Fundamental Elements of VHDL

A VHDL program consists of a collection of **design units**, each of which is defined using three components

Library and Package Declaration

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

Libraries and packages are collections of commonly used items, such as data types, subprograms and components

The above two packages define *std_logic* and *std_logic_vector* data types, as well as *signed* and *unsigned*

I also find it extremely useful to create a file with my own data types and constants, that are then included declared below the *ieee* packages

```
library work;
use work.DataTypes_pkg.all;
```

Fundamental Elements of VHDL

Entity Declaration

```
entity entity_name is
  port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type;
  );
end entity_name;
```

port_names are the **formal** signal names of the design unit, which are used to connect this design unit to pins on an FPGA or to other design units

The *mode* component can be **in**, **out** or **inout** (for bi-directional port)

ALWAYS use *std_logic* and *std_logic_vector* as the data_type in **entity** declarations

Fundamental Elements of VHDL

A common mistake with *mode* is to try to use a signal of mode **out** as an *input signal* within the architecture body

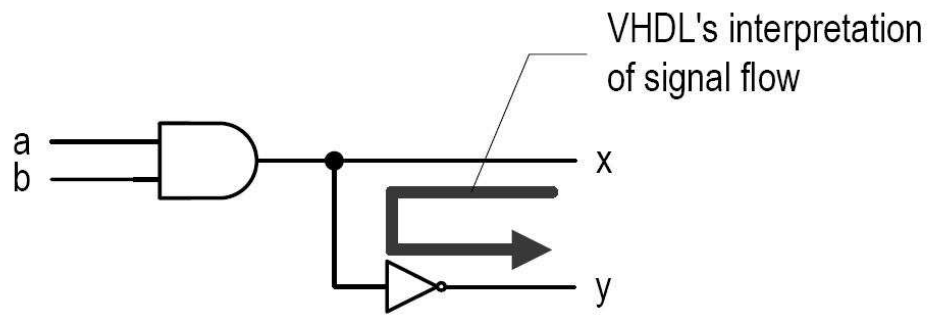
Consider:

```
library ieee;
use ieee.std_logic_1164.all;
entity mode_demo is
  port(
       a, b: in std_logic;
       x, y: out std_logic);
end mode_demo;

architecture wrong_arch of mode_demo is
  begin
  x <= a not b;
  y <= not x;   -- ERROR!!!!
end wrong_arch;
```

Fundamental Elements of VHDL

Port signals defined to be *out* can NOT be read



This code reads and writes *x* so it must be defined as **inout** to avoid a syntax error

But *x* is really not a bi-directional signal in the true sense of the word

The solution you will be forced to adopt is to create an *internal* signal as follows

```
architecture ok_arch of mode_demo is
  signal ab: std_logic;
  begin
  ab <= a and b;
  x <= ab;
  y <= not ab;
end ok_arch;
```

Fundamental Elements of VHDL

Architecture Body

The architecture body specifies the logic functionality of the design unit

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

The *declaration* part is optional and can include **internal signal declarations** or **constant declarations**

There are several possibilities for *concurrent_stmts*, which we will cover soon

Fundamental Elements of VHDL

Comments start with two dashes, e.g.,

```
-- This is a comment in VHDL
```

An **identifier** can only contain alphabetic letters, decimal digits and underscore; the first character *must be a letter* and the last character **cannot** be an underscore

VHDL is case **IN**sensitive, i.e., the following identifiers are the same

nextstate, NextState, NEXTSTATE, nEXTsTATE

Smart convention: Use CAPITAL_LETTERs for constant names and the suffix _*n* to indicate active-low signals

Signal declaration

```
signal signal_name, signal_name, ... : data_type
```

Fundamental Elements of VHDL

The **std_logic_vector** is an array of elements with *std_logic* data type

```
signal a: std_logic_vector(7 downto 0);
```

