DAT093
Introduction to Electronic System Design
VHDL update

Sven Knutsson
svenk@chalmers.se
Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg
Sweden

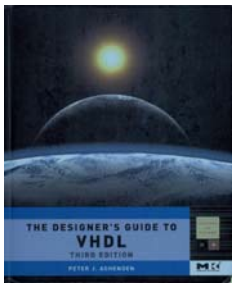
Literature

We do not require any advanced book on VHDL in this course

Since many of you already have some book on VHDL you can probably go on using that if it is not to elimentary

but if you need a book we recommend

Peter J. Ashenden:
The Designer´s Guide to VHDL, 3 ed
ISBN 978-0-12-088785-9



$48:04 on amazon.com

619:00 SEK on bokus.se

509:00 SEK on adlibris.com

older editions will do just as well

Your can find the book as an ebook at Chalmers library

If you google you can also find it as a PDF on the net

How to describe your electronic design

Schematic

- Components and nets
- Analog or digital

Netlist

- Component descriptions and node connections describing a schematic
- Analog or digital
- EDIF - Electronic Design Interchange Format

'Programming' language

- Program code converted (syntesized) to electronics
- Digital with some, not that successful, attempts to go analog

Programming languages

Can describe systems on a high system level

- On that level both electronic and mechanical parts can be included
- Electronics can be both analog and digital

If we move down to design (synthesis) level we will in most cases have to restrict the description to digital electronics

Programming languages cont.

VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

- IEEE (Institute of Electrical and Electronics Engineers) standard
- Strictly typed language Type conversions needed
- Popular in Europe

Verilog

- IEEE standard
- More loose description than VHDL
- Can give different results in different compilers
- Popular in USA

SystemVerilog

- IEEE standard
- Extention to Verilog with a more strict description style
- Seems to slowly become the new standard description language

Programming languages cont.

System Verilog has so far only been successful for system description and verification.

When it comes to synthesis we have to stay with VHDL or Verilog

Programming languages cont.

There are atempts to use C or C++ for electronic design since these languages are so widely spread

Most atempts have resulted in languages that just use a restricted subset of C or C++ or the language have been changed, sometimes almost beyond recognition

Examples

System C

Mostly for simulation

Handel C

One vendor (Celoxica)

Bought by Mentor, will be integrated into Catapult C

Catapult C

One vendor (Mentor) Sold to Calypto on August 26, 2011

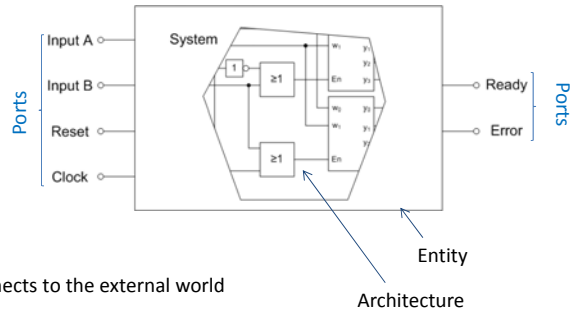
Expensive

more are comming but the hype seem to be gone

VHDL basics

Basically a system description falls into two parts

- The outside, the interface. How to connect to other systems or to the external world?
- The internals, the functional description of the design

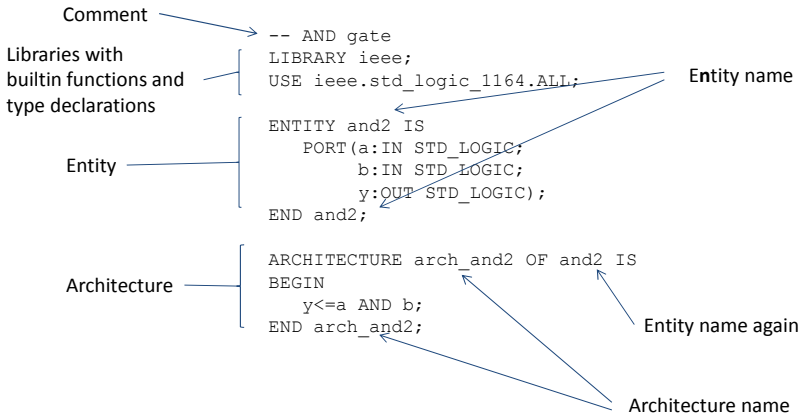


In VHDL

- An **entity** connects to the external world
- An **architecture** describes the internal functionality

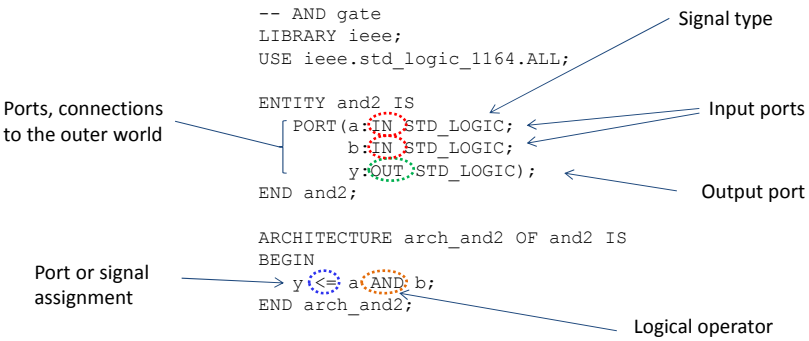
VHDL basics

Example: Simple AND gate



VHDL basics cont.

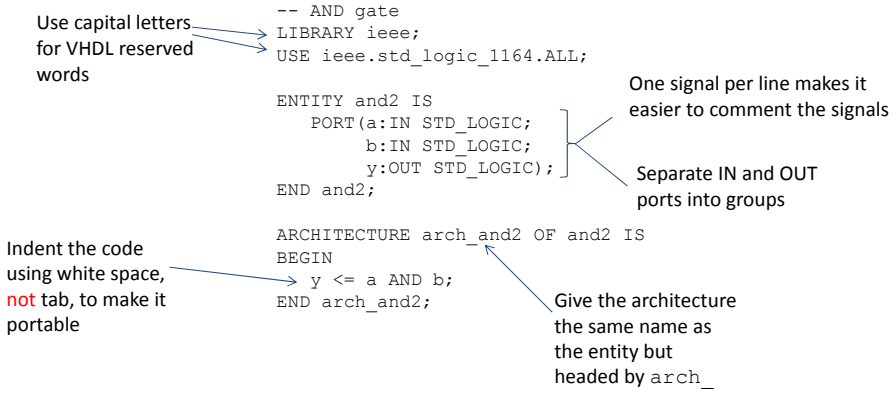
Example: Simple AND cont.



VHDL basics cont.

Example: Simple AND cont.

Let´s look at the coding style



VHDL basics cont.

We can write our code in two different ways

- Structural code

  The code is like a netlist (schematic) with components (sub blocks) that are interconnected

  Only suitable in smaller designs

  and to interconnect several designs into a larger design

- Behavioral code

  The code describes the functionallity we like to achieve, not the structure of the design

  In many cases the best choice

  In a practical design we often split the construction into blocks where we use behavioral code to describe the internals of these blocks and structural code for the interconnection of the blocks

VHDL basics

Let´s look at our simple AND gate

```
-- AND gate
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END and2;

ARCHITECTURE arch_and2 OF and2 IS
BEGIN
   y<=a AND b;
END arch_and2;
```

This is a structural design since we are using the function, or component if you like, AND

VHDL basics cont.

```
-- AND gate
LIBRARY ieee;
USE
ieee.std_logic_1164.ALL;
```

```
ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END and2;
```

```
ARCHITECTURE arch_and2 OF and2 IS
BEGIN
   y <= a AND b;
END arch_and2;
```

If we rewrite the code for the AND device in a behavioral way we will get

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

```
ENTITY and2 IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END and2;
```

The entity is the same

```
ARCHITECTURE arch_and2_behavioral OF and2 IS
BEGIN
   y <= '1' WHEN (a='1') AND
                 (b='1') ELSE
        '0';
END arch_and2_behavioral;
```

The architecture has changed

In this case the behavioral desciption is somewhat more complicated but this is no general rule

VHDL basics cont.

If we look att the code from another perspective we can also see two types of code

- Concurrent code

  Parallel code, things happen at the same time, in parallel
  We have different elctronic structures

- Sequential code

  The code is interpreted as a sequence, line by line (compare to programming code), things happen in sequence

  This code has to be written in a **process**

  The whole process is concurrent with the rest of the code

VHDL basics cont.

Example

Entity

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

VHDL basics cont.

Example

Architecture

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
   SIGNAL x_seq_signal:STD_LOGIC;

BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
   BEGIN
      x_seq_signal <= a AND b;
      y_seq <= x_seq_signal OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

Internal signals interconnections

Concurrent code

Process name

Sequential code

Sensitivety list

The signals that **trigger** (activate) the process

VHDL basics cont.

Example cont.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
   SIGNAL x_seq_signal:STD_LOGIC;

BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
   BEGIN
      x_seq_signal <= a AND b;
      y_seq <= x_seq_signal OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

The `x_seq_signal` value is updated when we leave the process

The value `x_seq_signal` had when we entered the process is used, the assignment on the line above isn´t effective yet

VHDL basics cont.

Example alternative

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
BEGIN
   x_conc_signal <= a AND b;
   y_conc <= x_conc_signal OR c;

   seq_proc:
   PROCESS(a,b,c)
      VARIABLE x_seq_variable:STD_LOGIC;
   BEGIN
      x_seq_variable := a AND b;
      y_seq <= x_seq_variable OR c;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

Variable, local to the process, not visable outside of the process

The `x_seq_variable` value is immediately updated and passed on to the OR statement

Variable asignment

The new value of `x_seq_variable` is used

VHDL basics cont.

What if we change the order of the statements?

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and_or IS
   PORT(a:STD_LOGIC;
        b:STD_LOGIC;
        c:IN STD_LOGIC;
        y_conc:OUT STD_LOGIC;
        y_seq:OUT STD_LOGIC);
END and_or;
```

```
ARCHITECTURE arch_and_or OF and_or IS
   SIGNAL x_conc_signal:STD_LOGIC;
BEGIN
   y_conc <= x_conc_signal OR c;
   x_conc_signal <= a AND b;

   seq_proc:
   PROCESS(a,b,c)
      VARIABLE x_seq:STD_LOGIC;
   BEGIN
      y_seq <= x_seq_variable OR c;
      x_seq_variable := a AND b;
   END PROCESS seq_proc;
END arch_and_or;
```

The `x_conc_signal` value is immediately updated and passed on to the OR statement

change of order

`x_seq_variable` is updated after the use in the `y_seq` assignment because of the order of the statements

The statement uses the `x_seq_variable` value we had when we entered the process

change of order

VHDL basics cont.

How do we test our code?

Simulation

The standard tool for simulation is ModelSim/QuestaSim from Microtech (bought by Mentor)

An aid in the simulation is the test bench

A test bench is a VHDL structure where we encapsulate our design as a component and generate stimuli to the inputs of the design and watch the results at the outputs (and internal nodes)

- The description above is for the basic test bench (type 1)
- We can improve the test bench by adding code that tests that the output signals are as expected when we apply the input stimuli and indicates correct or not by this using an OK signal (type 2)
- We can also improve the test bench by adding code that tests that the output signals are as expected when we apply the input stimuli and write information to the simulator´s output window if an error occurs, describing the type of error and at what simulation time the error occured and other information that we like to come out (type 3)

VHDL basics cont.

Before we move on with VHDL

Number representation

Number bases

Decimal

Base 10

Values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Integer example

$2346 = 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0$

Position weights

Real example

$3.27 = 3 \cdot 10^0 + 2 \cdot 10^{-1} + 7 \cdot 10^{-2}$

MSD (Most significant digit)

LSD (Least significant digit)

Position weights

VHDL basics cont.

Number bases cont.

Hexadecimal

Base 16

Values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Integer example

Position weights

$92A = 9 \cdot 16^2 + 2 \cdot 16^1 + A \cdot 16^0$

Real example

$3.45 = 3 \cdot 16^0 + 4 \cdot 16^{-1} + 5 \cdot 16^{-2}$

MSD (Most significant digit)

LSD (Least significant digit)

Position weights

VHDL basics cont.

Number bases cont.

Binary

Base 2

Values: 0, 1

Position weights

Integer example

$100100101010 = 1 \cdot 2^{11} + 0 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 +$

$+ 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

Position weights

Real example

$10.01 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$

MSB (Most significant bit)

LSB (Least significant bit)

Position weights

VHDL basics cont.

Number bases cont.

Since we deal with logical, digital signals, we will concentrate on binary numbers but internally within a design it is often convenient to use integers.
Make sure to restrict the values to those needed

Since we are designing hardware, actual wires, we will have to deside how many wires to use, that is how many binary bits we will use to represent our binary numbers

The number of bits might change as needed in different parts of the design

What about negative numbers?

We use 2-complement representation for negative values

VHDL basics cont.

2-complement

To represent a negative number in 2-complement form we take the positive number with the same magnitude, invert all bits and add one to LSB

Example: Write the decimal number -10 in binary 2-complement form using 8 bits

You need to decide on the number of bits

1. Convert the positive number (10) to binary base

$$(10)_{10} = 1 \cdot 2^3 + 1 \cdot 2^1 = (00001010)_2$$

2. Invert all bits

$$00001010 \rightarrow 11110101$$

3. Add 1 to LSB

