Digital Design Through Verilog Hdl

IV-YEAR II-SEM

Dept. of Electronics and Communication Engineering

LIST OF CONTENTS

GEETHANJALI COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF *Electrical and Electronics Engineering*

(Name of the Subject / Lab Course) : DIGITAL DESIGN THROUGH VERILOG HDL

(JNTU CODE -58033) **Programme : *UG***

Branch: ECE	**Version No : *04***
Year: IV-Year Updated on :26 -11-15	
Semester: II-sem	**No.of pages :186**

Classification status (Unrestricted / ~~Restricted~~)

Distribution List :

Prepared by :	**1) Name :JUGAL KISHORE BHANDARI**	**1) Name :**
	2) Sign :	**2) Sign :**
	3) Design : Asst. Prof.	**3) Design :**
	4) Date :	**4) Date :**

Verified by : 1) Name :	*** For Q.C Only. 1) Name :**
2) Sign :	**2) Sign :**
3) Design :	**3) Design :**
4) Date :	**4) Date :**

Approved by : (HOD) 1) Name : Dr. P. Srihari

2) Sign :

3) Date :

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

DIGITAL DESIGN THROUGH VERILOG (ELECTIVE – III)

UNIT I

INTRODUCTION TO VERILOG:

Verilog as HDL, Levels of Design Description, Concurrency, Simulationand Synthesis, Functional Verification, System Tasks, Programming Language Interface (PLI), Module,Simulation and Synthesis Tools.

UNIT II

LANGUAGE CONSTRUCTS AND CONVENTIONS:

Introduction, Keywords, Identifiers, White Space Characters, Comments, Numbers, Strings, Logic Values, Strengths, Data Types, Scalars and Vectors, Parameters, Memory, Operators.

UNIT III

GATE LEVEL MODELING:

Introduction, AND Gate Primitive, Module Structure, Other Gate Primitives,Illustrative Examples, Tri-State Gates, Array of Instances of Primitives, Additional Examples, Design of Flip-flopswith Gate Primitives, Delays, Strengths and Contention Resolution, Net Types, Design of BasicCircuits.

UNIT IV

BEHAVIORAL MODELING:

Introduction, Operations and Assignments, Functional Bifurcation, *Initial*Construct, *Always* Construct, Examples, Assignments with Delays, *Wait* construct, Multiple Always Blocks,Designs at Behavioral Level, Blocking and Non blocking Assignments, The *case* statement, Simulation Flow.*if* and *if-else* constructs, *assign-deassign* construct, *repeat* construct, *for* loop, the *disable* construct, *while*loop, *forever* loop, parallel blocks, *force-release* construct, Event.

UNIT V

MODELING AT DATA FLOW LEVEL:

Introduction, Continuous Assignment Structures, Delays and Continuous Assignments, Assignment to Vectors, Operators.

SWITCH LEVEL MODELING:

Basic Transistor Switches, CMOS Switch, Bi-directional Gates, Time Delays with Switch Primitives, Instantiation with 'Strengths' and 'Delays', Strength Contention with Trireg Nets.

UNIT VI

SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES:

Introduction, Parameters, Path Delays, Module Parameters, System Tasks and Functions, File-Based Tasks and Functions, Compiler Directives, Hierarchical Access, User- Defined Primitives (UDP).

UNIT VII

SEQUENTIAL CIRCUIT DESCRIPTION:

Sequential models – feedback model, capacitive model, implicit model, basic memory components, functional register, static machine coding, sequential synthesis.

UNIT VIII

COMPONENT TEST AND VERIFIACTION:

Test bench – combinational circuit testing, sequential circuit testing, test bench techniques, design verification, assertion verification.

COURSE DESCRIPTION:

This course covers the use of Verilog and Systemverilog Languages (IEEE Std. 1800) for the design and development of digital integrated circuits, including mask-programmed integrated circuits (ASICs) and field programmable devices (FPGAs). Hierarchical top down vs. bottom up design, synthesizable vs. non-synthesizable code, design scalability and reuse, verification, hardware modeling, simulation system tasks, compiler directives and subroutines are all covered and illustrated with design examples.

3 Credits are allocated for this subject.

3. VISION OF THE DEPARTMENT:

To impart quality technical education in Electronics and Communication Engineering emphasizing analysis, design/synthesis and evaluation of hardware/embedded software using various Electronic Design Automation (EDA) tools with accent on creativity, innovation and research thereby producing competent engineers who can meet global challenges with societal commitment.

4. MISSION OF THE DEPARTMENT:

I. To impart quality education in fundamentals of basic sciences, mathematics, electronics and communication engineering through innovative teaching-learning processes.

II. To facilitate Graduates define, design, and solve engineering problems in the field of Electronics and Communication Engineering using variousElectronic Design Automation (EDA) tools.

III. To encourage research culture among faculty and students thereby facilitating them to be creative and innovative through constant interaction with R & D organizations and Industry.

IV. To inculcate teamwork, imbibe leadership qualities, professional ethics and social responsibilities in students and faculty.

5. Program Educational Objectives of B. Tech (ECE) Program:

I. To prepare students with excellent comprehension of basic sciences, mathematics and engineering subjects facilitating them to gain employment or pursue postgraduate studies with an appreciation for lifelong learning.

II. To train students with problem solving capabilities such as analysis and design with adequate practical skills wherein they demonstrate creativity and innovation that would enable them to develop state of the art equipment and technologies of multidisciplinary nature for societal development.

III. To inculcate positive attitude, professional ethics, effective communication and interpersonal skills which would facilitate them to succeed in the chosen profession exhibiting creativity and innovation through research and development both as team member and as well as leader.

Program Outcomes of B.Tech ECE Program:

1. An ability to apply knowledge of Mathematics, Science, and Engineering to solve complex engineering problems of Electronics and Communication Engineering systems.
2. An ability to model, simulate and design Electronics and Communication Engineering systems, conduct experiments, as well as analyze and interpret data and prepare a report with conclusions.
3. An ability to design an Electronics and Communication Engineering system, component, or process to meet desired needs within the realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability and sustainability.
4. An ability to function on multidisciplinary teams involving interpersonal skills.
5. An ability to identify, formulate and solve engineering problems of multidisciplinary nature.
6. An understanding of professional and ethical responsibilities involved in the practice of Electronics and Communication Engineering profession.
7. An ability to communicate effectively with a range of audience on complex engineering problems of multidisciplinary nature both in oral and written form.
8. The broad education necessary to understand the impact of engineering solutions in a global, economic, environmental and societal context.
9. Recognition of the need for, and an ability to engage in life-long learning and acquire the capability for the same.
10. A knowledge of contemporary issues involved in the practice of Electronics and Communication Engineering profession
11. An ability to use the techniques, skills and modern engineering tools necessary for engineering practice.
12. An ability to use modern Electronic Design Automation (EDA) tools, software and electronic equipment to analyze, synthesize and evaluate Electronics and Communication Engineering systems for multidisciplinary tasks.
13. Apply engineering and project management principles to one’s own work and also to manage projects of multidisciplinary nature.

6. Course objectives and outcomes

Course Objectives:

- The ability to code and simulate any digital function in Verilog HDL.
- Know the difference between synthesizable and non-synthesizable code.
- Understand library modeling, behavioral code and the differences between them.
- Understand the differences between simulator algorithms.
- Learn good coding techniques per current industrial practices.
- Understand logic verification using Verilog simulation.

Course outcomes:

Subject: Digital Design through Verilog HDL

.

CO 1: Students will have an ability to describe Verilog hardware description languages (HDL).

CO 2: Students will be able to Design Digital Circuits in Verilog HDL.

CO 3: Ability to write behavioral models of digital circuits.

CO 4: Ability to write Register Transfer Level (RTL) models of digital circuits.

CO 5: Ability to verify behavioral and RTL models.

CO 6: Students will have an ability to describe standard cell libraries and FPGAs.

CO 7: To Synthesize RTL models to standard cell libraries and FPGAs.

CO 8: To Implement RTL models on FPGAs and Testing & Verification.

Brief note on importance of course

This course is intended to provide a thorough coverage of Verilog HDL concepts based on fundamental principles of VLSI Design.

1) This is the basic fundamental subject for the programming of the digital Electronics.
2) This subject is required to understand the programming of the combinational and sequential circuit designs.

3) By studying this subject, the students can design and understand digital systems and its importance.
4) Large and complicated digital circuits can be incorporated into hardware by using Verilog, a hardware description language (HDL). Design through Verilog HDL affords novices the opportunity to perform all of these tasks, while also offering seasoned professionals a comprehensive resource on this dynamic tool.
5) Describing a design using Verilog is only half the story: writing test-benches, testing a design for all its desired functions, and how identifying and removing the faults remain significant challenges. Design Through Verilog HDL addresses each of these issues concisely and effectively.
6) The students logical thinking capability will be improved which will help in placements and in their future technical assignments.

8. Prerequisites:

1) Concepts of switching theory and logic design.
2) A basic understanding of digital hardware design and verification.

9. Instructional objectives and Learning outcomes:

UNIT I

Introduction to Verilog HDL:

After completion of this unit, students are able to:

1. Students understand the importance of HDL (Hardware Descriptive Language) and apply the knowledge ofBoolean algebra to design and development digital Systems.
2. Understand the difference between concurrent and sequential programming.
3. Issues related to simulation and synthesis models.

UNIT II

Language Constructs and Conventions:

After completion of this unit, students are able to:

1. Knowledge of language constructs
2. Pertaining to Semantic and syntactical errors in programming using HDL

3. Limitation of HDL

UNIT III

Gate Level Modeling:

After completion of this unit, students are able to:

1. Student will learn conventional structural modeling of digital systems.
2. Learn to model language defined primitive gates
3. Understand importance of component structure in Verilog.
4. Learn Hierarchical digital system building

UNIT IV

Modeling at Dataflow Level:

After completion of this unit, students are able to:

1. Continuous assignment operator based model construction will be learnt.

UNIT V

After completion of this unit, students are able to:

Behavioral Modeling:

1. Students will be familiarized to high level abstraction of digital systems with behavioral modeling of systems.
2. RTL modeling of digital systems
3. Will be made familiar to behavioral constructs like ‘always’ ,’initial’,’if’, ‘if-else’,’case’..etc
4. Register and array modeling.

Switch Level Modeling:

5. Students will learn low-level abstraction of digital systems.
6. Switch level primitives will be learnt
7. Made to familiarize to different strengths of logic values

UNIT VI

System Tasks, Functions and Compiler Directives:

After completion of this unit, students are able to:

1. Understand the importance of system tasks and functions
2. Understand compiler directives.
3. Understand user defined primitives and learn to model systems using UDP
4. Learn the intricacies associated with usage of functions and tasks in packages
5. Learn package declaration and package usage in project building

UNIT VII

Sequential Circuit Description:

After completion of this unit, students are able to:

1. Learn to model Sequential circuits at higher level of abstraction using RTL modeling
2. Will be able to design static and dynamic memories
3. Will learn to model in behavioral style of binary encoding and one hot encoding

UNIT VIII

Component Test and Verification:

After completion of this unit, students are able to:

1. Students understand Test bench generation
2. Producing of test vectors to test the digital systems at higher level of abstraction.

10. Mapping of Course outcomes with Programme outcomes:

*When the course outcome weightage is < 40%, it will be given as moderately correlated (1).

*When the course outcome weightage is >40%, it will be given as strongly correlated (2).

POs	1	2	3	4	5	6	7	8	9	10	11	12	13
Digital Design Through Verilog HDL	**2**	**2**	**2**	**2**	**2**	**1**		**1**	**2**	**2**	**2**		**1**
CO 1: Students will be able to describe Verilog hardware description languages (HDL).	**2**	**2**	**2**	**2**	**2**	**1**			**2**	**2**	**2**		**1**
CO 2: Students will be able to Design Digital Circuits in Verilog HDL.	**2**	**2**	**2**	**2**	**2**				**2**	**2**	**1**		
CO3: Ability to write behavioral models of digital circuits.	**1**	**1**		**2**				**1**	**2**	**1**	**1**		
CO4: Ability to write Register Transfer Level (RTL) models of digital circuits.	**1**	**2**	**2**		**2**			**1**	**2**	**1**	**1**		
CO5: Ability to verify behavioral and RTL models.	**2**	**2**	**2**	**2**	**2**				**2**	**1**	**1**		
CO 6: Students will have an abilityto describe standard cell libraries and FPGAs.	**2**	**2**	**2**		**2**				**2**	**1**	**1**		
CO 7: To Synthesize RTL models to standard cell libraries and FPGAs.	**1**	**2**	**2**		**2**					**1**	**1**		
CO 8: To Implement RTL models on FPGAs and Testing & Verification.	**1**	**2**	**2**		**2**			**1**	**2**	**1**	**2**		**1**

11. Class Time Table:

12. Individual Time Table:

13. Micro Plan With Dates & Closure Report:

SL. NO	Unit No.	Total no. of periods	Date	Topics to be covered in one lecture	Regular/ Additional	Teaching aids used LCD/ OHP/ BB	Remarks
1	**I**	4	11/12/15	Verilog as HDL, Levels of Design Description, Concurrency, Verilog as HDL, Levels of Design Description	Regular	OHP,BB	
2			11/12/15	Concurrency Simulation and Synthesis, Functional Verification, System Tasks	Regular	OHP,BB	
3			12/12/15	Programming Language Interface (PLI), Module, Simulation and Synthesis Tools	Regular	OHP,BB	
4			12/12/15	Tutorial class-1			

5	**II**	3	18/12/15	LANGUAGE CONSTRUCTS AND CONVENTIONS:Introduction, KeywordsIdentifiers, White Space Characters, Comments, Numbers, Strings, Logic Values, Strengths	Regular	BB	
6			18/12/15	Data Types, Scalars and Vectors, Parameters, Memory, Operators, System Tasks, Exercises.	Regular	OHP,BB	
7			19/12/15	Tutorial class-2		BB	
8	**III**	4	26/12/15	GATE LEVEL MODELING: Introduction, AND Gate Primitive, Module Structure	Regular	OHP,BB	
9			26/12/15	Other Gate Primitives, Illustrative Examples, Tri-State Gates	Regular	OHP,BB	
10			02/01/16	Array of Instances of Primitives, Additional Examples	Regular	OHP,BB	
11			02/01/16	Design of Flip-flops with Gate Primitives, Delays	Regular	BB	
12			08/01/16	Strengths and Contention Resolution, Net Types, Design of Basic Circuits.	Regular	BB	
13			08/01/16	Verilog designs for various rounding methods	**Additional**	OHP,BB	
14			08/01/16	Tutorial class-3			
15	**IV**	4	22/01/16	BEHAVIORAL MODELING: Introduction, Operations and Assignments, Functional Bifurcation, *Initial* Construct, *Always* Construct, Examples	Regular	OHP,BB	
16			22/01/16	Assignments with Delays, *Wait* construct, Multiple Always Blocks, Designs at Behavioral Level	Regular	OHP,BB	
17			29/01/16	Blocking and Non blocking Assignments, The *case* statement, Simulation Flow. *if* and *if-else* constructs		BB	

18			29/01/16	*Assign-de-assign* construct, *repeat* construct, *for* loop, the *disable* construct,*while* loop, *forever* loop, parallel blocks, *force-release* construct, Event.		BB	
19			05/02/16	Tutorial class-4		BB	
20			05/02/16	Solving university papers			
21			06/02/16	Assignment test-1			
22				Mid test-1			
23	**V**	6	13/02/16	MODELING AT DATAFLOW LEVEL: Introduction, Continuous Assignment Structures, Delays and Continuous Assignments	Regular	OHP,BB	
24			13/02/16	Assignment to Vectors, Operators.	Regular	OHP,BB	
25			19/02/16	SWITCH LEVEL MODELING: Introduction Basic Transistor Switches, CMOS Switch, Bi-directional Gates	Regular	OHP,BB	
26			19/02/16	Time Delays with Switch Primitives, Instantiations with Strengths and Delays	Regular	OHP,BB	
27			20/02/16	Strength Contention with Trireg Nets.	Regular	BB	
28			20/02/16	Combinational synthesis	**Additional**	BB	
29	**VI**	4	26/02/16	Introduction, Parameters, Path Delays, Module Parameters, System Tasks and Functions	Regular	OHP,BB	
30			26/02/16	File-Based Tasks and Functions, Compiler Directives, Hierarchical Access,	Regular	OHP,BB	
31			27/02/16	User- Defined Primitives (UDP)	Regular	OHP,BB	

32			27/02/16	Tutorial class- 5 & 6		BB	
33	**VII**	4	04/03/16	Sequential Models – FeedBack Model, Capacaitive Model, Implicit Model	Regular	OHP,BB	
34			04/03/16	Basic Memory Components, Functional Register	Regular	OHP,BB	
35			05/03/16	Static Machine Coding	Regular	OHP,BB	
36			05/03/16	Sequential Synthesis	Regular	OHP,BB	
37			11/03/16	Tutorial class – 7		BB	
38	**VIII**	4	12/03/16	Component Test and Verification: Test Bench – Combinational Circuit Testing	Regular	OHP,BB	
39			18/03/16	Test Bench – Sequential Circuit Testing	Regular	OHP,BB	
40			19/03/16	Test Bench Techniques	Regular	OHP,BB	
41			26/03/16	Design Verification	Regular	OHP,BB	
42			01/04/16	Assertion Verification	Regular	OHP,BB	
43			01/04/16	Tutorial Class – 8		BB	
44			02/04/16	Solving university papers		BB	
45			09/04/16	Assignment test-2			
46				Mid test-2			

Course closer Report on Digital Design Through Verilog HDL:

1.	No. of classes Planned to complete the course	:	36
2.	No.of classes conducted	:	ECE-A
			ECE-B
			ECE-C
3.	No. of students appeared the External Examination	:	ECE-A
			ECE-B
			ECE-C
4.	No. of students passed in the External Examination	:	ECE-A
			ECE-B
			ECE-C
5.	No. of Students failed in the External Examination	:	ECE-A
			ECE-B
			ECE-C
6.	Pass percentage of each section	:	ECE-A
			ECE-B
			ECE-C
7.	Maximum marks obtained in the Course	:	ECE-A
			ECE-B
			ECE-C

14. Detailed Notes:

1

INTRODUCTION TO VLSI DESIGN

1.1 INTRODUCTION

The word digital has made a dramatic impact on our society. More significant is a continuous trend towards digital solutions in all areas – from electronic instrumentation, control, data manipulation, signals processing, telecom-munications *etc*., to consumer electronics. Development of such solutions has been possible due to good digital system design and modeling techniques.

1.2 CONVENTIONAL APPROACH TO DIGITAL DESIGN

Digital ICs of SSI and MSI types have become universally standardized and have beenaccepted for use. Whenever a designer has to realize a digital function, he uses a standard set of ICs along with a minimal set of additional discrete circuitry. Consider a simple example of realizing a function as

$Q_{n+1} = Q_n + (A\ B)$

Here Q_n,A, and B are Boolean variables, with Q_n being the value of Q at the nth time step. Here $A\ B$ signifies the logical AND of A and B; the '+' symbol signifies the logical OR of the logic variables on either side. A circuit to realize the function is shown in Figure 1.1. The circuit can be realized in terms of two ICs – an A-O-I gate and a flip-flop. It can be directly wired up, tested, and used.

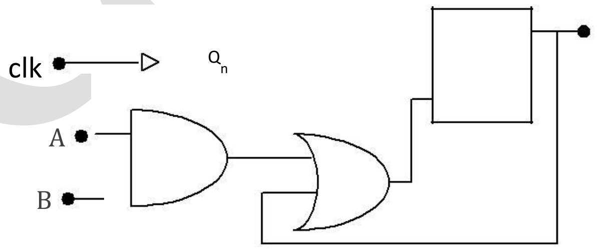


Figure 1.1 A simple digital circuit.

With comparatively larger circuits, the task mostly reduces to one of identifying the set of ICs necessary for the job and interconnecting; rarely does one have to resort to a microlevel design [Wakerly]. The accepted approach to digital design here is a mix of the top-down and bottom-up approaches as follows [Hill & Peterson]:

Decide the requirements at the system level and translate them to circuit requirements.

Identify the major functional blocks required like timer, DMA unit, register-file *etc*., say as in the design of a processor.

Whenever a function can be realized using a standard IC, use the same –for example programmable counter, mux, demux, *etc*.

Whenever the above is not possible, form the circuit to carry out the block functions using standard SSI – for example gates, flip-flops, *etc*.

Use additional components like transistor, diode, resistor, capacitor, *etc*., wherever essential.

Once the above steps are gone through, a paper design is ready. Starting with the paper design, one has to do a circuit layout. The physical location of all the components is tentatively decided; they are interconnected and the 'circuit-on-paper' is made ready. Once a paper design is done, a layout is carried out and a net-list prepared. Based on this, the PCB is fabricated, and populated and all the populated cards tested and debugged. The procedure is shown as a process flowchart in Figure 1.2.

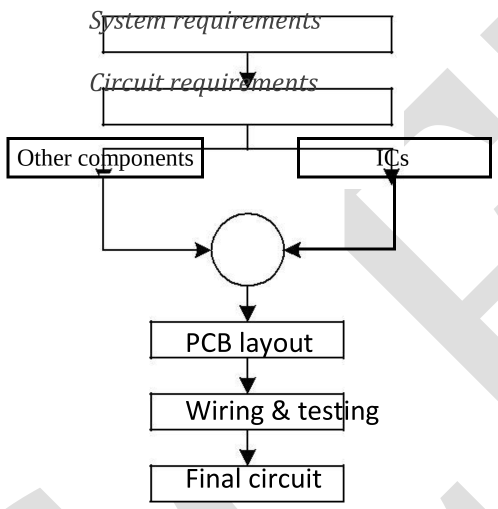


Figure 1.2 Sequence of steps in conventional electronic circuit design.

At the debugging stage one may encounter three types of problems:

Functional mismatch: The realized and expected functions are different. Onemay have to go through the relevant functional block carefully and locate any error logically. Finally the necessary correction has to be carried out in

hardware.

Timing mismatch: The problem can manifest in different forms. Onepossibility is due to the signal going through different propagation delays in two paths and arriving at a point with a timing mismatch. This can cause faulty operation. Another possibility is a race condition in a circuit involving asynchronous feedback. This kind of problem may call for elaborate debugging. The preferred practice is to do debugging at smaller module stages and ensuring that feedback through larger loops is avoided: It becomes

essential to check for the existence of long asynchronous loops.

Overload: Some signals may be overloaded to such an extent that the signaltransition may be unduly delayed or even suppressed. The problem manifests as reflections and erratic behavior in some cases (The signal has to be suitably buffered here.). In fact, overload on a signal can lead to timing mismatches.

The above have to be carried out after completion of the prototype PCB manufacturing; it involves cost, time, and also a redesigning process to develop a bugfree design.

1.3 VLSI DESIGN

The complexity of VLSIs being designed and used today makes the manual approach to design impractical. Design automation is the order of the day. With the rapid technological developments in the last two decades, the status of VLSI technology is characterized by the following [Wai-kai, Gopalan]:

A steady increase in the size and hence the functionality of the ICs.

A steady reduction in feature size and hence increase in the speed of operation
as well as gate or transistor density.

A steady improvement in the predictability of circuit behavior.

A steady increase in the variety and size of software tools for VLSI design.

The above developments have resulted in a proliferation of approaches to VLSI design. We briefly describe the procedure of automated design flow [Rabaey, Smith MJ]. The aim is more to bring out the role of a Hardware Description Language (HDL) in the design process. An abstraction based model is the basis of the automated design.

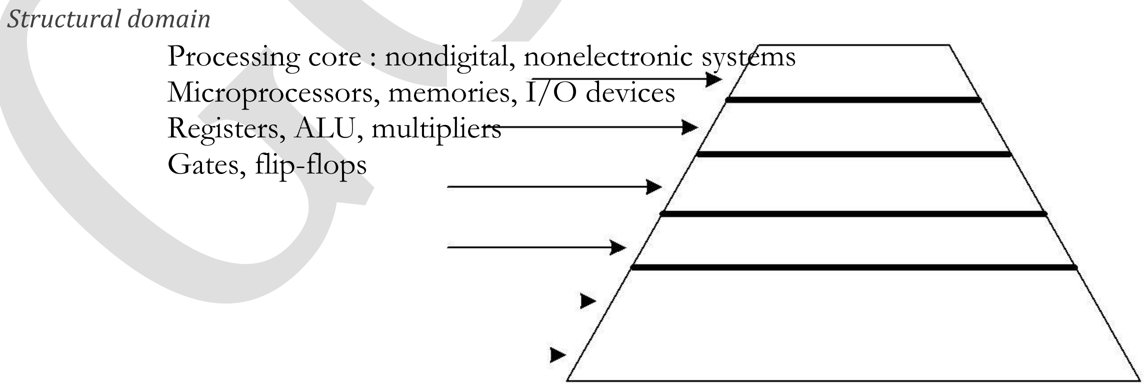
Abstraction Model

The model divides the whole design cycle into various domains (see Figure 1.3). With such an abstraction through a division process the design is carried out in different layers. The designer at one layer can function without bothering about the layers above or below. The thick horizontal lines separating the layers in the figure signify the compartmentalization. As an example, let us consider design at the gate level. The circuit to be designed would be described in terms of truth tables and state tables. With these as available inputs, he has to express them as Boolean logic equations and realize them in terms of gates and flip-flops. In turn, these form the inputs to the layer immediately below. Compartmentalization of the approach to design in the manner described here is the essence of abstraction; it is the basis for development and use of CAD tools in VLSI design at various levels.

The design methods at different levels use the respective aids such as Boolean equations, truth tables, state transition table, *etc*. But the aids play only a small role in the process. To complete a design, one may have to switch from one tool to another, raising the issues of tool compatibility and learning new environments.

1.4 ASIC DESIGN FLOW

As with any other technical activity, development of an ASIC starts with an idea and takes tangible shape through the stages of development as shown in Figure 1.4 and shown in detail in Figure 1.5. The first step in the process is to expand the idea in terms of behavior of the target circuit. Through stages of programming, the same is fully developed into a design description – in terms of well defined standard constructs and conventions.


Structural domain
Processing core : nondigital, nonelectronic systems
Microprocessors, memories, I/O devices
Registers, ALU, multipliers
Gates, flip-flops

Transistors, L, R, C — Circuit (differential equations)

Geometric objects — Silicon (*none*)

Figure 1.3 Design domain and levels of abstraction.

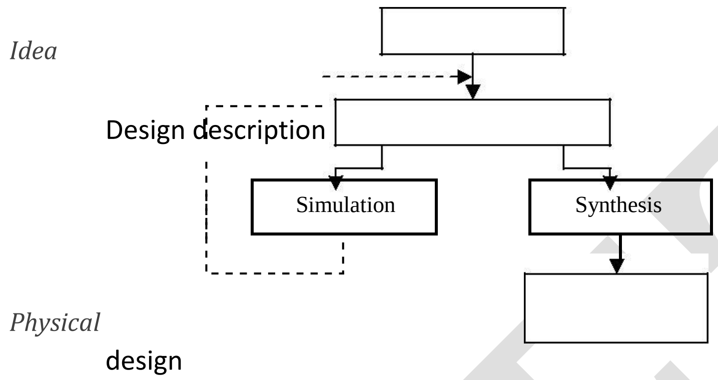


Figure 1.4 Major activities in ASIC design.

The design is tested through a simulation process; it is to check, verify, and ensure that what is wanted is what is described. Simulation is carried out through dedicated tools. With every simulation run, the simulation results are studied to identify errors in the design description. The errors are corrected and another simulation run carried out. Simulation and changes to design description together form a cyclic iterative process, repeated until an error-free design is evolved.

Design description is an activity independent of the target technology or manufacturer. It results in a description of the digital circuit. To translate it into a tangible circuit, one goes through the physical design process. The same constitutes a set of activities closely linked to the manufacturer and the target technology

1.4.1 Design Description

The design is carried out in stages. The process of transforming the idea into a detailed circuit description in terms of the elementary circuit components constitutes design description. The final circuit of such an IC can have up to a billion such components; it is arrived at in a step-by-step manner.

The first step in evolving the design description is to describe the circuit in terms of its behavior. The description looks like a program in a high level language like C. Once the behavioral level design description is ready, it is tested extensively with the help of a simulation tool; it checks and confirms that all the expected functions are carried out satisfactorily. If necessary, this behavioral level routine is edited, modified, and rerun – all done manually. Finally, one has a design for the expected system – described at the behavioral level. The behavioral design forms the input to the synthesis tools, for circuit synthesis. The behavioral constructs not supported by the synthesis tools are replaced by data flow and gate level constructs. To surmise, the designer has to develop synthesizable codes for his design.

The design at the behavioral level is to be elaborated in terms of known and acknowledged functional blocks. It forms the next detailed level of design description. Once again the design is to be tested through simulation and iteratively corrected for errors. The elaboration can be continued one or two steps further. It leads to a detailed design description in terms of logic gates and transistor switches.

1.4.2 Optimization

The circuit at the gate level – in terms of the gates and flip-flops – can be redundant in nature. The same can be minimized with the help of minimization tools. The step is not shown separately in the figure. The minimized logical design is converted to a circuit in terms of the switch level cells from standard libraries provided by the foundries. The cell based design generated by the tool is the last step in the logical design process; it forms the input to the first level of physical design [Micheli].

1.4.3 Simulation

The design descriptions are tested for their functionality at every level – behavioral, data flow, and gate. One has to check here whether all the functions are carried out as expected and rectify them. All such activities are carried out by the simulation tool. The tool also has an editor to carry out any corrections to the source code. Simulation involves testing the design for all its functions, functional sequences, timing constraints, and specifications. Normally testing and simulation at all the levels – behavioral to switch level – are carried out by a single tool; the same is identified as "scope of simulation tool" in Figure 1.5.

Synthesis

With the availability of design at the gate (switch) level, the logical design is complete. The corresponding circuit hardware realization is carried out by a synthesis tool. Two common approaches are as follows:

The circuit is realized through an FPGA [Oldfield]. The gate level design description is the starting point for the synthesis here. The FPGA vendors provide an interface to the synthesis tool. Through the interface the gate level design is realized as a final circuit. With many synthesis tools, one can directly use the design description at the data flow level itself to realize the final circuit through an FPGA. The FPGA route is attractive for limited

volume production or a fast development cycle.

The circuit is realized as an ASIC. A typical ASIC vendor will have his own library of basic components like elementary gates and flip-flops. Eventually the circuit is to be realized by selecting such components and interconnecting them conforming to the required design. This constitutes the physical design. Being an elaborate and costly process, a physical design may call for an intermediate functional verification through the FPGA route. The circuit realized through the FPGA is tested as a prototype. It provides another opportunity for testing the design closer to the final circuit.

1.4.5 Physical Design

A fully tested and error-free design at the switch level can be the starting point for a physical design [Baker & Boyce, Wolf]. It is to be realized as the final circuit using (typically) a million components in the foundry's library. The step-by-step activities in the process are described briefly as follows:

System partitioning: The design is partitioned into convenient compartmentsor functional blocks. Often it would have been done at an earlier stage itself and the software design prepared in terms of such blocks. Interconnection of

the blocks is part of the partition process.

Floor planning: The positions of the partitioned blocks are planned and theblocks are arranged accordingly. The procedure is analogous to the planning and arrangement of domestic furniture in a residence. Blocks with I/O pins are kept close to the periphery; those which interact frequently or through a large number of interconnections are kept close together, and so on. Partitioning and floor planning may have to be carried out and refined

iteratively to yield best results.

Placement: The selected components from the ASIC library are placed in position on the "Silicon floor." It is done with each of the blocks above.

Routing: The components placed as described above are to be interconnectedto the rest of the block: It is done with each of the blocks by suitably routing the interconnects. Once the routing is complete, the physical design cam is taken as complete. The final mask for the design can be made at this stage and the ASIC manufactured in the foundry.

1.4.6 Post Layout Simulation

Once the placement and routing are completed, the performance specifications like silicon area, power consumed, path delays, *etc*., can be computed. Equivalent circuit can be extracted at the component level and performance analysis carried out. This constitutes the final stage called "verification." One may have to go through the placement and routing activity once again to improve performance.

1.4.7 Critical Subsystems

The design may have critical subsystems. Their performance may be crucial to the overall performance; in other words, to improve the system performance substantially, one may have to design such subsystems afresh. The design here may imply redefinition of the basic feature size of the component, component design, placement of components, or routing done separately and specifically for the subsystem. A set of masks used in the foundry may have to be done afresh for the purpose.

1.5 ROLE OF HDL

An HDL provides the framework for the complete logical design of the ASIC. All the activities coming under the purview of an HDL are shown enclosed in bold dotted lines in Figure 1.4. Verilog and VHDL are the two most commonly used HDLs today. Both have constructs with which the design can be fully described at all the levels. There are additional constructs available to facilitate setting up of the test bench, spelling out test vectors for them and "observing" the outputs from the designed unit.

IEEE has brought out Standards for the HDLs, and the software tools conform to them. Verilog as an HDL was introduced by Cadence Design Systems; they placed it into the public domain in 1990. It was established as a formal IEEE Standard in 1995. The revised version has been brought out in 2001. However, most of the simulation tools available today conform only to the 1995 version of the standard.

Verilog HDL used by a substantial number of the VLSI designers today is the topic of discussion of the book.

1.6 VERILOG AS AN HDL

Verilog has a variety of constructs as part of it. All are aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC. The language has a dual function – one fulfilling the need for a design description and the other fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, slack, setup, and hold times [Smith DJ, Wai-Kai].

Verilog as an HDL has been introduced here and its overall structure explained. A widely used development tool for simulation and synthesis has been introduced; the brief procedural explanation provided suffices to try out the Examples and Exercises in the text.

1.7 LEVELS OF DESIGN DESCRIPTION

The components of the target design can be described at different levels with the help of the constructs in Verilog.

1.7.1 Circuit Level

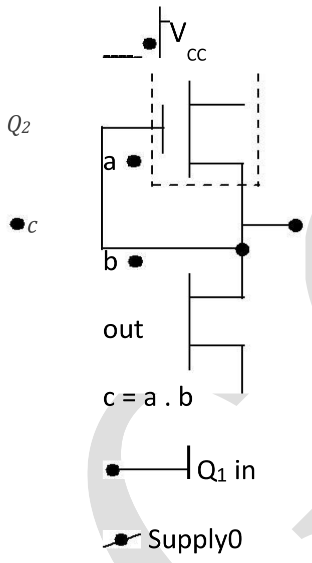
At the circuit level, a switch is the basic element with which digital circuits are built. Switches can be combined to form inverters and other gates at the next higher level of abstraction. Verilog has the basic MOS switches built into its constructs, which can be used to build basic circuits like inverters, basic logic gates, simple 1-bit dynamic and static memories. They can be used to build up larger designs to simulate at the circuit level, to design performance critical circuits. Figure 2.1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

1.7.2 Gate Level

At the next higher level of abstraction, design is carried out in terms of basic gates. All the basic gates are available as ready modules called "Primitives." Each such primitive is defined in terms of its inputs and outputs. Primitives can be incorporated into design descriptions directly. Just as full physical hardware can be built using gates, the primitives can be used repeatedly and judiciously to build larger systems. Figure 2.2 shows an AND gate suitable for description using the gate primitive of Verilog. The gate level modeling or structural modeling as it is sometimes called is akin to building a digital circuit on a bread board, or on a PCB. One should know the structure of the design to build the model here. One can also build hierarchical circuits at this level. However, beyond 20 to 30 of such gate primitives in a circuit, the design description becomes unwieldy; testing and debugging become laborious.

