vhdl-cheatsheet

summarizes most important things to recognize about vhdl

librarys

```
library IEEE;
use IEEE.std_logic_1164.all;      -- contains std_logic and std_logic_vector (and more)
use IEEE.numeric_std.all;         -- contains unsigned and signed arithmetics
```

entities

```
entity multiplexer is
  port(
    a, b: in std_logic;
    s: in std_logic;
    c: out std_logic    -- no semicolon here!
  );
end entity multiplexer;
```

- blackbox view defining inputs and outputs

architectures

```
architecture multiplexer_logic of multiplexer is
begin
  c <= a when s = '0' else
```

```
       b when s = '1';
end multiplexer_logic;
```

- whitebox view defining the logic, always belonging to an entity
- there can be multiple architectures for a single entity
- you cannot write to inputs!
- you cannot read from outputs!
- all assignments are processes!
  - all processes happen in parallel!
  - a signal takes its state at the end of a process!
  - that means that assigning a value to a signal in one process won't affect another process using that signal (not in the same "loop")!
- use signals to reuse your "output"!

types of architectures

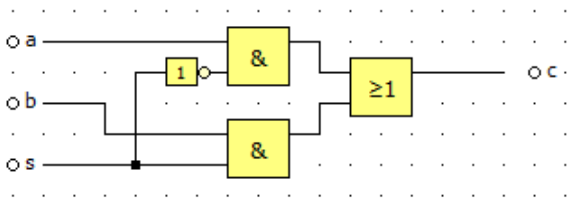
There are three types of architectures. I'll explain them by implementing following 1-mux:

```
entity multiplexer is
  port(
    a, b: in std_logic;
    s: in std_logic;
    c: out std_logic
  );
end entity multiplexer;
```

1. structural

"The description of a structural body is based on component instantiation and generate statements."

That means first of all we create a component for each logic gate we need. Then we use these logic gates to exactly define the multiplexer's structure.




```
architecture structural of multiplexer is

component and_gate
  port(a, b: in std_logic; c: out std_logic);
end component;
component or_gate
  port(a, b: in std_logic; c: out std_logic);
end component;
component not_gate
```

```
  port(a: in std_logic; b: out std_logic);
end component;

signal s_not: std_logic;
signal first_and: std_logic;
signal second_and: std_logic;

begin
   n1: not_gate port map(s, s_not);
   a1: and_gate port map(a, s_not, first_and);
   a2: and_gate port map(b, s, second_and);
   o1: or_gate port map(first_and, second_and, c);
end structural;
```

2. dataflow

"The Dataflow description is built with concurrent signal assignment statements. Each of the statements can be activated when any of its input signals changes its value."

That means we just use simple assignments and logical operators.

```
architecture dataflow of multiplexer is

begin
   c <= (a and not s) or (b and s);
end dataflow;
```

3. behavioural

"The architecture body describes only the expected functionality (behavior) of the circuit, without any direct indication as to the hardware implementation. Such description consists only of one or more processes, each of which contains sequential statements."

```
architecture behavioural of multiplexer is

begin
   process(a, b, s)
   begin
      if(s = '1') then
        c <= b;
      else
        c <= a;
      end if;
   end process;
end behavioural;
```

reusing entities

```
architecture structural of full_adder is
signal s1, s2, s3: bit;
component half_adder  -- 1. define the reused component's black box view inside the architectur
```

```
  port (
    x, y : in bit;
    sum, cout : out bit
  );
end component;
begin
  u1 : half_adder port map (x, y, s1, s2);  -- 2. map (input/output) signals to the reused comp
  u2 : half_adder port map (s1, cin, sum, s3);
  cout <= s2 OR s3;
end structural;
```

signals

```
architecture some_random_architecture of some_random_entity is
signal internal_signal: std_logic;
begin
  internal_signal <= a xor internal_signal;
  c <= internal_signal;
end some_random_architecture;
```

- useful for reusing "outputs"
- are declared inside the architecture's declarative part

initialising signals

```
signal internal_signal: std_logic := '1';
```

- remember: intialization isn't synthesizable!

processes

```
process(clk)
begin
  if(rising_edge(clk) then
    a <= '1';
  else
    a <= '0';
  end if;
end process;
```

- processes are only "called" if the value of at least one signal in the sensitivity list changes
- common assignments (outside a process) are processes too, all processes happen in parallel!
- signals take their states at the end of a process

if-statements

```
if (a = '1') or (c > d) then
  statements;
elsif condition then
  statements;
else
  statements;
end if;
```

