VHDL Language

Introduction

VHDL is an acronym for VHSIC Hardware Description Language. VHSIC is an acronym for Very High Speed Integrated Circuits.

VHDL was developed by IBM, Texas Instruments and Intermetrics in response from a request by the US Department of Defence, in order to provide a uniform standard for VLSI design, so that firstly different designs could be evaluated and different design modules could be shared. Version 7.2 was developed and released in 1985. Some enhancements were incorporated and VHDL became an IEEE standard in 1987. (IEEE Std 1076-1987). Further enhancements were incorporated in 1993 (IEEE Std 1076-1993), 1999 (IEEE Std 1076-1999), and 2002 (IEEE Std 1076-2002),

As part of this standard the STD_LOGIC_1164 library was developed. This is IEEE Std 1164-1993 and is normally included in VHDL designs.

Other computer languages such as C, Java and Basic and their variants describe procedures for computing, in a step by step form, a mathematical function or manipulation of data. VHDL describes a digital system. As a result the operations are initiated on an event such as a Clock-pulse and the digital system performs all the relevant operations immediately, allowing for propagation delays. This non-sequential nature of VHDL is a trap for many programmers.

Texts:

1 “VHDL Primer”, J. Bhasker, 3rd Edition, 1999, Prentice Hall. ISBN 0-13-096575-8.

2 “Introduction to Digital systems”, Milos Ercegovac, Tomas Lang, Jaime H. Moreno, John Wiley & Sons, 1999. ISBN 0-471-57299-8. [Includes Alterra Max+Plus II 7.21 Student Edition Programmable Logic Development Software]

3 “Digital Design, principles and practices”, John. F. Wakerley, Fourth Edition. (Chapter 5.3). [Includes Xilinx Foundation software, Student Edition.]

4 “Synplicity FPGA Synthesis” Synplify for Lattice Reference Manual, Synplicity, September 2005, 513 pages (On the Lattice-EE4306 CD. Chapter 9 deals with VHDL)

5 VHDL Language Reference Guide. 1997, ALDEC, Inc. Located on the EE4306 web site.

Features

The Language can be used as an exchange medium between chip vendors and CAD and CAE design tool users or between different CAD and CAE tools. As schematic diagram can thus generate a VHDL source code, which is as an input to a simulation, an input to an EPLD or FPGA fitter program or a VLSI IC process. The results should be consistent.

© 1999-2008, C. J. Kikkert, James Cook University

1 VHDL supports “hierarchy” so that a system can be modelled as a series of interconnected components, which in turn can be modelled as a series of other interconnected components etc.

2 The language is an IEEE, ANSI and Military (Mil Std 454) Standard. The language is publicly available and is non-proprietary.

3 Test benches can be written in VHDL, to test VHDL models.

4 Synchronous and Asynchronous sequential logic, Propagation delays and spike detection all can naturally be done using the language.

5 The capability of defining new data types, allows the modelling of new design techniques, without any concern about the implementation details.

Example

The VHDL file below is a Dual Electronic Dice, which is the same function as is used as an example in CC2510, where the code is written in ABEL. Note at this stage this code is simply shown to show an overview of the way VHDL is structured.

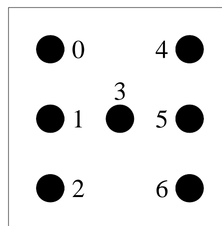

Figure 1. Dice layout.

The Dice is shown above and the 7 LEDs are numbered 0 to 6 and in the code are accessed as a 7 bit vector (0 to 6).

```
-- An dual electronic Dice. An example for EE4306.
-- Author C. J. Kikkert, James Cook University
-- original 2000 for isp2032, changes 25 june 2006 for ispM4A5

library ieee;
use ieee.std_logic_1164.all;

entity ElecDice is
    port(clk,run: in std_logic;
        LED1: out std_logic_vector (0 to 6);
        LED2: out std_logic_vector (0 to 6);
        Count1: inout integer range 0 to 7;
        Count2: inout integer range 0 to 7);
        attribute loc: string;
        attribute loc of clk : signal is "p11" ;      -- pin allocations on the CPLD
        attribute loc of run : signal is "p31" ;      --SW3 used for Run
        attribute loc of LED1 : signal is "p7, p2, p3, p8, p6, p5, p4" ;
        attribute loc of LED2 : signal is " p19, p14, p15, p20, p18, p17, p16" ;
            -- Seven Segment display order f a b g e d c for Dice like display
        attribute loc of Count1 : signal is "p41, p36, p40" ; --top half display
        attribute loc of Count2 : signal is "p398, p39, p40" ; --bottom half display
end ElecDice;
```

```
architecture ElecDice_arc of ElecDice is
begin
	p_counter:	process (clk)
	--counter module
	begin
		if rising_edge (clk)	then
			if run = '0' then
				if (Count1 = 6) then
					Count1 <= 1;
					if (Count2 = 6) then
						Count2 <= 1;
					else
						Count2 <= Count2 + 1;
					end if;
				else
					Count1 <= Count1 + 1;
				end if;
			else
				Count1 <= Count1;
				Count2 <= Count2;
			end if;
		end if;
	end process p_counter;

	p_leds1: process (Count1)
	--module to generate the LED display for the Dice
	begin
		case Count1 is
			when 0 => LED1 <= "0000000" ;
			when 1 => LED1 <= "0001000" ;
			when 2 => LED1 <= "1000001" ;
			when 3 => LED1 <= "1001001" ;
			when 4 => LED1 <= "1010101" ;
			when 5 => LED1 <= "1011101" ;
			when 6 => LED1 <= "1110111" ;
			when others => LED1 <= "0000000" ;
		end case;
	end process p_leds1;

	p_leds2: process (Count2)
	--module to generate the LED display for the Dice
	begin
		case Count2 is
			when 0 => LED2 <= "0000000" ;
			when 1 => LED2 <= "0001000" ;
			when 2 => LED2 <= "1000001" ;
			when 3 => LED2 <= "1001001" ;
			when 4 => LED2 <= "1010101" ;
			when 5 => LED2 <= "1011101" ;
			when 6 => LED2 <= "1110111" ;
			when others => LED2 <= "0000000" ;
		end case;
	end process p_leds2;
end ElecDice_arc;
```

As will be discussed later, this program consists of three parts. The first part indicates which libraries are used, like the #include statements in C. The second part “entity”

© 1999-2008, C. J. Kikkert, James Cook University

contains the connections to the outside world through inputs and outputs of a port, it also contains the pin allocations. The third part "architecture" contains the programming statements, which perform the required operations. In this situation the architecture contains three processes, or subprograms. The first process (p_counter) has two counters, which count from 1 to 6, thus indicating the number on the die. The first counter (Count1) operates six times as fast as the second counter (Count2), so that the whole process is repeated in 36 (6*6) clockpulses. The second process (p_leds1) is the output coding for the first die and the third process (p_leds2).

Just like subroutines in most programming languages, it is possible for the code in one "architecture" to call another "entity", so that very complicated functions can be realised.

Fundamental difference between VHDL and other languages

A fundamental difference between VHDL and other programming languages like C is that when the code is compiled, it is turned into a wiring connection specification, which produces a hardware IC with the operation specified in the code. Each of the processes form a sub-circuit that has inputs and outputs as specified and performs all the operations specified, either immediately when one of the inputs changes if the circuit is a combinational logic circuit or when a clock event occurs if the circuit is a synchronous sequential logic circuit. As a result in most cases order in which the statements occur in a procedure can be changed without effecting the resulting hardware.

A second difference is that since the outputs are specified for different input signals, any conditional branches must specify all the outputs for each branch, since otherwise the output is not fully specified and depending on the synthesis software (compiler) can result in undesired outputs.

