This document is meant to be an introduction to VHDL both as a simulation language and an input language for automatic logic synthesis. It is based on material originally prepared for the ASIC Design Laboratory taught at the University of Twente in the years 1993-2002. The text has undergone a major revision in order to be suitable for use in the elective course VLSI System Design.

Before presenting the syntax of the language, first some general background information on top-down design and the design trajectory is presented. The document then continues with a short explanation of the simulation principles that the language assumes. The last part of the document deals with synthesis issues.

Contents

1 VHDL History

2 The ASIC/FPGA Design Trajectory

3 The VHDL Approach to Design

4 Elementary VHDL Language Constructs

4.1 Libraries, Packages, and Entities

4.2 Architectures, Processes, Signals, and Variables
1 VHDL History

The essence of *top-down design* is that one starts with the specifications of a system and goes through a process of step-by-step refinement that culminates in a completed design. A formal language can be quite helpful in that process. It allows to define and document all intermediate design steps plus the final design, leaving no room for misinterpretation. It is possible to use a familiar programming language for that purpose, which is sometimes actually done, but the formal specification of hardware usually works better with a so-called *hardware description language* (HDL).

Many HDLs have been developed in the past, each with its specific strengths and weaknesses. Since these were not standardized and since the average design was less complex than is the case nowadays, the development and use of HDLs often remained an academic issue. This situation has changed, however. At the result of initiatives of the U.S. Defense Department, experts developed an HDL in
the 1980s for use in all military projects. This language was called VHDL, which stands for “VHSIC Hardware Description Language”. (VHSIC in turn stands for “Very High Speed Integrated Circuit”). The language quickly also became popular for non-military applications, especially in Europe. In the United States, the HDL called Verilog is widely used for civilian applications. Both VHDL and Verilog have been accepted as a standard by the IEEE, the Institute of Electrical and Electronics Engineers. VHDL has actually been standardized twice, in 1987 and 1993. There are small differences between the two standards. It is recommended to adhere to the 1993 standard.

Nowadays, many commercial software packages provide support for designing with VHDL. One can even say that VHDL has a key position in the design trajectory as will be shown in more detail in the next section.

2 The ASIC/FPGA Design Trajectory

One way to look at the type of electronic systems that are considered here, is to see them as a mere collection of large numbers of CMOS transistors that are interconnected in a specific way. However, the knowledge of transistors alone is not sufficient to build these systems. Insight in the hierarchical structuring of these systems is necessary for the design of both analog and digital systems.

In the digital domain, one can interconnect transistors to obtain elementary gates such as a 2-input NAND and a D-flipflop. These gates can be combined for building more complex units such as adders, multipliers and registers. These units, on their turn, can be parts of processors. Multiple processors may be required to obtain an entire data processing system. The larger the blocks become, the higher the level of abstraction. For each level of abstraction specific design knowledge is required.

At the highest levels of abstraction, one is hardly aware that hardware is being designed. Only functional relations matter. Designers want to experiment with executable specifications to have an idea of the complexity of the design, the bottlenecks, etc. At this stage simulations using a general-purpose language such as C are often used, although VHDL and specific system-level description languages may be used as well.

In a next stage, properties of hardware, mainly the possibility to perform calculations is parallel have to be dealt with. One should decide about the hardware units to be used and the mapping of computations on the hardware. Two issues have to be settled: on which unit will some calculation take place and when. This is the problem of scheduling. The scheduling problem can either be solved manually or using architectural synthesis tools. The latter often requires applications from a specific area, such as video processing. Tools are already becoming available in which the executable specifications given in C are automatically translated into register-transfer (RT) level VHDL.

At the RT level, the timing of a design is fixed at the resolution of clock cycles: one knows what has to happen from the moment that a register output value changes until new values become available to update the registers in the next clock cycle. If one sees a design as a state machine in which the registers hold the system state, one can say that an RT-level description specifies the “next-state” function.

At this stage logic synthesis can be performed to design the combinational logic that will implement the next-state function. If the input description for logic synthesis is given in VHDL, the process is called VHDL synthesis. Logic synthesis is common practice nowadays and will be covered in detail in later on in this document. A convenient property of VHDL synthesis is that the VHDL code that can
be processed by the synthesis tools, is in principle independent of the target implementation, whether it be an application-specific integrated circuit (ASIC) or a field-programmable gate array (FPGA). Both type of implementations differ at the level of basic building blocks, the so-called standard cells. All available cells are part of a library. The VHDL synthesis tools do not need to know all details of library cells. What matters is the functionality (e.g. 2-input NAND, positive edge-triggered D-flipflop) and the delays associated to the propagation of the signals through the gates.

After logic synthesis, the design will consist of an interconnection of library cells, the so-called netlist. The netlist needs to be processed by backend tools that are specific for the target implementation.

In the case of an ASIC, the backend tools will generate the layout of the entire chip by placing and routing the cells (decide on where to put each cell and determine how the wires between the cells run). The result is a specification of all masks that are needed in the IC production process. As you probably know, the fabrication of an IC is a complex process in which masks are used to selectively etch on silicon, deposit dopants, grow oxide layers, etc.

An FPGA is an integrated circuit itself and is, therefore, produced in the same way. Its main characteristic, however, is that its functionality is electrically programmable. Without going into the details of the different FPGA architectures, it is sufficient to state here that they contain memories (permanent or volatile) that determine the functionality of small logic units (combinational gates of, say, 4 inputs, a single-bit flipflop that may be bypassed, etc.) as well as the way the units are interconnected. Changing the contents of these memories amounts to reconfiguring the FPGA to become a new system.

Backend tools for FPGAs also need to perform placement and routing. As opposed to ASICs where additional space for wiring can be created by pulling cells apart, the wiring capacity in an FPGA is fixed in advance. The routing task is therefore more difficult. The result produced by the backend tools is a specification of the memory contents for the FPGA device. In a prototyping environment the backend tools will transmit the memory patterns directly to an FPGA mounted on a board such that the design can be verified in a practical setting.

Clearly, FPGAs are an ideal platform for prototyping purposes. They are significantly cheaper than ASICs for situations in which the system specifications are subject to change. Once large series of a chip are needed, it becomes profitable to design ASICs. In ASICs the silicon area required for the same functionality is far less, the power consumption is lower and higher operating frequencies may be possible.

In the analog domain, fewer levels of abstraction exist. One can e.g. distinguish current mirrors, amplifiers, etc. that can be used to digital-to-analog (D/A) converter bit cell and combine these cells to obtain a multibit D/A converter. In general, analog circuits are harder to design than digital circuits. As all voltage and current values matter, parasitic capacitors and resistors have to be carefully taken into account during design. Obviously, the state-of-the-art in automatic synthesis in the analog domain is less advanced than for the digital domain.

One can look at top-down design as a process in which gradually more and more detail is added to a specification. The introduction of more detail also involves the risk of the introduction of errors. This is not only true when a human person is in charge of the design, but also when automatic synthesis tools are used. Unfortunately, the synthesis tools themselves, which can be considerably complex, can contain bugs. For these reasons, verification of intermediate design stages by simulation is extremely important.
Analog circuits will in general require full-custom layout. This means that the designer can fully control the shapes of the mask patterns. Composing a circuit by merely placing and routing cells from a library is called semi-custom design. Note that the design of the library cells themselves, is a full-custom activity.

