"IMPLEMENTATION OF SPECIFIC CONTROL FUNCTIONS OF POWER ELECTRONIC CONVERTERS WITH PROGRAMMABLE LOGIC DEVICES"

Prof. ILYA GALKIN, Riga Technical University
Contents

Part I. Introduction to Programmable Logic .......................................................... 2
  Genesis of Programmable Logic ...................................................................... 2
  Simple Programmable Logic Devices ............................................................... 6
  Complex Programmable Logic Devices ........................................................... 7
  Field Programmable Gate Arrays ...................................................................... 9
  Design Process .................................................................................................. 10
  Manufacturers of Programmable Logic Devices ............................................ 14

Part II. Basics of VHDL ....................................................................................... 15
  Design Approaches .......................................................................................... 15
  Generalised Structure of VHDL File ................................................................. 15
  Lexical Elements of VHDL ............................................................................... 17
  Data Types ....................................................................................................... 19
  Data objects ..................................................................................................... 26
  VHDL Operators ............................................................................................. 29
  Entity’s Declaration .......................................................................................... 33
  Entity’s Architecture Description ..................................................................... 34

Part III. VHDL programming .............................................................................. 36
  VHDL Tools for Dataflow Modelling (Concurrent Statements) ......................... 36
  VHDL Tools for Structural Modelling (Component Statement) ......................... 40
  VHDL Tools for Behavioural Modelling (Sequential Statements) ....................... 48

Appendix A: Internal structure of Atmel’s SPLD ATF16V8B .................................... 57
Appendix B: Basic Elements of Altera’s MAX3000 CPLDs ..................................... 58
Appendix C: Basic Elements of Altera’s EP2CS FPGAs ......................................... 60
Appendix D: Schematic of CPLD Target Board (MAX3064) .................................... 62
Appendix E: Pinout of FPGA Target Board (EC2PS) ............................................ 63
This material gives a brief introduction to the world of programmable logic devices, as well as basic skills of VHDL programming. It is intended for students who are not familiar in this field. However, basic knowledge of general electronics and digital technique is welcome.

**Part I. Introduction to Programmable Logic**

There are quite many kinds and manufacturers of programmable logic devices. However, it is quite possible to emphasize two main stream of their development. The first one includes arrays of logical gates equipped with reconfigurable connections between them. The other group is based on memory elements that also have programmable connections between them. Let’s discuss programmable logic from this point of view.

**Genesis of Programmable Logic**

Let’s discuss a one-bit full-adder which states are given in Table I. As it is known from the course of basic digital technique it has three inputs (two operands $A$, $B$ and carry from the previous digit $C_{IN}$) and two outputs (result $S$ and carry to the next digit $C_{OUT}$). In fact the full-adder consists of two independent circuits (one is for sum and one more – for carry generation) that have the same inputs.

Table I. State table of a one-bit full-adder

<table>
<thead>
<tr>
<th>$C_{IN}$</th>
<th>$A$</th>
<th>$B$</th>
<th>$S$</th>
<th>$C_{OUT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If the Sum of Products (SOP) approach is utilized then the logical equations, producing $S$ and $C_{OUT}$, are constructed as a logical sum of logical products of all three input variables taken with or without logical negation. Each of these products correspond to a ‘1’ in the state table (therefore if there are 4 ‘1’ in the state table the will be 4 products in the sum). An input variable is inverted if its value in the corresponding line of the state table is ‘0’ and is taken without negation if its value is ‘1’. These rules applied to Table I produce the following equations of the full cum $S$ and carry $C_{OUT}$:

$$ S = \overline{C_{IN}} \cdot \overline{A} \cdot \overline{B} + \overline{C_{IN}} \cdot A \cdot \overline{B} + \overline{C_{IN}} \cdot \overline{A} \cdot B + C_{IN} \cdot A \cdot B, $$  

$$ C_{OUT} = \overline{C_{IN}} \cdot A \cdot B + C_{IN} \cdot \overline{A} \cdot \overline{B} + C_{IN} \cdot A \cdot \overline{B} + C_{IN} \cdot A \cdot B. $$

Logical circuits, corresponding to SOP, contain sequential connection of NOT, AND and OR elements. The circuits that correspond to (1) and (2) are given in Fig. 1. The circuit in Fig. 1-a produces the result, while the circuit in Fig. 1-b – the output carry signal. From these diagrams is seen that both theses circuits have the same logical gates: three inverters, four 3-input OR gates and one 4-input AND gate. The difference between these circuits is in their connections that on an experimental board are usually made mechanically, but could also be done electronically.
a) result (unique connections);
d) result (configured matrix);

b) carry (unique connections);
e) carry (configured matrix);

c) matrix of connections;
f) Equality detector (XNOR);

Fig. 1. Programmable matrix of connections
Let's represent the full-adder as a matrix of vertical and horizontal wires Fig. 1-c. If necessary, nodes of this matrix can connect its horizontal and vertical lines. The configuration of this matrix for $S$ signal of the full-adder result is given in Fig. 1-d (blue dots), while its configuration for the output carry – in Fig. 1–e (red dots). It is also possible to configure this matrix in some other way. For example, one more configuration given in Fig. 1-f represents an equality detector (XNOR gate). In this circuit some gates have unused inputs. Depending on their function they are connected to power supply or ground through a resistor (that is then known as pull-up or pull-down resistor). These default pull-up and pull-down connections are not shown in some further diagrams.

There two basic types of programmable connections one-time programmable and reprogrammable. The first kind is based on either fuse (Fig. 2-a...c) or anti-fuse (Fig. 2-d...f) technologies. It is much cheaper, but provides only one opportunity to configuring. The second kind of configurable connections usually utilise FET switches driven from a memory cell – RAM or ROM (Fig. 2-g...h illustrates basics of this principle). It provides the opportunity of reconfiguring but is more expensive. Nowadays programmable logic devices are controlled from memory and the configuration memory is either SRAM or Flash EEPROM.

![Configurations of programmable connections](image)

a) fuse before programming; b) fuse during programming; c) fuse after programming;

![Additional configurations](image)

d) anti-fuse before programming; e) anti-fuse during programming; f) anti-fuse after programming;

![Additional diagrams](image)

g) connected transistor; h) disconnected transistor;

Fig. 2. Basic principles of programmable connections
The full-adder is a circuit with complete function. Therefore both the result and carry sub-circuits has to be built. This requires two gate arrays shown in Fig. 1-c with connections programmed in the right way. In practice programmable gate arrays have more AND gates and OR gates associated with them (in Fig. 3-a a gate array with 4 AND elements is shown). Then it is possible to obtain several functions in the same device. For example in Fig. 3-b a simple Arithmetically-Logical Unit (ALU) is presented. It includes a full-adder, equality detector and un-equality detector.

An alternative approach to synthesis of combinational logic is memory approach. It considers any combinational logical circuit as memory – look-up table (LUT) in read mode. The Inputs of combinational logic are then considered as address signals, but its output as a read data signal. If there is 1 output and N inputs the capacity of such memory is $2^N$ bits (Fig. 4). Like programmable connections with SOP approach the content of LUT has to be pre-programmed in correspondence with the function of combinational circuit.
Simple Programmable Logic Devices

The structure presented in Fig. 3-a could be referred to a class of Simple Programmable Logic Devices (SPLDs) and entitled as PAL4H4 (Programmable Array Logic which is one-time programmable) or GAL4H4 (Generic Array Logic – reprogrammable), where the first number – in number of inputs, the last one – number of outputs, but the character H in the middle means that it is an active-HIGH logic device (Fig. 5-a). Other possible values of this parameter for real SPLD are L – active-LOW logic (Fig. 5-b) and V – if the output can be configured as active-HIGH or active-LOW logic (Fig. 5-c).

In practice real SPLDs have bigger number of inputs and outputs. Moreover some outputs can be utilized again as inputs (Fig. 5-c and Fig. 5-d) that allows cascading of the SOPs. For example, SPLD 18V8 (basic structure in Fig. 6) has 10 distinct inputs, 2 distinct outputs and 6 pins that can be used as outputs, inputs or both (input and output). Therefore the maximal number of inputs is 18, but the maximal number of outputs is 8 that give the title 18V8 (detailed structure of Atmel’s ATF18V8 is given in Appendix A). Other significant feature of the real SPLDs is a flip-flop installed in the output logic (Fig. 5-e). This allows development of counters, registers, timers and other dynamic digital devices with SPLDs. The flip-flops have common clock input. Also outputs of SPLDs can be enabled/disabled with common signal (Fig. 5-e). The flip-flop, output pin logic and OR element together are called microcell. It is obvious then the number of macrocells in SPLD is equal to the maximal number of its outputs. For instance 18V8 includes 8 macrocells.

![Diagram of SPLD output pin logic](image)

**Fig. 5.** Basic types of SPLDs output pin logic (types of macrocells)
Fig. 6. Bloc diagram of 18V8 SPLD

Complex Programmable Logic Devices

A Complex Programmable Logic Device (CPLD) includes several gate arrays similar to those previously described as SPLDs (which in CPLDs are usually called Logic Array Blocks – LABs). In CPLDs connections are provided by Programmable Interconnection Array (PIA). Any input of any LAB, any output of LABs, as well as any pin of the chip can be connected one to another through the PIA (Fig. 7).

CPLDs themselves, as well as LABs of CPLDs are quite advanced devices. For example LABs of Altera’s MAX3000 CPLDs can be described as 36V16 (see Appendix B for more details). I.e. they include 16 macrocells and have up to 36 inputs connected to the PIA. The lowest member of the family EPM3032 includes 2 LABs or 32 macrocells, but the highest EPM3512 – 32 LABs or 512 macrocells.
Macrocells of CPLDs are also more complicated. For example, macrocells of MAX3000 includes a flip-flop that can be programmed for D, T, RS and JK operation. They also include hardware dedicated for cascading of macrocells: parallel and shared expanders. The parallel expansion assumes that the Term Allocation Matrix (TAM) of macrocell (connection of OR to AND elements) has special inputs that can be directly connected to other outputs of macrocells that allows their cascaded use (Fig. 8-a). The shared expanders are based on de Morgan’s law. In order to provide that TAM has special path that connects an AND element to PIA through an inverter. Inverted product then is passed through another AND element thus providing and additional sum op products (Fig. 8-b). Configurations of CPLDs are directly controlled from their internal “Flash” memory that allows using them with less number of extra components.

![Fig. 7. Generic CPLD](image)

Macrocell Expanders in Altera’s MAX7000 CPLD

![Fig. 8. Use of Macrocell Expanders](image)
Field Programmable Gate Arrays

Heart of a CPLD is its interconnection matrix (PIA) that provides versatile communication between its blocks. The blocks themselves are rather complex. This lets to consider CPLDs as digital circuits with concentrated resources. In contrast, resources of Field Programmable Gate Arrays (FPGAs) are distributed (Fig. 9-a). The main kinds of these resources are Configurable Logical Blocks (CLBs), their interconnections and input/output (I/O) blocks.

CLBs are less complex than LABs of CPLDs. They contain several logic modules with a local interconnect array (Fig. 9-b). Logic modules are the fundamental “bricks” of FPGAs like macrocells are fundamental blocks of CPLDs. Logic modules are based on LUT approach (Fig. 4-b and Fig. 9-c) and are less complex than macrocells.

I/O blocks are located around the perimeter of the gate array. They are configurable and can operate as input, output or bidirectional buffers.

The global connection resources CLB-CLB and CLB-I/O are distributed and locally limited. This makes connection of the neighbour CLBs and I/O blocks more preferable than the farther ones. These connections are divided into two big groups - vertical (column) and horizontal (row) connections.

FPGAs often incorporate additional hardware (memory blocks, DSP cores, transceivers etc.) thus providing an exceptional basis for comprehensive intellectual control solutions. From this point of view FPGAs are often divided into two groups: high-end and low-cost. The high-end FPGAs (Altera’s STRATIX and Xilinx’s VIRTEX) incorporates more specific hardware resources that the low-cost ones (Altera’s CYCLONE and Xilinx’s STARTAN).

![Diagram of FPGA with I/O blocks, CLBs, and logic module connections](image)

**Fig. 9.** Generic FPGA
An FPGA may have huge number of logic modules and CLBs. Therefore its configuration memory is also extremely large. Due to this huge volume FPGAs are either one-time programmable (usually with anti-fuse technology) or configured from RAM. The RAM is loaded with configuration data right at power-up from a boot Flash EEPROM that may be internal, but usually is external.

As an example, let’s take a look at the lowest member of Cyclone II family (not the most advanced) – FPGA EP2C5. However, it includes 4608 logic modules (called in Altera’s documentation LE – Logic Element), 26 RAM blocks 4kb (kilobit) each, 13 embedded 18×18 multipliers, as well as 2 phase-locked loops (PLLs) that provide general-purpose clocking with clock synthesis and phase shifting.

Each LE contains 4-input LUT, programmable flip-flop, carry chain circuit and register chain circuit (refer Appendix C for details). Each 16 LEs are grouped in CLB (which are called LABs – Logic Array Blocks in Altera’s documentation) equipped also with local interconnect matrix (that provides connection of LEs to up to 48 other LEs). Altogether there are 288 LABs organized in a matrix of 12 rows and 24 columns. The configuration memory has capacity of 1265792 bits that (taking into account 50% decompression rate) requires 1Mb (megabit) configuration Flash EEPROM EPCS1.

