TSIU03: A Fairly Small VHDL Guide

Petter Källström, Mario Garrido
{petterk, mariog}@isy.liu.se

Version: 2.0

Abstract

This VHDL guide is aimed to show you some common constructions in VHDL, together with their hardware structure. It also tells the difference between concurrent and sequential VHDL code. The emphasize is on RTL level (synthesizable code), but some high level VHDL code are also presented.

Contents

This document is a brief VHDL summary, intended as a help during the labs, rather than a read-through document. For an introduction to VHDL, consider taking Alteras 90 minutes online class in basic VHDL[1], or attend the course lectures.

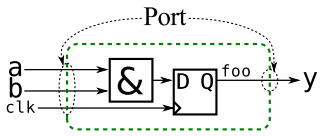
1 Introduction

Very high speed integrated circuit (VHSIC) Hardware Description Language (VHDL) was initially aimed as a way to describe good models of how digital circuits behaved. Soon, VHDL started to be used to describe how circuits could be built.[citation needed]

1.1 A Simple Example

The structure of a VHDL file is depicted in Code 1.

- **library** ⇒ Gives you access to the library `ieee`, which contains all standard functions defined in VHDL.
- **use ieee.std_logic_1164.all;** ⇒ Lets you simpler access members from the package `ieee.std_logic_1164`, e.g. `std_logic`.
- **Comments** starts with "`--`" on a line.
- **entity** *ename* **is** {defs} **end entity;** ⇒ Defines the "public interface" of the module *ename*.
- **port(**{pins}**);** ⇒ If the module is a chip, {pins} is the pins.
- **std_logic** ⇒ "*The*" data type for digital logic. Mostly '0' or '1'.
- **architecture** *aname* **of** *ename* **is** {decl} **begin** {body} **end architecture** ⇒ Defines the "engine" of module *ename*. {decl} contains all declaration of internal signals in the module. {body} contains the actual logic definition. *aname* is the name of the "engine". "RTL" stands for "Register Transfer Level".
- **signal foo : std_logic;** ⇒ Declares an internal signal named `foo` of type `std_logic`.
- **process(clk) begin** {body} **end process;** ⇒ Defines a sequential block that is "executed" on all events of the signal `clk`.
- **if rising_edge(clk) then** {stats} **end if;** ⇒ The code "{stats}" will be "executed" only on positive clock flanks. Hence each assignment here acts as a D-type flip flop (DFF, or "D-vippa" in Swedish).
- **foo <= a and b;** ⇒ Implement the logic AND gate, and assign the result to (the input of) a DFF.
- **y <= foo;** ⇒ Y is continuously assigned the value of signal `foo`. E.g., directly connected to `foo`.




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

-- this is a comment
entity and_dff is
  port(clk : in std_logic;
       a,b : in std_logic;
       y : out std_logic);
end entity;

architecture rtl of and_dff is
  signal foo : std_logic;
begin
  process(clk) begin
    if rising_edge(clk) then
      foo <= a and b;
    end if;
  end process;
  y <= foo;
end architecture;
```

Code 1: A simple VHDL example.

1.2 RTL vs Behavioral VHDL

VHDL can, in some sense, be divided into **RTL** and **behavioral** code.

1.2.1 RTL VHDL

RTL ("Register Transfer Level") code can be directly synthesized into hardware, in terms of gates, registers etc. This is exemplified throughout the documentation.

1.2.2 Behavioral VHDL

In addition to what can be described as RTL code, the behavioral models can use much more complex constructions. A few examples;

- Sequential execution tricks;
  - Pause execution somewhere (e.g. `wait until ack='1';`).
- Time effects;
  - Assign signals after a certain amount of time (`y <= x or z after 10ns;`).
  - Pause execution for some time (`wait for 30ns;`).

1.3 Concurrent vs Sequential Syntax

The architecture body contains concurrent and sequential code. Some syntax is the same in both cases, but more advanced language constructions have different syntaxes.

By default, the code in the architecture is concurrent, which means all statements are executed in parallel, all the time (and hence, it does not matter in which order you write them). You can have processes, and within those, the code is sequential.

1.3.1 Concurrent VHDL

Remember that you want to create hardware. The hardware is there all the time. If you have 200 gates, all 200 gates will "execute" the incoming signal continuously. You describe the different gates using a number of statements, but it does not matter in which order the statements are written (in most cases).

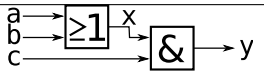
Concurrent VHDL will always generate combinational logic[1].

Code 2 shows three ways of writing the logic net in (d). (a) and (b) are exactly the same thing, while (c) will not define the intermediate signal `x`.

```
x <= a or b;
y <= x and c;
```
(a)

```
y <= x and c;
x <= a or b;
```
(b)

```
y <= (a or b) and c;
```
(c)



(d)

Code 2: Some examples of the same thing.

1.3.2 Sequential VHDL

In processes the code is sequential.

You can have processes that generates only combinational signals, but we do not recommend those constructions. Instead, we recommend processes that looks like those in Code 3, where a DFF is implemented with (a) asynchronous reset and (b) synchronous reset.

```
process(clk,rstn) begin
  if rstn = '0' then
    Q <= '0';
  elsif rising_edge(clk) then
    Q <= D;
  end if; -- rstn,clk
end process;
```
(a)

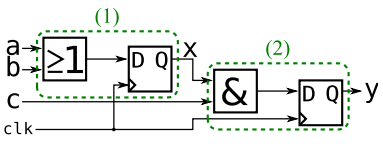
```
process(clk) begin
  if rising_edge(clk) then
    if rstn = '0' then
      Q <= '0';
    else
      Q <= D;
    end if; -- rstn
  end if; -- clk
end process;
```
(b)

Code 3: A DFF with (a) asynchronous and (b) synchronous reset.

The signals are updated when the execution comes to the end of the process. Hence, the order of signal assignments does not matter, just like in the concurrent case. Version (a) and (b) in Code 4 gives the behavior depicted in (c).