1.7.3 Data Flow

Data flow is the next higher level of abstraction. All possible operations on signals and variables are represented here in terms of assignments. All logic and algebraic operations are accommodated. The assignments define the continuous functioning of the concerned block. At the data flow level, signals are assigned through the data manipulating equations. All such assignments are concurrent in nature. The design descriptions are more compact than those at the gate level. Figure 2.3 shows an A-O-I relationship suitable for description with the Verilog constructs at the data flow level.



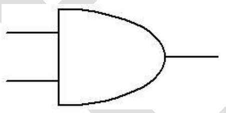


Figure A simple Inverter circuit at the gate level. **Figure** A simple AND gate representedat the switch level.

1.7.4 Behavioral Level

Behavioral level constitutes the highest level of design description; it is essentially at the system level itself [Bhaskar]. With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a "C" program. The statements involved are "dense" in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient. Figure 2.4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of

Verilog.

1.7.5 The Overall Design Structure in Verilog

The possibilities of design description statements and assignments at different levels necessitate their accommodation in a mixed mode. In fact the design statements coexisting in a seamless manner within a design module is a significant characteristic of Verilog. Thus Verilog facilitates the mixing of the above-mentioned levels of design. A design built at data flow level can be instantiated to form a structural mode design. Data flow assignments can be incorporated in designs which are basically at behavioral level.

1.8 CONCURRENCY

In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change. Simulation of such a circuit in an HDL calls for concurrency of operation. A number of activities – may be spread over different modules – are to be run concurrently here. Verilog simulators are built to simulate concurrency. (This is in contrast to programs in the normal languages like C where execution is sequential.) Concurrency is achieved by proceeding with simulation in equal time steps. The time step is kept small enough to be negligible compared with the propagation delay values. All the activities scheduled at one time step are completed and then the simulator

$e \ \overline{a.b \ \ c.d}$	If (a, b, c or d changes) Compute e as $e \ \ \overline{a.b \ \ c.d}$

Figure An A-O-I gate represented as a code at data flow type of relationship. **Figure** An A-O-I gate in pseudo behavioral level.

advances to the next time step and so on. The time step values refer to simulation time and not real time. One can redefine timescales to suit technology as and when necessary and carry out test runs.

In some cases the circuit itself may demand sequential operation as with data transfer and memory-based operations. Only in such cases sequential operation is ensured by the appropriate usage of sequential constructs from Verilog HDL.

1.9 SIMULATION AND SYNTHESIS

The design that is specified and entered as described earlier is simulated for functionality and fully debugged. Translation of the debugged design into the corresponding hardware circuit (using an FPGA or an ASIC) is called "synthesis." The tools available for synthesis relate more easily with the gate level and data flow level modules [Smith MJ]. The circuits realized from them are essentially direct translations of functions into circuit elements. In contrast many of the behavioral level constructs are not directly synthesizable; even if synthesized they are likely to yield relatively redundant or wrong hardware. The way out is to take the behavioral level modules and redo each of them at lower levels. The process is carried out successively with each of the behavioral level modules until practically the full design is available as a pack of modules at gate and data flow levels (more commonly called the "RTL level").

1.10 FUNCTIONAL VERIFICATION

Testing is an essential ingredient of the VLSI design process as with any hardware circuit. It has two dimensions to it – functional tests and timing tests. Both can be carried out with Verilog. Often testing or functional verification is carried out by setting up a "test bench" for the design. The test bench will have the design instantiated in it; it will generate necessary test signals and apply them to the instantiated design. The outputs from the design are brought back to the test bench for further analysis. The input signal combinations, waveforms and sequences required for testing are all to be decided in advance and the test bench configured based on the same.

The test benches are mostly done at the behavioral level. The constructs there are flexible enough to allow all types of test signals to be generated.

In the process of testing a module, one may have to access variables buried inside other modules instantiated within the master module. Such variables can be accessed through suitable hierarchical addressing.

Test Inputs for Test Benches

Any digital system has to carry out a number of activities in a defined manner. Once a proper design is done, it has to be tested for all its functional aspects. The system has to carry out all the expected activities and not falter. Further, it should not malfunction under any set of input conditions. Functional testing is carried out to check for such requirements. Test inputs can be purely combinational, periodic, numeric sequences, random inputs, conditional inputs, or combinations of these. With such requirements, definition and design of test benches is often as challenging as the design itself.

As the circuit design proceeds, one develops smaller blocks and groups them together to form bigger circuit units. The process is repeated until the whole system is fully built up. Every stage calls for tests to see whether the subsystem at that layer behaves in the manner expected. Such testing calls for two types of observations:

Study of signals within a small unit when test inputs are given to the whole unit.

Isolation of a small element and doing local test to facilitate debugging.

Verilog has constructs to accommodate both types of observation through a hierarchical description of variables within.

1.10.1 Constructs for Modeling Timing Delays

Any basic gate has propagation delays and transmission delays associated with it. As the elements in the circuit increase in number, the type and variety of such delays increase rapidly; often one reaches a stage where the expected function is not realized thanks to an unduly large time delay. Thus there is a need to test every digital design for its performance with respect to time. Verilog has constructs for modeling the following delays:

Gate delay Net delay Path delay

Pin-to-pin delay

In addition, a design can be tested for setup time, hold time, clock-width time specifications, *etc*. Such constructs or delay models are akin to the finite delay time, rise time, fall time, path or propagation delays, *etc*., associated with real digital circuits or systems. The use of such constructs in the design helps simulate realistic conditions in a digital circuit. Further, one can change the values of delays in different ways. If a buffer capacity is increased, its associated delays can be reduced. If a design is to migrate to a better technology, the delay values can be rescaled. With such testing, one can estimate the minimum frequency of operation, the maximum speed of response, or typical response times.

1.10.2 SYSTEM TASKS

A number of system tasks are available in Verilog. Though used in a design description, they are not part of it. Some tasks facilitate control and flow of the testing process. The values of signals in a module can be displayed in the course of simulation. The tasks available for the purpose display them in desired formats. Reading data from specified files into a module and writing back into files are also possible through other tasks. Timescale can be changed prior to simulation with the help of specific tasks for the purpose.

A set of system functions add to the flexibility of test benches: They are of three categories:

Functions that keep track of the progress of simulation time

Functions to convert data or values of variables from one format to another Functions to generate random numbers with specific distributions.

There are other numerous system tasks and functions associated with file operations, PLAs, *etc*.

1.10.3 PROGRAMMING LANGUAGE INTERFACE (PLI)

PLI provides an active interface to a compiled Verilog module. The interface adds a new dimension to working with Verilog routines from a C platform. The key functions of the interface are as follows:

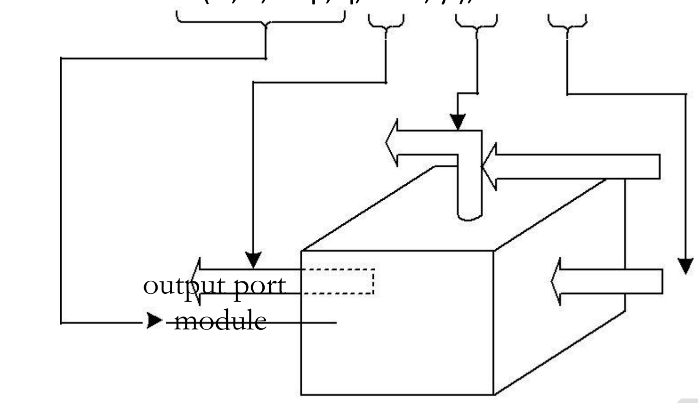
One can read data from a file and pass it to a Verilog module as input. Such data can be test vectors or other input data to the module. Similarly, variables

in Verilog modules can be accessed and their values written to output devices. Delay values, logic values, *etc*., within a module can be accessed and altered. Blocks written in C language can be linked to Verilog modules.

1.11 MODULE

Any Verilog program begins with a keyword – called a "**`module`**." A **`module`** is the name given to any system considering it as a black box with input and output terminals as shown in Figure 2.5. The terminals of the module are referred to as 'ports'. The ports attached to a module can be of three types:

module adder(a, b, . . .p, q, . . . x, y);



input por

Figure Representation of a module as black box with its ports.

input ports through which one gets entry into the module; they signify the input signal terminals of the module.

output ports through which one exits the module; these signify the output signal terminals of the module.

inout ports: These represent ports through which one gets entry into themodule or exits the module; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

Whether a module has any of the above ports and how many of each type are present depend solely on the functional nature of the module. Thus one module may not have any port at all, another may have only input ports, while a third may have only output ports, and so on.

All the constructs in Verilog are centered on the module. They define ways of building up, accessing, and using modules. The structure of modules and the mode of invoking them in a design are discussed here.

A module comprises a number of "lexical tokens" arranged according to some predefined order. The possible tokens are of seven categories:

White spaces Comments Operators

Numbers Strings

Identifiers Keywords

The rules constraining the tokens and their sequencing will be dealt with as we progress. For the present let us consider modules. In Verilog any program which forms a design description is a "module." Any program written to test a design description is also a "module." The latter are often called as “stimulus modules” or “test benches.” A module used to do simulation has the form shown in Figure 2.6. Verilog takes the active statements appearing between the “**`module`**” statement and the “**`endmodule`**” statement and interprets all of them together as forming the body of the module. Whenever a module is invoked for testing or for incorporation into a bigger design module, the name of the module (“test” here) is used to identify it for the purpose.

A digression into design using SSI ICs is in order here. Consider the IC 7430, an eight input NAND gate. In any design using it, the IC can be looked up on as a black box with eight input leads and one output lead (Figure 2.7a). Three aspects characterize the IC – its function, its input leads, and its output lead. Other ICs may have more output leads. A NAND gate module is defined in an analogous manner in terms of its function, input leads and the output lead. The module used to describe the circuit here also follows the earlier format; that is, the "**`module`**" statement signifies the beginning of the module, the "**`endmodule`**" statement signifies the end of the module. However, the initial statement "**`module`**" has to be more elaborate with the input and the output ports forming part of it (see Figure).

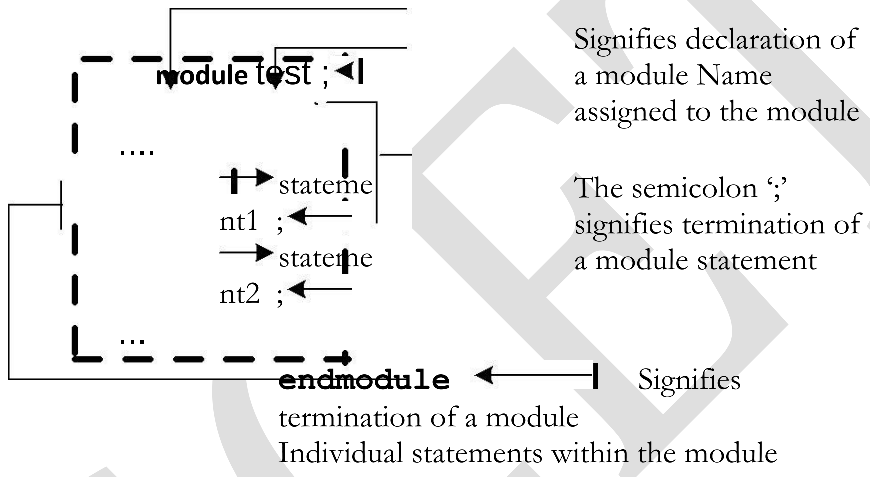

module test ;
....
stateme nt1 ;
stateme nt2 ;
...
endmodule
Signifies declaration of a module Name assigned to the module
The semicolon ';' signifies termination of a module statement
Signifies termination of a module
Individual statements within the module

Figure Structure of a typical simulation module.

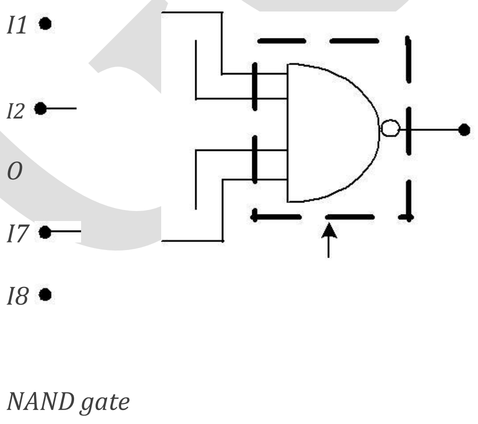
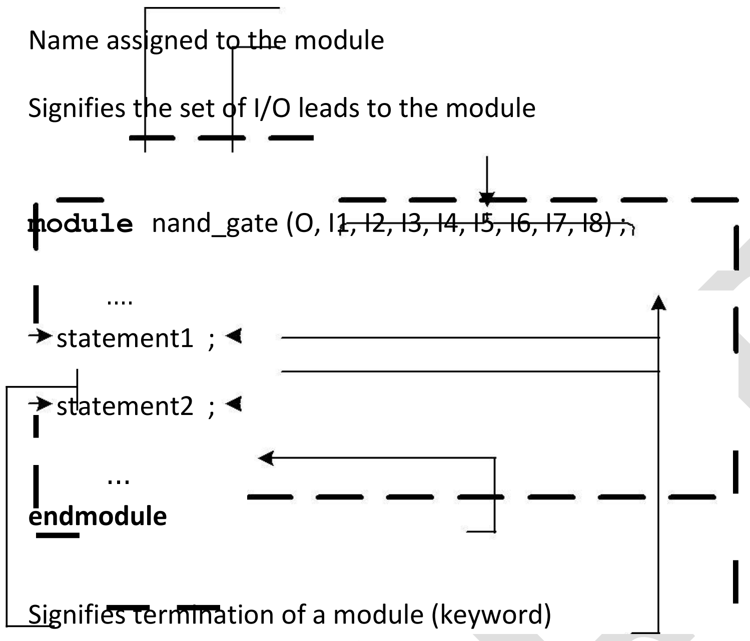


Figure (a) Eight input NAND gate (IC 7430). Gate proper with terminals.

Signifies declaration of a module(keyword)

Name assigned to the module

Signifies the set of I/O leads to the module



```
module  nand_gate (O, I1, I2, I3, I4, I5, I6, I7, I8) ;
      ....
   statement1  ;
   statement2  ;
      ...
endmodule
```

Signifies termination of a module (keyword)

Individual statements within the module

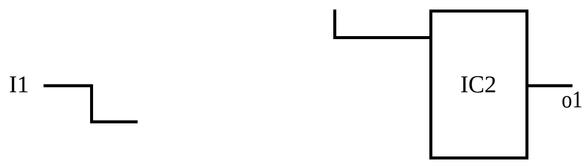
The semi-colon ‘;’ signifies termination of a module statement

Figure (b) Eight input NAND gate (IC 7430). Structure of the gate module.

The same type of IC – 7430 – may be repeatedly used in a circuit. Each time it is used, a different name is assigned to it in the design sheet. Part of such a typical design sheet will look as in Figure 2.8. The associated table (Table 2.1) allows us to identify each type of IC to be used and put in its proper place. An automated design description can use a module defined above, repeatedly in a number of places as in the circuit of Figure . Each such use is an “instantiation.” A typical instantiation of the module defined above has the form shown in Figure 2.9. The following observations are in order here:

Table Partial list of IC numbers and their types for a typical design

IC No	IC1	IC2	IC3	…	IC9	…
IC type	7430	7430		…	7405	…



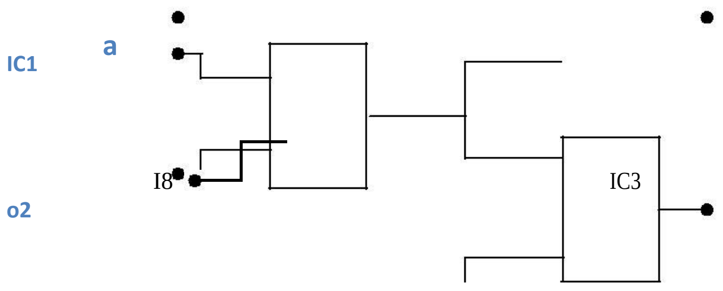
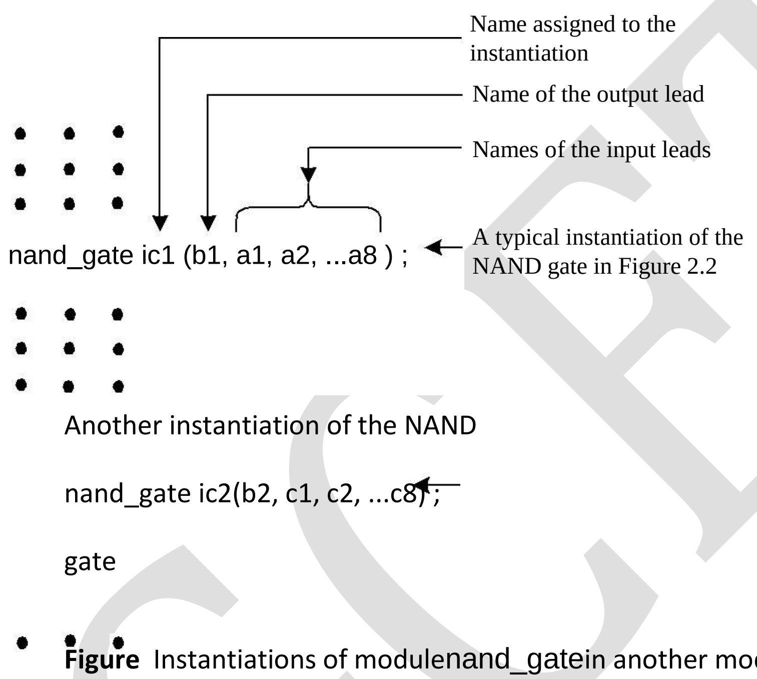


Figure Part of the circuit diagram of a typical digital circuit.


Name assigned to the instantiation
Name of the output lead
Names of the input leads
A typical instantiation of the NAND gate in Figure 2.2

nand_gate ic1 (b1, a1, a2, ...a8) ;

Another instantiation of the NAND

nand_gate ic2(b2, c1, c2, ...c8);

gate

Figure Instantiations of modulenand_gatein another module.

The designer has defined a specific function within a module; the module is assigned the name "nand_gate."

The nand_gate can be invoked (instantiated) by him in a design as many times as desired.

Each instantiation has to be assigned a separate identifier name by him (called "IC1", "IC2", *etc*.). As part of the instantiation declaration, the input and output terminals are to be defined. The convention followed is to stick to the same order as in the module declaration. It is further illustrated in Figure 2.9.

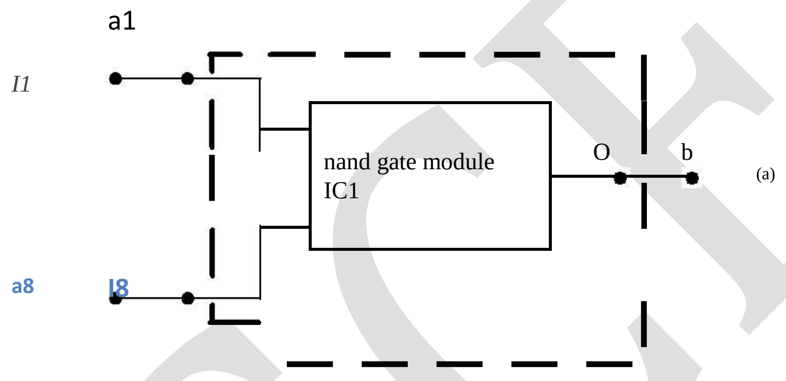
Some modules may have a large number of ports. Sticking to the order of the ports in an instantiation is likely to cause (human) errors. An alternative (and sometimes more convenient) form of instantiation is also possible – shown in Figure 2.10. The terminal identifications are explicit (though elaborate) here. Further one need not stick to the order of the ports as they appear in the module definition. With such a form of port assignments, the possibility of errors is considerably reduced.

The following aspects of the modules and their instantiation are noteworthy:

Each module can be defined only once.
Module definitions are to be done independently. One module cannot be defined inside another – they cannot be nested.

Any module can be instantiated inside another any number of times. Each instantiation has to be done with a separate name assigned to it.

The various constructs and features available in Verilog are discussed in the following chapters. However, certain conventions and constructs essential for the progress of the book at this stage are discussed in Chapter 3.



nand_gate ic1(O(b), I8(a8), ... I1(a1)); (b)

Figure A typical circuit block and (b) its instantiation.

1.12 SIMULATION AND SYNTHESIS

A variety of Software tools related to VLSI design is available**.** We discuss here two of them directly relevant to us – Modelsim and Leonardo Spectrum of Mentor Graphics. Modelsim has been used to simulate the designs. Simulation results presented for the variety of examples discussed in the book have been obtained using it. Leonardo Spectrum has been used to obtain the synthesized circuits presented. We would like to draw the attention of the readers to the following in this context:

Only the essential aspects of the tools are presented – those essential for the progress of the book.

For more details of the tools and the variety of facilities they offer, one

can refer to the respective user manuals and the Help menus.

Tools from other sources are similar in essentials. Any of them can be used.

1.12.1 Use of Modelsim SE 5.5

The procedure to invoke the tool and use it is briefly described here. The tool can be used to prepare a source file, edit and compile it, and simulate the compiled version.

Editing and Compilation

Open the Modelsim Window. We get the following menus listed at the top:

File	Edit	Design	View	Project	Run	Compare	Macro

Click on "View." We get the following menus:

All

Hide Workspace

Sources

Structure

Variables

Signals

List

Process

Wave

Data flow

Data sets

New

Other

Click on “Source.” The “Source” window opens with the following set of menus listed at the top:

File Edit Object Options Window

Click on “File” option. We get the following options:

New

Open

Use source

Source directory

Properties

Save

Save as

Compile

Close

Click on “New.” We get the following options:

VHDL

Verilog

Other

Click on "Verilog." A "Source_edit-new.v" opens.

The Verilog design can be keyed in. It forms the source file. The source file considered in various examples in the book can be created in this manner

(*e.g.*, Example 4.2 and Figure 4.4).

Click on "File" option. We get a pull down menu. Click on "Save as."

Select a Directory of your choice. Give a suitable filename with extension ".v" (Say "demo.v"). Click on "Save" and save the file. The source (design) file has been created and saved. Now it is ready for compilation.

Click on "Compile." "Compile HDL Source Files" window opens. File name "demo" is displayed. Library "Work" is displayed. The selected file (demo.v) will be compiled and loaded into Work. The lines of display in the

main window confirm this.

If the source file has any syntax or logical errors, compilation will not take place. The errors will be indicated in the main window. The source file can be opened (by clicking on the main menu) and edited. Once again compilation can be attempted. The procedure has to be repeated iteratively until all the errors in the source file have been removed and compilation is successfully completed.

Simulation

In the main window click on "Design" pulldown menu.

In the options displayed, click on "Load Design." The following options are displayed at the top:–

Design	VHDL	Verilog	Libraries	SDF

Select "Design" and click on it. A small window appears on the screen. "Library: Work" is displayed, implying that the working library is open. The module name "demo" is displayed under it. In the normal course the names of all the compiled files will be listed alphabetically one below the other. The

specific file to be simulated is to be selected by clicking on the same.

The "Load" button below gets highlighted. Click on it. The design gets loaded and is ready for simulation run.

Click the "Run" menu in the Modelsim main window. Select 100 ns runtime. The design runs for 100 ns (by default) and the output list appears in the main window. The listing can be selected, copied, and pasted to another file. The simulation results for the various examples in the book have been obtained in this manner. If necessary, the time duration of simulation can be altered in

the main window.

Observing Waveforms

Simulation results can alternately be viewed as waveforms with the following procedure:

In the main Modelsim window click on "Signals." The signals window opens with the following options displayed at the top:

File	Edit	View	Window

Click on the "View" pulldown menu. We get the options as shown below:

Wave List Log Filter

Amongst the options available, click on "Wave." We get the following options:

Selected Signals Signals in Region Signals in Design

Select "Signals in Design." The "Waveform Window" opens and shows the signals in the design. The Window has a "Run" option.

Click on "Run" to run the design and get the waveforms displayed.

The waveforms shown as simulated outputs for different examples in the book have been obtained in this manner.

One can practice simulation of a few examples given in the book. Subsequently options available at the different stages can be tried, and the tool with its full versatility can be mastered.

1.12.2 Synthesis

Conversion of the code into hardware logic and fitting it into an FPGA or ASIC to realize the circuit is termed "Synthesis." We have used the Mentor Graphics Synthesis tool called "Leonardo Spectrum" for the purpose. The synthesis procedure is briefly described here:

Double click on "Leonardo Spectrum 2000.1b."

The Main Window named "Examplar Logic – Leonardo Spectrum Level *3*"opens with a pulldown menu as follows:

File | Edit | View | Tools | Window | Help

Click on "File". A pulldown menu opens with options such as the following:

New

Open

Save

…

…

Select "New." A window named "untitled" opens. We can type in a new program and save it as a file with a name assigned to it (Say "name.v") in a directory of our choice. The procedure is similar to that followed above to create and save a new file with extension ".v" (signifying that it is a Verilog file). The file is now ready for synthesis. However, it is always preferable to simulate a file and be fully satisfied with at the simulation stage itself before

synthesizing it.

Click on the "Tools" menu on the main window. A set of options appear on the screen.

Select "Quick Set up." A window of the type shown in Figure 2.11 appears. All the settings necessary to complete the synthesis can be carried out with it. Click on "Open files." Select the Verilog source file to be synthesized. It will

be visible under "Input" in the figure.

Under "Technology" select "FPGA." Select a device of (say) Xilinx – for example, XC4000XL. The selected Xilinx device name is displayed under

'Device'.

Select a "Clock Frequency" – say 10 MHz.

Click on the "Run Flow" button. The synthesis program runs and completes the synthesis. Summarized results will be displayed on the screen.

If the coding is correct and synthesizable, the display "Ready" appears highlighted at the bottom left-hand corner. If not, error details will be displayed. The program may be rectified and synthesis attempted again. Icons for "RTL Schematic", "Gate Level Schematic" and "Critical Path

Schematic" at the top become active.

We can click on each of them in succession. The circuit schematic can be viewed at the RTL level or the gate level. The critical path can be viewed – it represents the path that takes the maximum time of operation on a pin-to-pin basis. It sets the upper limit to the speed of operation of the circuit.

The synthesized circuits shown for the different examples in the book have been obtained in this manner. The device selected to synthesize the design, is called the “Target Device.” One can select any other suitable target device of Xilinx or other FPGA vendors like Actel, Altera, Cypress, Lattice, Lucent, Quicklogic, *etc*.

The program generates a summary of the synthesis activity and displays it as a “Sum File.” It gives a report on the utilization of the “Target Device” by the

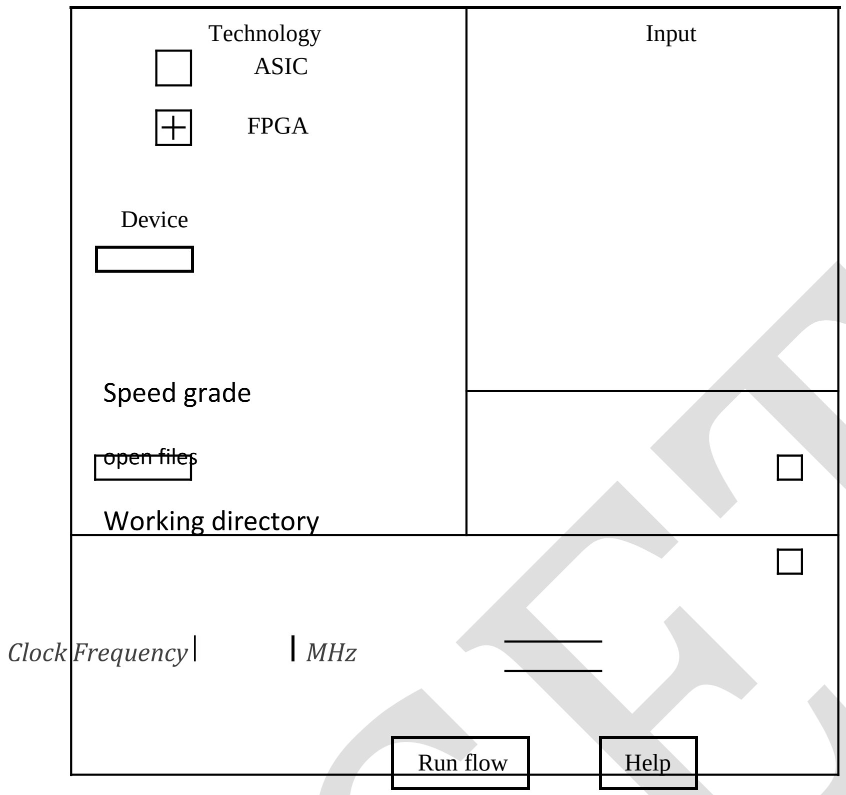


Figure 2.11 The Window in Leonardo Spectrum to do the settings for synthesis.

design that was synthesized. It also generates and displays some timing information like "Critical Path Timing."

1.13 TEST BENCHES

Any digital circuit that has been designed and wired goes through a testing process before being declared as ready for use. Testing involves studying circuit behavior under simulated conditions for the following:

Check and ensure that all functions are carried out as desired. It is the test for the static behavior of the circuit. A set of logic input values are applied at

selected points and the logic values at another set of points observed.

Check and ensure that all the functional sequences are carried out as desired. It is one of the tests for the dynamic behavior of the circuit. It may call for the generation of specific input sequences with respect to time, applying them to the circuit and observing selected outputs.

Check for the timing behavior: One tests for the propagation and other types of delays here. A variety of tests may have to be carried out. It may involve observation of variations in the signals at selected points, measuring the time delay between specified events, measuring pulse widths, and so on.

Verilog has the provision for all the above. One sets up a “test bench” in software and caries out a simulated test. The facilities required to set up test benches are discussed in detail in Chapters 7 and 8. However, the need to test the designs in Chapters 4 to 6 warrants a brief introduction to them here; only the essentials are discussed. Further, the “test benches” up to Chapter 7 are kept simple and easily understandable.

Simulated testing is a time-based activity. It is usually carried out in simulated time. With any simulation tool the simulation progresses through equal simulation time steps. The time step can be 1 fs, 1 ps, 1 ns and so on. In the text the default value is taken as 1 ns. In some cases it is mentioned explicitly; in other situations it is implicit, *that is,* whenever ‘time step’ is mentioned, it implies 1ns of simulation time. If required, the simulation time step can be altered (see Chapter 11).

Consider the group of statements below reproduced from the test bench of Figure :

```
Initial

Begin
    a1 = 0;

  a2 = 0; #3 a1 = 1; #1 a1 = 0; #2 a2 = 1; #4 a1 = 1; #3 a2 = 0; #1 a2 = 1; end

        and g1(b, a1, a2);
        initial $monitor ( $time, “a1 = %b, a2 = %b, b = %b”’ a1,
        a2, b);#100 $finish;
```

The **keyword** initial is followed by a sequence of statements between the keywords **begin** and **end**. Usually the **initial** banner signifies a setting done on a once or a “once for all” basis. The “# 3” implies a time delay or wait time of 3 time steps in simulation. Thus the sequence implies the following:

At 0 simulation time the logic variables a1 and a2 are assigned the logic level 0.

With a delay of 3 ns a1 is reassigned the logic value of 1.

With a further delay of 1 ns – that is, at the 4th ns - a1 is reverted to the logic level 0.

Similarly at the 6th, 10th, 13th and 14th ns values of simulation time, further changes are made to a1 and a2.

Note that every time value specified here is an increment in simulation time.

The values of a1 and a2 are not changed beyond the 14th ns. The statement

initial # 100 $finish;

implies that the simulation is to be continued up to the 100th ns of simulation time and then stopped.

The above constitutes the generation of the test sequence for testing. Such test signals are applied to the designed circuit through instantiation; the statement

and g1(b, a1, a2);

implies as much. The statement

initial $monitor (**$time**, “a1 = %b, a2 = %b, b = %b”’ a1, a2, b);

monitors a1, a2, and a3 for changes; whenever any of them changes, all of them are sampled and the sampled values displayed.

Summarizing testing constitutes three activities:

*Generation of the test signals – under the “**initial**” banner*

Application of the test signal to the circuit under test – through instantiation Observing selected signal values – through the **$monitor** statement

Many of the test benches for the subsequent examples are also structured in a similar fashion. Changes are kept to the minimum to ensure focus on the example concerned. As and when such changes are made, the same is explained.

2
LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

INTRODUCTION

The constructs and conventions make up a software language. A clear understanding and familiarity of these is essential for the mastery of the language. Verilog has its own constructs and conventions [IEEE, Sutherland]. In many respects they resemble those of C language [Gottfried]. We discuss the constructs and conventions essential to the progress of the book. More of these follow in the ensuing chapters.

Any source file in Verilog (as with any file in any other programming language) is made up of a number of ASCII characters. The characters are grouped into sets — referred to as “lexical tokens.” A lexical token in Verilog can be a single character or a group of characters. Verilog has 7 types of lexical tokens

- operators, keywords, identifiers, white spaces, comments, numbers, and strings. Operators are introduced in Chapter 6. All the other tokens are discussed here. Some other aspects of Verilog essential to the progress of the book are also discussed subsequently.

2.1.1 Case Sensitivity

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,… *etc.*, are all treated as different entities / quantities in Verilog.

2.2 KEYWORDS

The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. As such, a programmer cannot use a keyword for any purpose other than that it is intended for. All keywords in Verilog are in small letters and require to be used as such (since Verilog is a case-sensitive language). All keywords appear in the text in New Courier Bold-type letters.

Examples

`module` signifies the beginning of a module definition. **`endmodule`** signifies the end of a module definition. **`begin`** signifies the beginning of a block of statements. **`end`** signifies the end of a block of statements.
`if` signifies a conditional activity to be checked **`while`** signifies a conditional activity to be carried out.
A list of keywords in Verilog with the significance of each is given in Appendix A.

2.3 IDENTIFIERS

Any program requires blocks of statements, signals, *etc*., to be identified with an attached nametag. Such nametags are identifiers. It is good practice for us to use identifiers, closely related to the significance of variable, signal, block, *etc*., concerned. This eases understanding and debugging of any program.
e.g.,clock,enable,gate_1, . . .

There are some restrictions in assigning identifier names. All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (_), or the dollar ($) sign – for example

name, _name. Name, name1, name_$, . . . all these are allowed asidentifiers
name aa not allowed as an identifier because of the blank ("name" and "aa"are interpreted as two different identifiers)
$name not allowed as an identifier because of the presence of "$" as the firstcharacter.
1_name not allowed as an identifier, since the numeral "1" is the firstcharacter
@name not allowed as an identifier because of the presence of the character"@".
A+b not allowed as an identifier because of the presence of the character "+".

An alternative format makes it is possible to use any of the printable ASCII characters in an identifier. Such identifiers are called "escaped identifiers"; they have to start with the backslash (\) character. The character set between the first backslash character and the first white space encountered is treated as an identifier. The backslash itself is not treated as a character of the identifier concerned.

Examples

\b=c \control-signal\&logic
\abc // Here the combination "abc" forms the identifier.
It is preferable to use the former type of identifiers and avoid the escaped identifiers; they may be reserved for use in files which are available as inputs to the design from other CAD tools.

