VHDL for Simulation and Synthesis

Sabih H. Gerez
University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science (EWI-CAES)
s.h.gerez@utwente.nl

Version 5.0* (September 2, 2016)

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.[1] The text has undergone a major revision in order to be suitable for use in the elective course *VLSI System Design* around 2000 and once more for adaptation to the course *System-on-Chip Design* in 2004.[2]

Suggestions to improve the text are always welcome.

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.

Design and coding rules, indicated by the keywords "D/C Rule" and typeset in a framed box, are included in this text. Consider them to be mandatory and respect them in your designs.

Contents

*Version history: Version 1 was released in 2003, Versions 2 and 3 in 2004, Version 4 in 2010.

[1] In the course of those years, I have received feedback from many persons involved in teaching the laboratory course. The list of people that I would like to acknowledge includes Hans Snijders, Johan Wesselink, Javier Olivan, Frank te Beest, Erik Roos and many others.

[2] For more information on past and current courses, see: http://wwwhome.cs.utwente.nl/~gerezsh/.

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 initially remained an academic issue. This situation has changed in the 1980s, however. With the support of the U.S. Defense Department, experts then developed an HDL 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. Already for decades, there are just two widely-used HDLs, the second one being *Verilog*. They can more or less be used to describe the same things and are supported by the major vendors of computer-aided design tools. 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 multiple times; the most important standards date from 1987, 1993 and 2008. The differences between the standards are not relevant in the context of the current document which adheres to the 1993 standard.

While VHDL was the outcome of work by a large group of programming-language experts, Verilog was much more an ad-hoc language created for commercial product which turned out to receive wide acceptance. As such, it has several weaknesses such as tolerating signals that were nowhere declared. For this reason, VHDL was the language of choice for the System-on-Chip Design course.

Before presenting VHDL in later sections, this document pays attention to the chip design flow, the sequence of design steps, in the next one. Knowledge of the flow should make it easier to understand how design can be supported by a language like VHDL.

2 The ASIC/FPGA Design Flow

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 on a single chip. 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 based on a general-purpose language such as C is 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 in 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. These are the problems of *assignment* and *scheduling*. They can either be solved manually or using *architectural synthesis* (also called *high-level synthesis*) tools.[3]

[3]The elective course *Implementation of Digital Signal Processing* dedicates significant attention to architectural synthesis.

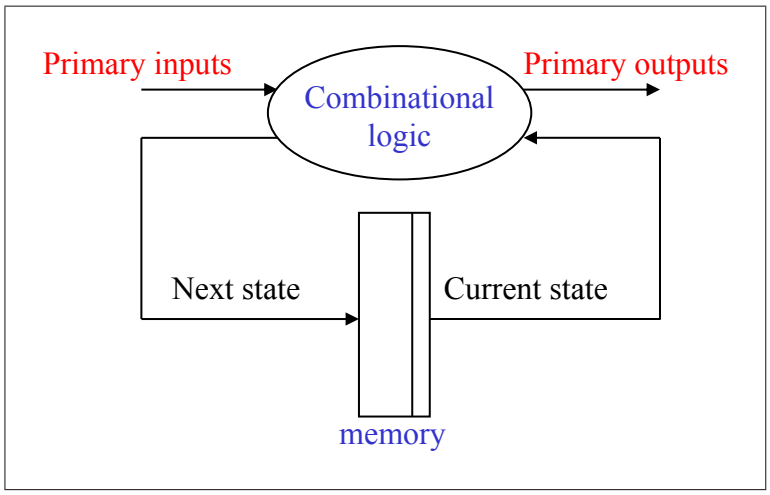

Figure 1: *Hardware model at the RT level, corresponding to a Mealy machine.*

At the register-transfer (RT) level, the timing of a design is specified 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, hardware at the RT level obeys the model of Figure 1. The figure depicts a so-called Mealy-type finite state machine. Combinational logic computes the next state and outputs from the current state and current inputs.

At this stage *logic synthesis* can be performed to design the combinational logic that will implement the next-state function. Logic synthesis is the process of optimizing Boolean expressions and finding the best mapping on the gates available in the chosen technology. 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 build a 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. Whereas automatic synthesis can deal with thousands or even millions of transistors for digital circuits, the opportunities for automatic synthesis of a analog circuits are far more limited.

For this reason, 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.

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.

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 a mathematical proof does not need to substitute all possible values for variables in an equation. Few commercial products for formal verification exist, while the topic continues to receive attention from academic researchers. Such tools are not used in this course.

Given the importance of simulation in the design process and the many levels of abstractions that exist, VHDL emerges as a powerful language because 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 with VHDL-AMS: 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 flow 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 interconnection between the various blocks.

- *Hierarchy and abstraction.* The subblocks making up a block that has a structural description, can on their turn have their own structural description. This can 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 *decomposition*. It results in a structural description at the highest level while the subblocks initially get a behavioral description. These are on their turn divided into interconnected blocks with a behavioral description each. In this way, a completely structural description is ultimately obtained. The behavior of the blocks at higher abstraction levels follows *bottom-up* from the behavior of the elementary building blocks and the structure.

These concepts are illustrated in Figure 2. In Figure 2(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 2(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 2(c) Z is split up further into Z1 and Z2. The recursive division of the design can be reflected in a *decomposition tree* as shown in Figure 2(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 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 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*, as mentioned in Section 2. 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 is called *automatic synthesis*. If the tools do not produce errors, then verification of

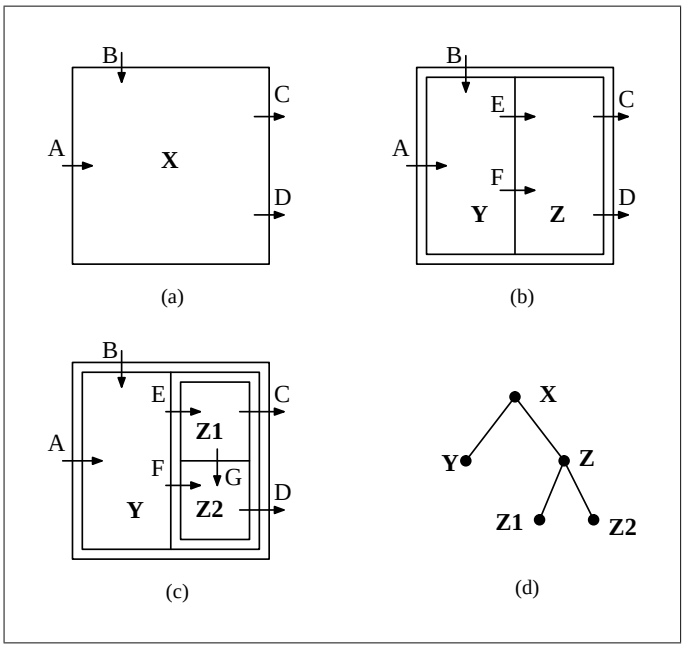

Figure 2: *A block with a purely behavioral description (a), its division into two subblocks (b), a further subdivision (c), and the decomposition tree (d).*

the decomposition is not needed. This is called *correctness by construction*. On the other hand, the complexity of automatic synthesis tools is so high that some verification of its results is still desired to obtain confidence in the quality of the design.

