhdl
architecture ..... 
signal user_signal : integer range 0 to 15 := 11;
begin
user_out <= unsigned("0000") + user_signal;
```

**Range**

permits an interval of allowed values to be explicitly defined (see subtype). The following example declares a range of integers:

```vhdl
signal user_signal : integer range -5 to 10; -- the range of integers is from -5 to 10
```
Record

permits a collection of data (with the same or different types) to be represented. A record can be declared as a type (see additional information in type). Record type signals can be assigned using aggregates (see additional information in aggregates). The following example demonstrates how a type record can be declared for a serial package that might be used in communications through RS232 interface:

```vhdl
type serial_package is record  -- type definition
    start_bit    : std_logic;
    data_bits    : std_logic_vector (7 downto 0);
    parity_bit   : std_logic;
    stop_bit     : std_logic;
    number       : integer range 0 to 127;
end record;

signal my_sp : serial_package;  -- declaration of my_sp signal of type serial_package
```

The following example shows how to access and assign individual fields:

```vhdl
my_sp.number <= 10;  my_sp.start_bit <= '1';
my_sp.data_bits <= (others => '1');
```

Relational Operators

= (equal to), /= (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to).

Remainder—rem

is a binary operator for remainder that is predefined for any integer type. The result has the sign of the first operand and is defined as: \( a \text{ rem } b = a - (a / b) \times b \). See examples in Sect. 2.2.

Report

is a statement for generating report messages (see Sect. 2.5 for details).

Return

terminates a function with this statement and passes control to the calling module. Any function must have a return statement.
**Select**

can be used in signal assignments in the body of an architecture. For example, the following architecture describes the full adder from Sect. 2.1:

```vhdl
architecture STRUCT of FULLADD is
signal three_bits : std_logic_vector(2 downto 0);
begin
    three_bits <= A & B & CIN;
    with three_bits select
        SUM <= '1' when "100"|"010"|"001"|"111", -- SUM=1 for the listed vectors
            '0' when others; -- SUM=0 for non-listed vectors
    with three_bits select
        COUT <= '1' when "011"|"101"|"110"|"111", -- COUT=1 for the listed vectors
            '0' when others; -- COUT=0 for non-listed vectors
end STRUCT; -- another example is given in subtype
```

**Severity**

is a predefined type with the values note, warning, error and failure (see Sect. 2.5 for details).

**Shared Variable**

**Shared** keyword allows different processes to access the same variable. Shared variables can be declared only in entities, architectures, and generates (which are places where normal variables cannot be declared) according to the following syntax:

```
shared variable <VARIABLE_NAME> : <NAME OF TYPE> := <EXPRESSION>;
```

For example (N is the number of RAM words, M is the size of the words):

```vhdl
type type_of_the_RAM_block is array (0 to N-1) of std_logic_vector (M-1 downto 0);
shared variable RAM_block : type_of_the_RAM_block;
```

Xilinx recommends shared variables to be used to model a RAM with two write ports (examples are given in [1]).

**Shift Operators rol, ror, sla, sll, sra, srl**

The document [1] indicates that these operators are defined for a one-dimensional array with bit or Boolean elements. There are two arguments: A and B, where A is the array and B is the number of the array positions which are either shifted or rotated. Assuming that the operand A has N bits (N-1 downto 0) and the operand B is an integer, the following logical equivalence can be given:
Appendix A: VHDL Constructions Used in the Book

- **rol** (rotate left): \( A(N-B-1 \text{ downto } 0) \) & \( A(N-1 \text{ downto } N-B) \);
- **ror** (rotate right): \( A(B-1 \text{ downto } 0) \) & \( A(N-1 \text{ downto } B) \);
- **sla** (shift left arithmetic): \( A(N-B-1 \text{ downto } 0) \) & \( (B-1 \text{ downto } 0 => A(0)) \);
- **sll** (shift left logic): \( A(N-B-1 \text{ downto } 0) \) & \( (B-1 \text{ downto } 0 => '0') \);
- **sra** (shift right arithmetic): \( (B-1 \text{ downto } 0 => A(N-1)) \) & \( A(N-1 \text{ downto } B) \);
- **srl** (shift right logic): \( (B-1 \text{ downto } 0 => '0') \) & \( A(N-1 \text{ downto } B) \);

The following example provides an additional demonstration:

```vhdl
entity L_shift is -- the library numeric_std has to be included (use ieee.numeric_std.all)
port(clk  : in std_logic; -- system clock 100 MHz
     sw : in unsigned(15 downto 0); -- switches of the Nexys-4 or any other board
     led : out unsigned(13 downto 0) ); -- LEDs of the Nexys-4 or any other board
end L_shift;
architecture Behavioral of L_shift is
signal data_in  : unsigned(13 downto 0); -- an input vector from the switches
signal data_tmp : unsigned(13 downto 0); -- a temporary vector that is rotated
signal sel     : unsigned(1 downto 0); -- selects the number of positions to rotate
signal divided_clk : std_logic; -- low frequency (1Hz) to make the rotations visible
begin
  data_in  <= sw(13 downto 0); -- taking an input vector from the switches
  sel     <= sw(15 downto 14); -- taking the sel value from the switches
  process (divided_clk) -- sequential process that demonstrates the use of the rol operator
  begin
    if rising_edge(divided_clk) then
      case sel is
        when "00" => data_tmp <= data_in ; -- taking an initial vector from the switches
        when "01" => data_tmp <= data_tmp rol 1; -- rotate one position (B=1)
        when "10" => data_tmp <= data_tmp rol 2; -- rotate two positions (B=2)
        when "11" => data_tmp <= data_tmp rol 3; -- rotate three positions (B=3)
        when others => data_tmp <= data_in ;
      end case;
      -- the operators ror, sll, srl can be used above instead of the operation rol
    end if;
  end process;
  led <= data_tmp; -- showing rotated data on leds
end Behavioral;
```

We would prefer to use logically equivalent operators shown above instead of the described here shift operators.

### Signal

**Signals** model physical wires in hardware circuits. They are assigned with a pair of symbols <= and any assignment involves a delay (one delta delay by default). The latter applies when an assignment is done within a block or as a part of sequential statements within a process (see TestProc entity in Sect. 2.3.2 for
Signals differ from variables which are assigned immediately. Signals are declared in architectures (and cannot be declared in processes, procedures or functions) in the following form:

```
signal <NAME OF SIGNAL> : <TYPE OF SIGNAL>);
signal <NAME OF SIGNAL> : <TYPE OF SIGNAL> := <INITIAL VALUE>);
```

Signals can be used in bodies of architectures, processes, procedures or functions and can be formal parameters of a function or a procedure. A sensitivity list of a process cannot include variables and includes only signals.

Concurrent signal assignments (<=), conditional signal assignments (when ... else) and selected signal assignments (with ... select ... when) can be used in architecture body. In processes (procedures) normally only sequential signal assignments (<=) are allowed. The following rules are the most important:

1. Sequential signal assignments will be done in a process only when the process suspends.
2. If there are several assignments in the process to the same signal only the last one takes effect.

**Subtype**

Introduces constraints or subsets of values for the chosen base type. It is declared in the following general form:

```
subtype <NAME OF SUBTYPE> is <BASE_TYPE>
  range <VALUES IN RANGE>;
```

The use of subtypes is considered on an example below.

```
entity types_and_subtypes is
  port ( switches : in std_logic_vector(1 downto 0); -- two switches
         leds : out std_logic_vector (3 downto 0)); -- four LEDs
end types_and_subtypes;

architecture Behavioral of types_and_subtypes is
  subtype four_bits_std_logic_vector is std_logic_vector (3 downto 0);
  type my_pack is array (0 to 3) of four_bits_std_logic_vector; -- a subtype of std_logic_vector
  constant set_of_lines : my_pack := (x"F", b"00_11", o"6'0", "0101"); -- defining a constant value
begin
  with switches select
  begin
    leds <= set_of_lines(0) when "00", -- displayed value is "1111" = x"F"
           set_of_lines(1) when "01", -- displayed value is "0011" = b"00_11"
           set_of_lines(2) when "10", -- displayed value is "1100" = o"6'0"
           set_of_lines(3) when "11", -- displayed value is "0101" = "0101"
           (others => '0') when others;
  end Behavioral;
```

To
declares a direction in a range, for example, A(0 to 7).
Type

is declared in the following general form:

```
type <NAME OF TYPE> is <SPECIFICATION OF TYPE>; -- see also enumerated type
```

Table A.4 summarizes information about types most commonly used in synthesizable VHDL (resolved type permits signals to be driven by more than one source). Note that there are many restrictions for using the type real.

Each type allows a set of values and a set of associated operations. There are several groups of predefined types such as scalar (bit, boolean, character, enumerated, integer, physical, real, severity) and composite (array, bit_vector, record, string).

Unsigned vector “1111” corresponds to integer 15 and signed vector “1111” corresponds to integer −1. The latter is represented in two’s complement notation, i.e. the most significant bit indicates the sign (1 is minus ‘−’ and 0 is plus ‘+’) and has a negative weight −2^3, while all the remaining bits have positive weights (2^0, 2^1 and 2^2, accordingly) which are equal to 2^x where x is the index of the respective bit (the least significant bit has an index 0).
Type Conversions

are frequently required. They are provided either automatically, through type casts, or with the aid of conversion functions (see also Sect. 2.2). Type cast is used to convert equal sized signed or unsigned to std_logic_vector and vice versa:

\[
\begin{align*}
\text{signed_vector} & \leq\text{signed}(\text{std_logic_vector}_{\text{signal}}) \\
\text{unsigned_vector} & \leq\text{unsigned}(\text{std_logic_vector}_{\text{signal}}) \\
\text{std_logic_vector}_{\text{signal}} & \leq\text{std_logic_vector}(\text{signed_vector}) \\
\text{std_logic_vector}_{\text{signal}} & \leq\text{std_logic_vector}(\text{unsigned_vector})
\end{align*}
\]

The following assignments need conversion functions:

\[
\begin{align*}
\text{integer}_{\text{signal}} & \leq\text{conv_integer}(\text{unsigned}_{\text{vector}}) \\
\text{integer}_{\text{signal}} & \leq\text{conv_integer}(\text{signed}_{\text{vector}}) \\
\text{integer}_{\text{signal}} & \leq\text{conv_integer}(\text{std_logic_vector}_{\text{signal}}) \\
\text{unsigned}_{\text{vector}} & \leq\text{conv_unsigned}(\text{integer}_{\text{signal}}, \text{size}_{\text{of unsigned}_{\text{vector}}}) \\
\text{signed}_{\text{vector}} & \leq\text{conv_signed}(\text{integer}_{\text{signal}}, \text{size}_{\text{of signed}_{\text{vector}}}) \\
\text{std_logic_vector}_{\text{signal}} & \leq\text{conv_std_logic_vector}(\text{integer}_{\text{signal}}, \text{size})
\end{align*}
\]

Using conversion functions requires the relevant libraries to be included. For example the conv_integer function is defined in the library std_logic_unsigned (or std_logic_signed) and conv_std_logic_vector function is defined in the library std_logic_arith.

Until

is used in the condition of a wait statement (see on). An example is given below. The support is limited [1].

\[
\text{process} \\
\text{begin} \\
\text{wait until rising_edge(divided_clk) and BTNC = '1';} \\
\text{count} \leq\text{count} + 1; \\
\text{end process;}
\]

Use

enables functions, procedures, constants, and types of a package to become accessible (visible) in an associated entity/architecture.

Variable

Variables in VHDL are very similar to variables in general-purpose programming languages. They can be declared and used in processes, procedures, and functions. Assignments are allowed from signals to variables (<variable> := <signal>; ) and vice
versa (<signal> <= <variable>), however type match has to be satisfied and the proper operator (:= or <=) must be chosen. Variable assignments take effect without a delay (as distinct from signal assignments).

**Wait**

suspends a process. The document [1] recommends describing processes with a sensitivity list and indicate the following limitations: (1) only one wait statement is allowed and it must be the first in the process; (2) the condition in the wait statement has to describe a clock signal. See also on and until.

**When**

can be used in case statements and in signal assignments (see examples in case and select).

**While**

statement permits repeated operations to be implemented in replicated logic. It has the following general form:

```
<OPTIONAL LABEL>: while <CONDITION> loop
    -- sequential statements;
end loop <OPTIONAL LABEL>;
```

Let us consider an example:

```
process (vector) -- this process finds the position (from 1 to 8) of the first '1' in the vector
variable first_right : integer range 0 to N;
variable i  : integer range 0 to N;
begin
    first_right := 0; -- variables have to be used here
    i := 0;
    while i < N loop  -- vector is declared as std_logic_vector (7 downto 0);
        if vector(i) = '1' then
            first_right := i+1; exit;
        else
            i := i+1;
        end if;
    end loop;  -- positions of the vector bits are: 8 for bit 7, 7 for bit 6, 6 for bit 5, etc.
    led <=conv_std_logic_vector(first_right, 8); -- if vector = "00010100" then the result is 0011
end process;  -- the result 0011 indicates position 3 (for bit 2) which is the first '1' from the right
```

**With**

is used in a selected signal assignment (see select).
References


Appendix B
Coding Examples

Abstract Appendix B includes coding examples for frequently needed modules, any of which can easily be located by name. Entities have exactly the same names that are used for the relevant components described in Chaps. 3 and 4. VHDL codes, user constraints files, and bitstreams for all the projects are available online at http://sweet.ua.pt/skl/Springer2014.html.

Binary to BCD Converters (BinToBCD8)

The following VHDL code is a complete description of a module, which converts 8-bit binary numbers (binary) to binary coded decimal (BCD) numbers (BCD2, BCD1, BCD0):

```vhdl
library IEEE; -- a conversion can also be done on request and this will be
use IEEE.STD_LOGIC_1164.all; -- shown after the next example BinToBCD16
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity BinToBCD8 is -- Binary to BCD converter for 8-bit numbers of std_logic_vector type
generic( size_of_data_to_convert : integer := 8 );
port ( clk : in std_logic;
reset : in std_logic; -- ready is 0 when the number is being converted
binary : in std_logic_vector (size_of_data_to_convert-1 downto 0);
BCD2 : out std_logic_vector (3 downto 0); -- BCD code for the most significant digit
BCD1 : out std_logic_vector (3 downto 0); -- BCD code for the digit in the middle
BCD0 : out std_logic_vector (3 downto 0)); -- BCD code for the least significant digit
end BinToBCD8;

architecture Behavioral of BinToBCD8 is
type state is (idle, op, done);
signal c_s, n_s : state;
signal BCD2_c, BCD1_c, BCD0_c, BCD2_n, BCD1_n, BCD0_n : unsigned(3 downto 0);
signal BCD1_tmp, BCD0_tmp : unsigned(3 downto 0);
signal BCD2_tmp : unsigned(2 downto 0);
signal int_rg_c, int_rg_n : std_logic_vector (size_of_data_to_convert-1 downto 0);
```

V. Sklyarov et al., Synthesis and Optimization of FPGA-Based Systems, Lecture Notes in Electrical Engineering 294, DOI: 10.1007/978-3-319-04708-9, © Springer International Publishing Switzerland 2014
Figure B.1a explains an interface with the module BinToBCD8. Signal ready is valid during one clock cycle and it indicates that the result of conversion is ready to be used. A new data item for conversion can be prepared when ready=0. The code
may be slightly changed in such a way that as soon as ready is active the FSM in the module above is continued to be in the idle state until a request for a new conversion is received. Additional details will be given after the next example.

**Binary to BCD Converters (BinToBCD16)**

The following VHDL code is a complete description of a module, which converts 16-bit binary numbers (binary) to binary coded decimal (BCD) numbers (BCD4, BCD3, BCD2, BCD1, BCD0):

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity BinToBCD16 is
-- binary to BCD converter for 16-bit numbers of std_logic_vector type
generic(size_of_data_to_convert : integer := 16);
port(
  clk : in std_logic;
  reset : in std_logic;
  ready : out std_logic; -- ready is 0 when the number is being converted
  binary : in std_logic_vector(size_of_data_to_convert-1 downto 0);
  request : in std_logic; -- a request is assumed to be sent when ready is active (1)
  BCD4 : out std_logic_vector(3 downto 0); -- BCD code for the most significant digit
  BCD3 : out std_logic_vector(3 downto 0);
);
end entity BinToBCD16;
```

Fig. B.1 Interface with the BinToBCD8 module (a), an example of conversion (b), and interface with the BinToBCD16 module (c)
process (clk, reset)
begin
  if rising_edge(clk) then
    if reset = '1' then
      c_s <= idle;
      BCD4_c <= (others => '0'); BCD3_c <= (others => '0'); BCD2_c <= (others => '0');
      BCD1_c <= (others => '0'); BCD0_c <= (others => '0');
      BCD0 <= (others => '0'); BCD1 <= (others => '0'); BCD2 <= (others => '0'); BCD3 <= (others => '0');
      BCD4 <= (others => '0');
    else
      c_s <= n_s;
      BCD4_c <= BCD4_n; BCD3_c <= BCD3_n; BCD2_c <= BCD2_n;
      BCD1_c <= BCD1_n; BCD0_c <= BCD0_n;
      index_c <= index_n; int_rg_c <= int_rg_n;
      if (get_outputs = '1') then
        BCD0 <= std_logic_vector(BCD0_n); BCD1 <= std_logic_vector(BCD1_n);
        BCD2 <= std_logic_vector(BCD2_n); BCD3 <= std_logic_vector(BCD3_n);
        BCD4 <= std_logic_vector(BCD4_n);
      end if;
    end if;
  end if;
end process;

process (c_s, BCD4_c, BCD3_c, BCD2_c, BCD1_c, BCD0_c, BCD4_tmp, BCD3_tmp, BCD2_tmp, BCD1_tmp, BCD0_tmp,
  binary, int_rg_c, index_c, index_n, request)
begin
  get_outputs <= '0';
  n_s <= c_s; BCD4_n <= BCD4_c; BCD3_n <= BCD3_c; BCD2_n <= BCD2_c;
  BCD1_n <= BCD1_c; BCD0_n <= BCD0_c; index_n <= index_c; int_rg_n <= int_rg_c;
  ready <= '0';

  case c_s is
    when idle =>
      n_s <= op; ready <= '1'; int_rg_n <= binary; index_n <= "10000";
      if request /= '1' then
        n_s <= idle; -- transition to the op state is
      end if;
    when op =>
      ready <= '0';
      int_rg_n <= int_rg_c(size_of_data_to_convert-2 downto 0) & '0';
      BCD0_n <= BCD0_tmp(2 downto 0) & int_rg_c(size_of_data_to_convert-1);
      BCD1_n <= BCD1_tmp(2 downto 0) & BCD0_tmp(3);
      BCD2_n <= BCD2_tmp(2 downto 0) & BCD1_tmp(3);
  end case;
end process;
The module BinToBCD16 operates slightly different comparing with the module BinToBCD8. Now the ready signal is active in the idle state (\(\text{ready} = '1'\)) and the module waits for a request for a new conversion. As soon as the signal request becomes active, new data item is taken and a new conversion will be done (see Fig. B.1c). To use the module BinToBCD16 in the entity TopForInteractingWitIPCores (see Sect. 4.1) the following small change can be done:

1. New signals have to be declared: \(\text{signal request, ready: std_logic;}\)
2. The request has to be generated, for example: \(\text{request} <= \text{ready and BTNR;}\)
3. Mapping in the \text{port map} is done as follows:

\[
\text{port map (clk, reset, open, To_BCD, '1', BCD4, BCD3, BCD2, BCD1, BCD0);}
\]

The following changes can be done in the entity TopForInteractingWitIPCores that enable the request signal to be involved:

1. New signals have to be declared: \(\text{signal request, ready: std_logic;}\)
2. The request has to be generated, for example: \(\text{request} <= \text{ready and BTNR;}\)
3. Mapping in the \text{port map} is done as follows:

\[
\text{port map (clk, reset, ready, To_BCD, request, BCD4, BCD3, BCD2, BCD1, BCD0);}
\]

Now the conversion will be done on the request from the onboard button BTNR.

\text{Clock Divider (clock_divider)}

A \text{clock divider} permits system clock to be divided by \(2^{\text{how_fast} + 1}\). The module with the reset signal is the following (from comments it is clear that reset can be removed if required):

\[
\text{library IEEE;}
\text{use IEEE.STD_LOGIC_1164.all;}
\text{use IEEE.STD_LOGIC_UNSIGNED.all;}
\text{entity clock_divider is}
\]

BCD3_n <= BCD3_tmp(2 downto 0) & BCD2_tmp(3);
BCD4_n <= BCD4_tmp(2 downto 0) & BCD3_tmp(3);
index_n <= index_c - 1;
if \(\text{index_n} = 0\) then \(n_s <= \text{done; get_outputs <= '1';}\)
end if;
when done => \(n_s <= \text{idle};\)
\(\text{BCD4_n <= (others} => '0');\)
\(\text{BCD3_n <= (others} => '0');\)
\(\text{BCD2_n <= (others} => '0');\)
\(\text{BCD1_n <= (others} => '0');\)
\(\text{BCD0_n <= (others} => '0');\)
\(\text{ready <= '1'};\)
end case;
end process;

BCD0_tmp <= BCD0_c + 3 when BCD0_c > 4 else BCD0_c;
BCD1_tmp <= BCD1_c + 3 when BCD1_c > 4 else BCD1_c;
BCD2_tmp <= BCD2_c + 3 when BCD2_c > 4 else BCD2_c;
BCD3_tmp <= BCD3_c + 3 when BCD3_c > 4 else BCD3_c;
BCD4_tmp <= BCD4_c(2 downto 0) + 3 when BCD4_c > 4 else BCD4_c(2 downto 0);
end Behavioral;
DSP-Based Hamming Weight Counter/Comparator for N = 32
(Test_HW32)

The following VHDL code is a complete description of the Hamming weight counter/comparator (with a fixed threshold) for N = 32:

```
library IEEE; -- The top-level module to test the 32-bit Hamming weight counter/comparator
use IEEE.STD_LOGIC_1164.all; -- this circuit occupies 0 logical slices and 2 DSP48 slices
use IEEE.STD_LOGIC_UNSIGNED.all; -- the maximum combinational path delay is 3.9 ns

entity Test_HW32 is -- the project was tested in the Nexys-4 board
port ( Sw   : in std_logic_vector (15 downto 0); -- Nexys-4 onboard switches
       led     : out std_logic_vector (5 downto 0); -- Nexys-4 onboard LEDs
       in16bit : in std_logic_vector(15 downto 0); -- signals from Nexys-4 PMod connectors
       led_comp : out std_logic); -- the result of comparison
end Test_HW32;

architecture Behavioral of Test_HW32 is

signal threshold   : std_logic_vector(5 downto 0); -- not "011010" + 1; -- this value of threshold was taken just for test
signal HW1, HW2   : std_logic_vector(4 downto 0);  
signal remaining_inputs1 : std_logic_vector(11 downto 0);  
signal remaining_inputs2 : std_logic_vector(11 downto 0);  
signal remaining_outputs1 : std_logic_vector(5 downto 0);  
signal remaining_outputs2 : std_logic_vector(5 downto 0);  

begin

threshold <= not "011010" + 1;  
remaining_inputs1 <= '0' & HW1 & '0' & HW2;  
remaining_inputs2 <= remaining_outputs1 & threshold;  
led     <= remaining_outputs1;

HWCC16_1: entity work.HW_counter_comparator_16bit -- see the code below
```

Appendix B: Coding Examples
Figure B.2a demonstrates a possible interface with the circuit. Clearly, the value of the threshold can also be taken from outside and any mode from Fig. 3.30 can easily be added with just one additional look-up table (see Fig. B.2b). Xilinx primitive CFGLUT5 [1] is a runtime, dynamically reconfigurable 5-input LUT that enables the implemented logical function (configuration of the LUT) to be changed during the circuit operation. Hence, the bounds/thresholds can be modified during run-time if required.
The following VHDL code is a complete description of the Hamming weight counter/comparator (with a fixed threshold) for $N = 64$:

```vhdl
library IEEE; -- the top-level module to test the 64-bit Hamming weight counter/comparator
use IEEE.STD_LOGIC_ARITH.all; -- the project was tested in the Nexys-4 board
use IEEE.STD_LOGIC_UNSIGNED.all; -- the maximum combinational path delay is 6.1 ns

entity Test_HW64 is -- this projects takes 64-bit vectors from FPGA pins
  port (Sw : in std_logic_vector(15 downto 0); -- part of the vector from switches
         led : out std_logic_vector(6 downto 0); -- Nexys-4 onboard LEDs
         in48bit : in std_logic_vector(47 downto 0); -- the remainder
         led_comp : out std_logic); -- the result of comparison
end Test_HW64;

architecture Behavioral of Test_HW64 is -- this circuit occupies 1 logical slice and 4 DSP48 slices

signal threshold : std_logic_vector(6 downto 0);
signal HW1, HW3 : std_logic_vector(4 downto 0);
signal HW2, HW4 : std_logic_vector(5 downto 0);
signal remaining_inputs1 : std_logic_vector(11 downto 0);
signal remaining_inputs2 : std_logic_vector(11 downto 0);
signal remaining_inputs3 : std_logic_vector(11 downto 0);
signal remaining_inputs4 : std_logic_vector(11 downto 0);

begin
  threshold <= not "0110010" + 1; -- this value of threshold was taken just for test

  remaining_inputs1 <= '0' & HW1 & HW2;
  remaining_inputs2 <= remaining_inputs1 & remaining_inputs3;
  remaining_inputs3 <= '0' & HW3 & HW4;
  remaining_inputs4 <= remaining_outputs2(5 downto 0) & threshold(5 downto 0);

  led_comp <= comp or remaining_outputs2(6);
  led <= remaining_outputs2;

  HWCC16_1: entity work.HW_counter_comparator_16bit -- see the code above
    port map (Sw, HW1, remaining_inputs1, remaining_outputs1, open);
  HWCC16_2: entity work.HW_counter_comparator_16bit_m
    port map (in48bit(15 downto 0), HW2, remaining_inputs2, remaining_outputs2, open);
  HWCC16_3: entity work.HW_counter_comparator_16bit -- see the code above
    port map (in48bit(31 downto 16), HW3, remaining_inputs3, remaining_outputs3, open);
  HWCC16_4: entity work.HW_counter_comparator_16bit_m -- the code above is slightly changed
    port map (in48bit(47 downto 32), HW4, remaining_inputs4, open, comp);

end Behavioral;
```

**DSP-Based Hamming Weight Counter/Comparator for $N = 64$**

(Test_HW64)

