ANDREW KENNINGS

FAST INTRODUCTION TO VHDL FOR ECE 124

1 *Preface*

The purpose of these notes is to provide a fast introduction to **VHDL** which is suitable for the laboratories in ECE 124.

2 *Introdution*

The acronym **VHDL** stands for **V**HSIC **H**ardware **D**ecsciption **L**anguage. The acronym **VHSIC**, in turn, stands for **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit. **VHDL** was a result of a program sponsored by the US Department of Defense with the goal of developing a new generation of high-speed circuits. **VHDL** is one of several hardware descriptive languages that can be used to describe circuits.

At a certain point, digital circuit design became increasingly complex and therefore a standarized representation of a digital circuit was required such that circuit descriptions could be more easily shared amoung different designers. This standard representation became a *language* for describing digital circuits. The language itself because the **IEEE Standard 1076-1987** with new features added later which resulted in the **IEEE Standard 1076-1993**.

The main point is that schematic diagrams became an ineffective way to design and describe digital circuits. It is more common to “program” the description of a circuit in a “programming language” and let a software tool convert the description into logic gates and so forth.

3 First look

We can get a gentle introduction to **VHDL** and observe some of the necessary pieces of the language by considering how to implement a simple combinational circuit. Consider the truth tables in Figure 3.1 and the corresponding circuit in Figure 3.2 which implements these truth tables.

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

a	b	c	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 3.1: Two truth tables

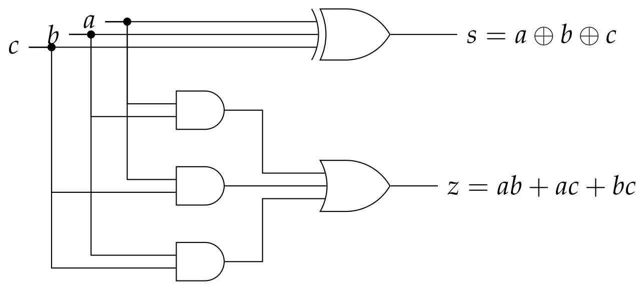

Figure 3.2: Circuit implementing the truth tables in Figure 3.1.

The **VHDL** description of this circuit is provided in Figure 3.3. We can examine this simple **VHDL** description to understand some of the basic syntax and structure of a **VHDL** description.

```
-- VHDL code for First Look
library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
  port( a,b,c   : in std_logic;
        s,z     : out std_logic);
end FullAdder;
architecture prototype of FullAdder is
begin
  s <= a xor b xor c;                        -- equation for s
  z <= (a and b) or (a and c) or (b and c);   -- equation for z
end prototype;
```

Figure 3.3: **VHDL** description of the circuit in Figure 3.2.

3.1 Comments

We can have comments in a VHDL description just like in any programming language. Comments can be on a line by themselves or after other **VHDL** syntax (e.g., at the end of a line). A comment is indicated by a double dash "--".

3.2 Entities and architectures

A **VHDL** description of a circuit is called a *design entity* and consists of two parts:

1. Entity declaration;

2. Architecture definition.

The **entity declaration** describes the *interface* of the circuit to the rest of the world (e.g., the inputs and outputs of the circuit). This is similar to the function prototype in a programming language. The entity declaration has a specific syntax shown in Figure 3.4.

```
ENTITY entity_name IS
  PORT(
    SIGNAL signal_name : mode type ;
    SIGNAL signal_name : mode type ;
    ...
    SIGNAL signal_name : mode type ) ;
END entity_name;
```

Figure 3.4: Syntax for an entity declaration. **VHDL** keywords needed for the entity declaration are UPPERCASE.

Entity declarations have a name and a port. The port is where the input and output signals of the circuit are declared. Signals declared here must have a **mode** and a **type** which we will talk about later.

The **architecture definition** describes one *implementation* of the circuit. It's like the body of a function in a programming language. The architecture definition has a specific syntax shown in Figure 3.5. The architecture definition itself has a declarative section and an implementation section. The declarative and implemention sections may or may not include other types of statements.

In our VHDL example, we have an entity declaration with a port that defines 5 signals: a, b, c, s and z. We have an architecture definition that provides the circuit implementation. Our architecture definition has nothing in its declarative section. The statements inside of the implementation section of the architecture definition are examples of CONCURRENT ASSIGNMENT statements.

```
ARCHITECTURE architecture_name OF entity_name IS
    -- declarative section
    [SIGNAL declarations]
    [CONSTANT declarations]
    [TYPE declarations]
    [COMPONENT declarations]
    [ATTRIBUTE declarations]
BEGIN
    -- implementation
    [COMPONENT instantiation statements]
    [CONCURRENT ASSIGNMENT statements]
    [PROCESS statements]
    [GENERATE statements]
END architecture_name ;
```

Figure 3.5: Syntax for an architecture definition. **VHDL** keywords are shown in UPPERCASE. Items shown in "[" and "]" are different optional pieces of syntax that might appear.

