Design Through Verilog HDL

IEEE Press
445 Hoes Lane
Piscataway, NJ 08854

Technical Reviewers

Robert S. Hanmer, Lucent Technologies, Naperville, IL
Zhou Feng, Fudan University, China

Design Through Verilog HDL

T. R. Padmanabhan
B. Bala Tripura Sundari

IEEE PRESS

A JOHN WILEY & SONS, INC., PUBLICATION

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Padmanabhan, T. R.
Design through Verilog HDL / T. R. Padmanabhan, B. Bala Tripura Sundari.
p. cm.
Includes bibliographical references and index.
ISBN 0-471-44148-1 (cloth)
1. Verilog (Computer hardware description language) I. Tripura Sundari, B. Bala. II. Title.

TK7885.7.P37 2003
621.39'2–dc22 2003057671

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my parents

B. Bala Tripura Sundari

To Ravi and Chandra

T.R. Padmanabhan

CONTENTS

PREFACE

Verilog has rapidly become a widely accepted language for VLSI design. The language is well-structured and defined to cater to the steady increase in the size of ICs to be designed without sacrificing the advantages associated with design at the "grass roots" level. A designer aspiring to master the language in its versatility should become familiar with the various constructs in it, practice their use in real applications, and use them in combinations to be successful.

Describing a design using Verilog is only half the story: Writing Test benches, testing a design for all its desired functions, and identifying the faults and removing them remain equally challenging tasks. This book is an attempt to address these issues effectively. The constructs in Verilog are discussed through apt illustrative examples. Equal importance is given to design description and test benches. The examples have been tested with popular and commonly used simulation packages and the results reproduced. In many of the cases the tested designs have been synthesized, and the synthesized circuit has also been reproduced. "Seeing is believing": Seeing a design available as a software routine, transformed to a circuit, will add a lot to the confidence level of novices who use the book. flip-flops, counters, registers, coders, decoders, mux, demux *etc*., have been considered at different levels of design; this should help in clarifying the perspectives regarding levels, need, and significance.

Place and significance of Verilog in VLSI design have been brought out in Chapters 1 and 2. Basics of the language, its conventions, *etc*., are dealt with in Chapters 2 and 3. Chapters 4 and 5 form an introduction to design through Verilog. It is done at the gate level, which may be the most comfortable for the beginner. Any design, however involved it may be, can be completely realized in terms of the gate primitives of Verilog. We hope that the illustrative examples considered and the exercises at the end of the chapters, impart such a confidence to a designer. Chapter 6 is devoted to design at the data flow level. Continuous assignments using operators linking operands, which allow designs to be described more compactly but still close enough to the circuit level, form the theme of this chapter. Behavioral level design is discussed in Chapters 7 and 8. Mastery at this level – akin to the C language – is essential for a successful designer working at the system level. Functions and tasks, which facilitate structuring of designs and their orderly description, form the theme of Chapter 9. The switch primitives in Verilog constitute the link with actual VLSI implementation although their mastery is not essential to many of the designers with their higher level activities. Chapter 10 is devoted exclusively to switch level design; since it stands out from

the main text flow so far, its discussion is consciously deferred to this stage. Chapter 11 forms an introduction to the system tasks and functions in Verilog and their use in typical environments. Chapter 12 deals with design using PLDs and FSMs. Though subdued, the treatment is enough to give the necessary lead to more comprehensive designs.

All the chapters have enough exercises at the end. Some help mastery of the material in the chapter, through practice; others are structured to stimulate the users to explore avenues of their own. The step-by-step build-up of a processor in Chapter 12 is of this type.

All simulation results presented in the text as part of illustrative examples, have been obtained using the "Modelsim" software of Mentor Graphics. All synthesis results wherever presented, have been obtained using the "Leonardo Spectrum" software of Mentor Graphics. These have been reproduced by courtesy of Mentor Graphics.

Users' views and suggestions are welcome; for this purpose, the website *www.aitec.amrita.edu/publications* may be accessed.

T. R. PADMANABHAN
B. BALA TRIPURA SUNDARI

July 2003

ACKNOWLEDGEMENTS

Many of our acquaintances and associates have contributed to the fruition of this venture. K.N.C. Eswaran is responsible for all the delicate and subtle touches with Word. Our colleagues — Subha, Sathyapriya, and Rajagopal — have made many useful suggestions. Anand Srinivasan helped with simulation in his own way. Ajai Narendran of the Systems Wing of our Institute has been helpful in many ways. Our families — Krishna Sudarshan, Saketh, Srikanth, Ravi, Chandra, and Uma — have put up with our transient oddities. Brahmachari Abhayamrita Chaitanya — Chief Operating Officer of Amrita Vishwa Vidyapeetham — made the Institute facilities, especially the VLSI laboratory, available for us. Dr. N. Narayana Pillai, Dean (Students), and Prof. R. Sundararajan of our Institute have been of great encouragement to us. Ms Christina Kuhnen, Associate Acquisitions Editor at IEEE Press, has been quite helpful throughout; she has effectively bridged the distance between New York and Coimbatore. The painstaking efforts of the Referees to wade through the manuscript, understand the matter and their constructive suggestions have conspicuously contributed to the book in its present form. We give our sincere thanks to all of them.

