VHDL: A Language for Specifying Logic

Table of Contents

1 INTRODUCTION

VHDL stands for VHSIC Hardware Description Language. VHSIC in turn stands for Very High Speed Integrated Circuit. This is essentially a language for specifying the operation of logic systems. VHDL was developed by the United States Government and later adopted by the IEEE as IEEE standard IEEE1164 and IEEE1076. There are two VHDL standards, one set out in 1987 and one set out in 1993. The later standard extended the first standard. Further changes were made in 2000, 2002 and 2009.

VHDL is one of several logic specification languages. The other hugely accepted language in industry is Verilog HDL. VHDL and Verilog each have strengths and weaknesses. Verilog is probably the dominant language used by the IC industry in the design and implementation of VHSIC, especially in the USA. VHDL has found favour in Europe, and is widely used in the implementation of complex digital signal processing and computing systems.

Logic specification is useful for a number of reasons. The most obvious is for programming CPLD's (complex programmable logic devices) and FPGA's (Field Programmable Gate Arrays). These days it is possible to manufacture certain types of devices based purely on VHDL specification. This offers advantages in terms of speed of modification, design verification and ease of design.

The very programmability of FPGAs makes them ideal for applications produced in relatively small quantity (Radio Astronomy, Radar, segments of the Telecommunications industry).FPGAs currently operate at fairly low clock speeds (compared to an IC). However, they are massively parallel, and it is thus possible to implement computer algorithms (for example evaluating share options in investment banking, gene folding, and so on) in parallel, with consequent speedup compared to a serial implementation on a conventional processor. Another bonus is much reduced power consumption: power consumption goes up to the square of clock speed. Commercial boards and software development suites now constitute a small but growing industry around high performance computing.

FPGAs are heavily utilised in high speed, digital, signal processing. Again, the parallelism and programmability are key factors. The jury is still out, however, on the best software tools utilised to design, build and test these systems.

Currently some microprocessor manufacturers use high level logic specification to design entire microprocessors. These devices are known as synthesized processors. Most FPGA vendors now offer a selection of “soft cores” which are complete microprocessor cores that have been programmed using a hardware description language. These cores are then programmed onto a large FPGA, along with other subsystems to form a “System On a Programmable Chip” (SOPC). Ideally the SOPC concept will allow electronic system designers to use a single large FPGA to implement an entire complex system.

As an example think of a miniature MP3 player. The system will have a microprocessor, to control the system. It will have an MP3 CODEC (CODEC=coder/decoder). It will need to have a Flash memory interface, to speak to a flashcard, such as SD or CompactFlash. Then the system will probably also have a USB interface to communicate with a computer. All of these functions require logic functions. In the past this would have been implemented using separate chips for low production volumes or an ASIC (Application Specific Integrated Circuit) for high production volumes.

Using separate chips is bulky and not cost effective. To some extent the microprocessor can incorporate multiple functions, but fast CODEC's can be disproportionately expensive to implement in this way. ASIC's require large capital investment to set them up and this effectively restricts their use to high volume applications, where the setup costs can be amortized over millions of units. With SOPC designers can develop applications based around FPGA's. This reduces setup costs drastically. If the sales volumes justify it then the manufacturers can migrate towards ASIC-like devices which are programmed according to HDL specifications.

Lastly, HDL's provide a means of specifying hardware used for test purposes. Imagine testing a counter. You will at least need to pulse the clock line repeatedly while watching the output code. The hardware that stimulates the clock line could be written in an HDL and linked to the counter in an entirely simulated test. In this case the hardware that provides the stimulus is called a “Testbench” and the counter itself is known as a DUT, Device Under Test.

VHDL will provide us with a means of specifying the operation of the complex digital systems that will be examined in the rest of this course. We will be able to describe these systems and the synthesize and simulate those systems. The course will culminate in the development of a very simple soft core which will illustrate many concepts in a practical way.

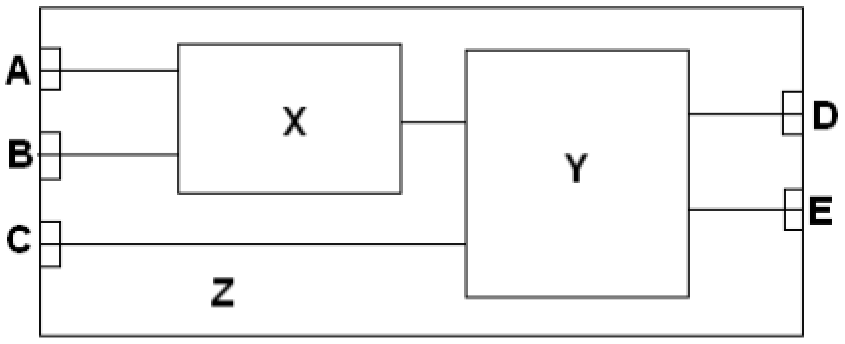
You will soon see that programming logic is somewhat different to the sequential or procedural systems that you used for programming computing devices.

2 Structure and Behavior

There are two fundamentally different ways of specifying logic. The first method is to specify the structure of the logic and the other method is to specify the behavior of the system.

2.1 Structure

The structure of a system describes that system in terms of "what is connected to what". The system is thus broken down into smaller units which are connected together to form a whole. The whole unit is called an *entity*. Inputs and outputs to/from the entity are called *ports*.



In this illustration the overall entity is called Z. The input ports are called A, B and C. The output ports are called D and E. The interconnections inside the entity Z are called *signals.* You will observe that X and Y are themselves entities. The entities X and Y would in turn have to be specified in terms of either their behavior or their structure. This nested definition system implies that by using structural descriptions very big and complex blocks may be built up out of a number of simple blocks. This is fundamental to problem solving of almost any type.

