



IEEE

Alexandria
Student Branch

Digital Design Using VHDL

Prepared by:

Eng. Wael Mahmoud El Sharkasy

Teacher Assistant

EE Department, Alexandria University

Contents

Session 1: INTRODUCTION

Session 2: VHDL BASICS

Session 3: SEQUENTIAL STATEMENTS IN VHDL

Session 4: MODELING SEQUENTIAL CIRCUIT

Session 5: FINITE STATE MACHINE

Session 6: FPGAs

بسم الله الرحمن الرحيم

Digital Design using VHDL

Eng. Wael Mahmoud El Sharkasy

Teacher Assistant

EE Department , Alexandria University

Objectives

- I. Learning the digital design flow from writing VHDL code down to physical device verification.
- II. Learning some Design Tools from Xilinx and Mentor Graphics.
- III. Learning the basics of FPGAs and how to implement designs on it.
- IV. Working on a final big project to apply what we have studied.

Session 1

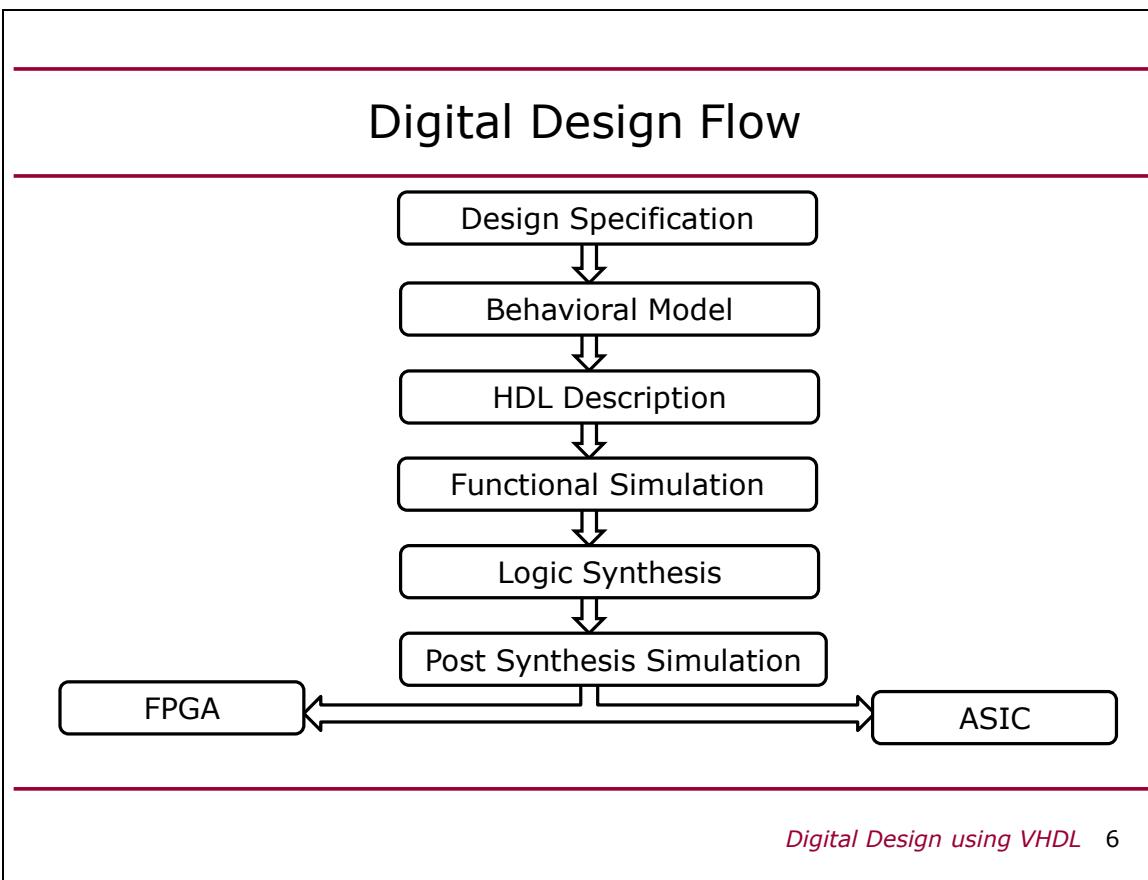
INTRODUCTION

Digital Design using VHDL 4

Topics for this Lecture

- Digital Design Flow
- Why do we need HDL?
- What is VHDL?
- History of VHDL
- IEEE Standard Extension
- Scope of VHDL
- Test Benches
- Structure of a VHDL Program

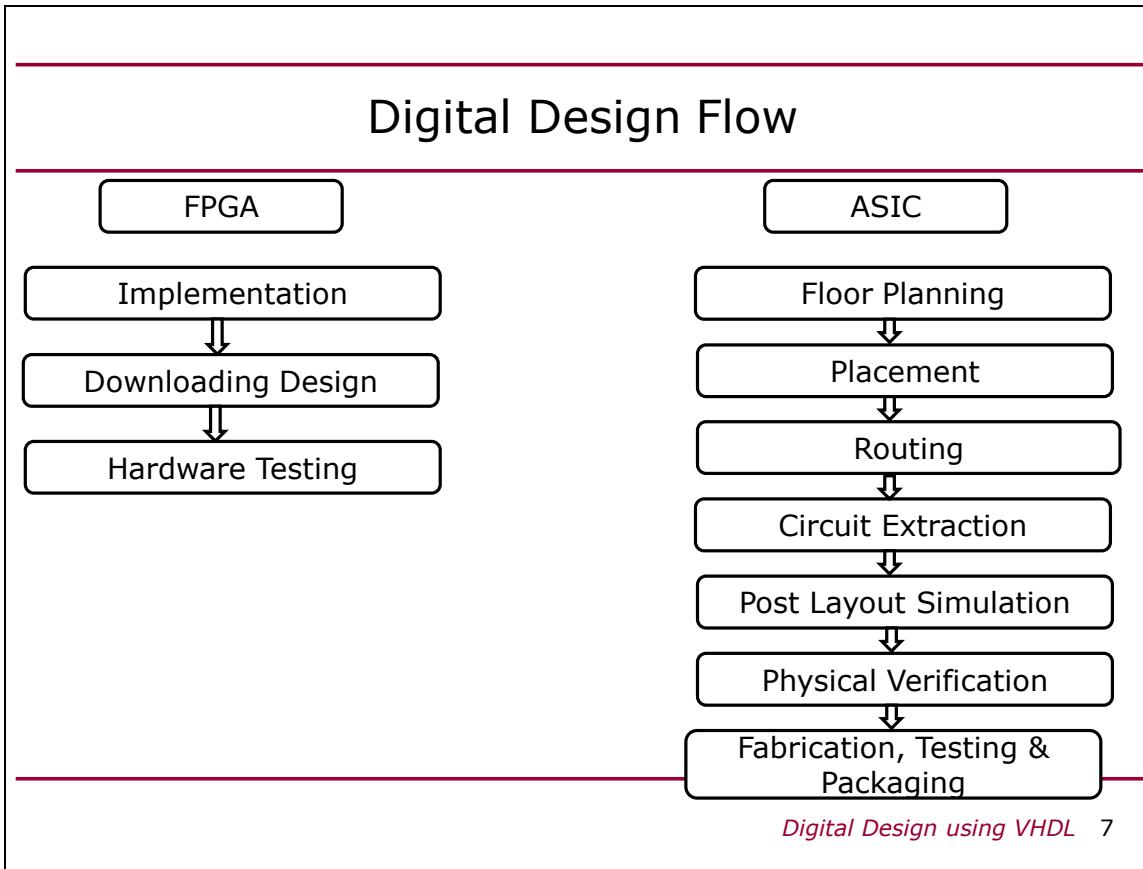
Digital Design using VHDL 5



Digital Design Flow:

1. **Design Specification:** Determining the specification of the design, its function, its performance requirement and its physical requirements such as size and power dissipation.
2. **Behavioral Model:** The design specification is used to create a high level behavioral model usually with a high level programming language such as C/C++ and Matlab. It can also be done by a HDL such as Verilog, VHDL and SystemC. Behavioral simulation can be made to verify that the desired functionality are reached. The design could be divided into smaller blocks for implementation.
3. **HDL Description (also known as RTL Description):** The design is implemented using HDL language such as VHDL and Verilog. The design is usually described at the level of registers, memories, arithmetic units and state machines which is known as Register-Transfer-Level (RTL).

4. Functional Simulation: The HDL code must be simulated to verify that it meets the required function. The simulation here doesn't have any timing constraints, only we simulate the function of the design.
 5. Logic Synthesis: It is the process in which the HDL code is converted to the gate level. The inputs to the synthesis process are the HDL code and the technology files. The output of the synthesis process is called Netlist and it contains gates connected to each other to perform the same function as the HDL code. Also the synthesis process gives another output which describes the delay of the gates in the netlist.
 6. Post Synthesis Simulation: To verify that the gate level circuit provides the desired functionality and meet the timing requirements.
- The rest of the design process is mainly physical design stage so it depends on the target implementation whether the design will be implemented on FPGA or will be fabricated to give an IC with a specific application (ASIC).



➤ For FPGA Implementation:

7. Implementation (Placing & Routing): For the moment, think of an FPGA as hardware device that provides a large number of gates and flip flops that we can connect as we wish according to our design. This gate level circuit will be placed on the FPGA using the gates and flip flops provided by the chip. The corresponding gates and flip flops must now be connected by routing signals between them on the chip. We can do another timing simulation since we can now derive accurate estimates of the delays through the wires and gates since these delays are available from the FPGA chip specification.
8. Downloading the design on the FPGA and testing it.

➤ For ASIC Implementation:

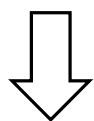
9. Floor Planning: Arranging the blocks of the netlist on the chip.



10. Placement: Decide the location of cells in the blocks.
11. Routing: Make the connection between cells and blocks.
12. Circuit Extraction: Determine the parasitic resistance and capacitance of each net.
13. Post Layout Simulation. Check to see the design still works with the added loads of the interconnects and still meet the timing constraints.
14. Physical Verification: This includes DRC (Design Rule Checking) and LVS (Layout vs. Schematic).
15. Fabrication of the design, testing the fabricated chip and packaging it to have the final product IC.

Why do we need HDL ?

- We want to describe Hardware.
- Hardware works concurrently.
- General purpose programming languages such as C works sequentially.
- We need programming languages, that works concurrently as the hardware, to describe hardware.



Hardware Description Language
(HDL)

- Examples: VHDL, Verilog.

Notes:

There are wide varieties of computer programming languages, from Fortran to C to Java. Unfortunately, they are not suitable for describing and modeling digital hardware. This is because most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be changed.

The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When a signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete.



After completion, each part updates the value of the corresponding output port.

The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages (i.e., HDLs) that are designed to model digital hardware.

HDL program is used in documentation, simulation and modeling of digital systems.

What is VHDL ?

- VHDL stands for VHSIC(Very High Speed Integrated Circuit) Hardware Description Language.
- It describes the behavior and structure of digital electronic system.
- It is an IEEE standard.

Digital Design using VHDL 9

Notes:

VHDL is adopted as a standard language in the electronic design community. Using a standard language guarantees that you will not have to throw away your designs simply because the design entry method is not supported in a newer generation of design tools.



History of VHDL

- 1981 : Initiated by US DoD to address hardware life-cycle Crisis.
- 1983-5 : Development of baseline language by Intermetrics, IBM and TI.
- 1986 : All rights transferred to IEEE
- 1987 : Publication of IEEE Standard 1076-1987
- 1994 : Revised standard, VHDL 1076-1993
- 2000 : Revised Standard, VHDL 1076-2000.
- 2002 : Revised Standard, VHDL 1076-2002.
- 2008 : VHDL-2008 was released.

Digital Design using VHDL 10

Notes:

The development of VHDL was initiated in 1981 by the United States Department of Defense to address the hardware life cycle crisis. The cost of reprocuring electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would work the same on any simulator and was independent of technology or design methodology. In July 1983, a team of Intermetrics, IBM and Texas Instruments were awarded a contract to develop VHDL. In August 1985, the final version of the language under government contract was released.

In 1986, all rights to the language definition were given away by the DoD to the IEEE in order to encourage industry acceptance and investment.

In 1987 VHDL became an IEEE standard.

As an IEEE standard, VHDL must undergo a review process every 5 years (or sooner) to ensure its ongoing relevance to the industry. The first such revision was completed in September 1993, and VHDL '93 is officially the current VHDL standard.



IEEE Standard Extensions

- 1076.1: VHDL-AMS (Analog and Mixed Signal)
- 1076.2: VHDL Math Package
- 1076.3: VHDL Synthesis Package (NUMERIC_STD)
- 1076.4: VHDL Timing Methodology (VITAL)
- 1076.6: VHDL Subset for RTL Synthesis
- 1164: VHDL Model Practices (STD_LOGIC_1164)

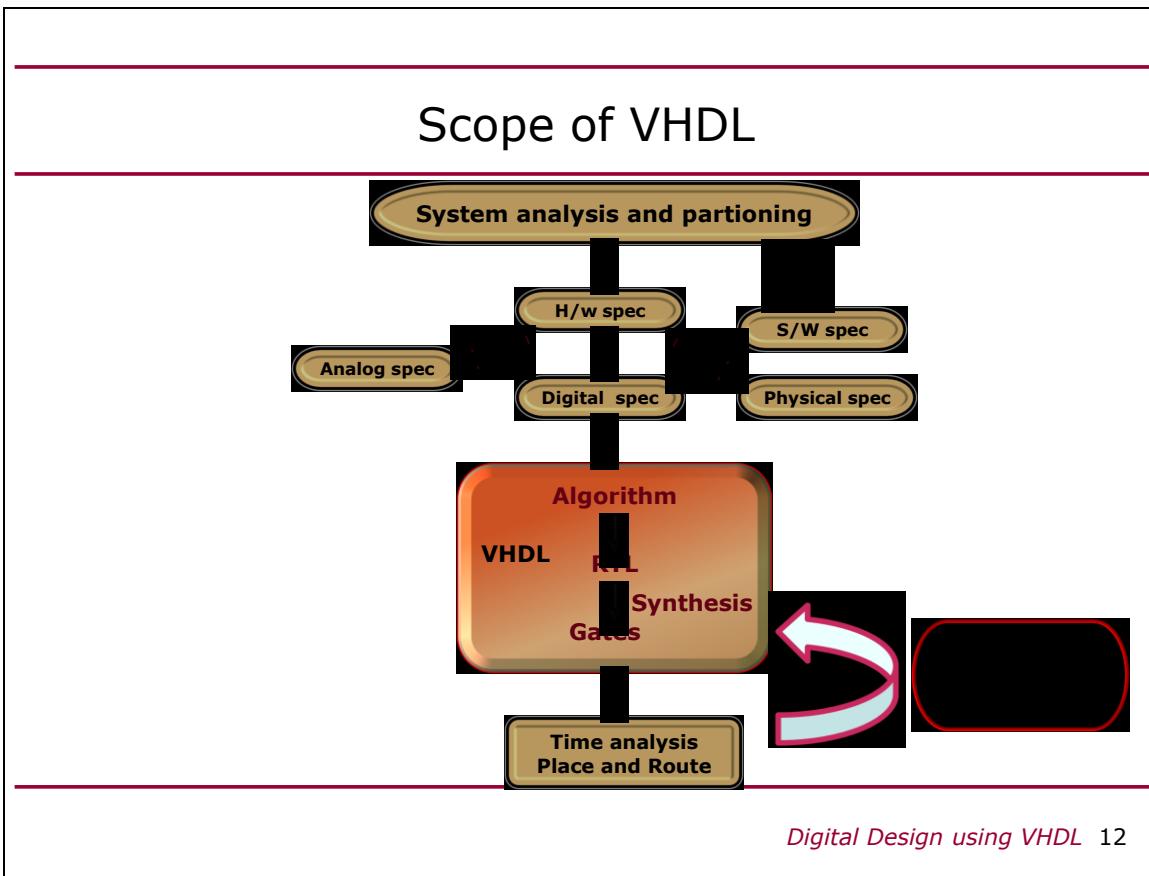
Digital Design using VHDL 11

Notes:

After the initial release of the standard, various extensions were developed to facilitate various design and modeling requirements. These extensions are documented in several IEEE standards:

- IEEE standard 1076.1-1999, VHDL Analog and Mixed Signal Extensions (VHDL-AMS): defines the extension for analog and mixed-signal modeling and simulation.
- IEEE standard 1076.2-1996, VHDL Mathematical Packages: defines extra mathematical functions for real and complex numbers.
- IEEE standard 1076.3- 1997, Synthesis Packages: defines arithmetic operations over a collection of bits.
- IEEE standard 1076.4-1995, VHDL Initiative Towards ASIC Libraries (VITAL): defines a mechanism to add detailed timing information to ASIC cells.
- IEEE standard 1076.6-1999, VHDL Register Transfer Level (RTL) Synthesis: Defines how VHDL is to be written for RTL synthesis.

-IEEE standard 1164- 1993 :Multivalue Logic System for VHDL Model inter operability
(std_logic_1164): defines new data types to model multivalue logic.



Notes:

VHDL is suited to the specification, design and description of digital electronic hardware.

➤ System Level:

VHDL is not ideally suited for abstract system-level simulation, prior to the hardware-software split. VHDL has been used in this area with some success.

➤ Digital:

VHDL is suitable for use today in the digital hardware design process, from specification through high-level functional simulation, manual design and logic synthesis down to gate-level simulation. The gate-level netlist which is the output of the synthesis process is represented in structure VHDL description. The placement and routing tool will generate the layout or configuration files, which are not in VHDL.

➤ Analog:

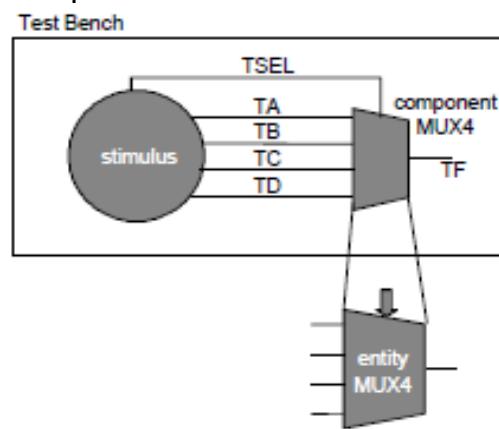
Because of VHDL's flexibility as a programming language, it has been stretched to handle analog and switch level simulation in limited cases. A separate standard named VHDL-AMS provides analog and mixed signal extensions to VHDL.

➤ Design Process:

The diagram opposite shows a very simplified view of the electronic system design process incorporating VHDL. The central portion of the diagram shows the parts of the design process that will be impacted by VHDL

Test Benches

- VHDL can be used as a stimulus definition language as well as a hardware description language.
- We can develop VHDL code to generate a test vector and to collect and compare the output responses.
- This is known as a test bench.



Digital Design using VHDL 13

Notes:

Simulating our design can be done similar to doing an experiment with a physical circuit, in which we connect the circuit's input to a stimulus (e.g., a function generator) and observe the output (e.g., by logic analyzer). Simulating a VHDL description is like doing a virtual experiment, in which the physical circuit is replaced by the corresponding VHDL description while the stimulus and the output observer is replaced by a test bench.

In a simple test bench, the test bench simply generates some stimulus to drive the inputs of the design-under-test. You would have to rely on the display facilities of the VHDL simulator to examine the values of the outputs and confirm that the design is functionally correct. You can also write more sophisticated test benches that are self-checking, i.e. the test bench will contain code to check that the actual value of the output is equal to some expected value, and give an error message if it fails.

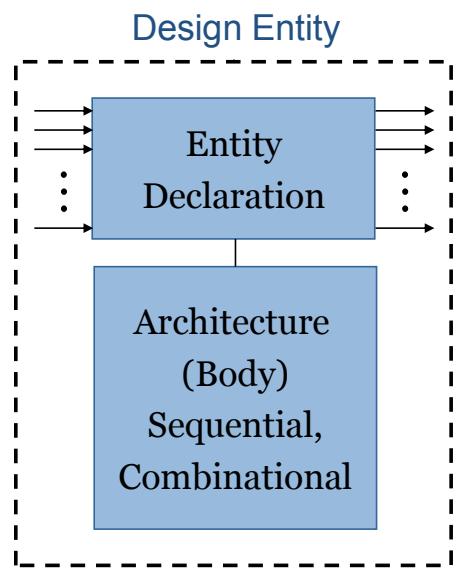
VHDL Syntax

- Not case sensitive
- Comments begin with --
- Statements end with ;
- Signal/entity names: letter followed by letters, digits.

Digital Design using VHDL 14

Structure of a VHDL Program

- Design Entity
 - Entity Declaration
 - Architecture Body
- Library and Packages



Digital Design using VHDL 15

Notes:

- A VHDL design entity represents a block of hardware with well-defined inputs and outputs and a well-defined function.



Design Entity

Entity Declaration

- It describes the external interface or the outline of the design.
- It includes the name of the design and the names & the basic characteristics of its input output ports.
- Syntax:

```
entity NAME_OF_ENTITY is
    port (port_names: mode data_type;
          port_names: mode data_type;
          :
          port_names: mode data_type);
end NAME_OF_ENTITY ;
```

Digital Design using VHDL 16

Notes:

➤ Ports:

A port is like a pin of an IC. Each port declaration must include the name of one or more ports, mode and data type.

-The mode indicates the direction of data flow through this port. It may be (in, out, inout or buffer). The port can be input or output or bidirectional. We will not use the buffer mode as it can cause some problems in synthesis.

-The data_type can be (bit, bit_vector, std_logic, std_logic_vector, integer,.....). We will discuss them later. We will mainly use std_logic and std_logic_vector as data types.

➤ The words in bold and blue are reserved VHDL keywords.

➤ In VHDL 87, the entity ends by “end NAME_OF_ENTITY”. In VHDL 93, it can also ends by “end entity NAME_OF_ENTITY”.

➤ Note that each statement ends by a semicolon (;) except the last port declaration.

Design Entity

Entity Declaration

- Example: 2 input AND gate:

```
entity AND2 is
  port (A, B : in std_logic;
        Y: out std_logic);
end AND2 ;
```

- Example: 4-to-1 MUX each input is 8 bits

```
entity mux4_to_1 is
  port (I0 ,I1 ,I2 ,I3 : in std_logic_vector(7 downto 0);
        S : in std_logic_vector(1 downto 0);
        OUT1: out std_logic_vector(7 downto 0));
end mux4_to_1;
```

Digital Design using VHDL 17

Design Entity

Architecture Body

- It specifies the internal operation or organization of the design.
- In VHDL, we can develop multiple architecture for the same entity and later we can choose one of them to bind with the entity for simulation or synthesis.
- It contains concurrent signal assignments which describe the function of the design entry.
- It may includes an optional declaration section which consists of declarations of some objects.

Design Entity

Architecture Body

- Syntax:

```
architecture architecture_name of NAME_OF_ENTITY is
    -- Declarations
    -- components declarations
    -- signal declarations
    -- constant declarations
    -- type declarations
    :
    begin
    -- concurrent statements
    -- concurrent statements
    :
end architecture_name;
```

Digital Design using VHDL 19

Notes:

- The first line of the architecture body contains the name of the body and the corresponding entity.
- The main part of the architecture body consists of the concurrent statements that describe the behavior or the structure of the design. Sequential statements can also be written inside a process as we will show later.



Library & Packages

- A VHDL package is a file or module that contains declarations of commonly used items such as data type, component declarations, signal and functions.
- A library can be considered as a place where the compiler stores information about a design project.
- To specify the library and package, use the **library** and the **use** keywords.
- Example: to include the **std_logic_1164 package** that exists in the **library ieee**.
`library ieee;`
`use ieee.std_logic_1164.all;`
- The **.all** extension indicates to use all of the `ieee.std_logic_1164` package.

Digital Design using VHDL 20

Notes:

➤ Library:

It is the place to store design units. It is normally mapped into a directory in the computer's hard disk.

By VHDL default the design units will be stored in a library named work.

➤ Packages:

To facilitate the synthesis, IEEE has developed several VHDL packages, including the std-logic-1164 package and the numeric-std package, which are defined in IEEE standards 1164 and 1076.3. To use a predefined package, we must include the library and use statements before the entity declaration.

In the above example, the first line invokes a library named ieee, and the second line makes the std-logic-1164 package visible to the subsequent design unit. We must invoke this library because we want to use some predefined data types, std_logic and std_logic_vector, of the std_logic_1164 package.

Example 1

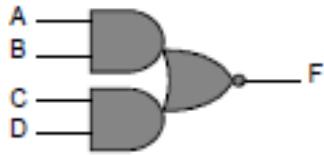
We want to write a VHDL code for the following AND-OR

Invert gate:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity AOI is
    port (A, B, C, D: in STD_LOGIC;
          F : out STD_LOGIC);
end AOI;

architecture V1 of AOI is
begin
    F <= not ((A and B) or (C and D));
end V1;
```



Digital Design using VHDL 21

Notes:

- The “`<=`” means assignment operation. The assignments within the architecture are concurrent signal assignments. Such assignments execute whenever a signal on the right hand side of the assignment changes value.
- Delays:

Each of the concurrent signal assignments has a delay. The expression on the right hand side is evaluated whenever a signal on the right hand side changes value, and the signal on the left hand side of the assignment is updated with the new value after the given delay if a delay is given.

Example 2: Half Adder

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Half_adder is
    port (A,B: in STD_LOGIC;
          carry,result : out STD_LOGIC);
end Half_adder;

architecture V1 of Half_adder is
begin
    carry<= A and B after 2ns;
    result<= A xor B after 3ns;
end V1;

```



Digital Design using VHDL 22

Notes:

➤ The assignments within the architecture are concurrent signal assignments. Because of this, the order in which concurrent assignments are written has no effect on their execution. The assignments are concurrent because potentially the two assignments executes at the same time.

➤ Delays:

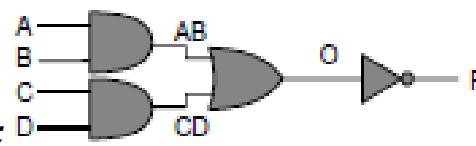
Each of the concurrent signal assignments has a delay. The expression on the right hand side is evaluated whenever a signal on the right hand side changes value, and the signal on the left hand side of the assignment is updated with the new value after the given delay.

If no delay is mentioned, VHDL specifies that there is an implicit δ -delay (delta delay) associated with the operation. A delta delay is an infinitesimal delay that is greater than zero but smaller than any physical number.

Example 3

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity AOI is
    port (A, B, C, D: in STD_LOGIC;
          F : out STD_LOGIC);
end AOI;

architecture V2 of AOI is
signal AB, CD, O: STD_LOGIC;
begin
AB <= A and B after 2 NS;
CD <= C and D after 2 NS;
O <= AB or CD after 2 NS;
F <= not O after 1 NS;
end V2;
```



Digital Design using VHDL 23

Notes:

➤ Signal:

Signals can be interpreted as wires that connect the internal parts. A signal is declared before the begin of an architecture, and has its own data type. The architecture here contains three signals AB, CD and O, that are used internally within the architecture.

➤ Delay:

In this example, a change on the port A would propagate through the AOI entity to the port F with a total delay of 5 NS.

➤ The order in which concurrent assignments are written has no effect on their execution because the assignments are concurrent which means that two assignments could execute at the same time (if two inputs changed simultaneously).

Session 2

VHDL BASICS

Digital Design using VHDL 2

Topics for this Lecture

- Lexical Elements
- Data Objects
- Data Types and Operators
- Modeling in VHDL
- Concurrent Signal Assignment Statements
- Structure Modeling
- Compilation
- Library & Library Declaration
- Configuration

Digital Design using VHDL 3



Lexical Elements

- **Comments:** - Starts with two dashes --
 - Has no effect on the code
- **Identifier:** - It is the name of an object in VHDL.
 - Can contain only alphabetic letters, decimal digits & underscores.
 - The first character must be a letter.
 - The last character cannot be an underscore.
 - Successive underscores are not allowed.

Digital Design using VHDL 4

Notes:

- The lexical elements are the basic syntactical units in a VHDL program (language vocabulary). They include comments, identifiers, reserved words, numbers, characters and strings.
- Comments: The comment is for documentation purposes only and has no effect on the code.
- Identifiers: It is also good to use descriptive identifier for better readability. For example, consider the name for a signal that enables the memory address buffer. The mem_addr_en is good, mae is too short, and memory_address_enable is probably too long.



Lexical Elements

➤ Reserved words:

abs	default	label	package	sla
access	disconnect	library	parameter	sll
after	downto	linkage	port	sra
alias	else	literal	postponed	srl
all	elsif	loop	procedure	strong
and	end	map	process	subtype
architecture	entity	mod	property	then
array	exit	nand	protected	to
assert		new	pure	transport
assume		next		type
assume guarantee	fairness	nor		
attribute	file	not		unaffected
begin	for	null		unit
block	force	of		until
body	function	on		use
buffer	generic	open		
bus	generate	or		variable
case	group	others		vmode
component	guarded	out		vprop
configuration				vunit
constant	if			wait
context	impure			when
cover	in			while
	inertial			with
	inout			xor
	is			xnor

Lexical Elements

- **Numbers:**
 - Can be integer (ex: 1234, 98E7), real (ex: 1.234).
 - Can be represented in other bases(ex: 2#1011#, 16#2D#).
 - Underscores can be added (ex: 12_345, 2#1011_1111_1000#).
- **Character:** - Enclosed in single quotation marks(ex: 'A', '3').
- **String:** - Sequence of characters
 - Enclosed in double quotation marks(ex: "Hello").

Digital Design using VHDL 6

Notes:

➤ Numbers: A number in VHDL can be integer, such as 0,1234 and 98E7, or real, such as 1.23456 or 9.87E6. It can be represented in other number bases. For example, 45 can be represented as 2#101101# in base 2 and 16#2D# in base 16. We can also add an underscore to enhance readability. For example, 123_456 is the same as 123456, and 2#0011_1010_1101# is the same as 2#001110101101#.



Data Objects

➤ **Signal:**

- Declared in the architecture body's declaration section.
- Syntax: **signal** signal_name , signal_name , . . . : data_type ;
- An optional initial value can be specified for simulation purpose.
Ex: signal a, b, c: std_logic := '0';
- Signal Assignment: signal_name <= projected_waveform;
- From the synthesis point of view, a signal represents a wire or "a wire with memory" (i.e., a register or latch).
- Updated when their signal assignment statement is executed, after a certain delay.

Digital Design using VHDL 7

Notes:

An object in VHDL is a named item that holds the value of a specific data type. There are four kinds of objects: signal, variable, constant and file. We do not discuss the file object since it cannot be synthesized.

➤ **Signal:** The signal is the most common object. It has to be declared in the architecture body's declaration section. The simplified syntax of signal declaration is :
signal signal_name , signal_name , . . . : data_type ;
For example, the following line declares the a, b and c signals with the std_logic data type: signal a, b, c: std_logic;
According to the VHDL definition, we can specify an optional initial value in the signal declaration. For example, we can assign an initial value of '0' to the previous signals: signal a, b, c: std_logic := '0';
While this is sometimes handy for simulation purposes, it should not be used in synthesis since not many physical devices can implement the desired effect.

The simplified syntax of signal assignment is
`signal_name <= projected_waveform;`

From the synthesis point of view, a signal represents a wire or “a wire with memory” (i.e., a register or latch).



Data Objects

➤ **Variable:**

- Declared and used in a process and is local to that process.
- Symbolic memory location where a value can be stored and modified.
- Syntax: **variable** variable_name , . . . : data_type;
- Variable Assignment: variable_name := value_expression;
- There is no direct mapping between a variable and a hardware part.
- No timing information is associated with a variable.
- Updated without any delay as soon as the statement is executed.

Digital Design using VHDL 8

Notes:

A variable is a concept found in a traditional programming language. It can be thought of as a “symbolic memory location” where a value can be stored and modified. The main application of a variable is to describe the abstract behavior of a system. It can only be declared and used in a process and is local to that process.

No timing information is associated with a variable, and thus only a value, not a waveform, can be assigned to a variable. Since there is no delay, the assignment is known as an immediate assignment and the notion `:=` is used.



Data Objects

➤ **Constant:**

- Declared in the architecture body's declaration section.
- Holds a value that cannot be changed.
- Syntax: **constant** constant_name: data_type := value_expression ;
- Improve the readability of the code and allow easy updating.

Digital Design using VHDL 9

Notes:

The value_expression term specifies the value of the constant. A simple example is:

```
constant BUS_WIDTH: integer := 32;  
constant BUS_BYTES: integer := BUS_WIDTH / 8;
```

The proper use of constants can greatly enhance readability of the VHDL code and make the code more descriptive. A simple example:

```
architecture beh1_arch of even_detector is  
signal odd : std_logic ;  
constant BUS_WIDTH: integer := 3;  
begin  
....  
tmp := '0';  
for i in (BUS_WIDTH-1) downto 0 loop  
    tmp := tmp xor a( i );  
end loop;  
....
```



Data Types and Operators

- In VHDL, each object has a data type.
- A data type is defined by:
 - A set of values that an object can take.
 - A set of operations that can be performed on objects of this data type.

Digital Design using VHDL 10

Notes:

In VHDL, an object can only be assigned a value of its type, and only the operations defined with the data type can be performed on the object. If a value of a different type has to be assigned to an object, the value must be converted to the proper data type by a type conversion function or type casting. For example, if a Boolean value is assigned to a signal of integer type or an arithmetic operation is applied to a signal of character type, the software can detect the error during the analysis stage.

To facilitate modeling and simulation, VHDL is rich in data types. Our focus is on a small set of predefined data types that are relevant to synthesis. For a signal, we are mainly confined to the std_logic, std_logic_vector, signed and unsigned data types.



Data Types and Operators

➤ **Predefined data types in VHDL:**

- Integer: -Range is from $-(2^{31} - 1)$ to $2^{31} - 1$.
-Two related subtypes: natural (0 & +ve) and positive.
- Boolean: defined as (false, true).
- Bit : defined as ('0', '1').
- Bit vector: one-dimensional array with elements of the bit data type.

Digital Design using VHDL 11

Notes:

There are large numbers of predefined data types in VHDL. Only we focus on data types that are relevant to synthesis.

➤integer: VHDL does not define the exact range of the integer type but specifies that the minimal range is from $-(2^{31} - 1)$ to $2^{31} - 1$, which corresponds to 32 bits. Two related data types (formally known as subtypes) are the natural and positive data types. The former includes 0 and the positive numbers and the latter includes only the positive numbers.

Data Types and Operators

➤ **Predefined operators in VHDL:**

operator	Description	Data type operand a	Data type operand b	Data type of result
a ** b	Exponentiation	Integer	integer	integer
abs a	absolute value	Integer		integer
not a	negation	boolean, bit bit_vector		boolean, bit bit_vector
a * b a / b a mod b a rem b	multiplication division modulo remainder	integer	integer	Integer
+ a - a	identity negation	integer		Integer
a + b a - b	Addition Subtraction	integer	integer	Integer
a & b	concatenation	1-D array, element	1-D array element	1-D array

Data Types and Operators

➤ **Predefined operators in VHDL:**

a sll b a srl b asla b a sra b a rol b a ror b	shift-left logical shift-right logical shift-left arithmetic shift-right arithmetic rotate left rotate right	bit vector	integer	bit vector
a = b a /= b a < b a <= b a > b a >= b	equal to not equal to less than less than or equal to greater than greater than or equal to	any	same as a	boolean
		scalar or 1D-array	same as a	boolean
a and b a or b a xor b a nand b a nor b a xnor b	and or xor nand nor xnor	boolean ,bit, bit vector	same as a	same as a



Data Types and Operators

➤ **Data types in the IEEE std_logic_1164 package :**

- To better reflect the electrical property of digital hardware, several new data types were developed by IEEE.
- To use the new data types, we must include the necessary library and use statements before the entity declaration:

```
library ieee ;  
use ieee. std_logic _1164.all ;
```

- std_logic and std_logic_vector :

- Extension to the bit and bit_vector data types.
- Has nine possible value:
('U','0', '1', 'Z', 'L', 'H', 'X', 'W', '-')
- only '0', '1' and 'Z' are used in synthesis.

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

Digital Design using VHDL 14

Notes:

➤ The original intention of the bit data type is to represent the two binary values used in Boolean algebra and digital logic. However, in a real design, a signal may assume other values, such as the high impedance of a tri-state buffer's output. To solve the problem, a set of more versatile data types, std_logic and std_logic_vector, are introduced in the IEEE std_logic_1164 package. To achieve better compatibility, we should avoid using the bit and bit_vector data types.

➤ The std_logic data type consists of nine possible values, which are shown in the following list: ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'). These values are interpreted as follows:

- '0' and '1' : stand for "forcing logic 0" and "forcing logic 1," which mean that the signal is driven by a circuit with a regular driving current.
- 'Z' : stands for high impedance, which is usually encountered in a tri-state buffer.



- 'L' and 'H': stand for "weak logic 0" and "weak logic 1," which means that the signal is obtained from wired-logic types of circuits, in which the driving current is weak.
- 'X' and 'W' : stand for "unknown" and "weak unknown." The unknown represents that a signal reaches an intermediate voltage value that can be interpreted as neither logic 0 or logic 1. This may happen because of a conflict in output (such as a logic 0 signal and a logic 1 signal being tied together). They are used in simulation for an erroneous condition.
- 'U' : stands for uninitialized. It is used in simulation to indicate that a signal or variable has not yet been assigned a value.
- '-' : stands for don't-care.

➤ A VHDL array is defined as a collection of elements with the same data type. Each element in the array is identified by an index. The std_logic_vector data type is an array of elements with the std_logic data type. It can be thought of as a group of signals or a bus in a logic circuit.

The use of std_logic_vector can best be explained by a simple example. Let us consider an 8-bit signal, a. Its declaration is

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

It indicates that the a signal has 8 bits, which are indexed from 7 down to 0. The most significant bit (MSB, the leftmost bit) has the index 7, and the least significant bit (LSB, the rightmost bit) has the index 0. We can access a single bit by using an index, such as a(7) or a(2), and access a portion of the index by using a range, such as a(7 downto 3) or a(2 downto 0).

Data Types and Operators

➤ **Data types in the IEEE std_logic_1164 package :**

- To assign value to std_logic : ex: `a<='1'`
- Different methods can be used to assign value to std_logic_vector

Ex:

```
a <= "10100000";
a <= ('1', '0', '1', '0', '0', '0', '0', '0');
a <= (7=>'1', 6=>'0', ..... ,0=>'0');
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
a <= (7|5=>'1', others=>'0');
```

Ex: `a <= (others=>'0');` equivalent to `a <= "00000000";`

Data Types and Operators

➤ **Overloaded operators in the IEEE std_logic_1164 package :**

overloaded operator	Data type Of operand a	Data type Of operand b	Data type Of result
not a	std_logic_vector std_logic		Same as a
a and b a or b a xor b a nand b a nor b a xnor b	std_logic_vector std_logic	Same as a	Same as a

- Also Concatenation operator "&" can be applied on std_logic_vector
- Arithmetic operation can't be done on std_logic but only on integer data type which is difficult to realize in hardware (range is not indicated).

Digital Design using VHDL 16

Notes:

- In VHDL, we can use the same function or operator name for operands of different data types. There may exist multiple functions with the same name, each for a different data type. This is known as overloading of a function or operator.
- In the std_logic_1164 package, all logical operators, which include not, and, nand, or, nor, xor and xnor, are overloaded with the std_logic and std_logic_vector data types. In other words, we can perform the logical operations over the objects with the std_logic or std_logic_vector data types.

Note that the arithmetic operators are not overloaded, and thus these operations cannot be applied.

➤ The concatenation operator, &, is very useful for array manipulation. We can combine segments of elements and smaller arrays to form a larger array. For example, we can shift the elements of the array to the right by two positions and append two 0's to the front:

```
y <= "00" & a(7 downto 2);
```

or append the MSB to the front (known as an arithmetic shift):

```
y <= a(7) & a(7) & a(7 downto 2);
```

or rotate the elements to the right by two positions:

```
y <= a(1 downto 0) & a(7 downto 2);
```



Data Types and Operators

➤ **Data types in the IEEE numeric std package :**

- Two new data types (signed and unsigned) are defined.
- Both are array of elements with the std_logic data type but they can be interpreted as unsigned or signed number.
- In signed number data type: MSB represents the sign.
Ex: "1100" is interpreted as 12 if unsigned and -4 if signed.
- To use signed and unsigned data types:

```
library ieee ;  
use ieee. std_logic _1164.all ;  
use ieee. numeric_std.all ;
```

- Ex: signal x,y: signed(15 downto 0);

Digital Design using VHDL 17

Notes:

In addition to logical operations, digital hardware frequently involves arithmetic operation as well. If we examine VHDL and the std_logic_1164 package, the arithmetic operations are defined only over the integer data type. To perform addition of signals as an example, we must use the integer data type.

Ex: signal a, b, sum: integer;

.....

sum <= a + b;

It is difficult to realize this statement in hardware since the code doesn't indicate the range (number of bits) of the a and b signals. Although this does not matter for simulation, it is important for synthesis since there is a huge difference between the hardware complexity of an 8-bit adder and that of a 32-bit adder.



A better alternative is to use an array of 0's and 1's and interpret it as an unsigned or signed number. We can define the width of the input and the size of the adder precisely, and thus have better control over the underlying hardware. The IEEE numeric_std package was developed for this purpose. The IEEE numeric_std package is apart of IEEE standard 1176.3. Two new data types, signed and unsigned, are defined in the package. Both data types are an array of elements with the std_logic data type. For the unsigned data type, the array is interpreted as an unsigned binary number, with the leftmost element as the MSB of the binary number. For the signed data type, the array is interpreted as a signed binary number in 2's-complement format. The leftmost element is the MSB of the binary number, which represents the sign of the number.

Since the signed and unsigned data types are arrays, their declarations are similar to that of the std_logic_vector data type.

Ex: signal x,y: signed(15 downto 0);

To use the signed and unsigned data types, we must include the library statement before the entity declaration. Furthermore, we must include the std_logic_1164 package since the std_logic data type is used in the numeric_std package. These statements are:

```
library ieee ;  
use ieee.std_logic_1164.all ;  
use ieee .numeric_std.all ;
```

Data Types and Operators

➤ **Overloaded operators in the IEEE numeric std package :**

Overloaded operator	Description	Data type Of operand a	Data type Of operand b	Data type Of result
abs a - a	absolute value negation	signed		signed
a * b a / b a mod b a rem b a + b a - b	arithmetic operation	Unsigned Unsigned, natural signed Signed, integer	Unsigned, natural Unsigned Signed, integer signed	Unsigned Unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	Unsigned Unsigned, natural signed Signed, integer	Unsigned, natural Unsigned Signed, integer signed	Boolean Boolean Boolean Boolean

Digital Design using VHDL 18

Notes:

Since the goal of the numeric_std package is to support the arithmetic operations, the relevant arithmetic operators, which include abs, *, /, mod, rem, + and -, are overloaded. These operators can now take two operands, with data types unsigned and unsigned, unsigned and natural, signed and signed as well as signed and integer. For example, the following are valid assignment statements:

signal a , b , c , d: unsigned(7 downto 0);

....

a <= b + c;

d <= b + 1;
e <= (5 + a + b) - c;

Data Types and Operators

➤ **New defined functions in the IEEE numeric std package :**

function	description	data type of operand a	data type of operand b	data type of result
shift_left (a,b) shift_right (a,b) rotate_left (a,b) rotate_right (a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize (a,b) std_match(a,b)	resize array compare '-'	unsigned, signed	natural	same as a
		unsigned, signed std_logic_vector std_logic	same as a	boolean
to_integer(a) to_unsigned(a,b) to_signed(a,b)	data type conversion	unsigned, signed		integer
		natural	natural	unsigned
		integer	natural	signed

Digital Design using VHDL 19

Notes:

The numeric_std package defines several new functions. These are new functions, not the overloaded VHDL operators. The new functions include:

- shift_left , shift_right , rotate_left, rotate_right: used for shifting and rotating operations.
- resize: used to convert an array to different sizes.
- std_match: used to compare objects with the '-' value.
- to_unsigned, to_signed, to_integer: used to do type conversion between the two new data types and the integer data type.



Data Types and Operators

➤ **Type Conversion:**

Data type of a	To data type	Conversion function /type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Digital Design using VHDL 20

Notes:

Conversion between two different data types can be done by a type conversion function or type casting. There are three type conversion functions in the numeric_std package: to_unsigned, to_signed, and to_integer.

The to_integer function takes an object with an unsigned or signed data type and converts it to the integer data type.

The to_unsigned and to_signed functions convert an integer into an object with the unsigned or signed data type of a specific number of bits. It takes two parameters. The first is the integer number to be converted, and the other specifies the desired number of bits (or size) in the new unsigned or signed data type.

The std_logic_vector, unsigned and signed data types are all defined as an array with elements of the std_logic data type. They are known as closely related data types in VHDL. Conversion between these types is done by a procedure known as type casting. To do type casting, we simply put the original object inside parentheses prefixed by the new data type.



Data Types and Operators

➤ **Type Conversion:**

Ex:

```
library ieee ;
use ieee. std_logic_1164. all ;
use ieee . numeric_std. all ;

...
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0 ) ;
...
u3 <= u2 + u1;
u4 <= u2 + 1;
u5 <= unsigned(sg);
u6 <= to_unsigned (5,4) ;
s3 <= std_logic_vector(u3);
s4 <= std_logic_vector(to_unsigned(5,4));
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));
s6 <= std_logic_vector(unsigned(s2) + 1);
```

Modeling in VHDL

- Behavioral Modeling
 - Using Sequential Statements
 - Using Concurrent statements (dataflow)
- Structural Modeling

Digital Design using VHDL 22



Concurrent Signal Assignment Statements

- Simple signal assignment statement
- Conditional Signal assignment statement
- Selected Signal assignment statement

Digital Design using VHDL 23



Simple Signal Assignment Statement

Syntax

- signal name <= projected_waveform;
- projected_waveform : consists of a new value for the signal and the time when the new value takes place.
Ex: `y <= a + b + 1 after10 ns;`
- For synthesis: timing information is not specified in the code.
`signal_name <= value_expression;`
Ex: `status <= '1';`
Ex: `even <= (p1 and p2) or (p3 and p4);`
Ex: `arith_out <= a + b + c - 1;`

Digital Design using VHDL 24

Notes:

A simple signal assignment statement is a conditional signal assignment statement without the condition expression and thus is a special case of a conditional signal assignment statement.

In VHDL definition, the simplified syntax of the simple signal assignment statement can be written as: `signal_name <= projected_waveform;`

The `projected_waveform` clause consists of two kinds of specifications: the expression of a new value for the signal and the time when the new value takes place. For example, consider the statement

`y <= a + b + 1 after 10 ns;`

which indicates that whenever the `a` or `b` signal changes, the expression `a+b+1` will be evaluated, and its result will be assigned to the `y` signal after 10 ns.

The time aspect of `projected_waveform` normally corresponds to the internal propagation delay to complete the computation of the expression. However, since the propagation delay depends on the components, device technology,



routing, fabrication process and operation environment, it is impossible to synthesize a circuit with an exact amount of delay. Therefore, for synthesis, explicit timing information is not specified in VHDL code. The default delta delay is used in the projected waveform. The syntax becomes:

```
signal_name <= value_expression;
```

The value_expression clause can be a constant value, logical operation, arithmetic operation and so on.

Simple Signal Assignment Statement

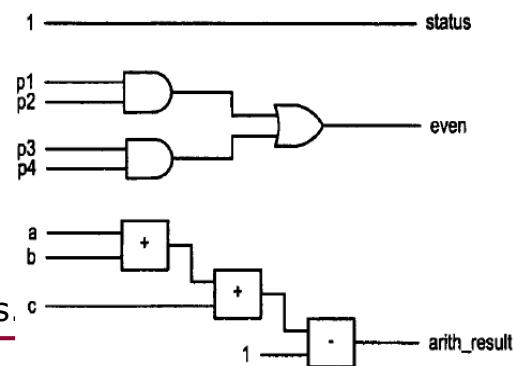
Conceptual Implementation

- The statement can be thought of as a circuit block. The output of the circuit is the signal in the left-hand side of the statement, and the inputs are all the signals that appear in the right-hand-side value expression. We then map each operator of the value expression into a smaller circuit block and connect their inputs and outputs accordingly.

Ex: For the previous three example:

```
status <= '1';
even <= (p1 and p2) or (p3 and p4);
arith_out <= a + b + c - 1;
```

- Note that these diagrams are only conceptual sketches. They will be transformed and simplified during synthesis.



Digital Design using VHDL 25



Conditional Signal Assignment Statement

Syntax

```
signal name <= value_expr_1 when boolean_expr_1 else  
                     value_expr_2 when boolean_expr_2 else  
                     value_expr_3 when boolean_expr_3 else  
                     . . .  
                     value_expr_n;
```

- The boolean_expr_i term is a Boolean expression that returns true or false.
- These Boolean expressions are evaluated successively in turn until one is found to be true, and the corresponding value expression is assigned to the output signal.

Digital Design using VHDL 26

Notes:

The boolean_expr_i ($i = 1, 2, 3, \dots, n$) term is a Boolean expression that returns true or false. These Boolean expressions are evaluated successively in turn until one is found to be true, and the corresponding value expression is assigned to the output signal. In other words, the first Boolean expression, boolean_expr_1, is checked first. If it is true, the first value expression, value_expr_1, will be assigned to the output signal. If it is false, the second Boolean expression, boolean_expr_2, will be checked next. This process continues until all Boolean expressions are checked. The last value expression, value_expr_n, will be assigned to the signal if none of the Boolean expressions is true.



Conditional Signal Assignment Statement

Example 1: 8-bit 4-to-1 MUX

```
library ieee ;
use ieee. std_logic_1164. all ;
entity mux4 is
port (
    a,b,c,d: in std_logic_vector(7 downto 0);
    s: in std_logic_vector (1 downto 0) ;
    x: out std_logic_vector(7 downto 0)
);
end mux4;
architecture cond_arch of mux4 is
begin
    x <= a when ( s = "00") else
        b when ( s = "01" ) else
        c when ( s = "10" ) else
        d;      --error if use d when (s="11");
end cond_arch ;
```

Input	Output
s	x
0 0	a
0 1	b
1 0	c
1 1	d

Digital Design using VHDL 27

Notes:

The first two lines are used to invoke the IEEE std_logic_1164 package so that the std_logic data type can be used. The next part is the entity declaration, which specifies the input and output ports of this circuit. The input ports include a, b, c and d, which are four 8-bit input data, and s, which is the 2-bit control signal. The output port is the 8-bit x signal. The architecture part uses one conditional signal assignment statement. The boolean condition s="00" is evaluated first. If it is true, the first value expression, a, is assigned to x. If it is false, the next boolean condition, s="01", will be evaluated. If it is true, b is assigned to x or the next Boolean expression, s="10", will be evaluated. If all three Boolean expressions are false, the last value expression, d, is assigned to x.

There is an issue about the use of the std_logic data type. At first glance, it seems that s is implied to be "11" when the first three Boolean expressions are



false, and thus d is assigned to x. However, there are nine possible values in std_logic data type and, for the 2-bit s signal, there are 81 (i.e., 9*9) possible combinations, including the expected "00", "01", "10" and "11" as well as the combinations, such as "0Z", "UX", "0-" and so on. Therefore, d is assigned to x for the "11" condition, as well as other 77 combinations. However, these 77 combinations can exist only in simulation. In a real circuit, comparisons as in s="0Z", cannot be implemented, and sometimes is meaningless, as in s="UX". In general, except for the limited use of 'Z' , the other 6 values of the std_logic data type will be ignored by synthesis software, and thus the final circuit will be synthesized as we originally expected.

Conditional Signal Assignment Statement

Example 2: 2-to-4 Binary Decoder

```
library ieee ;
use ieee. std_logic_1164. all ;
entity decoder4 is
port (
    s: in std_logic_vector (1 downto 0) ;
    x: out std_logic_vector (3 downto 0)
);
end decoder4;
architecture cond_arch of decoder4 is
begin
    x <= "0001" when ( s = "00") else
    "0010" when ( s = "01" ) else
    "0100" when ( s = "10" ) else
    "1000";
end cond_arch ;
```

Input	Output
s	x
0 0	0001
0 1	0010
1 0	0100
1 1	1000

Digital Design using VHDL 28

Conditional Signal Assignment Statement

Example 3: 2-to-4 priority encoder

```

library ieee ;
use ieee. std_logic_1164. all ;
entity prio_encoder42 is
port (
    r: in std_logic_vector (3 downto 0) ;
    code: out std_logic_vector (1 downto 0) ;
    active: out std_logic
);
end prio_encoder42;
architecture cond_arch of prio_encoder42 is
begin
    code <= "11" when ( r(3) = '1' ) else
        "10" when ( r(2) = '1' ) else
        "01" when ( r(1) = '1' ) else
        "00";
    active <= r(3) or r(2) or r(1) or r(0);
end cond_arch ;

```

Input	Output	
r	code	active
1 ---	11	1
0 1 --	10	1
0 0 1 -	01	1
0 0 0 1	00	1
0 0 0 0	00	0

Digital Design using VHDL 29

Notes:

A priority encoder checks the input requests and generates the code of the request with highest priority. The function table of a 4-to-2 priority encoder is shown in the above table. There are four input requests, r(3), r(2), r(1) and r(0). The outputs include a 2-bit signal, code, which is the binary code of the highest-priority request, and a 1-bit signal, active, which indicates whether there is an active request. The r(3) request has the highest priority. When it is asserted, the other three requests are ignored and the code signal becomes "11". If r(3) is not asserted, the second highest request, r(2), is examined. If it is asserted, the code signal becomes "10". The process repeats until all the requests are checked. The code signal returns "00" when only r(0) is asserted or no request is asserted. The active signal can be used to distinguish these two conditions.

Since operation of the priority encoder is similar to the definition of the conditional signal assignment statement, it is a good way to code this type of circuit.



Conditional Signal Assignment Statement

Example 4: Simple ALU

```

library ieee ;
use ieee. std_logic_1164. all ;
use ieee. numeric_std. all ;
entity simple_alu is
port (
    ctrl: in std_logic_vector (2 downto 0) ;
    src0,src1: in std_logic_vector (7 downto 0) ;
    result: out std_logic_vector (7 downto 0)
);
end simple_alu;
architecture cond_arch of simple_alu is
signal sum, diff , inc : std_logic_vector (7 downto 0) ;
begin
    inc <= std_logic_vector (signed(src0)+1) ;
    sum <= std_logic_vector(signed(src0)+signed(src1));
    diff <= std_logic_vector (signed(src0)-signed(src1)) ;

```

Input	Output
ctrl	result
0 --	src0 + 1
1 0 0	src0 + src1
1 0 1	src0 - src1
1 1 0	src0 and src1
1 1 1	src0 or src1

Digital Design using VHDL 30

Conditional Signal Assignment Statement

Example 4: Simple ALU (cont.)