Our obeisance goes to *Mata Amritanandamayi Devi* for her commitment to societal transformation through quality education; this is a humble attempt to add another brick to the edifice being built by her.

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, telecommunications *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.

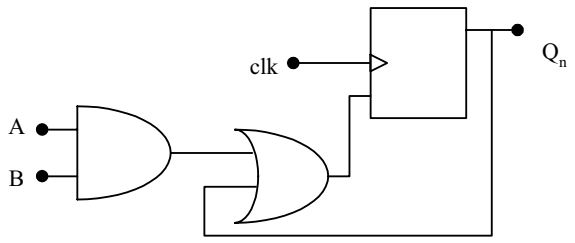


Figure 1.1 A simple digital circuit.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

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.

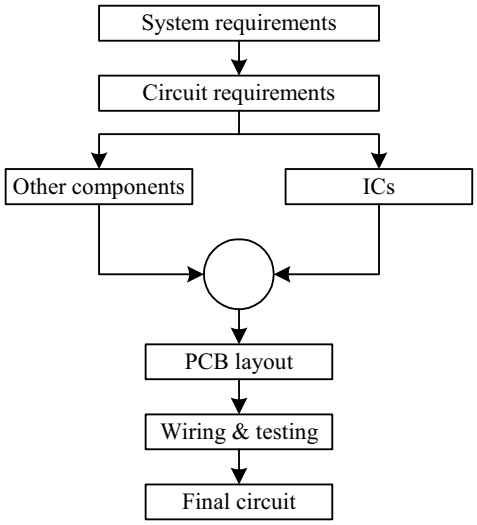


Figure 1.2 Sequence of steps in conventional electronic circuit design.

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

- *Functional mismatch*: The realized and expected functions are different. One may 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. One possibility 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 signal transition 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.

1.3.1 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.

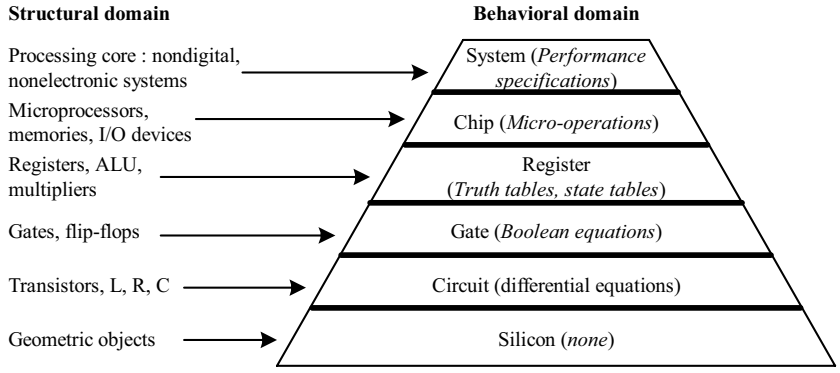


Figure 1.3 Design domain and levels of abstraction.

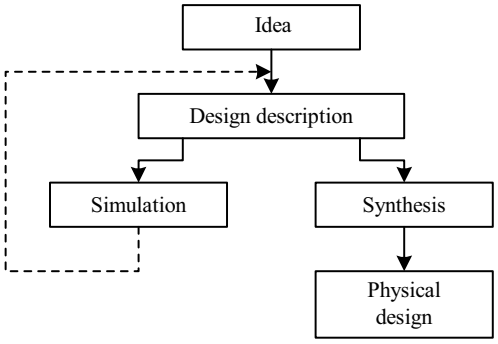


Figure 1.4 Major activities in ASIC design.

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.