```
        1
 11110101
+00000001
 11110110
```

We´re done!

To find the magnitute of a negative number we do the exact same thing –(-x)=x

VHDL basics cont.

What if our number has a value that can´t be representated with the choosen number of bits?

We get overflow!

Example: Use 8 bit words and add the decimal numbers 70 and 80 in binary form

(The largest signed number we can represent with 8 bits is 127, but 70 + 80 = 150)

$(70)_{10} \rightarrow (01000110)_2$

$(80)_{10} \rightarrow (01010000)_2$

```
  1
 01000110
+01010000
 10010110
```

2-complement

```
10010110 → 01101001
         +        1
          01101010 = 106
```

The number is -106

One in MSB indicates a negative number. But what number?

The result has been corrupted. The phenomenon is called **wrap around**

VHDL basics cont.

Wrap around

Wrap around means that if the value of a binary number, positive or negative, won´t fit into the used number of bits then the result will change sign and be totally wrong although there is a system in the incorrectness

In many cases this is unacceptable

There are two remedies

1. Increase the number of bits to handle larger values

   This is a non-destructive solution but increases the amount of hardware

2. Saturate the result

   If the value is to large, saturate, set the value to the largest value we can represent with the given number of bits

   This is a destructive solution since the result vill be corrupted. This solution will also increase the amount of hardware since we need hardware to test for overflow

VHDL basics cont.

Wrap around or saturate?

Wrap around means that we just discard the bits that don´t fit within the given number of bits. This doesn´t take any extra logic, but we might check to indicate wrap around

Saturation means that we have to introduce extra logic to investigate if overflow have occured

The application decides what route to take

VHDL basics cont.

Value holders

Ports

Connections to the external world or to other components

Constants

Symbolic names instead of numbers to simplify programming

Generics

Values that are used to specify the instantiation of a subprogram, for example to give the number of bits in a vector without the need to rewrite the code

Signals

Internal signal connections visible in the whole architecture see them as wires between blocks, if a signal is assigned a value inside a process this value will not be be updated until we leave the process

Variables

Internal signals that are local to a process, they are not accessable outside of the process, the values are immediately updated

VHDL basics cont.

Value assignment

We assign values to our value holders.
The syntax differs somewhat between the types

Generics, constants and variables are assigned values using :=

```
GENERIC (width:INTEGER:=16);

CONSTANT size_const:INTEGER:=8;

VARIABLE index_variable:INTEGER;
……
  index_variable:=3;
```

The variable and the signal could be given a value at declaration but this value will only transfer to simulation, not to synthesis so don´t do that. Assign values after declaration instead using some kind of reset phase

Ports and signals are assigned values using <=

```
SIGNAL error_signal:STD_LOGIC;
……
  error_signal <= '1';

PORT(……
     outport:OUT STD_LOGIC);
……
  outport <= error_signal;
```

VHDL basics cont.

Data types

Scalar types

Type declarations

```
TYPE ubyte IS RANGE 0 TO 255;
TYPE nibble IS RANGE -8 TO 7;
```

Placed in the architecture before the first BEGIN

Signal declarations

```
SIGNAL xint_signal:INTEGER;
SIGNAL xubyte1_signal:ubyte;
SIGNAL xubyte2_signal:ubyte;
SIGNAL xnibble_signal:nibble;
```

Predefined type with range $-2^{31} – (2^{31}-1)$ = = -2,147,483,648 – 2,147,483,647

Our declared types

What about assignments? The integer type could represent all the values in the ubyte range so

```
xint_signal <= xubyte1_signal;
```

would be OK, wouldn´t it?

No! They are two different types and VHDL is strictly typed. To go between types we need conversion functions

```
xubyte1_signal <= xubyte2_signal;
```

is OK though. They are of the same type

VHDL basics cont.

```
TYPE ubyte IS RANGE 0 TO 255;
TYPE nibble IS RANGE -8 TO 7;
```

Scalar types cont.

Why not just use INTEGER as in software?

Our VHDL code will be synthesized to hardware and this hardware must be able to handle all possible values of a signal.

In the hardware our signals are represented by binary bits.

An integer will have to be represented by 32 bits to cover all possible values and that would have to be the width of our signal paths then.

If we only use a fraction of the integer range that would be a waste of hardware.

Even worse if the signal is to be stored along the signal path. In every place where we want to store the signal we would have to include 32 flip-flops to do this.

We can restrict the integer range though.

The `ubyte` type would take 8 bits and the `nibble` type only 4 bits.

A word of warning. The simulator will give an error if we try to use values outside of the range of the type but the hardware won´t

VHDL basics cont.

Scalar types cont.

Subtypes

We can declare a subtype if we like.

A subtype is a new type that only covers a part of the range of another type

```
TYPE ubyte IS RANGE 0 TO 255;
SUBTYPE subunibble IS ubyte RANGE 0 TO 15;
SIGNAL xubyte_signal:ubyte;
SIGNAL subxunibble_signal:subunibble;
```

It is OK to move values between signals of the type and the subtype as opposed to between types

```
xbyte_signal <= subxunibble_signal;
subxunibble_signal <= xbyte _signal;
```

A word of warning. The subtype `subunibble` can´t take all the values of the `ubyte` type so the last assignment is dangerous

VHDL basics cont.

Scalar types cont.

Enumeration types

Symbolic names for the values of a signal

```
TYPE weekday IS (sun,mon,tue,wed,thu,fri,sat);
TYPE washing_machine IS (pre_wash,wash,rinse,dry);
```

Typically used to name the states in a state machine,
like the phases for the program of a traffic light (green, yellow, red)

Some useful preedefined enumeration types

```
TYPE boolean IS (false,true);
```
← Useful in conditional code

```
TYPE bit IS ('0','1');
```
← Logical values. Not recommended use std_logic

The '-signs indicate that these values are actually characters

VHDL basics cont.

Scalar types cont.

Enumeration types cont.

The rest of the line is a comment

Standard logic unsigned

```
TYPE std_ulogic IS ('U',  -- uninitialized
                    'X',  -- forcing unknown
                    '0',  -- forcing zero
                    '1',  -- forcing one
                    'Z',  -- high impedance
                    'W',  -- weak unknown
                    'L',  -- weak zero
                    'H',  -- weak one
                    '-'); -- don´t care
```

Only relevant in simulation

Only relevant at compilation

Standard logic (`std_logic`) is a type that is formed from `std_ulogic`

Signed or unsigned has no meaning for single bits

Standard logic is our recommended type for all binary signals

VHDL basics cont.

Scalar types cont.

Fixed and floating point types

VHDL can use fixed and floating point values
but they are of limited use for synthesis

When we get to filter implementation we will see that
fixed point representation have some relevance

In many cases using floating point values would increase the accuracy in our calculations but the required amount of hardware will increase drastically

VHDL basics cont.

Scalar types cont.

Physical types

These are used to represent real-world physical quantities, such as length, mass, time and current

The only physical unit of use to us is time. It has no meaning for synthesis but is very useful for giving times in test benches for simulation

```
TYPE time IS RANGE implementation defined
  UNITS
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min
  END UNITS;
```

VHDL basics cont.

Scalar type attributes

Scalar types have some attributes that could be useful

`typename'LEFT` – the first (leftmost) value in `typename`

`typename'RIGHT` – the last (rightmost) value in `typename`

`typename'LOW` – the smallest value in `typename`

`typename'HIGH` – the largest value in `typename`

These attributes are very useful when we design subprograms that use generic signals

VHDL basics cont.

Scalar operators

Scalar signals have a number of operators but not all of them apply to all types of scalars

Logical operators

Apply to bit and std_logic

```
NOT
AND
OR
NAND
NOR
XOR
XNOR
```

Observe that in VHDL all logical operators have the same presidence.
Therefore: be generous with parenthesis

VHDL basics cont.

Scalar operators

Arithmetic operators

The operators apply to all numeric values exept where noted

- `*` multiplication
- `/` division
- `mod` modulo (apply to integer)
- `rem` reminder (apply to integer)
- `-` negation
- `+` addition
- `-` subtraction

The operators are given in order of presidence with multiplication having the highest presidence

Modulo and reminder has the same presidence and the same goes for addition and subtraction

VHDL basics cont.

```
TYPE bit IS ('0','1');
```

```
TYPE std_ulogic IS ('U',  -- uninitialized
                    'X',  -- forcing unknown
                    '0',  -- forcing zero
                    '1',  -- forcing one
                    'Z',  -- high impedance
                    'W',  -- weak unknown
                    'L',  -- weak zero
                    'H',  -- weak one
                    '-'); -- don´t care
```

Scalar operators

Operators for comparision

These operators apply to all scalar operators

- `=` equality
- `/=` unequality
- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to

Equality and unequality have higher presidence than the others

When it comes to enumerated types a value to the left in the sequence is smaller than a value to the right

For bits `0` is smaller then `1`

For std_logic `X` is less than `W`

VHDL basics cont.

```
TYPE ubyte IS RANGE 0 TO 255;
```

Composite data types

Arrays, vectors

Since we have to form our multi-value signals from binary values in our hardware implementation, binary vectors are our basic form for signal description besides single binary bits

8 bits

```
TYPE byte IS ARRAY (0 TO 7) OF std_logic;
```

Observe that this is not the same as the earlier type definition of `ubyte`. Both can take on 256 different values but the types are not interchangable.

Indexes can have any range, they don´t have to start with zero (0) or one (1)
Increasing indexes use `TO`, descending indexes use `DOWNTO`

We can address individual bits and vector ranges in the array using indexes

```
SIGNAL xbit_signal:std_logic;
SIGNAL xbyte_signal:byte;
SIGNAL xnibble_signal:std_logic(0 TO 3);
……
   xbit_signal <= xbyte_signal(3);
   xnibble_signal <= xbyte_signal(2 TO 5);
```

Single bit

Subarray

VHDL basics cont.

Arrays, vectors cont

In many cases our vector represents a binary value,
In these cases it is more natural to use descending indexes

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF std_logic;
```

This type definition is in most cases not necessary since we have predefined vector types for bits and std_logic

VHDL basics cont.

Arrays, vectors cont

Predefined types

```
SIGNAL bitword_signal:bit_vector(15 DOWNTO 0);
SIGNAL stdbyte_signal:std_logic_vector(7 DOWNTO 0);
SIGNAL std_signal:std_logic_vector(1 TO 12);
```

For these pre-defined types the indexes must be natural numbers, that is positive or zero (0)

When we write a value to a std_logic_vector we treat the value as a string

```
stdbyte_signal <= ”00110110”;
```
← Double quotations indicate string

It is recommended to **avoid using bit_vector** and **always use std_logic_vectors**

Use descending indexes if there is no good reason to do otherwise

VHDL basics cont.

Arrays, vectors cont

We can also read or write parts of a vector

```
stdbyte_signal(4 DOWNTO 2) <= ”110”;
```

or use concatination (&) to manipulate our vectors

A new line within the code line is fully acceptable

```
stdbyte_signal <=
      stdbyte_signal(4 DOWNTO 2) & ”00110”;
```

The number of bits on the two sides of the assingment sign must of course match

VHDL basics cont.

Unconstrained arrays

So far we have seen constrained arrays, that is arrays where the size is declared in the type declaration

Sometimes it is practical to just declare the type of the values in the type declaration and leave the size declaration to the instantiation

We have an unconstrained array type

Just a placeholder

```
TYPE uncon IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

Defines the range for the allowed indexes. These can be

`POSITIVE` – positive integer values

`NATURAL` – natural integer values, positive values and zero (0)

`INTEGER` – integer values, both positive and negative

When we declare a signal of the unconstrained array type we have to set the size by setting the index range of the declared signal

```
SIGNAL uncon_signal:uncon(10 TO 53);
```

VHDL basics cont.

Multidimensional arrays

An array can have more than one dimension

```
TYPE multiarray IS ARRAY (0 TO 9,0 TO 4) OF STD_LOGIC;
```

We address the individual elements using two indexes

```
SIGNAL ma_signal:multiarray;
……
  ma_signal(5,3) <= '1';
```

VHDL basics cont.

Arrays of arrays

In some cases it is more practical to be able to address the rows of the multi dimensional array and not the individual elements.
This could be the case when we create a memory for byte sized data.
In these cases it is better to define a array of vectors

```
TYPE memory IS ARRAY (0 TO 9) OF
                STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Here we address the rows of the array, that is the bytes and not the individual bits

```
SIGNAL mem_signal:memory;
……
  mem_signal(5) <= ”00110110”;
```

In this case we have no simple way of addressing the individual elements.

To do this we have to first read the row vector, address the individual bit in the row and then write the row vector back to its place

VHDL basics cont.

Array type attributes

Array types have some attributes that could be useful

`typename'LEFT(N)` – left bound of index range of dimension `N` of `typename`

`typename'RIGHT(N)` – right bound of index range of dimension `N` of `typename`

`typename'LOW(N)` – lower bound of index range of dimension `N` of `typename`

`typename'HIGH(N)` – upper bound of index range of dimension `N` of `typename`

`typename'RANGE(N)` – index range of dimension `N` of `typename`

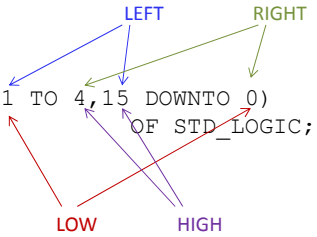
For one dimensional arrays `N` can be left out

This is very useful for unbound and generic vectors where we don´t know the index range

VHDL basics cont.