The following VHDL code is a complete description of the Hamming weight counter/comparator (with a fixed threshold) for $N = 64$:
The next module may be helpful when only the onboard switches for the Nexys-4 board are used to supply 64-bit binary vectors as a sequence of four 16-bit fragments from the 16 available switches. Each fragment is saved when the associated onboard button is pressed (BTNL for in_16bit1, BTNC for in_16bit2, BTNR for in_16bit3, and BTND for in_16bit4).

```vhdl
library IEEE; -- the top-level module to test the 64-bit Hamming weight counter/comparator
use IEEE.STD_LOGIC_ARITH.all; -- the project was tested in the Nexys-4 board
use IEEE.STD_LOGIC_UNSIGNED.all;

entity Test_HW64 is
  port ( clk : in std_logic; -- for reading and saving 16-bit segments of 64-bit vector
          Sw : in std_logic_vector(15 downto 0); -- segments of the vector from Nexys-4 switches
          led : out std_logic_vector(6 downto 0); -- Nexys-4 onboard LEDs
          BTNL, BTNC, BTNR, BTND : in std_logic; -- Nexys-4 onboard buttons
          led_comp : out std_logic); -- the result of comparison
end Test_HW64;

architecture Behavioral of Test_HW64 is
  -- the same signal declarations as in the previous example
signal in_16bit1, in_16bit2, in_16bit3, in_16bit4 : std_logic_vector(15 downto 0);
begin
  process(clk)
  begin -- reading and saving 16-bit fragments of 64-bit vector
    if rising_edge(clk) then
      if BTNL = '1' then -- saving the first 16 bits if BTNL is pressed
        in_16bit1 <= Sw;
      elsif BTNC = '1' then -- saving the second 16 bits if BTNC is pressed
        in_16bit2 <= Sw;
      elsif BTNR = '1' then -- saving the third 16 bits if BTNR is pressed
        in_16bit3 <= Sw;
      elsif BTND = '1' then -- saving the forth 16 bits if BTND is pressed
        in_16bit4 <= Sw;
      end if;
    end if;
  end process;
end Behavioral;
```
The network uses two components EvenOddMerge8Sort described in Sect. 3.4.1.

```vhdl
library IEEE; -- the project was tested in the Atlys board involving interactions with a host PC
use IEEE.STD_LOGIC_1164.all; -- interactions with a host PC are not shown here
use work.set_of_data_items.all; -- see the given below user-defined package
entity EvenOddMerge16Sort is -- this circuit occupies 187 logical slices (including interactions)
  generic (M : integer := 4; -- generic size of data items
          N : integer := 16); -- generic number of data items (cannot be changed for this project)
  port (input_1items : in set_of_8items;
         input_2items : in set_of_8items;
         sorted : out set_of_16items);
end EvenOddMerge16Sort;
architecture Structural of EvenOddMerge16Sort is
begin
  sort8items1: entity work.EvenOddMerge8Sort
                generic map(M => M, N => 8); -- even-odd merge sorter for 8 items
  port map(input_1items, sorted1);
  sort8items2: entity work.EvenOddMerge8Sort
                generic map(M => M, N => 8); -- even-odd merge sorter for 8 items
  port map(input_2items, sorted2);
  stage4: for i in N/2-1 downto 0 generate
    group1stage4: entity work.Comparator
                  generic map(M => M);
    port map(sorted1(i), sorted2(i), out1_in2(i), out1_in2(i+8));
    step1stage4: if (i >= 4) generate
      group2stage4: entity work.Comparator
                     generic map(M => M);
      port map(out1_in2(i), out1_in2(i+4), out2_in3(i), out2_in3(i+4));
    end generate;
    step2stage4: if (i < 4) generate
      out2_in3(i) <= out1_in2(i);
      out2_in3(i+12) <= out1_in2(i+12);
    end generate;
    step3stage4: if (i < 3) generate
      incide_stage4: for j in 0 to N/8-1 generate
```
The following package set_of_data_items has been used:

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

package set_of_data_items is
constant N  : integer := 8;
constant M  : integer := 4; -- for different values of M this constant needs to be changed

type set_of_8items is array (N-1 downto 0) of std_logic_vector (M-1 downto 0);

end set_of_data_items;

end package body set_of_data_items;

library IEEE;
-- the project was tested for the Nexys-4 board and occupies 3 slices
use IEEE.STD_LOGIC_1164.all;
-- maximum combinational path delay is 2.5 ns

entity HammingWeightComparator is
port ( Sw  : in std_logic_vector (14 downto 0); -- input 15-bit vector
       LedC : out std_logic); -- the result of comparison
end entity HammingWeightComparator;

architecture Behavioral of HammingWeightComparator is
signal Upper_bits, Middle_bits, Bottom_bits : std_logic_vector(2 downto 0);
signal ToLast : std_logic_vector(5 downto 0);
signal comp : std_logic;

begin

end architecture Behavioral;
```

**Hamming Weight Comparator for** \(N = 15\) \((\text{HammingWeightComparator})\)

The following VHDL code is a complete synthesizable specification of the Hamming weight comparator in Fig. 3.31a (for any module below the final comparison circuits from Fig. 3.25 can also be used):

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

entity HammingWeightComparator is
port ( Sw  : in std_logic_vector (14 downto 0); -- input 15-bit vector
       LedC : out std_logic); -- the result of comparison
end entity HammingWeightComparator;

architecture Behavioral of HammingWeightComparator is
signal Upper_bits, Middle_bits, Bottom_bits : std_logic_vector(2 downto 0);
signal ToLast : std_logic_vector(5 downto 0);
signal comp : std_logic;

begin

end architecture Behavioral;
```
The following code is for the component LUT_5to3:

```vhdl
begin

LUT_5_3_upper : entity work.LUT_5to3
    port map(Sw(14 downto 10), Upper_bits);

LUT_5_3_middle : entity work.LUT_5to3
    port map(Sw(9 downto 5), Middle_bits);

LUT_5_3_bottom : entity work.LUT_5to3
    port map(Sw(4 downto 0), Bottom_bits);

LUT6_1_comp: entity work.LUT6_1
    port map (ToLast, LedC);

end Behavioral;

The following code is for the final comparator in Fig. 3.29a (LUT6_1):
Table B.1 below explains how to configure the LUT $LUT_{6\_1}$ for the final comparator in Fig. 3.29a.

The $SixIn$ column shows input vectors that are represented by three 2-bit sub-vectors. The most significant sub-vector has the weight 4, the middle sub-vector—the weight 2 and the least significant sub-vector—the weight 1. Thus, the code 000101 has the value $0 \times 4 + 1 \times 2 + 1 \times 1 = 3$ and this is the settled bound 3. The next code 000110 has the value $0 \times 4 + 1 \times 2 + 2 \times 1 = 4$ and this is the value above the bound 3. All the subsequent values until the number 011001 are within the settled bounds (more than 3 and less than 10). The number 011010 has the first value $1 \times 4 + 2 \times 2 + 2 \times 1 = 10$ outside the bounds. Hexadecimal numbers from Table B.1 are used to configure the LUT. They have to be taken from the bottom right part to the upper left part giving the following constant: $FFFFFFF800003F$ that is used for the INIT statement: $INIT \Rightarrow X''8000003F''$.

The circuit has been tested in the Nexys-4 board. Input vectors were taken from 15 onboard switches: 14, 13, …, 0 (switch 15 was not used). The result of comparison is shown on LED 0.

**Hamming Weight Counter for $N = 31$ and Comparator for $N = 32$ ($HW31\_HWC32$)**

VHDL code below can be used directly for the circuit in Fig. 3.32 which counts the Hamming weight of any input binary vector for $N = 31$ (i.e. for $B = \{B_0, \ldots, B_{30}\}$)
### Table B.1 Configuring the LUT6_1 for the final comparator (see also Fig. 3.29a)

<table>
<thead>
<tr>
<th>SixIn</th>
<th>Comp</th>
<th>SixIn</th>
<th>Comp</th>
<th>SixIn</th>
<th>Comp</th>
<th>SixIn</th>
<th>Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>1 F</td>
<td>010000</td>
<td>0 0</td>
<td>100000</td>
<td>1 F</td>
<td>110000</td>
<td>1 F</td>
</tr>
<tr>
<td>000001</td>
<td>1</td>
<td>010001</td>
<td>0</td>
<td>100001</td>
<td>1</td>
<td>110001</td>
<td>1</td>
</tr>
<tr>
<td>000010</td>
<td>1</td>
<td>010010</td>
<td>0</td>
<td>100010</td>
<td>1</td>
<td>110010</td>
<td>1</td>
</tr>
<tr>
<td>000011</td>
<td>1</td>
<td>010011</td>
<td>0</td>
<td>100011</td>
<td>1</td>
<td>110011</td>
<td>1</td>
</tr>
<tr>
<td>000100</td>
<td>1 3</td>
<td>010100</td>
<td>0 0</td>
<td>100100</td>
<td>1 F</td>
<td>110100</td>
<td>1 F</td>
</tr>
<tr>
<td>000101</td>
<td>1</td>
<td>010101</td>
<td>0</td>
<td>100101</td>
<td>1</td>
<td>110101</td>
<td>1</td>
</tr>
<tr>
<td>000110</td>
<td>0</td>
<td>010110</td>
<td>0</td>
<td>100110</td>
<td>1</td>
<td>110110</td>
<td>1</td>
</tr>
<tr>
<td>000111</td>
<td>0</td>
<td>010111</td>
<td>0</td>
<td>100111</td>
<td>1</td>
<td>110111</td>
<td>1</td>
</tr>
<tr>
<td>001000</td>
<td>0 0</td>
<td>011000</td>
<td>0 C</td>
<td>101000</td>
<td>1 F</td>
<td>111000</td>
<td>1 F</td>
</tr>
<tr>
<td>001001</td>
<td>0</td>
<td>011001</td>
<td>0</td>
<td>101001</td>
<td>1</td>
<td>111001</td>
<td>1</td>
</tr>
<tr>
<td>001010</td>
<td>0</td>
<td>011010</td>
<td>1</td>
<td>101010</td>
<td>1</td>
<td>111010</td>
<td>1</td>
</tr>
<tr>
<td>001011</td>
<td>0</td>
<td>011011</td>
<td>1</td>
<td>101011</td>
<td>1</td>
<td>111011</td>
<td>1</td>
</tr>
<tr>
<td>001100</td>
<td>0 0</td>
<td>011100</td>
<td>1 F</td>
<td>101100</td>
<td>1 F</td>
<td>111100</td>
<td>1 F</td>
</tr>
<tr>
<td>001101</td>
<td>0</td>
<td>011101</td>
<td>1</td>
<td>101101</td>
<td>1</td>
<td>111101</td>
<td>1</td>
</tr>
<tr>
<td>001110</td>
<td>0</td>
<td>011110</td>
<td>1</td>
<td>101110</td>
<td>1</td>
<td>111110</td>
<td>1</td>
</tr>
<tr>
<td>001111</td>
<td>0</td>
<td>011111</td>
<td>1</td>
<td>101111</td>
<td>1</td>
<td>111111</td>
<td>1</td>
</tr>
</tbody>
</table>
and provides comparison of any binary vector for \( N = 32 \) with fixed bounds (any set of bounds from Fig. 3.30 may be chosen).