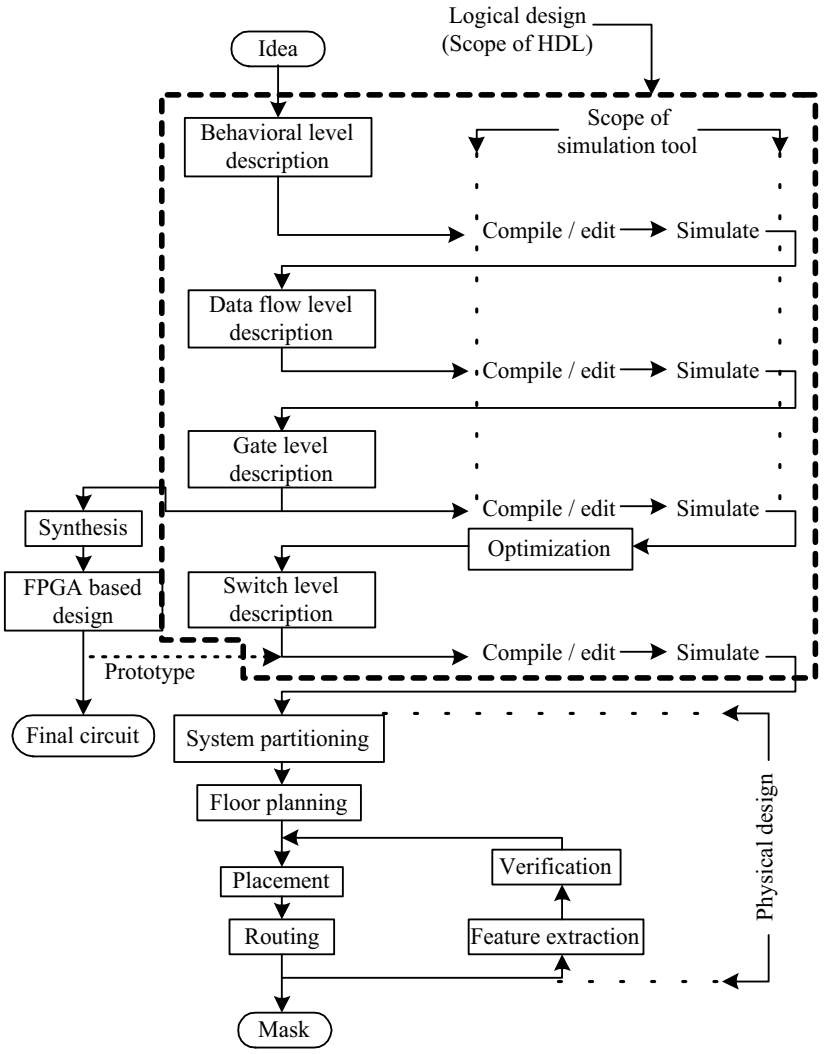


Figure 1.5ASIC design and development flow.

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.

1.4.4 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 compartments or 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 the blocks 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 interconnected to 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.

2

INTRODUCTION TO VERILOG

2.1 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.

2.2 LEVELS OF DESIGN DESCRIPTION

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

2.2.1 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.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

2.2.2 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.

2.2.3 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.

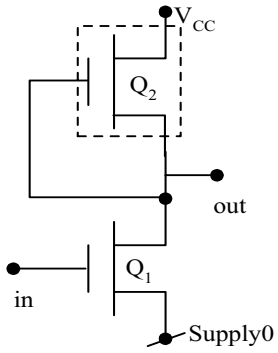


Figure 2.1 A simple Inverter circuit at the switch level.

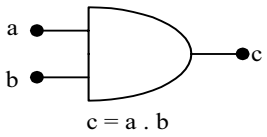


Figure 2.2 A simple AND gate represented at the gate level.

2.2.4 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.

2.2.5 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.

2.3 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

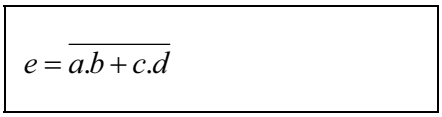


Figure 2.3 An A-O-I gate represented as a data flow type of relationship.

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

Figure 2.4 An A-O-I gate in pseudo code at 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.

2.4 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").

2.5 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.

2.5.1 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.

2.5.2 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.

2.6 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*.

2.7 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.

2.8 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:

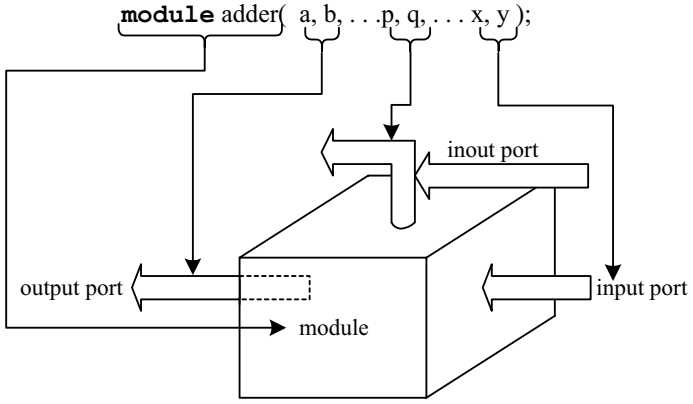


Figure 2.5 Representation of a module as black box with its ports.

- **`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 the module 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 2.7b).

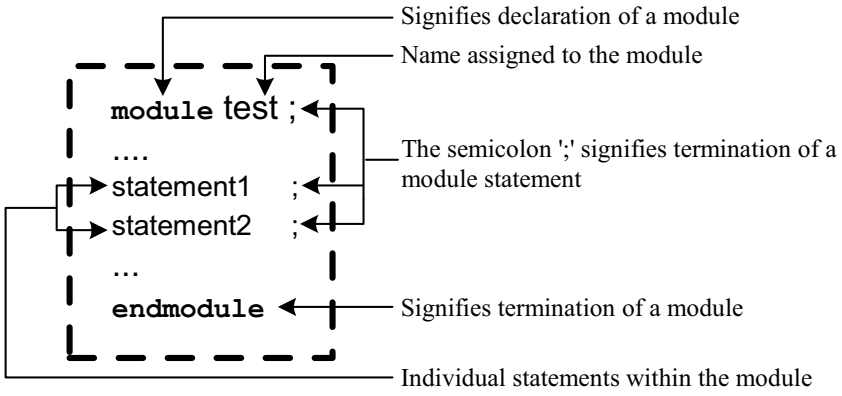


Figure 2.6 Structure of a typical simulation module.