Array type attributes cont.

Example




```
TYPE arraytype IS ARRAY (1 TO 4,15 DOWNTO 0)
                         OF STD_LOGIC;

arraytype'LEFT(1) = 1
arraytype'LEFT(2) = 15

arraytype'RIGHT(1) = 4
arraytype'RIGHT(2) = 0

arraytype'LOW(1) = 1
arraytype'LOW(2) = 0

arraytype'HIGH(1) = 4
arraytype'HIGH(2)= 15
```

VHDL basics cont.

Array type attributes cont.

Another example

```
TYPE arraytype IS ARRAY (7 DOWNTO 0)
                              OF STD_LOGIC;
SIGNAL x_array_signal:arraytype;
…
VARIABLE count_variable:INTEGER RANGE 0
                           TO x_array'HIGH+1;
count_variable:=0;
FOR index IN x_array'RANGE LOOP
   IF x_array_signal(index)='1' THEN
      count_variable:=
            count_variable+1;
   END IF;
END LOOP;
```

We have eight bits, `HIGH` is seven

`RANGE` is `7 DOWNTO 0`

The `index` variable does not have to be declared

The example counts the number of ones in `x_array` without knowing the size of `x_array`, it is generic

Observe that we must use a variable not a signal for the counter since the loop must be placed within a process and we need the updated values of `count_variable` imediately for the next loop round

VHDL basics cont.

Array operators

Array signals have a number of operators but not all of them apply to all types of arrays

Logical operators

Apply to boolean, bit and std_logic vectors

```
NOT
AND
OR
NAND
NOR
XOR
XNOR
```

Observe that in VHDL all logical operators have the same presidence.
Therefore: be generous with parenthesis

For std_logic arrays the two arrays involved need to be of the same length and the operators work bit by bit

For boolean and bit arrays we have a somewhat broader span of applications for the operators but we won´t get into those

VHDL basics cont.

Shift operators

Apply to boolean and bit vectors, **not** std_logic vectors

```
SLL  shift left logically
SRL  shift right logically
ROL  rotate left
ROR  rotate right
SLA  shift left aritmethic
SRA  shift right aritmethic
```

The operators need to be complemented by the number of bits to shift or rotate, `y <= SLL x 2`

Logical and aritmethic shifts to the left give the same result and fill the empty bits with zeros

Logical shifts to the right fill the empty bits with zeros while aritmethic shifts to the right fill the empty bits with sign bits

Since the operators only work on boolean and bit arrays we need to use conversion functions to use them with std_logic vectors, see the following exemple

VHDL basics cont.

Shift operators cont.

Example: Shift the std_logic vector `x_signal` three bits to the right logically

First we need to convert the std_logic vector to a bit vector to do the shift and then we need to convert the shifted bit vector back to std_logic_vector to get our original std_logic vector again

```
SIGNAL x_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL y_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bitvector1_signal:BIT_VECTOR(7 DOWNTO 0);
SIGNAL bitvector2_signal:BIT_VECTOR(7 DOWNTO 0);
……
  bitvector1_signal <= TO_BITVECTOR(y_signal);
  bitvector2_signal <= bitvector1_signal SRL 3;
  y_signal <= TO_STDLOGICVECTOR(bitvector2_signal);
```

Conversion functions

VHDL basics cont.

```
bitvector1 <= TO_BITVECTOR(b);
  bitvector2 <= bitvector1 SRL 3;
  y <= TO_STDLOGICVECTOR(bitvector2);
```

Example cont.

You might think that one bit vector would be enough so that we could write

```
SIGNAL x_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL y_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bitvector_signal:BIT_VECTOR(7 DOWNTO 0);
……
  bitvector_signal <= TO_BITVECTOR(x_signal);
  bitvector_signal <= bitvector_signal SRL 3;
  y_signal <= TO_STDLOGICVECTOR(bitvector_signal);
```

But since the code isn´t written inside a process our statements are concurrent, this means that the first two lines of code both tries to write values to `bitvector` at the same time which obviously won´t work.

We could skip the bit vectors all together though and write the whole function in one single line

```
Y_signal <=TO_STDLOGICVECTOR((TO_BITVECTOR(x_signal)
                                                SRL 3));
```

VHDL basics cont.

Shift and rotate operators

There are no explicite functions to shift and rotate std_logic_vectors, we have to convert to bit vector or implement the functions ourselves or use the standard logic subtypes SIGNED or UNSIGNED for which there are such functions. We will get back to these.

The operations can be logic or arithmetic shifts or rotations and they can be to the left or to the right.

The shift and rotate operations can be done using vector manipulation

Example: logical shift two steps to the left

```
SIGNAL shift_v_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
……
shift_v_signal<=shift_v_signal(5 DOWNTO 0)&”00”;
```

Example: arithmetic shift one steps to the right

Sign extention

```
shift_v_signal<=shift_v_signal(7)&
                shift_v_signal(7 DOWNTO 1);
```

Example: rotate three steps to the right

```
shift_v_signal<=shift_v_signal(2 DOWNTO 0)&
                shift_v_signal(7 DOWNTO 3);
```

VHDL basics cont.

Shift and rotate operators cont.

This is a suitable place to use vector attributes so the same code could be used for vectors of different lengths

Don´t try to write functions where variables in your code change the number of shifting or rotating steps since this will generate an awful lot of logic

Let´s take the last example again but define the vector size using a constant

VHDL basics cont.

Shift and rotate operators cont.

Example: logical shift two steps to the left

```
CONSTANT WIDTH:INTEGER:=8;
SIGNAL shift_v_signal:STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
……
shift_v_signal<=shift_v_signal(WIDTH-3 DOWNTO 0)&"00";
```

Example: arithmetic shift one steps to the right

```
shift_v_signal<=shift_v_signal(WIDTH-1)&
                shift_v_signal(WIDTH-1 DOWNTO 1);
```

Example: rotate three steps to the right

```
shift_v_signal<=shift_v_signal(2 DOWNTO 0)&
                  shift_v_signal(WIDTH-1 DOWNTO 3);
```

Observe that for synthesis the number of shift steps must be fixed and not variable in an instantiation since the hardware is a fixed structure

VHDL basics cont.

Array operators

Arithmetic and comparison operators

These operators require the inclusion of a numberic package called `numeric_std` to handle std_logic vectors

Most arithmetic operations can give different results depending on if the operands are interpreted as signed or unsigned vectors.

Is 1100 (12 or -4) larger or smaller than 0011 (3)?
It depends on signed or unsigned interpretation

Because of this we need to specify how to interpret the vectors

We do this by using the subtypes **SIGNED** and **UNSIGNED** to std_logic_vector when we perform the arithmetic operations and then we go back to std_logic again. The subtypes are declared in `numeric_std`

As mentioned before there are defined shift and rotate operations for these subtypes, together with arithmetic operations

VHDL basics cont.

Array operators

Arithmetic and comparison operators cont.

Example

```
SIGNAL a:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL b:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL y:STD_LOGIC_VECTOR(7 DOWNTO 0);
……
y <= STD_LOGIC_VECTOR(SIGNED(a) + SIGNED(b));
```

Interpret as signed

Go back to interpretation as std_logic_vector

Observe that since we only use subtypes and don´t do type conversion we do not use the conversion function `TO_STDLOGICVECTOR` but the type declaration function `STD_LOGIC_VECTOR`

VHDL basics cont.

Array operators

Arithmetic and comparison operators cont

Earlier it was common to use the types `std_logic_signed` and `std_logic_unsigned` to handle these situations.

In recent years it has been recommended to use `std_logic_vector` and the subtypes `signed` and `unsigned` instead since the earlier types could lead to confusion in some instances

Follow the new recommendation and use SIGNED and UNSIGNED!

Don´t use std_logic_signed and std_logic_unsigned!

Be observant on this if you use example code from somewhere on the net or others

VHDL basics cont.

Array operators

Arithmetic operators

`*`	multiplication
`/`	division
`mod`	modulo
`rem`	reminder
`-`	negation
`+`	addition
`-`	subtraction

These operations can be performed between two vectors or between a vector and an integer, the operands can be in any order.
If the vector is to be interpreted as UNSIGNED then the integer value is limited to natural values.
The same operator can be used on different types of objects because there are several versions of the operators and the operator types decide which version that will be used.
This is called **overloading**.

VHDL basics cont.

Array operators

Arithmetic operators cont.

In all operations exept multiplication the result from operations on vectors will be a vector of the same size as the largest input vector.

When one of the operands is an integer number and the other a vector the operation will give a vector of the same size as the vector.

In multiplication between two vectors the size of the result will be the sum of the sizes of the two input vectors.

If one of the operands in the multiplication is a integer and the other a vector then the size of the result vector will be twice the size of the input vector

These multiplications will not perform the left shift needed for fractional numbers. You need to do this yourself

VHDL basics cont.

Array operators

Arithmetic operators cont.

Examples

```
SIGNAL a:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL b:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL c:STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL y:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL z:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL q:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL w:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL t:STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL u:STD_LOGIC_VECTOR(15 DOWNTO 0);
……
y <= STD_LOGIC_VECTOR(SIGNED(a)+UNSIGNED(b));
z <= STD_LOGIC_VECTOR(SIGNED(a)+SIGNED(c));
q <= STD_LOGIC_VECTOR(SIGNED(a)-5);
w <= STD_LOGIC_VECTOR(SIGNED(a)/SIGNED(b));
t <= STD_LOGIC_VECTOR(SIGNED(a)*SIGNED(b));
u <= STD_LOGIC_VECTOR(SIGNED(a)*9);
```

VHDL basics cont.

Array operators

Operators for comparision

=	equality
/=	unequality
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The operators apply to one dimensional arrays where the elements can be of any discrete type. The operators will return a value of type boolean.

The arrays don´t need to be of the same size as long as the elements are of the same type.

The equality function can only be true if the two arrays are of the same size and all the element values are the same

VHDL basics cont.

Array operators

Operators for comparision cont.

The less and greater functions are performed element by element extending the shortest array

These comparision operations can also be performed between a vector and a integer, the operands can be in any order.
If the vector is to be interpreted as UNSIGNED then the integer value is limited to natural values.

VHDL basics cont.

Array operators

Operators for comparison cont.

Examples

```
SIGNAL a_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL b_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL c_signal:STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL x_signal:BOOLEAN;
SIGNAL y_signal:BOOLEAN;
SIGNAL z_signal:BOOLEAN;
……
x_signal<=STD_LOGIC_VECTOR(SIGNED(a_signal)>
                                UNSIGNED(b_signal));
y_signal<=STD_LOGIC_VECTOR(SIGNED(a_signal)/=
                                  SIGNED(c_signal));
z_signal<=STD_LOGIC_VECTOR(SIGNED(a_signal)<=5);
```

VHDL basics cont.

Array operators

Concatination

We can create longer arrays by concatination (&) of shorter arrays or elements

Some examples

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL b_signal:STD_LOGIC;
SIGNAL c_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL d_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL e_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL f_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL g_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL h_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);

e_signal<=c_signal & d_signal;

f_signal<="000" & '1' & "0000";

g_signal<=e_signal (1 DOWNTO 0) &
                    e_signal (7 DOWNTO 2);

h_signal <= c_signal & "000" & a_signal;
```

Two vectors

Constants

Rotate operation

Vector, constant and scalar

VHDL basics cont.

Basic VHDL structures

Entity

The external, visual part of aVHDL design is the entity that defines the connections (ports) in and out of the design.

The entity can also contain generics, attributes that are used to control the design, for example the width of vectors.

The entity has the following structure

```
ENTITY entity_name IS
  [GENERIC (generic_name:data_type[:=value]);]
  PORT(port_name1:connection_type datatype;
       port_name2:connection_type datatype);
END entity_name;
```

Observe where the semicolon (;) separators are placed

VHDL basics cont.

Basic VHDL structures

Comments

Comments can be placed in all parts of the code if you follow some rules.

- Comments start with the sign −− and continues to the end of the line
- If a line starts with −− then the whole line is a comment
- You can not have comments within lines, with code before and after the comment
- You can not comment out more then one line of code at once, multiple line comments have to be commented out line by line. QuestaSim have tools to comment out several marked lines at once

Instead of commenting out longer code sections it might be easier to temporarily cut the code section out and store it by paisting it into another file for the time being and then paiste it back in later on when you need it again

VHDL basics cont.

Basic VHDL structures

Generics

The generic can be of any datatype and since it only modifies the design instantiation it doesn´t have to be a synthesizable datatype.

The generic does not have to be given a value in the entity. That might come later when we instantiate our design as a subdesign (component) in a larger design.

In that case we create a generic design that can adapt to the application.

We create a generic component or function

VHDL basics cont.

Basic VHDL structures

Ports

The ports are our connections in and out of the design

If they are to be connected to the outer world they have to be of synthesizable types, preferably std_logic or std_logic_vector

We can use other datatypes but only if the design is supposed to be used as part of a larger design where the ports will be internal connections to other parts of the design that use the same datatype and not connected to the outside world. In this cases INTEGER, SIGNED and UNSIGNED can be practical

We can have four different connection types for the ports

`IN`	data path directed into the design
`OUT`	data path directed out of the design
`INOUT`	bidirectional data path
`BUFFER`	a readable output

VHDL basics cont.

Basic VHDL structures

Ports cont.

The `IN` port can only be read, we can not write to it

The `OUT` port can only be written, we can not read from it

The `INOUT` port can be both read and written

The `BUFFER` port is a buffered outport where the value before the buffer can be read

Not all pins on a FPGA can be used as bidirectional or buffered pins so I would recommend that you try and use `IN` or `OUT` ports exclusively

Do not define a port as `INOUT` just to be able to read it. Reserve this for truly bidirectional ports

In most cases we can avoid `INOUT` and `BUFFER` ports by using internal signals that can be both read and written and then transfer the value to the `OUT` ports. This will not give extra hardware. Se example below

VHDL basics cont.

Basic VHDL structures

Ports cont.

Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY inout_port IS
   PORT(reset:IN STD_LOGIC;
        clk:IN STD_LOGIC;
        a:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END inout_port;
```

VHDL basics cont.