An alternative to simulation is formal verification. Simulation has the strong disadvantage that any nontrivial circuit has too many different input patterns and too many internal states to be exhaustively verified. The goal of formal verification is to reason about circuits in a mathematical way and prove that a detailed design behaves fully according to specification. The necessity to consider all possible input combinations is e.g. avoided in a similar way that mathematical proof does not need to substitute all possible values for variables in an equation. Formal verification tools have for a long time been mainly a topic of academic research. However, some commercial products have become available recently.

Given the importance of simulation in the design process and the many levels of abstractions that exist, VHDL emerges as a powerful language because of it is meant in the first place exactly to support simulations at many levels of abstraction, from the bit level where each separate wire carrying binary signals is distinguished, to the system level at which data types may be used that are not directly related to hardware equivalents. Even more levels can be covered when VHDL-AMS is available: it allows the description of circuits containing analog parts (AMS stands for “analog and mixed-signal”).

3 The VHDL Approach to Design

A number of concepts that were presented during the explanation of the design trajectory in the previous section, are clearly recognizable in VHDL. The most important of these are the following.

- **Behavior versus structure.** A behavioral description of a hardware building block, regardless of whether the block covers the overall design or only a part, strictly documents the relation between the input and output signals. It does not say anything about the division of the block into subblocks. If such a division exists, then we have a structural description. You should note that a structural description not only specifies the subblocks that make up the block, but also the exact interface between the various blocks. In other words, when there is a division into subblocks, a description is needed of the various input and output signals of each block, as well as of how the signal interaction between the different blocks is effected.

- **Hierarchy and abstraction.** The subblocks making up a block that has a structural description, can on their turn have their own structural description. This could go on recursively until we finally come to the elementary or atomic building blocks of the design. In this lab course, for example, these blocks are the elements from the cell library. Under different circumstances the individual transistors might be the elementary building blocks. The recursive division of the building blocks results in a hierarchical description of the design. A concept that is related to hierarchy is abstraction. At a given level in the hierarchy, not all details of the underlying levels are important. By eliminating those details, abstraction enables us to refer to the calculations at a specific level in a meaningful way. It might be useful, for example, to express a calculation at a certain abstraction level in integers, while at a lower level the same calculation might be described in terms of the bits in the binary representation of those numbers.

- **Top-down design.** This design methodology starts with a behavioral description of the overall system to be designed. The system is then subdivided into a number of subblocks. This is called
Figure 1: A block with a purely behavioral description (a), its division into two subblocks (b), a further subdivision (c), and the decomposition tree (d).

decomposition. It results in a structural description at the highest level. Only a behavioral description, however, is available of the subblocks that are referred to in this structural description. These are on their turn divided into blocks with a behavioral description. In this way, a completely structural description is ultimately obtained. A behavioral description is only known for the elementary building blocks. The behavior of the blocks at higher abstraction levels follows bottom-up from the structure.

These concepts are illustrated in Figure 1. In Figure 1(a) the full circuit X is shown with its input and output signals A through D. The first step in a top-down design process is to divide X into its subblocks Y and Z as given in Figure 1(b). Note that the signals on the outside of the circuit are not affected in any way, even though two internal signals E and F have been added. In Figure 1(c) Z is split up further into Z1 and Z2. The recursive split of the design can be reflected in a decomposition tree as shown in Figure 1(d).

The advantage of using VHDL or another hardware description language in a top-down design methodology is that each decomposition step can be verified immediately. This is done by simulating the description before and after decomposition using the same input signals. This approach is used as much as possible during this lab course.

It should be noted that, while simulation is a common and useful tool to verify designs, it does not provide any guarantee of correctness. That is because the number of possible combinations of input patterns for circuits is hardly manageable (except for small and trivial circuits). An alternative for verification through simulation is formal verification. This proves mathematically that a decomposition step preserves the behavior of the circuit. Formal verification is not covered further in the course as this is not applied much in practice (only a few CAD tools). Until now it was assumed that a decomposition step would be performed directly by the designer. It can also be done, however, using CAD tools. This
library ieee;
use ieee.std_logic_1164.all;

entity gcd is
  port (int1, int2: in std_logic_vector(7 downto 0);
   inp_valid: in std_logic;
   clk: in std_logic;
   reset: in std_logic;
   result: out std_logic_vector(7 downto 0);
   ready: out std_logic);
end gcd;

Figure 2: The entity declaration for the GCD circuit.

is called automatic synthesis. If the tools do not produce errors, then verification of the decomposition is not needed. We then speak of correctness by construction.

4 Elementary VHDL Language Constructs

This section deals with a number of important VHDL constructions. This is done with a simple example, namely a circuit that calculates the greatest common divisor (GCD) of two integers.

Note: VHDL does not distinguish between capital and small letters (except in character and string constants). Only small letters are used in this text.

4.1 Libraries, Packages, and Entities

As mentioned in section 3, it is important to define the signals through which a hardware unit communicates with the outside world during the design process. The actual content of the unit, which can consist of behavior or structure, is largely independent from those signals. In VHDL, the specification of communication takes place through the declaration of an entity. Figure 2 presents the declaration of the entity gcd.

All information that is presented in VHDL to a CAD system is supposed to be stored in a library. All libraries have a name that serves as a reference to the library and its contents. The concept of libraries enables designers to organize their design data, to make well-considered use of the data of others, and to store designs and components for later use. The actual design that is being worked on is normally stored in the library work. The designer can also indicate in his VHDL code that he wants to use data from other libraries. The GCD circuit uses the type definitions std_logic and std_logic_vector which are defined in the package std_logic_1164 of the library ieee.

In general, a package contains definitions of data types, procedures, and functions that have been taken together for specific reasons. The package std_logic_1164 contains a nine-valued data type which has been standardized by the IEEE, and functions based on this data type. In addition to the “normal” values ’0’ and ’1’ (for “strong” binary signals), the values that are possible for a signal of this type include ’2’ (for a “tristate” or high-impedant signal), ’X’ for an unknown signal and ’U’ for an uninitialized signal (the remaining values are not relevant for the purposes of this document).
Multiple assignments on the same signal (multiple “drivers” on the same wire) are not permitted in VHDL since the value of a signal is not well defined at the moment when two or more different values are placed on a signal carrier. This restriction is not valid for so-called resolved data types such as std_logic. A resolved data type has a resolution function that maps two or more different values of a certain type on a single value of the same type. Suppose that a bus signal is driven by two sources, one with value ‘2’ and one with value ‘1’. The resolution function will combine these two values into the value ‘1’ for the bus. The combination of ‘1’ and ‘0’, which amounts to a short circuit, however, will result in value ‘X’.

In its simplest form the body of an entity declaration consists of the keyword port, followed by a specification in parentheses of the signals that are used for the communication with the outside world. Input signals are indicated by the keyword in and output signals by the keyword out. In addition, two-way communication can be indicated through the keyword inout.

The input of the GCD circuit consists of two 8-bit integers of the type std_logic_vector (int1 and int2) and the logic signals inp_valid, clk and reset. inp_valid becomes high when the numbers have a valid value such that the calculation can start. reset is necessary to initialize the internal memory elements to a defined value. clk is the clock signal on the rising edge of which the internal memory elements change their values. There are two output signals: an 8-bit integer res that indicates the final value of the calculation and a logic signal ready that indicates that the circuit is ready to receive new values.