2.4 WHITE SPACE CHARACTERS

Blanks (\b), tabs (\t), newlines (\n), and formfeed form the white space characters in Verilog. In any design description the white space characters are included to improve readability. Functionally, they separate legal tokens. They are introduced between keywords, keyword and an identifier, between two identifiers, between identifiers and operator symbols, and so on. White space characters have significance only when they appear inside strings.

2.5 COMMENTS

It is a healthy practice to comment a design description liberally – as with any other program. Comments are incorporated in two ways. A single line comment begins with "//" and ends with a new line – for example

module d_ff(Q,dp,clk); //This is the design description of a D flip-flop.
//Here Q is the output.
// dp is the input and clk is the clock.

One can incorporate multiline comments also without resorting to "//" at every line. For such multiline comments "/*" signifies the beginning of a comment and "*/" its end. All lines appearing between these two symbol combinations are together treated as a single block comment – for example

module d_ff(Q,dp,clk);
/* This module forms the design description of a d_flip_flop wherein Q is the output of the flip-flop ,
dp is the data input and clk the clock input*/

Multiline comments cannot be nested. For example, the following comment is not valid.

/*The following forms the design description of a D flip-flop /*which can be modified to form other types of flip-flops*/ with clock and data inputs.*/

A valid alternative can be as follows: -

/*The following forms the design description of a D flip-flop (which can be modified to form other types of flip-flops) with clock and data inputs.*/

2.6 NUMBERS

Frequently numbers need to be specified in a design description. Logic status of signal lines, buses, delay values, and numbers to be loaded in registers are examples. The numbers can be of integer type or real type.

2.6.1 Integer Numbers

Integers can be represented in two ways. In the first case it is a decimal number – signed or unsigned; an unsigned number is automatically taken as a positive number. Some examples of valid number representations of this category are given below:

2
25
253

- 253

The following are invalid since nondecimal representations are not permissible.

2a

B8

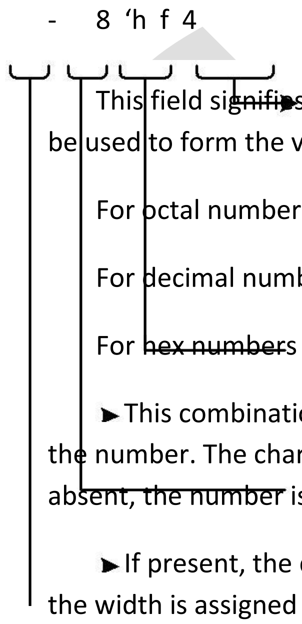
- 2a
- B8

Normally the number is taken as 32 bits wide. Thus all the following numbers are assigned 32 bits of width:
2
25253

- 2
- 25
- 253

If a design description has a number specified in the form given here, the circuit synthesizer program will assign 32 bits of width to it and to all the related circuits. Hence all such number specifications – despite their simplicity – may be avoided in design descriptions. Number representation in this form may preferably be restricted to test benches.

The alternate form of number representation is more specific – though elaborate. The number can be specified in binary, octal, decimal, or hexadecimal form. The representation has three tokens with an optional sign preceding it. Figure 3.1 shows typical number representations with the significance of each field explained separately.



This field signifies the value of the number. For binary numbers the characters 0, 1, x, z can be used to form the value.

For octal numbers the numerals 0 to 7, x, z can be used to form the value.

For decimal numbers all the numerals, x, z can be used to form the value.

For hex numbers all the numerals, a, b, c, d, e, f, x, z can be used to form the numbers.

➤This combination - the single quote character followed by b, o, d or h - specifes the base of the number. The character signifies binary, octal, decimal or hexadecimal base. If this field is absent, the number is taken as a dcimal one.

➤If present, the decimal number in this field signifies the bit width of the number. If absent the width is assigned a default value by the compiler.

$\longrightarrow$ This field(optional) is for the sign bit. It is allowed only with the decimal numbers. If absent, the number is taken as positive. For a number with a negative sign the number is represented in 2’s complement form

Figure Representation of a number in Verilog: One can use capital letters instead ofsmall letters in the last two fields.

Observations:

The characters used to specify the base number, the sign or the magnitude can be in either case (Thus A, B, C, D, E, or F can be used in place of a, b, c, d,e, or f, respectively, to specify the concerned hex digit.**X**or**Z**can be used in

place of **x** or **z** value, respectively).

The single quote character in the base field has to be immediately followed by the character representing the base. Intervening white spaces are not allowed.

However, such white spaces can precede the magnitude field. Negative numbers are represented in 2’s complement form.

The question mark character – “?” – can be used in place of **z**. The underscore character can be used anywhere after the first character. It adds to

the readability. It is normally ignored.

If the number size is smaller than the size specified, the size is made up by padding 0’s to the left. However, if the leftmost bit is a **x** or **z**, the same is

padded to the left.

Left truncation and right extension can often be confusing. It is preferable to specify the numbers fully.

Table 3.1 shows the format of specifications of the integer type numbers along with illustrative examples.

2.6.2 Real Numbers

Real numbers can be specified in decimal or scientific notation. The decimal notation has the form

- a.b

where a, b, the negative sign, and the decimal point have the usual significance. The fields a and b must be present in the number. A number can be specified in scientific notation as

4.3e2
where 4.3 is the mantissa and 2 the exponent. The decimal equivalent of this number is 430. Other examples of numbers represented in scientific notation are –4.3e2, –4.3e–2, and 4.3e–2. The representations are common.

2.7 STRINGS

A string is a sequence of characters enclosed within double quotes. A string must be contained on a single line; that is, it cannot be carried over to two lines with a

Table Different ways of number representations in Verilog

Representation	Remarks
33 'd33	Both of these represent decimal numbers of unspecified size – normally interpreted by Verilog as 32 bitwide, *i.e.*, 0000 0000 0000 0000 0000 0000 0010 0001
9'd439 9'D439 9'D4_39	All these represent 3 digit decimal numbers. D&d both specify decimal numbers. "_" (underscore) is ignored
9'b1_1011__1x01 9'b11011x01 9'B11011x01	All these represent binary numbers of value 11011x01. B& b specify binary numbers. "_" is ignored. **x** signifies the concerned bit to be of unknown value.
9'o123 9'O123 9'o1x3 9'o12z	All these represent 9-bit octal numbers. The binary equivalents are 001 010 011, 001 010 011, 001 **xxx** 011, 001 010 **zzz** respectively. **z** signifies the concerned bits to be in the high impedance state.
'o213	An octal number of unspecified size having octal value 213.
8'ha5 8'HA5 8'hA5 8'ha_5	All these are 8 bit-wide-hex numbers of hex value a5h. The equivalent binary value is 1010 0101.
11'hb0	A 11 bit number with a hex assignment. Its value is 000 1011 0000. The number of bits specified is more than that indicated in the value field. Enough zeros are padded to the left as shown.
9'hza	A hex number of 9 bits. Its value is taken as **zzzzz** 1010.
5'hza	A 5-bit hex number. Its value is taken as **z** 1010.
5'h?a	A 5-bit hex number. Its value is taken as **z** 1010. '?' is another representation for '**z**'.
-5'h1a -3'b101	Negative numbers. Negative numbers are represented in 2's complement form.
-4'd7	A 4 bit negative number. Its value in 2's complement form is 7. Thus the number is actually – (16 – 7) = –9.

carriage return. Special characters are specified by preceding them with the "\" character. Verilog treats a string as a sequence of ASCII characters – for example,
"This is a string"
"This string is one \t with a gap in between"
"This is called a \"string\"".
When a string of ASCII characters as above is an operand in an expression, it is treated as a binary number. This binary number is formed by replacing each ASCII character by 8 bits – a 0 bit followed by the 7-bit ASCII equivalent – and treating the resulting binary sequence as a single binary number. For example, the statement (with P defined as a 32-bit vector beforehand)
P = "numb"
assigns the binary value
0110 1110 0111 0101 0110 1101 0110 0010
to P (0110 1110, 0111 0101, 0110 1101 and 0110 0010 are the 8-bit equivalents of

the letters n, u, m, and b, respectively).

2.8 LOGIC VALUES

Signal lines, logic values appearing on signal lines, *etc*., can normally take two logic levels:
1 signifies the 1 or high or true level 0 signifies the 0 or low or false level.

Two additional levels are also possible – designated as **x** and **z**. Here **x** represents an unknown or an uninitialized value. This corresponds to the don't-care case in logic circuits. **z** represents / signifies a high impedance state. This is possible when a signal line is tri-stated or left floating. The following are noteworthy here:

When a variable in an expression is in the **z** state, the effect is the same as it having **z** value. But when an input to a gate is in the **z** state (see Chapter 4), it

is equivalent to having the **x** value.

The MOS switches discussed in Chapter 10 form an exception to the above. If the input to a MOS switch is in the **z** state, its output too remains at the **z**

state.

With a few exceptions all data types in Verilog can take on all the 4 logic values or levels. The **event** (see Section 8.11) is an exception to this. It cannot store any value. The **trireg** cannot take on the **z** value (see Chapter 5).

A logic state can have a "strength" associated with it. It is a quantitative representation of the internal impedance value of the corresponding hardware circuit; a change in the internal impedance is reflected as a corresponding change in the strength level. Whenever the logic values from two sources are combined, there can be a conflict and the resulting contention has to be resolved. The strength values are discussed below.

2.9 STRENGTHS

The logic levels are also associated with strengths. In many digital circuits, multiple assignments are often combined to reduce silicon area or to reduce pin-outs. To facilitate this, one can assign strengths to logic levels. Verilog has eight strength levels – four of these are of the driving type, three are of capacitive type and one of the hi-Z type. Details are given in Table 3.2 (see also Section 5.4).

When a signal line is driven simultaneously from two sources of different strength levels, the stronger of the two prevails. A few illustrative examples are considered here.

If a signal line a is driven by two sources – b at 1 level with strength "**`strong1`**" and c at level 0 with strength "**`pull0`**"– a will take the value 1.

Details of strengths in Verilog

Strength name	Strength level (signifies inverse of source impedance)	Specification keyword	Abbreviation	Element modeled
Supply drive	7	**`Supply1`** **`Supply0`**	**`Su1`** **`Su0`**	Power supply connection
Strong drive	6	**`Strong1`** **`Strong0`**	**`St1`** **`St0`**	Default gate and assign output strength
Pull drive	5	**`Pull1`** **`Pull0`**	**`Pu1`** **`Pu0`**	Gate and assign output strength
Large capacitor	4	**`Large1`** **`Large0`**	**`La1`** **`La0`**	Size of trireg net capacitor
Weak drive	3	**`Weak1`** **`Weak0`**	**`We1`** **`We0`**	Gate and assign output strength
Medium capacitor	2	**`Medium1`** **`Medium0`**	**`Me1`** **`Me0`**	Size of trireg net capacitor
Small capacitor	1	**`Small1`** **`Small0`**	**`Sm1`** **`Sm0`**	Size of trireg net capacitor
High impedance	0	**`Highz1`** **`Highz0`**	**`Hi1`** **`Hi0`**	Tri-stated line

If a signal line a is driven by two sources – b at 1 level with strength "**pull1**" and c at level 0 with strength "**strong0**," a will take the value 0. If a signal line a is driven by two sources – b at 1 level with strength

"**strong**1" and c at level 0 with strength "**strong0**," a will take the value

x (indeterminate).

If a signal line a is driven by two sources – b at 1 level with strength "**weak1**" and c at level 0 with strength "**large0**," a will take the value 0. (Note that **large** signifies a capacitive drive on a tri-stated line whereas **weak** signifies a gate / assigned output drive with a high source impedance;despite this, due to the higher strength level, the **large** signal prevails.)

The significance of strengths is further explained in Chapter 5.

2.10 DATA TYPES

The data handled in Verilog fall into two categories:

(i) Net data type

(ii) Variable data type

The two types differ in the way they are used as well as with regard to their respective hardware structures. Data type of each variable or signal has to be declared prior to its use. The same is valid within the concerned block or module.

2.10.1 Nets

A net signifies a connection from one circuit unit to another. Such a net carries the value of the signal it is connected to and transmits to the circuit blocks connected to it. If the driving end of a net is left floating, the net goes to the high impedance state. A net can be specified in different ways.

wire: It represents a simple wire doing an interconnection. Only one output isconnected to a wire and is driven by that.

tri: It represents a simple signal line as a wire. Unlike the wire, a tri can bedriven by more than one signal outputs.

Functionally, **wire** and **tri** are identical. Distinct nomenclatures are provided for the convenience of assigning roles.

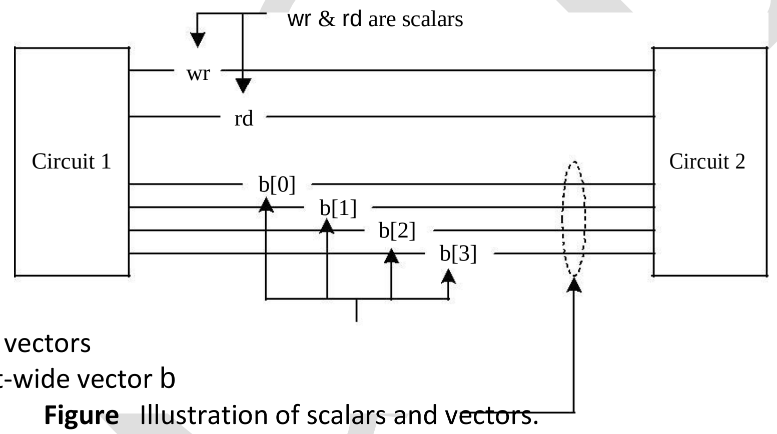
2.10.2 Variable Data Type

A variable is an abstraction for a storage device. It can be declared through the keyword **reg** and stores the value of a logic level: 0, 1, **x**, or **z**. A net or wire connected to a **reg** takes on the value stored in the **reg** and can be used as input to other circuit elements. But the output of a circuit cannot be connected to a **reg**. The value stored in a **reg** is changed through a fresh assignment in the program. **time**, **integer**, **real,** and **realtime** are the other variable types of data;these are dealt with later.

2.11 SCALARS AND VECTORS

Entities representing single bits — whether the bit is stored, changed, or transferred — are called "scalars." Often multiple lines carry signals in a cluster – like data bus, address bus, and so on. Similarly, a group of **reg**s stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector." Figure 3.2 illustrates the difference between a scalar and a vector. wr and rd are two scalar nets connecting two circuit blocks circuit1 and circuit2. b is a 4-bit-wide vector net connecting the same two blocks. b[0], b[1],b[2], and b[3] are the individual bits of vector b. They are "part vectors."

A vector **reg** or net is declared at the outset in a Verilog program and hence treated as such. The range of a vector is specified by a set of 2 digits (or expressions evaluating to a digit) with a colon in between the two. The combination is enclosed within square brackets.



part vectors
4-bit-wide vector b

Figure Illustration of scalars and vectors.

Examples:

wire[3:0]a; /*ais a four bit vector of net type; the bits are designated asa[3], a[2], a[1] and a[0]. */

reg[2:0]b; /*bis a three bit vector of **reg** type; the bits are designated asb[2], b[1] and b[0]. */

reg[4:2]c; /*cis a three bit vector of **reg** type; the bits are designated asc[4], c[3] and c[2]. */

wire[-2:2] d ;/*dis a 5 bit vector with individual bits designated asd[-2],d[-1], d[0], d[1] and d[2]. */

Whenever a range is not specified for a net or a **reg**, the same is treated as a scalar – a single bit quantity. In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values. These can also be expressions evaluating to integer constants – positive or negative.

Normally vectors – nets or **regs** – are treated as unsigned quantities. They have to be specifically declared as "**signed**" if so desired.

Examples

wire signed[4:0]num; //numis a vector in the range -16 to +15. **reg signed** [3:0]num_1; //num_1 is a vector in the range -8 to +7.

2.12 PARAMETERS

In some designs, certain parameter values are not committed at the outset. Proportionality constants, frequency-scaling levels, number of taps in digital filters, *etc*., are typical examples. There are also situations where the size of the design is left open and decided at a later stage. Bus width, LIFO depth, and memory size are such quantities which may be committed later. All such constants can be declared as parameters at the outset in a Verilog module, and values can be assigned to them; for example,

parameter word_size= 16;

parameter word_size= 16,mem_size= 256;

Such parameter assignments are made at compiler time. The parameter values cannot be changed (normally) at runtime. However, a parameter that has been assigned a value in a module definition can have its value changed at runtime – that is, when the module is used at runtime in some other design (*i.e*., instantiated) or when it is tested. Such modifications are carried out through a "**defparameter**" statement. The parameter assignment done as part of parameter declaration can have the appropriate constant on the right-hand side of the assignment statement, as was the case above. The assignment can also have algebraic expressions on the right hand side. Such expressions can involve constants and other parameters declared already; for example

Parameter word_size= 16,factor= word_size/2;

2.13 MEMORY

Different types and sizes of memory, register file, stack, *etc*., can be formed by extending the vector concept. Thus the declaration

Reg [15:0]memory[511:0];

declares an array called "memory"; it has 512 locations. Each location is 16 bits wide. The value of any chosen location can be assigned to a selected register or *vice versa*; this constitutes memory reading or writing [see Example 8.10]. Theindex used to refer a memory location can be a number or an algebraic expression which reduces to an integral value – positive, zero, or negative. As an example, consider the assignment statement

B = mem[(p-q)/2];

The simulator first evaluates (p - q)/2 (which should be an integer): Let it reduce to 3. Then the data stored at mem[3] is assigned to B. Stack pointer, program counter, index register, *etc.*, can be implemented through the above concept. Different types of memory addressing like indirect, indexed, *etc.*, can also be accommodated. Page addressing can be accomplished by a slight adaptation of the concept.

2.14 OPERATORS

Verilog has a number of operators akin to the C language. These are of three types:

1. Unary: the unary operator is associated with a single operand. The operator precedes the operand – for example, ~a.
2. Binary: the binary operator is associated with two operands. The operator appears between the two operands – for example, a&b.
3. Ternary: the ternary operator is associated with three operands. The two operators together constitute a ternary operation. The two operators separate the three operands – for example

   a?b:c // Here the operators "?" and ":" together define an operation.

2.15 SYSTEM TASKS

During the simulation of any design, a number of activities are to be carried out to monitor and control simulation. A number of such tasks are provided / available in Verilog. Some other tasks serve other functions. However, a few of these are used commonly; these are described here. The "$" symbol identifies a system task. A task has the format

$<keyword>

2.15.1 $display

When the system encounters this task, the specified items are displayed in the formats specified and the system advances to a new line. The structure, format, and rules for these are the same as for the "printf" / "scanf" function in C. Refer to a standard text in "C" language for the text formatting codes in common usage [Gottfried].

Examples

$display ("The value ofais : a = , %d",a);

Execution of this line results in printing the value of a as a decimal number (specified by "%d"). The string present within the inverted commas specifies this. Thus if a has the value 3.5, we get the display

The value of a is : a = 3.5.

After printing the above line, the system advances to the next line.

$display; /* This is a display task without any arguments. It advancesoutput to a new line. */

2.15.2 $monitor

The **$monitor** task monitors the variables specified whenever any one of those specified changes. During the running of the program the monitor task is invoked and the concerned quantities displayed whenever any one of these changes. Following this, the system advances to the next line. A monitor statement need appear only once in a simulation program. All the quantities specified in it are continuously monitored. In contrast, the **$display** command displays the quantities concerned only once – that is, when the specific line is encountered during execution. The format for the **$monitor** task is identical to that of the **$display** task.

Examples

$monitor ("The value ofais :a= , %d",a);
With the task, whenever the value of a changes during execution of a program, its new value is printed according to the format specified. Thus if the value of a changes to 2.4 at any time during execution of the program, we get the following display on the monitor.
The value of a is: a = 2.4.

2.15.3 Tasks for Control of Simulation

Two system tasks are available for control of simulation:
$**finish** task, when encountered, exits simulation. Control is reverted to the Operating System. Normally the simulation time and location are also printed out by default as part of the exit operation.
$**stop** task, suspends simulation; if necessary the simulation can be resumed by user intervention. Thus with the stop task, the simulator is in an interactive mode. In contrast with $finish, simulation has to be started afresh.

2.16 EXERCISES

1. Run the Verilog program in Figure 3.3. Observe the output.

```
module fancy2; integer i,j; initial repeat(5) begin

    #1      j=0; while(j<=10) begin

    j=j+1;

    for(i=0;i<=j;i=i+1) $write(" b"); $display("*");

    end

#1      while(j>=0)
        begin
    for(i=0;i<=j;i=i+1) $write(" c"); $display("*");

    j=j-1;

    end
        end
        initial #12 $stop; endmodule
```

Figure A simple Verilog module.

2. In Exercise 3.1 above, delete b and c in the write statement lines. Rerun the program.
3. Try other combinations of I and j values and repeat the run.
4. Run the Verilog program in Figure 3.4.
5. In the program of Figure 3.4 replace the "`always`" statement by "`initial`" statement and run the program.
6. In the program of Figure 3.4 replace the "a=a+7" statement by "a=a-7" statement and run the program.

```
          module fancy3; reg[11:0]a; always
          begin
#0     $display("See this:      ah=%d, ad=%h, ao=%o, ab=%b",a,a,a,a);
#1     $display("How about this? ah=%0d, ad=%0h, ao=%0o, ab=%0b",a,a,a,a); a=a+7;
          end initial begin
       a=0;

       #10 $stop;

          end endmodule
```

Figure Another simple Verilog module.

3

GATE LEVEL MODELING

INTRODUCTION

Digital designers are normally familiar with all the common logic gates, their symbols, and their working. Flip-flops are built from the logic gates. All other functionally complex and more involved circuits can also be built using the basic gates. All the basic gates are available as "Primitives" in Verilog. Primitives are generalized modules that already exist in Verilog [IEEE]. They can be instantiated directly in other modules. Further design description using gate primitives is quite close to the actual circuits (design description using the switch primitives dealt with in Chapter 10 are still closer). We describe features of gate level primitives, ways of working with them, and ways of building more involved circuits with them [Palnitkar, Lee]. In this process some of the commonly used features of Verilog are also brought out.

3.1 AND GATE PRIMITIVE

The AND gate primitive in Verilog is instantiated with the following statement: **`and`** g1 (O, I1, I2, . . ., In);
Here '**`and`**' is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. O is the gate output; I1, I2, *etc*., are the gate inputs. The following are noteworthy:

The AND module has only one output. The first port in the argument list is the output port.

An AND gate instantiation can take any number of inputs — the upper limit is compiler-specific.

A name need not be necessarily assigned to the AND gate instantiation; this is true of all the gate primitives available in Verilog.

3.1.1 Example 3.1

Figure 4.1 shows the stimulus program for testing the AND gate g1. The output obtained by stimulating the program is shown in Figure 4.2. Some explanation regarding the simulation program is in order here.
The module test_and has no port. It instantiates the AND module once.
The test input sequence is specified within the **`initial`** block – the sequence of statements between the **`begin`** and **`end`** statements together form

this block.

The keyword "**initial**" signifies the settings done initially — that is, only once for the whole routine.

The first set of statements within the **initial** block
a1 = 0;

a2 = 0;make

a1 = a2 = 0
at zero simulation time.

After 3 time steps, a1 is set to one but a2 remains at 0. The expression "#3" means "after 3 time steps". Subsequent changes in a1 and a2 also can be explained in the same manner.

```
module test_and; reg a1, a2; wire b;
Initial
Begin
    a1 = 0;
 a2 = 0; #3 a1 = 1; #1 a1 = 0; #2 a2 = 1; #4 a1 = 1; #3 a2 = 0; #1 a2 = 1; end
    and g1(b, a1, a2);
    initial $monitor ( $time, "a1 = %b, a2 = %b, b = %b"" a1, a2, b);
    initial #100 $finish;
    endmodule
```

Figure 3.1 A module to instantiate the AND gate primitive and test it.

```
0 a1 = 0 a2 = 0 b = 0
3 a1 = 1 a2 = 0 b = 0
4 a1 = 0 a2 = 0 b = 0
6 a1 = 0 a2 = 1 b = 0
10 a1 = 1 a2 = 1 b = 1
13 a1 = 1 a2 = 0 b = 0
14 a1 = 1 a2 = 1 b = 1
```

Figure 3.2 The output obtained by running the module of Figure 4.1.

The program displays the variable values – that is, the values of o, a1, and a2 whenever any one of these changes. This is evident from the printout on the

monitor, which has been reproduced in Figure 4.2.

A pair of variables a1 and a2 are declared in the program, and the values stored in them are given as inputs to the AND gate instantiation.

Any variable not declared in the module is by default taken as a net of wire type; it is also taken as a scalar. The same is true of all modules in Verilog.

The term **$time** in the **$monitor** statement signifies the running time of the program. Here it causes the value of time at the instant of capturing the

data for display, to be displayed. The statement

#100$finish;

signifies that the program will stop simulation and exit the operating system at the end of 100 time steps.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table 4.1. It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

Table 3.1 Truth table of AND gate primitive

		Input 1			
		0	1	*x*	*z*
Input2	0	0	0	0	0
	1	0	1	*x*	*x*
	x	0	*x*	*x*	*x*
	z	0	*x*	*x*	*x*

If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, **x** or **z** state.

The output is at 1 state if and only if every one of the inputs is at 1 state. For all other cases the output is at the **x** state.

Note that the output is never at the **z** state – the high impedance state. This is true of all other gate primitives as well.

3.2 MODULE STRUCTURE

Figure 4.1 shows a typical module. In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

The first statement of a module starts with the keyword **`module`**; *it may be* followed by the name of the module and the port list if any (see Section 2.8). All the variables in the ports-list are to be identified as **`inputs`**, **`outputs,`** or **`inouts`**. The corresponding declarations have the form shown below:

Input a1, a2; Output b1, b2; Inout c1, c2;

The port-type declarations here follow the module declaration mentioned above.

The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case. The respective declaration statements follow the port-type declaration statements.

Examples:

`wire` a1, a2, c; **`reg`** b1, b2;

The type declaration must necessarily precede the first use of any variable or signal in the module.

The executable body of the module follows the declaration indicated above. The last statement in any module definition is the keyword "**`endmodule`**". Comments can appear anywhere in the module definition.

3.3 OTHER GATE PRIMITIVES

All other basic gates are also available as primitives in Verilog. Details of the facilities and instantiations in each case are given in Table 4.2. The following points are noteworthy here:

In all cases of instantiations, one need not necessarily assign a name to the instantiation. It need be done only when felt necessary – say for clarity of

circuit description.

In all the cases the output port(s) is (are) declared first and the input port(s) is

(are) declared subsequently.

The buffer and the inverter have only one input each. They can have any number of outputs; the upper limit is compiler-specific. All other gates have one output each but can have any number of inputs; the upper limit is again compiler-specific.

3.3.1 Truth Table

Extending the concepts of Section 4.2.2, one can form the truth tables of all other gate primitives. The basic features of each are given in Table 4.3. The truth tables themselves are given in Appendix B.

3.4 ILLUSTRATIVE EXAMPLES

The examples considered here illustrate the use of gate primitives in designs. Further, they bring out how one can build fairly large designs by judiciously combining smaller modules in a repeated fashion [Bignel, Sedra].

Table 3.2 Basic gate primitives in Verilog with details

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	**and** ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	**or** gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	**nand** gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	**nor**gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	**xor** gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	**xnor** gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	**buf** gb (o1, o2, …. i);	o1, o2, o3, . .	i
NOT	**not** gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Table 3.3 Rules for deciding the output values of gate primitives for different input combinations

Type of gate	0 output state	1 output state	**x** output state
AND	Any one of the inputs is zero	All the inputs are at one	All other cases
NAND	All the inputs are at one	Any one of the inputs is zero	
OR	All the inputs are at zero	Any one of the inputs is one	
NOR	Any one of the inputs is one	All the inputs are at zero	
XOR	If every one of the inputs is definite at zero or one, the output is zero or one as decided by the XOR or XNOR function		If any one of the inputs is at **x** or **z** state, the output is at **x** state
XNOR			
BUF	If the only input is at 0 state	If the only input is at 1 state	All other cases of inputs
NOT	If the only input is at 1 state	If the only input is at 0 state	

3.4.1 Example 3.2

The commonly used A-O-I gate is shown in Figure 4.3 for a simple case. The module and the test bench for the same are given in Figure 4.4. The circuit has been realized here by instantiating the AND and NOR gate primitives. The names of signals and gates used in the instantiations in the module of Figure 4.4 remain the same as those in the circuit of Figure 4.3. The module aoi_gate in the figure has input and output ports since it describes a circuit with signal inputs and an output. The module aoi_st is a stimulus module. It generates inputs to the aoi_gate module and gets its output. It has no input or output ports.

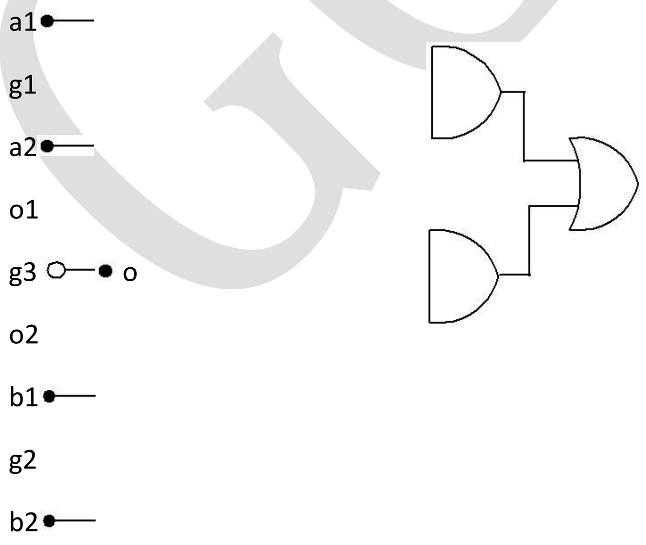


Figure 3.3 A typical A-O-I gate circuit.

```
/*module for the aoi-gate of figure 4.3 instantiating the gate primitives - fig4.4*/
module aoi_gate(o,a1,a2,b1,b2);
input a1,a2,b1,b2;// a1,a2,b1,b2 form the input //ports of the module
output o;//o is the single output port of the module wire o1,o2;//o1 and o2 are
intermediate signals //within the module
and g1(o1,a1,a2); //The AND gate primitive has two and g2(o2,b1,b2);//
instantiations with assigned //names g1 & g2.
nor g3(o,o1,o2);//The nor gate has one instantiation //with assigned name g3.
endmodule
//Test-bench for the aoi_gate above module aoi_st;
reg a1,a2,b1,b2;
//specific values will be assigned to a1,a2,b1, // and b2 and these connected
//to input ports of the gate insatntiations; //hence these variables are declared as reg
wire o;
initial begin
a1 = 0;
a2 = 0;

b1 = 0;

b2 = 0; #3 a1 = 1; #3 a2 = 1; #3 b1 = 1; #3 b2 = 0; #3 a1 = 1; #3 a2 = 0; #3 b1 = 0;

end
initial #100 $stop;//the simulation ends after //running for 100 tu's.
initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b
",o,a1,a2,b1,b2); aoi_gate gg(o,a1,a2,b1,b2);
endmodule
```

Figure 3.4 Module for the AOI gate of Figure 4.3 and a test bench for the same.

The A-O-I gate module has three instantiations – two of these being AND gates and the third a NOR gate; this conforms to the circuit of A_O_I gate in Figure 4.3. Within the aoi_gate module, all signals are of type net. The aoi_ gate module in Figure 4.4 is instantiated once in the module aoi_st for testing. Any such instantiation of a user-defined module in another module has to be assigned a name. (As mentioned earlier, this is not mandatory with the instantiation of gate primitives available in Verilog.) The instantiation is given the name gg here. Note that all the inputs to the instantiation of aoi_gate in the test bench are fed through **`reg`**s.

The aoi_gate and aoi_st are compiled and run. Different combinations of values are assigned to a1, a2, b1, and b2 in the test bench at regular intervals of 3 time steps. At all such time steps at least one of the signals included in the monitor statement changes. Hence all the signal values are displayed on the monitor at three time step intervals. The results of running the test bench are reproduced in Figure 4.5, which confirms this.

The module aoi_gate has been synthesized and the synthesized circuit shown in Figure 4.6; the figure does not warrant any detailed explanation.

Both the modules can do with some elegant simplification. First consider the stimulus module aoi_st in Figure 4.4. All the four inputs can be clubbed together and treated as a “vector” input. Often this may be possible to be identified with a four-bit-wide bus in a system. It makes the vector representation all the more meaningful. With this, the variables together can be declared as a single vector. The value taken by the vector can be defined with relevant time delays. To accommodate such a change, the AOI module of Figure 4.4 is recast in Figure 4.7. The compactness achieved here is carried over to the instantiation of the module for its test bench aoi_st2, which is also shown in the figure.

The AOI gate itself (aoigate2 in Figure 4.7) has been made compact on two counts: All the four inputs have been clubbed together and treated as a four-bit vector. Further, the two and gate instantiations are clubbed together into one statement. Note the format of the statement – a comma separates the two instantiations, and as usual a semicolon signifies the end of the statement. In any set of instantiations, all similar instantiations in a module can be combined in this manner. The module aoigate2 has an input/output port since it describes a circuit with signal inputs and outputs. aoi_st2 is a stimulus module. It generates inputs

```
#   0  o = 1 ,  a1 = 0 , a2 = 0 , b1 = 0 ,b2 = 0
#   3  o = 1 ,  a1 = 1 , a2 = 0 , b1 = 0 ,b2 = 0
#   6  o = 0  , a1  = 1 , a2  = 1 , b1  = 0 ,b2 = 0
#   9  o = 0  , a1  = 1 , a2  = 1 , b1  = 1 ,b2 = 0
#  18  o = 1  , a1  = 1 , a2  = 0 , b1  = 1 ,b2 = 0
#  21  o = 1  , a1  = 1 , a2  = 0 , b1  = 0 ,b2 = 0
```

Figure 3.5 Results of running theaoi_sttest bench of Figure 3.3.

to the module from within the stimulus module and gets its output. It has no input or output port. In a more general case one may have a number of modules defined at different levels, which are repeatedly instantiated in bigger modules. The stimulus module may be at the apex. It may carry out the stimulus activity by generating the inputs to the other ports in the hierarchy and receiving their outputs.

```
module aoi_gate2(o,a);
input [3:0]a;//A is a vector of 4 bits width output o;// output o is a scalar
wire o1,o2;//these are intermediate signals and (o1,a[0],a[1]),(o2,a[2],a[3]);
nor (o,o1,o2);/*The nor gate has one instantiation with assigned name g3.*/
endmodule
module aoi_st2; reg[3:0] aa; aoi_gate2 gg(o,aa); initial
begin

aa = 4'b000;//a being a vector, all its #3 aa = 4'b0001;//bit components are
```

#3 aa = 4’b0010;//assigned values at one go. #3 aa = 4’b0100;//Similarly their changes are #3 aa = 4’b1000;//combined in the assignments #3 aa = 4’b1100;

#3 aa = 4’b0110; #3 aa = 4’b0011;

end initial

$monitor($time , “ aa = %b , o = %b “ , aa,o); initial #24 $stop;
endmodule

Figure 3.7 Another realization of the A-I-O gate with the input declared as a vector; the testbench for the module is also shown in the figure.