3.3 *Data objects and signals*

Before we go any further, we need to discuss signals. **VHDL** stores information via **data objects**. There are three types of data objects, namely **signals**, **constants** and **variables**. We will be mostly concerned with signals. Basically, a signal can be thought of as a wire that connects things together and carries logic values.

Signals have names and these names can contain any alphanumeric characters and the underscore symbol. Some restrictions apply:

- Signal names must begin with a letter;
- Signal names cannot have two consecutive underscores;
- Signal names cannot end with an underscore;
- Signal names cannot be the same as any **VHDL** reserved word;
- Signals names are case insensitive.

Signals can be declared in three places in a **VHDL** description:

- The port of an entity declaration;
- The declarations section if an architecture description;
- The declarations section of a package.

We won't talk too much about packages. **In our VHDL example, we have 5 signals: a, b, c, s and z which are all declared in port of the entity declaration**.

Signals must have a **type**. Declaring a signal to have a certain type is identical to declaring a variable in a conventional programming language to be of type `int`, `unsigned`, `float`, etc. **In our VHDL example, our signals are all of type std_logic**. Finally, if declared in the port of the entity declaration, signals must also have a **mode**. The mode describes the direction and use of a signal which enters (or leaves) the circuit.

3.4 *Operators*

We can have several different sorts of operators in **VHDL** descriptions in different statements which appear in the implementation section of the architecture description. Such operators include:

- **Boolean operators** including AND, OR, NOT, XOR, NAND, NOR, etc.;
- **Relational operators** including $=$ (equality), $/=$ (not equality), $<$ (less than), $>$ (greater than), etc.;
- **Arithmetic operators** including $+$, $-$, &, etc.

In our VHDL example, we've used some Boolean, but we have not used and relational or arithmetic operators.

3.5 *Concurrent signal assigments*

We need to be able to assign values to signals. Compared to a schematic, this is comparable to connecting inputs to gates, connecting the gates together and then taking the output of the assignment to be the output of the circuit. One way to assign values to signals is to use a **concurrent signal assignment** which has the syntax shown in Figure 3.6.

```
signal_name <= expression;
```

Figure 3.6: Syntax for the concurrent signal assigment statement.

In our first example, we have 2 concurrent signal assigment statements in order to generate the signals s **and** z.

The concept of concurrency is important in **VHDL**. **VHDL** describes a digital circuit — in a circuit everything is always operating in parallel. Therefore, **VHDL** is an example of a concurrent programming language. **All concurrent signal assignments are operating or being evaluated at the same time**. Hence, the order of concurrent signal assignments is **irrelevant**. If we consider Figure 3.7, both descriptions in (a) and (b) are equivalent. We should think of concurrent signal assignments working as follows: At time t, the values of all signals on the right-hand side of concurrent signal assignment statements are saved. All left-hand side values are computed and updated at time $t + \delta t$ using the right-hand side values *from time* t. This is entirely different from a sequential programming language — in such a language the order of the statements in Figure 3.7 is relevant.

```
 e   <=   c or d;        d   <=   a and b;
 d   <=   a and b;       e   <=   c or d;       .
         (a)                      (b)
```

Figure 3.7: The order of **VHDL** concurrent signal assignmenta statements doesn't matter; (a) and (b) will produce the same circuit.

3.6 *Signal modes*

Recall that signals can be declared inside of the port of an entity declaration require a **mode**. The mode indicates the direction of the signal and how the value is used inside of the circuit. There are four possible modes:

1. **IN** — the signal is generated outside of our ciruit and is therefore an input to our circuit. It connects to inputs of elements inside our circuit;

2. **OUT** — the signal is generated within our circuit and is output from our circuit. However, the signal is *not* used to drive other circuit elements within the circuit.

3. **BUFFER** — the signal is generated within our circuit and is output from our circuit. Further, the signal is used to drive other circuit elements within the circuit.

4. **INOUT** means that the signal is generated either within our circuit or outside of our circuit; i.e., the signal corresponds to a wire which is bi-directional.

These different signals modes are described in Figure 3.8. Note that I haven't shown **INOUT** (explaining the **INOUT** requires drawing a circuit element that we likely don't know about yet so I'll avoid that confusion right now).