```
process(clk) begin
  if rising_edge(clk) then
    x <= a or b; -- (1)
    y <= x and c; -- (2)
  end if;
end process;
```
(a)

```
process(clk) begin
  if rising_edge(clk) then
    y <= x and c; -- (2)
    x <= a or b; -- (1)
  end if;
end process;
```
(b)



(c)

Code 4: Two ways of writing the same thing. Note that `c` is "AND:ed" with the *old* version of (`a OR b`).

[1]There are dirty ways to generate registers or flip flops in concurrent VHDL, but we don't teach dirty codes.

2 Data Types

There are some data types in VHDL that is good to know about.

2.1 std_logic Based Data Types

The package `ieee.std_logic_1164` contains the data type `std_logic`, and a set of operations on this, and some derived data types from this, e.g., `std_logic_vector`.

2.1.1 std_logic

In digital theory, you learned that the logic level can be zero or one. In VHDL, there are nine digital states for the type `std_logic`.

A `std_logic` constant is written using the `'` (single citation mark), e.g. `'0'` or `'1'`.

The states of std_logic are:

- `'0'`, `'1'` – The standard zero and one.
- `'L'`, `'H'` – A kind of weak low or high, corresponds to if a resistor is pulling it toward '0' or '1'.
- `'Z'` – High impedance, drive neither to '0' nor to '1'. The "readers" of a signal should do this.
- `'U'` – Unknown. This is default for a signal in simulation, until the value is set.
- `'W'` – Weak unknown.
- `'X'` – Strong unknown, e.g. when short circuit a '0' and a '1', or performing operations on a 'U'.
- `'-'` – Don't care. Used in comparisons.

There are rules for what happens in a short circuit. For instance, if you short circuit 'L' and a 'H', you will get 'W', and if you short circuit 'L' and '1', you will get '1'. 'H' and 'Z' becomes 'H', and so on.

In RTL code, you will most likely only use '0' and '1', and you should typically not short circuit anything.

2.1.2 std_logic_vector

A `std_logic_vector` is an array of `std_logic`. It must have non-negative indices. The array spans from left to right, and the index can be increasing or decreasing.

A `std_logic_vector` constant is given using the `"` (double citation mark), e.g. `"1001"`.

When you declare a signal, you specify the left most and the right most index of it, and the direction (using the `to` or `downto` keywords), e.g.:

- `std_logic_vector(0 to 2)` – A three bit vector, indexed 0, 1, 2.
- `std_logic_vector(2 downto 0)` – A three bit vector, indexed 2, 1, 0.
- `std_logic_vector(5 to 200)` – A 196 bit vector, indexed 5, 6, 7, ..., 200.

A good (but not so widely used) practice is to use `(N-1 downto 0)` for vectors representing values (`N` bits signed or unsigned numbers), and to use `(0 to N-1)` for vectors representing `N` bits, e.g., some control bits.

If you need to type long constants, it can be handy to add some kind of delimiter, which can be done using the "b" prefix, which allows the character "_" in the string. E.g., the constant 10^6, as a 24 bit vector will be

$$\begin{aligned} 10^6 &= \texttt{"000011110100001001000000"} \\ &= \texttt{b"0000\_1111\_0100\_0010\_0100\_0000"} \\ &= \texttt{X"0F4240"} \text{ (hexadecimal)} \end{aligned}$$

The packages `ieee.std_logic_signed` and`ieee.std_logic_unsigned` contains arithmetic operations on those.

2.1.3 signed, unsigned

The package `ieee.numeric_std` declares the data types `SIGNED` and `UNSIGNED`, both have the same definition as `std_logic_vector`. They are treated as unsigned and two's complement signed number respectively in all arithmetic operations.

2.2 High Level Data Types

There are also some more data types, e.g.

- `integer` – an integer, signed 32 bit value.
- `unsigned` – an integer, unsigned 32 bit value.
- `boolean` – a boolean, can be `true` or `false`.

Those *can* be used for synthesis, but we recommend they are not.

2.3 Signal Attributes

A vector have some attributes you can access, e.g. the left most index of vector `vec` is `vec'left` (pronounced "vec-tic-left". Some of those attributes are exemplified in Table 1

```
signal vec_up :  std_logic_vector(4 to 6);
signal vec_dn :  std_logic_vector(7 downto 2);
```

	X = vec_up	X = vec_dn
X'left	4	7
X'right	6	2
X'high	6	7
X'low	4	2
X'length	3	6
X'range	4 to 6	7 downto 2

Table 1: Some signal attributes

There are more attributes as well, for instance all signals have the attribute `'event` (see Section 3.6, "Process Definitions").

3 Declarations and Definitions

So much to declare...

3.1 Use Package Declarations

Some examples:

- `library ieee;` – Declares that we want to access the entire content defined by ieee.
- `use ieee.std_logic_1164.all;` – We want simple access to all declarations in the package.
- `use ieee.std_logic_unsigned.CONV_INTEGER;` – We want to use simplified access to the function `CONV_INTEGER` from the package.
- `use ieee.std_logic_signed."+";` – We want to perform additions in signed values (e.g. sign extending the shorter value).

If you do not use the `library` {lib}`;` command, you have no way to access the functions in the library. If you want to use the `+` operator from the signed package, you can write;

`res <= ieee.std_logic_signed."+"(a,b);`	Without "`use ieee.std_logic_signed."+";`"
`res <= a + b;`	With "`use ieee.std_logic_signed."+";`"

You can find a good list of the standard packages, and what they contains on the web page [2], and in App A.

3.2 Entity Definitions

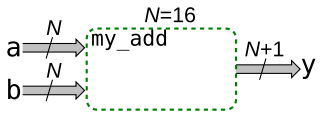
The syntax for the entity definition is