The stimulus module need not necessarily have a port; aoi_st in Figure 4.4 and aoi_st2 in Figure 4.7 are typical examples. The results of running the test bench aoi_st2 of Figure 4.7 are shown in Figure 4.8.

To facilitate involved design descriptions, some additional flexibility is available in Verilog.

Signals at the ports can be identified by a hierarchical name. Such addressing may become useful when displaying them in the stimulus module.

Signal instantiations illustrated above specify inputs and outputs in the same sequence as was done in the definition. The procedure is simple and acceptable in situations with only a few numbers of inputs and outputs. But in modules with a comparatively large number of inputs and outputs, sticking to the sequence and keeping track of it becomes strenuous. In such situations the instantiation can be done by identifying the inputs and outputs on a one-to-one basis [see Section 2.8]. Thus the instantiation of the aoi_gate2 in the test bench of Figure 4.7 can be described alternately as

```
aoigate2 gg (.o(o), .a[1](aa[1]), .a[2](aa[2]), .a[3](aa[3]), .a[4](aa[4]) );
```

Here one need not stick to the same order of assignment of the ports as in the module definition. Thus the instantiation entered as

```
aoigate2 gg (.a[1](aa[1]), .o(o),.a[2](aa[2]), .a[4](aa[4]), a[3](aa[3]) );
```

is equally valid.

3.4.2 Example 4.3: 4-to-16 Decoder

Decoder design using gates can be described in various ways. Here we define a 2-to-4 decoder module and instantiate it repeatedly and judiciously to realize a 4-to-16 decoder. The procedure is not necessarily the best or most elegant.

```
#        0  aa = 0000 , o = 1
#        3  aa = 0001 , o = 1
#        6  aa = 0010 , o = 1
#        9  aa = 0100 , o = 1
#       12  aa = 1000 , o = 1
#       15  aa = 1100 , o = 0
#       18  aa = 0110 , o = 1
#       21  aa = 0011 , o = 0
```

Figure 3.8 Results of running theaoi_st2test bench of Figure 4.7.

Figure 3.9© shows the formation of the 4-to-16 decoder in terms of two numbers of 3-to-8 decoders. The 3-to-8 decoders have an “Enable” input each (designated ‘en’ – one being of the active high and the other of the active low type); these are connected to the most significant bit of the 4-bit input to form the 4-to-16 decoder. The 3-to-8 decoder can again be formed in terms of two 2-to-4 decoders in the same manner as shown in Figure 4.9(b). The 2-to-4 decoder block used here is shown in Figure 4.9(a). The logic of building a complex circuit unit in terms of repeated use of smaller and smaller circuit units followed here is used in the design description as well. Figure 4.10 shows the design description of a 2-to-4 decoder module and a test bench for the same. The decoder module (dec2_4) accepts a 2-bit-wide vector input b and decodes it into a 4-bit-wide vector output a. It has an additional “Enable” input designated “en”; the outputs are enabledonly if en = 1. The input en has been introduced to facilitate expansion of the decoder capacity by repeated instantiation as explained above. The test bench for the decoder is more illustrative than exhaustive; that is, it does not test the module for all possible input values. Results of the simulation run are shown in Figure 4.11.

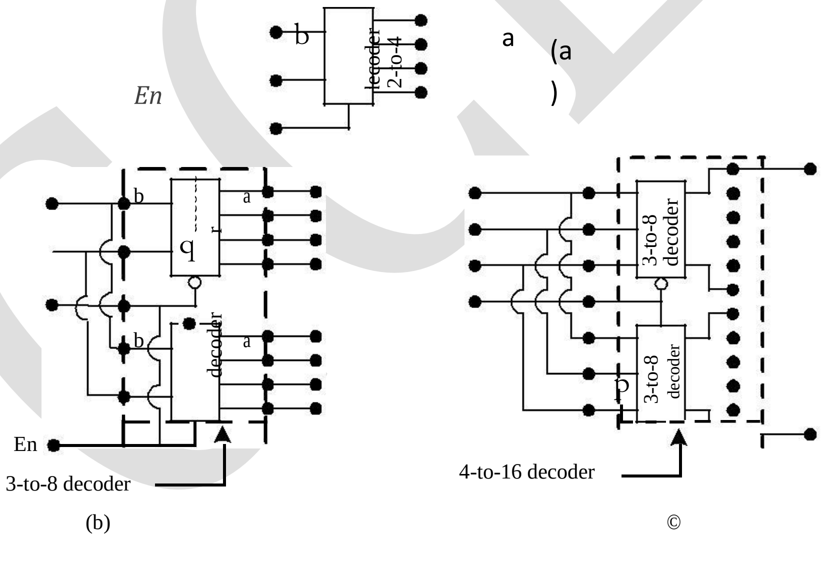


Figure 3.9 Formation of 4-to-16 decoder circuit in terms of smaller decoders: (a) 2-to-4decoder, (b) 3-to- 8 decoder in terms of two 2-to-4 decoders, and (c) 4-to-16 decoder in terms of two 3-to-8 decoders.

```
module dec2_4 (a,b,en); output [3:0] a;

input [1:0]b; input en; wire [1:0]bb;

not(bb[1],b[1]),(bb[0],b[0]);

and(a[0],en, bb[1],bb[0]),(a[1],en, bb[1],b[0]), (a[2],en,
b[1],bb[0]),(a[3],en, b[1],b[0]); endmodule

//test bench

module tst_dec2_4(); wire [3:0]a; reg[1:0] b; reg en; dec2_4
dec(a,b,en); initial

begin

{b,en} =3'b000; #2{b,en} =3'b001; #2{b,en} =3'b011; #2{b,en}
=3'b101; #2{b,en} =3'b111; end

initial

$monitor ($time , "output a = %b, input b = %b ", a, b);

endmodule
```

Figure 3.10 Design description of a 2-to-4 decoder circuit and its test bench.

Figure 4.12 shows a 3-to-8 decoder module formed by repeated instantiation of the 2-to-4 decoder of Figure 4.10. The eight AND gate instantiations ensure that the outputs are enabled only when enn — a separate "Enable" signal — goes active. Following the same logic, the module for the 4-to-16 decoder is described in Figure 4.13. A test bench to test the module through all the possible input states is also included in the figure. Figure 4.14 shows the results of running the test-bench.

```
//output
//#         0 output a =  0000, input b   = 00
//#         2 output a =  0001, input b   = 00
//#         4 output a =  0010, input b   = 01
//#         6 output a =  0100, input b   = 10
//#         8 output a =  1000, input b   = 11
```

Figure 3.11 Results of running the test bench of Figure 4.10.

```
module dec3_8(pp,q,enn); output[7:0]pp; input[2:0]q;
input enn; wire qq; wire[7:0]p; not(qq,q[2]);
dec2_4 g1(.a(p[3:0]),.b(q[1:0]),.en(qq)); dec2_4 g2(.a(p[7:4]),.b(q[1:0]),.en(q[2])); and
g30(pp[0],p[0],enn);
and g31(pp[1],p[1],enn); and g32(pp[2],p[2],enn); and g33(pp[3],p[3],enn); and
g34(pp[4],p[4],enn); and g35(pp[5],p[5],enn); and g36(pp[6],p[6],enn); and
g37(pp[7],p[7],enn); endmodule
```

Figure 3.12 A 3-to-8 decoder module formed by repeated instantiation of the 2-to-4decoder module in Figure 4.10.

```
module dec4_16(m,n); output[15:0]m; input[3:0]n;
wire nn; //wire en; not(nn,n[3]);
dec3_8 g3(.pp(m[7:0]),.q(n[2:0]),.enn(nn)); dec3_8
g4(.pp(m[15:8]),.q(n[2:0]),.enn(n[3])); endmodule
//test-bench
module dec4_16_stimulus; wire[15:0]m;
//wire l,m,n; reg[3:0]n; dec4_16 gg(m,n); initial
begin n=4'b0000;#2n=4'b0000;#2n=4'b0001;
#2n=4'b0010;#2n=4'b0011;#2n=4'b0100;
#2n=4'b1000;#2n=4'b1001;#2n=4'b1010;

#2n=#2n=4'b0101;#2n=4'b0110;#2n=4'b0111;

4'b1011;#2n=4'b1100;#2n=4'b1101;
#2n=4'b1110;#2n=4'b1111;#2n=4'b1111; end
initial $monitor($time," m = %b ,n = %b , gg.g3.qq = %b , gg.g4.g1.bb = %b " ,
m,n,gg.g3.qq,gg.g4.g1.bb); //gg.g3.qq displays the enable line of dec3_8 called g3-g1
//gg.g4.g1.bb displays the bb wire in dec2_4 initial #40 $stop ;
endmodule
```

Figure 3.13 A 4-to-16 decoder module formed by repeated instantiation of the 3-to-8decoder module of Figure 4.12. A test bench for the same is also shown.

```
//output
//#       0 m = 0000000000000001 ,n = 0000 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 11
//#       4 m = 0000000000000010 ,n = 0001 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 10
//#       6 m = 0000000000000100 ,n = 0010 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 01
//#       8 m = 0000000000001000 ,n = 0011 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 00
//#      10 m = 0000000000010000 ,n = 0100 ,
gg.g3.qq = 0 , gg.g4.g1.bb = 11
//#      12 m = 0000000000100000 ,n = 0101 ,
gg.g3.qq = 0 , gg.g4.g1.bb = 10
//#      14 m = 0000000001000000 ,n = 0110 ,
gg.g3.qq = 0 , gg.g4.g1.bb = 01
//#      16 m = 0000000010000000 ,n = 0111 ,
gg.g3.qq = 0 , gg.g4.g1.bb = 00
//#      18 m = 0000000100000000 ,n = 1000 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 11
//#      20 m = 0000001000000000 ,n = 1001 ,
gg.g3.qq = 1 , gg.g4.g1.bb = 10
//#      22 m = 0000010000000000 ,n = 1010 ,
```

```
gg.g3.qq = 1 , gg.g4.g1.bb = 01
 //#      24 m = 0000100000000000 ,n = 1011 ,
 gg.g3.qq = 1 , gg.g4.g1.bb = 00
 //#      26 m = 0001000000000000 ,n = 1100 ,
 gg.g3.qq = 0 , gg.g4.g1.bb = 11
 //#      28 m = 0010000000000000 ,n = 1101 ,
 gg.g3.qq = 0 , gg.g4.g1.bb = 10
 //#      30 m = 0100000000000000 ,n = 1110 ,
 gg.g3.qq = 0 , gg.g4.g1.bb = 01
 //#      32 m = 1000000000000000 ,n = 1111 ,
 gg.g3.qq = 0 , gg.g4.g1.bb = 00
```

Figure 3.14 Results of running the test bench of Figure 4.13 for the 4-to-16 decoder

Observations:–

The nested tree of modules with the inputs and outputs in each case are shown in Figure 3.15.

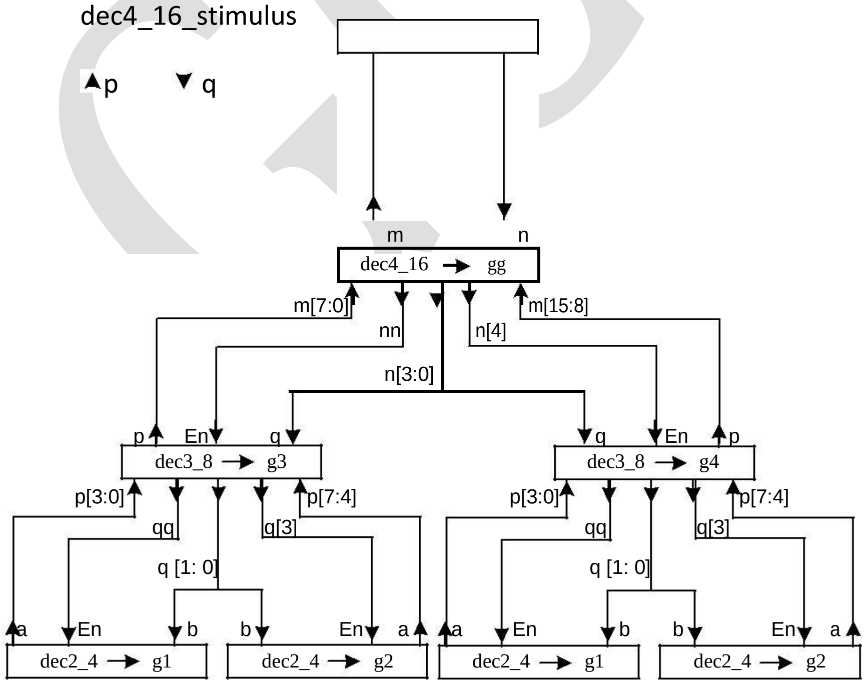


Figure 3.15 Block diagram representation of the module instantiations and signalassignments for the stimulus module of Figure 4.10.

Two signals within the two nested modules are monitored in dec4_16_stimulus. Formation of their hierarchical addresses is also shown in

Figure 3.15. (Hierarchical addressing is addressed in detail in Chapter 11.)

The module dec3_8 is instantiated twice in the module dec4_16. Here the port declarations are done by declaring the port names on a one-to one basis. The order has not been maintained as in the defining module.

4.5.2.1 Decoder Synthesis

The synthesized circuit of the 2-to-4 decoder module of Figure 4.10 (dec2_4) is shown in Figure 4.16. The AND gate cells available in the library are all of the two-input type; hence six such cells (designated as ix5, ix7, ix11, ix13, ix15, and ix19) are utilized to realize the four numbers of three-input AND gates instantiated in the design module. The NOT gates are realized through two NOT gate cells in the library (designated as ix1 and ix3). The wider lines in the figure signify bus-type interconnections. The synthesized circuit of the 3-to-8 decoder module of Figure 4.12 (dec3_8) is shown in Figure 4.17. The two instantiations of the dec2_4 module (g1 and g2) are shown as black boxes. Similarly, Figure 4.18 shows the synthesized circuit of the 4-to-16 decoder module of Figure 4.13 (dec4_16). The two instantiations of the dec3_8 module (g3 and g4) appear as black boxes inside. Figure 4.19 shows the complete hierarchy of instantiations in the synthesized circuit. In the figure boxes g3 and g4 represent instantiations of the 3-to-8 decoders used in the module. Each of these has two numbers of the 2-to-4 decoders – designated as g1 and g2; these are shown enclosed inside boxes.

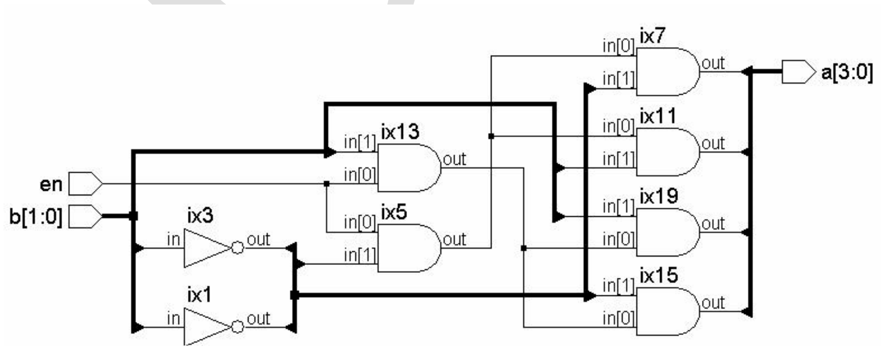


Figure 3. 16 The synthesized circuit of the 2-to-4 decoder of Figure 4.10.

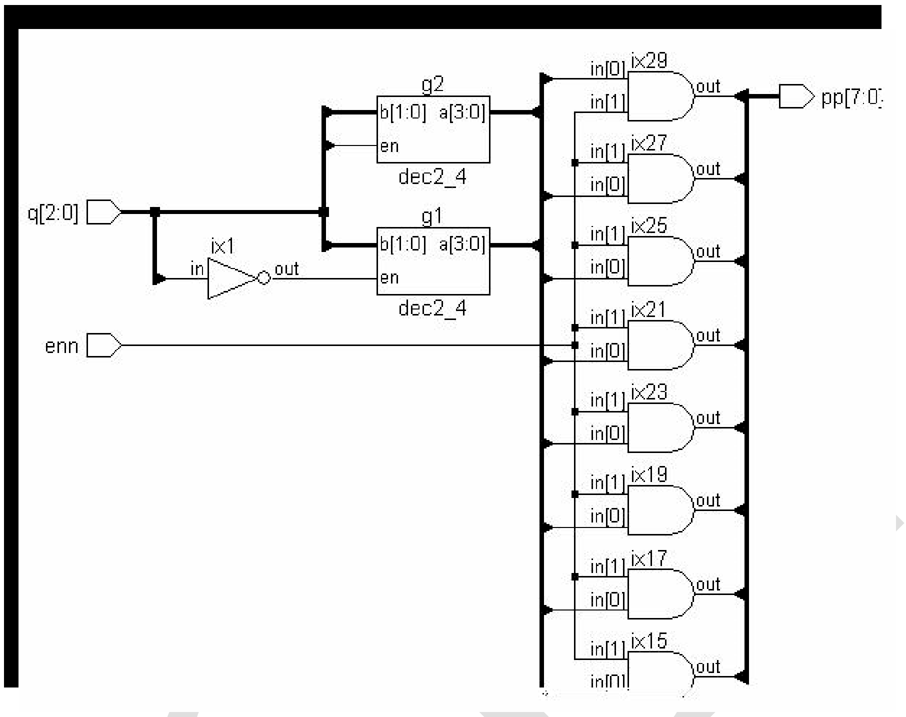


Figure 3.17 The synthesized circuit of the 3-to- 8 decoder of Figure 4.12.

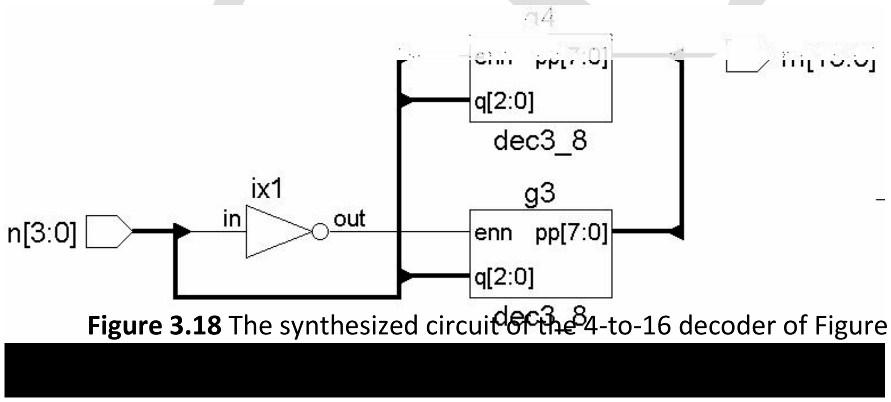


Figure 3.18 The synthesized circuit of the 4-to-16 decoder of Figure 4.13.

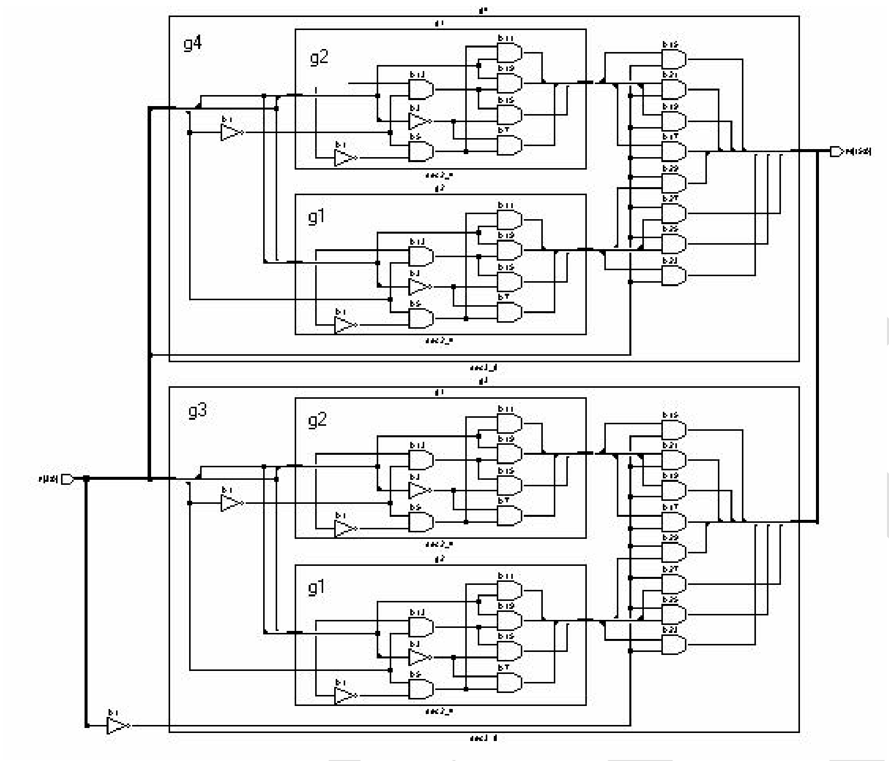


Figure 3.19 Four-to-sixteen decoder – hierarchy of instantiations.

3.6TRI-STATE GATES

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a control signal. The direct buffer is instantiated as
`Bufif1` nn (out, in, control);

The symbol of the buffer is shown in Figure 4.20. We have
out as the single output variable in as the single input variable and

control as the single control signal variable.
When control = 1, out = in.

in out

control

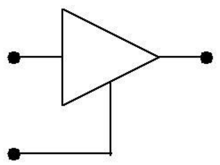
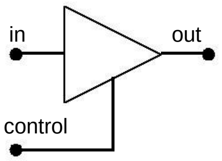
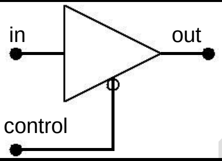
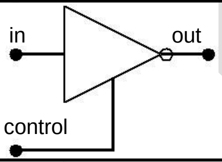
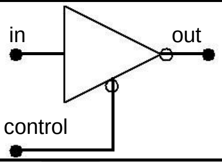


Figure 3.20 *A tri-state buffer.*

When control = 0,
out is cut off from the input and tri-stated. The output, input and control signalsshould appear in the instantiation in the same order as above. Details of bufif1 as well as the other tri-state type primitives are shown in Table 4.4. In all the cases shown in Table 4.4, out is the output, in is the input, and control, the control variable.

Table 3.4 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
bufif1 (out,in, control);		Out = in if control = 1; else out = z
bufif0 (out,in, control);		Out = in if control = 0; else out = z
notif1 (out,in, control);		Out = complement of in if control = 1; else out = z
notif0 (out,in, control);		Out = complement of in if control = 0; else out = z

The truth tables of the tri-state buffers are given in Appendix B. The following observations are common to all the tri-state buffer primitives:

If the control signal has a value that corresponds to the buffer being on, two possibilities exist:

The output has the same value as the input if the input is 0 or 1. The output is at **x** otherwise (*i.e.*, if the input is **x** or **z**).

If the control signal has a value that corresponds to the control signal being off, the output is at **z** state irrespective of the value of the input. If the control signal is at **x** or **z**, three possibilities arise:

If the input is at **x** or **z**, the output is at **x**.

If the input is at 0 state, the output is **L** for bufif1 and bufif0. It is at

H for notif1 and notif0.

If the input is at 1 state, the output is **H** for bufif1 and bufif0. It is at **L** for notif1 and notif0.

Note that **H** corresponds to 1 or **z** state while **L** corresponds to 0 or **z** state.

3.7ARRAY OF INSTANCES OF PRIMITIVES

The primitives available in Verilog can also be instantiated as arrays. A judicious use of such array instantiations often leads to compact design descriptions. A typical array instantiation has the form

and gate [7 : 4] (a, b, c);

where a, b, and c are to be 4 bit vectors. The above instantiation is equivalent to combining the following 4 instantiations:

and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1],c[1]), gate [4] (a[0], b[0], c[0]);

The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself. A more general instantiation of array type has the form

and gate[mm : nn](a, b, c);

where mm and nn can be expressions involving previously defined parameters, integers and algebra with them. The range for the gate is 1+ (*mm-nn*); *mm* and *nn* do not have restrictions of sign; either can be larger than the other.

3.7.1 Example 4.4 A Byte Comparator

A circuit to compare two variables each of one byte is given in Figure 3.21. The circuit outputs a flag d; d is 1 if the two bytes are equal; else it is 0. The output is activated only if the enable signal en = 1. If en = 0, the circuit output is tri-stated. The module description is given in Figure 3.22 along with a test-bench. The simulated output is in Figure 3.23.

Observations:

In all array-type instantiations, the array sizes are to be matched.

The order of assignments to outputs, inputs, *etc*., in the individual gates will be decided by the order of the bits. Thus the array instantiation

or gg[3:1] (a[3:1], b[4:2], c);

is equivalent to the combination of instantiations

or gg[3] (a[3], b[4], c[2]), gg[2] (a[2], b[3], c[1]), gg[1] (a[1], b[2], c[0]);

If the vector sizes in the port list do not match the array size specified, assignments will be done starting from the right; that is, the rightmost instantiation will be assigned the rightmost inputs and outputs and the following instantiations will be made assignments in the order specified. However, it is desirable to avoid such ill-matched instantiations.

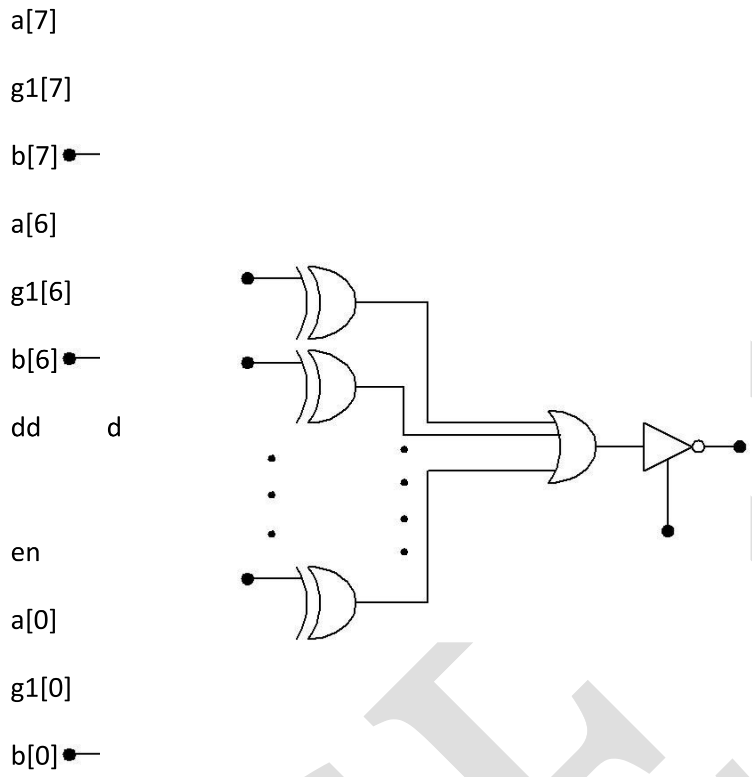


Figure 3.21 A byte comparator.

In the general case the array size is specified in terms of two constant expressions. These can involve constants, previously defined parameters and algebraic operators: Such an instantiation can have a form as

and gate [offset*2+size-1: offset*2] (a, b, c);

where 'offset' and 'size' are parameters whose values should have been assigned earlier (operators are discussed in detail in Chapter 6).

```
        module comp(d,a,b,en); input en; input[7:0]a,b;
        output d; wire [7:0]c; wire dd;
        xor g1[7:0](c,b,a); or(dd,c); notif1(d,dd,en); endmodule
        module comp_tb; reg[7:0]a,b; reg en;
        comp gg(d,a,b,en); initial
begin

a = 8'h00; b = 8'h00; en = 1'b0; end

        always
        #2 en = 1'b1; always
        begin
        #2 a = a+1'b1; #2 b = b+2'd2; end
        initial $monitor($time," en = %b , a = %b ,b = %b ,d = %b
        ",en,a,b,d);
        initial #30 $stop; endmodule
```

Figure 3.22 Module of an 8-bit comparator and its test bench.

```
#    0 en = 0, a = 00000000, b = 00000000, d = z
#    2 en = 1, a = 00000001, b = 00000000, d = 0
#    4 en = 1, a = 00000001, b = 00000010, d = 0
#    6 en = 1, a = 00000010, b = 00000010, d = 1
#    8 en = 1, a = 00000010, b = 00000100, d = 1 #10 en = 1, a =
     00000011, b = 00000100, d = 0 #12 en = 1, a = 00000011, b =
     00000110, d = 0 #14 en = 1, a = 00000100, b = 00000110, d = 1
     #16 en = 1, a = 00000100, b = 00001000, d = 1 #18 en = 1, a =
     00000101, b = 00001000, d = 0 #20 en = 1, a = 00000101, b =
     00001010, d = 0 #22 en = 1, a = 00000110, b = 00001010, d = 1
     #24 en = 1, a = 00000110, b = 00001100, d = 1 #26 en = 1, a =
```

Figure 3.23 Results of the simulation run of the test bench in Figure 4.22.

3.9ADDITIONAL EXAMPLES

A set of representative examples is discussed here with the following aims:–

Bring out the flexibility associated with the use of primitives and their instantiations.

Illustrate the use of different features of Verilog discussed in the chapter.

Focus attention on the fact that any combinational circuit can be designed at the gate level.

Details of the examples considered are summarized in Table 3.5

Table 3.5 Summary of the examples considered in Section 3.8

Circuit function	Figure numbers			Remarks
	Module & Test-bench	Simulation results	Synthesized circuit	
Half-adder	4.24	4.25	4.26	
Full-adder	4.27	4.28	4.29 & 4.30	Instantiates the half-adder twice as ha1 and ha2 in Figure 4.27
2-to-1 Mux	4.37	4.38	4.39	Realized with tri-state buffers
4-to-1 Mux	4.31	4.32	4.33	Simple & direct
	4.34	4.35	4.36	The above type with an additional tri-state output facility
	4.40	4.41	4.42	Realized with tri-state buffers

```
module ha(s,ca,a,b); input a,b;
output s,ca; xor(s,a,b); and(ca,a,b); endmodule
//test-bench module tstha(); reg a,b;
wire s,ca;
ha hh(s,ca,a,b); initial
begin a=0;b=0; end always begin
#2 a=1;b=0; #2 a=0;b=1; #2 a=1;b=1; #2 a=0;b=0; end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum =
%b " ,a,b,ca,s);
initial #24 $stop; endmodule
```

Figure 3.24 Design module and a test bench for a half-adder.

output

```
#    0  a = 0 , b = 0 ,out carry = 0 , outsum = 0
#    2  a = 1 , b = 0 ,out carry = 0 , outsum = 1
#    4  a = 0 , b = 1 ,out carry = 0 , outsum = 1
#    6  a = 1 , b = 1 ,out carry = 1 , outsum = 0
#    8  a = 0 , b = 0 ,out carry = 0 , outsum = 0
#    10  a = 1 , b = 0 ,out carry = 0 , outsum = 1
#    12  a = 0 , b = 1 ,out carry = 0 , outsum = 1
#    14  a = 1 , b = 1 ,out carry = 1 , outsum = 0
#    16  a = 0 , b = 0 ,out carry = 0 , outsum = 0
#    18  a = 1 , b = 0 ,out carry = 0 , outsum = 1
#    20  a = 0 , b = 1 ,out carry = 0 , outsum = 1
#    22  a = 1 , b = 1 ,out carry = 1 , outsum = 0
```

Figure 3.25 Results of running the test bench of the half-adder module in Figure 4.24.

```
module fa(sum,cout,a,b,cin); input a,b,cin;
output sum,cout; wire s,c1,c2;
ha ha1(s,c1,a,b), ha2(sum,c2,s,cin); or(cout,c2,c1);
endmodule
//test-bench module tst_fa(); reg a,b,cin;
fa ff(sum,cout,a,b,cin); initial
begin
a =0;b=0;cin=0; end
always begin

#2 a=1;b=1;cin=0;#2 a=1;b=0;cin=1; #2 a=1;b=1;cin=1;#2 a=1;b=0;cin=0; #2
a=0;b=0;cin=0;#2 a=0;b=1;cin=0; #2 a=0;b=0;cin=1;#2 a=0;b=1;cin=1; #2
a=1;b=0;cin=0;#2 a=1;b=1;cin=0; #2 a=0;b=1;cin=0;#2 a=1;b=1;cin=1; end

initial $monitor($time ,” a = %b, b = %b, cin = %b, outsum = %b, outcar
= %b “, a,b,cin,sum,cout); initial #30 $stop ;
endmodule
```

Figure 3.27 Design module and a test bench for a full-adder.

```
//output

#0 a = 0, b = 0, cin = 0, outsum = 0, outcar = 0 #2 a = 1, b = 1, cin = 0,
outsum = 0, outcar = 1 #4 a = 1, b = 0, cin = 1, outsum = 0, outcar = 1 #6 a =
1, b = 1, cin = 1, outsum = 1, outcar = 1 #8 a = 1, b = 0, cin = 0, outsum = 1,
outcar = 0 #10 a = 0, b = 0, cin = 0, outsum = 0, outcar = 0 #12 a = 0, b = 1, cin
= 0, outsum = 1, outcar = 0 #14 a = 0, b = 0, cin = 1, outsum = 1, outcar = 0
#16 a = 0, b = 1, cin = 1, outsum = 0, outcar = 1 #18 a = 1, b = 0, cin = 0,
outsum = 1, outcar = 0 #20 a = 1, b = 1, cin = 0, outsum = 0, outcar = 1 #22 a
= 0, b = 1, cin = 0, outsum = 1, outcar = 0 #24 a = 1, b = 1, cin = 1, outsum = 1,
outcar = 1 #26 a = 1, b = 1, cin = 0, outsum = 0, outcar = 1 #28 a = 1, b = 0, cin
= 1, outsum = 0, outcar = 1
```

Figure 3.28 Results of running the test bench of the full-adder module in Figure 4.27.

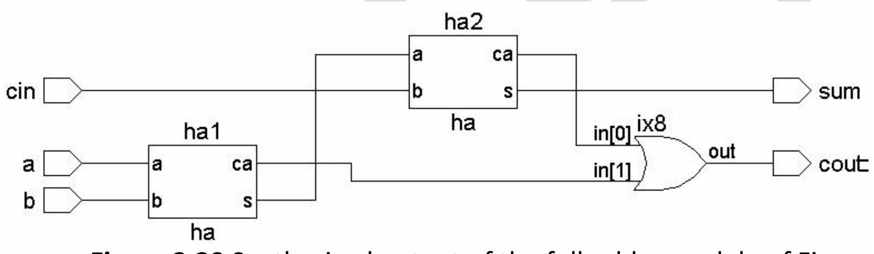


Figure 3.29 Synthesized output of the full-adder module of Figure 3.27.