3.7 *Signal types*

We might think to define a signal in **VHDL** to be something like a `bool`. However, this is not sufficient since signals represent physical wires in a circuit and wires can experience more "conditions" that

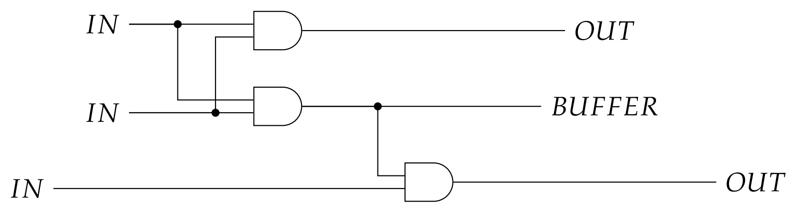

Figure 3.8: Illustration of signals modes by means of a circuit diagram.

just being 0 or 1. For example, wires can become disconnected or shorted. Voltages on wires might have more or less driving strength (ability to provide sufficient power).

The short story is that a standard definition for signals in **VHDL** was introduced in **IEEE Standard 1164**. This standard introduces the signal type **std_logic** which can be used for signal definitions in **VHDL** and accounts for situations other that 0 or 1 values on wires. There is also a type **std_logic_vector** if we want to declare a bundle of wires at once. These types are defined in a **library** as part of a **package**.

To quickly get to the point — we want to use the signal type **std_logic**. To define signals to be this type and to perform operations with these types of signals, we need to have the statements shown in Figure 3.9 at the top of our **VHDL** code. These statements basi-

```
-- following lines before every VHDL entity declaration.
library ieee;
use ieee.std_logic_1164.all;
```

Figure 3.9: Statements at the top of a **VHDL** description in order to declare and manipulate signals of **std_logic**.

cally say that we want to use **all** of (everthing defined inside of) the package **std_logic_1164** which is defined (is part of) the library **ieee**. These statements are similar in concept to an `#include` statement in a conventional programming language.

3.8 *Another example*

Now that we have read through and clarified the syntax of a simple **VHDL** description, we should be able to write some code ourselves. Consider writing **VHDL** code for the sum of product equation $f = f(a, b, c) = \sum(0, 1, 3, 4, 5)$. The **VHDL** code is shown in Figure 3.10 with comments.

Figure 3.10: The **VHDL** description to implement $f = f(a,b,c) = \sum(0,1,3,4,5)$.

```
-- needed to define signals to be std_logic
-- and to manipulate such signals
library ieee;
use ieee.std_logic_1164.all;

-- entity for our function.  the entity declares three
-- inputs and one output signal
entity SomeFunction is
  port( a, b, c     : in    std_logic;
        f           : out   std_logic
        );
end SomeFunction;

-- architecture describing the implementation.
architecture prototype of SomeFunction is
    -- nothing in the declarative portion
begin
    -- one concurrent signal statement to implement the SOP
    f <=            ((not a) and (not b) and (not c))   -- minterm 0
                 or ((not a) and (not b) and c)         -- minterm 1
                 or ((not a) and b       and )          -- minterm 3
                 or (a       and (not b) and (not c))   -- minterm 4
                 or (a       and (not b) and c);        -- minterm 5
end prototype;
```

4 Component instantiations

Perhaps we have already designed a circuit that would be useful to use in the creation of another circuit. One simple example would be the creation of a n-bit ripple adder using multiple copies of a 1-bit full adder circuit that has previously been designed.

We can use one **VHDL** description inside of another **VHDL** description via **component instantiations**. To use a previously created circuit inside of a new **VHDL** description requires three things:

1. Declaring the components that we wish to use;

2. Instantiating copies of the components;

3. Connecting the components together (which might require declaring new signals).

For example, consider that we have already designed a 1-bit full adder in **VHDL** as shown in Figure 4.1. We would now like to design a 4-bit ripple adder using 4 copies of the 1-bit full adder.

```
-- VHDL for 1-bit full adder.
library ieee;
use ieee.std_logic_1164.all;
entity fulladder1 is
    port( a,b,c     : in    std_logic;
          s,z       : out   std_logic);
end fulladder1;
architecture prototype of fulladder1 is
begin
    s <= a xor b xor c;
    z <= (a and b) or (a and c) or (b and c);
end prototype;
```

Figure 4.1: **VHDL** description of a 1-bit full adder.

The **VHDL** description of our 4-bit full adder is shown in Figure 4.2. We can disect this **VHDL** in Figure 4.2 to learn some more **VHDL**.