Note that the standard type integer is used here as the data type for the operands. The type has an “infinite” range and is therefore not really suited as data type for a signal in hardware. That is because a signal in hardware consists of a finite number of bits (the word length) and thus has limited range. The exact size depends on the representation form that is selected, such as the two-complement representation. Still, it is often customary during the initial phase of the design process not to define the word length to be used (unless the word length is part of the specification). Various considerations during this phase can lead to the insight as to what word length is appropriate or achievable. In the case of the GCD circuit, the type integer is used primarily not to detract attention from the main items through the specific language constructions that VHDL uses when working with a finite number range.

4.2 Architectures, Processes, Signals, and Variables

The interface of the GCD circuit has been specified by the entity declaration, but nothing has yet been said about its content. This is done through an architecture body. Several architectures can be associated to a single entity, each with its own name. A single entity can have one or more behavioral descriptions, as well as one or more structural descriptions, so that descriptions can be available simultaneously at different abstraction levels (see also section 3). In the case of the GCD circuit, the architecture with the name rtl, as presented in Figure 3, is a possible behavioral description. This description is based on Euclid’s algorithm. It states that the GCD of two numbers can be found by repetitively subtracting the smaller number from the larger, and continuing this until two equal numbers are left that are equal to the GCD (check for yourself that this algorithm always gives the correct GCD).

The VHDL code of Figure 3 is synthesizable, which means that it makes use of that subset of the full VHDL syntax that can be automatically mapped onto hardware.

Behavior is specified in VHDL by means of a process, of which an architecture can possess several.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

architecture rtl of gcd is
  signal num1, num2: unsigned(7 downto 0);
  signal num1_next, num2_next: unsigned(7 downto 0);
  signal ready_next: std_logic;

begin
  seq: process(clk)
  begin
    if rising_edge(clk) then
      if (reset = '1') then
        num1 <= (others => '0');
        num2 <= (others => '0');
        ready <= '1';
      else
        if (inp_valid = '1') then
          num1 <= unsigned(int1);
          num2 <= unsigned(int2);
          ready <= '0';
        else
          num1 <= num1_next;
          num2 <= num2_next;
          ready <= ready_next;
        end if;
      end if;
    end if
  end process seq;

  next_val: process(num1, num2)
  begin
    if (num1 > num2) then
      num1_next <= num1 - num2;
      num2_next <= num2;
      ready_next <= '0';
    elsif (num1 < num2) then
      num1_next <= num1;
      num2_next <= num2 - num1;
      ready_next <= '0';
    else
      num1_next <= num1;
      num2_next <= num2;
      ready_next <= '1';
    end if;
  end process next_val;

  result <= std_logic_vector(num1);
end rtl;

Figure 3: Behavioral description of the entity gcd based on Euclid’s algorithm.
A process itself is a *sequential* computation. This means that the statements in the body of a process are carried out in the order in which they appear in the code. The *parallel* nature of hardware expresses itself through the presence of various processes in a single hardware description. The description of a process is quite comparable to that in a traditional programming language such as C. Declarations of variables and constants are followed by the body of a process between the keywords `begin` and `end`. This consists of assignments, if statements, while statements and more.

The architecture of Figure 3 consists of two processes: a process called `seq` that describes the memory elements and a process `next_val` that corresponds to combinational logic for computing the new values of the memory elements.

VHDL distinguishes between *signals* and *variables* (variables do not yet occur in this example). Signals transfer data between different processes. Those that are visible from the outside world are declared after the keyword `port` in an entity. Local signals also exist; these can be stated within an architecture between the keywords `is` and `begin`. A variable, on the other hand, is private to a process and cannot be accessed by any other process. Variables in a process keep their values from one process invocation to the next.

An assignment to a signal is indicated by the symbol `<=`. The value change resulting from the assignment can go into effect either immediately or after a certain amount of time (see also section 6). This last situation is expressed by the keyword `after` that is followed by an expression that must have a result of type `time` and the value of which corresponds with the desired delay. Example:

```vhdl
duml_next <= duml - dnum2 after 5ns;
```

The delay that occurs in hardware is modeled through this mechanism. An assignment to a variable is indicated by the symbol `:=`. The related change always takes place immediately.

Note that the use of the keyword `after` typically belongs to VHDL as a simulation language. The keyword is ignored by logic synthesis tools. Their goal is to take into account the delay of the standard-cell library cells on which they map a design and to find a solution circuit that meets user constraints on delay. It is not their intention to generate hardware that is exclusively meant for delaying signals.

The signal names in parentheses that follow the keyword `process`, form the *sensitivity list*. Each value change in any of the signals in the sensitivity list causes the process to be activated. In the example of Figure 3, the sequential process is only sensitive to the clock signal `clk`. In addition, the assignment to the memory elements only takes place when the clock has a rising edge (any signal change in a logic signal implies either a rising or falling edge). This expresses exactly the fact that the memory elements are supposed to be implemented by *positive edge-triggered flipflops*.

Note that a conditional statement in VHDL is built using the keywords `if`, `then`, `else`, and `end if`. The `else` branch is optional. The keyword `elsif` is available for testing for conditions in decreasing order of priority. An example can be found in the process `next_val`. One can use `elsif` multiple times in an `if` statement. The `case` statement is available for testing on multiple conditions of equal priority. Examples will follow later on in this text.

The second process is a combinational process that computes the new values of the memory elements. The signals occurring in a combinational process can be partitioned in the disjoint sets of input and output signals. Having one signal to be both input and output would imply a feedback and therefore a memory element. In the example, the inputs of process `next_val` are `duml` and `dnum2`. The outputs
are num1_next, num2_next and ready_next. All input signals of a combinational process must occur in its sensitivity list.

4.3 Structural Description: Data Path and Control

In this section make a decomposition of the GCD circuit will be presented. When considering Euclid’s algorithm, we can see that we have to process the two numbers, which calls for two registers, that we need to compare the two numbers, which calls for a “comparator”, and that we must be able to subtract, which must take place at a subtraction unit. Also we need a number of multiplexers to be able to route the signals to the right hardware unit, depending on the state of the algorithm. These considerations can lead to the data path as presented in Figure 4. Note that the “subtract” and “compare” operations are related to each other. They can thus be executed in an alternative (and cheaper) data path on the same hardware unit.

Now that we have introduced the data path, the time has come to introduce and clarify the concept of instantiation, which is closely related to structural descriptions. When describing a hardware unit in VHDL by means of an entity declaration and an architecture, we establish a kind of template for that unit. When we incorporate this particular unit in a larger hardware unit, we can speak of instantiation. The template, with its formal parameters, is then used to create a piece of hardware whose actual parameters are provided by the instantiating environment. The instantiated piece of hardware is called an instance of the template. The data path of the GCD circuit contains, for example, four instances of a multiplexer (see Figure 4). While the four multiplexers are identical in behavior, they each carry out different computations since they are connected to different signals. The instance and signal names that are used in Figure 4 correspond to the names in the fragments of the VHDL code to be presented later on. One of the multiplexers, mux1, is enlarged in the figure to show not only its actual parameters (external naming of the ports), but also its formal parameters (internal naming of the ports).