2.2 Behavior

You will see that structural descriptions are extremely useful for building systems from smaller components. Ultimately we need to specify what each component actually does. In this case structural descriptions are of limited use. We need some way of describing the behavior of the entities that we use. At the very minimum we need to specify the behavior of the entities at the bottom of the hierarchical structure. VHDL thus provides a means of specifying the behavior of entities by means of a *behavioral description.*

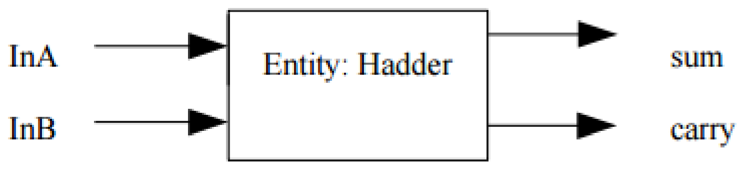
Behavioral descriptions resemble a programming language. We will discuss syntax as we need it to implement the concepts that follow. Please bear in mind as we go that, despite the similarities, a VHDL description is a hardware description, and that the analogy to a programming language is actually fairly weak.

2.3 ENTITIES

As we have mentioned an entity is a "box" with inputs and outputs called ports. We need to give the ports names and associate them with an entity. Here is an example of this:

```
entity hadder is
   port
      ( inA, inB: in std_logic;
      sum,carry: out std_logic);--note the semicolon
end hadder;
```

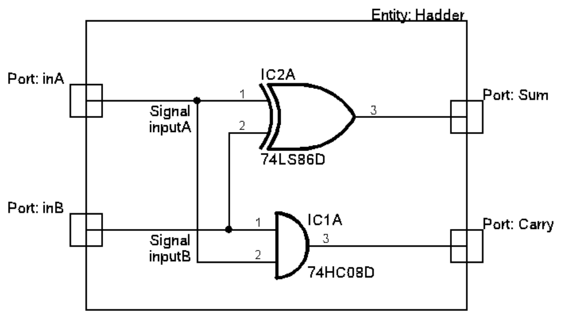
This entity declaration states that we are creating an entity called hadder. This entity has two inputs called inA and inB and two outputs called sum and carry. Further we have declared that the ports are all 'std_logic' values. A std_logic value is basically a binary bit, which can take on the value 0 and 1. Std_logic bits can also take on a few other states, such as high impedance and undefined, in order to closely model the behaviour of real logic bits. The entity is shown graphically here:



2.4 ARCHITECTURES

Once we have declared the entity we need to describe how it works. We use an *architecture* to do this. We could use either a structural description or a behavioral one.

Suppose for the sake of the illustration that the above entity implements a half adder. Our adder's circuit diagram is as follows:



Here is an example of structural description:

```
architecture structure of hadder is
--We now need to describe the interface to the components that we are
--using. These components are described in the ALTERA MAXPLUS2 library.
component a_7408    --AND gate
    port (a_2: in  STD_LOGIC;
          a_3: in  STD_LOGIC;
          a_1: out STD_LOGIC);
end component;

component a_7486    --XOR gate
    port (a_2: in  STD_LOGIC;
          a_3: in  STD_LOGIC;
          a_1: out STD_LOGIC);
end component;

--Signals are like wires used to connect components
signal inputA,inputB : std_logic;
begin
   --see equivalent diagram in notes to understand this.
   inputA<=inA;
   inputB<=inB;
   myAND: a_7408 port map (a_2=>inputA,a_3=>inputB,a_1=>carry);
   myXOR: a_7486 port map (a_2=>inputA,a_3=>inputB,a_1=>sum);
end structure;
```

The first thing that we do is state which components are contained in our box. These are an AND gate and an XOR gate. We also define the ports of these components. Note that the components were created previously and we are only using them here.

We then define two signals. These signals are the internal "wires" which connect components together. The signals are inputA and inputB.

After we have defined all of the types of things that go into our description we need to *instantiate* them. We do that in the next section of code. We make an instance of the AND gate called myAND and an instance of the XOR gate called myXOR. When we instantiate the components we list how the inputs are connected to the signals. This effectively creates a *netlist*. A netlist is a list of how components connect together.
The first two statements connect the internal signals inputA and inputB to the input ports, as shown in the circuit diagram.

Here is an example of a behavioral description:

```
architecture behave of hadder is
 begin
    carry  <= inA and inB;
    sum    <= inA xor inB;
 end behave;
```

You will immediately notice that this architecture seems to be at a higher level than the previous one. While we laboriously explained *how* the previous circuit would work we now explain *what* this circuit must do. Notice that we have not called on pre-defined components at all in this description. This is clearly essential at some level of a design.

In general, VHDL is very conducive to hierarchical design techniques and many of the VHDL tools available use this to good effect to produce structured, maintainable descriptions.

3 PROCESSES

We sometimes want a set of VHDL statements to execute sequentially, one after the other. This allows us to build up a set of instructions that are 'executed' one after the other. The execution of this list is triggered by some event happening, very often on a port of the entity.