Design Process

A programmable logic device can be compared with an empty experimenting board where a digital electronic expert put digital gates and connects them together by wires. In the case of programmable logic this design and assembling process occurs virtually with assistance of dedicated software (for example, Altera’s Quartus II or Xilinx’s Web Pack) installed on an instrumental computer. Hardware is required at the very final stage in order to test the designed circuit in reality.

![Design flow diagram](image)

**Fig. 10. Project Development and Debugging**
The most advanced approach assumes that the design is deployed in a programmable logic device (target
device) that can be programmed “in-system”. Then the target is all the time attached to the instrumental
computer (usually through a device called “in-system programmer/debugger” that in fact is an interface
between the target and instrumental computer – Fig. 10-a). An alternative approach assumes that the
target is programmed in the dedicated device called “programmer” but not in the system.

The design process of the programmable logic devices (often called “design flow”) consists of several steps:
1) design entry/editing; 2) functional simulation; 3) synthesis; 4) implementation; 5) timing simulation; 6) download and in-system debugging (Fig. 10-b).

The design entry/editing is the stage of the design flow at which the developed system is described. This
can be done either with a schematic (Fig. 11) or text entry tool. The text entry is done with one of Hardware
Description Languages (HDLs): VHDL (Very high speed integrated circuits Hardware Description Language –
IEEE St. 1076), Verilog HDL (IEEE St. 1364), AHDL (Altera’s HDL) or ABEL (Advanced Boolean Expression
Language). The device independent entry is done with schematic, VHDL or Verilog (AHDL is owned by Altera,
but ABEL – by Xilinx). The entered design is compiled. During the compilation the source code of the design
is translated into object code that can be used by simulators or can be written into the target device. Note
that “source code” and “object code” definitions regards rather structural than algorithmic description.

The functional simulation stage (device independent) verifies if the design operates as it is expected. It is
checked if correct output is produced for typical combinations of inputs, which is produced by a software
tool called “waveform editor”. If this stage reveals mistakes the design flow returns to entry/editing stage.

The synthesis stage (device independent) provides translation of the design into a standard form netlist.
The next stage is called **implementation** (Fig. 12). At this stage the standard netlist generated during the synthesis phase is mapped in an actual device. At this stage all inputs and outputs of the design are tied to the particular pins of the device, as well as constraints of deployment in the device are taken into account. At the end of this stage an output file suitable to deployment in the target device, so called bitstream, is generated. This phase is device dependant.

The **timing simulation** stage verifies if the design is able to operate in the particular programmable device. All propagation delays that take place in the chosen programmable device are taken into account. If the timing simulation reveals that the design deployed in the particular chip does not satisfy the timing requirements the design must be either revised or deployed in another chip.

In order to implement the design in hardware the **download** stage is necessary. At this stage the previously obtained bitstream is deployed in the chosen programmable device (target device). As it has been mentioned the most comprehensive kind of debugging assumes that the target chip is attached to the instrumental computer through an in-system programmer/debugger. Then the design can be deployed in the chip and tested in the real operation environment and in the real time. “USB-Blaster” is example of in-system programmer for Altera’s devices. It is attached to the instrumental computer through USB, but to the target through JTAG interface.

![Screenshot of design implementation stage](image-url)
Fig. 13. Screenshot of design download stage (programming tool)

The JTAG for boundary scan logic (a kind of test interface and procedure) is described on IEEE St. 1149.1. It specifies that JTAG Test Access Port (TAP) consists of 4 mandatory inputs/outputs and 1 optional (Fig. 14-a).

The mandatory signals are:

Test Data In (TDI) – input for serially shifting programming and test data;
Test Data Out (TDO) – output of serially shifting programming and test data;
Test Mode Select (TMS) – input that enables/disables TAP;
Test Clock (TCK) – input that provides clock for TAP controller.

The optional signal is:

Test Reset (TRST) – optional input for TAP controller reset.

Besides TAP IEEE St. 1149.1 specifies a set of control registers for boundary scan logic:

1. Boundary Scan Register – is composed of Boundary Scan Cells (BSCs) that are primary sources of test information;
2. Bypass Register – a “shortcut” register that consists of one flip-flop and used if none of other registers is used;
3. Instruction Register – stores instructions for boundary scan operation;
4. Identification Register – stores a code that identifies the particular device.
IEEE St. 1149.1 identifies also a set of instructions that are written in the instruction register and control boundary scan logic.

It has also to be noted that several devices with JTAG TAP can be connected together in a test chain for joint programming and testing. An example of such chain is given in Fig. 14.

Manufacturers of Programmable Logic Devices

The most significant players on the market of the programmable logic devices are Xilinx, Altera and Actel corporations. The total volume of market of semiconductor devices in 2009 was 230 billions USD. 3.3 of them define the volume of PLD segment. 55% of this segment is taken by Xilinx and 39% – by Altera.
Part II. Basics of VHDL

The simplest kind of the design entry tool is a schematic editor. However, the schematic editors have a number of drawbacks. They are not convenient for complicated design with a big number of components, with big number of hierarchy levels, to implement algorithmic descriptions and mix them with structural descriptions etc. For these reasons, significant designs are usually made with Hardware Description Languages (HDLs). The most widely used HDLs are open and in standards specified languages: VHDL (Very high speed integrated circuits Hardware Description Language – IEEE St. 1076) and Verilog HDL (IEEE St. 1364). Of them VHDL became a standard tool in hardware description like C++ is a standard tool in programming. Although HDLs look like traditional programming languages they have significant differences. HDLs describe hardware and their commands that correspond to logic gates are executed in parallel (at once). This makes description of a design equal to an interconnection of components.

Design Approaches

There are three basic approaches how to describe a design with VHDL for further physical implementation: structural, data flow and algorithmic (Fig. 15). The first one, structural approach, represents the design as a combination of elements and their interconnections, the second one, dataflow – as a set of formulas that process input data and produce some outputs, but the third one, behavioural or algorithmic – as a combination of states, conditions and events. Finally, nevertheless the chosen description and design method, the design is implemented on the physical level (the corresponding PLD is configured in correspondence with the described function). This implementation may look quite different (especially optimized) from the initial description, but always keeps its function.

Generalised Structure of VHDL File

A VHDL design is called entity which consists of the entity’s declaration (interface) and entity’s architecture description, as well as includes declarations of libraries and packages used in the entity (Fig. 16). VHDL entity may also contain other entities that are then considered as components of the top-level entity. The entity declaration can be considered as the interface to the external circuits that consists of the input and output signals. At the same time the architecture body includes the description of the entity that may be composed of low-level entities and their interconnections (structural approach), or described as a set of processes/state flow/ formulas (behavioural/dataflow approach) or may combine the above methods.
Let’s take a half-adder as an example. Its dataflow description obtained as a SOP formula looks like:

\[
HS = \overline{A} \cdot B + A \cdot \overline{B},
\]

(3)

\[
HC = A \cdot B.
\]

(4)

The corresponding architecture may be considered as a black-box with the corresponding set of formulas describing the half-adder in it (Fig. 17-a). The same half-adder can be presented with structural approach. Then the corresponding black-box contains several logical gates (Fig. 17-b). VHDL provides specific lexical constructions for both structural and dataflow approach, as well as for behavioural.
Lexical Elements of VHDL

Before discussing the statements that declares entity and describe its architecture let’s take a look at basic lexical elements of VHDL. The following elements can be emphasized: basis identifiers, extended identifiers, numerical literals and symbolic literals.

Basic Identifiers

Identifiers are user-defined words that name objects in VHDL. There are two kinds of identifiers in VHDL: basic and extended. The basic identifiers are composed in correspondence to following rules:

1) basic identifiers may contain only Roman alphabet letters (A to Z), decimal digits (0-9) and the underscore (_);
2) the first character of a basic identifier must be a letter, but the last one cannot be an underscore;
3) basic identifiers cannot include two consecutive underscores;
4) basic identifiers can be of any length;
5) basic identifiers are not case sensitive («And2», «AND2» as well as «and2» refer to the same object);
6) basic identifiers cannot be a keyword (Table II).

Table II. Keywords (Reserved words) in VHDL

<table>
<thead>
<tr>
<th>abs</th>
<th>disconnect</th>
<th>is</th>
<th>out</th>
<th>sli</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>downto</td>
<td>label</td>
<td>package</td>
<td>sra</td>
</tr>
<tr>
<td>after</td>
<td>else</td>
<td>library</td>
<td>port</td>
<td>srl</td>
</tr>
<tr>
<td>alias</td>
<td>elsif</td>
<td>linkage</td>
<td>postponed</td>
<td>subtype</td>
</tr>
<tr>
<td>all</td>
<td>end</td>
<td>literal</td>
<td>procedure</td>
<td>then</td>
</tr>
<tr>
<td>and</td>
<td>entity</td>
<td>loop</td>
<td>process</td>
<td>to</td>
</tr>
<tr>
<td>architecture</td>
<td>exit</td>
<td>map</td>
<td>pure</td>
<td>transport</td>
</tr>
<tr>
<td>array</td>
<td>file</td>
<td>mod</td>
<td>range</td>
<td>type</td>
</tr>
<tr>
<td>assert</td>
<td>for</td>
<td>nand</td>
<td>record</td>
<td>unaffected</td>
</tr>
<tr>
<td>attribute</td>
<td>function</td>
<td>new</td>
<td>register</td>
<td>units</td>
</tr>
<tr>
<td>begin</td>
<td>generate</td>
<td>next</td>
<td>reject</td>
<td>until</td>
</tr>
<tr>
<td>block</td>
<td>generic</td>
<td>nor</td>
<td>return</td>
<td>use</td>
</tr>
<tr>
<td>body</td>
<td>group</td>
<td>not</td>
<td>rol</td>
<td>variable</td>
</tr>
<tr>
<td>buffer</td>
<td>guarded</td>
<td>null</td>
<td>ro</td>
<td>wait</td>
</tr>
<tr>
<td>bus</td>
<td>if</td>
<td>of</td>
<td>select</td>
<td>when</td>
</tr>
<tr>
<td>case</td>
<td>impure</td>
<td>on</td>
<td>severity</td>
<td>while</td>
</tr>
<tr>
<td>component</td>
<td>in</td>
<td>open</td>
<td>signal</td>
<td>with</td>
</tr>
<tr>
<td>configuration</td>
<td>inertial</td>
<td>or</td>
<td>shared</td>
<td>xnor</td>
</tr>
<tr>
<td>constant</td>
<td>inout</td>
<td>others</td>
<td>sla</td>
<td>xor</td>
</tr>
</tbody>
</table>


Extended Identifiers

In VHDL it is possible to overcome limitations of the basic identifiers through the applying of a set of extended rules that allow identifiers with any sequence of characters. The identifiers defined with this extended rules are called «extended identifiers». So the rules for their definition are:

1) extended identifiers are enclosed between by two backslash “\” characters, for example«\AND\»;
2) extended identifiers may consist of any combination of any symbols including backslashes. Then the backslashes that are found in the middle of a sequence are interpreted as its part. For example, sequence of symbols «\MCU_IOs\Outputs\» defines the identifier «\MCU_IOs\Outputs\»;

3) extended identifiers are case sensitive;

4) extended identifiers are not the same class of text elements as basic identifiers or reserved words; for this reason may exist extended identifiers equal to reserved words or existing basic identifiers;

5) Extended identifiers are not allowed in the VHDL versions earlier than VHDL-93.

**Numerical Literals**

VHDL allows integer and real literals. The integer literals consist of whole numbers that never include a decimal point, while the real literals always include it. Both kinds of literals may have an exponential part that starts after “E” or “e” symbol. The exponent of integer literals must always be positive. The literals «2000», «2E3» and «2e+3» are integer literals, but «90.0», «0.9e2» and «314.0E-2» – real literals. Negative numbers are represented as a combination of a negation operator and a literal. The default numerical system in VHDL is decimal. To express a numbers in a different numerical system the following construction is used:

```
<BASE>##<NUMBER>##
```

where:

- `<BASE>` – base of the numerical system;
- `<NUMBER>` – the number represented in the chosen numerical system;

For example «2#1011010#» is a binary number 1011010 (equal to decimal 90). At the same time its representation in octal system is «8#132#», but in the hexadecimal – «16#5A#». For better readability underscores can be used, for example «2#0101_1010#».

**Character and Strings Literals**

Character literals in VHDL are enclosed in apostrophes, for example: ‘a’, ‘B’, ‘2’ etc. At the same time strings of characters are placed in quotation marks, like: “My string”, “90”, “This is an example of “String in the string”“ - placed in repeated double quotation marks”. A string may contain any printing character.

A bit-string represents a sequence of bit values. Bit strings are recognized by B in beginning of the string, for example B”1011010”. In fact, bit strings are character strings with a limited set of characters. For this reason the string “1011010” is also a bit string. Use of B, therefore, is not obligatory but increases readability of VHDL script. Also hexadecimal and octal strings are available. Then X or O is placed before the string, for example: X” 5A”, O”132”.

