TSIU03: A Tiny 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).

Contents

1 Introduction

This document is a very brief VHDL summary, intended as a simple non-covering help during the labs.

1.1 A Simple Example

A simple example of a VHDL file is depicted in Code 1.

- **library,use ieee.std_logic_1164.all;** ⇒ Access the standard types and functions defined in VHDL.
- **entity port(...); end entity;** ⇒ Defines the "public interface".
- **std_logic** ⇒ "*The*" data type for digital logic. Mostly '0' or '1'.
- **architecture ... end architecture** ⇒ The "engine".
- **RTL** stands for "Register Transfer Level".
- **signal foo : std_logic;** ⇒ Declares an internal signal.
- **process(clk)**, **rising_edge(clk)** ⇒ Generates the D-flip-flop.
- **foo <= a and b;** ⇒ The AND gate, assigned to the DFF.
- **y <= foo;** ⇒ direct connection.

```
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.

1.2.2 Behavioral VHDL

Behavioral VHDL is used for simulation only. In addition to what can be described as RTL code, it can use much more complex constructions, e.g. file access.

1.3 Concurrent vs Sequential Syntax

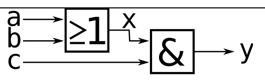
VHDL code can, in some sense, be divided into concurrent and sequential code.

By default, the code in the architecture is concurrent. Each statement corresponds to a hardware block. You can have processes, and within those, the code is sequential.

1.3.1 Concurrent VHDL

Concurrent VHDL will always generate combinational logic.

Code 2 shows three ways of writing the logic net in (d). The intermediate signal `x` is not defined in (c).

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

Code 2: Some examples of the same thing.

1.3.2 Sequential VHDL

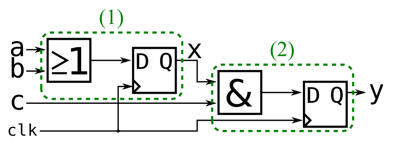
Use processes to generate registers, DFFs etc. The code in the process is understandable if you think it as a sequential execution, that starts on the clock edge, and where all assignments are updated when the "execution" is done. Versions (a) and (b) in Code 3 gives the behavior depicted in (c).

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

(a)

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

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

The type `std_logic` has binary values, as `'0'`, `'1'` or `'-'` (don't care).

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, e.g. `(0 to 2)` or `(5 downto 1)`.

Constants are given as `"1001"`. Hexadecimal constants can be written as `X"a3"`.

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.

3 Declarations and Definitions

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;` – Simplified access to `CONV_INTEGER`.
- `use ieee.std_logic_signed."+";` – The "+" operator (e.g. `a + b`).

Without the `use` command, you can access the "+" operator as `ieee.std_logic_signed."+"(a,b)` instead of `"a+b"`.

You can find a good list of the standard packages, and what they contains on the web page [1], and in App A.

3.2 Entity Definitions

The entity describes the module I/O pins. The definition usually looks like:

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

- {ename} ⇒ The name of the entity.
- {plist} ⇒ A list of design "pins", on the form "`a1,a2,... :` {dir} `typeA; b1,b2,... :` {dir} `typeB; ...`".
- {dir} ⇒ the direction of the pins. Typically `in` or `out`. You can read from (but not write to) an `in` pin. You can write to (but not read from) an `out` pin.

3.3 Architecture Definitions

The architecture is like the engine. 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} ⇒ Declare/define signals, functions, aliases, constants, component etc. here.
- {body} ⇒ Here is the body of the architecture – the logic definition.

3.4 Signal Declarations

Signals are declared before the `begin` in the architecture. It can look like in Code 4.

```
architecture rtl of foo is
  signal sl1,sl2 : std_logic;                   -- two signals of type std_logic
  signal sl3 : std_logic := ’0’;                -- initiates to ’0’
  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 4: Examples of signal declaration.

3.5 Process Definitions

A process is placed in the concurrent code, and contains sequential code.

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

- {pname} : ⇒ An optional name of the process.
- {sensitivity list} ⇒ A list of signals that should “trig the process to start”. Most often just `(clk)`, or `(clk,reset)`.
- {body} ⇒ The sequential code.

4 Basic VHDL

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

4.1 Assignments

The “<=” operator is used to assign signals.

4.2 Logic Operations

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

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

Example of a multiplexer implemented with logic gates:

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

4.3 Arithmetic Operations

You can use arithmetic operations like `a+b`, `a-b`, `-a` or `a*b`.

Those operations works on numerical data types, like `std_logic_vector`, `signed` or `unsigned`. 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).

4.4 Test Operations

Those operates on numerical data types. The operations returns the data type boolean, used by, e.g. `if` statements.

- `=`, `/=` $\Rightarrow$ Equal or not equal. Those also works on `std_logic`.
- `<`, `<=` $\Rightarrow$ Less than (or equal).
- `>`, `>=` $\Rightarrow$ Greater than (or equal).

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

4.5 Vectors and Indexing

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

We use the signals in Table 1 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)	"$b_{n-1} \dots b_0$"
	Examples	
a4	std_logic_vector(3 downto 0)	"$a_3a_2a_1a_0$"

Table 1: Declaration of signals used in examples.

4.5.1 Vector Indexing

Indexing is illustrated by the examples in Table 2.

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 = "$a_4 10 a_1a_0$"

Table 2: Examples of vector indexing.