Example:

A typical example is the following code, which is part of a larger program and checks if an ASCII character is part of a specified string (note AscDollar is specified as a constant elsewhere in program):

```
p_LetterMatch: process
    begin
    wait until rising_edge(ASCIIClk);       --new ASCII letter into buffer
        if (ASCIIData = AscDollar) then --received $
            WordCount <= X"0";          -- clear word Count correct first character
            SecDis <= SecDis;           -- hold the seconds display
        elsif (WordCount = X"F") then
            WordCount <= X"F";          -- past end of string or no match, wait for $
            SecDis <= SecDis;           -- hold the seconds display
        elsif ((WordCount < X"3") ) then       --next 4 characters ignore
            WordCount <= WordCount+X"1";
            SecDis <= SecDis;           -- hold the seconds display
        elsif (WordCount = X"3") then
            if (ASCIIData = AscG) then       --correct fourth character G
                WordCount <= WordCount+X"1";      -- correct string
                SecDis <= SecDis;           -- hold the seconds display
            else  --wrong string
                WordCount <= X"F";          --stop string compare, wait next $
                SecDis <= SecDis;           -- hold the seconds display
            end if;
```

```
        elsif (WordCount < X"B") then       --next 7 characters ignore
            WordCount <= WordCount+X"1";
            SecDis <= SecDis;               -- hold the seconds display
        elsif (WordCount = X"B") then       --correct seconds character, display
            WordCount <= WordCount+X"1";
            SecDis <= ASCIIData;     --display ,
        else  -- past required character for display or wrong string
            WordCount <= X"F";
            SecDis <= SecDis; -- hold the seconds display
        end if;
    end process p_LetterMatch;
```

Note that for each conditional branch "WordCount" and SecDis are specified. In C it can simply be assumed that values of variables do not change.

A third difference is that since each process specifies a hardware output, different processes cannot produce the same output, since that corresponds to shorting two outputs together.

Language Structure

Data Types

Data Object Names

Any alpha-numeric character as well as '_' (underscore) can be used in any name, provided the name is not a VHDL keyword. Data object names must begin with a letter and cannot end with an underscore. VHDL is not case sensitive Cout is the same as COUT. Often capitals, bold face or (blue) coloured text are used for keywords to make the code more readable. In this document capitals or bold are used depending on the context.

Comments are preceded by two consecutive dashes '--'. The compiler shows comments in a green colour.

Data Object values and numbers

Signal, Variable or **Constant** data objects are used to represent individual or multiple, logic inputs or outputs of a circuit, or binary (integer) numbers. The value of an individual logic signal is specified using apostrophes '0' and '1' are correct. The value of a multi-bit signal is specified using double quotes "1001" is a valid representation of a four-bit signal. Integers can be specified as decimal numbers such as 9 or 127, integers can also be specified as binary numbers by the use of double quotes ie "1001" for 9. Octal numbers can be written as O"244" for 164, Hexadecimal numbers have a prefix X like X"A4" for 164.

Units

VHDL has five different types of primary modules, called units. They are:

1. Entity declaration
2. Architecture body
3. Configuration declaration

© 1999-2008, C. J. Kikkert, James Cook University

4 Package declaration
5 Package Body.

Package declarations

The VHDL package serves as a package that holds VHDL code that is of general use, like code that defines data types. The Package can then be included in many other codes. The general package structure is:

```
Package package_name is
[type declarations]
[signal declarations]
[component declarations]
end package_name ;
```

In the above code, the **bold** or blue type represents the VHDL keywords. In the ispLever Compiler, those are shown as blue text. In that compiler comments are shown as green and numbers as purple. In these notes **bold** is normally used for language structure statements (as obtained from text books and blue type is normally used for code examples, that can be implemented directly into a program.

A package is used as follows:

```
library library_name;
use library_name.package_name.all;
```

The library_name represents the location in the file where the package is stored. The ieee system library is normally used and it contains four packages within that library:

std_logic_1164, Defines std_logic and std_logic_vector types and overloads

std_logic_signed, use this if you want std_logic_vector to be "signed"

std_logic_unsigned, - use this if you want std_logic_vector to be "unsigned"

std_logic_arith. - Basic arithmetic types (signed & unsigned) and overloads

Starting a new project under ispLever Classic, the std_logic_1164, std_logic_unsigned and std_logic_arith are automatically included in the VHDL template. These libraries are written in VHDL and are contained in ispTools\synbase\lib\vhdl_sim and ispTools\synbase\lib\vhdl.

The package body of std_logic_1164 is shown in Bhasker on page 349 and ispTools\odelsim\vhdl_src\ieee. A VHDL program will normally start with the following lines:

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

If additional packages within the IEEE library are required these can simply be added as:

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

Signal Types

BIT and BIT_VECTOR types are predefined in the VHDL standards IEEE 1076 and IEEE 1164. To use them no library is needed. The BIT type can have values of ‘0’ or ‘1’. The BIT_VECTOR type is a linear array of BIT objects.

© 1999-2008, C. J. Kikkert, James Cook University

Example:

```
signal X1 : bit ;
signal Byte : bit_vector (7 downto 0) ;
```

The ieee.std_logic_1164 package provides for the use of STD_LOGIC and STD_LOGIC_VECTOR types.

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

STD_LOGIC data objects are: 0,1, Z, –, L, H, U, X, W.

0 Forcing logic level 0 (same as BIT type)

1 Forcing logic level 1 (same as BIT type)

Z high impedance

– don't care

L weak 0 (treated as low by some compilers)

H weak 1 (treated as high by some compilers)

U un-initialised or weak unknown (treated as don't care by some compilers)

X Forcing unknown (treated as don't care by some compilers)

W Weak unknown (treated as don't care by some compilers)

Operators

Logical **and** **or** **nand** **nor** **xor** **xnor** **not**

Note **nand** and **not** are not associative so that 'A **nand** B **nand** C' is illegal. The function can be evaluated as '(A **nand** B) **nand** C' or as '**not** (A **and** B **and** C)'.

Relational = /= < <= > >=

Note /= is not equal.

Shift Operators

sll shift left logical -- L logically shifted left by R index positions.

srl shift right logical -- L logically shifted right by R index positions.

sla shift left arithmetic -- L arithmetically shifted left by R index positions.

sra shift right arithmetic -- L arithmetically shifted right by R index positions.

rol rotate left logical -- L rotated left by R index positions.

ror rotate right logical -- L rotated right by R index positions.

For the logical shifts, the vacated locations are filled with zeros. For arithmetic shifts, the vacated locations are filled with the most significant bit for a right shift and the least significant bit for a left shift. Shifts by a constant do not require any additional logic and are cheap to implement in hardware. Shifts by a –ve number imply a shift in the opposite direction. sra –2 is the same as sla 2.

© 1999-2008, C. J. Kikkert, James Cook University

Example

```
"1001010" sll 2; --results in "0101000"
"1001011" sra -2; --results in "0101111"
"1001010" sla 2; --results in "0101000"
```

NOTE: sll, srl, sla, sra, rol, ror are defined for type BIT and BIT_VECTOR only. For standard logic vectors, the same operation can easily be obtained using a simple assignment.

For example if PRS is a 11 bit standard logic vector, PRS must be declared as:

```
PRS: inout std_logic_vector (10 downto 0);
```

then

```
PRS sll 1;    --Left shift by one bit
```

Causes a compiler error, but the required result can be produced by a simple vector assignment:

```
PRS(10 downto 1) <= PRS(9 downto 0);
```

Arithmetic Operators

+ addition

- subtraction

& concatenation (adds an element to an array, "010001" & '1' results in "0100011")

* multiplication

/ division (Supported for compile-time constants and when divisor is a power of 2.)