It is known that each digit of hexadecimal system correspond 4 digits (bits) of binary system, but of the octal – to 3 bits. This has direct impact on the correspondence between binary, hexadecimal and octal

---

1 In this template, as well as later in templates positions for user defined identifiers or literals are capitalized and put in corner brackets (like `<BASE>` in this template), reserved keywords are regular and emphasized with blue colour (for example `type`), comments are green (like `--declarations`), but regular elements in templates (like `<type_definition>`) are user defined elements of limited choice. Optional parts of the templates are put in square brackets.
strings. One character in the hexadecimal string corresponds to 4 characters in the binary string and 1 in the octal – to 3 in the binary. For this reason X"5A" is not equivalent of B"1011010", but is equivalent of B"01011010". In the same way O"132" does not correspond to B"1011010", but does correspond to B"001011010".

Data Types

In VHDL data objects have to be declared specifying a particular data type for it. However, VHDL itself does not define any data type, but provides tools for doing it so that all used data types must be declared utilizing these tools. For this reason all data types can be divided into two groups: standard (declared in the attached libraries) and user defined (declared in the given architecture).

The standard library and package can be attached with the following statement:

```
library std, work;  
use std.standard.all;
```

One more significant package where important data types are defined is «std_logic_1164» in library «ieee». This package is hooked to VHDL project as follows:

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

Depending on the type declaration kind it is also possible to emphasize integer, real, physical and enumerated data types (Fig. 18). Beside that there are two categories of data types: scalar and composite that also can be integer, real, physical or enumerated.

![Data Types in VHDL](image)

**Fig. 18. Data Types in VHDL**

---

2 In examples: user definer identifiers are capitalized (for example, «DUTY_CYCLE»), but standard or defined in libraries (as well as names of libraries, packages and modules are regular, like «std.standard.all»). Comments in examples are green, but keywords are blue (like `library`). Note also that complete VHDL statements are recognized with a semicolon sign «; ».
Some standard (library defined) data types

The scalar data types correspond to objects that can hold only one value at any time instant. The scalar data types are integer, real, physical and enumerated. Read chapter «Declaration of data types and subtypes» for more details about data type declaration.

Standard integer data types – are mathematically integer numbers. The integer data types are defined with integer literals. The minimum range of integer numbers is specified by the Standard Package contained in the Standard Library as the range $-2,147,483,647$ to $+2,147,483,647$. Besides that the Standard Package defines also integer types «natural» (with values from 0 to the specified maximum) and «positive» (from 1 to the specified maximum).

Standard real data types – are used to declare objects that emulate mathematically real numbers. The real data types are defined with real literals. The minimum range of real numbers specified by the Standard Package in the Standard Library is from $-1.0E+38$ to $+1.0E+38$.

Standard physical data types – are used to represent physical quantities. A physical type provides not only range of values but also a base unit, and related units that are defined in terms of the base unit. Read chapter «Declaration of data types and subtypes» for more details about physical data type definition. The Standard Package in the Standard Library defines physical data type «TIME», which main unit is «sec» (seconds) but which can also be defined in «min» (minutes), «hr» (hours), «ms» (milliseconds), «us» (microseconds), «ns» (nanoseconds), «ps» (picoseconds) and «fs» (femtoseconds).

Standard enumerated data types – consist of lists of values. A digital circuit designer can use an enumerated type to represent exactly the values required for a specific operation. The values of an enumerated data type are user-defined. Read the chapter entitle «Declaration of data types and subtypes» for more details about enumerated data type definition. The Standard Package in the Standard Library defines enumerated data types «bit» (with values ‘0’ and ‘1’), «boolean» (with values FALSE and TRUE), «character» (any legal VHDL character), «severity_level» (with values note, warning, error and failure), «file_open_kind» (with values read_mode, write_node and append_mode) and «file_open_status» (with values open_or, status_error, name_error and mode_error).

One more important enumerated type that is defined in the «std_logic_1164 package» is the «std_ulogic» type. This type has values (from left to right) are: ‘U’ (uninitialized), ‘X’ (forcing unknown), ‘0’ (forcing 0), ‘1’ (forcing 1), ‘Z’ (high impedance), ‘W’ (weak unknown), ‘L’ (weak 0), ‘H’ (weak 1) and ‘-‘ (don’t care).

Standard array types are composed of several elements of the same type. The most important arrays declared in standard library are «string» and «bit_vector» arrays. The «string» is a one-dimensional array (vector) composed of «character» type elements. The «bit_vector» is also vector, but composed of «bit» type elements (‘0’ and ‘1’).
**Declaration of data types and subtypes**

A new type can be declared with the following statement:

```plaintext
type <TYPE_IDENTIFIER> is <type_definition>;
```

where:

- `type` is specific keywords that declares a type;
- `<TYPE_IDENTIFIER>` is user-defined identifier of the type;
- `<type_definition>` is definition of the type; most frequent type definitions are:
  1. ascending range (with minimum `<MIN>` and maximum `<MAX>`):
     ```plaintext
     range <MIN> to <MAX>
     ```
  2. descending range (with the same limits):
     ```plaintext
     range <MAX> downto <MIN>
     ```
  3. list of values (identifiers or character literals):
     ```plaintext
     (<list_of_values>)
     ```

A subtype is a subset of a previously defined type. A subtype within an existing type can be declared with the following statement:

```plaintext
subtype <SUBTYPE_IDENTIFIER> is <TYPE_IDENTIFIER> <subtype_definition>;
```

where:

- `subtype` is specific keywords that declares a subtype;
- `<SUBTYPE_IDENTIFIER>` is user-defined identifier of the subtype;
- `<TYPE_IDENTIFIER>` is the identifier of an existing data type;
- `<subtype_definition>` is definition of the subtype.

Below some examples of type and subtype definitions are given.

**User defined integer types** are declared with integer literals. There are few examples of new integer types:

```plaintext
type SMALL_INT is range 0 to 1024;
type MY WORD_LENGTH is range 31 downto 0;
subtype DATA_WORD is MY WORD_LENGTH range 7 downto 0;
subtype INT_SMALL is INTEGER range -1024 to +1024;
```

The last two examples show the use of subtypes. The first one defines a type called «DATA_WORD» that is a subtype of the previously defined type «MY_WORD_LENGTH», but have even narrower data range from 7 to 0. In the second example an integer type «INT_SMALL» is defined based on the standard integer type.

**User defined real types** are declared with real literals. The next three examples shows definition of new floating-point types:

```plaintext
type PROBABILITY is range 0.0 to 1.0;
type CMOS_LEVEL is range 0.0 to 3.3;
subtype CMOS_LOW is CMOS_LEVEL range 0.0 to +1.8;
```
User defined physical types as well as all physical data types definitions includes not only type declaration, but also declaration of units:

```vhdl
type <PH_TYPE_IDENTIFIER> is <type_definition>;
    units
        <BASE_UNIT>;
        ...
        <RELATED_UNITS>;
        ...
    end units [<PH_TYPE_IDENTIFIER>];
```

where:
- `type` – specific keywords that declares a physical type;
- `is` – specific keywords that declares a unit system for the type;
- `units` – specific keywords that declares a unit system for the type;
- `<TYPE_IDENTIFIER>` – user-defined identifier of the type;
- `<type_definition>` – definition of a numeric type;
- `<BASE_UNIT>` – the base unit of the type; a space must be left before the unit name;
- `<RELATED_UNITS>` – all related units of the type and their definition on terms of the base type; a space must be left before the unit name.

An example of physical type declaration:

```vhdl
type CURRENT is range 0.0 to 10.0
    units
        A;
        mA = 0.001 A;
        uA = 0.001 mA;
        nA = 0.001 uA;
    end units CURRENT;
```

User defined enumerated types are declared with a list of values (identifiers or character literals):

```vhdl
type <TYPE_IDENTIFIER> is (LIST_OF_VALUES);
```

Some examples of user defined enumerated types are listed below:

```vhdl
type MY_VALUES is ('0', '1', 'Z');
type INSTRUCTION is (load, store, add, sub, div, mult, shiftl, shiftr);
```

The «std_ulogic» type is defined in the «std_logic_1164» package as an enumerated type:

```vhdl
type STD_ULOGIC is (
    'U', -- uninitialized
    'X', -- forcing unknown
    '0', -- forcing 0
    '1', -- forcing 1
    'Z', -- high impedance
    'W', -- weak unknown
    'L', -- weak 0
    'H', -- weak 1
    '-') -- don't care
```
As soon as data types are defined (in the architecture body file of in the attached libraries) it becomes possible to declare data objects of the defined data types. Here are some object declarations that use the types from above examples:

```vhdl
variable BUS_WIDTH: SMALL_INT := 24;
signal DATA_BUS: MY_WORD_LENGTH;
variable VAR1: CMOS_LEVEL range 0.0 to 2.5;
constant LEAKAGE: CURRENT := 125 nA;
signal SIG1: MY_VALUES;
variable ALU_OP: INSTRUCTION;
```

**Composite data types**

Composite data types correspond to data objects that can hold more than one value at any time instant. Such data objects consist of multiple related data elements. There are two kinds of the composite data types (do not confuse them with data objects): arrays and records.

**Arrays types** correspond to the objects that are composed of several elements of the same scalar type. There are two kinds of array types – constrained and unconstrained. A *constrained array type* is of definite dimensions and it is declared as follows:

```vhdl
type <ARRAY_IDENTIFIER> is array (<indexing scheme>) of <TYPE_IDENTIFIER>;
```

where:
- `type` is specific keywords that declares an array type;
- `is array` is of `<ARRAY_IDENTIFIER>` is user-defined identifier of the array type;
- `<TYPE_IDENTIFIER>` is the identifier of an existing scalar data type;
- `<indexing scheme>` is way of numbering of elements:
  1. ascending range (with minimum `<MIN>` and maximum `<MAX>`):
     - `<MIN>` to `<MAX>`
  2. descending range (with the same limits):
     - `<MAX>` downto `<MIN>`
  3. list of values (identifiers or character literals):
     - `<LIST_OF_VALUES>`

Some examples of the array type declaration:

```vhdl
type WORD16A is array (15 downto 0) of std_logic;
type WORD16D is array (0 to 15) of std_logic;
type EIGHT_V is array (0 to 7) of integer;
type STD_LOGIC_1D is array (std_ulogic) of std_logic;
```

In the first example a one-dimensional array type of `std_logic` type elements indexed in the descending order (0 to 15) is defined. In the second example a similar array type is indexed in the ascending order (from 15 down to 0). The third array type consists of 8 integer elements. The last example defines an array of `std_logic` type elements that uses the type `std_ulogic` to define the indexes. This array looks as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td>4th</td>
<td>5th</td>
<td>6th</td>
<td>7th</td>
<td>8th</td>
<td>9th</td>
</tr>
</tbody>
</table>
The data objects that correspond to the previously defined types can be declared as follows:

```vhdl
signal ADDR_WORD: WORD16A;
signal DATA_WORD: WORD16D := B"1101100101010110";
constant SETTING: EIGHT_V := (2,4,6,8,10,12,14,16);
```

In the first example the signal «ADDR_WORD» is defined as an array of 16 elements, of «std_logic» type each. The initial values of all these bits are ‘0’ s. In the second example the initial values of similar array «DATA_WORD» are defined directly as a bit string. In the third example an integer array «SETTING» is also initiated directly.

Individual elements of an array are accessible with the proper index. The use of the indexes is positional taking into account their order. For example, ADDR_WORD(15) accesses the most left bit of this array (because its order is descending), while DATA_WORD(15) accesses the most right bit of the array (which default value is ‘0’). It is also possible to access a range of elements, applying the proper index range. For example the left 8 elements of the mentioned arrays are accessed with the following statements:

```vhdl
ADDR_WORD(15 downto 8)
and
DATA_WORD(0 to 7)
```

It is also possible to declare multidimensional arrays. This can be done with a similar syntax, but several indexing schemes (for each dimension). Some examples of array type declarations:

```vhdl
type MATRIX3X2 is array (1 to 3, 1 to 2) of natural;
type MATRIX4X2 is array (1 to 4, 1 to 2) of integer;
type STD_LOGIC_2D is array (std_ulogic, std_ulogic) of std_logic;
```

and data objects corresponding to the declared array types:

```vhdl
variable DATA_MATRIX: MATRIX4X2 :=((0,2), (1,3), (4,6), (5,7));
```

The variable array «DATA_MATRIX» will then be initialized with the following initial content:

```
0  2
1  2
4  6
5  7
```

Particular elements can be accessed with both indexes, for example DATA_MATRIX(3,1) returns the value 4.