4 VHDL Libraries, Packages, and Entities

This section presents a first set of important VHDL constructions. They are presented in the context of a simple circuit called `siso8` based on 8-bit serial-in serial-out communication.

Note: VHDL is not case sensitive (except in character and string constants). Only lower-case letters are used in this text.

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 3 presents the declaration of the entity `siso8`.

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 `siso8` 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

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

entity siso8 is
   port (data_in: in std_logic_vector(7 downto 0);
         clk: in std_logic;
         reset: in std_logic;

         req: out std_logic;
         data_out: out std_logic_vector(7 downto 0);
         ready: out std_logic);
end siso8;
```

Figure 3: *The entity declaration for the* `siso8` *circuit.*

together for specific reasons. The package `std_logic_1164` defines a nine-valued data type called `std_logic` 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 `'Z'` (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). The package `std_logic_1164` also defines the data type `std_logic_vector` that is meant for multi-bit signals each of the type `std_logic`.

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 `'Z'` 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`.

D/C Rule 1 *Two-way communication should not be used in any design.*

The serial-in serial-out device `siso8` has an 8-bit data input called `data_in` and an 8-bit data output called `data_out`. Their data type is `std_logic_vector`. It has two single-bit inputs of the type `std_logic`: `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. The device also has two single-bit outputs: `req` is a request signal indicating that new data should be provided to the `data_in` input while `ready` signals that the `data_out` output is valid and can be read.

```
architecture copy of siso8 is
begin
  -- the next process is sequential and only sensitive to clk and reset
  seq: process(clk, reset)
  begin
    if (reset = '1')
    then
      data_out <= (others => '0');
      ready <= '0';
    elsif rising_edge(clk)
    then
      data_out <= data_in;
      ready <= '1';
    end if;
  end process seq;

  -- the system is intended to receive new data at each clock cycle
  -- (after reset)
  req <= '1';
end copy;
```

Figure 4: *Architecture description for the entity* `siso8` *that simply copies the input to the output.*

5 Architectures, Processes, Signals, and Variables

The interface of the `siso8` 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 or structural descriptions, so that descriptions at different levels of abstraction can be available simultaneously (see also Section 3).

The `siso8` circuit will actually be used to implement a wide range of designs. The design with architecture `copy` as given in Figure 4, is a possible behavioral description. This design requests new data at each clock cycle and stores this data immediately at its output register. In VHDL, any text following two dashes until the end of the line is considered to be *comment*.

> **D/C Rule 2** *Be generous in inserting comments to make your code more readable.* Indent *code to emphasize its structure.*

A more complex architecture with name `gcd` implements a *greatest common divider* (GCD) circuit. It is presented in Figure 5. 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).

Behavior is specified in VHDL by means of a `process`, of which an architecture can possess several. 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.

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

architecture gcd of siso8 is
  -- registers
  signal num1, num2: unsigned(7 downto 0);
  signal odd, req_i: std_logic;
  -- wires
  signal num1_next, num2_next: unsigned(7 downto 0);
  signal odd_next, req_i_next, ready_next: std_logic;
begin
  seq: process(clk, reset) -- process is sequential
  begin
    if (reset = '1')
    then
      num1 <= (others => '0');
      num2 <= (others => '0');
      odd <= '0';
      req_i <= '1'; -- the system is ready to receive data after reset
      ready <= '0';
    elsif rising_edge(clk)
    then
      if ((req_i = '1') and (odd = '0'))
      then
        num1 <= unsigned(data_in);
        odd <= '1';
        ready <= '0';
      elsif ((req_i = '1') and (odd = '1'))
      then
        num2 <= unsigned(data_in);
        odd <= '0';
        req_i <= '0';
        ready <= '0';
      else
        num1 <= num1_next;
        num2 <= num2_next;
        req_i <= req_i_next;
        ready <= ready_next;
      end if; -- ((req_i = '1') and (odd = '0'))
    end if; -- (reset = '1')
  end process seq;
  next_val: process(num1, num2) -- combinational process
  begin
    if (num1 > num2)
    then
      num1_next <= num1 - num2;
      num2_next <= num2;
      ready_next <= '0';
      req_i_next <= '0';
    elsif (num1 < num2)
    then
      num1_next <= num1;
      num2_next <= num2 - num1;
      ready_next <= '0';
      req_i_next <= '0';
    else
      num1_next <= num1;
      num2_next <= num2;
      ready_next <= '1';
      req_i_next <= '1';
    end if;
  end process next_val;
  data_out <= std_logic_vector(num1); -- output can be any of num1 or num2
  req <= req_i; -- req wires to req_i
end gcd;
```

Figure 5: *Architecture description for the entity* `siso8` *that computes the greatest common divider of two subsequent inputs.*

The architecture of Figure 5 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 8). 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:

```
c <= a or b after 55ps;
```

Such a statement could be used to model an OR-gate. 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 examples of Figure 4 and 5, the sequential process is only sensitive to the clock and reset signals. The process first checks the value of the reset signal. Only when the reset is not active, it checks whether the clock had a rising edge and then updates the memory elements. This expresses exactly the fact that the memory elements are supposed to be implemented by *positive edge-triggered flipflops with asynchronous resets*.

> **D/C Rule 3** *All registers in your designs should be updated on the rising clock edge. All resets should be asynchronous.*

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 has the danger of creating an unwanted feedback and possibly a memory element. In the example, the inputs of process `next_val` are `num1` and `num2`. The outputs are `num1_next`, `num2_next`, `ready_next` and `req_i_next`. Note that all four signals contain the new values of signals stored in registers. Envisioning the hardware that is designed, this, for example, means that `num1` refers to the outputs of the flipflops holding value `num1` whereas `num1_next` refers to the inputs of these flipflops.

D/C Rule 4 *Use separate VHDL processes to describe combinational and sequential logic.*

All input signals of a combinational process *must* occur in its sensitivity list. Omitting a signal may result in unexpected behavior and different behavior after synthesis.

D/C Rule 5 *Be keen on adding all input signals of a combinational process to its sensitivity list.*

The VHDL code of the two architectures that are presented here, is *synthesizable* (see Sections 11 and 12). This means that it makes use of that subset of the full VHDL syntax that can be automatically mapped onto hardware. Looking into more detail to the `copy` architecture of Figure 4, one sees two memory elements: `data_out` and `ready`. They are updated at each rising clock edge and should preserve their content until next rising edge. The value of `data_out` after reset is not the result of any "computation"; for this reason the `ready` signal has reset value `'0'` and becomes permanently `'1'` afterwards. Signal `req`, on the other hand can be permanently high as the system is supposed to process all inputs directly after reset. Note that the assignment to `req` occurs outside any process. Such an assignment is called a *concurrent assignment* and is equivalent to the process with just the assignment in its body that is sensitive to all signals in the right-hand side of the assignment.