Figure 4.2: **VHDL** for a 4-bit adder using component istantiations.

```
-- VHDL for a 4-bit adder built from 1-bit adders.
library ieee;
use ieee.std_logic_1164.all;

entity fulladder4 is
    port (  x,y   : in  std_logic_vector(3 downto 0);
            cin   : in  std_logic;
            sum   : out std_logic_vector(3 downto 0);
            cout  : out std_logic);
end fulladder4;
architecture prototype of fulladder4 is
    -- declare the component we will instantiate.
    component fulladder1
      port (a,b,c : in std_logic; s,z : out std_logic);
    end component;

    -- temporary signals required to connect things.
    signal c1, c2, c3 : std_logic;
begin
    -- instantiate multiple copies of the 1-bit adder.
    bit0: fulladder1
        port map (a=>x(0),b=>y(0),c=> cin,s=> sum(0),z=>   c1);
    bit1: fulladder1
        port map (a=>x(1),b=>y(1),c=>  c1,s=> sum(1),z=>   c2);
    bit2: fulladder1
        port map (a=>x(2),b=>y(2),c=>  c2,s=> sum(2),z=>   c3);
    bit3: fulladder1
        port map (a=>x(3),b=>y(3),c=>  c3,s=> sum(3),z=> cout);
end prototype;
```

To use a component inside of another **VHDL** description, we need to declare (in the architecture declaration portion) the **component** that we are want to use. This was done with the statement:

```
component fulladder1
  port (a,b,c : in std_logic; s,z : out std_logic);
end component;
```

The declaration of the component closely follows the same syntax is the entity declaration of the component that we want to use.

This description also requires that we create some new signals that *exist only within our circuit* in order to connect some things together. The declaration of new signals is done inside of the declaration portion of the architecture as follows:

```
signal c1, c2, c3 : std_logic;
```

We have *instantiated* or *created* 4 separate copies of the 1-bit adder component. To be clear — unlike a software program where we might "call" a function 4 separate times, this is not what is happening here. We are, in fact, creating 4 separate copies of the component. The syntax for instantiating a copy of a component is shown in Figure 4.3. In Figure 4.3, the port map is where we connect signals

```
instance_name : component_name
    port map (  signal_in_component => signal_in_current_circuit,
                signal_in_component => signal_in_current_circuit,
                ...
                signal_in_component => signal_in_current_circuit
             );
```

Figure 4.3: **VHDL** syntax to instantiate and to connect signals to the inputs and outputs on the component.

in the current circuit to the inputs and outputs of the circuit we've just made a copy of. The syntax "=>" inside of the port map is intended to mean that we are connecting or wiring the signal in the current circuit to an input or output pin of the component. Finally, in this example, we've used **std_logic_vector** to allocate a bundle of wires at the same time. An example of such syntax is:

```
sum   : out std_logic_vector(3 downto 0);
```

This means we will have signals `sum(0)`, `sum(1)`, `sum(2)`, and `sum(3)` — 4 signals as part of the bundle of signals "sum".

5 More combinational assignment statements

The concurrent signal assignment "<=" is not the only means by which we can perform assignments in **VHDL**. There are a couple of other statements which provide a more *behavioural* way to express how a circuit operates. We'll discuss these here. Also, we can give some examples of *constants* to both **std_logic** types and **std_logic_vector** types.

5.1 Conditional signal assignments

There is another operator construct called **conditional signal assignment** which we can use. The syntax of the conditional signal assignment is shown in Figure 5.1. Note that we *require* a *default*

```
signal  <=  other_signal_1 WHEN (condition1) ELSE
            other_signal_2 WHEN (condition2) ELSE
            other_signal_3 WHEN (condition3) ELSE
            ...
            other_signal_n WHEN (conditionn) ELSE
            default_signal ;
```

Figure 5.1: **VHDL** syntax for conditional signal assignment.

assignment for the situation when none of the conditions are true. The purpose of the conditional signal assignment is to provide a *higher level of abstraction* when describing a particular operation. For example, in Figure 5.2 is the **VHDL** description of a 4-to-1 multiplexer using a concurrent signal assignment. The use of a concurrent signal assignment requires that we determine the logic equation for the multiplexer output in terms of the logic gates required — this is a very "low level" or "structural description" of the multiplexer and, although it correctly implements the multiplexer, doesn't really "describe the behaviour" of the multiplexer.

The description of the same multiplexer, but using a conditional

```
-- 4-to-1 multiplexer using concurrent signal assigment
library ieee;
use ieee.std_logic_1164.all;

entity mux4to1_version1 is
    port (  x1, x2, x3, x4 : in std_logic;      -- inputs
            s              : in std_logic_vector(1 downto 0); -- controls
            f              : out std_logic); -- output
    end entity;
architecture prototype of mux4to1_version1 is
begin
    f <= (not s(0) and not s(1) and x1) or
         (not s(0) and s(1)     and x2) or
         (s(0)     and not s(1) and x3) or
         (s(0)     and s(1)     and x4);
end prototype;
```

Figure 5.2: **VHDL** description of a 4-to-1 MUX using a concurrent signal assignment.

signal assignment, is shown in Figure 5.3. This version is more descriptive. Also note that we can use the "=" as an equality operator to compare the contents of a **std_logic_vector** to a constant bit vector of "00". Finally, note that there is an implied ordering or priority when using a conditional signal assignment. The ordering of the "when-else" is important — it can be the case that multiple conditions are true, but the first true condition encountered is the one that will determine the resulting assignment.