```
library IEEE;  -- the project was tested for the Nexys-4 board and occupies 14 logical slices
use IEEE.STD_LOGIC_1164.all;  -- the maximum combinational path delay is 4.4 ns
use IEEE.STD_LOGIC_ARITH.all;  -- constant compare configured for two bounds: 1) 10\( \leq \)weight \(-\) LedC is OFF; 2) 10<weight<20 - LedC is ON; 3) 20\( \leq \)weight<30 - LedC is OFF;
use IEEE.STD_LOGIC_UNSIGNED.all;  -- and 30\( \leq \)weight - LedC is ON;

entity HW31_HWC32 is  -- the names of used components are the same as in Fig. 3.32
  port (Data_in : in std_logic_vector (31 downto 0);  -- 32-bit binary vector (Vector31_in in Fig. 4.6)
        led : out std_logic_vector (4 downto 0);
        LedC : out std_logic);  
end HW31_HWC32;

architecture Mixed of HW31_HWC32 is

  signal HW15_1: std_logic_vector(3 downto 0);  -- the Hamming weight for Data_in(14 downto 0)
  signal HW15_2: std_logic_vector(3 downto 0);  -- the Hamming weight for Data_in(29 downto 15)
  signal LUT5_3 : std_logic_vector(3 downto 0);  -- LUT5_3 in Fig. 3.32 (see block D)
  signal LUT4_3 : std_logic_vector(3 downto 0);  -- LUT4_3 in Fig. 3.32 (see block C)
  signal Out5_3 : std_logic_vector(2 downto 0);  -- LUT5_3 in Fig. 3.32 (see block D)
  signal Out4_3 : std_logic_vector(2 downto 0);  -- LUT4_3 in Fig. 3.32 (see block C)
  constant compare : std_logic_vector(127 downto 0) :=  
                      "X"FEE0000077FFCC000FCC0000FFFF880000";

  constant bit0 : std_logic_vector(63 downto 0) := X"AAAAAAAAAAAAAAAA";
  constant bit1 : std_logic_vector(63 downto 0) := X"CCCCCCCCCCCCCCCC";
  constant bit2 : std_logic_vector(63 downto 0) := X"0FF00FF00FF00FF00";
  constant bit3 : std_logic_vector(63 downto 0) := X"0FFFF0000FFFF0000";
  constant bit4 : std_logic_vector(63 downto 0) := X"0FFFFFFFF00000000";

begin

  LUT_based1: entity work.HW15Counter  -- see block B in Fig. 3.32  
        port map (Data_in(14 downto 0), HW15_1);

  LUT_based2: entity work.HW15Counter  -- see block A in Fig. 3.32  
        port map (Data_in(29 downto 15), HW15_2);

  LUT4_3 <= HW15_1(3 downto 2) & HW15_2(3 downto 2);  -- see LUT4_3 lines in Fig. 3.32
  LUT5_3 <= HW15_1(1 downto 0) & HW15_2(1 downto 0);  -- see LUT5_3 lines in Fig. 3.32

  LUT_4_3: entity work.LUT4to3  -- see block C in Fig. 3.32  
        port map(LUT4_3, Out4_3);

  LUT_5_3 : entity work.LUT5to3  -- see block D in Fig. 3.32  
        port map(LUT5_3, Data_in(30), Out5_3);

  LedC <= compare(conv_integer(Data_in(31) & Out4_3 & Out5_3));  -- the result of comparison (block E)

  led <= bit4(conv_integer(Out4_3 & Out5_3)) & bit3(conv_integer(Out4_3 & Out5_3)) &  
        bit2(conv_integer(Out4_3 & Out5_3)) & bit1(conv_integer(Out4_3 & Out5_3)) &  
        bit0(conv_integer(Out4_3 & Out5_3));  -- computation of Hamming weight is not shown in Fig. 3.32

