Technical Report 2003-01P(2003/2/23)

A First Look at VHDL For Digital Design

An Introduction to the Fundamental Concepts of the Language

Luk Michel St.Onge[1] **Shawki Areibi**[2]

School of Engineering
University of Guelph
Albert Thornbrough Bld #159
Guelph, Ontario,
Canada, N1G 2W1

[1]lstonge@uoguelph.ca
[2]sareibi@uoguelph.ca (http://www.uoguelph.ca/~sareibi)

Publication History

This manual is meant to give students a “first look” at the hardware description language VHDL (namely 2nd year students taking ENGG*2410 Digital Design at the University of Guelph School of Engineering). It is a derivative of the original SOE Technical Report *VHDL for Digital Design*. The pages herein are not meant to be exhaustive; the goal is for the student to develop an understanding of some of the fundamental concepts of the VHDL language, necessary to pursue further (more advanced) studies in this area. Some knowledge of digital systems design will be assumed, although only at a moderate level. Notably, the most influential book used in the creation of this text is the *VHDL Starter’s Guide* by Sudhakar Yalamanchili [1] (highly recommended for a more complete understanding of the VHDL language). Some examples may also be based (at least in part) on examples from the book *VHDL for Programmable Logic* by Kevin Skahill [2]. Other important references include [3] and [4].

Contents

List of Figures

List of Tables

1 Introduction

The term "VHDL" is a very popular buzz word in industry, especially in the search for qualified engineers who are familiar with it. But what is VHDL, and why all the fuss?

1.1 Overview of VHDL

"VHDL" stands for VHSIC Hardware Description Language. VHSIC in turn stands for Very High Speed Integrated Circuit, which was a U.S. Department of Defense (DoD) program to encourage research on high-performance IC technology. The emergence of VHDL was a result of an obvious need for standardization by the DoD in describing digital systems, as it had many sub-contractors working on the VHSIC project. VHDL was later adopted by the Institute of Electrical and Electronic Engineers (IEEE) for standardization, which has thus aided in increasing its appeal in the electronic systems and Computer Aided Design (CAD) industries.

VHDL is a language, just as C and Java are languages. VHDL is used to describe, model, and synthesize (make) a circuit. Like C, VHDL supports libraries (design libraries that contain common or reusable components, such as `nand` and `nor` gates). VHDL also allows us to create modular designs, so that we can take advantage of hierarchical design (building a big, complex circuit from a bunch of smaller, simpler circuits).

Like Java, VHDL is "device independent". That is, we can design a circuit before we know which type of device it will be implemented on. In fact, we can take the same design and "target" many different device architectures. Once you have designed a circuit, there are two main tasks that you can accomplish: you can simulate the circuit or you can synthesize the circuit.

Simulation is usually done before synthesis, since design flaws that would prevent the actual realization of the circuit from working can be determined in advance. Simulation is a way to test a hardware circuit in software, before we go through the time and expense of implementing the hardware. There is a danger with relying on simulation alone however. In software, we can design a circuit that cannot be easily realized (synthesized) in hardware. Indeed we can (inadvertently, of course) design a circuit that is physically meaningless!

1.2 Power of HDLs

Hardware description languages (HDLs) have a resemblance to programming languages, but are specifically oriented to describing hardware structures and behaviour. They differ markedly from the typical programming language in that they represent extensive parallel operations (often referred to in this context as *concurrent signal assignments*), whereas most programming languages represent serial operations (or *sequential signal assignments*). In the past, HDLs were developed for specific CAD tools using specific design methodologies (and thus at different levels of abstraction, see Figure 1 on the following page [1]). However, with the fast growth in digital design complexity, designs began to use an increasingly diverse set of tools in their development and data sharing among these tools became extremely important.

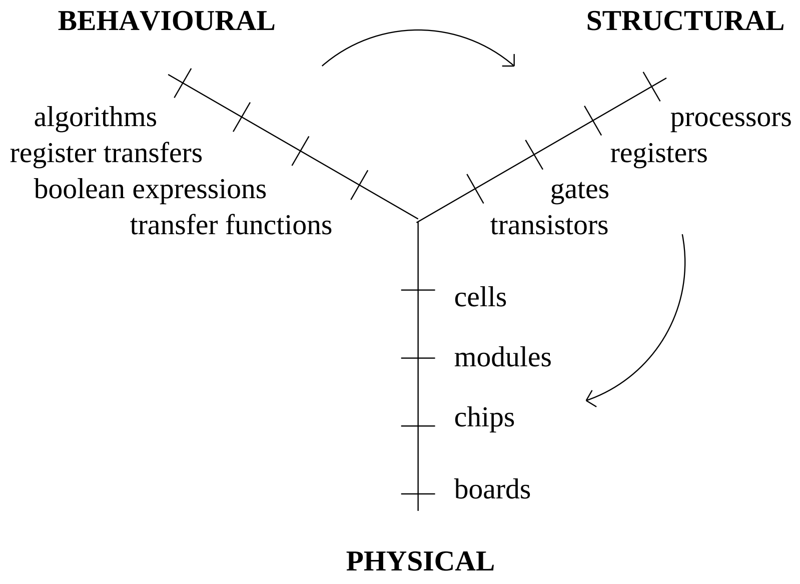


Figure 1: Design Views and Corresponding Levels of Abstraction

VHDL is capable of representing multiple levels of abstraction in the process. This further encourages standardization and use, since different design tools (with different design methodologies) using a standardized HDL can create portable models, reducing the complexity of data sharing among designs. In the rapidly developing technology sector of digital systems, design philosophy will be frequently changing, and it is therefore desirable that the HDL language used be flexible enough to accommodate these changes, something for which VHDL is seemingly well suited for.

Due to VHDL's rich language subset, it is also technology independent. Designs can be synthesized to Field Programmable Gate Arrays (FPGAs) or Complex Programmable Logic Devices (CPLDs) using specialized CAD tools. CAD tools also exist for Application Specific Integrated Circuit (ASIC) designs in order to simulate operations before physically implementing them. Once a physical design is available, the timing and performance can also be verified in simulation.

As previously alluded to, with the ability to create portable models among CAD tools, it is not a far leap for VHDL to permit the re-use of designs. Libraries of digital models are available across a diverse range of platforms, and additional ones can be created within a particular group. It is also possible to mix hardware and software in the process of a design!

1.3 Using VHDL in the Design Process

In general, there is a recipe of steps to follow when designing a circuit. These steps can be described as[2]:

1. Define the design requirements;
2. Define (code) the design in VHDL;
3. Simulate the VHDL "source" code;
4. Synthesize the design[1];
5. Fit the design into a given device architecture;
6. Program the device.

