TR 1999- 1 0.4pt 0.4pt

Technical Report 1999-03P(1999/9/2)

VHDL for ELE548

Heather Hinton[1]

Electrical and Computer Engineering,
Ryerson Polytechnic University,
350 Victoria Street,
Toronto, Ontario,
Canada, M5B 2K3

[1] **hhinton@ee.ryerson.ca**

ECE Technical Reports

This technical report series allows faculty of the Department of Electrical and Computer Engineering to publish detailed and recent research results in a timely manner. It is *not* intended that these technical reports duplicate outside publications. However, due to the time lag in publishing results in formal, peer reviewed venues, many of these technical reports will be submitted for review and publication elsewhere. In such cases, it is intended that the technical reports will contain additional details and results that cannot be included elsewhere due to space limitations.

In addition to technical reports pertaining to research conducted within the Department, the technical report series may also be used to publish "pedagogical" results and methods. Ryerson has a strong tradition and committment to high-quality teaching and teaching methods. Many of our faculty are actively engaged in developing new pedagogical techniques, including the use of multi-media and Web-based tools for instructional purposes. We believe that it is equally important to make these results available to the academic and education community.

While all reports will be numbered sequentially, a research report will be identified by the technical report number and the code **R**. Likewise, a pedagogical report will be identified by the technical report number followed by the code **P**.

For more information about this technical report series, please contact Heather Hinton `hhinton@ee.ryerson.ca` or Andrew Kennings `akenning@ee.ryerson.ca`.

Publication History

This manual is an introductory tutorial in VHDL for third year students taking ELE548, Computer Architecture at Ryerson Polytechnic University. Initially, the manual contains only the introductory VHDL tutorial. This manual consists of an introduction to VHDL configured for the course (ELE548), followed by a series of exercises that are to be completed by the student. These exercises are intended to introduce all the concepts required to complete the ELE548 project. The examples used in this document are based the examples discussed in Kevin Skahill's book, "VHDL for Programmable Logic" [1].

Contents

List of Figures

1 Introduction

VHDL is an acronym for Very High-Speed-Integrated-Circuit Description Language, which pretty much describes what VHDL actually is. It is a language, just as C and Java are languages. VHDL is used to describe, model, and synthesize (make) a circuit, just as C is used to describe, model and implement a solution to a problem. So, don't be surprised when we refer to your VHDL solutions as "code"!

Like C, VHDL supports libraries (design libraries that contain common or reusable components, such as **and** 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 *synthesized* the circuit or you can *simulate* the circuit. Simulation is usually done before synthesis. In ELE548 we will focus on simulation. Why? By simulating a VHDL design of a circuit, we can "run" the VHDL code and determine if their are flaws that will prevent the actual realization of the circuit from working. 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 only on simulation, however. In software, we can design a circuit that cannot be easily realized (synthesize) in hardware. Indeed we can (inadvertently, of course) design a circuit that is physically meaningless! For this reason, we will (try very hard to have the equipment to allow you to) synthesize and implement a VHDL design at the end of this course.

1.1 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 (p. 8, [1]):

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. Programme the device

[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.

In this course we will stick to steps 1, 2, and 3. That will be plenty for our purposes, especially because steps 2 and 3 are often iterative (again, like C, there will be debugging, but no hacking).

Note: This tutorial should provide you with the background information you need to complete the ELE548 project. There are several other good sources of VHDL information that you may wish to investigate, including

- The Max+plusII VHDL Help (Help Menu, VHDL Help) has lots of information, including a syntax definition, information on templates and how to use the VHDL compiler within Max+plusII.
- Keven Skahill's book, *VHDL for Programming Logic*, [1], is also very good, but at a bit higher level than required for this course. Nevertheless, a good reference and may be handy for later courses. This is Prof. Hinton's favourite VHDL reference.
- The book by Charles Roth, [2], *Digital Systems Design Using VHDL* is another good reference book that happens to be Prof. Kennings' favourite. This book is more "textbook-like" then the Skahill book.

2 A Simple Circuit

To make our VHDL design as modular as possible (so that we can take advantage of the benefits of hierarchical design), VHDL forces us to keep the circuit interface and internals separate. Consider a black box, shown in Figure 1, that takes as input two four-bit vectors and produces a four-bit vector output. We know nothing about how the input or the output are related and what the functionality of this black box actually accomplishes. But, we do know a great deal about the interface to this black box. If we think about this black box as a C-function, what we know from this figure is the equivalent of the function declaration, called the **entity declaration** in VHDL.

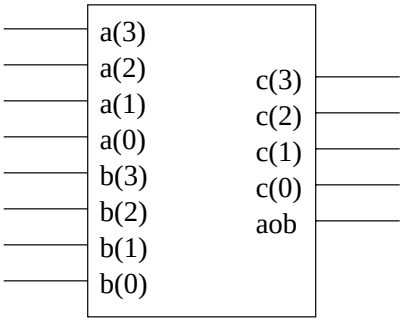

Figure 1: Black Box Representation of Circuit

2.1 The Entity Declaration

In VHDL, we describe the interface to this black box/entity using an **entity declaration**, as shown in Figure 2. Like a function declaration in C, an entity declaration describes the inputs and outputs to an entity (the black box). This entity has two inputs, the four-bit vectors **a** and **b**, and two outputs, a four-bit vector **c** and a single bit output **aob**. The entity has the name **bbox**, and the definition of the entity is *bounded* by the **entity bbox** and **end bbox** statements.

```
entity bbox is port(
     a,b : in std_logic_vector(3 downto 0);
     c   : out std_logic_vector(3 downto 0);
     aob : out std_logic);
end bbox;
```

Figure 2: Code: Entity Declaration for Black Box Circuit

The words **entity, is, port, end, in, out, std_logic, std_logic_vector**, and **downto** are *reserved words* in VHDL, meaning that the VHDL compiler knows what these words mean and you cannot use them as variable names.

What is a **port**? It is an I/O signal within an entity declaration. In the code shown above, there are four ports. Each port is declared with a **signal name** (such as **a** or **aob**), a **mode** (or direction, such as **in** or **out**), and a **data type** (such as **std_logic_vector**).

The code above shows two types of mode in the entity declaration, **in** and **out**. These can be thought of as "single-use" or "unidirectional" modes: the source of an in-mode signal is external to the entity, and the destination of an out-mode signal is external to the entity.

An additional, useful, mode is the **inout** mode. This mode is used to declare a signal that acts as both an input and an output signal (think of feedback).

All we know about the inside of the black box from the mode description is which signals are used as inputs and which signals are used as outputs. The data types of a port declaration tells us how to treat the signal on the port. Just as in C, declaring a variable as an integer or a floating point tells us how to interpret the variable, declaring a port with a data type tells us how to interpret the data on the port. So, **std_logic_vector(3 downto 0)** describes a four-bit vector (bits 0, 1, 2, 3) where the order of significance is from bit(3) downto bit(0). **std_logic** without the vector qualifier describes a single bit.

