VHDL Tutorial

Goals

- Introduce the students to the following:
  - VHDL as Hardware description language.
  - How to describe your design using VHDL.
  - Why use VHDL as an alternative to schematic capture.
  - Syntax of VHDL.
  - Hierarchical Design.

VHDL

- VHDL stands for VHSIC (Very High Speed Integrated Circuit) HDL (Hardware Description Language).
- HDLs are used to model hardware
- VHDL is used to describe digital systems.
- Initially was intented for documentation, and simulation.
- Now used for synthesis.

VHDL program components

- Library Declaration.
- Entity Declaration.
- Architecture Body.

Library Declaration

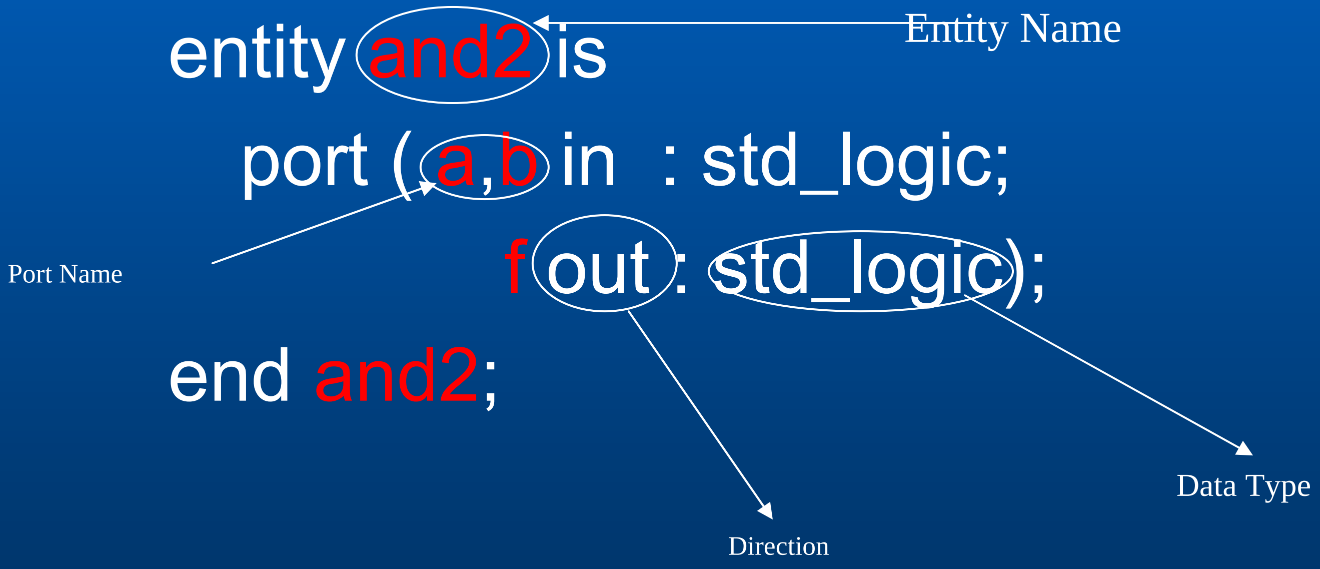
- This declare standard data types and some procedures used in modelling the design.

```
Library IEEE;                        -- Declare the IEEE library
Use IEEE.STD_LOGIC.1164.all;   --Use package 1164
```

- Packages are containers for related functional units.
- Library contains declaration of different packages and components.

Entity Declaration

- Entity describes the input/output configuration for the modelled system.



```
entity and2 is
   port (a,b in  : std_logic;
             f out : std_logic);
end and2;
```

Architecture Body

- Architecture body is used to describe the internal structure of the modelled system.

```
architecture dataflow of and2 is
--signal and component declaration here
begin
  f <= a and b;
end dataflow;
```

Entity

Architecture Name

Concurrent Statements

Complete Model

```
Library IEEE;
Use IEEE.STD_LOGIC.1164.all;

entity and2 is
   port ( a,b in  : std_logic;
                  f out : std_logic);
end and2;

architecture dataflow of and2 is
begin
   f <= a and b;            --Data flow model
end dataflow;
```