Of course, steps 2 and 3 are most often iterative (again, like C and Java, there is a great deal of debugging involved).

1.4 Summary

VHDL is a powerful language used to describe hardware structures and behaviour. As a standard of the IEEE, it is commonly used among CAD vendors and digital designers in general. Its power and appeal are derived in part from its ability to represent different levels of abstraction in a digital design, coupled with a rich language set that allows it to be used in a diverse range of design philosophies. The point is this: VHDL $\Rightarrow$ Large cost savings in the design of digital systems.

Among HDLs, VHDL is also accompanied by another popular language named Verilog. Verilog is also an approved IEEE standard and therefore most CAD tools support it as well as VHDL. This standardization gives HDLs another advantage over schematics, namely that HDLs are portable across CAD tools whereas schematic capture tools are typically unique to a particular vendor.

[1] At the same time we may also optimize the design, so that it will perform better for a given device architecture. We will probably also concern ourselves with placing and routing the design, that is, making the VHDL design fit within the constraints of a programmable logic device or a field-programmable gate array.

2 First Concepts

Upon first encounter, VHDL can seem overly complicated and intimidating. Not only does it provide a very rich language set capable of describing a variety of levels of abstraction, but the language is also meant to describe physical systems. This makes programming in VHDL much different than in conventional languages such as C or Java. Rather than develop algorithms and sequences of manipulations on data in order to achieve a certain goal, the intent in a VHDL program is to model a physical system, to mimic its behaviour. The first and most obvious example of these differences is the use of parallel operations in VHDL. Programming in VHDL will require a different approach and way of thinking to problems compared to the conventional programming languages most are familiar with.

Nonetheless, these differences just described can also be of great aide to the new-comer. Since VHDL is meant to described physical systems, many of the language constructs that will be introduced in this text can be paralleled with real physical objects, removing a great deal of abstractness. In this section, some of the most fundamental language constructs will be introduced in order to start building the necessary knowledge required to develop any VHDL program.

2.1 Signals

In digital circuits we concern ourselves with signals, and likewise in VHDL. To accommodate all the possible signal values that could occur in a digital circuit, dependent in part on design style, the IEEE approved a 9 value system in order to construct a standard that would allow for portable models. This value system is termed the IEEE 1164 standard, can can be seen in Table 1 (the meaning of each signal value will not be discussed here as most are evident and there importance will be seen as the student develops greater knowledge of VHDL and digital simulation).

Table 1: IEEE 1164 Standard Signal Value System

Value	**Meaning**
`U`	Uninitialized
`X`	Forcing Unknown
`0`	Forcing 0
`1`	Forcing 1
`Z`	High Impedance
`W`	Weak Unknown
`L`	Weak 0
`H`	Weak 1
`-`	Don't Care

In order to furnish these signal values and simulate the behaviour of digital signals, VHDL introduces the `signal` object type[2]. Signals can be of different types, such as integer or real (used at a more abstract level), but the most common and useful to be introduced as new material at this point are `std_logic` and `std_logic_vector` (defined in the IEEE library package `std_logic_1164`). Both signal types provide the values in Table 1 on the page before, except that the latter allows the initialization of a "bank" of `std_logic` signals.

Included in the IEEE 1164 standard is a means of dealing with conflicting signals. For example, say a signal is being driven to both `0` and `1` simultaneously. In this case, the IEEE 1164 standard defines the resulting signal to be `X`, forcing unknown. As another example, take the case where a signal is being driven to both `1` and `Z` (high impedance). Here the resulting signal would be `1`. These results are determined by a *resolution function*, and the signals which are defined with such a function are said to be of a *resolved type*.

Signal declaration is best demonstrated by example, as it differs in two contexts to be discussed shortly. Nonetheless, it is generally of the form:

identifier-list: type := expression;

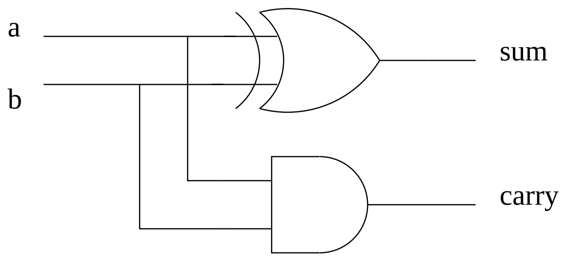
where "*:= expression*" is an optional initialization. Signal declaration will be regarded in greater detail once the next two sections are covered, namely the discussion of the two fundamental programming abstractions in which these declarations differ, as well as some other important programming concepts.

2.2 Entity Declaration & Architecture Body

In order to define a component in VHDL in which we intend to simulate the behaviour of — such as a chip, board, or transistor — we must declare a *design entity*. The behaviour is not described in this declaration, rather the intent is only to define the interface of the component. This interface will be used to input test signals to the design and to read the resulting outputs, not to mention combining the component with other such components in order to create larger and more complex designs. In essence, the VHDL language enforces modular design.

Take for example the half-adder circuit and entity declaration in Figure 2 on the following page. Here the entity is named `half_adder` (error to use hyphens in label names, thus `half-adder` would be incorrect), and with respect to the entity the signals are called *ports*, and thus are defined within the `is port` declaration. We see that the signals `a` and `b` are defined to be inputs and that the signals `sum` and `carry` are defined to be outputs, each of type `std_logic`, which means that they adhere to the IEEE 1164 standard signal value system shown in Table 1 on the page before.

[2]Signals can have a *time value* that describes the delay in the signal – differentiating it from variables in common programming languages – but this will not be discussed in this manual (simply regard it as another of the many features available in VHDL, present to describe specific levels of abstraction).



```
entity half_adder is port(
   a,b      : in  std_logic;
   sum,carry: out std_logic);
end half_adder;
```

Figure 2: Half-Adder Circuit (left) and Entity Declaration (right)

At this point it is important to note that bold face type is used in the VHDL code presented in this document for reserved words, a list of which can be seen in Table 4 on page 27. So for the code just presented, the key words `entity`, `is`, `port`, and `end` are all reserved by the VHDL language, and therefore they can not be used as labels in the code (such as for the name of an entity or a signal). Also, VHDL is not generally case sensitive, so the following are equivalent:

```
entity half_adder is port(...
ENTITY half_adder IS PORT(...
eNtItY hAlF_aDdEr Is PoRt(...
```

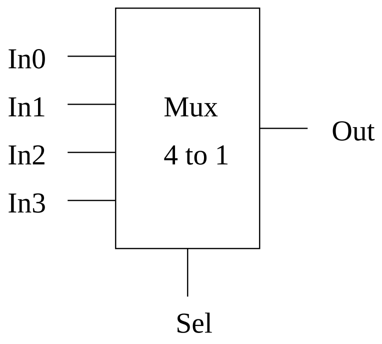
The first two lines in the above example are typically used in VHDL code, the latter to distinguish key words, but it is really just a matter of style (of course the last line of the example is only meant to show that case in not important, it has no real use other than this).