- can only be used inside processes
- elsif and else clauses are optional

case-statements

```
case a is
  when "00" => b <= "1000";
  when "01" => b <= "0100";
  when "10" => b <= "0010";
  when "11" => b <= "0001";
  when others => b <= "0000"; -- fallback case
  -- when others =>; -- commonly used to do "nothing"
end case;
```

assignments

simple assignments

```
a <= '1';
```

when else assignments

```
a <= "11" when b = "00" else
     "10" when b = "01" else
     "01" when b = "10" else
     "00" when b = "11";
```

its also possible to define a fallback value

```
a <= "11" when b = "00" else
     "10" when b = "01" else
     "01" when b = "10" else
     "00" when b = "11" else
     "00";
```

with select assignments

this is equal to the code above

```
with b select a <=
     "11" when "00",  -- use commas here
     "10" when "01",
     "01" when "10",
     "00" when "11",
     "00" when others; -- fallback value
```

operators

arithmetical

`+`, `-`, `/`, `*`, `mod`

relational

`=`, `<`, `<=`, `>`, `>=`

logical

`not`, `or`, `and`, `xor`, `nand`, `nor` and `xnor`.
`not` has the highest precendence, all other operators have the same precendence (!). so just use parenthesis^^.

```
c <= a or not b or (f and (g xor h));
```

concatentation-operator

`&` is an operator allowing you to concatenate vectors.

```
signal a: std_logic_vector(3 downto 0);
signal b: std_logic_vector(3 downto 0);
signal c: std_logic_vector(8 downto 0);
(...)
c <= a & b;
```

datatypes

simple

boolean

can be `true` or `false`. used for conditions.

bit

can be `1` or `0`. represents a signal.

std_logic

the better bit. can be `0` (logic 0, all fine), `1` (logic 1, all fine), `Z` (high impendence), `-` (don't care), `U` (uninitialised, cannot know), `X` (unknown, can't determine), `L` (weak signal, probably 0), `H` (weak signal, probably 1), `W` (weak signal, can't tell).
useful for detecting errors while simulating.

vectors / arrays

vectors are array-like one-dimensional collections of signals having the same type. they are indexed by an integer number starting with 0.

defining ranges

- `downto` corresponds to *little endian*
- `to` correspons to *big endian*

e.g.

```
signal big_endian : std_logic_vector(0 to 7) := "0000_0001";        -- results in index 7 havir
signal little_endian : std_logic_vector(7 downto 0) := "0000_0001"; -- results in index 0 havir
```

assigning arrays

assigning single elements

```
my_array(0) <= '1';
```

assigning a part of the array

```
my_array(3 downto 0) <= "0101";
```

positional association

```
my_array <= ('0', '1', '0', '1');
```

named association

```
my_array <= (0 => '1', others => '0');            -- 0000_0001
my_array <= (0 | 3 | 5 => '1', others => '0');    -- 0010_1001
my_array <= (4 downto 1 => '1', others => '0');   -- 0001_1110
```

creating custom vector types

first you have to define your array type:

```
type 4bit_vector is array(3 downto 0) of bit;
```

after that you can simple use it:

```
signal a: 4bit_vector;
```

std_logic_vector

represents a vector of bits (std_logic). does not support arithmetics.

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

unsigned

represents a vector of bits (std_logic) supporting unsigned arithmetics.

```
signal a: unsigned(7 downto 0) := 7;  -- default value is 'U' (uninitialised) to every bit
(...)
a <= a + 1;
```

if you use arithmetical operators on two unsigned signals, remember that they should have the same bit count!

```
signal a: unsigned(7 downto 0) := 17;
signal b: unsigned(4 downto 0) := 2;
(...)
a <= a + b; -- doesn't work
a <= a + ("000"&b); -- does work
```

signed

represents a vector of bits (std_logic) supporting signed arithmetics.

```
signal a: signed(7 downto 0) := -7;  -- default value is 'U' (uninitialised) to every bit
(...)
a <= a + 1;
```

if you use arithmetical operators on two signed signals, remember that they should have the same bit count!

```
signal a: signed(7 downto 0) := 17;
signal b: signed(4 downto 0) := 2;
(...)
a <= a + b; -- doesn't work
a <= a + ("000"&b); -- does work
```