```
entity {ename} is generic({glist}); port({plist}); end entity;
```

- {ename} ⇒ The name of the entity.
- `generic(`{glist}`)` – Optional input of design constants. Must contain at least one constant.
- {glist} ⇒ The design constants, on the form "`const1 : type1 := default1; const2 : type2 := default2; ...`". The semicolon is used as separator, do not use a semicolon before the "`)`".
- `port(`{plist}`)` ⇒ optional pin declarations. Must contain at least one pin.
- {plist} ⇒ A list of design "pins", on the form "`a1,a2,... :` {dir} `typeA := initA; ...`".
  - {dir} ⇒ the direction of the pins. Any of `in`, `out`, `inout`, `buffer`. You will typically only use `in` and `out`.
  - `in` ⇒ input. You can read from, but not write to this signal.
  - `out` ⇒ output. You can write to, but not read from this signal.
  - `inout` ⇒ bidirectional. You can read and write to this signal. Write 'Z' to "release" the signal, so someone else can write to it. This is used for e.g. memory buses.
  - `buffer` ⇒ output. You can write and read. This is considered as dirty code in the VHDL community. We do not accept any usage of this.
- `end entity` ⇒ Here you can instead write "`end` {ename}".

An example is depicted in Code 5

```
entity my_add is
  generic(N : integer := 16);
  port(a,b : in std_logic_vector(N-1 downto 0);
       res : out std_logic_vector(N downto 0));
end entity;
```



Code 5: An Entity Example.

Here, the owner (if this is a sub module) can decide how wide the buses should be (by changing N). By default, the input buses are 16 bits, and the result is 17 bits.

In simulations, you can have, e.g. `generic(delay : time := 10ns);`, if you want to model different delays in the design.

3.3 Architecture Definitions

The syntax for the architecture definition is

```
architecture {aname} of {ename} is {declarations} begin {body} end architecture;
```

- {aname} ⇒ The name of the architecture, e.g., `rtl`.
- {ename} ⇒ The name of the entity it implements.
- {declarations} ⇒ Here you can declare/define signals, functions, aliases, constants, component, ...
- {body} ⇒ Here is the body of the architecture – the logic definition.
- `end architecture` ⇒ You can instead write `end` {aname}.

3.4 Signal Declarations

Signals are declared before the `begin` in the architecture. The syntax is

`signal` {snames} `:` {type} `:=` {initial value}`;`

- {snames} ⇒ mandatory signal names. One or several comma separated names.
- {type} ⇒ mandatory data type, including eventual vector borders.
- {initial value} ⇒ optional initial value. std_logic will default to `'U'` if not initiated, in simulation.

Some examples are given in Code 6.

```
architecture rtl of foo is
  signal sl1,sl2 : std_logic; -- initiates to '0' in synthesis, and 'U' in simulation.
  signal sl3 : std_logic := '0'; -- initiates to '0' even in simulation.
  signal slv1 : std_logic_vector(7 downto 0); -- a byte.
  signal slv2 : std_logic_vector(11 downto 0) := X"3ff"; -- initial value = 1023.
begin
```

Code 6: Examples of signal declaration.

3.5 Function Definitions

You can also declare/define functions before the `begin` statement in the architecture.

`function` {fname}`(`{args}`)` `return` {type} `is` {declarations} `begin` {body} `end function;`

- {fname} ⇒ The name of the function.
- {args} ⇒ A list of arguments, on the form `(a1,a2,... : typeA; b1,b2,... : typeB; ...)`. The types should *not* include vector borders. E.g., declare just "`std_logic_vector`", rather than "`std_logic_vector(3 downto 0)`".
- {type} ⇒ The return type. Also without vector borders.
- {declarations} ⇒ You can declare variables (but not signals). Variables are explained in Sections 3.7 and 6.4.
- {body} ⇒ Here is the body of the function. The body must `return` a value.
- `return` {val} ⇒ Will return the value {val} and quit the function.
- `end function` ⇒ You can instead write `end` {name}.

Some key points:

- Functions are always executed sequentially, so you must use the sequential syntax.
- Functions can be called from both concurrent and sequential VHDL code.
- You can override or redefine operators. E.g. `function "+"(lhs,rhs : type)`.
- Operators will have the same priority as always.
- You cannot define new operator names, like the `!` operator, whatever you would like that to do.

Some examples are given in Code 7, where the function `rising_edge` are the actual definition from the `ieee.std_logic_1164` package. The function uses the keyword "`signal`" before the argument `foo`, which forbids calls with e.g. variables.

```
architecture rtl of foo is
  function mux(s,a0,a1 : std_logic) return std_logic is begin
    if s = '0' then return a0; end if; return a1;
  end mux;
  function "and"(lhs : std_logic_vector; rhs : std_logic) return std_logic_vector is
    variable res : std_logic_vector(lhs'range);
  begin
    for i in lhs'range loop
      res(i) := lhs(i) and rhs;
    end loop;
    return res;
  end function;
  function rising_edge(signal foo : std_logic) return boolean is begin
    return foo'event and foo = '1';
  end function;
  signal bit1,bit2,bit3,bit4 : std_logic;
  signal vec1,vec2 : std_logic_vector(3 downto 0);
begin
  bit1 <= mux(bit2,bit3,bit4);
  vec1 <= vec2 and bit1;
  -- vec1 <= bit1 and vec2; will fail - no such function.
  process(clk)
    variable foo : std_logic;
  begin
    foo := clk;
    -- we can't call rising_edge(foo), since foo is a variable.
    if rising_edge(clk) then -- using our function
      foo := bit2; -- This will be assigned directly.
      vec2 <= inp_vec1 and foo; -- this is the "new" version of foo, i.e. bit2
    elsif ieee.std_logic_1164.rising_edge(clk) then -- using the standard function
      bit2 <= mux(inp_bit1, bit3,bit4);
    end if;
  end process;
end architecture;
```

Code 7: Examples of function declarations and how to use them.

3.6 Process Definitions

A process is placed in the concurrent code, and can be seen as a "sequential island in the concurrent sea". The formal syntax is:

{pname} `: process(`{sensitivity list}`) `{declarations}` begin `{body}` end process;`

- {pname} `:` ⇒ An optional name of the process. E.g. `rx_proc : process(clk) ....`
- {sensitivity list} ⇒ A list of signals that should trig the process to start (in a simulation). If one or more signals in the sensitivity list changes, those signals will get the attribute `'event` = true, and then the process will run. All other signals will have the `'event` = false.
- {declarations} ⇒ Here you can declare variables or functions. You cannot declare signals.
- {body} ⇒ Here is the body of the process (where you place the `if rising_edge` etc).
- `end process` ⇒ Here you can instead write `end` {pname}, if you specified a name.

3.7 Variable Declarations

Variables are declared just like signals, but use the keyword `variable` instead of `signal`, and are declared in processes or functions. Read more in Section 6.4

4 Basic VHDL

What is stated here holds in both concurrent and sequential VHDL.

4.1 Logic Operations

Those operations works typically on `boolean` and `std_logic`, and element wise on `std_logic_vector`.

Some operations are:

- `not`
- `and`, `nand`
- `or`, `nor`
- `xor`, `xnor`

Example of a multiplexer implemented with logic gates:

```
res <= (a0 and not s) or (a1 and s);
```

Some interesting functions from `ieee.std_logic_misc`, operating on `std_logic_vectors`.

- `fun_MUX2x1(in0,in1,sel)` ⇒ A multiplexer, that works on `std_logic`.
- {X}`_REDUCE(slv)` ⇒ An N input {X} gate, fed with a vector of size N. {X} is `AND`, `NAND`, `OR`, `NOR`, `XOR` or `XNOR`.

  E.g. `XNOR_REDUCE(x(2 to 4))` corresponds to "`not (x(2) xor x(3) xor x(4));`".

  It is not possible to mix `boolean` and `std_logic` (unless you define those operators yourself).

4.2 Arithmetic Operations

Those operations works typically on numerical data types, like `integer`, `natural`, `std_logic_vector`. When used on `std_logic_vector`, the functions are available in the packages `ieee.std_logic_unsigned` and `ieee.std_logic_signed`, that might behave differently (since, e.g. `"1011"` is -5 in a signed system, and $+11$ in an unsigned system).

It is often possible to mix std_logic_vectors with integers, but not always.

The **unary** operations are:

- `+` ⇒ No effect, e.g., `a <= +b;`
- `-` ⇒ Negating, e.g., `a <= -b;` (see `ieee.std_logic_signed`).
- `abs x` ⇒ absolute value of `x`.

The **binary** operations are:

- `+`, `-`, `*`, `/` ⇒ The classical four operations.
- `a ** b` ⇒ Exponential, a^b.
- `mod`, `rem` ⇒ modulo and reminder, e.g. `res <= a mod b;`.

In synthesis, you should only use the operations `+`, `-` and `*`. The other operations are really complicated to implement in hardware, and should not be used lightly.

Multiplications or divisions by two are just shift operations (no logical gates are needed). Those are handled in Section 4.4, "Vectors and Indexing".

4.3 Test Operations

Those also operates on numerical data types. The operations returns the data type boolean, which is used by the more advanced constructions.

- `=`, `/=` ⇒ Equal or not equal.
- `<`, `<=` ⇒ Less than (or equal), only numerical comparison.
- `>`, `>=` ⇒ Greater than (or equal), only numerical comparison.

The `=` and `/=` also works for `std_logic` and vectors not treated as numbers. Two vectors are equal if all bits are equal.

Note that the operator `<=` is also an assignment operator.

4.4 Vectors and Indexing

VHDL have great support for vectors, e.g. `std_logic_vector`.

We use the signals in Table 2 to exemplify the operations.

Signal	Type	Content
`x,y`	`std_logic`	*'x', 'y'*
an	`std_logic_vector(`n`-1 downto 0)`	"$a_{n-1} \dots a_0$"
bn	`std_logic_vector(`n`-1 downto 0)`	"$b_{n-1} \dots b_0$"
cn	`std_logic_vector(`n`-1 downto 0)`	"$c_{n-1} \dots c_0$"
Examples		
`a4`	`std_logic_vector(3 downto 0)`	"$a_3a_2a_1a_0$"
`b5`	`std_logic_vector(4 downto 0)`	"$b_4b_3b_2b_1b_0$"

Table 2: Declaration of signals used in examples.

4.4.1 Vector Indexing

Indexing is illustrated by the examples in Table 3.

Expression	Result
`a4(2)`	'a_2', a `std_logic`
`a4(2 downto 2)`	"a_2", a vector with one element
`a4(2 downto 3)`	"", a vector with zero elements
`a4(3 downto 2)`	"a_3a_2"
`a5(3 downto 2) <= "10";`	`a5` = "$a_4$10a_1a_0"
`a6(0) <= 'X';`	`a6` = "$a_5a_4a_3a_2a_1$X"
`a4(2) <= "1";`	Error: Cannot assign a vector to a bit

Table 3: Examples of vector indexing.

4.4.2 Vector concatenation

The "`&`" operator is used to merge vectors, and works for both `std_logic` and `std_logic_vectors`. The result is always a vector. Some examples are shown in Table 4

Expression	Result
`x & y;`	"xy"
`a3 & b4`	"$a_2a_1a_0b_3b_2b_1b_0$"
`x & b3`	"$xb_2b_1b_0$"
`b5(0) & b5(4 downto 1)`	"$b_0b_4b_3b_2b_1$"
`a5 <= ('0' & b4) + '1'`	"$a_4a_3a_2a_1a_0$", where $a_{3..0}$ = `b4`+1, a_4 = carry out.

Table 4: Examples of vector concatenation

4.4.3 Aggregation: Generating Vectors (the "`(others=>'0')`" syntax)

Generate a vector by the syntax

`(ix1=>val1,ix2=>val2,..., others=>val`N`)`

- `ix`n ⇒ Any index, e.g. "`5`", "`2 to 3`" or "`5 downto 0`".
- `val`n ⇒ A vector element, e.g. `'1'`, `'0'` or `'Z'`. Must *not* be a std_logic_vector.
- `others=>val`N ⇒ An optional syntax when assigning signals/variables.

When assigning, using the "`others=>val`N", the indices must exactly match those of the receiving signal/variable. All non given indices will get the value `val`N.

When assigning, not using the "`others=>val`N", the direction of the aggregation is that of the receiving signal/variable. All indices in the range must be given.

When not assigning, the index of the result is increasing, so the lowest index is to the left.
Some examples are shown in Table 5.

Expression	Result	Aggregation Index
`(5=>'1', 4=>'0')`	`"01"`	`4 to 5`
`(4=>'0', 1 to 3=>'1', 0=>'Z')`	`"Z1110"`	`0 to 4`
`(4=>'0', 1 to 2=>'1', 0=>'Z')`	Error: index 3 is missing in range 0 to 4.	
`a5 <= (3=>'1', others=>'0')`	`a5` = `"01000"`	`4 downto 0`
`a5 <= (5=>'1', 4 downto 1=>'0')`	`a5` = `"10000"`	`5 downto 1`
`a5 <= ""&(5=>'1', 4 downto 1=>'0')`	`a5` = `"00001"`	`1 to 5`
`a5 <= (5=>'1', others=>'0')`	Error: index 5 is outside range 4 downto 0.	
`b5 = (b5'range => '1')`	test if all elements in `b5` is one.	

Table 5: Examples of aggregation.

Aggregation is most often used for resets, something like "`my_signal <= (others => '0');`", which fills `my_signal` with zeros.

4.4.4 Shifting

There are six built in shift operators, listed in Table 6. Unfortunately, those are only defined for the type `bit_vector`, and not for any vector type.

Operator	Operation	Shifted In	Example	Result
`sll`	logic shift left.	0	`a5 sll 1`	"$a_3a_2a_1a_0$0"
`srl`	logic shift right.	0	`a5 srl 2`	"00$a_4a_3a_2$"
`rol`	rotate left.	LMB[1]	`a5 rol 1`	"$a_3a_2a_1a_0a_4$"
`ror`	rotate right.	RMB[1]	`a5 ror 1`	"$a_0a_4a_3a_2a_1$"
`sla`	arithmetic shift left.	RMB	`a5 sla 1`	"$a_3a_2a_1a_0a_0$"
`sra`	arithmetic shift right.	LMB	`a5 sra 1`	"$a_4a_4a_3a_2a_1$"

[1] LMB = Left Most Bit. RMB = Right Most Bit.

Table 6: Shift operators for the `bit_vector` data type.

If you want to perform a shift operations on std_logic_vectors, the simplest way is often to write it in normal VHDL. Then you have also full control of what is shifted in. Examples are given in Table 7.

Example	Result	Operation
`x & b5(4 downto 1);`	"$xb_4b_3b_2b_1$"	shift right, shift in `x`.
`b5(3 downto 0) & x;`	"$b_3b_2b_1b_0x$"	shift left, shift in `x`.
`a5 <= b5(4) & b5(4 downto 1);`	`a5` = "$b_4b_4b_3b_2b_1$"	arithmetic shift right.
`b5(3 downto 0) <= b5(4 downto 1);`	`b5` = "$b_4b_4b_3b_2b_1$"	arithmetic shift right[1].

[1] This should be performed in a process.

Table 7: Shift operators for the `bit_vector` data type.

Since the shift operators exist at all, you can define them also for the std_logic_vector, as illustrated in Code 8.

The packages `ieee.std_logic_signed` and `ieee.std_logic_unsigned` provides some shift functions, listed in Table 8, where `slv`β is a std_logic_vector treated as an unsigned number in all four cases.

4.5 Assignment

Assignments are done using the `<=` operator for signals, and the `:=` operator for variables. The assignment do not return a value. If a value is expected from the formula, the "$\leq$" operator is used instead. Some examples are given in Table 9.

The signal/variable to the left of the assignment must have the same data type as the value to the right. In cases of vectors, the direction should also match.

