ECE 545—Digital System Design with VHDL
Lecture 1

Introduction to VHDL

8/26/08

Lecture Roadmap

- Syllabus and Course Objectives
- History of VHDL
- VHDL Design Flows
- Introduction to VHDL for synthesis

Syllabus and Course Objectives

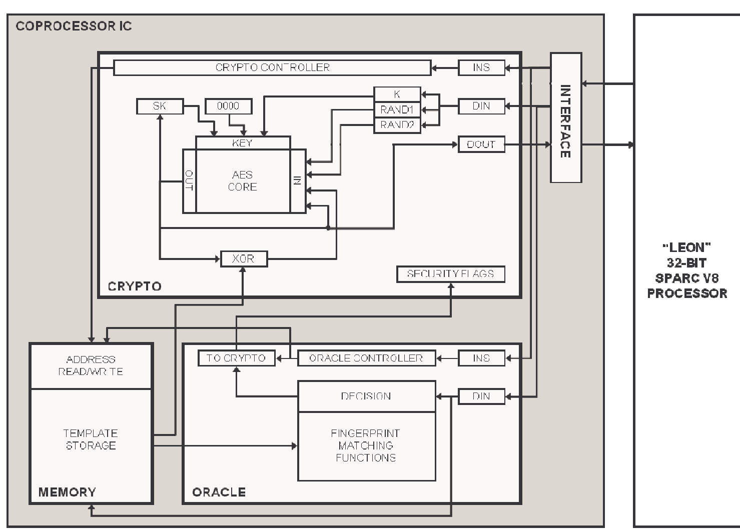
About the Instructor

- Dr. David Hwang
  - PhD in Electrical Engineering, UCLA 2005
    - System Architectures and VLSI Implementations of Secure Embedded Systems
  - Worked in industry designing VLSI signal processing algorithms and circuits
  - Research Interests
    - Secure embedded systems
    - Cryptographic hardware and circuits
    - VLSI digital signal processing
    - VLSI systems and circuits
- Anyone interested in these topics?
  - Do well in ECE 545 and show interest in the material
  - We can discuss M.S. or Ph.D. research

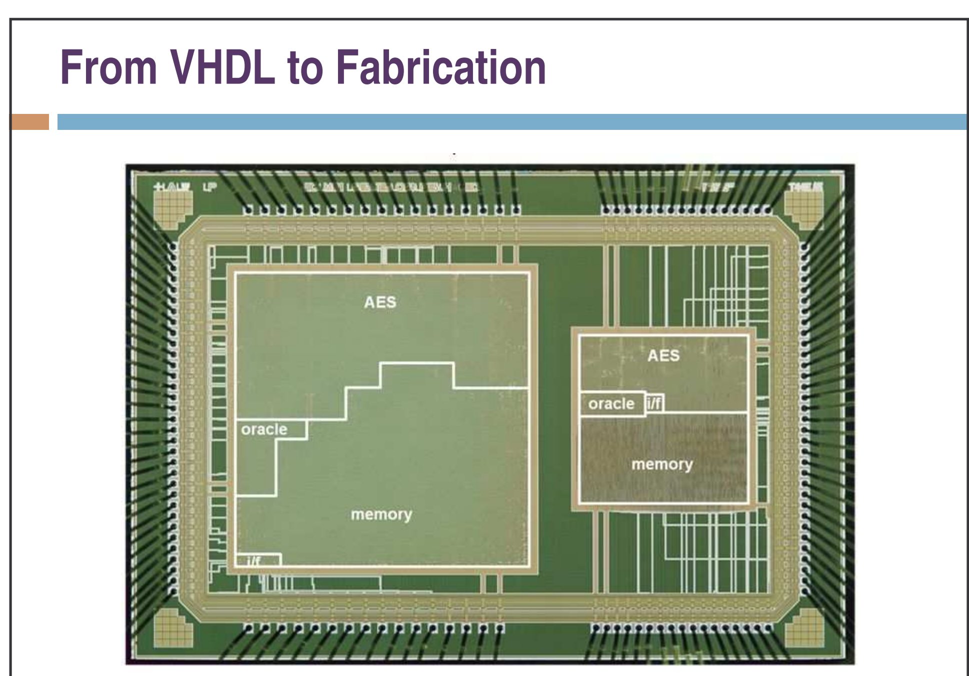
Course Objectives

- At the end of this course you should be able to:
  - Code in VHDL for synthesis
  - Decompose a digital system into a controller (FSM) and datapath, and code accordingly
  - Write VHDL testbenches
  - Synthesize and implement digital systems on FPGAs
  - Understand behavioral, non-synthesizable VHDL and its role in modern design
  - Effectively code digital systems for cryptography, signal processing, and microprocessor applications
- This knowledge will come about through homework, exams, and an extensive project
  - The project in particular will help you know VHDL and the FPGA design flow from beginning to end

Example Design: Cryptographic Coprocessor Resistant to Side-Channel Attacks

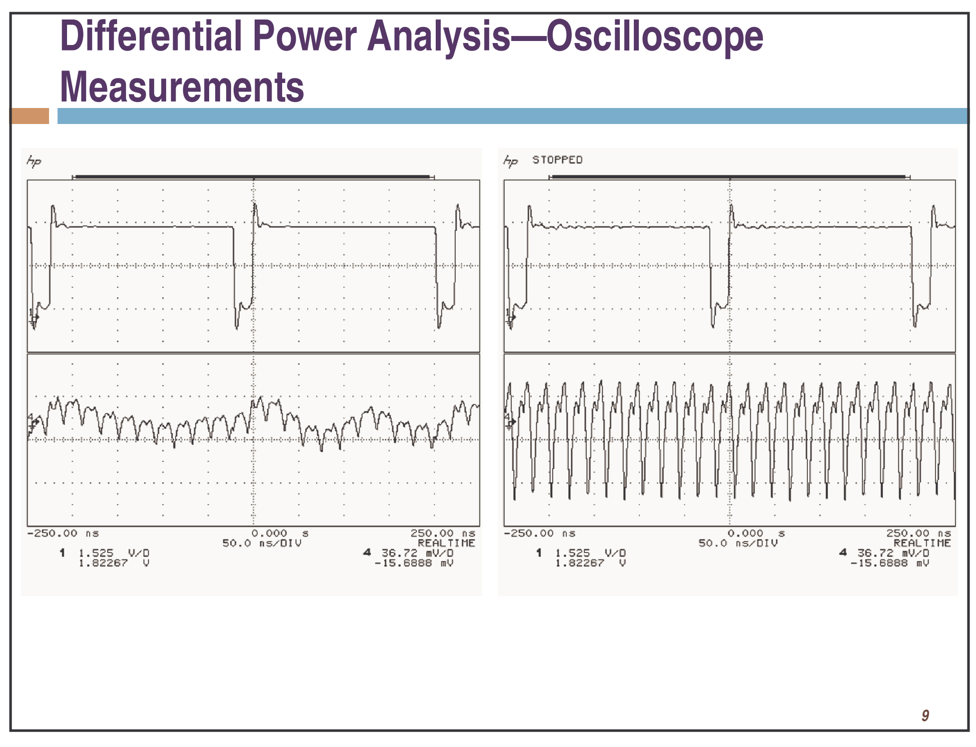


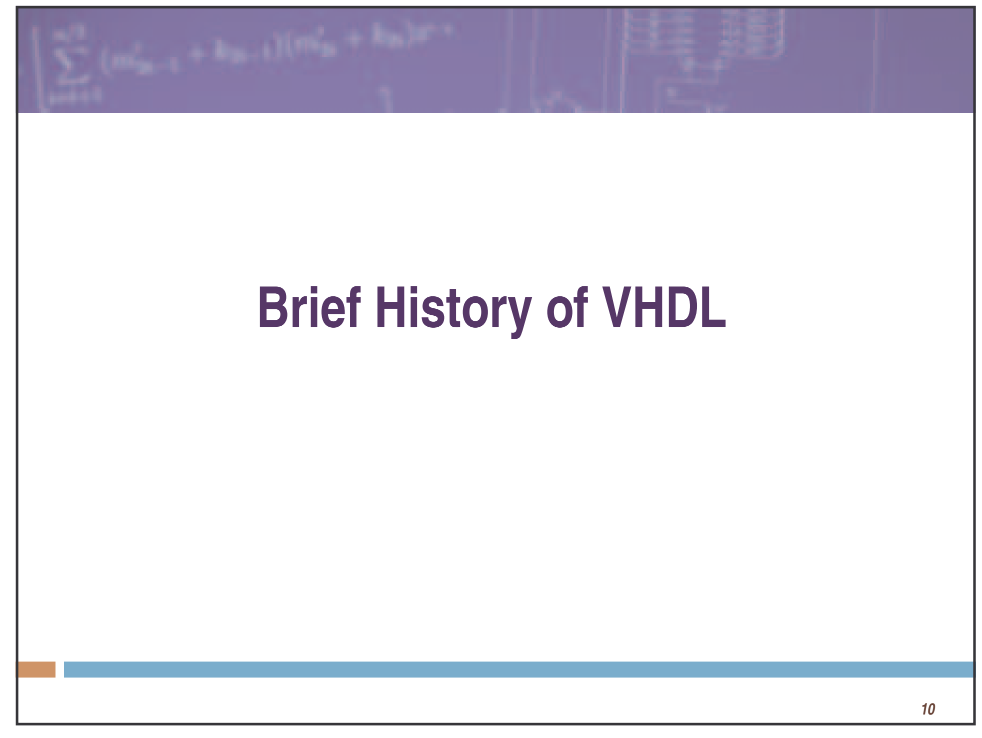
From VHDL to Fabrication



Measurement Test Setup

FPGA BOARD
ASIC BOARD


Differential Power Analysis—Oscilloscope Measurements
hp
hp STOPPED
-250.00 ns
0.000 s
50.0 ns/DIV
250.00 ns
REALTIME
1 1.525 V/D
1.82267 V
4 36.72 mV/D
-15.6888 mV
9