It is also possible to declare an array type without specifying its dimensions at the moment of the declaration. Such array types are called unconstrained array types. The syntax of their declaration is:

```vhdl
type <ARRAY_IDENTIFIER> is array (<<TYPE1> range <> ) of <TYPE_IDENTIFIER>;
```

where:

- `type` – specific keywords that declares an array type;
- `is array` – user-defined identifier of the array type;
- `of` – the identifier of an existing scalar data type;
- `range <>` – user-defined identifier of the type for array dimensioning;
Some examples of unconstrained array types:

```vhdl
  type VECTOR_UNCI is array (integer range <>) of integer;
  type VECTOR_UNCN is array (natural range <>) of integer;
  type MATRIX_UNC is array (natural range <>, natural range <>) of std_logic;
```

The corresponding data array objects are declared as follows:

```vhdl
  variable MTX1: VECTOR_UNCI (2 downto -7) := (3, 5, 1, 4, 7, 9, 2, 1, 20, 8);
  variable A4x2: MATRIX_UNC (1 to 4, 1 to 2) := (('1','0'), ('0','-''), ('1','2'),
             ('X','1'));
```

A record is a composite type that consists of multiple elements that may be of different types. The syntax of declaration of a record is the following:

```vhdl
  type <RECORD_IDENTIFIER> is
    record
      <FIELD_IDENTIFIER1> :<TYPE_IDENTIFIER1>;
      <FIELD_IDENTIFIER2> :<TYPE_IDENTIFIER2>;
      ...
      <FIELD_IDENTIFIERn> :<TYPE_IDENTIFIERn>;
    end record;
```

where:

- **type** — specific keywords that declares a record type;
- **is** — record
- **end record** — user-defined identifier of the record type;
- **<FIELD_IDENTIFIERk>** — user-defined identifier of field of the record type;
- **<TYPE_IDENTIFIERk>** — the identifier of an existing scalar data type;

Below an example of a record declaration:

```vhdl
  type PWM is
    record
      RISE_TIME  :TIME;
      FALL_TIME  :TIME;
      PERIOD     :TIME;
      DUTY_CYCLE :REAL range 0.0 to 1.0;
    end record;
```

and example of declaration of the corresponding data object:

```vhdl
  signal A, B: PWM;
```

as well as example of access to the separate fields or of the defined object (that is accessed utilising together object’s name and field’s name separated by a dot character) of the record type are given:

```vhdl
  A.RISE_TIME <= 1 ns;
  A.FALL_SIZE <= 5 ns;
  A.PERIOD <= 10 us;
  A.DUTY_CYCLE <= 0.5;
  B <= A;
```

---

3 Since VHDL operators and statements are ended with a semicolon sign `;`, the longer lines can be easily split into a set of shorter ones. In this example the continuation is emphasized with an indent.
Data objects

A data object is one of the following: a constant, a variable or a signal. Data objects have to be declared in the proper part of VHDL model. Data objects are recognized by their identifier, type and value. Signals in VHDL can be compared with connections in a schematic that are characterized by their values (like electrical node is characterized by its voltage), while constants and variables have the same meaning as they have in programming languages and are used to estimate behaviour.

Constants

A constant is a named number that is used in various locations of a VHDL model. A constant can be declared and its value can be defined only once. Constants are declared with the following statement:

```
constant <LIST_OF_NAMES_OF_CONSTANT>: <type> [:= <INITIAL_VALUE>];
```

where:
- `constant` – specific keyword that declares a constant;
- `<type>` – built-in or user-defined type of the constant;
- `<LIST_OF_NAMES_OF_CONSTANT>` – user defined name or names of the constant;
- `<INITIAL_VALUE>` – optional initial value of the constant.

The constants declared in the beginning of an architecture are valid within the architecture. Constants can also be declared for narrower region. For example, the constants declared in the beginning of a process are valid only in the process. Some examples:

```
constant PERIOD: time := 10 us;
constant RISE_TIME, FALL_TIME: time:= 1 ns;
constant MD_BUS, MA_BUS: integer:= 16;
```

Variables

Variables are local data storages allowed in processes (as well as in subprograms). Unlike constants variables can be updated and their update occurs without any delay as soon as the update statement is executed. Variables are declared in a process with the following statement:

```
variable <LIST_OF_VARIABLES>: type [:= <INITIAL_VALUE>];
```

where:
- `variable` – specific keyword that declares a variable;
- `<type>` – built-in or user-defined type of the variable;
- `<LIST_OF_VARIABLES>` – user defined name (or names) of the variable (variables);
- `<INITIAL_VALUE>` – optional initial value of the variable.

Some examples of variable declarations:

```
variable VAR_BIT: bit :=0;
variable VAR_BOOLEAN: boolean :=FALSE;
variable VAR_INTEGER: integer :=1000;
variable CNTR: integer range 0 to 15;
variable VAR_BIT_V: bit_vector (3 downto 0);
```
The variable «CNTR», in the fourth example, is an integer with a restricted range of values (0...15). The last example defines a 4 element bit vector «VAR_BIT_V» with elements: VAR_BIT_V(3), VAR_BIT_V(2), VAR_BIT_V(1) and VAR_BIT_V(0).

Variable are updated using the following variable assignment statement:

\[ \text{<VARIABLE_NAME>} := \text{<expression>} ; \]

For example, statement

\[ \text{CNTR} := \text{CNTR} + 1 ; \]

updates the previously defined variable «CNTR».

**Signals**

A signal is an information stream transferred between parts (components, processes and equations) of VHDL entity, as well as its input and output data streams (Fig. 19).

![Signals in VHDL model](image)

Fig. 19. Signals in VHDL model

Signals declaration statement is the following:

\[ \text{signal } \text{<LIST_OF_SIGNALS>}: \text{<type>} [ := \text{<INITIAL_VALUE>} ] ; \]

where:

- \( \text{signal} \) – specific keyword that declares a signal;
- \( \text{type} \) – built-in or user-defined type of the signal;
- \( \text{<LIST_OF_SIGNALS>} \) – user defined name (or names) of the signal (signals);
- \( \text{<INITIAL_VALUE>} \) – optional initial value of the signal.

For example:

\[ \text{signal A_NOT, A_NOT, INT_1, INT_2: std_logic; } \]
\[ \text{signal MY_VALUE: integer :=0; } \]
\[ \text{signal CNTR: integer range 0 to 15; } \]
\[ \text{signal MD_BUS: bit_vector (0 to 15); } \]
All signals are updated with the following statement:

\[
\langle \text{SIGNAL\_NAME} \rangle \ <= \ \langle \text{expression} \rangle \ [\text{after} \ \langle \text{DELAY} \rangle];
\]

where:

\[
\langle \text{DELAY} \rangle \quad – \quad \text{optional delay of signal changes.}
\]

For example the output signal of a gate with a propagation delay 2ns is described as:

\[
\text{HS} \ <= \ (\text{not} \ \text{A} \ \text{and} \ \text{B}) \ \text{or} \ (\text{not} \ \text{B} \ \text{and} \ \text{A}) \ \text{after} \ 2 \ \text{ns};
\]

It is also possible to define multiple waveforms using multiple events:

```vhdl
signal WAVEFORM: std_logic;
WAVEFORM <= "0", "1" after 5ns, "0" after 10ns, "1" after 15 ns;
```

The main difference between variables and signals becomes obvious after taking a look at their reaction to input changes. A variable changes as soon as the variable assignment is executed. This regards only the variable which assignment is executed.

Signals change as soon as the corresponding inputs alter. In fact, the value of a signal is fixed in a buffer called «signal driver». All operations with signals are, in fact, made with their drivers which update the corresponding variables when the corresponding inputs change. It means that all signals that depend on the same inputs alter at once (taking into account a delay that is explicitly defined). Let’s discuss two examples of data processing.

**Example 1: Data processing using variables**

```
VARIABLE1 := VARIABLE2;
VARIABLE2 := VARIABLE1 + VARIABLE3;
VARIABLE3 := VARIABLE2;
RESULT <= VARIABLE1 + VARIABLE2 + VARIABLE3;
```

**Example 2: Data processing using signals**

```
SIGNAL1 <= SIGNAL2;
SIGNAL2 <= SIGNAL1 + SIGNAL3;
SIGNAL3 <= SIGNAL2;
RESULT <= SIGNAL1 + SIGNAL2 + SIGNAL3;
```

Let’s assume that the initial value of variables and numbers correspond to their number (i.e. 1, 2 and 3). In Example 1 statements are executed one by one. For this reason firstly «variable1» takes on a value of 2 (initial value of «variable2»), secondly «variable1» takes on a value of 5 (initial value of «variable3»=3 plus updated value of «variable1»=2 - because this variable has been changed first) and then «variable3» takes on a value of 5 (updated value of «variable2»). Finally, «Result» is equal to 12 (the sum of updated values of all variables.

In Example 2 all signals take on new values simultaneously and operate with the initial values of inputs. For this reason «signal1»=2, «signal2»=1+3=4 and «signal3»=2. Then «Result» is finally equal to 1+2+3=6.
**VHDL Operators**

VHDL supports different classes of operators that can process signals, variables and constants. The VHDL operators and their classes are briefly described in Table III.

<table>
<thead>
<tr>
<th>Class (and priority)</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Logical operators</td>
<td>and, or, nand, nor, xor, xnor</td>
</tr>
<tr>
<td>2. Relational operators</td>
<td>=, /=, &lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>3. Shift operators</td>
<td>sll, srl, sla, sra, rol, ror</td>
</tr>
<tr>
<td>4. Addition operators</td>
<td>+, =</td>
</tr>
<tr>
<td>5. Unary operators</td>
<td>+, -</td>
</tr>
<tr>
<td>6. Multiplying operators</td>
<td>*, /, mod, rem</td>
</tr>
<tr>
<td>7. Miscellaneous operators</td>
<td>**, abs, not</td>
</tr>
</tbody>
</table>

VHDL operators are of different priority. Operators of the class 7 (miscellaneous) are of the highest priority, but of the class 1 (logical) – of the lowest priority. Operators of the same priority are processed from left to right. The order of processing can be changed with parentheses.

Below a more detailed description of the VHDL operators is given.

**Logical operators**

The logic operators («and», «or», «nand», «nor», «xor» and «xnor») are valid with the «bit», «boolean», «std_logic» and «std_ulogic» types and their arrays. The logical operators are used to define logic expressions or to perform bitwise operations on bit arrays. The logical operators produce a result of the same type as their operands. Logical operators can be applied to signals, variables and constants.

Logical «and» also have sign «&», but logical «or» - «|».

Notice that the «nand» and «nor» operators are not associative. For this reason, for example, expression «X nand Y nand Z» is logically incorrect and will lead to a syntax error. It has to be replaced with expression «(X nand Y) nand Z» or «X nand (Y nand Z)».

**Relational operators**

A relational operator produce a Boolean output of «TRUE» or «FALSE» value depending on the validity of the condition of the operator.

The operator «smaller or equal to» looks exactly as assignment operator used to write a value to a signal or variable (both operators look as «<=»). In the examples below the first «<=» symbol is the assignment operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Types</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = B</td>
<td>Equality of A and B</td>
<td>any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>A /= B</td>
<td>Inequality of A and B</td>
<td>any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>A &lt; B</td>
<td>Is A smaller than B</td>
<td>scalar or enumerated types</td>
<td>Boolean</td>
</tr>
<tr>
<td>A &lt;= B</td>
<td>If A smaller than or equal B</td>
<td>scalar or enumerated types</td>
<td>Boolean</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>If A greater than B</td>
<td>scalar or enumerated types</td>
<td>Boolean</td>
</tr>
<tr>
<td>A &gt;= B</td>
<td>If A greater than or equal B</td>
<td>scalar or enumerated types</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Some examples of relational operations are:

```vhdl
variable CMP : boolean;
constant A : integer := 24;
constant B : integer := 32;
constant C : integer := 14;
CMP <= (A < B);
CMP <= ((A >= B) or (A > C));
CMP <= ((A >= B) and (A > C));
```

The first comparison produces «TRUE» because A < B. The second comparison also produces «TRUE» obtained as a result of disjunction of «TRUE» coming from A > C and «FALSE» coming from A ≥ B. In the similar way the third comparison produces «FALSE» as a result of disjunction of the same operands.

For data types defined as enumerated, the comparison is done on a «left to right» gradient basis. This means that the values defined in the right part of the corresponding enumeration list are more significant that those defined in its left part. In the next example the comparison will produce «TRUE» because ‘1’ stands to the left of ‘Z’ in the corresponding type definition list.

```vhdl
type NEW_STD_LOGIC is ('0', '1', 'Z', '-');
variable A1: NEW_STD_LOGIC := '1';
variable A2: NEW_STD_LOGIC := 'Z';
CMP <= (A1 < A2);
```

For array types, the comparison is done on an element-per-element basis. «TRUE» is produced for an array comparison only if comparison of all their elements produces «TRUE». For this reason the following comparison produces «FALSE»:

```vhdl
CMP <= (std_logic('1','0','1') < std_logic('0','1','1'));
```

### Shift operators

Built-in shift and rotate operators are included in VHDL93 standard (Table V). These operators allow shifting and rotating operations with one-dimensional array types (with elements of «bit» or «std_logic» or «boolean» type).

**Table V. Shift and Rotate Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Types</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sll B</td>
<td>Shift A left logical (B times; fill right vacated bits with 0)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
<tr>
<td>A srl B</td>
<td>Shift A right logical (B times; fill left vacated bits with 0)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
<tr>
<td>A sla B</td>
<td>Shift A left arithmetical (B times; fill right vacated bits with the most right bit)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
<tr>
<td>A sra B</td>
<td>Shift A right arithmetical (B times; fill left vacated bits with the most left bit)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
<tr>
<td>A rol B</td>
<td>Rotate A left circular (B times)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
<tr>
<td>A ror B</td>
<td>Rotate A right arithmetical (B times)</td>
<td>A: one-dimensional bit array type, B: integer</td>
<td>Same as A</td>
</tr>
</tbody>
</table>
When B is a negative integer, the opposite action occurs. For example shift to the left will be a shift to the right. The results of shift and rotate operation for \( A='10100101' \) and \( B=2 \) are given in Table VI.