```
function "sll"(vec:std_logic_vector; n : integer) return std_logic_vector is
begin
  if vec'ascending then -- index defined with "to", and not "downto"
    return vec(vec'left+n to vec'right) & (1 to n => '0');
  else
    return (1 to n => '0') & vec(vec'left to vec'right-n);
  end if;
end "sll";
```

Code 8: Define the "`sll`" operator for the std_logic_vector.

Function	**Operation**	**Shifted in**
`ieee.std_logic_unsigned.SHL(slv`α`, slv`β`)`	Shift `slv`α left `slv`β steps.	`'0'`
`ieee.std_logic_unsigned.SHR(slv`α`, slv`β`)`	Shift `slv`α right `slv`β steps.	`'0'`
`ieee.std_logic_signed.SHL(slv`α`, slv`β`)`	Shift `slv`α left `slv`β steps.	`'0'`
`ieee.std_logic_signed.SHR(slv`α`, slv`β`)`	Shift `slv`α right `slv`β steps.	`MSB`

Table 8: Shift functions for the std_logic_vector data type.

Operation	**Description**
`a <= b;`	**signal** `a` is assigned the content of `b`.
`a := b;`	**variable** `a` is assigned the content of `b`.
`a <= b <= c;`	`b` is compared with `c`, the result (`boolean`) is assigned to `a`.
`a(2) <= b;`	Element 2 from signal `a` is assigned the content of `b`.

Table 9: Four assignment examples.

The assignment operator cannot be overloaded (you can for instance not define the assignment operator that assigns an integer to a `std_logic_vector`).

5 Concurrent Constructions

Concurrent VHDL statements are "executed" continuously, and corresponds to combinational logic.

5.1 When-Else: Multiplexer Net

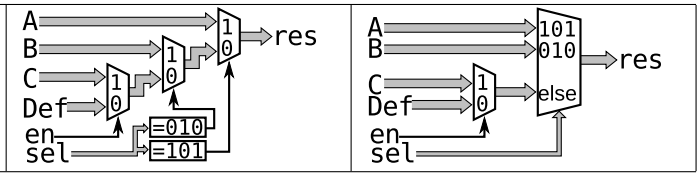
The syntax for the `when else` statement is

{res} `<=` {val1} `when` {cond1} `else` {val2} `when` {cond2} `else ...  else` {val$N$};

- {res} ⇒ Name for the signal that should be assigned.
- {val$n$}, $n = 1, 2, \ldots, N$ ⇒ Data values to select between.
- {cond$n$}, $n = 1, 2, \ldots, (N-1)$ ⇒ Conditions of type Boolean.

If {cond1} is true, then {res} is assigned the value {val1}. Otherwise {cond2} is tested, and so on. If no {cond$n$} is true, {val$N$} is used. See example in Code 9.