There is more to the hardware than just the data path. As shown in the figure, the data path contains elements that need a control signal. In particular, the selection signals of the multiplexers and the enable signals of the registers must have the right value at the right moment. There is also the comparator which produces signals, that are not connected within the data path. The same holds for the input signal inp_valid and the output signal ready. All of these signals are connected to the controller. This can best be described as a state machine. A state machine that correctly controls the data path of Figure 4 is shown in Figure 5. Each state is depicted by a circle. The name given to the state is shown in the circle. The output signals belonging to the state, sel1, sel2, sel3, enable1, enable2 and ready, are shown below the state. An output signal with value ‘-’ means don’t care. Note that the multiplexers are controlled in pairs by the same signal. The arrows between the circles indicate state transitions that are allowed. Such a transition only takes place if the condition stated in the diagram is met. It is also assumed that state transitions are synchronized by the system clock, the clock that is also connected to the registers. The circuit enters the starting state start when the reset signal becomes high.

The state machine is of the Moore type, in which the outputs depend solely on the current state and not on the value of the current input signals. There are also Mealy state machines, in which the current value of the inputs influences the output. In the course we will work strictly with Moore machines. These have the advantage that their outputs always change synchronously with the clock.

If it is assumed that the clock and reset signals are generated outside the actual hardware (in the so-called testbench, see Section 5), the structural description of the GCD circuit will contain one or more
Figure 4: A possible data path for the GCD circuit.
instances of:

- `int_reg`, a register that can store integers,
- `int_mux`, a two-input multiplexer for integers,
- `int_sub`, which calculates the difference between two integers,
- `int_comp`, which compares two integers, and
- `controller`, which serves to control the GCD circuit.

### 4.4 More Language Constructs: generic, wait, type, etc.

We will now consider a number of components of the structural description with the goal of introducing new language constructions of VHDL.

The description of the multiplexer is shown in Figure 6. The following characteristics hereof can be mentioned:

- In an entity declaration, the keyword `generic` relates to parameters that can be referred in an architecture body. This mechanism allows that the architecture description is first compiled and stored in a library, and that afterwards, when the entity is instantiated, a value can be assigned to the generic parameters. If several instances exist, each may be assigned its own parameter value. The mechanism is useful because in a top-down design approach the values of physical parameters, such as delay times, are not yet known in early design stages. When these become known later, the entity does not need to be compiled again. If the parameters had been stated as...
library ieee;
use ieee.std_logic_1164.all;

entity int_mux is
  generic (delay: time := 5 ns);
  port (int0, int1: in std_logic_vector(7 downto 0);
    sel: in std_logic;
    out_int: out std_logic_vector(7 downto 0));
end int_mux;

architecture behavior of int_mux is
begin
  out_int <= int0 after delay when sel = '0' else int1 after delay;
end behavior;

Figure 6: The entity declaration and a behavioral description of the multiplexer int_mux.

constants, this would have been necessary. In the case of this example, the mechanism allows to instantiate multiple multiplexers each with its own delay.

Note: this way of using generics only makes sense for simulation models. In synthesis, generics could be used for models that have variable word lengths. However, the synthesis tool needs to be told the value of the generic before it can carry out the synthesis.

• Although a behavioral description is involved here, no process is defined in the architecture. If the behavior can be described exclusively through signal assignments, it is unnecessary to describe a process explicitly. One can say that a signal assignment is a process of itself.

• Conditional signal assignments can be expressed through the keyword when.

• The description is incomplete since not all possible values for signals of the std_logic type are dealt with. As mentioned earlier, std_logic is a nine-valued type. A complete description would have to evidence appropriate related behavior for all possible values of the signal sel instead of merely distinguishing between equal or unequal to '0'.

Next we consider an entity clock as described in Figure 7, for an explanation of the wait statement. This statement effectuates that the execution of a process is interrupted during a time indicated by the value after the keyword for. In this case, the two wait statements ensure that the output signal clock repetitively changes value after a given time. Note that the mechanism is quite different from that of a delayed signal assignment that uses the keyword after. In the latter case, the process goes on without interruption while the assignment takes effect later, whereas when using wait the execution of the process is interrupted for a certain period of time. The use of wait cannot be combined with the use of a sensitivity list (see Section 4.2). A process with a sensitivity list is not allowed to have a wait statement in its body. A process without a sensitivity list, on the other hand, needs to contain a wait statement for each “path” through the code in order to guarantee that the process is not started up again immediately after it has been gone through. In such case, the other processes within the simulation would not be executed anymore.

The last component of the GCD circuit that we examine, is the controller that corresponds with the state machine in Figure 5. Its entity declaration and part of the architecture is shown in Figures 8.

In describing the state machine, the various states have been defined by means of a type declaration.
library ieee;
use ieee.std_logic_1164.all;

entity clock is
  generic (lowperiod: time := 10 ns; highperiod : time := 10 ns);
  port (clock_out : out std_logic);
end clock;

architecture behavior of clock is
begin
process
begin
  clock_out <= '0';
  wait for lowperiod;
  clock_out <= '1';
  wait for highperiod;
end process;
end behavior;

Figure 7: The entity declaration and architecture description of clock.

library ieee; use ieee.std_logic_1164.all;

entity gcd_controller is
  port (inp_valid, reset, equal, greater, clock: in std_logic;
    sel1_2, sel3_4, enable1, enable2, ready: out std_logic);
end gcd_controller;

architecture behavior of gcd_controller is
  type state is (start, read_input, compare, greater1, greater2);
  signal current_state: state;
begin
  new_state: process (clock)
  begin
    if rising_edge(clock) then
      if reset = '1' then
        current_state <= start;
      else
        case current_state is
        when start =>
          if inp_valid = '1' then
            current_state <= read_input;
          else current_state <= current_state;
          end if;
          -- The remaining entries have been omitted.
        end case;
      end if;
    end if;
  end process new_state;

  outputs: process(current_state)
  begin
    case current_state is
    when start =>
      sel1_2 <= '1'; sel3_4 <= '1';
      enable1 <= '0'; enable2 <= '0'; ready <= '1';
      -- The remaining entries have been omitted.
    end case;
  end process outputs;
end behavior;

Figure 8: Partial description of the controller.
architecture structure of gcd is
  component int_mux
    port (int0, int1: in std_logic_vector(7 downto 0);
           sel: in std_logic;
           out_int: out std_logic_vector(7 downto 0));
  end component;

  component gcd_controller
    port (inp_valid, reset, equal, greater, clock: in std_logic;
          sel1_2, sel3_4, enable1, enable2, ready: out std_logic);
  end component;

  signal mux1_out, mux2_out, mux3_out, mux4_out,
          reg1_out, reg2_out, sub_out: std_logic_vector(7 downto 0);
  signal equal, greater, sel1_2, sel3_4, enable1, enable2: std_logic;

begin
  mux1: int_mux
    port map (int1, sub_out, sel1_2, mux1_out);
  mux2: int_mux
    port map (sel => sel1_2, int0 => int2, 
              int1 => sub_out, out_int => mux2_out);
  reg1: int_reg
    port map (mux1_out, enable1, reset, clk, reg1_out);
  reg2: int_reg
    port map (mux2_out, enable2, reset, clk, reg2_out);
  compare1: int_comp
    port map (reg1_out, reg2_out, greater, equal);
  mux3: int_mux
    port map (reg1_out, reg2_out, sel3_4, mux3_out);
  mux4: int_mux
    port map (reg2_out, reg1_out, sel3_4, mux4_out);
  sub1: int_sub
    port map (mux3_out, mux4_out, sub_out);
  cont1: gcd_controller
    port map (inp_valid, reset, equal, greater, clk, 
              sel1_2, sel3_4, enable1, enable2, ready);
  result <= reg1_out;  -- not a purely structural description here
end structure;

Figure 9: Part of the structural description of the GCD circuit.