Basic VHDL structures

Ports cont.

Example cont.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY inout_port IS
   PORT(reset:IN STD_LOGIC;
        clk:IN STD_LOGIC;
        a:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END inout_port;
```

```
ARCHITECTURE arch_inout_port OF inout_port IS
   SIGNAL y_signal:SIGNED(7 DOWNTO 0);
BEGIN
   clocked_add_proc:
   PROCESS(reset,clk)
   BEGIN
      IF reset = '1' THEN
         y_signal <= (OTHERS=>'0');
      ELSIF (rising_edge(clk)) THEN
         y_signal <= y_signal + SIGNED(a);
      END IF;
   END PROCESS clocked_add_proc;
   y <= STD_LOGIC_VECTOR(y_signal);
END arch_inout_port;
```

Readable temporary signal

Internal signal can be of type SIGNED

Asynchronous reset signal

Set all bits to zero

Positive clock edge

Treat a as SIGNED

Transfer internal signal to output and convert to STD_LOGIC_VECTOR

VHDL basics cont.

Basic VHDL structures

Ports cont.

Example cont.

```
ARCHITECTURE arch_inout_port OF inout_port IS
   SIGNAL y_signal:SIGNED(7 DOWNTO 0);
BEGIN
   clocked_add:PROCESS(reset,clk)
   BEGIN
      IF reset = '1' THEN
         y_signal <= (OTHERS=>'0');
      ELSIF clk'EVENT AND clk='1' THEN
         y_signal <= y_signal + SIGNED(a);
      END IF;
   END PROCESS clocked_add;
   y <= STD_LOGIC_VECTOR(y_signal);
END arch_inout_port;
```

The output assignment can be placed inside the process

```
      END IF;
      y <= STD_LOGIC_VECTOR(y_signal);
   END PROCESS clocked_add;
END arch_inout_port;
```

but then it will have to be registered (clocked) to remember the value from one process triggering to the next, that is we need registers for both `y_signal` and `y`. This will require extra flip-flops.

We will also delay `y` by one clock cycle if we put `y` within the process.

If we use a variable instead of a signal we remove the delay but can not get out of the process.

When we place the assignment of `y` outside of the process it will only be a set of wires between the signal and the port and no extra registers are required.

VHDL basics cont.

Basic VHDL structures

Architecture

The architecture describes the internals of our design

It has the following structure

```
ARCHITECTURE architecture_name OF entity_name IS
[Constant declarations]
[Type definitions]
[Signal declarations]
[Component declarations]
BEGIN
   parallel (concurrent) code
   sequential code (processes)
END architecture_name;
```

VHDL basics cont.

```
ARCHITECTURE architecture_name OF entity_name IS
[Constant declarations]
[Type definitions]
[Signal declarations]
[Component declarations]
BEGIN
   parallel (concurrent) code
   sequential code (processes)
END architecture_name;
```

Basic VHDL structures

Architecture cont.

Constants

The constant declarations are a way of giving symbolic names to objects used in the code. It will also make it possible to make the change in just one single place in the code if we want to change an object.

The declaration has the form

```
CONSTANT CONSTANT_NAME:constant_type:=value;
```

The constant can be of have any type since it will not necessarily have to be synthesizable

Type definitions

Declaration of our own data types. We´ve talked about this before

Components

Subdesigns that we use to build more complex designs

We will get back to this later on

VHDL basics cont.

Basic VHDL structures

Architecture Body

The architectural body contains our design code describing the functionality.

All statements in the body are concurrent, they are asserted in parallel, at the same time. There is no sequence between events.

The exception to this is the process where the internal code of the process is sequential.

But the whole process is a concurrent statement that is executed in parallel with the rest of the code.

The process reads input values when the process is entered and new values are written to outputs of the process when the process is exited.

The architectural code can be written in two different ways

- **Behavoiral code** that describes the functionality of the design
- **Structural code** that describes the design as blocks (components) interconnected by signals

In reality the two approaches are often combined

VHDL basics cont.

Basic VHDL structures

Signal attributes

A signal can have a number of attributes.
We will only use the EVENT attribute.

We´ll get back to this when we talk about clocked processes

Signal assignment

As we have seen earlier signals are assigned values using the symbol <=

The same goes for output ports

Variables, constants and generics are assigned values using the symbol :=

Variables can only exist in sequential code, that is in a process,
and they are local to the process

The other signal types can exist in both concurrant and sequential code
and are visible in the entire architecture

Constants can be local to processes

VHDL basics cont.

Basic VHDL structures

Conditional signal assignment

In concurrent code we have two structures for conditional signal assigment.

The first one, the **WHEN** statement, is simular to what we know as an
IF statement from software programming

Example

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  y_signal <= '1' WHEN (a_signal='0') ELSE
              '0';
```

The parantheses are not necessary but increase the readability

The signal must be assigned a value under all conditions which means that the else clause is necessary

VHDL basics cont.

Basic VHDL structures

WHEN statement cont.

The statement could be expanded

Example

```
SIGNAL y_signal:STD_LOGIC;
SIGNAL a_signal:STD_LOGIC_VECTOR(1 DOWNTO 0);
……
  y_signal <= '1' WHEN (a_signal="00") ELSE
              '1' WHEN (a_signal="01") ELSE
              '0';
```

This could also be rewritten as

```
  y_signal <= '1' WHEN ((a_signal="00") OR
                        (a_signal="01")) ELSE
              '0';
```

I think this is less readable though

VHDL basics cont.

Basic VHDL structures

WHEN statement cont.

Observe that there is nothing to say that the selection condition must be of the same type in all clauses

Example

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
SIGNAL b_signal:STD_LOGIC_VECTOR(1 DOWNTO 0);
……
  y_signal <= '1' WHEN (a_signal='1') ELSE
              '1' WHEN (b_signal="01") ELSE
              '0';
```

This kind of coding is very confusing and should not be used

VHDL basics cont.

```
SIGNAL a_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  y_signal<='1' WHEN (a_signal='0') ELSE
            '0';
```

Basic VHDL structures

WITH statement

The other concurrent conditional signal assignment is the **WITH** statement.
It has simularities with the CASE statement in software programming

Example

We repeat our first WHEN exampl using the WITH statemente

```
SIGNAL x_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  WITH x_signal SELECT
     y_signal <= '1' WHEN '0',
                 '0' WHEN '1';
```

condition signal

This code won´t compile though it is formally correct and we have covered both the high and the low signal values.

Why?

VHDL basics cont.

```
SIGNAL x_signal:STD_LOGIC;
SIGNAL y_signal:STD_LOGIC;
……
  WITH x_signal SELECT
     y_signal <= '1' WHEN '0',
                 '0' WHEN '1';
```

Basic VHDL structures

WITH statement cont.

In the statement all possible values of the selector, here `x_signal`, has to be covered and the std_logic variable `x_signal` has nine (9) possible values (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `–`) that must be handled

VHDL basics cont.

Basic VHDL structures

WITH statement cont.

We rewrite the code

```
WITH x_signal SELECT
   y_signal <= '1' WHEN '0',
               '0' WHEN OTHERS;
```

The OTHERS clause covers all cases when `x/='0'`

Since the synthesized code only has values `0` and `1` (and `Z`, but not as an input value), this covers all cases.

The code gets somewhat clearer if we rewrite it as

```
WITH x SELECT
   y_signal <= '1' WHEN '0',
               '0' WHEN '1',
               '0' WHEN OTHERS;
```

The synthisized result will be the same though

VHDL basics cont.

```
SIGNAL y_signal:STD_LOGIC;
SIGNAL a_signal:STD_LOGIC_VECTOR(1 DOWNTO 0)
……
  y_signal <= '1' WHEN (a_signal="00") ELSE
              '1' WHEN (a_signal="01") ELSE
              '0';
```

Basic VHDL structures

WITH statement cont.

Let´s rewrite the other WHEN statement, the one with the vector

```
WITH a_signal SELECT
   y_signal <= '1' WHEN "00",
               '1' WHEN "01",
               '0' WHEN OTHERS;
```

In the WITH statement we can only have one selecter so we can not have the mixed condition of the scalar `a` and vector `b` that we had in the WITH case.

The two cases that give the same result could be combined

```
WITH a_signal SELECT
   y_signal <= '1' WHEN "00" | "01",
               '0' WHEN OTHERS;
```

OR statement

For enumerated types we can also give ranges using TO or DOWNTO

We´ll see this for the CASE statement later on

VHDL basics cont.

Basic VHDL structures

GENERATE statement

There is one more concurrent assignment statement, the GENERATE statement.

We will get back to this when we talk about components

VHDL basics cont.

Basic VHDL structures

Sequential code, processes

Sequential code is code where the statements are evaluated line by line in a sequence and not all statement at the same time, in parallel, as in concurrent code.

Sequential code is written within **processes**

VHDL basics cont.

Basic VHDL structures

The process structure

The process label is optional but it might encance the readability of the code

```
[process_name:]
PROCESS[(sensitivity list)]
constant declarations
variable declarations
BEGIN
   [WAIT statement]
   sequential statements
END PROCESS [process_name];
```

The process must have either a sensitivity list or a WAIT statement, but it can not have both

A process won´t execute until it is triggered and this is where the sensitivity list or WAIT statement comes into play

VHDL basics cont.

Basic VHDL structures

The sensitivety list

The sensitivity list is a list of the signals that trigger the process, that is the signals that make the process execute when any of them change value.

Examples

`PROCESS(a)` ← Process triggered by `a`

`compare_proc:PROCESS(a,b)` ← Process named `compare_proc` triggered by signals `a` and/or `b`

The process will not execute if none of the signals in the sensitivity list change value

Make sure that all signals that should make the process execute are in the sensitivity list but don´t add any other signals

Synthisized hardware are somewhat more forgiving to this than simulators that are very strict

VHDL basics cont.

Basic VHDL structures

The sensitivety list cont.

Example

```
AND_proc:PROCESS(a)
BEGIN
   y <= a AND b;
END process AND_proc;
```

The process should only trigger when `a` changes value but not when `b` changes value and this is true in simulation

In synthesis though the result will be an ordinary AND circuit that will trigger on both `a` and `b`

VHDL basics cont.

Basic VHDL structures

The WAIT statement

The WAIT statement has the same function as the sensitivity list but can give some more options.

The WAIT statement can be placed anywhere in the process and there can be more than one WAIT statement in a process.

The process will execute up until the WAIT statement and then wait for the trigging condition.

The statement has three different forms

- WAIT ON has the same function as the sensitivety list, it is triggered when the signal changes value
- WAIT UNTIL is triggered when some condition is fulfilled
- WAIT FOR waits for a specified time

The last one can not be synthesized but is useful in simulation. Notably in test benches

VHDL basics cont.

Basic VHDL structures

The WAIT statement cont.

Examples

```
WAIT ON a;
```
Waits until a changes value

```
WAIT ON a,b;
```
Waits until a and/or b change value(s)

```
WAIT UNTIL (a='1');
```
Waits until a changes value to one (1)

```
WAIT FOR 10ns;
```
Waits for 10 ns Only used in simulation!

I find WAIT statements a bit tricky to handle and would recommend using sensitivity lists

VHDL basics cont.

Basic VHDL structures

The WAIT statement cont.

WAIT FOR example

We can use a WAIT FOR statement to genarate a simulation clock within a test bench

```
clock_proc:PROCESS
BEGIN
   WAIT FOR 50 ns;
   clk_tb_signal<=NOT(clk_tb_signal);
END PROCESS clock_proc;
```

For this to work `clk_tb_signal` must have a start value so this is a rare moment when it is recommended to give the signal an initial value at declaration

```
SIGNAL clk_tb_signal:STD_LOGIC:='0';
```

This is OK since it is only used in simulation

VHDL basics cont.

Basic VHDL structures

Conditional signal assignment in sequential code

The WHEN and WITH statements used in concurrent code can not be used in sequential code.

We have a couple of replacements

VHDL basics cont.

Basic VHDL structures

IF statement

The IF statement has simularities to the WHEN statement

The structure is

```
[IF_label:]
IF condition THEN
   sequential code;
[ELSIF condition THEN
   sequential code;]
[ELSE
   sequential code;]
END IF [IF_label];
```

The IF label is optional but it might encance the readability of the code

VHDL basics cont.

Basic VHDL structures

IF statement cont.

Examples

```
IF (a='0') THEN
   y <= '1';
END IF;
```

Since the behavior at all values of a is not declared a memory element must be used

```
IF (a='0') THEN
   y <= '1';
ELSE
   y <= '0';
END IF;
```

IF structure with complete assignment.
When all possibilities are fully declared no memory element is needed

Use this instead!

VHDL basics cont.

Basic VHDL structures

Examples cont.