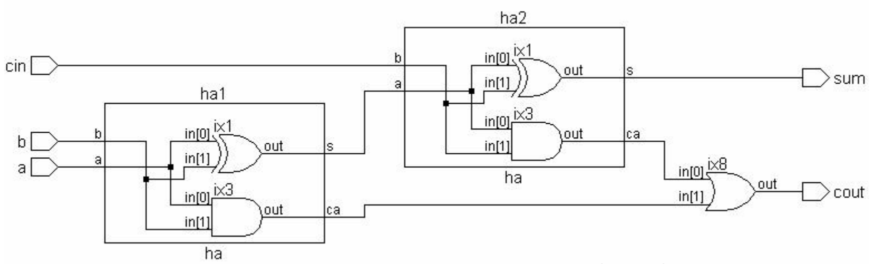


Figure 3.30 Synthesized circuit hierarchy of the full-adder module in Figure 3.27.

```
module mux4_1(y,i,s); input [3:0] i;
input [1:0] s; output y; wire [1:0] ss; wire [3:0]yy;
not (ss[0],s[0]),(ss[1],s[1]); and (yy[0],i[0],ss[0],ss[1]); and (yy[1],i[1],s[0],ss[1]);
and (yy[2],i[2],ss[0],s[1]); and (yy[3],i[3],s[0],s[1]);
or (y,yy[3],yy[2],yy[1],yy[0]); endmodule
//test-bench
module tst_mux4_1(); reg [3:0]i;
reg [1:0] s; mux4_1 mm(y,i,s); initial
begin

#2{i,s} = 6'b 0000_00; #2{i,s} = 6'b 0001_00; #2{i,s} = 6'b 0010_01; #2{i,s} = 6'b
0100_10; #2{i,s} = 6'b 1000_11; #2{i,s} = 6'b 0001_00; end

initial
$monitor($time," input s = %b,y = %b" ,s,y); endmodule
```

Figure 3.31 Design module and a test bench for a 4-to-1 mux module.

```
//output
//#          0 input s = xx ,y = x
//#          2 input s = 00 ,y = 0
//#          4 input s = 00 ,y = 1
//#          6 input s = 01 ,y = 1
//#          8 input s = 10 ,y = 1
//#         10 input s = 11 ,y = 1
//#         12 input s = 00 ,y = 1
```

Figure 3.32 Results of running the test bench of the 4-to- mux module in Figure 3.31.

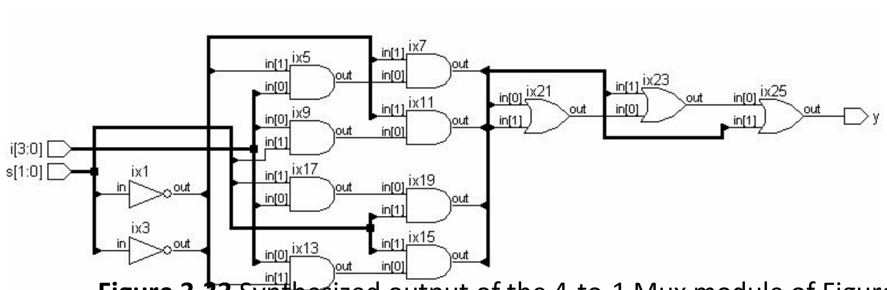


Figure 3.33 Synthesized output of the 4-to-1 Mux module of Figure 3.31.

```
        module trimux4_1(o,e,i,s); input e;
        input [1:0]s; input [3:0]i; output o; tri o;
        wire y,y1,y2,y3,y4; wire [1:0]ss;
        not(ss[0],s[0]),(ss[1],s[1]); and g1(y1,ss[0],ss[1],i[0]); and g2(y2,ss[1],s[0],i[1]); and
        g3(y3,ss[0],s[1],i[2]); and g4(y4,s[1],s[0],i[3]); or(y,y3,y2,y1,y2);
        bufif1 buf2(o,y,e); endmodule
        //TESTBENCH
        module tst_trimux4_1(); reg [1:0]s;
        reg [3:0]i; reg e; wire o;
        trimux4_1 tmx4_1(o,e,i,s); initial
        begin
        e =0;i =2'b00; end
always begin

#6 e=0;s=2'b00;i=4'b0001; #6 e=1;s=2'b01;i=4'b0010;

#6 e=1;s=2'b10;i=4'b0100; #6 e=1;s=2'b10;i=4'b1000; end

        initial $monitor($time ," input e = %b , s= %b , i = %b , output o = %b " ,e,s,i,o);
        initial #48 $stop; endmodule
```

Figure 3.34 Design module and a test bench for a 4-to-1 mux module with tri-state output.

```
output

#    0 input e = 0 , s= xx , i = 0000 , output o = z
#    6 input e = 0 , s= 00 , i = 0001 , output o = z #12 input e = 1 , s= 01 , i = 0010 , output o = 1 #18
     input e = 1 , s= 10 , i = 0100 , output o = 1 #24 input e = 1 , s= 10 , i = 1000 , output o = 0 #30
     input e = 0 , s= 00 , i = 0001 , output o = z #36 input e = 1 , s= 01 , i = 0010 , output o = 1 #42
     input e = 1 , s= 10 , i = 0100 , output o = 1
```

Figure 3.35 Results of running the test bench of the 4-to-1 mux module in Figure 3.34.

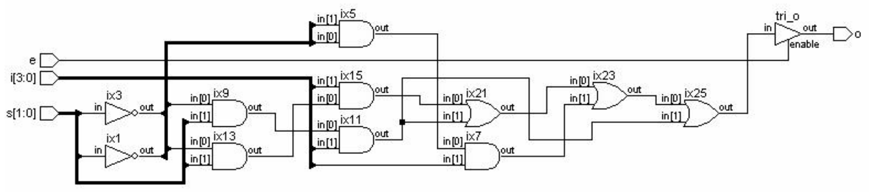


Figure 3.36 *Synthesized output of the 4-to-1 mux module of Figure 3.34*

```
module ttrimux2_1(out,e,i,s); input[1:0]i;
input e; input s; output out; wire o;
bufif0 g1(o,i[0],s); bufif1 g2(o,i[1],s);
bufif1 g3(out,o,e); endmodule
//testbench
module ttst_ttrimux2_1(); reg e;
reg [1:0]i; reg s;
ttrimux2_1 mm(out,e,i,s); initial
begin
e =0; i = 2'b 00;end always
begin

#4 e =0;{i,s} = 3'b 01_0; #4 e =1;{i,s} = 3'b 01_0; #4 e =1;{i,s} = 3'b 10_1; #4 e =1;{i,s} = 3'b 00_1;
#4 e =1;{i,s} = 3'b 10_1; #4 e =1;{i,s} = 3'b 01_0; #4 e =1;{i,s} = 3'b 00_0; #4 e =1;{i,s} = 3'b 11_0; end

initial $monitor($time ," enable e = %b ,
s= %b , input i = %b ,output out = %b ",e ,s,i,out); initial #48 $stop;
endmodule
```

Figure 3.37 Design module and a test bench for a 2-to-1 mux module formed with tri-statebuffers.

output

```
#    0 enable e = 0, s= x, input i = 00,output out = z
#    4 enable e = 0, s= 0, input i = 01,output out = z
#    8 enable e = 1, s= 0, input i = 01,output out = 1 #12 enable e = 1, s= 1, input i = 10,output out =
     1 #16 enable e = 1, s= 1, input i = 00,output out = 0 #20 enable e = 1, s= 1, input i = 10,output
     out = 1 #24 enable e = 1, s= 0, input i = 01,output out = 1 #28 enable e = 1, s= 0, input i =
     00,output out = 0 #32 enable e = 1, s= 0, input i = 11,output out = 1 #36 enable e = 0, s= 0, input
     i = 01,output out = z #40 enable e = 1, s= 0, input i = 01,output out = 1 #44 enable e = 1, s= 1,
     input i = 10,output out = 1
```

Figure 3.38 Results of running the test bench of the 2-to-1 mux module in Figure 3.37.

```
module ttrimux4_1(out,e,i,s); input[3:0]i;

input e; input[1:0]s; output out; tri o;
```

```
tri [1:0]o1;
bufif0 g1(o1[0],i[0],s[0]); bufif1 g2(o1[0],i[1],s[0]); bufif0 g3(o1[1],i[2],s[0]); bufif1
g4(o1[1],i[3],s[0]); bufif0 g5(o,o1[0],s[1]); bufif1 g6(o,o1[1],s[1]); bufif1 g7(out,o,e);
endmodule
//testbench
module ttst_ttrimux4_1(); reg e;
reg [3:0]i; reg [1:0]s;
ttrimux4_1 mm(out,e,i,s); initial
begin
e = 0;
i = 4'b 0000;

end always
begin

#4 e =0;{i,s} = 6'b 0001_00; #4 e =1;{i,s} = 6'b 0001_00; #4 e =1;{i,s} = 6'b 0010_01; #4 e =1;{i,s}
= 6'b 0000_01; #4 e =1;{i,s} = 6'b 0100_10; #4 e =1;{i,s} = 6'b 0101_10; #4 e =1;{i,s} = 6'b 1000_11;
#4 e =1;{i,s} = 6'b 0000_11; end

initial $monitor($time ," enable e = %b , s= %b , input i = %b ,output out = %b ",e
,s,i,out);
initial #48 $stop; endmodule
```

Figure 3.40 Design module and a test bench for a 4-to-1 mux module formed with tri-statebuffers.

output

```
#   0 enable e =0,s=xx, input i =0000, output out = z
#   4 enable e =0,s=00, input i =0001, output out = z
#   8 enable e =1, s=00,input i =0001 ,output out = 1 #12 enable e =1, s=01,input i =0010 ,output
    out = 1 #16 enable e =1, s=01,input i =0000 ,output out = 0 #20 enable e =1, s=10,input i =0100
    ,output out = 0 #24 enable e =1, s=10,input i =0101 ,output out = 1 #28 enable e =1, s=11,input
    i =1000 ,output out = 1 #32 enable e =1, s=11,input i =0000 ,output out = 0 #36 enable e =0,
    s=00,input i =0001 ,output out = z #40 enable e =1, s=00,input i =0001 ,output out = 1 #44
    enable e =1, s=01,input i =0010 ,output out = 1
```

Figure 3.41 Results of running the test bench of the 4-to-1 mux module in Figure 3.40.

3.10 DESIGN OF FLIP-FLOPS WITH GATE PRIMITIVES

The basic RS latch can be designed using gate primitives. Two instantiations of NAND or NOR gates suffice here. More involved flip-flops, registers, *etc*., can be built around these. Some of the level triggered versions of such flip-flops are taken up for design. Subsequently, the edge-triggered flip-flop of the 7474 type is developed in a skeletal form. More generalized versions are left as exercises.

Example 3.10.1 A Simple Latch

Figure 5.1 shows the design description of a simple latch formed with two NAND gates. A test bench for the same is shown in Figure 5.2 along with the results of the simulation run for 20 time steps. The test-bench has a block within a **`begin-end`** construct which reassigns values to`rb`and`sb`at two successive time stepintervals. The whole sequence described within the block lasts for 10 ns. Defining the block within the **`always`** construct repeats the above assignment sequence cyclically until the simulation stops. The latch has been synthesized, and the synthesized circuit is shown in Figure.

```
module sbrbff(sb,rb,q,qb); input sb,rb;
output q,qb; nand(q,sb,qb); nand(qb,rb,q); endmodule
```

Figure A module to instantiate the AND gate primitive and test it.

```
module tstsbrbff; //test-bench reg sb,rb;
wire q,qb;
sbrbff ff(sb,rb,q,qb); initial
begin
sb =1'b1; rb =1'b0;

end always begin
#2 sb =1'b1;rb =1'b1; #2 sb =1'b0;rb =1'b1; #2 sb =1'b1;rb =1'b1; #2 sb =1'b1;rb
=1'b0; #2 sb =1'b1;rb =1'b1;

end
initial $monitor($time, " sb = %b, rb = %b, q = %b, qb =
%b",sb,rb,q,qb);
initial #20 $stop; endmodule
```

Simulation results

```
11.5     0 sb = 1 , rb = 0 , q = 0 , qb  = 1
11.6     2 sb = 1 , rb = 1 , q = 0 , qb  = 1
11.7     4 sb = 0 , rb = 1 , q = 1 , qb  = 0
11.8     6 sb = 1 , rb = 1 , q = 1 , qb  = 0
11.9     8 sb = 1 , rb = 0 , q = 0 , qb  = 1
11.10    10 sb = 1 , rb = 1 , q = 0 , qb  = 1
11.11    14 sb = 0 , rb = 1 , q = 1 , qb  = 0
11.12    16 sb = 1 , rb = 1 , q = 1 , qb  = 0
11.13    18 sb = 1 , rb = 0 , q = 0 , qb  = 1
```

Figure A test bench for the flip-flop of Figure 5.1 and results of running the test bench.

Example 3.10.2 An RS Flip-Flop

The design module of an RS flip-flop along with a test bench for the same is shown in Figure 5.4. The module is a slight modification of the flip-flop of Figure 5.1. The simulation results are shown in Figure 5.5. The synthesized circuit is shown in Figure 5.6. One can easily relate the difference between this circuit and that of Figure 5.3 to the corresponding difference between the respective design modules.

```
module srff(s,r,q,qb); input s,r;
output q,qb; wire ss,rr; not(ss,s),(rr,r); nand(q,ss,qb); nand(qb,rr,q); endmodule
module tstsrff; //test-bench reg s,r;
wire q,qb;
srff ff(s,r,q,qb); initial
begin
s =1'b1; r =1'b0;

end always begin
#2 s =1'b0;r =1'b0; #2 s =1'b0;r =1'b1; #2 s =1'b0;r =1'b0; #2 s =1'b1;r =1'b0; #2 s =1'b0;r =1'b0;

end
initial $monitor($time, " s = %b, r = %b, q = %b, qb = %b ",s,r,q,qb);
initial #20 $stop; endmodule
```

Figure 5.4 Module of an RS flip-flop with NAND gates and a test bench for the same.

```
#    0 s = 1 , r = 0 , q = 1 , qb  = 0
#    2 s = 0 , r = 0 , q = 1 , qb  = 0
#    4 s = 0 , r = 1 , q = 0 , qb  = 1
#    6 s = 0 , r = 0 , q = 0 , qb  = 1
#    8 s = 1 , r = 0 , q = 1 , qb  = 0
#    10 s = 0 , r = 0 , q = 1 , qb  = 0
#    14 s = 0 , r = 1 , q = 0 , qb  = 1
#    16 s = 0 , r = 0 , q = 0 , qb  = 1
#    18 s = 1 , r = 0 , q = 1 , qb  = 0
```

Figure *Results of running the test bench for the flip-flop*

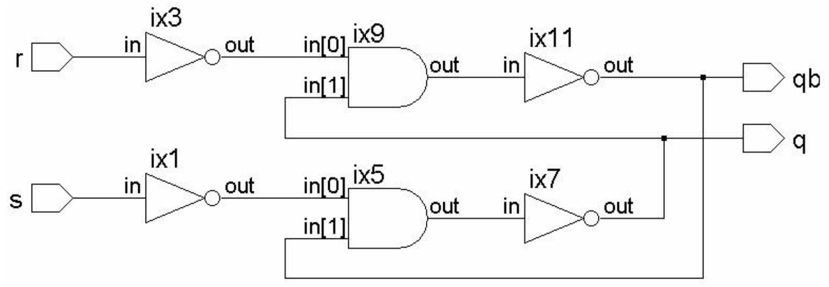


Figure *Synthesized circuit of the flip-flop module*

3.12 DELAYS

Verilog has the facility to account for different types of propagation delays of circuit elements. Any connection can cause a delay due to the distributed nature of its resistance and capacitance. Due to the manufacturing tolerances, these can vary over a range in any given circuit [Bignel, Sedra]. Similar delays are present in gates too. These manifest as propagation delays in the 0 to 1 transitions and 1 to 0 transitions from input to the output. Such propagation delays can differ for the two types of transitions. A variety of such delays can be accommodated in Verilog. Sometimes manufacturers adjust input and output impedances of circuit elements to specific levels and exploit them to reduce interface hardware. These too can be accommodated in Verilog design descriptions [Ciletti, Palnitkar].

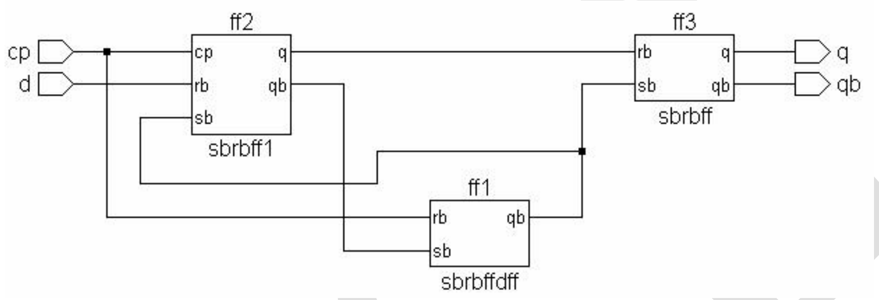


Figure *Synthesized circuit of the flip-flopdffgatnew1*

3.12.1 Net Delay

One of the simplest delays is that of a direct connection – a net. It can be part of the declaration statement

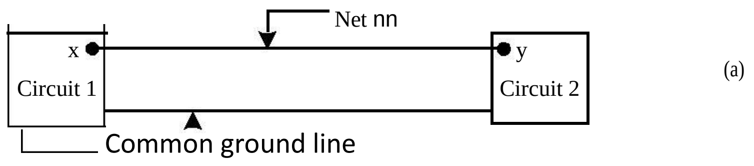
```
wire #2 nn; //nnis declared as a net with a propagation delay of 2 time steps
```

Here nn is declared as a net with an associated propagation delay of 2 time steps. The delay is the same for the positive as well as the negative transitions. The same is illustrated in Figure 5.21(a), which connects two circuit blocks through a net nn with a delay of 2 time steps associated with it. The module in Figure 5.22 is a simple realization of the same. A test bench for the module is also shown in the figure. The simulation results are shown in Figure 5.21(b), which bring out the effect of the net delay clearly.

Similar delays can be assigned to other types of nets as well. Whenever a variable or a signal is defined as a net and no delay is specified for it, the associated delay is taken as zero. This is true of instantiations of modules as well. The impedance connected as well as the type of loading can differ for the two transitions. The propagation delay too can differ accordingly. Two such delays can be specified as follows:

```
Wire    # (2, 1) nm;
```

Here nm is declared as a net with two distinct propagation delays; the positive (0 to 1) transition has a delay of 2 time steps associated with it. The negative



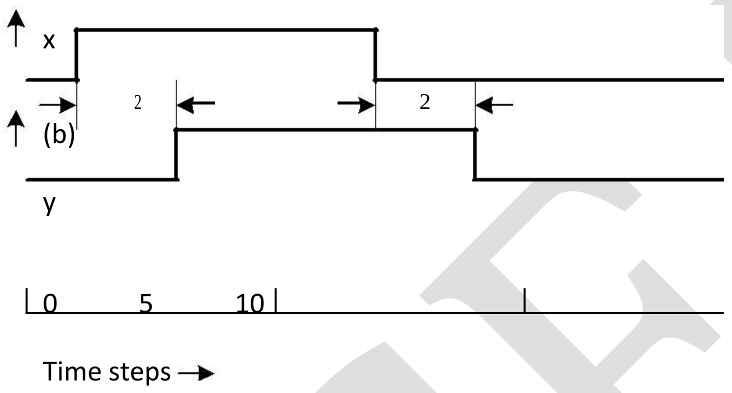


Figure A net connecting two circuit blocks and the delay through it: (a) Connectiondiagram (b) Typical signal waveforms at the input and output ends of the net.

```
module netdelay(x,y); input x;
output y; wire #2 nn;
not (nn,x); //circuit1 in Figure 5.21 buf y = x; //circuit2 in Figure 5.21 endmodule
module tst_netdelay ; //test-bench reg x;
wire y;
netdelay nd(x,y); initial
begin
x =1'b0; #6 x =~x;

end
initial #20 $stop; endmodule
```

Figure A module to illustrate net delay and a test bench for the same.

(1 to 0) transition has a delay of 1 time step. The delays are explained in Figure 5.23. The module of Figure 5.22 has been modified and shown in Figure 5.24; the propagation delays are different for rise and fall here.

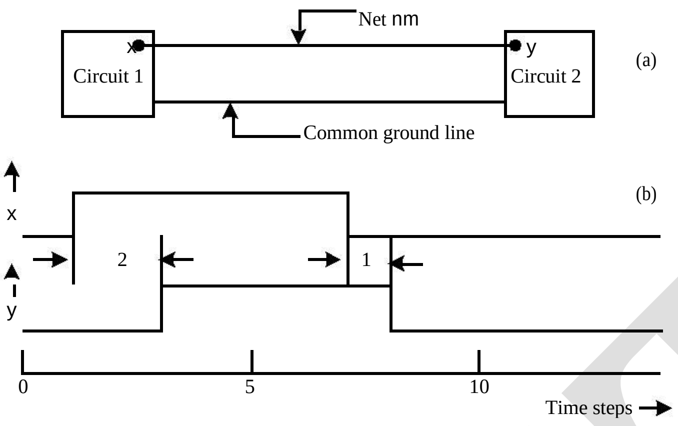


Figure A net connecting two circuit blocks and the delays through it: (a) Connectiondiagram (b) Typical signal waveforms at the input and output ends of the net.

```
module netdelay1(x,y); input x;
output y;
wire #(2,1) nn; not (nn,x); y=nn; endmodule
module tst_netdelay1; //test-bench reg x;
wire y;
netdelay1 nd(x,y); initial
begin
x =1'b0; #6 x =~x;

end
initial #20 $stop; endmodule
```

Figure A module to demonstrate different delays for rise and fall times on a net.

3.12.2 Gate Delay

Gates too can have delays associated with them. These can be specified as part of the instantiation itself.

```
and #3 g ( a, b, c);
```

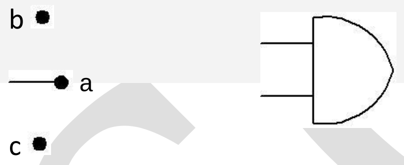
The above represents an AND gate description with a uniform delay of 3 ns for all transitions from input to output. A more detailed description can be as follows:

```
and #(2, 1) (a, b, c);
```

With the above statement the positive (0 to 1) transition at the output has a delay of 2 time steps while the negative (1 to 0) transition has a delay of 1 time step. Figure shows a module to illustrate the delays associated with gate primitives. A test bench for the same is also shown in the figure. The results of running the test bench are shown in Figure 5.27. The AND gate instantiation in Figure has different delays for the output transitions; respective waveforms are shown in Figure

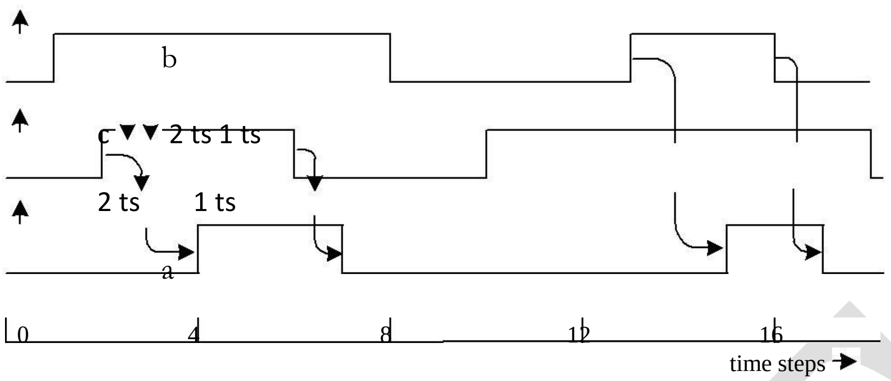
```
module gade(a,a1,b,c,b1,c1); input b,c,b1,c1;
output a,a1;
or #3gg1(a1,c1,b1); and #(2,1)gg2(a,c,b); endmodule
module tst_gade();//test-bench reg b,c,b1,c1;
wire c,c1;
gade ggde(a,a1,b,c,b1,c1); initial
begin
b =1'b0;c =1'b0;b1 =1'b0;c1=1'b0; end
always begin
#5 b =1'b0;c =1'b0;b1 =1'b1;c1=1'b1; #5 b =1'b1;c =1'b1;b1 =1'b0;c1=1'b0; #5 b =1'b1;c
=1'b0;b1 =1'b1;c1=1'b0; #5 b =1'b0;c =1'b1;b1 =1'b0;c1=1'b1; #5 b =1'b1;c =1'b1;b1 =1'b1;c1=1'b1;
#5 b =1'b1;c =1'b1;b1 =1'b1;c1=1'b1;

end
initial $monitor($time , " b= %b , c = %b , b1 = %b ,c1 = %b , a = %b ,a1 = %b"
,b,c,b1,c1,a,a1);
initial #30 $stop; endmodule
```

Figure 5.25 Module to demonstrate the delays with gates.



(a)

Figure AND gate instantiation with different delays for the positive and negativetransitions and associated waveforms: (a) Gate instantiated.



(b)

Figure (cont'd**) (b)** associated waveforms (time step has been abbreviated to "ts" inthe diagram).

In a more detailed design description, delays can be associated with nets as well as gates. Consider the design description shown in Figure 5.28(a). It has a total of 8 different time delay values specified. All these are hypothetical and different from each other. It is done intentionally to bring out the effect of each of them on the concerned gates and signals. The circuit for this design description is shown in Figure 5.28(b). Typical waveforms of input signals as well as other signals are shown in Figure 5.29, to illustrate the different delays in the design description. Figures 5.29(a) and 5.29(b) illustrate how changes in one of the inputs

- b1 – affect the other signals; the signals and gates affected are shown

```
#    0 b= 0, c = 0 , b1 = 0 ,c1 = 0 , a = x ,a1 = x
#    1 b= 0, c = 0 , b1 = 0 ,c1 = 0 , a = x ,a1 = 0
#    3 b= 0, c = 0 , b1 = 0 ,c1 = 0 , a = 0 ,a1 = 0
#    5 b= 0, c = 0 , b1 = 1 ,b1 = 1 , a = 0 ,a1 = 0
#    7 b= 0, c = 0 , b1 = 1 ,c1 = 1 , a = 0 ,a1 = 1
#    10 b= 1, c = 1 , b1 = 0 ,c1 = 0 , a = 0 ,a1 = 1
#    11 b= 1, c = 1 , b1 = 0 ,c1 = 0 , a = 0 ,a1 = 0
#    13 b= 1, c = 1 , b1 = 0 ,c1 = 0 , a = 1 ,a1 = 0
#    15 b= 1, c = 0 , b1 = 1 ,c1 = 0 , a = 1 ,a1 = 0
#    17 b= 1, c = 0 , b1 = 1 ,c1 = 0 , a = 1 ,c1 = 1
#    18 b= 1, c = 0 , b1 = 1 ,c1 = 0 , a = 0 ,c1 = 1
#    20 b= 0, c = 1 , b1 = 0 ,c1 = 1 , a = 0 ,a1 = 1
#    25 b= 1, c = 1 , b1 = 1 ,c1 = 1 , a = 0 ,a1 = 1
#    28 b= 1, c = 1 , b1 = 1 ,c1 = 1 , a = 1 ,a1 = 1
```

Figure Results of running the test bench of above module in Figure 5.25.

highlighted in Figure 5.29(a). Throughout this period, input c1 is taken as at 1 state while inputs b2 and c2 remain at 0 state. The propagation delays of signals at point P and Q and that for the signal a are shown in Figure 5.29(b). These conform to the delays specified in the design segment of Figure 5.28(a). Subsequently, input c1 goes down to 0 state and input b1 remains at 0 state itself. Only signal b2 changes. The affected signals and gates are shown highlighted in Figure 5.29(c). The waveforms of signals affected and the associated propagation designs are shown in Figure 5.29(d). These too conform to the program segment of Figure 5.28(a).

```
        module gates(b1,b2,c1,c2,a); input b1,b2,c1,c2;
        wire #(2,1)a1,a2; output a;
        and #(3,4)g1(a1,b1,c1); and #(5,6)g2(a2,b2,c2); or #(8,7)g3(a,a1,a2);
        endmodule
        module tst_gates;//test-bench reg b1,b2,c1,c2;
        gates gg(b1,b2,c1,c2,a); initial
        begin
    b1=1'b0;c1=1'b0;b2=1'b0;c2=1'b0;

        end
        initial #100 $stop;
        always begin
#2b1=1'b0;c1=1'b0;b2=1'b1;c2=1'b1;
#2b1=1'b1;c1=1'b1;b2=1'b0;c2=1'b0;
#2b1=1'b0;c1=1'b1;b2=1'b0;c2=1'b0;
#2b1=1'b0;c1=1'b0;b2=1'b1;c2=1'b0;
#2b1=1'b1;c1=1'b0;b2=1'b1;c2=1'b1;
#2b1=1'b1;c1=1'b1;b2=1'b0;c2=1'b0;

#2b1=1'b1;c1=1'b1;b2=1'b1;c2=1'b0;
#2b1=1'b0;c1=1'b0;b2=1'b1;c2=1'b1;

        end
        initial $monitor($time," b1= %b , c1 = %b ,b2 = %b , c2 = %b , a = %b
        ",b1,c1,b2,c2,a);
        endmodule
```

Figure A design having eight different time delay values.

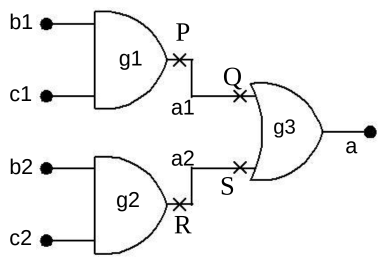


Figure (b) The circuit for the module considered in Figure 5.28(a).

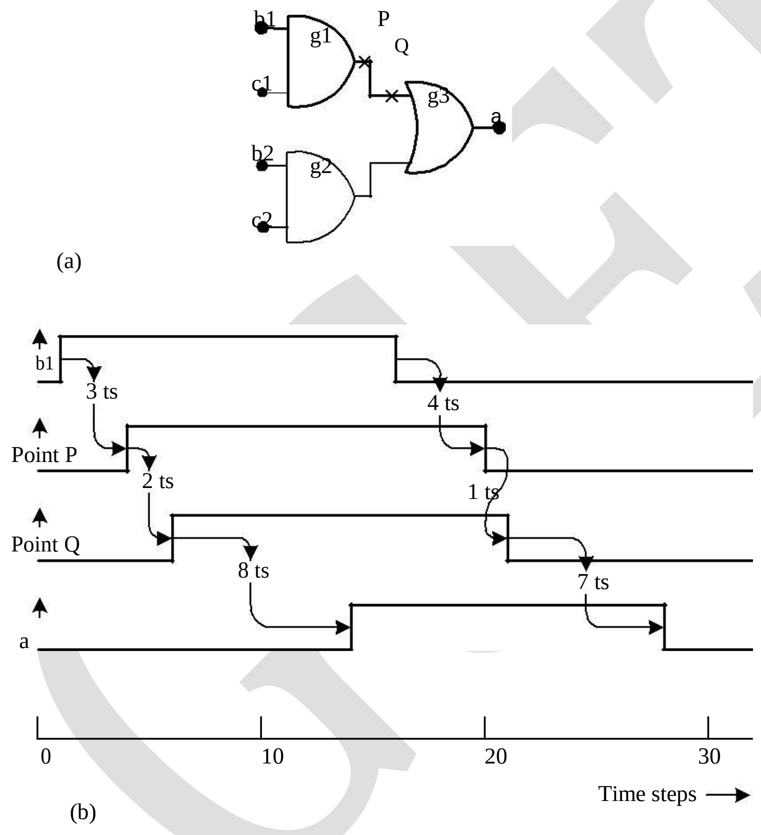
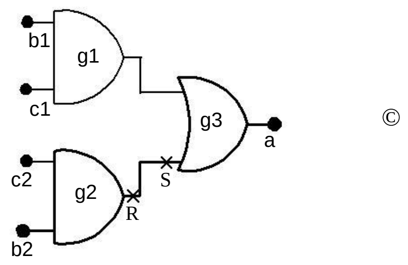


Figure Illustration of signal delays in the design description segment in Figure 5.28:

(a) The circuit portion active during changes to signal b1. (b) Signal waveforms following changes to signal b1 (time step has been abbreviated as ts in the diagram).



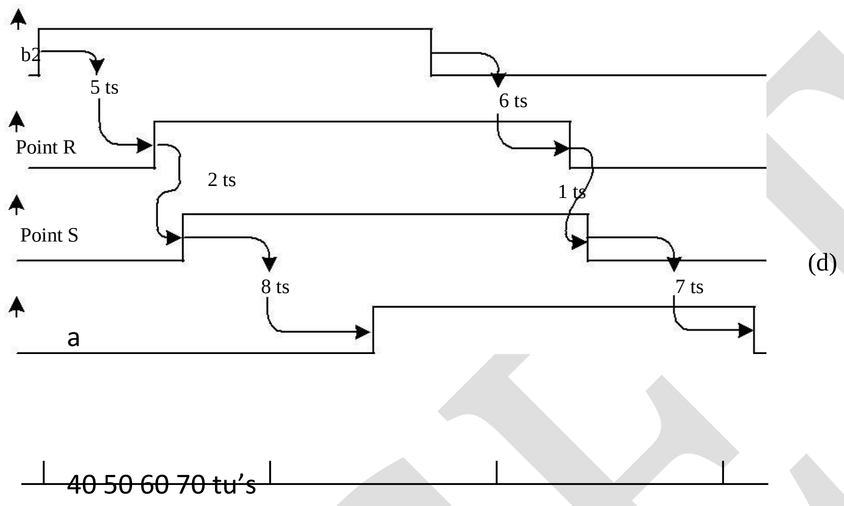


Figure (cont'd) © The circuit portion active during changes to signal b2. (d) Signalwaveforms following changes to signal b2 (time step has been abbreviated as ts in tbe diagrams).

3.12.3 Delays with Tri-state Gates

For tri-state gates the delays associated with the control signals can be different from those of the input as well as the output. The instantiation inclusive of this is shown in Figure 5.30 for a tri-state buffer of the **`bufif1`** type. Three time delay values are specified:

1. The first number represents the delay associated with the positive (0 to 1) transition of the output.
2. The second number represents the delay associated with the negative (1 to 0) transition of the output.
3. The third number represents the delay for the output to go to the hi-Z state as the control signal changes from 1 to 0 (*i.e.*, ON to OFF command).

```
bufif1 @ (1, 2, 3) b1(ao, ai, c);
```

Delay for the 0 to 1 transition of ao

Delay for the 1 to 0 transition of ao

Delay for the output to go to the hi-z state as c changes from 1 to 0

Figure Delays associated with a typical tri-state gate.

Delays for the other tri-state buffers – namely **`bufif0`**, **`notif1`** and **`notif0`** – may be specified in a similar manner.

The turn-off time — 2 time steps here — represents the time for which the charge will be stored in the output line after the control line turns off. Values of delay time and storage time can be specified in this manner for all the types of tri-state type gates. The following are noteworthy here:

Delays and storage times can be specified on the gate primitives and the nets but not on regs.

All three time values are separately specified in the most versatile case.

If only two time-values are specified, these are interpreted by Verilog as the rise (0 to 1) and fall (1 to 0) time, respectively. The turn-off time (delay) is

taken as the smaller of these two.

If only one time value is specified, it is taken as the rise time, the fall time, and the turn-off time.

If no time value is specified, the rise and fall times at the output are taken as zero. The turn-off is taken as instantaneous.

Normally the delay time of any IC varies over a range for ICs from different production batches (as well as in any one batch). It is customary for manufacturers to specify delays and their range in the following manner:

Max delay: The maximum value of the delay in a batch; that is, the delay encountered in practice is guaranteed to be less than this in the worst case.

Min. delay: Minimum value of delay in a batch; that is, the specified signal is guaranteed to be available only after a minimum of time specified. *Typ. delay*: Typical or representative value of the delay.