```

result <= inc when ( ctrl(2) = '0' ) else
    sum when ( ctrl(1 downto 0) = "00" ) else
    diff when ( ctrl(1 downto 0) = "01" ) else
    src0 and src1 when ( ctrl(1 downto 0) = "00" ) else
    src0 or src1;
end cond_arch ;

```

Input	Output
ctrl	result
0 --	src0 + 1
1 0 0	src0 + src1
1 0 1	src0 - src1
1 1 0	src0 and src1
1 1 1	src0 or src1

Digital Design using VHDL 31



Notes:

An ALU performs a set of arithmetic and logical operations. The function table of a simple ALU is shown in the table above. The inputs include two 8-bit data sources, `src0` and `src1`, and a control signal, `ctrl`, which specifies the function to be performed. The output is the 8-bit result signal, which is the computed result. There are five functions, including three arithmetic operations, which are incrementing, addition and subtraction, and two logical operations, which are bitwise *and* and *or* operations.

For this circuit, the input data are interpreted as a collection of bits for the logical operation and as a signed number for the arithmetic operation. To achieve better portability, we normally use the `std_logic_vector` data type in the port declaration and then convert it to the desired data type in architecture body.

The IEEE `numeric_std` package and its `signed` data type are used to facilitate the arithmetic operation. When an addition, subtraction or incrementing operation is specified, we first convert the input to the `signed` data type, perform the operation and then convert the result back to the `std_logic_vector` data type. To make the code clear, we introduce three separate simple signal assignment statements and the `sum`, `diff`, and `inc` signals for the intermediate results of arithmetic operations.

Conditional Signal Assignment Statement

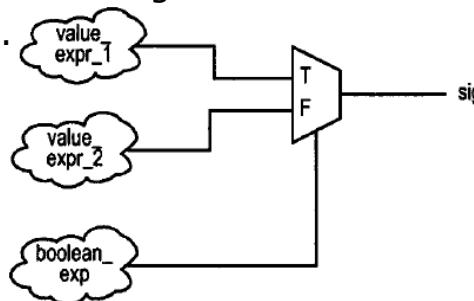
Conceptual Implementation

- For a simple conditional signal assignment with one when clause:

```
sig <= value_expr_1 when boolean_expr_1 else
    value_expr_2;
```

If the boolean_exp_1 is true, value_expr_1 is assigned to sig. If it is false, value_exp_2 is assigned to sig.

⇒ This is like a 2-1 MUX.



Digital Design using VHDL 32

Notes:

Constructing the conditional signal assignment statement requires three groups of hardware:

- Value expression circuits
- Boolean expression circuits
- Priority routing network

Value expression circuits realize the value expressions, value_expr_1, . . . , value_expr_n, and one of the results is routed to the output. Boolean expression circuits realize the Boolean expressions, boolean_expr_1, . . . , boolean_expr_n, and their values are used to control the priority routing network. The priority routing network is the structure that routes and controls the desired value to the output signal.

A priority network can be implemented by a sequence of 2-to-1 multiplexers. There will be two input ports, designated as T (for true) and F (for false)

respectively. If the selection signal has a value of true, the data from the T port will be routed to output. On the other hand, if the selection signal has a value of false, the data from the F port will be selected.

Conditional Signal Assignment Statement

Conceptual Implementation

- For a general conditional signal assignment with multiple when clauses:

```
sig <= value_expr_1 when boolean_expr_1 else
    value_expr_2 when boolean_expr_2 else
    ...
    value_expr_n;
```

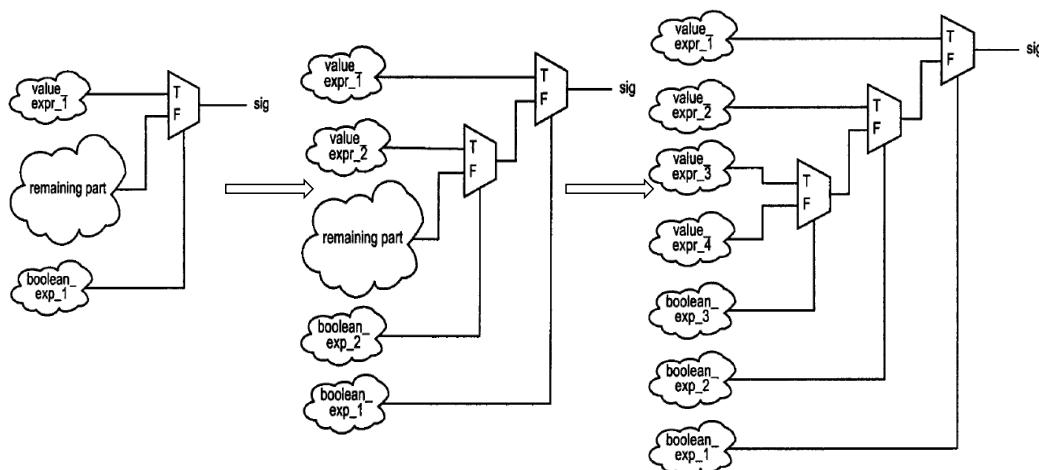
-We can perform the previous operation repetitively .

-ex: sig <= value_expr_1 when boolean_expr_1 else
 value_expr_2 when boolean_expr_2 else
 value_expr_3 when boolean_expr_3 else
 value_expr_4;

Digital Design using VHDL 33

Conditional Signal Assignment Statement

Conceptual Implementation



Digital Design using VHDL 34



Notes:

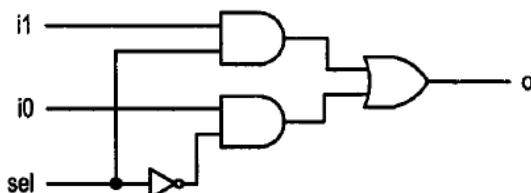
First we construct the first when clause, which corresponds to the highest_priority condition. If the result of boolean_expr_1 is true, the result of the corresponding value expression, value_expr_1, is routed to output. On the other hand, if the result of boolean_expr_1 is false , the result from the remaining part of the statement, which is shown as a single cloud, will be used. This cloud can be constructed using a multiplexer similar to the first when clause, with its output connected to the F port of the rightmost multiplexer. After repeating this process one more time, we construct the third when clause and complete the conceptual implementation.

The construction process can be applied repeatedly to any number of when clauses. Since each clause will introduce one extra stage of multiplexer network, the depth of the network grows as the number of clauses increases.

Conditional Signal Assignment Statement

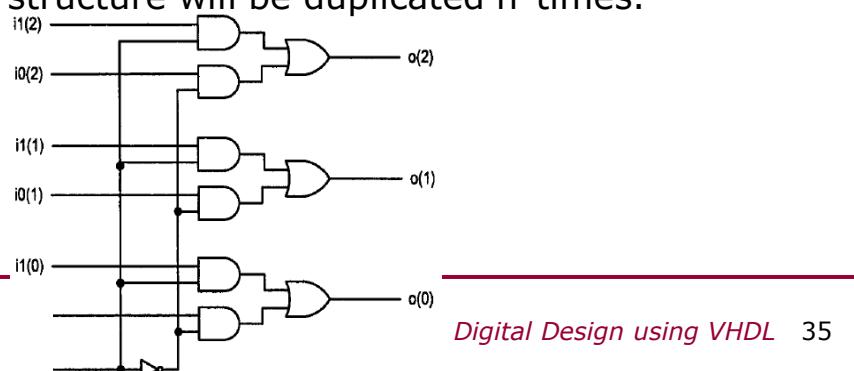
Gate Level Implementation

- Implementation of the 1-bit 2-1 MUX:



- For a n-bit 2-1 MUX: the control signal will remain the same while the gating structure will be duplicated n-times.

ex: 3-bit 2-1 MUX:

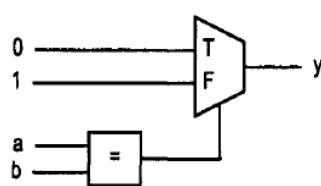


Conditional Signal Assignment Statement

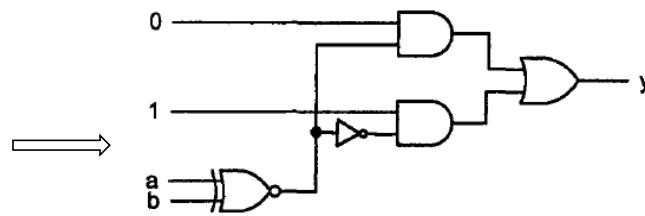
Implementation Example 1

Truth table of a 1-bit comparator.

...				
input	output			
a	b	a=b		
0	0	1		
0	1	0		
1	0	0		
1	1	1		



(a) Conceptual diagram



(b) Gate-level diagram

Digital Design using VHDL 36

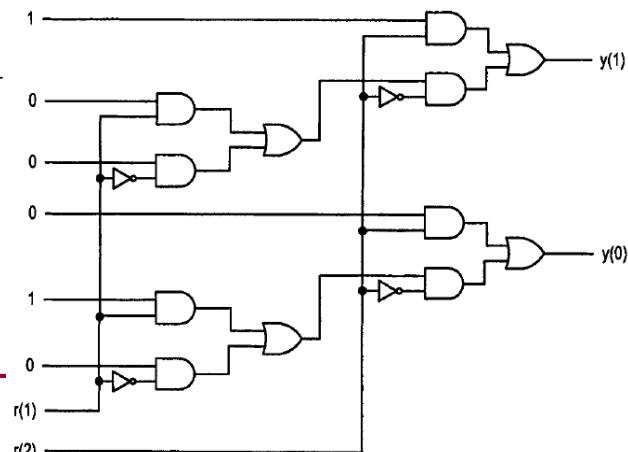
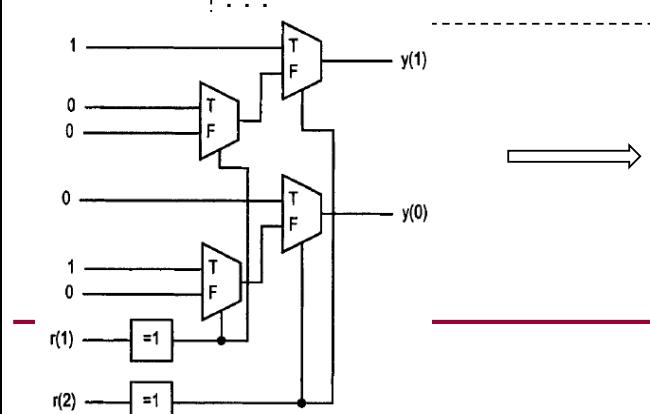
Notes:

For the gate level circuit, we can derive the logic expression of this circuit using boolean algebra. Reducing the expression using boolean algebra, we can find that this circuit can be simplified to a single xor gate.

Conditional Signal Assignment Statement

Implementation Example 2

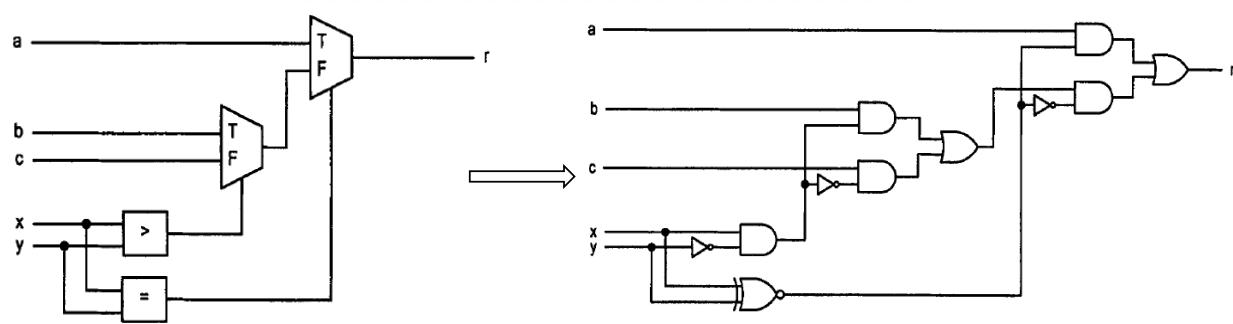
```
...
signal r: std_logic_logic (2 downto 1);
signal y: std_logic_logic (1 downto 0);
...
y <= "10" when r(2)='1' else
  "01" when r(1)='1' else
  "00" ;
...
```



Conditional Signal Assignment Statement

Implementation Example 3

```
...
signal a,b,c,x,y,r: std_logic;
...
r <= a when x=y else
      b when x>y else
      c;
```



Digital Design using VHDL 38



Selected Signal Assignment Statement

Syntax

```
with select_expression select
  signal_name <= value_expr_1 when choice_1,
                value_expr_2 when choice_2,
                value_expr_3 when choice_3,
                ...
                value_expr_n when choice_n;
```

- All possible values of select_expression must be covered by one and only one choice.
- The reserved word “others” can be used in the last choice (choice_n) to represent the unused values.

Digital Design using VHDL 39

Notes:

The selected signal assignment statement assigns an expression to a signal according to the value of select_expression. It is somewhat like a case statement in a traditional programming language. The select_expression term is used as the key for selection and it must result in a value of a discrete type or one-dimensional array. In other words, the evaluated result of select_expression can have only a finite number of possibilities. For example, a signal of the bit_vector(1 downto 0) data type can be used as select_expression since it contains only four possible values: “00”, “01”, “10” or “11”. A choice (i.e., choice_i) must be a valid value or a set of valid values of select_expression. The values of choices have to be mutually exclusive (i.e., no value can be used more than once) and all inclusive (i.e., all values have to be used). In other words, all possible values of select_expression must be covered by one and only one choice. The reserved word, others, can be used in the last choice (i.e., choice_n) to represent all the previously unused values.



Selected Signal Assignment Statement

Example 1: 8-bit 4-to-1 MUX

```

library ieee ;
use ieee. std_logic_1164. all ;
entity mux4 is
port (
    a,b,c,d: in std_logic_vector(7 downto 0);
    s: in std_logic_vector (1 downto 0) ;
    x: out std_logic_vector(7 downto 0)
);
end mux4;
architecture sel_arch of mux4 is
begin
with s select
    x <= a when "00",
    b when "01",
    c when "10",
    d when others; --syntax error if write d when "11";
end sel_arch ;

```

Input	Output
s	x
0 0	a
0 1	b
1 0	c
1 1	d

Digital Design using VHDL 40

Notes:

We need to take care of the possible values of the std_logic and std_logic_vector data types when they are used for select_expression. Recall that there are nine possible values in std_logic data type and there are 81 (i.e., 9*9) possible combinations for the 2-bit s signal, including the expected "00", "01", "10" and "11" as well as 77 other combinations, such as "ZZ", "UX" and "0-", which are not meaningful in synthesis and will be ignored accordingly. In the code, the when others clause covers the "11" choice as well as the other 77 combinations. We cannot simply list the last choice as "11". This causes a syntax error since only 4 of 81 values are covered, and thus the choices are not all-inclusive.

Selected Signal Assignment Statement

Example 2: 2-to-4 Binary Decoder

```
library ieee ;
use ieee. std_logic_1164. all ;
entity decoder4 is
port (
    s: in std_logic_vector (1 downto 0) ;
    x: out std_logic_vector (3 downto 0)
);
end decoder4;
architecture sel_arch of decoder4 is
begin
with s select
    x <= "0001" when "00",
    "0010" when "01",
    "0100" when "10",
    "1000" when others;
end sel_arch ;
```

Input	Output
s	x
0 0	0001
0 1	0010
1 0	0100
1 1	1000

Digital Design using VHDL 41

Selected Signal Assignment Statement

Example 3: 2-to-4 priority encoder

```

library ieee ;
use ieee. std_logic_1164. all ;
entity prio_encoder42 is
port (
    r: in std_logic_vector (3 downto 0) ;
    code: out std_logic_vector (1 downto 0) ;
    active: out std_logic
);
end prio_encoder42;
architecture sel_arch of prio_encoder42 is
begin
with r select
    code <= "11" when "1000"|"1001"|"1010"|"1011"|
                    "1100"|"1101"|"1110"|"1111" ,
    "10" when "0100"|"0101"|"0110"|"0111" ,
    "01" when "0010"|"0011",
    "00" when others;
Active <= r(3) or r(2) or r(1) or r(0);
end sel_arch ;

```

Input r	Output code	active
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

Digital Design using VHDL 42

Notes:

Recall that "11" will be assigned to code if r (3) is '1'. This consists of eight possible input combinations of the r signal, which are "1000", "1001", "1010", . . . , "1111". All of them are listed in the first choice. Note that the symbol | is used for specifying multiple values.

Intuitively, we may wish to use the '-' (don't-care) value of the std_logic data type to make the code compact:

```

with r select
    code <= "11" when "1---",
    "10" when "01--",
    "01" when "001-",
    "00" when others ;

```

While this is syntactically correct, the code does not describe the intended circuit. In VHDL, the '-' value is treated just as an ordinary value of std_logic.

Since the ‘ - ’ value will never occur in the physical circuit, the “1---”, “01—” and “001-” choices will never be met and the code is the same as: code <= “00”;



Selected Signal Assignment Statement

Example 4: Simple ALU

```

library ieee ;
use ieee. std_logic_1164. all ;
use ieee. numeric_std. all ;
entity simple_alu is
port (
    ctrl: in std_logic_vector (2 downto 0) ;
    src0,src1: in std_logic_vector (7 downto 0) ;
    result: out std_logic_vector (7 downto 0)
);
end simple_alu;
architecture sel_arch of simple_alu is
signal sum, diff , inc : std_logic_vector (7 downto 0) ;
begin
    inc <= std_logic_vector (signed(src0)+1) ;
    sum <= std_logic_vector(signed(src0)+signed(src1));
    diff <= std_logic_vector (signed(src0)-signed(src1)) ;

```

Input	Output
ctrl	result
0--	src0 + 1
100	src0 + src1
101	src0 - src1
110	src0 and src1
111	src0 or src1

Digital Design using VHDL 43

Selected Signal Assignment Statement

Example 4: Simple ALU (cont.)

```

with ctrl select
result <= inc when "000"|"001"|"010"|"011",
      sum when "100",
      diff when "101",
      src0 and src1 when "110",
      src0 or src1 when others ;
end sel_arch ;

```

Input	Output
ctrl	result
0--	src0 + 1
100	src0 + src1
101	src0 - src1
110	src0 and src1
111	src0 or src1

Digital Design using VHDL 44

Selected Signal Assignment Statement

Example 5: Truth Table Implementation

```
library ieee ;  
use ieee. std_logic_1164. all ;  
Entity truth_table is  
port (  
    a,b: in std_logic;  
    y: out std_logic  
);  
end truth_table;  
architecture arch of truth_table is  
    signal tmp: std_logic_vector (1 downto 0) ;  
begin  
    tmp <= a & b ;  
    with tmp select  
        y <= '0' when "00" ,  
            '1' when "01" ,  
            '1' when "10" ,  
            '1' when others;  
End arch;
```

Input	Output
a b	y
00	0
01	1
10	1
11	1

Digital Design using VHDL 45

Notes:

A truth table can be used to specify any combinational function. It is a simple and useful way to describe a small, random combinational circuit.

Because the choices list all the possible combinations, the selected signal assignment statement is a natural match for the truth table description.

Selected Signal Assignment Statement

Conceptual Implementation

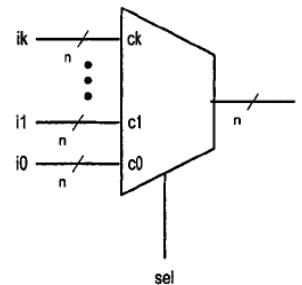
- For a selected signal assignment :

with select_expression select

```
signal_name <= value_expr_1 when choice_1,
               value_expr_2 when choice_2,
               value_expr_3 when choice_3,
               ...
               value_expr_k when choice_k;
```

⇒ This is like a k-1 MUX.

sel: selection_expression
MUX i/p: value_expr



Digital Design using VHDL 46

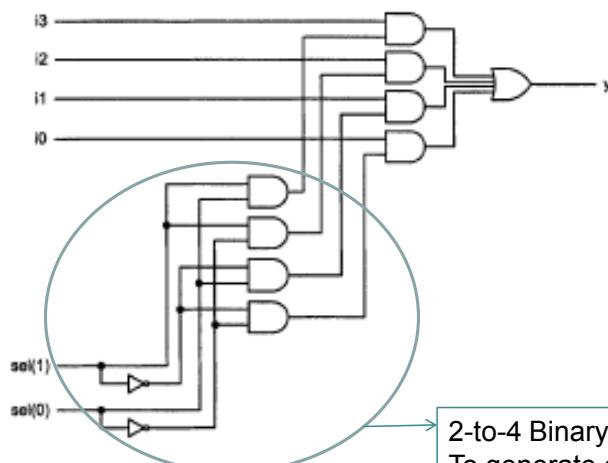
Notes:

Conceptually, the selected signal assignment statement can be thought as an abstract multiplexing circuit that utilizes a selection signal to route the result of the designated expression to output. In this multiplexing circuit, each possible value of select_expression has a designated input port in the multiplexer, and select_expression works as the selection signal of this multiplexer. Once its value is determined, the result of the designated value expression is passed to the output port of the multiplexer.

Selected Signal Assignment Statement

ex: 4-1 MUX:

Gate Level Implementation



2-to-4 Binary Decoder
To generate enable signal

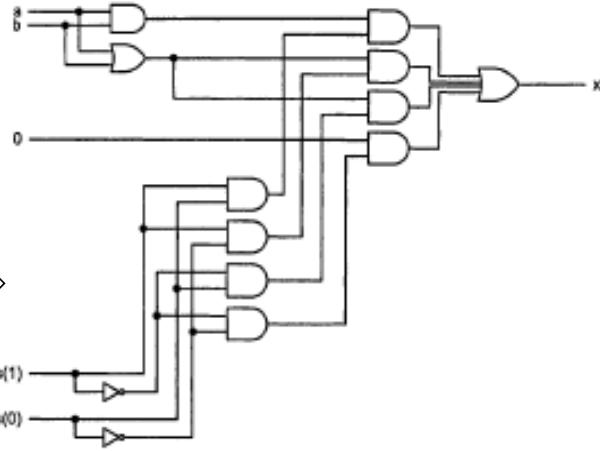
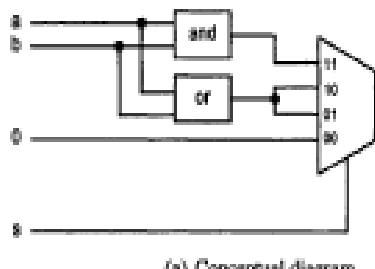
Digital Design using VHDL 47

Selected Signal Assignment Statement

Implementation Example

```
...
signal s: std_logic_vector(1 downto 0);
...
with s select
X <= (a and b) when "11",
(a or b)  when "01"|"10",
'0'        when others
...

```



Digital Design using VHDL 48

Notes:

This is a simple selected signal assignment statement. The selection expression has a data type of `std_logic_vector(1 downto 0)`. Although there are 81 possible values, only "00", "01", "10" and "11" are meaningful for synthesis. Thus, only a 4-to-1 multiplexer is needed.



Conditional & Selected Signal Assignment Statements

Conditional

- Good for circuits that need preferential treatment for certain conditions or to prioritize the operation.

Ex: Priority encoder.

- Less effective to describe a truth table.
- Can handle complicated conditions.

Ex: `pc_next <=
 pc_reg + offset when (state=jump and a=b) else
 pc_reg + 1 when (state=skip and flag='1') e l s e`

Selected

- Good for circuit described by truth table or truth table-like function.

Ex: Decoder, MUX.

- Less effective when certain input conditions are given preferential treatment.



Structure Modeling

- The design is described in terms of smaller parts called components interconnected with each other.
- The structure description specifies what types of parts are used and how these parts are connected.
- Structural modeling is very good to describe complex digital systems, though a set of components in a hierarchical fashion.
- At the lowest hierarchy each component is described as a behavioral model.
- The component has to be declared (make known) and then can be instantiated (actually used) in the architecture body as needed.

Digital Design using VHDL 50

Notes:

A structural description of a system is expressed in terms of subsystems interconnected by signals. Each subsystem may in turn be composed of an interconnection of sub-subsystems, and so on, until we finally reach a level consisting of primitive components, described purely in terms of their behavior. Thus the top-level system can be thought of as having a hierarchical structure.



Structure Modeling

Component Declaration

- The component declaration consists of the component name and the interface (ports).

- Syntax:

```
component component_name
    port (port_signal_names: mode type;
          port_signal_names: mode type;
          :
          port_signal_names: mode type);
end component ;
```

- The information contained inside the declaration is similar to that of entity declaration of the component.

Digital Design using VHDL 51

Notes:

The component must be declared in the architecture declaration region. The information of the component declaration must be taken from the entity declaration of this component design.

The ports in a component declaration must usually match the ports in the entity declaration one-for-one. The component declaration defines the names, order, mode and types of the ports to be used when the component is instanced in the architecture body. Declaration of a component implies making a local copy of the corresponding design entity. A component is declared once within any architecture, but may be instanced any number of times.



Structure Modeling

Component Instantiation

- The component instantiation statement initiates an instance of the component already defined.

- Syntax:

- Positional Association:

```
instance_name : component_name
    port map (signal1, signal2,...signaln);
```

- Named Association:

```
instance_name : component_name
    port map (comp_port=>signal1, comp_port=> signal2,... comp_port=> signaln);
```

Digital Design using VHDL 52

Notes:

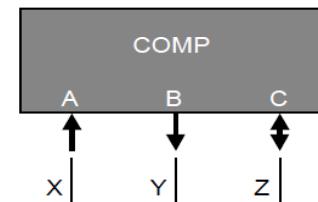
Signals in architecture are associated with ports on a component using a port map. In effect, a port map makes an electrical connection between “pieces of wire” in architecture (which are signals) and pins on a component (which are the ports). The same signal may be associated with several ports - this is the way to define interconnections between components. Port maps can be written using either named association or positional association. With named association, the port names appear in the port map, whereas with positional association, it is the order of the signals in the port map which determines what is connected to what. In positional association, the signal position must be in the same order as the declared component’s ports.

Structure Modeling

Component Instantiation

Example:

```
component COMP
    port (A: in STD_LOGIC;
          B: buffer STD_LOGIC;
          C: inout STD_LOGIC);
    end component;
```



- Positional Association:

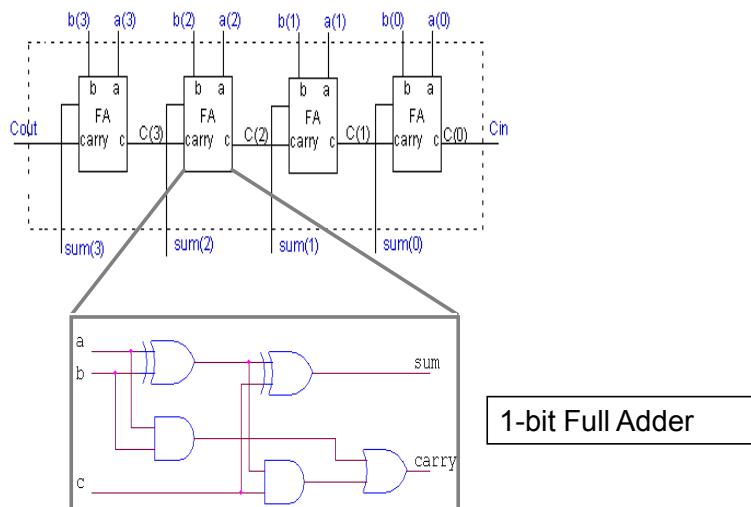
```
C1 : COMP port map ( X , Y , Z );
```

- Named Association:

```
C1 : COMP port map (A => X, B => Y, C => Z);
```

Structure Modeling

Example: 4-bit Full Adder



Digital Design using VHDL 54

Notes:

The 4-bit full adder can be constructed using four 1-bit full adders connecting them as shown in the above slide.

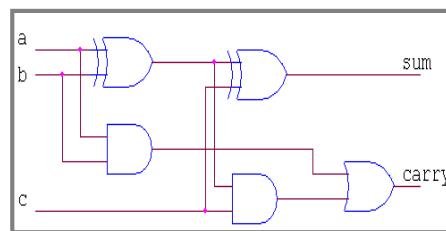
We will write a separate code for the 1-bit full adder to describe it behaviorally using concurrent statements and then we will declare it as a component in a new code and instantiate 4 times to build the 4-bit full adder.

Structure Modeling

Example: 4-bit Full Adder

➤ 1-bit Full Adder:

```
library ieee;
use ieee.std_logic_1164.all;
entity fulladder is
    port (a, b, c: in std_logic;
          sum, carry: out std_logic);
end fulladder;
architecture fulladder_behav of fulladder is
begin
    sum <= (a xor b) xor c;
    carry <= (a and b) or (c and (a xor b));
end fulladder_behav;
```



Digital Design using VHDL 55

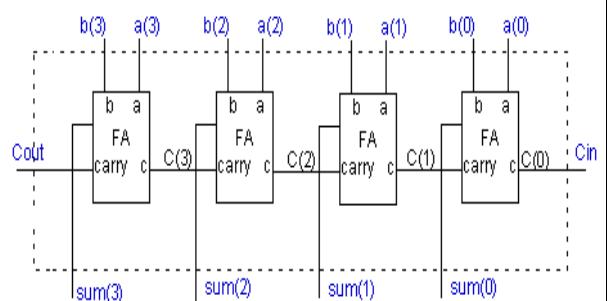
Structure Modeling

Example: 4-bit Full Adder

➤ 4-bit Full Adder:

```

library ieee;
use ieee.std_logic_1164.all;
entity FOURBITADD is
  port (a, b: in std_logic_vector(3 downto 0);
        Cin : in std_logic;
        sum: out std_logic_vector (3 downto 0);
        cout: out std_logic);
end FOURBITADD;
architecture STRUC of FOURBITADD is
  signal c: std_logic_vector (3 downto 1);
  component fulladder
    port (a, b, c: in std_logic;
          sum, carry: out std_logic);
  end component;
  
```



Structure Modeling

Example: 4-bit Full Adder

➤ 4-bit Full Adder:

begin

FA0: fulladder

port map (a(0), b(0), Cin, sum(0), c(1));

--OR: **port map** (a=>a(0), b=b>(0), c=>Cin, sum=>sum(0), carry=>c(1));

FA1: fulladder

port map (a(1), b(1), C(1), sum(1), c(2));

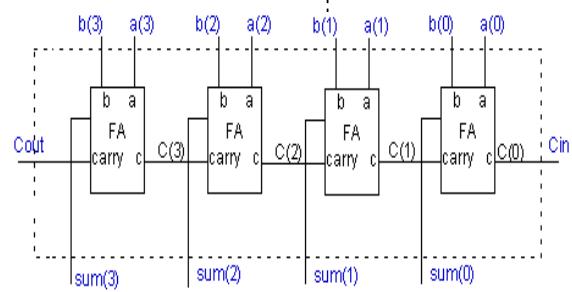
FA2: fulladder

port map (a(2), b(2), C(2), sum(2), c(3));

FA3: fulladder

port map (a(3), b(3), C(3), sum(3), cout);

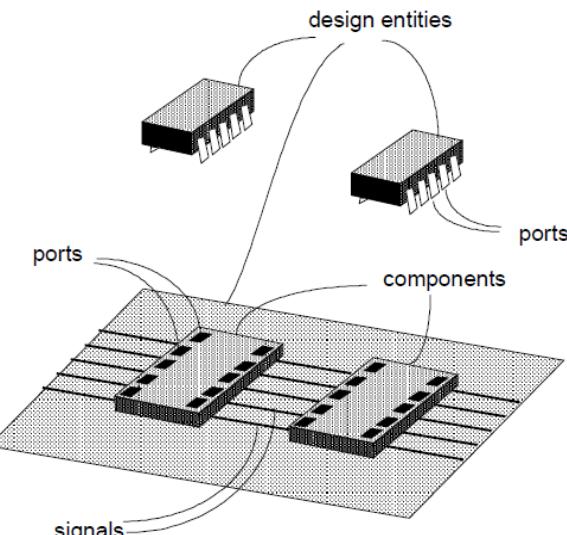
end STRUC;



Digital Design using VHDL 57

Structure Modeling

Chip Socket Analogy



Digital Design using VHDL 58

Notes:

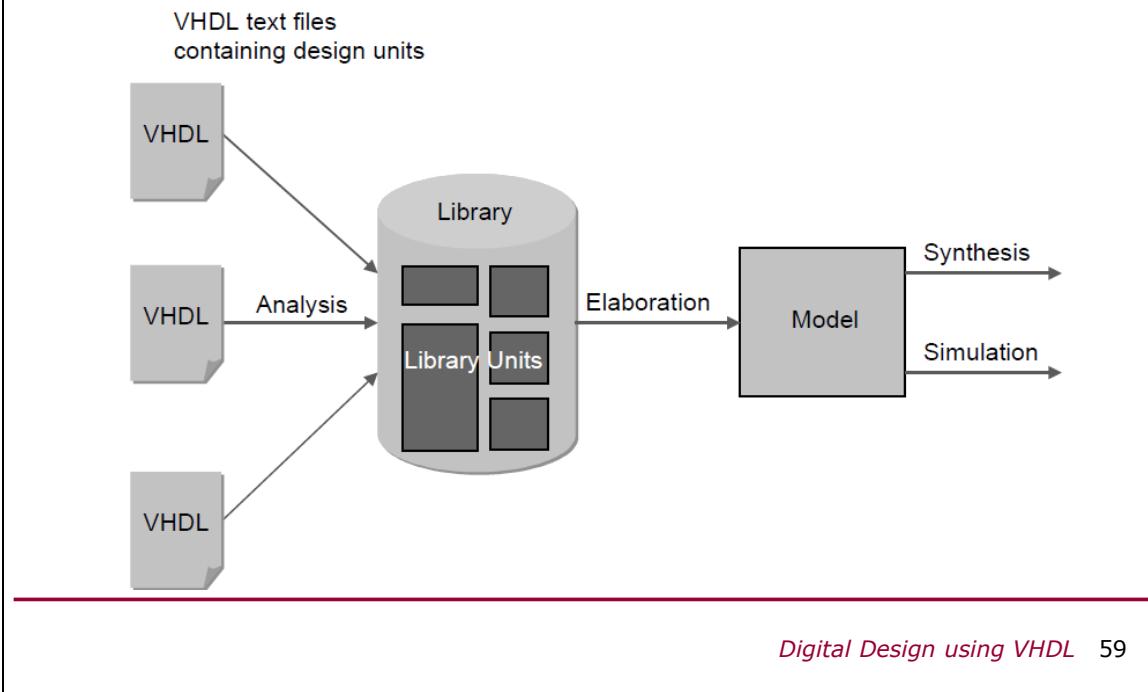
The diagram opposite illustrates the analogy between a VHDL component and a chip socket. A component corresponds to a chip socket; a design entity corresponds to a chip or a printed circuit board, a port in an entity declaration corresponds to a pin on an IC package, a port in a component declaration corresponds to a pin on a chip socket, and a signal corresponds to solder track on the printed circuit board.

Extending the analogy further, a VHDL configuration performs the task of plugging the chip into the socket, as we will see later on.

The purpose of components is to permit the design hierarchy to be coded and compiled top-down as well as bottom-up. A structural description making use of VHDL components is self-contained, and can be compiled separately before the lower level design entities have even been written. This means that you do not have to start at the bottom of the hierarchy and work upwards, connecting

together design entities written earlier. You can equally well start at the top and work down, and still compile each design entity as you write it. The use of components allows another level of flexibility, in that you can decide later on which specific design entity to use for each component instantiation. You will learn later that VHDL configurations can be used to name the entity and architecture to be used for each component instance.

Compilation



Notes:

Analysis

Units of VHDL text that can be processed by VHDL tools are called design units. A tool called an analyzer is responsible for checking the syntax of design units and putting them onto a library, where each design unit becomes a library unit. A VHDL library stores design units in a semi-compiled form, and can function as a database for tool integration. A single design can be spread over multiple design libraries.

Elaboration

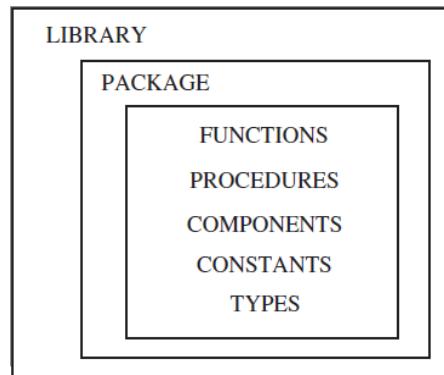
A set of design units is linked together into a design hierarchy in a process called elaboration. During the elaboration stage, the software starts from the designated top-level entity declaration and binds its architecture body according to the configuration specification. If there are instantiated components, the software replaces each instantiated component with the corresponding architecture body description. The process may repeat.



recursively until all instantiated components are replaced. The result of elaboration is the creation of a single model for the complete design hierarchy. Then the analyzed and elaborated description is usually fed to simulation or synthesis software. The former simulates and “runs” the description in a computer, and the latter realizes the description by physical circuits. Many tools implement elaboration as a process which occurs at the start of a simulation run, i.e. the “simulate” command causes the elaboration of the design hierarchy, followed by the execution of the simulation model.

Library

- A LIBRARY is a collection of commonly used pieces of code.
- The code is usually written in the form of functions, procedures, or components, which are placed inside packages, and then compiled into the destination library.
- Placing such pieces inside a library allows them to be reused or shared by other designs.



Digital Design using VHDL 60

Notes:

For a complex design, there may exist a large number of design units. It is desirable to organize these units and store them in separate places. Also, we may have a collection of commonly used design units that are shared by many different designs. It is more effective to save these units in a common library rather than duplicating them in every design directory.

To access the content of a library, we must first make it known by using a *library* statement.



Library Declaration

- Declaring a library is to make it visible to the design.

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

- At least three packages, from three different libraries, are usually needed in a design:
 - ieee.std_logic_1164 (from the ieee library)
 - standard (from the std library)
 - work (work library)
- Libraries std and work are made visible by default.

Digital Design using VHDL 61

Notes:

To declare a library (that is, to make it visible to the design) two lines of code are needed, one containing the name of the library, and the other a use clause. At least three packages, from three different libraries, are usually needed in a design:

- ieee.std_logic_1164 (from the ieee library)
- standard (from the std library)
- work (work library)

Their declarations are as follows:

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
LIBRARY std;  
USE std.standard.all;  
LIBRARY work;  
USE work.all;
```



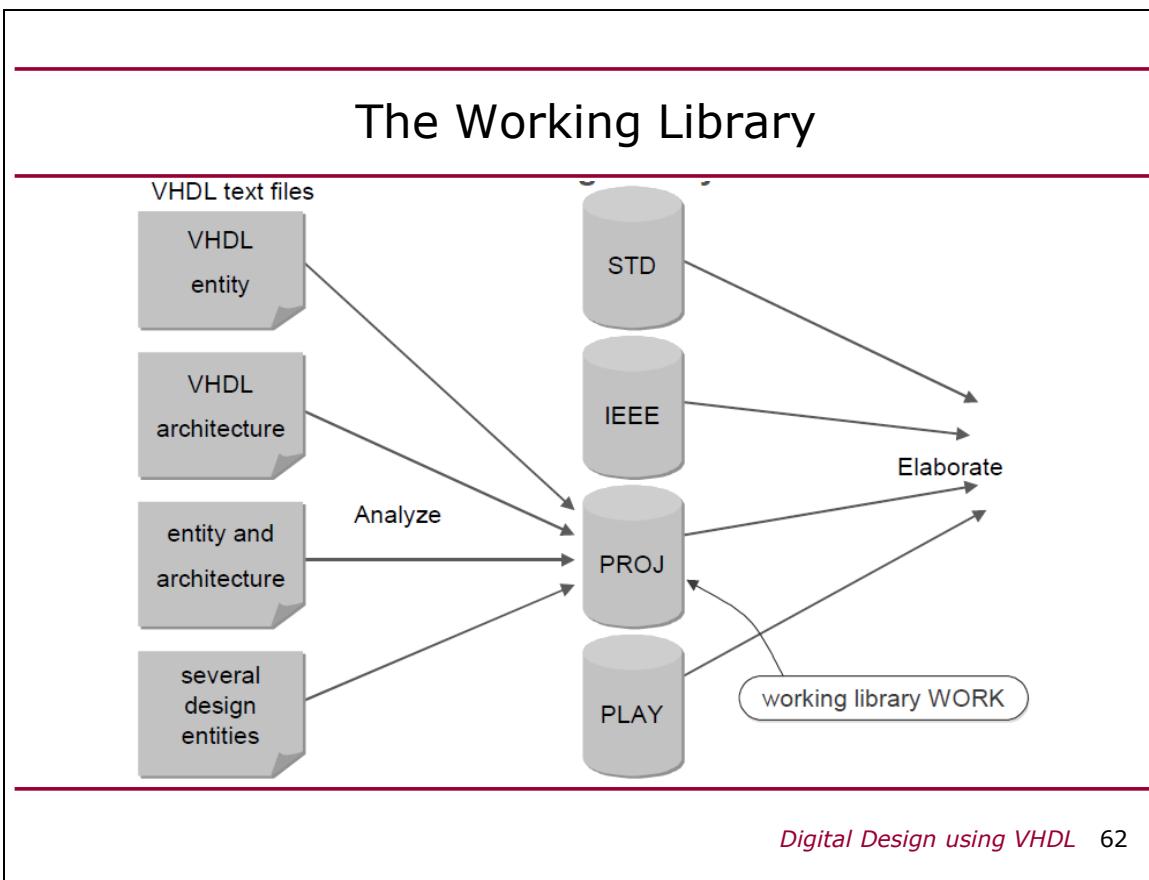
The libraries std and work shown above are made visible by default, so there is no need to declare them; only the ieee library must be explicitly written.

However, the latter is only necessary when the STD_LOGIC data type is employed in the design.

The purpose of the three packages/libraries mentioned above is the following: the std_logic_1164 package of the ieee library specifies a multi-level logic system; std is a resource library (data types, text i/o, etc.) for the VHDL design environment; and the work library is where we save our design (the .vhd file, plus all files created by the compiler, simulator, etc.).

➤ STANDARD

The package STD.STANDARD is defined in the Language Reference Manual, and is built into every VHDL tool. The package STD.STANDARD provides the predefined environment for every design unit, and this includes a set of data types including Integer, Bit, Boolean, String and Time.



Notes:

Libraries are a means of organizing and controlling VHDL design data and resources across an engineering team.

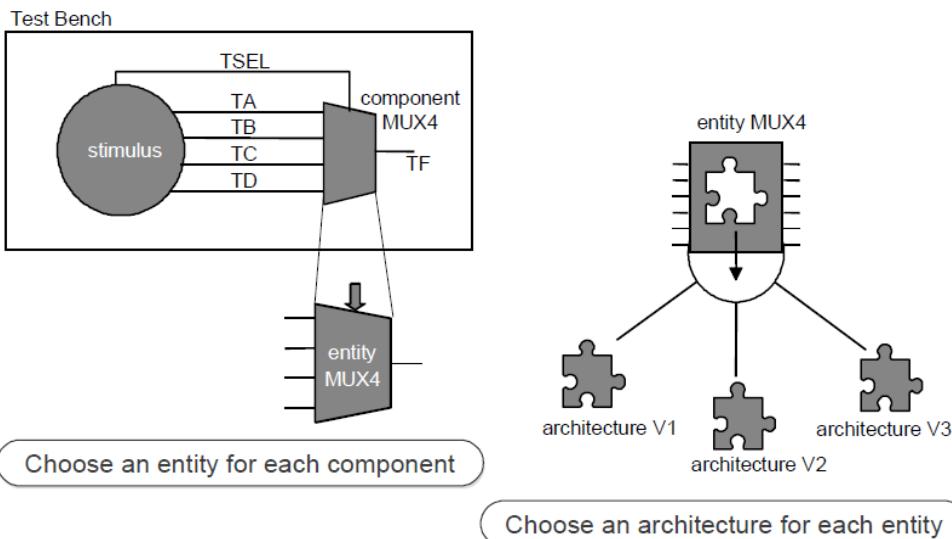
A VHDL design hierarchy can use design units (entities and packages) from many libraries - in fact any VHDL library accessible on the host computer's disk or network. These are termed resource libraries.

At any particular moment within a VHDL "session" only one library is designated as the working library. The VHDL compiler places design units into the working library. The working library is whichever VHDL library the VHDL analyzer is compiling the VHDL into. It is possible to reference the current working library from VHDL source text using the library name **WORK**. Note that there need never be a library named **WORK** - the name **WORK** points to whichever resource library happens to be the working library at the time the design unit is compiled.

Different VHDL tools use the name WORK in different ways. With some tools, you set the working library by defining the name WORK (as a logical library name). In other tools, the name WORK is defined automatically as a side effect of setting the working library some other way (e.g. through a GUI).