Brief History of VHDL
10

VHDL

- VHDL is a language for describing digital hardware used by industry worldwide
  - **VHDL** is an acronym for **V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage

Genesis of VHDL

- State of the art circa 1980
  - Multiple design entry methods and hardware description languages in use
  - No or limited portability of designs between CAD tools from different vendors
  - Objective: shortening the time from a design concept to implementation from 18 months to 6 months

A Brief History of VHDL

- June 1981: Woods Hole Workshop
- July 1983: contract awarded to develop VHDL
  - Intermetrics
  - IBM
  - Texas Instruments
- August 1985: VHDL Version 7.2 released
- December 1987: VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI standard
- Four versions of VHDL:
  - IEEE-1076 1987
  - IEEE-1076 1993 ← **most commonly supported by CAD tools**
  - IEEE-1076 2000 (minor changes)
  - IEEE-1076 2002 (minor changes)

Verilog

- Essentially identical in function to VHDL
  - No generate statement
- Simpler and syntactically different
  - C-like
- Gateway Design Automation Co., 1983
- Early de facto standard for ASIC programming
- Open Verilog International Standard

VHDL vs. Verilog

Government Developed	Commercially Developed
Ada based	C based
Strongly Type Cast	Mildly Type Cast
Difficult to learn	Easier to Learn
More Powerful	Less Powerful

Examples

- **VHDL Example:**

```
process (clk, rstn)
begin
    if (rstn = '0') then
        q <= '0';
    elseif (clk'event and clk = '1') then
        q <= a + b;
    end if;
end process;
```

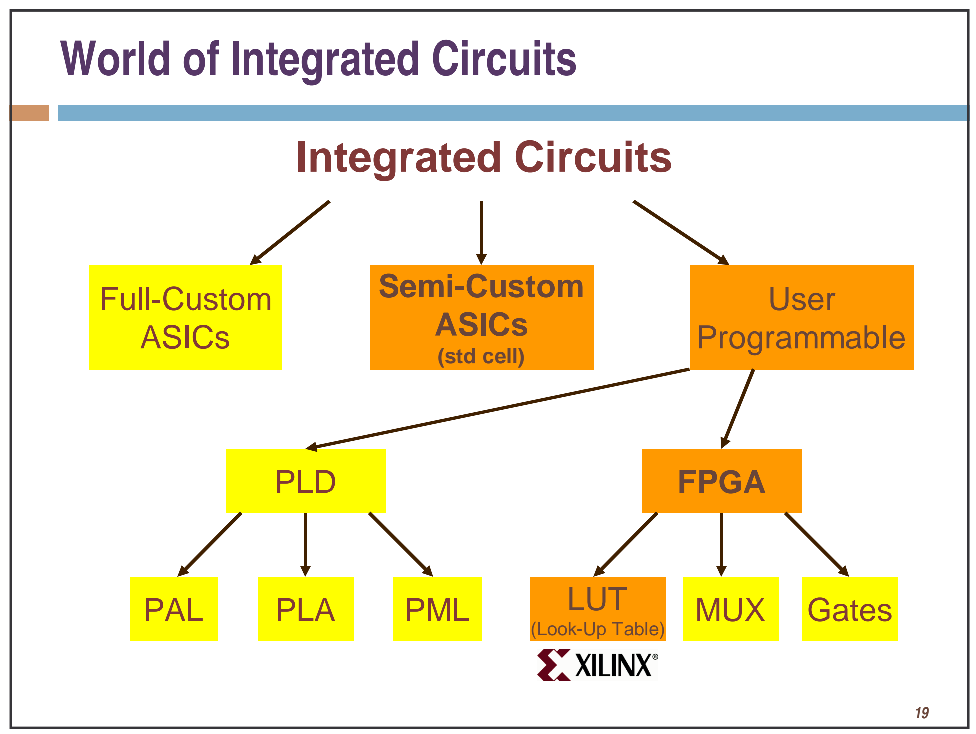
- **Verilog Example:**

```
always@(posedge clk or negedge rstn)
begin
    if (! rstn)
        q <= 1'b0;
    else
        q <= a + b;
end
```

Features of VHDL and Verilog

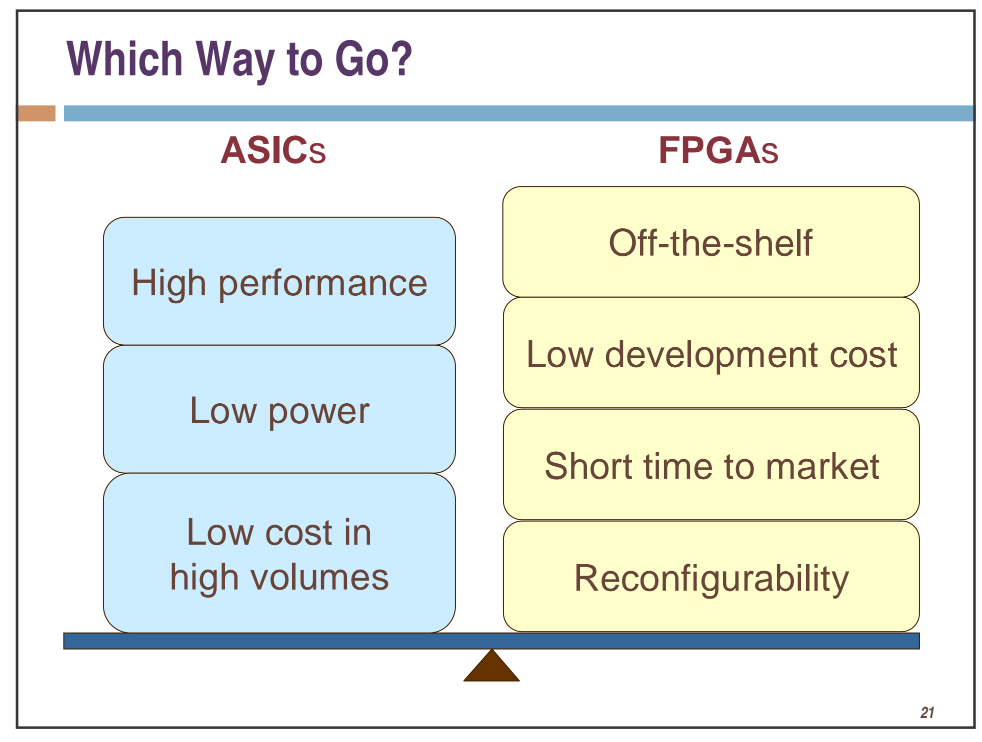
- Technology/vendor independent
- Portable
- Reusable

