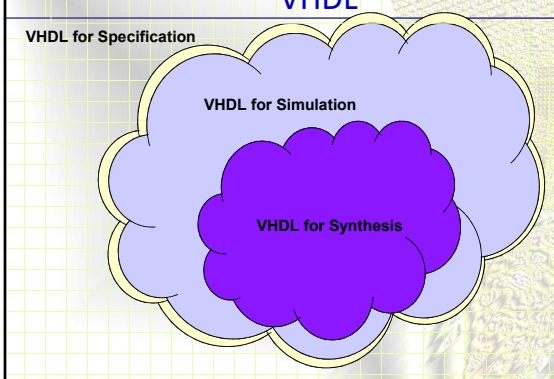
# Embedded Systems

## Register Transfer-level Design: VHDL for Synthesis

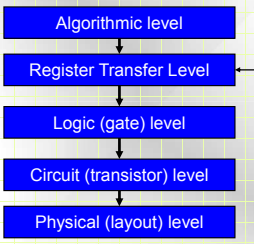
George Kornaros

## VHDL





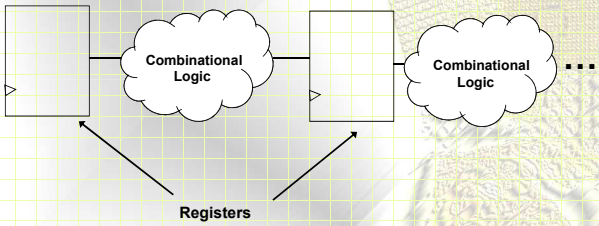

## Levels of design description




Level of description most suitable for synthesis



## Register Transfer Logic (RTL) Design Description



# VHDL Fundamentals



## Naming and Labeling (1)

- VHDL is not case sensitive

  Example:

  Names or labels

  **databus**
  **Databus**
  **DataBus**
  **DATABUS**

  are all equivalent



## Naming and Labeling (2)

General rules of thumb (according to VHDL-87)

1. All names should start with an alphabet character (a-z or A-Z)
2. Use only alphabet characters (a-z or A-Z) digits (0-9) and underscore (_)
3. Do not use any punctuation or reserved characters within a name (!, ?, ., &, +, -, etc.)
4. Do not use two or more consecutive underscore characters (__) within a name (e.g., Sel__A is invalid)
5. All names and labels in a given entity and architecture must be unique



## Free Format

- VHDL is a "free format" language

  No formatting conventions, such as spacing or indentation imposed by VHDL compilers. Space and carriage return treated the same way.

  Example:

```
if (a=b) then
```

*or*

```
if  (a=b)        then
```

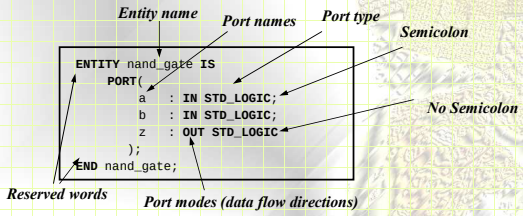
*or*

```
if (a =
b) then
```

are all equivalent

## *Entity* Declaration

- *Entity Declaration* describes the interface of the component, i.e. *input* and *output* ports.






## Entity declaration – simplified syntax

```
ENTITY entity_name IS
  PORT (
     port_name :  signal_mode  signal_type;
     port_name :  signal_mode  signal_type;
     ……..
     port_name :  signal_mode  signal_type);
END entity_name;
```



## *Architecture*

- Describes an implementation of a design entity.
- Architecture example:

```
ARCHITECTURE model OF nand_gate IS
BEGIN
    z <= a NAND b;
END model;
```



## Architecture – simplified syntax

```
ARCHITECTURE architecture_name OF entity_name IS
   [ declarations ]
BEGIN
    code
END architecture_name;
```

## Entity Declaration & Architecture

nand_gate.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
    PORT(
        a   : IN STD_LOGIC;
        b   : IN STD_LOGIC;
        z   : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
    z <= a NAND b;
END model;
```



## Library declarations

Library declaration

Use all definitions from the package std_logic_1164

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
    PORT(
        a   : IN STD_LOGIC;
        b   : IN STD_LOGIC;
        z   : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE model OF nand_gate IS
BEGIN
    z <= a NAND b;
END model;
```