Complete Model

```
entity and2 is
    port ( a,b in  : std_logic;
               f out : std_logic);
end and2;
architecture dataflow of and2 is
begin
        f <= a and b;
end dataflow;


architecture structured of and2 is
begin
        u1 : oldand2 port map (a,b,f);
end structured;
```

```
--Three Model Styles
--1. Data Flow
--2. Structured
--3. Behavioural
```

```
architecture behaviour of and2 is
begin
and_proc: process (a,b)
begin
    if a = b then
        f <= '1';
    else
        f <= '0';
    end if;
end behaviour;
```

Data Types

- Every data object in VHDL can hold a value that belongs to a set of values.
- This set of values is specified using a type declaration.
  - Predefined types.
  - User defined types

Predefined Data Types

- Boolean “False, True”
- Bit (0,1)
  - Bit_Vector -array of bits (100011)
- Character ‘a’ ,”ASCII”
- INTEGER (3 , 12)
- REAL (1.5 , 0.23)

STD_LOGIC Data Type

- This data type is used to define signals that could be found in standard digital system.
- This data type is defined in the IEEE library Package IEEE.STD_LOGIC.1164.
- It could have the following values:
  - ‘1’ => Forcing Logic 1
  - ‘0’ => Forcing Logic 0
  - ‘Z’ => High Impedance
  - ‘U’ => Un-initialized
  - ‘X’ => Forcing Unknown
  - ‘-’ => Don’t care
  - ‘W’=> Weak Unknown
  - ‘L’ => Weak 0
  - ‘H’ => Weak 1

STD_LOGIC_VECTOR Data Type

- Array of STD_LOGIC.
- It could be used to represent a bus in digital systems.

```
--MSB in the left  and LSB in the right
signal data_bus : std_logic_vector (7 downto 0);

--LSB in the left  and MSB in the right
signal data_bus : std_logic_vector (0 to 7);
```

STD_LOGIC_VECTOR Data Type

- Signals or Variables of this data type could be accessed completely , partially, or bit by bit.

```
--MSB in the left  and LSB in the right
signal data_bus : std_logic_vector (7 downto 0);
signal data_bus_nipple : std_logic_vector (3 downto 0);


--Inside the architecture
data_bus_nipple <= “0101”;   --load data in 4 bit signal
data_bus (3 downto 0) <= data_bus_nipple; --connect it to the first
    4
data_bus (6 downto 4) <= “100”; --assign the other 3 bits
data_bus (7) <= not data_bus_nipple (3); --and the final bit
```

Concurrent Statements

- Inside the architecture body we use concurrent statements.
  - Signal assignment

    ```
    f  <= a and b;
    ```

  - Processes

    ```
    and_proc : process (a,b)
    begin
      .
      --sequential statements
      .
    end process;
    ```

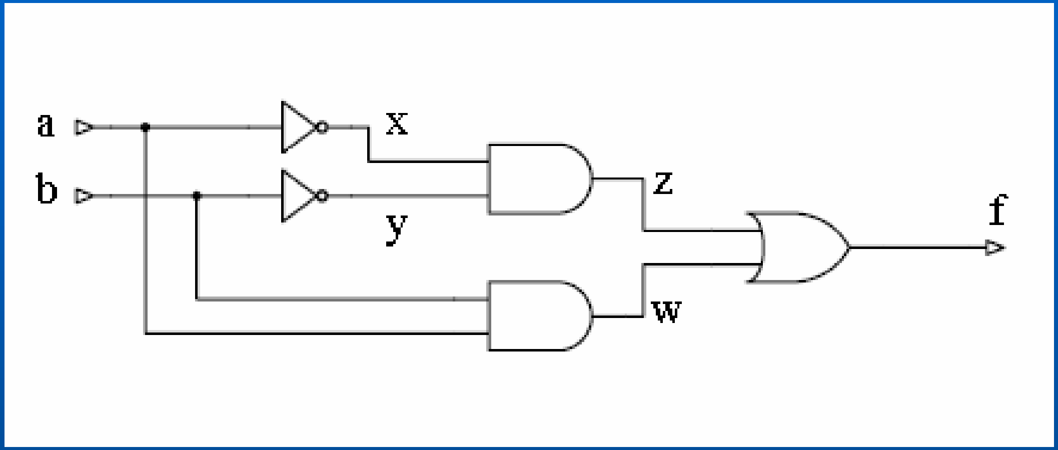
  - Component instantiation

    ```
    u1 : and2 port map (as,bs,fs);
    ```

Concurrent Statements

- The concurrent statements are executed without any specific order.
- The architecture body could contain any combination of the 3 types of concurrent statements.