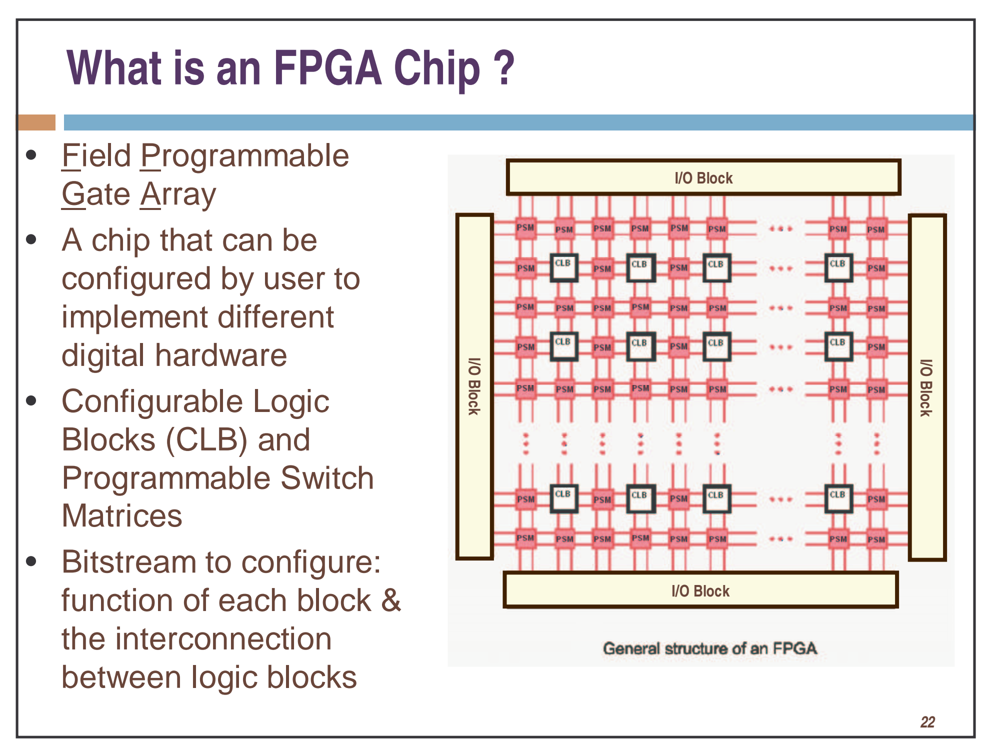
VHDL Design Flows

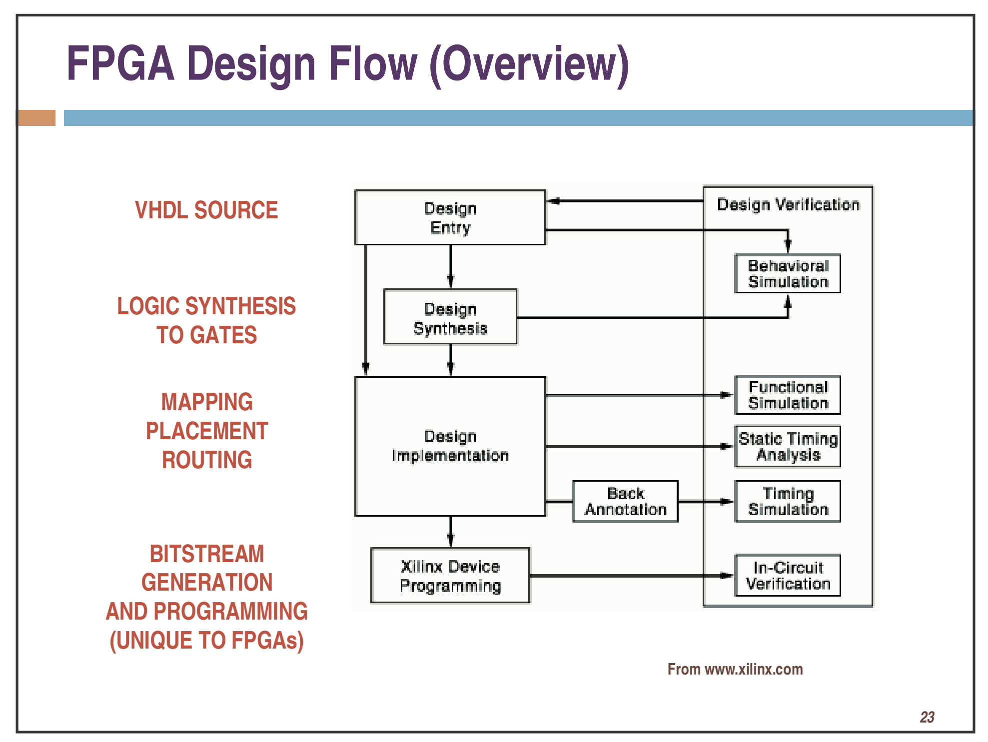


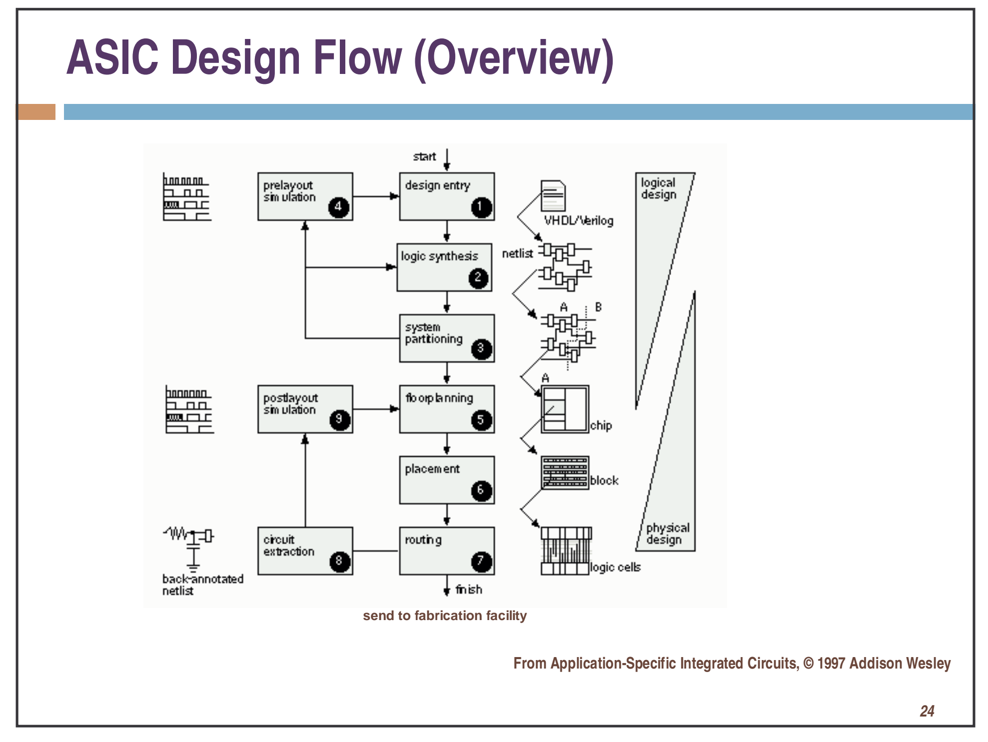
ASIC versus FPGA

ASIC **A**pplication **S**pecific **I**ntegrated **C**ircuit	**FPGA** **F**ield **P**rogrammable **G**ate **A**rray
• designs must be sent for expensive and time consuming fabrication in semiconductor foundry	• bought off the shelf and reconfigured by designers themselves
• designed all the way from behavioral description to physical layout	• no physical layout design; design ends with a bitstream used to configure a device


Which Way to Go?
ASICs
FPGAs
High performance
Low power
Low cost in
high volumes
Off-the-shelf
Low development cost
Short time to market
Reconfigurability
21


What is an FPGA Chip ?
• Field Programmable Gate Array
• A chip that can be configured by user to implement different digital hardware
• Configurable Logic Blocks (CLB) and Programmable Switch Matrices
• Bitstream to configure: function of each block & the interconnection between logic blocks
I/O Block
I/O Block
I/O Block
I/O Block
PSM
CLB
General structure of an FPGA
22

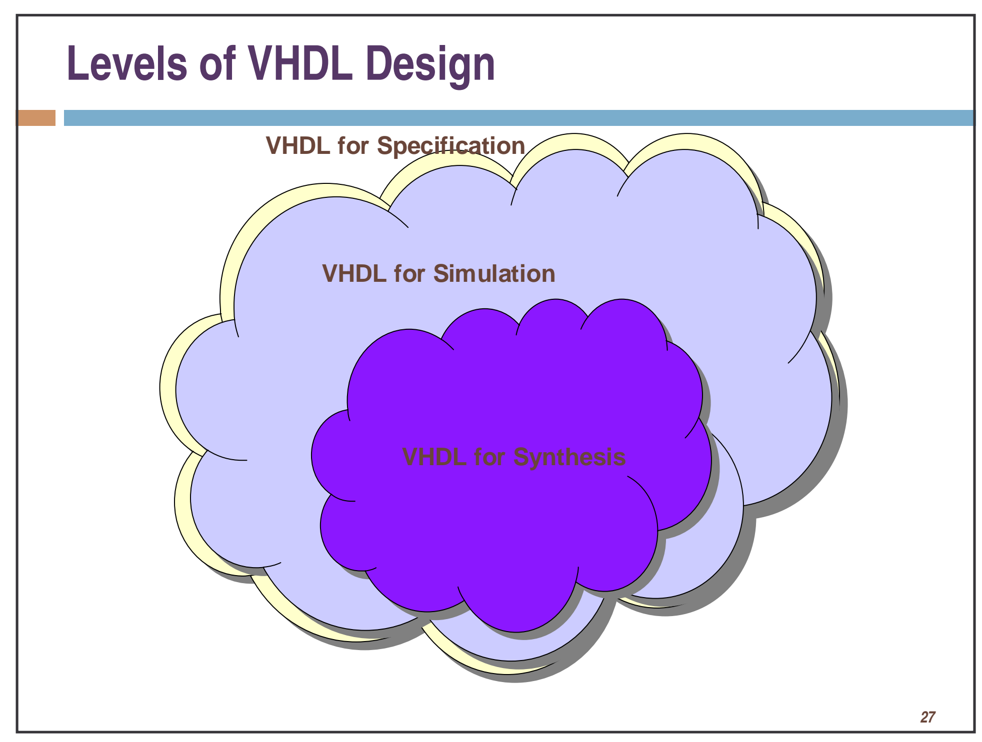

FPGA Design Flow (Overview)
VHDL SOURCE
LOGIC SYNTHESIS
TO GATES
MAPPING
PLACEMENT
ROUTING
BITSTREAM
GENERATION
AND PROGRAMMING
(UNIQUE TO FPGAs)
Design Entry
Design Synthesis
Design Implementation
Back Annotation
Xilinx Device Programming
Design Verification
Behavioral Simulation
Functional Simulation
Static Timing Analysis
Timing Simulation
In-Circuit Verification
From www.xilinx.com
23

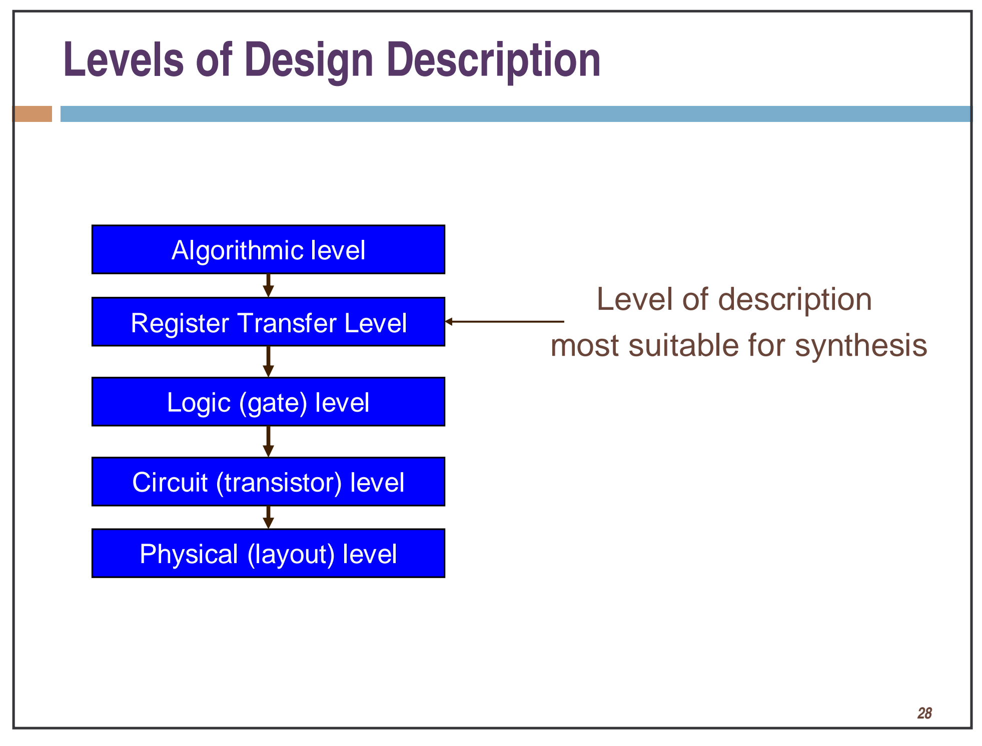

ASIC Design Flow (Overview)
start
prelayout simulation 4
design entry 1
VHDL/Verilog
logical design
logic synthesis 2
netlist
system partitioning 3
A
B
postlayout simulation 9
floorplanning 5
chip
placement 6
block
circuit extraction 8
routing 7
logic cells
physical design
back-annotated netlist
finish
send to fabrication facility
From Application-Specific Integrated Circuits, © 1997 Addison Wesley
24