The **std_logic** type is an IEEE standard, provided by the IEEE std_logic_1164 package. In order to use this library, we must include it (remember including library files in C?). We include the library and the packages used *before* the entity declaration, as shown in Figure 3.

```
library ieee;
use ieee.std_logic_1164.all;
entity entity_name is port(
   -- stuff missing here
end entity_name;
```

Figure 3: Code: Declaring and Using Library Packages

The words **library** and **use** are also reserved words in VHDL. Also, the **ieee** library is "built-in" to VHDL (remember how C knows about system include files?).

2.2 Architecture Body

What about the internal workings of the entity? We know what the entity interface looks like, thanks to the entity declaration. We must now define the **architecture body**, the internal working, or behaviour, of the entity. We can chose between one of three "ways" to describe the architecture body: behavioural, dataflow, or structural descriptions. The main difference between these approaches is the level of detail required (or, conversely, the level of abstraction allowed).

2.2.1 Behavioural Description

Consider the architecture body shown in Figure 4. This behaviour description is quite reminiscent of a C-language programme in many ways. Behavioural descriptions are high-level, just as C is a high-level language. The architectural description is bounded by the **architecture** and **end arch_bbox** statements. When declaring the **architecture arch_bbox**, we define which entity the architecture belongs to (**of** bbox **is**). The **process** statement is used to enclose an algorithm.

The process in Figure 4 is named **comp** and the **sensitivity list** of comp is declared as (a,b). The sensitivity list identifies the signals that will cause the process to execute. In this

```
architecture arch_bbox of bbox is
begin
  comp: process (a,b) begin
    c <= b;
    if a = b then
        aob <= '1';
     else
       aob <= '0';
     end if
  end process comp;
end arch_bbox;
```

Figure 4: Code: Behavioural Description of Black Box Circuit

```
architecture arch_bbox of bbox is
begin
  comp: process (a,b) begin
    c <= b;
    aob <= '0';
    if a = b then
        aob <= '1';
    end if
  end process comp;
end arch_bbox;
```

Figure 5: Code: Alternate Behavioural Description of Black Box Circuit

case, the circuit is sensitive to changes in the two input signals, **a** and **b**. This means that whenever **a** or **b** changes, the **comp** process changes.

Note that the assignment statement **aob <= '1'** indicates that the variable **aob** is assigned the (bit) '1'. To see how this reads, try reading this statement from right to left, instead of the usual left to right. You can then pronounce this statement as '1' is assigned to **aob**.

Why is this description called behavioural? Because it is fairly easy to read the behaviour from the description: this VHDL listing describes a 4-bit equality comparison function. Because this behavioural description is given by an algorithm, we may suspect, that like a high-level programme written in C, this is not the only possible description. So, the VHDL code shown in Figure 5 is equivalent to the code shown in Figure 4.

2.2.2 DataFlow Description

A dataflow description is very similar to a behavioural description. In fact, the two are often both referred to as behavioural. The main difference is that a dataflow description does not use the **process** construct. Clearly the dataflow description is easy to understand for a simple example, such as the one we are looking at. With a more complicated algorithm

```
architecture dataflow of bbox is
begin
   aob <= '1' when (a=b) else '0';
end dataflow;
```

Figure 6: Code Fragment: DataFlow Description of Black Box Circuit

is required, such as one with nested sequential statements, a behavioural description will probably make more sense.

The big difference between behavioural and dataflow can be seen when we consider a circuit where the inputs may change at any time, but where we only want these (possibly changed) inputs to be noticed when a clock pulse triggers the circuit. We can easily describe this using a behavioural description where **process(clk)** identifies the clock signal as causing the circuit to activate [2]. With a dataflow description, we cannot as easily or neatly control "when" the circuit activates.

2.2.3 Structural Descriptions

A structural description consists of VHDL **netlists**, lists of signals and how they are "joined" by components, such as **and** or the hierarchically created **xnor** (a combination of **nor** and **not**).

```
use work.gatespkg.all;
architecture struct of bbox is
    signal tmp : std_logic_vector(0 to 3);
begin
   u0: xnor2 port map (a(0),b(0),tmp(0));
   u1: xnor2 port map (a(1),b(1),tmp(1));
   u2: xnor2 port map (a(2),b(2),tmp(2));
   u3: xnor2 port map (a(3),b(2),tmp(3));
   u4: and4  port map (tmp(0),tmp(1),tmp(2),tmp(3),aob);
end struct;
```

Figure 7: Code: Structure Description of Black Box Circuit

Take a look at the code shown in Figure 7. This is the structural equivalent of the behavioural and dataflow descriptions already discussed. The netlists in this description relate the inputs **a(1)** and **b(1)** with the output **tmp(1)** using the component **xnor2** [3]. Netlists are not as easy to read or understand as the behavioural or dataflow descriptions we have already seen. For this reason, we will focus on behavioural and/or dataflow descriptions in this course.

[2] See the section on clocked circuits for more details.

[3] The components **xnor2** and **and4** must have been defined elsewhere, and compiled into the library **work.gatespkg.all**. This allows us to create our **bbox** component hierarchically, building on already defined components. If we compile **bbox** and include it in a library, it to can be used to hierarchically create more complex components. We will see how to do this in one of the exercises.

2.3 Signals

So far, we have focused on input and output signals defined in an entity declaration. These signals define the interface to the circuit that we are designing. These signals may have the "directions" (or modes) of **in**, **out**, or **inout** [4]. Other useful signals include internal signals (discussed in next section), signal aliases (following next section) and clock and reset signals (both synchronous and asynchronous), discussed in Section 4.1.

```
library work;
use work.gatespkg.all;
entity bbox is port(
   a, b   : in  std_logic_vector(3 downto 0);
   axnorb : out std_logic);
end bbox;

architecture struct of bbox is
    signal tmp : std_logic_vector(0 to 3);
    signal out_and4 : std_logic;
begin
   -- instantiate components
   u0: xnor2 port map (a(0),b(0),tmp(0));
   u1: xnor2 port map (a(1),b(1),tmp(1));
   u2: xnor2 port map (a(2),b(2),tmp(2));
   u3: xnor2 port map (a(3),b(2),tmp(3));
   u4: and4  port map (tmp(0),tmp(1),tmp(2),tmp(3),out_and4);
   -- extract output signal
   axnorb <= int_out_and4;
end struct;
```

Figure 8: Code: Internal Signals

2.3.1 Internal Signals

In section 2.2.3, we saw a signal defined in the architecture body that did not have a **mode** (signal **tmp**). This signal is an *internal* signal, meaning that it is not part of the interface defined in the entity declaration. Internal signals are very useful as they provide a means of "gluing" components together. In Figure 7, the internal signal **tmp** is used to glue the outputs of the **xnor** components to the input of the 4-input **and4** component.

In fact, we can also use internal signals to define the output of the **and4** component. If we do this, however, we must find a way to *extract* this internal signal, and map it to an interface signal (defined as mode **out** in the entity declaration). This is actually quite easy, and is shown in Figure 8.

[4] Additionally, these signals could be defined as **buffer**, although we will not discuss this mode in this course.