Concurrent Statements



This circuit could be modelled as following:

```
f<= z or w;
z<= x and y;
x<= not a;
w<= a and b;
y<= not b;
```

Generate Statement

- Used to generate multiple concurrent statements with the same pattern.

```
signal x : std_logic_vector (3 downto 0);
signal y,z : std_logic_vector (3 downto 0);

for i in 0 to 2 generate
    x(i) <= (y (i) and y (i+1) ) or z (i);
end generate;
```

Process Block

Process name

Sensitivity List

Variable Deceleration

Sequential Statements

```
proc_name: process (x,y)
variable z: std_logic;
begin
    z:= x or y;
    if z= ‘0’ and y=‘1’ then
        m<= z;
    else
        m<= not x;
    end if;
end process;
```

Note: The process block is considered a single concurrent statement.

Component Instantiation

- It has two parts:
  - Component Declaration in arch. Body before the begin line:

```
component and2                        --like entity declaration
   port (a,b : in STD_LOGIC;
            f    : out STD_LOGIC);
end component;
```

  - Component Instantiation inside the arch. Body:

```
u1: and2 port map (a=>x,f=>z,b=>y); --No order required
                   or simply
u1: and2 port map (x,y,z);           --Has to be in order
```

Component port

Arch. Signal

Sequential Statements

- Sequential Statements are used inside the process, function or procedure blocks.
- This may be regarded as normal programming language (The order of the statements affect the result of execution).
- Can make use and change the values of signals and variables

Sequential Statements

- If statement

```
if  x = y then
      z<= ‘1’;                    --if true
else
      z<= ‘0’;                    --if false
end if;
```

- Case statement

```
signal y : std_logic_vector (1 downto 0);
signal m : std_logic_vector (3 downto 0);

  case (y) is
      when "00" =>          m <=“1001”;
      when "01" =>          m<=“0101”;
      when "10" =>          m<=“1100”;
      when "11" =>          m<=“0001”;
      when others =>        m<=“0000”;
  end case;
```

Sequential Statements

- Other statements like *for* and *while* are also existing but requires attention.

```
signal v: std_logic_vector (3 downto 0);


    for i in 0 to 2 loop                      --shifting right using for loop
        v(i) <= v(i+1);
    end loop;

    v(3) <= ‘0’;
```

Note: In this example the signal is regarded as a register.

Object Types

- Signals
  - Ports.

    ```
    port  ( a: in std_logic; ……
    ```
  - Internal Signals.

    ```
    signal x : std_logic;
    ```
- Variable

  ```
  variable x : integer;
  x:= 5;
  ```
- Constants

  ```
  constant gnd : bit := 0;
  ```

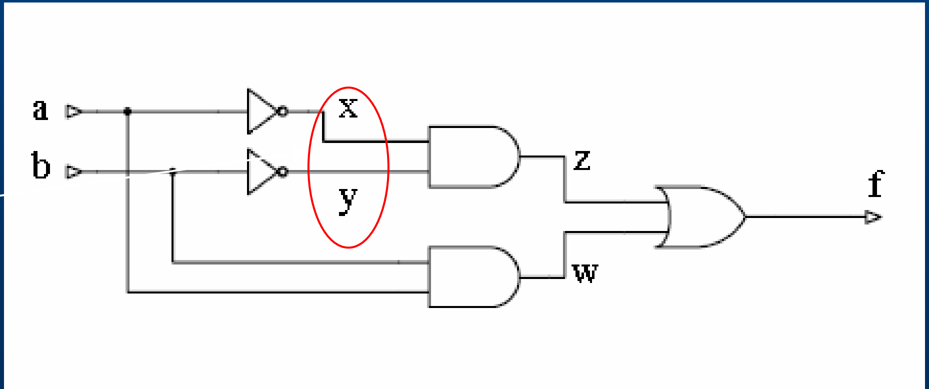
Signals

- Declared as ports or inside the architecture body.
  - Ports.

    port (a: in std_logic; ……
  - Internal Signals.

    signal x : std_logic;
- Signals are regarded is wires in some models and as memory storage (register) in other models.