### Table VI. Results of shift and rotate operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sll B</td>
<td>1001000</td>
<td>1001011</td>
</tr>
<tr>
<td>A sla B</td>
<td>1001011</td>
<td>1001010</td>
</tr>
<tr>
<td>A srl B</td>
<td>0010010</td>
<td>1110100</td>
</tr>
<tr>
<td>A sra B</td>
<td>1110100</td>
<td>0110100</td>
</tr>
<tr>
<td>A rol B</td>
<td>1001001</td>
<td>0110100</td>
</tr>
</tbody>
</table>

### Addition and subtraction operators

The addition and subtraction operators perform the corresponding arithmetic operation on operands of any numeric type. The concatenation (\&) operator is used to concatenate two one-dimensional arrays (vectors) together into a longer one. In order to use these operators it is necessary to attach the «std_logic_unsigned» or «std_logic_arith» packages in addition to the «std_logic_1164» package with the following statements:

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

or

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

### Table VII. Addition and Subtraction Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Types</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>Adds A and B</td>
<td>A: any numerical type, B: same as A</td>
<td>Same as A</td>
</tr>
<tr>
<td>A - B</td>
<td>Subtracts B from A</td>
<td>A: any numerical type, B: same as A</td>
<td>Same as A</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>Concatenate array A and array B into a bigger one</td>
<td>A: any array, B: same as A</td>
<td>Same as A and B</td>
</tr>
</tbody>
</table>

An example of concatenation is given below:

```vhdl
constant DOCTYPE : string := "MY_REPORT";
custom DOCYEAR : string := ".2009";
custom DOCMONTH : string := ".12";
MY_DATA <= "REPORT" & ".2009" & ".12"
```

The result of this operation will be a string “MY_REPORT.2009.12”.

### Unary operators

The unary operators “+” and “−” specify the sign of a numeric type (Table VIII).

### Table VIII. Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Types</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+A</td>
<td>Specify identity of A</td>
<td>A: any numerical type</td>
<td>Same as A</td>
</tr>
<tr>
<td>−A</td>
<td>Specify negation of A</td>
<td>A: any numerical type</td>
<td>Same as A</td>
</tr>
</tbody>
</table>
Operators of multiplying and dividing

The multiplying and dividing operators provide the corresponding mathematical functions with data objects of numeric types – integer or real (Table IX). The multiplication operator is also capable of operation with one operand of physical data type. Also dividend can be of physical type.

Table IX. Multiplying and Dividing Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*B</td>
<td>Multiplication</td>
<td>Any integer or real type</td>
<td>Same type as A</td>
<td>Same type as A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any physical type</td>
<td>Any integer or real type</td>
<td>Same type as A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any integer or real type</td>
<td>Any physical type</td>
<td>Same type as B</td>
</tr>
<tr>
<td>A/B</td>
<td>Division</td>
<td>Any integer or real type</td>
<td>Same type as A</td>
<td>Same type as A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any physical type</td>
<td>Any integer or real type</td>
<td>Same type as A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any integer or real type</td>
<td>Same type as A</td>
<td>Same type as A</td>
</tr>
<tr>
<td>A mod B</td>
<td>Modulus</td>
<td>Any integer type</td>
<td>Same type as A</td>
<td>Same type as A</td>
</tr>
<tr>
<td>A rem B</td>
<td>Remainder</td>
<td>Any integer type</td>
<td>Same type as A</td>
<td>Same type as A</td>
</tr>
</tbody>
</table>

The result of «rem» operation is a reminder after division. It has the same sign as its first operand (dividend). The result of the «rem» operation is defined as follows:

\[ A \text{ rem} B = A - \text{integer(A/B)} \cdot B, \]  

where A/B is an integer result of division.

In VHDL «mod» is also a kind or reminder finding. This result of «mod» operation is defined as follows:

\[ A \text{ mod} B = A - B \cdot N, \]  

where N is an integer. In the case both operands are of the same sign N is result of division and «mod» works exactly as «rem». If the signs of the operands are opposite then N is bigger by 1 so that the sign of «mod» operation is always the same as of the second operand.

Here are some examples of «rem» operator:

11 rem 4 = 3;
\((-11) \text{ rem} 4 = -3;\)

but for «mod» operator:

7 mod 4 = 3;
\(-7 \text{ mod} (-4) = -3;\)

7 mod (-4) = -1;

Miscellaneous operators

This group contains operators of exponentiation, finding absolute value and logical negation. The first two operators perform operations with numeric types. The logical negation results process logical types and returns an inverted value (Table X).

Table X. Miscellaneous operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A**B</td>
<td>Exponentiation</td>
<td>Any numerical type</td>
<td>Any integer type</td>
<td>Same type as A</td>
</tr>
<tr>
<td>abs A</td>
<td>Absolute value</td>
<td>Any numerical type</td>
<td>Any integer type</td>
<td>Same type as A</td>
</tr>
<tr>
<td>not A</td>
<td>Logical negation</td>
<td>«bit» «boolean» types</td>
<td></td>
<td>Same type as A</td>
</tr>
</tbody>
</table>
Entity's Declaration

Entities are declared and their connections (ports) are defined with the following construction:

```
entity <NAME_OF_ENTITY> is
    [generic (generic_declarations)];
port (<SIGNAL_NAMES1>: <MODE1> <TYPE1>;
    <SIGNAL_NAMES2>: <MODE2> <TYPE2>;
    ...
    <SIGNAL_NAMESn>: <MODEn> <TYPEEn>);
end [<NAME_OF_ENTITY>];
```

where:

- `entity` is mandatory keywords that defines entity’s declaration;
- `<NAME_OF_ENTITY>` is a user defined entity’s identifier;
- `<SIGNAL_NAMEsk>` are a user defined names of entity’s input and output signals;
- `<MODEk>` is reserved word that defines signal’s direction; may be either «in» (defines an input), «out» (defines an output), «buffer» (defines an output readable inside of entity’s architecture) or «inout» (signal that can be either input or output);
- `<TYPEk>` is built-in or user-defined type of the corresponding signal, for example: «bit» - bit type that can be of 0 or 1 value (built-in type); «bit_vector» - a vector of bit values (built-in type); «std_logic» - one of 9 values that defines strength and value of a signal (user defined type – library defined); etc.;
- `generic` is an optional declaration of the local constants usually used for time and bus widths definitions; It is defined as follows:

```
generic (  
    <CONSTANT_NAME1>: <TYPE1> [:=<VALUE1>] ;
    <CONSTANT_NAME2>: <TYPE2> [:=<VALUE2>] ;
    ...
    <CONSTANT_NAMEN>: <TYPEEn> [:=<VALUEEn>] );
```

where `<CONSTANT_NAMEk>` is free name of a constant of value `<VALUEk>` and type `<TYPEk>`

For example entity of the half-adder presented in Fig. 17 can be declared with the following VHDL script:

```
entity HALF_ADDER is
    port (A, B: in std_logic;
        HS, HC: out std_logic);
end HALF_ADDER;
```
Entity's Architecture Description

The architectures of the declared entities are described with the following construction:

```
architecture <ARCHITECTURE_NAME> of <NAME_OF_ENTITY> is
    -- components declarations
    -- signal declarations
    -- constant declarations
    -- function declarations
    -- procedure declarations
    -- type declarations
    ...
begin
    -- Statements (describes the design)
end [architecture] <ARCHITECTURE_NAME>;
```

where:
- `architecture` – specific mandatory keywords that defines the structure for entity’s architecture description;
- `of` – the identifier of the previously declared entity;
- `<ARCHITECTURE_NAME>` – a user defined name of architecture for the previously defined.

The same entity declaration may be associated with several architecture descriptions or with none of them.

Library and package declarations are often placed before the declarations of entities and descriptions of their architectures. A library is a file where the compiler takes particular information about design project. A package and module are pieces of the library that contain declarations of often used objects, data types, component declarations, signal, procedures and functions that can be shared among VHDL descriptions. Libraries, packages and modules are declared with the construction

```
library <LIBRARY_NAME>;
use <LIBRARY_NAME>.<PACKAGE_NAME>.<MODULE_NAME>;
```

or with the construction

```
library <LIBRARY_NAME>;
use <LIBRARY_NAME>.<PACKAGE_NAME>.all;
```

where:
- `library` – specific mandatory keywords that defines the structure for library, package and module declarations;
- `use` – name of a library;
- `<PACKAGE_NAME>` – name of a package in the library;
- `<MODULE_NAME>` – name of a module in the library;
- `all` – keyword that lets to utilize all modules of the package.

The mentioned library, package and module declarations are related to the subsequent entity statement. These declarations have to be repeated for each entity declaration.
Let's take the previously described half-adder as an example of entity declaration and architecture description. Example 3 contains a VHDL description of this half-adder made with behavioral (data flow) approach. Then the architecture of this entity contains only one line with logical expression of the half-adder.

Example 3: Half-adder (behavioral approach)

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

-- Entity's Declaration
entity HALF_ADDER is
  port (A, B: in std_logic;
        HS, HC: out std_logic);
end HALF_ADDER;

-- Entity's Architecture
architecture HALF_ADDER_DATAFLOW of HALF_ADDER is
begin
  HS <= (not A and B) or (not B and A);
  HC <= A and B;
end HALF_ADDER_DATAFLOW;
```

In this example «and», «or» and «not» are the keywords of basic logical operations that can be made with declared inputs and outputs of «std_logic» type. More information about them can be found in the corresponding chapter below.
Part III. VHDL programming

VHDL provides constructions for all three mentioned design approaches (Fig. 15): dataflow, behavioural and structural. Below these tools are briefly described and explained with examples.

VHDL Tools for Dataflow Modelling (Concurrent Statements)

VHDL dataflow approach, presented in this section, uses concurrent (parallel) statements to describe behaviour of a digital system. In other words, it describes a digital circuit in terms of its equations and shows how data are converted in the circuit. At the lowest level all elements are described with dataflow approach. With this approach concurrent signal assignments are event triggered and executed as soon as an event on one of the signals occurs.

Direct concurrent (parallel) signal assignments

The most direct way to apply the dataflow approach — is simply use of concurrent (parallel) signal assignments. The syntax of such assignment is the following:

\[ <\text{TARGET\_SIGNAL}> <= <\text{expression}>; \]

where the value of the EXPRESSION is applied to the TARGET\_SIGNAL. As soon as an event occurs on one of the signals that compose the expression it will be re-evaluated. The type of the TARGET\_SIGNAL and the type of the value of the EXPRESSION must be the same.

An example, the simple dataflow approach with the concurrent statements is a 4-bit adder (Example 4). In order to use the addition operator «+» with «std\_logic_array» types the package «IEEE.std\_logic\_unsigned» is attached.

In this example the architecture body consists of three lines. In the first one (Line I) A and B operands, as well as input carry are completed to a 5-bit format and added together. In the second line (Line II) the most significant fit of the obtained 5-bit result forms the output carry. Then, in the third line (Line III) 4 least significant bits of the 5-bit result forms the output sum. Notice that here «&» will be interpreted as concatenation operator (but not as logical AND)

Example 4: Four bit Adder using concurrent assignments

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

-------------------------------- Entity’s Declaration --------------------------------
entity ADDER4 is
  port (A4: in STD_LOGIC_VECTOR (3 downto 0);
        B4: in STD_LOGIC_VECTOR (3 downto 0);
        CINP: in STD_LOGIC;
        S4: out STD_LOGIC_VECTOR (3 downto 0);
        COUTP: out STD_LOGIC);
end ADDER4;

-------------------------------- Entity’s Architecture --------------------------------
architecture ADDER4_DATAFLOW of ADDER4 is
  signal SUMINT: STD_LOGIC_VECTOR(4 downto 0);
begin
  SUMINT <= ('0' & A4) + ('0' & B4) + ("0000" & CINP); -- Line I
  COUTP <= SUMINT(4); -- Line II
  S4 <= SUMINT(3 downto 0); -- Line III
end ADDER4_DATAFLOW;
```
Conditional concurrent signal assignments

The concurrent signal assignment can be conditional if necessary. The syntax for the conditional signal assignment is following:

\[
\text{<TARGET\_SIGNAL>} \leftarrow \begin{array}{ll}
\text{<EXPRESSION1>} & \text{when <CONDITION1>} \text{ else} \\
\text{<EXPRESSION2>} & \text{when <CONDITION2>} \text{ else} \\
& \ldots \\
\text{<EXPRESSIONn>} & \text{else}
\end{array}
\]

or (in the simplest occasion) following:

\[
\text{<TARGET\_SIGNAL>} \leftarrow \begin{array}{ll}
\text{<EXPRESSION1>} & \text{when <CONDITION>} \text{ else} \\
\text{<EXPRESSION2>} & \text{else}
\end{array}
\]

where:

- \text{when else} -- specific mandatory keywords that forms a statement of conditional concurrent signal assignment;
- \text{<TARGET\_SIGNAL>} -- a signal to be modified;
- \text{<CONDITIONk>} -- a Boolean expression that if \text{TRUE} activates the corresponding \text{EXPRESSIONk};
- \text{<EXPRESSIONk>} -- the expression that is assigned to the \text{TARGET\_SIGNAL} if the corresponding \text{CONDITIONk} is of \text{TRUE} value.
- \text{<EXPRESSIONn>} -- the expression that is assigned to the \text{TARGET\_SIGNAL} if none of conditions produces \text{TRUE} value.