In the architecture `gcd` of Figure 5, any signal that occurs at the left-hand side of an assignment in process `seq` is a memory element. The signal `req_i` has been introduced because `req` is an output port of `siso8`. The semantics of VHDL do not allow that the value of an output port is consulted within the entity. Hence the introduction of an intermediate signal. The final assignment `req <= req_i` connects the internal signal to the output. The architecture has two internal registers `num1` and `num2`. It first takes care of copying input data sequentially into these registers. Then Euclid's algorithm is executed. When doing arithmetic with bit vectors of the type `std_logic_vector`, one needs to agree on how numbers are encoded in bits. The `unsigned` data type used in the code tells e.g. that the bit pattern should be interpreted as a positive number. The next section gives more information on data types.

So, processes in synthesizable VHDL are either sequential or combinational:

- A sequential process has only the clock and reset signals in its sensitivity list. The process consists of an `if` statement checking the reset and then one checking for the rising edge of the clock. The latter one has no `else` branch. The same signal may be present in both the left-hand and right-hand side of an assignment. When used in the left-hand side, the corresponding register input is meant. When used in the right-hand side, the register output. At the rising clock edge the register input is copied to the register output.
- A combinational process is sensitive to *all* its inputs. All signals in the process except for those occurring in the left-hand side of an assignment are considered inputs. Those at the left-hand side are the outputs of the process. The sets of input and outputs signals should be disjoint as there may otherwise be a feedback path through the logic that is not interrupted by registers which is against the principles of synchronous design.

6 Data Types and Functions for VHDL Synthesis

VHDL has a few built-in data types (such as `integer` and `character`) and it also has a powerful mechanism for defining new data types. The standardized type `std_logic` is an example of a data type that is not built in. It becomes available by declaring the package that defines it (`std_logic_1164` in the library `ieee`), before using it. In this section, additional information will be given regarding the data types standardized by the IEEE for synthesis.

6.1 Data types

In all examples given above, most 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 logic 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 (e.g. "plain" binary, Grey-coded binary, 2's complement signed, 1's complement signed, fixed point, floating point, etc.). The IEEE standard for VHDL synthesis defines two types that are both 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 according to 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 counterpart of `std_logic_vector` is a bundle of wires identified with a numeric index. 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:

```
a <= std_logic_vector(b); and
b <= unsigned(a);
```

"Casting" amounts to *reinterpretation* of the same pattern. In VHDL, the types match correctly while in hardware nothing happens: the bundle of wires stays the same; only the interpretation of the signals that they carry, changes.

6.2 Functions

VHDL has the feature found in many object-oriented languages that a function with some name can be made to behave differently depending on the data types of its arguments. Using so-called *overloading*, existing functions as well as 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 <= x nand y;` will compute the bitwise NAND of `x` and `y` and assign it to `z`.

The package `numeric_std` located in the `ieee` library contains many useful functions related to the use of the data types `integer`, `unsigned` and `signed`. Everything mentioned below for `unsigned` has a counterpart for `signed`. 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`. The result size for addition and subtraction is the maximum of the sizes of its operands. For multiplication, the result size is the sum of the sizes of its operands. 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.
- The infix operators `/` (division), `mod` (modulo) and `rem` (remainder) are usually supported. However, they will generate expensive hardware, unless the second operand has a constant value that is a power of 2.
- 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.

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.

```
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 6: *The synthesizable VHDL description of an "exotic" counter.*

6.3 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 6 and 7. Both descriptions should lead to the same hardware when input to a synthesis tool. The use of the first style is recommended.

6.4 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 8 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

```
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: 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 7: *An alternative description of the counter of Figure 6.*

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 8. 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.

As opposed to all other examples in this document, the design of Figure 8 uses a *synchronous reset* which means that the contents of the memory can only be reset on the rising edge of the clock and no reset is possible in the absence of a clock. Student designs should only use asynchronous resets. Note that the process has the clock as the only signal in its sensitivity list.

7 The Testbench Concept, Structural Descriptions, and Configurations

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.

```
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 8: *A synthesizable multibit shift register using a* synchronous *reset.*

The entirety of DUV and models that drive its inputs and process its outputs is called a *testbench*. It is recommended to build a testbench that at least consists of the following models:

- A "test-vector controller" (TVC) 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.
- A top-level entity without any inputs or outputs. This top level will have a *structural* architecture that consists of the DUV and the test-vector controller.

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 9 that depicts the two entities mentioned above for the case of the `siso8` circuit.

Its VHDL description is then given in Figure 10. Before commenting on the VHDL code of the testbench, the concept of *instantiation* will be introduced. When describing a hardware unit in VHDL by means of an entity declaration and an architecture, one establishes a kind of template for that unit. Instantiation is the incorporation of this particular unit in a larger hardware unit. The template, with its

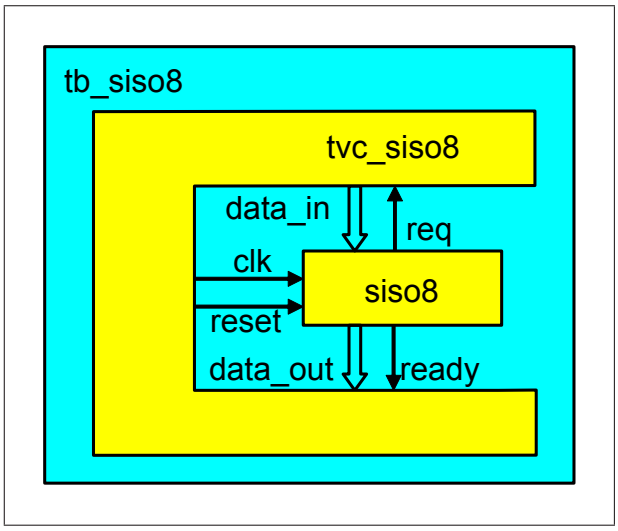

Figure 9: *The testbench for DUV* `siso8`.

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

entity tb_siso8 is
end tb_siso8;

architecture structure of tb_siso8 is
  -- declare components to be instantiated
  component siso8
    port (data_in: in std_logic_vector(7 downto 0);
          clk: in std_logic;
          reset: in std_logic;

          req: out std_logic;
          data_out: out std_logic_vector(7 downto 0);
          ready: out std_logic);
  end component;

  component tvc_siso8
    port (data_in: out std_logic_vector(7 downto 0);
          clk: out std_logic;
          reset: out std_logic;

          req: in std_logic;
          data_out: in std_logic_vector(7 downto 0);
          ready: in std_logic);
  end component;

  -- declare local signals
  signal data_in, data_out: std_logic_vector(7 downto 0);
  signal clk, reset, req, ready: std_logic;
begin
  -- instantiate and interconnect components
  duv: siso8
    port map (data_in => data_in, clk => clk, reset => reset,
              req => req, data_out => data_out, ready => ready);
  tvc: tvc_siso8
    port map (data_in => data_in, clk => clk, reset => reset,
              req => req, data_out => data_out, ready => ready);
end structure;
```

Figure 10: *Entity and architecture for a testbench of the* `siso8` *circuit.*

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 `structure` architecture of the testbench first declares the components that it needs and then instantiates them. *Components* and *signals* are declared in the declaration part of the description (before the keyword `begin`). Note that component declarations 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 from the "configuration" of the hardware (see below).