Types can be declared locally, within an architecture or process, if they are only needed locally. If that is not the case, they can also be defined in a package.

The architecture contains two processes: a combinatorial one that computes the outputs given the current state (outputs), and a sequential one (new_state) that computes the new values of the state variables. This is very similar to the setup used in Figure 3. An alternative description would use a third combinational process that would compute the new state from the current state while the sequential process would be reduced to a mere assignment of the new state to the current state (and the check for the rising clock edge and reset).

4.5 Instantiation and Configuration

We have now come to the architecture of the structural description of the GCD circuit. A fragment corresponding with the data path of Figure 4 is shown in Figure 9. Components and signals are declared in the declaration part of the description (before the keyword begin). Note that component declarations
configuration combine of gcd is
for structure
for all: intmux use entity work.int_mux(behavior);
end for;
for all: intreg use entity work.int_reg(behavior);
end for;
for compare1: intcomp use entity work.int_comp(behavior);
end for;
for sub1: intsub use entity work.int_sub(behavior);
end for;
for cont1: gcd_controller use entity work.gcd_controller(behavior);
end for;
end for;
end combine;

Figure 10: The configuration that fully specifies the structural description of the GCD circuit.

strongly resemble entity declarations. In the body of a structural architecture, the part of the code that comes after the keyword begin, the components that have been declared in this way are instantiated one or more times. The declared signals serve to connect the instances from the body. During instantiation, an instance is connected either with an internal signal or with one of the input or output signals. Each instance is given a name in the body. This instance name is the label that precedes the component name. The instance name is referred to during the “configuration” of the hardware (see below).

Note: The component declaration of int mux does not refer to the entity depicted in Figure 6, but to a version without generics (otherwise, the generic declaration should be included here as well).

During instantiation, the keywords port map precede a list with signals; these “actual” signals are linked to the “formal” signals from the component declaration. This can be done in two ways. The first method uses an argument list separated by commas using the same ordering as the formal signals such as in mux1 in Figure 9. The second method uses an association list in which formal signals are explicitly linked with the actual signals, such as in mux2 in Figure 9. When this method is used, the ordering of signals can be arbitrary. Similarly, generic parameters can be assigned a value during instantiation by means of a construction that is preceded by the keywords generic map.

Lastly, note that the description in Figure 9 contains a signal assignment that expresses behavior although it is used in a structural description: res <= reg1 out. A pure structural description would require an additional component (and related entity) that connects the two signals to each other and that would be instantiated in the architecture body. However, that is rather cumbersome. It is also hardly appropriate to leave out the internal signal reg1out and replace it with res. That would detract from the fact that textttres is an output signal and cannot act as an input for certain components without introducing changes to the entity declaration.

While it may appear at first sight that a description such as in Figure 9 contains all information needed for a structural description, that is not the case. The component declarations may establish a link with the entities, but an entity generally has more than one architecture. The structural description must indicate which of the architectures needs to be instantiated for the purpose of simulation. This specification is achieved by the declaration of a configuration. For the GCD circuit such a declaration might look like Figure 10. The outer for statement indicates that it concerns the configuration of the architecture structure of the entity gcd. The other for statements establish a link between an instance name and an entity-architecture combination by supplying the architecture name between
parentheses after the entity name. If all instances of a type have the same architecture, then this is indicated by the keyword all. Note that the library work is explicitly referred to. All entities must be present in this library in compiled format. Note also that a configuration declaration in VHDL can be omitted if all instantiated entities have a single architecture in work.

5 The Testbench Concept

As already mentioned several times, VHDL modeling (or hardware modeling in general) has at least two uses: simulation for the purpose of verification and synthesis for the automatic transformation of a relatively abstract description into a collection of gates from a library. The entire model of the hardware that one wants to build is called the design under verification (DUV).

A VHDL simulator has various features to control the simulation. A user can indicate the time stretch that the simulation should cover, the sequence of test signals or stimuli that should be provided to the DUV, etc. In spite of these facilities, it is a better idea to control the simulation as much as possible from VHDL itself. The advantage of this is that it requires only minimal knowledge of the simulator and that one becomes independent of the simulator.

The models that describe an environment for the DUV form the so-called testbench. It is recommended to build a testbench that at least consists of the following models:

- A top-level entity without any inputs or outputs.
- A “test-vector generator” entity that has I/O ports that are exactly complementary to those of the DUV. So, the entity has outputs for each input of the DUV and can provide appropriate signals in this way.

More complex testbenches may have more than one entity to generate inputs for the DUV or process its outputs.

The idea of a testbench is illustrated in Figure 11 that depicts the two entities mentioned above for the case of the GCD circuit. The entity testvec_gcd will typically take care of clocks and resets as well as the regular data processing. It is a good habit to read an input data stream for the DUV from a file and write the output data stream to a file (or compare the outputs with a reference output stream stored in a file). In this way, one can experiment with different I/O streams without needing to recompile the models. One can also stop the simulator by means of a VHDL assert statement (such a statement instructs the simulator to interrupt simulation and print an error message).

VHDL’s configuration mechanism especially shows its power in the context of a testbench. The different DUVs that a designer creates throughout the design process should behave the same when simulated in the same testbench. One does not need to modify the testbench models. Instead one writes a separate configuration for each DUV version that one wants to simulate. Configurations for the two versions of the GCD DUV presented earlier are shown in Figure 12. Note that a configuration can be composed of entity-architecture combinations or other configurations.
library ieee;
use ieee.std_logic_1164.ALL;

entity tb_gcd is
end tb_gcd;

architecture structural of tb_gcd is

component gcd
    port (int1, int2: in std_logic_vector(7 downto 0);
          inp_valid: in std_logic;
          clk: in std_logic;
          reset: in std_logic;
          result: out std_logic_vector(7 downto 0);
          ready: out std_logic);
end component;

component testvec_gcd
    port (int1, int2: out std_logic_vector(7 downto 0);
          inp_valid: out std_logic;
          clk: out std_logic;
          reset: out std_logic;
          result: in std_logic_vector(7 downto 0);
          ready: in std_logic);
end component;

signal clk, reset, inp_valid, ready: std_logic;
signal int1, int2, result: std_logic_vector(7 downto 0);

begin
    gc: gcd
    port map (clk => clk, reset => reset, inp_valid => inp_valid,
              int1 => int1, int2 => int2, result => result, ready => ready);
    tv: testvec_gcd
    port map (clk => clk, reset => reset, inp_valid => inp_valid,
              int1 => int1, int2 => int2, result => result, ready => ready);
end structural;

---

6 The Operation of the VHDL Simulator

Before we introduce the VHDL description of the dalmemo, it is useful to have a brief look at how the VHDL simulator works. We will confine ourselves to the most important aspects, even though much more can be said about the structure of the VHDL simulator and about simulation techniques in general.3

Part of the information below has already been discussed earlier in the text. It is repeated and expanded on here in the hope that further insight arises into the operation of the simulator.

The simulator regards a circuit as a collection of signals and processes. Signals can change in value over time under the impact of processes. A signal change is called a transaction.