end Mixed;
```
Table B.2 explains how the constants compare, bit4, bit3, bit2, bit1, bit0 have been prepared.

The result of the comparison is changed twice on the left-hand side of Table B.2. Let us consider the first change: 001110 (10) and 001111 (11). The values in parenthesis (in the SixIn column) indicate the decimal numbers corresponding to the neighboring code. For the vector 001 110 the decimal number is formed as 110 \times 4^{10} + 6^{10} = 1010 (see also Fig. 3.32). The value in parenthesis for the column Hamming weight/comparator indicates the result of comparison. It is equal to 0 for (10_{10}) and it is equal to 1 for (11_{10}). For the second vector 010 010 (10)

<table>
<thead>
<tr>
<th>SixIn</th>
<th>Hamming weight/comparator</th>
<th>SixIn</th>
<th>Hamming weight/comparator</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 (0)</td>
<td>00000 (0)</td>
<td>000000 (16)</td>
<td>10000 (1)</td>
</tr>
<tr>
<td>000001 (1)</td>
<td>00001 (0)</td>
<td>100001 (17)</td>
<td>100001 (1)</td>
</tr>
<tr>
<td>000010 (2)</td>
<td>00010 (0)</td>
<td>100010 (18)</td>
<td>10010 (1)</td>
</tr>
<tr>
<td>000011 (3)</td>
<td>00111 (0)</td>
<td>100111 (19)</td>
<td>10111 (1)</td>
</tr>
<tr>
<td>000100 (4)</td>
<td>01000 (0)</td>
<td>00FCA</td>
<td>10100 (0)</td>
</tr>
<tr>
<td>000101 (5)</td>
<td>01010 (0)</td>
<td>101010 (22)</td>
<td>10110 (0)</td>
</tr>
<tr>
<td>000110 (6)</td>
<td>01110 (0)</td>
<td>101111 (23)</td>
<td>10111 (0)</td>
</tr>
<tr>
<td>000111 (7)</td>
<td>01111 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001000 (8)</td>
<td>00000 (0)</td>
<td>00FCA</td>
<td>10100 (0)</td>
</tr>
<tr>
<td>001001 (9)</td>
<td>00011 (0)</td>
<td>101011 (23)</td>
<td>11011 (0)</td>
</tr>
<tr>
<td>001010 (10)</td>
<td>00101 (0)</td>
<td>101101 (24)</td>
<td>11010 (0)</td>
</tr>
<tr>
<td>001011 (11)</td>
<td>00111 (0)</td>
<td>101110 (25)</td>
<td>11010 (0)</td>
</tr>
<tr>
<td>001100 (12)</td>
<td>01000 (0)</td>
<td>8 0F0CA</td>
<td>11000 (0)</td>
</tr>
<tr>
<td>001101 (13)</td>
<td>01001 (0)</td>
<td>11001 (0)</td>
<td>11010 (0)</td>
</tr>
<tr>
<td>001110 (14)</td>
<td>01010 (0)</td>
<td>11010 (0)</td>
<td>11010 (0)</td>
</tr>
<tr>
<td>001111 (15)</td>
<td>01011 (0)</td>
<td>11011 (0)</td>
<td></td>
</tr>
<tr>
<td>010000 (16)</td>
<td>10000 (0)</td>
<td>F 0FFCA</td>
<td>11100 (0)</td>
</tr>
<tr>
<td>010001 (17)</td>
<td>10001 (0)</td>
<td>11101 (0)</td>
<td>11110 (1)</td>
</tr>
<tr>
<td>010010 (18)</td>
<td>10010 (0)</td>
<td>11110 (31)</td>
<td>11111 (1)</td>
</tr>
<tr>
<td>010011 (19)</td>
<td>10011 (0)</td>
<td>11111 (35)</td>
<td>100011 (1)</td>
</tr>
<tr>
<td>010100 (20)</td>
<td>10100 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010101 (21)</td>
<td>10101 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010110 (22)</td>
<td>10110 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010111 (23)</td>
<td>10111 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011000 (24)</td>
<td>11000 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011001 (25)</td>
<td>11001 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011010 (26)</td>
<td>11010 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011011 (27)</td>
<td>11011 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 (28)</td>
<td>11100 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011101 (29)</td>
<td>11101 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011110 (30)</td>
<td>11110 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 (31)</td>
<td>11111 (0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011000 (32)</td>
<td>100000 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011001 (33)</td>
<td>100100 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011010 (34)</td>
<td>101000 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011011 (35)</td>
<td>101100 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011100 (36)</td>
<td>110000 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011101 (37)</td>
<td>110100 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011110 (38)</td>
<td>111000 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011111 (39)</td>
<td>111100 (1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.2: Preparing the constants for the module HW31_HWC32
the value \((10_{10})\) is formed as \(2_{10} \times 4_{10} + 2_{10} = 10_{10}\) (see also Fig. 3.32). Additional explanations are given in Fig. B.3. The upper one-digit hexadecimal numbers are used to configure the comparator and the lower five-digits hexadecimal numbers—to configure the counter. Figure B.3 explains how the constants have been prepared. This figure depicts the bottom right part of the Table B.2. Hexadecimal numbers for forming the comparison result are produced as shown in Fig. B.3. Thus, the 16-digit constant \(FCC0000FFFF88000\) is defined. The most significant 16-digit part of the constant \((FEE00007FFCC000)\) is formed using the same technique but considering also the most significant bit \(Data_{in}(31)\) (this part is not shown in Table B.2 and in Fig. B.3). Hexadecimal constants for the Hamming weight are built similarly, but now five columns with five-digit hexadecimal numbers are used (the rightmost column for \(bit0\) and the leftmost column for \(bit4\); the remaining constants are built from digits in the middle).

Thus, the following hexadecimal values are used: 000CA (the bottom right part of Table B.2), FFFCA, FFFCA, F00CA, F00CA, F00CA, F00CA, F00CA, 0FFCA, 0FFCA, 0FFCA, 0FFCA, 0FFCA, 000CA (the upper left part of Table B.2). Such constants have been defined only for five least significant digits in the column Hamming weight/comparator (because the Hamming weight is computed only for 31-bit vectors and five binary digits are sufficient). Now the constant is prepared for each hexadecimal digit. For the most significant digit the constant \(bit4\) is \(0FFFFFFFF0000000\) (composed of the most significant digits in each hexadecimal number). The next constant for \(bit3\) is: \(0FFFFFFFF0000000\), etc.

The component HW15Counter is very similar to the described above component HammingWeightComparator. The only difference is in computing the Hamming weight of a 15-bit input vector instead of the result of comparison. VHDL code below is used for the component HW15Counter that computes the Hamming weight:

---

Fig. B.3 Preparing constant values
The component LUT_5to3 is the same as in the described above entity HammingWeightComparator. The component LUT6_4 has the following VHDL code:

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

entity HW15Counter is
  port ( Data_in : in std_logic_vector (14 downto 0); -- input binary vector
         HW15 : out std_logic_vector(3 downto 0)); -- Hamming weight of the input vector
end HW15Counter;

architecture Structural of HW15Counter is -- it is very similar to the entity HammingWeightComparator
signal Upper, Middle, Bottom : std_logic_vector(2 downto 0);
signal ToLast   : std_logic_vector(5 downto 0);
begin
  LUT_5_3_upper : entity work.LUT_5to3
    port map(Data_in(14 downto 10), Upper);
  LUT_5_3_middle : entity work.LUT_5to3
    port map(Data_in(9 downto 5), Middle);
  LUT_5_3_bottom : entity work.LUT_5to3
    port map(Data_in(4 downto 0), Bottom);
  LUT6_4_comp_HW: entity work.LUT6_4
    port map (ToLast, HW15);
  FA_generate: for i in 0 to 2 generate
    FA: entity work.FullAdder -- see entity FullAdder in section 3.7
      port map(Bottom(i), Middle(i), Upper(i), ToLast(2*i), ToLast(2*i+1));
  end generate
end Structural;

The component LUT_5to3 is the same as in the described above entity HammingWeightComparator. The component LUT6_4 has the following VHDL code:

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

library UNISIM; -- for FPGA LUTs
use UNISIM.vcomponents.all;

entity LUT6_4 is
  port ( Data_in    : in std_logic_vector (5 downto 0); -- input binary vector
         Data_out : out std_logic_vector (3 downto 0)); -- Hamming weight for the input vector
end LUT6_4;

architecture Structural of LUT6_4 is
begin
  LUT6_inst2: LUT6
    generic map (INIT => X"003f3f00000c0000") -- LUT Contents
    port map (Data_out(3), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Data_in(4), Data_in(5));
  LUT6_inst3 : LUT6
    generic map (INIT => X"c03f3fc0c03f3fc0") -- LUT Contents
    port map (Data_out(2), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Data_in(4), Data_in(5));
  LUT6_inst4 : LUT6
    generic map (INIT => X"3c3c3c3c3c3c3c3") -- LUT Contents
    port map (Data_out(1), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Data_in(4), Data_in(5));
  LUT6_inst5 : LUT6
    generic map (INIT => X"aaaaaaaaaaaaaaaa") -- LUT Contents
    port map (Data_out(0), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Data_in(4), Data_in(5));
end Structural;
```
The component LUT4to3 has the following VHDL code:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library UNISIM; -- for FPGA LUTs
use UNISIM.vcomponents.all;

entity LUT4to3 is
  port ( Data_in : in std_logic_vector (3 downto 0);
         Data_out : out std_logic_vector (2 downto 0));
end LUT4to3;
architecture Structural of LUT4to3 is
begin
  LUT4_inst1 : LUT4
    generic map (INIT => X"EE80")
    port map (Data_out(2), Data_in(0), Data_in(1), Data_in(2), Data_in(3));
  LUT4_inst2 : LUT4
    generic map (INIT => X"936C")
    port map (Data_out(1), Data_in(0), Data_in(1), Data_in(2), Data_in(3));
  LUT4_inst3 : LUT4
    generic map (INIT => X"5A5A")
    port map (Data_out(0), Data_in(0), Data_in(1), Data_in(2), Data_in(3));
end Structural;
```

The component LUT5to3 (note that this component is not the same as the considered above component LUT_5to3 in the entity HammingWeightComparator) has the following VHDL code:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library UNISIM; -- for FPGA LUTs
use UNISIM.vcomponents.all;

entity LUT5to3 is
  port ( Data_in  : in std_logic_vector (3 downto 0);
         Extra_bit : in std_logic;
         Data_out  : out std_logic_vector (2 downto 0));
end LUT5to3;
architecture Structural of LUT5to3 is
begin
  LUT5_inst1 : LUT5
    generic map (INIT => X"FEC8EE80")
    port map (Data_out(2), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Extra_bit);
  LUT5_inst2 : LUT5
    generic map (INIT => X"C936936C")
    port map (Data_out(1), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Extra_bit);
  LUT5_inst3 : LUT5
    generic map (INIT => X"A5A5A5A5")
    port map (Data_out(0), Data_in(0), Data_in(1), Data_in(2), Data_in(3), Extra_bit);
end Structural;
```
The respective project is ready to be tested and the circuit (with the maximum combinational path delay equal to 4.4 ns) occupies just 14 Artix-7 FPGA slices. It can be used as a Hamming weight counter and comparator. If just one from two such functions (i.e. either counting or comparison) is required then unnecessary fragment can be removed. The project above will also be used as a component of the last example in Appendix B.

**Hamming Weight Counter for \( N = 36 \) (HammingWeightCounter\(36\)bits)**

The following VHDL code is a complete synthesizable specification of the Hamming weight counter in Figs. 3.27 and 3.28 (any final comparison circuit from Fig. 3.25 can be used for the Hamming weight comparator):

```vhdl
library IEEE; -- the project was tested for the Nexys-4 board and occupies 15 slices
use IEEE.STD_LOGIC_1164.all; -- the maximum combinational path delay is 3.5 ns

entity HammingWeightCounter36bits is
  generic (N : integer := 36);
  port (Data_in : in std_logic_vector(N-1 downto 0); -- inputs \( a_0, a_1, \ldots, a_{35} \) in Fig. 3.27
        Data_out : out std_logic_vector(5 downto 0)); -- the Hamming weight in Fig. 3.28
end entity HammingWeightCounter36bits;

architecture Behavioral of HammingWeightCounter36bits is
  type array_of_inputs is array (N/6-1 downto 0) of std_logic_vector(5 downto 0);
  type array_of_outputs is array (N/4-1 downto 0) of std_logic_vector(5 downto 0);
  signal Out18_bits, In6_bits, Res9_bits : std_logic_vector(N-1 downto 0);
  begin
    generate_LUTs_at_level_0: for i in N/6-1 downto 0 generate
      one_slice: entity work.LUT_6to3
        port map(Data_in(6*i+5 downto 6*i), Out18_bits(3*i+2 downto 3*i));
    end generate

    generate_LUTs_at_level_1: for i in N/12-1 downto 0 generate
      one_slice: entity work.LUT_6to3
        port map(In6_bits(i), Res9_bits(3*i+2 downto 3*i));
    end generate

    FinalCircuit: entity work.Final_LUT_based_adders
      port map (Res9_bits(7 downto 0), Res9_bits(8), Data_out);
  end Behavioral;
```