As an example consider this simple flip flop design:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY test_proc1 IS
    PORT
    (D_input     : IN     STD_LOGIC;
     clk_input   : IN     STD_LOGIC;
     Q_output    : OUT    STD_LOGIC);
END test_proc1;
ARCHITECTURE a OF test_proc1 IS
BEGIN
    PROCESS (clk_input)   --trigger this when clk_input changes
    BEGIN
        if (clk_input='1') then
             Q_output <= D_input;
        end if;
    END PROCESS;
END a;
```

In this example we are describing the behavior of a D type flip flop. The flip flop's D input is transferred to the Q output and held there on the rising edge of the clock signal.

We declare an entity as before.

The architecture begins with the keyword 'Process' followed by a list of port names, in this case clk_input. This list is called a 'sensitivity list'. Whenever one (or more) of the items in the sensitivity list changes the code that is contained in the process is executed. In this example when clk_input changes, the code within the process is executed. This code examines the state of the clk_input bit and if it is high the Q output is updated to be equal to the D input. We achieve the rising edge triggered action by looking for a change in clk_input and then examining to see if the change was a rising or a falling edge.

4 VARIABLES

It is possible to have variables in an architecture. Variables are used to store data within a process. Signals differ from variables in that variables cannot trigger events, nor can a change in a variable trigger a process. Furthermore, signal values within a process change only when the process is completed, whereas variables are evaluated immediately.

Variables are modified with the variable assignment operator which is :=, thus assigning one variable to another is done as follows:

```
one_variable:=another_variable
```

5 VHDL SYNTAX

5.1 COMMENTS

Comments are preceded by "--" (two dashes) and extend to the end of the line. Multiline comments are not available, but some code editors provide a multiline comment feature.

5.2 IDENTIFIERS

Identifiers are names of signals, variables, components etc. etc. The names must all begin with a letter. After the initial letter they may have any combination of letters, numbers and underscores. **VHDL is not case sensitive. vhdl IS NOT CASE SENSITIVE.**

5.3 NUMBERS

VHDL has support for a very wide range of number representations.

Numbers may be in any base between 2 and 16. The hash (#) separator is used to separate the base from the rest of the number, by surrounding the number. e.g. 2#10101010# =10101010 base 2. Note that the base is an integer expressed in base 10.

Real literals may have decimal points. These numbers are real numbers with a fixed fractional part. The VHDL standard supports floating point numbers although some VHDL compilers do not. Floating point operations are extremely expensive to implement because of the inefficiency incurred in synthesizing the floating point operations. e.g. 3.1415 is a fixed point real number.

Remember that VHDL only looks like a programming language. Bear in mind that when you use a floating point number the required operation has to be constructed out of gates when the design is put onto a chip. Numbers may have exponents. The exponents are expressed in decimal and may only be integers. Exponents may be positive or negative. The exponent is signified by an 'E'. **The exponent is raised to the base.** For example, a number expressed in base 2 will be multiplied by 2 to the power of the exponent.

Example: 2#1010#E1=1010*2^1=10100=20 decimal.

Here is an example of VHDL using integer types:

```
entity Code_Example is
port (clock  : In   std_logic;
         sel  : In   std_logic;
       output: Out  integer range 0 to 255);
end Code_Example;

architecture archy of Code_Example is
begin
process (clock)
 begin
   if clock='1' then
       if sel='1' then
          output<=16#55#;
       else
          output<=2#1010_1010#;
       end if;
   end if;
end process;
end archy;
```

Integer types are used for representing numbers with no fractional part. VHDL guarantees that integers will be represented by at least 32 bits. Integers are signed. We can limit the range of integers by using the *range* keyword. This is useful for limiting the size of the representation and for making your code more compiler independent. In general limit your range to the size that you need. When we limit the range of a type we obtain a *Subtype*.

When limiting the range of an integer only the size of the representation is limited, not the actual range of the number. As an example if we said `dout: OUT integer range 0 to 2` in the example above we would still have had a 2 bit counter. The code that could be represented on those 2 bits would still include “11”, even though it is outside of our defined range. Some simulation packages would flag a warning if this occurs, but many would not.

When using an integer as a port, the port will have as many physical lines as are needed to represent the integer using standard binary coding. Thus, in the example above, 'dout' will be represented by two physical port lines.

5.4 Characters and Strings

Characters are enclosed in single quotation marks and are equivalent to their ASCII number. e.g. '0'=48

Strings are enclosed in double quotation marks. Example "Samuel" To include a double quotation mark inside quotation marks use it twice "Fred said""hello""

Strings may be formed in other number bases. These bases are binary octal and hex.

```
entity Code_Example is
port (clock : In  std_logic;
      sel   : In  Std_logic;
      output: Out std_logic_vector (7 downto 0));
end Code_Example;

architecture archy of Code_Example is
begin
process (clock)
 begin
  if clock='1' then
     if sel='1' then
       output<="01010101"; --binary string. Use a string to fill vector
     else
       output<=X"FF"; --Hexadecimal string
     end if;
   end if;
end process;
end archy;
```

5.5 ENUMERATED TYPES

VHDL provides enumerated types for convenience. We have already come across one of these, std_logic, in which the logic states are represented as an enumerated type having nine different states which represent different electrical possibilities available on logic signals.

```vhdl
entity Code_Example is
port (clock : In std_logic;
      output: Out integer range 0 to 1);
end Code_Example;

architecture archy of Code_Example is
type colour is (green, blue);
    --declare types in architecture declarative section

begin
process (clock)
 variable status: colour;
    --declare variables in process declarative section
 begin
   if clock='1' then
      if status =green then
        output<=1;
        status:=blue;
      else
        output<=0;
        status:=green;
      end if;
   end if;
end process;
end archy;
```

5.6 Arrays

These have the same meaning that you are used to. Arrays may have any number of dimensions and may be formed from any supported type, or previously defined type.

There are predefined types, such as strings and bit vectors.
The definition for the string type is given here:

```
type string is array (positive range <>) of character;
```

The definition for bit vectors is this:

```
type bit_vector is array (natural range <>) of bit;
```

The "<>" angle brackets are called a box. They are placeholders which indicate that the range will be filled in when the types are used.

Bit vectors are essentially an array of bits used for binary numbers. Here is an example of the usage of a bit vector:

```
din: in bit_vector (2 downto 0);
```

In this case the box will be filled in with the range 2 downto 0.

Arrays may be indexed to access individual elements. If we wish to access the first bit in din then we can say:

```
if din(2)='1' then....
```

Note that in the case of bit vectors when we specify the range we are specifying an array index. As a result, we are not specifying the numerical range, but rather the actual number of bits in the array.

If we use the “downto” keyword then the most significant bit will occupy the highest numbered array position. The opposite will apply if we use the “to” keyword.

We will not use bit vectors much, rather we will use 'standard logic vectors' which are more widely supported. The reason for this is that most of the libraries that are available for our VHDL system are defined in terms of standard logic vectors, rather than bit vectors. The behavior of std_logic_vectors is however very similar to the bit vector, except that, as noted above, std_logic has more possible states.

See above for examples of std_logic_vector usage.

6 VHDL OPERATORS

Like any language VHDL has a set of operators. These operators may be divided into three types: Logical, arithmetic and comparison.

6.1 LOGICAL OPERATORS

Logical operators are used to derive logical results from bits and bit_vectors. When used with bit_vectors the operators act in a bitwise manner. When used with bit_vectors, those vectors must be of the same length. The logical operators are as follows:

NOT

AND

OR

NAND

NOR

XOR

These should require no further explanation.

In addition there is a concatenation operator. This is the ampersand "&". This concatenates two bit vectors.

Example:

```
signal a: bit_vector (1 to 3);
signal b: bit_vector (1 to 5);
signal c: bit_vector (1 to 8);
c <= a & b;  --legal concatenation operation.
```

6.2 ARITHMETIC OPERATORS

These operators perform basic arithmetic operations on numbers. Normally these operations are only intended for use on integers. Some VHDL suites come with libraries which contain arithmetic functions for use with bit_vectors and std_logic vectors.

The operators are:

+ addition

- subtraction

* multiplication

/ division

The only caution that applies here is that some packages only support multiplication and division by integer powers of two. This in effect reduces multiplication and division to the status of arithmetic shift operations.

6.3 Comparison operators

These operators are used to compare two scalar or bit_vector arguments and they return 'true ' or 'false'. If bit_vectors are compared then the vectors must be of the same length.

The operators are:

<	smaller than
<=	smaller than or equal
>	greater than
>=	greater than or equal
=	equal
/=	not equal

7 ATTRIBUTES OF OBJECTS

Attributes are properties of VHDL types that can be very useful. We use attributes to extract further information about a VHDL object.

All attributes are accessed by using the apostrophe ' operator, sometimes pronounced “tick”. In theory attributes are classified by what sort of thing they return, but in authoritative sources (compare Ashenden 2008 to Perry 2002) they are dealt with in quite different ways. For purposes of this course we will be quite loose in our handling of attributes, aiming for utility rather than formal correctness.

7.1 ATTRIBUTES OF SIGNALS

These attributes either derive information about signals or define new signals based on other signals. We have seen one of these already, the 'event attribute. This attribute creates a signal that is true when its base signal has changed in the last cycle. Examples have been shown above.
There is also a 'last_value attribute which returns the value of a signal prior to its last transition.

There are many more of these attributes, however many of them are for simulation purposes only and cannot be directly synthesized to hardware. These would typically be used for generating testbenches.

7.2 ATTRIBUTES OF SCALAR TYPES

'pos – str'pos ("fred") gives the position of fred in the string str.
'val – Var'val(x) gives the value at position x of variable var.
'leftof – value in a variable to the left of a specified position.
'rightof – value in a variable to the right of a specified position.
'pred – value of a preceding position. Depends on range direction.
'succ – value of a succeeding position. Depends on range direction.

Note that not all of these attributes apply to all types. Most of the above are useful with enumerated types. If an array is declared with a range of indices from 10 to 30 then Array'left will return 10.

Example:

Suppose we have the following declarations:

```
lsb_output: OUT std_logic  --declare a bit-wide port
signal count_signal_name : STD_LOGIC_VECTOR (3 downto 0);
                      --declare a 4 bit wide signal
```

Then the following line extracts the least significant bit from the signal and assigns it to the port:

```
lsb_output <= count_signal_name (count_signal_name'right);
```

This is useful for making procedures which handle arrays independently of the array width.

8 More on Processes

Processes are fundamental to the specification of sequential circuitry. Here are some examples of the usage of processes.

This is the behavioral description of a 2 bit counter.

```
entity testvhdl is
 port (ain: IN std_logic;
       dout: OUT integer range 3 downto 0);
end testvhdl;

architecture behave of testvhdl is

begin
 fred: process (ain)
 variable output: integer range 3 downto 0;
 begin
   if ain='1' then output:=output+1;
     dout<=output;
   end if;
 end process;
end behave;
```

Note that the ain is the counter clock. Dout is the counter's output. When there is a positive edge on ain then the code contained in the 'process' will be executed and dout will advance by 1. We will look more carefully at processes later.

You will see that we use a variable here. This is because when we increment output we effectively need to read the value of output. This is illegal in the case of an output port. If we made the port an input/output type (inout) then we could do away with the variable, but this would be poor practice as it implies that the counter output would have an input mode, which is nonsensical.

We can have more than one item in our sensitivity list. As an example consider a flip flop with a reset input. The process must be sensitive to both the clock input and the reset input. Different action must be taken in either case. Here is an example which illustrates this:

(if you want to run this code you will need to remove the line numbers)

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;

3   ENTITY test_proc1 IS
4   PORT
5       (d_input   : IN  STD_LOGIC;
6        clk_input : IN  STD_LOGIC;
7        clr_input : IN  STD_LOGIC;
8        q_output  : OUT STD_LOGIC);
9    END test_proc1;

10  ARCHITECTURE a OF test_proc1 IS
11  BEGIN
12  PROCESS (clk_input, clr_input)
13  BEGIN
14      IF clr_input = '1' THEN
15          q_output <= '0';
16      ELSIF (clk_input'EVENT AND clk_input = '1') THEN
17          q_output <= d_input;
18      END IF;
19  END PROCESS;
20  END a;
```