## Library declarations - syntax

```
LIBRARY  library_name;
USE library_name.package_name.package_parts;
```



## Fundamental parts of a library

LIBRARY

PACKAGE 1

TYPES
CONSTANTS
FUNCTIONS
PROCEDURES
COMPONENTS

PACKAGE 2

TYPES
CONSTANTS
FUNCTIONS
PROCEDURES
COMPONENTS

## Libraries

- ieee
  Specifies multi-level logic system, including STD_LOGIC, and STD_LOGIC_VECTOR data types

  Need to be explicitly declared

- std
  Specifies pre-defined data types (BIT, BOOLEAN, INTEGER, REAL, SIGNED, UNSIGNED, etc.), arithmetic operations, basic type conversion functions, basic text i/o functions, etc.

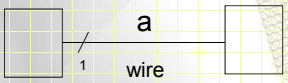
- work
  Current designs after compilation

  Visible by default

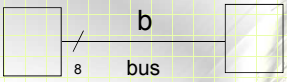


## Signals

SIGNAL a : STD_LOGIC;




SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);






## Standard Logic Vectors

```
SIGNAL a: STD_LOGIC;
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL f: STD_LOGIC_VECTOR(8 DOWNTO 0);
                    …….
a <= '1';
b <= "0000";          -- Binary base assumed by default
c <= B"0000";         -- Binary base explicitly specified
d <= "0110_0111";     -- You can use '_' to increase readability
e <= X"AF67";         -- Hexadecimal base
f <= O"723";          -- Octal base
```



## Vectors and Concatenation

```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c, d, e: STD_LOGIC_VECTOR(7 DOWNTO 0);

a <= "0000";
b <= "1111";
c <= a & b;                    -- c = "00001111"

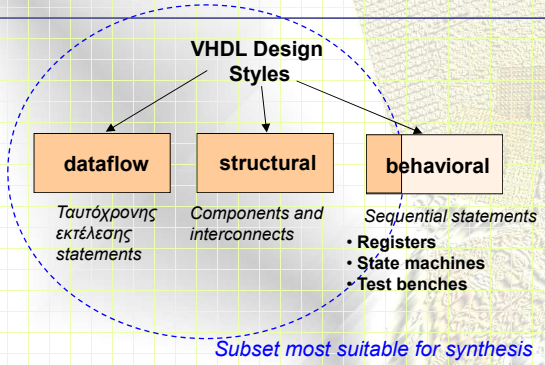
d <= '0' & "0001111";          -- d <= "00001111"

e <= '0' & '0' & '0' & '0' & '1' & '1' &
     '1' & '1';
                               -- e <= "00001111"
```

## VHDL Design Styles

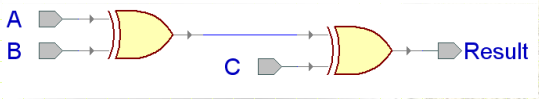


## VHDL Design Styles






## xor3 Example



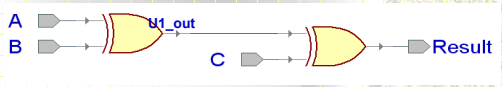



## Entity xor3

```
ENTITY xor3 IS
   PORT(
      A : IN STD_LOGIC;
      B : IN STD_LOGIC;
      C : IN STD_LOGIC;
      Result : OUT STD_LOGIC
         );
end xor3;
```

## Dataflow Architecture (xor3 gate)

```
ARCHITECTURE dataflow OF xor3 IS
SIGNAL U1_out: STD_LOGIC;
BEGIN
        U1_out <=A XOR B;
        Result  <=U1_out XOR C;
END dataflow;
```






## Dataflow Description

- Describes how data moves through the system and the various processing steps.
- Data Flow uses series of concurrent statements to realize logic. Concurrent statements are evaluated at the same time; thus, order of these statements doesn't matter.
- Data Flow is most useful style when series of Boolean equations can represent a logic.