** exponentiation (Supported for compile-time constants and when left operand is 2.)

abs absolute value

mod modulus (Supported for constants and when right operand is a power of 2.)

rem remainder (Supported for constants and when right operand is a power of 2.)

VHDL will allow division for any integer value, however multiplication and division for any number not a power of 2 is very expensive and should be avoided if possible. Multiplication or division by a constant required far less hardware than multiplication or division or two variables or signals.

1164 and 1067 Packages

The ieee.std_logic_1164 defines std_logic types, bit and vector, and Boolean operations. It does not handle relational or arithmetic comparisons

The ieee.std_logic_unsigned package allows arithmetic operators for logic vectors to be performed. The package does not perform arithmetic on integers or signed types.

The implementation of the std_logic_arith package does depend on the compiler realisation. For example Synopsys provides for the use of arithmetic operators such as +, -, * and relational for signed and unsigned types. Conversions to and from integer and logic vectors are included. The Synopsys package does not support arithmetic operation on logic vectors. The Mentor Graphics std_logic_arith package does do arithmetic operations on signed, unsigned and logic vector types. In addition it does division, mod and rem. Before the std_logic_arith package is used, one should

© 1999-2008, C. J. Kikkert, James Cook University

determine if it supports the types required. An easy way to determine this is to simply compile the required VHDL code with the appropriate library package.

The IEEE std 1076-1993 is an extension to VHDL to support the description and simulation of event driven systems. A further extension IEEE std 1067.6-1999 is an extension to VHDL to support the description and simulation of analogue and mixed-signal circuits and systems. Work on this standard is continuing with new amendments continuously being added.

The ieee.numeric_std will perform arithmetic opearations +, -, *. /, rem and mod on signed and unsigned numbers. It will also convert from logic vectors to unsigned and signed types. This library is part of the IEEE std 1076.3-1995. Most of our work will be done using the 1164 packages.

Declaring and Assigning Objects

Declaring Objects

Syntax:

[shared] <object_class> <object_name> : <data_type> [:= <initial _value>];

Where: Shared is an optional keyword used to create a shared variable in an entity or architecture. The value of the shared variable can be read or written anywhere within the corresponding architectures. (Like Global in C.)

The object_class is **signal**, **variable** or **constant**.

Signal is an object with a past history of values. A signal may have multiple drivers, each with a current value and projected future values. The term signal refers to objects declared by signal declarations and port declarations. Signals go into and out of the design. A signal can be declared inside an architecture body after the **architecture** line and before the **begin** line, as is shown in the wide adder example using components, later in these notes.

Variable is an object with a single current value. Variables are used inside a **process** or **function**. Examples of their use can be found in the VHDL examples included as part of the Lattice Software and available on the web.

Constant is an object whose value cannot be changed once defined for the design. Constants may be explicitly declared or they may be sub-elements of explicitly declared constants, or interface constants. Constants declared in packages may also be deferred constants.

data_type can be any predefined or user defined data type, such as std_logic_vector.

Assignments

The signal assignment is '<=' and the variable or constant assignment is ':='. The compiler will generate a warning if the wrong assignment is used.

Examples:

```
signal my_sig : integer range 0 to 15;       --  a 4 bit value.
my_sig <= 12;                                -- initialise it to 12.
variable my_var : std_logic_vector (7 downto 0);        -- a 8 bit vector.
my_var := "10100100"; -- set to 164 decimal. X"A4" for hex or O"244" for octal.
my_var(6) := '1';   -- set bit 6 to logical 1.
```

© 1999-2008, C. J. Kikkert, James Cook University

```
constant num_bits : integer := 7;
variable my_var : std_logic_vector (num_bits downto 0);     -- a 8 bit vector.
```

Entity declaration

The entity declaration specifies the name of the entity and lists the set of interface ports, which communicate with the outside world. The general entity structure is:

```
Entiry entity_name is
Port ([signal signal_name {,signal_name} : [mode] type_name { ;
        [signal signal_name {,signal_name} : [mode] type_name }) ;
end entity_name ;
```

For example a full adder has inputs A, B and $Carry_{in}$ and outputs Sum and $Carry_{out}$. The VHDL entity declaration for a full adder is:

```
entity Full_Adder is
        port (A, B, Cin : in BIT; Sum, Cout : out BIT) ;
end Full_Adder ;
```

Alternately the line can be broken up. In addition the word STD_LOGIC can be used instead of BIT (data objects 0,1). The STD_LOGIC type declaration is better, as it is part of the later standard, Std_Logic_1164 library and is an extension of the BIT type to include STD_LOGIC data objects: 0,1, Z, –, L, H, U, X, W, as outlined on page 5 of these notes.

```
entity Full_Adder is
        port (A, B, Cin : in STD_LOGIC ;
                Sum, Cout : out STD_LOGIC );
end Full_Adder;
-- This is a comment line, A Full Adder Entity declaration.
```

This second form is better when there are many input and output variables to be handled. Notice the Entity Declaration simply specifies the connections to the outside world and does not specify anything about the function of the device. This functional specification is done in the Architecture Body.

Architecture Body

The architecture declaration contains one or more of the following:

1 A set of interconnected components, such as the full adder above to make a complete circuit.
2 A set of combinational assignment statements to represent dataflow or combinational logic.
3 A set of sequential assignment statements to represent behaviour or sequential logic.

The logic for our full adder must firstly be evaluated:

Sum = A•/B• /Cin + /A• /B•Cin + /A•B•/Cin + A•B•Cin
CarryOut = A•B + A• Cin + B•Cin

For our full adder, a suitable architecture block is:

```
architecture FullAdd_arch of Full_Adder is
begin
        process (A, B, Cin)
        begin
```

© 1999-2008, C. J. Kikkert, James Cook University

```
        Sum <= (A and (not B) and (not Cin)) or ((not A) and (not B) and Cin)  or
              ((not A) and B and (not Cin)) or (A and B and Cin) ;
        Cout <= (A and B) or (A and Cin) or (B and Cin) ;
        end process;
    end FullAdd_arch ;
```

Process

Inside the Architecture body, one or more processes reside. Each process is event driven. The process statement either contains a "sensitivity list", where the expressions are evaluated every time one of the variables in the sensitivity list changes, or they must contain a wait statement, like "wait until rising_edge(clk)" and the process is started when the clock has a rising edge.

In the above example for the full adder, the process is executed every time a parameter in the sensitivity list changes. So a change in A, B or Cin, causes the Sum and Cout to be re-evaluated. The process is not run is there is no change in A, B or Cin.

In programming languages like Basic, C or C++, the statements are executed sequentially. In VHDL the statements in a process are executed on-demand and in parallel, when a signal in the sensitivity list changes.

In many cases there are more than one event driven function incorporated in the architecture block. Each of those is then contained in a separate process. Since an output is generated by a process and if the same event triggers two processes, both those processes are executed at the same time. This is very different from conventional programs like C code where the instructions are generated sequentially.

As a result in every process an output signal can only be modified once each time the process is done. The same signal cannot be generated by more than one process, since they could both operate at the same time.

This is very different from other programming languages and can cause confusion and unexpected results. For example the following code:

```
architecture behave of counter is
begin
    p1: process
    begin
        wait until rising_edge (clk);
        count <= count + 1;
        if ( reset = '1' or count = 255 ) then
        -- if synchronous reset comes along or max count, go to 0
            count <= 0;
        end if;
    end process p1;
end behave;
```

is wrong since if reset = '1' the counter count is both incremented and set to zero. Only one operation can be done and the compiler makes a choice which operation is to be performed, leading to unexpected results.