To illustrate the usefulness of the conditional signal assignment and the priority of the "when-else" one can consider the **VHDL** description of an 8-to-3 priority encoder which is shown in Figure 5.4. In this description, it is almost always the case that multiple "when-else" conditions are true at the same time and therefore the order in which they are listed becomes important to ensure the encoder works as expected. The priority encode also shows how to compare a **std_logic** signal to a constant; the constant is enclosed by single tick marks (unlike a **std_logic_vector** which requires quotation marks for comparison to a constant).

Figure 5.3: **VHDL** description of a 4-to-1 MUX using a conditional signal assignment.

```
-- 4-to-1 multiplexer using conditional signal assigment
library ieee;
use ieee.std_logic_1164.all;

entity mux4to1_version2 is
    port (  x1, x2, x3, x4 : in std_logic;      -- inputs
            s              : in std_logic_vector(1 downto 0); -- controls
            f              : out std_logic); -- output
    end entity;
architecture prototype of mux4to1_version2 is
begin
    f <= x1 when (s = "00") else
         x2 when (s = "10") else
         x3 when (s = "01") else
         x4;
end prototype;
```

Figure 5.4: **VHDL** description of a 8-to-3 priority encoder using a conditional signal assignment.

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

entity priority8 is
    port (x     : in  std_logic_vector(7 downto 0);
          valid : out std_logic;
          y     : out std_logic_vector(2 downto 0));
end priority8;

architecture prototype of priority8 is
begin
    -- encoded output.
    y <=    "111" when (x(7) = ’1’) else -- value of lower inputs not important
            "110" when (x(6) = ’1’) else
            "101" when (x(5) = ’1’) else
            "100" when (x(4) = ’1’) else
            "011" when (x(3) = ’1’) else
            "010" when (x(2) = ’1’) else
            "001" when (x(1) = ’1’) else
            "000" when (x(0) = ’1’) else
            "000";

    -- valid output.
    valid <= ’0’ when x = "00000000" else ’1’;
end prototype;
```

5.2 *Selected signal assignment*

The **selected signal assignment** is very similar to the conditional signal assignment and again offers a higher level of abstraction compared to the concurrent signal assignment. The value to be assigned to a signal is determined by the value of a select expression, much like a case statement in a conventional programming language. The syntax for the selected signal assignment is given in Figure 5.5.

```
   WITH (some_signal) SELECT
    other_signal <= value1 WHEN (condition1) ,
                    value2 WHEN (condition2) ,
                    ...,
                    default_value WHEN OTHERS;
```

Figure 5.5: **VHDL** syntax for the selected signal assignment.

the **VHDL** description of a 4-to-1 multiplexer using the selected signal assignment is shown in Figure 5.6 and can be compared to the previous descriptions in Figures 5.2 and 5.3. The selected signal

```
-- 4-to-1 multiplexer using conditional signal assigment
library ieee;
use ieee.std_logic_1164.all;

entity mux4to1_version2 is
    port (  x1, x2, x3, x4 : in std_logic;      -- inputs
            s              : in std_logic_vector(1 downto 0); -- controls
            f              : out std_logic); -- output
    end entity;
architecture prototype of mux4to1_version2 is
begin
    with s select
        f <= x1 when "00" ,
             x2 when "10" ,
             x3 when "01" ,
             x4 when others;
end prototype;
```

Figure 5.6: **VHDL** description of a 4-to-1 MUX using a selected signal assignment.

assignment differs from the conditional signal assignment in that there is no priority and all choices are evaluated at once (and therefore only one choice can be true at any time). All possible choices need to be specified, but we can capture values that we are not interested in using the **others** keyword. The others keyword conveniently captures all the other possible values of the select signal —

we should always include the "when others;" in a selected signal assignment.

5.3 *Constants and signals*

In the explanation of the conditional and selected signal assignment statements, we saw the assignment of constants to signals and to vectors of signals. To assign a constant to a signal, or to a single bit within a vector of signals, we enclose the constant inside of single tick marks. For example,

```
f <= ’1’; -- f is std_logic
y(4) <= ’0’; -- y is std_logic_vector and this assigns y(4)
```

To assign a vector of signals all at once, we assign each bit, but enclose the bits inside of double quotation marks. For example,