Although hardware is parallel by nature, it is generally simulated on a sequential machine. In one way or the other, processes that are active simultaneously, as well as signals that can change in value

---


© Sabih H. Gerez, University of Twente, The Netherlands January 19, 2004
Figure 12: Two testbench configurations for the GCD circuit.

simultaneously, must be dealt with in such a way that the differences between simulation and the real world are as small as possible.

Section 4.2 already stated that a process must either have a “sensitivity list”, meaning that its body is evaluated once each time when one of the signals in the list changes in value, or it must contain wait statements. A process that involves wait statements is immediately started up again when its entire body has been gone through, but the evaluation is stopped when a wait statement is encountered. We thus see that both types of processes involve a finite simulation time. When a process is inactive, the simulator has the possibility to evaluate another process. A process that has neither a sensitivity list nor a wait statement is hardly meaningful: once activated it no longer becomes inactive and fully occupies the simulator.

What the simulator must do at a given moment is indicated through a list of actions that is sorted by time. This is the event list. These actions indicate that, at the related moment, a signal must change value or that a process must be activated at that moment. For example, if the process that is active at moment t = t₀ encounters the statement a <= '1' after 10 ns, then transaction <= '1' is placed on the event list at moment t = t₀ + 10 ns. A transaction never takes effect immediately, not even if the process does specify any delay (for example, through a signal assignment without the keyword after). In that case, the transaction is placed on the event list before moment t = t₀ + Δ. Δ is equal to zero (or better: infinitesimally small), but it allows processes that take place simultaneously to be ordered in time. This is possible because the following applies: 0 < Δ < 2Δ . . .

The simulation starts with the construction of the event list. All processes in the VHDL description are placed in the right position in the list. (Most processes start at time zero, applying the rule that a minimal time of Δ must occur between two activations.) During simulation, the event list is processed in the order of increasing time. New events that result from this are added in the event list at the right position. The simulation is ended when the event list becomes empty, when the simulation is forced to
be terminated by the initiative of the user or by an error.

Using an event list allows reduction of the calculation work to be performed by the simulator. Processes are evaluated only when this is strictly necessary. This method is called the *event-driven* simulation technique. It is used in one way or other by practically all digital simulators.

## 7 VHDL Synthesis Basics

It has been mentioned already that VHDL was primarily designed for purposes of *simulation*, but that more and more tools are becoming available that can *synthesize* (some subset of) VHDL. Synthesis means here that a VHDL description provided by the user is taken as the *specification* of the hardware and mapped to either an IC or an FPGA layout that shows the same behavior as the specification.

One can say that the synthesis tools perform *silicon compilation*. In a way similar to the process of software compilation where the specification of some computation in some high-level language such as C++ or Java is automatically translated into machine instructions, a silicon compiler translates a high-level specification of hardware behavior into a set of mask patterns on chip that realizes the desired behavior (or into a configuration pattern of an FPGA).

With some simplification, the VHDL synthesis process can be seen as consisting of first deriving Boolean equations from the VHDL code and then optimizing these equations such that they can be realized with the standard cells from a given library. The remaining part of this text presents typical examples of VHDL code that can be synthesized. Because of its intricacy, some additional attention is paid on how to specify arithmetic circuits in synthesizable VHDL.

VHDL is a standardized language. Standardization efforts for a subset of VHDL for synthesis have led to first results: the data types to be used in synthesis. This means that an interpretation of the std_logic data type (and other types not covered here) has been fixed for the purpose of synthesis. New standards to define the unambiguous hardware equivalents of subsets of VHDL are in preparation. In practice various synthesis tools support almost the same VHDL language subset.

One of the main lessons of this text is that VHDL can be the core of an IC design project. One starts with a formal VHDL description of the behavior of the circuit to be designed. It can be verified through simulation. This “executable specification” can be refined using a top-down design approach until a VHDL description is obtained that can be synthesized, while at the same time simulation is used to continually verify the correctness of the description. After VHDL synthesis, the resulting netlist of standard cells can again be described in VHDL. It will, of course, be a structural description where instances of standard cells are interconnected. Behavioral descriptions of the individual standard cells themselves are given in the library. This final VHDL description of the design can again be simulated using the original testbench. There are two reasons for simulating the final description. First of all, the final description will contain timing information based on a realistic modeling of delays. It may turn out that the circuit does not work properly due to timing problems. They may be solved by a revision of the design. A second reason for postsynthesis simulation is that the synthesis tools cannot always be trusted; due to the complexity of the algorithms, bugs may exist in the software.
8 VHDL Synthesis Through Examples

As was mentioned before, only a subset of VHDL can be synthesized by commercially available synthesis tools (this subset is different across the tools; however, some standardization exists). Because of its intricacy, it is not possible to present here the exact subset. Instead, an even smaller subset that is sufficient to complete the design of the dialmemo, will be informally defined here.

This section will first give some characteristics of the VHDL subset to be used and then explain the subset by means of some examples.

8.1 General Remarks on Synthesizable VHDL

These are the main properties of the synthesizable subset of VHDL:

- Only a single architecture for each entity to be synthesized is allowed. A second architecture presented to the system will result in the first one to be ignored. Configurations do not make sense because no confusion between multiple architectures is possible.

- The architecture of an entity can either be a behavioral one or a structural one composed of instantiations of other entities. So, hierarchical descriptions can be used. Multiple entities per file are allowed.

- Behavioral descriptions of an entity will have one or more processes in the architecture body. It is a good custom to separate combinational and sequential logic into separate processes. Examples are given later on.

- Synthesizable VHDL should not contain references to absolute time such as in assignments with the after keyword. Signals can be delayed, but only using synchronization mechanisms with respect to some clock signal.

- Although the synthesizer can deal with many data types, it is strongly recommended to exclusively use the std_logic and std_logic_vector data types for the I/O signals of the top-level entities. These are namely the data types used in the VHDL descriptions of the synthesized circuits. Sticking to them facilitates the reuse of testbenches.

8.2 Combinational Logic at the Bit Level

In Table 1 an example of a function with 3 inputs and 2 outputs is given. The synthesizable VHDL equivalent of such a function is shown in Figure 13. It is the synthesizable VHDL equivalent of the truth table given in Table 1. As can be seen from the VHDL description, the code has a one-to-one correspondence to the truth table. The example teaches a few points that are valid for VHDL synthesis in general:

- Processes that represent combinational logic, have a sensitivity list that should contain all inputs of the hardware unit.

---

Table 1: An example of a Boolean function with 3 inputs and 2 outputs.

<table>
<thead>
<tr>
<th>Input $x_1x_2x_3$</th>
<th>Output $y_1y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
</tr>
<tr>
<td>010</td>
<td>01</td>
</tr>
<tr>
<td>011</td>
<td>01</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>101</td>
<td>1D</td>
</tr>
<tr>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>111</td>
<td>D1</td>
</tr>
</tbody>
</table>

Table 1: An example of a Boolean function with 3 inputs and 2 outputs.

Figure 13: Truth-table style specification of combinational logic.

- The data types `std_logic` and `std_logic_vector` that are used widely for simulation, are also synthesizable. All value combinations with '0' and '1' for the input signals should be specified in the VHDL description. Specifying the behavior for input signal values other than '0' and '1' does not make sense for synthesis, but is necessary for the simulation of the description prior to synthesis: hence, the `others` clause in the `case` statement of Figure 13. This clause is ignored by synthesis tools. The value ' - ' for don’t care signals can be used for output values to allow the logic synthesis algorithms to minimize the hardware. Other data types that can be used for signals in VHDL synthesis will be discussed later.