## Structural Architecture (xor3 gate)

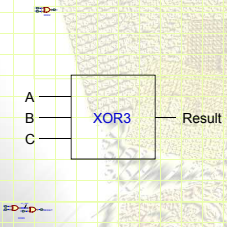
```
ARCHITECTURE structural OF xor3 IS
SIGNAL U1_OUT: STD_LOGIC;

COMPONENT xor2 IS
    PORT(
        I1 : IN STD_LOGIC;
        I2 : IN STD_LOGIC;
        Y : OUT STD_LOGIC
    );
END COMPONENT;

BEGIN
    U1: xor2 PORT MAP (I1 => A,
                       I2 => B,
                       Y  => U1_OUT);

    U2: xor2 PORT MAP (I1 => U1_OUT,
                       I2 => C,
                       Y  => Result);

END structural;
```






## Component and Instantiation (1)

- Named association connectivity (recommended)

```
COMPONENT xor2 IS
    PORT(
        I1 : IN STD_LOGIC;
        I2 : IN STD_LOGIC;
        Y : OUT STD_LOGIC
        );
END COMPONENT;

U1: xor2 PORT MAP (I1 => A,
                   I2 => B,
                   Y  => U1_OUT);
```

## Component and Instantiation (2)

- Positional association connectivity (not recommended)

```
COMPONENT xor2 IS
      PORT(
         I1 : IN STD_LOGIC;
         I2 : IN STD_LOGIC;
         Y : OUT STD_LOGIC
         );
END COMPONENT;

U1: xor2 PORT MAP (A, B, U1_OUT);
```



## Structural Description

- Structural design is the simplest to understand. This style is the closest to schematic capture and utilizes simple building blocks to compose logic functions.
- Components are interconnected in a hierarchical manner.
- Structural descriptions may connect simple gates or complex, abstract components.
- Structural style is useful when expressing a design that is naturally composed of sub-blocks.



## Behavioral Architecture (xor3 gate)

```
ARCHITECTURE behavioral OF xor3 IS
BEGIN
xor3_behave: PROCESS (A,B,C)
BEGIN
   IF ((A XOR B XOR C) = '1') THEN
         Result <= '1';
   ELSE
         Result <= '0';
   END IF;
END PROCESS xor3_behave;
END behavioral;
```



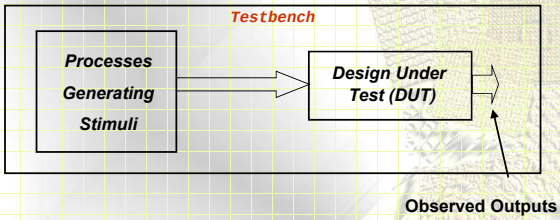
## Behavioral Description

- It accurately models what happens on the inputs and outputs of the black box (no matter what is inside and how it works).
- This style uses PROCESS statements in *VHDL.*

# Testbenches



## Testbench Block Diagram






## Testbench Defined

- *Testbench* applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
- The results can be viewed in a waveform window or written to a file.
- Since *Testbench* is written in VHDL, it is not restricted to a single simulation tool (portability).
- The same *Testbench* can be easily adapted to test different implementations (i.e. different *architectures*) of the same design.



## Testbench Anatomy

```
ENTITY tb IS
                --TB entity has no ports
END tb;

ARCHITECTURE arch_tb OF tb IS

              --Local signals and constants

  COMPONENT TestComp --All Design Under Test component declarations
            PORT (  );
  END COMPONENT;
-------------------------------------------------------
BEGIN
  testSequence: PROCESS                    }
                                           }   -- Input stimuli
  END PROCESS;

  DUT:TestComp PORT MAP(       -- Instantiations of DUTs
                       );
END arch_tb;
```

## Testbench for XOR3 (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xor3_tb IS
END xor3_tb;

ARCHITECTURE xor3_tb_architecture OF xor3_tb IS
-- Component declaration of the tested unit
COMPONENT xor3
PORT(
 A : IN STD_LOGIC;
 B : IN STD_LOGIC;
 C : IN STD_LOGIC;
 Result : OUT STD_LOGIC );
END COMPONENT;

-- Stimulus signals - signals mapped to the input and inout ports of tested entity
SIGNAL test_vector: STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL test_result : STD_LOGIC;
```



## Testbench for XOR3 (2)