```
res <= A when sel = "101" else
       B when sel = "010" else
       C when en = '1' else
       Def;
```



Code 9: When-else: A multiplexer net, in VHDL and as a schematic (before and after optimization).

5.2 With-Select: One Hugh Multiplexer

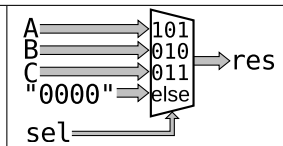
The syntax for the `With-Select` statement is

```
with {expr} select {res} <= {val1} when {choice1}, {val2} when {choice2}, ...  {valN} when others;
```

- {expr} ⇒ Signal or expression to test against.
- {res} ⇒ Signal that should be assigned.
- {val$n$}, $n = 1, 2, \ldots, N$ ⇒ Data values to select between.
- {choice$n$}, $n = 1, 2, \ldots, (N-1)$ ⇒ Select values to compare with {expr}.
- If {expr} = {choice1}, then {res} is assigned the value {val1}.
- Otherwise {expr} = {choice2} is tested, and so on.
- If {expr}≠{choice$n$}, $n = 1, 2, \ldots, (N-1)$, then {val$N$} is used.

See example in Code 10.

```
with sel select
  res <= A when "101",
         B when "010",
         C when "011",
       "0000" when others;
```



Code 10: With-select: One big multiplexer, in VHDL and as a schematic.

5.3 Instantiation Of Other Modules

If we want to use sub modules, they can be implemented according to the examples in Code 11, where (a) instantiates the module in Code 1, and (b) instantiates the module in Code 5

```
architecture
  component and_dff is
    port(clk : in std_logic;
         a,b : in std_logic;
         y : out std_logic);
  end component;
begin
  inst1 : and_dff
    port map(clk => iclk, a=>ia, b=>ib, y=>iy);
  inst2 : and_dff
    port map(iclk, ia, ib, iy);
end architecture;
```

(a)

```
architecture
  component my_add is
    generic(N : integer := 16);
    port(a,b : in unsigned(N-1 downto 0);
         res : out unsigned(N downto 0));
  end component;
begin
  inst1 : my_add
    generic map(N => 12)
    port map(a=>ia, b=>ib, res=>ires);
  inst2 : my_add
    generic map(12)
    port map(ia,ib,ires);
end architecture;
```

(b)

Code 11: Instantiation of modules and_dff and my_add.

The `=> i*` signals are local signals, that are connected to the port on the instantiated module. An error will occure, since the signals `iy` and `ires` are written to from both instances `inst1` and `inst2`. The "`=>`" operator connects the pin with the local signal, independen of the direction of the pin.

In some cases the component declaration is unnecessary.

6 Sequential Constructions

Sequential VHDL statements are "executed" sequential, when something happens on a signal in the sensitivity list in a process, or when a function is called.

In concurrent constructions, a signal that is assigned should always be assigned some value.

In sequential VHDL, a signal does not have to be assigned, and will then keep it's value (by pulling the `en` signal to the DFF/reg low).

6.1 If-Then: Multiplexer Net

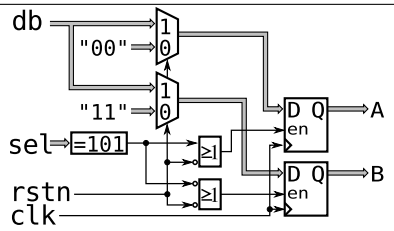
The if statement works like in any programming language (from a programming perspective). The syntax is:

```
if {cond1} then {stats1} elsif {cond2} then {stats2} elsif ...else {statsN} end if;
```

- {cond$n$}, $n = 1, 2, \ldots, N$ ⇒ Conditions of type boolean.
- {stats$n$}, $n = 1, 2, \ldots, (N-1)$ ⇒ Statements that should be "executed".
- `elsif ... then` ⇒ Optional.
- `else` ⇒ Optional.

An example is given in Code 12, where the signal `A` is assigned `"00"` when `rstn='0'`, or the signal `db` when `rstn='1'` and `sel="101"`. In other cases `A` should keeps its value. The enable signal to register `A` will be one if `rstn='0'` or `sel="101"`. The input of the `A` register must be `"00"` when `rstn='0'`, otherwise it should be `db` when `sel="101"`. Since it does not matter what's on the input when `sel`≠`"101"`, it's easiest to simply ignore that condition, and let the input be `db` all the time (except when `rstn='0'`).

```
if rstn = '0' then
  A <= "00";
  B <= "11";
elsif sel="101" then
  A <= db;
else
  B <= db;
end if;
```



Code 12: An if-then statement, and it corresponding net.

6.2 Case-Is: A Hugh Multiplexer

The case-is construction have the syntax:

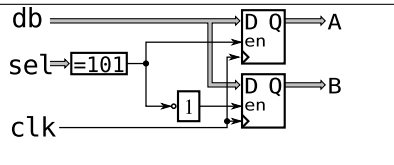
```
case {expr} is when {choice1} => {stats1} when {choice2} => ...  when others => {statsN} end case;
```

- {expr} ⇒ Signal or expression to test against.
- {choice$n$}, $n = 1, 2, \ldots, (N-1)$ ⇒ Constant values to compare with {expr}.
- {stats$n$}, $n = 1, 2, \ldots, N$ ⇒ Statements to execute.

This do not have to result in a big multiplexer, but will most often do so if the same signal is assigned in all `stat`n's. The case-is construction is efficient when building state machines.

An example is given in Code 13, which have the same functionality as Code 12, except the reset.

```
case sel is
  when "101" => A <= db;
  when others => B <= db;
end case;
```



Code 13: A case-is statement, with corresponding net.

6.3 For Loop: Simplifies Some Tasks

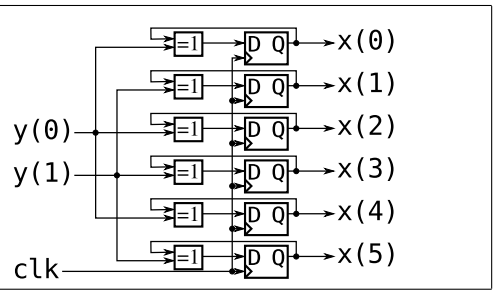
The for loop have the syntax:

{lname} : for {id} in {range} loop {stats} end loop;

- {lname} : ⇒ Optional name for the loop.
- {id} ⇒ An integer, that will step through the range.
- {range} ⇒ A range, like "5 to 9" or "a'range".
- {stats} ⇒ The statements that should be performed.

Code 14 illustrates the use of a for loop, where the operator "mod" is used in a "safe" way (i.e. not resulting in arithmetic hardware).

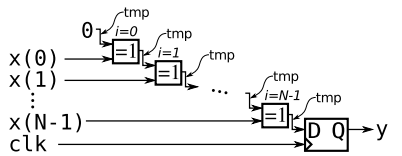
```
for i in 5 downto 0 loop
  y(i) <= y(i) xor x(i mod 2);
end loop;
```



Code 14: A for loop, with corresponding net.

Code 15 illustrates a chain of XOR gates (the "XOR_REDUCE" function).