```
compare_ab:
IF ((a='1') AND (b='0')) THEN
   a_high <= '1';
   b_high <= '0';
   equal <= '0';
ELSIF ((a='0') AND (b='1')) THEN
   a_high <= '0';
   b_high <= '1';
   equal <= '0';
ELSIF..
..
ELSE
   a_high <= '0';
   b_high <= '0';
   equal <= '1';
END IF compare_ab;
```

The IF clause is evaluated first

The ELSIF clause is only evaluated if the IF clause is false

The ELSE clause is only evaluated if the IF and the ELSIF clause(s) are false

We have a priority-encoded structure with dominance for the first IF statement

Try using ELSIF instead of separate IF statements

VHDL basics cont.

Basic VHDL structures

CASE statement

The CASE statement has simularities to the WITH statement.

The structure is

```
[CASE_label:]
CASE selectorSignal IS
   WHEN value1 =>
      sequential code;
   WHEN value1 =>
      sequential code;
  [WHEN value2 =>
      sequential code;]
  [WHEN others =>
      sequential code;]
END CASE [CASE_label];
```

The CASE label is optional but it might encance the readability of the code

If the `WHEN` cases don´t cover all of the possible values for `selectorSignal` we must include the `OTHERS` clause. It could actually be there even when it is not needed so make it a habit to include it

All cases have the same priority and they may not overlap

VHDL basics cont.

Basic VHDL structures

```
[Case label:]
CASE selectorSignal IS
   WHEN value1 =>
      sequential code;
   WHEN value1 =>
      sequential code;
  [WHEN value2 =>
      sequential code;]
  [WHEN others =>
      sequential code;]
END CASE [Case label];
```

CASE statement cont.

`selectorSignal` is an input port, a signal or a variable.

The `valueX` could be one single value of `selectorSignal`.

It could also be more than one value if we combine them using the OR symbol `|`

or it could be a range of values if we use `TO` or `DOWNTO`

VHDL basics cont.

Basic VHDL structures

CASE statement cont.

Examples

```
SIGNAL tal_signal:INTEGER RANGE 0 TO 20;
SIGNAL output_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);

   selector:
   CASE tal_signal IS
      WHEN 1 =>
         output_signal <= "0001";
      WHEN 2 =>
         output_signal <= "0010";
      WHEN OTHERS =>
         output_signal <= "0000";
   END CASE selector;
```

selector

selector values

OTHERS clause

VHDL basics cont.

Basic VHDL structures

Examples cont.

```
SIGNAL tal_signal:INTEGER RANGE 0 TO 20;
SIGNAL output_signal:STD_LOGIC_VECTOR(3 DOWNTO 0);

   CASE tal_signal IS
      WHEN 1 => 
         output_signal <= "0001";
      WHEN 2 TO 4 => 
         output_signal <= "0011";
      WHEN 5 | 9 => 
         output_signal <= "0101";
      WHEN OTHERS =>
         output_signal <= "0000";
   END CASE;
```

Single selector value

Range of selector values

Group of selector values

VHDL basics cont.

Basic VHDL structures

LOOP statement

We have one sequential statement that has no concurrent correspondance, the **LOOP** statement.

The statement has a number of forms

- Infinite loop
- WHILE loop
- FOR loop

The FOR loop is the only LOOP statement that is synthesizable and only under some circumstances that we will get back to

VHDL basics cont.

Basic VHDL structures

Infinite LOOP statement NOTICE! This is not synthesizable

As the name suggests this loop goes on forever

The structure is

```
[LOOP_label:]
LOOP
   sequential code;
END LOOP [LOOP_label];
```

The LOOP label is optional but it might encance the readability of the code

Example

```
VARIABLE counter_variable:NATURAL;
……
count_variable:=0;
counter12:LOOP
   WAIT UNTIL rising_edge(clk);
   count_variable:=
        (count_variable+1) MOD 12;
END LOOP counter12;
```

Infinite loop

Triggered every clock cycle on positive clock edge

The counter counts from 0 to 11 on the rising edge of `clk` and then restarts

VHDL basics cont.

Basic VHDL structures

WHILE LOOP statement NOTICE! This is not synthesizable

The WHILE LOOP goes on while some condition is true

The structure is

```
[LOOP_label:]
WHILE condition LOOP
   sequential code;
END LOOP [LOOP_label];
```

The LOOP label is optional but it might encance the readability of the code

Example

```
VARIABLE sum_variable:NATURAL:=0;
……
add_loop:
WHILE (sum_variable < 100) LOOP
   sum_variable:=sum_variable + 3;
END LOOP add_loop;
```

VHDL basics cont.

Basic VHDL structures

FOR LOOP statement

This loop goes on for some range of an identifier

The structure is

Parentheses not allowed

```
[Loop label:]
FOR identifier IN discrete_range LOOP
   sequential code;
END LOOP [Loop label];
```

The LOOP label is optional but it might enhance the readabillity of the code

Example

To be synthesizable these must be constant values

```
one_fill:
FOR index IN 15 DOWNTO 0 LOOP
   vector(index)<='1';
END LOOP one_fill;
```

A way to fill the vector with ones.

Could be replaced by

```
vector <= (OTHERS=>'1');
```

VHDL basics cont.

Basic VHDL structures

LOOP control

We have a couple of functions to control the loop

With the **EXIT** statement we can break out of the loop and leave it.

The basic form is

```
IF condition THEN
   EXIT;
END IF;
```

The code could be shortened to

```
EXIT WHEN condition;
```

VHDL basics cont.

Basic VHDL structures

EXIT statement cont.

Example

```
VARIABLE count_variable:NATURAL;

……
count_variable:=0;
LOOP
   WAIT UNTIL ((clk='1') OR (reset='1'));
   EXIT WHEN (reset='1');
   count_variable:=
            (count_variable+1) MOD 12;
END LOOP;
```

Start value for the count

Triggered by `clk` or `reset`

Leave the loop and start all over again when `reset` is activated

The process continously counts from 0 to 11 on positive edge of the clock signal and is restarted when `reset` is one (1)

VHDL basics cont.

Basic VHDL structures

NEXT statement

The **NEXT** statement breaks the current loop round and moves on to the next round

The basic form is

```
IF condition THEN
   NEXT;
END IF;
```

The code could be shortened to

```
NEXT WHEN condition;
```

VHDL basics cont.

Basic VHDL structures

NEXT statement cont.

Example

```
ones_variable:=0;
FOR index IN WIDTH-1 DOWNTO 0 LOOP
      NEXT WHEN vector(index)='0';
      ones_variable:=ones_variable+1;
 END LOOP;
```

Move to the next bit in the vector when the current bit is zero

The code counts the number of ones (1) in `vector`

VHDL basics cont.

Basic VHDL structures

Asynchronous and synchronous code

Our VHDL code consists of two different types of code

- Asynchronous code

  The code is purely logical and not controlled by any clock

- Synchronous code

  The code is controlled by a trigger signal, a clock and will only execute when we have a new clock tick

In most cases our code use a combination of the two

VHDL basics cont.

Basic VHDL structures

Asynchronous and synchronous code cont.

In most practical cases it is easier to make synchronous code work as intended

Synchronous code must be written inside processes while asynchronous code can be written as concurrant code or written within an asynchronous process

We can find two basic synchronous structures

- Synchronous code with asynchronous reset
- Synchronous code with synchronous reset

VHDL basics cont.

Basic VHDL structures

Synchronous code with asynchronous reset

Basic structure

```
async_reset:PROCESS(clk,reset)
BEGIN
   IF (reset = '1') THEN
      asynchronous_reset_code;
   ELSIF (rising_edge(clk)) THEN
      synchronous_main_code;
   END IF;
END PROCESS async_reset;
```

Both signals must be able to trigger the process. No other signals should be included in the sensitivity list

Level triggered `reset`

Positively edge triggered `clk` signal

`reset` is dominant

VHDL basics cont.

Basic VHDL structures

Synchronous code with asynchronous reset cont.

Basic structure cont.

A positive clock edge is detected by

```
rising_edge(clk)
```

This can also be written as

```
clk'EVENT AND clk = '1'
```

A negative clock edge is detected by

```
falling_edge(clk)
```

This can also be written as

```
clk'EVENT AND clk = '0'
```

`signal'EVENT` is a signal attribute that is triggered when something has happened on the signal.

This kind of code might mistrigger so avoid it

Do not trigger on both edges of a clock.
This is not synthesizable, use only one of the edges.
In most cases it is best to use the same edge in the whole design

VHDL basics cont.

Basic VHDL structures

Synchronous code with synchronous reset

Basic structure

Since the `reset` signal is synchronized to the `clk` signal only the `clk` signal should trigger the process

```
PROCESS(clk)
BEGIN
   IF (rising_edge(clk)) THEN
      IF (reset = '1') THEN
         synchronous_reset_code;
      ELSE
        synchronous_main_code;
      END IF;
   END IF;
END PROCESS;
```

Can only be activated on the rising edge of clk

The same goes for this code

`clk` is dominant

VHDL basics cont.

Basic VHDL structures

Enable signal

Sometimes we also include a enable signal.

This would for the asynchronous reset case give the structure

```
PROCESS(clk,reset)
BEGIN
   IF (reset = '1') THEN
      asynchronous_reset_code;
   ELSIF (rising_edge(clk)) THEN
      IF (enable = '1') THEN
         synchronous_main_code;
      END IF;
   END IF;
END PROCESS;
```

Execute only when `enable` is activated and the clock has generated a positive flank

Notice that the `enable` signal doesn´t trigger the process

VHDL basics cont.

Basic VHDL structures

Time in hardware circuits

When we design hardware we often want to create delays.
A specific time period.

The concept of time really doesn´t exist in hardware besides the unavoidable delay we get when the signals pass through the electronic blocks.

In most cases the only time reference we have in our design is the period of the system clock we use.

Our usual way to create specific time periods is to count a number of clock cycles from the system clock.

This means that our smallest time tick, the time resolution we can use, is the period of the system clock.

We could of course also count changes on some external signal, use an external clock. This would i most cases be a slow clock.

VHDL basics cont.

Basic VHDL structures

Clocks in hardware circuits

When we design hardware we often want to create clock signals with other (lower) frequencies than the system clock.

To do this we have to divide the system clock down to a lower frequency, which means that these clocks will have frequencies that are the system clock frequency divided by some integer number.

We do this by counting clock pulses.

Most FPGA circuits use dedicated nets for the clock signals and there are not that many of these nets.

Because of this it is unwise to use many different clock signals in a design.

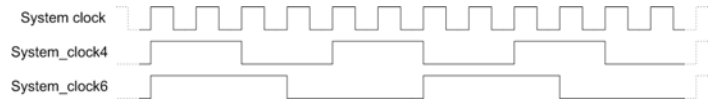
We can often solve this by using clock enable signals instead of derived clocks.

We´ll get back to this in lab assignment 5

VHDL basics cont.

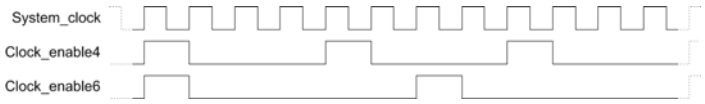
Basic VHDL structures

Clocks in hardware circuits cont.



Do not use

```
Rising_edge(system_clk4)
```



Instead use

```
IF (rising_edge(system_clk)) THEN
   IF (Clock_enable4='1') THEN
```

Experience says that the code works better if you don´t combine the two `if` caluses into one

VHDL basics cont.

Basic VHDL structures

Subprograms

We use subprograms to structure our code and when we want to instantiate the same code sequence more than once in our design or when we want to reuse codeblocks we have designed earlier.

Don´t confuse these subprograms with functions, procedures or subroutines used in software programming. Each instantiation of the routine will generate its own hardware and in most cases we can not actually share a subprogram.

If used in concurrent code then every instance of the subprogram has to be instantiated as it´s own hardware.
In sequential code we might be able to reuse the same block of hardware since different parts of the sequential code execute at different times. We use multiplexing

We have two kinds of subprograms

- Procedure — Doesn´t return any value but we can override this by sending a signal with the procedure call
- Function — Returns a value

VHDL basics cont.

Basic VHDL structures

Procedures

We have the structure

```
PROCEDURE identifier [(parameter_list)] IS
   [declarations of local signals,
    variables and constants]
BEGIN
   sequential statements;
END [PROCEDURE][identifier];
```

NOTICE!

The name and the label is optional but they might encance the readability of the code

The procedure description should be placed in the architecture of the program, at the same place as constants and signals are declared, that is before the BEGIN of the architecture

VHDL basics cont.

Basic VHDL structures

Procedures cont.

Example

Procedure name

```
PROCEDURE max_proc(a:IN STD_LOGIC_VECTOR
                        (7 DOWNTO 0);
                   b:IN STD_LOGIC_VECTOR
                        (7 DOWNTO 0);
                   SIGNAL max_signal:OUT
                          STD_LOGIC_VECTOR
                           (2 DOWNTO 0)) IS
BEGIN
   IF (a > b) THEN
      max_signal <= "100";
   ELSIF (a < b) THEN
      max_signal <= "001";
   ELSE
      max_signal <="010";
   END IF;
END PROCEDURE max_proc;
```

Calling parameters

Return parameter

Observe that it is declared as a signal

VHDL basics cont.

Basic VHDL structures

Example cont.

Procedure call