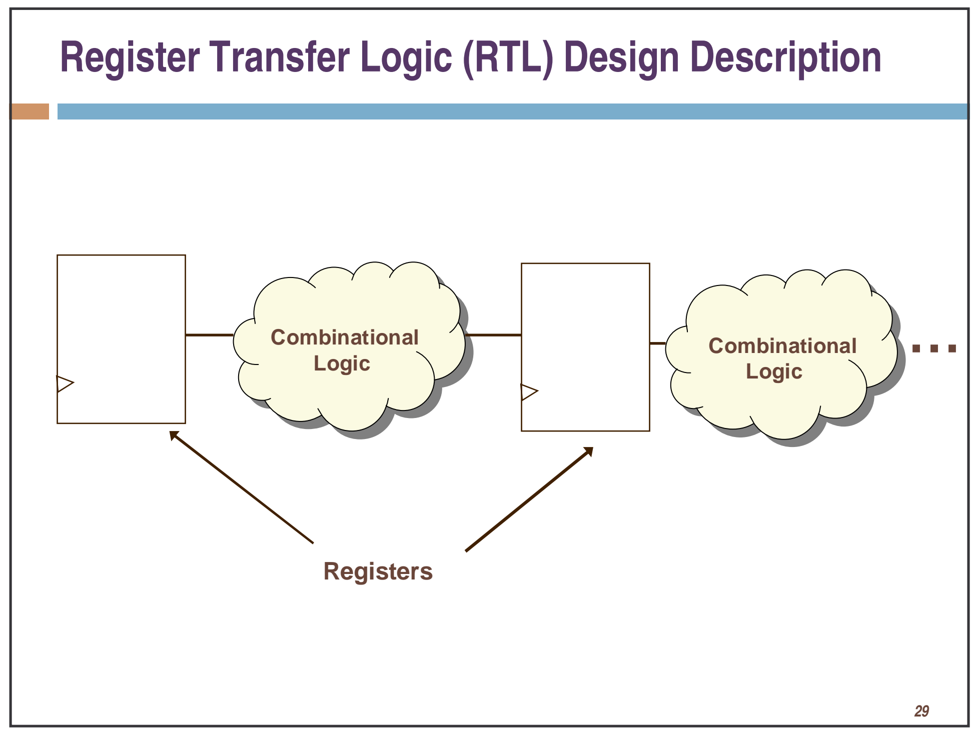
ECE 545 Design Flow

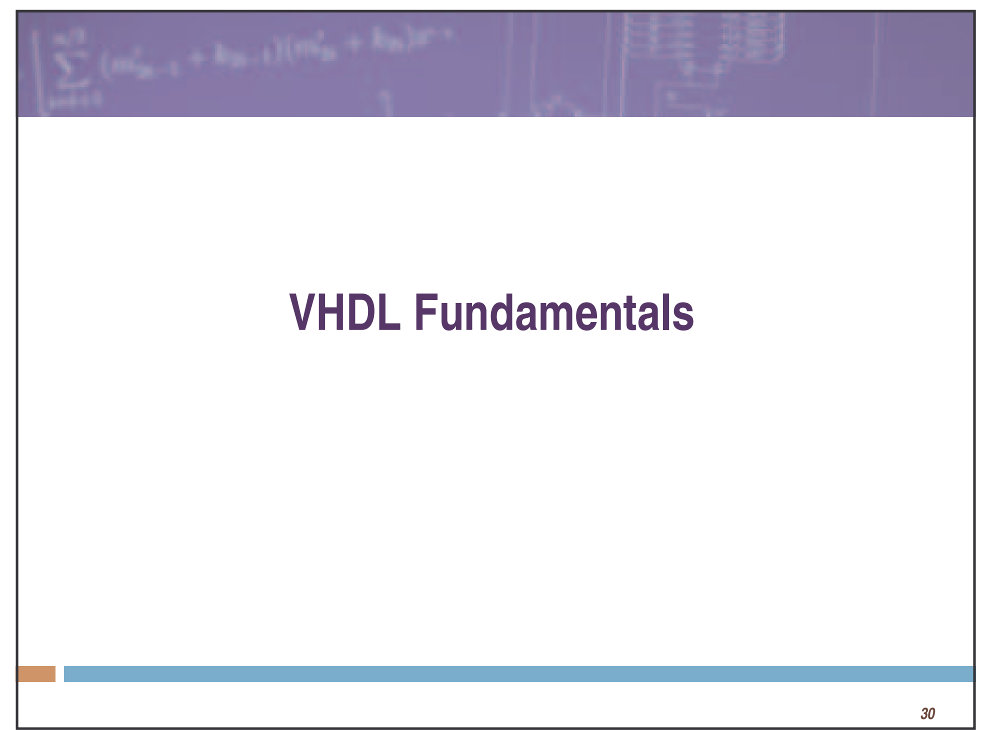
- This course will focus on FPGA design flows
- ECE 681 focuses on ASIC design flows

VHDL for Synthesis


Levels of VHDL Design
VHDL for Specification
VHDL for Simulation
VHDL for Synthesis
27


Levels of Design Description
Algorithmic level
Register Transfer Level
Logic (gate) level
Circuit (transistor) level
Physical (layout) level
Level of description
most suitable for synthesis
28


Register Transfer Logic (RTL) Design Description
Combinational Logic
Combinational Logic
Registers
29


VHDL Fundamentals
30

Case Sensitivity

- VHDL is not case sensitive

Example:

Names or labels

```
databus
Databus
DataBus
DATABUS
```

are all equivalent

Naming and Labeling

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

Readability Standards

ESA VHDL Modelling Guidelines

published by

European Space Research and Technology Center

in September 1994

available at the course web page

Readability Standards

Selected issues covered by ESA Guidelines:

- Consistent writing style
- Consistent naming conventions
- Consistent indentation
- Consistent commenting style
- Recommended file headers
- File naming and contents
- Number of statements/declarations per line
- Ordering of port and signal declarations
- Constructs to avoid

Comments

- Comments in VHDL are indicated with a “double dash”, i.e., “--”
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:

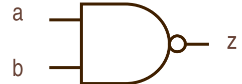
```
-- main subcircuit
Data_in <= Data_bus;    -- reading data from the input FIFO
```

Comments

- Explain function of module to other designers
- Explanatory, not just restatement of code
- Locate close to code described
  - Put near executable code, not just in a header

Design Entity

Example: NAND Gate



a	b	z
0	0	1
0	1	1
1	0	1
1	1	0

Example VHDL Code

- 3 sections to a piece of VHDL code
- File extension for a VHDL file is .vhd
- Name of the file is usually the entity name (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 DECLARATION

ENTITY

ARCHITECTURE

Fundamental Parts Of A Library

- Library is a collection of commonly used pieces of code, grouped for reuse.