4.5.2 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 3

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 3: Examples of vector concatenation

4.5.3 Aggregation: The “`(others=>’0’)`” Syntax

In assignments, you can fill the target signal with, e.g., zeros, by `a5 <= (others=>’0’);`

4.5.4 Shifting

The easiest way of shifting is to use a combination of aggregation and indexing, as in Table 4.

Example	**Result**	**Operation**
`x & b5(4 downto 1);`	"$xb_4b_3b_2b_1$"	shift in `x` from left (right shift).
`b5(3 downto 0) & x;`	"$b_3b_2b_1b_0x$"	shift in `x` from right (left shift).
`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, or $b_0 = b_1 = b_2 = b_3 = b_4$, e.g. just a wire with five names.

Table 4: Shift operators for the `bit_vector` data type.

5 Concurrent Constructions

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

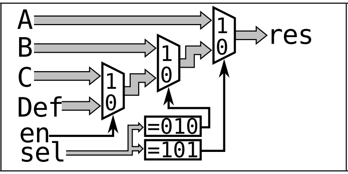
5.1 When-Else: Multiplexer Net

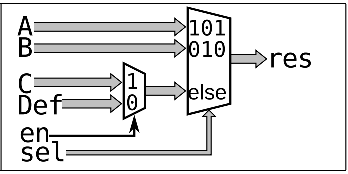
The syntax for the `when else` assignment is

```
{res} <= {val1} when {cond1} else {val2} when {cond2} else ... else {valN};
```

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 5.

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





Code 5: When-else: A multiplexer net, in VHDL and as a schematic (before and after optimization).

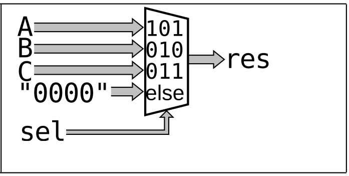
5.2 With-Select: One Hugh Multiplexer

The syntax for the `With-Select` statement is

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

- 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.

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



Code 6: With-select: One big multiplexer, in VHDL and as a schematic.

6 Sequential Constructions

In sequential VHDL, the signal assignments are made to the input of DFFs/regs. If a signal is not assigned during a clock cycle, it will keep it's value (by pulling the `en` signal to the DFF/reg low).

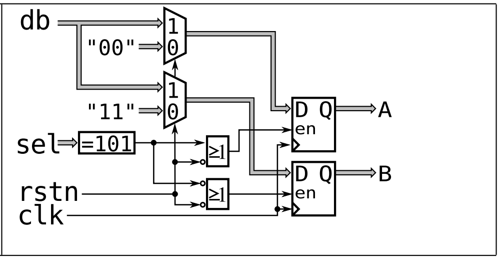
6.1 If-Then: "Multiplexer Net"

The if statement works like in any programming language. 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.

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



Code 7: An if-then statement, and it corresponding net.

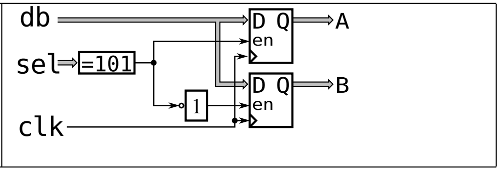
6.2 Case-Is: "A Hugh Multiplexer"

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.

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



Code 8: A case-is statement, with corresponding net.

References

[1] 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 some `ieee` packages. They are explained more in details in [1].

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` ⇒ An integer (e.g. `48`).
- `S` ⇒ `signed`.
- `U` ⇒ `unsigned`.
- `US` ⇒ `U` or `S`.
- $\substack{\leq =\\ > \neq}$ ⇒ `<`, `<=`, `=`, `/=`, `>=` or `>`.
- *aox* ⇒ `and`, `or`, `xor`, `nand`, `nor` or `xnor`.

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 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.

Types

- `SIGNED`,`UNSIGNED` ⇒ identical definition as `slv`, but own types.

Functions/operators

- `U+U`, `S+S`, `+US`.
- `U-U`, `S-S`, `-S`.
- `U*U`, `S*S`.
- `abs S`.
- `RESIZE(US,int)`.
- `TO_UNSIGNED(int,int)` ⇒ 2nd arg = size.
- `TO_SIGNED(int,int)`.
- `U`$\substack{\leq =\\ > \neq}$`U`, `S`$\substack{\leq =\\ > \neq}$`S`, `US`$\substack{\leq =\\ > \neq}$`int`, `int`$\substack{\leq =\\ > \neq}$`US`
- `not U`, `not S` ⇒ bitwise not.
- `U` *aox* `U`, `S` *aox* `S` ⇒ logic operators.
- `STD_LOGIC_VECTOR(...)`, `UNSIGNED(...)`, `SIGNED(...)` ⇒ convert between `S`, `U` and `slv`.

A.3 `ieee.std_logic_(un)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-slv`, `slv-int`, `int-slv`, `slv-sl`, `sl-slv`.
- `slv*slv`.
- `-slv` ⇒ only in `std_logic_signed`.
- `ABS(slv)` ⇒ only in `std_logic_signed`.
- `slv`$\substack{\leq =\\ > \neq}$`slv`, `slv`$\substack{\leq =\\ > \neq}$`int`, `int`$\substack{\leq =\\ > \neq}$`slv`.