EEC 242 - VHDL cheat sheet[1]
Chapter 1 and 2

1 Chapter 1

- VHDL ⇒ **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage
- VHDL is a concurrent language.
- VHDL is not case sensitive.
- Purpose:
  - Documentation
  - Simulation
- Documentation : write a comment
  Characters "- -"

1.1 Entity

- Entity deceleration defines the name of the entity and lists the input and output ports.

```
entity example1 is
   port(x1, x2, x3     :in    std_logic;
        in1            :in    integer;
        val1           :out   std_logic;
        val2           :out   std_logic_vector(3 downto 0));
end example1;
```

1.1.1 Signals Names

- Signal Name specifies external interface signals
- It can be any alphanumeric character
- Rules:
  - Must begin with a letter (Don't begin with number or _)
  - Don't end with underscore _
  - No two successive underscores _ _

1.1.2 Modes

- Indicate signal direction:

```
- in      : input
- out     : output   (whose value can only be read by other entities)
- buffer  : output   (whose value can be read inside the entity's architecture)
- inout   : can be an input or an output.
```

1.1.3 Types

- A built in or user defined signal type:

```
- bit              : 0 or 1.
- bit_vector       : vector of bits.
- std_logic        : 0, 1, Z, -, L, H, U, X, W.
- std_logic_vector : vector of std_logics.
- boolean          : TRUE or FALSE.
- integer          : integer numbers.
- real             : real numbers.
- character        : character.
- time             : time.
```

- bit & bit_vector :
  - bit : 0 or 1.
  - ```
    port (C    : in bit;
          BYTE : in bit);
    ```
  - bit_vector : multi-bit data
  - Diffrence between `to` and `downto`

    ```
    port (to_example     : out bit-vector(1 to 8);
          downto_example : out bit-vector(7 downto 0));
    ```
  - `to_example <= "10011000"` results in

    ```
    to_example(1) = 1
    to_example(2) = 0
    to_example(3) = 0
    to_example(4) = 1
    to_example(5) = 1
    to_example(6) = 0
    to_example(7) = 0
    to_example(8) = 0
    ```

[1] Taha Ahmed
    ```

  - downto_example <= "10011000" results in
    downto_example(7) = 1
    downto_example(6) = 0
    downto_example(5) = 0
    downto_example(4) = 1
    downto_example(3) = 1
    downto_example(2) = 0
    downto_example(1) = 0
    downto_example(0) = 0

- std_logic & std_logic_vector.
  - to use this type

    ```
    Library IEEE;
    Use IEEE.std_logic_1164.all;
    ```

  - 1
    0
    Z: high impedance
    -: don't care
    U: uninitialized
    X: unknown
    W: weak unknown
    L: weak high
    H: weak low

- Integer:
  - represents a binary number
  - by default, it has 32 bits, range from $-(2^{31}-1)$ to $2^{31}-1$.
  - you can declare integer with fewer bits by Range keyword :

    ```
    Integer Range -127 to 127;
    ```

- Generic:

  - ```
    entity example1 is
       Generic (Delay : Time := 10ns);
       port(x1, x2, x3   :in    std_logic;
            in1          :in    std_logic;
            in2          :in    std_logic;
            output       :out   std_logic;
            val2         :out   std_logic_vector(3 downto 0))
    end example1;
    Architecture Behavior of example1 is
    begin
    ...
     output  <=  in1 or in2 after Delay;
    ...
    end Behavior;

    ____________

    entity CPU is
       Generic (BusWidth : Integer := 16);
       port(DataBus : inout std_logic_vector (BusWidth downto 0));
    end Cpu;
    ```

1.2 Architecture

- specifies how the circuit operates and how it is implemented.
- Ways
  - Behavioral (Data flow / Sequential)
  - Structural
  - Combination of both

1.2.1 Operators

- From Highest Precedence to Lowest:
  ** : exponentiation
  ABS
  Not
  *
  /
  MOD
  REM
  +
  -
  & : Concatenation
  =
  /= : Not equal
    ```

```
<
<=
>
>=
AND
OR
NAND
NOR
XOR
XNOR
NOT
```

```
entity example1 is
   port(x1, x2, x3   :in    std_logic;
        output       :out   std_logic);
end example1;

Architecture Behavior of example1 is
begin

 output  <=  (x1 and x2) or ( not x2 and x3);

end Behavior;
```

1.2.2 Signals

- Represents logic signals or wires in a circuit.
- Signals assignment operator `<=`
- `Signal signal_name :  type;`
-
```
entity example1 is
   port(x1, x2, x3   :in    std_logic;
        output       :out   std_logic);
end example1;

Architecture Behavior of example1 is
Signal A, B : std_logic;
begin

A <= x1 and x2;
B <= not x2 and x3;
output  <=  A or B;

end Behavior;
```

1.3 Concurrent Assignment Statements

- to **assign a value to a signal** in an architecture body.
- Four types:
  - Simple signal assignment.
  - Selected signal assignment.
  - Conditional signal assignment.
  - Generate statements.