The **downto** syntax puts the most significant bit (7) on the left, which is the natural representation for numbers (I rarely use the **(0 to n**) syntax)

std_logic constants are enclosed in single quotes: '1' and '0'

std_logic_vector constants are enclosed in double quotes: "00101"

Signal **assignment** uses '<=', NOT '=' as in programming languages

```
signal_name <= constant or signal_name;
```

Constant declaration

```
constant const_name, ... : data_type := value_expr;
```

For example

```
constant BUS_WIDTH_LB: integer := 5;
constant BUS_WIDTH_NB: integer := 2**BUS_WIDTH_LB;
```

Fundamental Elements of VHDL

operator	description	data type of operand a	data type of operand b	data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a **mod** b	modulo			
a **rem** b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a **sll** b	shift left logical	bit_vector	integer	bit_vector
a **srl** b	shift right logical			
a **sla** b	shift left arithmetic			
a **srl** b	shift right arithmetic			
a **rol** b	rotate left			
a **ror** b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a **and** b	and	boolean, bit, bit_vector	same as a	boolean, bit, bit_vector
a **or** b	or			
a **xor** b	xor			
a **nand** b	nand			
a **nor** b	nor			
a **xnor** b	xnor			

Not automatically synthesizable

Precedence	Operator
Highest	`**` **abs not**
	`*` `/` **mod rem**
	`+ - (ident/neg)`
	`& + - (add/sub)`
	sll srl sla sra rol ror
Lowest	**and or nand nor xor xnor**

Note: **and** and **or** have SAME precedence -- use parenthesis!

Fundamental Elements of VHDL

You will use *std_logic_vector* instead of *bit_vector* as defined in the table

Division by powers of 2 can be used in signal assignment stmts, e.g., *a/16*

This is implemented by the synthesis tool as a right shift operation

Division by other numbers requires a design unit that implements the division!

VHDL is a strongly-typed language, requiring frequent type casting and conversion

This is particularly evident with the *shift* operator

```
a <= resize(unsigned(b), 10) sll
        to_integer(unsigned(c));
```

Here, *a* is a *unsigned* of size 10 elements, and *b* and *c* are *std_logic_vector*

Bits or a range of bits can be referenced as

```
a(1)
a(7 downto 3)
```

Fundamental Elements of VHDL

VHDL relational operations, >, =, etc, must have operands of the same element type but their **widths may differ**

Avoid comparing operands of different widths, it's error prone

Concatenation operator (&) constructs and/or extends operands on the right

Also used to force a match between width of the operands on left and right

```
y <= "00" & a(7 downto 2);
y <= a(7) & a(7) & a(7 downto 2);
y <= a(1 downto 0) & a(7 downto 2);
```

Also useful when defining a *shift register* as we will see later

Array aggregate

```
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
a <= (7|5=>'1', others=>'0');
a <= (7 downto 3 => '0') & b(7 downto 5);
a <= (others=>'0');
```

Last assignment is very useful and works independent of the data type

Fundamental Elements of VHDL

IEEE numeric_std package

Standard VHDL and the *std_logic_1164* package support arithmetic operations only on *integer* data types

```
signal a, b, sum: integer;
. . .
sum <= a + b;
```

But this is inefficient in hardware because integer does NOT allow the range (number of bits) to be specified

We certainly don’t want a 32-bit adder when an 8-bit adder would do

The *numeric_std* package allows an array of 0’s and 1’s to be interpreted as an *unsigned* or *signed* number, using these names as the data type

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

signal x, y: signed(15 downto 0);
```

Fundamental Elements of VHDL

For *signed*, the array is interpreted in **2's-complement** format, with the MSB as the sign bit

Therefore "1100" represents 12 when interpreted as an unsigned number but -4 as a signed number

The *numeric_std* package supports arithmetic operations, including those involving integer constants