```
variable tmp : std_logic;
---
tmp := '0';
for i in x'range loop
  tmp := tmp xor x(i);
end loop;
y <= tmp;
```



Code 15: Another for loop, with corresponding net.

6.4 Variables vs Signals in Processes

How signals are treated in processes (that "runs" on clk):

- A signal is global within a module. All processes can access it, but only one should write to it.
- A signal is assigned a value using the "<=" operator.
- A signal corresponds to a DFF (or several, if the signal is a vector).
- Reading from the signal is reading the state of the output of the DFF.
- From a programming perspective, the signal value is updated at "end process;"
- A DFF that is not written to in a clock cycle, should keep it's value ⇒ en = false.
- A DFF that is written to several times in a clock cycle, should get the last written value.

Variables can be used as indexing variables (e.g. i in Code 15), or writing a temporary wire (e.g. tmp).

How variables are treated in processes:

- A variable is local within a process or function.
- A variable is assigned a value using the ":=" operator.
- An assignment of run time data to a variable corresponds to writing to a wire.
- If a variable is assigned several times in the same clock cycle, that corresponds to writing to several wires (as illustrated in Code 15).
- From a programming perspective, the variable value is updated immediately.
- If a variable is not assigned a value, it should keep it's previous value. Hence you *can* use it as a DFF, but you should avoid it.

7 Pipelining

Pipelining is used to speed up logics. A logic gate always contains a small delay (e.g. 3 ns). Lets say we have the combinational net depicted in Fig. 1, which have the following propagation characteristics:

Shortest delay for any signal path	5 ns
Longest delay for any signal path	15 ns

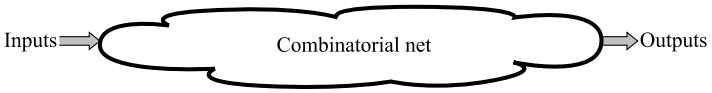


Figure 1: A combinational net.

The signals must have time to propagate from the input register to the output register within one clock cycles. Those the clock frequency can be at most $\frac{1}{15 \text{ ns}} \approx 66$ MHz.

If the requirement is a 150 MHz clock, this delay is not acceptable. We analyze and decompose the net into three different nets, as depicted in Fig. 2, with the following delays:

	Net 1	**Net 2**	**Net 3**
Shortest delay	1.2 ns	2 ns	2.6 ns
Longest delay	6.4 ns	6.3 ns	4.5 ns

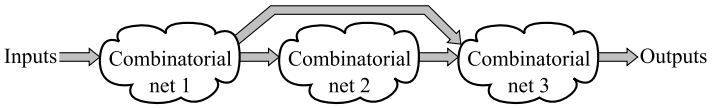


Figure 2: The net decomposed into three different nets.

Now it is time to insert pipeline registers in the buses between the nets, as depicted in Fig. 3.

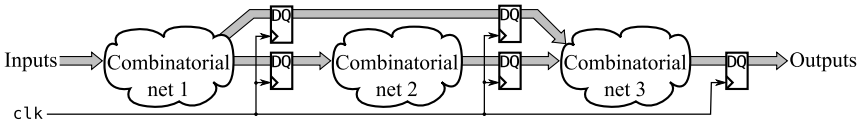


Figure 3: Pipeline registers inserted into the net.

The effect is that the longest delay in the system (from input or register to register or output) is now only 6.4 ns. Hence we can clock the system in at most $\frac{1}{6.4 \text{ ns}} \approx 156$ MHz.

One important change: The output is not available at once. The different stages of the pipeline processes data that is a number of clock cycles late, as depicted in Table 10.

Clock cycle	**net1**	**net 2**	**net 3**	**Output**
1	1	(old)	(old)	(old)
2	2	1	(old)	(old)
3	3	2	1	(old)
4	4	3	2	1
5	5	4	3	2

Table 10: The data sets processed by the different nets from Fig. 3.

This is the main idea with pipelining. Split the design into several stages, with one clock cycle delay per stage. If you need to feed data backwards, it's more tricky. **The pitfall:** Watch out so you don't mix data from different time sets, that belongs in different pipeline stages.

References

[1] `http://www.altera.com/customertraining/webex/VHDL/player.html`

[2] `http://www.csee.umbc.edu/portal/help/VHDL/stdpkg.html`

Appendix A Misc Package Declarations

This appendix aims to give a quick-and-sloppy overview of the `ieee` packages. They are explained more in details in [2].

Some packages are developed by IEEE (who defined the language), and some are developed (and owned) by Synopsys.

Notations in this appendix:

- `sl` ⇒ `std_logic`.
- `slv` ⇒ `std_logic_vector`.
- `int` ⇒ `integer`.
- `nat` ⇒ `natural` (≥ 0).
- `S` ⇒ `signed`.
- `U` ⇒ `unsigned`.
- `SI` ⇒ `S` or `int` or a combination, except `(int,int)`.
- `UN` ⇒ `U` or `nat` or a combination, except `(nat,nat)`.
- `US` ⇒ `U` or `S`.
- $\overset{<=}{>\neq}$ ⇒ the six comparison operators `<`, `<=`, `=`, `/=`, `>=`, `>`.
- *aox* ⇒ the operators `and`, `nand`, `or`, `nor`, `xor` and `xnor`.