Now consider the 4 to 1, 8 Bit Multiplexer Circuit and Entity Declaration in Figure 3 on the following page. The entity is named `mux` and in this case the input and output signals are declared to be `std_logic_vector`'s, meaning that each encompasses a group of `std_logic` signals (which adhere to the IEEE 1164 value system). For example, the input signals `In0` to `In3` each contain 8 `std_logic` signals, numbered from 7 down to 0. Basically a `std_logic_vector` is a bus, grouping several signals into one for convenience and clarity. The rest of this entity declaration should now seem straight-forward.

Having described the entity declaration — the interface — we must now move on to describing the behaviour of these entities. This is done using an `architecture` body. The basic syntax is:

```
architecture arch_type of the_entity is
   -- architecture declarations
begin
   -- architecture body
end arch_type;
```

The architecture name is just another label, in this called `arch_type`, usually given to distinguish it from other possible variants. Examples could be `behavioural`, `structrual`,



```
entity mux is port(
   In0,In1: in  std_logic_vector(7 downto 0);
   In2,In3: in  std_logic_vector(7 downto 0);
   Sel    : in  std_logic_vector(1 downto 0);
   Out    : out std_logic_vector(7 downto 0));
end mux;
```

Figure 3: 4 to 1, 8 Bit Multiplexer Symbol (left) and Entity Declaration (right)

`high_speed`, `low_power`, etc. The architecture will describe the inner workings of the entity, which is associated to it by placing its name after the key word `of`, in this case called `the_entity`. Obviously then the architecture follows from the entity, but the entity is meaningless without an architecture, hence one usually speaks of an *entity-architecture pair*.

Architecture bodies differ greatly due to the various levels of abstraction for which VHDL is capable of modeling. Hence, these will be discussed in the sections that follow.

2.3 Return to Signal Assignment

In this section we wish to discuss some important concurrent signal assignments (CSAs), necessary to begin building our knowledge of the rich VHDL language. Other important aspects of the language will also be introduced, such as the ability to include libraries, as the need arises.

2.3.1 Simple

Consider a full implementation of the half-adder in Figure 4 on the next page. The CSAs shown here are as straight forward as can be. However, being the first full piece of code presented, its description will require more detail than in following examples.

First notice the `libary` clause, which identifies the library to be used. On the other hand it is coupled with a `use` clause, which identifies the packages or design units to be used in the current design. Here the package name is `std_logic_1164` and `all` components in it will be used. This can be regarded as a standard declaration for the purpose of this document and any designs created subsequent to the information gathered from it should include this library and package. Next is the entity declaration previously seen in Figure 2 on the preceding page (and therefore need not be discussed further). The architecture has been called `concurrent_behaviour`, and is linked to the `half_adder` entity.

Although the CSAs themselves are simple enough (the statements contained within the `begin` and `end` of the `concurrent_behaviour` architecture), it is important not to loose sight of the fact that they are concurrent, so that each statement is evaluated concurrently,

```
library ieee;                    -- standard library
use ieee.std_logic_1164.all;    -- required to use std_logic

entity half_adder is port(
   a,b      : in  std_logic;
   sum,carry: out std_logic);
end half_adder;

architecture concurrent_behaviour of half_adder is
begin
   sum   <= (a xor b);    -- concurrent signal assignments
   carry <= (a and b);
end concurrent_behaviour;
```

Figure 4: Half-Adder Code Using Concurrent Signal Assignments

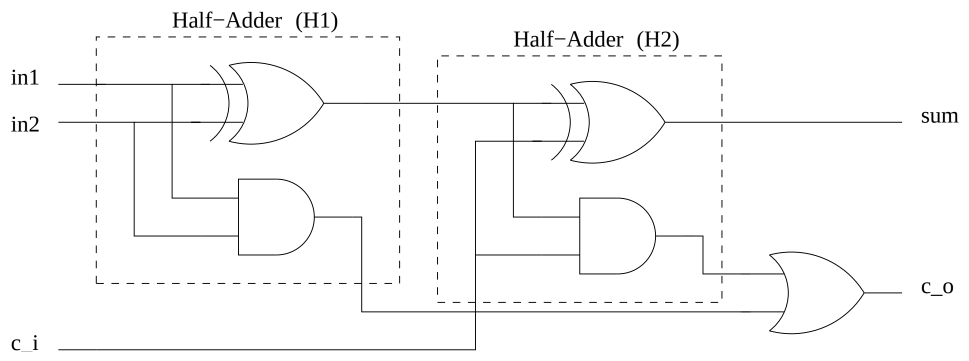
regardless of order. Hence, the `sum` and `carry` signal assignments could be reversed and the model would be unaffected (albeit not the greatest example since either way order is not relevant to this example). So,

```
sum   <= (a xor b);    - same as -    carry <= (a and b);
carry <= (a and b);                   sum   <= (a xor b);
```

CSAs are evaluated when a term on the right hand side changes, in this case the signals `a` or `b` (which in this example both CSAs are dependent upon, although this will not always be the case).

Now consider a more complicated example, that of the full-adder in Figure 5 on the next page. For clarity, both the digital circuit and a full implementation in VHDL have been included together. Of course, the `ieee` library and `std_logic_1164` package have been included, now considered standard for out purposes. Notice that the `full_adder` entity includes only the input and output signals to the design unit, those necessary for interfacing purposes.

In the `dataflow` architecture described, internal signals have been declared before the architecture body, namely `s1` to `s3`, which are also present in the circuit. These internal signals are then used to link various components in the design, and thus complete the physical model of the circuit. As a result, this architecture contains 5 CSA statements, one for each signal. Again recall that the order of the signal assignments is irrelevant, as they are evaluated not in order but concurrently. Whenever a signal changes, if it is contained on the right hand side of a CSA, that CSA will be evaluated. The sequence of execution could be considered for some change in input signals, but as one can see from this example, a change in one signal can set off a whole chain of events!