The VHDL compiler (or analyzer) does not care whether the design units are placed in separate text files or in the same text file. Although you may place several design units in one file and compile them in one go, it is usually good practice to place each design entity and each package in a separate file. This minimizes the amount of re-compilation that you need to do when you change a design unit.

Configuration



Digital Design using VHDL 63

Notes:

A VHDL description may consist of many design entities, each with several architectures, and organized into a design hierarchy. The configuration does the job of specifying the exact set of entities and architectures used in a particular simulation or synthesis run.

A configuration does two things. Firstly, a configuration specifies the design entity used in place of each component instance (i.e. it plugs the chip into the chip socket). Secondly, a configuration specifies the architecture to be used for each design entity.

There are two ways to specify the configuration. It can be described in an independent design unit, which is known as configuration declaration, or included in the declaration section of the architecture body, which is known as configuration specification(in which the relevant configuration is in the



declaration section of the architecture body). The IEEE RTL standard supports only the configuration declaration method.

Also VHDL 93 provides a much simpler mechanism. It allows a component to be bound directly to an entity and an architecture in component and no component declaration or any additional configuration construct is needed.



Configuration Declaration

- We create a new kind of design unit, known as configuration, to specify the binding of a component.
- Syntax:

```
configuration conf_name of entity_name is
    for architecture_name
        for instance_label : component_name
            use entity lib_name . bound_entity_name (bound_arch_name) ;
        end for ;
        for instance_label : component_name
            use entity lib_name . bound_entity_name (bound_arch_name) ;
        end for ;
        ...
    end for ;
end ;
```

- In the place of instance_label, we can use the all & others keywords.

Digital Design using VHDL 64

Notes:

In the configuration declaration method, we create a new kind of design unit, known as configuration, to specify the binding of a component. A configuration unit is an independent design unit in VHDL , just like an entity declaration and an architecture body. It is analyzed and stored independently when the VHDL code is processed.

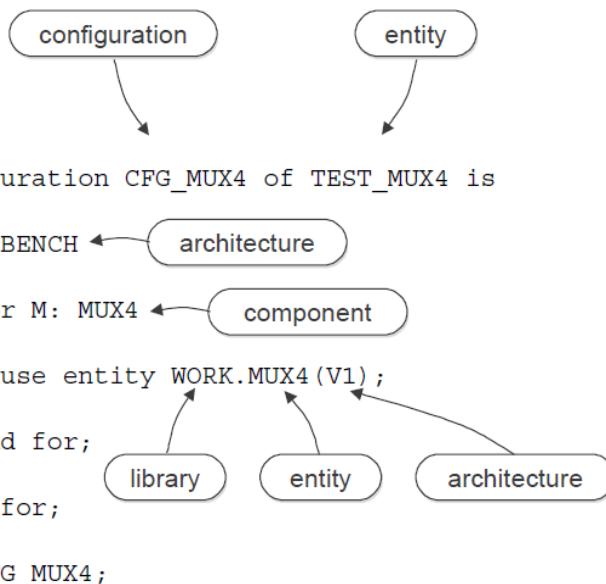
The conf_name term is the unique identifier for this configuration unit. The entity_name and architecture_name terms identify the entity and the architecture for which the configuration is intended. The instance_label term specifies a specific component instance, and the following “use . . .” clause indicates the entity and architecture to be bound to the instance. The lib_name term is the name of the library in which the entity and architecture reside.

In the place of instance_label, we can use the all keyword to represent all instances of this particular component, or use others in the end to represent all the unbound instances of the component.

Configuration Declaration

Example:

```
configuration CFG_MUX4 of TEST_MUX4 is
    for BENCH
        use entity WORK.MUX4 (V1);
    end for;
end CFG_MUX4;
```



Digital Design using VHDL 65

Notes:

The second line of the configuration (BENCH in this example) specify the name of an architecture belonging to the entity named on the first line of the configuration (TEST_MUX4 in this example). The third line of the configuration identifies one of the component instances within the architecture (M is an instance label in this example) and the forth line says which particular design entity is to be bound to that component instance.

If there is more than one component instantiation within an architecture, then the middle three lines can be repeated for each instance. Each for must be balanced with a corresponding "end for;", and the entire configuration declaration must end with "end;" or in this example with "end CFG_MUX4;" or in VHDL 93 with "end configuration;" or even "end configuration CFG_MUX4;".

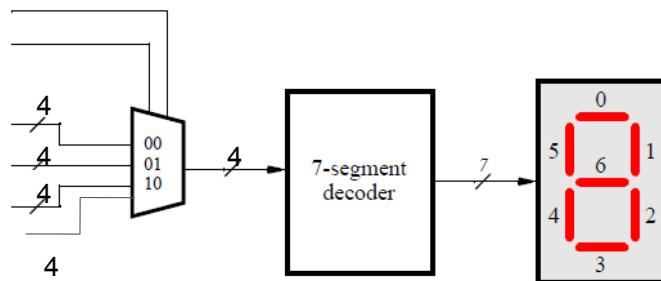


Default Configuration

- In the absence of any explicit configuration information, one or more component instances can be configured by default.
- The default rules are as follows:
 - The component is bound to an entity with the identical name.
 - Component ports are bound to entity ports of the same names.
 - If there is more than one architecture, then the most recently compiled (analyzed) architecture is bound to the entity declaration.
- The default binding should be satisfactory most of the time, and no explicit configuration statement is needed.

Design Example 1

- Design the following Multiplexed 7-segment decoder.

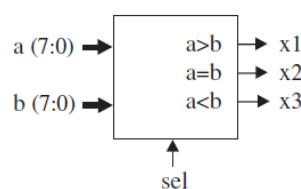


Digital Design using VHDL 71

Design Example 2

Comparator

Construct a circuit capable of comparing two 8-bit vectors, a and b . A selection pin (sel) should determine whether the comparison is signed ($sel = '1'$) or unsigned ($sel = '0'$). The circuit must have three outputs, x_1 , x_2 , and x_3 , corresponding to $a > b$, $a = b$, and $a < b$, respectively.



Digital Design using VHDL 72



Design Example 3

- Design a 2-to-4 decoder with an enable signal 'en'. When en is '1', the decoder functions as usual. When en is '0', the decoder is disabled and output becomes "0000".

Digital Design using VHDL 73

Design Example 4

- Consider a 2-by-2 switch. It has two input data ports, $x(0)$ and $x(1)$, and a 2-bit control signal, ctrl. The input data are routed to output ports $y(0)$ and $y(1)$ according to the ctrl signal. The function table is specified below.

Input	Output	Function
ctrl	y1	y0
00	x1	x0
01	x0	x1
10	x0	x0
11	x1	x1

Digital Design using VHDL 74



Session 3

SEQUENTIAL STATEMENTS IN VHDL

Digital Design using VHDL 2



Topics for this Lecture

- What are Sequential Statements?
- VHDL Process
- Types of process
- Signals and variables.
- Sequential statements: If statement, Case statement & for loop statement.

Digital Design using VHDL 3



Introduction

- Sequential statements are executed in sequence.
- They are more like that of a traditional programming language.
- The main purpose of sequential statements is to describe and model a circuit's behavior.
- Since VHDL is executed concurrently, sequential statements have to be enclosed inside a construct known as a **process**.
- Unlike concurrent signal assignment statements, there is no clear mapping between sequential statements and hardware components

Digital Design using VHDL 4

Notes:

As the name suggests, sequential statements are executed in sequence. The semantics of these statements is more like that of a traditional programming language. Since they are not compatible with the general concurrent execution model of VHDL, sequential statements have to be enclosed inside a construct known as a process. The main purpose of sequential statements is to describe and model a circuit's "abstract behavior." Unlike concurrent signal assignment statements, there is no clear mapping between sequential statements and hardware components. Some statements and coding styles are difficult or even impossible to synthesize. To use processes and sequential statements for synthesis, the VHDL description has to be coded in a disciplined way so that the code can be faithfully mapped into the intended hardware configuration.



VHDL Process

- It contains a set of sequential statements.
- The process itself is a concurrent statement.
- It can be interpreted as a black box whose behavior is described by the sequential statements.
- The order of statements inside a process is important since they are executed sequentially.
- A process and its internal sequential statements can be used to describe a combinational or sequential circuit.
- The process has two basic forms: process with a sensitivity list & process with wait statements.

Digital Design using VHDL 5

Notes:

A process is a VHDL construct that contains a set of actions to be executed sequentially. These actions are known as sequential statements. The process itself is a concurrent statement. It can be interpreted as a circuit part enclosed inside a black box whose behavior is described by the sequential statements. We may or may not be able to construct physical hardware that exhibits the desired behavior. The execution inside a process is sequential, and thus the order of the statements is important.

Note that we should not confuse sequential statements with sequential circuits. Sequential statements are VHDL statements inside a process, and sequential circuits are circuits with internal states. A process and its internal sequential statements can be used to describe a combinational or sequential circuit. We will focus, in this session, on combinational circuits.

The process has two basic forms. The first form has a sensitivity list but no wait statement inside the process. The second form has one or more wait statements but no sensitivity list.

Process with a Sensitivity List

➤ Syntax:

```
process(sensitivity_list)
  declarations ;
begin
  sequential statement ;
  sequential statement ;
  ...
end process;
```

- The sensitivity_list is a list of signals to which the process responds.
- The declarations part consists of various declarations that are local to the process.

Digital Design using VHDL 6

Notes:

Labeling:

Optionally, a process statement may be labeled. That's mean it can take any label we choose. The label is written before the "process" keyword (ex: first_process: process(sensitivity_list).....)



Process with a Sensitivity List

- A VHDL process is not called by another routine.
- It acts like a circuit part, which is either active (known as activated) or inactive (known as suspended).
- A VHDL process is activated when a signal in the sensitivity list changes its value.
- Once a process is activated, its statements will be executed sequentially until the end of the process.
- The process is then suspended until the next change of signal.

Digital Design using VHDL 7

Process with a Sensitivity List

- Example:

```
signal a,b,c,y : std_logic ; -- in architecture declaration
...
process ( a , b , c )
Begin
y <= a and b and c ;
end process;
```

- When any input (i.e., a, b or c) changes, the process is activated and its statement is executed.
- The statement evaluates the expression and assigns the result to the y signal.
- This process simply describes a three-input and circuit with a, b and c inputs and y output.

Digital Design using VHDL 8



Process with an incomplete Sensitivity List

- It means a sensitivity list has one or more input signals missing.
- Example:

```
signal a,b,c,y : std_logic ; -- in architecture declaration
...
process ( a )
Begin
y <= a and b and c ;
end process;
```

- For a combinational circuit, all input signals of a combinational circuit should be included in the sensitivity list.
- A process with incomplete sensitivity can be used to describe a circuit with internal memory.

Digital Design using VHDL 9

Notes:

When the 'a' signal changes, the process is activated and the circuit acts as expected. On the other hand, when the 'b' or 'c' signal changes, the process remains suspended and the 'y' signal keeps its previous value. This implies that the circuit has some sort of memory element that is triggered at both positive and negative edges of the 'a' signal. When the 'a' signal changes, the expression is evaluated and the result is stored in the memory element. This is not the circuit behavior we expected, and it cannot be synthesized by regular hardware components.

For a combinational circuit, the output is a function of input. This implies that the circuit responds to any input change. Thus, all input signals of a combinational circuit should be included in the sensitivity list. A process with incomplete sensitivity can be used to describe a circuit with internal memory and thus infer a memory element.



Process with a wait Statement

- A process with wait statements has one or more wait statements but no sensitivity list.
- Syntax:

The wait statement has several forms:

wait on signals ;

wait until boolean_expression;

wait for time_expression ;

Process with a wait Statement

➤ Example:

```
signal a,b,c,y : std_logic ; -- in architecture declaration
...
process
Begin
y <= a and b and c ;
wait on a, b , c ;
end process;
```

- The process starts automatically after the system initialization.
- It continues the execution until a wait statement is reached and then becomes suspended.
- “wait on a, b , c;” means that the process waits for a change in the a or b or c signal. When one of them changes value, the process is activated.

Digital Design using VHDL 11

Notes:

Note that there is no sensitivity list. The process starts automatically after the system initialization. It continues the execution until a wait statement is reached and then becomes suspended. The statement “wait on a, b , c” means that the process waits for a change in the a or b or c signal. When one of them changes value, the process is activated. It executes to the end of the process, then returns to the beginning of the process and continues execution. It becomes suspended again when reaching the wait statement. The overall effect of this process describes a three-input and gate.

The behavior of the two other types of wait statements is similar except that the process waits until a special Boolean condition is asserted or waits for a specific amount of time.

Since multiple wait statements are allowed, a process with wait statements can be used to model complex timing behavior and sequential events.



However, in synthesis, only few well-defined forms of wait statements can be used, and normally only one wait statement is allowed in a process.



Sequential Statements

- We will focus on the following sequential statements:
 - Sequential signal assignment statement
 - Variable assignment statement
 - If statement
 - case statement
 - for loop statement

Digital Design using VHDL 12

Notes:

Sequential statements include a rich variety of constructs, and they can exist only inside a process. Many sequential constructs don't have clear counterparts in hardware implementation, and are difficult, if not impossible, to synthesize.



Sequential Signal Assignment Statement

- The syntax of a sequential signal assignment is identical to that of the simple concurrent signal assignment of except that it is written inside a process.
- Syntax:
 - With a time delay: `signal_name <= projected_waveform;`
 - With a delta delay: `signal_name <= value_expression;`
- Any assignments to a signal takes effect only when the process suspends. Until this happens, all signals keep their previous values.
- Only last assignment to a signal listed inside the process is effective.

Digital Design using VHDL 13

Notes:

The `projected_waveform` clause consists of a value expression and a time expression, which is generally used to represent the propagation delay. As in the concurrent signal assignment statement, the delay specification cannot be synthesized and we always use the default delta delay.

The concurrent conditional and selected signal assignment statements cannot be used inside the process.

For a signal assignment with delta-delay, the behavior of a sequential signal assignment statement is somewhat different from that of the concurrent one. If a process has a sensitivity list, the execution of sequential statements is treated as a “single abstract evaluation,” and the actual value of an expression will not be assigned to a signal until the end of the process. This is consistent with the black box interpretation of the process; that is, the entire process is



treated as one indivisible circuit part, and the signal is assigned a value only after the completion of all sequential statements.

Inside a process, a signal can be assigned multiple times. If all assignments are with delta delays, only the last assignment takes effect. Because the signal is not updated until the end of the process, it never assumes any “intermediate” value.

Sequential Signal Assignment Statement

➤ Example:

```
Process (C,D)
Begin
A <= 2 ;
B <= A+C ;
A <= D+1;
E <=A*2;
end process;
```

Assume initial values are:
A =1
B=1
C=1
D=1
E=1

- If D is changed to 2, A will change to 3 and B & E to 2.

Digital Design using VHDL 14

Notes:

Signal D changes its value from 1 to 2. Since D is on the sensitivity list, the process is activated. The first statement is executed but the assignment is postponed till the process is suspended. As a result, signal A is still 1. Then the second statement is executed but the assignment is postponed till the process is suspended. B is still 1. The third statement and then the forth statement is executed but the assignment is postponed till the suspension of the process. Till now A, B, E are still 1.

Now the process is suspended and signal assignment will take place depending on the actual old values of the signal. Thus $A \leq 3$, $B \leq 2$ and $E \leq 2$. These is the true value not what was expected ($A=3$, $B=3$, $E=6$).

Sequential Signal Assignment Statement

➤ Example:

```
Process (a,b,c,d)
Begin
y <= a or c ;
y <= a and b ;
y <= c and d ;
end process;
```

```
Process (a,b,c,d)
Begin
y <= c and d ;
end process;
```

- The 2 previous codes are the same !

Digital Design using VHDL 15

Notes:

Although this segment is easy to understand, multiple assignments may introduce mistakes in a more complex code and make synthesis very difficult. Unless there is a compelling reason, it is a good idea to avoid assigning a signal multiple times. The only exception is the assignment of a default value in the if and case statements as we will see later.

The result will be very different if the multiple assignments are the concurrent signal assignment statements. Assume that the previous three assignment statements are concurrent signal assignment statements (i.e., not inside a process). The code segment becomes:

...

-- the statements are not inside a process

```
y <= a or c;
y <= a and b;
y <= c and d;
```



The code is syntactically correct since multiple assignments are allowed for a signal with the std_logic data type. Although the syntax is fine, the design is incorrect because of the potential output conflict. The y signal may get a value of 'X' in simulation if any two of the output values of the three gates are different.



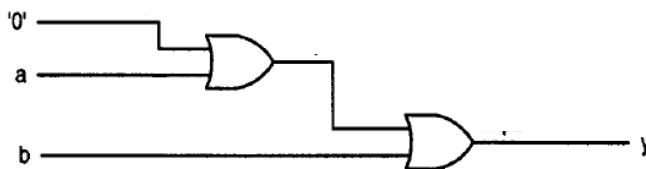
Variable Assignment Statement

- Syntax: `variable_name := value_expression;`
- The immediate assignment notion, `: =`, is used for the variable assignment.
- There is no time dimension (i.e., no propagation delay) and the assignment takes effect immediately.
- The behavior of the variable assignment is just like that of a regular variable assignment used in a traditional programming language.

Variable Assignment Statement

➤ Example:

```
signal a, b, y: std_logic;  
...  
process (a, b)  
    variable tmp : std_logic ;  
begin  
    tmp := '0';  
    tmp := tmp or a;  
    tmp := tmp or b;  
    y <= tmp;  
end process;
```



Digital Design using VHDL 17

Notes:

The tmp variable assumes the value immediately in each sequential statement and assigns its value, which is equal to a or b, to the y signal.

Note that the variables are “local” to the process and have to be declared inside the process.

Although the behavior of a variable is easy to understand, mapping it into hardware is difficult. Because of the lack of clear hardware mapping, we should try to use signals in code in general and resort to variables only for the characteristics that cannot be described by signals.



Variables & Signals

- A variable can only be declared and used within a single process - unlike signals.
- Variables cannot be used for inter-process communication.
- Variables cannot appear in a sensitivity list, so there is no way to force a process to wake up when a variable is updated.
- Signals are always assigned with delay, variables are assigned immediately.
- Signals can have a future waveform consisting of a series of events yet to occur on that signal, whereas variables have only a current value.
- Sometimes variables can be used in place of signals, sometimes they cannot.

Digital Design using VHDL 18

Notes:

Variables can only be declared inside a process, and can only be used inside the process in which they are declared.

Variables & Signals

```
signal a, b, y: std_logic;
...
process (a, b)
    variable tmp : std_logic ;
begin
    tmp := '0';
    tmp := tmp or a;
    tmp := tmp or b;
    y <= tmp;
end process;
```

```
signal a, b, y,tmp: std_logic;
...
process (a, b , tmp)
begin
    tmp <= '0';
    tmp <= tmp or a;
    tmp <= tmp or b;
    y <= tmp;
end process;
```

 the same as

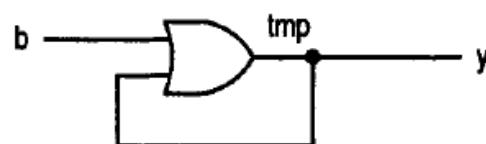
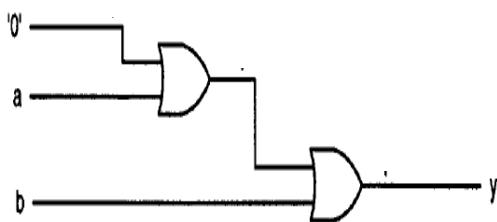
```
signal a, b, y,tmp: std_logic;
...
process (a, b , tmp)
begin
    tmp <= tmp or b;
    y <= tmp;
end process;
```

Digital Design using VHDL 19

Notes:

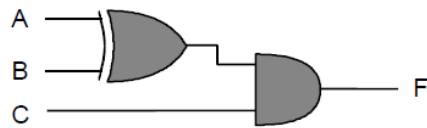
Note that the signals have to be “global” and declared outside the process, and the tmp signal has to be included in the sensitivity list.

Variables & Signals



Digital Design using VHDL 20

Variables & Signals



```
F <= (A xor B) and C;
```

```

process (A, B, C)
  variable Tmp: STD_LOGIC;
begin
  Tmp := A xor B;
  F <= Tmp and C;
end process;
  
```

```

signal Tmp: STD_LOGIC;
...
process (A, B)
begin
  Tmp <= A xor B;
end process;

process (Tmp, C)
begin
  F <= Tmp and C;
end process;
  
```

Digital Design using VHDL 21

Notes:

The three boxes opposite show equivalent fragments of code. The first is a signal assignment that uses bracketing to control the order of evaluation of an expression. The second uses a variable to store the intermediate result so must be coded up as a single process. The third uses a signal to store the intermediate result, so can be coded up as two processes.

If Statements

➤ Syntax:

```
if boolean_expr_1 then
    sequential_statements;
elsif boolean_expr_2 then
    sequential_statements ;
elsif boolean_expr_3 then
    sequential_statements;
...
else
    sequential_statements;
end if ;
```

- Branches elsif and else are optional.

Digital Design using VHDL 22

Notes:

An if statement has one then branch, one or more optional elsif branches and one optional else branch. The boolean_expr_i term is a boolean expression that returns true or false. These Boolean expressions are evaluated sequentially. When an expression is evaluated as true, the statements in the corresponding branch will be executed sequentially and the remaining branches will be skipped. If none of the expressions is true and the else branch exists, the statements in the else branch will be executed.

If Statements

Example 1 : 8 bit 4-1 MUX

```
....  
architecture if_arch of mux4 is  
begin  
    process (a ,b ,c , d , s )  
    begin  
        if ( s = "00") then  
            x <= a;  
        elsif ( s = "01") then  
            x <= b;  
        elsif ( s = "10") then  
            x <= c;  
        else  
            x <= d;  
        end if ;  
    end process;  
end if_arch;
```

Digital Design using VHDL 23

Notes:

Since the multiplexer is a combinational circuit, all input signals, including a, b, c, d and s, are in the sensitivity list. Note that the signals used in the boolean expressions are also the input signals.

If Statements

Example 2 : 2-to-4 Binary Decoder

```
....  
architecture if_arch of decoder4 is  
begin  
    process (s )  
    begin  
        if ( s = "00") then  
            x <= "0001";  
        elsif ( s = "01") then  
            x <= "0010";  
        elsif ( s = "10") then  
            x <= "0100";  
        else  
            x <= "1000";  
        end if ;  
    end process;  
end if _arch;
```

Digital Design using VHDL 24

If Statements

Example 3 : 4-to-2 Priority Encoder

```
....  
architecture if_arch of prio_encoder42 is  
begin  
    process ( r )  
    begin  
        if ( r(3) = '1' ) then  
            code <= "11";  
        elsif (r(2) = '1' ) then  
            code <= "10";  
        elsif (r(1) = '1' ) then  
            code <= "01";  
        else  
            code <= "00";  
        end if ;  
    end process;  
    active <= r(3) or r(2) or r(1) or r(0);  
end if _arch;
```

Digital Design using VHDL 25



If Statements

Example 4 : Simple ALU

```
....  
architecture if_arch of simple_alu is  
signal src0s, src1s: signed(7 downto 0);  
begin  
    src0s <= signed(src0);  
    src1s <= signed(src1);  
    process ( ctrl , src0 , src1, src0s , src1s)  
    begin  
        if ( ctrl(2) = '0' ) then  
            result <= std_logic_vector (src0s + 1) ;  
        elsif (ctrl(1 downto 0) = "00") then  
            result <= std_logic_vector (src0s + src1s) ;  
        elsif (ctrl(1 downto 0) = "01") then  
            result <= std_logic_vector(src0s - src1s);
```

Digital Design using VHDL 26

If Statements

Example 4 : Simple ALU

```
elsif (ctrl(1 downto 0) = "10") then  
    result <= src0 and src1 ;  
else  
    result <= src0 or src1 ;  
end if ;  
end process;  
end if_arch;
```

Digital Design using VHDL 27

If Statements

Comparison to a conditional signal assignment statement

- If the sequential statements inside an if statement consist of only the signal assignment of a single signal, the two statements are equivalent.

```
sig <= value_expr_1 when boolean_expr_1 else
Value_expr_2 when boolean_expr_2 else
Value_expr_3 when boolean_expr_3 else
...
Value_expr_n;
```

```
process ( . . . )
begin
if boolean_expr_1 then
sig <= value_expr_1 ;
elsif boolean_expr_2 then
sig <= value_expr_2;
elsif boolean_expr_3 then
sig <= value_expr_3 ;
.
.
else
sig <= value_expr_n;
end if ;
end process ;
```

Digital Design using VHDL 28



If Statements

Comparison to a conditional signal assignment statement

- An if statement is much more general since a branch of the if statement can be a sequence of sequential statements.
- Since an if statement is a sequential statement, it can be nested in a branch of another if statement.

Ex: Determining the maximum of 3 signals.

```
process (a, b, c)
begin
if (a > b) then
  if (a > c) then
    max <= a;
  else
    max <= c ; -- a>b and c>=a
  end if ;
else
  if ( b > c ) then
    max <= b ; -- b>=a and b>c
  else
    max <= c ; - b>=a and c>=b
  end if ;
end if ;
end process;
```

Digital Design using VHDL 29

Notes:

An if statement is much more general since a branch of the if statement can be a sequence of sequential statements. Proper and disciplined use of an if statement can make code more descriptive and sometimes even more efficient. For example, an if statement is a sequential statement, and thus it can be nested in a branch of another if statement



If Statements

Incomplete Branch

- In VHDL, the else branch can be omitted.

➤ Ex:

```
process (a, b)
begin
  if (a=b) then
    eq <= '1';
  end if ;
end process;
```

is the same as

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

- This implies a circuit with a closed feedback loop, which constitutes unwanted memory and doesn't meet the wanted specification.

Digital Design using VHDL 30

Notes:

We learned that an incomplete sensitivity list, in which one or more input signals are omitted, may lead to unexpected circuit behavior. This may also happen to the incomplete branch and incomplete signal assignment. According to VHDL semantics, the else branch is optional and a signal does not need to be assigned in all branches. Although syntactically correct, the omissions introduce unwanted memory elements (i.e., latches).

For example, the above statement is an attempt to code a comparator that compares the a and b inputs and asserts the eq output when a and b are equal. The code is syntactically correct. When a is equal to b, the eq signal becomes '1'. When a is not equal to b, there is no else branch and thus no action is taken. VHDL semantics specifies that the eq signal does not change and keeps its previous value. This implies a circuit with a closed feedback loop, which constitutes internal states or memory. Clearly, this description does not meet the intended specification.

If Statements

Incomplete Branch

- The correct is:

```
process (a, b)
begin
    if (a=b) then
        eq <= '1' ;
    else
        eq <='0';
    end if ;
end process;
```

- For a combinational circuit, the else branch should always be included to avoid the unwanted memory or latch.



If Statements

Incomplete Signal Assignment

- It is possible that a signal is assigned only in some branches of if statements, but not all branches.
- Although syntactically correct, the incomplete signal assignment infers unwanted memory.

➤ Example:

```
process (a, b)
begin
    if (a>b) then
        gt <= '1' ;
    elsif (a=b) then
        eq <='1' ;
    else
        lt <= '1';
    end if ;
end process;
```

Digital Design using VHDL 32

Notes:

The following statement attempts to describe a comparator with three outputs, gt, lt and eq, which indicate the conditions “a is greater than b,” “a is less than b” and “a is equal to b”.

The VHDL semantics specifies that a signal will keep its previous value if it is not assigned. When a is greater than b, the first branch is taken and the gt signal becomes '1'. The eq and lt signals keep their previous values since they are not assigned. A similar situation occurs in two other branches since only one output signal is assigned. This implies that three unwanted memory elements are inferred from the code.

If Statements

Incomplete Signal Assignment

- The correct is:

```
process (a, b)
begin
    if (a>b) then
        gt <= '1';
        eq <='0';
        lt <= '0';
    elsif (a=b) then
        gt <= '0';
        eq <='1';
        lt <= '0';
    else
        gt <= '0';
        eq <='0';
        lt <= '1';
    end if ;
end process;
```

Digital Design using VHDL 33

If Statements

Incomplete Signal Assignment

- One way to make the code compact and clear is to assign a *default value* for each signal in the beginning of the process:

```
process (a, b)
begin
    gt <= '0';
    eq <='0';
    lt <= '0';
    if (a>b) then
        gt <= '1';
    elsif (a=b) then
        eq <='1';
    else
        lt <= '1';
    end if ;
end process;
```

Digital Design using VHDL 34

Notes:

Recall that, in a process, only the last signal assignment takes effect. If a signal is assigned in a branch of the if statement, that assignment takes effect. If it is not assigned in any branch, the default assignment takes effect. The output signals are therefore always assigned. We can treat the assignment of a default value as shorthand for the previous code segment.

For a combinational circuit, an output signal should be assigned in all branches of an if statement. It is a good practice to assign a default value at the beginning of the process to cover the unassigned branches.

If Statements

Conceptual Implementation

- Since a simple one-output-signal if statement is equivalent to a conditional signal assignment statement, We can apply the same procedure as in the conditional signal assignment statement to derive the conceptual block diagram.
- It will be identical to that of conditional signal assignment statement.
- In addition, an if statement is more flexible and can accommodate more than one statement in each branch and can form nested statements.

Digital Design using VHDL 35

Notes:

An if statement evaluates a set of Boolean expressions in sequential order and takes action when the first Boolean condition is met. To achieve this in hardware, we need a priority routing network similar to that of a conditional signal assignment statement. A simple one-output-signal if statement is equivalent to a conditional signal assignment statement. We can apply the same procedure as that described for conditional signal assignment to derive the conceptual block diagram for the simple if statement.

Consider an if statement with four branches:

```
if boolean_expr_1 then
  sig <= value_expr_1 ;
elsif boolean_expr_2 then
  sig <= value_expr_2 ;
elsif boolean_expr_3 then
  sig <= value_expr_3 ;
```

```
else
sig <= value_expr_4 ;
end if ;
```

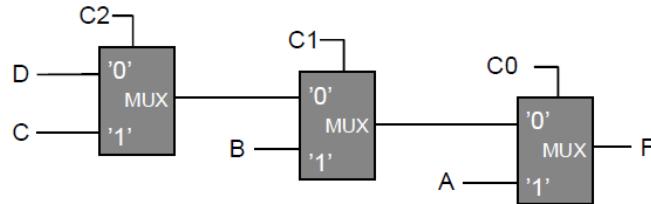
We first derive the circuit for the first branch by constructing the rightmost 2-to-1 multiplexing circuit and the boolean_expr_1 and value_expr_1 circuits. We can then repeat the process and complete the implementation branch by branch. The finished diagram is identical to that of conditional signal assignment.

If Statements

Conceptual Implementation

➤ Example:

```
process (C0, C1, C2, A, B, C, D)
begin
if C0 = '1' then
F <= A;
elsif C1 = '1' then
F <= B;
elsif C2 = '1' then
F <= C;
else
F <= D;
end if;
end process;
```



Digital Design using VHDL 36

Notes:

Testing a series of conditions is made easier by the `elsif` part of the `if` statement. Each condition is tested in turn until a condition is found which evaluates to true, then that branch of the `if` statement is executed, with the effect that earlier conditions take priority over later conditions.

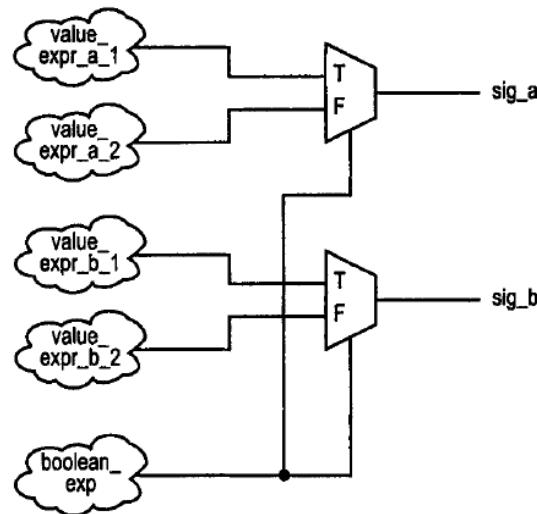
Note that `elsif` is a single keyword - writing `elseif` or `else if` are common mistakes!

If Statements

Conceptual Implementation

- Example (an if statement with two output signals):

```
if boolean_expr then
  sig_a <= value_expr_a_1;
  sig_b <= value_expr_b_1;
else
  sig_a <= value_expr_a_2;
  sig_b <= value_expr_b_2 ;
end if;
```



Digital Design using VHDL 37

Notes:

Since there are two signals, two separating routing networks are needed. When each routing network has its own multiplexer, the two networks use the same Boolean expressions to control the selection signals of the multiplexers. Thus, the boolean_exp circuit is actually shared. We can apply the same idea to derive a conceptual diagram for an if statement with more output signals.



If Statements

Conceptual Implementation

- Example (a nested if statement):

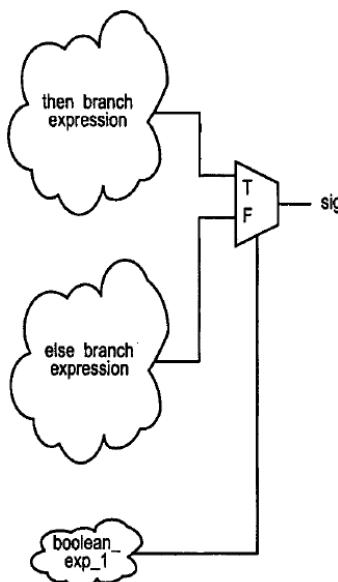
```
if boolean_expr_1 then
    if boolean_expr_2 then
        signal_a <= value_expr_1 ;
    else
        signal_a <= value_expr_2;
    end if ;
else
    if boolean_expr_3 then
        signal_a <= value_expr_3;
    else
        signal_a <= value_expr_4;
    end if;
end if ;
```

Digital Design using VHDL 38

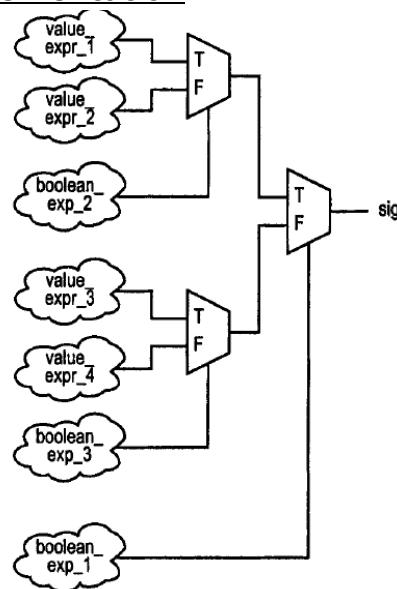
If Statements

➤ Exam

Conceptual Implementation



(a) Outer if statement



(b) Two inner if statements

HDL 39

Notes:

The conceptual diagram can be constructed in a hierachal manner. We first derive the routing structure for the outer if statement, as in Figure (a), and then realize the two inner if statements inside the then and else branches of the outer if statement, as in Figure (b). We can apply this procedure repeatedly if the code consists of more nested levels.



Case Statement

➤ Syntax:

```
case case_expression is
    when choice_1 =>
        sequential statements;
    when choice_2 =>
        sequential statements ;
    .
    .
    when choice_n =>
        sequential statements;
end case ;
```

- The choice values have to be used only once and all values must be included.
- “others” can be used instead of choice_n to cover all unused values.

Digital Design using VHDL 40

Notes:

A case statement uses the value of case_expression to select a set of sequential statements. The case_expression term functions just as the select_expression term of the concurrent selected signal assignment statement. Its data type must be a discrete type or one-dimensional array. The choice_i term is a value or a set of values that can be assumed by case_expression. The choices have to be mutually exclusive (i.e., no value can be used more than once) and all-inclusive (i.e., all values must be included). The keyword others can be used in choice-n in the end to cover all unused values.

Also the “null statement” can be used. The “null statement” consists of just the reserved word null (followed by a semicolon), and is an explicit way of doing nothing. This is actually useful in the context of a case statement to indicate that no action is to be taken within a particular branch.

Ex: when others =>
 null;

Case Statement

- Example: 8 bit 4-1 MUX

```
.....
architecture case_arch of mux4 is
begin
process ( a , b , c , d , s )
begin
case s is
when "00" =>
x <= a ;
when "01" =>
x <= b ;
when "10"=>
x <= c ;
when others =>
x <= d ;
end case;
end process;
end case_arch ;
```

Digital Design using VHDL 41

Notes:

Note that there are 81 (9×9) possible combinations for the 2-bit s signal, including the normal "00", "01", "10" and "11" combinations. In the code, we use the when others clause to cover "11" and all unused combinations. The signals used in case_expression are the inputs to the circuit and thus should be included in the sensitivity list.



Case Statement

- Exercise: Simple ALU

Input ctrl	Output result
0 --	$\text{src0} + 1$
1 0 0	$\text{src0} + \text{src1}$
1 0 1	$\text{src0} - \text{src1}$
1 1 0	src0 and src1
1 1 1	src0 or src1



Case Statement

Comparison to selected signal assignment

- A case statement is like a concurrent selected signal assignment statement.
- If each when branch of a case statement consists only of the assignment of a single signal, the two statements are equivalent.
- The case statement is much more flexible and general since each when branch can consist of a sequence of sequential statements.



Case Statement

Incomplete signal assignment

- Incomplete branch can't take place because any "incomplete when clause" will lead to a syntax error.
- Incomplete signal assignment can occur and infer unwanted memory.
- Example:

```
process (a)
begin
    case a is
        when "100"|"101"|"110"|"111" =>
            high <= '1';
        when "010"|"011"=>
            middle <= '1';
        when others =>
            low <= '1';
    end case ;
end process;
```

Digital Design using VHDL 44

Notes:

Unlike an if statement, the choices of a case statement have to be inclusive, and thus no omitted when clause is allowed. Any "incomplete when clause" will lead to a syntax error and thus be detected when the VHDL code is analyzed. However, incomplete signal assignment can still occur and infer unwanted memory.

In the shown example, statements attempts to describe a priority encoder with a 3-bit input request signal, a, and three output signals, high, middle and low. The a(3) signal has the highest priority. When it is '1', the high signal will be asserted. The two other output signals are for two other lower requests. Again, the VHDL semantics specifies that a signal will keep its previous value if it is unassigned. If the a signal is "111", the first when clause is taken and the high signal is assigned a '1'. Since the middle and low signals are unspecified, they keep their previous values. A similar situation occurs in other when clauses, and therefore three unwanted memory elements are inferred. To fix



the problem, we must make sure to have the signals assigned in all when clauses.

Case Statement

Incomplete signal assignment

- The correct is to use a default assignment:

```
process (a)
begin
    high <= '0';
    middle <= '0';
    low <= '0' ;
    case a is
        when "100"|"101"|"110"|"111" =>
            high <= '1';
        when "010"|"011"=>
            middle <= '1';
        when others =>
            low <= '1' ;
    end case ;
end process;
```



Case Statement

Conceptual Implementation

- A case statement with a single output signal can be implemented by multiplexer identical to the one used in the selected signal assignment statement.
- For multiple output statements:

```
case case_exp is
when c0 =>
sig_a <= value_expr_a_0;
sig_b <= value_expr_b_0;
when c1 =>
sig_a <= value_expr_a_1;
sig_b <= value_expr_b_1;
when others =>
sig_a <= value_expr_a_2;
sig_b <= value_expr_b_2 ;
end case;
```

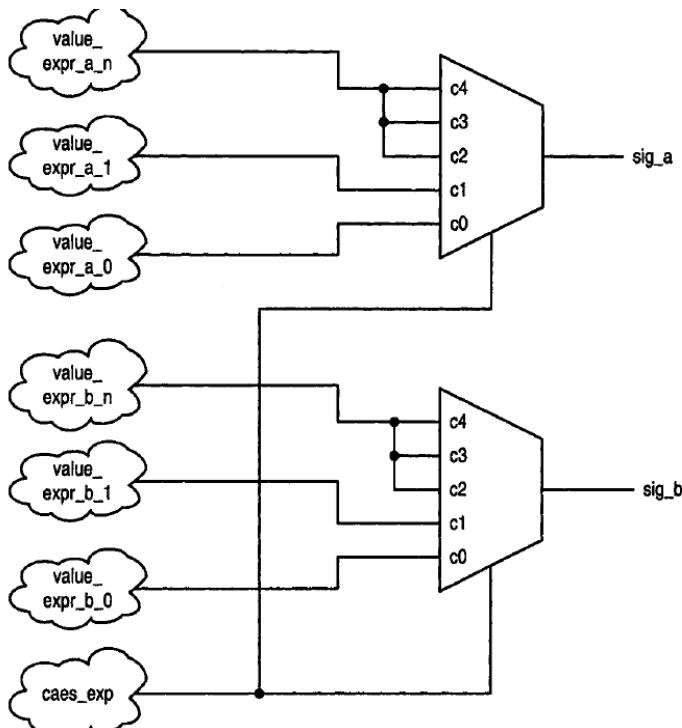
Digital Design using VHDL 46

Notes:

A case statement utilizes the value of case_expression to select a set of sequential statements to execute. Conceptually, it can be thought of as an abstract multiplexing circuit that utilizes case_expression as the selection signal to route the results of designated expressions to output signals. A case statement with a single output signal can be implemented by an abstract multiplexer identical to the one used in the selected signal assignment statement.

The previous scheme can easily be extended for a case statement with multiple output signals. We simply duplicate the abstract multiplexer for each signal and connect the case_exp to the selection signals of all multiplexers. In the shown example the case statement has two output signals. We assume that case_exp may result in one of five possible values: c0, c1 , c2, c3 and c4. The when others clause implicitly covers c2, c3 and c4.

Case Statement



using VHDL 47

Simple for loop Statement

- VHDL provides a variety of loop constructs, including simple infinite loop, for loop and while loop, as well as mechanisms to terminate a loop, including the exit statement, which skips the remaining iterations of the loop, and the next statement, which skips the remaining part of the current iteration.
- Only few, very restricted forms of loop can be realized by hardware and synthesized automatically.



Simple for loop Statement

➤ Syntax:

```
for index in loop_range loop
    sequential statements;
end loop;
```

- The loop body is repeated for a fixed number of iterations.
- The loop_range specifies a range of values.
- A loop index is used to keep track of the iteration.
- The loop index automatically takes the data type of loop_range's element.
- For synthesizable code, loop_range must be determined at the time of synthesis.

Digital Design using VHDL 49

Notes:

The for loop repeats the loop body of sequential statements for a fixed number of iterations. The loop_range term specifies a range of values between the left and right bounds. A loop index, index, is used to keep track of the iteration and takes a successive value from loop_range in each iteration, starting with the leftmost value. The loop index automatically takes the data type of loop_range's element and does not need to be declared.

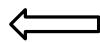
For synthesizable code, loop_range must be determined at the time of synthesis (i.e., be static) and cannot change with the input signal. The loop body is a sequence of sequential statements. It is very flexible and versatile but can be difficult or impossible to synthesize.



Simple for loop Statement

- Example 1:
4-bit xor gate

y <= a xor b;



```
library ieee;
use ieee. std_logic_1164. all ;
entity bit_xor is
port (
    a, b: in std_logic_vector(3 downto 0);
    y : out std_logic_vector(3 downto 0));
end bit_xor ;
architecture demo_arch of bit_xor is
constant WIDTH: integer := 4;
begin
process (a, b)
begin
    for i in (WIDTH-1) downto 0 loop
        y(i) <= a(i) xor b(i);
    end loop ;
end process ;
end demo_arch ;
```

Digital Design using VHDL 50

Notes:

The for loop performs bitwise xor operation on two 4-bit signals. The operation is done one bit at a time. The loop range is WIDTH-1 downto 0. We use a symbolic constant here to make the code more readable and to facilitate future modification. The loop index is i. It is local to the loop and does not need to be declared. The index assumes a value of 3, the leftmost value in the range, in the first iteration, and then assumes a value of 2 in the second iteration. The iteration continues until the value of the rightmost value, 0, is used.

The code here is just for demonstration purposes. The same operation can actually be achieved by a single statement: y <= a xor b;

Simple for loop Statement

- Example 2:
reduced xor circuit

```
library ieee;
use ieee. std_logic_1164. all ;
entity reduced_xor_demo is
port (
    a: in std_logic_vector(3 downto 0);
    y : out std_logic);
end reduced_xor_demo;
architecture demo_arch of reduced_xor_demo is
constant WIDTH: integer := 4;
signal tmp: std_logic_vector (WIDTH-1 downto 0) ;
begin
process (a,tmp)
Begin
    tmp(0) <= a(0); -- boundary bit
    for i in 1 to (WIDTH-1) loop
        tmp(i) <= a(i) xor tmp(i-1);
    end loop ;
end process ;
y <= tmp(WIDTH-1);
end demo_arch ;
```

Digital Design using VHDL 51

Notes:

The second example is a reduced-xor circuit, which performs the xor operation over a group of four signals. The reduced-xor operation of this four signals can also be achieved by the following statement

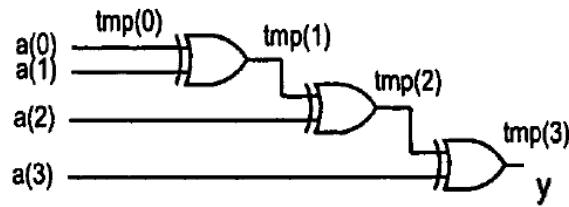
$y \leq a(0) \text{ xor } a(1) \text{ xor } a(2) \text{ xor } a(3);$

Simple for loop Statement

Conceptual Implementation

- To realize a for loop in hardware, unroll the loop and convert it into code that contains no loop constructs.
- Then the implementation can be constructed for the flattened code.
- Example: reduced xor gate

```
tmp(0) <= a(0) ;
tmp(1) <= a(1) xor tmp(0);
tmp(2) <= a(2) xor tmp(1);
tmp(3) <= a(3) xor tmp(2);
y <= tmp(3);
```



Digital Design using VHDL 52

Notes:

The basic way to realize a for loop in hardware is to unroll or flatten the loop and convert it into code that contains no loop constructs. The flattened code can then be constructed accordingly. This implies that we repeat the hardware described by the loop body for each iteration.

To unroll a loop, the range has to be constant and has to be known at the time of synthesis. That is why the range has to be static. We cannot, for example, use the value of an input signal to set the range's right boundary.

For the reduced xor code example, The for loop can be unrolled by manually substituting index i into the loop body for four iterations. Then we can derive the conceptual implantation accordingly.

Other Loop Statements

```
for PARAMETER in LOOP_RANGE loop  
    ...  
end loop;
```

```
while CONDITION loop  
    ...  
end loop;
```

```
LOOP_LABEL: loop  
    ...  
    exit;  
    ...  
    exit LOOP_LABEL;  
    ...  
    next LOOP_LABEL;  
    ...  
end loop LOOP_LABEL;
```

Jump out of loop

Jump back to top

Digital Design using VHDL 53

Notes:

Loop statements are used to execute a sequence of statements repeatedly. The for loops determine the number of repetitions before the first iteration and while loops test a condition before each iteration to determine whether to leave the loop.

Also exit and next statements can be used with any kind of loop statement. Exit causes control to jump directly out of the loop, next causes control to jump back to the top of the loop. Labeling loops allows the use of exit and next to jump out of nested loop structures. The use of an exit statement is necessary in the case of a loop without an iteration scheme (i.e. loop...end loop;) in order to terminate the loop.



Tasks

Develop a behavioral model for a limiter with three 8-bit inputs: "data_in", "lower" and "upper"; an 8-bit output, "data_out"; and a bit output, "out_of_limits".

The data_out output follows data_in so long as it is between lower and upper.

If data_in is less than lower, data_out is limited to lower.

If data_in is greater than upper, data_out is limited to upper.

The out_of_limit output indicates when data_out is limited.



Session 4

MODELING SEQUENTIAL CIRCUIT

Digital Design using VHDL 2



Topics for this Lecture

- Combinational & Sequential Circuits
- Basic Memory Elements
- Synchronous & Asynchronous Circuits
- Basic Model of Sequential Circuits
- Modeling of Basic Memory Elements
- Simple Design Examples
- More Design Examples



Combinational & Sequential Circuits

Combinational

- Circuit has no internal states.
- Its output is a function of current input only. (memory-less about the past events).

Sequential

- Circuit has an internal state, or memory.
- Its output is a function of current input as well as the internal state. (memorizes the effect of the past input values).
- The output is affected by current input value and past input values

Digital Design using VHDL 4

Notes:

A combinational circuit, by definition, is a circuit whose output, after the initial transient period, is a function of current input. It has no internal state and therefore is “memoryless” about the past events (or past inputs).

A sequential circuit, on the other hand, has an internal state, or memory. Its output is a function of current input as well as the internal state. The internal state essentially “memorizes” the effect of the past input values. The output thus is affected by current input value as well as past input values (or the entire sequence of input values). That is why we call a circuit with internal state a sequential circuit.



Basic Memory Elements

- We will use predesigned memory components.
- These can be divided into two broad categories: latch and flip-flop (FF).
- We will concentrate on D-type latch (D latch) and D-type FF (D FF).

Digital Design using VHDL 5

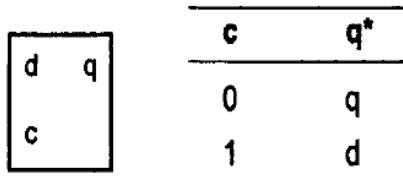
Notes:

We can add memory to a circuit in two ways. One way is to add closed feedback loops in a combinational circuit, in which the memory is implicitly implemented as system states. Because of potential timing hazards and racing, this approach is very involved and not suitable for synthesis.

The other way is to use predesigned memory components. All device libraries have certain memory cells, which are carefully designed and thoroughly analyzed. These elements can be divided into two broad categories: latch and flip-flop (FF).