```
PROCEDURE max_proc(a:IN STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
                   b:IN STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
                   SIGNAL max_signal:OUT
                          STD_LOGIC_VECTOR
                           (2 DOWNTO 0)) IS
BEGIN
   IF (a > b) THEN
      max_signal <= "100";
   ELSIF (a < b) THEN
      max_signal <= "001";
   ELSE
      max_signal <="010";
   END IF;
END PROCEDURE max_proc;
```

```
SIGNAL a_signal:STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
SIGNAL b_signal:STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
SIGNAL max_signal:STD_LOGIC_VECTOR
                              (2 DOWNTO 0);
……
  max_proc(a_signal,b_signal,max_signal);
```

New line within a statement doesn´t affect the code

The value returned from procedure goes here

Observe! max_signal on top level and down in the procedure are not the same

VHDL basics cont.

Basic VHDL structures

Return from procedure

We can return from a procedure before we reach the end of the procedure code using a RETURN statement

Example

```
IF (a = ”00000000”) OR
   (b = ”00000000”) THEN
   max_signal <= ”000”;
   RETURN;
ELSIF (a > b) THEN
   max_signal <= "100";
ELSIF (a < b) THEN
   max_signal <= "001";
ELSE
   max_signal <="010";
END IF;
```

VHDL basics cont.

Basic VHDL structures

Functions

We have the structure

```
FUNCTION identifier [(inparameter_list)]
   RETURN return_type IS
   [declarations of local signals,
    variables and constants]
BEGIN
   sequential statements;
END [FUNCTION][identifier];
```

NOTICE!

The name and the label is optional but might encance the readability of the code

The function description should be placed in the architecture of the program, at the same place as constants and signals are declared, that is before the BEGIN of the architecture

VHDL basics cont.

Basic VHDL structures

Functions cont.

Example

Function name

Calling parameters

```
FUNCTION max_func(a:STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
                  b:STD_LOGIC_VECTOR
                              (7 DOWNTO 0)
                  RETURN STD_LOGIC_VECTOR IS
BEGIN
   IF (a > b) THEN
      RETURN "100";
   ELSIF (a < b) THEN
      RETURN "001";
   ELSE
      RETURN "010";
   END IF;
END FUNCTION max_func;
```

Type of return parameter

Notice that there is no size declaration of the vector and it has no name

VHDL basics cont.

Basic VHDL structures

Example cont.

Function call

```
FUNCTION max_func(a:STD_LOGIC_VECTOR
                              (7 DOWNTO 0);
                  b:STD_LOGIC_VECTOR
                              (7 DOWNTO 0)
                  RETURN STD_LOGIC_VECTOR IS
BEGIN
   IF (a > b) THEN
      RETURN "100";
   ELSIF (a < b) THEN
      RETURN "001";
   ELSE
      RETURN "010";
   END IF;
END FUNCTION max_func;
```

```
SIGNAL a_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL b_signal:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL max_signal:STD_LOGIC_VECTOR(2 DOWNTO 0);
……
  max_signal <= max_func(a_signal,b_signal);
```

The value returned from the function goes here

VHDL basics cont.

Basic VHDL structures

Overloading of subprograms

In many cases we want to use the same subprogram more than once, but with different types of calling or returning parameters.

To do this we can write more than one version of the subprogram with the same name but different parameter types or even different number of parameters.

The number of calling parameters and their type(s) will decide which version of the subprogram that will be used.

This is called **overloading**

VHDL basics cont.

Basic VHDL structures

Packages

Procedures and functions are ways to create subprograms that we want to use more than once in a design

With packages we go one step further and collect these structures into a separate file that can be reused in more than one design

We can also use a package to define types and subtypes, constants and signals and to create components

The package is divided into two separate parts

- The **package declaration** where we declare our subprograms, types, signals and so on
- The **package body** where we instantiate the subprograms and components

VHDL basics cont.

Basic VHDL structures

Package declaration

We have the syntax

The name of the package

```
PACKAGE identifier IS
   item_declarations;
END [PACKAGE][identifier];
```

The name and the label is optional but they might encance the readability of the code

Package body

We have the syntax

The same name as in the package declaration

```
PACKAGE BODY identifier IS
   item_instantiations;
END [PACKAGE BODY][identifier];
```

The name and the label is optional but they might encance the readability of the code

VHDL basics cont.

Basic VHDL structures

Packages

Example

Let´s move the procedure max_proc and the function max_func into a package

Package declaration

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE max_package IS
   PROCEDURE max_proc(a:IN STD_LOGIC_VECTOR
                                 (7 DOWNTO 0);
                      b:IN STD_LOGIC_VECTOR
                                 (7 DOWNTO 0);
                      SIGNAL max_signal:OUT
                              STD_LOGIC_VECTOR
                                 (2 DOWNTO 0));
   FUNCTION max_func(a:STD_LOGIC_VECTOR
                                 (7 DOWNTO 0);
                     b:STD_LOGIC_VECTOR
                                 (7 DOWNTO 0))
                     RETURN STD_LOGIC_VECTOR;
END PACKAGE max_package;
```

Procedure declaration

Function declaration

VHDL basics cont.

Basic VHDL structures

Example cont.

Package body

```
PACKAGE BODY max_package IS

   PROCEDURE max_proc(a:IN STD_LOGIC_VECTOR
                                 (7 DOWNTO 0);
                      b:IN STD_LOGIC_VECTOR
                                 (7 DOWNTO 0);
                    SIGNAL max_signal:OUT
                           STD_LOGIC_VECTOR
                                 (2 DOWNTO 0)) IS
   BEGIN
      {same code as before}
   END PROCEDURE max_proc;
(cont)
```

Procedure instantiation

VHDL basics cont.

Basic VHDL structures

Example cont.

Function instantiation

```
    FUNCTION max_func(a:STD_LOGIC_VECTOR
                                (7 DOWNTO 0);
                      b:STD_LOGIC_VECTOR
                                (7 DOWNTO 0))
                      RETURN STD_LOGIC_VECTOR IS
    BEGIN
       {same code as before}
    END FUNCTION max_func;
END PACKAGE BODY max_package;
```

VHDL basics cont.

Basic VHDL structures

To use a package

We have already seen how to use the standard packages like `std_logic_1164`.

We use our own packages in the same way

Syntax

The name of the library

```
USE library_catalogue.library_identifier.ALL;
```

If we have the library in our project then the library will be compiled to the subfolder `work` in the project catalogue

To use libraries in other catalogues the search paths to these have to be defined in the compiler system

Indicates that we want to use all declarations in the package

VHDL basics cont.

Basic VHDL structures

To use a package cont.

Example

Let´s use our package `max_package` in a main program

We write a main program

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.max_package.ALL;

ENTITY package_main IS
   PORT(a:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        max1:OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        max2:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END package_main;
```

Our package is placed within our project

VHDL basics cont.

Basic VHDL structures

To use a package cont.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.max_package.ALL;

ENTITY package_main IS
    PORT(a:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         b:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         max1:OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
         max2:OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END package_main;
```

Example cont.

```
ARCHITECTURE arch_package_main OF
package_main IS
BEGIN
   max_proc(a,b,max1);
   max2 <= max_func(a,b);
END arch_package_main;
```

Return value

Defined in package `max_package`

Return value

VHDL basics cont.

Basic VHDL structures

Making the package generic

To make our package more useful we can make it work for any vector size by leaving out the size declarations of the parameters

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE max_package IS
   PROCEDURE max_proc(a:IN STD_LOGIC_VECTOR;
                      b:IN STD_LOGIC_VECTOR;
                      SIGNAL max:OUT
                           STD_LOGIC_VECTOR);
   FUNCTION max_func(a:STD_LOGIC_VECTOR;
                     b:STD_LOGIC_VECTOR)
                     RETURN STD_LOGIC_VECTOR;
END PACKAGE max_package;
```

No sizes

The signal sizes will be decided by the sizes of the calling parameters

VHDL basics cont.

Basic VHDL structures

Standard packages

VHDL includes a number of standard packages that are part of the IEEE standard package collection.

We will discuss a few of these and some of their contents

- **standard** defines the character set, integer, real, time, string, boolean_vector, bit_vector, integer_vector, real_vector
- **math_real** works on real numbers and has declarations for mathematical constants like pi, square root, exponential and logarithmic functions and trigonomethric functions
- **math_complex** has mathematical constants and functions for complex numbers
- **std_logic_1164** declares the `std_logic` data type, declares logical functions, shift operators and conversions between `std_logic` and `bit`

VHDL basics cont.

Basic VHDL structures

Standard packages cont.

- **numeric_bit** has numerical and logical operations for bits
- **numeric_std** has more or less the same declarations as `numeric_bit` but for `std_logic`
- **fixed_generic_pkg** has numerical and logical operations for fixed point numbers
- **float_generic_pkg** has numerical and logical operations for floating point numbers

A search of the internet will give the declarations within these packages

We will in most cases create our designs using only the `std_logic_1164` library or complement it with use of the `numeric_std` library and in some rare cases `math_real`

Let´s have a look at these two packages

Demonstration!

VHDL basics cont.

Basic VHDL structures

Components

Components are subdesigns that we use as building blocks to build larger designs

We will get a hierarchical design that might be easier to grasp

Splitting the design into components makes it easier to simulate and test the separate blocks on their own

It is also a way to reuse building blocks from earlier designs and to interconnect two designs into one larger design

When we use a component in a design we have to declare the component and connect its ports to the signals in the higher level design

Let´s illustrate with an example

VHDL basics cont.

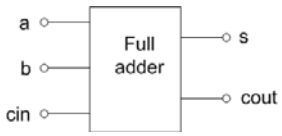
Basic VHDL structures

Components cont.

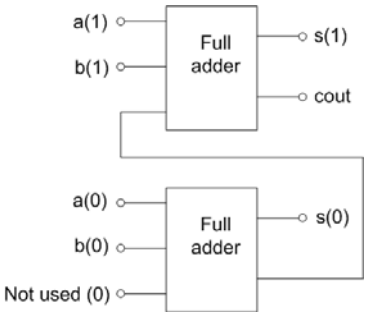
Example

We want to build a two-bit adder.

We start by building a one-bit full adder and then use two blocks of this kind as components in the two bit version.

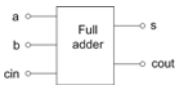


One-bit adder



Two-bit adder

VHDL basics cont.



Basic VHDL structures

Components cont.

Example cont.

Our one-bit adder has the code

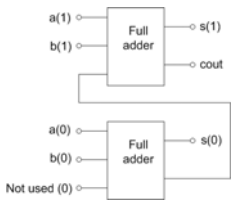
```
ENTITY full_adder IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        cin:IN STD_LOGIC;
        s:OUT STD_LOGIC;
        cout:OUT STD_LOGIC);
END full_adder;

ARCHITECTURE arch_full_adder OF full_adder IS
BEGIN
   s<=a XOR b XOR cin;
   cout<=(a AND b) OR
         (a AND cin) OR
         (b AND cin);
END arch_full_adder;
```

Separate lines increases readability

VHDL basics cont.

Basic VHDL structures



Example cont.

We move on to the two-bit adder.

We have the entity

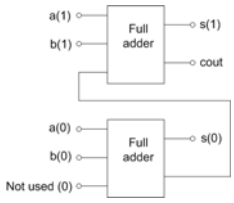
```
ENTITY adder_2_bit IS
   PORT(a:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        s:OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END adder_2_bit;
```

The entity is the same as for the one bit adder exept that `a`, `b` and `s` have changed from calars to vectors

VHDL basics cont.

Basic VHDL structures

```
ENTITY adder_2_bit IS
   PORT(a:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END adder_2_bit;
```



Example cont.

and the architecture

```
ARCHITECTURE arch_adder_2 OF adder_2 IS
   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL cint_signal:STD_LOGIC;
BEGIN
   full_adder_comp0:COMPONENT full_adder
      PORT MAP(a=>a(0),b=>b(0),
                  cin=>'0',s=>s(0),cout=>cint_signal);
   full_adder_comp1:COMPONENT full_adder
      PORT MAP(a=>a(1),b=>b(1),
                 cin=>cint_signal,s=>s(1),cout=>cout);
END arch_adder_2;
```

Component declaration

Component instantiations

VHDL basics cont.

Basic VHDL structures

Example cont.

Lets´s look at the different parts of our architecture.

First the component declaration

```
COMPONENT full_adder IS
    PORT(a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         cin:IN STD_LOGIC;
         s:OUT STD_LOGIC
         cout:OUT STD_LOGIC);
END COMPONENT full_adder;
```

If we compare this to the entity of our one bit full adder

```
ENTITY full_adder IS
    PORT(a:IN STD_LOGIC;
         b:IN STD_LOGIC;
         cin:IN STD_LOGIC;
         s:OUT STD_LOGIC
         cout:OUT STD_LOGIC);
END full_adder;
```

We see that they are the same exept that the word ENTITY has been replaced by the word COMPONENT and we have added the word COMPONENT at the end as well

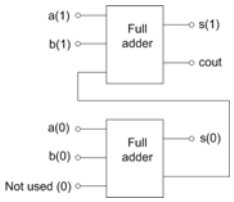
VHDL basics cont.

Basic VHDL structures

Example cont.

In the architecture body we instantiate two full adders as components.

Let´s look at the first one




```
ENTITY adder_2_bit IS
   PORT (a:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         b:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
         cout:OUT STD_LOGIC);
END adder_2_bit;
```

The same signal name can be used on different design levels without interference

Name of component port

Top level signal connected to the component port

```
full_adder_comp0:COMPONENT full_adder
  PORT MAP(a=>a(0),b=>b(0),
           cin=>'0',s=>s(0),cout=>cint);
```

Transfer carry out to the next bit using an internal signal

The instantiation must have a label

Connect `cin` to a constant value zero (0) since we have no carry in

The port map connects the ports of the component to the signals in the top design

VHDL basics cont.

Basic VHDL structures

Example cont.