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

entity full_adder is port(
   In1,In2,c_i: in  std_logic;
   sum,c_o    : out std_logic);
end full_adder;

architecture dataflow of full_adder is
   signal s1,s2,s3: std_logic;   -- architecture declarations
begin
   s1  <= (In1 xor In2);   -- architecture body with CSAs
   s2  <= (c_i and s1);
   s3  <= (In1 and In2);
   sum <= (s1 xor c_i);
   c_o <= (s2 or s3);
end dataflow;
```

Figure 5: Full-Adder Circuit (top) and Code Using Concurrent Signal Assignments (bottom)

2.3.2 Conditional

In the previous 2 examples boolean expressions were used with CSAs to model the circuits. Although the models are legitimate, not all circuits can be described in this fashion without great tedium. Wherefore more complex constructs are now introduced.

Examine the code for the 4-to-1, 8 bit multiplexer using a conditional signal assignment in Figure 6 on the following page. The entity is no different than that presented in Figure 3 on page 7, thus only the architecture need be discussed. A conditional signal assignment is one statement, and thus if there were additional signal assignments in the architecture body besides it they would operate concurrently. Consider it one long CSA, with added flavour.

As in the simple CSAs presented in the preceding section, any change on the right hand side of the signal assignment (i.e. on the right hand side of the `<=`) will cause it to be

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

entity mux is port(
   In0,In1,In2,In3: in  std_logic_vector(7 downto 0);
   Sel            : in  std_logic_vector(1 downto 0);
   Out            : out std_logic_vector(7 downto 0));
end mux;

architecture behavioural of mux is
begin
   Out <= In0 when Sel = "00" else   -- evaluated in order
          In1 when Sel = "01" else
          In2 when Sel = "10" else
          In3 when Sel = "11" else
          "00000000";
end behavioural;
```

Figure 6: 4 to 1, 8 Bit Multiplexer Code Using Conditional Signal Assignment

updated. Thus, if the signals `In0` to `In3` or `Sel` change, `Out` will be evaluated. The most important factor to remember when dealing with conditional signal assignments is that the conditional expressions are evaluated in order. Therefore, when updated, each conditional expression is evaluated from first to last, and once one of the expressions is found to be valid, evaluation terminates (hence the proceeding conditional expressions are not evaluated).

Perhaps now it should be mentioned that when specifying a grouping of bits, such as in a `std_logic_vector`, quotation marks are used (`"..."`). On the other hand, if only one bit where being specified, as for a `std_logic` signal, single quotations are used (`'.'`). Take for example the `Sel` signal above, used to select the input to be directed to the output. If "`Sel = "10"`" then `Out` is connected to `In2`. If this signal were instead split up into two , say `Sel1` and `Sel0`, then for the same results "`Sel1 = '1' and Sel0 ='0'`" would be required.

2.3.3 Selected

A selected signal assignment is very similar to a conditional signal assignment. Consider the code for the 4-to-1, 8 bit multiplexer using a selected signal assignment in Figure 7 on the following page. As in the previous case, this signal assignment is evaluated as one statement, except that here the order in which the select expressions appear is irrelevant, i.e. select expressions are evaluated concurrently.

One element may seem strange in the architecture body, that of the `others` word present in the last select expression. The keyword `others` is used to denote the remaining signal values not yet defined. This brings up a tremendously important aspect of a selected sig-

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

entity mux is port(
   In0,In1,In2,In3: in  std_logic_vector(7 downto 0);
   Sel            : in  std_logic_vector(1 downto 0);
   Out            : out std_logic_vector(7 downto 0));
end mux;

architecture behavioural of mux is
begin
   with Sel select
   Out <= In0 when "00",   -- evaluated concurrently
          In1 when "10",
          In2 when "10",
          In3 when "11",
          "00000000" when others;   -- must cover full range
end behavioural;
```

Figure 7: 4 to 1, 8 Bit Multiplexer Code Using Selected Signal Assignment

nal assignment, that the full range of signal values must be covered. Recall that the IEEE 1164 standard defines a 9 value system (Table 1 on page 4). Attempting to define a select expression for every possible combination would be exhaustive, thus the use of the `others` keyword.

2.4 Summary

Basic building blocks have now been put in place for creating simple VHDL models. The following topics should now be considered essential and basic knowledge:

- IEEE 1164 Standard (Table 1 on page 4) — `std_logic` and `std_logic_vector`, resolved type;
- Entity-Architecture Pair — Physical interpretation, modularity;
- Concurrent Statements — Simple CSAs, conditional signal assignment (evaluated in order), selected signal assignment (evaluated concurrently, must cover full range);
- Basic Constructs — Above topics as well as other fine points described.

3 Behavioural Description

This section will be mainly concerned with presenting a more in depth knowledge of language constructs (as will be the following), focused on behavioural descriptions. Programming a behavioural description of a model will seem more natural to those with previous programming experience (than in the following section regarding structural descriptions). Language constructs presented in this section will follow closely with conventional programming languages. This is mainly since behavioural models can be described in terms of the processing performed on input signals and the output signals produced. Hence an introduction to the `process` construct wherein signal assignments are sequential rather than concurrent.

3.1 Process

Using sequential statements within the `process` construct, more abstract behavioural models will be possible[3]. Take for example the modeling of a CPU or memory module. Approaching such a task using only the CSAs presented thus far would be overwhelming. However, using more conventional programming techniques such as if statements, loops, etc. (to be discussed shortly), the task is greatly simplified.

A process is declared inside an architecture body in two general forms, the first to be discussed being:

```
architecture arch_type of the_entity is
   -- architecture declarations
begin
   label: process(sensitivity list)
   -- process declarations
   begin
      -- sequential statements
   end process label;
end arch_type;
```

Note that the process label is optional, although recommended for clarity.

A *sensitivity list* identifies the signals that will cause the process to execute. Whenever an event occurs on a signal in the sensitivity list, the process is executed sequentially until completion. This is similar to CSAs whereas they are executed when an event occurs on a signal on the right hand side of the assignment. Process declarations will be the variables that are to be used locally (whose declaration will been seen shortly in an example), which can be standard data types such as integers, characters, and real numbers. Moreover, the process can contain assignments to signals declared externally.

[3]Synthesizing these abstract models into real digital circuits is a complex task still being researched and optimized today and thus not suitable for discussion in this introductory manual.

Although statements are strictly sequential within a process, it is possible to modify the order of execution using more advanced constructs. Introducing these constructs now will allow for some much needed examples.

3.2 Order of Execution

3.2.1 If Statement

An `if` statement should not present difficulties to anyone with at least an introduction to any conventional programming language. An expression is executed (relational operators in VHDL include =, /=, <, <=, >, >=) and if true the block of sequential statements that follow are executed. In VHDL a `then` clause must precede the sequential statements (i.e. `if` *expression* `then`, where the expression may be surrounded by parentheses if desired) and the end of the if statement is signified with an `end if`. An `elsif` statement may follow (note the syntax carefully, there is only one 'e' present), as well as an `else`. Consider the first process named `sum_proc` in Figure 8 on the next page for the half-adder circuit.

3.2.2 Case Statement

This construct is well suited for situations in which several branches of execution need be selected based on the value of a single expression. Note however that the case statement must cover all possible values of the expression upon which selection is being made. The `others` key word may again be used as a "catch-all" once the desired selections have already been covered. See again the code in Figure 8 on the following page, this time considering the process named `carry_proc`.

Notice that there is only one sequential statement in each if and case block. This is only due to the simplicity of the example and is by no means a restriction. Also, since two processes have been declared, each will execute concurrently (only the statements within each process are evaluated sequentially, but independent of the other process). The processes are concurrent with respect to one another and can be considered more complex CSAs.

3.2.3 Loops

And of course this section would be incomplete without the all too familiar loop, necessary to perform repetitive operations. There are two general loops in VHDL, a `for` and a `while` loop. However, there are some distinct features to these loops not usually seen elsewhere. Let us begin with some very simple examples, say initializing the elements of a `std_logic_vector` for some unknown architecture, seen in Figures 9 and 10 on page 15.

In order to clarify the examples, some points should be first considered. Notice that an entity has not been explicitly defined here and is being assumed. This is merely intended to simplify the examples and should not be a point of confusion. Also, note that the 8 bit vector could also be initialized using one statement, namely "`sum <= "00000000"`" and therefore these are trivial examples only intended to demonstrate the use of such loops. And finally,

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

entity half_adder is port(
   a,b      : in  std_logic;
   sum,carry: out std_logic);
end half_adder;

architecture behavioural of half_adder is
begin
   sum_proc: process(a,b)       -- using if statement
   begin
      if a = b then
         sum <= '0';
      else
         sum <= (a or b);
      end if;
   end process sum_proc;

   carry_proc: process(a,b)     -- using case statement
   begin
      case a is
      when '0' =>
         carry <= a;
      when '1' =>
         carry <= b;
      when others =>
         carry <= 'x';
      end case;
   end process carry_proc;
end behavioural;
```

Figure 8: Half-Adder Code Using Process with If and Case Statements

```
-- (standard) ieee library; entity declaration

architecture behavioural of the_entity is
   signal sum: std_logic_vector(7 downto 0);
begin
   init_proc: process(clk)
   begin
      for i in 7 downto 0 loop
         sum(i) <= ‘0’;
      end loop;
      -- sequential statements
   end process init_proc;
end behavioural;
```

Figure 9: Simple For Loop, Initializing an 8-bit Vector

```
-- (standard) ieee library; entity declaration

architecture behavioural of the_entity is
   signal sum: std_logic_vector(7 downto 0);
begin
   init_proc: process(clk)
      variable i: integer := 0;
   begin
      while i < 8 loop
         sum(i) <= ‘0’;
         i := i + 1;
      end loop
      -- sequential statements
   end process init_proc;
end behavioural;
```

Figure 10: Simple While Loop, Initializing an 8-bit Vector

note the method upon which vector elements are accessed, using parentheses and the bit number to access with the vector name (i.e. `vector_name(bit_number)`).

First consider the `for` loop in Figure 9 on the preceding page. Notice that the index `i` has not been declared in advance. In VHDL a for loop index is implied and thus only a name need be provided. As the index is implied for the loop, it is only local to it. However, unlike most conventional languages, the for loop index may not be modified inside of the loop. It is only an index, and nothing more.

Now we turn our attention to the `while` loop in Figure 10 on the page before. In this case a loop index is not implied and instead the while loop relies on a condition, like that used in an `if` statement. In this example the condition relies on an unimplied index, created to act like the previous `for` loop. This then introduces a new construct, the `variable`. Note however that a `variable` need not have an initial value. Observe also that a variable assignment uses the "`:=`" operator, and that it is local to the process.

3.3 Wait Statement

In the examples discussed so far in this section, process execution has relied on a sensitivity list, but there is an alternative method that can be used. A process can be activated using a `wait` statement, which will define a specific event to wait for. This is useful when a process will depend upon a clock signal in synchronous circuits. General forms of the `wait` statement include:

wait for *time expression* ;
wait on *signal* ;
wait until *condition* ;
wait;

Using such statements not only can a process be made to wait for specific events, such as the rising edge of a clock, but the process can also be suspended at various points inside of the process (as opposed to just at the beginning).

In order to present the workings of the `wait` statement we need first discuss the *function attributes* in Table 2 and the *value attributes* in Table 3 on the following page. Since the tables give brief descriptions of each attribute, they will not be discussed in greater detail here, but rather will be seen in examples that follow.

Now take the simple example of the code for a positive edge-triggered D flip-flop in Figure 11 on page 18. Here it can be noted that the process does not contain a sensitivity list, but instead has been replaced by a `wait` statement. This wait statement is rather important as it designates a positive edge to trigger the D flip-flop. Such a condition is also sometimes used in an if statement when asynchronous set and reset signals exist (in which case a sensitivity list would be required, since they are not limited by the clock).

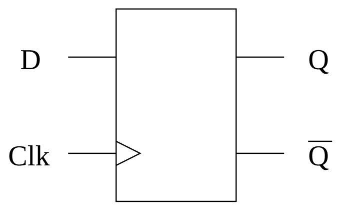
Recall that an IEEE 1164 signal may have up to 9 distinct values, and hence the wait statement used here is somewhat flawed. *Any* event that ends with the `Clk` being the value 1 will cause the D flip-flop to trigger, whereas we would like it to trigger only when the `Clk`

Table 2: Most Common Signal Attributes

Function Attribute	**Function**
`signal_name'`**event**	Return Boolean value if change occurred;
`signal_name'`**active**	Return Boolean value if signal assignment made (not necessarily new);
`signal_name'`**last_event**	Return time since last event occurred;
`signal_name'`**last_active**	Return time since signal last active;
`signal_name'`**last_value**	Return previous value of signal.

Table 3: Most Common Value Attributes

Value Attribute	**Value**
`scaler_name'`**left**	Return left most value in defined range;
`scaler_name'`**right**	Return right most value in defined range;
`scaler_name'`**high**	Return highest value in its range;
`scaler_name'`**low**	Return lowest value in its range;
`scaler_name'`**ascending**	Return true if values in ascending range;
`array_name'`**length**	Return number of elements.
`array_name'`**range**	Return range for a loop.