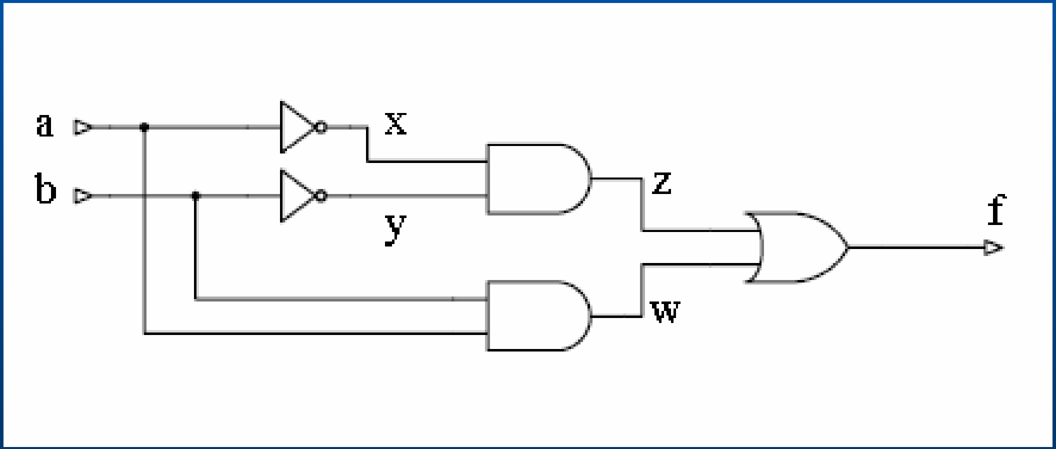
signals



Signals

- The order of signal assignments in the same block is not important.

```
f<= z or w;
z<= x and y;
x<= not a;
w<= a and b;
y<= not b;
```



Signals

- Signals can’t have two drivers.

Note: In this example the process is considered a single driver

```
signal x,y,x : std_logic; --signal declaration
.
.
x<= y or z;                    -- first driver

uproc : process (y,z)          -- second driver
Begin
    if y = z then
            x <= ‘1’;
   else
            x <= ‘0’;
    end if;
end process;
```

Conditional Signal Assignment

- Signal Assignment could be conditional.

```
signal x : STD_LOGIC;
signal y : STD_LOGIC_VECTOR (2 downto 0);

with y select
 x<= ‘0’ when “00” ,
       ‘0’ when “01”,
       ‘0’ when “10”,
       ‘1’ when “11”,
       ‘0’ when others;
```

Conditional Signal Assignment

- Signal Assignment could be conditional.

```
signal x : STD_LOGIC;
signal y : STD_LOGIC_VECTOR (2 downto 0);

x<= ‘0’ when y = “00” else
       ‘0’ when y = “01” else
       ‘0’ when y = “10” else
       ‘1’ when y = “11” else
       ‘0’;
```

Variables

- Same variable concept as computer programming languages.
- Declared inside a process block ,function or procedure.

```
uproc: process (x,y)
variable z : std_logic;
begin
    z := ‘0’;          --variable could have different values
   z := x or y;
end process;
```

Variables

- The scope of the variable is inside the unit it declared in (process, function or procedure).

```
uproc: process (x,y)
variable z : std_logic;
begin
   z := ‘0’;  --variable could have different values
   z := x or y;
end process;
~~z:= ‘1’;~~      -- This variable is out of scope
```

Operators

- Logic operators: They are bit-wise operators
  (and , or , xor , xnor , nand , nor , not)

```
x<= y and z;
a<= (b and c) or (d and e);
j <= (h and i and k)
```

Note: Parenthesis are used to group terms.