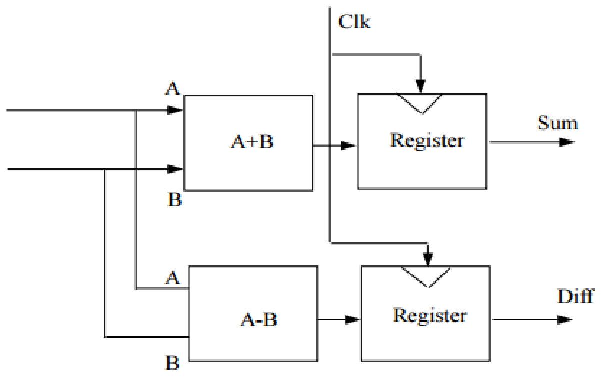
In line 12 we declare a process. This process is sensitive to clk_input as well as clr_input. When either of these ports changes state the process is launched. On line 14 the clr_input is examined. If the input is high then the output of the flip flop is cleared.

Line 16 is used to examine the clock line. If the process is launched it may be caused by the reset line, rather than the clock line. We want our flip flop to be edge triggered, not level triggered and so we must check the clock line to see if it caused the process. We do this by examining the attribute 'event of the clock line. If the clock line changed then we check to see if it was a positive edge. Only if there was a positive edge on the clock line do we transfer the D input to the Q output.

Notice over here that we have a synchronous data line and an asynchronous reset line in the same architecture.

It is possible to have more than one process in an architecture. Each process is triggered independently, based on the sensitivity lists, and each process will execute concurrently. This means that the processes will effectively execute in parallel. There is no time correlation between lines in different processes. Note that if different processes attempt to update common signals or ports then the result is undefined as it is not always possible to tell which process caused the last update.
Many compilers will issue a warning or error message about this.

As an example suppose you wish to implement the following hardware:



Effectively we have parallel hardware. We want the addition and subtraction to occur simultaneously if possible and we need to latch the results of these operations in the registers on the rising clock edge.
We can do this by using two processes, both triggered by the clk port pin. Here is the VHDL:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY test_proc1 IS
    PORT
    (clk   :  IN STD_LOGIC;
     inA   :  IN INTEGER range 0 to 255;
     inB   :  IN INTEGER range 0 to 255;
     sum   : OUT INTEGER range 0 to 255;
     diff  : OUT INTEGER range 0 to 255);
END test_proc1;
ARCHITECTURE a OF test_proc1 IS
    BEGIN
    PROCESS (clk)
    BEGIN
     if clk='1' then
       sum <= inA + inB;
     end if;
        END PROCESS;
    PROCESS (clk)
    BEGIN
     if clk='1' then
      diff <=inA - inB;
     end if;
        END PROCESS;
END a;
```

When we have multiple processes we can let each process have a different sensitivity list. As an example suppose we wish to have two extra inputs, one to clear the 'sum' register and another to clear the 'diff' register. We can implement this as follows:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY test_proc1 IS
    PORT
    (clk        :  IN STD_LOGIC;
     clr_sum    :  IN STD_LOGIC;
     clr_diff   :  IN STD_LOGIC;
     inA        :  IN INTEGER range 0 to 255;
     inB        :  IN INTEGER range 0 to 255;
     sum        : OUT INTEGER range 0 to 255;
     diff       : OUT INTEGER range 0 to 255);
END test_proc1;

ARCHITECTURE a OF test_proc1 IS
    BEGIN
    PROCESS (clk,clr_sum)
    BEGIN
       if clr_sum = '1' then
         sum <= 0;
       elsif clk'event and clk='1' then
         sum <= inA + inB;
       end if;
    END PROCESS;
    PROCESS (clk,clr_diff)
    BEGIN
       if clr_diff = '1' then
         diff <= 0;
       elsif clk'event and clk='1' then
         diff <=inA - inB;
       end if;
    END PROCESS;
END a;
```

Note that the clear inputs act asynchronously to the clock input. It can sometimes be tricky to arrange the clocking inputs so that the compiler can handle them. QUARTUS gives an error if the order of clocking is changed slightly in the above example.

It is possible to declare a “shared variable” in the architecture declarative section which is accessible to all of the processes in that architecture. This must be carefully thought out however because of the possibility of synchronization problems. To understand this problem consider the following example:

```
entity Code_Example is
port (clock1: In  std_logic;
      clock2: In  std_logic;
      output: Out std_logic);
end Code_Example;

architecture archy of Code_Example is
    shared variable sync:  std_logic;
    begin
    process (clock1)
    begin
        if clock1='1' then
        sync:='0';
        end if;
    end process;
    process (clock2)
    begin
        if clock2='1' then
        sync:='1';
        end if;
    end process;
    output<=sync;
end archy;
```

The compiler gives the following error:

Error (10028): Can't resolve multiple constant drivers for net "sync" at Code_Example.vhd(22)

The reason for the error is that it is quite possible for there to be simultaneous clock edges on clock1 and clock2. In this case what value should “sync” take on? There are two different values being sent at the same time to one variable which is going to produce undefined behaviour. The same problem must be avoided when working with signals shared between processes and output ports which are updated by multiple processes.

9 Changing Execution Flow in VHDL

In order for any sort of program to be truly versatile there must exist a means of changing the "program flow". You will have seen examples of this in any programming language that you may have used. VHDL offers similar facilities.

9.1 Variable assignment

Variables are needed to hold data temporarily. *Variables are distinctly different from ports and signals*. A value may be assigned to a variable by means of the variable assignment operator ":=". The type of the variable(s) and data must be the same.