Clk	D	Q	$\overline{Q}$
$\uparrow$	0	0	1
$\uparrow$	1	1	0

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

entity dff is port(
   D,Clk : in  std_logic;
   Q,Qbar: out std_logic);
end dff;

architecture behavioural of dff is
begin
   output: process     -- no sensitivity list
   begin
      wait until (Clk'event and Clk='1');  -- edge triggered
      Q <= D;                               -- by wait statement
      Qbar <= not D;
   end process output;
end behavioural;
```

Figure 11: Positive Edge-Triggered D Flip-Flop Symbol, Truth Table (top) and Code (bottom)

changes from 0 to 1. Hence, the reason for which the `ieee` library defines two functions to simplify this problem, namely `rising_edge(Clk)` and `falling_edge(Clk)` (of course `Clk` can be any desired signal). These functions return Boolean values and thus are used in a condition (in this case replacing the attributes used for the `wait until` construct).

Further examples of the use of attributes and wait statements can be found in other texts. Nonetheless their importance and use should be understood.

3.4 Summary

The constructs needed to create behavioural models in VHDL have been described in this section, including:

- Processes – Declared in architecture body, sequential statements, sensitivity list or wait statement;

- If statement — `if` *condition* `then`, `elsif` (only one 'e'), `else`, `end if`;
- Case statement — Several branches of execution, must cover full range;
- Loops:
  - `for` — Implied index, cannot modify index;
  - `while` — Relies on condition, usually uses a declared variable local to the process;
- Attributes — Function and signal attributes, often used in wait statements.

4 Structural Description

A structural description of a digital circuit provides a representation of the connections between various components. The behaviour of each component is not dealt with, rather the focus is on the connections themselves. Considering the previous section, this type of model could be regarded as less abstract, and we soon realize that several structural models can be conceived from a single behavioural model. Nonetheless, down the chain of hierarchy (say for example that each component is also modeled using structural descriptions), at some point the models must be behavioural, even if this must be attained at the gate level. This is much like the actual wiring of digital circuits, whereas chips are created and then used in a design, and then this design may be packaged and then used in another design, and so on. The task of this section will be to introduce such structural models in VHDL.

4.1 Describing Connections

Recall the full-adder circuit presented in Figure 5 on page 9. The simplest means of describing this circuit structurally is to consider each half-adder circuit and the 2-input or gate as individual components. Do not consider the inner workings of these components. Instead, focus on the interconnections between them, this is what needs to be described in a structural model.

In order to describe the interconnections between components, the components themselves must be distinguishable from one another. There could be multiple components being used that are of the same type, as is the case here, where two half-adder circuits are being used in the construction of the full-adder circuit. Although the components are the same, their connections do differ and as such must be distinguishable from one another. Notice that in the full-adder circuit the labels `H1`, `H2`, and `O1` have been used to distinguish the two half-adder circuits and the 2-input or gate, respectively.

Now examine the structural description of the full-adder circuit in Figure 12 on the next page. First notice that the entity declaration is no different than in previous examples; the structural description is in the architecture only, as expected. Next we see the `component` declarations. Basically they are internal entity declarations for the components to be used, rightfully so since entity-architecture pairs describing the inner workings of each of component is required. Hence, the component declarations must match an entity-architecture pair. Also, be aware that the number of components is not specified. The number of components used will become evident in the architecture body when the interconnections are specified for each component, thus eliminating redundancy. And next in the architecture we see the signal declarations, which specify the signals that will make the interconnections between components.

Finally we arrive at the architecture body, where the interconnections we have discussed so far are actually specified. Although arguably the most important part of a structural description, with little effort one sees that they are relatively straight-forward. Components are given unique labels to distinguish them from one another, components are

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

entity full_adder is port(
   in1,in2,c_i: in std_logic;
   sum,c_o    : out std_logic);
end full_adder;

architecture structural of full_adder is
   component half_adder port(        -- component declarations
      a,b      : in std_logic;
      sum,carry: out std_logic);
   end component
   component or_2 port(
      a,b:in std_logic;
      c  :out std_logic);
   end component
   signal s1,s2,s3: std_logic;        -- signal declarations
begin
   H1: half_adder port map(           -- component interconnections
      a=>in1, b=>in2, sum=>s1, carry=>s3);
   H2: half_adder port map(
      a=>s1, b=>c_i, sum=>sum, carry=>s2);
   O1: or_2 port map(
      a=>s2,b=>s3,c=>c_o);
end structural;
```

Figure 12: Full-Adder Code Using Structural Description

then specified with their name, and then `port map`'s are made. The latter is nothing more than the specification of the ports connecting to interconnections, using '`=>`' to specify the connection from the component to the signal which will provide the interconnection between it and other components.

4.2 Hierarchy & Abstraction

It is important to understand and remember that structural models require entity-architecture pairs to match each component. These components could be structural as well, with entity-architecture pairs matching its components, but at some point the hierarchy must end with behavioural models. Take for example the full-adder model previously described. If the half-adders used to describe it were also structural models, than it would require entity-architecture pairs for a 2-input and gate and a 2 input xor gate (see the original circuit for the half-adder in Figure 2 on page 6). The resulting hierarchy for the full-adder would be that shown in Figure 13.