The component Final_LUT_based_adders describes the functionality of the circuit in Fig. 3.28a and it is coded in VHDL as follows (similar but simpler circuit can be built for Fig. 3.28b):
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity Final_LUT_based_adders is -- the mapping is described below by constants
port ( A_3bits_B_3bits_C_2bits: in std_logic_vector(7 downto 0);
       C_last_bit   : in std_logic;  -- C_last_bit is the symbol $\chi_3$ in Fig. 3.28a
       Data_out    : out std_logic_vector(5 downto 0)); -- Hamming weight (N=36)
end Final_LUT_based_adders;

architecture Behavioral of Final_LUT_based_adders is

-- only 3 least significant bits in 4-bit vectors are used below for the INIT statement for $\gamma_\alpha \gamma_\beta \gamma_\chi$ in Fig. 3.28a
constant LUTs1 : for_LUT :=
    (x"0", x"1", x"2", x"3", x"4", x"5", x"6",
     x"2", x"3", x"4", x"5", x"3", x"4", x"5", x"6",
     x"3", x"4", x"5", x"5", x"4", x"5", x"6",
     x"4", x"5", x"5", x"5", x"6", x"7", x"7", x"8");

-- A_3bits/B_3bits/C_2 bits are associated with the symbols $\alpha_\alpha \beta_\beta \beta_\chi$ in Fig. 3.28a
signal A1A2A3, B1B2B3, C1C2C3 : std_logic_vector(2 downto 0);

begin -- the lines below describe the circuit in Fig. 3.28a
    \text{--- \textit{LUT1} is the bottom block in Fig. 3.28a and LUT2 is the upper block in Fig. 3.28a}\n    A1A2A3 <= C_last_bit & A_3bits_B_3bits_C_2bits(7 downto 6); -- signals $\alpha_\alpha$ for the upper block
    B1B2B3 <= A_3bits_B_3bits_C_2bits(5 downto 3); -- signals $\beta_\beta$ (upper block) and $\beta_\beta$ (bottom block)
    C1C2C3 <= A_3bits_B_3bits_C_2bits(2 downto 0); -- signal $\beta_\chi$ (direct output) and $\beta_\chi$ (bottom block)

    O5_3 <= LUTs2(conv_integer(CmClO2O1(3 downto 2) &
                              A1A2A3(2 downto 1) & B1B2B3(2)))(2 downto 0);
    CmClO2O1 <= LUTs1(conv_integer(A1A2A3(0) & B1B2B3(1 downto 0) &
                                   C1C2C3(2 downto 1)));
    Data_out <= O5_3 & CmClO2O1(1 downto 0) & C1C2C3(0); -- concatenation of $(\gamma_\alpha \gamma_\beta \gamma_\chi)$ and $(\gamma_\alpha \gamma_\beta \gamma_\chi)$
end Behavioral;

The respective project is ready to be tested and the circuit (with the maximum combinational path delay equal to 3.5 ns) occupies just 15 Artix-7 FPGA slices. It only counts the Hamming weight of 36-bit vectors. A simple addition enables the same project to be used as a Hamming weight comparator.

Note that we have described many different projects and they may be chosen dependently on available embedded to FPGA components. Indeed, if embedded DSP slices are available then DSP-based projects are perhaps the best. If only logical slices can be used then one of the described here projects may be helpful.

Random Number Generator (RanGen)

The module generates random numbers with generic size width and it has the following VHDL code (for the default value width = 32):
In each clock cycle a new 32-bit pseudorandom number is generated. The size \( width = 32 \) is generic and can easily be changed.

**Segment Decoder (segment_decoder)**

The decoder converts 4-bit binary codes in such a way that the respective digits become visible on 7-segment displays such as those available on the Nexys-4 board. VHDL code for the decoder is given below:

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

entity segment_decoder is -- any one hexadecimal or BCD code can be used as an input
port ( BCD : in std_logic_vector (3 downto 0); -- decoder input
       segments : out std_logic_vector (7 downto 1)); -- decoder output
end segment_decoder;

architecture Behavioral of segment_decoder is
begin -- segment is active when the signal is '0' and passive when the signal is '1'
    segments <= "1000000" when BCD = "0000" else 0
                "1111001" when BCD = "0001" else 1
                "0100100" when BCD = "0010" else 2
                "0110000" when BCD = "0011" else 3
                "0011001" when BCD = "0100" else 4
                "0010010" when BCD = "0101" else 5
                "0000010" when BCD = "0110" else 6
                "1111000" when BCD = "0111" else 7;
end Behavioral;
```

In each clock cycle a new 32-bit pseudorandom number is generated. The size \( width = 32 \) is generic and can easily be changed.
Segment Display Control (EightDisplayControl)

This component controls eight 7-segment displays available on the Nexys-4 board. Functionality of the module is explained in Fig. B.4 and its VHDL code is given below:

```vhdl
library IEEE; -- this code is for 8 7-segment displays available on the Nexys-4 board
use IEEE.STD_LOGIC_1164.all; -- small changes permit the same code to be used for many
use IEEE.STD_LOGIC_UNSIGNED.all; -- prototyping boards, for example, Nexys-2/Nexys-3

entity EightDisplayControl is -- FourDisplayControl for Nexys-2/Nexys-3 can be also based on the code below
  port ( clk  :
         in  std_logic;
         leftL, near_leftL :
         in  std_logic_vector (3 downto 0);
         near_rightL, rightL :
         in  std_logic_vector (3 downto 0);
         leftR, near_leftR :
         in  std_logic_vector (3 downto 0);
         near_rightR, rightR :
         in  std_logic_vector (3 downto 0);
         select_display :
         out std_logic_vector (7 downto 0);
         segments :
         out std_logic_vector (6 downto 0));
end EightDisplayControl;

architecture Behavioral of EightDisplayControl is
  signal Display  : std_logic_vector(2 downto 0);
  signal div     : std_logic_vector(16 downto 0);
  signal convert_me : std_logic_vector(3 downto 0);
begin
  div<= div + 1 when rising_edge(clk);
  Display <= div(16 downto 14);

  process(Display, leftL, near_leftL, near_rightL, rightL, leftR, near_leftR, near_rightR, rightR) begin -- sequential activation of the displays with proper control of the segments of the selected display
    if  Display="111" then select_display <= "11111110"; convert_me <= leftL;
    elsif Display="110" then select_display <= "11111101"; convert_me <= near_leftL;
    elsif Display="101" then select_display <= "11111111"; convert_me <= near_rightL;
    elsif Display="100" then select_display <= "11110111"; convert_me <= rightL;
    elsif Display="011" then select_display <= "11011111"; convert_me <= leftR;
    elsif Display="010" then select_display <= "10111111"; convert_me <= near_leftR;
    elsif Display="001" then select_display <= "10111111"; convert_me <= near_rightR;
    else select_display <= "01111111"; convert_me <= rightR;
  end if; -- the display is active when the corresponding bit in 8-bit vector above is zero
end process;

  decoder : entity work.segment_decoder
        port map (convert_me, segments); -- segment decoder (see above)
end Behavioral;
```
Four-bit codes (either BCD or binary) leftL, near_leftL, near_rightL, rightL, leftR, near_leftR, near_rightR, rightR are associated with different displays shown in Fig. B.4. These codes are sent to the inputs of the segment decoder. Since only one display is active at a time, scanning all the displays enables different numbers to be shown on each of them. Sequential activation of the displays is achieved with the aid of the lines: div<= div + 1 when rising_edge(clk) and Display <= div(16 downto 14). If a converter from binary to BCD codes is also used then binary numbers (see Fig. B.1b) will be displayed in decimal format.

Let us consider an example in which the considered above four components EightDisplayControl, segment_decoder, BinToBCD8, and HW31_HWC32 are used:

```vhdl
library IEEE;  -- the project was tested for the Nexys-4 board and occupies 34 slices
use IEEE.STD_LOGIC_1164.all;  -- the project shows on segment displays the Hamming weight of
use IEEE.STD_LOGIC_UNSIGNED.all;  -- 32-bit input binary vector and the result of comparison