Similarly if there are two processes operating on the same event, and the same signal is operated on in each of the processes, then unexpected results will again occur. An example of such wrong code is:

```
architecture behave of counter is
begin
        p1: process
        begin
                wait until rising_edge (clk);
                count <= count + 1;
        end process p1;

        p2: process
        begin
                wait until rising_edge (clk);
                -- if synchronous reset comes along or max count, go to 0
                if ( reset = '1' or count = 255 ) then
                        count <= 0;
                end if;
        end process p2;
end behave;
```

The signal "count" is assigned a different value in both processes operating at the same event. Particularly with very large programs such errors can easily occur.

The following is the correct way to write the code. The output count is only changed once

```
architecture behave of counter is
begin
        p1: process
        begin
                wait until rising_edge (clk);
                if ( reset = '1' or count = 255 ) then
                        -- if synchronous reset comes along or max count, go to 0
                        count <= 0;
                else
                        count <= count + 1;
                end if;
        end process p1;
end behave;
```

If the counter is to be controlled by another process, then that process can set a flag, which is acted on by the process controlling the counter. For examples of this see the VHDL Examples lecture notes.

In many cases the program can be made easier to follow, by splitting it into different processes, for example the Sum and Carry assignment for our full adder can be put as separate processes as follows:

```
architecture FullAdd_arch of Full_Adder is
begin
        Sump: process (A, B, Cin)
        begin
                Sum <= (A and B and (not Cin)) or (A and (not B) and Cin) or
                        ((not A) and B and Cin) or (A and B and Cin) ;
        end process Sump;

Carry: process (A, B, Cin)
        begin
                Cout <= (A and B) or (A and Cin) or (B and Cin) ;
        end process Carry;
end FullAdd_arch ;
```

It is possible to change the statements, so that they are a little bit simpler, using XOR gates instead of AND and OR gates.

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

entity adder is
        port (a, b, cin : in std_logic;
                sum, cout: out std_logic);
end adder;

architecture behave of adder is
begin
        sum <= (a xor b) xor cin;
        cout <= (a and b) or (a and cin) or (b and cin);
end behave;
```

This one bit adder can be cascaded to form a wide adder as shown in figure 4of these notes. Using the arithmetic library significant savings in code can be achieved. This is illustrated using the following code for a vector adder, the number of bits can easily be changed in the **entity** statement. Since the same size std_logic_vector must be used for all the signals in an assignment, the overflow, or carry out from the most significant bit is not produced and overflows could go undetected.

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

entity addern is
        port ( a, b: in std_logic_vector (7 downto 0);
                result: out std_logic_vector (7 downto 0));
end addern;

architecture behave of addern is
begin
        result <= a + b;
end behave;
```

The above example is one of the examples from the ispLever installation in the synbase\examples\vhdl\combinat directory. In this example, the line "**use** ieee.std_logic_arith.**all;**" can actually be removed as the ieee.std_logic_unsigned package contains the required addition operation. The 1164, arith and unsigned libraries are all automatically included when a new VHDL project is started using ispLever. Since extra libraries normally do not cause any problems, it is advisable to leave these in the code.

Note a, b and result, are all 8 bit vectors, so that any overflow is ignored. This last realisation is very simple. The word length specification can easily be included, and for a 16 bit wide adder is:

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

entity addern is
        generic (msb_operand: integer := 15;
```

© 1999-2008, C. J. Kikkert, James Cook University

```
            msb_sum: integer := 15);
        port ( a, b: in std_logic_vector (msb_operand downto 0);
            result: out std_logic_vector (msb_sum downto 0));
end addern;

architecture behave of addern is
begin
        result <= a + b;
end behave;
```

The **generic** is an interface constant, declared in an entity declaration. Generics provide a channel for static information to be communicated to a block from its environment. Unlike constants, however, the value of a generic can be supplied externally, either in a component instantiation statement or in a configuration specification. The generic declaration allows us to thus easily change the number of bits used.

Clocked operations

For synchronous sequential logic, the command: **wait until** rising_edge(clk); causes the process to wait till a clock pulse, which then causes a change. This can easily be illustrated using a counter. Note that this wait until command will only cause that process to wait, any other processes in the architecture are triggered differently and will be executed when their sensitivity values change.

```
-- 8 bit counter with synchronous reset
-- Design from Synplify user guide

library ieee;
use ieee.std_logic_1164.all;

entity counter is
        port (clk, reset : in std_logic;
            count : buffer integer range 0 to 255);
end counter;

architecture behave of counter is
begin
        p1: process
        begin
            wait until rising_edge (clk);
            -- if synchronous reset comes along or max count, go to 0
            if ( reset = '1' or count = 255 ) then
                count <= 0;
            else
                count <= count +1;
            end if;
        end process p1;
end behave;
```

Concurrent and Sequential Signal Assignment

In VHDL an architecture is a logic module that specifies the connections for a circuit. As a result when the appropriate input on the sensitivity list is changed, a new output is generated. The hardware generates all the outputs at the same time. This is in start contrast to other programming languages like Fortran, C, Basic or assembly language for DSP etc, where the program represents a succession of operations which are

© 1999-2008, C. J. Kikkert, James Cook University

performed sequentially. This difference is a trap for the VHDL programmer and the subtle differences are illustrated in the following examples. Consider the following architecture:

```
architecture Seq_Sig_Arc of Flow is
signal A: std_logic;        --A and B are signals
signal B: std_logic;
begin
        process (B)
        begin
                A <= B;
                Z <= A;
        end process;
end;
```

When B changes all the statements inside the process are executed simultaneously. The value of Z is thus the initial value of A and is not the same as B. **A signal change inside a process statement is not available for use by that process until that process is executed again. As a result, the order is which these statements are placed inside the process is irrelevant. Note the synthesiser will give a warning that A should be included on the sensitivity list and automatically include A as in 2 below.**

In the following three equivalent realisations, are equivalent and for these architectures, when B changes, the process is executed resulting in a change of A, which then causes the process to be run again, resulting in a change of Z.

1 No Process used

```
architecture Conc_Sig_Arc of Flow is
signal A: std_logic;        --A and B are signals
signal B: std_logic;
begin
        A <= B;
        Z <= A;
end;
```

In this case any changes in A or B cause the statements to be executed. As soon as B changes the first statements is executed, which causes A to change, resulting in the execution of the second statement.

2 Fully specified sensitivity list

```
architecture Conc_Sig_Arc of Flow is
signal A: std_logic;        --A and B are signals
signal B: std_logic;
begin
        process (A, B)
        A <= B;
        Z <= A;
        end process;
end;
```

In this case A and B are on a specified sensitivity list and as soon as B changes the process is executed, which causes A to change, resulting in a re-execution of the process.

3 Implied sensitivity list

```
architecture Conc_Sig_Arc of Flow is
signal A: std_logic;        --A and B are signals
```

```
signal B: std_logic;
begin
        process
        A <= B;
        Z <= A;
        end process;
end;
```

In this case A and B are on an implied sensitivity list and as soon as B changes the process is executed, which causes A to change, resulting in a re-execution of the process.

The process is only executed upon a change of the signals used in the sensitivity list. For example, if a process has a sensitivity list of **wait until** rising_edge (clk); then no changes of the outputs from the process will occur until after the rising edge of the clock. If a variable is used inside that process, then a change in a variable is available immediately to the process and that can result in a different operation from above as illustrated in the following example.

Example

Write the VHDL Code for a circuit to generate the sequence: 1, 2, 3, 5, 8, 13, 21, 34, 55. and then stop at this last value and as soon as this last value is reached produce an output LED which is a logical 1.

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

entity SeqCount is
        port (Clk ,Rset: in STD_LOGIC;
                LED : out STD_LOGIC;
                Count : inout integer range 0 to 127);
end;

architecture SeqCount_Arch of SeqCount is
        signal Countlast : integer range 0 to 127;
begin
        process
        begin
                wait until rising_edge (clk);
                if (Rset = '0') then
                --must specify each of the 3 signals in each of the branches
                        Count <= 1;
                        Countlast <= 0;
                        Led <= '0';
                elsif (Count < 50 ) then
                        Countlast <= Count;
                        Count <= Countlast + Count;
                        Led <= '0';
                else
                        LED <= '1';
                        Countlast <= Countlast;
                        Count <=  Count;
                end if;
        end process;
end SeqCount_Arch;
```

The resulting waveforms are shown in figure 2.