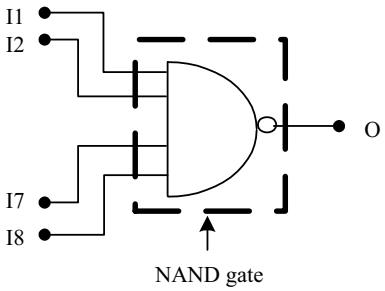


Figure 2.7(a) Eight input NAND gate (IC 7430). Gate proper with terminals.

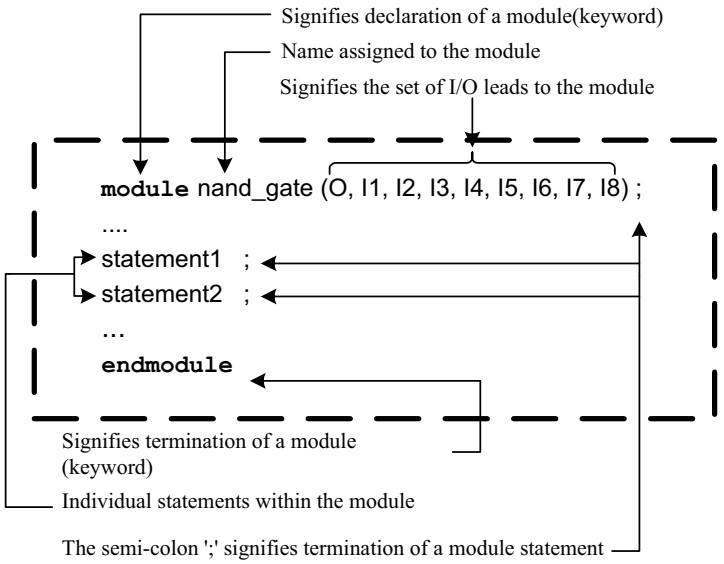


Figure 2.7(b) Eight input NAND gate (IC 7430). Structure of the gate module.

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 2.8. 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 2.1 Partial list of IC numbers and their types for a typical design

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

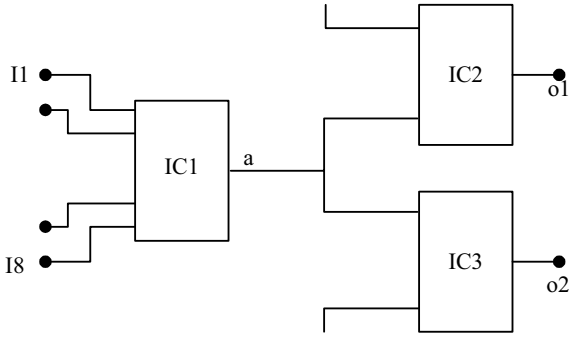


Figure 2.8 Part of the circuit diagram of a typical digital circuit.

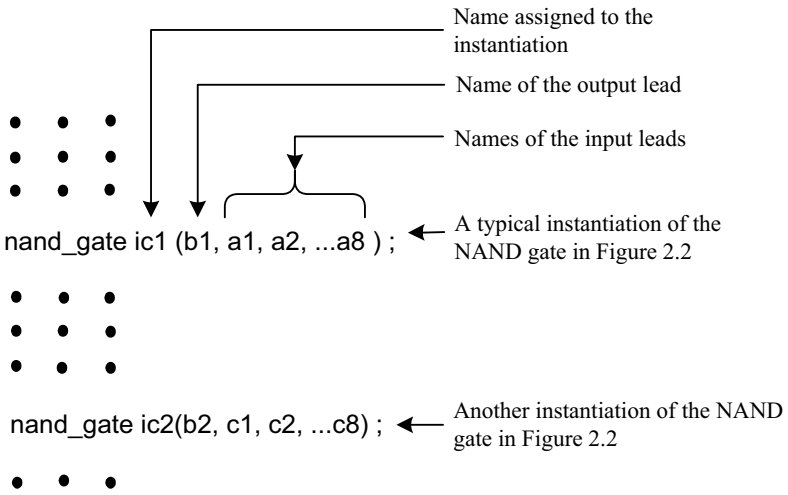


Figure 2.9 Instantiations of module nand_gate in another module.

- 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*.).