```
signal a, b, c, d, e: unsigned(7 downto 0);
...
a <= b + c;
d <= b + 1;
e <= (5 + a + b) - c;
```

Note that the sum "wraps around" when overflow occurs, so BE VERY CAREFUL when choosing a size

Fundamental Elements of VHDL

numeric_std
package definitions

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a	absolute value	signed		signed
- a	negation			
a * b				
a / b		unsigned	unsigned, natural	unsigned
a **mod** b	arithmetic	unsigned, natural	unsigned	unsigned
a **rem** b	operation	signed	signed, integer	signed
a + b		signed, integer	signed	signed
a - b				
a = b				
a /= b		unsigned	unsigned, natural	boolean
a < b	relational	unsigned, natural	unsigned	boolean
a <= b	operation	signed	signed, integer	boolean
a > b		signed, integer	signed	boolean
a >= b				

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b)	shift left	unsigned, signed	natural	same as a
shift_right(a,b)	shift right			
rotate_left(a,b)	rotate left			
rotate_right(a,b)	rotate right			
resize(a,b)	resize array	unsigned, signed	natural	same as a
std_match(a,b)	compare '-'	unsigned, signed std_logic_vector, std_logic	same as a	boolean
to_integer(a)	data type	unsigned, signed		integer
to_unsigned(a,b)	conversion	natural	natural	unsigned
to_signed(a,b)		integer	natural	signed

Fundamental Elements of VHDL

There are three ***type conversion functions*** in *numeric_std* package

```
to_unsigned, to_signed and to_integer
```

Use *to_unsigned* and *to_signed* when assigning constants to *unsigned* and *signed* signals

```
a <= to_unsigned(2048, 13);
```

Assumes *a* is *unsigned* and of width 13

a is assigned the constant 2048

```
a <= resize(unsigned(b), 10) sll
        to_integer(unsigned(c));
```

Looked at this earlier -- *sll* operator requires an integer type as last operand

a must be *unsigned* of width 10

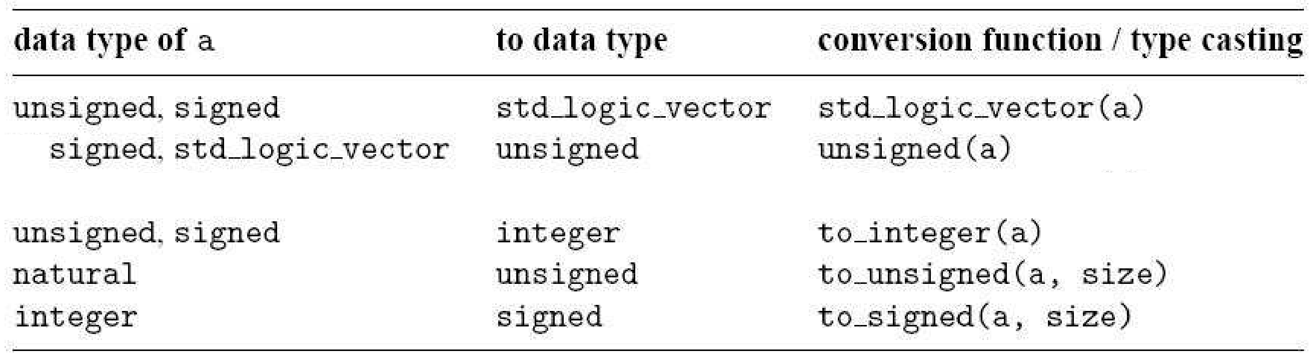
```
a(to_integer(b)) <= '1';
```

Indexing into *std_logic_vector* requires an integer data type

Here *b* must be *unsigned*

Fundamental Elements of VHDL

Type casting is also possible between 'closely related' data types



data type of a	to data type	conversion function / type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Type casting

Type conversion

```
signal u1, u2: unsigned(7 downto 0);
signal v1, v2, v3: std_logic_vector(7 downto 0);
signal sg: signed(7 downto 0);

u1 <= unsigned(v1);
v2 <= std_logic_vector(u2);
u2 <= unsigned(sg) + u1;
v3 <= std_logic_vector(unsigned(v1) + unsigned(v2));
```

Use *resize* to deal with width differences if they exist