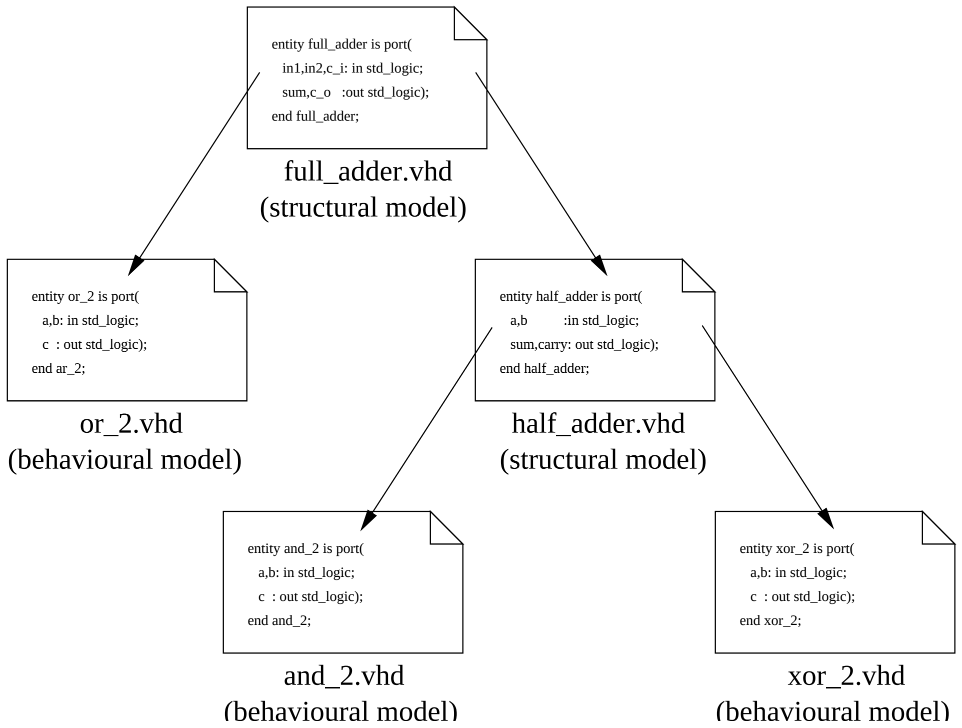


Figure 13: Hierarchy of Full-Adder Structural Model

Although simple in structure, it is evident that for large designs the connections in a structural model could be substantially complex and numerous, the reason for which a hierarchical approach is most often taken. I.e. the top-level design is broken up into large blocks (components) and interconnections are made, then the process is repeated iteratively for each block (component) until the final descriptions provided are behavioural. The point at which behavioural descriptions would be provided is dependent on the current desired level of abstraction of the design. As the design progresses and comes closer to being implemented, the behavioural descriptions can be replaced with structural descriptions decreasing the level of abstraction until, perhaps, the design attains a gate level representation (of course, the hierarchical representation ends at behavioural models, hence the gate level models would have to be behavioural).

One last point that should be obvious, once considered. Since the structural models depend on components not described in that model, the components must be analyzed in advance (i.e. the entity-architecture pairs of components must be analyzed in advance). Analyzing a structural model without having defined the entity-architecture pairs for each component would result in errors, since the model would have no meaning. Interconnections between components that don't exist are hence meaningless.

4.3 Generics

Now would be an appropriate time to introduce the `generic` construct. This is a convenient feature used to parameterize a model. Figure 14 shows a generic behavioural model of an exclusive-or gate. The number of inputs are parameterized by the value `n`, whose default value is 2. The rest of the code need not be described as it should be somewhat self explanatory. Nonetheless, examine the code to ensure that it is clear and well understood.

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

entity generic_xor is generic(n: positive:=2) port(
   xor_in : in  std_logic_vector((n-1) downto 0);
   xor_out: out std_logic);
end generic_xor;

architecture behavioural of generic_xor is
begin
   process(xor_in)
      variable result: std_logic := '0';
   begin
      result := '0';
      for i in 0 to (n-1) loop
         result := result xor xor_in(i);
      end loop;
      xor_out <= result;
   end process;
end behavioural;
```

Figure 14: Generic Exclusive-Or Gate

Now we continue with an example of using the generic exclusive-or gate in Figure 15 on the next page. In this example the `generic_xor` component instantiation is the same as the entity declaration for the `generic_xor` except that the value of `n` is unspecified, and a **generic map** is used to declare the value of `n` to be 2. Note that this is the same as the default value specified in the `generic_xor` entity declaration, and is somewhat redundant. If the `generic` and `generic map` constructs had not been included in the half-adder code, this default value would have been used. However, not only does this code demonstrate how to use the `generic map` to specify the value of `n` but it also ensures that changes in the default value of `n` in the `generic_xor` entity declaration will not affect the half-adder model.

The value of `n` could also have been specified in the component declaration by changing the generic parameter such that:

```
generic(n: positive:=2);
```

which thus would make it match the `generic_xor` entity declaration exactly, but of course the value of `n` specified (i.e. 2) could differ[4].

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

entity half_adder is port(
   a,b      : in  std_logic;
   sum,carry: out std_logic);
end half_adder;

architecture structural of half_adder is
   component generic_xor generic(n: positive) port(
      xor_in : in  std_logic_vector((n-1) downto 0);
      xor_out: out std_logic);
   end component;
   component and_2 port(
      and_in0,and_in1: in  std_logic;
      and_out        : out std_logic);
   end component;
begin
   X1: generic_xor generic map(n => 2) port map(
      xor_in(0)=>a, xor_in(1)=>b, xor_out=>sum);
   A1: and_2 port map(
      and_in0=>a, and_in1=>b, and_out=>carry);
end structural;
```

Figure 15: Half-Adder Code Using Structural Description and Generic Exclusive-Or Gate

4.4 Configurations

Since many variations in model type are possible in VHDL, it is important to know how to *bind* models together. For example, one architecture could be optimized for speed whereas another for lower power consumption. If several architectures exist for the same component, then a *configuration* must be specified. This can become quite complicated as there is a

[4]Actually, it is also possible to pass down generic values from multiple levels of hierarchy, but to alleviate any further possible confusion our discussion will not go any further than it already has.

substantial amount of flexibility and options available and hence only one simple method of combining components will be discussed herein, once default configurations are discussed.

In the structural model of the full-adder previously discussed (See Figure 12 on page 21), no configuration was specified. It was assumed that the component declarations would match entity declarations (found in the working directory). However, if there were multiple architectures for an entity then the architecture most recently analyzed would be used to analyze the structural model. On the other hand, if it were desired that another architecture be used (rather than the last one analyzed), or if the component did not have a matching entity (which should of course be avoided for the sake of clarity), then a configuration would have to be specified.

A configuration can be specified among the architecture declarations (before the `begin` clause). The general syntax is:

```
for label: component_name use entity library_name.entity_name (arch_name);
```

This should be fairly straight-forward and thus not require additional explanation (if not clear the following example should alleviate confusion).