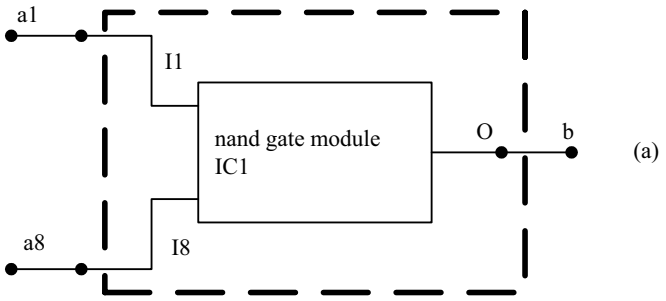
- As part of the instantiation declaration, the input and output terminals are to be defined. The convention followed is to stick to the same order as in the module declaration. It is further illustrated in Figure 2.9.

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 2.10 (a) A typical circuit block and (b) its instantiation.

2.9 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.

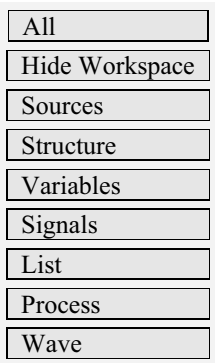
2.9.1 Use of Modelsim SE 5.5

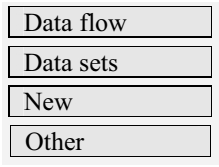
The procedure to invoke the tool and use it is briefly described here. The tool can be used to prepare a source file, edit and compile it, and simulate the compiled version.

Editing and Compilation

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

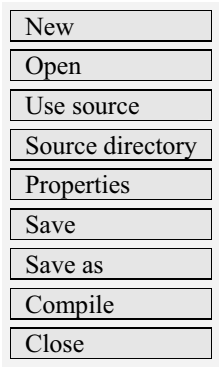
- Click on "View." We get the following menus:





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

- Click on "File" option. We get the following options:



- Click on "New." We get the following options:

- 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:–

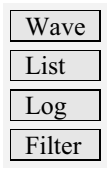
- Select “Design” and click on it. A small window appears on the screen. “Library: Work” is displayed, implying that the working library is open. The module name “demo” is displayed under it. In the normal course the names of all the compiled files will be listed alphabetically one below the other. The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

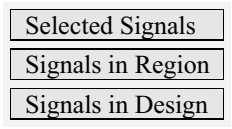
Simulation results can alternately be viewed as waveforms with the following procedure:

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

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



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



- 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.

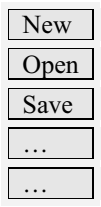
One can practice simulation of a few examples given in the book. Subsequently options available at the different stages can be tried, and the tool with its full versatility can be mastered.

2.9.2 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:

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



- 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

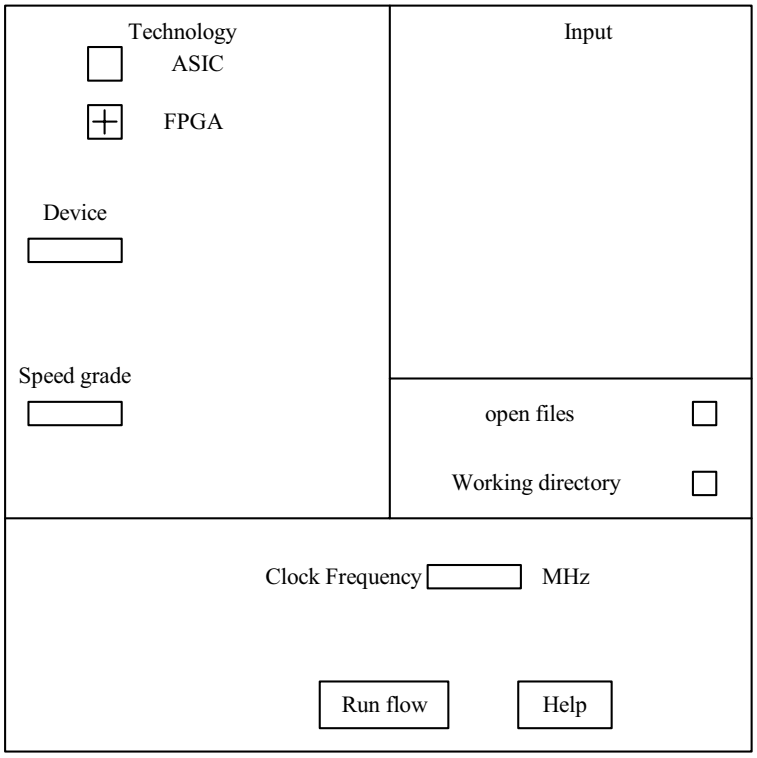


Figure 2.11 The Window in Leonardo Spectrum to do the settings for synthesis.