By default (without any `library` or `use` statement), VHDL defines basic types (bit, bit_vector, integer, natural, boolean, string, real, time) and some operations on those.

Warning: There are two packages called `ieee.std_logic_arith`. One from IEEE, and one from Synopsys. A computer can have any of them installed, which reduces the portability of the code. Avoid those.

A.1 `ieee.std_logic_1164`

The "standard" package for synthesizable code.

Types

- `std_logic` ⇒ {`'U'`,`'X'`,`'0'`,`'1'`,`'Z'`,`'W'`,`'L'`,`'H'`,`'-'`}.
- `std_logic_vector` ⇒ array (natural range <>) of std_logic.
- ...and more.

Functions/operators

- `not sl`.
- `not slv`.
- `sl` *aox* `sl`.
- `slv` *aox* `slv`.
- `rising_edge(sl)`.
- `falling_edge(sl)`.

A.2 `ieee.numeric_std`

Contains the definitions of the types `SIGNED` and `UNSIGNED`, and the operators on those and on `int`/`nat`.

Types

- `SIGNED`,`UNSIGNED` ⇒ same def as `slv`, but own types.

Functions/operators

- `UN+UN`, `SI+SI`, `+US`.
- `UN-UN`, `SI-SI`, `-S`.
- `UN*UN`, `SI*SI`.
- `UN/UN`, `SI/SI` ⇒ Do not use.
- `UN mod UN`, `SI mod SI` ⇒ Do not use.
- `UN rem UN`, `SI rem SI` ⇒ Do not use.
- `abs S`.
- `UN` $\overset{<=}{>\neq}$ `UN`, `SI` $\overset{<=}{>\neq}$ `SI`.
- `SHIFT_LEFT(US,nat)`, `SHIFT_RIGHT(US,nat)`.
- `ROTATE_LEFT(US,nat)`, `ROTATE_RIGHT(US,nat)`.
- `US sll int`, `US srl int`.
- `US rol int`, `US ror int`.
- `RESIZE(US,nat)`.
- `TO_INTEGER(US)` ⇒ S → int, U → nat.
- `TO_UNSIGNED(nat,nat)` ⇒ First operand = "the signal".
- `TO_SIGNED(int,nat)`.
- `not US` ⇒ bitwise not.
- `U` *aox* `U`, `S` *aox* `S`.
- `STD_LOGIC_VECTOR(...)`, `UNSIGNED(...)`, `SIGNED(...)` ⇒ convert between `S`, `U` and `slv`.

A.3 `ieee.std_logic_arith` (IEEE)

Contains arithmetic definitions for `slv`, treating it as unsigned.

Functions/operators

- `"+"` ⇒ combinations of `slv`, `int` and `sl`.
- `"-"` ⇒ the same combinations.
- `+slv`.
- `"*"` ⇒ combinations of `slv` and `sl`.
- `"/"` ⇒ combinations of `slv` and `sl`.
- `cond_op(boolean,...)` ⇒ mux for `slv` or `sl`.
- `"`$\overset{\leq =}{>\neq}$`"` ⇒ comparison between `slv` and `int`.
- `sh_left(slv,nat)`, `sh_right(slv,nat)` ⇒ shift a number of steps.
- `to_integer(...)` ⇒ convert `slv` or `sl` to integer.
- `To_StdLogicVector(int,nat)`.

A.4 `ieee.std_logic_arith` (Synopsys)

Contains own definitions of the types `SIGNED` and `UNSIGNED`, and a huge set of operators on those and on `int`.

Types

- `SIGNED`,`UNSIGNED` ⇒ same def as `slv`, but own types.

Functions/operators

All those functions, involving `S` will return `S`, the rest will return `U`. All functions also exist in a version returning `slv`. The comparison functions only returns boolean.

- (no logical operators are defined)
- `"+"` ⇒ all combinations of `S`, `U`, `int` and `sl`. No combinations of only `int` and `sl` are available.
- `"-"` ⇒ the same combinations.
- `+U`, `+S`.
- `-S`.
- `abs(S)`.
- `"*"` ⇒ all combinations of `S` and `U`.
- `"`$\overset{\leq =}{>\neq}$`"` ⇒ comparison operators for `S`, `U`, `int` and any combination of them. Returns only boolean.
- `SHL(...,U)` ⇒ shift `S` or `U` left a number of steps.
- `SHR(...,U)` ⇒ shift `S` or `U` right a number of steps.
- `CONV_INTEGER(...)` ⇒ convert `int`, `U`, `S` or `sl` to `int`.
- `CONV_UNSIGNED(..., int)`, `CONV_SIGNED(..., int)` ⇒ convert `int`, `U`, `S` or `sl` to `U` or `S`.
- `CONV_STD_LOGIC_VECTOR(..., int)` ⇒ convert `int`, `U`, `S` or `sl` to `U` or `slv`.
- `UNSIGNED(...)`, `SIGNED(...)`, `STD_LOGIC_VECTOR(...)` ⇒ convert between `S`, `U` and `slv`.
- `EXT(slv,int)`, `SXT(slv,int)` ⇒ zero resp. sign extend tt slv.

A.5 `ieee.std_logic_misc` (Synopsys)

Contains miscellaneous functions for `slv`, whereof the interesting functions are listed here.

Functions/operators

- *aox*`_REDUCE(slv)` ⇒ A big *aox* gate, fed with all bits in a `slv`. E.g. `NOR_REDUCE(x)` is zero if any bit in `x` is 1.

A.6 `ieee.std_logic_unsigned` and `ieee.std_logic_signed` (Synopsys)

Arithmetic operations on slv, that treat those as unsigned and signed values respectively.

Functions/operators

- `slv+slv`, `slv+int`, `int+slv`,`slv+sl`,`sl+slv`.
- `+ slv` (unary operator).
- `- slv` (only in `signed`).
- `ABS(slv)` (only in `signed`).
- `slv-slv`, `slv-int`, `int-slv`,`slv-sl`,`sl-slv`.
- `slv*slv`.
- `slv`$\overset{\leq =}{>\neq}$`slv`, `slv`$\overset{\leq =}{>\neq}$`int`, `int`$\overset{\leq =}{>\neq}$`slv`.
- `SHL(slv,slv)`.
- `SHR(slv,slv)`.
- `CONV_INTEGER(slv)`.