```
   z <= "10011"; -- z is a 5-bit std_logic_vector
   z(3 downto 0) <= "1110"; -- z is std_logic_vector and this assigns lowest 4-bits.
```

6 Process statements

Recall that **VHDL** concurrent statements all operate in paralle. Occasionally, it is useful to be able to evaluate things sequentially. The **VHDL** *process statement* is effectively a "block" around a bunch of logic and control statements.

1. Inside of a process statement, statements are executed sequentially which allows us to introduce sequencing and control;

2. Inside a process statemetn, we can use additional **VHDL** syntax like *case* statements and *if-then-else* statements.

Each **VHDL** process statement operates in parallel with other process statements and in parallel with other concurrent **VHDL** statements. Note that operations with sequencing like *case* and *if-then-else* must be — and can only be — used inside of a process statement.

6.1 First example

Figure 6.1 implements a 2-to-1 multiplexer as an example of a **VHDL** process statement. The syntax of a process statement is given in Figure 6.2. The syntax is as follows: A process can be named; the naming of a process is optional. The statements inside of the *begin...end* are evaluated sequentially. A process has a *sensitivity list*. The execution of the statements inside of the process happens when the value of any signal in the sensitivity list changes. Therefore, the signals that should be listed in the sensitivity list are those that can cause outputs to change inside of the process. It's also useful to note that signals changed inside of a process statement are *not changed or updated* until the process has completed evaluating every statement inside of the *begin...end*.

It wasn't mentioned, but Figure 6.1 demonstrates the syntax for a simple *if...else...end if* statement. If we have more conditions to check, we might need to use and *if...elsif...else...end if* statement. An example of this sort of *if* statement is illustrated in Figure 6.3 which shows the implementation of a 4-to-1 multiplexer using a process statement.

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

entity multiplexer_2to1 is
    port (x0,x1 : in std_logic;  -- inputs.
          s     : in std_logic;  -- control line.
          f     : out std_logic  -- multiplexer output.
         );
end multiplexer_2to1;

architecture prototype of multiplexer_2to1 is
begin
    process (x0,x1,s) -- process has a sensitivity list.
    begin
        if (s = '0') then -- can use if-else inside of a process
            f <= x0;
        else
            f <= x1;
        end if;
    end process;
end prototype;
```

Figure 6.1: First example of a **VHDL** process statement.

```
    [process_name] : process sensitivity_list
                     begin
                        -- statements.
                     end process;
```

Figure 6.2: Syntax for a process statement.

Figure 6.3: Implementation of a 4-to-1 multiplexer to demonstrate the use of *if...elsif...else* inside of a process statement.

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

entity multiplexer_4to1 is
    port (x0,x1,x2,x3   : in std_logic;                      -- inputs.
          s             : in std_logic_vector(1 downto 0);  -- controls.
          f             : out std_logic);                    -- output.
    end multiplexer_4to1;

architecture prototype of multiplexer_4to1 is
begin
    process (x0,x1,x2,x3,s)
    begin
        if (s = "00") then
            f <= x0;
        elsif (s = "01") then   -- notice the syntax âĂIJelsifâĂİ without spaces
            f <= x1;
        elsif (s = "10") then
            f <= x2;
        else
            f <= x3;
        end if;
    end process;
end prototype;
```

.

6.2 *Case statements*

Another statement which can be used only inside of a process statement is the *case statement*. To illustrate, let's consider imlementing a circuit that receives as input the decimal digits $0 \cdots 9$ represented in 4 bits. We have 7 outputs that need to be set differently depending on the inputs as shown in Figure 6.4. The **VHDL** description

inputs				**decimal**	**outputs**						
c_0	c_1	c_2	c_3	**value**	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	1	0	0	0	0	1	1	0
0	0	1	0	2	1	1	0	1	1	0	1
0	0	1	1	3	0	0	0	1	1	1	1
0	1	0	0	4	0	0	1	0	1	1	1
0	1	0	1	5	1	0	1	1	0	1	1
0	1	1	0	6	1	1	1	1	0	1	1
0	1	1	1	7	0	0	0	1	1	1	0
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	0	0	1	1	1	1	1

Figure 6.4: Truth table used to illustrate **VHDL** for the case statement.

that implements the circuit description from Figure 6.4 is shown in Figure 6.5.

The syntax for the case statement is shown in Figure 6.6. Notice that the case statement has a "default" situation for the situation when no conditions match — the *others* keyword is used here. Finally, in the default statement, we use the **VHDL** keyword *null* which is equivalent to saying "value doesn't matter, so do nothing/whatever".

6.3 *Comments on process statements*

In addition to statements like *if...else... end if*, *if...elsif...else...end if* and *case* statements, we can use *concurrent signal assignment* (i.e., lhs <= rhs). However, the other concurrent signal assignments including *selected* and *conditional* signal assignment statements **are not allowed** inside of a *process* statement. This is to keep concurrent and sequential statements **separate** from each other.

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

entity bcd_decoder is
    port (code : in std_logic_vector(3 downto 0);   -- inputs
          leds : out std_logic_vector(6 downto 0)); -- outputs
    end bcd_decoder;

architecture prototype of bcd_decoder is
begin
    process (code)
    begin
        case code is
            when "0000" => leds <= "0111111" ;
            when "0001" => leds <= "0110000" ;
            when "0010" => leds <= "1011011" ;
            when "0011" => leds <= "1111000" ;
            when "0100" => leds <= "1110100" ;
            when "0101" => leds <= "1101101" ;
            when "0110" => leds <= "1101111" ;
            when "0111" => leds <= "0111000" ;
            when "1000" => leds <= "1111111" ;
            when "1001" => leds <= "1111100" ;
            when others => leds <= null ;
            end case;
end process;
end prototype;
```