2.3.2 Signal Aliases

Another useful thing that we can do is create a signal **alias**. Aliases are useful for allowing us to rename a signal, perhaps into a more meaningful signal name, for the scope of a description. The declaration of an alias is shown in Figure 9.

```
signal input_vector: std_logic_vector(15 downto 0);
alias  op_vector: std_logic_vector(7 downto 0) is input_vector(15 downto 8);
alias  op1 : std_logic_vector(3 downto 0) is input_vector(7 downto 4);
alias  op2 : std_logic is input_vector(3);
```

Figure 9: Code Fragment: Declaring Aliases to a Signal

An alias is an "alternative identifier" for an existing object. A change to an alias is equivalent to a change to the original signal. For example, assigning a value to **op2** in Figure 9 has the same affect as assigning that value to **input_vector(3)**.

Aliases are really useful if you have a vector signal, where individual bits within the vector have distinct meanings. Using an alias, we can create an identifier to refer to these bits individually.

2.4 Variables

How do we store local values in VHDL? A signal doesn't really allow us to do this. Instead, we resort to **variables**, declared as follows:

```
variable var_name : var_type := var_initial_value;
```

Variables must be declared in a process, and are local to that process (recall that signals, on the other hand, are defined *outside* of a process).

```
architecture arch_bbox of bbox is
begin
  comp: process (a,b)
     variable inc_amt: integer := 2
  begin
     -- code that includes use of integer variable inc_amt
  end process comp;
end arch_bbox;
```

Figure 10: Code: Alternate Behavioural Description of Black Box Circuit

Useful variable types that you may require in this course include **bit**, **boolean**, and **integer**

If you need a variable with a constant value, you can declare a **constant**:

```
constant constant_name : const_type := const_initial_value;
```

The constant types are the same as the variable types. Like a variable, a constant must be declared within a process and is local to that process.

3 Combinational Logic

Combinational logic can be written with both *concurrent* and *sequential* statements. Concurrent statements may be executed in parallel (concurrently) and are found in dataflow and structural descriptions of a circuit. Sequential statements must be executed in a given sequential order and are used in behavioural descriptions (hint: what is the big difference between behavioural and dataflow descriptions?)

3.1 Concurrent Statements

Concurrent statements fall outside of the process statement (and hence fit nicely with dataflow descriptions).

3.1.1 Boolean Statements

The most "obvious" of concurrent statements are boolean statements. As an example, suppose we wish to build a circuit that will produce the logical-and and logical-or of two bits. We can accomplish this using boolean statements, as shown in Figure 11.

```
library ieee;
use ieee.std_logic_1164.all;
entity cct1 is port(
   a,b      : in std_logic;
   land,lor : out std_logic);
end cct1;
architecture archcct1 of cct1 is
begin
   land <= a and b;
   lor  <= a or b;
end archcct1;
```

Figure 11: Code: Concurrent Boolean Statement Circuit

The output of this circuit is the two signals, **land** and **lor**, produced concurrently (simultaneously).
The boolean statements that are available in the **ieee 1164** library are:

and, **or**, **nand**, **not**, **xor**, **xnor**

These data types can be used with bit and Boolean variables (**std_logic**) and with one-dimensional arrays of bits and Boolean variables (such as **std_logic_vector(3 downto 0)**), where both variables have the same length.

If you have an equation with multiple boolean operations, you must use parenthesis to force VHDL into order of operations (otherwise you will get a compile-time error).

3.1.2 With-Select-When Statements

There may be cases where a signal value is assigned based on the value of another signal (a *selection signal*). In this case, the **with-select-when** statements come in handy.

For example, consider a circuit where the output, **z**, will be assigned the value of signals **a** or **b**, depending on the value of a selection signal, **s**. We can represent this in VHDL as shown in Figure 12.

```
ARCHITECTURE cct OF testcct IS
BEGIN
with s select
  z <= a when '0',
       b when '1';
END cct;
```

Figure 12: Code Fragment: With-Select-When Statements

Careful inspection of this code fragment should convince you that it is remarkably similar to the circuit that you designed in the "Introduction to Max+plusII" tutorial. In fact, we have used the with-select-when statement to implement a single-bit multiplexor. **Note:** What happens if you put this code fragment into a proper VHDL structure and try to compile it? To see why this happens, read the next section... This example is easily expanded to create higher-order multiplexors.

3.1.3 With-Select-When-When Others

Because of how **std_logic** is defined, a single bit does not necessarily have only two values (unless it is explicitly **Boolean**). Other possible values include high impedance, unspecified, low impedance, and so on. For this reason, we have the choice of specifying the case **when others** as a catch-all for all other, not already specified, values of the selection signal. This idea is similar to the use of **default** in a C-language **case** statement.

Figure 13 shows the **when-others** "equivalent" of Figure 12.

```
ARCHITECTURE cct OF testcct IS
BEGIN
with s select
  z <= a when '0',
       b when '1',
       '0' when others;
```

Figure 13: Code: With-Select-When-When Others Statements

The code in Figure 13 states that for *any* value of **s** other than "**0**", the signal **z** will have the same value as the signal **b**.

3.1.4 When-Else Statements

The **when-else** statements are a version of the **when-select** statements where assignment is based on a condition that may or may not revolve around a single signal. The condition evaluated in this type of statement may be based on a single signal, like the **when-else** statements, or on multiple signals, or multiple conditions involving different signals. For this reason, there is an order of preference within a **when-else** statement; once a successful, or true, condition is encountered, the assignment specified by the **when-else** statement is executed and the entire clause is "exited".

For example, the **when-else** equivalent of the code of Figures 12 and 13 is shown in Figure 14.

```
ARCHITECTURE cct OF testcct IS
BEGIN
  z <= a when (s='0') else
       b;
END cct;
```

Figure 14: Code Fragment: When-Else Statements

Suppose we only want **z** to be assigned the value of **a** or **b** based on the select signal **s** and some other condition. We can create compound conditional statements within a **when-else** statement. All that we have to do enclose the compound statement in a set of parentheses, to "create" a simple statement, as seen in Figure 15.

```
ARCHITECTURE cct OF testcct IS
BEGIN
  z <= a when (s='0' and ocond1=true) else
       b when (s='1' and ocond2=true) else
       z;
END cct;
```

Figure 15: Code Fragment: When-Else Statements

3.2 Sequential Statements

The combinational circuit(s) that we saw in the previous sections were fairly simple: they could be implemented using simple gate logic, and represented with simple combinational expressions.

In this section, we will look at sequential statements, those that are used within behavioural descriptions.

3.2.1 If-Then-Else Statements

The **if-then-else** statements have the same meaning in VHDL as they do in the C-language. By comparison with the combinational statements of the previous section, these statements

are the sequential equivalents of the **with-select-when** and **when-else** statements.