D Latch

- c and d inputs are control signal and data input.
- When c is '1' , input data, d, is passed directly to output, q.
- When c is '0' , the output remains the same.
- Since the operation of the D latch depends on the level of the control signal, we say that it is *level sensitive*.



Digital Design using VHDL 6

Notes:

Since the latch is “transparent” when c is asserted, it may cause racing if a loop exists in the circuit. Because of the potential complication of timing, we normally do not use latches in synthesis.

D Flip-Flop

positive-edge-triggered D FF

- D FF has a special control signal known as a clock signal, which is labeled 'clk' .
- The D FF is activated only when the clock signal changes from '0' to '1', which is known as the rising edge of the clock. At other times, its output remains the same as its previous value.
- Since operation of the D FF depends on the edge of the clock signal, we say that it is *edge sensitive*.

clk	q*
0	q
1	q
↑	d

Digital Design using VHDL 7

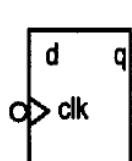
Notes:

We can say in other words, at the rising edge of the clock, a D FF takes a sample of input data, stores the value into memory, and passes the value to output.

D Flip-Flop

negative-edge-triggered D FF

- It is the same except that sampling is performed at the falling edge of the clock.



clk	q^*
0	q
1	q
↓	d

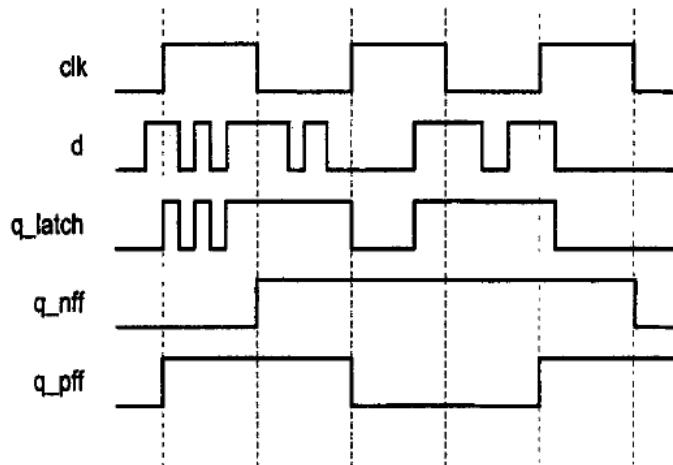
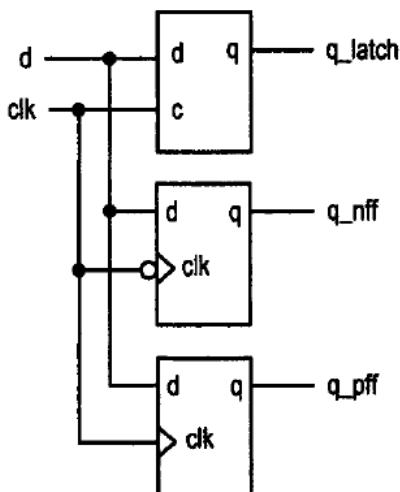
- Today's sequential circuits uses D FFs as the storage elements.

Digital Design using VHDL 8

Notes:

The sampling property of FFs has several advantages. First, variations and glitches between two rising edges have no effect on the content of the memory. Second, there will be no race condition in a closed feedback loop. The disadvantage of the D FF is its circuit size, which is about twice as large as that of a D latch. Since its benefits far outweigh the size disadvantage, today's sequential circuits normally utilize D FFs as the storage elements.

D Latch & D FlipFlop



Digital Design using VHDL 9



Synchronous versus asynchronous circuits

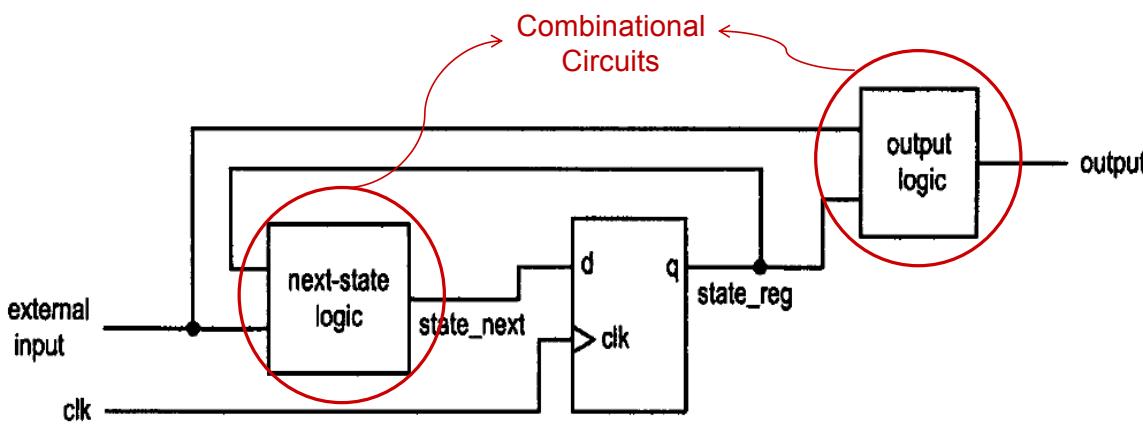
- According to the arrangement of the clock, we can mainly divide the sequential circuits into the following classes:
 - synchronous circuit: It uses FFs as memory elements, and all FFs are controlled (i.e., synchronized) by a single global clock signal. Synchronous design is the most important.
 - asynchronous circuit: It does not use a clock signal to coordinate the memory operation. The state of a memory element changes independently.
- Our discussion is focused mainly on synchronous circuits.

Digital Design using VHDL 10

Notes:

- The asynchronous circuits are divided into two categories:
 - The first category comprises circuits that consist of FFs but do not use the clock in a disciplined way. One example is the ripple counter, in which the clock port of a FF is connected to the output of the previous FF. Utilizing FFs in this way is a poor design practice.
 - The second category includes the circuits that contain “clockless” memory components, such as a latch or a combinational circuit with closed feedback loops. This kind of circuit is sometimes simply referred to as an asynchronous circuit. The design of asynchronous circuits is very different from that of synchronous circuits and is not recommended for HDL synthesis.
- Synchronous design is the most important methodology used to design and develop large, complex digital systems. It not only facilitates the synthesis but also simplifies the verification, testing, and prototyping process. Our discussion is focused mainly on this type of circuit.

Basic model of a synchronous circuit



Digital Design using VHDL 11

Notes:

The memory element, frequently known as a state register, is a collection of D FFs, synchronized by a common global clock signal. The output of the register (i.e., the content stored in the register), the `state_reg` signal, represents the internal state of the system. The next-state logic is a combinational circuit that determines the next state of the system. The output logic is another combinational circuit that generates the external output signal. Note that the output depends on the external input signal and the current state of the register.



Basic model of a synchronous circuit

Operates as follows:

- At the rising edge of the clock, the value of the state_next signal (appearing at the d port) is sampled and propagated to the q port, which becomes the new value of the state_reg signal. The value is also stored in FFs and remains unchanged for the rest of the clock period. It represent the current state of the system.
- Based on the value of the state_reg signal and external input, the next-state logic computes the value of the state_next signal and the output logic computes the value of external output.
- At the next rising edge of the clock, the new value of the state_next signal is sampled and the state_reg signal is updated. The process then repeats.

Digital Design using VHDL 12

Notes:

There are several advantages of synchronous design. First, it simplifies circuit timing. Satisfying the timing constraints is one of the most difficult design tasks. When a circuit has hundreds or even thousands of FFs and each FF is driven by an individual clock, the design and analysis will be overwhelming. Since in a synchronous circuit all FFs are driven by the identical clock signal, the sampling of the clock edge occurs simultaneously. We only need to consider the timing constraints of a single memory component.

Second, the synchronous model clearly separates the combinational circuits and the memory element. We can easily isolate the combinational part of the system, and design and analyze it as a regular combinational circuit.

Third, the synchronous design can easily accommodate the timing hazards. In a synchronous circuit, inputs are sampled and stored at the rising edge of the clock. So the glitches do not matter as long as they are settled at the time of sampling.

Modeling of Basic Memory Elements

D Latch

```
library ieee ;
use ieee. std_logic_1164. all ;
Entity dlatch is
port (
    c : in std_logic;
    d : in std_logic;
    q : out std_logic);
end dlatch ;
architecture arch of dlatch is
begin
    process ( c , d )
begin
    if ( c = '1' ) then
        q <= d ;
    end if ;
end process,
end arch ;
```

Digital Design using VHDL 13

Notes:

In this code, the value of d is passed to q when c is '1'. Note that there is no else branch in the if statement. According to the VHDL definition, q will keep its previous value when c is not '1' (i.e., c is '0'). This is just what we want for the D latch.



Modeling of Basic Memory Elements

D FF (Positive-edge-triggered)

```
library ieee ;  
use ieee. std_logic_1164. all ;  
Entity dff is  
port (  
    clk : in std_logic;  
    d : in std_logic;  
    q : out std_logic);  
end dff ;  
architecture arch of dff is  
begin  
    process ( clk )  
    begin  
        if ( clk'event and clk = '1' ) then  
            q <= d ;  
        end if ;  
    end process;  
end arch ;
```

Digital Design using VHDL 14

Notes:

The key expression to infer the D FF is the Boolean expression: `clk'event and clk='1'`. The “’event“ term is a VHDL attribute returning true when there is a change in signal value (i.e., an event). Thus, when `clk` event is true, it means that the value of `clk` has changed. When the `clk= '1'` expression is true, it means that the new value of `clk` is ‘1’.

When both expressions are true, it indicates that the `clk` signal changes to ‘1’, which is the rising edge of the `clk` signal.

The `if` statement states that at the rising of the `clk` signal, `q` gets the value of `d`. Since there is no `else` branch, it means that `q` keeps its previous value otherwise. Thus, the VHDL code accurately describes the function of a D FF. Note that the `d` signal is not in the sensitivity list. It is reasonable since the output only responds to `clk` and does nothing when `d` changes its value.

Modeling of Basic Memory Elements

D FF (Positive-edge-triggered)

```
architecture arch of dff is
begin
  process ( clk )
  begin
    if rising_edge(clk) then
      q <= d ;
    end if ;
  end process;
end arch ;
```



```
architecture wait_arch of dff is
begin
  process
  begin
    wait until clk'event and clk= '1 ' ;
    q <= d ;
  end process;
end wait_arch ;
```

Digital Design using VHDL 15

Notes:

There is a defined function, `rising_edge()`, in the IEEE std_logic_1164 package. It also returns true when the checked signal is changed to '1'.

We can also use `wait` statement inside the process. However, since the sensitivity list makes the code easier to understand, we do not use it.

Modeling of Basic Memory Elements

D FF (Negative-edge-triggered)

```
architecture arch of dff is
begin
  process ( clk )
  begin
    if ( clk'event and clk = '0' )then
      q <= d ;
    end if ;
  end process;
end arch ;
```



```
architecture arch of dff is
begin
  process ( clk )
  begin
    if falling_edge(clk) then
      q <= d ;
    end if ;
  end process;
end arch ;
```

Digital Design using VHDL 16

Notes:

Negative-edge-triggered D FF is similar to a positive-edge-triggered D FF except that the input data is sampled at the falling edge of the clock.

To specify the falling edge, we must revise the Boolean expression of the if statement: if (clk'event and clk='0') then

We can also use the function, falling_edge (), defined in the IEEE std_logic_1164 package.

Modeling of Basic Memory Elements

D FF with asynchronous reset

```
library ieee ;
use ieee.std_logic_1164.all ;
Entity dffr is
port ( clk,reset : in std_logic;
       d : in std_logic;
       q : out std_logic);
end dffr ;
architecture arch of dffr is
begin
process ( clk ,reset)
begin
    if ( reset= '1' ) then
        q <= '0';
    elsif ( clk'event and clk = '1' ) then
        q <= d ;
    end if ;
end process;
end arch ;
```

reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

Digital Design using VHDL 17

Notes:

A D FF may contain an asynchronous reset signal that clears the D FF to '0'. Note that the reset operation does not depend on the level or edge of the clock signal. Actually, we can consider that it has a higher priority than the clock-controlled operation

Both the reset and clk signals are in the sensitivity list since either can activate the process. When the process is activated, it first checks the reset signal. If it is '1', the D FF is cleared to '0'. Otherwise, the process continues checking the rising-edge condition, as in a regular D FF. Note that there is no else branch.

Asynchronous reset, as its name implies, is not synchronized by the clock signal and thus should not be used in normal synchronous operation. The major use of a reset signal is to clear the memory elements and set the system to an initial state. Once the system enters the initial state, it starts to operate synchronously and will never use the reset signal again. In many digital systems, a short reset pulse is generated when the power is turned on.

Modeling of Basic Memory Elements

D FF with asynchronous reset & preset

```
Entity dffrp is
port ( clk,reset ,preset: in std_logic;
       d : in std_logic;
       q : out std_logic);
end dffrp ;
architecture arch of dffrp is
begin
  process ( clk,reset,preset )
begin
  if ( reset= '1' ) then
    q <= '0';
  elsif ( preset= '1' ) then
    q <= '1';
  elsif ( clk'event and clk = '1' ) then
    q <= d ;
  end if ;
end process;
end arch ;
```

Digital Design using VHDL 18

Modeling of Basic Memory Elements

Register

A register is a collection of a D FFs that is driven by the same clock and reset signals.

```
Entity reg8 is
  port ( clk,reset : in std_logic;
         d : in std_logic_vector( 7 downto 0);
         q : out std_logic_vector(7 downto 0) );
end reg8 ;
architecture arch of reg8 is
begin
  process ( clk ,reset)
  begin
    if ( reset= '1' ) then
      q <= (others=>'0');
    elsif ( clk'event and clk = '1' ) then
      q <= d ;
    end if ;
  end process;
end arch ;
```

Digital Design using VHDL 19

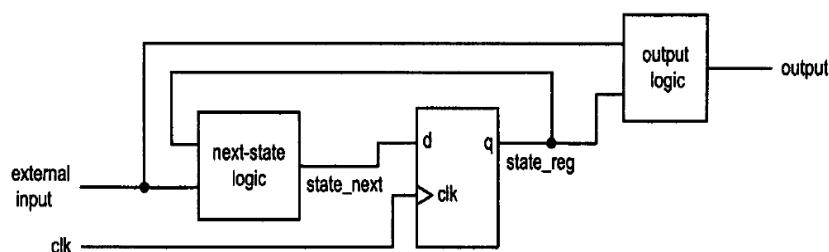
Notes:

The code is similar to D FF except that the d input and the q output are now 8 bits wide.

We use the symbol of D FF for the register. The size of the register can be derived by checking the bus width marks of the input and output connections.

Simple Design Examples

- We will first identify and separate the memory elements and then derive the next_state logic and output logic which are combinational circuits.
- A clear separation between memory elements and combinational circuits is essential for the synthesis of large, complex design and is helpful for the verification and testing processes.



Digital Design using VHDL 20

Notes:

The most effective way to derive a sequential circuit is to follow the block diagram. We first identify and separate the memory elements and then derive the next_state logic and output logic. After separating the memory elements, we are essentially designing the combinational circuits. A clear separation between memory elements and combinational circuits is essential for the synthesis of large, complex design and is helpful for the verification and testing processes. Our VHDL code description follows this principle and we always use an isolated VHDL segment to describe the memory elements.

Since identifying and separating the memory elements is the key in deriving a sequential circuit, we utilize the following coding practice to emphasize the existence of the memory elements:

- Use an individual VHDL code segment to infer memory elements. The segment should be the standard description of a D FF or register.



- Use the suffix reg to represent the output of a D FF or a register.
- Use the suffix next to indicate the next value (the d input) of a D FF or a register.



Example 1:D FF with enable

```

Entity dff_en is
port ( clk,reset ,en: in std_logic;
      d : in std_logic;
      q : out std_logic);
end dff_en;
architecture arch of dff_en is
signal q_reg : std_logic ;
signal q_next : std_logic ;
Begin
--D FF
process ( clk,reset)
begin
    if ( reset= '1' ) then
        q_reg <= '0';
    elsif ( clk'event and clk = '1' ) then
        q_reg <= q_next ;
    end if ;
end process;
```

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	1	0	q
0	1	1	d

```

-- next state logic
q_next <= d when en ='1' else
          q_reg;
--output-logic
q <= q_reg;
end arch ;
```

Digital Design using VHDL 21

Notes:

Note that the enable signal, en, has an effect only at the rising edge of the clock. This means that the signal is synchronized to the clock. At the rising edge of the clock, the FF samples both en and d. If en is '0', which means that the FF is not enabled, FF keeps its previous value. On the other hand, if en is '1', the FF is enabled and functions as a regular D FF.

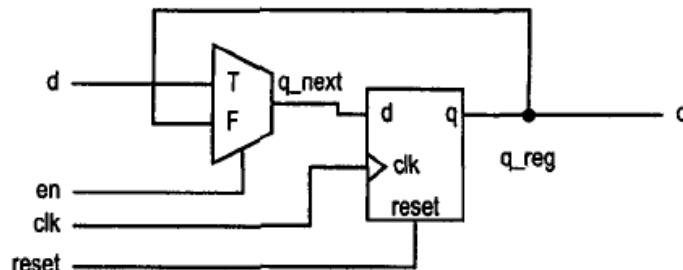
The VHDL code follows the basic sequential block diagram and is divided into three segments: a memory element, next-state logic and output logic. The memory element is a regular D FF. The next-state logic is implemented by a conditional signal assignment statement. The q_next signal can be either d or the original content of the FF, q_reg, depending on the value of en. At the rising edge of the clock, q_next will be sampled and stored into the memory element. The output logic is simply a wire that connects the output of the register to the q port.

Example 1:D FF with enable

```

Entity dff_en is
port ( clk,reset ,en: in std_logic;
      d : in std_logic;
      q : out std_logic);
end dff_en;
architecture arch of dff_en is
signal q_reg : std_logic ;
signal q_next : std_logic ;
Begin
--D FF
process ( clk,reset)
begin
    if ( reset= '1' ) then
        q_reg <= '0';
    elsif ( clk'event and clk = '1' ) then
        q_reg <= q_next ;
    end if ;
end process;
-- next state logic
q_next <= d when en ='1' else
          q_reg;
--output-logic
q <= q_reg;
end arch ;

```



Digital Design using VHDL 22



Example 2:T FF

```

Entity tff is
port ( clk,reset : in std_logic;
       t : in std_logic;
       q : out std_logic);
end tff;
architecture arch of tff is
signal q_reg : std_logic ;
signal q_next : std_logic ;
Begin
--D FF
process ( clk,reset)
begin
    if ( reset= '1' ) then
        q_reg <= '0';
    elsif ( clk'event and clk = '1' ) then
        q_reg <= q_next ;
    end if ;
end process;
```

reset	clk	t	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	q'

```

-- next state logic
q_next <= q_reg when t ='0' else
not(q_reg);
--output-logic
q <= q_reg;
end arch ;
```

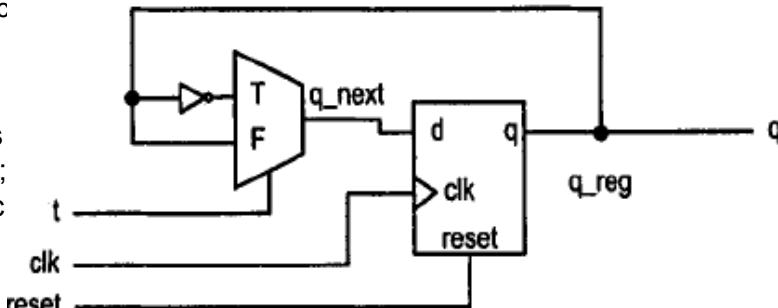
Digital Design using VHDL 23

Notes:

A T FF has a control signal, t, which specifies whether the FF to invert (i.e., toggle) its content. Note that the t signal is sampled at the rising edge of the clock.

Example 2:T FF

```
Entity tff is
port ( clk,reset : in std_logic;
       t : in std_logic;
       q : out std_logic);
end tff;
architecture arch of tff is
signal q_reg : std_logic ;
signal q_next : std_logic
Begin
--D FF
process ( clk,reset)
begin
    if ( reset= '1' ) then
        q_reg <= '0';
    elsif ( clk'event and clk = '1' ) then
        q_reg <= q_next ;
    end if ;
end process;
```

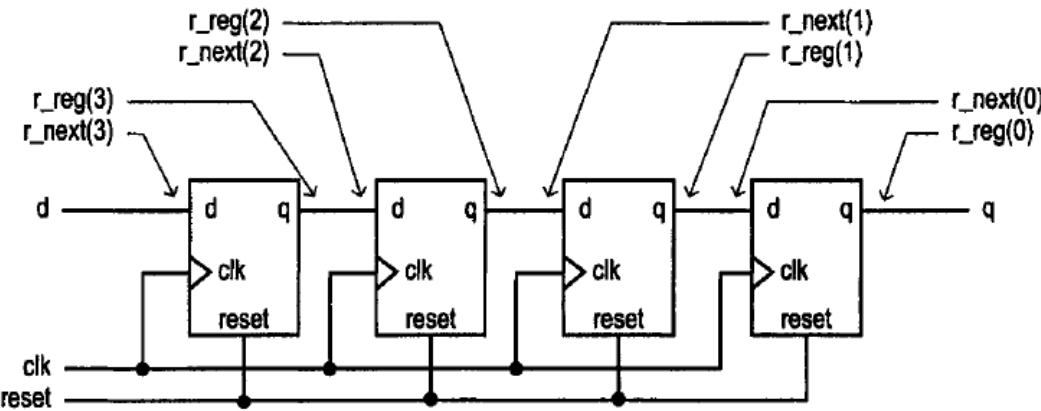


```
-- next state logic
q_next <= q_reg when t ='0' else
not(q_reg);
--output-logic
q <= q_reg;
end arch ;
```

Digital Design using VHDL 24

Example 3:Free-running shift-right register

- A free-running shift register performs the shifting operation continuously. It has no other control signals.



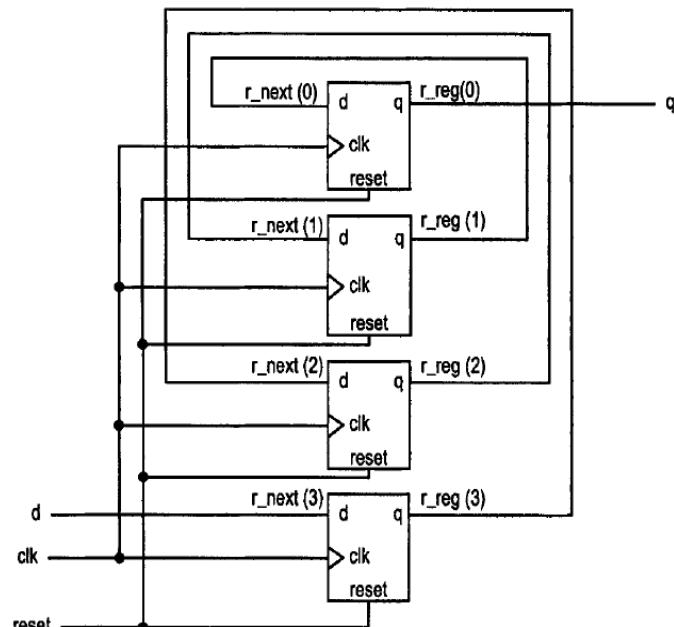
- We will rearrange the FFs and align them vertically.

Digital Design using VHDL 25

Notes:

A shift register shifts the content of the register left or right 1 bit in each clock cycle. One major application of a shifter register is to send parallel data through a serial line. In the transmitting end, a data word is first loaded to register in parallel and is then shifted out 1 bit at a time. In the receiving end, the data word is shifted in 1 bit at a time and reassembled.

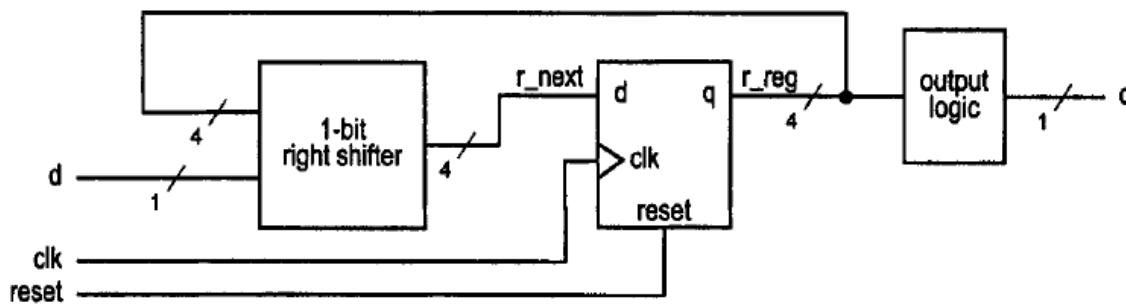
Example 3:Free-running shift-right register



Digital Design using VHDL 26

Example 3:Free-running shift-right register

- Grouping the four FFs together and treating them as a single memory block, we transform the circuit into the basic sequential circuit block diagram.



Digital Design using VHDL 27

Example 3:Free-running shift-right register

```

library ieee;
use ieee.std_logic_1164.all;
entity shift_right_register is
port(
    clk, reset: in std_logic;
    d: in std_logic;
    q: out std_logic
);
end shift_right_register;

architecture two_seg_arch of shift_right_register is
    signal r_reg: std_logic_vector(3 downto 0);
    signal r_next: std_logic_vector(3 downto 0);
begin
    -- register
    process(clk,reset)
        begin
            if (reset='1') then
                r_reg <= (others=>'0');
            elsif (clk'event and clk='1') then
                r_reg <= r_next;
            end if;
        end process;
        -- next-state logic (shift right 1 bit)
        r_next <= d & r_reg(3 downto 1);
        -- output
        q <= r_reg(0);
    end two_seg_arch;

```

Digital Design using VHDL 28

Notes:

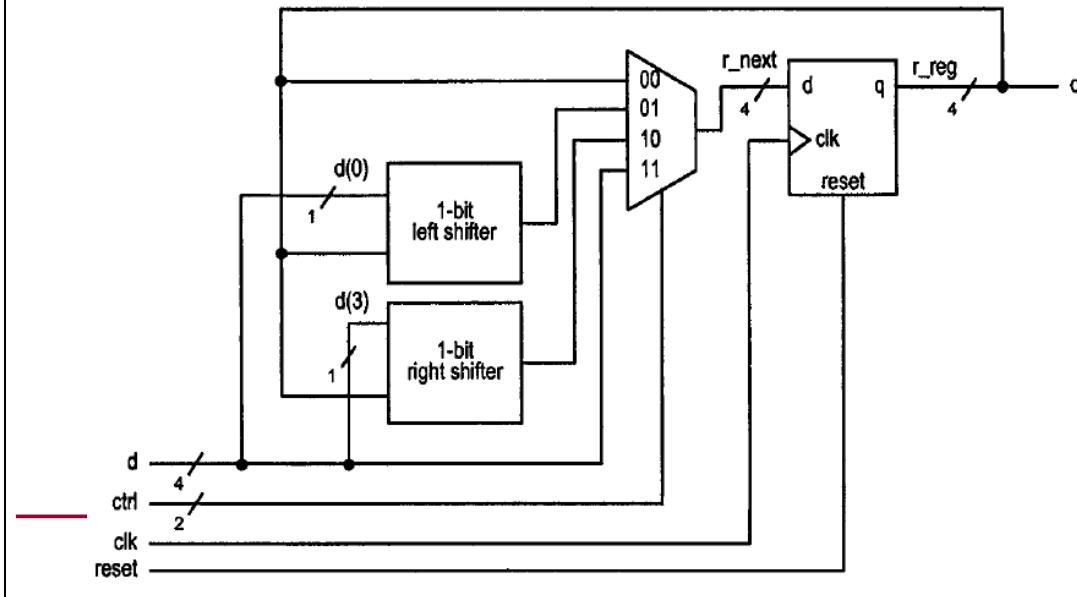
The VHDL code follows the basic sequential circuit block diagram, and the key is the code for the next-state logic. The statement

r_next <= d & r_reg(3 downto 1);

indicates that the original register content is shifted to the right 1 bit and a new bit, d, is inserted to the left. The memory element part of the code is the standard description of a 4-bit register.

Example 4: Universal Shift Register

- There are four operations: load, shift right, shift left and pause.
- A control signal, ctrl, specifies the desired operation.



29

Notes:

A universal shift register can load a parallel data word and perform shifting in either direction. There are four operations: load, shift right, shift left and pause. A control signal, ctrl, specifies the desired operation.

Note that the d(0) input and the q(3) output are used as serial-in and serial-out for the shift-left operation, and the d(3) input and the q(0) output are used as serial-in and serial-out for the shift-right operation.



Example 4: Universal Shift Register

```

library ieee;
use ieee.std_logic_1164.all;
entity shift_register is
port(
    clk, reset: in std_logic;
    ctrl: in std_logic_vector(1 downto 0);
    d: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0)
);
end shift_register;

architecture two_seg_arch of shift_register is
    signal r_reg: std_logic_vector(3 downto 0);
    signal r_next: std_logic_vector(3 downto 0);
begin
    -- register
    process(clk,reset)
        begin
            if (reset='1') then
                r_reg <= (others=>'0');
            elsif (clk'event and clk='1') then
                r_reg <= r_next;
            end if;
        end process;
        -- next-state logic
        with ctrl select
            r_next <=
                r_reg           when "00",
                r_reg(2 downto 0) & d(0) when "01",
                d(3) & r_reg(3 downto 1) when "10",
                d           when others;
        -- output
        q <= r_reg;
    end two_seg_arch;

```

Digital Design using VHDL 30



Example 5: Arbitrary-sequence counter

- A sequential counter circulates a predefined sequence of states. The next-state logic determines the patterns in the sequence.
- For example, if we need a counter to cycle through the sequence of "000", "011", "110", "101" and "111", we can construct a combinational circuit with a function table that specifies the desired patterns.

Input pattern	Next pattern
000	011
011	110
110	101
101	111
111	000

Example 5: Arbitrary-sequence counter

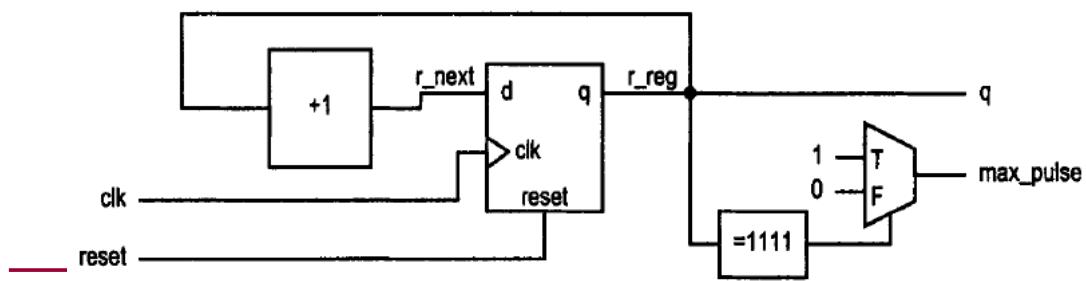
```
library ieee;
use ieee.std_logic_1164.all;
entity arbi_seq_counter4 is
port(
    clk, reset: in std_logic;
    q: out std_logic_vector(2 downto 0)
);
end arbi_seq_counter4;

architecture two_seg_arch of
arbi_seq_counter4 is
    signal r_reg: std_logic_vector(2 downto 0);
    signal r_next: std_logic_vector(2 downto 0);
begin
    -- register
    process(clk,reset)
        begin
            if (reset='1') then
                r_reg <= (others=>'0');
            elsif (clk'event and clk='1') then
                r_reg <= r_next;
            end if;
        end process;
        -- next-state logic
        r_next <= "011" when r_reg="000" else
                    "110" when r_reg="011" else
                    "101" when r_reg="110" else
                    "111" when r_reg="101" else
                    "000"; -- r_reg="111"
        -- output logic
        q <= r_reg;
    end two_seg_arch;
```

Digital Design using VHDL 32

Example 6: Free-running binary counter

- An n-bit binary counter has a register with n FFs, and its output is interpreted as an unsigned integer.
- A free-running binary counter increments the content of the register every clock cycle, counting from 0 to $(2^n - 1)$ and then repeating.
- In addition to the register output, we assume that there is a status signal, max_pulse, which is '1' when the counter is in the all-one state.



Digital Design using VHDL 33

Notes:

A binary counter circulates through a sequence that resembles the unsigned binary number. For example, a 3-bit binary counter cycles through "000", "001", "010", "011", "100", "101", "110" and "111", and then repeats.

Example 6: Free-running binary counter

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter4_pulse is
  port(
    clk, reset: in std_logic;
    max_pulse: out std_logic;
    q: out std_logic_vector(3 downto 0)
  );
end binary_counter4_pulse;

architecture two_seg_arch of
binary_counter4_pulse is

  signal r_reg: unsigned(3 downto 0);
  signal r_next: unsigned(3 downto 0);
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  r_next <= r_reg + 1;
  -- output logic
  q <= std_logic_vector(r_reg);
  max_pulse <= '1' when r_reg="1111" else
  '0';
end two_seg_arch;

```

Digital Design using VHDL 34

Notes:

The next_state logic consists of an incrementor, which calculates the new value for the next state of the register. Note that the definition requests the 4-bit binary counter counts in a wrapped-around fashion; i.e., when the counter reaches the maximal number, "1111", it should return to "0000" and start over again. This is achieved here because in the IEEE numeric_std package, the definition of '+' on the unsigned data type is modeled after a hardware adder, which behaves like wrapping around when the addition result exceeds the range.



Example 7: Featured Binary Counter

- Rather than leaving the counter in the free-running mode, we can add more control. In the counter, we can synchronously clear the counter to 0, load a specific value, and enable or pause the counting.

syn_clr	load	en	q*	Operation
1	-	-	00...00	synchronous clear
0	1	-	d	parallel load
0	0	1	q+1	count
0	0	0	q	pause

Example 7: Featured Binary Counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter4_feature is
  port(
    clk, reset: in std_logic;
    syn_clr, en, load: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0)
  );
end binary_counter4_feature;

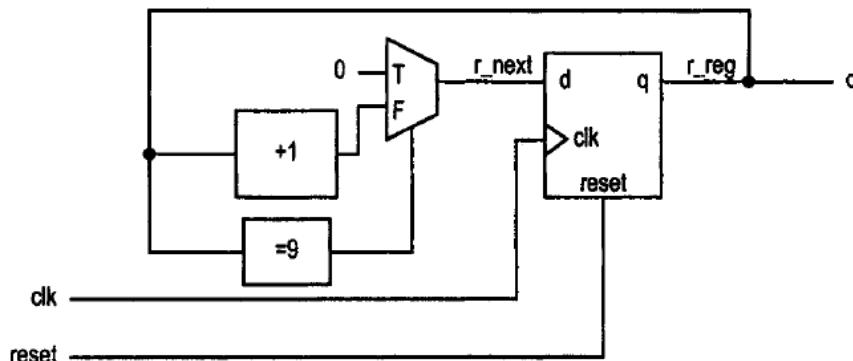
architecture two_seg_arch of
binary_counter4_feature is

  signal r_reg: unsigned(3 downto 0);
  signal r_next: unsigned(3 downto 0);
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  r_next <= (others=>'0') when syn_clr='1' else
    unsigned(d) when load='1' else
      r_reg + 1 when en ='1' else
        r_reg;
  -- output logic
  q <= std_logic_vector(r_reg);
end two_seg_arch;
```

Digital Design using VHDL 36

Example 8: Decade Counter

- Instead of utilizing all possible 2^n states of an n-bit binary counter, we sometime only want the counter to circulate through a subset of the states.
- Ex: A decade counter (mod-10 counter) which counts from 0 to 9 and then repeats.



Digital Design using VHDL 37

Notes:

Instead of utilizing all possible 2^n states of an n-bit binary counter, we sometime only want the counter to circulate through a subset of the states. We define a mod-m counter as a binary counter whose states circulate from 0 to $m - 1$ and then repeat. Here we consider the design of a mod-10 counter, also known as a decade counter. The counter counts from 0 to 9 and then repeats. We need at least 4 bits to accommodate the 10 possible states, and the output is 4 bits wide.

Example 8: Decade Counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity decade_counter is
port(
    clk, reset: in std_logic;
    q: out std_logic_vector(3 downto 0)
);
end decade_counter;

architecture two_seg_arch of
decade_counter is

constant TEN: integer := 10;
signal r_reg: unsigned(3 downto 0);
signal r_next: unsigned(3 downto 0);
begin
    -- register
    process(clk,reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
    -- next-state logic
    r_next <= (others=>'0') when r_reg=(TEN-1) else
        r_reg + 1;
    -- output logic
    q <= std_logic_vector(r_reg);
end two_seg_arch;
```

Digital Design using VHDL 38

Notes:

The key to this design is the next_state logic. When the counter reaches 9, as indicated by the condition $r_reg=(TEN-1)$, the next value will be 0. Otherwise, the next value will be incremented by 1.



Example 9: Programmable mod-m counter

- We will design a "programmable" 4-bit mod-m counter.
- m is a 4-bit input signal which is interpreted as an unsigned number.
- The range of m is from "0010" to "1111".
- The maximal number in the counting sequence of a mod-m counter is $m-1$.
- when the counter reaches $m-1$, the next state should be 0.

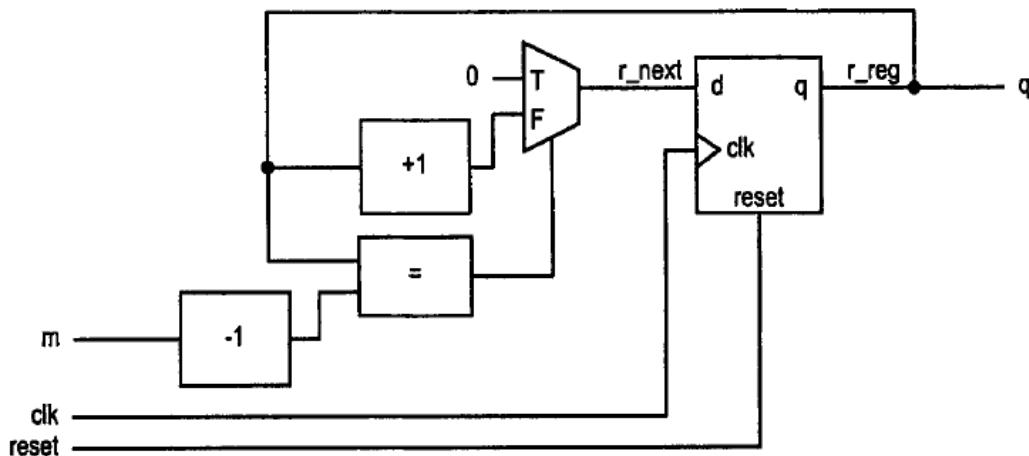
Digital Design using VHDL 39

Notes:

We can easily modify the code of the previous decade counter to a mod-m counter for any m . However, the counter counts a fixed, predefined sequence. In this example, we design a "programmable" 4-bit mod-m counter, in which the value of m is specified by a 4-bit input signal, m , which is interpreted as an unsigned number. The range of m is from "0010" to "1111", and thus the counter can be programmed as a mod-2, mod-3, . . . , or mod- 15 counter. The maximal number in the counting sequence of a mod-m counter is $m - 1$. Thus, when the counter reaches $m - 1$, the next state should be 0. Our first design is based on this observation. The VHDL code is similar to the decade counter except that we need to replace the $r_reg=(TEN-1)$ condition of the next-state logic with $r_reg=(\text{unsigned}(m)-1)$.

Example 9: Programmable mod-m counter

First Design



Digital Design using VHDL 40

Notes:

When the counter reaches $m - 1$, the next state should be 0. Our first design is based on this observation. The VHDL code is similar to the decade counter except that we need to replace the $r_reg=(TEN-1)$ condition of the next-state logic with $r_reg=(\text{unsigned}(m)-1)$.

Example 9: Programmable mod-m counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity prog_counter is
port(
    clk, reset: in std_logic;
    m: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0)
);
end prog_counter;

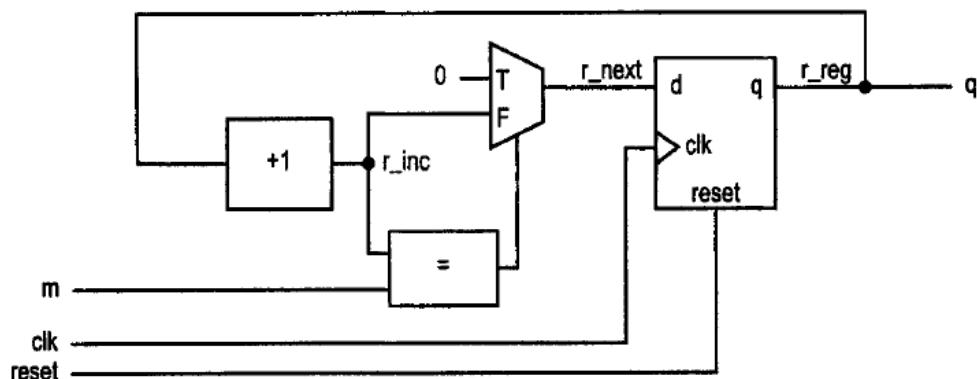
architecture two_seg_clear_arch of
prog_counter is

signal r_reg: unsigned(3 downto 0);
signal r_next: unsigned(3 downto 0);
begin
    -- register
    process(clk,reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
    -- next-state logic
    r_next <= (others=>'0') when r_reg=(unsigned(m)-1)
    else
        r_reg + 1;
    -- output logic
    q <= std_logic_vector(r_reg);
end two_seg_clear_arch;
```

Digital Design using VHDL 41

Example 9: Programmable mod-m counter

Second Design (More Efficient)



Digital Design using VHDL 42

Example 9: Programmable mod-m counter

```
architecture two_seg_effi_arch of
prog_counter is
    signal r_reg: unsigned(3 downto 0);
    signal r_next, r_inc: unsigned(3 downto 0);
begin
    -- register
    process(clk,reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
    -- next-state logic
    r_inc <= r_reg + 1;
    r_next <= (others=>'0') when r_inc=unsigned(m) else
        r_inc;
    -- output logic
    q <= std_logic_vector(r_reg);
end two_seg_effi_arch;
```

Digital Design using VHDL 43



Home Work

1. A JK FF is defined as in the table below. Use a D FF and a combinational circuit to design this FF. Derive the VHDL code and draw the conceptual diagram for this circuit.

j	k	clk	q*
-	-	0	q
-	-	1	q
0	0	J	q
0	1	K	0
1	0	J	1
1	1	K	q'

JK FF

Digital Design using VHDL 44

Home Work

2. Expand the design of the universal shift register to include rotate-right and rotate-left operations. To accommodate this, the ctrl signal has to be extended to 3 bits. Derive the VHDL code for this circuit.
3. Consider an 8-bit free-running up-down binary counter. It has a control signal, up. The counter counts up when the up signal is '1' and counts down otherwise. Derive the VHDL code for this circuit and draw the conceptual top-level diagram.

Digital Design using VHDL 45

Home Work

4. Consider a 4-bit counter that counts from 3 ("0011") to 12 ("1100") and then wraps around. If the counter enters an unused state (such as "0000") because of noise, it will restart from "0011" at the next rising edge of the clock. Derive the VHDL code for this circuit and draw the conceptual top-level diagram.



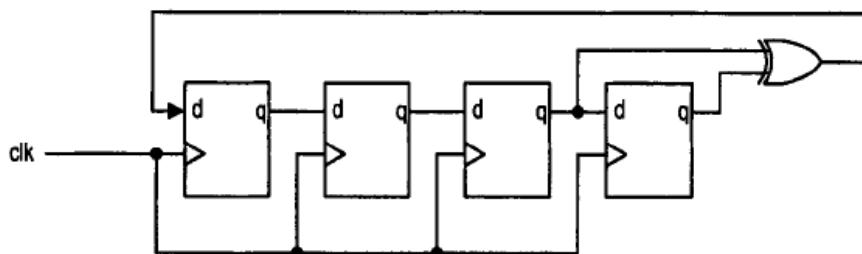
More Design Examples

- LFSR (linear feedback shift register)
- Decimal counter
- Pulse width modulation circuit
- Register file
- Register-based synchronous FIFO buffer

Digital Design using VHDL 47

LFSR (Linear Feedback Shift Register)

- The linear feedback shift register (LFSR) is a shift register that utilizes a special feedback circuit to generate the serial input value.
- It performs xor operation on certain bits of the register to force the register to cycle through certain states.
- For n-bit LFSR, the register circulates through $(2^n)-1$ states.
- Ex: 4-bit LFSR (circulates through 15 state)



Digital Design using VHDL 48

Notes:

The linear feedback shift register (LFSR) is a shift register that utilizes a special feedback circuit to generate the serial input value. The feedback circuit is essentially the next-state logic. It performs xor operation on certain bits of the register and forces the register to cycle through a set of unique states. In a properly designed n-bit LFSR, we can use a few xor gates to force the register to circulate through $2^n - 1$ states.

For a 4-bit LFSR is, the two LSB signals of the register are xored to generate a new value, which is fed back to the serial-in port of the shift register. Assume that the initial state of register is "1000". The circuit will circulate through the 15 (i.e., $2^4 - 1$) states as follows:

"1000", "0100", "0010", "1001", "1100", "0110", "1011", "0101", "1010",
"1101", "1110", "1111", "0111", "0011", "0001".



Note that the "0000" state is not included and constitutes the only missing state. If the LFSR enters this state accidentally, it will be stuck in this state.

The construction of LFSRs is based on the theoretical study of finite fields. The term linear comes from the fact that the general feedback equation of an LFSR is described by an expression of the and and xor operators, which form a linear system in algebra.



LFSR (Linear Feedback Shift Register)

- The LFSR has some interesting properties:
 - An n-bit LFSR can cycle through up to $2^n - 1$ states. The all-zero state is excluded (ex: For 4-bit LFSR, the "0000" state is not included and if the LFSR enters this state accidentally, it will be stuck in this state).
 - A feedback circuit to generate maximal number of states exists for any n.
 - The sequence generated by the feedback circuit is pseudorandom, which means that the sequence exhibits a certain statistical property and appears to be random.
- Note that the LFSR cannot be initialized with the all-zero pattern. The initial value of the sequence is known as a seed. We use a constant to define the initial value and load it into the LFSR during system initialization.

Digital Design using VHDL 49

Notes:

The feedback circuit depends on the number of bits of the LFSR. Despite its irregular pattern, the feedback expressions are very simple, involving either one or three xor operators most of the time.

Table lists the feedback expressions for register sizes between 2 and 8 as well as several larger values.

Register size	Feedback expression
2	$q_1 \oplus q_0$
3	$q_1 \oplus q_0$
4	$q_1 \oplus q_0$
5	$q_2 \oplus q_0$
6	$q_1 \oplus q_0$
7	$q_3 \oplus q_0$
8	$q_4 \oplus q_3 \oplus q_2 \oplus q_0$
16	$q_5 \oplus q_4 \oplus q_3 \oplus q_0$
32	$q_{22} \oplus q_2 \oplus q_1 \oplus q_0$
64	$q_4 \oplus q_3 \oplus q_1 \oplus q_0$
128	$q_{29} \oplus q_{17} \oplus q_2 \oplus q_0$

We assume that the output of the n-bit shift register is $q_{n-1}, q_{n-2}, \dots, q_1, q_0$. The result of the feedback expression is to be connected to the serial-in port of the shift register (i.e., the input of the $(n - 1)$ th FF). Once we know the feedback expression, the coding of LFSR is straightforward.

The unique properties of LFSR make them useful in a variety of applications. The first type of application utilizes its pseudo randomness property to scramble and descramble data, as in testing, encryption and modulation. The second type takes advantages of its simple combinational feedback circuit. For example, we can use just three xor gates in a 32-bit LFSR to cycle through $2^{32} - 1$ states. By comparison, we need a fairly large 32-bit incrementor to cycle through 2^{32} states in a binary counter. Thus, an LFSR can replace other counters for applications in which the order of the counting states is not important.



4-bit LFSR

```
library ieee;
use ieee.std_logic_1164.all;
entity lfsr4 is
port(
    clk, reset: in std_logic;
    q: out std_logic_vector(3 downto 0)
);
end lfsr4;

architecture no_zero_arch of lfsr4 is

signal r_reg, r_next: std_logic_vector(3 downto 0);
signal fb: std_logic;
constant SEED: std_logic_vector(3 downto 0):="0001";
begin

    -- register
    process(clk,reset)
    begin
        if (reset='1') then
            r_reg <= SEED;
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
    -- next-state logic
    fb <= r_reg(1) xor r_reg(0);
    r_next <= fb & r_reg(3 downto 1);
    -- output logic
    q <= r_reg;
end no_zero_arch;
```

Digital Design using VHDL 50



4-bit LFSR

- It is possible to use an additional circuit to insert the all-zero state into the counting sequence so that an n-bit LFSR can circulate through all 2^n states.
- The revised design will insert the all-zero state, "00 . . . 00", between the "00 . . . 01" and "10 . . 00" states.
- The modified feedback fzero has the following expression:

$$f_{\text{zero}} = f_b \oplus (q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1)$$

- The modified design is known as a Bruijn counter.
- Note that the all-zero state can be loaded into the register during system initialization.

Digital Design using VHDL 51

Notes:

The all-zero state is excluded in a pure LFSR. It is possible to use an additional circuit to insert the all-zero state into the counting sequence so that an n-bit LFSR can circulate through all 2^n states. This scheme is based on the following observation. In any LFSR, a '1' will be shifted in after the "00 . . . 01" state since the all-zero state is not possible. In other words, the feedback value will be '1' when the n - 1 MSBs are 0's and the state following "00 . . . 01" will always be "10 . . 00". The revised design will insert the all-zero state, "00 . . . 00", between the "00 . . . 01" and "10 . . 00" states.