1.3.1 Simple Signal Assignment

- For a logic or arithmetic expression.

```
output  <=  (x1 and x2) or ( not x2 and x3);
```

- Assign using Others:
  Instead of writing `S <= "00000000";`
  write `S <= (others => '0');`

1.3.2 Selected Signal Assignment

- Assign the value of a signal to a **one of several alternatives based on a selection** criterion

```
WITH some_selector SELECT
            output <= x1 WHEN '0',
                      x2 WHEN OTHERS;
```

1.3.3 Conditional Signal Assignment

- Assign the value of a signal to a **one of several alternatives based on a Condition**

```
    output <= '1' WHEN x1 = x2 ELSE
                  '0';
```

```vhdl
-- Priority Circuit

-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity priority is
   port(x1, x2, x3   :in    std_logic;
        output       :out   std_logic_vector (1 downto 0));
end priority;

-- Architecture
Architecture Behavior of priority is

begin

output  <=  "01" when x1 = '1' else
            "10" when x2 = '1' else
            "11" when x3 = '1' else
            "00" ;


end Behavior;
```

1.3.4 Generate Signal Assignment

- types
  - `for` generate.
  - `if` generate.

```vhdl
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity example is
   port(input_signal     :in    std_logic_vector (15 downto 0);
        first_output     :out   std_logic_vector (7 downto 0);
        second_output    :out   std_logic_vector (0 to 7));
end example;

-- Architecture
Architecture Behavior of example is

begin

GENERATE_EXAMPLE : for i in 0 to 7 generate
                       first_output(i) <= input_signal(i);
                       second_output(7 - i) <= input_signal(i + 8);

end generate GENERATE_EXAMPLE;

end Behavior;

-- take a moment and try to understand this code.
-- the code breaks the input to two halves, assign
-- the first half to the first out and
-- the reverse of the second halve to the second out
-- input = "1110101100010110"
-- first_output = "00010110"
-- second_output = "11010111"
```

2 Chapter 2

2.1 Components

|

- Intro:
  VHDL entity defined in one file can be used as a subcircuit in another file.
- The subcircuit is called a component.
- Where is the deceleration?
  in the declaration region of an architecture.
- Example: Full adder

```
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity fullAdder1bit is
   port(x, y, carryIn     :in    std_logic;
        sum, carryOut     :out   std_logic);
end fullAdder1bit;

-- Architecture
Architecture Behavior of example is

begin

sum <= x xor y xor carryIn;
carryOut <= (x and y) or (x and carryIn) or (y and carryIn);

end Behavior;
```

- Example: 4 bit Full adder

```
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity fullAdder4bit is
   port(x, y     :in    std_logic_vector(3 downto 0);
        carryIn  :in    std_logic;
        sum      :out   std_logic_vector(3 downto 0);
        carryOut :out   std_logic);
end fullAdder4bit;

-- Architecture
Architecture Behavior of example is

-- Here we declare an intermediate signal to help us to contain carry of each stage

Signal helpMe :  std_logic_vector(1 to 3);

-- Here we declare a component, our 1 bit full adder

Component fullAdder1bit
   port(x, y, carryIn     :in    std_logic;
        sum, carryOut     :out   std_logic);
end Component;

begin

stage0 : fullAdder1bit PORT MAP (carryIn, x(0), y(0), sum(0), helpMe(1));
stage1 : fullAdder1bit PORT MAP (helpMe(1), x(1), y(1), sum(1), helpMe(2));
stage2 : fullAdder1bit PORT MAP (helpMe(2), x(2), y(2), sum(2), helpMe(3));
stage3 : fullAdder1bit PORT MAP (helpMe(3), x(3), y(3), sum(3), carryOut);

-- YOU CAN ALSO TYPE
-- stage3 : fullAdder1bit PORT MAP (x => x(3), y => y(3),
--                        carryIn => helpMe(3), sum => sum(3), carryOut => carryOut);

end Behavior;
```

- Example: 4 bit Full adder (another implementation using for generate)

```
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity fullAdder4bit is
   port(x, y     :in    std_logic_vector(3 downto 0);
        carryIn  :in    std_logic;
        sum      :out   std_logic_vector(3 downto 0);
        carryOut :out   std_logic);
end fullAdder4bit;

-- Architecture
```

```
Architecture Structure of example is

-- Here we declare an intermediate signal to help us to contain carry of each stage

Signal helpMe :  std_logic_vector(0 to 4)


Component fullAdder1bit
   port(x, y, carryIn     :in    std_logic;
        sum, carryOut     :out   std_logic);
end Component;


begin

helpMe(0) <= CarryIn;

GENERATE_LABEL : for i in 0 to 3 generate
                    fullAdder1bit PORT MAP (helpMe(i), x(i), y(i), sum(i), helpMe(i + 1));

end generate GENERATE_LABEL;
carryOut <= helpMe(4);
end Structure;
```