Figure 16 represents the **if-then-else** equivalent representation of the multiplexor-type functionality described in the previous section.

```
ARCHITECTURE cct OF testcct IS
BEGIN
   test: process(a,b,s)
   begin
       if s='0' then
           z <= a;
       elsif (s='1') then
           z <= b;
       end if;
   end process;
END cct;
```

Figure 16: Code Fragment: If-Then-Else Statements

3.2.2 Case-When Statements

A **case-when** statement is the sequential equivalent of a **with-select-when** statement. Figure 17 shows the **case-when** equivalent of the simple, single-bit multiplexor-type circuit of Figures 12 to 16. Note that this code must account for the other possible values of the signal **s**.

```
ARCHITECTURE cct OF testcct IS
BEGIN
 test: process(s)
   begin
     case s is
        when '0' => z <= a;
        when '1' => z <= b;
        when others => z <= b;
     end case;
   end process;
```

Figure 17: Code Fragment: Case-When Statements

3.2.3 Loop Statements

There are two types of loop statements that we can use in VHDL: a **for** loop and a **while** loop. They tend to be used when a set of repetitive operations need to be executed, usually a bit-wise operation on a bit-vector. Figure 18 shows a simple **for-loop** used to initialize an 8-bit vector.

```
ARCHITECTURE cct OF test_cct IS
   signal sum: std_logic_vector(7 downto 0);
BEGIN
 test: process(clk)
   begin
      for i in 7 downto 0 loop
         sum(i) <= '0';
      end loop;
      -- code requiring clk and sum
   end process;
 END cct;
```

Figure 18: Code: For-Loop Statements

Unlike a **for-loop**, which has a predefined number of iterations, controlled by the counter (**i** in Figure 18), a **while-loop** will execute as long as a "controlling" condition evaluates to true (just like in the C-programming language) [5]. The **while-loop** equivalent of Figure 18

```
ARCHITECTURE cct OF test_cct IS
   signal sum: std_logic_vector(7 downto 0);
BEGIN
 test: process(clk)
      variable i: integer := 0;
   begin
      while i < 7 loop
         sum(i) <= '0';
         i = i+1;
      end loop;
      ...
   end process;
```

Figure 19: Code: While-Loop Statements

is shown in Figure 19.

Note that the **while-loop** of Figure 19 requires that we declare and initialize the variable **i**. Again, we can see the similarities to the C-language, where **i** is local to the scope of a **for** loop and need not be declared, but is not "built-in" to a **while** loop and therefore must be declared and initialized.

[5] Note that this construct is *not* supported by Max+PlusII.

4 Synchronous Logic

What happens when we need to synchronize our actions, for example, when a circuit is clocked? It shouldn't be too surprising to realize that a dataflow description does not handle synchronous logic well: this is due in large part to the lack of a **process** statement in a dataflow description.

4.1 Clocked Circuits

A **process** statement easily handles synchronous logic, by allowing us to specify a clock signal as one of the triggers to the circuit. Figure 20 shows a circuit that is sensitive *only* to changes in the clock signal.

```
ARCHITECTURE cct OF testing IS
BEGIN
   process (clk)
      begin
         ...
   end process;
END cct;
```

Figure 20: Code Fragment: A Clock-Sensitive Circuit

Let's think about what a clock signal, or any waveform for that matter, looks like. There will be a rising edge and a falling edge. Do we want our circuit to be triggered by any change in the clock signal, or {positive,negative} edge-triggered? How do we represent edge-triggering in VHDL?

Edge-triggering requires *two* conditions to be true: 1) the clock signal must change, and 2) it must change in the positive (negative) direction, as required. Representing edge-triggering in VHDL therefore requires a compound statement, specifying each of these conditions.

To represent the change in a clock signal, we need some way to record or recognize that an event (corresponding to the change in clock value) has occurred. To do this, we consider the **event attribute** of the clock signal, given by **clk'event** [6].

Unfortunately, all that the the **clk'event** attribute tells us is that we have encountered an edge in the clock signal, and not whether it was a rising or falling edge. To further specify the type of edge encountered, we also specify the value of the clock *after the edge has "completed"*. Thus a value of **clk='1'** would indicate that a rising edge had just occurred, and a value of **clk='0'** would indicate that a falling edge had just occurred.

Figure 21 shows the clock-sensitive circuit of Figure 20 re-written to specify a negative-edge triggered circuit.

[6] See the subsection on Attributes at the end of this section.

```
ARCHITECTURE cct OF testing IS
BEGIN
   process (clk) begin
     if (clk'event and clk='0') then
        ...
     end if;
   end process;
END cct;
```

Figure 21: Code Fragment: A Clocked, Negative-Edge Triggered Circuit

4.2 Reset Signals

Resets are a generally very useful thing. A reset signal in a circuit can be used to restore initial conditions, such as resetting a counter to its initial count value. The problem with resets is that they are generally asynchronous signals. How do we work a reset into an synchronous circuit?

This is actually quite easy if we remember a couple of things about VHDL. The first is that we can specify in the process sensitivity a list of all signals that affect the circuit outputs. So, we can specify a sensitivity list that includes both the (synchronous) clock signal and the (asynchronous) reset signal.

The second thing that we must remember that in the specification of a sequential statement such as an **if-then-else** statement, there is an order of preference that is followed. The first conditions that are encountered are of higher precedence, even if subsequent conditions are also true. So, as long as the **reset** conditions are tested first, we should be okay.

Figure 22 shows how to incorporate an asynchronous reset signal into a synchronous circuit. The sensitivity list tells us that the circuit outputs are sensitive to changes in the **clk** signal and the **reset** signal. The first condition included in the **if** statement concerns the **reset** signal; if this (asynchronous) signal is set, then the reset actions will be executed. If the (asynchronous) **reset** signal is *not* set, and the **rising edge** of the clock has occurred (note that **rising_edge** == **(clk'event and clk='1')**) then the synchronous rising edge triggered actions occur.

```
ARCHITECTURE cct OF testing IS
BEGIN
   process (clk, reset) begin
     if (reset = '1') then
        -- do (asynch) reset actions
     elsif rising_edge(clk) then
        -- do (synch) rising edge clock triggered actions
     end if;
   end process;
END cct;
```

Figure 22: Code Fragment: Asynchronous Resets in Synchronous Circuits

Attributes

In VHDL, we often need to consider some **attribute** of a signal, such as recognizing when a signal changes. When a signal changes, an event is said to have occurred; we recognize this by looking at the **event attribute** of a signal [7].

An attribute is simply a (predefined) means of providing information about an item. An attribute is represented with a tick-mark **'** and the attribute-reserved word. The attribute that identifies that a signal has changed is given as **'event**. If we wish to specify that a given signal, for example, the clock, has changed, we can write:

clk'event

The value of **clk'event** is false, unless an event has just occurred (the clock has changed value) when **clk'event** will be true.

[7] We can also consider attributes of entities, architectures, etc. In this course, we will need only consider **event attributes** of signals.

5 Using Components

One thing that we would like to be able to do in VHDL is access components that have been defined in separate files (same as we want to include and use functions defined in different files when programming in C). To do this, we take advantage of the **component** structure in VHDL.

Suppose that we know that the file **add4.vhd** contains the VHDL code for a 4-bit adder, and we wish to use a 4-bit adder in our larger circuit. We can cut-and-paste the code from **add4.vhd** into our larger circuit, or we can treat this code/file as a component that we will use in our larger circuit.