Let the output of an n-bit shift register be $q_{n-1}, q_{n-2}, \dots, q_1, q_0$ and the original feedback signal of the LFSR be f_b .

The modified expression can be analyzed as follows:



- The expression $q'_{n-1} \cdot q'_{n-2} \dots \cdot q'2 \cdot q'1$ indicates the condition that the $n - 1$ MSBs are 0's. This condition can only be true when the LFSR is in "00 . . . 01" or "00 . . . 00" state.
- If the condition above is false, the value of fzero is fb since $fb \text{ xor } 0 = fb$. This implies that the circuit will shift in a regular feedback value and follow the original sequence except for the "00 . . . 01" or "00 . . . 00" states.
- If the current state of the register is "00 . . . 01", the value of fb should be '1' and the expression $fb \text{ xor } (q'_{n-1} \cdot q'_{n-2} \dots \cdot q'2 \cdot q'1)$ becomes 1 xor 1. Thus, a '0' will be shifted into the register at the next rising edge of the clock and the next state will be "00 . . . 00".
- If the current state of the register is "00 . . . 00", the value of fb should be '0' and the expression $fb \text{ xor } (q'_{n-1} \cdot q'_{n-2} \dots \cdot q'2 \cdot q'1)$ becomes 0 or 1. Thus, a '1' will be shifted into the register at the next rising edge of the clock and the next state will be "10 . . . 00".
- Once the shift register reaches "10 . . . 00", it returns to the regular LFSR sequence.

The analysis clearly shows that the modified feedback circuit can insert the all-zero state between the "00 . . . 01" and "10 . . . 00" states. Technically, the and operation of the revised feedback expression destroys the "linearity," and thus the circuit is no longer a linear feedback shift register. The modified design is sometimes known as a Bruijn counter.

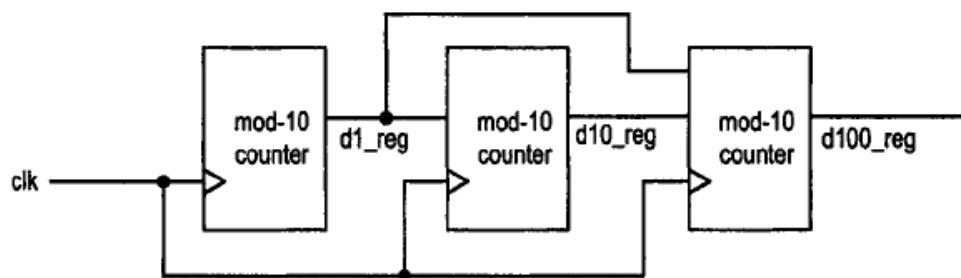
4-bit LFSR

```
architecture with_zero_arch of lfsr4 is
  signal r_reg, r_next: std_logic_vector(3 downto 0);
  signal fb, zero, fzero: std_logic;
  constant seed: std_logic_vector(3 downto 0):="0000";
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= seed;
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  fb <= r_reg(1) xor r_reg(0);
  zero <= '1' when r_reg(3 downto 1)="000" else
    '0';
  fzero <= zero xdr fb;
  r_next <= fzero & r_reg(3 downto 1);
  -- output logic
  q <= r_reg;
end with_zero_arch;
```

Digital Design using VHDL 52

Decimal Counter

- A decimal counter circulates the patterns in BCD format.
- One possible way to construct a decimal counter is to divide the counter into stages of decade counters and use special enable logic to control the increment of the individual decade counters.
- We will design a 3-digit (12 bit) decimal counter that counts from 000 to 999.



Digital Design using VHDL 53

Notes:

A decimal counter circulates the patterns in binary-coded decimal (BCD) format. The BCD code uses 4 bits to represent a decimal number. For example, the BCD code for the three digit decimal number 139 is "0001 0011 1001". The decimal counter follows the decimal counting sequence and the number following 139 is 140, which is represented as "0001 0100 0000".

Consider a 3-digit (12-bit) decimal counter that counts from 000 to 999 and then repeats. It can be implemented by cascading three special decade counters. The leftmost decade counter represents the least significant decimal digit. It is a regular mod-10 counter that counts from 0 to 9 (i.e., from "0000" to "1001") and repeats. The middle decade counter is a mod-10 counter with a special enable circuit. It increments only when the least significant decimal digit reaches 9. The rightmost decade counter represents the most significant

decimal digit, and it increments only when the two least significant decimal digits are equal to 99. Note that when the counter reaches 999, it will return to 000 at the next rising edge of the clock.

Decimal Counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity decimal_counter is
port(
    clk, reset: in std_logic;
    d1, d10, d100: out std_logic_vector(3 downto 0)
);
end decimal_counter;

architecture concurrent_arch of decimal_counter is
    signal d1_reg, d10_reg, d100_reg: unsigned(3 downto 0);
    signal d1_next, d10_next, d100_next: unsigned(3 downto 0);
begin
```

Digital Design using VHDL 54

Decimal Counter

```
-- register
process(clk,reset)
begin
    if (reset='1') then
        d1_reg <= (others=>'0');
        d10_reg <= (others=>'0');
        d100_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
        d1_reg <= d1_next;
        d10_reg <= d10_next;
        d100_reg <= d100_next;
    end if;
end process;
```

Digital Design using VHDL 55

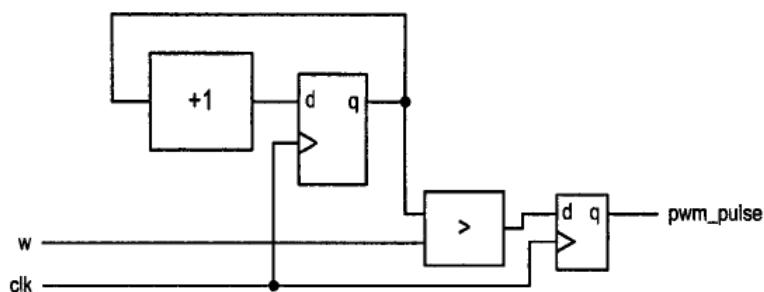
Decimal Counter

```
-- next-state logic
d1_next <= "0000" when d1_reg="1001" else
      d1_reg + 1;
d10_next <= "0000" when (d1_reg="1001" and d10_reg="1001") else
      d10_reg + 1 when d1_reg="1001" else
      d10_reg;
d100_next <=
      "0000" when (d1_reg="1001" and d10_reg="1001" and d100_reg="1001") else
      d100_reg + 1 when (d1_reg="1001" and d10_reg="1001") else
      d100_reg;
-- output
d1 <= std_logic_vector(d1_reg);
d10 <= std_logic_vector(d10_reg);
d100 <= std_logic_vector(d100_reg);
end concurrent_arch;
```

Digital Design using VHDL 56

Pulse Width Modulation (PWM) Circuit

- A PWM circuit generates an output pulse with an adjustable duty cycle.
- We will design a PWM circuit whose duty cycle can be adjusted in increments of 1/16. A 4-bit control signal, w , which is interpreted as an unsigned integer, specifies the desired duty cycle. The duty cycle will be 16/16 when w is "0000", and will be $w/16$ otherwise.



Digital Design using VHDL 57

Notes:

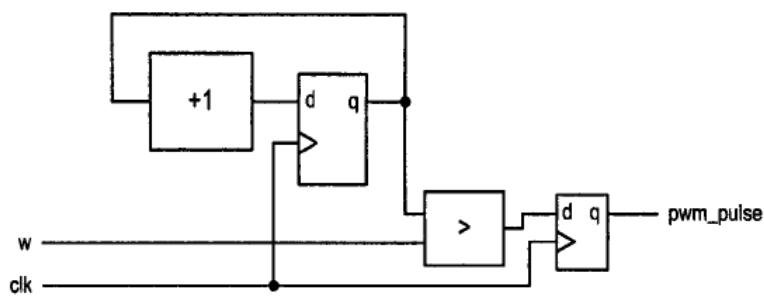
Instead of using the counting patterns directly, some applications generate output signals based on the state of the counter. One example is a pulse width modulation (PWM) circuit.

In a square wave, the duty cycle is defined as the percentage of time that the signal is asserted as '1' in a period. For example, the duty cycle of a symmetric square wave is 50% since the signal is asserted half of the period. A PWM circuit generates an output pulse with an adjustable duty cycle. It is frequently used to control the on-off time of an external system.

Consider a PWM circuit whose duty cycle can be adjusted in increments of 1/16, i.e., the duty cycle can be 1/16, 2/16, 3/16, . . . , 15/16, 16/16. A 4-bit control signal, w , which is interpreted as an unsigned integer, specifies the desired duty cycle. The duty cycle will be 16/16 when w is "0000", and will be $w/16$ otherwise. This circuit can be implemented by a mod-16 counter with a special output circuit.

Pulse Width Modulation (PWM) Circuit

- The mod-16 counter circulates through 16 patterns.
- An output circuit compares the current pattern with the w signal and asserts the output pulse when the counter's value is smaller than w .
- The output pulse's period is 16 times the clock period, and $w/16$ of the period is asserted.



Digital Design using VHDL 58

Pulse Width Modulation (PWM) Circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pwm is
port(
    clk, reset: in std_logic;
    w: in std_logic_vector(3 downto 0);
    pwm_pulse: out std_logic
);
end pwm;

architecture two_seg_arch of pwm is
    signal r_reg: unsigned(3 downto 0);
    signal r_next: unsigned(3 downto 0);
    signal buf_reg: std_logic;
    signal buf_next: std_logic;
begin
```

Digital Design using VHDL 59

Pulse Width Modulation (PWM) Circuit

```
-- register & output buffer
process(clk,reset)
begin
    if (reset='1') then
        r_reg <= (others=>'0');
        buf_reg <= '0';
    elsif (clk'event and clk='1') then
        r_reg <= r_next;
        buf_reg <= buf_next;
    end if;
end process;
-- next-state logic
r_next <= r_reg + 1;
-- output logic
buf_next <=
    '1' when (r_reg<unsigned(w)) or (w="0000") else
    '0';
pwm_pulse <= buf_reg;
end two_seg_arch;
```

Digital Design using VHDL 60

Notes:

Note that an additional Boolean expression, $w="0000"$, is included to accommodate the special condition. We also add an output buffer to remove any potential glitch.

REGISTERS AS TEMPORARY STORAGE

- RAM
 - RAM cell designed at transistor level.
 - Cell use minimal area.
 - Behave like a latch.
 - For mass storage.
- Register
 - D FF requires much larger area than RAM cell (Using registers as massive storage is not cost-effective).
 - Synchronous.
 - For small, fast temporal storage in a large digital system.
 - Ex: register file, fast FIFO.



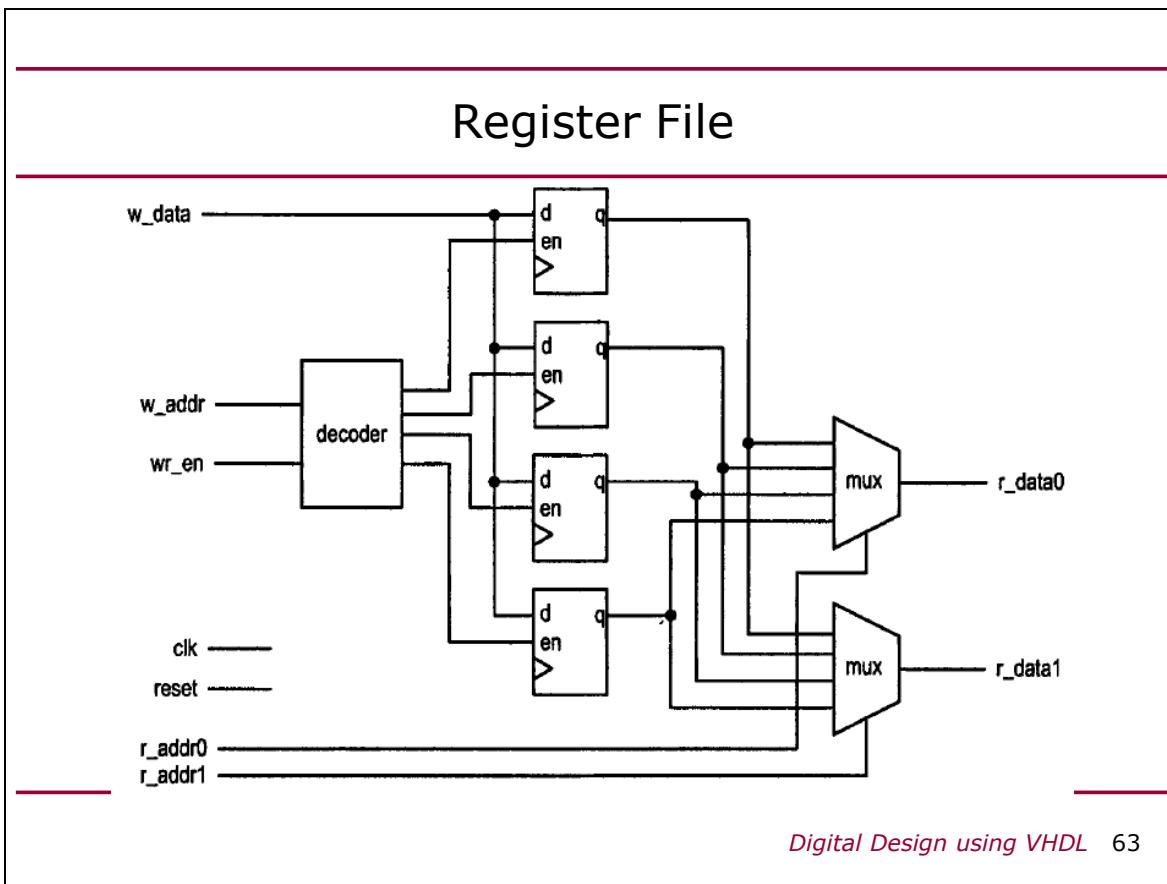
Register File

- It consists of a set of registers.
- Each register is identified with an address.
- For data access, the register file provides only one write port (with write enable signal) and few read ports, which are shared by all registers.
- Ex: register file with four 16-bit registers and three I/O ports, which include one write port and two read ports.

Digital Design using VHDL 62

Notes:

Each register is assigned a binary address as an identifier, and an external system uses the address to specify which register is to be involved in the operation. The storage and retrieval operations are known as the write and read operations respectively. A processor normally includes a register file as fast temporary storage.



Notes:

The data signals are labeled w_data, r_data0 and r_data1, and the port addresses are labeled w_addr, r_addr0 and r_addr1. There is also a control signal, wr_en, which is the write enable signal to indicate whether a write operation can be performed.

The design consists of three major parts: registers with enable signals, a write decoding circuit, and read multiplexing circuits. There are four 16-bit registers, each register with an individual enable signal, en. The en signal is synchronous and indicates whether the input data can be stored into the register.

The write decoding circuit examines the wr_en signal and decodes the write port address. If the wr_en signal is asserted, the decoding circuit functions as a regular 2-to-4 binary decoder that asserts one of the four en signals of the

corresponding register. The w_data signal will be sampled and stored into the corresponding register at the rising edge of the clock.

The read multiplexing circuit consists of two 4-to-1 multiplexers. It utilizes r_addr0 and r_addr1 as the selection signals to route the desired register outputs to the read ports.

Register File

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_file_wael is
port(
    clk, reset: in std_logic;
    wr_en: in std_logic;
    w_addr: in std_logic_vector(1 downto 0);
    w_data: in std_logic_vector(15 downto 0);
    r_addr0, r_addr1: in std_logic_vector(1 downto 0);
    r_data0, r_data1: out std_logic_vector(15 downto 0)
);
end reg_file_wael;

architecture no_loop_arch of reg_file_wael is
    signal r_reg0,r_reg1,r_reg2,r_reg3,r_next0,r_next1,r_next2,r_next3:
        std_logic_vector(15 downto 0);
    signal en: std_logic_vector(3 downto 0);
```

Digital Design using VHDL 64



Register File

```
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg3 <= (others=>'0');
      r_reg2 <= (others=>'0');
      r_reg1 <= (others=>'0');
      r_reg0 <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg3 <= r_next3;
      r_reg2 <= r_next2;
      r_reg1 <= r_next1;
      r_reg0 <= r_next0;
    end if;
  end process;
```

Digital Design using VHDL 65

Notes:

The register and corresponding enabling circuit are described by two processes. The decoding circuit is described in another process. The read ports are described as two multiplexers.



Register File

```
-- enable logic for register
process(r_reg0,r_reg1,r_reg2,r_reg3,en,w_data)
begin
    r_next3 <= r_reg3;
    r_next2 <= r_reg2;
    r_next1 <= r_reg1;
    r_next0 <= r_reg0;
    if en(3)='1' then
        r_next3 <= w_data;
    end if;
    if en(2)='1' then
        r_next2 <= w_data;
    end if;
    if en(1)='1' then
        r_next1 <= w_data;
    end if;
    if en(0)='1' then
        r_next0 <= w_data;
    end if;
end process;
```

Digital Design using VHDL

66

Notes:

If wr_en is '0' , en will be "0000" and no register will be updated. Otherwise, one bit of the en signal will asserted according to the value of the w_addr signal.



Register File

```
-- decoding for write address
process(wr_en,w_addr)
begin
  if (wr_en='0') then
    en <= (others=>'0');
  else
    case w_addr is
      when "00" => en <= "0001";
      when "01" => en <= "0010";
      when "10" => en <= "0100";
      when others => en <= "1000";
    end case;
  end if;
end process;
```

Digital Design using VHDL 67

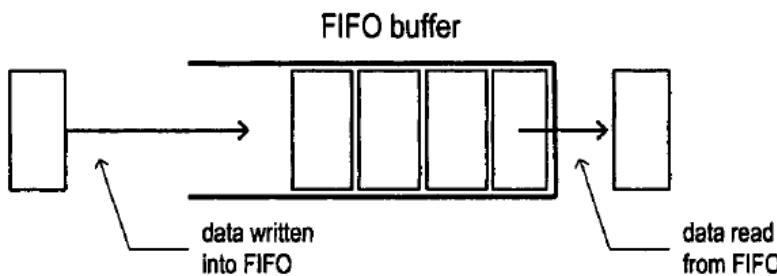
Register File

```
-- read multiplexing
with r_addr0 select
  r_data0 <= r_reg0 when "00",
              r_reg1 when "01",
              r_reg2 when "10",
              r_reg3 when others;
with r_addr1 select
  r_data1 <= r_reg0 when "00",
              r_reg1 when "01",
              r_reg2 when "10",
              r_reg3 when others;
end no_loop_arch;
```

Digital Design using VHDL 68

Register-based synchronous FIFO buffer

- A first-in-first-out (FIFO) buffer acts as "elastic" storage between two subsystems.



- One subsystem stores (i.e., writes) data into the buffer, and the other subsystem retrieves (i.e., reads) data from the buffer and removes it from the buffer.
- The order of data retrieval is same as the order of data being stored, and thus the buffer is known as a first-in-first-out buffer.

Digital Design using VHDL 69

Notes:

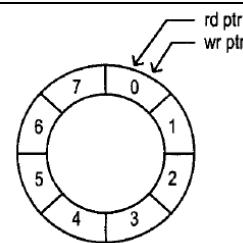
If two subsystems are synchronous (i.e., driven by the same clock), we need only one clock for the FIFO buffer and it is known as a synchronous FIFO buffer.

Register-based synchronous FIFO buffer

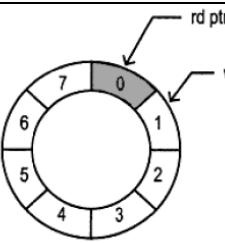
- The most common way to construct a FIFO buffer is to add a simple control circuit to a memory array, such as a register file or RAM.
- The memory array is arranged as a circular queue and use two pointers to mark the beginning and end of the FIFO buffer.
 - Write pointer: point to the empty slot before the head of the queue.
 - Read pointer: point to the tail of the queue.
- During a write operation, the data is stored in this designed slot, and the write pointer advances to the next slot (i.e., incremented by 1).
- During a read operation, the data is retrieved and the read pointer advances one slot, effectively releasing the slot for future write operations

Digital Design using VHDL 70

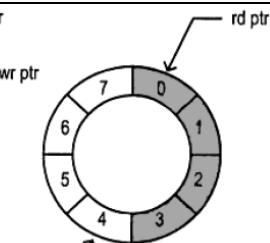
Registers



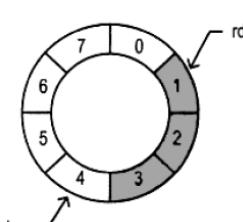
(a). initial (empty)



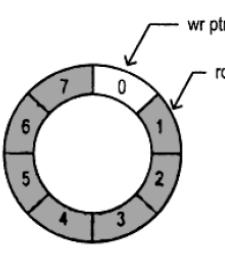
(b). after a write



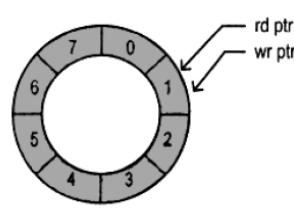
(c). 3 more writes



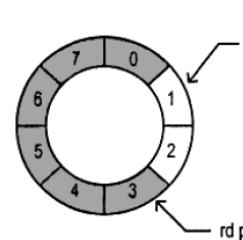
(d). after a read



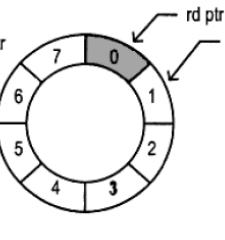
(e). 4 more writes



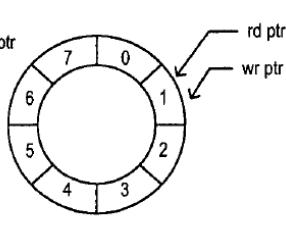
(f). 1 more write (full)



(g). 2 reads



(h). 5 more reads



(i). 1 more read (empty)

Buffer

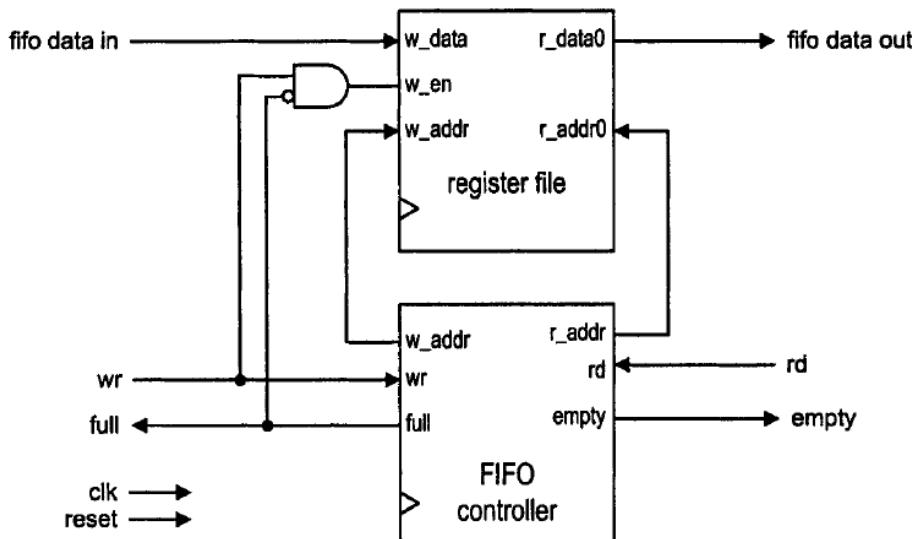
using VHDL 71

Notes:

Initially, both the read and write pointers point to the 0 address, as in Figure (a). Since the buffer is empty, no read operation is allowed at this time. After a write operation, the write pointer increments and the buffer contains one item in the 0 address, as in Figure (b). After a few more write operations, the write pointer continues to increase and the buffer expands accordingly, as in Figure (c). A read operation is performed afterward. The read pointer advances in the same direction, and the previous slot is released, as in Figure (d). After several more write operations, the buffer is full, as in Figure (f), and no write operation is allowed. Several read operations are then performed, and the buffer eventually shrinks to 0, as in Figures (g), (h) and (i).

Register-based synchronous FIFO buffer

- Block diagram of register-based FIFO



Digital Design using VHDL 72

Notes:

It consists of a register file and a control circuit, which generates proper read and write pointer values and status signals. Note that the FIFO buffer doesn't have any explicit external address signal. Instead, it utilizes two control signals, wr and rd, for write and read operations.



Register-based synchronous FIFO buffer

- At the rising edge of the clock, if the wr signal is '1' and the buffer is not full, the corresponding input data will be sampled and stored into the buffer.
- The output data from the FIFO is always available. The re signal might better be interpreted as a "remove" signal. If it is asserted at the rising edge and the buffer is not empty, the FIFO's read pointer advances one position and makes the current slot available. After the internal delays of the incrementing and routing, new output data is available in FIFO's output port.
- To ensure correct operation, a FIFO buffer must include the full and empty status signals for the case when the FIFO buffer is full or empty.

Digital Design using VHDL 73

Notes:

During FIFO operation, an overflow occurs when the external system attempts to write new data when the FIFO is full, and an underflow occurs when the external system attempts to read (i.e., remove) a slot when the FIFO is empty. To ensure correct operation, a FIFO buffer must include the full and empty status signals for the two special conditions. In a properly designed system, the external systems should check the status signals before attempting to access the FIFO.



Register-based synchronous FIFO buffer

- The major components of a FIFO control circuit are two counters, whose outputs function as write and read pointers respectively.
- During regular operation, the write counter advances one position when the wr signal is '1' at the rising edge of the clock, and the read counter advances one position when the re signal is '1' at the rising edge of the clock.
- There must be some safety precautions to ensure that data will not be written into a full buffer or removed from an empty buffer. Under these conditions, the counters will retain the previous values.

Digital Design using VHDL 74

Register-based synchronous FIFO buffer

- The difficult part of the control circuit is the handling of two special conditions in which the FIFO buffer is empty or full.
- When the FIFO buffer is empty, the read pointer is the same as the write pointer. Unfortunately, this is also the case when the FIFO buffer is full. Thus, we cannot just use read and write pointers to determine full and empty conditions.
- There are several methods to generate the status signals. We will consider only one method.
 - We will use the binary counters for the read and write pointers and increase their sizes by 1 bit. We can determine the full or empty condition by comparing the MSBs of the two pointers.



Register-based synchronous FIFO buffer

Ex: a FIFO with 3-bit address (i.e., 8 words)

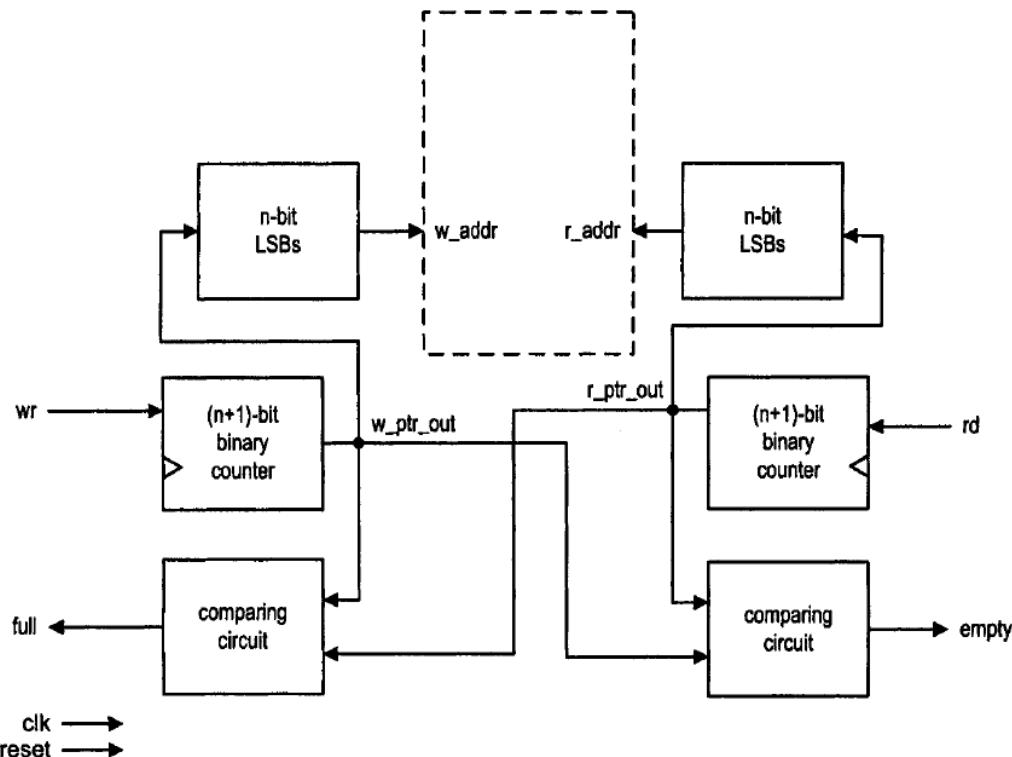
Write pointer	Read pointer	Operation	Status
0 000	0 000	initialization	empty
0 111	0 000	after 7 writes	
1 000	0 000	after 1 write	full
1 000	0 100	after 4 reads	
1 100	0 100	after 4 writes	full
1 100	1 011	after 7 reads	
1 100	1 100	after 1 read	empty
0 011	1 100	after 7 writes	
0 100	1 100	after 1 write	full
0 100	0 100	after 8 reads	empty

Digital Design using VHDL 76

Notes:

Two 4-bit counters will be used for the read and write pointers. The three LSBs of the read and write pointers are used as addresses to access the register file and wrap around after eight increments. They are equal when the FIFO is empty or full. The MSBs of the read and write pointers can be used to distinguish the two conditions. The two bits are the same when the FIFO is empty. After eight write operations, the MSB of the write pointers flips and becomes the opposite of the MSB of the read pointer. The opposite values in MSBs indicate that the FIFO is full. After eight read operations, the MSB of the read pointer flips and becomes identical to the MSB of the write pointer, which indicates that the FIFO is empty again.

Register-based synchronous FIFO buffer



'DL 77

Register-based synchronous FIFO buffer

➤ The VHDL code of a 4-word FIFO controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fifo_sync_ctrl4_wael is
port(
    clk, reset: in std_logic;
    wr, rd: in std_logic;
    full, empty: out std_logic;
    w_addr, r_addr: out std_logic_vector(1 downto 0)
);
end fifo_sync_ctrl4_wael;

architecture enlarged_bin_arch of fifo_sync_ctrl4_wael is
constant N: integer:=2;
signal w_ptr_reg, w_ptr_next: unsigned(N downto 0); --i.e. 2 downto 0 (3 bits)
signal r_ptr_reg, r_ptr_next: unsigned(N downto 0);
signal full_flag, empty_flag: std_logic;
```

3

Notes:

A constant, N, is used inside the architecture body to indicate the number of address bits. Note that the w_ptr_reg and r_ptr_reg signals, which are the write and read pointers, are increased to N + 1 bits.

Register-based synchronous FIFO buffer

```
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      w_ptr_reg <= (others=>'0');
      r_ptr_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      w_ptr_reg <= w_ptr_next;
      r_ptr_reg <= r_ptr_next;
    end if;
  end process;
```

Digital Design using VHDL 79

Register-based synchronous FIFO buffer

```
-- write pointer next-state logic
w_ptr_next <=
  w_ptr_reg + 1 when wr='1' and full_flag='0' else
  w_ptr_reg;

full_flag <=
  '1' when r_ptr_reg(N) /=w_ptr_reg(N) and
    r_ptr_reg(N-1 downto 0)=w_ptr_reg(N-1 downto 0)
  else
  '0';

-- write port output
w_addr <= std_logic_vector(w_ptr_reg(N-1 downto 0));
full <= full_flag;
```

Digital Design using VHDL 80

Register-based synchronous FIFO buffer

```
-- read pointer next-state logic
r_ptr_next <=
    r_ptr_reg + 1 when rd='1' and empty_flag='0'  else
    r_ptr_reg;

empty_flag <= '1' when r_ptr_reg=w_ptr_reg else
    '0';

-- read port output
r_addr <= std_logic_vector(r_ptr_reg(N-1 downto 0));
empty <= empty_flag;

end enlarged_bin_arch;
```

Digital Design using VHDL 81

Home Work

1. Add the 4X16 register file to the 4-word FIFO controller to form a complete register-based FIFO buffer.
2. The PWM circuit can control the duty cycle, but its frequency is fixed. If the original frequency of the clock signal is f_{clk} , the frequency of the PWM circuit in this lecture is $\frac{f_{clk}}{2^4}$. We can extend the PWM circuit to a programmable pulse generator by adding additional control signal to specify the desired frequency. Let k be a 4-bit signal that is interpreted as an unsigned divisor. The frequency of the new output pulse will be $\frac{f_{clk}}{k * 2^4}$ if k is not 0 and will be $\frac{f_{clk}}{2^4}$ if k is 0.

Design this circuit and derive the VHDL code.

Home Work

3. A stack is a buffer in which the data is stored and retrieved first-in-last-out fashion.

In a synchronous stack, it should consist of the following I/O signals:

- w_data and r_data: data to be written (also known as pushed) into and read (also known as popped) from the stack.
- push and pop: control signals to enable the push or pop operation.
- full and empty: status signals.
- clk and reset: the clock and reset signals.



Home Work

3.

We can use a register file to construct this circuit.

- (a) Draw a top-level diagram.
- (b) Consider a stack of four words. Derive the VHDL code of the control circuit. It has 4 inputs: clk, reset, push, pop. It has 2 outputs: empty, full. It has also an output signal stack pointer (sp) that points to the top of the stack (i.e., the available slot to be used in the next push operation). It is used to determine the register file address for reading and writing .

Session 5

FINITE STATE MACHINE

Digital Design using VHDL 2



Topics for this Lecture

- Overview of FSM
- FSM Representation
- State Diagram & ASM Chart
- Operation of a synchronous FSM
- Moore Machine vs Mealy Machine
- VHDL Description of an FSM
- State Assignment
- Design Examples

Digital Design using VHDL 3



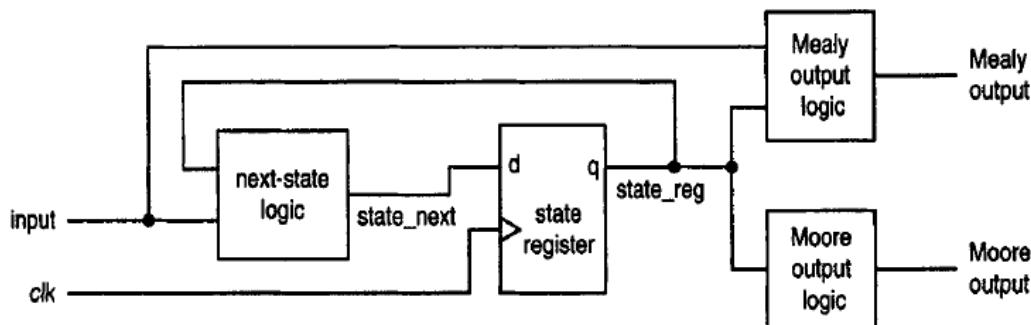
Overview of FSM

- A finite state machine (FSM) is a sequential circuit with “random” next-state logic.
- Unlike the regular sequential circuit, the state transitions and event sequence of an FSM do not exhibit a simple pattern.
- The derivation of an FSM starts with a more abstract model, such as a state diagram or an algorithm state machine (ASM) chart.
- Our emphasis is on the application of an FSM as the control circuit for a large, complex system, and our discussion focuses on the issues related to this aspect.
- As in previous lecture, our discussion is limited to the synchronous FSM, in which the state register is controlled by a single global clock.

Digital Design using VHDL 4

Overview of FSM

- The main application of an FSM is to realize operations that are performed in a sequence of steps.
- An FSM can function as the control circuit that coordinates and governs the operations of other units.



Digital Design using VHDL 5

Notes:

Formally, an FSM is specified by five entities: symbolic states, input signals, output signals, next-state function and output function. A state specifies a unique internal condition of a system. As time progresses, the FSM transits from one state to another. The new state is determined by the next-state function, which is a function of the current state and input signals. In a synchronous FSM, the transition is controlled by a clock signal and can occur only at the triggering edge of the clock. The output function specifies the value of the output signals. **If it is a function of the state only, the output is known as a Moore output.** On the other hand, **if it is a function of the state and input signals, the output is known as a Mealy output.** An FSM is called a Moore machine or Mealy machine if it contains only Moore outputs or Mealy outputs respectively. A complex FSM normally has both types of outputs.

The block diagram of an FSM is similar to the block diagram of a regular sequential circuit. The state register is the memory element that stores the state of the FSM. It is synchronized by a global clock. The next-state logic implements the next-state function, whose input is the current state and input signals. The output logic implements the output function. This diagram includes both Moore output logic, whose input is the current state, and Mealy output logic, whose input is the current state and input signals.

The main application of an FSM is to realize operations that are performed in a sequence of steps. A large digital system usually involves complex tasks or algorithms, which can be expressed as a sequence of actions based on system status and external commands. An FSM can function as the control circuit (known as the control path) that coordinates and governs the operations of other units (known as the data path) of the system. FSMs can also be used in many simple tasks, such as detecting a unique pattern from an input data stream or generating a specific sequence of output values.



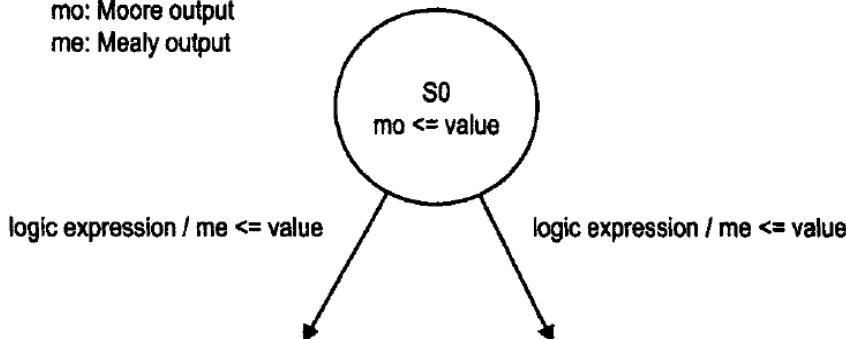
FSM REPRESENTATION

- The design of an FSM normally starts with an abstract, graphic description, such as a state diagram or an ASM chart.
- Both descriptions utilize symbolic state notations, show the transition among the states and indicate the output values under various conditions.
- A state diagram or an ASM chart can capture all the needed information (i.e., state, input, output, next-state function, and output function) in a single graph.

1. State diagram

- A state diagram consists of nodes, which are drawn as circles and one-direction transition arcs.
- A node represents a unique state of the FSM and an arc represents a transition from one state to another and is labeled with the condition that will cause the transition.

mo: Moore output
me: Mealy output



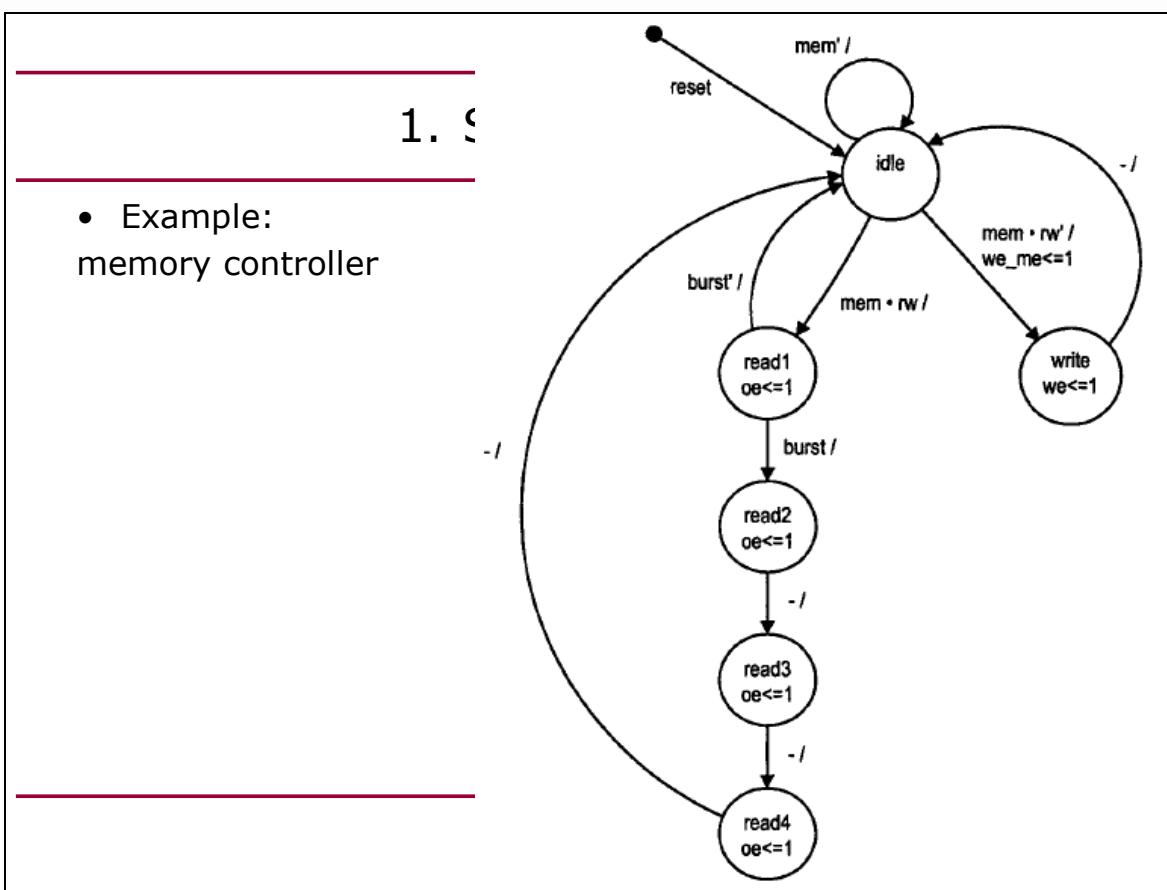
Digital Design using VHDL 7

Notes:

A state diagram consists of nodes, which are drawn as circles (also known as bubbles), and one-direction transition arcs.

A node represents a unique state of the FSM and it has a unique symbolic name. An arc represents a transition from one state to another and is labeled with the condition that will cause the transition. The condition is expressed as a logic expression composed of input signals. An arc will be taken when the corresponding logic expression is evaluated to be logic '1'.

The output values are also specified on the state diagram. The Moore output is a function of state and thus is naturally placed inside the state bubble. On the other hand, the Mealy output depends on both state and input and thus is placed under the condition expression of the transition arcs. To reduce the assignments, we list only the output signals that are activated or asserted. An output signal will assume the default, unasserted value ('0') if it is not listed inside the state bubble or under the logic expression of an arc.



Notes:

The controller is between a processor and a memory chip, interpreting commands from the processor and then generating a control sequence accordingly. **The commands, mem, rw and burst**, from the processor constitute the input signals of the FSM. **The mem signal** is asserted to high when a memory access is required. **The rw signal** indicates the type of memory access, and its value can be either '1' or '0', for memory read and memory write respectively. **The burst signal** is for a special mode of a memory read operation. If it is asserted, four consecutive read operations will be performed. The memory chip has two control signals, oe (for output enable) and we (for write enable), which need to be asserted during the memory read and memory write respectively. The two output signals of the FSM, oe and we, are connected to the memory chip's control signals. For comparison purpose, we also add an artificial Mealy output signal, we_me, to the state diagram.



Initially, the FSM is in the idle state, waiting for the mem command from the processor. Once mem is asserted, the FSM examines the value of rw and moves to either the read1 state or the write state. These input conditions can be formalized to logic expressions, as shown in the transition arcs from the idle state:

- mem': represents that no memory operation is required.
- mem . rw: represents that a memory read operation is required.
- mem . rw': represents that a memory write operation is required.

The results of these logic expressions are checked at the rising edge of the clock. If the mem' expression is true (i.e., mem is '0'), the FSM stays in the idle state. If the mem.rw expression is true (i.e., both mem and rw are '1'), the FSM moves to the read1 state. Once it is there, the oe signal is activated, as indicated in the state bubble. On the other hand, if the mem . rw' expression is true (i.e., mem is '1' and rw is '0'), the FSM moves to the write state and activates the we signal.

After the FSM reaches the read1 state, the burst signal is examined at the next rising edge of the clock. If it is '1', the FSM will go through read2, read3 and read4 states in the next three clock cycles and then return to the idle state. Otherwise, the FSM returns to the idle state. We use the notation “-” to represent the “always true” condition.

After the FSM reaches the write state, it will return to the idle state at the next rising edge of the clock.

The we_me signal is asserted only when the FSM is in the idle state and the mem . rw' expression is true. It will be deactivated when the FSM moves away from the idle state (i.e., to the write state). It is a Mealy output since its value depends on the state and the input signals (i.e., mem and rw).

In practice, we usually want to force an FSM into an initial state during system initialization. This is frequently done by an asynchronous reset signal, similar to the asynchronous reset signal used in a register of a regular sequential circuit. Sometimes a solid dot is used to indicate this transition. This transition is only for system initialization and has no effect on normal FSM operation.

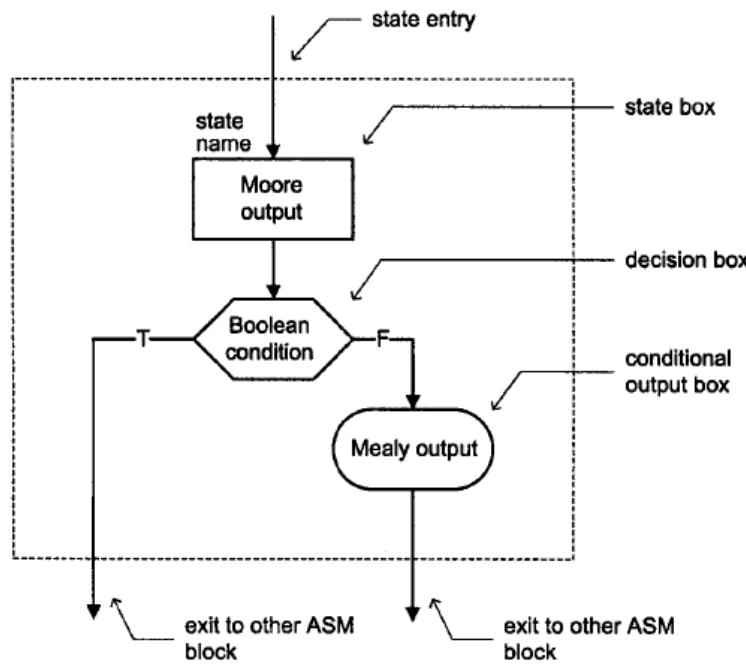


2. ASM Chart

- An algorithmic state machine (ASM) chart is an alternative method for representing an FSM.
- It contains the same amount of information as a state diagram but it is more descriptive.
- So, we can use an ASM chart to specify the complex sequencing of events involving commands (input) and actions (output).
- An ASM chart is constructed of a network of ASM blocks.
- An ASM block consists of one state box and an optional network of decision boxes and conditional output boxes.

2. ASM Chart

- ASM Block:



Notes:

The state box, as its name indicates, represents a state in an FSM. It is identified by a symbolic state name on the top left corner of the state box. The action or output listed inside the box describes the desired output signal values when the FSM enters this state. Since the outputs rely on the state only, they correspond to the Moore outputs of the FSM. We list only signals that are activated or asserted. An output signal will assume the default, unasserted value if it is not listed inside the box.

A decision box tests an input condition to determine the exit path of the current ASM block. It contains a Boolean expression composed of input signals and plays a similar role to the logic expression in the transition arc of a state diagram. Because of the flexibility of the Boolean expression, it can describe more complex conditions.

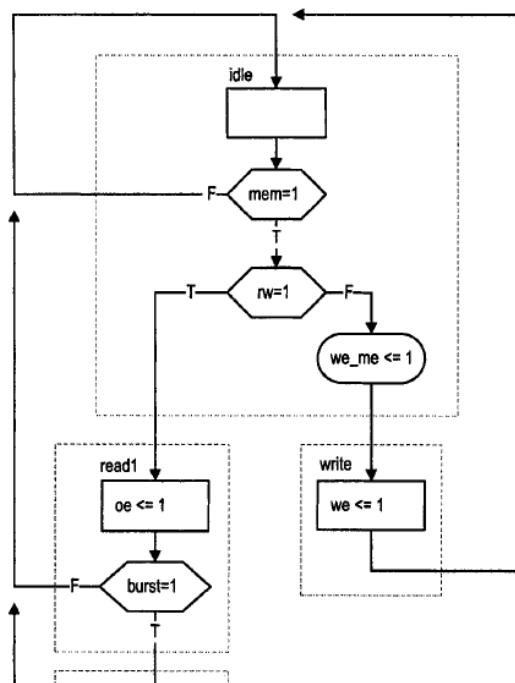
Depending on the value of the Boolean expression, the FSM can follow either the true path or the false path, which are labeled as T or F in the exit paths of

the decision box. If necessary, we can cascade multiple decision boxes inside an ASM block to describe a complex condition.