Note that if more than one condition is \text{TRUE}, the value of the first expression of \text{TRUE} condition will be assigned.

Let's consider a 4 to 1 multiplexer as an example of conditional signal assignments in which the conditions are formed on the basis of the select signals \text{S0} and \text{S1}. Four valid combinations of «std_logic» signals are directly listed. All other are not valid and produces ‘X’ on the output (in fact the number of combinations is \(8^1=9\)). Notice that the above described statement is useful to express logic function in the form of a truth table.

**Example 5: VHDL script of a 4 to 1 multiplexer (dataflow long)**

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

-- Entity’s Declaration
entity MUX4TO1 is
  port (S1, S0, A, B, C, D: in std_logic;
      Q: out std_logic);
end MUX4TO1;

-- Entity’s Architecture
architecture MUX4TO1_DATAFLOW of MUX4TO1
begin
  Q <=
  A when S1='0' and S0='0' else
  B when S1='0' and S0='1' else
  C when S1='1' and S0='0' else
  D when S1='1' and S0='1' else
             'X';
end MUX4TO1_DATAFLOW;
```
The above example can be presented in a more compact form (Example 6). In order to do that two «std_logic» signals replaced with one «std_logic_vector» that have two elements. With this modification all comparisons become shorter. From other points of view Example 6 is identical to Example 5.

**Example 6: VHDL script of a 4 to 1 multiplexer (dataflow short)**

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

-- Entity's Declaration
entity MUX4T01 is
  port (SEL in std_logic_vector (1 downto 0) A, B, C, D: in std_logic;
  Q: out std_logic);
end MUX4T01;

-- Entity's Architecture
architecture MUX4T01_DATAFLOW2 of MUX4T01 begin
  Q <=
    A when SEL="00" else
    B when SEL="01" else
    C when SEL="10" else
    D when SEL="11" else
    'X';
end MUX4T01_DATAFLOW2;
```

**Selected concurrent signal assignments**

The selected signal assignment is rather similar to the conditional one. Its syntax is the following:

```vhdl
with <CHOICE_EXPRESSION> select
  <TARGET_SIGNAL> <=
    <EXPRESSION1> when <CHOICE1>,
    <EXPRESSION2> when <CHOICE2>,
    ...
    <EXPRESSIONn> when others;
```

where:

- **with select** – specific mandatory keywords that forms a statement of selected concurrent signal assignment;
- **when** – an expression which value defines the particular choice;
- **<TARGET_SIGNALk>** – signals to be modified;
- **<EXPRESSIONk>** – expressions that define the values of the modified signals;
- **<CONDITIONk>** – a scalar value or a range of values that define the particular choice; choices cannot overlap; if «when others» option is not used then the choices must cover all range of possible values of <CHOICE_EXPRESSION>;
- **<EXPRESSIONn>** – the expression that is assigned to corresponding target signal if none of previous choices have been selected.

The previously described example of a 4 to 1 multiplexer can be formed also with the statement «with...select...when...when others» (Example 7).
Example 7: VHDL script of a 4 to 1 multiplexer (dataflow «select»)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
------------------------- Entity’s Declaration -----------------------------------
entity MUX4TO1 is
    port (SEL in std_logic_vector (1 downto 0) A, B, C, D: in std_logic;
             Q: out std_logic);
end MUX4TO1;
------------------------ Entity’s Architecture -------------------------------
architecture MUX4TO1_DATAFLOW3 of MUX4TO1
begin
    with SEL select
    Q <=
        A when "00",
        B when "01",
        C when "10",
        D when "11",
        'X' when others;
end MUX4TO1_DATAFLOW3;

The choices can express not only a single value or a range of values, but also a range or combined choices as it is shown in the following fragment (where «|» is equal to «or»):

```vhdl
target <= value1 when "000",
            value2 when "001" | "011" | "101",
            value3 when others;
```
**VHDL Tools for Structural Modelling (Component Statement)**

With VHDL structural approach the designer of digital circuits can describe them in terms of their components and interconnections. It is assumed that all the components have been already defined (either in a library or in the same file at the same level of hierarchy). They, in turn, can also be described as a structural or behavioural. However, at the lowest hierarchy level each component is described as a behavioural, using the basic logic operators. Structural VHDL approach has an analogy with circuit design in schematic when a hierarchical block hides a circuit composed of components and their interconnections. VHDL provides a formal way to utilize the structural approach. It includes: 4) definition of low level entities – they have to be declared and their architectures have to be defined previously within an attached library or directly in the same file; 2) declaration of components for further use; 2) declaration of signals that define interconnection nets between components; 3) defining unique (and usually multiple) instances of the same component. The components and signals are declared within the architecture body between the keywords «architecture <...> of <...> is» and «begin», but their instances – between «begin» and «end <...>» like it is stated below.

```
architecture <ARCHITECTURE_NAME> of <NAME_OF_ENTITY> is
  --component declarations
  --signal declarations
begin
  --components and their connections
...
end <ARCHITECTURE_NAME>;
```

All the components used in the current level of VHDL design have to be declared before their first use. The component declaration looks similar to entity declaration and consists of component name and the interface (ports). The syntax of the declaration is as follows:

```
component <COMPONENT_NAME> [is]
  [port (<port_name_1>: mode <type>; <port_name_2>: mode <type>; ...<port_name_N>: mode <type>);]
end component [<COMPONENT_NAME>];
```

where:
- **component** `<COMPONENT_NAME>` – specific mandatory keywords that declare a component and its inputs/outputs;
- **is** – a user defined component’s identifier;
- **port** `<<port_name_k>>` – names of component’s inputs and outputs in correspondence with a previously defined entity;
- **mode** – reserved word that defines signal’s direction;
- **<type>** – built-in or user-defined type of the corresponding signal.
As soon as components have been declared they can be used in the architecture body between the keywords «begin» and «end <...>». An instance of the declared component is defined as follows:

\[
\text{<INSTANCE_NAME>} : \text{<COMPONENT_NAME>}
\]

\[
\text{port map} (\text{<port_name_1> } => \text{<signal_name_1>}, \ldots);
\]

where:

- \text{port map} – specific mandatory keywords that defines connections of the instance;
- \text{<INSTANCE_NAME>} – a user defined identifier of the instance;
- \text{<COMPONENT_NAME>} – the identifier of the previously declared component that corresponds to the instance;
- \text{<port_name_k>} – the names of inputs and outputs of the component as they are stated in the component’s declaration;
- \text{<signal_name_k>} – is the name of the signal to which the corresponding port is connected;

In the above example of component’s instance definition its ports are associated to the signals through named association (defined directly). Then it is not important in which order signal and port are associated – only the correspondence of the ports and signal is significant. An alternative method is the positional association:

\[
\text{<INSTANCE_NAME>} : \text{<COMPONENT_NAME>}
\]

\[
\text{port map} (\text{<signal_name_1>}, \ldots, \text{<signal_name_N>});
\]

in which the signal position must be in the same order as the declared component’s ports. It is possible to mix named and positional associations, but all positional associations are placed before the named ones.

![Diagram of half-adder](image)

Fig. 20. Half-adder (structural approach)
The next VHDL script (Example 8) contains a description of the half-adder (Fig. 17-a) made with structural approach (Fig. 20-a). Note that the header the corresponding architecture in this example contains a declarative part that declares the components for further use. In Example 8 there are inverters, two-input AND, as well as two-input OR gates. These components have to be defined first – they must have an entity declaration and architecture description. In this script it is assumed that this information stored in the libraries referred in the header of the script. The declarations of these components proclaim their inputs IN1 and IN2, as well as an output OUT (Fig. 20-b...d).

Note that some of declared interface signals are reserved words IN, OUT and NOT. They cannot be declared as basic identifiers. For this reason they are declared as extended identifiers included in backslashes: \IN\, \OUT\ and \NOT\.

VHDL designer has also to define internal nets of his design or define signal names. In Example 8 the signals are called «A_NOT», «B_NOT», «INT_1» and «INT_2». Notice that the type of the signals has also to be declared.

**Example 8: Half-adder (structural approach)**

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

entity HALF_ADDER is
    port (A, B: in std_logic;
          HS, HC: out std_logic);
end HALF_ADDER;

architecture HALF_ADDER_STRUCTURAL of HALF_ADDER is
    -- Declarations of components
    component AND2
        port (IN1, IN2: in std_logic;
              \OUT\: out std_logic);
    end component;
    component OR2
        port (IN1, IN2: in std_logic;
              \OUT\: out std_logic);
    end component;
    component \NOT\ NOT
        port (IN: in std_logic;
              \OUT\: out std_logic);
    end component;
    -- Declaration of signals used to interconnect gates
    signal A_NOT, B_NOT, INT1, INT2: std_logic;
begin
    -- Component instantiations statements
    U1: \NOT\ port map (A, A_NOT);
    U2: \NOT\ port map (B, B_NOT);
    U3: AND2 port map (A, B_NOT, INT1);
    U4: AND2 port map (B, A_NOT, INT2);
    U5: OR2 port map (INT1, INT2, HS);
    U6: AND2 port map (A, B, HC);
end HALF_ADDER_STRUCTURAL;
```
Example 9: direct assignment of component inputs

architecture HALF_ADER_STRUCTURE2 of HALF_ADER is
  -- Declarations of components
...
begin
  -- Component instantiations statements
  U0: NOT1 port map (IN1 => A, OUT1 => A_NOT);
  U1: NOT1 port map (IN1 => B, OUT1 => B_NOT);
  U2: AND2 port map (IN1 => A, IN2 => B_NOT, OUT1 => INT_1);
  U3: AND2 port map (IN1 => B, IN2 => A_NOT, OUT1 => INT_2);
  U4: OR2 port map (IN1 => INT_1, IN2 => INT_2, OUT1 => HS);
  U5: AND2 port map (IN1 => A, IN2 => B, OUT1 => HC);
end HALF_ADER_STRUCTURE2;

The statements between the «begin» and «end» keywords describe the instantiations of the components and define their connections. These instantiations create a new level of hierarchy. At this level each line starts with an instance name (U1...6) followed by a colon, component name and the keyword «port map», which defines how the components are connected.

In Example 8 this is done through positional correspondence. For example, signal «A» corresponds to the input «IN1» of the gate «U3», «B_NOT» to its input «IN2», but signal «INT_1» to its output. An alternative way assumed explicit association between the ports and signals as it is shown in Example 9.

![Diagram of four bit adder with overflow (structural approach)](image)

Fig. 21. Four bit adder with overflow (structural approach)
Structural design approach itself corresponds to hierarchical one where once defined component can be used in a higher level entity. This reduces significantly the complexity of large designs. In the next example the structural approach is applied to the 4-bit adder (previously designed with dataflow approach). Then the adder (Fig. 21-a) consists of 4 one-bit full-adders (Fig. 21-b) and an overflow generation circuit, which is an XOR gate (Fig. 21-c). In turn, each full adder can be described by the Boolean expressions for the sum and carry out signals as

\[
S = (A \oplus B) \oplus \text{CIN} \quad \text{(7)}
\]

\[
\text{COUT} = A \cdot B + \text{CIN} \cdot (A \oplus B), \quad \text{(8)}
\]

but the overflow generator for k-bit adder – as

\[
V = C_{k-1} \oplus C_k \quad \text{(9)}
\]

that for the 4-bit adder gives

\[
V = C_3 \oplus C_4, \quad \text{(10)}
\]

Equations (7)...(8) provides a good basis for behavioural description of low-level gates. Note that equations that (7) and (8) have the same meaning as (1) and (2) but uses also XOR functions.

In the corresponding VHDL file (Example 10) a component for the full adder is defined first. Later several instantiations of the full adder builds the structure of the 4-bit adder.

Notice that in this example the same identifiers «A», «B», «S», «CIN» and «COUT» are applied to the input ports of the full adder and for the input ports of the 4-bit adder (however, then «A», «B» and «S» are 4-element vectors). Since they refer to different levels of the design this does not generate any error message.

The internal carry signals are defined as a vector «C(4:0)». The input carry signal CIN is the attached to the first input «C(0)», but the last one «C(4)» defines output COUT. Note that VHDL does not allow the use of outputs as internal signals.
Example 10: Four Bit Adder – an example of hierarchical approach

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

-- declaration of the full adder
entity FULLadder is
  port (A, B, CIN: in std_logic;
        S, COUT: out std_logic);
end FULLadder;

-- body of the full adder
architecture FULLadder_DATAFLOW of FULLadder is
begin
  S <= (A xor B) xor CIN;
  COUT <= (A and B) or (CIN and (A xor B));
end FULLadder_DATAFLOW;

library ieee;
use ieee.std_logic_1164.all;

-- declaration of the 4-bit adder
entity ADDER4 is
  port (A, B: in std_logic_vector (3 downto 0);
        CIN: in std_logic;
        S: out std_logic_vector (3 downto 0);
        COUT, V: out std_logic);
end ADDER4;

-- body of the 4-bit adder
architecture ADDER4_STRUCTURAL of ADDER4 is
begin
  signal C: std_logic_vector (4 downto 0);
  component FULLadder is
    port (A, B, CIN: in std_logic;
          S, COUT: out std_logic);
  end component;
  component XOR2
    port (IN1, IN2: in std_logic;
          OUT: out std_logic);
  end component;

  C(0) <= CIN;
  FA0: FULLadder port map (A(0), B(0), C(0), S(0), C(1));
  FA1: FULLadder port map (A(1), B(1), C(1), S(1), C(2));
  FA2: FULLadder port map (A(2), B(2), C(2), S(2), C(3));
  FA3: FULLadder port map (A(3), B(3), C(3), S(3), C(4));
  U0: XOR2 port map (C(3), C(4), V);
  COUT <= C(4);
end ADDER4_STRUCTURAL;
```
In Example 10 the full adder is constructed with dataflow approach utilising its description formulas (7) and (8). It is, however, possible to describe it also with structural approach as a set of AND, XOR and OR gates (Fig. 22). This forms a three level structure, where on the first level are simple gates (AND, OR, XOR), on the second – a full-adder composed of the simple gates and, finally, on the highest third level a 4 bit adder composed of the full-adders and XOR gate.