- Full specification of all input value combinations is important. According to VHDL semantics, signals that are not assigned during a process invocation maintain their values. For synthesis this would mean the insertion of latches to keep the old signal value for the unspecified input
library ieee;
use ieee.std_logic_1164.all;

entity example2 is
port (x: in std_logic_vector (1 to 3);
y1: out std_logic);
end example2;

architecture behavioral of example2 is
begin
react: process (x)
begin
if ((x(1) = '1') and (x(3) = '0')) or (x(2) = '0')
then
y1 <= '1';
elsif (x(1) = '1') and (x(2) = '1') and (x(3) = '1')
then
y1 <= '-';
else
y1 <= '0';
end if;
end process react;
end behavioral;

Figure 14: An alternative style for synthesizable VHDL at the bit-level.

combinations. This would make the hardware unit sequential instead of combinational.

Truth tables are not the only possibility to describe synthesizable VHDL at the bit level. The signal $y_1$ in Table 1 can e.g. be described as given in Figure 14. An important remark to be made about this example is that the boolean data type of VHDL should not be confused with the data type std_logic. The conditional expression of the if statement should evaluate to boolean. Although the package std_logic_1164 provides for the use of the operators such as and and or with values of the type std_logic, one cannot replace the first conditional expression by: $(x(1) \text{ and not } x(3)) \text{ or } \text{not } x(2)$. The results returned by the operators and, etc. are themselves of the type std_logic and not of the type boolean. Note: VHDL has the possibility of operator overloading as is e.g. the case in C++; this allows the use of the operators and etc. for data types other than boolean.

An example of a synthesizable combinational logic at the word level is given in Figure 15. The code describes a hardware unit that computes the exclusive or of two 12-bit signals after inverting the first signal depending on a control signal.

8.3 Sequential Logic: A Finite State Machine

As opposed to combinational logic, hardware units with sequential logic have an internal state and the output values of the unit not only depend on the actual input values but on the state as well. Any piece of sequential hardware that can be physically built, has a finite number of states (a finite number of logic gate outputs) and can therefore be called a finite state machine (FSM). The term FSM is often used for hardware in which the number of states is small such as in the example of Figure 5.

The discussion of VHDL synthesis for FSMs is limited to hardware in which states are stored in flipflops that are connected to a single clock. Figure 8 shows illustrative parts of the synthesizable description of
VHDL for Simulation and Synthesis

library ieee;
use ieee.std_logic_1164.all;

display entity cond_xor is
  port (a, b: in std_logic_vector(11 downto 0);
       c: in std_logic;
       result: out std_logic_vector(11 downto 0));
end cond_xor;

display architecture behavioral of cond_xor is
begin
  react: process (a, b, c)
  begin
    if c = '1' then
      result <= a xor b;
    else
      result <= (not a) xor b;
    end if;
  end process react;
  end behavioral;

Figure 15: The synthesizable description at the word level of a hardware unit.

display the controller given in Figure 5.

The following remarks related to synthesis can be made about the description:

- The description contains two processes: one sequential process new_state in which the next state is computed and one combinational process outputs for the computations of the outputs from the current state.

- The sequential process is only sensitive to the clock. The function rising_edge applied to a signal of the type std_logic returns the value true only if the signal makes a transition from '0' to '1'. So, what is actually described in VHDL is that signals in the sequential process only change value on the rising edge of the clock. The interpretation of this behavior for synthesis is that all signals that change value within the then part of the if statement, should be stored in flipflops. Because current_state is the only signal in the then part in the example to which an assignment is performed, flipflops will be created to store the state (the number of flipflops depends on the state assignment strategy used; in the default case, N flipflops will be used such that \( N = \lceil 2^{\log("number of states")} \rceil \)).

- As the FSM is of the Moore type, the outputs depend only on the current state. This is very clear in the process outputs which is only sensitive to the signal current_state.

Important note: Only synchronous sequential hardware is considered here. This means that all sequential processes in a synthesizable VHDL description can only be sensitive to the clock. Such processes are repetitively activated at the start of each new clock period. This means that any action should terminate within a single clock period. Iterative constructs that use e.g. while statements with the goal of performing actions that span multiple clock periods are taboo. The correct way of dealing with such actions is to introduce appropriate state variables that store the state until the next activation of the process in the next clock period (see e.g. the counter example of Section 9.3 where a state variable is incremented in each process activation). Iteration is implicit through of the periodic nature of the clock signal rather than explicit through the use of iterative language constructs.

© Sabih H. Gerez, University of Twente, The Netherlands January 19, 2004
8.4 Assignment of Multibit Signals

Another issue that may be useful in the design of hardware and has not yet been discussed here is the interconnection of multibit signals with unequal length. Different possibilities allowed by VHDL are shown in Figure 16. Two signals can be juxtaposed to form a wider signal by means of the signal-concatenation operator “&”. This is not only useful for composing buses. It can also be convenient in order to have compact and readable code as the wider signal can e.g. be used in a single case statement; multiple nested if or case statements would otherwise be necessary to describe the same.

One can also select a range of bits of a multibit signal both to be used at the left and right hand side of an assignment.

9 Data Types and Functions for VHDL Synthesis

In the previous section VHDL synthesis has been introduced by means of examples. In this section, additional information will be given regarding the data types standardized by the IEEE for synthesis.

9.1 Data types

In all examples given above, signals were either of the type std_logic or std_logic_vector. These types are defined in the package std_logic_1164 of the library ieee. Without any further measures, these data types can only be used in expressions involving Boolean functions such as not and xor. If one wants to use them as arguments for arithmetic functions, other data types should be used as explained further on.

A data type that is built into VHDL, is integer. After synthesis, signals of this type will be 32 bits wide (for most tools). VHDL allows, however, to constrain the range of integers. If one e.g. knows that some signal x will never be assigned a value greater than 10 and lower than 0, one can declare it as:
signal x: integer range 0 to 10. This mechanism will result in hardware that uses 4 bits instead of 32 after synthesis. The use of the data types signed and unsigned that are explained below, are to be preferred above integers as they force the designer to be better aware of the number of bits used.

A bit vector of the type std_logic_vector can, of course, represent a number. As you undoubtedly know, there are many different ways to encode a number as a bit vector. The IEEE standard for VHDL synthesis, therefore, defines two new types that are both defined to be arrays of std_logic. These are the types unsigned and signed. Bit vectors of the first type should be interpreted as positive integers whereas those of the second type require an interpretation of integers with a two's complement encoding. They are defined in the package numeric_std that is stored in the library ieee. This package should always be declared when signals or variables of type unsigned or signed are used (see later on for an example).

The hardware counterpart of a signal of type std_logic is a wire. The three array data types based on std_logic, viz. std_logic_vector, unsigned and signed all correspond to a set of wires (a bus) in hardware. VHDL knows that the three types are all arrays of the same type. Although type checking prevents that signals or variables of different types can directly be assigned to each other, a “casting” mechanism is available. Suppose, e.g. that a has type std_logic_vector and b has type unsigned and the same width, then the following assignments are legal:

\[
\begin{align*}
a & \leftarrow \text{std_logic_vector}(b); \\
b & \leftarrow \text{unsigned}(a);
\end{align*}
\]

In VHDL, the types match correctly while in hardware nothing happens: the set of wires stays the same; only the interpretation of the signals that they carry changes.