During instantiation, the keywords `port map` precede an *association list* with signals; in which formal signals are explicitly linked with the actual signals. The ordering of signals can be arbitrary (it does not need to follow the ordering of the component declaration).

The architecture of Figure 10 is a *purely* structural description: it solely contains instantiations of sub-blocks but no processes, so no behavioral code. Although it is allowed to mix structural and behavioral descriptions in one architecture in VHDL, it is strongly recommended not to do so.

D/C Rule 6 *Do not mix structural and behavioral descriptions in one VHDL architecture.*

The entity `tvc_siso8` (not shown in this document) 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).

While it may appear at first sight that a description such as in Figure 10 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 `siso8` testbench Figure 11 shows the two configurations to be used for the two architectures presented. The outer `for` statement indicates that the configuration is meant for the architecture `structure` of the entity `tb_siso8`. 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 (there are two instance names in this example: `duv` and `tvc`). 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 only a single architecture has been compiled of each instantiated entity. This is not recommended, though.

D/C Rule 7 *Define a configuration for each entity that you want to simulate.*

```
configuration conf_tb_siso8_copy of tb_siso8 is
  for structure
    for duv: siso8 use entity work.siso8(copy);
    end for;
    for tvc: tvc_siso8 use entity work.tvc_siso8(behavior);
    end for;
  end for;
end conf_tb_siso8_copy;

configuration conf_tb_siso8_gcd of tb_siso8 is
  for structure
    for duv: siso8 use entity work.siso8(gcd);
    end for;
    for tvc: tvc_siso8 use entity work.tvc_siso8(behavior);
    end for;
  end for;
end conf_tb_siso8_gcd;
```

Figure 11: *The configurations that fully specify simulation models for the* `siso8` *circuit.*

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. There may even be multiple versions of the TVC to simulate different operation modes of a DUV (e.g. one that verifies plain operation and one that verifies test modes such as the scan chain, see Section 9). Note that a configuration can be composed of entity-architecture combinations or other configurations.

8 The Operation of the VHDL Simulator

Before performing VHDL simulations in practice, it is useful to have a brief look at how the VHDL simulator works. The presentation is confined 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.[4]

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

Section 5 already stated that processes must have a "sensitivity list", meaning that their bodies are evaluated once each time when one of the signals in the list changes in value. Another category of processes contain `wait` statements and *no* sensitivity list (the combination of wait statements and a sensitivity list is not allowed). A process with wait statements is immediately restarted when its entire body has been executed, but the evaluation is stopped when a wait statement is encountered (improperly written code, e.g. with a wait statement in a branch of an `if` statement that is never selected, will lead

[4] See e.g. Gerez, S.H., *Algorithms for VLSI Design Automation*, John Wiley and Sons, Chichester, (1999).

to a process that runs forever). 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. Wait statements are not synthesizable; they are mainly used in system-level hardware models and testbenches.

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*. "Event" is the designation given to a signal change or a process activation at a specific time. For example, if the process that is active at moment $t = t_0$ encounters the statement `a <= '1' after 10 ns`, then transaction `a <= '1'` is placed on the event list at moment $t = t_0 + 10$ ns.

A transaction never takes effect immediately, not even if the code does not specify any delay (for example, through a signal assignment without the keyword `after`). In that case, the transaction is placed on the event list at moment $t = t_0 + \Delta$. 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: $0 < \Delta < 2\Delta \ldots$ The notion of an infinitesimally small delay in simulation is called a *delta delay*.

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 saves computation time. Processes are evaluated only when necessary. This method is called the *event-driven* simulation technique. It is used in one way or other by practically all digital simulators.

9 Towards Designing IP Blocks: Parameterizable Components and Test Interface

Designing *systems on chip* (SoCs) is only feasible by the availability of so-called *IP blocks*. IP stands for *intellectual property* and refers to the result of a design activity which has not necessarily materialized but consists of a collection of, for example, VHDL files. These files represent some economic value. Hence the name "intellectual property". In order to face the ever growing complexity in IC design, it is becoming more and more common practice that different parties concentrate on the design of standard components with a well-defined interface such as a microprocessor, a DMA (direct memory access) unit, a USB (universal serial bus) interface, etc. The SoC designer has then a relatively easy task to integrate the different components.

A desirable property of IP blocks is parameterizability. Examples of parameters are the widths of data and address buses, the size of available memory, etc. In this way, the same component can be reused in different contexts without the need to rewrite the VHDL code (supposing that the block has been designed in VHDL). The parameters should be given a value at the moment of component instantiation.

An important issue in IC design is *testability*. As a consequence of the delicate manufacturing process which is e.g. sensitive to dust particles, alignment of masks, etc., ICs that have been produced, are not guaranteed to function. Each IC needs to be tested before being shipped to the customer. Testing an IC becomes significantly easier if testing is taken into account during the design of the IC; this is called

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

entity siso_gen is
   generic (word_length: natural);
   port (data_in: in std_logic_vector(word_length-1 downto 0);
         clk: in std_logic;
         reset: in std_logic;

         req: out std_logic;
         data_out: out std_logic_vector(word_length-1 downto 0);
         ready: out std_logic;