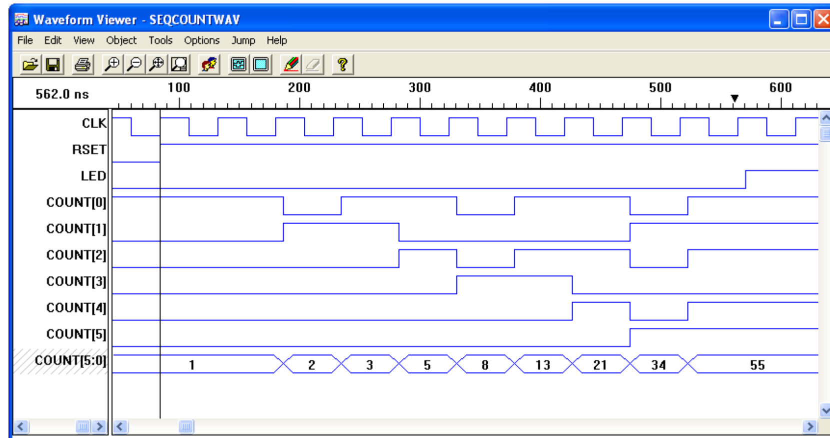

Figure 2. Waveforms for the counter using signals.

Note that only two signals, Count and CountLast are used. For a C program to be able to do this, 3 variables are required. In VHDL, the two statements:

```
Countlast <= Count;
Count <= Countlast + Count;
```

are executed simultaneously and the first line does not change the value of CountLast in the second line before it is calculated. We can have those two statements in any order without any change in the output.

The waveforms for this counter are shown below. Note that the bus display can be shown in Binary, Octal, Decimal or Hexadecimal, depending on what is most convenient.

Using Variables

Instead of using a signal, a variable can be used instead. The variable is valid inside a process and as a consequence is declared after the **process** statement. **A variable change inside a process statement is immediately available for use by that process. As a result the order of the statements is important and is just like other programming languages like C. As a result variables must be used in program loops in VHDL.** The assignments for Countlast must now use := instead of <= as shown below.

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

entity SeqCount is
        port (Clk ,Rset: in STD_LOGIC;
                LED : out STD_LOGIC;
                Count : inout integer range 0 to 127);
end;

architecture SeqCount_Arch of SeqCount is
begin
        process
        variable Countlast : integer range 0 to 127;
        begin
                wait until rising_edge (clk);
```

```
                if (Rset = '0') then
                --must specify each of the 3 signals in each of the branches
                        Count <= 1;--note <= since Count is a signal
                        Countlast := 0;      -- note := since Countlast is a variabl
                        Led <= '0';
                elsif (Count < 50 ) then
                        Countlast := Count;        -- note  := since Countlast is a variable
                        Count <= Countlast + Count; --note <= since Count is a signal
                        Led <= '0';
                else
                        LED <= '1';
                        Countlast <= Countlast;
                        Count <=  Count;
                end if;
        end process;
end SeqCount_Arch;
```

The code looks the same as before, except the as shown in figure 3, the output is different from that of figure 2.

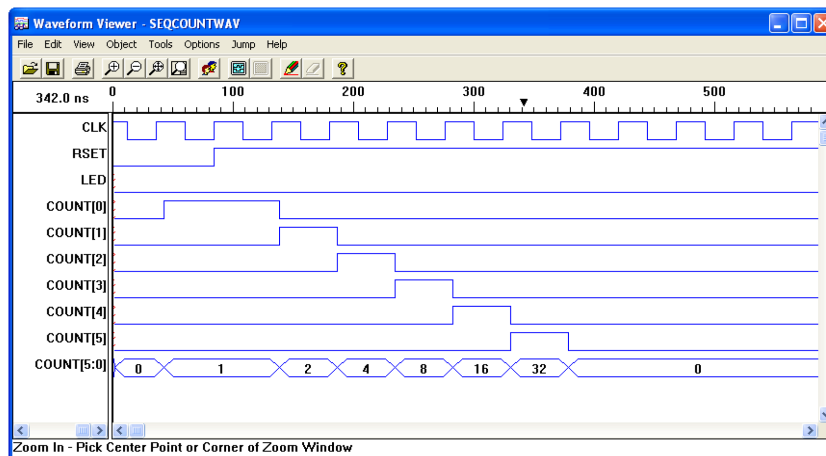

Figure 3. Waveforms for the counter, with using Variables

Consider the two important lines:

```
Countlast := Count;
Count <= Countlast + Count;
```

In this code Countlast is a **variable** and Count is a **signal.** When the variable is changed in the first line, this change is immediately available for the second line. The second line is then executed using the new first value. This then behaves like C code and gives the wrong result. Changing the order of these two statements gives the correct result. Note the second line above adds a variable and a signal. Because the result is a signal, the <= assignment needs to be used.

Program Flow

Wait Statement

```
wait on sensitivity-list;
wait until Boolean-expression;
wait for time expression;
wait on sensitivity-list until Boolean-expression for time expression;
```

Examples:

```
wait until A = B;
wait for 10 ns;
wait until rising_edge (clk);
```

IF Statement

```
if boolean-expression then
        sequential-statements
{elsif boolean-expression then
        sequential-statements}
[else
        sequential-statements]
end if;
```

Example:

```
if sum <= 100 then
        sum := sum +1;
elsif sum = 101 then
        sum := 127;
else
        sum := 0;
end if;
```

Case Statement

```
case expression is
        when choices => sequential-statements;
        when choices => sequential-statements;
        when others => sequential-statements;
end case;
```

Example:

The case statement can be used to implement a truth table, as implemented here for a hexadecimal to seven segment display conversion. For the equivalent truth table see the CC2510 lecture notes.

```
architecture SevenSeg_arch of SevenSeg is
begin
        process(Hexin)
        begin
                Lab0: case Hexin is
                -- need to have inverse of output coded since the board includes inverters
                        when X"0" => SevSegOut <= "0000001";   --0
                        when X"1" => SevSegOut <= "1001111";   --1
```

© 1999-2008, C. J. Kikkert, James Cook University

```
            when X"2" => SevSegOut <= "0010010";  --2
            when X"3" => SevSegOut <= "0000110";  --3
            when X"4" => SevSegOut <= "1001100";  --4
            when X"5" => SevSegOut <= "0100100";  --5
            when X"6" => SevSegOut <= "0100000";  --6
            when X"7" => SevSegOut <= "0001111";  --7
            when X"8" => SevSegOut <= "0000000";  --8
            when X"9" => SevSegOut <= "0000100";  --9
            when X"A" => SevSegOut <= "0001000";  --A
            when X"B" => SevSegOut <= "1100000";  --b
            when X"C" => SevSegOut <= "0110001";  --C
            when X"D" => SevSegOut <= "1000010";  --d
            when X"E" => SevSegOut <= "0110000";  --E
            when others => SevSegOut <= "0111000"; --F
        end case Lab0;
    end process;
end SevenSeg_arch;
```

The last case selection must always be when others even if there is only one value left, as is in the above example. A compiler error will result if this is not done.

In signal declarations an implied case statement can be used to initialise signals.

signal Init_P: std_logic_vector(7 downto 0) := (0 =>'1', others =>'U');

declares an 8 element vector and initialises element 0 to logic 1 and the other elements to logic value U (Unknown or don't' care for some compilers).

Null Statement