To use your 4-bit adder, we must declare it as a **component**. This is how we make your compiled adder circuit available to other VHDL files. A component declaration is not unlike a function declaration in the C-programming language: all it does is declare the component interface. By identifying the component input and output signals in a component declaration, we allow a designer to determine if the interface matches the interface required/implemented by their larger circuit.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY addfour IS
    PORT(
       cin      : in  std_logic_vector(3 downto 0);
       ain,bin  : in  std_logic_vector(3 downto 0);
       sum      : out std_logic_vector(3 downto 0);
       carry    : inout std_logic_vector(3 downto 0);
       cout     : out std_logic);
END addfour;
ARCHITECTURE addfour OF addfour IS
   -- Instantiated Component Declarations
   COMPONENT add4
      PORT(
       ci       : in  std_logic_vector(3 downto 0);
       a,b      : in  std_logic_vector(3 downto 0);
       s        : out std_logic_vector(3 downto 0);
       c        : inout std_logic_vector(3 downto 0);
       co       : out std_logic);
    END COMPONENT;

    -- Internal Signal Declarations
BEGIN
   -- instantiate and connect components
   add_low: add4 port map (cin, ain, bin, sum, carry, cout);
END addfour
```

Figure 23: Code: Instantiating A Component

We access this adder circuit by *instantiating* it within our file (to *instantiate* a component is to make an instance of it, or to make a copy of it that we can use). When using instantiated components, we must fill in the entity declaration for our larger circuit normally (there is no indication in the entity declaration that we will be using components). The signals that we declare in the entity declaration must contain at least the same signals (with local names) as the signals required by the components that we are going to use!

The component declaration is bounded by the reserved words **component** and **end component**. The component that is declared **must** have the same name as the VHDL file describing that component (so component **add4** would be found in file **add4.vhd**). The component declaration must also exactly match the entity declaration of the component file (in this case **add4.vhd**).

Note that in the entity declaration of Figure 23 we have used signal names that differ from the names in the component declaration: this is not necessary. We could have chosen to use the same signal names as in **add4.vhd**. We are using different names in this example to illustrate how the mapping of signals from **add4** to **addfour** is accomplished.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY sample_cct IS
     PORT(
        -- bunch of signals declared here
END sample_cct;
ARCHITECTURE sample_cct OF sample_cct IS
   -- Component Instantiations
   COMPONENT component1
      PORT(
      -- component1's entity declaration
   END COMPONENT;
   COMPONENT component2
      PORT(
      -- component2's entity declaration
   END COMPONENT:
   COMPONENT componentN
      PORT(
      -- componentN's entity declaration
   END COMPONENT;

   -- Internal Signal Declarations
BEGIN
-- instantiate and connect components
END sample_cct;
```

Figure 24: Code: Instantiating Multiple Components

The actual instantiation of the adder component occurs in the architecture body (see

Figure 23), where we declare a local copy (**add_low**) of the component **add4**. We can declare multiple local copies of **add4**, as long as each has a distinct local name.

The local declaration of **add_low** also maps the local signals (defined in our entity declaration) to the corresponding component signals of **add4**, defined in the component declaration. The **port map** clause identifies how the signals of your desired circuit are to be interconnected to the input and output signals of the library black box component. In this case, the signals identified in the **port map** clause are mapped, in the order given, to the signals in the component declaration, again in the order given. So, in Figure 23, the local signal **ain** maps to the component signal **a**, **sum** maps to the component signal **s**, and so on.

You may instantiate multiple components in a VHDL file, as shown in Figure 24.

6 Making Your Own Library

In this section, we will walk through how to create your own library. We will use a sample 4-bit adder circuit as our "working" example component, and add this file to a library file **mylibrary** so that you can access this component from other VHDL files.

6.1 Component Declaration

If a circuit is to be made available to other files, it must be declared as a component, as described in the previous section. A generic component declaration is given as:

```
component component_name port (
       define component interface (cut and paste from entity declaration)
end component;
```

The difference between a component that is declared inside the file in which it will be used, and one that is declared in a library, is that the library components must be declared within a *package*.

6.2 Package Declaration

If we think about the types of components that we may wish to include in a library, we may have multiple types of similar components. For example, we may wish to be able to include 4-bit, 8-bit, 12-bit and 16-bit adder components.

We use **packages** to group together components with the same functionality but different parameters. Each component within a package will have its own component declaration (so there would be a component declaration for the 4-bit adder, 8-bit adder, and so on). This makes sense: remember that a component declaration defines the component's interface. A 4-bit adder will have a different interface than an 8-bit adder (because the width of the input and output signals differ) and will therefore require a different component declaration.

For now, we will declare a package with a single component (simply because we have only defined the single 4-bit adder component)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
package package_name is
     component component_name port (
         define component interface (cut and paste from entity declaration)
     end component;
end package_name;
```

Note that we have included the **ieee** library as part of our package declaration. Any libraries required by the package components *must* be identified before the package is declared [8].

[8] This will have profound implications when you attempt to include multiple packages in a user-defined library. You must re-declare any and all libraries that are required by the package components before each package declaration. If you fail to do this, you will experience some wild and wonderful error messages.

6.3 Libraries

You have already used built-in libraries supplied by Max+PlusII when you invoked the boolean gates in the **ieee** library. Now we are going to create our own libraries, containing packages and components that we have compiled (and successfully simulated).

A library is a VHDL file that contains the package and component declarations that we will be accessing from another VHDL files. A library must be compiled, just as a regular VHDL file must be compiled. What makes a library different, however, is that it does not contain the component's architecture body. That is, all we need to include in the library is the component declarations (which look surprisingly like the component's entity declaration). Max+PlusII will look for the entity that matches the component name by looking for the VHDL file with the same name (remember that VHDL file names and entity names must match). This highlights one difference between a library file and a "regular" VHDL file: a library file may contain many components (entities) and therefore cannot match an entity name.

6.4 Building Your Library

Your library will contain (for now) an adder package, with the 4-bit adder component of the previous sections. Open a new file in the text editor and add the following VHDL code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
package adder_pkg is
     component add4 port (
         signals cut and paste from your entity declaration
     end component;
end adder_pkg;
```

Save this file as **mylibrary.vhd**. Set the project name to the current file (Cntrl-Shift-J) and then save and compile your library (Cntrl-L). You should get the following message:

```
Info: Compiling package "ADDER_PKG"
Info: File ``...\mylibrary.vhd" does not contain an Architecture Body
      - stopping compilation
```

This is okay. However, there is an important thing to note with this: any changes that you now make to your add4.vhd file, in particular the entity declaration, *must* be also made in your library file, and both files must be re-compiled.

Believe it or not, you have now created a library file. If you want to use it, you must declare it in the library statement of your VHDL file, and you must identify which packages in the library you wish to use (for now, and probably for always, you will use **all** packages).

6.5 Instantiating a Library Component in a VHDL File

The other big difference between user-defined libraries and the built-in VHDL/Max+PlusII libraries is in how you access the components. Before we look at a hierarchical interconnection of library components, let's look at how to instantiate a library component.