         -- scan-chain interface
         scan_in, scan_shift: in std_logic;
         scan_out: out std_logic);
end siso_gen;
```

Figure 12: *The SISO circuit with a generic word length and test interface*

.

design for testability (DFT). Different DFT strategies exist. If one agrees on one of these for all IP blocks, it becomes easier to combine them at the level of the SoC.

Figure 12 presents a new entity for the SISO example: `siso_gen`. With respect to the entity `siso8` (see Figure 3), it can be seen that the declaration not only contains I/O signals indicated by the keyword `port` but also parameters indicated by the keyword `generic`. The only declared parameter is actually `word_length`: it indicates the number of bits in the input and output words `data_in` and `data_out`. The generic parameter shows up in the port declaration and can also be used anywhere in an architecture declaration associated with the entity `siso_gen`.

The entity has provisions to include a *scan chain*. Although the topic is outside the scope of this document, the scan-chain principle will be shortly explained here. A scan chain is a DFT strategy. Changing the value a control signal, called `scan_shift` in this example, from `'0'` to `'1'`, puts all flipflops in the design (or a subset of them) in a *shift register*. In this mode, at each new rising edge of the clock, the flipflops copy the value of their predecessors in the chain rather than the intended value for normal (functional) operation. The input and output of this shift register are accessible from outside the block: they are called here `scan_in` and `scan_out` respectively.

The scan chain makes it possible to bring the hardware into a defined state using the shift mode. In this way, one can easily provide a *test pattern* at the inputs of all combinational logic in the design. Once the test pattern has been loaded, one executes one clock cycle in normal mode (making `scan_shift '0'`). This *captures* the response of the combinational logic into the flipflops. This response can be shifted out of the circuit while a new test pattern gets loaded. Faulty ICs can then be detected by comparing the measured response with the expected one.

Generic parameters can receive a value when a component is instantiated in a structural architecture. An example is shown in Figure 13. The testbench consists of two components which both have a generic parameter `word_length`. The parameter receives a value using the `generic map` construct which has a similar syntax as the `port map` construct that it precedes. In this example, the testbench itself has a generic `word_length` which it passes down to its subblocks. Note also that the *test-vector controller* component `tvc_siso_gen` has two more generics for the input and output files. These generics are not mapped at the moment of instantiation, which is allowed because default values have been provided for them.

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

entity tb_siso_gen is
   generic(word_length: natural := 8);
end tb_siso_gen;

architecture structure of tb_siso_gen is
   - - declare components to be instantiated
   component siso_gen
      generic (word_length: natural);
      port (data_in: in std_logic_vector(word_length-1 downto 0);
            clk: in std_logic;
            reset: in std_logic;

            req: out std_logic;
            data_out: out std_logic_vector(word_length-1 downto 0);
            ready: out std_logic;

            scan_in, scan_shift: in std_logic;
            scan_out: out std_logic);
   end component;

   component tvc_siso_gen
      generic (word_length: natural;
               in_file_name: string := "siso_gen.in";
               out_file_name: string := "siso_gen.out");
      port (data_in: out std_logic_vector(word_length-1 downto 0);
            clk: out std_logic;
            reset: out std_logic;

            req: in std_logic;
            data_out: in std_logic_vector(word_length-1 downto 0);
            ready: in std_logic;

            scan_in, scan_shift: out std_logic;
            scan_out: in std_logic);
   end component;

   - - declare local signals
   signal data_in, data_out: std_logic_vector(word_length-1 downto 0);
   signal clk, reset, req, ready: std_logic;
   signal scan_in, scan_shift, scan_out: std_logic;
begin
   - - instantiate and interconnect components
   - - note that the generic word_length is passed to the subblocks
   duv: siso_gen
      generic map (word_length => word_length)
      port map (data_in => data_in, clk => clk, reset => reset,
                req => req, data_out => data_out, ready => ready,
                scan_in => scan_in, scan_shift => scan_shift,
                scan_out => scan_out);
   tvc: tvc_siso_gen
      generic map (word_length => word_length)
      port map (data_in => data_in, clk => clk, reset => reset,
                req => req, data_out => data_out, ready => ready,
                scan_in => scan_in, scan_shift => scan_shift,
                scan_out => scan_out);
end structure;
```

Figure 13: *The testbench for the entity* `siso_gen` *illustrating the* `generic map` *construct.*

```
entity tb_siso_gen_top is
end tb_siso_gen_top;

architecture top of tb_siso_gen_top is
   component tb_siso_gen
      generic(word_length: natural := 8);
   end component;
begin
   tg: tb_siso_gen;
end top;
```

Figure 14: *The shell entity* `tb_siso_gen_top`.

```
configuration conf_tb_siso_gen_gcd of tb_siso_gen_top is
   for top
      for tg: tb_siso_gen use entity work.tb_siso_gen(structure)
            generic map (word_length => 16);
         for structure
            for duv: siso_gen use entity work.siso_gen(gcd);
            end for;
            for tvc: tvc_siso_gen use entity work.tvc_siso_gen(file_io)
                generic map (word_length => 16,
                             in_file_name => "gcd16.in",
                             out_file_name => "gcd16.out");
            end for;
         end for;
      end for;
   end for;
end conf_tb_siso_gen_gcd;
```

Figure 15: *The configuration that fully specifies the simulation model for the* `siso_gen` *circuit with a* `gcd` *architecture for the hardware and a file-I/O-based architecture for the test-vector controller.*

A useful feature of VHDL is that generic maps (and also port maps) can not only occur in structural *architecture declarations* but as well in *configuration declarations*. In order to be able to make use of this feature, a "shell" entity `tb_siso_gen_top` has been created. It is described in Figure 14. Its only function is encapsulate the testbench such that the top-level generic can be assigned a value.

Figure 15 illustrates the mapping of generics in a configuration declaration. The configuration is meant for a `gcd` architecture for `siso_gen` (not shown in this document, but very similar to the one of Figure 5) and a test-vector controller that performs a functional simulation based on inputs read from file and outputs written to file. As can be seen in the figure, generic mappings can be specified at various levels. The word length is given at the testbench level, while the file names meant for a GCD simulation are specified one hierarchical level lower.

10 Data Path and Controller Separation

The separation of hardware into *combinational* and (synchronous) *sequential* logic is clear: combinational logic does not have any internal memory and synchronous sequential logic basically changes value depending on a clock signal. In many cases, it is convenient to separate hardware in another way into the following parts: a *data path* and a *controller*. In the data path, the main data processing is done. The data path e.g. contains arithmetic units, registers, memories, buses, multiplexers, etc. *Control* sig-

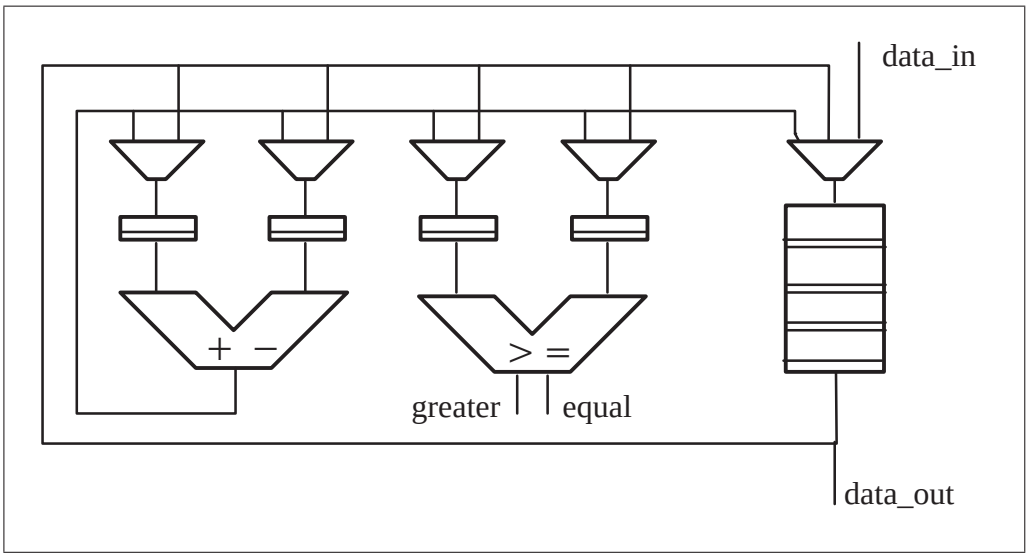

Figure 16: *An example data path for the* `siso_gen` *system.*

nals such as *select* signals for multiplexers, *enable* signals for registers, influence the functioning of the data path. They are generated by the controller. On the other hand, the data path may generate *status* signals that e.g. result from a comparison that act as inputs for the controller.