```
null;
```

Causes no action to take place and the next statement to be executed. This can be used in an IF or CASE statement to take no action under certain conditions. Since under every branch of an IF statement, all the outputs should be specified, the null statement should not occur in correctly written code.

Loop Statement

```
[loop-label:] iteration-scheme loop
    sequential statements
end loop [loop-label]
```

the iteration scheme can be:

```
for identifier in range
    while boolean-expression
    exit when expression
end loop
```

Notes on programming Loops with variables in CPLD devices: The Loop statement containing a variable permits a VHDL statement to be duplicated many times as part of this loop. A good example of this is the wide adder, which is the next example. For the required variables to be available immediately, an iterative requires the loop to be cycled through upon each event that causes the process to be executed. The example from Bhasker section 4.10:

```
variable Factorial : integer range 0 to N;  --where N is a constant set earlier
```

© 1999-2008, C. J. Kikkert, James Cook University

```
begin
Factorial :=1;
for Number in 2 to N loop;
        Factorial := Factorial*Number;
end loop;
```

Simply results in the VHDL program doing a calculation, which we can do manually and it does not produce any logic hardware. As a result the compiler may have difficulty compiling this and is likely to simply ignore this code as one that does not produce any hardware. The compilers included with the ispLever starter kit support loops if they are bound by constants or they have wait until statements to prevent combinational loops. The above example does not satisfy this. Loops can however be used very advantageously to do array indexing, as shown in the following example from the Synplify Reference Manual:

Example:

```
-- Loop Example from:
-- Synplify for Lattice Reference Manual (sep 2005) pp9-76
library ieee;
use ieee.std_logic_1164.all;

entity adder is
        generic(num_bits : integer := 4); -- Default adder size is 4 bits
        port (a : in std_logic_vector (num_bits downto 1);
                b : in std_logic_vector (num_bits downto 1);
                cin : in std_logic;
                sum : out std_logic_vector (num_bits downto 1);
                cout : out std_logic);
        end adder;

architecture behave of adder is
        begin
        process (a, b, cin)
                variable vsum : std_logic_vector (num_bits downto 1);
                variable carry : std_logic;
                -- vsun and carry variables to make them available immediately inside loop
                begin
                carry := cin;
                for i in 1 to num_bits loop
                        vsum(i) := (a(i) xor b(i)) xor carry;    -- := used since variable
                        carry := (a(i) and b(i)) or (carry and (a(i) or b(i)));
                end loop;
                sum <= vsum;        --converts variable to signal
                cout <= carry;      --converts variable to signal
        end process;
end behave;
```

Note any std_logic_vectors do not need to have the least significant bit as zero. In this instance the least significant bit is a 1. The block diagram with the lease significant bit as a zero is shown below.

© 1999-2008, C. J. Kikkert, James Cook University

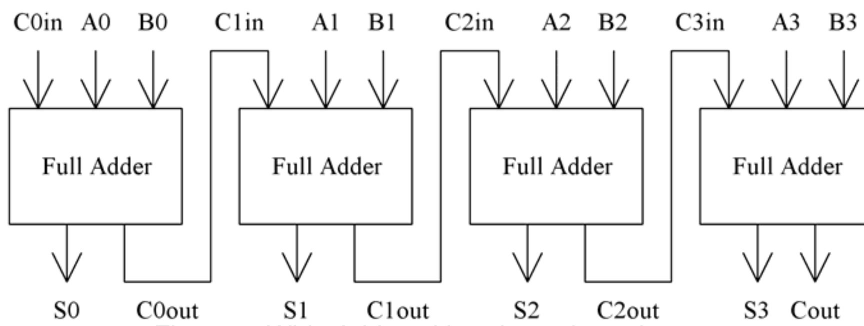

Figure 4. Wide Adder with a ripple through carry

The above code will generate a 4 bit wide adder, which includes a carry output, that can be used as an overflow indication. The size of the adder can easily be increased by simply changing the **generic**(num_bits : **integer** := 4); statement. The carrier rippling through the circuit will decrease the maximum operating speed and can cause some glitches as can be seen in the timing waveforms. To enhance the glitches, an 8 bit adder is used. The carry output works as expected.

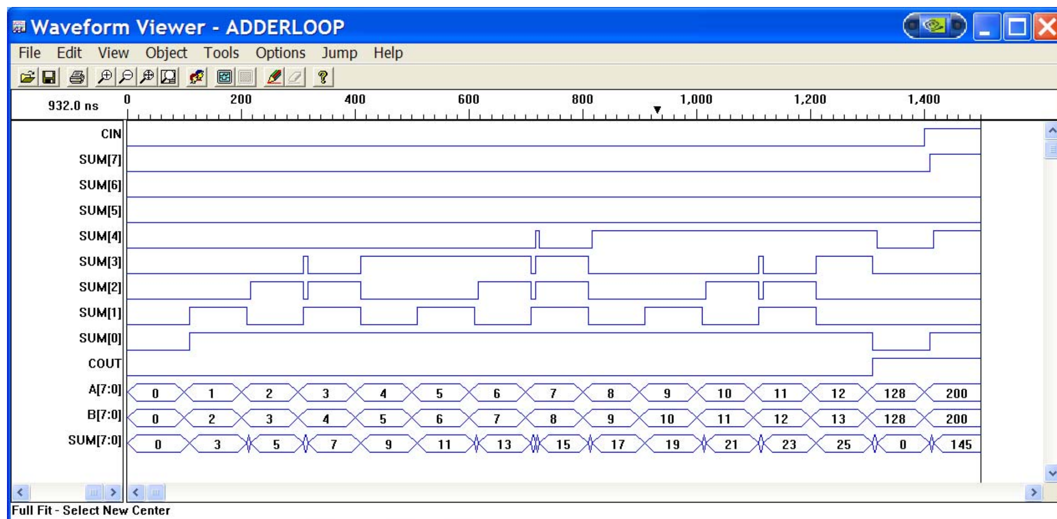

Figure 5. Waveforms for the adder, using timing simulation.

The glitches are due to the carry out from the least significant bit being the carry input for the second LSB and so on. If needed the glitches can be removed by using a series of flip-flops to store the correct values after the transitions have been completed.

Alternately the propagation delay can be reduced by making a 4 bit adder using a lookup table using 256 case statements, to produce a 5 bit output. Eight of these 4 bit adders can now be cascaded, as shown in figure 4 for the one bit adder, to produce a 32 bit wide adder, using only 8 propagation delays on the carry rippling through. The same 32 bit wide added made from single bit adders has a 32 gate propagation delay.

There is an optimum size for the number of bits added since if the Boolean algebra required to produce an output is bigger than the logic capacity of one Logical Unit, the hardware requirements are doubled.

Exit Statement

exit [loop-label] [**when** condition];

The exit statement causes the current (inner) loop to be terminated. It can only be used inside a loop. Note that the value of Count determines when the loop completes in the following example and the loop is executed 100 times.

Example:

```
Count := 0;
Clab: loop              --Clab is a loop label (optional)
        Count := Count +1;
        exit Clab when Count > 99;
end loop;
```

This is a simple counter that counts to 100.

Next Statement

next [loop-label] [when condition];

The next statement causes a jump to the end loop statement.

Example:

```
Count := 0;
Clab: for N in 0 to 200 loop              --Clab is a loop label (optional)
        when Count > 100 then
                next;
        else
        Count := Count +1;
end loop;
```

when the counter reaches 100, then the loop only increments N and does nothing else until N is 200 and the loop terminates.

Report Statement

report string-expression;

The report statement can be used to provide a message during execution and is useful in test benches. Example:

```
if CLK /= '0' and CLK /= '1' then
        report "Clock is not a valid logic level";
end if;
```

Since the Lattice Starter kit does not handle test benches, the report statement is of academic use only in this course.

Structural Modelling

Components

A component represents an entity/architecture pair. It specifies a subsystem, which can be instantiated in another architecture leading to a hierarchical specification. Component instantiation is like plugging a hardware component into a socket in a board

```
component component-name [is]
        [port (list –of –interface-ports);]
end component [component-name]
```

© 1999-2008, C. J. Kikkert, James Cook University

Example:

Consider the realisation of an 8 bit wide adder. If we use the arithmetic libraries, the carry-out signal will not be available. We can however make our own adder by simply using a half adder and a succession of full adders. This half adder and the full adder is thus used as components for an 8 bit wide adder.