LIBRARY

PACKAGE 1	PACKAGE 2
TYPES	TYPES
CONSTANTS	CONSTANTS
FUNCTIONS	FUNCTIONS
PROCEDURES	PROCEDURES
COMPONENTS	COMPONENTS

Library Declarations

```
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 declaration

Use all definitions from the package std_logic_1164

Library Declarations - Syntax

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

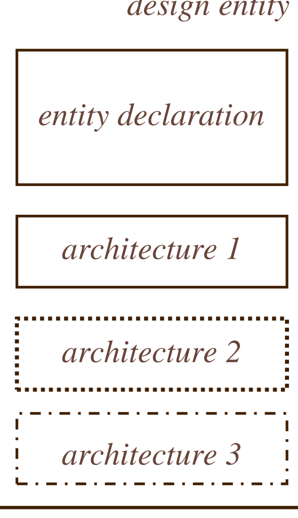

Commonly Used Libraries

- ieee
  - Specifies multi-level logic system including STD_LOGIC, and STD_LOGIC_VECTOR data types
- 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
  - User-created designs after compilation

Needs to be explicitly declared (ieee)

Visible by default (std, work)

Design Entity

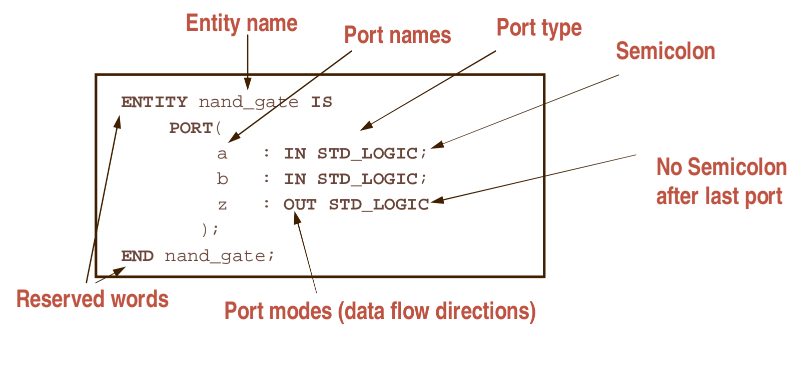


Design Entity - most basic building block of a design.

One *entity* can have many different *architectures.*

Entity Declaration

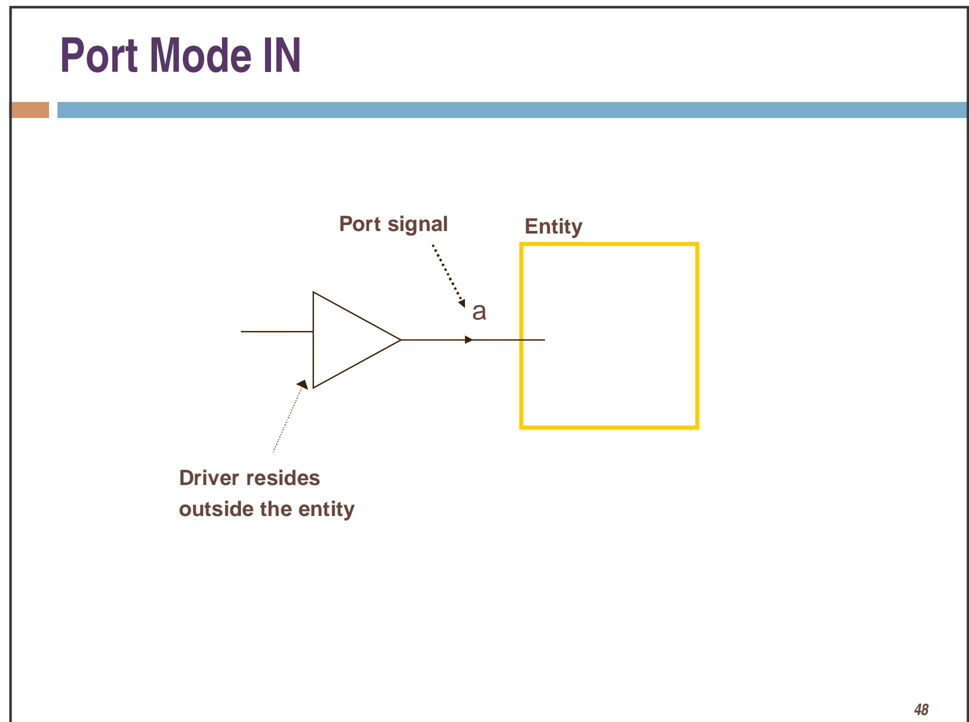
- Entity Declaration describes the interface of the component, i.e. input and output ports.

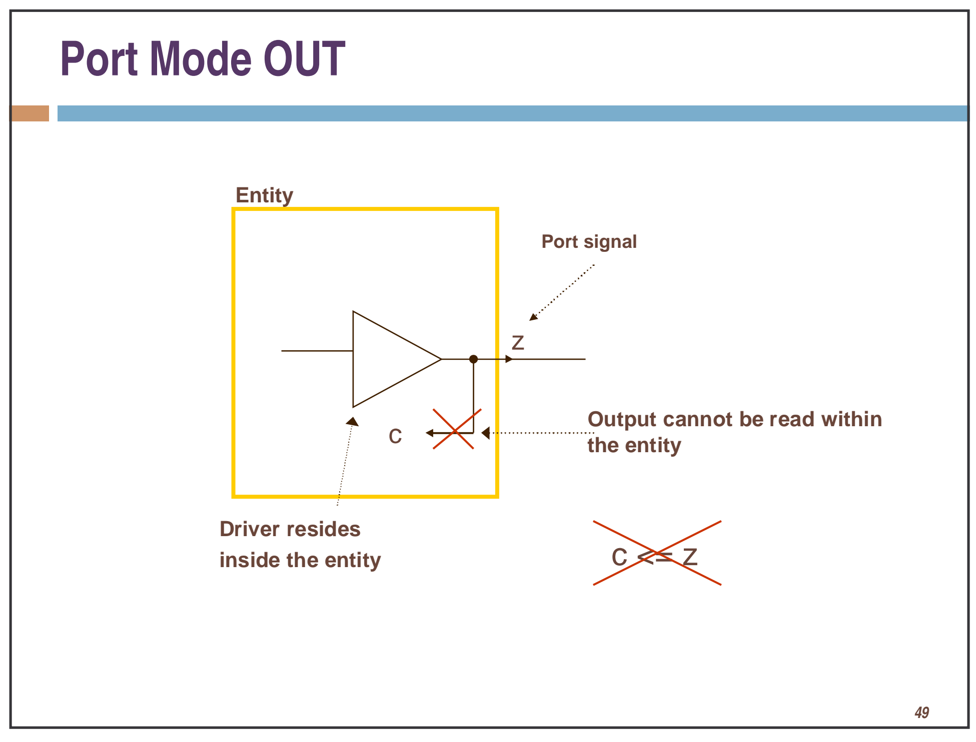


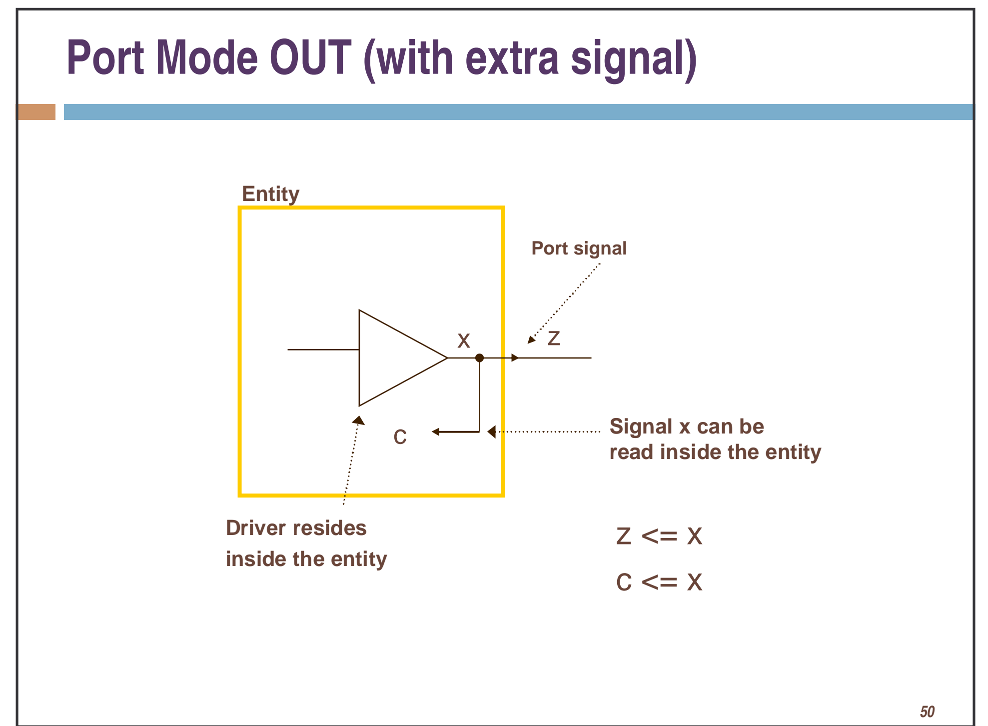
Entity Declaration – Simplified Syntax

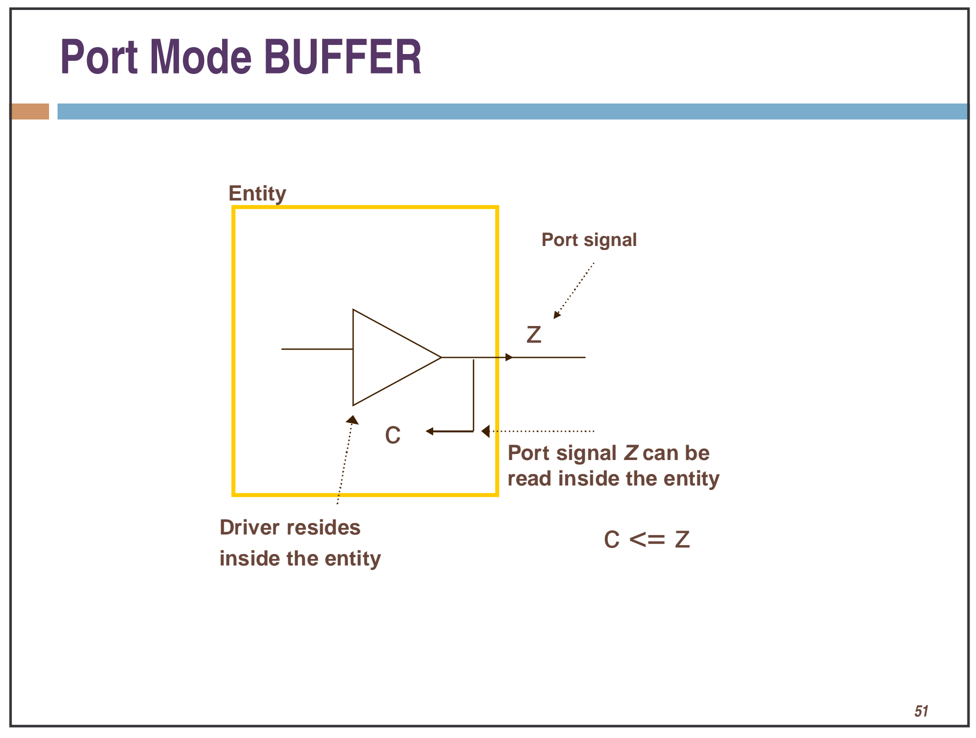
```
ENTITY entity_name IS
   PORT (
      port_name :  port_mode  signal_type;
      port_name :  port_mode  signal_type;
      ………….
      port_name :  port_mode  signal_type);
END entity_name;
```

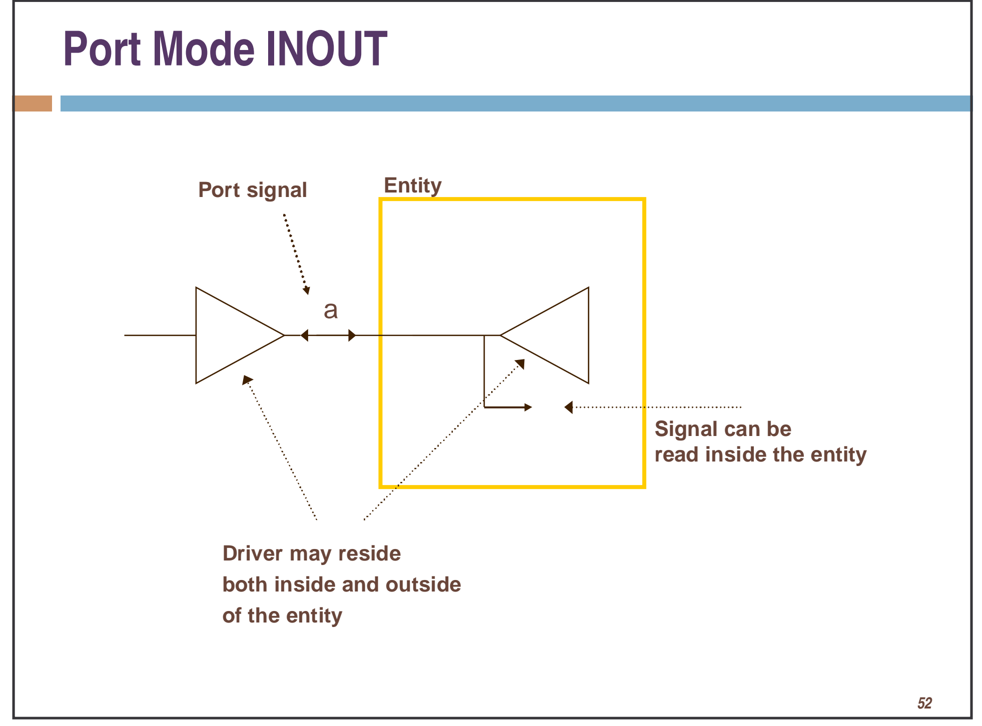

Port Mode IN




Port Mode OUT
Entity
Port signal
Z
C
Output cannot be read within the entity
Driver resides inside the entity
C <= Z
49


Port Mode OUT (with extra signal)
Entity
Port signal
X
Z
C
Signal x can be read inside the entity
Driver resides inside the entity
Z <= X
C <= X
50


Port Mode BUFFER
Entity
Port signal
Z
C
Port signal Z can be
read inside the entity
Driver resides
inside the entity
C <= Z
51


Port Mode INOUT
Port signal
Entity
a
Signal can be
read inside the entity
Driver may reside
both inside and outside
of the entity
52

Port Modes: Summary

The *Port Mode* of the interface describes the direction in which data travels with respect to the *component*

- **In**: Data comes in this port and can only be read within the entity. It can appear **only on the right side** of a signal or variable assignment.
- **Out**: The value of an output port can only be updated within the entity. **It cannot be read**. It can only appear **on the left side** of a signal assignment.
- **Inout**: The value of a bi-directional port can be read and updated within the entity model. It can appear on **both sides** of a signal assignment.
- **Buffer:** Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement the signal can appear on the left and right sides of the <= operator

INOUT AND BUFFER NOT USED IN ECE 545

Architecture

- Entity describes the **ports** of the module
- Architecture describes the **functionality** of the module (i.e. what does the module do)
- 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;
```