Each of the delays in a gate primitive or for a net can be specified in terms of these three values. For example

```
and #(2:3:4) g1(a0, a1, a2);
```

can instantiate an AND gate with the following time delay specifications: The 0 to 1 rise time and the 1 to 0 fall time are equal.

The minimum value of either is 2 time steps. Typical value is 3 time steps and the maximum value is 4 time steps.

Note that the colon that separates the numbers signifies that the timings specified are the minimum, typical, and maximum values. At the time of simulation, one can specify the simulation to be carried out with any of these three delay values. If the same is not specified, the simulation is carried out with the typical delay value.

The group of minimum, typical, and maximum delay values for the propagation delays can be specified separately for any gate primitive. Thus an AND gate primitive can be specified as

```
and #(1:2:3, 2:4:6) g2(b0, b1, b2);
```

Here for the 0 to 1 transition of the output (rise time) the gate has a minimum delay value of 1 ns, a typical value of 2 ns, and a maximum value of 3 ns. Similarly, for the 1 to 0 transition (fall time) the gate has a minimum delay value of 2 ns, a typical delay value of 4 ns, and a maximum delay value of 6 ns. Such delay specifications can be associated with nets as well as tri-state type gates also.

Examples

```
wire #(1:2:3) a;/* The net a has a propagation delay whose minimum,
typicaland maximum values are 1 ns, 2 ns, and 3 ns, respectively*/
bufif1 #(1:2:3, 2:4:6, 3:6:9) g3 (a0, b0, c0);
```

/* The different delay values for the buffer are as follows:
The output rise time (0 to 1 transition) has a minimum value of 1 ns, a typical value of 2 ns and a maximum value of 3 ns.

The output fall time (1 to 0 transition) has a minimum value of 2 ns, a typical value of 4 ns and a maximum value of 6 ns.

The output turn-off time (1 to 0) has a minimum value of 3 ns, a typical value of 6 ns, and a maximum value of 9 ns. */

A typical design can have a number of circuit blocks like gates, flip-flops, *etc*., with associated interconnections. The individual nets and gates may havetheir own separate delays. The following general observations are in order regarding the overall delays through the circuit:

A normal design can have many gates and nets in its signal paths. The delay through any path for a signal depends on the path and the type of transitions at each stage.

3.13 STRENGTHS AND CONTENTION RESOLUTION

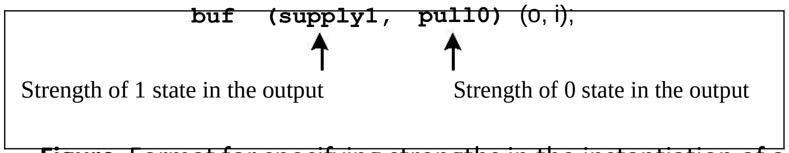
In practical situations, outputs of logic gates and signals on nets in a circuit have associated source impedances. When the outputs of two gates are joined together, the signal level is decided by the relative magnitudes of the source impedances. Sometimes a disparity between the impedances is intentionally introduced to minimize circuit hardware. Effects of such differences in the impedances are indirectly introduced in design descriptions by assigning "strengths" to specific signals (see also Section 3.9). Signal strength declarations are of two types – those associated with outputs of gate primitives and those with nets.

3.13.1 Strengths of Gate Primitives

Gate output strengths can be specified separately. Table 5.1 gives the names associated with strengths, respective abbreviations, and their order by weight. These hold good for logic 1 state as well as the 0 state.

Table Strength levels associated with outputs of gate primitives

Name	**supply**	**strong**	**pull**	**weak**	High impedance
Abbreviations	**su1** **su0**	**st1** **st0**	**pu1** **pu0**	**we1** **we0**	HiZ1 HiZ0
Strength	Strongest			Weakest	



```
buf (supply1, pull0) (o, i);
```

Strength of 1 state in the output
Strength of 0 state in the output

Figure Format for specifying strengths in the instantiation of a gate primitive.

The strengths associated with the output of a gate primitive can be specified separately for the two logic levels. The format for the same is shown in Figure 5.31 for a specific case; the format remains the same for all types of gate primitives.

3.14 NET TYPES

`wire` is possibly the simplest type of net declaration. **`trireg`** considered in thelast section is another. A variety of other net types are possible. Most of them are provided for specific types of contention resolution.

3.14.1 `wand` and `wor` Types of Nets

Strengths on nets can be decided in ways other than a direct declaration also. These offer additional flexibility to the circuit designer. Consider the example of Figure 5.33 for which the input–output values are shown in Table 5.3. For the signal input combination i1 = 0 and i2 = 1, signal o is indeterminate. However, it may be made specific in two alternate ways: '**wand**' and **wor** are two types of net declarations for such contention resolution. **wand** is a wire declaration, which resolves to AND logic in case of contention. **wor** is a wire declaration, which resolves to OR logic in case of a contention. Use of **wand** and **wor** nets is illustrated here through two simple examples crafted for the purpose.

Example 5.8 Illustration of `wand` type net

Figure 5.35 shows a design module where the outputs of two buffers drive the same net; the net has been declared to be a **wand** type, and any contention with the possibility of indeterminate output is resolved according to AND logic. A test bench and simulation results are also shown in the figure. The input and output logic values and the nature of contention resolutions wherever it occurs are listed out in Table 5.7 also. Contention can be seen to be resolved in two possible ways:

1. When `i1` = 1 and `i2` = 0, the stronger signal `i1` at the 1 level prevails and `o` = 1. The contention is resolved according to the strengths.
2. When `i1` = 0 and `i2` = 1, both signals being equally strong, the value of `o` is decided according to AND logic.

The synthesized version of the circuit is shown in Figure 5.36; the circuit translates into an AND gate which is erroneous (this is not consistent with the desired input–output relationship shown in Table 5.7).

```
module wand1(i1,i2,o); input i1,i2;
output o; wand o;
buf(strong1,pull0)g1(o,i1);
buf(pull1,pull0)g2(o,i2); endmodule
module tst_wand1; //testbench reg i1,i2;
wand1 ww(i1,i2,o); initial
begin
i1=0;i2=0;//o =0; no contention #2i1=0;i2=1;//o =0; contention resolved //according to wand
declaration

#2i1 =1;i2 =0;//out=1; contention resolved by //stronger signal

#2i1 =1;i2=1;//out =1; no contention.

end
initial $monitor($time,"i1=%b,i2=%b,o=%b",i1,i2,o); endmodule
```

```
output
    #                0i1=0,i2=0,o=0
    #                2i1=0,i2=1,o=0
    #                4i1=1,i2=0,o=1
    #                6i1=1,i2=1,o=1
```

Figure A design module to illustrate use of the wand-type net; a test bench and theresults of simulation are also shown.

Table Output values for different inputs of the design in Figure 5.35

Logic value of i1	Logic value of i2	Logic value of o	Remarks
0	0	0	No contention
0	1	0	Contention resolved according to **wand** declaration
1	0	1	Contention resolved by the stronger signal
1	1	1	No contention

Example 5.9 Illustration of `wor`-type net

Consider the design segment in Figure 5.35 with o being declared as a **`wor`** type of net instead of a **`wand`** type. The corresponding design module is shown in Figure 5.37. A test bench and simulation results are also shown in the figure. The outputs for all possible combinations of inputs are given in Table 5.8. Contention can be seen to be resolved in two possible ways:

1. When `i1` = 1 and `i2` = 0, the stronger signal `i1` at the 1 level prevails and `o` = 1. The contention is resolved according to the strengths.
2. When `i1` = 0 and `i2` = 1, both signals being equally strong, the value of `o` is decided according to OR logic.

The synthesized version of the circuit is shown in Figure 5.38; the circuit translates into an OR gate; this is consistent with the desired input–output relationship shown in Table 5.8.

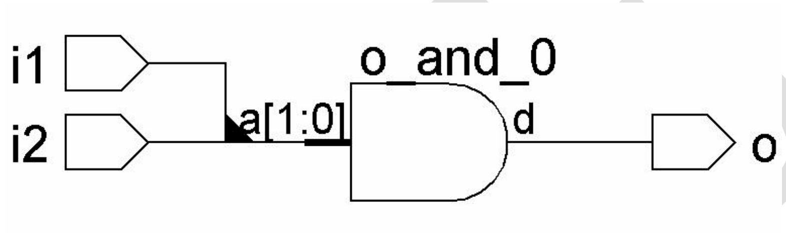


Figure 5.36 Synthesized version of the module with the wand-type net in Figure 5.35above.

```
module wor1(i1,i2,o); input i1,i2;
output o; wor o;
buf(strong1,pull0)g1(o,i1);
buf(pull1,pull0)g2(o,i2); endmodule
module tst_wor1;//testbench reg i1,i2;
wor1 ww(i1,i2,o); initial
begin
i1=0;i2=0;//out =0 no contention

#2 i1=0;i2=1;//out =1 contention resolved according //to wor
declaration
#2 i1 =1;i2 =0;//out=1 contention resolved by //stronger signal
#2 i1 =1;i2=1;//out =1 no contention. end
initial $monitor($time,"i1=%b,i2=%b,o=%b",i1,i2,o); endmodule
```

```
Output
        #               0 i1=0, i2=0, o=0
        #               2 i1=0, i2=1, o=1
        #               4 i1=1, i2=0, o=1
        #               6 i1=1, i2=1, o=1
```

Figure A design module to illustrate use of the**wor-**type net; a test bench and theresults of simulation are also shown.

5

MODELING AT DATA FLOW LEVEL

5.1 INTRODUCTION

Gate level design description makes use of the gate primitives available in Verilog. These are repeatedly and judiciously instantiated to achieve the full design description. Digital designers familiar with the basic logic gates and SSI / MSI circuits can describe the desired target circuit in terms of them on paper and proceed with the design description based on them. This was the approach followed in the last two chapters; it is practical for comparatively smaller designs – say those involving tens of gates. One can define modules in terms of primitives involving tens of gates and instantiate them in macro-modules. This increases the complexity of designs that can be handled by one order. Beyond that the gate level design description becomes too complicated to be practical.

Data flow level description of a digital circuit is at a higher level. It makes the circuit description more compact as compared to design through gate primitives. We have a number of operands and operations representing the simulations directly or indirectly. The operations are carried out on the operand(s) in singles or in combinations [IEEE]. The results are assigned to nets. The operand-operation-assignments representing data flow are carried out repeatedly to complete the design description [Thomas & Morby]. Further, these can be combined judiciously with the gate instantiations wherever necessary. With such combinations, design description of a comprehensive nature can be accommodated.

5.2 CONTINUOUS ASSIGNMENT STRUCTURES

A simple two input AND gate in data flow format has the form

assign c = a && b;

Here

"**assign**" is the keyword carrying out the assignment operation. This type of assignment is called a continuous assignment.

a and b are operands – typically single-bit logic variables.

"&&" is a logic operator. It does the bit-wise AND operation on the two operands a and b.

"=" is an assignment activity carried out.

c is a net representing the signal which is the result of the assignment.

In general, an operand can be of any one of the following types:

A constant number [including real].

Net of a scalar or vector type including part of a vector.

Register variable of a scalar or vector type including part of a vector. Memory element.

A call to a function that returns any of the above. The function itself can be a user-defined or of a system type [see Chapter 9].

There are other types of operators as well [see Section 6.5]. All types of combinational circuits can be modeled using continuous assignments. One need not necessarily resort to instantiation of gate primitives.

An AND gate module which uses the above assignment is shown in Figure 6.1. The test bench for the same is shown in Figure 6.2, and the waveforms of nets a, b, and c obtained with the simulation are shown in Figure 6.3. [The simulation software used has the facility to capture the waveforms of selected signals in the "run" phase; this has been invoked to get the waveforms in Figure 6.3. No separate **$monitor** command is included in the test bench of Figure 6.2. The same approach has been adopted with many of the test benches elsewhere in the book.]

Multiple assignments can be carried out through a direct extension of the structure adopted in the above case. Consider the AOI gate in Figure 6.4. A few patterns of the assignments for the circuit are given in Figure 6.5 to Figure 6.7.

```
module andgdf(c,a,b); output c;
input a,b; wire c;
assign c = a&&b; endmodule
```

Figure 5.1 A module with an AND gate at the data flow level.

```
module tst_andgdf; //TESTBENCH reg a,b;
wire c; initial begin
a = 1'b0; b = 1'b0; #4   a = 1'b1;
#4 b = 1'b1; #4 a = 1'b0; #4 b = 1'b0; #4 a = 1'b1; end
andgdf g1(c,a,b); initial #20 $stop; endmodule
```

Figure 5.2 A test bench for the module in Figure 6.1.

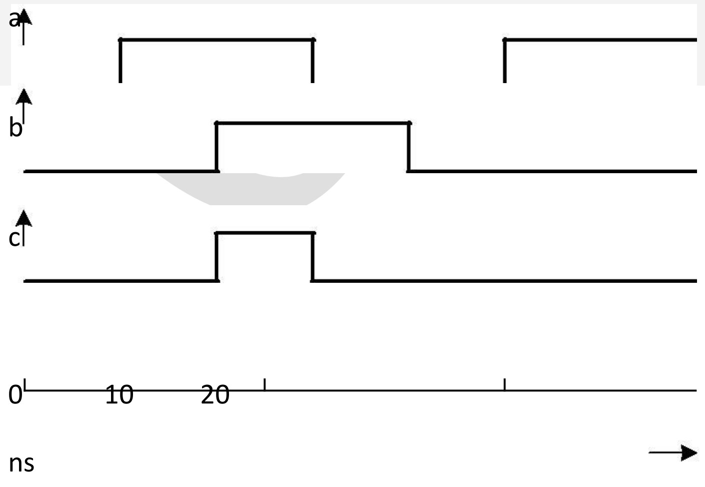


Figure 5.3 Waveforms of nets a, b, and c obtained with the simulation of the module

inFigure 5.1 with the test bench in Figure 5.2.

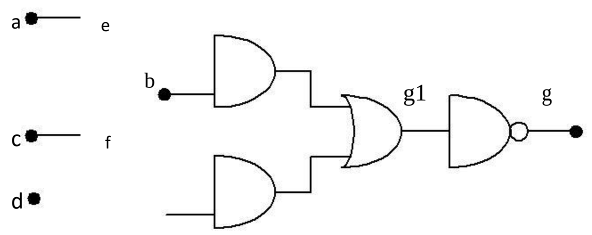


Figure 5.4 An A-O-I gate circuit.

assign e = a&&b, f = c&&d, g1 = e|f, g = ~g1;
Figure 5.5 A data flow level assignment statement to realize the A-O-I gate in Figure 6.4.

assign e = a & b, f = c & d; **assign** g1 = e|f, g = ~g1;
Figure 5.6 Another set of data flow level assignment statements to realize the A-O-I gate inFigure 5.4.

```
assign e  = a & b;
assign f  = c & d;
assign g1 = e ! f;
assign g  = ~g1;
```

Figure 5.7 Yet another set of data flow level assignment statements to realize the A-O-Igate in Figure 5.4.

Observations:

The semicolon terminates an assignment statement. Commas separate different assignments in an assignment statement.

“|” is the bit-wise OR operator and “~” the bit-wise negation operator in Verilog.

All the quantities in the left-hand side of a continuous assignment have to be of net type. Thus e, f, g, and g1 have to be declared as nets.

All the operations in an assignment are evaluated whenever any of the operands in the assignment changes value. Further, all the assignments are carried out concurrently. Hence the order of the assignments or the statements

is immaterial.

The right-hand sides of assignment statements can be nets, regs, or function calls. Here a, b, c, and d can be nets or regs. All other variables have to be nets.

The module for the A-O-I gate of Figure 5.4 is given in Figure 5.8 – it is formed around the assignment statement of Figure5.5. The same can be tested through a test bench.

5.2.1 Combining Assignment and Net Declarations

The assignment statement can be combined with the net declaration itself making the assignment implicit in the net declaration itself. Thus the two statements
wire c;
assign c = a & b;
can be combined as
wire c = a & b;
The above simplification cannot be carried over to multiple declarations. With this proviso, the module of Figure 5.8 can be modified as shown in Figure 5.9. In the modules of Figures 5.8 and 5.9, a, b, c, and d are declared as **input** and g as **output**. As was explained in Section 4.2, these would be taken as nets if thereare no separate declarations concerning their types. However, the intermediate quantities – e, f, and g1– should be declared as **wire**. Synthesized version of the A-O-I circuit is shown in Figure 5.10.

```
module aoi2(g,a,b,c,d); output g;
input a,b,c,d; wire e,f,g1,g;
assign e = a && b,f = c && d, g1 = e||f, g=~g1; endmodule
```

Figure 5.8 A compact description of the AOI module at the data flow level.

```
module aoi3(g,a,b,c,d); output g;
input a,b,c,d; wire g;
wire e = a && b; wire f = c && d; wire g1 = e||f; assign g = ~g1; endmodule
```

Figure 5.9 Alternate design module to realize the A-O-I gate in Figure 5.4.

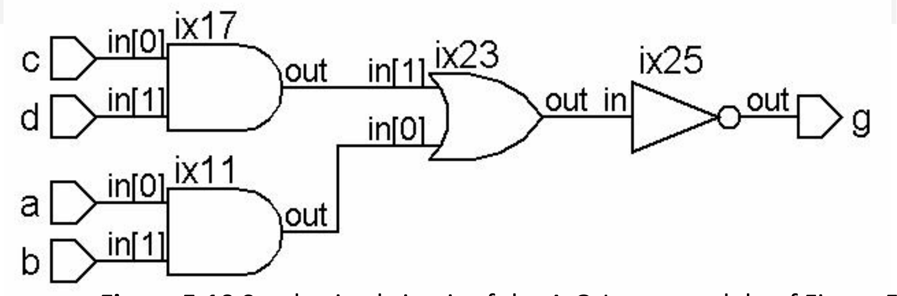


Figure 5.10 Synthesized circuit of the A-O-I gate module of Figure 5.9.

5.2.2 Continuous Assignments and Strengths

A net to which a continuous assignment is being made can be assigned strengths for its logic levels. The procedure is akin to the strength allocation to the outputs of primitives. The AOI gate of Figure 5.9 is modified with strength allocations to the output and is shown in Figure 5.11. The assignment to g can be combined with the wire declaration into a single statement as

wire (pull1, strong0)g = ~g1;

As mentioned earlier, one can have only one assignment in the statement here. In a bigger design, g in Figure 5.11 can be assigned to other expressions or primitives also. Any resulting contention in the output values will be resolved on the lines discussed in Chapter 4.

```
module aoi4 (g, a, b, c, d); output g;
input a, b, c, d; wire g;
wire e = a &&b; wire f = c &&d; wire g1 = e || f;
assign (pull1, strong0)g = ~g1; endmodule
```

Figure 5.11 The module of Figure 5.9 modified with strength allocation to the output.

5.3DELAYS AND CONTINUOUS ASSIGNMENTS

Delays can be incorporated at the data flow level in different ways [Ciletti]. Consider the combination of statements in Figure 5.12. The assignment takes effect with a time delay of 2 time steps. If a or b changes in value, the program waits for 2 time steps, computes the value of c based on the values of a and b at the time of computation, and assigns it to c. In the interim period, a or b may change further, but c changes and takes the new value only 2 time steps after the change in a or b initiates it. Typical waveforms for a, b, and c are shown in Figure 5.13. Note that the changes in a and b of duration less than 2 time steps are ignored *vis-à-vis* assignment to the net c. The following may be noted with respect to the waveforms: a changes at 0 ns, 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 0 ns, 2 ns, 5 ns, 8 ns and 13 ns. All these trigger changes to c.

In every case change to c comes into effect with a time delay of 2 time steps – that is, at the 2nd, 4th, 7th, 8th, 10th, 11th, 14th and 15th ns, respectively.

Whenever c changes, its new value is decided by the values of a and b at that instant of time. In effect, c changes at 2nd, 4th and 7th ns only.

```
wire c, a, b;

assign #2  c = a & b;
```

Figure 5.12 Illustration of combining delays with assignments.

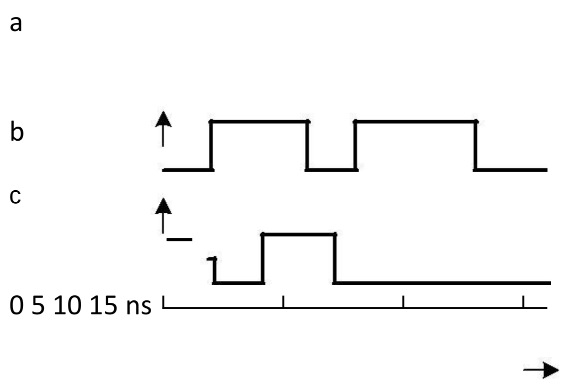


Figure 5.13 Waveforms of signals a, b, and c for the design segment of Figure 5.12.

9***
***-/-

41+
+6 l'\

The program segment in Figure 5.14 also gives the same output as shown in Figure 5.13. If the time delay is in the net and not in the assignment proper, its effect is not any different. Consider the program segment in Figure 5.15. Here the changes in the values of d are computed immediately following those in a and b. The assignment takes effect immediately. The delay in the net c causes a delay of 2 time steps in the assignment to c. Such a delay is not present for d. Typical waveforms for the program segment are shown in Figure 5.15. Note the following:

a changes at 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 2 ns, 5 ns, 8 ns and 13 ns. All these trigger changes to c and d also.

In every case, change to c comes into effect with a time delay of 2 time steps

- that is, in effect, c changes at 2nd, 4th and 7th ns only.

Whenever c changes, its new value is decided by the values of a and b at that instant of time.

In every case, changes to d come into effect immediately.

```
wire a, b;
wire   #2 c = a & b;
```

Figure 5.14 Alternate description for the program segment of Figure 5.10.

```
wire a, b, d;
wire #2      c;
```

assign c = a & b; **assign** d = a & b;

Figure 5.15 Illustration of combining delays with assignments.

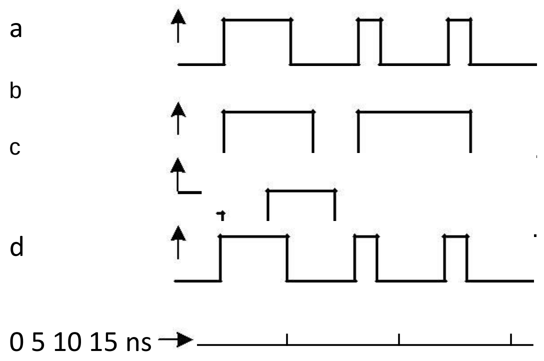


Figure 5.15 Waveforms of Signalsa,b,c,anddfor the design segment of Figure 5.15.

5.4ASSIGNMENT TO VECTORS

The continuous assignments are equally applicable to vectors. A single statement can describe operations involving vectors wherever possible. This is illustrated in the adder module in Figure 5.17, which adds two 8-bit numbers. Here it is assumed that the sum is also of 8 bits. However to account for the possibility of a carry bit being generated in the course of the addition process, it is desirable to increase the vector size of c by one bit.

5.4.1 Concatenation of Vectors

One can concatenate vectors, scalars, and part vectors to form other vectors. The concatenated vector is enclosed within braces. Commas separate the components – scalars, vectors, and part vectors. If a and b are 8- and 4-bit wide vectors, respectively and c is a scalar

{a, b, c}

stands for a concatenated vector of 13 bits width. The vector components are formed in the order shown – c is the least significant bit and a[7] the most significant bit and the other bits are in between in the order specified. The concatenation can be with selected segments of vectors also. For example,

{a(7:4), b(2:0)}

represents a 7-bit vector formed by combining the 4 most significant bits of vector a with the 3 least significant bits of vector b. The size of each operand within thebraces has to be specified fully to form the concatenated vector. Hence unsized constant numbers cannot be used as operands here.

Example 5.1 Eight-Bit Adder

Figure 5.18 shows the design description of an 8-bit adder, where the output vector is formed directly by concatenation. The adder takes a carry input and gives out a carry output. The adder module here can form the "seed" adder block in a multi-byte adder chain.

```
module add_8(a,b,c); input[7:0]a,b; output[7:0]c; assign c = a + b ; endmodule
```

Figure 5.17 An adder module at data flow level where the nets are vectors.

```
module add_8_c(c,cco,a,b,cci); input[7:0]a,b;
output[7:0]c; input cci; output cco;
assign {cco,c} = (a + b + cci); endmodule
```

Figure 5.18 A complete 8-bit adder module at data flow level.

When it is necessary to replicate vectors, scalars, *etc.,* to form other vectors, the same can be arrived at in a compact manner using the repetition multiplier again through concatenation. Thus,

{2{p}}
represents the concatenated vector
{p, p}
and
{2{p}, q}
represents the concatenated vector
{p, p, q}.
The two statements
assign GND=**supply0**;p={8{GND}};
together ground the 8 bits of the vector p.

Concatenation operation can be nested to form bigger vectors when component combinations are repeated. For example,

{a, 3 {2{b , c}, d}}
is equivalent to the vector
{a, b, c, b, c, d, b, c, b, c, d, b, c, b, c, d }

5.5 OPERATORS

A set of operators is available in Verilog. The operator symbols are similar to those in C language [Gottfried]. With these operators we can carry out specified operations on the operands and assign the results to a net or a vector set of nets as the case may be. A few such operands have already been used in the examples so far. We discuss here the different operators, their types, and the operations carried out by each. Subsequently the use of operators is illustrated through a set of examples.

5.5.1 Unary Operators

Unary operators do an operation on a single operand and assign the result to the specified net. The unary operators in Verilog are given in Table 5.1. All unary operators get precedence over binary and ternary operators. The operators "+" and "–" preceding an integer or a real number change its sign. These are also unary operators, though not separately listed in Table 5.1.

5.5.2 Binary Operators

Most operators available are of the binary type. A binary operator takes on two operands; the operator comes in between the two operands in the assignment. The binary operators are grouped into type categories and discussed separately. The following are to be noted:

The arithmetic operators treat both the operands as numbers and return the result as a number.

All net and **`reg`** operand values are treated as unsigned numbers. Real and integer operands may be signed quantities.

If either of the operand values has a zero value, the entire result has a zero value (?).

The result of any arithmetic operation — with the "+" or "–" or with any of the other arithmetic operators discussed later — will have an **`x`** value if any of the operand bits has an **`x`** or a **`z`** value.

5.5.2.1 Arithmetic Operators

The arithmetic operators of the binary type are given in Table 5.2. The modulus operand is similar to that in C language – It provides the remainder of the division

Table 5.1 Unary operators and their symbols

Operator type	Symbol	Remarks
Logical negation	!	Only for scalars
Bit-wise negation	~	For scalars and vectors
Reduction AND	&	For vectors – yields a single bit output
Reduction NAND	~&	
Reduction OR	|	
Reduction NOR	~|	
Reduction XOR	^	
Reduction XNOR	~^ or ^~	

Table 5.2 Arithmetic operators and their symbols

Operand type	Symbol	Remarks
Multiplication	*	

Division	/	The result is x if the denominator is zero
Modulus	%	
Addition	+	
Subtraction	–	

of two numbers. The module in Figure 5.17 is an example of the illustration of the use of the arithmetic binary operator "+" (for addition). Other arithmetic operators are also used in a similar manner.

Observations:

In integer division the fractional part of the result is truncated and ignored.

If any bit of an operand is **x** or **z** in an arithmetic operation, the result takes the **x** value.

If the first operand of a modulus operator is negative, the result is also a negative number.

Depending on the type of definition of a number, a modulus operation can lead to different results. Typical examples are given in Table 5.3.

5.5.2.2 Logical Operators

There are two logical operators involving two operands. The operands concerned can be variables or expressions involving variables. In both cases the result of the operation is a single bit of value 1 (true) or 0 (false). If a bit in one of the operands is x or z, the result of evaluation of the expression has an **x** value. The operator details are shown in Table 5.4. The modules in Figure 5.8 and Figure 5.9 are examples of the illustration of the use of logical binary operators.

5.5.2.3 Relational Operators

There are four relational operators; their details are shown in Table 5.5. A relational operator treats both the operands as binary numbers and compares them. The result is a 1 (true) bit or a 0 (false) bit. If a bit in either operand is **x** or **z**, the result has **x** (unknown) value. The operands can be variables or expressions involving variables. Operands of net or **reg** type are treated as unsigned numbers. Real and integers can be positive or negative (*i.e.*, signed) numbers.

Table 5.3 Typical modulus operations and their results

Expressions involving modulus operator	Result of the operation	Remarks
15 % 5	0	Results are obvious
14 % 5	4	
4'hf % 5	0	The numbers 4'hf and 4'he are in hex format with decimal values of 15 and 14, respectively. But the denominator 5 is in decimal form.
4'he % 5	4	
5'o15 % 5	3	5'o15 is an octal number with a decimal value of 13.
–4 % 3	–1	
4 % –3		Illegal form

Table 5.4 Binary logical operators and their symbols

Operator type	Symbol	Possible output value
AND	&&	Single-bit output
OR	||	

Table 5.5 Relational operators and their symbols

Operator type	Symbol	Possible output value
Greater than	>	Single-bit output
Less than	<	
Greater than or equal to	>=	
Less than or equal to	<=	

5.5.2.4 Equality Operators

The equality operator makes a bit-by-bit comparison of the two operands and produces a result bit. The result bit is a 1 (true) if the operand condition is satisfied; otherwise it is 0 (false). The operands can be variables or expressions involving variables. If the operands are of unequal length, the shorter one is zero filled to match the larger operand. The operators in this category are only of two types – those to test the equality and those to test inequality. The four operators in this category are given in Table 5.5.

5.5.2.5 Bit-wise Logical Operators

The operator does a specified bit-by-bit operation on the two operands and produces a set of result bits. The result is (bit-wise) as wide as the wider operand.

Table 5.5 Equality operators and their symbols

Operand symbol	Description of operand	Possible logical value of result
==	(The symbol comprises two consecutive equal signs.) If the two operands are equal bit by bit, the result is 1 (true); else the result is 0 (false). If either operand has a **x** or **z** bit, the result is **x**.	0, 1, or **x**
!=	(The symbol comprises of an exclamation mark followed by an equal sign.) A bit-by-bit comparison of the two operands is made. The result is a 1 if there is a mismatch for at least one bit position.	0, 1, or **x**
===	(The symbol comprises of three consecutive equal signs.) The operand bits can be 0, 1, **x**, or **z**. If the two operands match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never **x** here.	0 or 1
!==	(The symbol comprises an exclamation mark followed by 2 consecutive equal signs). The operand bits can be 0, 1, **x**, or **z**. If the two operands do not match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never	0 or 1

	x here.	

If the width of one of the operands is less than that of the other, it is bit-extended by filling zero bits and the widths are matched. Subsequently, the specified operation is carried out. If one of the operands has an **x** or **z** bit in it, the corresponding result bit is **x**. Either operand can be a single variable or an expression involving variables. Table 5.7 gives the four operators of this category.

5.5.2.6 Operator Truth Table

The truth tables for different types of bit-wise operators are given in Table 5.8. Note that an **z** input is treated as an **x** value (Compare these with their counterparts for respective gate primitives in Chapter 4.)

Table 5.7 Bit-wise logical operators and their symbols

Operator type	Symbol	Possible result
AND	&	0, 1, or **x**
OR	|	
XOR	^	
XNOR	~^ or ^~	

Table 5.8 Truth tables for bit-wise operators

AND

		Input 2		
Input 1		0	1	x
	0	0	0	0
	1	1	0	x
	x	0	x	x

OR

		Input 2		
Input 1		0	1	x
	0	0	1	x
	1	1	1	1
	x	x	1	x

XOR

		Input 2		
1		0	1	x
	1	1	0	x
t	x	x	x	x

XNOR

		Input 2		
Input 1		0	1	x
	0	1	1	x
	x	x	x	x

Negation

Input	0	1	x
Output	1	0	x

5.5.2.7 Shift Operators

Table 5.9 shows the two operators of this category. The << operator executes left shift operation, while the >> operator executes the right shift operation. In either case the operand specified on the left is shifted by the number of bits specified on the right. The shifting is done irrespective of whether the bits are 0, 1, **x**, or **z**. The bits shifted out are lost. The vacated positions created as a result of the shifting are filled with zeroes. If the right operand is **x** or **z**, the result has an x value. If the right operand is negative, the left operand remains unchanged.

5.5.3 Ternary Operator

Verilog has only one ternary operator – the conditional operator. It checks a condition and does a branching. It is a versatile and powerful operator. It enhances the potential of design description substantially (as can be seen through the examples below). The general form is

A?b:c

The conditional operation is made up of two operators – "?" and ":" – and three operands. The two operands separate the three operators in the order shown. The operational sequence of the operation is as follows:

Table 5.9 Shift-type operators and their symbols

Operand	Typical usage	Operation
>>	A >> b	The set of bits representing A are shifted right repeatedly b times.
<<	A<< b	The set of bits representing A are shifted left repeatedly b times.

"A" is evaluated first.

If A is true, b is evaluated. If A is false, c is evaluated.

If A evaluates to an ambiguous result, both b and c are evaluated. Then they are combined on a bit-by-bit basis to form the resultant bit stream. The result bit can have the following three possible values:

0 if the corresponding bits of b and c are 0. 1 if the corresponding bits of b and c are 1. **x** otherwise.

As an example, consider the assignment statement

assign y = w ? x : z;

where w, x, y and z are binary bits. If the bit w is true (1), y is assigned the value of x: otherwise – that is, if w is false (0) – y is assigned the value of z. The assignment statement here multiplexes x and z onto y; w is the control signal here. Consider the assignment

assign flag= (adr1 == adr2)?1'b1 : 1'b0;

Here adr1 and adr2 are two multibit vectors representing two addresses. If the two are identical, the flag bit is set to zero; else it is reset.

assgn zero_flag = (|byte)? 0:1;

All the bits of the byte are ORed together here. The zero_flag is set if the result is zero.

assign c = s ? a: b; //The netcis connected toaifs=1; else it is connected tob

The statement realizes a 2 to 1 mux. b and c have to be scalars or vectors of the same width. The assignment can be expanded to realize larger muxes.

The conditional operator can be nested [see Figure 5.19]. Nesting gives rise to a variety of uses of the operator. As an example, consider the formation of an ALU. ALU can be defined in a compact manner using the ternary operator.

assign d = (f==add)?(a+b): ((f==subtract)?(a-b): ((f==compl)?~a: ~b));

assign o = (s == 2'b00) ? I0 : ((s == 2'b01) ? I1 :

((s == 2'b10) ? I2 : I3));

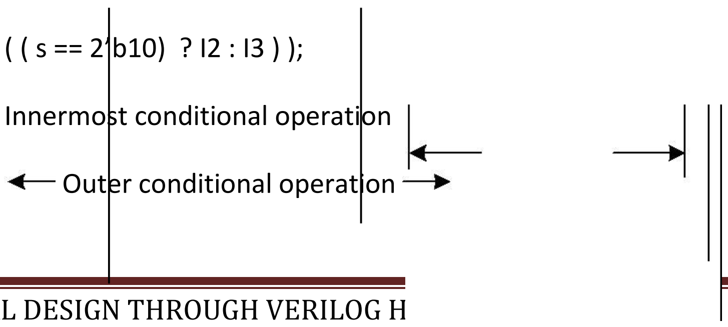


Figure 5.19 Illustration of nested conditional operations.