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

2.10 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 4.1:

```
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.

3

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

3.1 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.

3.1.1 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.

3.2 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

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

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.

3.3 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 as identifiers
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 first character.
1_name ← not allowed as an identifier, since the numeral "1" is the first character
@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.

3.4 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.

3.5 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.*/

3.6 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.

3.6.1 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
25

253
–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.

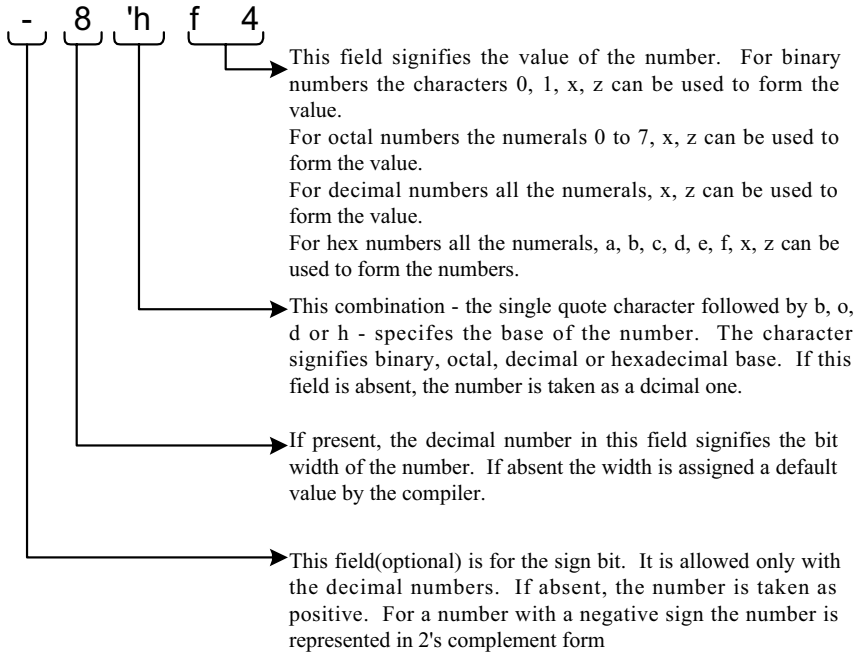


Figure 3.1 Representation of a number in Verilog: One can use capital letters instead of small letters in the last two fields.

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.

3.6.2 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.

3.7 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 3.1 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__1**x**01 9'b11011**x**01 9'B11011**x**01	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'o1**x**3 9'o12**z**	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'h**z**a	A hex number of 9 bits. Its value is taken as **zzzzz** 1010.
5'h**z**a	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).

3.8 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. Details of contention and its resolution are discussed in Chapter 5.

3.9 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.

3.2 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.

3.10 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.

3.10.1 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 is connected to a wire and is driven by that.

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

Functionally, **wire** and **tri** are identical. Distinct nomenclatures are provided for the convenience of assigning roles. Other types of nets are discussed in Chapter 5.

3.10.2 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.

3.11 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.

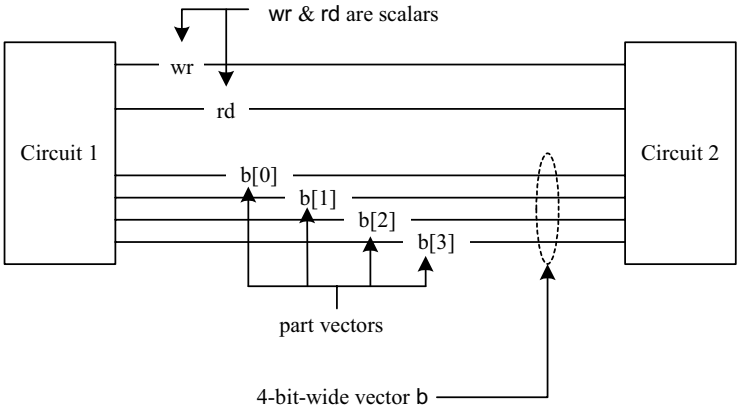


Figure 3.2 Illustration of scalars and vectors.

Examples:

```
wire[3:0] a;      /* a is a four bit vector of net type; the bits are designated as
                     a[3], a[2], a[1] and a[0]. */
reg[2:0] b;       /* b is a three bit vector of reg type; the bits are designated as
                     b[2], b[1] and b[0]. */
reg[4:2] c;       /* c is a three bit vector of reg type; the bits are designated as
                     c[4], c[3] and c[2]. */
wire[-2:2] d ;    /* d is a 5 bit vector with individual bits designated as d[-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;       // num is 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.
```

3.12 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;
```

3.13 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]. The index 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.

3.14 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.

Operators are discussed in detail in Chapter 6.

3.15 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>

3.15.1 $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 of a is : 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 advances output to a new line. */

3.15.2 $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 of a is : 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.

3.15.3 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.