```
BEGIN
UUT : xor3
 PORT MAP (
                A => test_vector(0),
                B => test_vector(1),
                C => test_vector(2),
                Result => test_result);
                                );
Testing: PROCESS
BEGIN
test_vector <= "000";
WAIT FOR 10 ns;
test_vector <= "001";
WAIT FOR 10 ns;
test_vector <= "010";
WAIT FOR 10 ns;
test_vector <= "011";
WAIT FOR 10 ns;
test_vector <= "100";
WAIT FOR 10 ns;
test_vector <= "101";
WAIT FOR 10 ns;
test_vector <= "110";
WAIT FOR 10 ns;
test_vector <= "111";
WAIT FOR 10 ns;
END PROCESS;
END xor3_tb_architecture;
```



## What is a PROCESS?

✓ A process is a sequence of instructions referred to as sequential statements.

- A process can be given a unique name using an optional LABEL
- This is followed by the keyword PROCESS
- The keyword BEGIN is used to indicate the start of the process
- All statements within the process are executed SEQUENTIALLY. Hence, order of statements is important.
- A process must end with the keywords END PROCESS.

The keyword PROCESS

```
Testing: PROCESS
BEGIN
        test_vector<="00";
        WAIT FOR 10 ns;
        test_vector<="01";
        WAIT FOR 10 ns;
        test_vector<="10";
        WAIT FOR 10 ns;
        test_vector<="11";
        WAIT FOR 10 ns;
END PROCESS;
```



## Execution of statements in a PROCESS

- The execution of statements continues sequentially till the last statement in the process.
- After execution of the last statement, the control is again passed to the beginning of the process.




```
Testing: PROCESS
BEGIN
        test_vector<="00";
        WAIT FOR 10 ns;
        test_vector<="01";
        WAIT FOR 10 ns;
        test_vector<="10";
        WAIT FOR 10 ns;
        test_vector<="11";
        WAIT FOR 10 ns;
END PROCESS;
```

Program control is passed to the first statement after BEGIN

## PROCESS with a WAIT Statement

- The last statement in the PROCESS is a WAIT instead of WAIT FOR 10 ns.
- This will cause the PROCESS to suspend indefinitely when the WAIT statement is executed.
- This form of WAIT can be used in a process included in a testbench when all possible combinations of inputs have been tested or a non-periodical signal has to be generated.

```
Testing: PROCESS
BEGIN
        test_vector<="00";
        WAIT FOR 10 ns;
        test_vector<="01";
        WAIT FOR 10 ns;
        test_vector<="10";
        WAIT FOR 10 ns;
        test_vector<="11";
        WAIT;
END PROCESS;
```

Order of execution

Program execution stops here

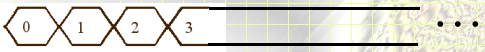


## WAIT FOR vs. WAIT

WAIT FOR: waveform will keep repeating itself forever




WAIT : waveform will keep its state after the last wait instruction.






## Loop Statement

- Loop Statement

```
FOR i IN range LOOP
    statements
END LOOP;
```

- Repeats a Section of VHDL Code
  - Example: process every element in an array in the same way



## Loop Statement – Example (1)

```
Testing: PROCESS
BEGIN
        test_vector<="000";
        FOR i IN 0 TO 7 LOOP
            WAIT FOR 10 ns;
            test_vector<=test_vector+"001";
        END LOOP;
END PROCESS;
```

## Loop Statement – Example (2)