Example 11: Four Bit Adder – an example of hierarchical approach

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

entity FULLADDER is
  port (A, B, CIN: in std_logic;
        S, COUT: out std_logic);
end FULLADDER;

architecture FULLADDER_STRUCTURAL of FULLADDER is
  component AND2
    port (IN1, IN2: in std_logic;
             \OUT\: out std_logic);
  end component;
  component OR2
    port (IN1, IN2: in std_logic;
             \OUT\: out std_logic);
  end component;
  component XOR2
    port (IN1, IN2: in std_logic;
             \OUT\: out std_logic);
  end component;
  signal AxorB, AandB, INT: std_logic;
  begin
    U1: AND2 port map (A, B, AandB);
    U2: XOR2 port map (A, B, AxorB);
    U3: XOR2 port map (AxorB, CIN, S);
    U4: AND2 port map (AxorB, CIN, INT);
    U5: OR2 port map (INT, AandB, COUT);
  end FULLADDER_STRUCTURAL;
```

Fig. 22. Full adder represented as a set of gates

Continued on the next page...
library ieee;
use ieee.std_logic_1164.all;
-- declaration of the 4-bit adder
entity ADDER4 is
  port (A, B: in std_logic_vector (3 downto 0);
      CIN: in std_logic;
      S: out std_logic_vector (3 downto 0);
      COUT, V: out std_logic);
end ADDER4;
-- body of the 4-bit adder
architecture ADDER4_STRUCTURAL of ADDER4 is
  signal C: std_logic_vector (4 downto 0);
  component FULLadder is
    port (A, B, CIN: in std_logic;
          S, COUT: out std_logic);
  end component;
  component XOR2
    port (IN1: in std_logic;
          OUT1: out std_logic);
  end component;
begin
  C(0) <= CIN;
  FA0: FULLadder port map (A(0), B(0), C(0), S(0), C(1));
  FA1: FULLadder port map (A(1), B(1), C(1), S(1), C(2));
  FA2: FULLadder port map (A(2), B(2), C(2), S(2), C(3));
  FA3: FULLadder port map (A(3), B(3), C(3), S(3), C(4));
  U0: XOR2 port map (C(3), C(4), V);
  COUT <= C(4);
end ADDER4_STRUCTURAL;
VHDL Tools for Behavioural Modelling (Sequential Statements)

**Process and wait statements**

The main element of behavioural VHDL modelling is a process statement that makes possible use of sequential statements to describe the behaviour of a system over time (in quite common algorithmic way). Several process statements inside of the same architecture are executed in parallel (so they are concurrent).

Other statements whose operation is parallel are «block», «generate» and «assert» operators. Since VHDL is excessive language and their operation is more or less similar their description is omitted here.

Like other concurrent statements, a process uses interface signals (input and output ports) to communicate with the rest of the architecture. At the same time the statements inside of the process statement are executed sequentially (like a typical program). It should be noted that variable assignments inside a process are executed immediately and denoted by the “:=” operator. This is in contrast to signal assignments denoted by “<=” and which changes occur after a delay (explicit or default). As a result, changes made to variables will be available immediately to all subsequent statements within the same process.

The syntax for a process statement is following:

```vhdl
[<PROCESS_LABEL>:] process [<sensitivity_list>] [is]
[<process_declarations>]
begin
  <list of sequential statements such as: signal assignments
  variable assignments
  case statement
  exit statement
  if statement
  loop statement
  next statement
  null statement
  procedure call
  wait statement>
end process [<PROCESS_LABEL>];
```

where:

- `process` — specific mandatory keywords that defines a process;
- `is` — a user defined identifier of the process;
- `<sensitivity_list>` — the sensitivity list is a list of the signals that will cause the process to execute; any change in the value of the signals in the sensitivity list will cause immediate execution of the process; the process has to have an explicit sensitivity list or a WAIT statement inside of it; notice that both an explicit sensitivity list and a WAIT statement cannot be included in a process together;
- `<process_declarations>` — the process declarative part declares local variables, signals and their types that can be used only inside of the process.

The keyword «begin» starts the computational part of the process (its body) which is ended with the keyword «end process».
The previously mentioned WAIT statement halt a process until an event occurs. There are several forms of the wait statement:

```
wait until <CONDITION>;
wait for <TIME>;
wait on <SIGNAL1> [,<SIGNAL2> ...];
wait;
```

where:

- `wait until` – specific mandatory keywords that constructs a wait statement;
- `<CONDITION>` – a Boolean expression that if TRUE stops the waiting state;
- `<TIME>` – explicit time limiter for the waiting state;
- `<SIGNALk>` – name of a signal(s) whose changes stop the waiting state.

Below the most typical applications of WAIT UNTIL statement are given:

```
wait until <SIGNAL = VALUE>;
```

The statement waits until the specified signal reaches the specified value.

```
wait until <SIGNAL’EVENT> and <SIGNAL = VALUE>;
```

The statement waits until the specified signal changes and reaches the specified value after the changes. Here «SIGNAL’S’EVENT» is an attribute of the signal that returns the Boolean value TRUE if an event on the signal has occurred. Otherwise it returns Boolean value FALSE.

```
wait until <SIGNAL’STABLE(TIME)> and <SIGNAL = VALUE>;
```

This statement waits until the specified signal stay unchanged for a certain time period and reaches the specified value after this interval; Here «SIGNAL’S’STABLE(TIME)» is an attribute of the signal that returns the Boolean value TRUE if no any event on the signal has occurred during defined <TIME> (default 0); otherwise it returns Boolean value FALSE;

There are some examples:

```
wait until CNTR=10;
```

The statement waits until signal COUNTER reaches value 20.

```
wait until CLK=’0’;
```

The statement waits until signal CLK reaches value ‘0’ (negative transient).

```
wait until CLK’EVENT and CLK=’0’;
```

This statement works as the previous one.

```
wait until CLK’STABLE(10ms) and CLK=’1’;
```

The statement waits until CLK signal stay unchanged for 10ms with value ‘1’.

---

4 Definition and meaning of attributes is not discussed in details in this material.
As it is seen from these example the statement «WAIT UNTIL» is capable of doing functions of other variants of «WAIT» statement. For this reason it is more frequently used.

Let’s discuss a full adder represented as a combination of the half-adders, which schematics are presented in Fig. 16 and which is described by equations (3) and (4). The functional diagram of such adder is presented in Fig. 23, but the corresponding VHDL script in Example 12.

Example 12: VHDL script of an adder composed of two half-adders

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

entity FULL_ADDER is
  port (A, B, CIN : in std_logic;
        S, COUT : out std_logic);
end FULL_ADDER;

architecture FULL_ADDER_BEHAVIOURAL of FULL_ADDER is
  signal INT1S, INT1C, INT2C: std_logic;
begin
  P1: process (A, B)
  begin
    INT1S <= A xor B;
    INT1C <= A and B;
  end process P1;

  P2: process (INT1S, INT1C, INTC)
  begin
    S <= INT1S xor CIN;
    INT2C <= INT1S and CIN;
    COUT <= INT1C or INT2C;
  end process P2;
end FULL_ADDER_BEHAVIOURAL;
```

In Example 12 the process P1 corresponds to the first half-adder, but P2 – to the second half-adder and an OR element forming output carry signal. Both processes are operating in parallel and use internal and external signals of architecture as inputs and outputs. In this example distribution of all expressions in two processes is not necessary and is done for demonstration purposes. Note also that there are no variables in this example (that is not very common). That is why new values to variables are assigned with <=.
**Condition statement**

The syntax of condition statement is the following:

```vhdl
if <condition 1> then <sequential statements 1...
  ...
[elsif <condition k> then <sequential statements k...
  ...
][else <sequential statements N>]
end if;
```

where:
- `if` then
- `else` then
- `end` is
- `<condition 1>` – user defined Boolean expression of the first condition;
- `<sequential statements 1>` – statements that are executed if `<CONDITION1>` produces TRUE;
- `<condition k>` – user defined Boolean expression of another condition;
- `<sequential statements k>` – statements that are executed if all previous conditions produce FALSE, but the current condition `<CONDITIONk>` produces TRUE;
- `<sequential statements N>` – statements that are executed if all conditions produce FALSE.

If all conditions produce a FALSE result, but «else» is not defined then no action is taken.

Let’s discuss a positive edge triggered D flip-flop with asynchronous clear as an example of condition statement used inside of a process (Example 13). In the given example its process statement deals only with interface signals of the architecture (inputs CLK, CLEAR, D and output Q). The process is triggered by CLK or clear signals. The corresponding action is chosen by «if» statement that has two conditions (for output reset and for output update from data input). «CLK’EVENT» is an attribute of CLK signal that returns Boolean value TRUE if an event on the signal has occurred. Otherwise it returns Boolean value FALSE.

**Example 13: VHDL script of a D flip-flop**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
--------------------------- Entity’s Declaration ---------------------------
entity DFF_CLEAR is
  port (CLK, CLEAR, D : in std_logic;
        Q : out std_logic);
end DFF_CLEAR;
--------------------------- Entity’s Architecture ---------------------------
architecture DFF_BEHAVIOURAL of DFF_CLEAR is
begin
  process (CLK, CLEAR)
  variable INT_Q: std_logic;
  begin
    if (CLEAR = '1') then
      INT_Q := '0';
    elsif (CLK'EVENT and CLK = '1') then
      INT_Q := D;
    end if;
    Q <= INT_Q;
  end process;
end DFF_BEHAVIOURAL;
```
Example 14 shows how one IF sequential statements can be placed inside of another one. This rule, of course, is valid for all sequential statements.

**Example 14: 4-bit counter with embedded IF statements**

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

entity COUNTER4 is
   port (CLK, RESET, ENABLE : in std_logic;
         Q : out integer range 0 to 15);
end COUNTER4;

architecture COUNTER4_BEHAVIOURAL of COUNTER4 is
begin
   process (CLK, RESET)
   variable CNT : integer range 0 to 15;
   begin
      if (CLK'EVENT and CLK='1') then
         if RESET = '1' then
            CNT := 0;
         elsif ENABLE = '1' then
            CNT := CNT + 1;
         end if;
      end if;
      Q <= CNT;
   end process;
end COUNTER4_BEHAVIOURAL;
```

**Choice statement**

The case statement evaluates its expression and compares the value to each of the choices. The syntax of choice statement is the following:

```vhdl
case <expression> is
   when <choice 1> => <sequential statements 1>
   ...
   when <choice k> => <sequential statements k>
   ...
   [when others => <sequential statements N>]
end case;
```

where:

- `case is` – mandatory keywords that form a choice statement;
- `when` – a user defined expression for evaluation; the expression must produce a result of an integer or an enumerated one-dimensional array (vector) type like «bit_vector»;
- `when others` – a value (or range of values) produced by the expression that selects the particular branch of sequential statements (next after the given «when» statement); choices must not overlap; if the choice
statement "when others" is not present, all possible values of the expression must be covered by other choices.

<sequential statements k> – statements that are executed if the expression produced the value <CHOICEk>;
<sequential statements N> – statements that are executed if all branches that have choice value have not been chosen.

Let’s take a look at a 4 to 1 multiplexer (Example 15). In the given example the choice is made by a 2 element vector named SEL of «std_logic_vector» type (in contrast to Example 5 where select signals are 2 separate «std_logic» signals). The number of possible combinations of this vector is \(9^2 = 81\) (it is \(9\) – number of possible «std_logic» values in power \(2\) – number of elements in SEL). Four of them (“00”, “01”, “10” and “11”) produce a valid result, but all other possible combinations (like “0X”, “X0”, “OZ”, “ZX” etc.) are covered by statement “when others”.