- Example: 4 bit Full adder (another implementation)

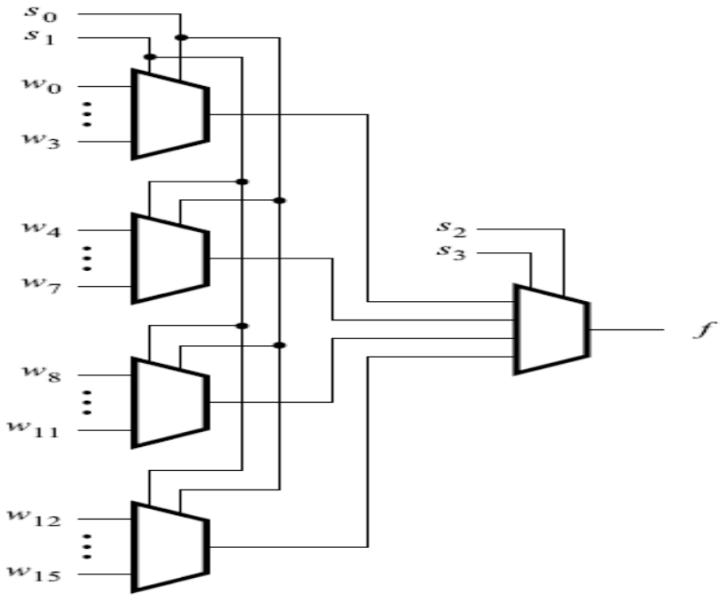
```
Library IEEE;
Use IEEE.std_logic_1164.all;
Use IEEE.std_logic_signed.all;

Entity fullAdder4bit is
   port(x, y     :in    std_logic_vector(3 downto 0);
        carryIn  :in    std_logic;
        sum      :out   std_logic_vector(3 downto 0);
        carryOut :out   std_logic);
end fullAdder4bit;


Architecture Behavior of example is

-- Here we declare an intermediate signal to help us, it will be the summation of x, y and carryIn, then we will break it to s

Signal helpMeSum :  std_logic_vector(4 downto 0);

-- No need to components here!

begin

helpMeSum <= ('0' & x) + y + carryIn;
sum <= helpMeSum(3 downto 0);
carryOut <= helpMeSum(4);

end Behavior;
```

- Example: Multiplexers 4to1

```
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity mux4to1 is
   port(w0,w1,w2,w3     :in    std_logic;
        selector        :in    std_logic_vector(1 downto 0);
        output          :out   std_logic);
end mux4to1;

-- Architecture
Architecture Behavior of mux4to1 is

begin

With selector select
        output <= w0 when "00",
              w1 when "01",
              w2 when "10",
```

```
              w3 when others;
end Behavior;
```

- Example: Multiplexers 16to1 using 4to1 mux



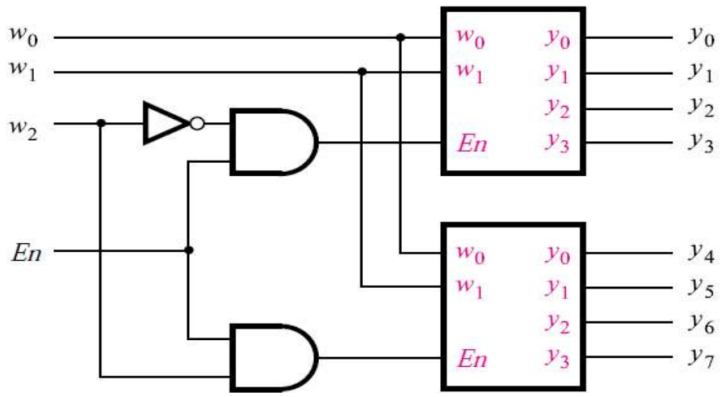

```
-- Libraries to use std_logic
Library IEEE;
Use IEEE.std_logic_1164.all;

-- Entity
Entity mux16to1 is
   port(w           :in    std_logic_vector (0 to 15);
        selector    :in    std_logic_vector(3 downto 0);
        output      :out   std_logic);
end mux16to1;

-- Architecture
Architecture Structure of mux16to1 is


-- Here we declare an intermediate signal to help us to contain outputs of first 4 muxs as input to the fifth mux

Signal helpMe :  std_logic_vector(0 to 3);

-- Here we declare a component, our mux4to1

Component mux4to1 is
   port(w0,w1,w2,w3     :in    std_logic;
        selector        :in    std_logic_vector(1 downto 0);
        output          :out   std_logic);
end Component;

begin


Mux1 : mux4to1 PORT MAP (w(0), w(1), w(2), w(3), selector(1 downto 0), helpMe(0));
Mux2 : mux4to1 PORT MAP (w(4), w(5), w(6), w(7), selector(1 downto 0), helpMe(1));
Mux3 : mux4to1 PORT MAP (w(8), w(9), w(10), w(11), selector(1 downto 0), helpMe(2));
Mux4 : mux4to1 PORT MAP (w(12), w(13), w(14), w(15), selector(1 downto 0), helpMe(3));
Mux5 : mux4to1 PORT MAP (helpMe(0), helpMe(1), helpMe(2), helpMe(3), selector(3 downto 2), output);
end Structure;
```