```
Testing: PROCESS
BEGIN
        test_ab<="00";
        test_sel<="00";
        FOR i IN 0 TO 3 LOOP
                FOR j IN 0 TO 3 LOOP
                        WAIT FOR 10 ns;
                        test_ab<=test_ab+"01";
                END LOOP;
                test_sel<=test_sel+"01";
        END LOOP;
END PROCESS;
```







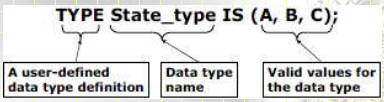
## Finite State Machines

- VHDL provides a number of constructs for designing finite state machines
- There is not a standard way for defining an FSM
- Basic approach
  - ✓ Create a user-defined data type to represent the possible states of an FSM
  - ✓ This signal represents the outputs (state variables) of the flip-flops that implement the states in the FSM
  - ✓ VHDL compiler chooses the appropriate number of flip-flops during the **synthesis** process
  - ✓ The state assignment can be done by the compiler or can be user specified



## User defined data types

- The **TYPE** keyword will be used to define a new data type used to represent states in the FSM
- Defines a data type (called State_type) that can take on three distinct values: A, B, or C).



## Declaring states

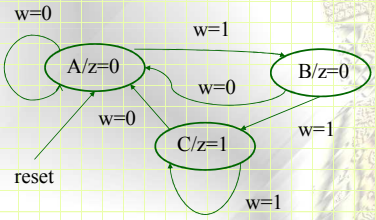
- A SIGNAL is defined, of the user-defined **State_type**, to represent the flip-flop outputs

**TYPE State_type IS (A, B, C);**
**SIGNAL y: State_type;**



## Example

- Create a VHDL description for a circuit that detects a '11' input sequence on an input, **w**






## FSM example in VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY detect IS
PORT( clk, resetn, w : IN STD_LOGIC ;
z : OUT STD_LOGIC) ;
END detect ;

ARCHITECTURE Behavior OF detect IS
   TYPE State_type IS (A,B,C) ;
   SIGNAL y: State_type ;
BEGIN
```



## FSM example in VHDL (cont.)

```
PROCESS ( resetn, clk )
BEGIN
IF resetn = '0' THEN
   y <= A;
ELSIF (clk'EVENT AND clk='1') THEN
  CASE y IS
  WHEN A =>
        IF w='0' THEN
        y <= A;
        ELSE
        y <= B;
        END IF;
  WHEN B =>
        IF w='0' THEN
        y <= A;
        ELSE
        y <= C;
        END IF;
  WHEN C =>
        IF w='0' THEN
        y <= A;
        ELSE
        y <= C;
        END IF;
  END CASE;
 END IF;
END PROCESS;

z <= '1' WHEN y=C ELSE '0';

END Behavior;
```

## Alternative VHDL style

- Another form for describing the circuit in VHDL is to define two signals to represent the state of the FSM
  - ✓ One signal, **y_present, defines the present state of the FSM**
  - ✓ The second, **y_next, defines the next state of the machine**
- This notation follows the **y (present state) and Y** (next state) notation
- Two PROCESS statements will be used to describe the machine
  - ✓ The first describes the STATE TABLE as a combinational circuit
  - ✓ The second describes the flip-flops, stating that **y_present** should take on the value of **y_next** after each positive clock edge



## Alternative VHDL style (cont.)

```
ARCHITECTURE Behavior OF detect IS
  TYPE State_type IS (A,B,C);
  SIGNAL y_present, y_next: State_type;
BEGIN

PROCESS(w,y_present)
BEGIN
   CASE y_present IS
   WHEN A =>
         IF w='0' THEN
         y_next <= A;
         ELSE
         y_next <= B;
   WHEN B =>
         IF w='0' THEN
         y_next <= A;
         ELSE
         y_next <= C;
   WHEN C =>
         IF w='0' THEN
         y_next <= A;
         ELSE
         y_next <= C;
   END CASE;
END PROCESS;

PROCESS(clk,resetn)
BEGIN
  IF resetn='0' THEN
         y_present <= A;
 ELSIF(clk'EVENT AND clk='1') THEN
         y_present <= y_next;
  END IF;
END PROCESS

z <='1' WHEN y_present=C ELSE '0';
END Behavior;
```



## Explicit state assignment

- With the previous designs, state assignment is done by the VHDL compiler
- State assignments can be user specified using the following constructs

```
ARCHITECTURE Behavior OF simple IS
TYPE State_type IS (A, B, C) ;
ATTRIBUTE ENUM_ENCODING : STRING ;
ATTRIBUTE ENUM_ENCODING OF State_type : TYPE IS "00 01 11" ;
SIGNAL y_present, y_next : State_type ;
BEGIN
Or
ARCHITECTURE Behavior OF simple IS
SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
CONSTANT A : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
CONSTANT B : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
CONSTANT C : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
```