```
-- An example of using components and VHDL hierarchy, to
-- create an 8 bit wide full adder with a carry out.
-- Author C. J. Kikkert James Cook University -- 4 August 2000

-- start of Full Adder module
library ieee;
use ieee.std_logic_1164.all;

entity FullAdd is
        port (FA, FB, FCin : in std_logic;
                FSum, FCout : out std_logic);
end;

architecture FullAdd_Arc of FullAdd is
begin
        FSum <= (FA xor FB) xor FCin;
        FCout <= (FA and FB) or (FA and FCin) or (FB and FCin);
end FullAdd_Arc;
--------------------------------------------
-- end of Full adder Module, which is used as a component later on
--------------------------------------------

-- start of Half Adder module
library ieee;
use ieee.std_logic_1164.all;

entity HalfAdd is
        port (HA, HB : in std_logic;
                HSum, HCout : out std_logic);
end;

architecture HalfAdd_Arc of HalfAdd is
begin
        HSum <= (HA xor HB);
        HCout <= (HA and HB);
end HalfAdd_Arc;
--------------------------------------------
-- end of Full adder Module, which is used as a component later on
--------------------------------------------

-- Start of Main Program, which calls the components
library ieee;
use ieee.std_logic_1164.all;

entity AddWide is
        port (A, B : in std_logic_vector (7 downto 0);
                Sum : out std_logic_vector (7 downto 0);
                Cout : out std_logic);
end;

architecture AddW_Arc of AddWide is
```

© 1999-2008, C. J. Kikkert, James Cook University

```
        component FullAdd
                port (FA, FB, FCin : in std_logic;
                FSum, FCout : out std_logic);
        end component;
        component HalfAdd
                port (HA, HB: in std_logic;
                HSum, HCout : out std_logic);
        end component;
        signal CInt : std_logic_vector (7 downto 1);
begin --the portmap statements make the connections as shown in figure 4
        F0: HalfAdd port map (A(0), B(0), Sum(0), CInt(1));
        F1: FullAdd port map (A(1), B(1), CInt(1), Sum(1), CInt(2));
        F2: FullAdd port map (A(2), B(2), CInt(2), Sum(2), CInt(3));
        F3: FullAdd port map (A(3), B(3), CInt(3), Sum(3), CInt(4));
        F4: FullAdd port map (A(4), B(4), CInt(4), Sum(4), CInt(5));
        F5: FullAdd port map (A(5), B(5), CInt(5), Sum(5), CInt(6));
        F6: FullAdd port map (A(6), B(6), CInt(6), Sum(6), CInt(7));
        F7: FullAdd port map (A(7), B(7), CInt(7), Sum(7), Cout);
end AddW_Arc;

--Note that the port mapping allows the input and output to be an array.
```

The hardware generated by this use of components is similar to the hardware generated by the use of the loop example on page 19 of these notes, the main difference being that a half adder is used as the first element in the above example and only full adders are used in the loop example.

Position and Naming Association

The port mapping shown above has the order of the signals in the architecture exactly the same as that in the component. This is a position association. It is possible to use a different order by specifying the names to be mapped using the => operator to explicitly reference the pairs of signals to be mapped. Since the pairs to be mapped are fully specified, the order in which they are specified can be changed if needed. This is a naming association and shown in the code below. Note that the order of port mapping of HA, HB, HSum and HCarry is different form the order in the component specification. The same can be applied to the port mapping for the full adder.

```
begin --the portmap statements make the connections as shown in figure 4
F0: HalfAdd port map (HSum=>Sum(0), HB=>B(0), HCout=>CInt(1), HA=>A(0));
........
end AddW_Arc;
```

The Half adder module and the full adder module can be in different VHDL files. It is thus convenient to build up a library of commonly used functions, like full and half adder or a Hexadecimal to seven-segment display. Those functions are then simply imported into the main program hierarchy.

Hierarchy

In the above code for the 8 bit wide adder, the Half Adder and the Full Adder modules are used in the top level (main) program as a component. A hierarchy is thus created, with the Half Adder and Full Adder being at a lower level and being called by the higher level. Extending this process allows very complex hardware to be produced.

© 1999-2008, C. J. Kikkert, James Cook University

Example

Hexadecimal counter and seven segment display. The M4A5 development boards used in this course have a seven segment display on them. The hardware for the displays includes a driver IC with inverters. The code for the seven segment display is best generated using a lookup table. In VHDL, that needs to be done using a case statement.

```
-- Up/Down Counter for EE4306
-- Author C. J. Kikkert, James Cook University
-- Date 16 Aug 2004
-- This is a counter to: Firstly test the functionality of the M4A5 Development boards
-- Secondly show the use of a hierarchical structure to reuse the seven segment display -
-- code
-- Thirdly show how a parts of long vector can be used
-- Fourthly how a lookup table needs to be implemented in VHDL.
-- Module for a 4 bit to seven segment display decoder
-- Author C. J. Kikkert, 16 Aug 2004

library ieee;
use ieee.std_logic_1164.all;

entity SevenSeg is
        Port (Hexin: in std_logic_vector (3 downto 0);
                SevSegOut: out std_logic_vector (6 downto 0));
end;

architecture SevenSeg_arch of SevenSeg is
begin
        process(Hexin)
        begin
                Lab0: case Hexin is
-- need to have inverse of output coded here since the board displays include inverters.
                        when X"0" => SevSegOut <= "0000001";   --0
                        when X"1" => SevSegOut <= "1001111";   --1
                        when X"2" => SevSegOut <= "0010010";   --2
                        when X"3" => SevSegOut <= "0000110";   --3
                        when X"4" => SevSegOut <= "1001100";   --4
                        when X"5" => SevSegOut <= "0100100";   --5
                        when X"6" => SevSegOut <= "0100000";   --6
                        when X"7" => SevSegOut <= "0001111";   --7
                        when X"8" => SevSegOut <= "0000000";   --8
                        when X"9" => SevSegOut <= "0000100";   --9
                        when X"A" => SevSegOut <= "0001000";   --A
                        when X"B" => SevSegOut <= "1100000";   --b
                        when X"C" => SevSegOut <= "0110001";   --C
                        when X"D" => SevSegOut <= "0110000";   --d
                        when X"E" => SevSegOut <= "0111100";   --E
                        when others => SevSegOut <= "0111000"; --F
                end case Lab0;
        end process;
end SevenSeg_arch;      -- end of 4 bit to seven segment display decoder module

-- Module for a 16 bit up/down counter
-- Author C. J. Kikkert, 16 Aug 2004
library ieee;
use ieee.std_logic_1164.all;
```

© 1999-2008, C. J. Kikkert, James Cook University