Figure 25 shows how to instantiate the **add4** component from your file **mylibrary** into a new file, **addfour.vhd**. Remember that the **add4** component is defined in the file **add4.vhd**.

```
-- examples using libraries and packages
LIBRARY ieee, mylibrary;
USE ieee.std_logic_1164.all;
USE mylibrary.adder_pkg.all;    -- your library file
ENTITY addfour IS
    PORT(
       cin       :-- fill in yourself;
       ain,bin   :-- fill in yourself;
       sum       :-- fill in yourself;
       carry     :-- fill in yourself;
       cout      :-- fill in yourself);
END addfour;
ARCHITECTURE addfour OF addfour IS
BEGIN
-- instantiate and connect components
adder: add4 port map (cin, ain, bin, sum,  carry, cout);
END addfour;
```

Figure 25: Code: User-Defined Libraries and Components

The first thing to note about the code in Figure 25 is that there are *two* libraries declared, the **ieee** library and your library, **mylibrary**. The next thing to notice is that that the usage of the **adder_pkg** in **mylibrary** is declared in the **USE** statement.

The entity declaration of your component is a "normal" entity declaration. In this entity declaration, the signals used by the component that you are building are declared; these signals are "local" to this file.

To instantiate the **add4** library component, we first give it a "local" name, in this case, **adder**. This creates a distinct copy of the library component **add4** in our local file.

We must then *map* the signals from the local component **adder** to the library component **add4**. Because we are treating the library component as a black box, we are only concerned with the interface signals, or the input and output ports of the library component. The **port map** defines the mapping of local signals onto the library component port signals. The listing of local signals shown in Figure 25 *must* be in the same order that the signals are declared in the library component's entity declaration. Why?

Having done all this, we have now successfully instantiated the library component **add4** in the library package **mylibrary.adder_pkg** and mapped the signals from our local component to the equivalent signals in the library component.

6.6 Compiling Your Library into a VHDL File

The only thing left to do is to figure out how to actually access your library as a library from within another VHDL file. First, you must declare your library using the **LIBRARY** clause (see below). Then you must identify which packages you wish to access (in this case, the **adder_pkg**), and which components within that package you wish to access (will almost always be **all**).

```
-- examples using libraries and packages
LIBRARY ieee, mylibrary;
USE ieee.std_logic_1164.all;
USE mylibrary.adder_pkg.all;
```

Before you compile your VHDL file, you must tell the compiler about the location of your **mylibrary** file:

1. Save your VHDL file, and bring up the **Compiler** menu (pull down from the Max+PlusII menu).
2. Pull down the **Interfaces** menu
3. Open the **Netlist Reader Settings** menu
4. Fill in the library name in the library box
5. Fill in the directory name containing the library, either by typing it in or highlight and double-click on the correct directory in the directory structure window.
6. Click **Add** to add your library to the existing (user-defined) libraries.
7. Click **OK**. You're done.

7 Using Max+PlusII for VHDL

1. Start up Max+PlusII. Create a new project (File / Project)
2. Start up the Max+PlusII text editor (Max+PlusII / Text Editor)
3. Insert the VHDL templates for **Entity Declaration** and **Architecture Body** (Templates / VHDL Templates)
4. Save the file **with the .vhd extension** (File / Save As). Explicitly enter **filename.vhd** Do not let the automatic extension of **.tdf** be used as this will tell Max+PlusII that you are entering an AHDL file, which you are not doing.
5. Fill in the blanks in the templates. Recall that __variable is not a legitimate variable name in VHDL, and so you must replace everything that is given in the template as _variable with a legitimate variable name. Re-save your file frequently.
6. Once you have entered your complete VHDL specification, you are ready to compile the code. Select the compiler (Max+PlusII / Compiler). Select the VHDL NetList reader sebfings (Interfaces / VHDL NetList). Set for 1993 VHDL. At this point, you can also define any user-defined libraries and packages.
7. Double check that your VHDL file has a **.vhd** extension and not a **.tdf** extension. The next step will fail miserably if you do not have the **.vhd** extension.
8. Start the compiler. Fix any errors and re-compile. Once you compile without errors, go to the next step. (Make sure that your entity name is the same as your filename.)
9. Go back to the compiler (Max+PlusII / Compiler). Select the Functional SNF Extractor (Processing / Functional SNF Extractor) and start.
10. Open SCF and set up the inputs for the waveform (see the Max+PlusII tutorial). Save the SCF file.
11. Go to the simulator (Max+PlusII / Simulator) and start simulating.
12. Verify the simulated waveform.

8 VHDL Syntax Primer

8.1 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 in ELE548.

alias	all	and	architecture	begin
body	case	component	constant	downto
else	elsif	entity	exit	for
if	in	inout	is	library
loop	map	nand	nor	not
null	of	on	or	others
out	package	port	process	signal
then	to	ttype	until	use
variable	wait	when	while	with
xnor	xor			

The boolean functions **and, nand, not, or, xnor, xor** are found in the **ieee** library, in the **std_logic_1164** package.

8.2 Declarations

8.2.1 Entity Declaration

entity entity_name **is**
    port (
        interface_signal_declarations
    );
    end entity_name;

8.2.2 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.

8.2.3 Library Declarations

library list_of_library_names;

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.

8.2.4 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 [pacakge_name];

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

8.2.5 Component Declarations

component component_name
    port (
        signal interface_signals : **mode** signal_type;
    );
end compnent [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:

internal_component_label: component_name
    port map (list_of_local_signals);

8.2.6 Signal Declarations

signal list_of_signal_names : type_name [:= initial_value];

interface_signal_declarations look like

signal list_of_signal_names: **mode** signal_type;

8.2.7 Constant Declarations

constant constant_name : type_name := constant_value;

8.2.8 Alias Declarations

alias identifier **is** item_name;

8.2.9 Variable Declarations

variable var_name : type_name [:= initial_value];

8.2.10 Integer Type Declarations

int_type type_name **is range** integer_range;

8.3 Simple Assignment Statements

8.3.1 Signal Assignment

signal <= expression

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

8.3.2 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.

8.4 Concurrent Statements

8.4.1 when-else

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

8.4.2 with-select-when

with selection_expresssion **select**
signal <= expression1 **when** condition1;
expression1 **when** condition2;
...
[expressionN **when others**];

8.5 Sequential Statements

8.5.1 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.