~~m<= (n not k);~~
~~g<= f and h or p;~~

m<= (n and (not k));
g<= (f and h) or p; ✓

Operators

- Arithmetic operators: Defined for predefined data types.
- For user defined data types they should be defined.

  - + Add , - Subtract , * multiply , / divide.
- Division is not always defined

```
signal v,w,x : integer;
w <= 5;
v  <= 3;
x <= w + v;    -- x = 8
```

Operators for STD_LOGIC

- Although STD_LOGIC type is defined in VHDL libraries. It is considered a user defined data type.
- Some extra package declaration are required.

```
library IEEE;
use IEEE.STD_LOGIC.1164;
use IEEE.STD_LOGIC.ARITH.all;
use IEEE.STD_LOGIC.UNSIGNED.all;   --unsigned operations
    or
use IEEE.STD_LOGIC.SIGNED.all;     --signed operations
```

Operators for STD_LOGIC;

```
use IEEE.STD_LOGIC.unsigned.all;

signal x,y : std_logic_vector (3 downto 0);
signal za :std_logic_vector (3 downto 0);
signal zm :std_logic_vector (7 downto 0);

x<=“0101”;        --5
y<=“1101”;        --13

za<= x + y;       -- za = “0010”;   = 2; wrong answer
Zm<= x * y;       -- zm = “01000001”; 65; right answer
```

Operators for STD_LOGIC;

use IEEE.STD_LOGIC.unsigned.all;

signal x,y : std_logic_vector (3 downto 0);
signal za :std_logic_vector (4 downto 0);
signal zm :std_logic_vector (7 downto 0);

x<="0101";   --5
y<="1101";   --13

za<=~~x~~ + y;   -- za = "0010";   = 2; wrong answer
Zm<= x * y;   -- zm = "01000001"; 65; right answer

Operators for STD_LOGIC;

```
use IEEE.STD_LOGIC.unsigned.all;

signal x,y : std_logic_vector (3 downto 0);
signal za :std_logic_vector (3 downto 0);
signal zm :std_logic_vector (7 downto 0);

x<=“0101”;        --5
y<=“1101”;        --13

za<= ‘0’&x + ‘0’&y;       -- za = “10010”;= 18; right answer
Zm<= x * y;       -- zm = “01000001”; 65; right answer
```