9.2 Functions

VHDL has the property found in many object-oriented languages that a function can be applied to different data types and that the computation performed depends on the data types of the arguments. Using so-called overloading, also existing infix operators, such as and and + can be made applicable to newly defined data types.

The package std_logic_1164 of the library ieee defines the functions and, or, not, nand, nor and xor that are infix operators for two operands of the type std_logic or two operands of the type std_logic_vector (having the same length!). The resulting value has the same data type as the two operands. Example: if x, y and z are two signals of the type std_logic_vector with length 10, z \leftarrow x \text{ nand} y; will compute the bitwise NAND of x and y and assign it to z.

Among the many other things defined in the package std_logic_1164 of the library ieee, the “strength stripper” function to_X01 may be of use for the design of the dialmemo. It converts the nine possible values of std_logic to either 'X', '0' or '1'. This may be useful, e.g. to convert the “weak zero” 'L' that is produced by the keyboard, to a “strong zero” '0' in order to be able to write synthesizable VHDL (remember that only the values '0', '1' and '-' are meaningful for synthesis). Example: if x and y are of the type std_logic and the value of 'x' equals 'L', y \leftarrow \text{to_X01}(x); will assign the value '0' to 'y'. The synthesis tool will just connect x and y with a wire.
The package `numeric_std` located in the `ieee` library contains many useful functions related to the use of the data types `integer` and `unsigned` (and type `signed` as well; however, no more attention to `signed` will be paid as the design can entirely be designed without reference to negative numbers). A selection of these will be mentioned here:

- **to_integer** takes an `unsigned` as its operand and returns the corresponding integer value. Example: if `x` is of the type `unsigned` and has value "1010", `to_integer(x)` will evaluate to 10.

- **to_unsigned** is the reverse function and takes two integer operands, the first being the one to be converted to a vector and the second the length of the vector (the number of bits). Example: if `x` is of the type `integer` and has value 10, `to_unsigned(x, 5)` will evaluate to "01010".

- The infix operators `+` (addition), `-` (subtraction), and `*` (multiplication) are defined for two operands of type `unsigned`. Either of the operands can also be of the type `integer`.

- The following relational operators are defined for two operands of type `unsigned`: `=, /=, >=, <=, >` and `<`. All return the type `boolean` and can therefore be used in e.g. the condition of an `if` statement.

- Of course, all arithmetic operators just mentioned are also applicable to the type `integer`. However, the functions are not part of the two packages mentioned here, but are built into VHDL itself. For synthesis, the restriction for the second operand of division, etc. also applies to the type `integer`.

- The infix operators `/` (division), `mod` (modulo) and `rem` (remainder) are only supported when their operands are of type `integer` and when the second operand has a constant value that is a power of 2.

Just to be clear, all functions described above are fully specified in the packages mentioned. As long as the libraries and packages are properly mentioned before the entity declaration, one will be able to simulate VHDL code that uses the functions because the functions themselves have been precompiled and stored in the appropriate libraries. On the other hand, these functions are special functions that are recognized by a synthesis tool. It does not need to synthesize the associated function bodies, but will directly generate hardware for each function.

### 9.3 Example

In this section an example will be discussed in which some of the data types and functions presented above are used. Two different descriptions of the same hardware will be presented: the first uses the data type `unsigned` for all internal calculations, the second is based on the data type `integer`. The hardware is an "exotic" type of counter that should start to count up from 5 after a reset signal and should continue counting until 10. Then, as long as no reset signal is given, the counter should repeatedly count from 0 to 10. The data type of the output signal should be `std_logic_vector` because it is a primary output. The two versions of the counter are respectively shown in Figures 17 and 18. Both descriptions should lead to the same hardware when input to a synthesis tool. The use of the first style is recommended.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity my_counter is
port (clock, reset: in std_logic;
 count: out std_logic_vector(3 downto 0));
end my_counter;

architecture behavioral of my_counter is
signal local_count: unsigned(3 downto 0);
begin
sequential: process (clock)
begin
if rising_edge(clock) then
if reset = '1' then
 local_count <= to_unsigned(5, 4);
-- alternative: local_count <= "0101";
elsif local_count >= to_unsigned(10, 4) then
 local_count <= to_unsigned(0, 4);
else
 local_count <= local_count + 1;
end if;
end if;
end process sequential;
count <= std_logic_vector(local_count);
end behavioral;

Figure 17: The synthesizable VHDL description of an “exotic” counter.

9.4 Multidimensional Data Structures

In VHDL, any data structure that is an array, must first be declared as a new data type. For example, the data type std_logic_vector that has been used many times, is declared to be an array of the type std_logic in the package std_logic_1164.

The same mechanism can be used to create multidimensional data structures. In order to be able to use two-dimensional data structures, for example, one can first define a new type that is an array of a one-dimensional data type. One can then declare and use new signals or variables of this type. This is illustrated in Figure 19 that shows a simple circuit that can receive a multibit data word and store it in a shift register. The new two-dimensional data type is called memory here. It can store 10 data words of 8 bits.

A VHDL construct that has not been presented yet, but is very useful when dealing with arrays is the for loop. It is used twice in the example of Figure 19. Note that the loop counter counter has to be declared. By the way, there will not be any hardware in the actual realization that holds the counter; the meaning of the for loop for synthesis is a repetition in space rather than in time.

A final issue that can be illustrated by the example of Figure 19 is the distinction between a signal and a variable. From a synthesis point of view, there is no distinction between their hardware equivalents. They either become wires or flipflops. A variable is proprietary to a process and cannot be accessed from outside the process.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

descriptive my_counter is
  port (clock, reset: in std_logic;
        count: out std_logic_vector(3 downto 0));
end my_counter;

descriptive behavioral of my_counter is
  signal local_count: integer range 0 to 10;
begin
  sequential: process (clock)
  begin
    if rising_edge(clock)
    then
      if reset = '1'
      then
        local_count <= 5;
      elsif local_count >= 10
      then
        local_count <= 0;
      else
        local_count <= local_count + 1;
      end if;
      end if;
  end process sequential;
  count <= std_logic_vector(to_unsigned(local_count,4));
end behavioral;

Figure 18: An alternative description of the counter of Figure 17.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shift_in is
  port (clock, read_mode, reset: in std_logic;
        data_in: in std_logic_vector (7 downto 0);
        data_out: out std_logic_vector (7 downto 0));
end shift_in;

architecture behavioral of shift_in is
  type memory is array (1 to 10) of unsigned (7 downto 0);
  signal local_memory: memory;
begin
  shift: process (clock)
  variable counter: integer range 1 to 10;
  begin
    if rising_edge(clock) then
      if (reset = '1') then
        for counter in 1 to 10 loop
          local_memory(counter) <= to_unsigned(0, 8);
        end loop;
      else
        if (read_mode = '1') then
          for counter in 2 to 10 loop
            local_memory(counter) <= local_memory(counter - 1);
          end loop;
          local_memory(1) <= unsigned(data_in);
        end if;
      end if;
    end if;
  end process shift;
  data_out <= std_logic_vector(local_memory(10));
end behavioral;

Figure 19: A synthesizable multibit shift register.