9.2 If, Else and ElsIf Statements

The basic format of these statements is as follows:

```
if condition then
     series of statements
end if;
```

Or else you may want to use both if and else:

```
if condition then
     series of statements
else
     alternative series of statements
end if;
```

The elsif statement is a combination of both the if and else terms:

```
if condition1 then
     series of statements
elsif condition2 then
     alternative series of statements
end if;
```

Watch the spelling of the “elsif” statement.

Note that in all cases the condition must evaluate to a Boolean expression.

9.3 WHEN...ELSE STATEMENTS

This is a conditional assignment statement. This means that when a condition is met a signal assignment will occur. The else clause allows us to specify an alternative assignment for occasions when the first condition is not true.

The syntax is given here:

```
s<=signal1 when condition1 else signal2;
```

These constructs may be used in series to give multiple options:

```
s<=signal1 when condition1 else
   signal2 when condition2 else
   signal3;
```

Example of a When-Else statement:

```
Z<= A WHEN X > 5 ELSE
    B WHEN X < 5 ELSE
    C;
```

Think of the When..Else construct as a compressed version of the if..then..else statement.

9.4 CASE STATEMENTS

The case statement can be used select between a set of mutually exclusive cases based on the contents of a variable. This allows us to insert a single case statement in place of a series of if's and else's. (Note to C programmers: watch the syntax...). Case statements, like IF statements, can appear only inside a process.

The basic syntax is as follows:

```
case expression is
when alternative1=>
                     statements for alternative1
when alternative2=>
                     statements for alternative2
<other possible cases>
end case;
```

Example of a case statement:

```
PROCESS (my_vect)
BEGIN
    CASE my_vect IS
     WHEN "00" => my_int := 0;
     WHEN "01" => my_int := 1;
     WHEN "10" => my_int := 2;
     WHEN "11" => my_int := 3;
    END CASE;
END PROCESS;
```

We may use the or operator (|) within our alternative specification:

```
case expression is
when alternative1 | alternative2=>
                     statements for alternative1 or alternative2
when alternative3 | alternative4=>
                     statements for alternative3  or alternative4
<other possible cases>
end case;
```

The expression must evaluate to either a discrete type (e.g. std_logic or integer) or it may evaluate to a one dimensional array. (e.g. std_logic_vector or string).

Here is an example of a bit vector expression:

```
variable Fred: bit_vector (0 to 7);
...
...
case Fred is
  when "00100000"  => statements for when Fred equals 20hex
  when "0101_1010" => statements for when Fred equals 01011010
end case;
```

9.5 With..Select Statements

The with...select select structure is often an alternative to the case statement. The structure of the With Select construct is as follows:

```
with expression select
    s<=signal1 when condition1,
       signal2 when condition2,
       signal when others;
```

Example of a With-Select statement:

```
WITH command_signal SELECT
 DataOut <= signal_A and signal_B WHEN "00",
            signal_A or  signal_B WHEN "01",
            signal_C WHEN "10",
            '0' WHEN OTHERS;
```

When the expression equals the appropriate condition then the signal assignment occurs. If no condition is true then the "when others" will be activated and that assignment will occur.

You may use the or (|) operator within the condition specification to further enhance the selection process.

9.6 Loop Statements

The most basic loop statement is the infinite loop. This is implemented as follows:

```
loop
  a set of statements
end loop;
```

This is not always what we want. VHDL also supports the 'for' and 'while' looping constructs. Here are some examples:

```
while condition loop
    a set of statements
end loop;

for <some object> in <start of range> to <end of range> loop
   a set of statements
end loop;

for count in 1 to 10 loop
    Fred(count):=0;  --index an array
end loop;
```

Note that when using the for statement the object being used as the loop iterator is considered to be a constant within the loop body and it may not be assigned to. The iterator object does not exist outside of the loop.

There are two more loop modification keywords. The "next" keyword terminates the current iteration of the loop and proceeds to the next iteration.

The "exit" keyword terminates the entire loop immediately.

They are used as follows:

```
exit when condition1;
next when condition2;
```

We may also label our loops. This is useful for clarifying matters when we have nested loops. The loop label comes before the looping construct:

```
outer_loop : loop
   inner_loop : loop
     do_something;
        next outer_loop when temp = 0;
    do_something_else;
   end loop inner_loop;
end loop outer_loop;
```

Please remember when using loops that the code which you write is ultimately going to be converted to hardware, made of physical gates.

9.7 THE NULL STATEMENT

This statement does nothing. It is most commonly used to denote situations in which nothing must happen but where we wish/need to list an option. This is often done for clarity.

10 Dividing Big Jobs into little Jobs

Anyone who has worked on a big system will tell you that the way to approach such a system is to divide it into smaller modules. Programming languages all (?) provide some means of breaking tasks down. VHDL is no exception to this.

10.1 Procedures and Functions

Procedures are subroutines that do not have any return value. They are simply sequences of instructions that are to be executed in order. The syntax for declaring a procedure is as follows:

```
Procedure name parameter list;
```

Here you can see that the procedure has a name and a list of parameters (including their types) that must be fed into the procedure. Sometimes procedures do not have input parameters. Here is one:

```
procedure noparams;
```

Here is an example of a procedure that takes in two parameters. The first parameter is called var1. This parameter is of type 'inout' which means that it may be written to and read from. We declare the parameter to be a variable. This means that we can modify the value of var1 within the procedure. Strictly speaking the use of 'inout' makes the use of 'variable' redundant, although we may wish to include it for clarity.