3.16 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)
```

continued

continued

```
        begin
                for(i=0;i<=j;i=i+1) $write(" c");
                $display("*");
                j=j-1;
        end
end
initial #12 $stop;
endmodule
```

Figure 3.3 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 3.4 Another simple Verilog module.

4

GATE LEVEL MODELING – 1

4.1 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.

4.2 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.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

4.2.1 Example 4.1

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 4.1 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 4.2 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.

4.2.2 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 4.1 Truth table of AND gate primitive

		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*

- 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.

4.3 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.

4.4 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.

4.4.1 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.

4.5 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 4.2 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 4.3 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	

4.5.1 Example 4.2

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.

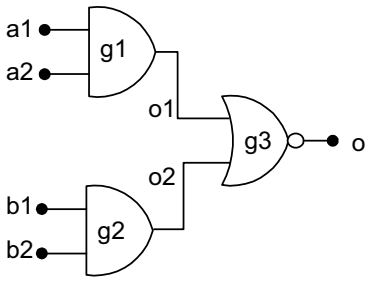


Figure 4.3 A typical A-O-I gate circuit.

```
/*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 4.4 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 4.5 Results of running the aoi_st test bench of Figure 4.3.

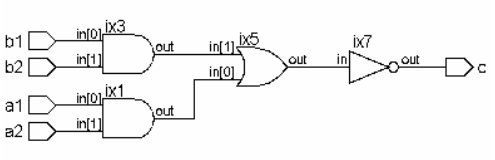


Figure 4.6 Synthesized version of the module aoi_gate of Figure 4.4.

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 4.7 Another realization of the A-I-O gate with the input declared as a vector; the test bench 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.

4.5.2 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 4.8 Results of running the aoi_st2 test bench of Figure 4.7.

Figure 4.9(c) 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 enabled only 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.

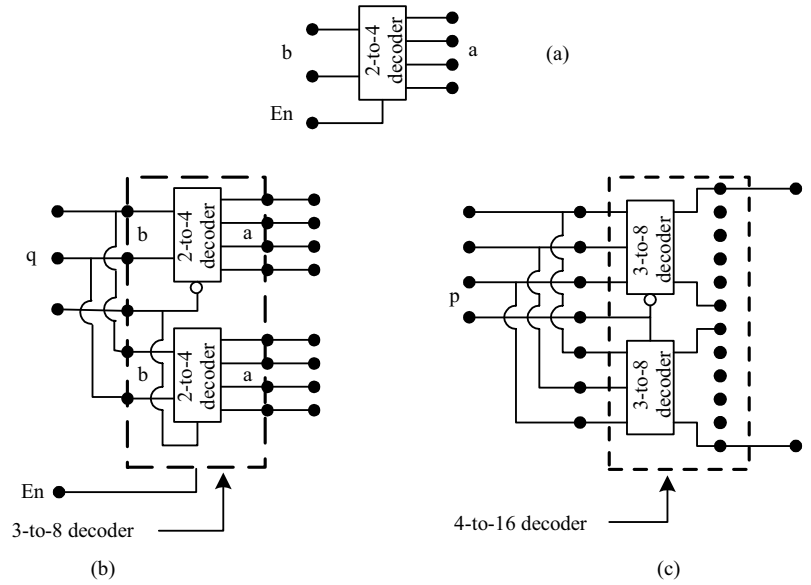


Figure 4.9 Formation of 4-to-16 decoder circuit in terms of smaller decoders: (a) 2-to-4 decoder, (b) 3-to- 8 decoder in terms of two 2-to-4 decoders, and (c) 4-to-16 decoder in terms of two 3-to-8 decoders.

```
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 4.10 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 4.11 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 4.12 A 3-to-8 decoder module formed by repeated instantiation of the 2-to-4 decoder 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
```

continued

continued

```
begin
  n=4'b0000;#2n=4'b0000;#2n=4'b0001;
#2n=4'b0010;#2n=4'b0011;#2n=4'b0100;
#2n=4'b0101;#2n=4'b0110;#2n=4'b0111;
#2n=4'b1000;#2n=4'b1001;#2n=4'b1010;
#2n=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 4.13 A 4-to-16 decoder module formed by repeated instantiation of the 3-to-8 decoder 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 ,
```