```
entity Counter is
        Port (Clk : in std_logic ; --2KHz
                SevSeg0: out std_logic_vector (6 downto 0);     -- display LS Digit
                SevSeg1: out std_logic_vector (6 downto 0);     -- display Digit 2
                SevSeg2: out std_logic_vector (6 downto 0);     -- display Digit 3
                SevSeg3: out std_logic_vector (6 downto 0);     -- display MS Digit
                Up: in std_logic;                               --Count UP button SW1
                Down: in std_logic);                            --Down button SW2
        attribute loc : string;
        attribute loc of Clk: signal is "P11" ;
        attribute loc of Up: signal is "P9";                    --Start button SW1
        attribute loc of Down: signal is "P31";                 --Stop button SW3
        attribute loc of SevSeg3: signal is "P2, P3, P4, P5, P6, P7, P8" ;--pin allocation of MS display
        attribute loc of SevSeg2: signal is "P14, P15, P16, P17, P18, P19, P20" ;
        attribute loc of SevSeg1: signal is "P24, P25, P26, P27, P28, P29, P30" ;
        attribute loc of SevSeg0: signal is "P36, P37, P38, P39, P40, P41, P42" ;
end;

architecture Counter_arch of Counter is
        component SevenSeg
                port (Hexin: in std_logic_vector;
                SevSegOut : out std_logic_vector);
        end component;
        signal Count: std_logic_vector (15 downto 0);       -- 16 bit counter for all bits.
        signal UpDown: std_logic;                        -- storage for Up/Down direction
begin
        D0: SevenSeg port map (Count(3 downto 0), SevSeg0); -- Display LS Digit
        D1: SevenSeg port map (Count(7 downto 4), SevSeg1); -- Display Digit 2
        D2: SevenSeg port map (Count(11 downto 8), SevSeg2); -- Display Digit 3
        D3: SevenSeg port map (Count(15 downto 12), SevSeg3); -- Display MS Digit

        P_Clock: process  --this does the up/down counting
        begin
        wait until rising_edge(Clk);
                if (UpDown = '1') then
                        Count <= Count + X"0001" ;          -- Count up
                else
                        Count <= Count - X"0001" ;          -- Count Down
                end if;
        end process;
        P_UpDown: process (Up, Down)       -- this sets the up/down flag
        begin
                if (Up = '0') then                       --Up button pressed Count Up.
                        UpDown <= '1';
                elsif (Down = '0') then                  --Down button pressed Count Down.
                        UpDown <= '0';
                end if;
        end process P_UpDown;
end Counter_arch;                                        -- end of up/down counter module
```

Conversion Functions

It is often necessary to convert from one type to another. Type conversion is allowed between related types. Types are related if it is obvious to the compiler what to do. For instance integer types are related, array types (std_logic_vector and bit_vector) of the same size are related. Simple conversions are pemitted using type declarations. The process is illustrated by the following example:

© 1999-2008, C. J. Kikkert, James Cook University

```
Type int16bit is integer range 0 to 16383;
Type int8bit is integer range 0 to 255;
Signal short: int8bit;
Signal long: int16bit;
Long <= int16bit(short); --converts the 8 bit short integer into a 16 bit value.
```

For components the type conversion can be done in the entity declaration. For example if the entity declaration of a counter written by someone else is:

```
entity Counter is
        port (Q: inout STD_LOGIC_VECTOR (3 downto 0);
                Clock, RESET: in STD_LOGIC;
                PARITY: out STD_LOGIC);
end COUNTER;
```

and we want to use this as a component in our architecture module, with the ports:

```
component COUNTER
        port (CTR : inout BIT_VECTOR;
                CLK, RST : in BIT;
                PAR : out BIT);
end component;
```

this can be done by using the port mapping, but instead of using the => mapping symbol conversion functions must be used as well.

without conversion the port map would be:

```
        port map( Q => CTR,             -- a bi-directional, inout function
                CLOCK => CLK,           -- an input ie from CLOCK to CLK
                RESET => RST,           -- an input ie from RESET to RST
                PARITY => PAR);         -- an output ie from PAR to PARITY
```

the conversion function needs to be added to the name that is to be converted to. The port map will thus become:

```
        port map( TO_BITVECTOR(Q) => TO_STDLOGICVECTOR(CTR),    --bi-directional.
                -- Since this is bidirectional both need to be able to be converted.
                -- Q is std_logic_vector converted to bit_vector and
                -- CTR is bit_vector converted to std_logic_vector
                CLOCK => TO_STDLOGIC(CLK),              -- from CLOCK to CLK
                RESET => TO_STDLOGIC (RST),             -- from RESET to RST
                -- CLOCK and RESET are inputs is std_logic. Clk and Rst need to be
                -- converted from bit to std_logic to be used in the component.
                TO_BIT(PARITY) => PAR);                 -- from PAR to PARITY
                --PAR is an output. PARITY needs to be converted to bit to be used in the
                --module one is writing.
```

The use of conversion as part of the component mapping is thus not simple. The conversion functions TO_STDLOGICVECTOR, TO_STDLOGIC, TO_BITVECTOR are not part of a standard set of functions and will need to be created, as a function similar to the TO_CHARACTER function below.

Functions and Procedures

Function and Procedures are both subprograms, which allow the code to be structured in a simple manner. A function returns a single value, a procedure maps input and output signals.

© 1999-2008, C. J. Kikkert, James Cook University

A good example of a function is a function, which converts a STD_LOGIC variable to a CHARACTER, to be displayed or written to a file. For a function the parameters inside the brackets are always inputs.

```
function TO_CHARACTER (ARG: STD_LOGIC)
        return CHARACTER is
begin
        case ARG is
        when 'U' => return 'U';
        when 'X' => return 'X';
        when '0' => return '0';
        when '1' => return '1';
        when 'Z' => return 'Z';
        when 'W' => return 'W';
        when 'L' => return 'L';
        when 'H' => return 'H';
        when '-' => return '-';
        when others => return 'E'; --this is not included in many implementations
end TO_CHARACTER;
```

A procedure is a subprogram where the parameters are passed in and out as indicated.

For example:

```
procedure (A,B: in INTEGER; D: out BIT; E inout STD_LOGIC_VECTOR(7 downto 0);
```

has an input A and B, which are integers, an output D, which is a single bit and E, which is an 8 bit vector, that are passed both in and out.

Overloading

Sometimes two functions, procedures or operators have the same name or symbol, but are different and have different input or output types. Such functions, procedures or variables are said to be “overloaded”.

For example:

```
function TO_CHARACTER (ARG: STD_LOGIC)
        return CHARACTER;
```

and the function

© 1999-2008, C. J. Kikkert, James Cook University

```
function TO_CHARACTER (ARG: STD_LOGIC_VECTOR)
        return CHARACTER;
```

and the function

```
function TO_CHARACTER (ARG: INTEGER)
        return CHARACTER;
```

are all different. They are overloaded functions when they are used in the same program. The compiler will select, which function is to be used, according to the type specified. When the compiler cannot select which function is to be used, because of a conflict in type specifications, a compiler error is generated.

The operator + is defined as the expected adder operator for INTEGER types in the ieee.std_logic_arith library. The same symbol is defined as an adder for STD_LOGIC_VECTOR types in the ieee.std_logic_unsigned library.

For example if 'Count' is used as an input and is of INTEGER type, then the code:

Count <= Count + 1; is correct since all operands are INTEGERS.

however if 'A' is used as an input and is of STD_LOGIC_VECTOR (7 **downto** 0) type, then the code:

A <= A + 1;

will generate an error since 'A' is a STD_LOGIC_VECTOR and '1' is an INTEGER.

The code:

A <= A + X"01"; is correct since all operands are STD_LOGIC_VECTOR, however care must be taken that the size of the std_logic_vector signals are all the same length, otherwise a compiler error occurs.

Similarly an error would occur for:

```
when (A < 126) then
```

But

```
when (A < X"7E") then
```

is correct and will compile properly.

Note some compilers are more tolerant than others for minor variations from the IEEE standards.

© 1999-2008, C. J. Kikkert, James Cook University