Figure 6.5: **VHDL** description to illustrate the use of the *case* statement inside of a *process* statement.

```
case signal_name is
    when (condition1) => (signal_assignment1) ;
    when (condition2) => (signal_assignment2) ;
    ...
    when others => (default_assignment);
end case;
```

Figure 6.6: Syntax for the **VHDL** case statement.

7 Sequential circuits

To implement sequential circuits, we must be able to describe things like latches, flip flops and state machines in **VHDL**. This is most certainly possible now that we have the process statement.

7.1 Latches, flip flops and signal attributes

The description of a latch is very straightforward; the **VHDL** for a gated *D*-type latch is shown in Figure 7.1.

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

entity dlatch is
    port ( d, g      : in std_logic;    -- data and gate inputs.
           q, qbar   : out std_logic    -- q and qbar outputs.
         );
end dlatch;
architecture prototype of dlatch is
begin
    process (d,g)
    begin
        if (g='1') then
            q    <= d;
            qbar <= not d;
        end if;
    end process;
end prototype;
```

Figure 7.1: **VHDL** for a gated *D*-type latch.

To define flip flops is slightly more complicated since we need the concept of **signal attributes**.

7.1.1 *Signal attributes*

When the value of a signal has **changed** we say that an **event** has occurred on the signal. Signals have properties that we can "pay attention" to and these properties are called **attributes**. An **event** is a type of **attribute**.

The syntax for observing an event on a signal is as follows:

```
signal_name'attribute_name
```

Notice the signal quotation (tick mark) between the signal name and the attribute name.

Consider that we have a **VHDL** signal called **temp** which is type **stf_logic** and we want to know if the signal has *just changed from a 0 to 1*. In this case, the value of the signal is 1 and an **event** has occurred on the signal. This is detected using the **VHDL** code *temp'event and temp='1')*. Once again, this statement detects when a change in the value of **temp** to a value of 1. Similarly, we can detect when the signal makes a transition from from *1 to 0* using the statement *temp'event and temp='0')*.

Using the **event** attribute, we can write the **VHDL** description for flip flops. As an example, the **VHDL** description of a positive edge triggered **DFF** is shown in Figure 7.4.

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

entity dff is
    port ( d, clk    : in std_logic; -- data and clock inputs.
           q         : out std_logic -- data output.
         );
end dff;

architecture prototype of dff is
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            -- q becomes d on rising edge otherwise does not change.
            q <= d;
        end if;
    end process;
end prototype;
```

Figure 7.2: **VHDL** description of a positive edge triggered **DFF**. Notice the use of the **event** attribute to detect the rising edge of the clock. We could change this to a falling edge triggered flip flop easily.

Note the power of **VHDL** — if we had multiple flip flops connected as a register, we only need to change the signals **d** and **q** in Figure 7.4 from **std_logic** to **std_logic_vector**. For example, Figure **??** shows the **VHDL** description for a 16 bit register consisting of 16 flip flops.

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

entity register16 is
    port ( d    : in  std_logic_vector(15 downto 0) ; -- data inputs.
           clk  : in  std_logic; -- clock input.
           q    : out std_logic_vector(15 downto 0) -- data outputs.
         );
end register16;