We also pass in a parameter called 'input'. This parameter is declared as a constant. This means that we may not modify it within the procedure. We declare 'input' as an 'in' mode parameter. This means that it is only an input to the procedure. Note that again the use of 'constant' is redundant when the parameter is 'in' mode. Be aware of the direction of the modes as this can lead to confusion. When the parameter is declared as input it means that that data is *input to the procedure.*

We also give the input parameter a default. The default in this case is '0000'.

```
procedure hasparams (variable var1: inout integer;
               constant input: in bit_vector (0 to 3):=b'0000');
```

We call the procedure as follows:

```
hasparams (fred,b'1010');
```

This calls hasparams and specifies both of the input parameters. Since the second parameter has a default we may not always wish to specify it. Here the second parameter reverts to its default.

```
hasparams (fred);
```

We declare a function as follows:

```
function name parameter list return return_type
```

This is very similar to the procedure declaration except that it has an additional return type. Here is an example:

```
function paramreturn (var1: integer) return integer;
```

The function paramreturn has an integer passed into it and it returns an integer. There is a further restriction on functions: The parameters must either be constants or they must be of mode "in". This effectively means that parameters can only pass data into a function.

We can call paramreturn as follows:

```
variable fred: integer;
fred:=paramreturn (222);
```

The definition for procedures and for functions is very similar:

```
function paramreturn (var1: integer) return integer is
   variable declarations if needed
   begin
     body of functions
   return <something of type integer>
   end paramreturn
```

The return statement will be omitted for procedures because they don't return any data. Note that Quartus does not allow return statements within procedures (as you would expect), nor does it allow return statements to be within loop constructs (which you may not have expected). There must be a return statement within every function.

The variables which are declared within the procedure or function have local scope. They are only visible within the procedure and they override all other variables of the same name declared anywhere else.

In addition to all of the above we have a further major restriction: Functions may not have any side effects. This means that they may not modify global variables and they may not modify their parameters. This means that functions may be called with no effect on the calling environment. This limitation makes function calls much 'safer'.

In the simplest usage functions and procedures need to be declared within the architecture that they are used.

Here is an example of a function call within a VHDL code segment.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY test_procedure IS
     PORT
     (clk_input   : IN   STD_LOGIC;
      clr_input   : IN   STD_LOGIC;
      count       : OUT  INTEGER);
END test_procedure;

ARCHITECTURE a OF test_procedure IS

function increment (parameter1:integer) return integer is
begin
   return (parameter1+1);
end increment;

BEGIN
     PROCESS (clk_input, clr_input)
       variable fred:integer;
     BEGIN
          IF clr_input = '1' THEN
               fred:= 0;
          ELSIF (clk_input'EVENT AND clk_input = '1') THEN
                    fred:=increment (fred);
          END IF;
      count<=fred;
     END PROCESS;
END a;
```

10.2 Overloading

VHDL allows us to overload procedures and functions. This means that we may use the same procedure name for more than one function provided the parameter list differs. The procedure may have a different number of parameters and/or the parameters may be of different types. This allows us to use the same procedure name on different data.

We can also overload operators.

```
function "+" (a, b : word_32) return word_32 is
begin
    return int_to_word_32( word_32_to_int(a) + word_32_to_int(b)); end
"+";
```

Within the body of this function, the addition operator is used to add integers, since its operands are both integers. However, in the expression:

```
X"1000_0010" + X"0000_FFD0"
```

the newly declared function is called, since the operands to the addition operator are both of type word_32. Note that it is also possible to call operators using the prefix notation used for ordinary subprogram calls, for example:

```
"+" (X"1000_0010", X"0000_FFD0")
```

Many functions are overloaded in the VHDL libraries. The ieee.std_logic libraries overload operators for use with the std_logic types and we will see more examples of this later on.

11 PACKAGES

A package is a collection of VHDL constructs. The constructs can be of totally unrelated types, the only commonality being that usually a package implements some particular service and all of the constructs assist in some way.

The package also provides a means of hiding the working code behind functions and procedures, which is good for security as well as for "information hiding and abstraction".
An example of a package declaration is given here:

```
package data_types is
   subtype address is bit_vector(24 downto 0);
   subtype data is bit_vector(15 downto 0);
   constant vector_table_loc : address;
   function data_to_int(value : data) return integer;
   function int_to_data(value : integer) return data;
end data_types;
```

Here we group all of the above data members and functions together. This declaration does not in any way define the constant or the function bodies. This must be done in a package definition. Here is an applicable package definition:

```
package body data_types is
    constant vector_table_loc : address := X"FFFF00";

    function data_to_int(value : data) return integer is
     --body of data_to_int...
    end data_to_int;

    function int_to_data(value : integer) return data is
    --body of int_to_data...
    end int_to_data;
end data_types;
```

Once we have declared and defined the package we will want to use its contents. This may be done by attaching the package name to the items from it that we are using. Here we use three of the items from the package.

```
variable PC : data_types.address;
int_vector_loc := data_types.vector_table_loc + 4*int_level;
offset := data_types.data_to_int(offset_reg);
```

We can also use a "use clause" which tells the compiler to use items from a package. For example:

```
use data_types.address
```

If we wish to use all of the items in a package we may say

```
use data_types.all
```

Packages are a useful way of writing functions and putting them together into a bigger program. Using this method avoids the messiness of putting all your functions into the architecture declarative part. Be aware that you need to compile packages in the same way that you compile normal VHDL code. In Quartus this might require changing the project file to the package file, compiling and then changing the package file back to the main VHDL file.

A complete example of a packaged parity-calculating function is shown here:
Main VHDL file:

```
use work.packdemo.all;    --include the package file

library ieee;
use ieee.std_logic_1164.all;

entity test is
  port (input: in std_logic_vector(1 to 32);
        output: out std_logic);
end test;
architecture a of test is
begin
  output<=parity (input);
end a;
```

The package file:

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

--Declare the package elements
package packdemo is
   function parity (num: std_logic_vector (1 to 32)) return std_logic;
end packdemo;

--Define the package elements
package body packdemo is
function parity(num:std_logic_vector (1 to 32))return std_logic is
        variable ret: std_logic;
    begin
     for count in 1 to 32 loop
        ret:=ret xor num (count);
     end loop;
     return ret;
    end to_vector;
   end packdemo;
```

12 CONVERSION FUNCTIONS

VHDL is a very strongly typed language. This means that it will not permit programmers to 'force' one type into another. This makes it more rigid and formal. It imposes a discipline on programmers to be rigorous with their types.