8.5.2 if-then-else

if condition **then**
sequential_statements
{**elsif** condition **then**
sequential_statements }
[**else** sequential_statements]
endif;

8.5.3 case-when

case expression **is**
when choice1 => sequential_statements
when choice2 => sequential_statements
...
[**when others** => sequential_statements]
end case;

8.5.4 for-loop

for identifier **in** range **loop**
sequential_statements
end loop

8.5.5 while-loop

while boolean_condition **loop**
sequential_statements
end loop

8.5.6 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];
```

8.6 Modes

8.6.1 in

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

8.6.2 out

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

8.6.3 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).

8.6.4 buffer

A buffer signal is an output signal, where the signal's values are also required internal to an entity.
This mode is not required in ELE548.

9 Exercise: A 4-bit Adder

Objective As the first part of this tutorial, you are going to implement a simple combinational circuit, in the form of a 4-bit carry-propagate adder, one bit of which is shown in Figure 26. You will implement this circuit using the Max+PlusII graphical editor and you will code this adder in VHDL. You will then simulate both in Max+PlusII and verify your implementations using a waveform analysis.

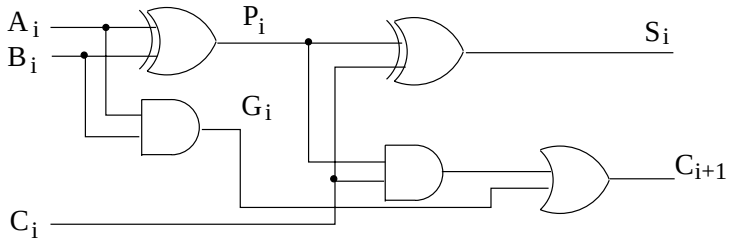

Figure 26: 1-Bit Carry-Propagate Adder

The inputs to the adder will be two 4-bit vectors and a carry-in bit. Assume that the initial carry-in bit will always be '0' (there is no carry-in). The output of the adder will be a single 4-bit vector and a carry-out bit.

9.1 A Schematic-Entry 4-Bit Adder

Design Requirements

Implement, using the graphical editor and the built-in Max+PlusII symbols, a schematic design for the 4-bit adder circuit based on the single-bit shown in Figure 26. Compile and simulate this design to be sure that it works :)
To Be Handed In: A schematic design of this circuit, built using the Max+PlusII Graphic Editor (see "A Quick Introduction to Altera Max+PlusII" handout) and a simple waveform analysis that demonstrates that your circuit is correct.

Hint: You can make this a lot easier on yourself by setting one input to a set of count values and setting the second input to a group value. If you cleverly chose the starting and group values, you can watch the carry-out bit being set in a relatively small time frame.

9.2 A VHDL 4-Bit Adder

Design Requirements

You are to design, implement, and simulate in VHDL, the 4-bit adder described above. You are to implement this adder from first principles (you must build the 4-bit equivalent circuit of Figure 26) with "basic" level components **only**: **and**'s, **or**'s, **nor**'s, **nand**'s, and **not**'s are allowed. You may employ either behavioural or dataflow descriptions.

Implementing the VHDL Code

The entity declaration of the 4-bit adder is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY add4 is
   PORT(
       ci        : IN   STD_LOGIC;
       a,b       : IN   STD_LOGIC_VECTOR(3 downto 0);
       s         : OUT  STD_LOGIC_VECTOR(3 downto 0);
       c         : INOUT STD_LOGIC_VECTOR(3 downto 0);
       co        : OUT   STD_LOGIC);
END add4;
ARCHITECTURE add4 OF add4 IS
   signal p,g     : std_logic_vector(3 downto 0);
BEGIN
  process( ?  )
  begin
       -- you fill in what goes here
  end process;
END;
```

To Be Handed In: A VHDL-source-code listing of your 4-bit adder. A printout from Max+PlusII is acceptable. You must also answer the following questions:

1. Why is signal `c` declared as `INOUT`?
2. Why are signals `p` and `g` declared in the architecture body and not the entity declaration?

Simulation

In Max+PlusII, compile and simulate your code. Verify that your adder is correct using Waveform simulation.
To Be Handed In: A waveform listing of your 4-bit adder as generated by Max+PlusII. On your waveform listing, show that the adder is correctly implemented. Remember the hint from the previous exercise.

10 Exercise: A 16-bit Adder

To illustrate one way that hierarchical design actually works, we are now going to build a 16-bit adder, from "scratch" and by interconnecting four of the 4-bit adders from the previous section.

10.1 Brute Force Schematic Entry 16-bit Adder

Design Requirements

Go back to the schematic design you generated of the 4-bit adder. Generate the equivalent schematic diagram of the 16-bit adder (meditate on the joys of cut and paste) and simulate your circuit.
To Be Handed In: The schematic diagram illustrating your complete 16-bit (brute force) adder and the corresponding waveform analysis proving that it is correct.

10.2 Brute Force VHDL 16-bit Adder

Design Requirements

Design the circuit 16-bit version of the circuit shown in Figure 26. This circuit is to be implemented with "basic" level components *only*: **and's, or's, nor's, nand's** and **not's** are allowed. Meditate once again on the joys of cut-and-paste.

Implementing the VHDL Code

Implement the corresponding the VHDL code for the design of the previous section. You may employ either behavioural or dataflow descriptions.

The entity declaration of the 16-bit adder is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY add4 is
   PORT(
       ci   : IN     STD_LOGIC;
       a,b  : IN     STD_LOGIC_VECTOR(15 downto 0);
       s    : OUT    STD_LOGIC_VECTOR(15 downto 0);
       c    : INOUT  STD_LOGIC_VECTOR(15 downto 0);
       co   : OUT    STD_LOGIC);
 END add4;
 ARCHITECTURE add16 OF add16 IS
 BEGIN
        -- you fill in what goes here
 END;
```

To Be Handed In: A VHDL-source-code listing of your 16-bit adder. A printout from Max+PlusII is acceptable.

Simulation

In Max+PlusII, compile and simulate your code. Verify that your adder is correct using Waveform simulation.
To Be Handed In: A waveform listing of your 16-bit adder as generated by Max+PlusII. On your waveform listing, show that the adder is correctly implemented.

10.3 16-bit Adder Using Components

Neither the brute force schematic nor cut and paste implementations of a large circuit work well if we already have "parts" of that circuit. Instead, hierarchical components, based on existing components, is a much better approach. You have already used this approach with Max+PlusII provided components, when you used the library symbols (for the schematic entry designs) and the boolean operations (such as **and** in the VHDL entry designs).

In this exercise, you will instantiate, as a component, the 4-bit adder of previous exercises, and the interconnect the instantiated components to create a 16-bit adder.

Design Requirements

On paper, represent the 4-bit adder of the previous sections as a black box. Identify the inputs and outputs to this box. Show how to use this black box component to implement a 16-bit adder (draw the required number of black box components and interconnect them, showing which outputs of which box are used as inputs to which other box). Be careful with the identification of the interconnects, and make sure that they are clearly labelled: this will be very useful for the implementation section...

Implementing the VHDL Code

You must declare the 4-bit adder component and any required interconnect signals. You must then instantiate the 4-bit adder some number of times, and interconnect these instantiated components to create a 16-bit adder.
To Be Handed In: A VHDL-source-code listing of your 16-bit adder and your component file(s). A printout from Max+PlusII is acceptable.

Simulating the Circuit

In Max+PlusII, compile and simulate your code. Verify that your adder is correct.
To Be Handed In: A waveform listing of your 16-bit adder as generated by MaxplusII. On your waveform listing, show that the adder is correctly implemented.

10.4 16-bit Adder Using User-Defined Library Components

In this exercise, you will put the 4-bit adder of the previous exercises into a package, compile it into a library and then hierarchically create a 16-bit adder.

Design Requirements

Same as for previous exercise.

Implementing the VHDL Code

You must declare the 4-bit adder as a **component**, within a **package**, and compile the **package declaration** into a library. See the "Making a Library" and "Using Components" sections of this tutorial for more details.
To Be Handed In: A VHDL-source-code listing of your 16-bit adder and your library file. A printout from Max+PlusII is acceptable.
HINT: Be careful about the exact implementation of the 4-bit adder that you chose to include in your library and use in this design. Now that your are hierarchically interconnecting components, an output from one component may also have to role of an input to another component. This will affect how you define your component/entity signals!

Simulating the Circuit

In Max+PlusII, compile and simulate your code. Verify that your adder is correct.
To Be Handed In: A waveform listing of your 16-bit adder as generated by MaxplusII. On your waveform listing, show that the adder is correctly implemented.

11 Exercise: D Flip-Flops

11.1 A VHDL Single-Bit D Flip Flop with Asynchronous Reset

Design Requirements

You are to design, implement, and simulate in VHDL a positive-edge-triggered D flip flop with an asynchronous reset. You may use either behavioural or dataflow descriptions (hint: one of these will be much more elegant than the other).

Implementing the VHDL Code

The entity declaration of the D-FF is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dfflop1 IS
   PORT(
      -- you fill in the blanks!
END dfflop1;

ARCHITECTURE dfflop_1bit OF dfflop1 IS
BEGIN
    -- you fill in the blanks!
END dfflop_1bit;
```