architecture prototype of register16 is
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            -- q becomes d on rising edge otherwise does not change.
            q <= d;
        end if;
    end process;
end prototype;
```

Figure 7.3: **VHDL** description of a positive edge triggered **DFF**. Notice the use of the **event** attribute to detect the rising edge of the clock. We could change this to a falling edge triggered flip flop easily.

Additional control signals can be added to describe a more complicated flip flop. For example, the **VHDL** description in Figure **??** is for a positive edge triggered D-type flip flop with asynchronous set and reset. Note that there is a precedence due to the use of the *if...elsif...else* statement. That is, the set signal has higher priority than the reset signal. Both set and reset have higher priority that the clock signal.

7.2 *State machines*

In **VHDL** it is very easy to describe state machines is a descriptive way and to let the **VHDL** compiler decide on how to best implement the state machine. We will consider implementing **Moore machines** in **VHDL**. **Moore machines** can be described in **VHDL** using *two* process statements:

1. The first process statement is coded to determine the next state of the state machine based on the current state and the current circuit

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

entity dffsr is
    port ( d         : in  std_logic; -- data input.
           s, r, clk : in  std_logic; -- set, reset and clock input.
           q         : out std_logic  -- data output.
         );
end dffsr;

architecture prototype of dffsr is
begin
    process (clk, s, r)
    begin
        if (s = ‘1’ ) then
            q <= ‘1’;
        elsif (r = ‘1’) then
            q <= ‘0’;
        elsif (clk’event and clk=‘1’) then
            q <= d;
        end if;
    end process;
end prototype;
```

Figure 7.4: **VHDL** description of a positive edge triggered **DFF** with asynchronous set and reset.

inputs;

2. The second process is coded to use the clock and its event attribute to update the current state with the next state in the active clock edge.

Finally, another interesting part is the introduction and use of an *enumerated type* to maintain the possible states of our state machine. We can implement an *enumerated type* in **VHDL**. The syntax to declare a new signal type in **VHDL** is

```
type type_name is ( value1, value2, ..., value_n );
```

Once defines, we can declare signals to be of this type.

To illustrate the implementation of a Moore machine in **VHDL** we will consider an example. Consider the following verbal description of a problem as shown in Figure 7.5. The state diagram

A circuit has one input X and one output Z. A "1" is asserted at its output Z when the circuit recognizes the following input bit sequence "1011". The circuit does not reset once the pattern is found, but continues to recognize the string. For example if the input is "...1011011...âĂİ, then the output will be high twice Z="...0001001...".

Figure 7.5: Verbal problem description to illustrate the implementation of a Moore machine in **VHDL**.

which corresponds to this verbal problem description is shown in Figure 7.6. We can see that this state diagram has one input x,

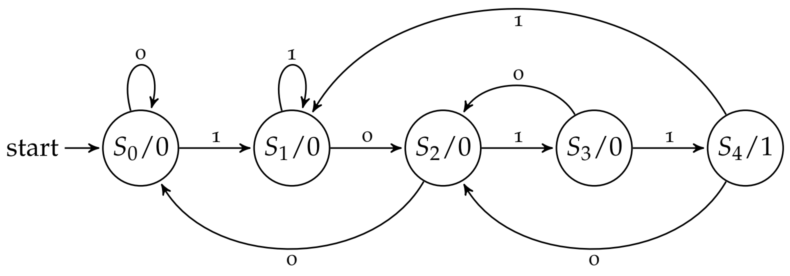

Figure 7.6: State diagram for a circuit to implement the verbal problem description in Figure 7.5.

one output z and requires 5 states. Obviously, a reset and clock signal are also required. The **VHDL** description is shown in Figure 7.7. Hopefully, the **VHDL** description is self-explanatory.

Figure 7.7: **VHDL** description for the state diagram in Figure 7.6.

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

entity moore_machine is
        port (x, reset, clk : in std_logic;
              z             : out std_logic);
end moore_machine;

architecture prototype of moore_machine is
    type moore_state is (moore_s0, moore_s1, moore_s2, moore_s3, moore_s4);
    signal curr_state, next_state : moore_state;
begin
    -- process statement to determine the next state.
    process (curr_state, x)
    begin
        case curr_state is
        when moore_s0 =>
            if (x = ‘1’) then
                next_state <= moore_s1;
            else
                next_state <= moore_s0;
            end if;
        when moore_s1 =>
            if (x = ‘1’) then
                next_state <= moore_s1;
            else
                next_state <= moore_s2;
            end if;
        when moore_s2 =>
            if (x = ‘1’) then
                next_state <= moore_s3;
            else
                next_state <= moore_s0;
            end if;
        when moore_s3 =>
            if (x = ‘1’) then
                next_state <= moore_s4;
            else
                next_state <= moore_s2;
            end if;
        when moore_s4 =>
            if (x = ‘1’) then
                next_state <= moore_s1;
            else
                next_state <= moore_s2;
            end if;
        end case;
    end process;

    -- process statement to update the state.
    process (clk, reset)
    begin
        if (reset = ‘1’) then
            curr_state <= moore_s0;
        elsif (clk = ‘1’ and clk’event) then
            curr_state <= next_state;
        end if;
    end process;

    -- simple concurrent assignment to determine outputs.
    z <= ’1’ when (curr_state = moore_s4) else ’0’;
end prototype;
```

8 *Conclusion*

Hopefully this document provides enough if an introduction to **VHDL** through examples to get you going. You can always find additional examples and explanations on the internet.