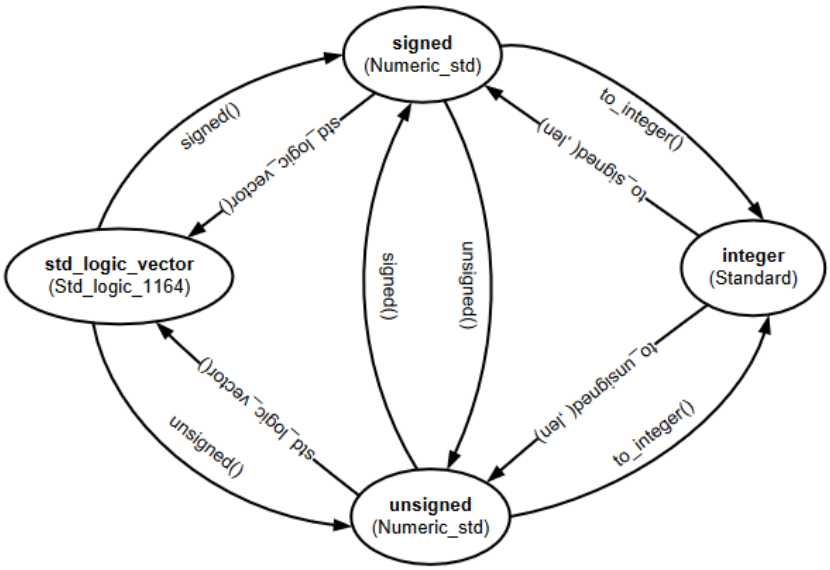
integer

represents an integer. similar to signed/unsigned.
but you

- cannot assign binary or hex values
- cannot use bitwise/logical operators
- cannot set the bit count using `(8 downto 0)` or `(0 to 3)`
  - instead you have to use `range value1 to value2`-syntax
- cannot access single bits

```
signal a_number: integer range -256 to 255;
```

converting between std_logic_vector, unsigned, signed and integer



Source

ghdl

```
sudo apt install ghdl
sudo apt install ghdl-gcc
```

gtkwave

```
sudo apt install gtkwave
```

testing

introduction to testbenches

To test our entities and their architectures we need *testbenches*. Testbenches are nothing more than entities feeding the entities we want to test with different inputs.
With the help of a few useful tools, we can create and run a simulation on the basis of our testbench and precisely analyse which inputs have led to which outputs.

Creating testbenches is not an exact science, as there are countless ways to approach it. In the following chapters I'm going to explain one simple practical example. I assume we want to test the following 1mux:

```
entity multiplexer is
  port(
    a, b: in std_logic;
    s: in std_logic;
    c: out std_logic
  );
end entity multiplexer;
```

the testbench's outer structure

```
entity testbench is
end entity testbench;

architecture testbench_logic of testbench is

component multiplexer
  port(
    a, b: in std_logic;
    s: in std_logic;
    c: out std_logic
  );
end component;

-- TODO

begin
  -- TODO
end testbench_logic;
```

- the testbench should be defined in a second file, you do **not** have to import the entity you want to test
- testbenches have no ports as they generate the input for our entities inside their architecture

- you have to embed the entity you want to test as component inside the testbench's architecture

generating input

Generating input is the key concept behind testbenches. First of all we have to determine which inputs are possible.

a	b	s
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

The 1mux has 7 different possible input combinations. To generate them, I use an unsigned (integer) having 3 bits (don't forget to import the library!).

```
(...)
signal input: unsigned(2 downto 0) := 0;
(...)
```

Now I use a process to count (binary) upwards.

```
(...)
process
begin
  input <= input + 1;
end process;
(...)
```

However, as processes are constantly re-executed, this incrementing happens at such a fast rate that it would hardly be visible in our simulation.
To fix this issue I use the `wait for <time><time unit>` statement. This statement tells our simulation to pause for some time before re-executing the process.

```
process
begin
  wait for 10ns;
  input <= input + 1;
end process;
```

mapping our generated input to our component

Now that we have successfully created our input signals, we need to feed them to our component. We accomplish this through port mapping.
In addition we need to store our component's output, that's why I declare another signal.

```
signal input: unsigned(2 downto 0);
signal output: std_logic; -- new signal storing our component's output
```

```
    (...)
  end process;

  m1: multiplexer port map(a => input(2), b => input(1), s => input(0), c => output);
end testbench_logic;
```

compiling files

To run our simulation, we first need to compile all the vhdl files involved.

```
ghdl-gcc -a *.vhd
```

generating the simulator

After that, we need to generate an executable program from our testbench.

```
ghdl-gcc -e testbench
```

- use the testbench's entity name

running the simulation

Finally we can run the simulation. The stop time must be chosen carefully. Since I test 7 different inputs and wait 10ns each time, I simulate 70ns.

```
ghdl-gcc -r testbench --vcd=output.vcd --stop-time=70ns
```

- use the testbench's entity name

gtkwave

The simulation's results are written to a .vcd file. *gtkwave* is a simple tool visualizing the results.

```
gtkwave file.vcd
```

- quick-tip: use WaveTrace plugin on vscode

links

- VHDL Online Help
- University of Minnesota Duluth PDF
- VHDL Grundlagen, Universität Ulm