To Be Handed In: A VHDL source-code listing of your single-bit D flip flop.

Simulation

Using Max+plusII, compile and simulate your code. Verify that you adder is correct using Waveform simulation.
To Be Handed In: A waveform simulation of your single-bit D flip flop. Show that your DFF is correctly implemented.

11.2 A VHDL 4-bit D Flip Flop with Asynchronous Reset

In this exercise, we will extend the single bit D flip flop of the previous exercise to implement a positive-edge-triggered D flip flop capable of handing an 4-bit input vector (and therefore an 4-bit output vector). Question: Does this implement a 4-bit latch or a 4-bit register?

Design Requirements

You are to design, implement, and simulate in VHDL a positive-edge-triggered 4-bit D flip flop with an asynchronous reset. You may use either behavioural or dataflow descriptions.

Implementing the VHDL Code

The entity declaration of the 4-bit D-FF is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dfflop4 IS
   PORT(
      -- you fill in the blanks!
END dfflop4;

ARCHITECTURE dfflop_4bit OF dfflop4 IS
BEGIN
    -- you fill in the blanks!
END dfflop_4bit;
```

To Be Handed In: A VHDL source-code listing of your 4-bit D flip flop.

Simulation

Using Max+plusII, compile and simulate your code. Verify that you adder is correct using Waveform simulation.

To Be Handed In: A waveform simulation of your 4-bit D flip flop. Show that your circuit is correctly implemented.

12 Exercise: Multiplexors

12.1 A VHDL 2-bit Multiplexor

In the "Introduction to Max+plusII" tutorial, you used schematic entry (via the graphical editor) to implement a single-bit multiplexor. In this exercise, you are to implement, in VHDL (via the text editor) a 2-bit multiplexor. The inputs to this multiplexor are the selection signal, `s`, and two 4-bit vectors, `a` and `b`. Depending on the value of the selection signal, the output vector `x` will contain either the logical and, or logical or of vectors `a` and `b`, or the logical inverse of vectors `a` or `b`, as shown in the table below:

Select Signal	Operation Performed
00	x ← a **and** b
01	x ← **not**(a)
10	x ← a **or** b
11	x ← **not**(b)

Design Requirements

You are to design, implement, and simulate in VHDL a 2-bit multiplexor. You may use either behavioural or dataflow descriptions.

Implementing the VHDL Code

The entity declaration of the 2-bit multiplexor is:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux IS
   PORT(
      -- you fill in the blanks!
END mux;

ARCHITECTURE mux2 OF mux IS
BEGIN
    -- you fill in the blanks!
END mux;
```

To Be Handed In: A VHDL source-code listing of your 2-bit multiplexor.

Simulation

Using Max+plusII, compile and simulate your code. Verify that you multiplexor is correct using Waveform simulation.
To Be Handed In: A waveform simulation of your 2-bit multiplexor. Show that your mux is correctly implemented.

13 Exercise: Using Components

13.1 An Adder-D Flip-Flop Circuit

In this exercise we are going to use components to build a cicuit from the 4-bit adder and 4-bit positive-edge-triggered flip flop circuits of the previous exercises. The composite circuit will take two 4-bit input vectors, add them together, and then feed the output into a positive-edge-triggered 4-bit D flip flop. The flip flop will have an asynchronous reset, which will reset the output of the flip flop to zero.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY addDff IS
    PORT(
       cin       :-- fill in yourself;
       ain,bin   :-- fill in yourself;
       q         :-- fill in yourself);
END addDff;
ARCHITECTURE arch_addDff OF addDff  IS
   -- Instantiated Component Declarations
   COMPONENT add4
      PORT(
         -- fill in to match your component's entity declaration
    END COMPONENT;
    COMPONENT dfflop4
      PORT(
         -- fill in to match your component's entity declaration
    END COMPONENT;
    -- Internal Signal Declarations
        -- include any internal signals required
BEGIN
   -- instantiate and connect components

END arch_addDff;
```

Figure 27: Code: Composite Adder-D Flip Flop Circuit

Design Requirements

You are to design, implement and simulate in VHDL this composite adder-D flip flop circuit. You may use either behavioural or dataflow descriptions.
To Be Handed In: A VHDL soruce-code listing of your composite circuit.

Implementing the VHDL Code

The template for the code for this composite circuit is shown in Figure 27.

Simulation

Using Max+plusII, compile and simulate your code. Verify that your composite circuit works as intended using Waveform simulation.
To Be Handed In: A waveform simulation of your circuit. Show that the circuit is correctly implemented.

13.2 Adder-D Flip Flop Circuit Using Library Components

In the previous exercise we declared and instantiated components within our circuit. In this exercise, we will create a library file with the required component instantiations and then use the library file to reference the adder and D flip flop components.

Design Requirements

Implement an adder package and a D flip flop package with the corresponding components. Compile this file as a library file.

```
LIBRARY ieee, mylibrary;
USE ieee.std_logic_1164.all;
USE                          -- identify adder package
USE                          -- identify flip flop package
ENTITY addDff IS
    PORT(
       cin       :-- fill in yourself;
       ain,bin   :-- fill in yourself;
       q         :-- fill in yourself);
END addDff;
ARCHITECTURE arch_addDff OF addDff  IS
   -- Internal Signal Declarations
        -- include any internal signals required
BEGIN
   -- instantiate and connect components

END arch_addDff;
```

Figure 28: Code: Adder-D Flip Flop Circuit Using Library Components

Implementing the VHDL Code

The template of this VHDL code for this question is given in Figure 28.
To Be Handed In: A VHDL-source-code listing of your composite circuit and your library file.

Simulation

Using Max+plusII, compile and simulate your code. Verify that your composite circuit is correct using Waveform simulation.
To Be Handed In: A waveform simulation of your composite circuit. Show that the circuit is correctly implemented.

References

[1] Kevin Skahill. *VHDL for Programming Logic.* Addison-Wesley, 1996.

[2] Jr Charles H. Roth. *Digital Systems Design Using VHDL.* ITP Nelson, 1997.