```
PORT MAP(a=>a(0),b=>b(0),
                  cin=>'0',s=>s(0),cout=>cint);
```

This way of connecting the signals to the component is called nominal mapping.

Here the order of the assignments doesn´t matter and we can even leave out the assignment of some outputs if we don´t want to use them. We could for example skip carry out if we don´t use it

```
PORT MAP(a=>a(0),b=>b(0),cin=>'0',s=>s(0));
```

All inputs must be there though since there values are needed to generate the output signal

VHDL basics cont.

Basic VHDL structures

Example cont.

I prefere to have one assignment per line when I do the port mapping. The code gets more readable and you can have comments on each assignment if you like

```
PORT MAP(a=>a(0),
         b=>b(0),
         cin=>'0',
         s=>s(0),
         cout=>cint);
```

I don´t do that in this presentation to same space on the slides

VHDL basics cont.

Basic VHDL structures

Example cont.

We can also use positional mapping where the possition of the signal within the PORT MAP desides where it is connected

```
PORT MAP(a(0),b(0),'0',s(0),cint);
```

Since the position is essential then all signals must be there. If we want to leave someone out we must keep the position.

If we once again leave out the carry out signal from the component.

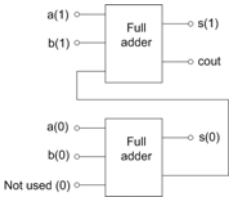
```
PORT MAP(a(0),b(0),'0',s(0),);
```

The positional mapping is shorter but often confusing since you must all the time check that you have got the positioning right.

Using nominal mapping and keeping the port order from the component entity makes the code less error prone

VHDL basics cont.

Basic VHDL structures



Example cont.

Let´s repeat the total architecture

```
ARCHITECTURE arch_adder_2 OF adder_2 IS
   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL cint_signal:STD_LOGIC;
BEGIN
   full_adder_comp0:COMPONENT full_adder
      PORT MAP(a=>a(0),b=>b(0),
                  cin=>'0',s=>s(0),cout=>cint_signal);
   full_adder_comp1:COMPONENT full_adder
      PORT MAP(a=>a(1),b=>b(1),
                 cin=>cint_signal,s=>s(1),cout=>cout);
END arch_adder_2;
```

```
ENTITY adder_2_bit IS
   PORT(a:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END adder_2_bit;
```

Transfer carry out to the next bit using a signal

VHDL basics cont.

Basic VHDL structures

This works fine for just a few instantiations of a component.

But it will be a lot of code, if we have many bits, for example 32 bits.

In this case there is a simplier way

Example

We continue with the adder but will use it for the addition of two 32 bit vectors

The entity will be the same as before but the vectors will have increased the number of bits to 32

```
ENTITY adder_32_bit IS
   PORT (a:IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         b:IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         s:OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
         cout:OUT STD_LOGIC);
END adder_32_bit;
```

VHDL basics cont.

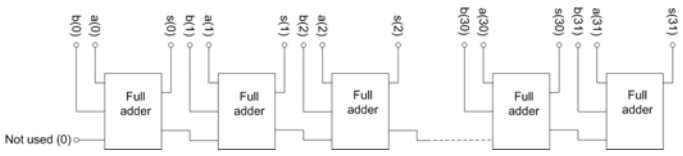
Basic VHDL structures

Example cont.

The component declaration will be exactly the same as in the 2-bit case since it is the same one-bit-adder component.

```
COMPONENT full_adder IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        cin:IN STD_LOGIC;
        s:OUT STD_LOGIC
        cout:OUT STD_LOGIC);
END COMPONENT full_adder;
```

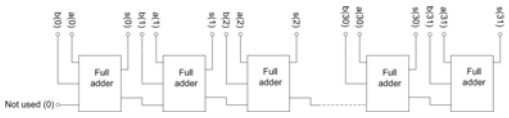
We have to change the architecture body with the component instantiations



VHDL basics cont.

Basic VHDL structures

Example cont.




```
ARCHITECTURE arch_adder_32_bit OF adder_32_bit IS
{declaration of component full_adder}
SIGNAL cint:STD_LOGIC_VECTOR(30 DOWNTO 0);
BEGIN
   full_adder_comp0:COMPONENT full_adder
         PORT MAP(a=>a(0),b=>b(0),cin=>'0',
                                  s=>s(0),cout=>cint(0));
   G:FOR i IN 1 TO 30 GENERATE
      full_adder_compi:COMPONENT full_adder
         PORT MAP(a=>a(i),b=>b(i),cin=>cint(i-1),
                                  s=>s(i),cout=>cint(i));
   END GENERATE;
   full_adder_comp31:COMPONENT full_adder
      PORT MAP(a=>a(31),b=>b(31),cin=>cint(30),
                                  s=>s(31));
END arch_adder_32_bit;
```

Signals that transfer the carrier bits between the com-ponents

No `cin`

Component instantiations

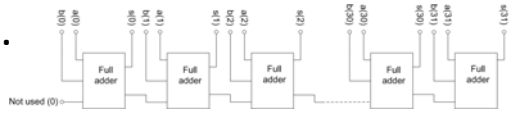
Carry ripples from one adder to the next

MSB and LSB are instantiated seperately since they have somewhat different in- and output signals

`cout` not connected, therefore missing

VHDL basics cont.

Basic VHDL structures



Example cont.

Let´s take a closer look at the generation of bit 1 to 30

The generate statement must have a label

The index variable does not have to be declared

```
G:FOR i IN 1 TO 30 GENERATE
      full_adder_compi:COMPONENT full_adder
         PORT MAP(a=>a(i),b=>b(i),cin=>cint(i-1),
                                  s=>s(i),cout=>cint(i));
   END GENERATE;
```

We need a component label but we do not need separate labels for the different instantiated components

We have what might look like a loop but the synthesize tool will unwrap this loop and instantiate 30 separate one bit adders

VHDL basics cont.

Basic VHDL structures

Now we might wonder:

Is there a way to generalize the adder so that we can use the same design no matter the number of bits we want to use?

Yes there is!

First of all we could define a constant to give the number of bits, which means that we only have to make a change in one place in the code when we change the number of bits

VHDL basics cont.

Basic VHDL structures

```
ARCHITECTURE arch_adder_x_bit OF adder_x_bit IS
{declaration of component full_adder}
CONSTANT WIDTH:NATURAL:=32;
SIGNAL cint:STD_LOGIC_VECTOR(WIDTH-2 DOWNTO 0);
BEGIN
   full_adder_comp0:COMPONENT full_adder
      PORT MAP(a=>a(0),b=>b(0),cin=>'0',
                                y=>y(0),cout=>cint(0));
   G:FOR i IN 1 TO WIDTH-2 GENERATE
       full_adder_compi:COMPONENT full_adder
          PORT MAP(a=>a(i),b=>b(i),cin=>cint(i-1),
                                   y=>y(i),cout=>cint(i));
   END GENERATE;
   full_adder_compN_1:COMPONENT full_adder
      PORT MAP(a=>a(WIDTH-1),b=>b(WIDTH-1),
                  cin=>cint(WIDTH-2),y=>y(WIDTH-1);
END arch_adder_x_bit_generate;
```

Constant declaration

We will have to edit the constant in the architecture to change the number of bits

VHDL basics cont.

Basic VHDL structures

As the next step we move the bit number constant out of the arcitecture and into the entity and define it as a **GENERIC**

```
ENTITY adder_x_bit IS
   GENERIC(WIDTH:NATURAL:=32);
   PORT(a:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END adder_x_bit;
```

The syntax for the generic is

```
GENERIC (generic_name:generic_type[:=value]);
```

We will get back to the case when we don´t need a value

VHDL basics cont.

Basic VHDL structures

All we have to do in the architecture is to remove the constant.
This is now replaced by the GENERIC in the entity

```
ARCHITECTURE arch_adder_x_bit OF adder_x_bit IS
{declaration of component full_adder}
SIGNAL cint:STD_LOGIC_VECTOR(WIDTH-2 DOWNTO 0);
BEGIN
   full_adder_comp0:COMPONENT full_adder
      PORT MAP(a=>a(0),b=>b(0),cin=>'0',
                                y=>y(0),cout=>cint(0));
   G:FOR i IN 1 TO WIDTH-2 GENERATE
      full_adder_compi:COMPONENT full_adder
         PORT MAP(a=>a(i),b=>b(i),cin=>cint(i-1),
                                  y=>y(i),cout=>cint(i));
   END GENERATE;
   full_adder_compN_1:COMPONENT full_adder
      PORT MAP(a=>a(WIDTH-1),b=>b(WIDTH-1),
            cin=>cint(WIDTH-2),y=>y(WIDTH-1);
END arch_adder_x_bit_generate;
```

VHDL basics cont.

Basic VHDL structures

This has improved the flexibility of our adder.

But we have to go in to the entity of the adder and edit it to change the number of bits.

What if we could describe the adder as a component and set the number of bits at instantiation?

To do this we keep the generic in the entity but we remove it´s value

Generic without value

```
ENTITY adder_x_bit IS
   GENERIC(WIDTH:NATURAL);
   PORT(a:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END adder_x_bit;
```

and set it when we instantiate the adder.

We can actually keep the value of the generic as a default. It will be overwritten by the instantiation

VHDL basics cont.

Basic VHDL structures

Example

Let´s try to implement two adders with different word lengths using our component. We add two 16 bit numbers and two 8 bit numbers.

We have the entity

We have used two new generics to set the two bit widths in the components

```
ENTITY multiple_adders IS
   GENERIC(WIDTH16:NATURAL:=16;
           WIDTH8:NATURAL:=8);
   PORT (a:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
         b:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
         c:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
         d:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
         y:OUT STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
         z:OUT STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
         cout16:OUT STD_LOGIC;
         cout8:OUT STD_LOGIC);
END multiple_adders ;
```

VHDL basics cont.

Basic VHDL structures

```
ENTITY multiple_adders IS
   GENERIC(WIDTH16:NATURAL:=16;
           WIDTH8:NATURAL:=8);
   PORT(a:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        c:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        d:IN STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(WIDTH16-1 DOWNTO 0);
        z:OUT STD_LOGIC_VECTOR(WIDTH8-1 DOWNTO 0);
        cout16:OUT STD_LOGIC;
        cout8:OUT STD_LOGIC);
END multiple_adders ;
```

Example cont.

In the architecture we instantiate two adders using the same component

```
ARCHITECTURE arch_multiple_adders OF multiple_adders
IS
COMPONENT adder_x_bit IS
   GENERIC(WIDTH:NATURAL);
   PORT(a:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        b:IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        cout:OUT STD_LOGIC);
END COMPONENT adder_x_bit;
BEGIN
   adder_x_bit_comp0:COMPONENT adder_x_bit
      GENERIC MAP(WIDTH=>WIDTH16)
      PORT MAP(a=>a,b=>b,y=>y,cout=>cout16);
   adder_x_bit1:COMPONENT adder_x_bit
      GENERIC MAP(WIDTH=>WIDTH8)
      PORT MAP(a=>c,b=>d,y=>z,cout=>cout8);
END arch_multiple_adders;
```

Generic component

Instantiation of 16 bit adder

Instantiation of 8 bit adder

VHDL basics cont.

Basic VHDL structures

Memories

Basically we use two types of memories

- ROM, read only memory where the data can be read but can not be changed (written)
- RAM, random access memory, read/write memory where the data can be both read and written

In most cases we don´t use memories with bit sized data but we read and write words of some size to a number of addresses

A suitable data type for the memory would then be an array of vectors

```
TYPE mem_array IS ARRAY (0 TO SIZE-1) OF
              STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
```

where we address each vector instead of the individual bits

VHDL basics cont.

Basic VHDL structures

Memories cont.