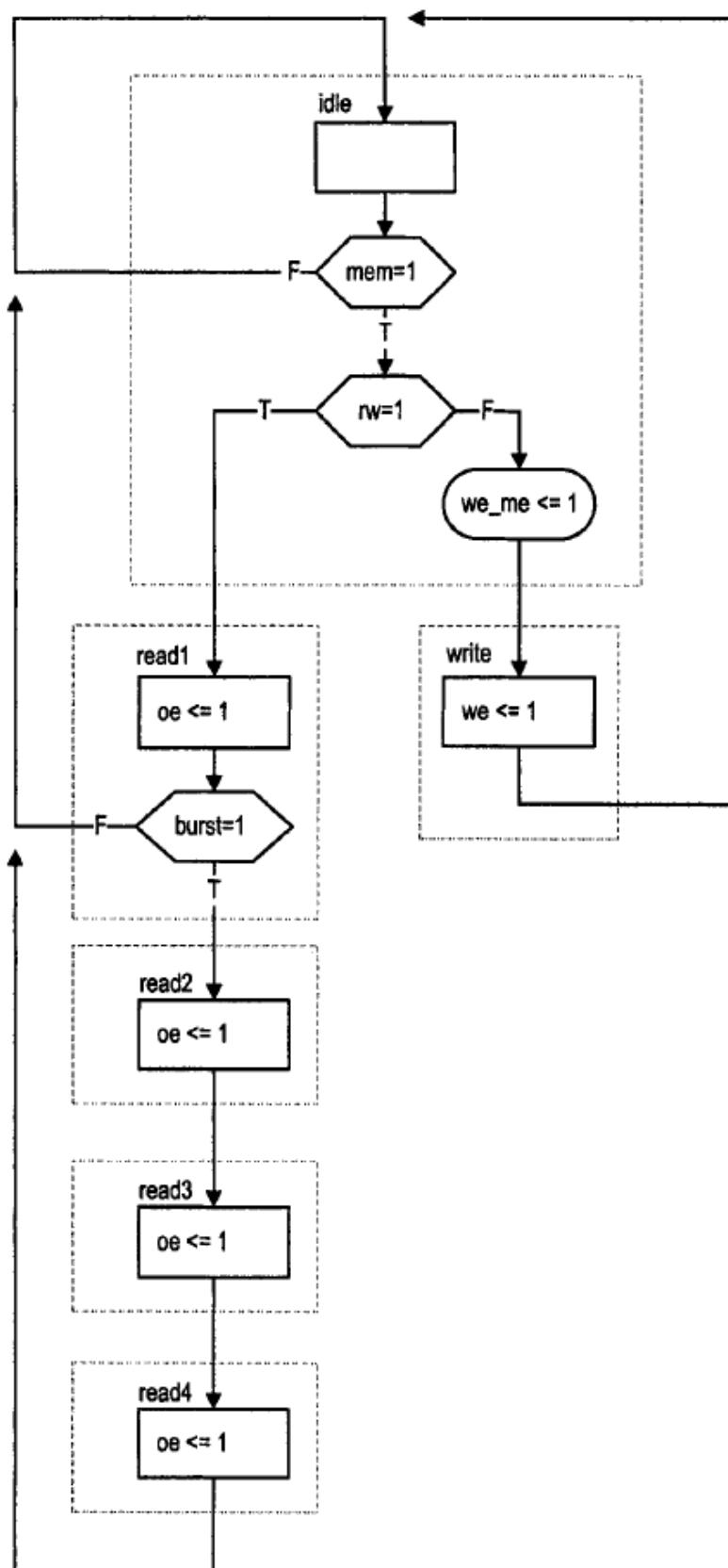
A conditional output box also lists asserted output signals. However, it can only be placed after an exit path of a decision box. It implies that these output signals can be asserted only if the condition of the previous decision box is met. Since the condition is composed of a Boolean expression of input signals, these output signals' values depend on the current state and input signals, and thus they are Mealy outputs. The output signal assumes the default, unasserted value when there is no conditional output box.

2. ASM Chart

- Example:
Memory Controller

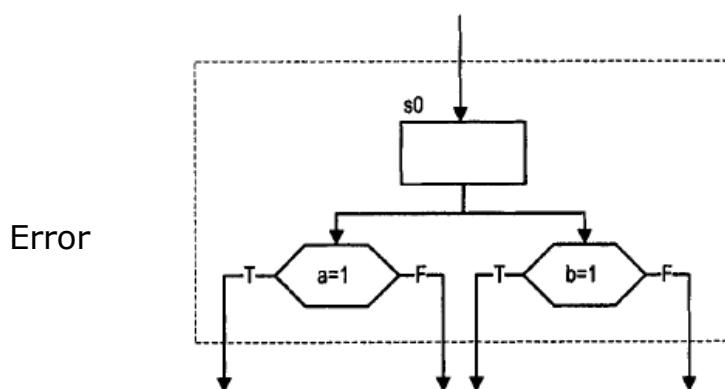


ng VHDL 11



2. ASM Chart

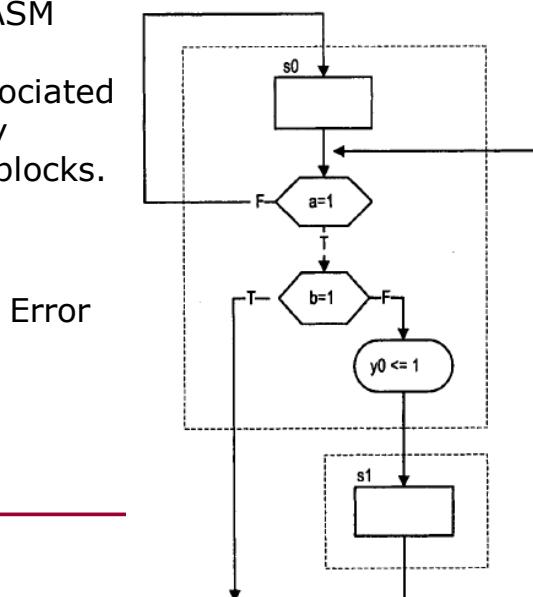
- Since an ASM chart is used to model an FSM, two rules apply:
 1. For a given input combination, there is one unique exit path from the current ASM block.



Digital Design using VHDL 12

2. ASM Chart

2. The exit path of an ASM block must always lead to a state box. The state box can be the state box of the current ASM block or a state box of another ASM block. The decision boxes and conditional output boxes are associated with a single ASM block and they cannot be shared by other ASM blocks.



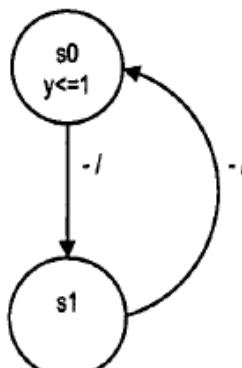


State Diagram & ASM Chart

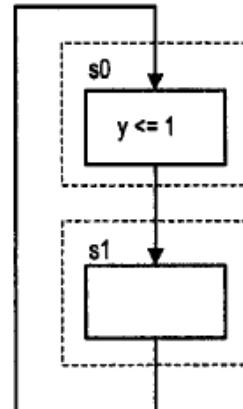
- ASM chart can accommodate the complex conditions involved in state transitions and Mealy outputs while a state diagram is preferred for an FSM with simple, straightforward state transitions.
- An ASM chart can be converted to a state diagram and vice versa.
- An ASM block corresponds to a state and its transition arcs of a state diagram.
- The key for the conversion is the transformation between the logic expressions of the transition arcs in a state diagram and the decision boxes in an ASM chart.
- This can be shown by the following examples.

State Diagram & ASM Chart

- Example 1:



(a)



(b)

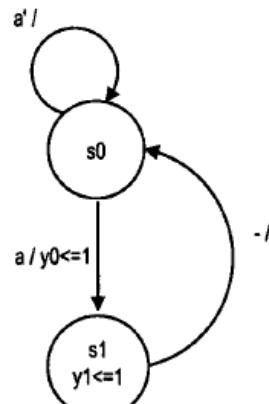
Digital Design using VHDL 15

Notes:

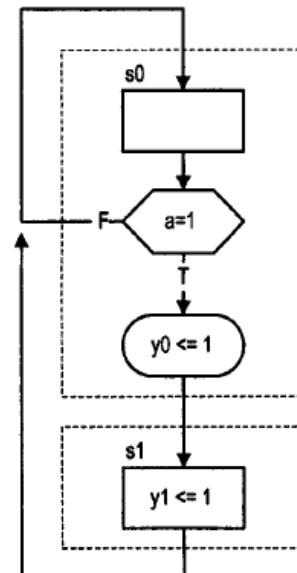
It is an FSM with no branching arches. The state diagram and the ASM chart are almost identical.

State Diagram & ASM Chart

- Example 2:



(a)

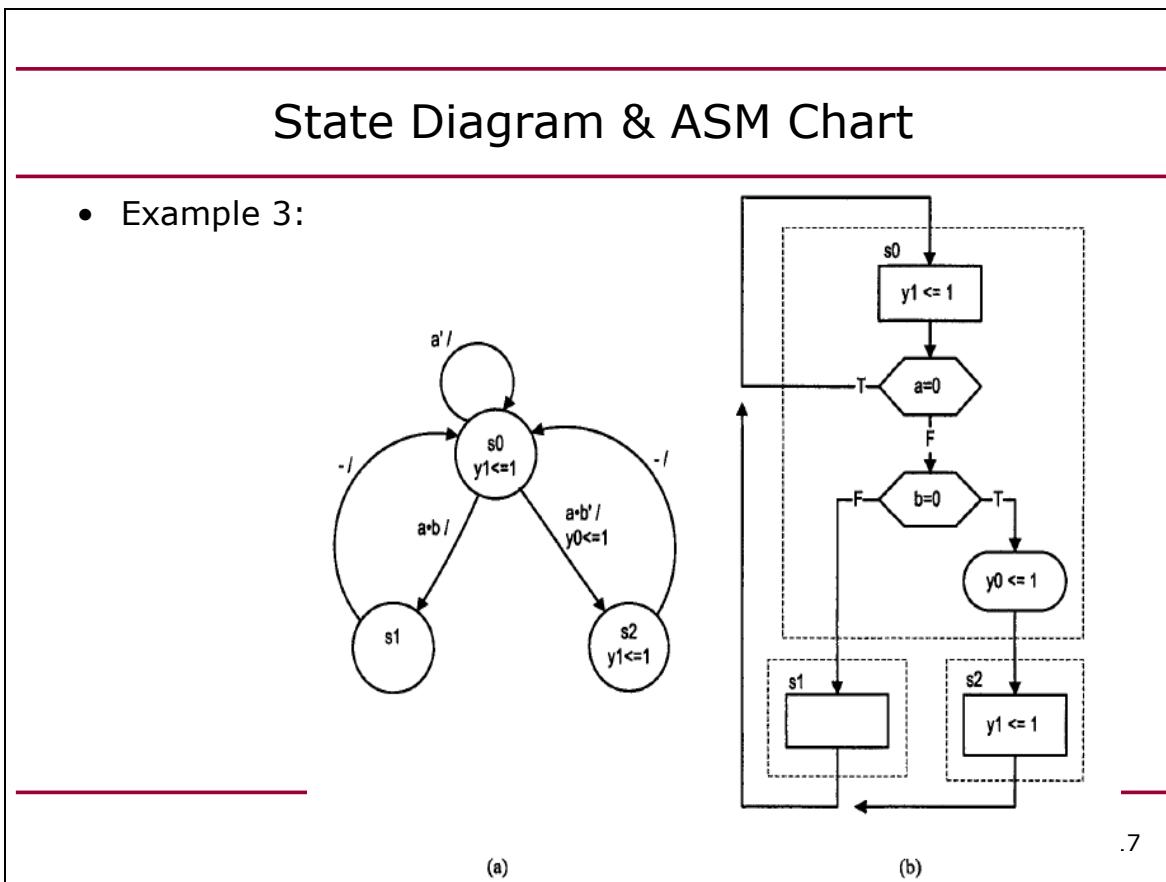


(b)

L6

Notes:

The FSM has two transition arcs from the S0 state and has a Mealy output, y. The logic expressions a and a' of the transition arches are translated into a decision box with Boolean expression $a = 1$. Note that the two states are transformed into two ASM blocks. The decision and conditional output boxes are not new states, just actions associated with the ASM block S0.

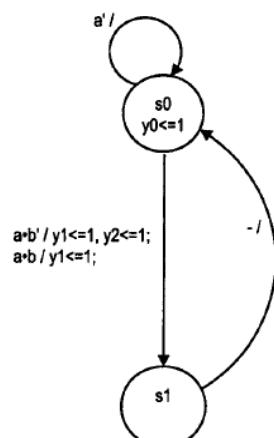


Notes:

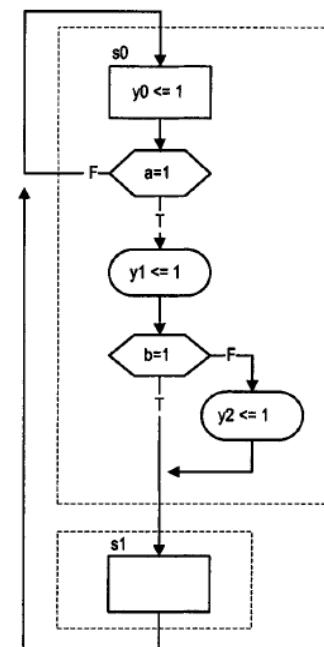
The transitions from the S0 state are more involved. We can translate the logic expressions a' and $a \cdot b'$ directly into two decision boxes of conditions $a = 0$ and $(a = 1 \text{ and } b = 0)$. However, closer examination shows that the second decision box is on the false path of the first decision box, which implies that a is '1'. Thus, we can eliminate the $a = 1$ condition from the second decision box and make the decision simpler and more descriptive.

State Diagram & ASM Chart

- Example 4:



(a)



(b)

18

Notes:

The output of the FSM is more complex and depends on various input conditions. The state diagram needs multiple logic expressions in the transition arc to express various input conditions. The ASM chart can accommodate the situation and is more descriptive than the state diagram.

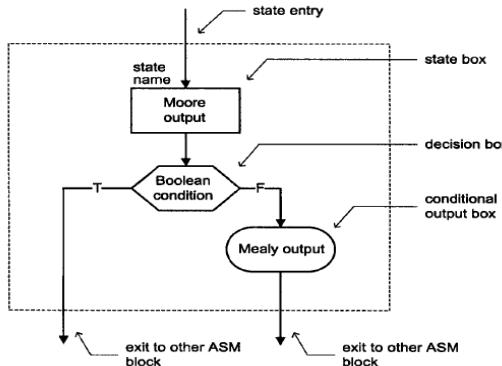


Operation of a synchronous FSM

- In a synchronous FSM, the state transition is controlled by the rising edge of the system clock.
- Mealy output and Moore output are not directly related to the clock but are responding to input or state change.
- A Moore output depends only on the state so its transition is indirectly synchronized by the clock.
- The timing of a synchronous FSM can best be explained by examining the operation of an ASM block. The transition can be interpreted as follows:
 1. At the rising edge of the clock, the FSM enters a new state (and thus a new ASM block).

Operation of a synchronous FSM

2. During the clock period, the FSM activates the Moore output signals asserted in this state. It also evaluates various Boolean expressions of the decision boxes and activates the Mealy output signals accordingly.
3. At the next rising edge of the clock (which is the end of the current clock period), the results of Boolean expressions are examined simultaneously, an exit path is determined, and the FSM enters the designated new ASM block.



Design using VHDL 20



Moore Machine vs Mealy Machine

- Theoretically , a Moore machine and a Mealy machine are considered to have similar computation capability, although a Mealy machine accomplishes the same task with fewer states.
- When the FSM is used as a control circuit, the control signals generated by a Moore machine and a Mealy machine have different timing characteristics.
This can be shown in the following example of an edge detection circuit.



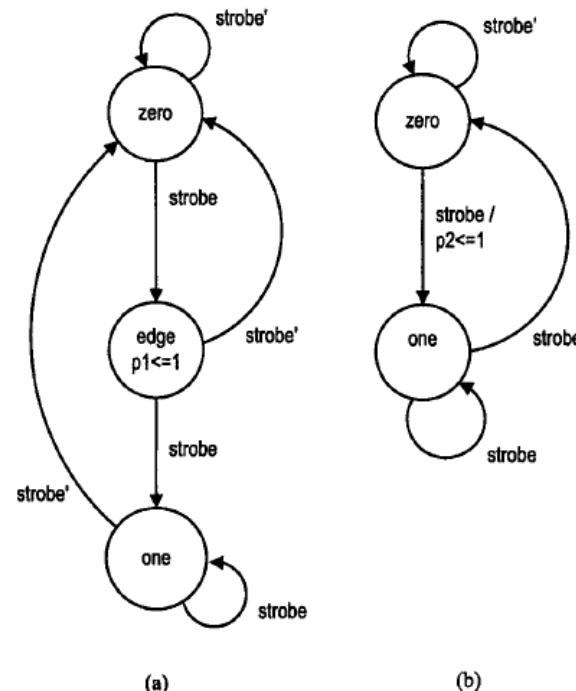
Moore Machine vs Mealy Machine

- Example: Edge Detection Circuit
 - An edge detection circuit is used to detect the rising edge of a slowly varying input strobe signal .
 - It generates a short pulse when the strobe signal changes from '0' to '1'. The width of the output pulse is about the same or less than a clock period of the FSM.
 - We will focus on the specification of the width and timing of the output pulse to show the difference between a Mealy machine and a Moore machine.

Moore Machine vs Mealy Machine

- Example:
Edge Detection Circuit

- (a) Moore Machine.
(b) Mealy Machine.



'3

Notes:

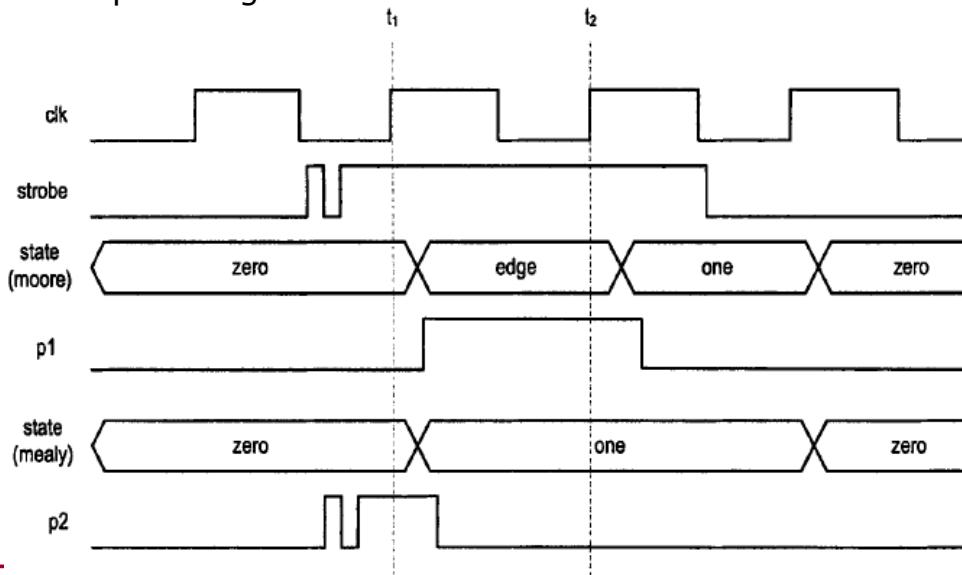
The basic design idea is to construct an FSM that has a zero state and a one state, which represent that the input has been '0' or '1' for a long period of time respectively. The FSM has a single input signal, strobe, and a single output signal. The output will be asserted momentarily when the FSM transits from the zero state to the one state.

The first design is based on a Moore machine. There are three states. In addition to the zero and one states, the FSM also has an edge state. When strobe becomes '1' in the zero state, it implies that strobe changes from '0' to '1'. The FSM moves to the edge state, in which the output signal, p_1 , is asserted. In normal operation, strobe should continue to be '1' and the FSM moves to the one state at the next rising edge of the clock and stays there until strobe returns to '0'. If strobe is really short and changes to '0' in the edge state, the FSM will return to the zero state.

The second design is based on a Mealy machine. It consists of only the zero and one states. When strobe changes from '0' to '1' in the zero state, the FSM moves to the one state. From the state diagram, it seems that the output signal, p2, is asserted when the FSM transit from the zero state to the one state. Actually, p2 is asserted in the zero state whenever strobe is '1'. When the FSM moves to the one state, p2 will be de-asserted.

Moore Machine vs Mealy Machine

- Example: Edge Detection Circuit



Digital Design using VHDL 24



Moore Machine vs Mealy Machine

There are three major differences between the Moore machine and Mealy machine-based designs:

- First, a Mealy machine normally requires fewer states to perform the same task.
- Second, a Mealy machine can generate a faster response.
- Third, the control of the width and timing of the output signal. In a Mealy machine, the width of an output signal is determined by the input signal and can be very narrow, while the output of a Moore machine is synchronized with the clock edge and its width is about the same as a clock period. The output of a Mealy machine is therefore susceptible to glitches from the input signal.

Digital Design using VHDL 25

Notes:

There are three major differences between the Moore machine and Mealy machine-based designs.

- First, a Mealy machine normally requires fewer states to perform the same task. This is due to the fact that its output is a function of states and external inputs, and thus several possible output values can be specified in one state. For example, in the zero state of the second design, p2 can be either '0' or '1', depending on the value of strobe. Thus, the Mealy machine-based design requires only two states whereas the Moore machine-based design requires three states.
- Second, a Mealy machine can generate a faster response. Since a Mealy output is a function of input, it changes whenever the input meets the designated condition. For example, in Mealy machine-based design, if the FSM



is in the zero state, p2 is asserted immediately after strobe changes from '0' to '1'. On the other hand, a Moore machine reacts indirectly to input changes.

The Moore machine-based design also senses the changes of strobe in the zero state. However, it has to wait until the next state (i.e., the edge state) to respond. The change causes the FSM to move to the edge state. At the next rising edge of the clock, the FSM moves to this state and p1 responds accordingly. In a synchronous system, the distinction between a Mealy output and a Moore output normally means a delay of one clock cycle. Recall that the input signal of a synchronous system is sampled only at the rising edge of the clock. Let us assume that the output of the edge detection circuit is used by another synchronous system. The p2 can be sampled at t1. However, the p1 signal is not available at that time because of the clock-to-q delay and output logic delay. Its value can be sampled only by the next rising edge at t2 .

- The third difference involves the control of the width and timing of the output signal. In a Mealy machine, the width of an output signal is determined by the input signal. The output signal is activated when the input signal meets the designated condition and is normally deactivated when the FSM enters a new state. Thus, its width varies with input and can be very narrow. Also, a Mealy machine is susceptible to glitches in the input signal and passes these undesired disturbances to the output. On the other hand, the output of a Moore machine is synchronized with the clock edge and its width is about the same as a clock period. It is not susceptible to glitches from the input signal.

As mentioned earlier, our focus on FSM is primarily on its application as a control circuit. From this perspective, selection between a Mealy machine and a Moore machine depends on the need of control signals. We can divide control signals into two categories: edge-sensitive and level-sensitive.

-An edge-sensitive control signal is used as input for a sequential circuit synchronized by the same clock. A simple example is the enable signal of a counter. Since the signal is sampled only at the rising edge of the clock, the width of the signal and the existence of glitches do not matter as long as it is stable during the setup and hold times of the clock edge. Both the Mealy and the Moore machines can generate output signals that meet this requirement.

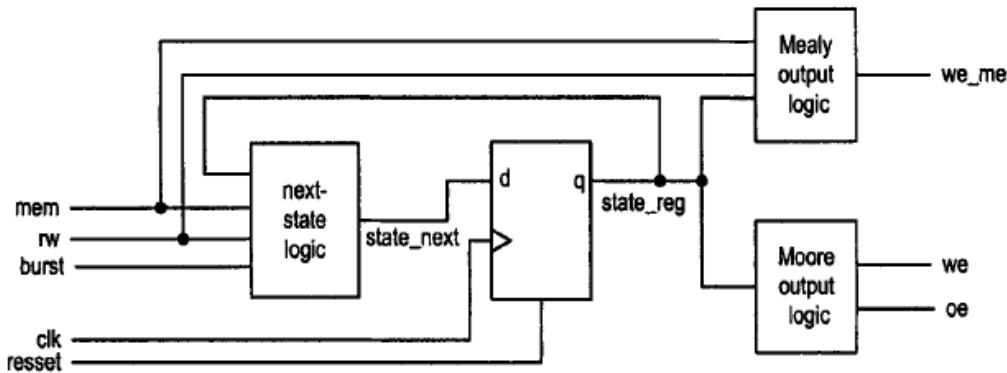
However, a Mealy machine is preferred since it uses fewer states and responds one clock faster than does a Moore machine.

- A level-sensitive control signal means that a signal has to be asserted for a certain amount of time. When asserted, it has to be stable and free of glitches.

A good example is the write enable signal of an SRAM chip. A Moore machine is preferred since it can accurately control the activation time of its output, and can shield the control signal from input glitches.

VHDL Description of an FSM

- Two coding styles are mainly used:
 - Multi-segment coding style.
 - Two-segment coding style.
- We will consider the memory controller example.



Digital Design using VHDL 26

Notes:

The block diagram of an FSM is similar to that of the regular sequential circuit. Thus, derivation of VHDL code for an FSM is similar to derivation for a regular sequential circuit. We first identify and separate the memory elements and then derive the next-state logic and output logic.

There are two differences in the derivation. The first is that symbolic states are used in an FSM description. To capture this kind of representation, we utilize VHDL's enumeration data type for the state registers. The second difference is in the derivation of the next-state logic. Instead of using a regular combinational circuit, such as an incrementor or shifter, we have to construct the code according to a state diagram or ASM chart.

VHDL Description of an FSM

1. Multi-segment coding style.

```
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
  port(
    clk, reset: in std_logic;
    mem, rw, burst: in std_logic;
    oe, we, we_me: out std_logic
  );
end mem_ctrl;

architecture mult_seg_arch of mem_ctrl is
  type mc_state_type is
    (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: mc_state_type;
begin
```

Digital Design using VHDL 27

Notes:

Inside the architecture declaration, we use the VHDL's enumeration data type. The data type is declared as:

type mc_state_type is (idle, read1, read2, read3, read4, write);

The syntax of the enumeration data type statement is very simple:

type type_name is (list_of_all_possible_values);

It simply enumerates all possible values in a list. In this particular example, we list all the symbolic state names.

The next statement then uses this newly defined type as the data type for the state register's input and output:

signal state_reg, state_next: mc_state_type;

VHDL Description of an FSM

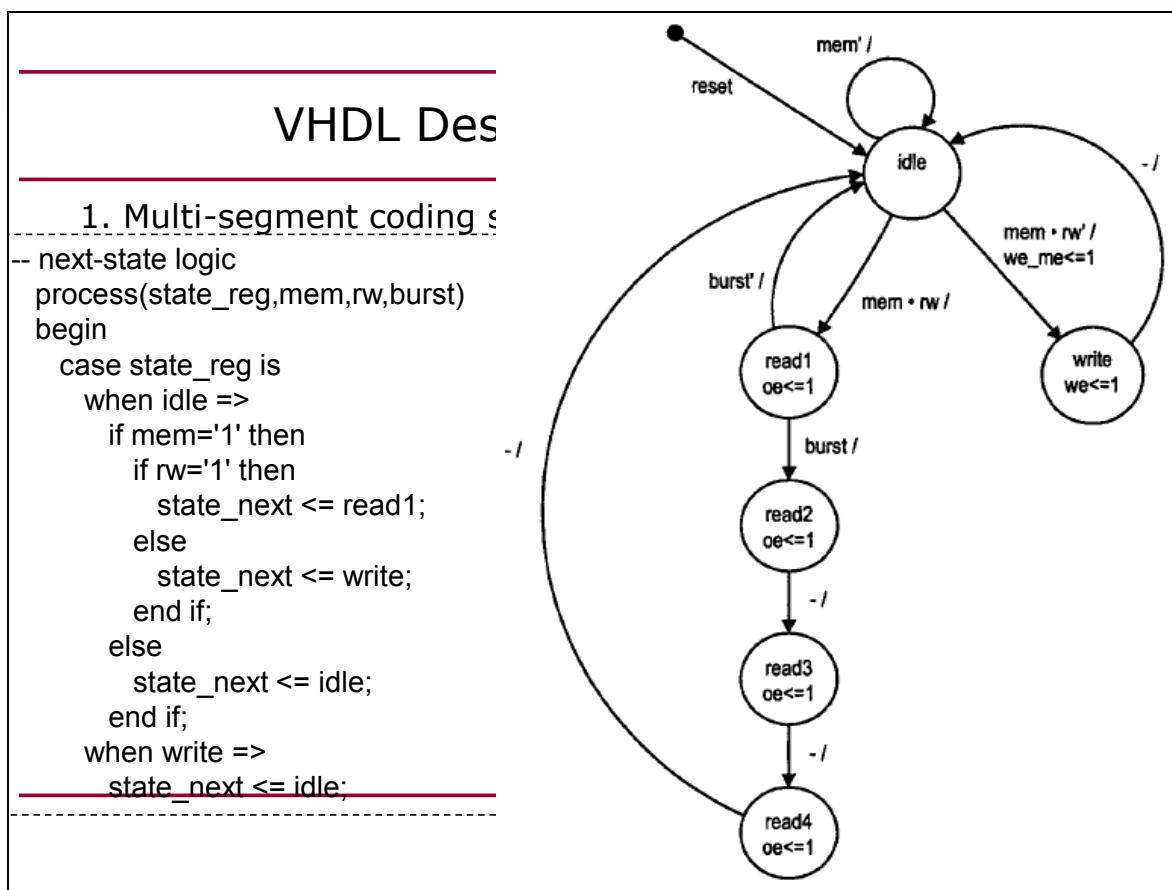
1. Multi-segment coding style.

```
-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= idle;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
```

Digital Design using VHDL 28

Notes:

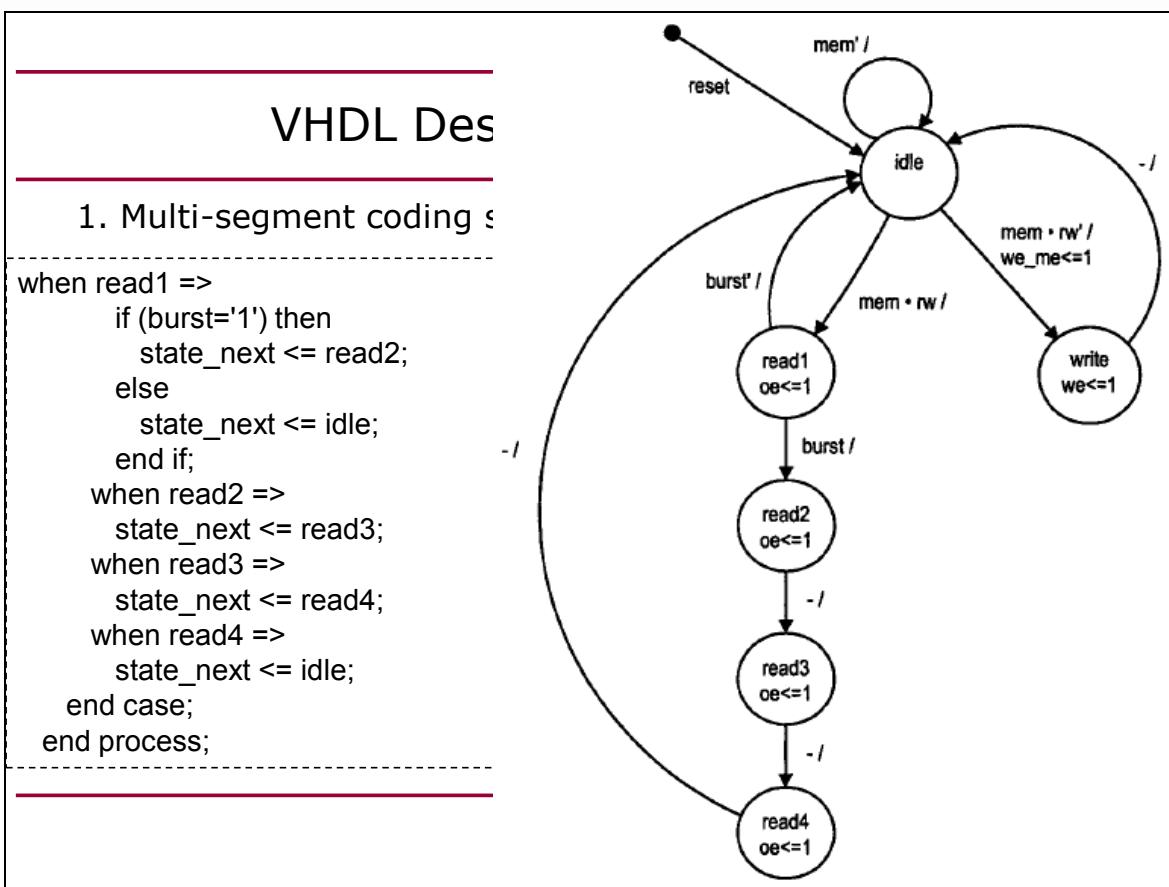
The architecture body is divided into four code segments. The first segment is for the state register. Its code is like that of a regular register except that a user-defined data type is used for the signal. We use an asynchronous reset signal for initialization. The state register is cleared to the idle state when the reset signal is asserted.



Notes:

The second code segment is for the next-state logic and is the key part of the FSM description. It is patterned after the ASM chart or the state diagram. We use a case statement with `state_reg` as the selection expression. The `state_reg` signal is the output of the state register and represents the current state of the FSM. Based on its value and input signal, the next state, denoted by the `state_next` signal, can be determined. The next state will be stored into the state register and becomes the new state at the rising edge of the clock. The `state_next` signal can be derived directly from the ASM block or the state diagram.

For a block with multiple exit paths, we can use if statements to code the decision boxes of the ASM chart or the logic expression on the arc of the state diagram. It can be directly translated to the Boolean expression of the if statement, and the two exit paths can be expressed as the then branch and the else branch of the if statement.



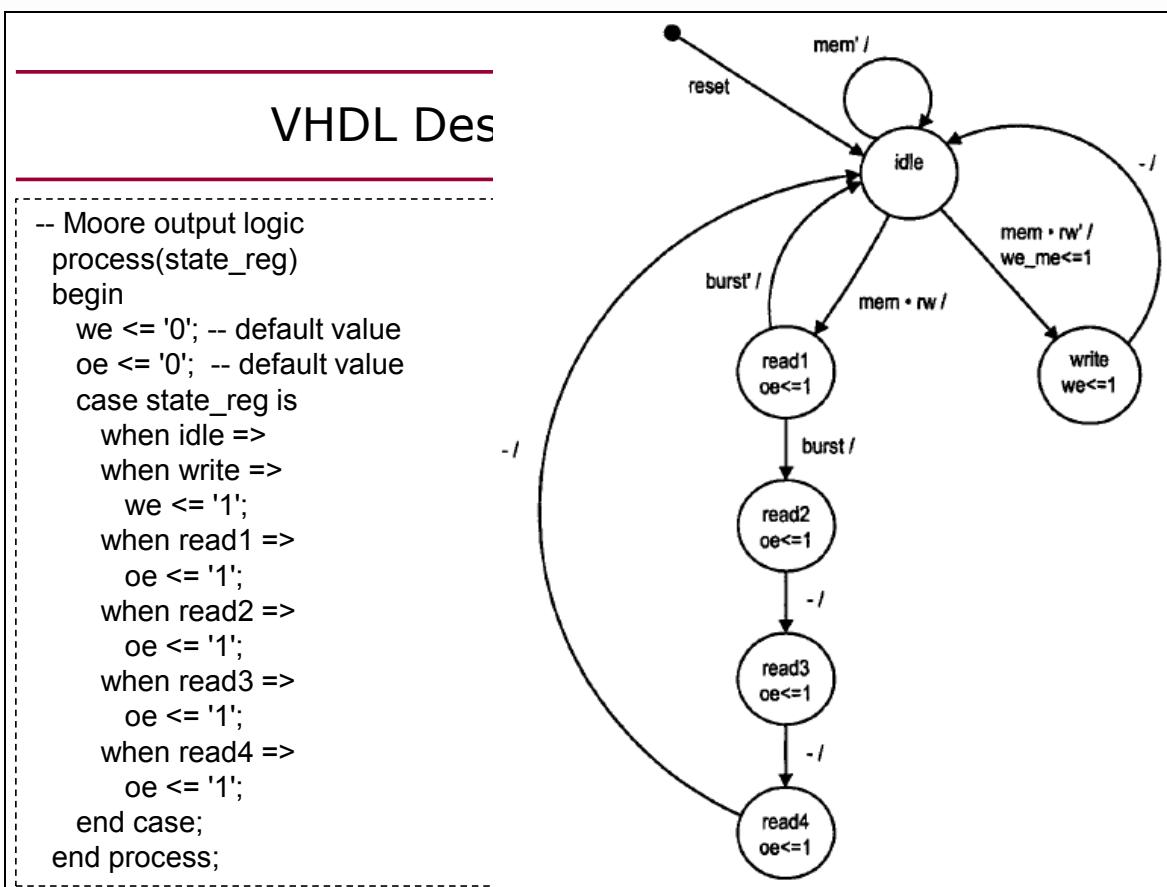
VHDL Des

1. Multi-segment coding s

```

when read1 =>
  if (burst='1') then
    state_next <= read2;
  else
    state_next <= idle;
  end if;
when read2 =>
  state_next <= read3;
when read3 =>
  state_next <= read4;
when read4 =>
  state_next <= idle;
end case;
end process;

```



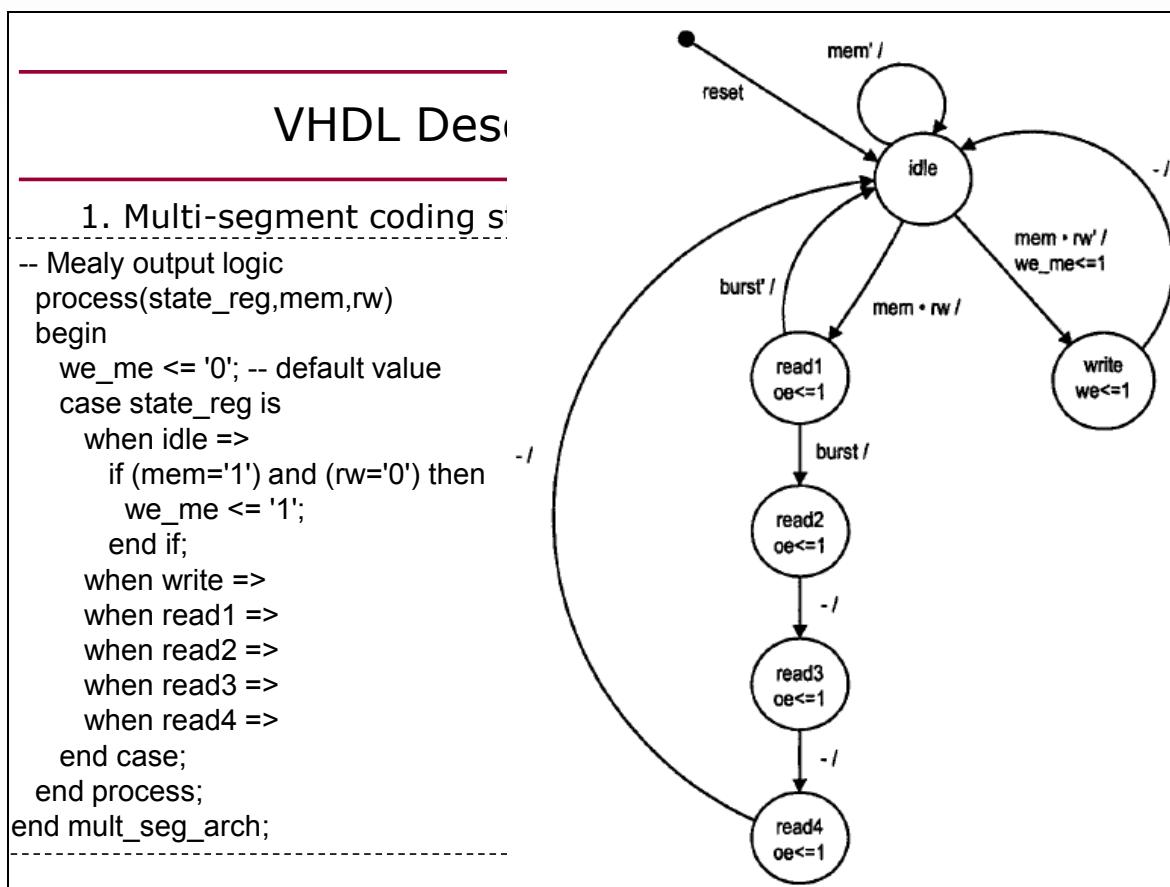
Notes:

The third code segment is the Moore output logic, Again, we use a case statement with state_reg as the selection expression. Note that since the Moore output is a function of state only, no input signal is in the sensitive list. Two sequential signal assignment statements are used to represent the default output value:

```

we <= '0' ;
oe <= '0' ;
  
```

If an output signal is asserted inside a state box, we put a signal assignment statement in the corresponding choice in the VHDL code to overwrite the default value.

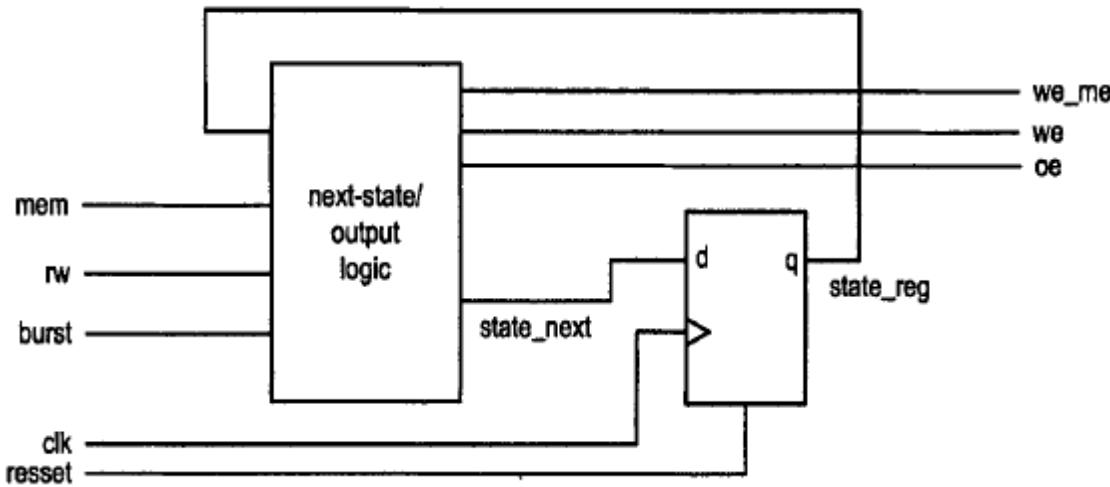


Notes:

The fourth code segment is the Mealy output logic. Note that some input signal is now in the sensitive list. Again, following the ASM chart, we use a case statement with `state_reg` as the selection expression and use an if statement for the decision box. The Mealy output, the `we_me` signal, will be assigned to the designated value according to the input condition.

VHDL Description of an FSM

2. Two-segment coding style.



Digital Design using VHDL 33

Notes:

The two-segment coding style divides an FSM into a state register segment and a combinational circuit segment, which integrates the next-state logic, Moore output logic and Mealy output logic. In VHDL code, we need to merge the three segments and move the `state_next`, `oe`, `we` and `we_me` signals into a single process.

VHDL Description of an FSM

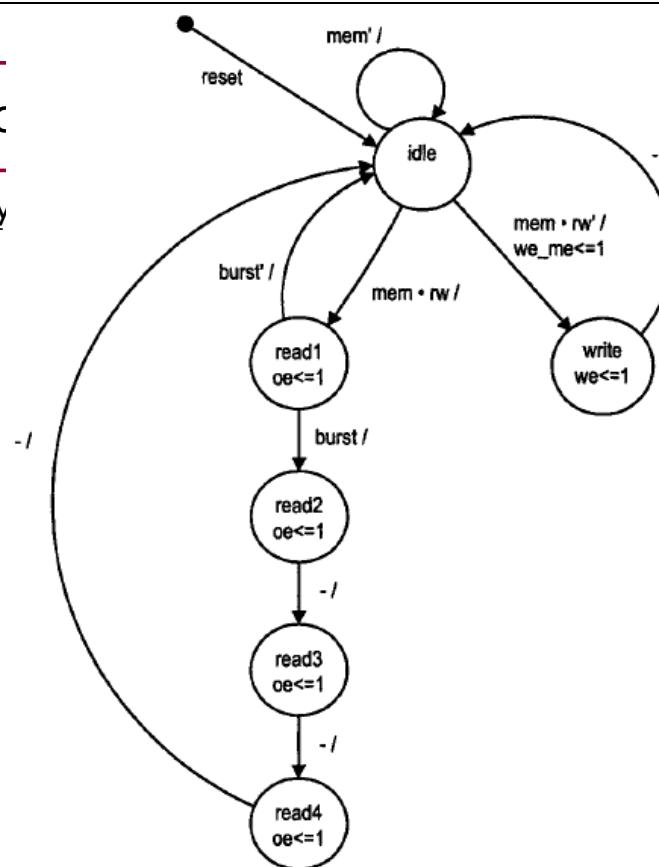
2. Two-segment coding style.

```
architecture two_seg_arch of mem_ctrl is
  type mc_state_type is
    (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: mc_state_type;
begin
  -- state register
  process(clk,reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;
```

VHDL Description

2. Two-segment coding style

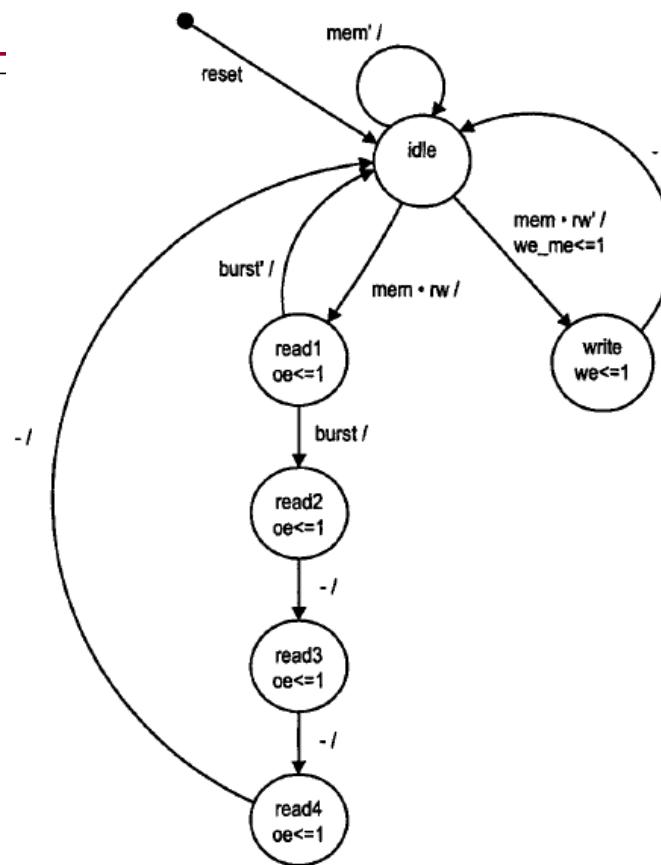
```
-- next-state logic and output logic
process(state_reg,mem,rw,burst)
begin
    oe <= '0'; -- default values
    we <= '0';
    we_me <= '0';
    case state_reg is
        when idle =>
            if mem='1' then
                if rw='1' then
                    state_next <= read1;
                else
                    state_next <= write;
                    we_me <= '1';
                end if;
            else
                state_next <= idle;
            end if;
```



```

when write =>
    state_next <= idle;
    we <= '1';
when read1 =>
    if (burst='1') then
        state_next <= read2;
    else
        state_next <= idle;
    end if;
    oe <= '1';
when read2 =>
    state_next <= read3;
    oe <= '1';
when read3 =>
    state_next <= read4;
    oe <= '1';
when read4 =>
    state_next <= idle;
    oe <= '1';
end case;
end process;
end two_seg_arch;

```





State Assignment

- During synthesis, each symbolic state has to be mapped to a unique binary representation so that the FSM can be realized by physical hardware.
- State assignment: assigning binary representations to symbolic states.
- Good assignment reduce the complexity of next state & output logic
- An FSM with n symbolic states requires a state register of at least $\lceil \log_2 n \rceil$ bits to encode all possible symbolic values but we sometimes utilize more bits for other purposes.

Digital Design using VHDL 37



State Assignment

- Example

	Binary assignment	Gray code assignment	One-hot assignment	Almost one-hot assignment
idle	000	000	000001	00000
read1	001	001	000010	00001
read2	010	011	000100	00010
read3	011	010	001000	00100
read4	100	110	010000	01000
write	101	111	100000	10000

Digital Design using VHDL 38

Notes:

- Binary (or sequential) assignment: assigns states according to a binary sequence. This scheme uses a minimal number of bits and needs only a $\lceil \log_2 n \rceil$ -bit register.
- Gray code assignment: assigns states according to a Gray code sequence. This scheme also uses a minimal number of bits. Because only one bit changes between the successive code words in the sequence, we may reduce the complexity of next-state logic if assigning successive code words to neighboring states.
- One-hot assignment: assigns one bit for each state, and thus only a single bit is '1' (or "hot") at a time. For an FSM with n states, this scheme needs an n -bit register.

- Almost one-hot assignment: is similar to the one-hot assignment except that the all-zero representation ("0 . . . 0") is also included. The all-zero state is frequently used as the initial state since it can easily be reached by asserting

the asynchronous reset signal of D FFs. This scheme needs an $(n - 1)$ -bit register for n states.

Although one-hot and almost one-hot assignments need more register bits, empirical data from various studies show that these assignments may reduce the circuit size of next-state logic and output logic.