STD_LOGIC Demystified

STD_LOGIC

```
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;
```

What is **STD_LOGIC** you ask?

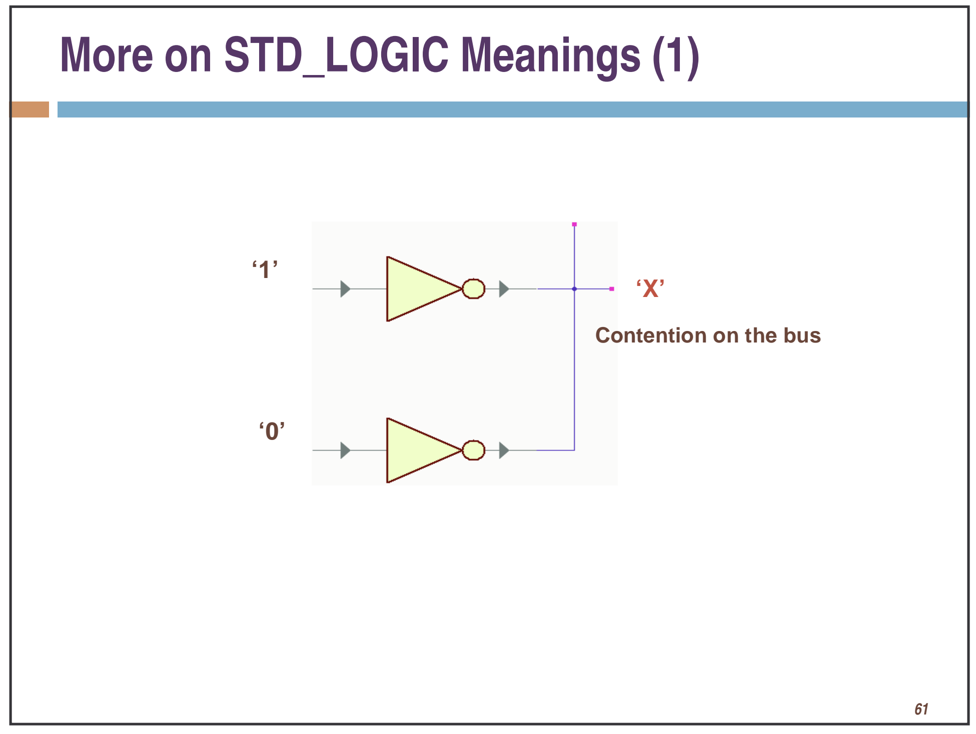
BIT versus STD_LOGIC

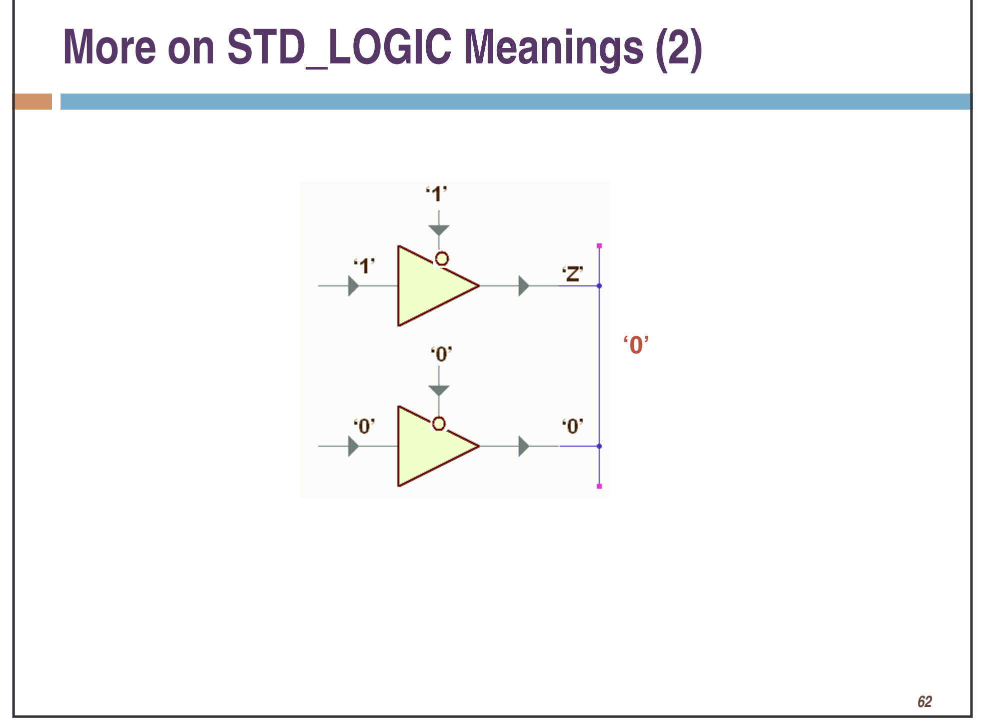
- BIT type can only have a value of ‘0’ or ‘1’
- STD_LOGIC can have nine values
  - 'U',‘0’,’1’,’X’,’Z’,’W’,’L’,’H’,’-’
  - Useful mainly for simulation
  - ‘0’,’1’, and ‘Z’ are synthesizable

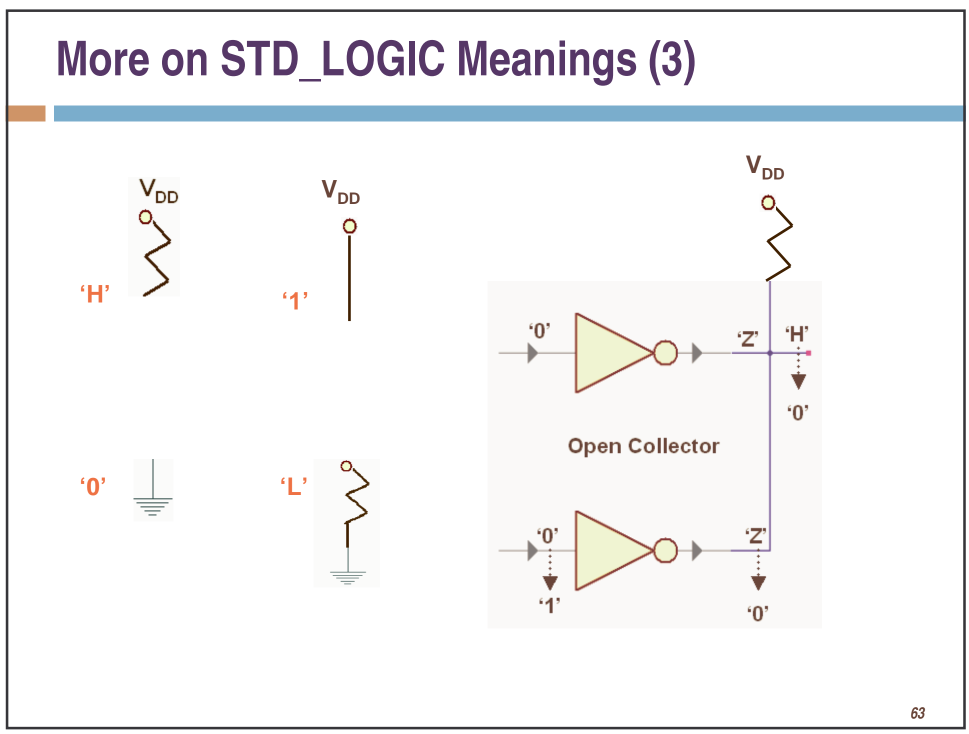
STD_LOGIC *type* demystified

Value	Meaning
'U'	Uninitialized
‘X’	Forcing (Strong driven) Unknown
‘0’	Forcing (Strong driven) 0
‘1’	Forcing (Strong driven) 1
‘Z’	High Impedance
‘W’	Weak (Weakly driven) Unknown
‘L’	Weak (Weakly driven) 0. Models a pull down.
‘H’	Weak (Weakly driven) 1. Models a pull up.
‘-’	Don't Care