The separation between data path and controller is not always sharp. An address for a memory may be generated in the controller but may also be computed in the data path (think of incrementing an index to access array elements).

In the designs presented until now, data paths and controllers are not explicitly represented. When one assigns different values to the same data signal in the `then` and `else` branches of an `if` statement, for example, one describes a multiplexer (in the data path) where the condition(s) of the `if` statement represent the select signals (the computation of the conditions belongs to the realm of the controller). Below, the description of a data-path-controller system will be presented.

Simplified schematics of an example data path suitable for the implementation of the `siso_gen` system are given in Figure 16. The data path consists of two arithmetic units that operate on *signed* operands. An *adder/subtractor* unit has two input registers (a *left* one and a *right* one) each with an *enable* signal and a control input signal to choose between addition and subtraction. A *comparator* unit also has two input registers. It generates two status outputs: `greater` becomes `'1'` when the left operand is greater than the right one; `equal` becomes `'1'` when both operands are equal. The third unit in the data path is a memory (or more precisely, a *register file*) with four locations (the address ranges from 0 to 3). The memory has a two-bit address to indicate the write location and a two-bit address to indicate the read location the contents of which are output. The memory output is also connected to the system output `data_out`. The memory input (which is possibly written to some location) comes from a multiplexer that takes its value from three sources: the memory output, the adder/subtractor output or the system input `data_in`. The fourth value of the two-bit control signal for this multiplexer indicates that writing the memory is disabled. The four input registers of the two arithmetic units are connected to two-way multiplexers to take data either from the adder/subtractor output or the memory output. The code for the data path is spread across three figures: Figure 17 shows the entity declaration; Figure 18 gives an architecture with a behavioral description; for reasons of space, the description of the combinational logic in this architecture is given in Figure 19.

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

entity cmp_add_dp is
   generic (word_length: natural);
   port (data_in: in std_logic_vector(word_length-1 downto 0);
         clk: in std_logic;
         reset: in std_logic;

         data_out: out std_logic_vector(word_length-1 downto 0);

         -- adder left/right register control
         add_l_sel, add_r_sel, add_l_en, add_r_en, sub: in std_logic;

         -- comparator left/right register control
         cmp_l_sel, cmp_r_sel, cmp_l_en, cmp_r_en: in std_logic;

         -- memory control
         rd_addr, wr_addr: in std_logic_vector(1 downto 0);
         wr_sel_en: in std_logic_vector(1 downto 0);

         -- comparator status
         equal: out std_logic;
         greater: out std_logic);
end cmp_add_dp;
```

Figure 17: *The entity declaration for the data path of Figure 16.*

The data path that has just been presented, can be used to implement various algorithms by combining it with an appropriate controller. A controller that implements Euclid's GCD algorithm is given by the finite-state machine (FSM) depicted in Figure 20. The controller has not been optimized. It implements the following behavior:

- Input data is first copied to memory locations 0 and 1 respectively. This requires two states.
- Then the two input registers of the comparator are loaded. This requires two states.
- Depending on the result of the comparison, the computation is either ready or a subtraction has to be performed.
- The operands of the subtraction are loaded in such a way that the left operand is always greater than the right one. This requires two states.
- The result of the subtraction is written into memory in such a way that the largest of the two operands is overwritten. This needs one state.

There are some subtleties involved in the timing of the controller and data path. It is a design-style decision to clock the flipflops in both the data path and the controller on the rising edge of the data path. If the control signals for the data path are derived from the *current state* of the controller, the data path will lag behind one clock cycle with respect to the controller: one clock edge is needed to enter the current state and one more for the data path to react. This means that two clock cycles are necessary to process state transitions involving status signals: as it takes one clock cycle for a state transition to be effective in the data path, the response to a status signal will take one clock cycle more.

The alternative chosen here derives the control signals for the data path from the *next state* in the controller. This makes that a state transition in the controller is simultaneous with the effect that the

```
architecture behavioral of cmp_add_dp is
  -- type declaration for memory
  type memory is array (0 to 3) of signed(word_length-1 downto 0);
  -- memory declaration
  signal mem: memory;
  -- other memory elements
  signal add_l, add_r, cmp_l, cmp_r: signed(word_length-1 downto 0);
  -- wires
  signal add_out, mem_out: signed(word_length-1 downto 0);
begin
  seq: process(clk, reset)
    variable counter: integer range 0 to 3;
  begin
    if (reset = '1')
    then
      for counter in 0 to 3 loop
        mem(counter) <= (others => '0');
      end loop;
      add_l <= (others => '0');
      add_r <= (others => '0');
      cmp_l <= (others => '0');
      cmp_r <= (others => '0');
    elsif rising_edge(clk)
    then
      -- memory write
      case wr_sel_en is
        when "00" => null; -- write is disabled
        when "01" => mem(to_integer(unsigned(wr_addr))) <= add_out;
        when "10" => mem(to_integer(unsigned(wr_addr))) <= mem_out;
        when "11" => mem(to_integer(unsigned(wr_addr))) <= signed(data_in);
        when others => null; -- not relevant for synthesis
      end case;
      -- register write
      if (add_l_en = '1')
      then
        if (add_l_sel = '1')
        then
          add_l <= add_out;
        else
          add_l <= mem_out;
        end if;
      end if;
      -- other registers left out!
    end if;
  end process seq;
  -- combinational processes left out
end behavioral;
```

Figure 18: *The architecture declaration for the data path of Figure 16.*

state transition should have on the data path. In this approach, the controller can react to status signals without delay, i.e. within one clock cycle.

The VHDL description of the controller entity is given in Figure 21. Note that, in accordance with the concept of separating data path and controller, the `siso_gen` control signals `req` and `ready` are generated in the controller.

The architecture for this entity implementing the GCD algorithm is shown in Figure 22. It has the typical structure of the VHDL description of an FSM:

- First a new *enumeration data type* is used to declare the states. Note that states have a symbolic encoding. No choice is made on how to encode the state in a binary pattern. This is left to the synthesis tool.