State Assignment in VHDL

- In some situations, we may want to specify the state assignment for an FSM manually. This can be done implicitly or explicitly.
- In implicit state assignment, we keep the original enumeration data type but pass the desired assignment by other mechanisms. We will use user attributes enum_encoding .

```
type mc_state_type is (idle,write,read1,read2,read3,read4);
attribute enum_encoding: string;
attribute enum_encoding of mc_state_type:
    type is "0000 0100 1000 1001 1010 1011";
```

Digital Design using VHDL 39

Notes:

In implicit state assignment, we keep the original enumeration data type but pass the desired assignment by other mechanisms. The VHDL standard does not define any rule for mapping the values of an enumeration data type to a set of binary representations. It is performed during synthesis. One way to pass the desired statement assignment to software is to use a VHDL feature, known as a user attribute, to set a directive to guide operation of the software. A user attribute has no effect on the semantics of VHDL code and is recognized only by the software that defines it. The IEEE 1076.6 RTL synthesis standard defines an attribute named enum_encoding for encoding the values of an enumeration data type. This attribute can be used for state assignment. For example, we can write the code above if we wish to assign the binary representations "0000", "0100", "1000", "1001", "1010" and "1011" to

the idle, write, read1, read2, read3 and read4 states of the memory controller FSM.

Synthesis software normally provides several simple state assignment schemes. If nothing is specified, the software will perform the state assignment automatically. It normally selects between binary assignment and one-hot assignment, depending on the characteristics of the targeting device technology. We can also use specialized FSM optimization software to obtain a good, suboptimal assignment.



State Assignment in VHDL

- In explicit state assignment, we specify the desired state assignment by replacing the symbolic values with the actual binary representations, and use the std_logic_vector data type for this purpose.

```
architecture state_assign_arch of mem_fsm is
  constant idle: std_logic_vector(3 downto 0) := "0000";
  constant write: std_logic_vector(3 downto 0) := "0100";
  constant read1: std_logic_vector(3 downto 0) := "1000";
  constant read2: std_logic_vector(3 downto 0) := "1001";
  constant read3: std_logic_vector(3 downto 0) := "1010";
  constant read4: std_logic_vector(3 downto 0) := "1011";
  signal state_reg, state_next: std_logic_vector(3 downto 0);
```

Digital Design using VHDL 40

Notes:

In explicit state assignment, we use std_logic_vector(3 downto 0) as the state register's data type. Six constants are declared to represent the six symbolic state names. Because of the choice of the constant names, the appearance of the code is very similar to that of the original code. However, the name here is just an alias of a binary representation, but the name in the original code is a value of the enumeration data type. One difference in the next-state logic code segment is an extra when clause:

```
when others =>
  state_next <= idle;
```

This revision is necessary since the selection expression of the case statement, state_reg, now is with the std_logic_vector(3 downto 0) data type, and thus has 9^4 possible combinations. The when others clause is used to cover all the unused combinations. This means that when the FSM reaches an unused binary representation (e.g., "1111"), it will return to the idle state in the next clock cycle.

Design Examples

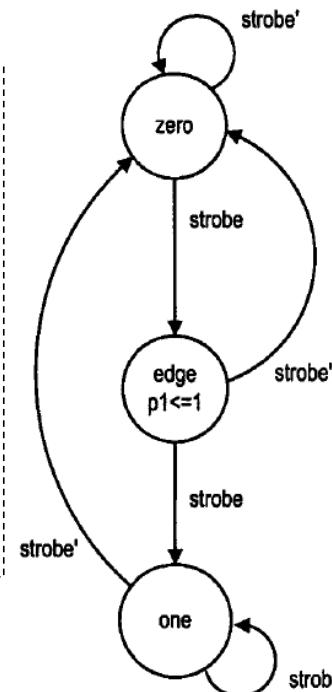
- Edge Detector Circuit
- Arbitrator
- Manchester Encoding Circuit
- FSM-based Binary Counter

Edge Detector Circuit

- Moore Machine Design:

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector1 is
port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    p1: out std_logic
);
end edge_detector1;

architecture moore_arch of edge_detector1 is
type state_type is (zero, edge, one);
signal state_reg, state_next: state_type;
begin
```

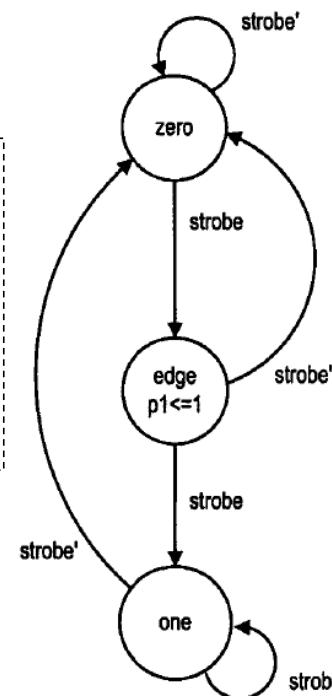


42

Edge Detector Circuit

- Moore Machine Design:

```
-- state register
process(clk,reset)
begin
  if (reset='1') then
    state_reg <= zero;
  elsif (clk'event and clk='1') then
    state_reg <= state_next;
  end if;
end process;
```

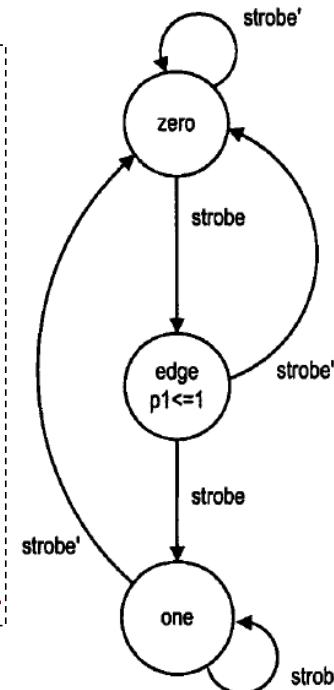


43

Edge Detector Circuit

- Moore Machine Design:

```
-- next-state logic
process(state_reg,strobe)
begin
    case state_reg is
        when zero=>
            if strobe= '1' then
                state_next <= edge;
            else
                state_next <= zero;
            end if;
        when edge =>
            if strobe= '1' then
                state_next <= one;
            else
                state_next <= zero;
            end if;
```



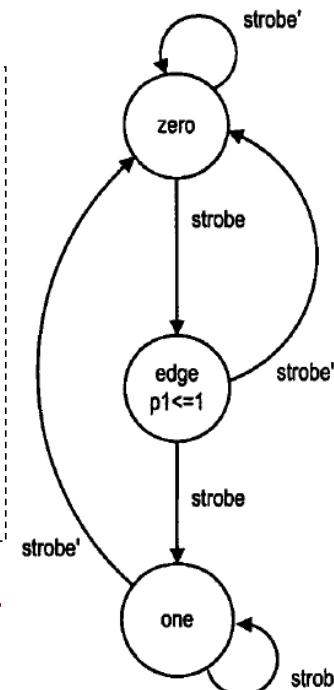
44

Edge Detector Circuit

- Moore Machine Design:

```
when one =>
    if strobe= '1' then
        state_next <= one;
    else
        state_next <= zero;
    end if;
end case;
end process;

-- Moore output logic
p1 <= '1' when state_reg=edge else
  '0';
end moore_arch;
```



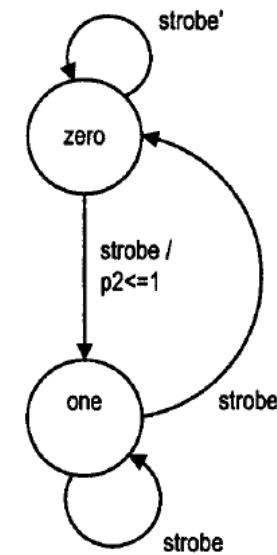
45

Edge Detector Circuit

- Mealy Machine Design:

```
library ieee;
use ieee.std_logic_1164.all;
entity edge_detector2 is
port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    p2: out std_logic
);
end edge_detector2;

architecture mealy_arch of edge_detector2 is
type state_type is (zero, one);
signal state_reg, state_next: state_type;
begin
```

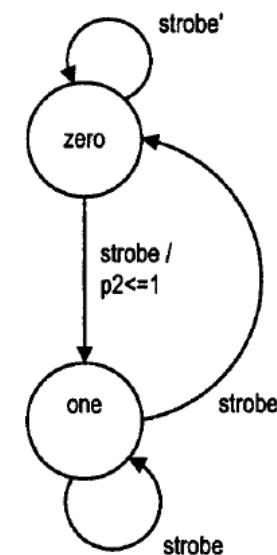


Digital Design using VHDL 46

Edge Detector Circuit

- Mealy Machine Design:

```
-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= zero;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;
```

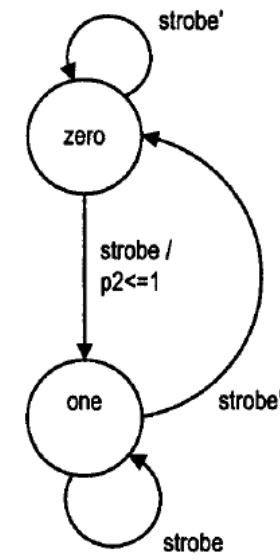


Digital Design using VHDL 47

Edge Detector Circuit

- Mealy Machine Design:

```
-- next-state logic
process(state_reg,strobe)
begin
    case state_reg is
        when zero=>
            if strobe= '1' then
                state_next <= one;
            else
                state_next <= zero;
            end if;
        when one =>
            if strobe= '1' then
                state_next <= one;
            else
                state_next <= zero;
            end if;
    end case;
end process;
```

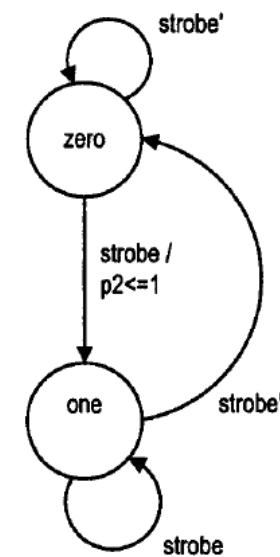


Digital Design using VHDL 48

Edge Detector Circuit

- Mealy Machine Design:

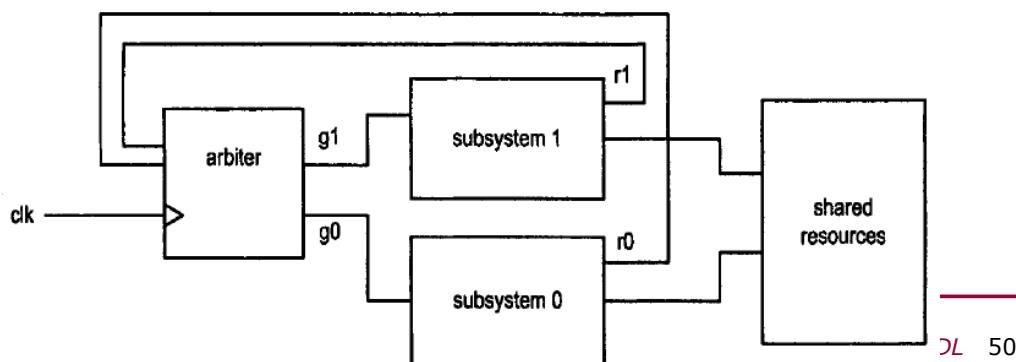
```
-- Mealy output logic
p2 <= '1' when (state_reg=zero) and (strobe='1')
else
    '0';
end mealy_arch;
```



Digital Design using VHDL 49

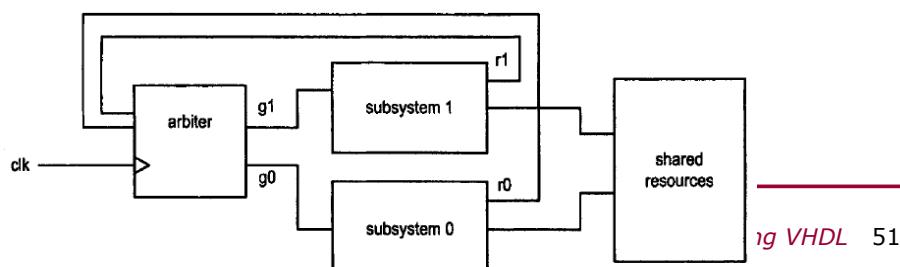
Arbiter

- In a large system, some resources are shared by many subsystems. For example, several processors may share the same block of memory, and many peripheral devices may be connected to the same bus.
- An arbiter is a circuit that resolves any conflict and coordinates the access to the shared resource.
- Example: an arbiter with two subsystems



Arbiter

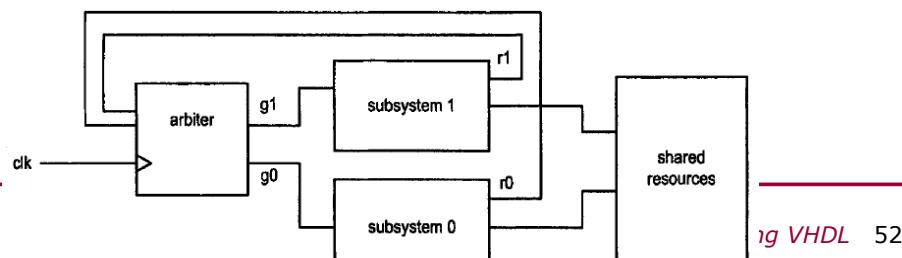
- The subsystems communicate with the arbiter by a pair of request and grant signals, which are labeled as $r(1)$ and $g(1)$ for subsystem 1, and as $r(0)$ and $g(0)$ for subsystem 0.
- When a subsystem needs the resources, it activates the request signal.
- The arbiter monitors use of the resources and the requests, and grants access to a subsystem by activating the corresponding grant signal.
- Once its grant signal is activated, a subsystem has permission to access the resources.



ig VHDL 51

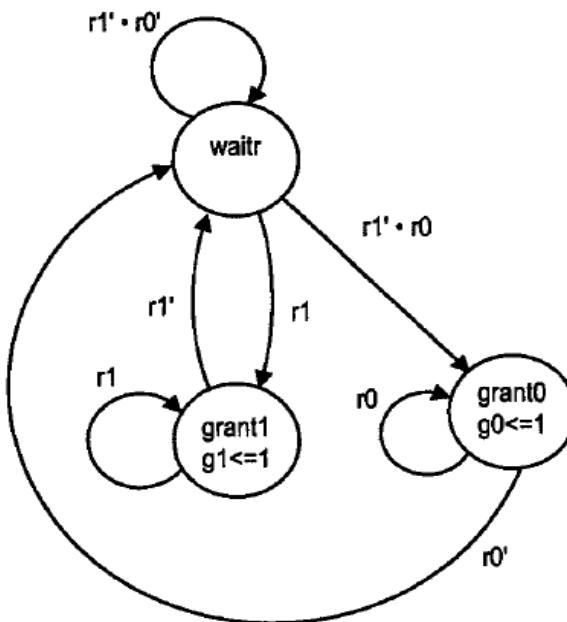
Arbiter

- After the task has been completed, the subsystem releases the resources and deactivates the request signal.
- Since an arbiter's decision is based partially on the events that occurred earlier (i.e., previous request and grant status), it needs internal states to record what happened in the past. An FSM can meet this requirement.
- One critical issue in designing an arbiter is the handling of simultaneous requests.
- Our first design gives priority to subsystem 1.



Arbiter

First Design: Fixed priority to subsystem 1



sign using VHDL 53

Notes:

The state diagram of the FSM consists of three states, waitr, grant1 and grant0. The waitr state indicates that the resources is available and the arbiter is waiting for a request. The grant1 and grant0 states indicate that the resource is granted to subsystem 1 and subsystem 0 respectively. Initially, the arbiter is in the waitr state. If the $r(1)$ input (the request from subsystem 1) is activated at the rising edge of the clock, it grants the resources to subsystem 1 by moving to the grant1 state. The $g(1)$ signal is asserted in this state to inform subsystem 1 of the availability of the resources. After subsystem 1 completes its usage, it signals the release of the resources by deactivating the $r(1)$ signal. The arbiter returns to the waitr state accordingly.

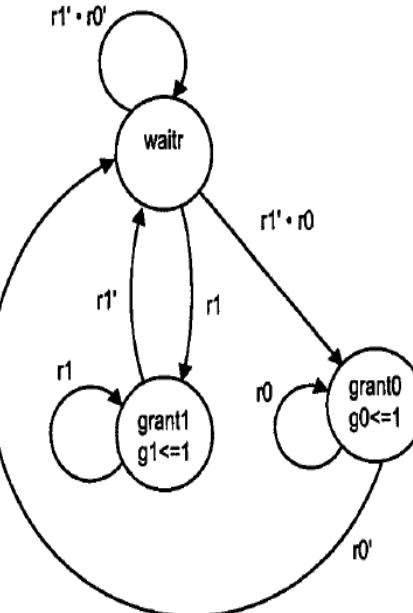
In the waitr state, if $r(1)$ is not activated and $r(0)$ is activated at the rising edge, the arbiter grants the resources to subsystem 0 by moving to the grant0 state and activates the $g(0)$ signal. Subsystem 0 can then have the resources until it releases them.

Arbiter

First Design: Fixed priority to subsystem 1

```
library ieee;
use ieee.std_logic_1164.all;
entity arbiter2 is
port(
    clk: in std_logic;
    reset: in std_logic;
    r: in std_logic_vector(1 downto 0);
    g: out std_logic_vector(1 downto 0)
);
end arbiter2;
```

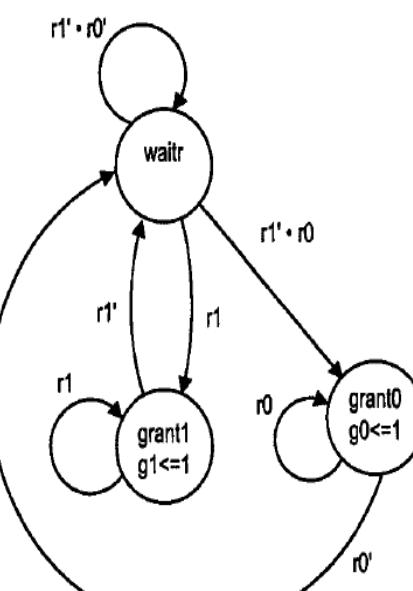
```
architecture fixed_prio_arch of arbiter2 is
type mc_state_type is (waitr, grant1, grant0);
signal state_reg, state_next: mc_state_type;
begin
```



Arbiter

First Design: Fixed priority to subsystem 1

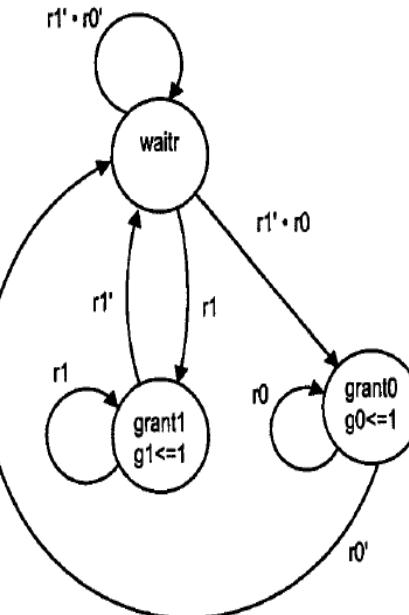
```
-- state register
process(clk,reset)
begin
  if (reset='1') then
    state_reg <= waitr;
  elsif (clk'event and clk='1') then
    state_reg <= state_next;
  end if;
end process;
```



Arbiter

First Design: Fixed priority to subsystem 1

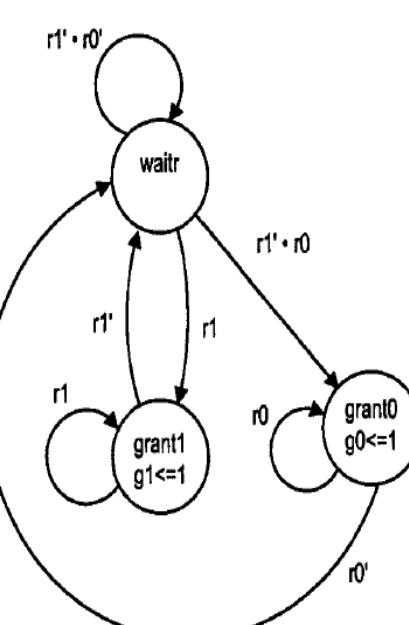
```
-- next-state and output logic
process(state_reg,r)
begin
  g <= "00"; -- default values
  case state_reg is
    when waitr =>
      if r(1)='1' then
        state_next <= grant1;
      elsif r(0)='1' then
        state_next <= grant0;
      else
        state_next <= waitr;
      end if;
```



Arbiter

First Design: Fixed priority to subsystem 1

```
when grant1 =>
  if (r(1)='1') then
    state_next <= grant1;
  else
    state_next <= waitr;
  end if;
  g(1) <= '1';
when grant0 =>
  if (r(0)='1') then
    state_next <= grant0;
  else
    state_next <= waitr;
  end if;
  g(0) <= '1';
end case;
end process;
end fixed_prio_arch;
```

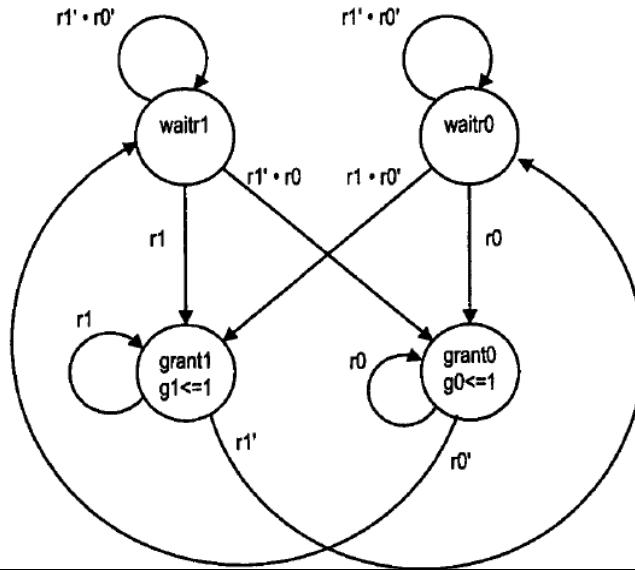


Arbiter

- The resource allocation of the previous design gives priority to subsystem 1.
- This may cause a problem if subsystem 1 requests the resources continuously.
- We can revise the state diagram to enforce a fairer arbitration policy. The new policy keeps track of which subsystem had the resources last time and gives preference to the other subsystem if the two request signals are activated simultaneously.
- The new design has to distinguish two kinds of wait conditions. The first condition is that the resources were last used by subsystem 1 so preference should be given to subsystem 0. The other condition is the reverse of the first.

Arbiter

- To accommodate the two conditions, we split the original waitr state into the waitr1 and waitr0 states, in which subsystem 1 and subsystem 0 will be given preferential treatment respectively.



Notes:

Note that FSM moves from the grant0 state to the waitr1 state after subsystem 0 deactivates the request signal, and moves from the grant1 state to the waitr0 state after subsystem 1 deactivates the request signal.

Arbiter

Second Design: Arbiter with alternating priority

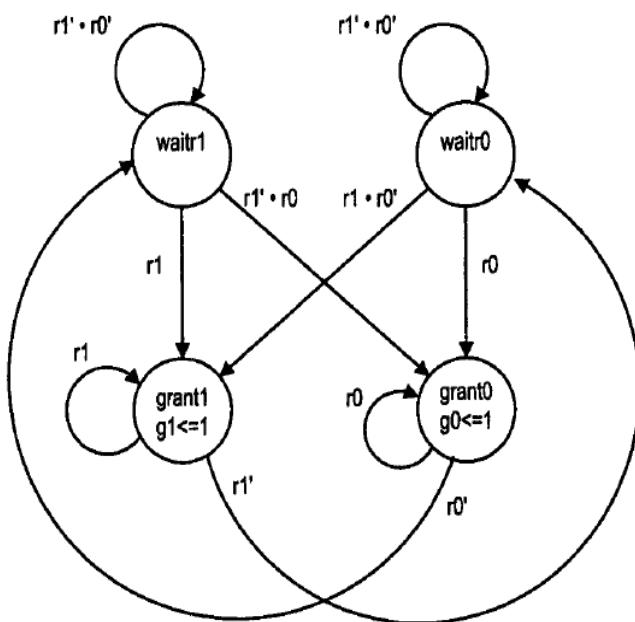
```
architecture rotated_prio_arch of arbiter2 is
  type mc_state_type is (waitr1, waitr0, grant1, grant0);
  signal state_reg, state_next: mc_state_type;
begin
  -- state register
  process(clk,reset)
  begin
    if (reset='1') then
      state_reg <= waitr1;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;
```

Digital Design using VHDL 60

Arbiter

Second Design: Arbiter with alternating priority

```
-- next-state and output logic
process(state_reg,r)
begin
  g <= "00";  -- default values
  case state_reg is
    when waitr1 =>
      if r(1)='1' then
        state_next <= grant1;
      elsif r(0)='1' then
        state_next <= grant0;
      else
        state_next <= waitr1;
      end if;
```



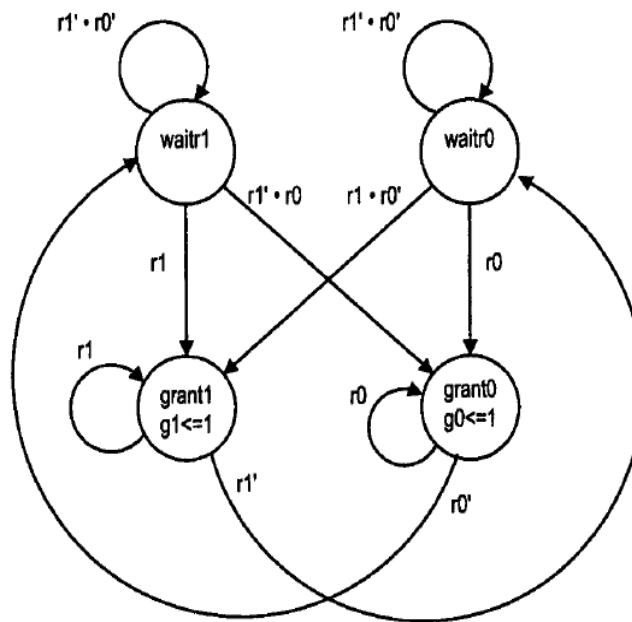
Arbiter

Second Design: Arbiter with alternating priority

```

when waitr0 =>
  if r(0)='1' then
    state_next <= grant0;
  elsif r(1)='1' then
    state_next <= grant1;
  else
    state_next <= waitr0;
  end if;
when grant1 =>
  if (r(1)='1') then
    state_next <= grant1;
  else
    state_next <= waitr0;
  end if;
  g(1) <= '1';

```



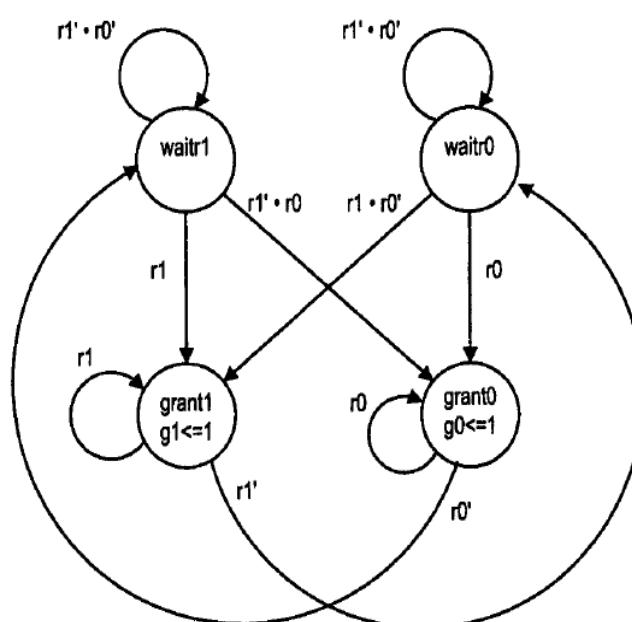
Arbiter

Second Design: Arbiter with alternating priority

```

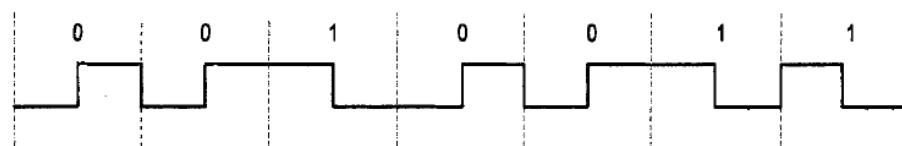
when grant0 =>
  if (r(0)='1') then
    state_next <= grant0;
  else
    state_next <= waitr1;
  end if;
  g(0) <= '1';
end case;
end process;
end rotated_prio_arch;

```



Manchester Encoding Circuit

- Manchester code is a coding scheme used to represent a bit in a data stream. A '0' value of a bit is represented as a 0-to-1 transition, in which the first half is '0' and the remaining half is '1' while a '1' value of a bit is represented as a 1-to-0 transition, in which the first half is '1' and the remaining half is '0'.



- The Manchester code is frequently used in a serial communication line. Since there is a transition in each bit, the receiving system can use the transitions to recover the clock information.

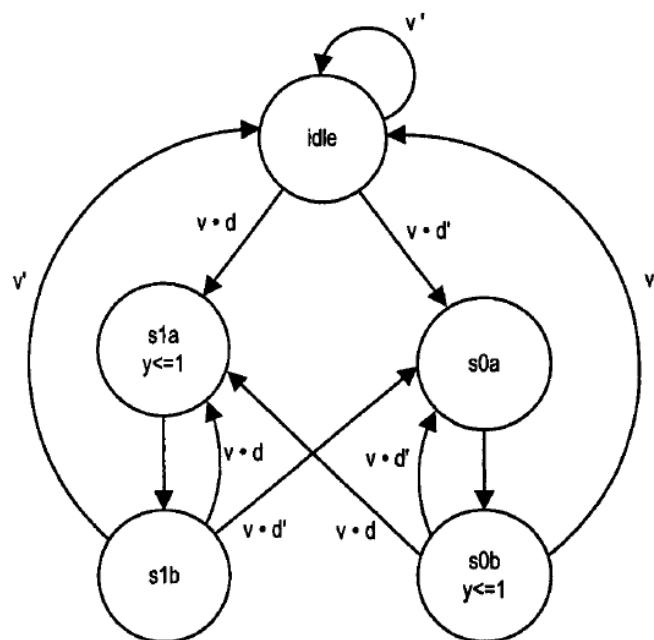


Manchester Encoding Circuit

- The Manchester encoder transforms a regular data stream into a Manchester-coded data stream.
- Two clock cycles are needed to represent each bit. Thus, the maximal data rate is only half of the clock rate because an encoded bit includes a sequence of "01" or "10".
- There are two input signals. The d signal is the input data stream, and the v signal indicates whether the d signal is valid (i.e., whether there is data to transmit).
- The d signal should be converted to Manchester code if the v signal is '1'. The output remains '0' otherwise.

Digital Design using VHDL 65

Manchester Encoding Circuit



Digital Design using VHDL 66

Notes:

While v is asserted, the FSM starts the encoding process. If d is '0', it travels through the $s0a$ and $s0b$ states. If d is '1', the FSM travels through the $s1a$ and $s1b$ states. Once the FSM reaches the $s1b$ or $s0b$ state, it checks the v signal. If the v signal is still asserted, the FSM skips the idle state and continuously encodes the next input data.

The Moore output is used because we have to generate two equal intervals for each bit.

Manchester Encoding Circuit

```

library ieee;
use ieee.std_logic_1164.all;
entity manchester_encoder is
port(
    clk, reset: in std_logic;
    v,d: in std_logic;
    y: out std_logic
);
end manchester_encoder;

architecture moore_arch of manchester_encoder is
type state_type is (idle, s0a, s0b, s1a, s1b);
signal state_reg, state_next: state_type;
begin

```

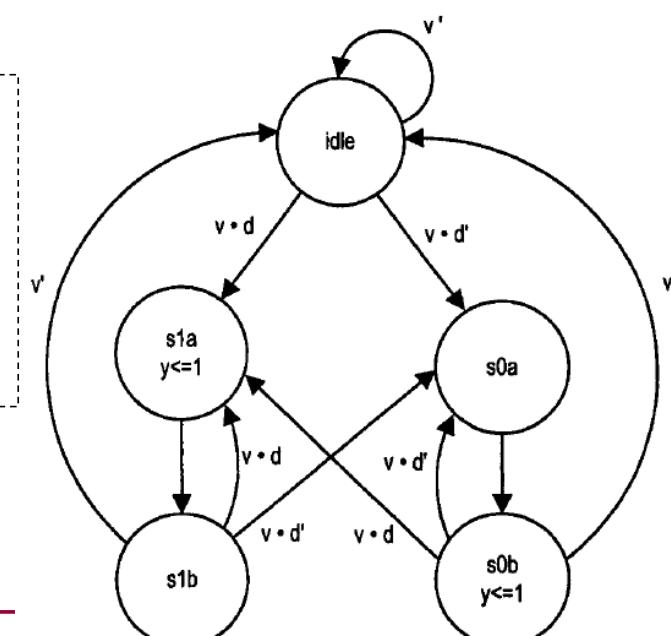
Digital Design using VHDL 67

Manchester Encoding Circuit

```

-- state register
process(clk,reset)
begin
    if (reset='1') then
        state_reg <= idle;
    elsif (clk'event and clk='1') then
        state_reg <= state_next;
    end if;
end process;

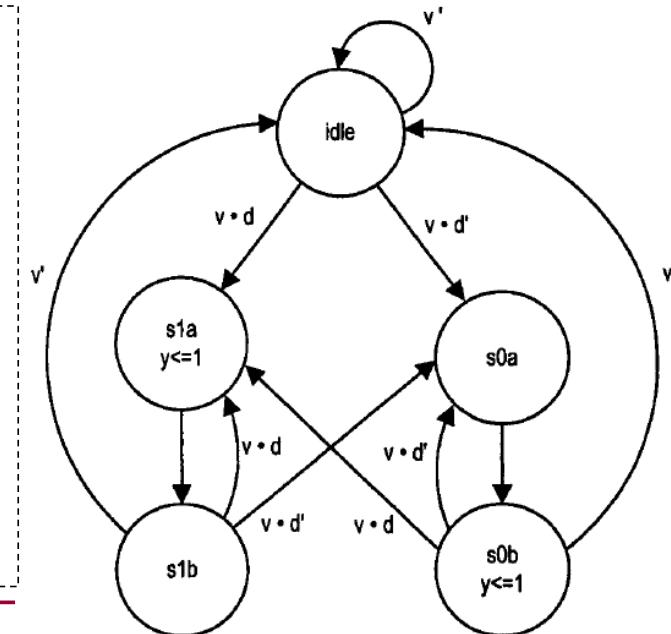
```



Digital Design using VHDL 68

Manchester Encoding Circuit

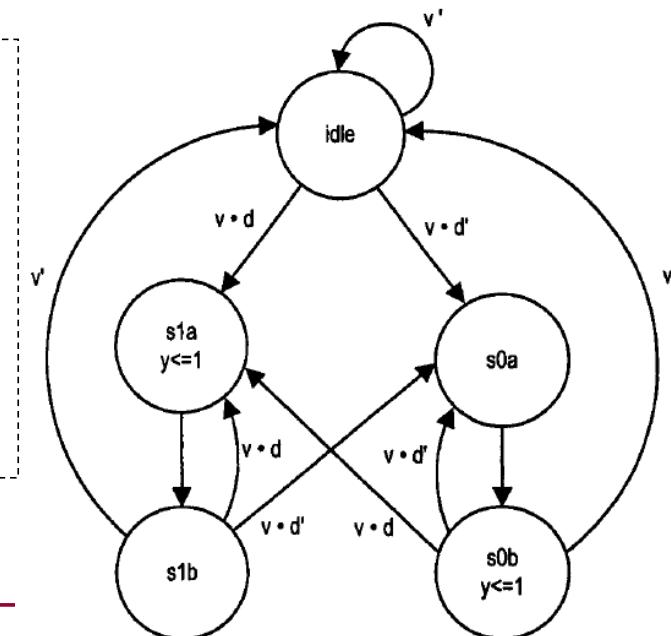
```
-- next-state logic
process(state_reg,v,d)
begin
  case state_reg is
    when idle=>
      if v= '0' then
        state_next <= idle;
      else
        if d= '0' then
          state_next <= s0a;
        else
          state_next <= s1a;
        end if;
      end if;
    when s0a =>
      state_next <= s0b;
```



Digital Design using VHDL 69

Manchester Encoding Circuit

```
when s1a =>
  state_next <= s1b;
when s0b =>
  if v= '0' then
    state_next <= idle;
  else
    if d= '0' then
      state_next <= s0a;
    else
      state_next <= s1a;
    end if;
  end if;
```



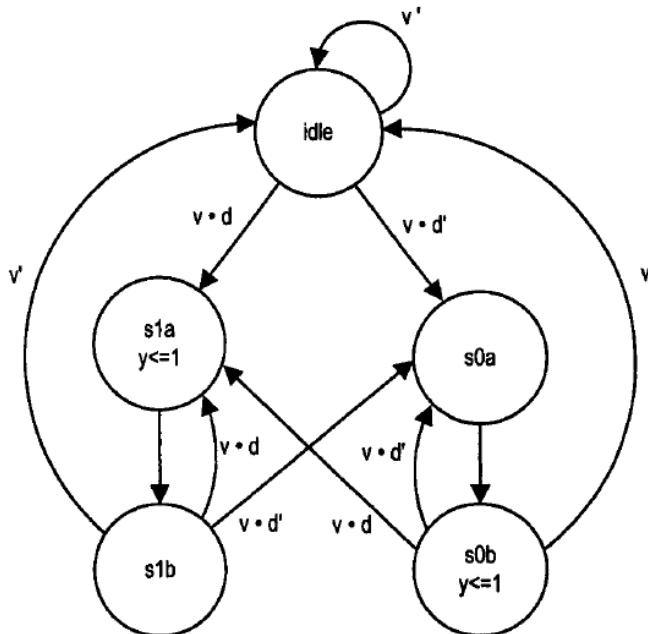
Digital Design using VHDL 70

Manchester Encoding Circuit