Pedroni is missing 'U' in the textbook


More on STD_LOGIC Meanings (1)
'1'
'0'
'X'
Contention on the bus
61


More on STD_LOGIC Meanings (2)
'1'
'1'
'Z'
'0'
'0'
'0'
'0'
62



More on STD_LOGIC Meanings (4)

- *Do not care.*
- *Can be assigned to outputs for the case of invalid inputs (may produce significant improvement in resource utilization after synthesis).*
- *Use with caution*
  '1' = '-' gives FALSE

Resolving Logic Levels

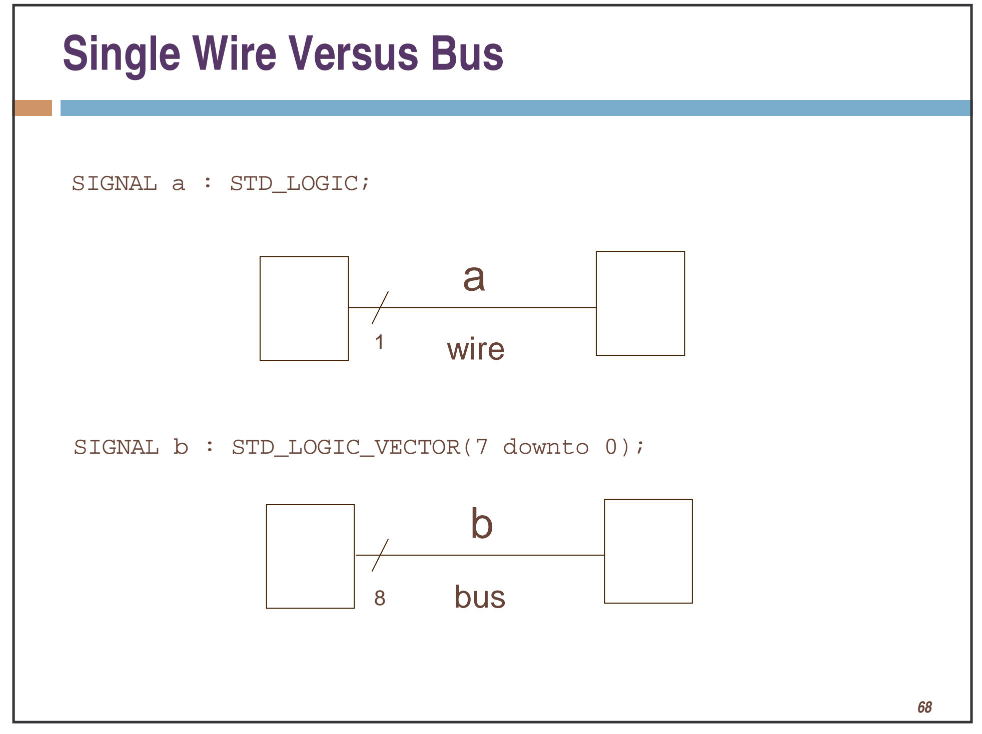
	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

STD_LOGIC Rules

- In ECE 545, use **std_logic** or **std_logic_vector** for all entity input or output ports
  - Do not use integer, unsigned, signed, bit for ports
  - Can use other types inside an architecture if desired
  - Can use other types in generics
- Instead use std_logic_vector and a conversion function inside your architecture
  - More on this in your book and in a later lecture

Modeling Wires and Buses

Single Wire Versus Bus



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
```


Single versus Double Quote

- Use single quote to hold a single bit signal
  - `a <= '0', a <='Z'`
- Use double quote to hold a multi-bit signal
  - `b <= "00", b <= "11"`

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"
```


STD_LOGIC versus STD_ULOGIC

- STD_LOGIC resolves types which conflict via the conflict resolution table
- STD_ULOGIC does not allow for conflicts on buses and will not resolve multiple sources for a signal
- We only use STD_LOGIC in this course

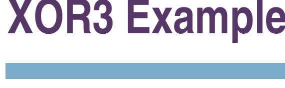
VHDL Design Styles

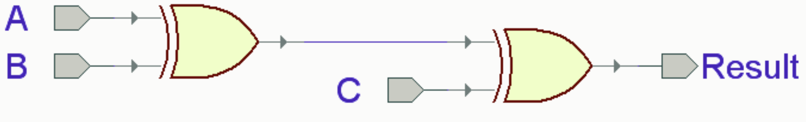
VHDL Design Styles (Architecture)

VHDL Design Styles

- DATAFLOW — *"concurrent" statements*
- STRUCTURAL — *components and interconnects*
- BEHAVIORAL — *"sequential" statements*
  - SYTHESIZABLE
    - State machines
    - Registers
  - NON-SYNTHESIZABLE
    - Test Benches
    - Modeling IP

VHDL subset most suitable for synthesis



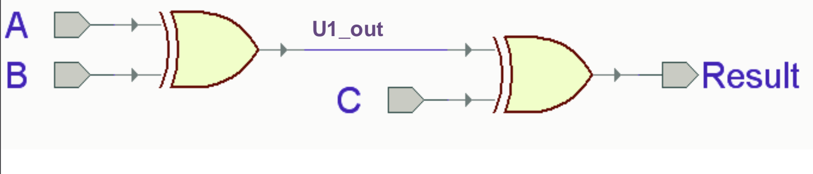


Entity XOR3 (same for all architectures)

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

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

Dataflow Architecture

```
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.
  - Dataflow uses series of concurrent statements to realize logic.
  - Dataflow is most useful style when series of Boolean equations can represent a logic → used to implement simple combinational logic
  - Dataflow code also called "concurrent" code (Pedroni)
- Concurrent statements are evaluated at the same time; thus, the order of these statements doesn't matter
  - This is not true for sequential/behavioral statements

This order…

```
U1_out <= A XOR B;
Result <= U1_out XOR C;
```

Is the same as this order…

```
Result <= U1_out XOR C;
U1_out <= A XOR B;
```

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;
```

A
B
C
XOR3
Result

Component and Instantiation

- Named association connectivity (recommended)

```
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);
...
```

COMPONENT PORT NAME LOCAL WIRE

Component and Instantiation

- Positional association connectivity (not recommended)

```
COMPONENT xor2 IS
     PORT(
       I1 : IN STD_LOGIC;
       I2 : IN STD_LOGIC;
       Y : OUT STD_LOGIC
          );
END COMPONENT;
BEGIN
U1: xor2 PORT MAP (A, B, U1_OUT);
...
```


Structural Description

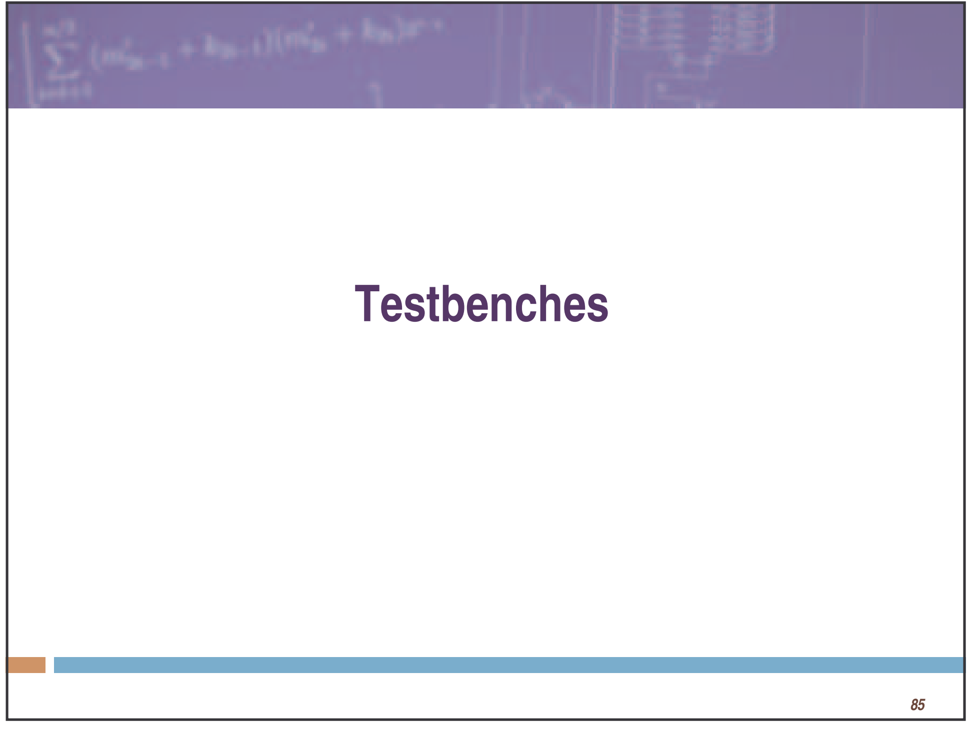
- Structural design is the simplest to understand. This style is the closest to schematic capture and utilizes simple building blocks to compose logic functions.
- Components are interconnected in a hierarchical manner.
- Structural descriptions may connect simple gates or complex, abstract components.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