In the example here, f is taken as a control word. If it is equal to the number add, d is to be equal to the sum of a and b. If f is equal to the number subtract, d is tobe equal to the difference between a and b. If it is equal to the number compl, d is to be the complement of a. Otherwise (*i.e.*, f = 3) d is taken as the complement of b. As another example consider a mux; the assignment statement in Figure 5.18represents a 4-to-1 mux formed with a nested set of ternary operators. The construct in the figure can be judiciously used to form muxes of larger sizes.

Example 5.2 ALU

Figure 5.20 shows an ALU module. It is built around a single executable statement present as a continuous assignment. A test bench for the ALU is also shown in the figure. The synthesized circuit is shown in Figure 5.21. Results of running the test bench are shown in Figure 5.22. Some of the combinational circuit operations required are realized inside the “modgen” blocks of the FPGA used. The nature of the ALU description in the module decides the translation into circuit. Contrast this with the ALU considered at the gate level of design in Section 5.7 where each functional block is instantiated separately and the selected set of outputs steered to the final output. Each such instantiated module translates into a separate circuit block. Their outputs are mux’ed into the final output vector. There is a one-to-one correspondence between the elements of the design description and their respective realizations.

```
module alu_df1 (d, co, a, b, f,cci); //a SIMPLE ALU FOR ILLUSTRATION
PURPOSES output [3:0] d;
output co; wire[3:0]d;wire co; input cci;
input [3 : 0 ] a, b;
input [1 : 0] f;//f is a two-bit function select input; assign
{co,d}=(f==2’b00)?(a+b+cci):((f==2’b01)?(a-b)
    :((f==2’b10)? {1’bz,a^b}:{1’bz,~a})); /*co is the carry bit in case of addition;it is the borrow bit in
case of subtraction. In the other two cases, co is not required. Hence it is assigned z value.*/

endmodule
module tst_aludf1; //test-bench reg [3:0]a,b;
reg[1:0] f; reg cci; wire[3:0]d; wire co;
alu_df1 aa(d,co,a,b,f,cci); initial
begin
cci= 1’b0; f = 2’b00; a = 4’b0;

b    = 4’h0;

end always
begin
```

```
#2 cci = 1’b0;f = 2’b00;a = 4’h1;b = 4’h0; #2 cci = 1’b1;f = 2’b00;a = 4’h8;b = 4’hf; #2 cci =
1’b1;f = 2’b01;a = 4’h2;b = 4’h1; #2 cci = 1’b0;f = 2’b01;a = 4’h3;b = 4’h7; #2 cci = 1’b1;f = 2’b10;a
= 4’h3;b = 4’h3; #2 cci = 1’b1;f = 2’b11;a = 4’hf;b = 4’hc; end

initial $monitor($time, “ cci = %b , a= %b ,b = %b , f = %b ,d =%b ,co= %b “,cci
,a,b,f,d,co);
initial #30 $stop; endmodule
```

Figure 5.20 A 4-bit 4-function ALU and a test bench for the same.

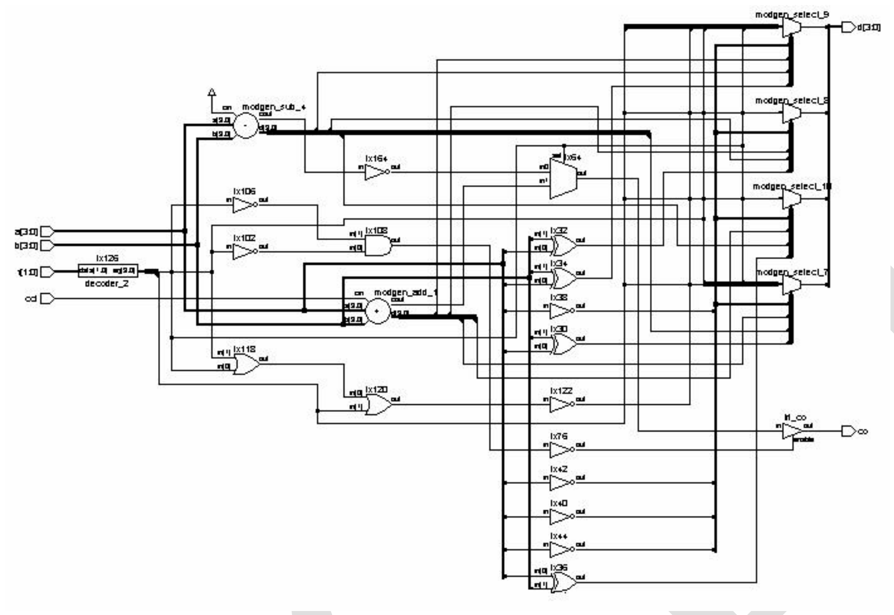


Figure 5.21 Synthesized circuit of the ALU in Example 5.18.

output listing

```
#    0 cci = 0 , a= 0000 ,b = 0000 ,f = 00 ,d =0000 ,co= 0
#    2 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0
#    4 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
#    5 cci = 1 , a= 0010 ,b = 0001 ,f = 01 ,d =0001 ,co= 0
#    8 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co= 1 #10 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d
     =0000 ,co= z #12 cci = 1 , a= 1111 ,b = 1100 ,f = 11 ,d =0000 ,co= z #14 cci = 0 , a= 0001 ,b =
     0000 ,f = 00 ,d =0001 ,co= 0 #15 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1 #18 cci = 1 ,
     a= 0010 ,b = 0001 ,f = 01 ,d =0001 ,co= 0 #20 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co=
     1 #22 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d =0000 ,co= z #24 cci = 1 , a= 1111 ,b = 1100 ,f = 11
     ,d =0000 ,co= z #25 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0 #28 cci = 1 , a= 1000 ,b =
     1111 ,f = 00 ,d =1000 ,co= 1
```

Figure 5.22 Results of running the test bench for the ALU module in Figure 5.20.

6 FUNCTIONS, TASKS, AND USER-DEFINED PRIMITIVES

6.1 INTRODUCTIUON

Bigger designs are better arranged in small functional blocks; it facilitates debugging and any reorganization. Thus a module can have well-defined sub-modules inside, treated as separate entities. Functions and Tasks are such entities inside modules. They play three broad roles:

A well-defined structure with a separate identity. They can hide some variables.

They can be repeatedly invoked within the module.

User-defined primitive (UDP) provides an alternative form of a submodule; it can realize specific outputs. The UDP has a specific format. It can be defined by the user and used wherever necessary. The fact that the UDP has a specific format allows a straightforward definition – often at the expense of flexibility.

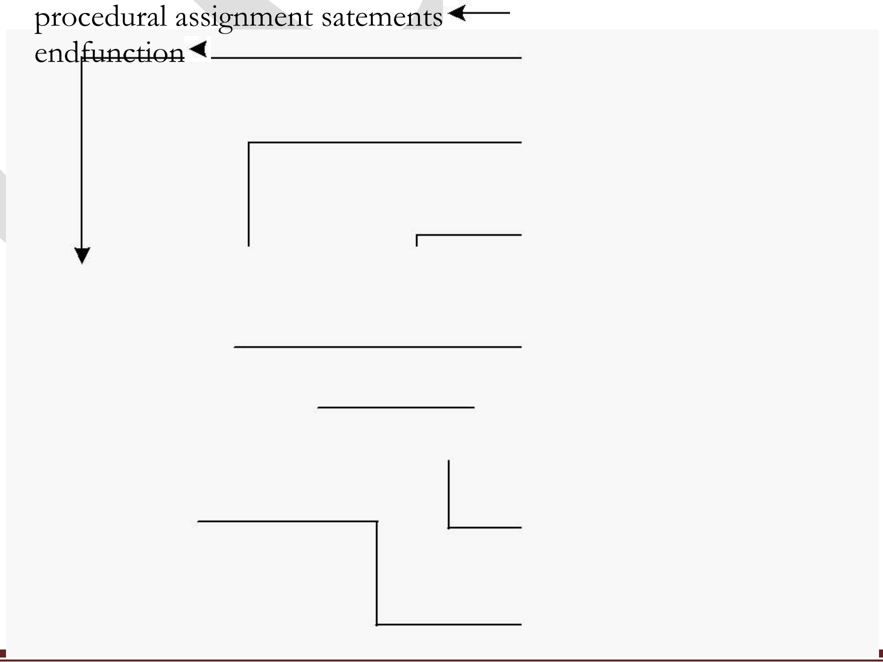
function declaration

The function can return a real or integer type data;it can be a vector with a specified size. The default value is a binary bit.

The name assigned to the function; the

▼▼ function is instantiated with this name. **function** type_or_size function_name ;

input declarations ◄
local variable declarations◄
procedural assignment satements◄
endfunction◄



All inputs to the function and their sizes are declared here. A function must have at least one input

Variables local to the function are declared. They are not available

outside the function

Represent the function body. It may be a single procedural assignment or a collection of them within a begin-end construct

Signifies termination of a function definition

Figure 6.1 Structure for function definition.

6.2 FUNCTION

A function is like a subroutine or a procedure in a program. It is defined separately within a module and can be called whenever necessary. When a function is declared with a function name, the system allocates a register for it. The name of the register is that of the function; and its type (as well as size) is also that of the function. When a function is called, the system executes the functional activity and generates the output. Eventually the output is assigned to the register identified for the function. The quantity returned by the function can be used as an operand in an assignment or in an expression. The structure of a function definition is shown in Figure 6.1. The significance of each of the quantities as well as the rules of using them is also explained in the figure. The use of functions is brought out through a set of examples.

Example 6.1

The function odd-parity is defined within the module parity-check in Figure 6.2. It generates a parity bit. The parity bit is 1 if the number of one-bits in the byte is odd. Otherwise it is zero. The module has an 8-bit vector input and a flag input – en. It has an output chk. Whenever the flag goes high, the function odd-parity iscalled. It returns the parity bit value and assigns it to chk in the module. parity-check is an example with a single-bit output-type function in it. The function hasno local variables in it.

```
        module parity_chk(a,en,chk); input[7:0]a;
        input en; output chk; wire[7:0] a; reg chk;
        always @(posedge en) begin
   chk=pb(a);

   $display("t=%0d, a = %b, en = %0b, pb = %0b ",$time,a,en,chk);

        end
  function pb; input[7:0]a; pb=^a; endfunction
        endmodule module tst_pchk; reg [7:0]a;
        reg en; wire chk; integer i;
        parity_chk pchk(a,en,chk); initial #0 en=1'b0;
        always #2 en = ~en; initial
        begin
   #1 a=8'h00; for(i=0;i<8;i=i+1) begin

   #4 a=a+3'o5;

   end

        end
        initial #40 $stop; endmodule
```

Figure 6.2 A module for parity generation through a function.

```
#    t=2, a = 00000000, en = 1, pb = 0
#    t=5, a = 00000110, en = 1, pb = 0
#    t=10, a = 00001100, en = 1, pb = 0
#    t=14, a = 00010010, en = 1, pb = 0
#    t=18, a = 00011000, en = 1, pb = 0
#    t=22, a = 00011110, en = 1, pb = 0
#    t=25, a = 00100100, en = 1, pb = 0
#    t=30, a = 00101010, en = 1, pb = 1
#    t=34, a = 00110000, en = 1, pb = 0
#    t=38, a = 00110000, en = 1, pb = 0
```

Figure 6.3 Simulation results of the test bench in Figure 6.2

Example 6.2

Figure 6.4 shows another module for parity generation. The module has a function to count the number of one-bits in the input byte. In the module the parity bit is decided by mod-2 division of the number returned by the function. The function has an integer declared and used within it. (In contrast, in the last example the parity bit was generated directly within the function defined.)

```
module parity(p,a,En); input[7:0]a;
input En; output p; reg p;
always @(posedge En) begin
p=n1(a)%2; //Use n1 & generate the parity bit. $display("t=%0d, a = %b, en = %b, p = %b
",$time,a,en,p);

end

function integer n1; //A function to count the number of 1 bits in a byte input[7:0]a;
integer i;
for(i=0;i!=8;i=i+1) begin

if(i==0) n1=0; if(a[i]) n1=n1+1;

end endfunction endmodule
```

Figure 6.4 A module to generate a parity bit: The parity bit is generated by counting thenumber of one-bits in a function and doing a mod-2 division.

Example 6.3

In the module of Figure 6.5 the number of one-bits is decided by shifting out the bits of the input vector and counting the ones in them. Otherwise the module is similar to the one in Figure 6.4. The module (as well as the previous ones) can be easily extended to generate the parity bit for wider binary streams.

```
module parity_a(p,a,En); input[7:0]a;
input En; output p; reg p;
always @(posedge En) begin
p=nn(a)%2;
$display("t=%0d, a = %b, En = %b, p = %b ",$time,a,En,p);

end
function integer nn; input[7:0]a; integer i;
begin
for(i=0;i!=8;i=i+1) begin

if(i==0) nn=0; if(a[i]) nn=nn+1; a=a>>1;

end

end endfunction endmodule
```

Figure 6.5 Another module to generate a parity bit similar to that in Figure 6.4.

Example 6.4

Figure 6.5 shows an adder module to add two 2-bit numbers. The module has two functions defined in it – a half-adder and a full-adder. Further, one can see that the full-adder function itself calls the half-adder function within it. The module calls the full-adder function repeatedly within itself. A test bench for the adder is also included in the figure. The simulation results are shown in Figure 6.7.

```
module adderfun(r,p,q,En);
input[1:0] p,q; input En; output [2:0] r; reg[2:0]r,c; integer i; always@(posedge En)
begin
for(i=0;i<2;i=i+1) begin

if(i==0) c[i]=1'b0; {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]);

end

r[2]=c[2];

$display("t=%0d, En = %b, p = %b, q = %b, r = %b ",$time ,En,p,q,r);

end
function[1:0] ha; input a,b; ha={a&b,a^b}; endfunction
function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2; begin
```

a1=ha(a,b);

aa2=ha(a1[0],c); a2[1] = (aa2[1]|a1[1]); a2[0] = aa2[0];

fa=a2;

end endfunction endmodule
module tst_adder_fun; //testbench; reg [1:0] p,q; reg En; wire [2:0] r; adderfun aa(r,p,q,En);
always #2 En=~En; initial begin
En=1'b0; p=2'b01;q=2'b00; #5 p=2'b10;q=2'b10;

#4 p=2'b10;q=2'b11; #4 p=2'b11;q=2'b11; #4 p=2'b01;q=2'b01;

end initial #30 $stop; endmodule

Figure 6.5 A module to illustrate a function calling another one; a test bench is alsoincluded in the figure.

```
#    t=2, En = 1, p = 01, q = 00, r = 001
#    t=5, En = 1, p = 10, q = 10, r = 100
#    t=10, En = 1, p = 10, q = 11, r = 101
#    t=14, En = 1, p = 11, q = 11, r = 110
#    t=18, En = 1, p = 01, q = 01, r = 010
#    t=22, En = 1, p = 01, q = 01, r = 010
#    t=25, En = 1, p = 01, q = 01, r = 010
```

Figure 6.7 Results of running the test bench in Figure 6.5.

Example 6.5

A module to add two 32-bit numbers is shown in Figure 6.8. It is essentially a scaled-up version of the one in Figure 6.5. The addition is initiated by the En input going high; it is carried out in one time step. A test bench is also included in the figure. The simulation results for a specific set of input number combinations are shown in Figure 6.6.
module add32(r,p,q,En);
input[31:0] p,q; input En; output [32:0] r; reg[32:0]r,c; integer i; always@(posedge En) begin
for(i=0;i<32;i=i+1) begin

if(i==0) c[i]=1'b0; {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]); end

r[32]=c[32];

```
        $display( "t=%0d, En = %b, p = %0h, q = %0h, r = %0h ",$time, En,p,q,r);

        end

            function[1:0] ha; input a,b; ha={a&b,a^b}; endfunction
            function [1:0]fa;
            input a,b,c; reg[1:0]a1,a2,aa2; begin
    a1=ha(a,b);
    aa2=ha(a1[0],c);

    a2[1] = (aa2[1]|a1[1]);
    a2[0] = aa2[0];

        fa=a2;

end
endfunction

            endmodule
            module tst_add32; //testbench;
            reg [31:0] p,q; reg En; wire [32:0] r;
add32 aa(r,p,q,En);
always #2 En=~En;

            initial     begin
    #0 En = 1'b0;
    #3 p  = 32'h1234;      q = 32'h4321;

        #4 p = 32'h12345578; q = 32'h68755432; #4 p = 32'habcdef12; q = 32'hbbccddee; #4 p =
    32'hfedcba36; q = 32'h13576bdf; #4 p = 32'h6875abcd; q = 32'hfedc8755; #4 p = 32'hf0e0d0c0; q
    = 32'h11020304;

    end initial #30 $stop; endmodule
```

Figure 6.8 A scaled-up version of the 2-bit adder in Figure 6.5 to add 32-bit numbers.

```
#   t=2, En = 1, p = x, q = x, r = x
#   t=5, En = 1, p = 1234, q = 4321, r = 5555
#   t=10, En = 1, p = 12345578, q = 68755432, r = aaaaaaaa
#   t=14, En = 1, p = abcdef12, q = bbccddee, r = 1576acd00
#   t=18, En = 1, p = fedcba36, q = 13576bdf, r = 112345518
#   t=22, En = 1, p = 6875abcd, q = fedc8755, r = 167533332
#   t=25, En = 1, p = f0e0d0c0, q = 11020304, r = 101e2d3c4
```

Figure 6.6 Results of running the test bench in Figure 6.8.

Example 6.5

A variant of the adder in Example 6.4 is shown in Figure 6.10: After the enable input en goes high, the full-adder function is called repeatedly in successive clock pulses and bit-wise addition is carried out. The figure also includes a test bench. As can be seen from the simulation results in Figure 6.11, each addition is spread over two clock periods.

```
module adderfunb(clk,r,p,q,En);
input[1:0] p,q; input En,clk; output [2:0] r; reg[2:0]r,c; integer i; always@(posedge En)
begin
for(i=0;i<2;i=i+1) begin @(posedge clk) if(i==0) c[i]=1'b0;

{c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]); end

r[2]=c[2];

$display(" t=%0d, clk = %b, En = %b, p = %b, q = %b, r = %b ",$time,clk,En,p,q,r);

end

function[1:0] ha; input a,b; ha={a&b,a^b}; endfunction
function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2; begin
a1=ha(a,b);

aa2=ha(a1[0],c); a2[1] = (aa2[1]|a1[1]); a2[0]=aa2[0];

fa=a2;

end endfunction endmodule
module tst_adder_funb();
reg [1:0] p,q; reg En,clk; wire [2:0] r; adderfunb bb(clk,r,p,q,En);
always #2 clk=~clk; initial begin
clk=1'b0; En=1'b0; p=2'b01; q=2'b00; #1 En=1'b1; #5 En=1'b0; p=2'b01; q=2'b10; #1 En=1'b1;
#7 En=1'b0; p=2'b01; q=2'b01; #1 En=1'b1; #7 En=1'b0; p=2'b10; q=2'b01; #1 En=1'b1; #7
En=1'b0; p=2'b10; q=2'b10; #1 En=1'b1; #7 En=1'b0; p=2'b10; q=2'b11; #1 En=1'b1; #7 En=1'b0;
p=2'b11; q=2'b11; #1 En=1'b1; #7 En=1'b0;

end initial #50 $stop; endmodule
```

Figure 6.10 A variant of the 2-bit adder in Figure 6.5; bit-wise addition is carried out insuccessive clock pulses.

```
#    t=5, clk = 1, En = 1, p = 01, q = 00, r = 001
#    t=14, clk = 1, En = 1, p = 01, q = 10, r = 011
#    t=22, clk = 1, En = 1, p = 01, q = 01, r = 010
#    t=30, clk = 1, En = 1, p = 10, q = 01, r = 011
#    t=38, clk = 1, En = 1, p = 10, q = 10, r = 100
#    t=45, clk = 1, En = 1, p = 10, q = 11, r = 101
#    t=54, clk = 1, En = 1, p = 11, q = 11, r = 110
```

Figure 6.11 Simulation results of the test bench for the adder module in Figure 6.10.

Example 6.7

A module to add 32-bit numbers is shown in Figure 6.12. It is a scaled-up version of that in the last example. The addition commences after the enable bit En goes high. Starting with the LSB, one bit is added at every succeeding clock pulse. Addition is completed in 32 clock pulses. The simulation results with a set of 32-bit numbers is shown in Figure 6.13.

```
module add32_a(clk,r,p,q,En);
input[31:0] p,q;input En,clk; output [32:0] r; reg[32:0]r,c; integer i; always@(posedge
En) begin
for(i=0;i<32;i=i+1) begin

@(posedge clk)  begin

if(i==0) c[i]=1'b0; {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]); end

end r[32]=c[32];

$display( "t=%0d, En = %b, p = %0h, q = %0h, r = %0h ",$time,En,p,q,r); end

function[1:0] ha;
input a,b; ha={a&b,a^b}; endfunction
function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2; begin
a1     = ha(a,b);
aa2    = ha(a1[0],c);
a2[1] =  (aa2[1]|a1[1]);
a2[0] = aa2[0]; fa = a2;
end endfunction endmodule
module tst_add32a();
reg [31:0] p,q; reg En,clk; wire [32:0] r; add32_a bb(clk,r,p,q,En);
always #1 clk=~clk; initial begin
clk=1'b0;En=1'b0;p=32'h1234;q=32'h4321;
```

```
        #1 En=1'b1;#100 En=1'b0;p=32'h12345578;q=32'h68755432; #1 En=1'b1;#66
        En=1'b0;p=32'habcdef12;q=32'hbbccddee; #1 En=1'b1;#66
        En=1'b0;p=32'hfedcba36;q=32'h13576bdf; #1 En=1'b1;#66
        En=1'b0;p=32'h6875abcd;q=32'hfedc8755; #1 En=1'b1;#66
        En=1'b0;p=32'hf0e0d0c0;q=32'h11020304; #1 En=1'b1;#66 En=1'b0;
end initial #600 $stop; endmodule
```

Figure 6.12 A 32-bit adder with the addition done in successive clock pulses.

```
# t=55, En = 1, p = 1234, q = 4321, r = 5555
# t=155, En = 1, p = 12345578, q = 68755432, r = aaaaaaaa
# t=255, En = 1, p = abcdef12, q = bbccddee, r = 1576acd00
# t=355, En = 1, p = fedcba36, q = 13576bdf, r = 112345518
# t=455, En = 1, p = 6875abcd, q = fedc8755, r = 167533332
# t=555, En = 1, p = f0e0d0c0, q = 11020304, r = 101e2d3c4
```

Figure 6.13 Simulation results of the test bench for the adder in Figure 6.12.

6.2.1 Trade-off Between Hardware and Speed

Examples 6.5 and 6.7 represent two extreme cases of a trade-off between speed and hardware. Minimal hardware is used in Example 6.7 to carry out the addition, but the execution time is a maximum here due to the repeated and sequential use of the same hardware block. In contrast, in Example 6.5 the same hardware is replicated to the maximum extent and the addition is carried out "at one go", that is, in minimum time. Circuit-wise, it is a trade-off between silicon area and speed. One can have nibble or byte adders and do nibble-wise or byte-wise addition; these represent intermediate levels of trade-offs. Algebraic or logic operations, register-based operations, *etc*., are other examples calling for similar trade-off decisions. Buswidth, memory organization, and ALU sizing all call for such trade-off decisions. In all such cases a decision may have to be based on considerations of speed of operation, power consumption, development time, cost, *etc*.

6.2.2 Scope of Functions

A few observations on functions and their use are in order here [IEEE].

A function has only input arguments. It is to have at least one input. When a function with multiple input ports is called, the order of arguments in the calling statement should match that of the input declarations within the

function definition.

A function returns an output. It has no separate output ports.

A function can have variables declared and used within it – these are variables local to the function.

A function can be defined anywhere within the module.

Event or timing based controls are not possible within a function. This restricts the function to be of a combinational logic type.

A function can be called from within another function. Both the functions are to be defined within the module.

A function in a module can be called from another module through proper hierarchical referencing.

A function can be called repeatedly within the module of definition. Expressions can be used as arguments while calling a function.

Definition of a function should not be within any initial or always block. or within another function.

A function uses a register of the declared type and size to return the value of the output. Such a returned value can be **`real`**, **`integer`**, **`time`**, or

`realtime` type. It can also be a vector with a range.

Every variable declared inside a function has a corresponding location inside. These locations are physical entities. Each time a function is called, the same set of locations is reused. This is in contrast to the instantiation of a module where with every instantiation, a fresh set of locations is assigned.

6.2.3 Recursive Functions

Consider a function to compute the sum of the squares of the first *n* natural numbers: The sum designated as S_n can be expressed as
$S_n = .n^2 + (n-1)^2 + + 3^2 + 2^2 + 1^2$

S_n can be expressed as
$S_n = n^2 + S_{n-1}$
where S_{n-1} represents the sum of the squares of the first $(n-1)$ natural numbers. Thus if S_{n-1} were known, S_n can be obtained by adding n^2 to it. Continuing the same argument one can recursively arrive at the following:
$S_{n-1} = (n-1)^2 + S_{n-2}$ $S_{n-2} = (n-2)^2 + S_{n-3}$
…
…

$S_2 = 2^2 + S_1$

We know that

$S_1 = 1$.

The actual computation is carried out in the reverse order; that is, one computes S_1 directly and the subsequent sums S2, S3, *etc*., are computed from it recursively – every sum by adding an increment to the previous sum.

A similar procedure can be adopted to compute factorials, infinite series and so on. Latest version of the LRM (2001) has expanded the scope of Functions to accommodate recursive functions. The keyword **`automatic`** following the keyword **`function`** implies it to be recursive. A recursive function can be called in the same manner as a nonrecursive function. Recursive function call is explained here through an example.

Example 6.8

The module sum_sq in Figure 6.14 computes the sum of the squares the first *n* natural numbers.

```
function automatic integer sum_sq; input n;
begin
if(n==1) sum_sq =1;

else sum_sq = sum_sq + n*n;

end endfunction
```

Figure 6.14 A module to compute the sum of squares of the first*n*natural numbers.

The term "**automatic**" in the function declaration statement ensures recursive computation. Thus if *n* is assigned the value 4, during compilation sum_sq (4) will be successively replaced by
sum_sq (3) + 4^2, sum_sq (4) + 3^2+ 4^2,
sum_sq (4) + 2^2+ 3^2+ $4^2$and finally by$1^2 + 2^2 + 3^2 + 4^2$.

6.3 TASKS

The role of a task in a module is similar to that of a subroutine in a program. It is defined within a module and can be called as many times as desired within a procedural block. Its scope and role are wider than those of a function.

6.3.1 Task Definition

The task definition is brought out in Figure 6.15. The first statement starts with the keyword task; it is followed by an identifier name and the customary semicolon. The input, inout, and the output declarations follow. Their order is not rigid. The body of the task comprises of a number of behavioral level statements. They may be executed in zero time or at specified time intervals or events. Thus the time of exit from a task can differ from that of entry to it.

6.3.2 Task Enabling

A task is enabled through a statement akin to the instantiation of a gate. It is enabled like a procedural assignment by specifying the task name followed by the list of arguments within brackets followed by the semicolon. A typical enabling statement has the form
Do_it (Expression1, Expression2, . .);
where
Do_it is the name of the task being enabled, Expression1 is the first argument, Expression2 is the second argument,
and so on.

The type and order of the arguments should match those of the respective declarations within the definition of the task. In a general case, an argument can be an expression. The following are characteristic of a task:

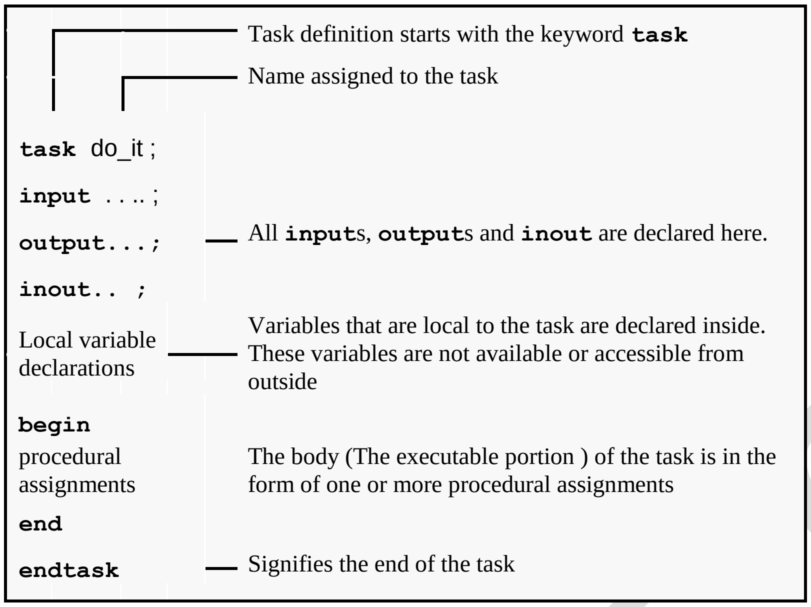

Task definition starts with the keyword **task**
Name assigned to the task
task do_it ;
input . . .. ;
output...;
inout.. ;
All **input**s, **output**s and **inout** are declared here.
Local variable declarations
Variables that are local to the task are declared inside. These variables are not available or accessible from outside
begin
procedural assignments
end
The body (The executable portion) of the task is in the form of one or more procedural assignments
endtask
Signifies the end of the task

Figure 6.15 Typical structure of a task.

A task can be activated by an event, sensitivity list, *etc*.

A task can have activities assigned within it which are event-controlled or time-controlled.

A task can have input, output and inout; however it need not necessarily have any of these; it can be complete in itself.

A task can enable other tasks and functions.

A task can call itself. The latest version of the LRM supports recursion. The keyword **`automatic`** is added to the keyword task to make it recursive.

All assignments to a task are passed to it by value and not through a pointer to the argument.

A task in a module can be invoked from another module through a hierarchical reference.

The arguments passed to a task retain their type within their environment of use. Thus a **`wire-`** type argument passed to a task as input cannot have its value altered within the task through an assignment.

There are no apparent restrictions on the input arguments of a task. They can be nets, regs, or expressions involving them. But any argument of inout or output type has to be a variable or of a similar type; the restrictions are similar to those on the quantities on the left side of procedural assignments. The use of tasks is illustrated through a set of four examples here.

6.4 USER-DEFINED PRIMITIVES (UDP)

The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives – called "user defined primitive (UDP)" and use them. A UDP can be defined anywhere in a source text and instantiated in any of the modules. Their definition is in the form of a table in a specific format. It makes the UDP types of functions simple, elegant, and attractive. UDPs are basically of two types – combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

6.4.1 Combinational UDPs

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An **inout** declaration is not supported by a UDP. The UDP definition is on par withthat of a module; that is, it is defined independently like a module and can be used in any other module. The definition cannot be within any other module.

Definition of a combinational type of UDP is illustrated through an example in Figure 6.22; it shows a simple UDP for an AND operation. The following are noteworthy:

The first statement starts with the keyword "primitive", it is followed by the name assigned to the primitive and the port declarations.

A UDP can have only one output port. It has to be the first in the port list. All the ports following the first are input ports and are all scalars.

inout ports are not permitted in a UDP definition.Output and input are declared in the body of the UDP.

```
          primitive udp_and (out, in1, in2);  output out;
          input in1, in2; table
//           In1          In2          Out
              0           0:            0;
              0           1:            0;
              1           0:            0;
              1           1:            1;
endtable
endprimitive
```

Figure 6.22 A two-input AND gate defined as a UDP.

The behavior block of the primitive is given in the form of a table. It is specified between keywords **table** and **endtable**.

The combinational function is defined as a set of rows (akin to the truth table).

All the input values are specified first – each in a separate field in the same order as they appear in the port declaration.

A colon and then the output value follow the set of input values. The statement ends with a semicolon – as with every statement in Verilog.

A comment line is inserted in the example following the "**`table`**" entry**.** It facilitates understanding the tabular entries.

All the inputs are nets – **`wire`**-type. Hence there is no need for a separate type definition.

Output can be of the net or **`reg`** type depending upon the type of primitive – explained later.

The last keyword statement – "**`endprimitive`**" – signifies the end of the definition.

6.4.2 More General Combinational UDPs

The UDP for the AND gate in Figure 6.22 specifies output values only for definite values of the inputs but not for their **x** states. A full and general definition of a UDP is characterized by the following additional factors:

The output can take on only three values – ***0****,* ***1****, or* ***x****. It cannot take the value* z.

Outputs can be defined for **0**, **1**, or **x** values of the inputs but not for the **z** state. However if an input takes the value **z**, it is taken as **x**.

All the undefined input combinations lead to **x** state in the output. Hence it is desirable to specify outputs for all the possible input combinations.

Figure 6.23 shows the UDP definition of an AND gate with all the input combinations included. A test-bench for the UDP and the simulation results are shown in Figure 6.24.

A two-input UDP has nine rows of tabular entries; their number increases rapidly as the number of input logic variables increases. LRM has the provision to make the UDP definition more compact. The symbol "?" can be used to signify all the possible values – that is, 0, 1, or **x**. Figure 6.25 shows the elaborate AND gate UDP of Figure 6.23 made compact in this manner. Wherever possible, one can use the symbol "**b**" to signify "0" or "1" values and reduce the table size further.

```
Primitive udp_and (out, in1, in2);
Output out; //UDP of an AND gate defined fully
Input in1, in2;
```

Table

//	In1	In2	Out
	0	0:	0;
	0	1:	0;
	1	0:	0;
	1	1:	1;
	X	0:	0;
	X	1:	X;
	X	X:	X;
	0	X:	0;
	1	X:	X;

Endtable
Endprimitive

Figure 6.23 A more exhaustive definition of the two2-input AND gate UDP of Figure 6.21.

```
module tst_udp_and(); reg in1,in2; wire out;
udp_and uand(out,in1,in2);
initial begin in1=1'b0;in2=1'b0; end always begin
#2 in1=1'b0;in2=1'b1; #2 in1=1'b1;in2=1'b0; #2 in1=1'b1;in2=1'b1; end

initial $monitor($time ,"in1 = %b ,in2 = %b ,out = %b ",in1,in2,out); initial #18
$stop;
endmodule
```

Simulation results

```
//#     0in1 = 0 ,  in2 = 0 , out = 0
//#     2in1 = 0 ,  in2 = 1 , out = 0
//#     4in1 = 1 ,  in2 = 0 , out = 0
//#     5in1 = 1 ,  in2 = 1 , out = 1
//#     8in1 = 0 ,  in2 = 1 , out = 0
//#     10in1 = 1 , in2 = 0 , out = 0
//#     12in1 = 1 , in2 = 1 , out = 1
//#     14in1 = 0 , in2 = 1 , out = 0
//#     15in1 = 1 , in2 = 0 , out = 0
```

Figure 6.24 A test bench for the UDP module of Figure 6.23 and the simulation results.

```
Primitive udp_and_b (out, in1, in2);

Output out;// UDP of an AND gate defined compactly
Input in1, in2;
```

Table

//	In1	In2	Out
	?	0:	0;
	0	?:	0;
	x	X	x
	1	1:	1;

Endtable
Endprimitive

Figure 6.25 The UDP of Figure 6.22 made compact using the symbol "?".

6.4.3 Instantiation of an UDP

UDPs are instantiated in the same manner as gate primitives (see the test bench in Figure 6.24). It is further illustrated here through an example.