Obviously it would be very inconvenient to be stuck without any means of converting types. For this reason there is a set of type converters provided. These functions take in a parameter of one type and return a different type. The output will be in some way equivalent to the input .

The basic available types are:

STD_LOGIC	a bit which may take on 0,1,X,Z
STD_LOGIC_VECTOR	a vector of std_logic bits
BIT	a bit which may be either 0 or 1
BIT_VECTOR	a vector of bits
CHARACTER	a single character
STRING	an array of characters
INTEGER	a signed number
SIGNED	a signed number
UNSIGNED	an unsigned number

Note that even when the types seem equivalent (INTEGER and SIGNED) a conversion function must still be used.

Some of the conversion functions that are provided are as follows:

- CONV_INTEGER--Converts a parameter of type INTEGER, UNSIGNED or SIGNED to an INTEGER value. The size of operands in CONV_INTEGER functions are limited to the range -2147483647 to 2147483647, i.e., to a 31-bit UNSIGNED value or a 32-bit SIGNED value.

- CONV_UNSIGNED--Converts a parameter of type INTEGER, UNSIGNED or SIGNED to an UNSIGNED value with SIZE bits.

- CONV_SIGNED--Converts a parameter of type INTEGER, UNSIGNED or SIGNED to a SIGNED value with SIZE bits.

- CONV_STD_LOGIC_VECTOR--Converts a parameter of type INTEGER, UNSIGNED, SIGNED, or STD_LOGIC to a STD_LOGIC_VECTOR value with SIZE bits.

Here is an example of a conversion in action:

```
ENTITY adder IS
    PORT (op1, op2 : IN UNSIGNED(7 DOWNTO 0);
         result    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END adder;

ARCHITECTURE maxpld OF adder IS
BEGIN
    result <= CONV_STD_LOGIC_VECTOR(op1 + op2, 8);
END maxpld;
```

There is a large collection of packages that are provided with the VHDL compiler. The following list is a sample of Altera's libraries.

File	**Package**	**Library**	**Contents**
std1164.vhd	std_logic_1164	ieee	Standard for describing interconnection data types for VHDL
arith.vhd	std_logic_arith	ieee	SIGNED and UNSIGNED types, arithmetic and comparison functions for use with SIGNED and UNSIGNED types, and the conversion functions CONV_INTEGER, CONV_SIGNED, and CONV_UNSIGNED. MAX+PLUS II does not support the shl(), shr(), ext(), or sxt() functions in the std_logic_arith package.
arithb.vhd			
signed.vhd	std_logic_signed	ieee	Functions that allow MAX+PLUS II to use STD_LOGIC_VECTOR types as if they are SIGNED types
signedb.vhd			
unsigned.vhd	std_logic_unsigned	ieee	Functions that allow MAX+PLUS II to use STD_LOGIC_VECTOR types as if they are UNSIGNED types
unsignedb.vhd			

If you use more than one of these packages in a single VHDL Design File, you must use them in the order in which they are listed in this table.

13 MEGAFUNCTIONS

Many of the library functions that are provided by Quartus are also available as modules which can be plugged into a graphical design. Some of these elements are extremely useful and are worth knowing about. Altera calls these components "Megafunctions" or LPM (Linear Parameterized Modules).

When you look at FPGA devices you will see that they have elements such as phase locked loops and RAM blocks. These are most efficiently used by the megafunction blocks. If you instantiate a megafunction the compiler matches desired block to the available chip features to produce a better solution than it would find if you implemented the block using normal VHDL.

The megafunctions are also available as VHDL components. If you need to incorporate them into a structural design then you can use the component declaration as you would expect.

For full information regarding the available megafunctions look in Quartus' help menu.

There are companies who write megafunctions and sell them. If you want to buy in intellectual property (IP) components then this is a way of doing it.

One of the biggest megafunctions around is the NIOS II microprocessor. This is a 32 bit CPU which fits onto a Cyclone FPGA (as used in the later labs) with ease. The NIOS II is capable of running uCLinux.

A huge collection of open source HDL cores can be found at www.opencores.org

14 VHDL Tutorial

1. What are the two fundamentally different methods of specifying the contents of a block of logic?

2. Explain, with a diagram, the following concepts:

Entities

Ports

Signals

3. What is the purpose of an entity declaration?

4. Declare an entity which contains one four input OR gate.

Exact syntax would not be required, as long as all the necessary parts are there.

5. Give the entity declaration and the structural architecture for a block which consists of two 2 input OR gates whose outputs are fed into a 2 input AND gate. You may assume the existence of the required components.

You would be given the basic skeleton in a test.

6. Explain the following code section:

```
entity testvhdl is
 port (ain: IN std_logic;
       dout: OUT integer range 3 downto 0);
end testvhdl;

architecture behave of testvhdl is
begin
 fred: process (ain)
 variable output: integer range 3 downto 0;
 begin
   if ain='1' then output:=output+1;
     dout<=output;
   end if;
 end process;
end behave;
```

7. What restrictions are placed on the parameters that are passed into VHDL functions? Give one reason that VHDL does not permit 'side effects', e.g. The modification of global variables.

8*. What is meant when it is said that VHDL is a strongly typed language? What type of functions are used to overcome the 'rigidness' of the strong typing?

9. What is the “sensitivity list” of a process?

10. What is the “mode” of a port?