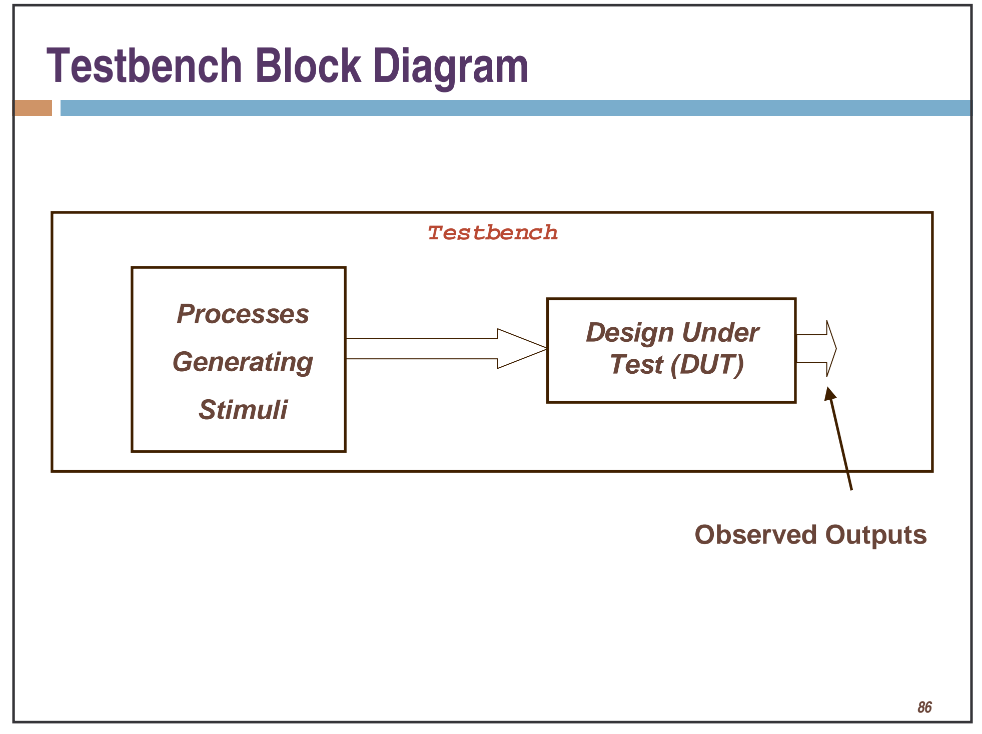
Behavioral Architecture (XOR3 gate)

```
ARCHITECTURE behavioral OF xor3 IS
BEGIN

  PROCESS (A,B,C)
  BEGIN
      IF ((A XOR B XOR C) = '1') THEN
             Result <= '1';
      ELSE
             Result <= '0';
      END IF;
  END PROCESS;
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.
- Statements are executed in sequence in a process statement → order of code matters!


Testbenches
85


Testbench Block Diagram
Testbench
Processes
Generating
Stimuli
Design Under
Test (DUT)
Observed Outputs
86

Testbench Defined

- A *testbench* applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
  - Design Under Test also called Unit Under Test (UUT)
- The results can be viewed in a waveform window or written to a file.
- Since a t*estbench* 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

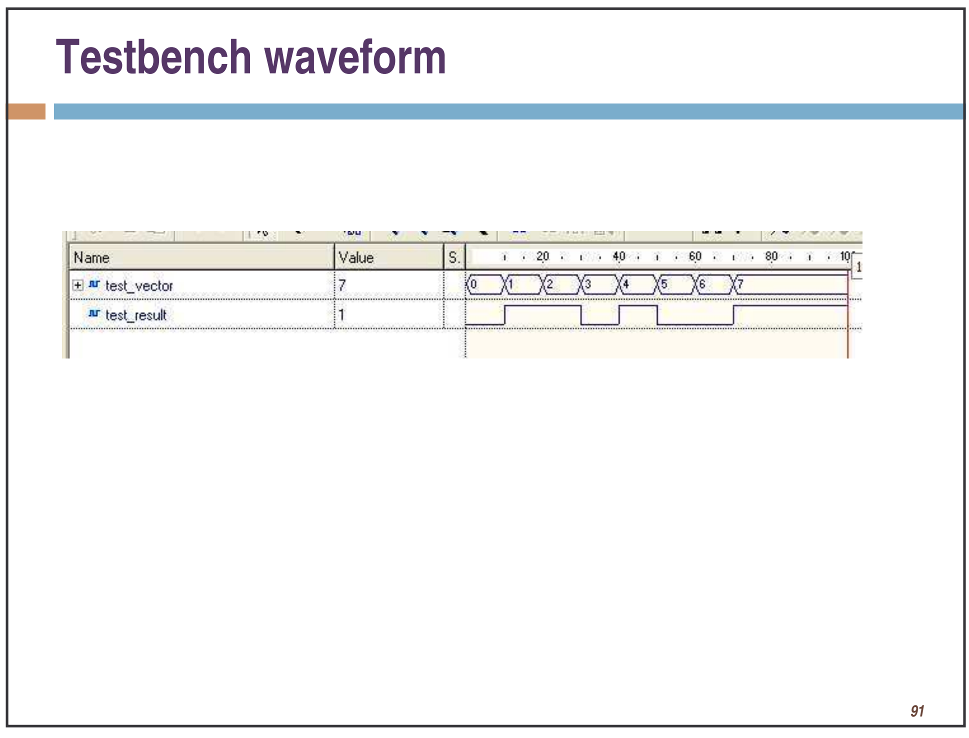
```
ENTITY tb IS
              --TB entity has no ports
END tb;

ARCHITECTURE arch_tb OF tb IS

  --Local signals and constants

  COMPONENT TestComp -- All Design Under Test component declarations
            PORT (   );
  END COMPONENT;
-----------------------------------------------------
BEGIN
   DUT:TestComp PORT MAP(  }   -- Instantiations of DUTs
                       );  }

   testSequence: PROCESS
                              -- Input stimuli
   END PROCESS;

END arch_tb;
```

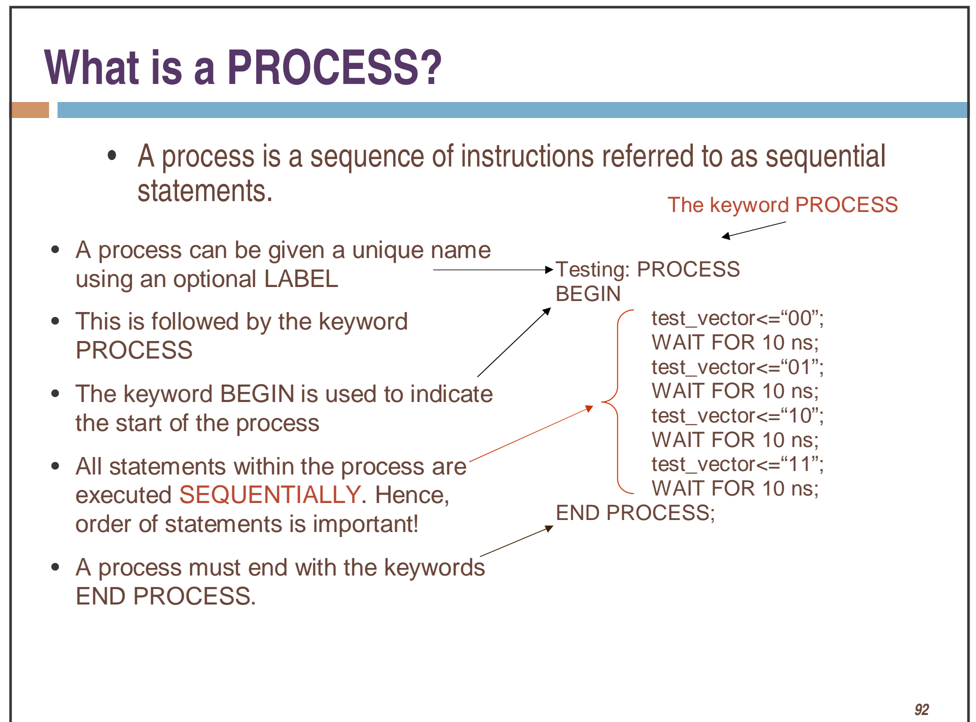
Testbench for XOR3

```
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 ports of tested entity
 SIGNAL test_vector: STD_LOGIC_VECTOR(2 DOWNTO 0);
 SIGNAL test_result : STD_LOGIC;
BEGIN
 DUT : xor3
    PORT MAP (
          A => test_vector(0),
          B => test_vector(1),
          C => test_vector(2),
          Result => test_result);
```


Testbench for XOR3 (2)

```
 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;
  END PROCESS;
END xor3_tb_architecture;
```


Testbench waveform
Name
Value
S.
20
40
60
80
test_vector
7
test_result
1
0
1
2
3
4
5
6
7
91


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;
92

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;
```

Order of execution

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.

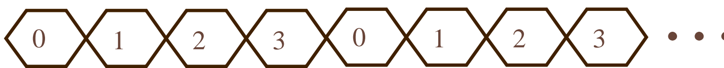
```
Testing: PROCESS
BEGIN
    test_vector<="00";
    WAIT FOR 10 ns;
    test_vector<="01";
    WAIT FOR 10 ns;
    test_vector<="10";
    WAIT FOR 10 ns;
    test_vector<="11";
    WAIT;
END PROCESS;
```

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;
```