Example 6.13

The full adder accepts three input bits and outputs two bits – a sum bit and a carry bit. Figure 6.25 shows UDPs for the sum and the carry bits as well as a full adder module using them. Figure 6.27 shows a test-bench for the Full Adder as well as the simulation results.

```
primitive udpsum(sum, in1,in2,carryi); output sum;
input in1, in2, carryi;
table
//      in1     in2     carryi:   sum
         0       0        0:       0;
         1       1        0:       0;
         0       1        1:       0;
         1       0        1:       0;
         1       0        0:       1;
         0       1        0:       1;
         0       0        1:       1;
         1       1        1:       1;
endtable endprimitive
primitive udpcar(caro,in1,in2,cari); // This udp is for carryout output caro; input in1,
in2, cari;
table
//      in1     in2     cari     caro
         0       0       ? :      0 ;
         0       ?       0 :      0 ;
         ?       0       0 :      0 ;
        b        1       1 :      1 ;
         1       b       1 :      1 ;
         1       1       b :      1 ;
endtable endprimitive
module fa (car_o, sum_o, in1, in2, car_i); input in1, in2, car_i; output car_o, sum_o;
udpcar aa(car_o,in1,in2,car_i);
udpsum bb(sum_o, in1,in2,car_i); endmodule
```

Figure 6.25 A full adder module with the sum and carry bits generated through UDPs.

```
module fa_tst;
reg [2:0] a;wire c,s;integer i; fa cc(c,s,a[0],a[1],a[2]); initial for(i=1;i<8;i=i+1) begin
a=i;

#1 $display($time, "a=%b, cs=%b%b",a, c, s); end
initial #10 $stop; endmodule
```

Simulation results

```
#   1a=001, cs=01
#   2a=010, cs=01
#   3a=011, cs=10
#    4a=100, cs=01
#    5a=101, cs=10
#    5a=110, cs=10
#    7a=111, cs=11
```

Figure 6.27 A test bench for the full adder module of Figure 6.25 and the simulation resultsfor the same.

Observations:

With three inputs and three states for each input (0, 1, and **x**), the full table of definition has 27 entries. Such definitions become cumbersome as the number

of inputs increase to even moderate values – say 4 or 5.

Only the entries essential to the definition of the primitive are included here. Others which lead to **x** output are left out intentionally. Thus with the carry primitive if any two inputs have **x** values, the output car_o too has **x** value.

Hence such a row has not been specified.

"?" and "**b**" have been used in the primitive definition to make the tables more compact

6.4.4 Combinational UDP and Function

Definition-wise, UDP and function are similar, though their formats differ (*i.e.*, a UDP definition is in the form of a table while the function definition is as a sequence of procedural assignments). UDPs are stand-alone-type primitives and can be instantiated in any module. In contrast, a function is defined within a module; it cannot be accessed anywhere outside the module of definition.

6.4.5 Sequential UDPs

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state [Wakerly]. A positive or a negative going edge or a simple change in a logic variable can trigger the transition from the present state of the circuit to the next state. A sequential UDP can accommodate all these. The definition still remains tabular as with the combinational UDP. The next state can be specified in terms of the present state, the values of input logic variables and their transitions. The definition differs from that of a combinational UDP in two respects:

*The output has to be defined as a **reg**. If a change in any of the inputs so*
demands, the output can change.

Values of all the input variables as well as the present state of the output can affect the next state of the output. In each row the input values are entered in different fields in the same sequence as they are specified in the input port list. It is followed by a colon (:). The value of the present state is entered in the next field which is again followed by a colon (:). The next state value of the output occupies the last field. A semicolon (;) signifies the end of a row definition (see the examples below).

6.4.6 Sequential UDPs and Tasks

Sequential UDPs and tasks are functionally similar. Tasks are defined inside modules and used inside the module of definition. They are not accessible to other modules. In contrast, sequential UDPs are like other primitives and modules. They can be instantiated in any other module of a design.

6.4.7 UDP Instantiation with Delays

Outputs of UDPs also can take on values with time delays. The delays can be specified separately for the rising and falling transitions on the output. For example, an instantiation as
udp_and_b # (1, 2) g1(out, in1, in2);
can be used to instantiate the UDF of Figure 6.25 for carry output generation. Here the output transition to 1 (rising edge) takes effect with a time delay of 1 ns. The output transition to 0 (falling edge) takes effect with a time delay of 2 ns. If only one time delay were specified, the same holds good for the rising as well as the falling edges of the output transition.

6.4.8 Vector-Type Instantiation of UDP

UDP definitions are scalar in nature. They can be used with vectors with proper declarations. For example, the full-adder module fa in Figure 6.25 can be instantiated as an 8-bit vector to form an 8-bit adder. The instantiation statement can be
fa [7:0] aa(co, s, a, b, {co[5:0],1'b0});
s (sum), co (carry output), a (first input), and b(second input) are all 8-bit vectors here. The vector type of instantiation makes the design description compact; however, it may not be supported by some simulators.

Truth Tables of Gates and Switches

The truth tables for gates are given with two inputs each; it remains the same for multiple inputs as well. The inputs are designated as 'Input 1' and 'Input 2'; the output values are in the respective cells of the table.

Table B.1 Truth table of AND gate

Input 1					
		0	1	x	z
Input 2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

Table B.2 Truth table of OR gate

Input 1					
		0	1	x	z
Input 2	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

Table B.3 Truth table of NAND gate

Input 1					
		0	1	x	z
Input 2	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

Table B.4 Truth table of NOR gate

Input 1					
		0	1	x	z
Input 2	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

Table B.5 Truth table of XOR gate

		Input 1			
		0	1	x	z
Input 2	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

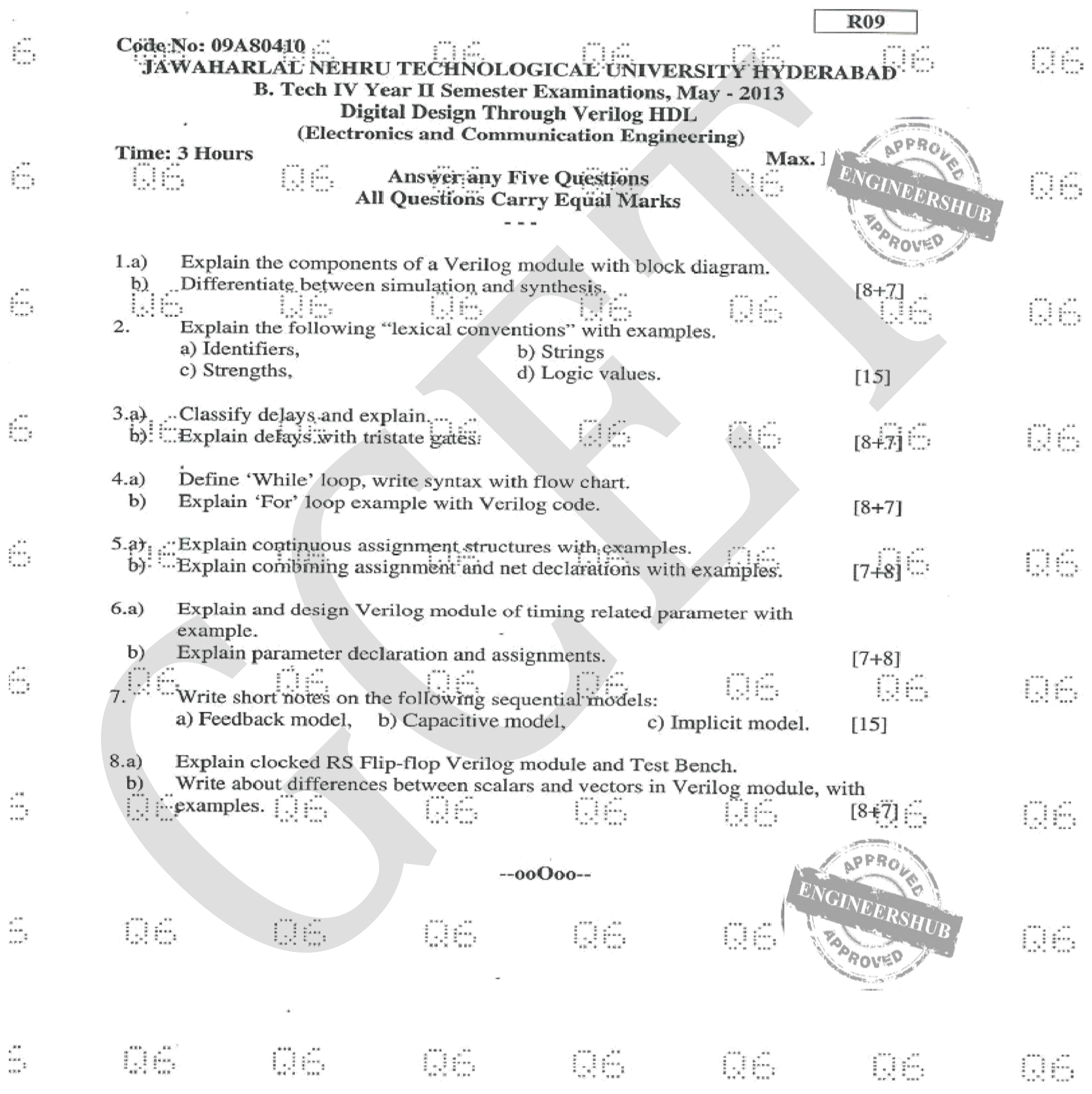
Table B.5 Truth table of XNOR gate

		Input 1			
		0	1	x	z
Input 2	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

15. Additional Topics :

Shall be provided later, as this has a revised syllabus and the course content is to be studied in details.

16.University previous Question papers:



R09

Code No: 09A80410

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B. Tech IV Year II Semester Examinations, May - 2013

Digital Design Through Verilog HDL

(Electronics and Communication Engineering)

Time: 3 Hours Max.]

Answer any Five Questions

All Questions Carry Equal Marks

- - -

1.a) Explain the components of a Verilog module with block diagram.
b) Differentiate between simulation and synthesis. [8+7]

2. Explain the following "lexical conventions" with examples.
a) Identifiers, b) Strings
c) Strengths, d) Logic values. [15]

3.a) Classify delays and explain.
b) Explain delays with tristate gates. [8+7]

4.a) Define 'While' loop, write syntax with flow chart.
b) Explain 'For' loop example with Verilog code. [8+7]

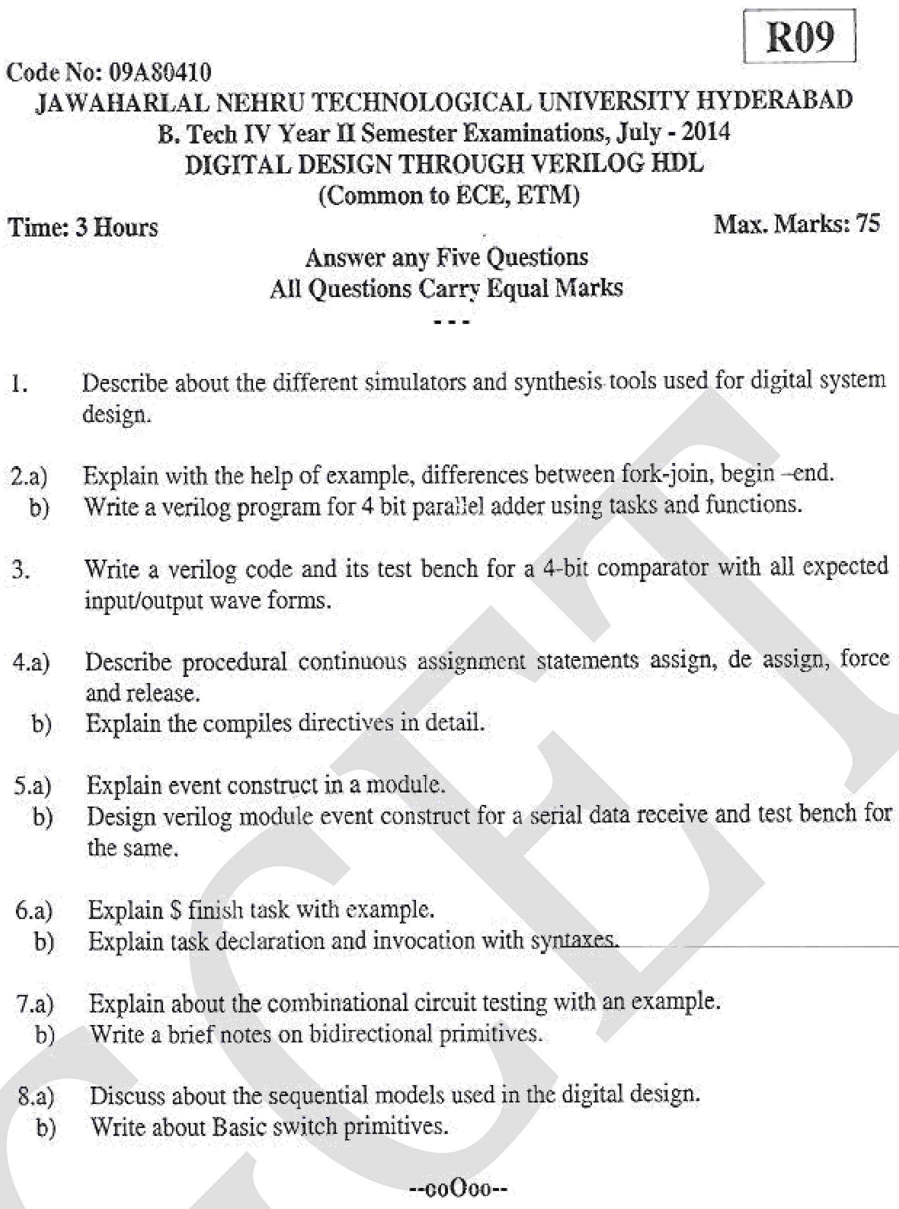
5.a) Explain continuous assignment structures with examples.
b) Explain combining assignment and net declarations with examples. [7+8]

6.a) Explain and design Verilog module of timing related parameter with example.
b) Explain parameter declaration and assignments. [7+8]

7. Write short notes on the following sequential models:
a) Feedback model, b) Capacitive model, c) Implicit model. [15]

8.a) Explain clocked RS Flip-flop Verilog module and Test Bench.
b) Write about differences between scalars and vectors in Verilog module, with examples. [8+7]

--ooOoo--



R09

Code No: 09A80410

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech IV Year II Semester Examinations, July - 2014
DIGITAL DESIGN THROUGH VERILOG HDL
(Common to ECE, ETM)

Time: 3 Hours Max. Marks: 75

Answer any Five Questions
All Questions Carry Equal Marks

- - -

1. Describe about the different simulators and synthesis tools used for digital system design.

2.a) Explain with the help of example, differences between fork-join, begin –end.
 b) Write a verilog program for 4 bit parallel adder using tasks and functions.

3. Write a verilog code and its test bench for a 4-bit comparator with all expected input/output wave forms.

4.a) Describe procedural continuous assignment statements assign, de assign, force and release.
 b) Explain the compiles directives in detail.

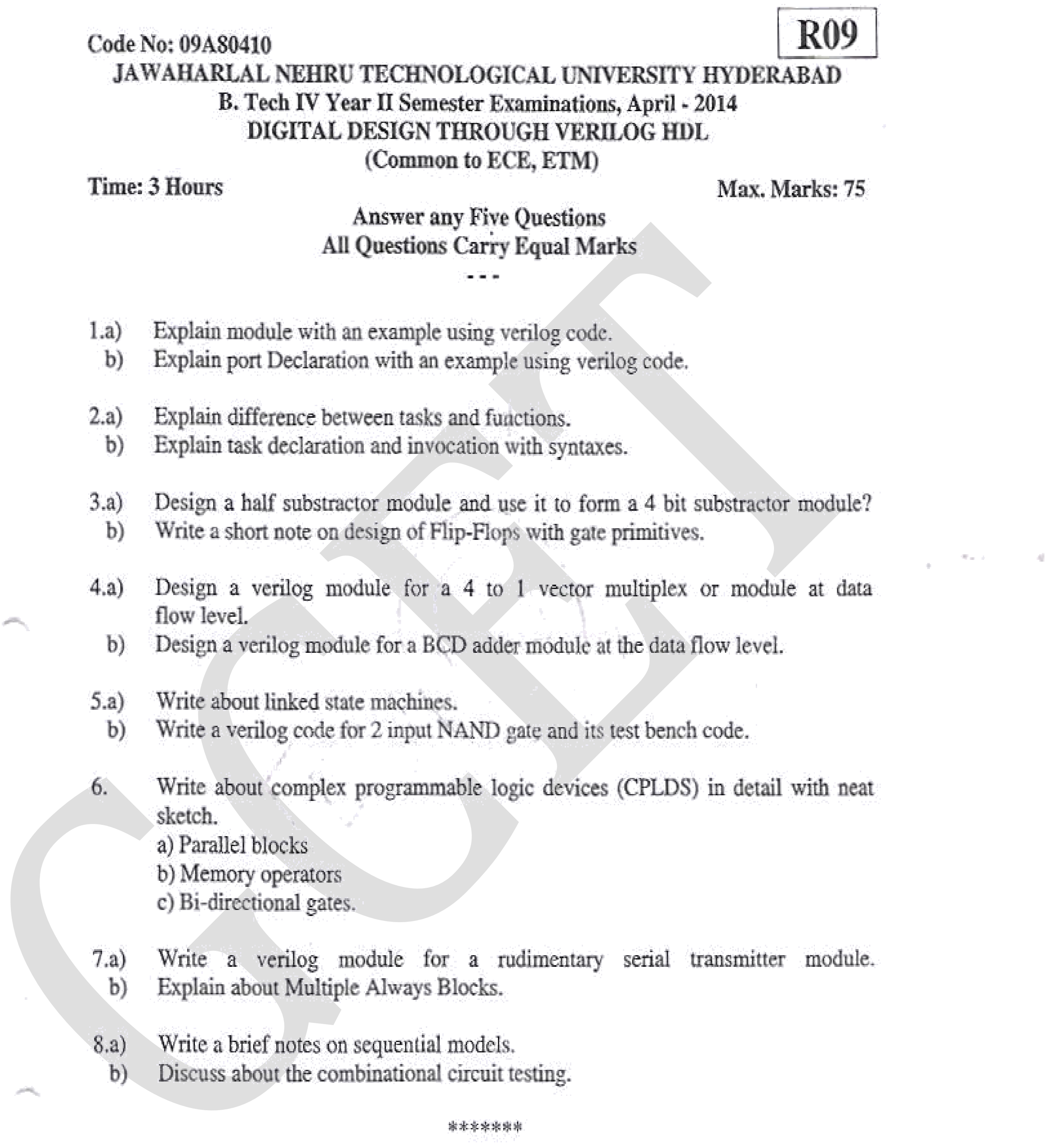
5.a) Explain event construct in a module.
 b) Design verilog module event construct for a serial data receive and test bench for the same.

6.a) Explain $ finish task with example.
 b) Explain task declaration and invocation with syntaxes.

7.a) Explain about the combinational circuit testing with an example.
 b) Write a brief notes on bidirectional primitives.

8.a) Discuss about the sequential models used in the digital design.
 b) Write about Basic switch primitives.

--ooOoo--



Code No: 09A80410 **R09**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B. Tech IV Year II Semester Examinations, April - 2014

DIGITAL DESIGN THROUGH VERILOG HDL

(Common to ECE, ETM)

Time: 3 Hours **Max. Marks: 75**

Answer any Five Questions

All Questions Carry Equal Marks

- - -

1.a) Explain module with an example using verilog code.
b) Explain port Declaration with an example using verilog code.

2.a) Explain difference between tasks and functions.
b) Explain task declaration and invocation with syntaxes.

3.a) Design a half substractor module and use it to form a 4 bit substractor module?
b) Write a short note on design of Flip-Flops with gate primitives.

4.a) Design a verilog module for a 4 to 1 vector multiplex or module at data flow level.
b) Design a verilog module for a BCD adder module at the data flow level.

5.a) Write about linked state machines.
b) Write a verilog code for 2 input NAND gate and its test bench code.

6. Write about complex programmable logic devices (CPLDS) in detail with neat sketch.
a) Parallel blocks
b) Memory operators
c) Bi-directional gates.

7.a) Write a verilog module for a rudimentary serial transmitter module.
b) Explain about Multiple Always Blocks.

8.a) Write a brief notes on sequential models.
b) Discuss about the combinational circuit testing.

****

Code No:07A80405 **R07** **SetNo. 2**

IVB.TechIISemesterExaminations,APRIL2011DIGITAL DESIGN THROUGH VERILOG

Common to Bio-Medical Engineering, Electronics AndComputerEngineering, Electronics And CommunicationEngineering

Time: 3hours Max Marks:80

Answer any FIVEQuestionsAll Questions carry equalmarks

1. (a) Classify and explain strengths and contentionresolution?
   (b) Designmoduletoillustrateuseofthewand-typenetandtestbenchwithstimulationresults? [8+8]
2. (a) Design half-adder module with time delay assignment through parameterdeclaration.
   (b) Write Test bench, simulation results for theabove. [8+8]
3. Explain the followingterms.
   (a) Simulation
   (b) Synthesis
   (c) Implementation
   (d) HDLS [16]
4. (a) Design verilog code of OR gate using for anddisable.
   (b) Write simulation results of above question with explanation. [8+8]
5. Design HDL module for UART Transmitter. [16]
6. (a)Designaverilogmoduleofa4bitbusswitcheratthedataflowlevel.
   (b) Design verilog module of an edge triggered flip-flop built with the latch atthedata flowlevel. [8+8]
7. Explain one hot state assignment withexample. [16]
8. (a)Drawtheblockdiagramforadividerthatdividesan8-bitdividendbya5-bitdivisortogivea3-bitquotient.Thedividendregistershouldbeloadedwhen$St=1$.
   (b) Draw an SM chart for the controlunit. [8+8]

Code No:07A80405 **R07** **SetNo. 4**

IVB.TechIISemesterExaminations,APRIL2011DIGITAL DESIGN THROUGH VERILOG

Common to Bio-Medical Engineering, Electronics AndComputerEngineering, Electronics And CommunicationEngineering

Time: 3hours Max Marks:80

Answer any FIVEQuestionsAll Questions carry equalmarks

1. (a) Design CMOSflipflop.
   (b) Design verilog module for CMOS flipflop. [8+8]

2. (a) Define While loop, write syntax with flowchart.
   (b) Explain for loop example with verilogcode. [8+8]

3. (a) Design a D flip flop using NAND gates.
   (b) Write a verilog code for D flip flop using NAND gates. [8+8]

4. (a) Explain the linked statemachines.
   (b) Explain the linked SM charts to Dicegame. [8+8]

5. (a) Write about $ readmemb with example.
   (b) Write value change dumpfile. [8+8]

6. Explain UART Receiver with SM Chart. [16]

7. Explain about flex 10k embedded arrayblock. [16]

8. (a) Explain simple latch with verilogmodule?
   (b) Explain RS Flip-flop with verilog module and TestBench? [8+8]

Code No: R05420405 Set No. 3

IV B.Tech II Semester supply Examinations, September/November 2009 DIGITAL DESIGN THROUGH VERILOG

(Common to Electronics & Communication Engineering, Bio-Medical Engineering and Electronics & Computer Engineering)

Time: 3 hours Max Marks: 80

Answer any FIVE Questions

All Questions carry equal marks

1. Explain Top-down Design methodology with example? [16]
2. (a) Design a Master Slave JK flip flop using NAND gates.

(b) Write a verilog code for Master Slave JK flip flop using NAND gates. [8+8]

3. Write Short Notes for following with Examples:

(a) Intra Assignment Delays
(b) Delay Assignments
(c) Zero Delay [16]

4. (a) Design CMOS switch with a single control line.

(b) Design code, testbench, results for CMOS switch with a single control line. [8+8]

5. (a) Explain module paths.

(b) Design verilog module using of path delay. [8+8]

6. (a) Explain Dice game with block diagram.

(b) Explain Dice game using flow chart. [8+8]

7. (a) Explain about XC4000 implementation of multiplier control.

(b) Write differences between FPGA and CPLD. [8+8]

8. Design HDL module for Baud rate generator.

Code No: R05420305 Set No. 1

IV B.Tech II Semester supply Examinations, September/November 2010 DIGITAL DESIGN THROUGH VERILOG

(Common to Electronics & Communication Engineering, Bio-Medical Engineering and Electronics & Computer Engineering)

Time: 3 hours Max Marks: 80

Answer any FIVE Questions
All Questions carry equal marks

1. (a) Design a JK flip flop using NAND gates.

   (b) Write a verilog code for JK flip flop using NAND gates.[8+8]
2. (a) Explain module with an example using verilog code?

   (b) Explain port Declaration with an example using verilog code?[8+8]
3. (a) Explain edge sensitive path using an example.

   (b) Explain over riding parameters.[8+8]
4. Explain UART Transmission with SM Chart. [16]
5. (a) Explain NMOS enhancement with conditions.

   (b) Write about Basic switch primitives.[8+8]
6. Explain parallel adder-subtractor with logic cell. [16]
7. (a) Write a verilog module for a rudimentary serial transmitter module.

   (b) Explain Multiple Always Blocks.[8+8]
8. (a) Construct an PLA and D-flip flop equivalent to the following state table. Test only one variable in each decision box. Try to minimize the number of decision boxes.

   (b) Write a VHDL description of the state machine based on the PLA and D-flip flop. [8+8]

17.Question Bank:

UNIT1:

1) Explain programming language interface
2) Explain levels of design description
3) Explain simulation and synthesis with differences
4) Write about system tasks with examples?

UNIT2:

1) Mention keywords and their significance?
2) Explain data types of Verilog.
3) Explain the following (a) scalars and vectors (b) parameters (c) white space.
4) Explain operator in Verilog.
5) Explain system tasks.

UNIT 3:

1) Explain gate level modeling with example?
2) Design half adder using gate level modeling?
3) Design full adder using gate level modeling?
4) Design full adder using half adder using gate level modeling?
5) Explain delays with an example
6) Explain net types
7) Design d flip flop with gate primitives
8) Explain tri state gates
9) Write about module structure
10) Write the Verilog program for 2 bit comparator in gate model?

UNIT 4

1) Write about continuous assignment structures
2) Explain assignment to vectors
3) Explain delays with a program
4) Explain wait construct with an example
5) Explain force release construct with an example
6) Explain forever loop
7) Explain the difference between blocking and non blocking assignments
8) Explain repeat construct
9) Explain design at behavioral levels
10) Explain if and else if constructs
11) Explain case statement with a program
12) Write about simulation flow?

UNIT 5:

1) Explain operators in data flow?
2) Design half adder using data flow modeling?
3) Design full adder using data flow modeling?
4) Design full adder using half adder using data flow modeling?

5) Write the Verilog code for cmos NOR in data flow model
6) Write the Verilog code for nmos NOR in data flow model
7) Write the Verilog code for cmos NAND in data flow model.
8) Explain basic transistor switches.
9) Explain CMOS switches.
10) Write the Verilog code for CMOS NOR in switch level model.
11) Write the Verilog code for NMOS NOR in switch level model.
12) Write the Verilog code for CMOS NAND in switch level model.

UNIT 6:

1) Explain parameters
2) Explain path delays
3) Explain file based tasks with an examples
4) Explain hierarchical access with a program
5) Explain system based tasks and functions
6) Explain sequence detector with FSM program
7) What are user defined primitives
8) What are complier directives
9) Explain module parameters
10) Explain Dice game with block diagram.Explain Dice game using flow chart.
11) Explain and design verilog module of timing related parameter with example.
12) Explain parameter declaration and assignments.

UNIT 7:

1) Write a brief notes on sequential models.
2) Write short notes on
    a) Feedback model
    b) Capacitive model
    c) Implicit model
3) Write about the differences between scalars and vectors in verlog modules with example.

UNIT 8:

1) Explain clocked rs flip-flop verilog module and test bench.
2) Write a verilog module for a rudimentary serial transmitter module.
3) Explain about multiple always blocks.
4) Discuss about the combinational circuit testing.

18.Assignment Topics: To be provided after revising previous question papers.

19.Unit wise bits:

Unit 1:

1) At the circuit level, a switch is the basic element with which digital circuits are built.
2) Verilog has the basic MOS switches built into its constructs.

3) All the basic gates are available as ready modules called "Primitives".
4) All possible operations on signals and variables are represented here in terms of assignments.
5) In an electronic circuit all the units are to be active and functioning concurrently.
6) Concurrency is achieved by proceeding with simulation in equal time steps.
7) Translation of the debugged design into the corresponding hardware circuit is called "synthesis."
8) The test benches are mostly done at the behavioral level.
9) Any basic gate has propagation delays and transmission delays associated with it.
10) PLI provides an active interface to a compiled Verilog module.
11) Any Verilog program begins with a keyword – called a "module."
12) A module comprises a number of "lexical tokens" arranged according to some predefined order.
13) Each module can be defined only once.
14) One module cannot be defined inside another – they cannot be nested.
15) Simulation results can alternately be viewed as waveforms.

Unit 2:

1) Any source file in Verilog is made up of a number of ASCII characters.
2) Verilog is a case-sensitive language like C.
3) A keyword signifies an activity to be carried out, initiated, or terminated.
4) Module signifies the beginning of a module definition. endmodule signifies the end of a module definition.
5) All characters of the alphabet or an underscore can be used as the first character.
6) 1_name not allowed as an identifier, since the numeral "1" is the first character.
7) \abc // Here the combination "abc" forms the identifier.
8) One can incorporate multiline comments also without resorting to "//" at every line.
9) Integers can be represented in two ways. In the first case it is a decimal number – signed or unsigned; an unsigned number.
10) When a signal line is driven simultaneously from two sources of different strength levels, the stronger of the two prevails.
11) The data handled in Verilog fall into two categories: Net data type and Variable data type
12) A net signifies a connection from one circuit unit to another.

Unit 3:

1) The last statement in any module definition is the keyword "endmodule".
2) In all cases of instantiations, one need not necessarily assign a name to the instantiation.
3) **and** ga (o, i1, i2, . . . i8); Output port is o
4) The A-O-I gate module has three instantiations – two of these being AND gates and the third a NOR gate.
5) Signals at the ports can be identified by a hierarchical name.
6) The primitives available in Verilog can also be instantiated as arrays.
7) The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself.
8) In all array-type instantiations, the array sizes are to be matched.
9) Verilog has the facility to account for different types of propagation delays of circuit elements.
10) Any connection can cause a delay due to the distributed nature of its resistance and capacitance.
11) One of the simplest delays is that of a direct connection – a net. It can be part of the declaration statement.
12) In practical situations, outputs of logic gates and signals on nets in a circuit have associated source impedances.

Unit 4:

1) Behavioral level modeling constitutes design description at an abstract level.
2) The description is carried out essentially with constructs similar to those in "C" language.
3) The design description at the behavioral level is done through a sequence of assignments.
4) Only the blocks which are to be identified and referred by the simulator need be named.
5) Regs declared and used within a block are static by nature.

6) A set of procedural assignments within an **initial** construct are executed only once – and, that too, at the times specified for the respective assignments.
7) Simulators have the facility to observe the waveforms and changes in the magnitudes of different variables with simulation time.
8) A module can have as many **initial** blocks as desired.
9) The progress of simulation time in different blocks is concurrent.
10) The **always** block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block.
11) A delay of 0 ns does not really cause any delay. However, it ensures that the assignment following is executed last in the concerned time slot.
12) The **wait** construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment.
13) Delays – of the assignment type and the intra-assignment type – can be associated with non-blocking assignments.
14) The **case** statement is an elegant and simple construct for multiple branching in a module.

Unit 5:

1) A simple two input AND gate in data flow format has the form **assign** c = a && b;
2) "**assign**" is the keyword carrying out the assignment operation.
3) Delays can be incorporated at the data flow level in different ways.
4) One can concatenate vectors, scalars, and part vectors to form other vectors.
5) The concatenated vector is enclosed within braces.
6) Operators can carry out specified operations on the operands and assign the results to a net or a vector set of nets.
7) Unary operators do an operation on a single operand and assign the result to the specified net.
8) If any bit of an operand is **x** or **z** in an arithmetic operation, the result takes the **x** value.
9) The equality operator makes a bit-by-bit comparison of the two operands and produces a result bit.

Unit 7:

1) In digital circuits , storage of data is done either by feedback or by gate capacitances that are refreshed frequently.
2) Feedback models and capacitive models are technology dependent.
3) Verilog provides timing check constructs for ensuring correct operation of implicit modeling.
4) A sequential UDP has the format of the combinational UDP except that its inputs, outputs and present state is also specified.
5) Q = d; q_b = ~d; are blocking assignments.
6) Q <= d; q_b <= ~d; are non-blocking assignments.
7) With each clock edge, the entire procedural block is executed once from begin to end.
8) A fork-join bracketing instead of begin-end causes all sequential statements to be executed in parallel.
9) A sequential assign statement forces a value into reg type variable, and a sequential deassign removes it.
10) Setup time is the minimum necessary time that a data input requires to setup before it is clocked into a flip-flop.
11) Hold time is the minimum necessary time that a flip-flop data must stay stable after it is clocked.
12) $readmemh and $readmemb tasks are for reading external data files and using them for initialization of memory blocks.
13) An inout bus is only considered as net and cannot be declared as a reg.
14) Verilog PLA modeling tasks use a personality memory whose contents determine PLA fusing.
15) A register is a group of flip-flops with a common clock.
16) A moore machine is a state machine in which all outputs are fully synchronized with the circuit clock.
17) In a mealy machine, its output depends on its current state and inputs.

Unit 8:

1) Verilog simulation environment provide tools for graphical or textual display of simulation results.

2) A Verilog testbench is a Verilog module that instantiates an MUT applies data to it and monitors its output.
3) Testing sequential circuits involves synchronization of circuit clock with other data inputs.
4) $stop and $finish are simulation control tasks.
5) Formal verification is a way of automating design verication by eliminating testbenches and problems associated with their data generation and response observation.
6) The assert_one_hot assertion monitor checks that while the monitor is active only one bit of its n-bit expression is 1.
7) The assert_cycle_sequence is a very useful assertion for verifying state machines.
8) A useful assertion for checking expected events or events implied by other events is the assert_implication assertion.
9) Assert_next assertion verifies that starting and an ending events occur with a specified number of clocks in between.
10) In assertion verification, in-code monitors take the responsibility of issuing a message if something happens that is not expected.

20.Tutorial class sheets:

Shall be provided later, the course content is to be studied in details.

21.Known gaps:

Shall be provided later, the course content is to be studied in details.

22. Discussion Topics if any

1) Design of digital circuits using different modeling levels of Verilog.
2) To write a synthesizable code.
3) To understand the use of test bench in real time projects.

23.References, Journals, websites and E-links:

REFERENCES:
1. Fundamentals of Logic Design with Verilog – Stephen. Brown and Zvonko Vranesic, TMH, 2005.
2. Digital Systems Design using VHDL – Charles H Roth, Jr. Thomson Publications, 2004.
3. Advanced Digital Design with Verilog HDL – Michael D. Ciletti, PHI, 2005.
4. Digital systems Design using VHDL – Charles H Roth, Jr. Thomson Publications, 2004.

24.Quality Control Sheets:

Shall be provided later

25. Students List

To be updated

26. Group-Wise students list for discussion topics

To be updated