Take for example the structural description of the full-adder again. Added to the architecture declarations could have been the lines shown in Figure 16. Although the first half-adder is expected to be found in the library `work` according to the first configuration, this is actually a keyword usually used to specify the current working directory. The two half-adders differ in only the fact that the first will use the `behavioural` architecture and the second the `structural` architecture. On the other hand, the 2-input or gate will be expected to be found in the `or_gates` library with the entity name `or` and architecture named `structural`.

```
-- (standard) ieee library; full_adder entity declaration
-- see Figure 12 on page 21

architecture structural of full_adder is
   -- architecture declarations; components and signals
   for H1: half_adder use entity work.half_adder (behavioural);
   for H2: half_adder use entity work.half_adder (structural);
   for O1: or_2 use entity or_gates.or (structural);
begin
   -- architecture body; port map's
end structural;
```

Figure 16: Full-Adder Configuration Specified in Architecture Declarations

4.5 Summary

This section has introduced the constructs needed to created structural models in VHDL, including:

- Describing Connections — `component` declarations (matching with entity-architecture pairs, unique labels), `port map`'s;
- Hierarchy and Abstraction — hierarchy of model must end with behavioural descriptions, entity-architecture pairs of components need to be analyzed in advance;
- Generic — parameterizing a model, `generic` construct and `generic map`'s;
- Configurations — to specify binding of components to entity-architecture pairs, default binding, configuration specification (an architecture declaration);

A Quick Reference

This section is not meant to cover *all* possible declarations, but hopefully the most relevant are present. Note that terms surrounded by square braces (`[...]`) in the code presented are optional, and therefore are not required (the square braces themselves are not to be included in the code).

A.1 Reserved Words

The following list of reserved words is by no means complete, but contains most (if not all) of the reserved words that are of interest to the beginner. The boolean functions `and`, `nand`, `nor`, `not`, `or`, `xnor`, `xor` are found in the `ieee` library, in the `std_logic_1164` package (as may be others).

Table 4: Reserved Words

abs	access	active	after	alias
all	and	architecture	array	ascending
assert	attribute	begin	bit	bit_vector
body	boolean	buffer	case	character
component	configuration	constant	downto	else
elsif	end	entity	error	event
exit	file	for	function	generic
high	if	in	inertial	inout
is	integer	last_active	last_event	last_value
left	length	library	line	loop
low	map	mod	nand	natural
nor	not	null	of	on
open	or	others	out	package
port	positive	procedure	process	range
read	real	reject	rem	report
return	right	rol	ror	severity
signal	sla	sll	sra	srl
string	subtype	text	transport	then
time	to	type	units	until
use	variable	wait	when	while
with	write	xnor	xor	-

A.2 Declarations

A.2.1 Entity Declaration

```
entity entity_name is port(
   -- interface signal declarations
);
end entity_name;
```

A.2.2 Architecture Body

```
architecture arch_name of entity_name is
   -- declarations:
   -- signal declarations, constant declarations
   -- component declarations, alias declarations
begin
   -- architecture body
end [architecture] arch_name;
```

The architecture body may or may not include processes.

A.2.3 Library Declarations

```
library library_name1,library_name2,...;
```

To use a library, we must declare a `use` statement as follows:

```
use library_name.package_name.item;
```

`.item` will usually be `.all` in this course.

A.2.4 Package Declarations

To create a user-defined library, we need to declare the packages contained in the library. This is done as follows:

```
package package_name is
   -- package declarations
end package [package_name];
```

Packages in turn contain components, defined in the next section.

A.2.5 Component Declarations

```
component component_name port(
   -- interface signal declarations
   );
end component [component_name];
```

Components can be declared within an architecture, or within a library. When declared within a library, components must be contained within packages (see above). Components are instantiated as follows:

```
label: component_name port map(port=>signal,...);
```

A.2.6 Signal Declarations

```
signal signal_name1,signal_name2,...: type [:= initial value];
```

Interface signal declarations look like:

```
signal_name1,signal_name2,...: mode signal type;
```

A.2.7 Constant Declarations

```
constant constant_name: type := constant_value;
```

A.2.8 Alias Declarations

```
alias identifier is item_name;
```

A.2.9 Variable Declarations

```
variable var_name: type [:= initial value];
```

A.2.10 Integer Type Declarations

```
int˙type type is range integer_range;
```

A.3 Simple Assignment Statements

A.3.1 Signal Assignment

```
signal <= expression;
```

Concurrent statements are recalculated every time the expression on the right-hand side of the equation changes.

A.3.2 Variable Assignment

```
variable := expression;
```

Variables can only be declared within a process (for our purposes) and are local to the process in which they are declared. Variables are updated immediately.

A.4 Concurrent Statements

A.4.1 when-else

```
signal <= expression1 when condition1 else
          expression2 when condition2 else
          ...
      [expressionN];
```

A.4.2 with-select-when

```
with select expression select
signal <= expression1 when condition1;
          expression2 when condition2;
       ...
      [expressionN when others];
```

A.5 Sequential Statements

A.5.1 Process Declaration

```
[process_label:] process (sensitivity list)
   [-- local constant/variable/alias declarations]
begin
   -- sequential statements:
   -- signal assignment, variable assignment,
   -- if statements, case statements, loop statements
end process [process_label];
```

Process labels are used to identify the functionality of the process; a process label is not mandatory but is strongly recommended.

A.5.2 if-then-else

```
if condition1 then
   -- sequential statements
[elsif condition2 then
   -- sequential statements]
[else
   -- sequential statements]
endif;
```

A.5.3 case-when

```
case expression is
```

```
    when choice1 =>
      -- sequential statements
    when choice2 =>
      -- sequential statements
    ...
   [when others =>
      -- sequential statements]
  end case;
```

A.5.4 for-loop

```
  for index in range loop
    -- sequential statements
  end loop;
```

A.5.5 while-loop

```
  while condition loop
    -- sequential statements
  end loop;
```

A.5.6 Synchronous Logic with Asynchronous Reset

```
  [process-label:] process (reset, clock)
    [-- local constant/variable/alias declarations ]
  begin
    if reset = '1' then
      -- asynchronous reset assignment statements
    elsif clock'event and clock = '0' then
      -- synchronous assignment statements
  end process [process-label];
```

A.6 Modes

A.6.1 in

Used to describe a signal that is an input to an entity. Such signals can ONLY be used as inputs.

A.6.2 out

Used to describe signals that may ONLY be used as outputs from an entity. The **out** signals are not required internally within an entity.

A.6.3 inout

Used to describe signals that may be used as inputs and outputs to an entity. Useful when hierarchically creating components or dealing with bi-directional signals (such as feedback signals).

A.6.4 buffer

A buffer signal is an output signal, where the signal's values are also required internal to an entity.

References

[1] Sudhakar Yalamanchili. *VHDL Starter's Guide*. Prentice Hall, New York, New York, 1998.

[2] Kevin Skahill. *VHDL for Programmable Logic*. Addison-Wesley, New York, New York, 1996.

[3] Charles H. Roth. *Digital Systems Design Using VHDL*. ITP Nelson, New York, New York, 1997.

[4] M.M. Mano and C. Kime. *Logic And Computer Design Fundamentals*. Prentice Hall, New York, New York, 2000.