```
- - adder/subtractor
add_sub: process(add_l, add_r, sub)
   variable add_r_in: signed(word_length-1 downto 0);
   variable carry: integer range 0 to 1; - - easy to add to "signed" operands
begin
   - - for substract, invert bits of 'add_r' and add a carry
   if (sub = '1')
   then
      add_r_in := not(add_r);
      carry := 1;
   else
     add_r_in := add_r;
     carry := 0;
   end if;
   add_out <= add_l + add_r_in + carry;
end process add_sub;
-- comparator
equal <= '1' when (cmp_l = cmp_r) else '0';
greater <= '1' when (cmp_l > cmp_r) else '0';
-- memory read
mem_out <= mem(to_integer(unsigned(rd_addr)));
-- main output
data_out <= std_logic_vector(mem_out);
```

Figure 19: *Description of the combinational logic belonging to the code of Figure 18.*

- There is then a sequential process to describe all memory elements including those holding `current_state`.
- There is a combinational process for the computation of the *next state*.
- Finally, there is a combinational process to calculate the outputs. As motivated above, the outputs are derived from then next state. The outputs therefore also depend of the (status) inputs making this FSM a so-called *Mealy* machine (in *Moore* machines the output only depends on the current state).

Note that one can implement a large range of algorithms on the same data path by specifying an appropriate controller architecture. One can make the system even more flexible by storing the control patterns in a memory rather than hard-coding them in an FSM. The patterns stored in the memory may be called the *firmware* or even the *software* depending on the actual approach chosen. Such a controller in combination with the data path may already be called a simple *processor*.

11 VHDL Synthesis Basics

It has been mentioned already that VHDL was primarily designed for purposes of *simulation* in the 1980s. In the 1990s tools became available that could synthesize well-defined subsets 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 design that shows the same behavior as the specification.

One can say that the synthesis tools perform *silicon compilation.* In a way similar to software compilation where the specification of some computation in a 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).

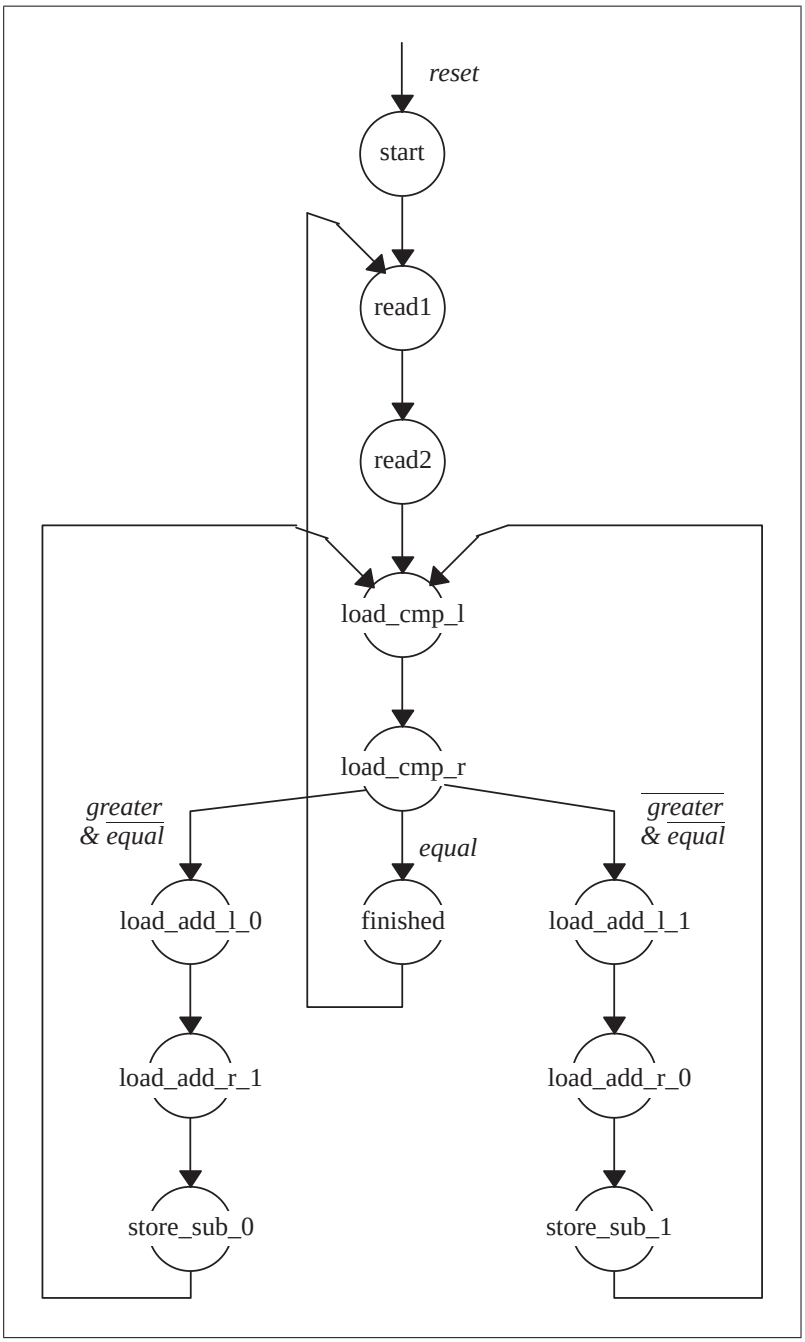

Figure 20: *The FSM computing Euclid's GCD algorithm on the data path of Figure 16.*

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 (see Section 2). 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.

As mentioned in Section 1, VHDL itself has been standardized several times. Synthesis standards also exist. They deal with two issues: data types to be used in synthesis (see Section 6) and the allowed language subset (see Section 12). In this subset, each language construct has an unambiguous hardware

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

entity cmp_add_ctrl is
   port (clk, reset: in std_logic;

         -- main outputs
         req, ready: out std_logic;

         -- status inputs from data path
         equal, greater: in std_logic;

         -- control outputs to data path

         -- adder left/right register control
         add_l_sel, add_r_sel, add_l_en, add_r_en, sub: out std_logic;

         -- comparator left/right register control
         cmp_l_sel, cmp_r_sel, cmp_l_en, cmp_r_en: out std_logic;

         -- memory control
         rd_addr, wr_addr: out std_logic_vector(1 downto 0);
         wr_sel_en: out std_logic_vector(1 downto 0));
end cmp_add_ctrl;
```

Figure 21: *The controller entity declaration.*

counterpart. 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 then 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 evolving 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 several 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 post-synthesis simulation is that the synthesis tools cannot always be trusted; due to the complexity of the algorithms, bugs may exist in the software. It may also be that the user has used non-synthesizable language constructs and then overlooked warnings issued by the synthesis tool.

12 VHDL Synthesis Through Examples

As was mentioned before, only a subset of VHDL can be synthesized by commercially available synthesis tools. It is not the intention here to exactly describe the subset as defined by the synthesis standards. Instead, a subset that is sufficient to complete the design exercises, 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.

```
architecture gcd of cmp_add_ctrl is
  - - enumeration type for states: "state"
  type state is (start,
                 read1, read2, load_cmp_l, load_cmp_r,
                 finished,
                 load_add_l_0, load_add_r_1, store_sub_0,
                 load_add_l_1, load_add_r_0, store_sub_1);
  signal current_state, next_state: state;
begin
  seq: process(clk, reset)
  begin
    if reset = '1'
    then
      current_state <= start;
      req <= '1';
      ready <= '0';
    elsif rising_edge(clk)
    then
      current_state <= next_state;
      if (next_state = read1) or (next_state = finished)
      then
        req <= '1';
      else
        req <= '0';
      end if;
      if next_state = finished
      then
        ready <= '1';
      else
        ready <= '0';
      end if;
    end if;
  end process seq;
  new_state: process(current_state, equal, greater)
  begin
    case current_state is
      when start => next_state <= read1;
      when read1 => next_state <= read2;
      when load_cmp_r =>
          if equal = '1'
          then
            next_state <= finished;
          elsif greater = '1'
          then
            next_state <= load_add_l_0;
          else
            next_state <= load_add_l_1;
          end if;
        - - other states left out!
    end case;
  end process new_state;
  outputs: process(next_state)
  begin
    case next_state is
      when read1 =>
          - - copy from data_in to memory address 0; rest is don't care
          add_l_sel <= '-'; add_l_en <= '-'; add_r_sel <= '-'; add_r_en <= '-';
          sub <= '-';
          cmp_l_sel <= '-'; cmp_l_en <= '-'; cmp_r_sel <= '-'; cmp_r_en <= '-';
          rd_addr <= "--"; wr_addr <= "00"; wr_sel_en <= "11";
        - - other states left out!
    end case;
  end process outputs;
end gcd;
```

Figure 22: *A controller architecture implementing the GCD algorithm.*

12.1 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. If they do, they are ignored. Signals can be delayed, but only by passing them through (a chain of) clocked registers.
- 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.