```
TYPE mem_array IS ARRAY (0 TO SIZE-1) OF
                  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
```

To create a memory we instantiate the type, in this case `mem_array`.

For a ROM memory the values should me initialized into the memory at design time and then never change. We use a constant declaration for the instantiation

Our ROM is named ROM

```
CONSTANT ROM:mem_array:=(value(1),value(2),
                         ..,
                         value(Size-1));
```

For a RAM memory we instantiate the memory by creating a signal without values

Our RAM is named RAM

```
SIGNAL RAM:mem_array;
```

We could fill the RAM with values at the instantiation but that will only work in simulation and will not be synthesized.

For syntesis we need to fill the RAM with values using VHDL code

VHDL basics cont.

Basic VHDL structures

Memories cont.

A small ROM can be created using a case statement

Hexadecimal base

```
BCD_7seg:PROCESS(bcd) IS
BEGIN
   CASE bcd IS
      WHEN X"0" => seg <= "0111111";
      WHEN X"1" => seg <= "0000110";
      WHEN X"2" => seg <= "1011011";
      WHEN X"3" => seg <= "1001111";
      WHEN X"4" => seg <= "1100110";
      WHEN X"5" => seg <= "1101101";
      WHEN X"6" => seg <= "1111101";
      WHEN X"7" => seg <= "0000111";
      WHEN X"8" => seg <= "1111111";
      WHEN X"9" => seg <= "1101111";
      WHEN OTHERS => seg <= "1000000";
   END CASE BCD_7seg;
END PROCESS;
```

This memory converts BCD code to 7 segment code

Observe the process

VHDL basics cont.

Basic VHDL structures

Memories cont.

We could do the same using a WITH statement

```
WITH bcd SELECT
seg <= ”0111111” WHEN X”0”,
       ”0000110” WHEN X”1”,
       ”1011011” WHEN X”2”,
       ”1001111” WHEN X”3”,
       ”1100110” WHEN X”4”,
       ”1101101” WHEN X”5”,
       ”1111101” WHEN X”6”,
       ”0000111” WHEN X”7”,
       ”1111111” WHEN X”8”,
       ”1101111” WHEN X”9”,
       ”1000000” WHEN OTHERS;
```

```
BCD_7seg:PROCESS(bcd) IS
BEGIN
   CASE bcd IS
      WHEN X”0” => seg <= ”0111111”;
      WHEN X”1” => seg <= ”0000110”;
      WHEN X”2” => seg <= ”1011011”;
      WHEN X”3” => seg <= ”1001111”;
      WHEN X”4” => seg <= ”1100110”;
      WHEN X”5” => seg <= ”1101101”;
      WHEN X”6” => seg <= ”1111101”;
      WHEN X”7” => seg <= ”0000111”;
      WHEN X”8” => seg <= ”1111111”;
      WHEN X”9” => seg <= ”1101111”;
      WHEN OTHERS => seg <= ”1000000”;
   END CASE BCD_7seg;
END PROCESS;
```

Observe that there is no process

VHDL basics cont.

Basic VHDL structures

Test benches

To verify that our design is correct we need to simulate our results.

We will be using the simulator QuestaSim (or ModelSim) from Mentor for this.

To assist in the simulation we can create a kind of test fixture in VHDL, a **test bench**.

This is a top level design where we instantiate our own design as a component and generate input stimuli for the component and watch or check the resulting output signals. We micht also check internal signals

We can have three different types of test benches

- **Type 1** only generates input stimuli and we have to watch the results in the simulator
- **Type 2** generates input stimuli, checks the results and gives an OK signal if the resulting output values are correct
- **Type 3** generates input stimuli and writes a message to the simulator output window if something goes wrong with the simulation results

VHDL basics cont.

Basic VHDL structures

Test benches

Example

Let´s take our one bit full adder as an example.

We have the entity

```
ENTITY full_adder IS
   PORT(a:IN STD_LOGIC;
        b:IN STD_LOGIC;
        cin:IN STD_LOGIC;
        s:OUT STD_LOGIC
        cout:OUT STD_LOGIC);
END full_adder;
```

VHDL basics cont.

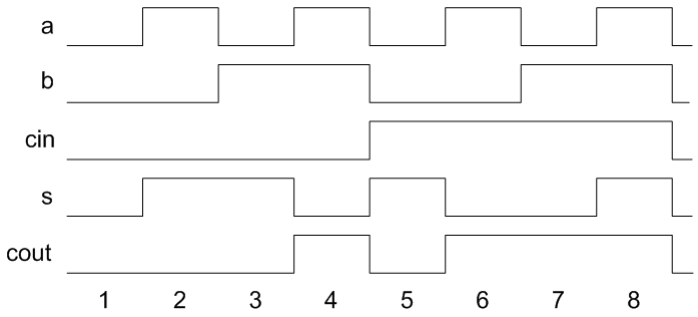
Basic VHDL structures

```
ENTITY full_adder IS
   PORT(   a:IN STD_LOGIC;
           b:IN STD_LOGIC;
         cin:IN STD_LOGIC;
           y:OUT STD_LOGIC
        cout:OUT STD_LOGIC);
END full_adder;
```

Test benches

Example cont.

We need to generate eight different input stimuli to fully test the circuit



To simplify things we will accept that more than one signal changes it´s value at a given time

VHDL basics cont.

Basic VHDL structures

Example cont.

Let´s start creating the test bench. We begin with test bench type 1

In this case we only watch the results from the simulation and we do not need any input or output ports to the test bench.

```
ENTITY full_adder_tb1 IS

END full_adder_tb1;
```

The entity is empty since we have no inputs and no outputs

This causes a problem in QuestaSim since by default signals that don´t connect to outputs are optimized away.

To overtcome this we can simulate without optimization. The paper on QuestaSim describes how to do this

VHDL basics cont.

Basic VHDL structures

```
ENTITY full_adder_tb1 IS
   PORT (   y_tb:OUT STD_LOGIC
         cout_tb:OUT STD_LOGIC);
END full_adder_tb1;
```

Example cont.

In the test bench architecture we instantiate our full adder as a component

```
ARCHITECTURE arch_full_adder_tb1 OF full_adder_tb1 IS
   COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL a_signal:STD_LOGIC;
   SIGNAL b_signal:STD_LOGIC;
   SIGNAL cin_signal:STD_LOGIC;
   SIGNAL s_signal:STD_LOGIC;
   SIGNAL cout_signal:STD_LOGIC;
BEGIN
   full_adder_comp:COMPONENT full_adder
      PORT MAP(a=>a_tb,b=>b_tb,cin=>cin_tb,
               s=>s_signal,cout=>cout_signal);
```

Component declaration

Signals to connect to the component

Component Instantiation

VHDL basics cont.

Basic VHDL structures

Example cont.

We complete the architecture with the input stimuli

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
            '1' AFTER 300 ns,
            '0' AFTER 400 ns,
            '1' AFTER 500 ns,
            '0' AFTER 600 ns,
            '1' AFTER 700 ns;
b_signal <= '0',
            '1' AFTER 200 ns,
            '0' AFTER 400 ns,
            '1' AFTER 600 ns;
cin_signal <= '0',
              '1' AFTER 400 ns;
END arch_full_adder_tb1;
```

This is one of the few times when we can and should use time in our designs

The exact times are not important since we deal with simulation of a design without circuit delays

But we should create all the input signal combinations we want to test for

VHDL basics cont.

Basic VHDL structures

Example cont.

We need a do file.

Since the instimuli is given in the test bench all the do file need to is to set up signals we like to watch and run the simulation time.

```
-- full_adder_tb1.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
add wave s_signal cout_signal
run 730ns
```

Signals to watch

Run the simulation for this time

VHDL basics cont.

Basic VHDL structures

Example cont.

We move on to test bench type 2

Here we will need a output signal that signals if something goes wrong with the output signals from the component during simulation.

We add an output to our test bench entity

```
ENTITY full_adder_tb2 IS
   PORT(test_OK:OUT STD_LOGIC);
END full_adder_tb2;
```

VHDL basics cont.

Basic VHDL structures


```
ENTITY full_adder_tb2 IS
   PORT(test_OK:OUT STD_LOGIC);
END full_adder_tb2;
```


Example cont.

In the architecture we keep the component declaration and do a component instantiation using signals and not outports

```
ARCHITECTURE arch_full_adder_tb2 OF full_adder_tb2 IS
 COMPONENT full_adder IS
      PORT(a:IN STD_LOGIC;
           b:IN STD_LOGIC;
           cin:IN STD_LOGIC;
           s:OUT STD_LOGIC
           cout:OUT STD_LOGIC);
   END COMPONENT full_adder;
   SIGNAL a_signal:STD_LOGIC;
   SIGNAL b_signal:STD_LOGIC;
   SIGNAL cin_signal:STD_LOGIC;
   SIGNAL s_signal:STD_LOGIC;
   SIGNAL cout_signal:STD_LOGIC;
BEGIN
   full_adder_comp:COMPONENT full_adder
      PORT MAP(a=>a_tb,b=>b_tb,cin=>cin_tb,
               s=>s_signal,cout=>cout_signal);
```

VHDL basics cont.

Basic VHDL structures

Example cont.

We keep the imput stimuli from test bench type 1

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
            '1' AFTER 300 ns,
            '0' AFTER 400 ns,
            '1' AFTER 500 ns,
            '0' AFTER 600 ns,
            '1' AFTER 700 ns;
b_signal <= '0',
            '1' AFTER 200 ns,
            '0' AFTER 400 ns,
            '1' AFTER 600 ns;
cin_signal <= '0',
              '1' AFTER 400 ns;
```

VHDL basics cont.

Basic VHDL structures

Example cont.

We have to complete the code with a test of the output signals

We write the code so that the `test_OK` signal will go low if an error occures and then stay low even if the next stimuli gives a correct result

VHDL basics cont.

Basic VHDL structures

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
...
b_signal <= '0',
            '1' AFTER 200 ns,
...
cin_signal <= '0',
              '1' AFTER 400 ns;
```

Example cont.

```
test_proc:PROCESS
BEGIN
   test_OK <= '1';
   WAIT FOR 50 ns; -- 000
   IF ((s_signal/='0') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 100
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 010
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   ……
END PROCESS test_proc;
END arch_test_bench_type2;
```

Default value for `test_OK`

Wait until the input signals have stabilized

If the result isn´t 00 then set test_OK low

Test for next stimuli

Continue for all eight combinations of input signals

VHDL basics cont.

Basic VHDL structures

Example cont.

We need a do file here too.

The only difference from the do file for test bench type 1 is that we have added the signal `test_OK` to the signals we watch

```
-- full_adder_tb2.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
add wave s_signal cout_signal test_OK
run 730ns
```

Signals to watch

Run simulation

Added signal

The only signal to watch in the test bench is really `test_OK` but it is practical to keep the rest of the signals for debugging

VHDL basics cont.

Basic VHDL structures

Example cont.

```
test:PROCESS
BEGIN
   test_OK <= '1';
   WAIT FOR 50 ns; -- 000
   IF ((s_signal/='0') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 100
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   WAIT FOR 100 ns; -- 010
   IF ((s_signal/='1') OR (cout_signal /= '0')) THEN
      test_OK <= '0';
   END IF;
   ……
END PROCESS test;
END arch_test_bench_type2;
```

Observe that as soon as a test sets `test_OK` to zero then it will stay at zero although following tests can be OK

We shouldn´t run this simulation longer than the added WAIT times since the process will restart when it reaches it´s end and then the results will most likely be incorrect

VHDL basics cont.

Basic VHDL structures

Example cont.

Now over to test bench type 3

In this case we don´t need any output signal either since the internal signals are used for our test and since these tests will give the written reports if something is wrong then we have an empty entity

```
ENTITY full_adder_tb3 IS

END full_adder_tb3;
```

We will rewrite the test process, that is replace the test_OK process, but keep the rest of the architecture code

VHDL basics cont.

Basic VHDL structures

Example cont.

```
a_signal <= '0',
            '1' AFTER 100 ns,
            '0' AFTER 200 ns,
...
b_signal <= '0',
            '1' AFTER 200 ns,
...
cin_signal <= '0',
              '1' AFTER 400 ns;
```

```
test_proc:PROCESS
BEGIN
   WAIT FOR 50 ns; -- 000
   ASSERT ((s_signal='0') AND (cout_signal = '0'))
   REPORT "000 50ns"
   SEVERITY warning;
   WAIT FOR 100 ns; -- 100

   ASSERT ((s_signal='1') AND (cout_signal = '0'))
   REPORT "100 150ns"
   SEVERITY warning;
   WAIT FOR 100 ns; -- 010
   ...
END PROCESS test_proc;
END arch_test_bench_type3;
```

If this condition is true then nothing is wrong with the signals so do nothing

The current simulation time and this text will be written to the simulators Transcript window if the output signals are incorrect

Continue for all eight combinations of input signals

VHDL basics cont.

Basic VHDL structures

Example cont.

If the **ASSERT** expression is true then the output signals have the correct values

If the expression is false then the test time and the **REPORT** message will be written to the simulators output window

We can have four different levels of **SEVERITY**

- **note**, the message will have the header Note
- **warning**, the message will have the header Warning
- **error**, the message will have the header Error
- **failure**, the message will have the header Failure

The simulation continues

The simulation will stop at current time

The severity levels are given in increasing order

The severity level should be choosen based on the kind of action the error calls for

VHDL basics cont.

Basic VHDL structures

Example cont.

The assert messages that you write can be simple or very detailed.

You decide!

For a more advanced design the test will be quit extensive and you need to write a lot of code just for the test

VHDL basics cont.

Basic VHDL structures

Example cont.

Once again we need a do file.

We´re back to the same do file as the one we used for test bench 1 since we have no output.

```
-- full_adder_tb3.do

restart -f -nowave
view signals wave
add wave a_signal b_signal cin_signal
Add wave s_signal cout_signal
run 730ns
```

Signals to watch

Run simulation

Strictly we don´t need to watch any signals since we have the assertions but like in test bench type 2 it is practical to keep the signals for debugging