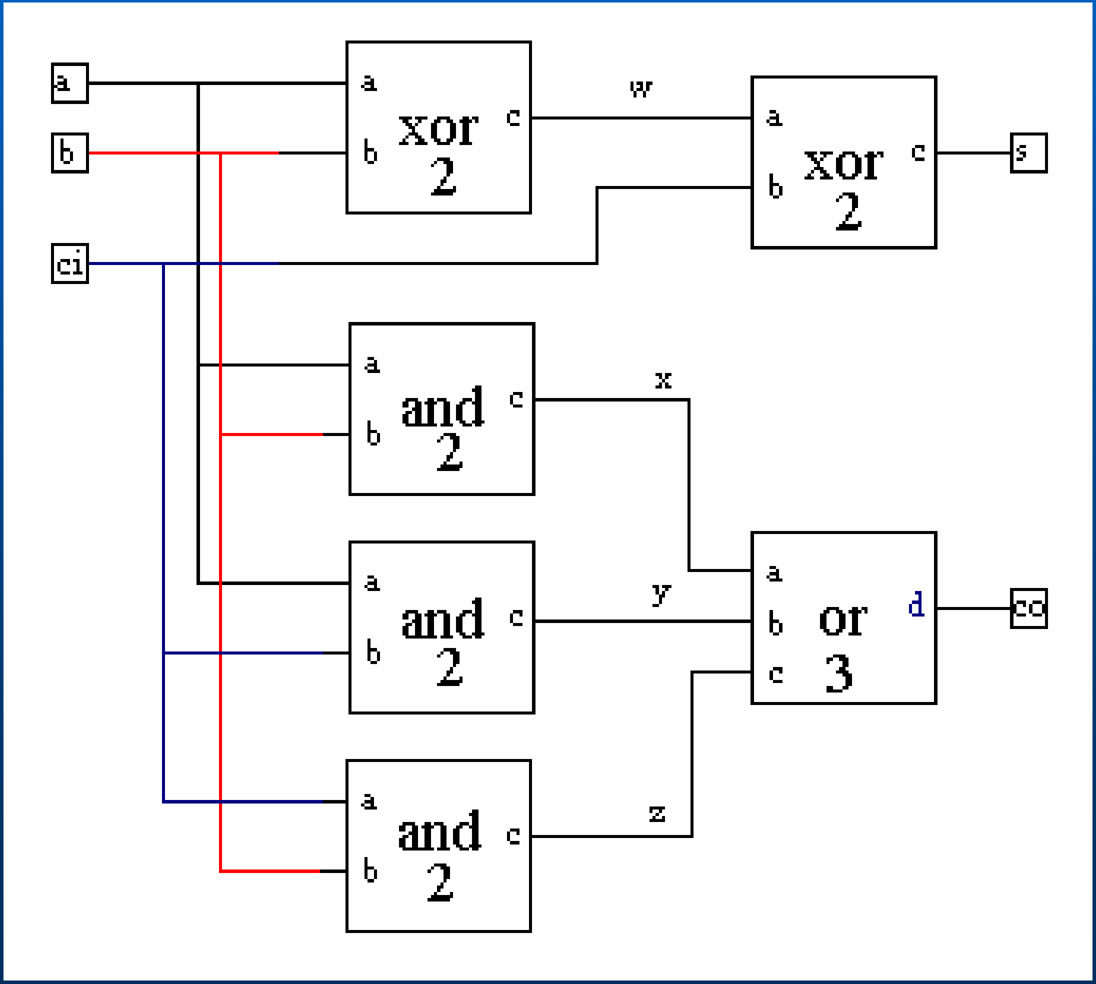
concatenation

Operators for STD_LOGIC;

```
use IEEE.STD_LOGIC.signed.all;

signal x,y : std_logic_vector (3 downto 0);
signal za :std_logic_vector (3 downto 0);
signal zm :std_logic_vector (7 downto 0);

x<=“0101”;        --5
y<=“1101”;        --(-3)

za<= ‘0’&x +’0’&y;        -- za = “00010”;= 2 (; wrong answer
Zm<= x * y;        -- zm = “11110001”; -15; right answer
```

Example

This Figure Shows a Full Adder Circuit



Example

- First declare Xor2 ,And2 and Or3 entities and architectures

```
entity and2 is
      port ( a,b : in   std_logic;
                  c : out std_logic );
end and2;
architecture dataflow of and2 is
begin
      c<= a and b;
end dataflow;
```

```
entity xo2 is
      port ( a,b : in    std_logic;
                  c : out std_logic );
end xor2;
architecture dataflow of and2 is
begin
      c<= a xor b;
end dataflow;
```

```
entity or3 is
   port ( a, b,c : in   std_logic;
                  d : out std_logic );
end and2;
architecture behavior of or3 is
begin
      and3_proc : process is
      begin
            if a = ‘0’ and b = ‘0’ and c=‘0’ then
                  d <= ‘0’;
            else
                  d <= ‘1’;
            end if;
      end process;
end behavior;
```

Example

- Now use them to implement a register

```
entity full_adder is
    port (a , b , ci: in std_logic;
          s ,co: out std_logic);
end entity;

architecture struct of full_adder is
--Internal signals
    signal w,x,y,z : std_logic;
--component declaration
    component and2
        port ( a,b : in bit;  c : out bit );
    end component;
    component xo2
        port ( a,b : in bit;  c : out bit );
    end component;
```

```
    component or3 is
      port ( a, b,c : in bit;  d : out bit );
    end component;
begin
    u0 : xor2  port map ( a, b , w);
    u1 : xor2  port map ( w, ci, s );
    u2 : and2 port map ( a, b, x);
    u3 : and2 port map ( a, ci, y );
    u4 : and2 port map ( b, ci, z );
    u5 : or2    port map (x,y,z, co);
end struct;
```