```

when s1b =>
    if v= '0' then
        state_next <= idle;
    else
        if d= '0' then
            state_next <= s0a;
        else
            state_next <= s1a;
        end if;
    end if;
end case;
end process;
-- Moore output logic
y <= '1' when state_reg=s1a or
state_reg=s0b else
    '0';
end moore_arch ;

```

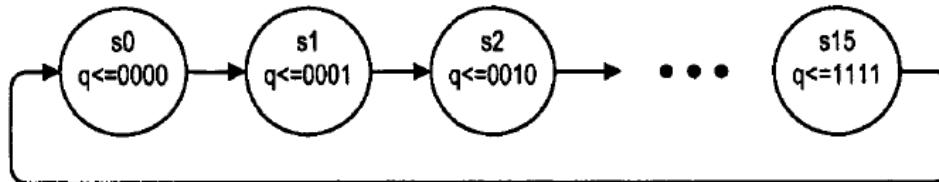


Digital Design using VHDL 71

FSM-based Binary Counter

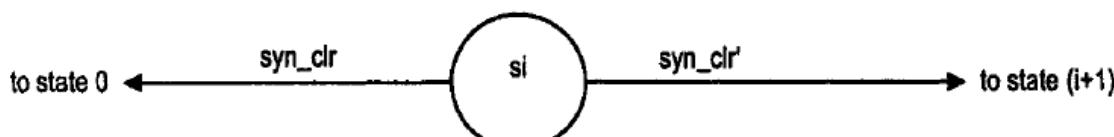
- In theory, all sequential circuits with finite memory can be modeled by FSMs and derived accordingly.
- We will start with a free running 4-bit binary counter and then we will gradually add some features.
- We will see that it is better to use FSM to model random sequential circuits rather than regular ones.

➤ Free running 4-bit binary counter

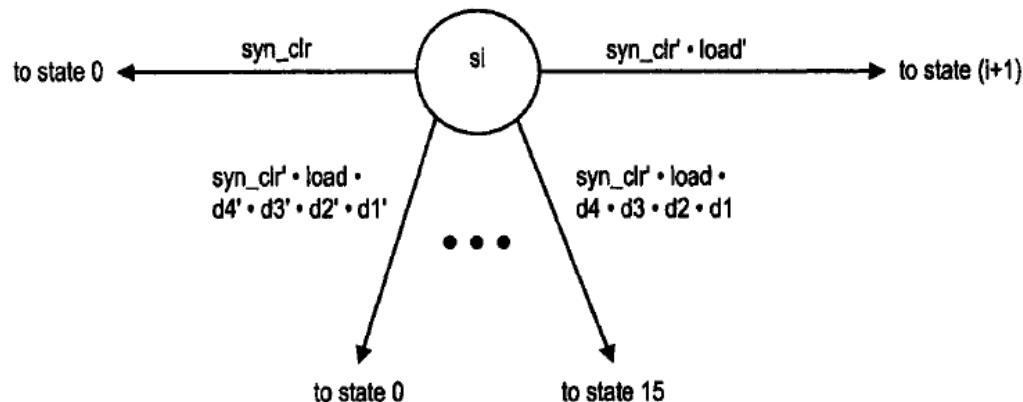


FSM-based Binary Counter

- 4-bit binary counter with synchronous clear



- 4-bit binary counter with synchronous clear & load



3

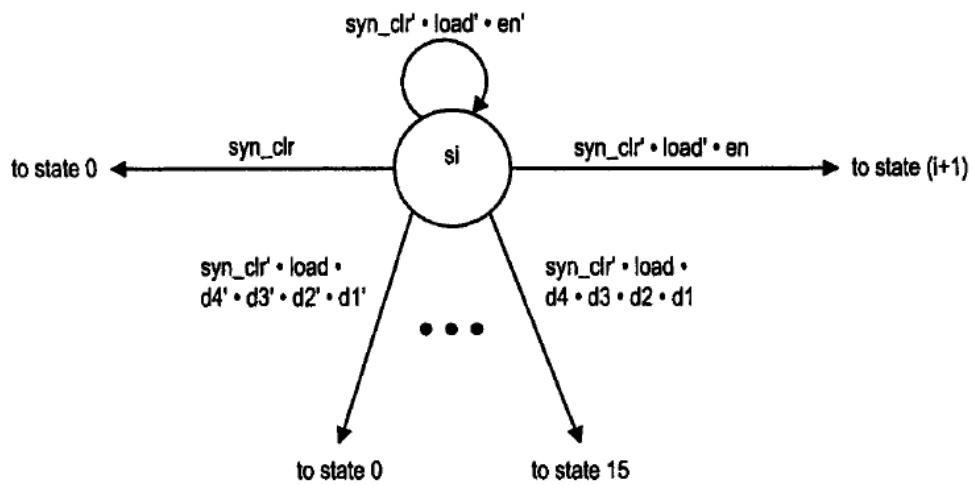
Notes:

We first add the synchronous clear signal, syn_clr , which clears the counter to 0. In the FSM, it corresponds to forcing the FSM to return to the initial state, s_0 . Note that the logic expressions give priority to the synchronous clear operation.

The next step is to add the load operation. This actually involves five input bits, which include the 1-bit control signal, $load$, and the 4-bit data signal, d . The d signal is the value to be loaded into the counter and it is composed of four individual bits, d_3 , d_2 , d_1 and d_0 . The load operation changes the content of the register according to the value of d . In terms of FSM operation, 16 transitions are needed to express the possible 16 next states.

FSM-based Binary Counter

- 4-bit binary counter with synchronous clear & load & enable



Digital Design using VHDL 74

Notes:

Finally, we add the enable signal, en, which can suspend the counting. In terms of FSM operation, it corresponds to staying in the same state.

Note that the logic expressions of the transition arches set the priority of the control signals in the order `syn_clr`, `load` and `en`.

FSM-based Binary Counter

- Although this design process is theoretically doable, it will be very large.
- Then what about a larger, say, a 16- or 32-bit, counter ?!

Digital Design using VHDL 75

Home Work

1. Revise the edge detection circuit to detect both 0-to-1 and 1-to-0 transitions; i.e., the circuit will generate a short pulse whenever the strobe signal changes state.

First: Use a Moore machine with a minimal number of states to realize this circuit.

Second: use a Mealy machine to realize the circuit. The Mealy machine needs only two states.

For each of them:

- (a) Derive the state diagram.
- (b) Convert the state diagram to an ASM chart.
- (c) Derive VHDL code according to the ASM chart.



Home Work

2. In digital communication, a special synchronization pattern, known as a preamble, is used to indicate the beginning of a packet. For example, the Ethernet preamble includes eight repeating octets of "10101010".
 - a) Design an FSM that generates the "10101010" pattern. The circuit has an input signal, start, and an output, data-out. When start is '1', the "10101010" will be generated in the next eight clock cycles.
 - b) Design an FSM to detect the "10101010" pattern in the receiving end. The circuit has an input signal, data_in, and an output signal, match. The match signal will be asserted as '1' for one clock period when the input pattern "10101010" is detected.



Home Work

3. A Manchester decoder transforms a Manchester-coded data stream back to a regular binary data stream. There are two output signals, data & valid. The data signal is the recovered data bit, which can be '0' or '1'. The valid signal indicates whether a transition occurs. The valid signal is used to distinguish whether the '0' of the data is due to the 0-to-1 transition or inactivity of the data stream.
 - (a) Derive the state diagram for this decoder.
 - (b) Convert the state diagram to an ASM chart.
 - (c) Derive VHDL code.



Home Work

4. Non-return to-zero invert-to ones (NRZI) code is another code used in serial transmission. The output of an NRZI encoder is '0' if the current input value is different from the previous value and is '1' otherwise.
 - a) Design an NRZI encoder using an FSM and derive the VHDL code accordingly.
 - b) Design an NRZI decoder, which converts a NRZI-coded stream back to a regular binary stream.



Session 6

FPGAs

Digital Design using VHDL 2



Topics for this Lecture

- What are FPGAs?
- What can FPGAs used to?
- The origin of FPGAs
- PLDs & ASICs
- FPGA Architecture
- Who are the vendors?

Digital Design using VHDL 3



What are FPGAs ?

- Field programmable gate arrays (FPGAs) are digital ICs that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks.
- Design engineers can configure (program) such devices to perform variety of tasks.
- Depending on the way in which they are implemented, some FPGAs may only be programmed a single time (OTP), while others may be reprogrammed over and over again.
- The “field programmable” portion of the FPGA’s name refers to the fact that its programming takes place “in the field” opposed to devices whose internal functionality is hardwired by the manufacturer.

Digital Design using VHDL 4

Notes:

“Programming in the field” may mean that FPGAs are configured in the laboratory, or it may refer to modifying the function of a device in an electronic system that has already been deployed in the outside world.



What can FPGAs be used for?

- ASIC and custom silicon: today's FPGAs are increasingly being used to implement a variety of designs that could previously have been realized using only ASICs and custom silicon.
- Digital signal processing: High-speed DSP has traditionally been implemented using specially tailored microprocessors called digital signal processors (DSPs). However, today's FPGAs can contain embedded multipliers, dedicated arithmetic routing, and large amounts of on-chip RAM, all of which facilitate DSP operations.



What can FPGAs be used for?

- Embedded microcontrollers: Small control functions have traditionally been handled by special-purpose embedded processors called microcontrollers. FPGA prices are falling and the smallest devices now can implement a soft processor core combined with a selection of custom I/O functions. The end result is that FPGAs are becoming increasingly attractive for embedded control applications.
- Physical layer communications: FPGAs have long been used to implement the interfaces between physical layer communication chips and high level networking protocol layers. Today's FPGAs can contain multiple high-speed transceivers so that communications and networking functions can be implemented into a single device.

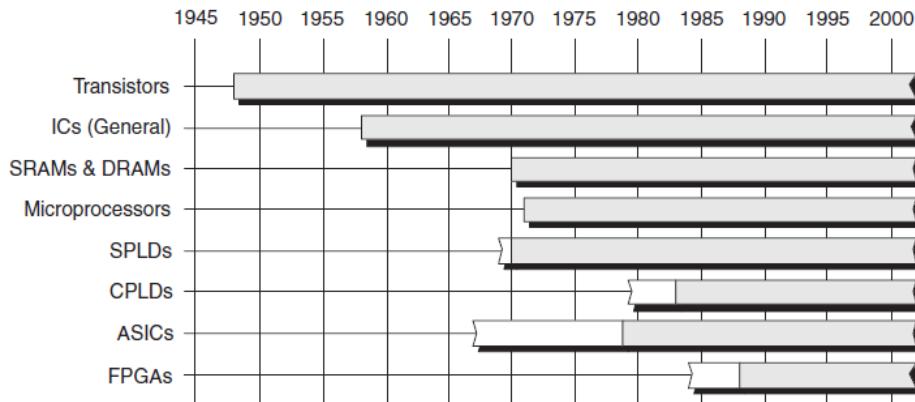
Digital Design using VHDL 6

Notes:

Microcontrollers -these low cost devices- contain on-chip program and instruction memories, timers, and I/O peripherals wrapped around a processor core.

The origin of FPGAs

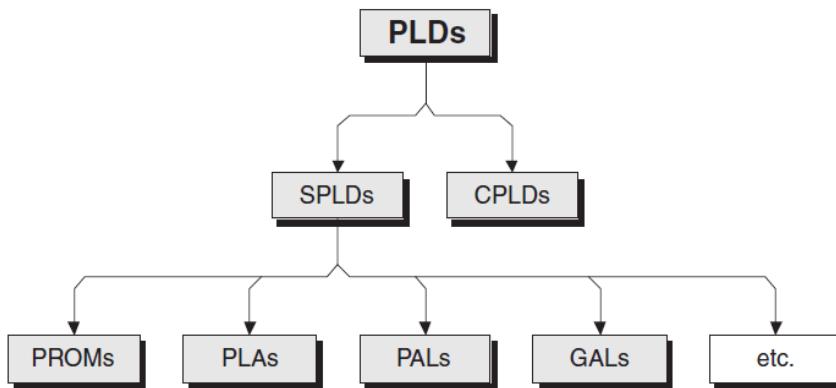
- In order to know the way in which FPGAs developed and the reasons why they appeared, we will consider other related technologies.



- We will focus on PLDs and ASICs.

PLDs

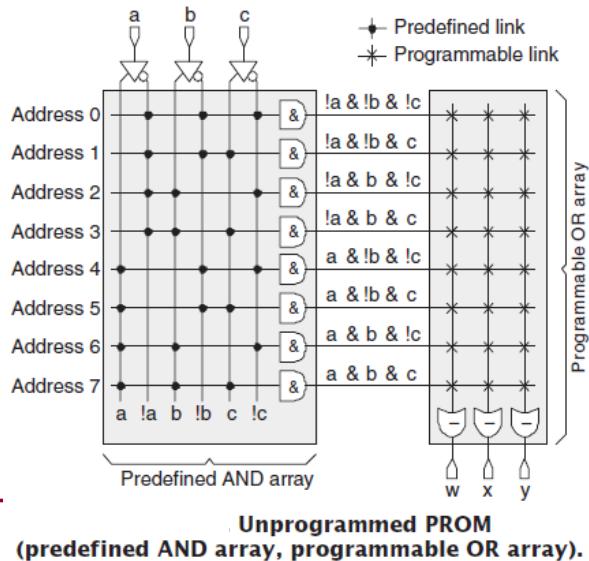
- The first programmable ICs were generically referred to as programmable logic devices (PLDs).
- It started arriving on the scene in 1970 in the form of PROMs.



Digital Design using VHDL 8

PROMs

- They are consisting of a fixed array of AND functions driving a programmable array of OR functions.
- Ex: 3-input, 3-output PROM



9

Notes:

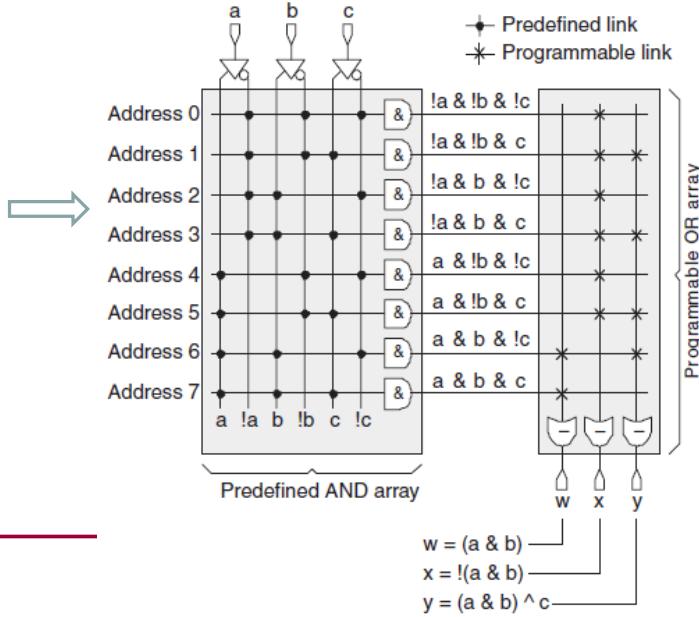
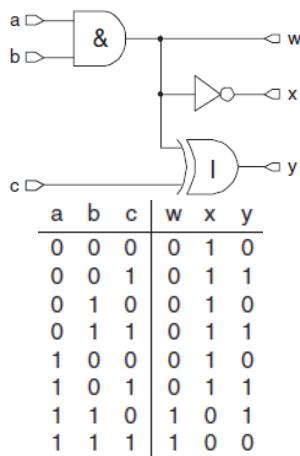
PROMs were originally intended for use as computer memories in which to store program instructions and constant data values. However, design engineers also used them to implement simple logical functions such as lookup tables and state machines.

The programmable links in the OR array can be implemented as fusible links, or as EEPROM transistors and EEPROM cells in the case of EPROM and EEPROM devices, respectively.

It is important to realize that this illustration is intended only to provide a high-level view of the way in which our example device works—it does not represent an actual circuit diagram. In reality, each AND function in the AND array has three inputs provided by the appropriate true or complemented versions of the *a*, *b*, and *c* device inputs. Similarly, each OR function in the OR array has eight inputs provided by the outputs from the AND array.

PROMs

- A PROM can be used to implement any block of combinational logic.
- Ex:



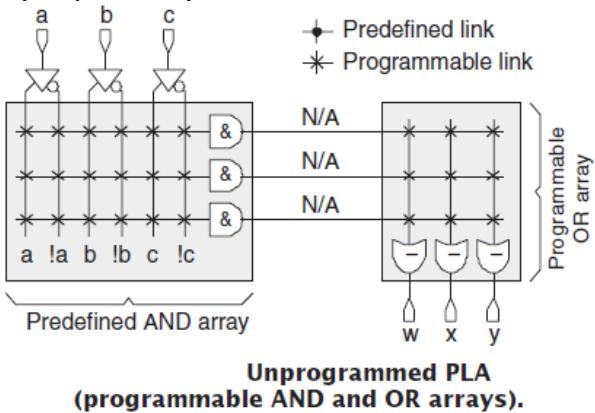
Notes:

The advantage of using PROMs rather than discrete logic component ICs is that quite a large number of these chips could be replaced with a single PROM resulted in circuit boards that were smaller, lighter, cheaper, and less prone to error (each solder joint on a circuit board provides a potential failure mechanism).

Furthermore, if any logic errors were subsequently discovered in this portion of the design (if the design engineer had inadvertently used an AND function instead of a NAND, for example), then these slipups could easily be fixed by blowing a new PROM (or erasing and reprogramming an EPROM or EEPROM). This was preferable to the ways in which errors had to be addressed on boards based on discrete ICs. These included adding new devices to the board, cutting existing tracks with a scalpel, and adding wires by hand to connect the new devices into the rest of the circuit.

PLAs

- In order to address the limitations imposed by the PROM architecture, programmable logic arrays (PLAs) appears.
- These were the most user configurable of the simple PLDs because both the AND and OR arrays were programmable.
- Ex: simple 3-input, 3-output PLA

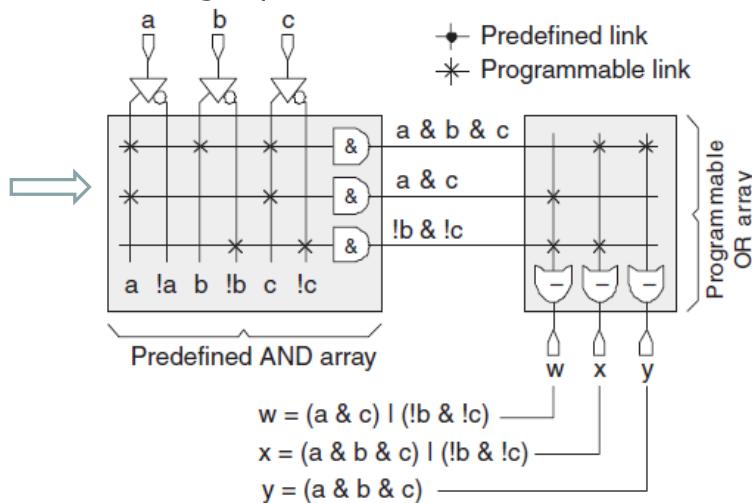


Unprogrammed PLA
(programmable AND and OR arrays).

Digital Design using VHDL 11

PLAs

- Ex: To implement the following equations



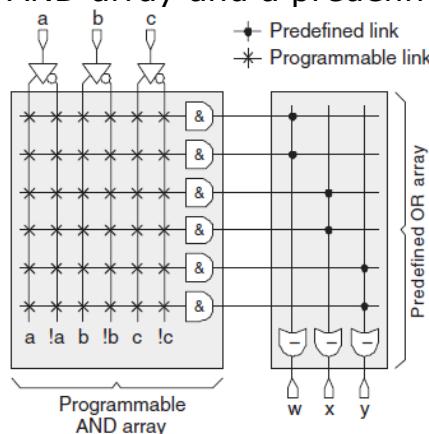
Digital Design using VHDL 12

Notes:

PLAs were touted as being particularly useful for large designs whose logical equations featured a lot of common product terms that could be used by multiple outputs; for example, the product term ($!b \& !c$) is used by both the w and x outputs. This feature may be referred to as product-term sharing. On the downside, signals take a relatively long time to pass through programmable links as opposed to their predefined counterparts. Thus, the fact that both their AND and OR arrays were programmable meant that PLAs were significantly slower than PROMs.

PALs

- In order to address the speed problems posed by PLAs, a new class of device called programmable array logic (PAL) was introduced in the late 1970s.
- Conceptually, a PAL is almost the exact opposite of a PROM because it has a programmable AND array and a predefined OR array.
- Ex: simple 3-input, 3-output PAL



Unprogrammed PAL
(programmable AND array, predefined OR array).

3

Notes:

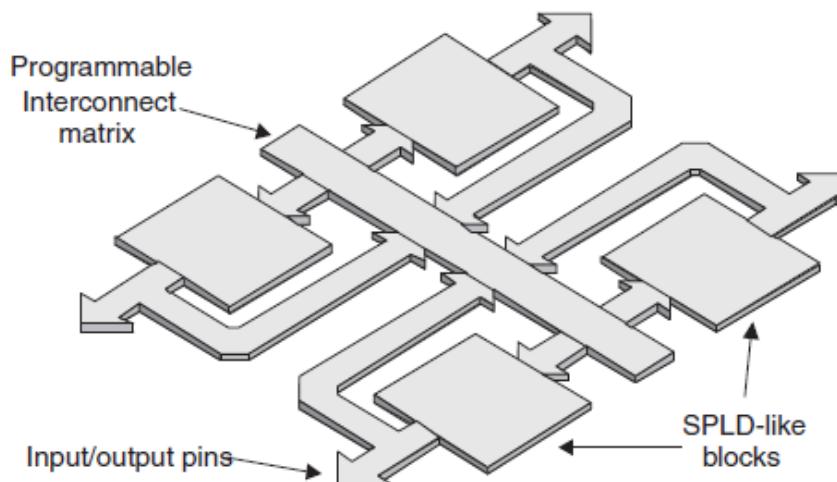
The advantage of PALs (as compared to PLAs) is that they are faster because only one of their arrays is programmable. On the downside, PALs are more limited because they only allow a restricted number of product terms to be ORed together.

CPLDs

- To have a larger functional capability, smaller physical size, faster, more powerful devices, the end of the 1970s and the early 1980s began to see the emergence of more sophisticated PLD devices that became known as complex PLDs (CPLDs).
- Although every CPLD manufacturer fields its own unique architecture, a generic device consists of a number of SPLD blocks (typically PALs) sharing a common programmable interconnection matrix.
- In addition to programming the individual SPLD blocks, the connections between the blocks can be programmed by means of the programmable interconnect matrix.

Digital Design using VHDL 14

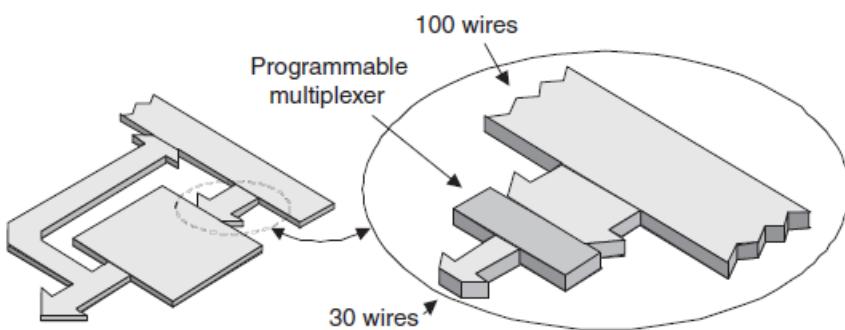
CPLDs



Digital Design using VHDL 15

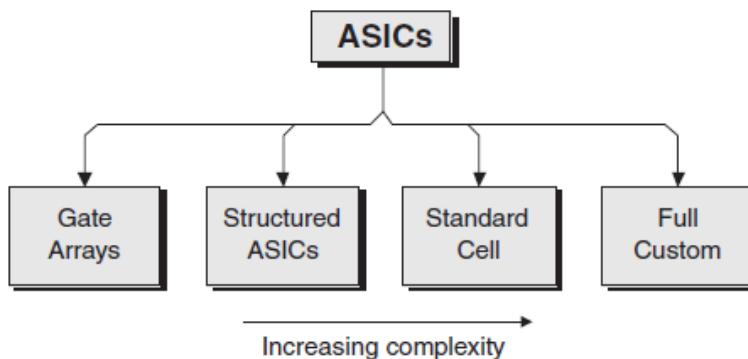
CPLDs

- The programmable interconnect matrix may contain a lot of wires (say 100), but this is more than can be handled by the individual SPLD blocks, which might only be able to accommodate a limited number of signals (say 30).
- Thus, the SPLD blocks are interfaced to the interconnect matrix using some form of programmable multiplexer.



ASICs

- ASICs stands for application-specific integrated circuits.
- It can contain hundreds of millions of logic gates and can be used to create incredibly large and complex functions.



- We will describe them in the sequence in which they appeared.

ASICs

➤ **Full custom:**

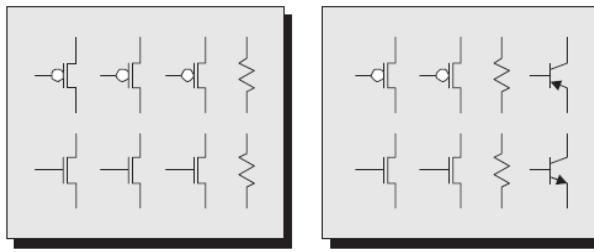
- Design engineers have complete control over every mask layer used to fabricate the silicon chip.
- The ASIC vendor does not prefabricate any components on the silicon and does not provide any libraries of predefined logic gates and functions.
- The engineers can determine the dimensions of individual transistors and then create higher-level functions based on these elements.
- The design of full-custom devices is highly complex and time-consuming, but the resulting chips contain the maximum amount of logic with minimal waste of silicon

Digital Design using VHDL 18

ASICs

➤ **Gate arrays:**

- Gate arrays are based on the idea of a basic cell consisting of a collection of unconnected transistors and resistors.
- Each ASIC vendor determines what it considers to be the optimum mix of components provided in its particular basic cell.



- The ASIC vendor commences by prefabricating silicon chips containing arrays of these basic cells.

Digital Design using VHDL 19



ASICs

- The ASIC vendor defines a set of logic functions such as primitive gates, multiplexers, and registers that can be used by the design engineers. Each of these building block functions is referred to as a cell and the set of functions supported by the ASIC vendor is known as the cell library.
- Design engineers end up with a gate-level netlist, which describes the logic gates they wish to use and the connections between them. Special mapping, placement, and routing software tools are used to assign the logic gates to specific basic cells and define how the cells will be connected together.
- The results are used to generate the photo-masks that are in turn used to create the metallization layers that will link the components inside the basic cells and also link the basic cells to each other and to the device's inputs and outputs.

Digital Design using VHDL 20

Notes:

Gate arrays offer considerable cost advantages in that the transistors and other components are prefabricated, so only the metallization layers need to be customized.

The disadvantage is that most designs leave significant amounts of internal resources unutilized, the placement of gates is constrained, and the routing of internal tracks is less than optimal. All of these factors negatively impact the performance and power consumption of the design.



ASICs

➤ **Standard cell:**

- The ASIC vendor defines the cell library that can be used by the design engineers.
- The vendor also supplies hard-macro and soft-macro libraries, which include elements such as processors, communication functions, and a selection of RAM and ROM functions. Design engineers may decide to reuse previously designed functions or to purchase blocks of IP.
- Design engineers end up with a gate-level netlist, which describes the logic gates they wish to use and the connections between them.
- Unlike gate arrays, standard cell devices do not use the concept of a basic cell, and no components are prefabricated on the chip.

Digital Design using VHDL 21

ASICs

➤ **Standard cell:**

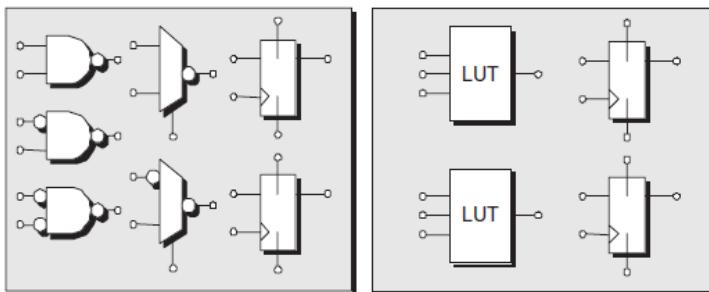
- Special tools are used to place each logic gate individually in the netlist and to determine the optimum way in which the gates are to be routed (connected together).
- The results are used to create custom photo-masks for every layer in the device's fabrication.
- The standard cell concept allows each logic function to be created using the minimum number of transistors with no redundant components.

Digital Design using VHDL 22

ASICs

➤ **Structured ASICs:**

- Each device commences with a fundamental element called a module or tiles.
- This element may contain a mixture of prefabricated generic logic (implemented either as gates, multiplexers, or a lookup table), one or more registers, and possibly a little local RAM.



(a) Gate, mux, and flop-based

(b) LUT and flop-based

Digital Design using VHDL 23

ASICs

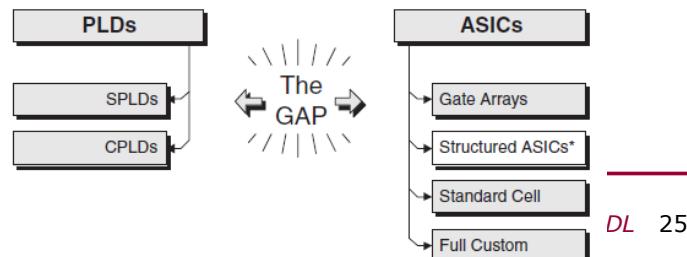
➤ **Structured ASICs:**

- An array of these elements is then prefabricated across the face of the chip.
- Also prefabricated (typically around the edge of the device) are functions like RAM blocks, clock generators,...etc.
- The idea is that the device can be customized using only the metallization layers just like gate array.
- The difference is that, due to the greater sophistication of the structured ASIC tile, most of the metallization layers are also predefined. Thus, many structured ASIC architectures require the customization of only two or three metallization layers.
- This reduces the time and costs associated with creating the remaining photo-masks used to complete the device.

Digital Design using VHDL 24

FPGAs

- Around the beginning of the 1980s, it became apparent that there was a gap in the digital IC continuum.
- At one end, there were programmable devices like SPLDs and CPLDs, which were highly configurable and had fast design and modification times, but which couldn't support large or complex functions.
- At the other end of the spectrum were ASICs which could support extremely large and complex functions, but they were expensive and time-consuming to design and once a design had been implemented as an ASIC it was effectively frozen in silicon.

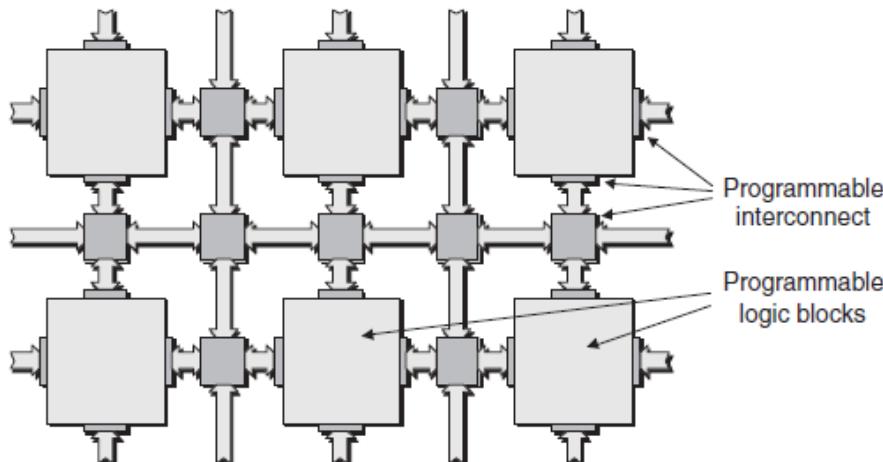


FPGAs

- In order to address this gap, Xilinx developed a new class of IC called a field-programmable gate array, or FPGA, which they made available to the market in 1984.
- FPGAs occupy a middle ground between PLDs and ASICs because their functionality can be customized in the field like PLDs, but they can contain millions of logic gates and be used to implement extremely large and complex functions that previously could be realized only using ASICs.
- The cost of an FPGA design is much lower than that of an ASIC (ASIC components are much cheaper in large production). At the same time, implementing design changes is much easier in FPGAs, and the time-to-market for such designs is much faster.

FPGA Architecture

- FPGA is based on the concept of a programmable logic blocks surrounded by programmable interconnects.



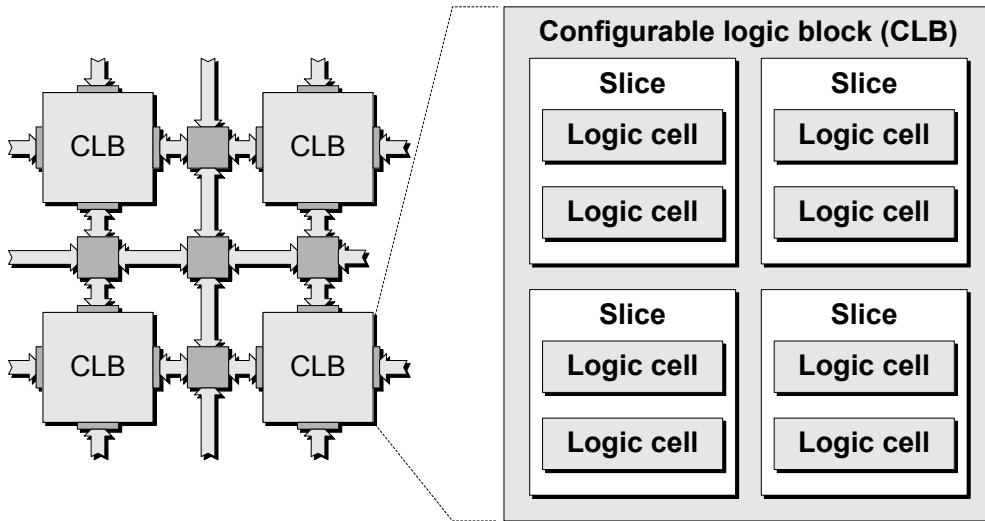
Digital Design using VHDL 27

FPGA Architecture

- The detailed architecture differs from one vendor to another.
- We will mainly concern Xilinx FPGAs Architecture.
- In Xilinx FPGAs the programmable logic blocks are called *configurable logic block (CLB)* while Altera called it *logic array block (LAB)*.
- CLB consists of two or four slices.
- Each slice contains two logic cells (LC).
- Logic cells are the core building block in modern FPGAs.
- Altera called it logic element (LE).

Digital Design using VHDL 28

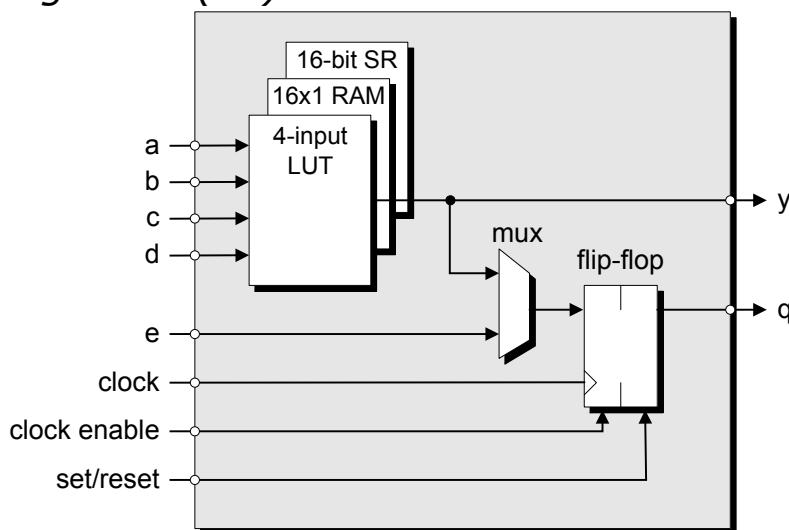
FPGA Architecture



Digital Design using VHDL 29

FPGA Architecture

➤ Simplified Logic Cell (LC):



Digital Design using VHDL 30

FPGA Architecture

➤ *Simplified Logic Cell (LC):*

- The register can be configured to act as a flip-flop, or as a latch.
- The polarity of the clock (rising-edge triggered or falling-edge triggered) can be configured, as can the polarity of the clock enable and set/reset signals (active-high or active-low).
- In addition to the LUT, MUX, and register, the LC also contains a smattering of other elements, including some special fast carry logic for use in arithmetic operations.

Digital Design using VHDL 31

FPGA Architecture

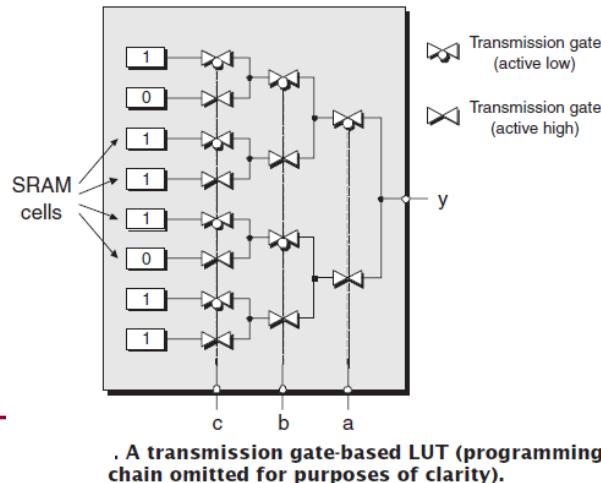
- This type of logic-block hierarchy— LC→ Slice (with two LCs)→ CLB (with four slices)—is that it is complemented by an equivalent hierarchy in the interconnect.
- There is fast interconnect between the LCs in a slice, then slightly slower interconnect between slices in a CLB, followed by the interconnect between CLBs.
- All these interconnects are programmable interconnects.

Digital Design using VHDL 32

FPGA Architecture

➤ Simplified Logic Cell (LC):

- LUT is formed from SRAM cells and a multiplexing circuit to route one of the SRAM cells to the output of the LUT based on the values of the LUT inputs.
- Ex: 3 i/p LUT
- SRAM cells are also connected through a chain for configuration.



33

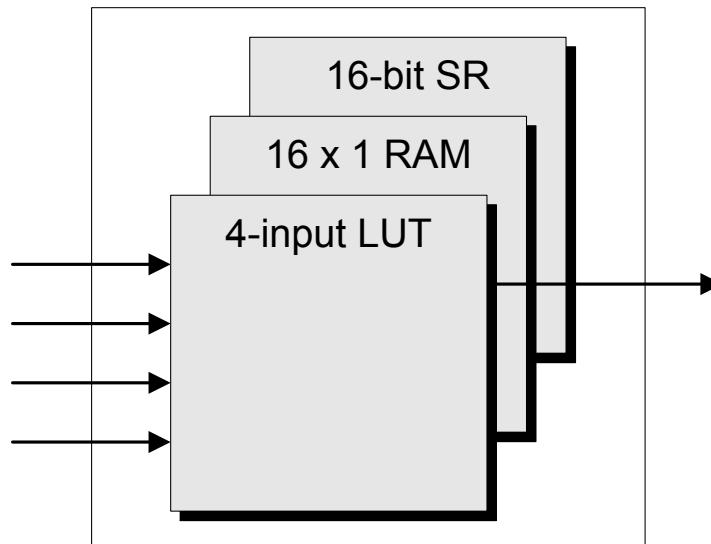
Notes:

If a transmission gate is enabled (active), it passes the signal seen on its input through to its output. If the gate is disabled, its output is electrically disconnected from the wire it is driving.

The transmission gate symbols shown with a small circle (called a “bubble”) indicate that these gates will be activated by a logic 0 on their control input. By comparison, symbols without bubbles indicate that these gates will be activated by a logic 1. Based on this understanding, it’s easy to see how different input combinations can be used to select the contents of the various SRAM cells.

FPGA Architecture

- Xilinx LUT is a multipurpose LUT.

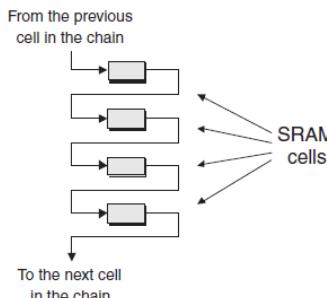


Digital Design using VHDL 34

FPGA Architecture

- Xilinx multipurpose LUT:

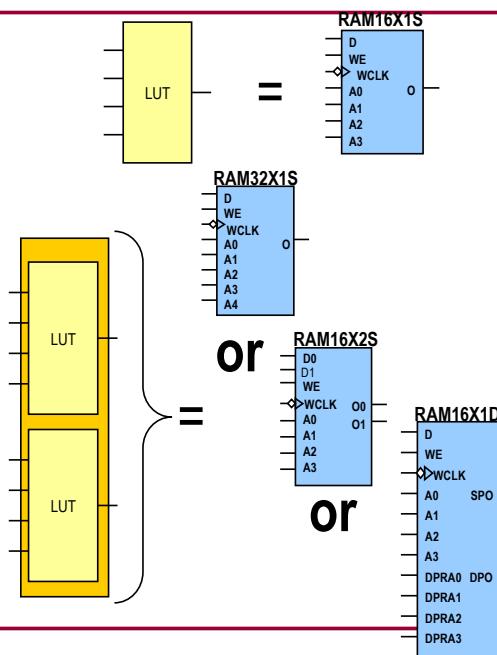
- In addition to its primary role as a lookup table, the cells forming the LUT can be used as a small block of RAM (the 16 cells forming a 4-input LUT, for example, could be cast in the role of a 16×1 RAM).
- This is referred to as distributed RAM because
 - (a) the LUTs are distributed across the surface of the chip
 - (b) this differentiates it from the larger chunks of block RAM
- some vendors allow the SRAM cells forming a LUT to be treated independently and to be used in the form of a shift register.



5

FPGA Architecture

- Distributed RAM:
 - A LUT equals 16x1 RAM
 - It can Implement Single and Dual-Ports
 - LUTs can be cascaded to increase RAM size



Digital Design using VHDL 36

FPGA Architecture

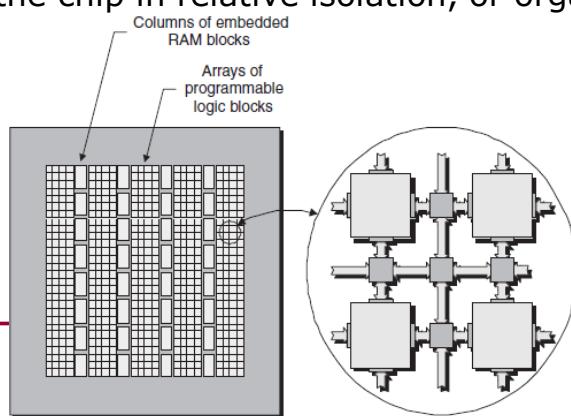
- Other FPGA Building Blocks
 - In addition to CLBs, FPGA can contain other embedded blocks which vary from one FPGA to another.
 - This contains embedded RAM, multipliers , adders, MACs and processor cores. This also contain Clock trees, Clock manager and general purpose I/O.

Digital Design using VHDL 37

FPGA Architecture

➤ Embedded RAM:

- A lot of applications require the use of memory, so FPGAs now include relatively large chunks of embedded RAM called block RAM.
- Depending on the architecture of the component, these blocks might be positioned around the device, scattered across the face of the chip in relative isolation, or organized in columns.

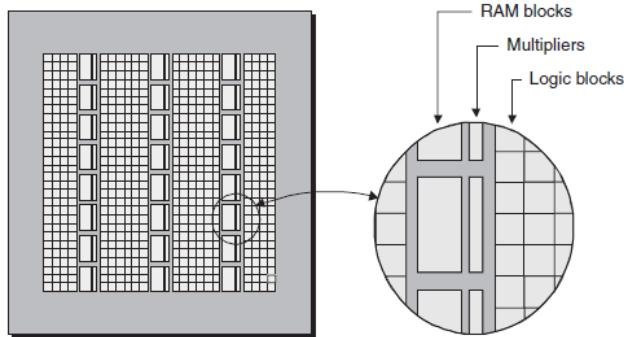


VHDL 38

FPGA Architecture

➤ Embedded Multiplier:

- Some functions, like multipliers, are inherently slow if they are implemented by connecting a large number of programmable logic blocks together.
- Since these functions are required by a lot of applications, many FPGAs incorporate special hardwired multiplier blocks.
- These are typically located in close to the embedded RAM blocks.

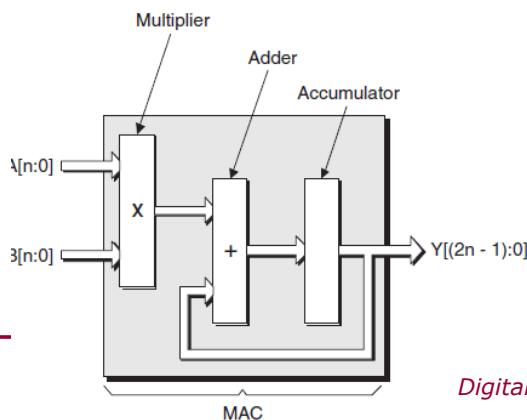


In using VHDL 39

FPGA Architecture

➤ Embedded MACs:

- One operation that is very common in DSP-type applications is called a multiply-and-accumulate (MAC).
- As name would suggest, this function multiplies two numbers together and adds the result to a running total stored in an accumulator.



Digital Design using VHDL 40

Notes:

If the FPGA you are working with supplies only embedded multipliers, you will have to implement this function by combining the multiplier with an adder formed from a number of programmable logic blocks, while the result is stored in some associated flip-flops, in a block RAM, or in a number of distributed RAMs. Life becomes a little easier if the FPGA also provides embedded adders, and some FPGAs provide entire MACs as embedded functions.

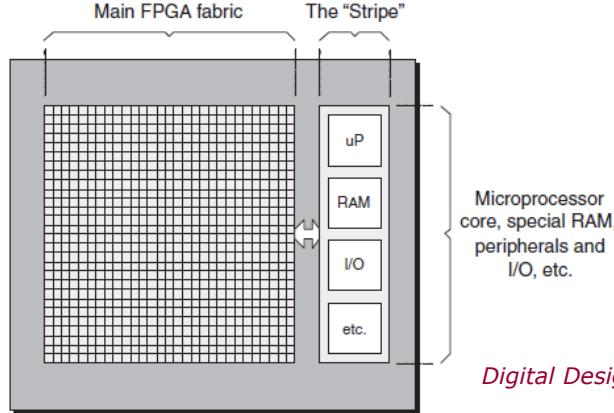
FPGA Architecture

- Embedded processors cores:
 - Majority of designs make use of microprocessors in one form or another.
 - Recently, FPGAs that contain one or more embedded microprocessors, which are typically referred to as microprocessor cores.
 - In this case, it often makes sense to move all of the tasks that used to be performed by the external microprocessor into the internal core.
 - This provides a number of advantages, not the least being that it saves the cost of having two devices; it eliminates large numbers of tracks, pads, and pins on the circuit board; and it makes the board smaller and lighter.
 - These cores can be soft cores or hard cores.

Digital Design using VHDL 41

FPGA Architecture

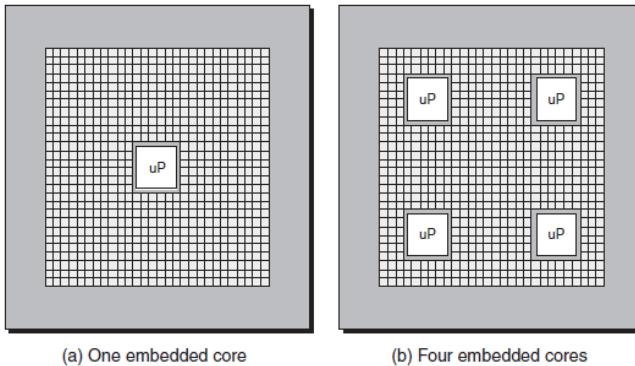
- Hard microprocessor cores:
 - A hard microprocessor core is implemented as a dedicated, predefined block.
 - There are two main approaches for integrating such a core into the FPGA.
 - The first is to locate it in a strip to the side of the main FPGA fabric.



Digital Design using VHDL 42

FPGA Architecture

- Hard microprocessor cores:
 - The second is to embed one or more microprocessor cores directly into the main FPGA fabric.
 - One, two, and even four core implementations are currently available.



Digital Design using VHDL 43

FPGA Architecture

- Soft microprocessor cores:
 - It implements a microprocessor core by configuring a group of programmable logic blocks to act as a microprocessor.
 - Soft cores are simpler (more primitive) and slower than their hard-core counterparts.
 - They have the advantage that you only need to implement a core if you need it and also that you can instantiate as many cores as you require until you run out of resources in the form of programmable logic blocks.

Digital Design using VHDL 44

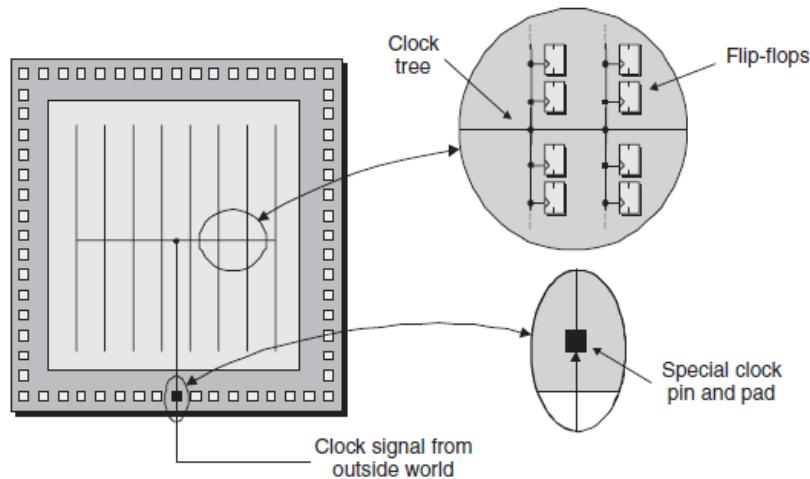
FPGA Architecture

- Clock trees and clock managers:
 - All of the synchronous elements inside an FPGA—for example, the registers configured to act as flip-flops inside the programmable logic blocks—need to be driven by a clock signal.
 - Such a clock signal typically originates in the outside world, comes into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate registers.

Digital Design using VHDL 45

FPGA Architecture

- Clock trees :



Digital Design using VHDL 46

FPGA Architecture

➤ Clock trees :

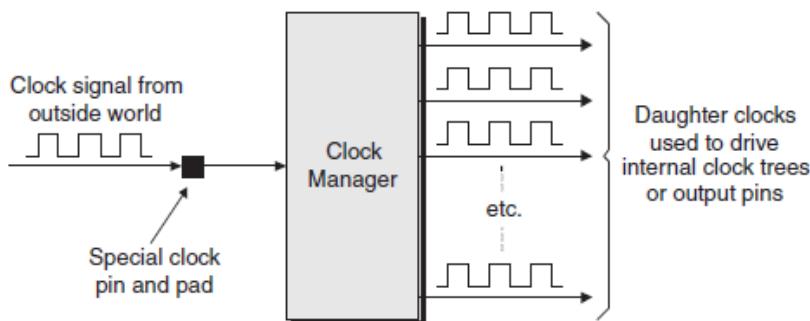
- This is called a “clock tree” because the main clock signal branches again and again (the flip-flops can be considered to be the “leaves” on the end of the branches).
- This structure is used to ensure that all of the flip-flops see their versions of the clock signal as close together as possible.
- If the clock were distributed as a single long track driving all of the flip-flops one after another, then the flip-flop closest to the clock pin would see the clock signal much sooner than the one at the end of the chain (skew).
- Even when using a clock tree, there will be a certain amount of skew.

Digital Design using VHDL 47

FPGA Architecture

➤ Clock managers :

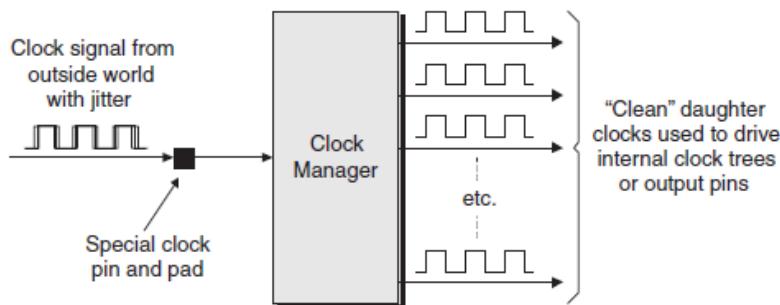
- Instead of configuring a clock pin to connect directly into an internal clock tree, that pin can be used to drive a special hard-wired function (block) called a clock manager that generates a number of daughter clocks.



Digital Design using VHDL 48

FPGA Architecture

- Clock managers :
- clock managers may support only a subset of the following features:
- Jitter removal

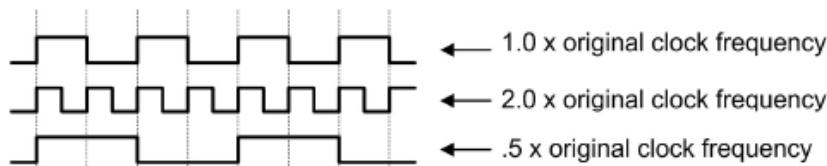


Digital Design using VHDL 49

FPGA Architecture

- Clock managers :
- Frequency synthesis

It may be that the frequency of the clock signal being presented to the FPGA from the outside world is not exactly what the design engineers wish for. In this case, the clock manager can be used to generate daughter clocks with frequencies that are derived by multiplying or dividing the original signal.



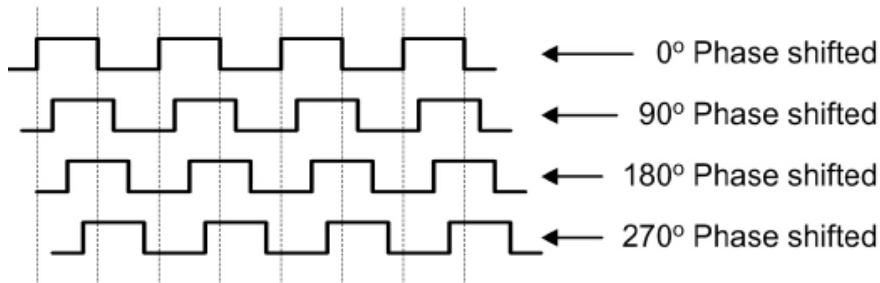
Digital Design using VHDL 50

FPGA Architecture

➤ Clock managers :

- Phase shifting

Certain designs require the use of clocks that are phase shifted (delayed) with respect to each other.

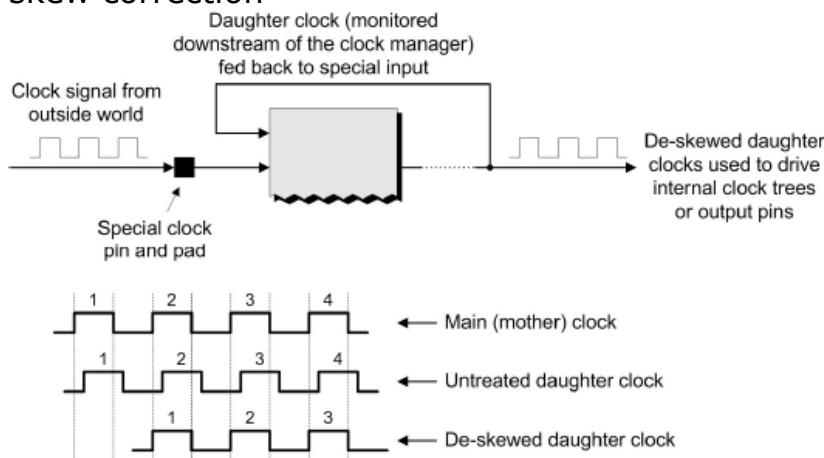


Digital Design using VHDL 51

FPGA Architecture

➤ Clock managers :

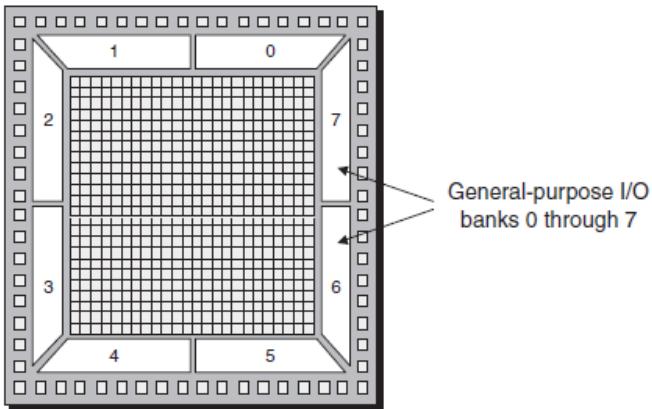
- Auto-skew correction



Digital Design using VHDL 52

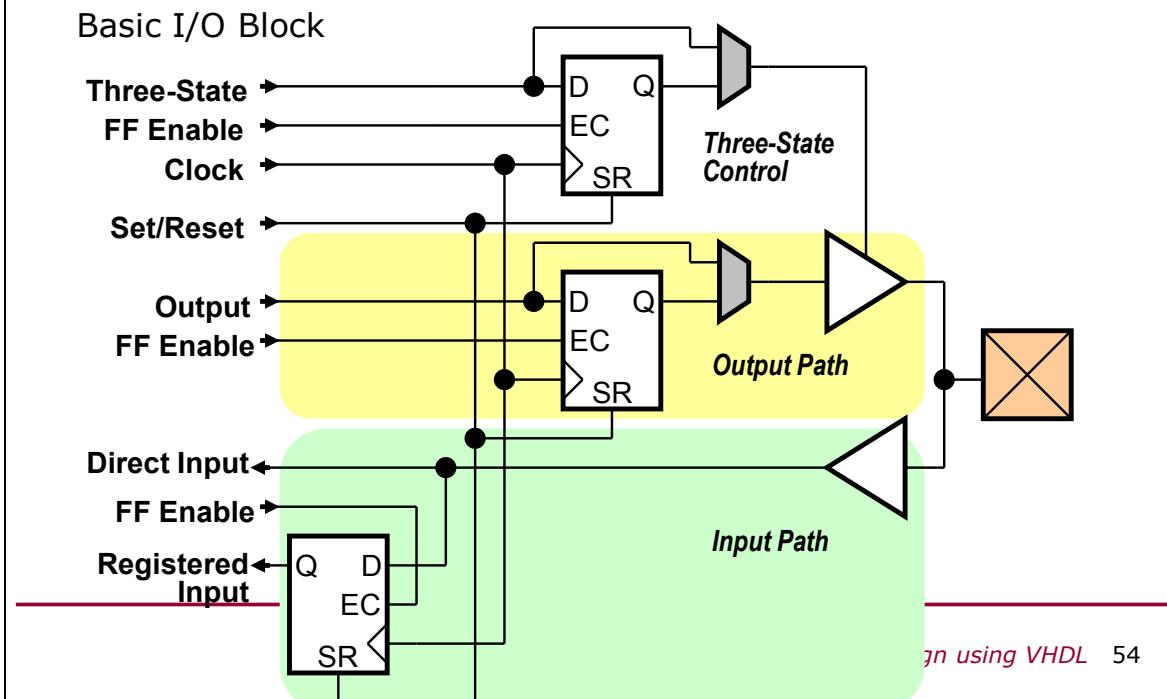
FPGA Architecture

- General-purpose I/O
- Today's FPGA packages can have large number of pins, which are arranged as an array across the base of the package.



Digital Design using VHDL 53

FPGA Architecture



Design using VHDL 54

Notes:

I/O Block provides interface between the package pins and CLBs. Each IOB can work as uni-directional or bi-directional I/O. Outputs can be forced into High Impedance. Inputs and outputs can be registered and Inputs can be delayed.



Who are the vendors ?

- FPGA Vendors:

Company	Web site	Comment
Actel Corp.	www.actel.com	FPGAs
Altera Corp.	www.altera.com	FPGAs
Anadigm Inc.	www.anadigm.com	FPAAs
Atmel Corp.	www.atmel.com	FPGAs
Lattice Semiconductor Corp.	www.latticesemi.com	FPGAs
Leopard Logic Inc.	www.leopardlogic.com	Embedded FPGA cores
QuickLogic Corp.	www.quicklogic.com	FPGAs
Xilinx Inc.	www.xilinx.com	FPGAs

Digital Design using VHDL 55

Who are the vendors ?

- Full Line EDA Vendors:

Company	Web site	Comment
Altium Ltd.	www.altium.com	System-on-FPGA hardware-software design environment
Cadence Design Systems	www.cadence.com	FPGA design entry and simulation (OEM synthesis)
Mentor Graphics Corp.	www.mentor.com	FPGA design entry, simulation, and synthesis
Synopsys Inc.	www.synopsys.com	FPGA design entry, simulation, and synthesis

Digital Design using VHDL 56

Who are the vendors ?

- Free EDA Tools:

Xilinx and Altera also offer free versions of their ISE and Quartus-II FPGA design environments, respectively.

Company	Website	Comment
Altera Corp.	www.altera.com	Synthesis and place-and-route
Xilinx Inc.	www.xilinx.com	Synthesis and place-and-route

References

1. PONG P. CHU, "RTL HARDWARE DESIGN USING VHDL Coding for Efficiency, Portability, and Scalability".
2. Pedroni, Volnei A., "Circuit design with VHDL".
3. Clive "Max" Maxfield, "The Design Warrior's Guide to FPGAs".
4. Peter J. Ashenden, "The Designer's Guide to VHDL".
5. Digital ASIC Group - Lund University, "Digital ASIC Design, A Tutorial on the Design Flow".
6. Doulos, "Comprehensive VHDL, Training Workbook".