Example 15: VHDL script of a 4 to 1 multiplexer (behavioural)

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

---------------------------------------- Entity's Declaration ----------------------------------------
entity MUX4TO1 is
  port (SEL: in std_logic_vector(2 downto 1);
        A, B, C, D: in std_logic;
        Q: out std_logic);
end MUX4TO1;

---------------------------------------- Entity's Architecture ----------------------------------------
architecture MUX4TO1_BEHAVIOURAL of MUX4TO1 is begin
  PR_MUX: process (SEL, A, B, C, D)
  variable TMP : std_logic;
  begin
    case SEL is
      when "00" => TMP := A;
      when "01" => TMP := B;
      when "10" => TMP := C;
      when "11" => TMP := D;
      when others => TMP := 'X';
    end case;
    Q <= TMP;
  end process PR_MUX;
end MUX4TO1_BEHAVIOURAL;
```

One more example is a decoder for 7 segment LED display (Example 16). Here its input and output are variables of integer type. All possible values of the input variable «InpData» are processed with «when» statements and no «when others» is necessary. Note also that both input and output variables are binary weighted. For the output variable «OutData» these weights have inverse meaning (obviously, for common anode indicator where segments are driven with active 0V). The least significant weight of the output variable corresponds to segment A, but the most significant – to segment G.
Example 16: VHDL script of a 7 segment LED decoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------------------- Entity's Declaration -------------------------------------------------------------------
entity LED_DECODER is
  port (InpDATA : in integer range 0 to 15;
         OutDATA : out integer range 0 to 127);
end LED_DECODER;
------------------------------------------------------------------- Entity's Architecture -------------------------------------------------------------------
architecture LED_DECODER_BEHAVIOURAL of LED_DECODER is
begin
  process (InpDATA)
  variable TMP : integer range 0 to 127;
  begin
    case InpDATA is
      when 0 => TMP := 64;
      when 1 => TMP := 121;
      when 2 => TMP := 36;
      when 3 => TMP := 48;
      when 4 => TMP := 25;
      when 5 => TMP := 18;
      when 6 => TMP := 2;
      when 7 => TMP := 120;
      when 8 => TMP := 0;
      when 9 => TMP := 16;
      when 10 => TMP := 8;
      when 11 => TMP := 3;
      when 12 => TMP := 70;
      when 13 => TMP := 33;
      when 14 => TMP := 6;
      when 15 => TMP := 14;
    end case;
    OutDATA <= TMP;
  end process;
end LED_DECODER_BEHAVIOURAL;
```

**Loop statements**

Loop statements provide repeated execution of a sequence of sequential statements. Three basic configurations of the loops statements can be emphasized: basic loops (unlimited loops), «while» loops (conditional) and «for» loops (counted). The syntax of a loop statement is the following:

```
[<LOOP_LABEL>]: <iteration scheme> loop
  <sequential statements>
  [next [<LOOP_LABEL>] [when <condition_n>]];
  [exit [<LOOP_LABEL>] [when <condition_e>]];
end loop [<LOOP_LABEL>];
```

where:
- **loop** – mandatory keywords that form a loop statement;
- **end loop** – mandatory keywords that close a loop statement;
- **<LOOP_LABEL>** – a user defined identifier of the loop that appoints its beginning;
- **<iteration scheme>** – defines how the number of loop repetition is limited;
- **next when** – optional keywords that interrupts the current loop sweep and start a new one;
Basic loops have no limitation on the number of their repetitions. They can be interrupted only with special statements «next» and «exit» placed inside of them. The iteration scheme in the basic loop statements does not exist. Therefore their syntax is the following:

```
[<LOOP_LABEL>:] loop
  <sequential statements>
  [next [<LOOP_LABEL>] [when <condition_n>];
  [exit [<LOOP_LABEL>] [when <condition_e>];
end loop [<LOOP_LABEL>];
```

The basic loops must have at least one wait statement. As an example of the basic loop a 5-bit unlimited (when it reaches 31 it starts again from 0) counter is considered (Example 17). A wait statement has been included so that the loop executes every time when the clock changes from ‘0’ to ‘1’. An internal process variable TMP is necessary because output ports of architecture cannot be modified inside the process. Note that the defined basic loop is endless and no NEXT or EXIT operator is necessary.

**Example 17: VHDL script of an unlimited 5 bit counter**

```
library ieee;
use ieee.std_logic_1164.all;
----------------------------------- Entity's Declaration -----------------------------------
entity CNT5U is
  port (CLK: in std_logic;
        COUNT: out integer);
end CNT5U;
----------------------------------- Entity's Architecture -----------------------------------
architecture CNT5U_BEHAVIOURAL of CNT5U is
begin
  COUNTING: process
  variable TMP: integer :=0;
  begin
    COUNT <= TMP;
    loop
      wait until CLK='1';
      TMP:=(TMP + 1) mod 32;
      COUNT <= TMP;
    end loop;
  end process COUNTING;
end CNT5U_BEHAVIOURAL;
```

**Conditional loops** repeats until their condition (Boolean expression) become TRUE. The syntax of a conditional loop statement is the following:

```
[<LOOP_LABEL>:] while <condition> loop
  -- sequential statements
  [next [<LOOP_LABEL>] [when <condition_n>];
  [exit [<LOOP_LABEL>] [when <condition_e>];
end loop [<LOOP_LABEL>];
```

where:
while loop
end loop
<condition>
other elements

– mandatory keywords that form a conditional loop statement;
– Boolean expressions at which if it is TRUE the loop is continued;
– like for basic loop.

Counted loops repeat a certain number of times. The syntax of a counted loop statement is the following:

```plaintext
[<LOOP_LABEL>:] for <IDENTIFIER> in <range> loop
    -- sequential statements
    [next [<LOOP_LABEL>]] [when <condition_n>];
    [exit [<LOOP_LABEL>]] [when <condition_e>];
end loop [<LOOP_LABEL>];
```

where:

for in loop
end loop

– mandatory keywords that form a counted loop statement;

<IDENTIFIER>

– an integer variable local for the loop; it is visible inside the loop and can be used inside of it; requires no declaration;
– defines a range of values of <IDENTIFIER> as well as counting direction; may be either:

```plaintext
<MIN> to <MAX>
```

or

```plaintext
<MAX> downto <MIN>
```

where <MIN> is an expression or integer literal that defines minimum, bit <MAX> that defines maximum of the <IDENTIFIER>;

other elements

– like for basic loop.
Appendix A: Internal structure of Atmel’s SPLD ATF16V8B
Appendix B: Basic Elements of Altera’s MAX3000 CPLDs

Figure 1. MAX 3000A Device Block Diagram

Note:
(1) EPM3032A, EPM3064A, EPM3128A, and EPM3256A devices have six output enables. EPM3512A devices have 10 output enables.

Logic Array Blocks

The MAX 3000A device architecture is based on the linking of high-performance LABs. LABs consist of 16–macrocell arrays, as shown in Figure 1. Multiple LABs are linked together via the PIA, a global bus that is fed by all dedicated input pins, I/O pins, and macrocells.

Each LAB is fed by the following signals:

- 36 signals from the PIA that are used for general logic inputs
- Global controls that are used for secondary register functions
Macrocells

MAX 3000A macrocells can be individually configured for either sequential or combinational logic operation. Macrocells consist of three functional blocks: logic array, product-term select matrix, and programmable register. Figure 2 shows a MAX 3000A macrocell.

Figure 2. MAX 3000A Macrocell

Combinatorial logic is implemented in the logic array, which provides five product terms per macrocell. The product-term select matrix allocates these product terms for use as either primary logic inputs (to the OR and XOR gates) to implement combinatorial functions, or as secondary inputs to the macrocell’s register preset, clock, and clock enable control functions.

Two kinds of expander product terms ("expanders") are available to supplement macrocell logic resources:

- Shareable expanders, which are inverted product terms that are fed back into the logic array
- Parallel expanders, which are product terms borrowed from adjacent macrocells

The Altera development system automatically optimizes product-term allocation according to the logic requirements of the design.
Appendix C: Basic Elements of Altera’s EP2C5 FPGAs

Logic Elements

phase-align double data rate (DDR) signals) provide interface support for external memory devices such as DDR, DDR2, and single data rate (SDR) SDRAM, and QDRII SRAM devices at up to 167 MHz.

Figure 2–1 shows a diagram of the Cyclone II EP2C20 device.

The number of M4K memory blocks, embedded multiplier blocks, PLLs, rows, and columns vary per device.

Logic Elements

The smallest unit of logic in the Cyclone II architecture, the LE, is compact and provides advanced features with efficient logic utilization. Each LE features:

- A four-input look-up table (LUT), which is a function generator that can implement any function of four variables
- A programmable register
- A carry chain connection
- A register chain connection
- The ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnects
- Support for register packing
- Support for register feedback
Figure 2–2 shows a Cyclone II LE.

**Figure 2–2. Cyclone II LE**

Each LE’s programmable register can be configured for D, T, JK, or SR operation. Each register has data, clock, clock enable, and clear inputs. Signals that use the global clock network, general-purpose I/O pins, or any internal logic can drive the register’s clock and clear control signals. Either general-purpose I/O pins or internal logic can drive the clock enable. For combinational functions, the LUT output bypasses the register and drives directly to the LE outputs.

Each LE has three outputs that drive the local, row, and column routing resources. The LUT or register output can drive these three outputs independently. Two LE outputs drive column or row and direct link routing connections and one drives local interconnect resources, allowing the LUT to drive one output while the register drives another output. This feature, register packing, improves device utilization because the device can use the register and the LUT for unrelated functions. When using register packing, the LAB-wide synchronous load control signal is not available. See “LAB Control Signals” on page 2–8 for more information.
Appendix D: Schematic of CPLD Target Board (MAX3064)
Appendix E: Pinout of FPGA Target Board (EC2P5)

<table>
<thead>
<tr>
<th>1st Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_A</td>
<td>139</td>
</tr>
<tr>
<td>2_A</td>
<td>129</td>
</tr>
<tr>
<td>3_A</td>
<td>51</td>
</tr>
<tr>
<td>4_A</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_B</td>
<td>136</td>
</tr>
<tr>
<td>2_B</td>
<td>126</td>
</tr>
<tr>
<td>3_B</td>
<td>55</td>
</tr>
<tr>
<td>4_B</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3rd Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_C</td>
<td>134</td>
</tr>
<tr>
<td>2_C</td>
<td>122</td>
</tr>
<tr>
<td>3_C</td>
<td>57</td>
</tr>
<tr>
<td>4_C</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4th Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_D</td>
<td>133</td>
</tr>
<tr>
<td>2_D</td>
<td>118</td>
</tr>
<tr>
<td>3_D</td>
<td>47</td>
</tr>
<tr>
<td>4_D</td>
<td>43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G/R LEDs</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED1_G</td>
<td>101</td>
</tr>
<tr>
<td>LED1_R</td>
<td>100</td>
</tr>
<tr>
<td>LED2_G</td>
<td>99</td>
</tr>
<tr>
<td>LED2_R</td>
<td>98</td>
</tr>
<tr>
<td>LED3_G</td>
<td>97</td>
</tr>
<tr>
<td>LED3_R</td>
<td>96</td>
</tr>
<tr>
<td>LED4_G</td>
<td>95</td>
</tr>
<tr>
<td>LED4_R</td>
<td>94</td>
</tr>
<tr>
<td>LED5_G</td>
<td>93</td>
</tr>
<tr>
<td>LED5_R</td>
<td>92</td>
</tr>
<tr>
<td>LED6_G</td>
<td>91</td>
</tr>
<tr>
<td>LED6_R</td>
<td>90</td>
</tr>
<tr>
<td>LED7_G</td>
<td>89</td>
</tr>
<tr>
<td>LED7_R</td>
<td>88</td>
</tr>
<tr>
<td>LED8_G</td>
<td>87</td>
</tr>
<tr>
<td>LED8_R</td>
<td>86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Push Buttons</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB1</td>
<td>104</td>
</tr>
<tr>
<td>PB2</td>
<td>103</td>
</tr>
<tr>
<td>PB3</td>
<td>102</td>
</tr>
<tr>
<td>PB4</td>
<td>101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACP with SPI</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>60</td>
</tr>
<tr>
<td>SCLK</td>
<td>61</td>
</tr>
<tr>
<td>SDATA</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gen. 80 MHz</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK_80MHz</td>
<td>91</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.EN</td>
<td>115</td>
</tr>
<tr>
<td>2.EN</td>
<td>114</td>
</tr>
<tr>
<td>3.EN</td>
<td>113</td>
</tr>
<tr>
<td>4.EN</td>
<td>112</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.EN</td>
<td>115</td>
</tr>
<tr>
<td>2.EN</td>
<td>114</td>
</tr>
<tr>
<td>3.EN</td>
<td>113</td>
</tr>
<tr>
<td>4.EN</td>
<td>112</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3rd Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.EN</td>
<td>115</td>
</tr>
<tr>
<td>2.EN</td>
<td>114</td>
</tr>
<tr>
<td>3.EN</td>
<td>113</td>
</tr>
<tr>
<td>4.EN</td>
<td>112</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4th Indicator</th>
<th>FPGA Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.EN</td>
<td>115</td>
</tr>
<tr>
<td>2.EN</td>
<td>114</td>
</tr>
<tr>
<td>3.EN</td>
<td>113</td>
</tr>
<tr>
<td>4.EN</td>
<td>112</td>
</tr>
</tbody>
</table>