entity HW32_HWC32 is  -- in the experiments 32-bit input binary vector is received from onboard
  -- in the considered above four components
  port ( clk : in std_logic;  -- switches of two Nexys-4 boards connected through PMod
    seg : out std_logic_vector(6 downto 0);  -- segments of onboard displays
    sel_disp : out std_logic_vector(7 downto 0);  -- control of onboard displays
    Data_in : in std_logic_vector(31 downto 0);  -- 32-bit input binary vector
    LedC : out std_logic);  -- the result of comparison (see the entity HW31_HWC32 above)
end HW32_HWC32;
```

Fig. B.4 Functionality of the module EightDisplayControl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity HW16_DISPLAY is -- Nexys-4 circuit occupies 4 logical slices and 1 DSP slice
port (ledL : out std_logic; -- ledL is the leftmost LED
seg : out std_logic_vector(6 downto 0); -- from segment decoder
sel_disp : out std_logic_vector(7 downto 0); -- pins:N6,M6,M3,N5,N2,N4,L1,M1
Sw : in std_logic_vector(15 downto 0)); -- input vector to count the HW
end HW16_DISPLAY;

architecture Mixed of HW16_DISPLAY is
signal HW16 : std_logic_vector(4 downto 0); --represents the HW
begin
-- DSP-based computing of the Hamming weight (HW16) for 16-bit binary vector Sw from Sect. 4.2
HWCC : entity work.Test_HW16 -- combining positional and named associations
port map (Sw,led=>HW16,led_comp=>open);
-- segment display decoder for hexadecimal input numbers
seg_dec : entity work.segment_decoder -- only named association is used
port map (BCD=>HW16(3 downto 0),segments=>seg);

ledL <= HW16(4); -- if HW16 = 16 then LedL is ON otherwise - OFF
sel_disp <= "11111110"; -- only the leftmost display is chosen
end Mixed;
At the beginning let us test the project above in ISE and then make some conversions that enable the project to be synthesized, implemented and tested in Vivado. Firstly, Nexys-4 UCF file has to be converted to XDC file as follows:

1) Run Xilinx PlanAhead software and open the project created in ISE;
2) Run synthesis in the PlanAhead and open synthesized design;
3) Run the following command: write_xdc c:/tmp/Nexys4.xdc from Tcl console of the PlanAhead (note that sub-directory tmp has to be manually created).

Then the following steps have to be done:

4) Create a new RTL Vivado project for FPGA available on the Nexys-4;
5) Copy all VHDL files from ISE project (4 files have to be copied for our project above) and the newly created XDC file to the new Vivado project;
6) Run synthesis, implementation and generate bitstream in Vivado;
7) Open hardware manager in Vivado and program the FPGA of Nexys-4;
8) Test the project in the Nexys-4 board.

To simplify testing the projects in ISE and Vivado all necessary components can be found at http://sweet.ua.pt/skl/Springer2014.html (either in ISE or in Vivado subdirectories). They contain all necessary files that have to be included in ISE/Vivado projects and, thus, only the steps 4)-8) need to be done. Note that if a converted project has XCO files they may need to be upgraded (see Sect. 1.5). If a converted project has COE/TXT files, they have to be either copied to Vivado project or their locations have to be explicitly indicated, for instance:

```vhdl
signal array_name : my_array := read_array("c:/tmp/data.txt");
```

Additional VHDL examples of different reusable blocks are available in [2,3]. Document [4] describes details about migration of ISE projects to Vivado projects.

References

# Index

| A | Adept software (Digilent), 5 |
|   | Arithmetical circuit, 95, 144 |
|   | with accumulators, 99 |
|   | Automaton |
|   | operational, 347 |
|   | of transitions, 347 |

| B | Barrel shifter, 96 |
|   | Binary-coded decimal (BCD), 145 |
|   | Block of |
|   | code transformer, 267 |
|   | input memory functions, 259 |
|   | microoperations, 260 |
|   | operations of transitions, 346 |
|   | replacement of logical conditions, 291 |
|   | state transformer, 262 |

| C | Boolean function, 89 |
|   | operation, 345 |

| Combinational circuit, 89, 101 |
| Communication overhead, 35 |
| Comparator, 95 |
| Computer-aided design system, 4 |
| Configurable logic block, 8 |
| Consecutive ones, 137 |
| Constraints files |
| UCF, 27, 428 |
| XDC, 27, 428 |
| Counting network, 127 |
| components, 127 |
| data-independent segment, 127 |
| mapping onto LUTs, 129 |
| Counter, 60, 83, 98 |

| D | Data-independent segment, 127 |
|   | Decomposition |
|   | functional, 260 |
|   | structural, 260 |
|   | Decoder, 93 |
|   | for segment displays, 93 |
|   | Design |
|   | hierarchical, 7 |
|   | steps, 25 |
|   | summary report, 7 |
|   | tools, 21 |
|   | Digital signal processing |
|   | slice, 4, 17, 153 |
|   | bitwise operations, 153 |
|   | splitting, 154 |
|   | Display controller, 146 |

| E | Embedded memory block, 264 |

---

V. Sklyarov et al., *Synthesis and Optimization of FPGA-Based Systems*, Lecture Notes in Electrical Engineering 294, DOI: 10.1007/978-3-319-04708-9, © Springer International Publishing Switzerland 2014
EMBer, 268
Encoder, 92

F
Feedback register, 114, 118, 121, 138
Field-programmable gate array (FPGA), 3
  bitstream, 5, 21
  characteristics, 29
  clock, 20
    management, 20, 152
    skew, 20
  configurable logic block, 4, 8
  DSP slice, 4, 17, 153
    bitwise operations, 153
    splitting, 154
  embedded blocks, 13
  flip-flop, 4
  interactions, 34
    through Digilent Adept virtual window, 35
    with higher-level systems, 34, 158
  look-up table, 4, 8
  memory, 4, 11, 13
    coefficient (COE) file, 15
  FIFO, 147
  single port, 14
  prototyping, 5, 25
  reset, 20
  slice, 4, 8
Finite state machine, 100
  clocking
    one edge, 103, 208, 212, 216
    two edges, 212, 214, 215, 218, 223
  hierarchical, 111, 210
    implementation of software programs, 239
    intelligent control, 254
    optimization, 245
    stacks, 242
    synthesis, 222
      with explicit modules, 212, 222
      template, 213
      with implicit modules, 220, 229
      template, 220
  Mealy, 101
  Moore, 101
  parallel, 230
  reconfigurable, 254
  with datapath, 102
Frequent item computation, 138
Full adder, 23, 47, 124, 132
Function of transition, 348

G
Generalized formula of transitions, 268
Graph-scheme of algorithm, 269
Greatest common divisor
  iterative implementation, 102, 208, 213, 231
  recursive implementation, 214

H
Half adder, 23, 47, 127
Hamming weight, 57, 64, 73, 131, 149, 155, 200
  comparison, 64, 124, 130
  counter
    DSP-based, 155
    LUT-based, 130
    network-based, 127
    Parallel, 123
    simplest counter (SHWC), 130
Hardware description language, 7
Hierarchical graph-scheme, 207

I
Integrated software environment, 5
  ISE, 26, 428
  Vivado, 26, 428
Intellectual property cores, 143
Intersection operation, 90
Interactions with higher-level systems, 33, 158
  through Digilent parallel port interface, 159, 181
  timing diagrams, 160
  through UART interface, 167, 188
  types, 110
IOExpansion component (from Digilent), 28, 37, 78

K
Karnaugh map, 275

L
Logic element, 12
Look-up-table, 4, 8
  LUTs and constants, 87
LUTer, 268

M
Macrocell, 263
Matrix covering, 250
Max-min network, 68, 118
  application for sorting, 121
  combinational, 118
  complexity and delays, 119, 122
  iterative, 118
  large-scale data sets, 122
Merging of sorted subsets, 111, 198
  Migration from ISE to Vivado projects, 27, 428
Multiplexer, 94

O
Orthogonality operation, 90

P
Parallel counters, 123
  characteristics and formulae, 124
Pipeline, 114, 199, 249, 252
Processing system, 197
Product term, 263
Programmable logic, 197
Programmable system-on-chip, 197, 254
Prototyping boards
  Atlys, 5, 27, 31
  interaction with PC, 158, 167, 181
  DE2-115, 29, 85
  Nexys-4, 32, 33
  interaction with PC, 167, 188

R
Random number generator, 423
Register, 97
  shift, 10, 61, 98
Return flag, 245

S
Schematic
  editor, 5
Sequential circuit, 97
Set of
  additional variables, 261
  logical conditions, 261
  states, 272
Simulation, 22, 81
  behavioral, 81
Software/hardware co-design, 179
Sorting network
  bitonic merge, 105, 113
  depth, 113
  even-odd merge, 66, 105, 113
example with N=8, 106
example with N=16, 108
example with N = 32, 109
  number of comparators, 109
even-odd transition, 113
  iterative, 113, 180
  enable signal, 114
  number of comparators, 113
  pipeline, 114
  propagation delay, 108, 113
  regularity, 113, 119
Specification
  hierarchical, 205
  parallel, 230
State of FSM
  current, 348
  initial, 272
  internal, 319
  pseudoequivalent, 266
State variable, 267
Structural diagram, 259
Structure table, 265
  reduced, 268
  transformed, 303
Sum-of-products, 264
Synthesis, 268
System of
  Boolean functions, 270
  generalized formulae of transitions, 268
  microoperations, 260

T
Table of
  microoperations, 272
  replacement of logical conditions, 279
Test bench, 22
Threshold, 124
  LUT-based bounds, 129, 133
Tree
  $N$-ary, 111, 247
  traversing, 112

U
User constraints file, 5, 26, 52, 428

V
Vector-based computations, 138
Vertex
  conditional, 265
  operator, 265
VHDL, 43
behavioral specification, 45
block, 69
constructions in alphabetical order, 375
coding examples, 403
constant-based conversions, 87, 94
files, 79, 384
function, 63
  impure, 65
  pure, 65
generic parameter, 70
generate statement, 74
library, 76
limitations (and assumptions) in the book, 34, 50
mixed specification, 46
objects, 51
operators, 53
package, 76
procedure, 66
process, 55
  combinational, 56
  sequential, 59
sensitivity list, 55
signals and variables, 51, 58, 62
simulation, 81
structural specification, 44
types, 50
Virtual window (Digilent Adept), 37