- Example 2 to 4 Decoder:

```
Library IEEE;
Use IEEE.std_logic_1164.all;


Entity decoder2to4 is
   port(w           :in    std_logic_vector(1 downto 0);
        Enable      :in    std_logic;
```

```
        output      :out   std_logic_vector(0 to 3));
end decoder2to4;


Architecture Behavior of decoder2to4 is


-- Here we declare an intermediate signal to help us, we concatenate Enable and w with & operator

Signal Enw :  std_logic_vector(2 downto 0);

begin

Enw <= Enable & w;
with Enw select
--            0123

   output <= "1000" when "100",
             "0100" when "101",
             "0010" when "110",
             "0001" when "111",
             "0000" when others;

end Behavior;
```

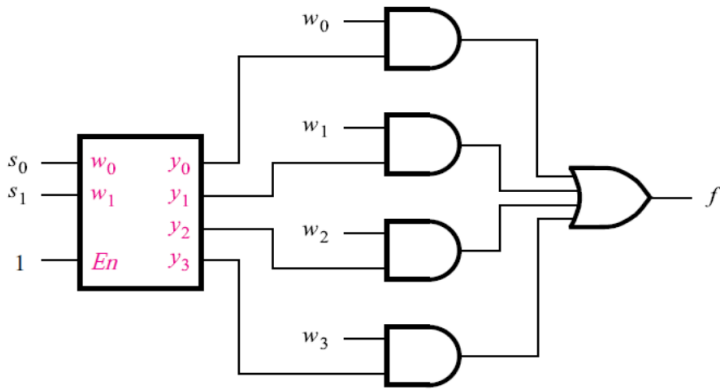
- Example: 3 to 8 Decoder using 2 to 4 decoder:




```
Library IEEE;
Use IEEE.std_logic_1164.all;


Entity decoder3to8 is
   port(w8           :in    std_logic_vector(2 downto 0);
        Enable8      :in    std_logic;
        output8      :out   std_logic_vector(0 to 7));
end decoder3to8;


Architecture Behavior of decoder3to8 is


-- Here we declare an intermediate signal to help us, we use it as intermediate enable

Signal inter_enable :  std_logic_vector(0 to 1);

-- Here we declare a component, our decoder2to4



Component decoder2to4 is
   port(w           :in    std_logic_vector(1 downto 0);
        Enable      :in    std_logic;
        output      :out   std_logic_vector(0 to 3));
end Component;


begin

inter_enable(0) <= not w8(2) and Enable8;
inter_enable(1) <= w8(2) and Enable8;
```

```
Stage1 : decoder2to4 PORT MAP (w8(1 downto 0), inter_enable(0), output8(0 to 3) );
Stage2 : decoder2to4 PORT MAP (w8(1 downto 0), inter_enable(1), output8(4 to 7) );
end Behavior;
```

- Example: 4 to 1 multiplexer using 2 to 4 decoder:




```
Library IEEE;
Use IEEE.std_logic_1164.all;


Entity mux4to1 is
   port(w0, w1, w2, w3 :in    std_logic;
        selector       :in    std_logic_vector(1 downto 0);
        output         :out   std_logic);
end mux4to1;


Architecture Behavior of mux4to1 is


-- Here we declare an intermediate signal to help us, we use it to contain the output of the decoder

Signal decoder_output :  std_logic_vector(0 to 3);

-- decoder is always enabled

constant enable : std_logic := '1';

-- Here we declare a component, our decoder2to4



Component decoder2to4 is
   port(w           :in    std_logic_vector(1 downto 0);
        Enable      :in    std_logic;
        output      :out   std_logic_vector(0 to 3));
end Component;


begin


Stage : decoder2to4 PORT MAP (selector(1 downto 0), enable, decoder_output(0 to 3) );

with decoder_output select

   output <= w0 when "1000",
             w1 when "0100",
             w2 when "0010",
             w3 when "0001";

end Behavior;
```