D/C Rule 8 *Only use `std_logic` or `std_logic_vector` data types for the top-level input and output signals of your design.*

12.2 Combinational Logic at the Bit Level

In Table 1 an example of a function with 3 inputs and 2 outputs is given.[5] *Don't care* outputs are indicated by a 'D'. The synthesizable VHDL equivalent of such a function is shown in Figure 23. 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.
- 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 23. 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.

[5] The example has been taken from: R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1984.

Input $x_1x_2x_3$	Output y_1y_2
000	11
001	10
010	01
011	01
100	10
101	1D
110	11
111	D1

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

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

entity example1 is
   port (x: in std_logic_vector (1 to 3);
         y: out std_logic_vector (1 to 2));
end example1;

architecture tabular of example1 is
begin
   react: process (x)
   begin
      case x is
         - - Note: you can't use don't cares for the input patterns
         - - when using this style of description.
         when "000" => y <= "11";
         when "001" => y <= "10";
         when "010" => y <= "01";
         when "011" => y <= "01";
         when "100" => y <= "10";
         when "101" => y <= "1-";
         when "110" => y <= "11";
         when "111" => y <= "-1";
         when others => y <= "--";
      end case;
   end process react;
end tabular;
```

Figure 23: *Truth-table style specification of combinational logic.*

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

D/C Rule 9 *Always check the warnings issued by the synthesis tool and be especially keen on inserted "latches". Latch insertion should not happen. Fix your VHDL such that synthesis does not insert any latch.*

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 24. An important remark to be made about

```
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 24: *An alternative style for synthesizable VHDL at the bit-level.*

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

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;

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 25: *The synthesizable description at the word level of a hardware unit.*

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) and not x(3)) or 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 25. The code describes a hardware unit that computes the *exclusive or* of two 12-bit signals after inverting the first

```
signal a, b: std_logic_vector (1 downto 0);
signal c: std_logic_vector (3 downto 0);

-- Concatenation:

c <= a & b;

case c is
   when "0000" => y <= '1'; z <= '0';
   - - other possibilites should follow here ...
end case;

-- Splitting:

a <= c (2 downto 1);

-- Range assignment:

c (2 downto 1) <= a;
```

Figure 26: *Different possibilities for assigning multiple-bit signals.*

signal depending on a control signal.

12.3 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 20.

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. An example of synthesizable VHDL code for an FSM has already been given in Figure 22.

Important note: Only positive-edge-triggered synchronous sequential hardware with an asynchronous reset is considered here. This means that all sequential processes in a synthesizable VHDL description should only be sensitive to the clock and reset. Such processes are repetitively activated at each clock edge, but solely specify behavior for the rising clock edge (and for the reset). 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 not compatible with the specification style. 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 6.3 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.

12.4 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 26. Two signals can be juxtaposed to form a wider signal by means of the signal-

```
if cond = '1'
then
   y <= a + b;
else
   y <= c + d;
end if;
```

Figure 27: *A fragment of VHDL showing a conditional addition.*

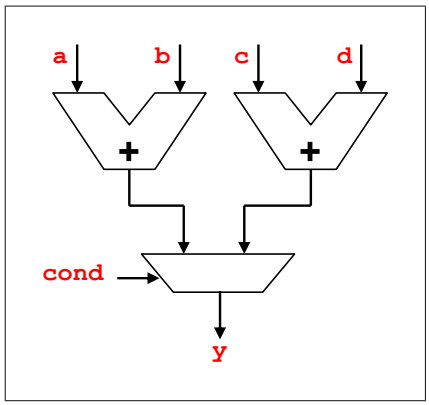

Figure 28: *A 1-to-1 realization of the code of Figure 27.*

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.

12.5 Resource Sharing

Consider the fragment of code as shown in Figure 27. A literal interpretation of this code would be that one computes the additions `a + b` and `c + d` and then passes one of the two results to output signal `y`. This implies two adders and one multiplexer as shown in the block diagram of Figure 28.

This is an expensive solution for the intended behavior: two additions are performed and one result is discarded. As the hardware costs of a multiplexer are lower than the costs of an adder, it is wiser to multiplex the inputs and perform one addition rather than perform two additions and multiplex their outputs. So, one would prefer the hardware of Figure 29 above the one of Figure 28. One says that the adder *resource* is *shared* between the two branches of the `if` statement.

The optimization that was presented, is relatively simple. One would expect that the synthesis tool should be able to perform it. Many synthesis tools actually have this possibility. However, as such an optimization modifies the hardware structure implied by the code, it is seen as an option that the tool user can control. It is recommended not to depend on the peculiarities of the tool but rather explicitly code the intended hardware structure in VHDL. The code corresponding to Figure 29 is given in Figure 30. It is supposed that the code is part of the body of a single combinational VHDL process. As the wires `t1` and `t2` are internal, they are coded as *variables* rather than *signals*. One could

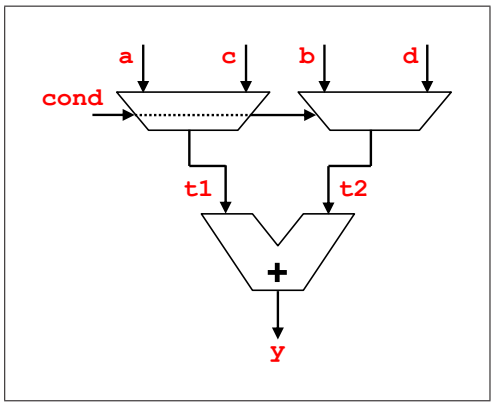

Figure 29: *A cheaper realization of the design of Figure 28.*

```
if cond = '1'
then
   t1 := a;
   t2 := b;
else
   t1 := c;
   t2 := d;
end if;

y <= t1 + t2;
```

Figure 30: *The description of the hardware of Figure 29 in VHDL.*

also opt to use two combinational processes for the hardware of Figure 29: one combinational block of which `t1` and `t2` are the outputs and another one of which they are the inputs. Then `t1` and `t2` should be declared as signals at the level of the VHDL architecture that contains the two processes.