continued

continued

```
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 4.14 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 4.15.

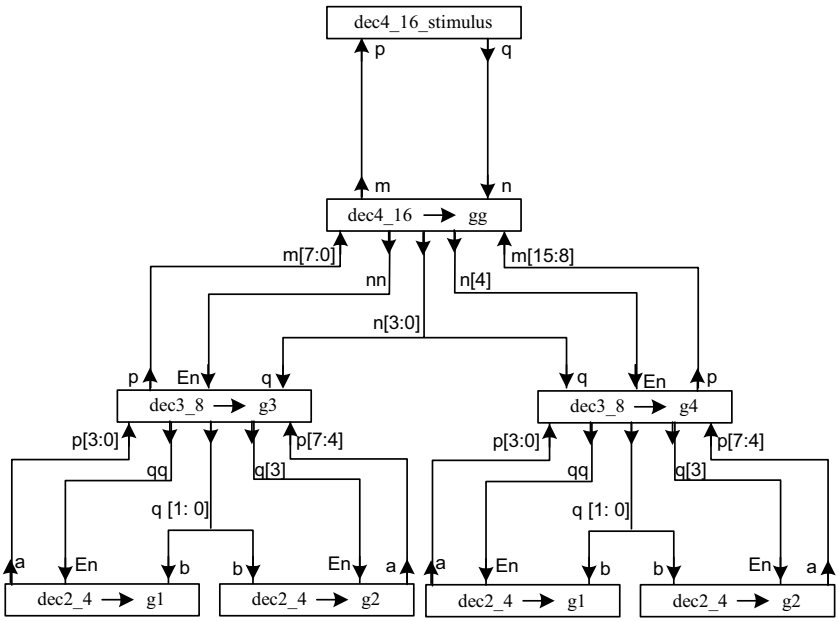


Figure 4.15 Block diagram representation of the module instantiations and signal assignments for the stimulus module of Figure 4.10.

- Two signals within the two nested modules are monitored in dec4_16_stimulus. Formation of their hierarchical addresses is also shown in Figure 4.15. (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.

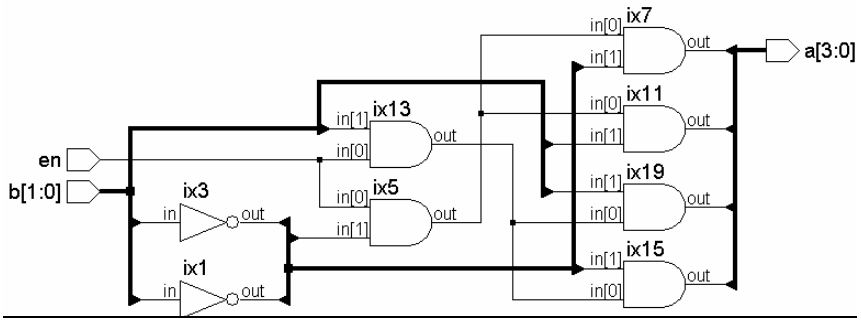


Figure 4. 16 The synthesized circuit of the 2-to-4 decoder of Figure 4.10.

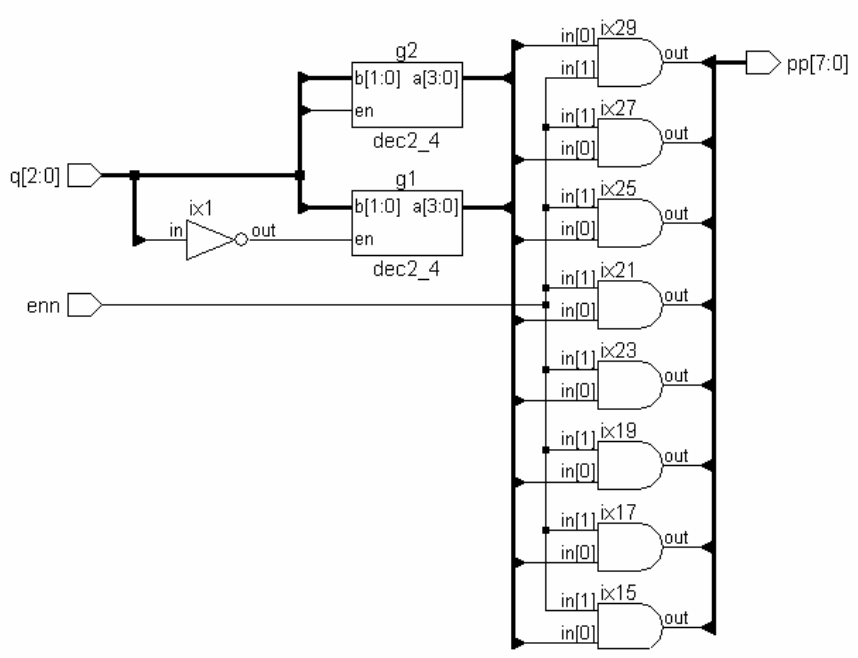


Figure 4.17 The synthesized circuit of the 3-to- 8 decoder of Figure 4.12.

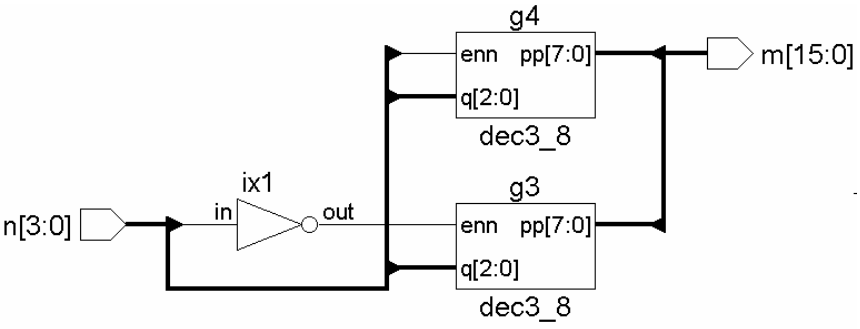


Figure 4.18 The synthesized circuit of the 4-to-16 decoder of Figure 4.13.

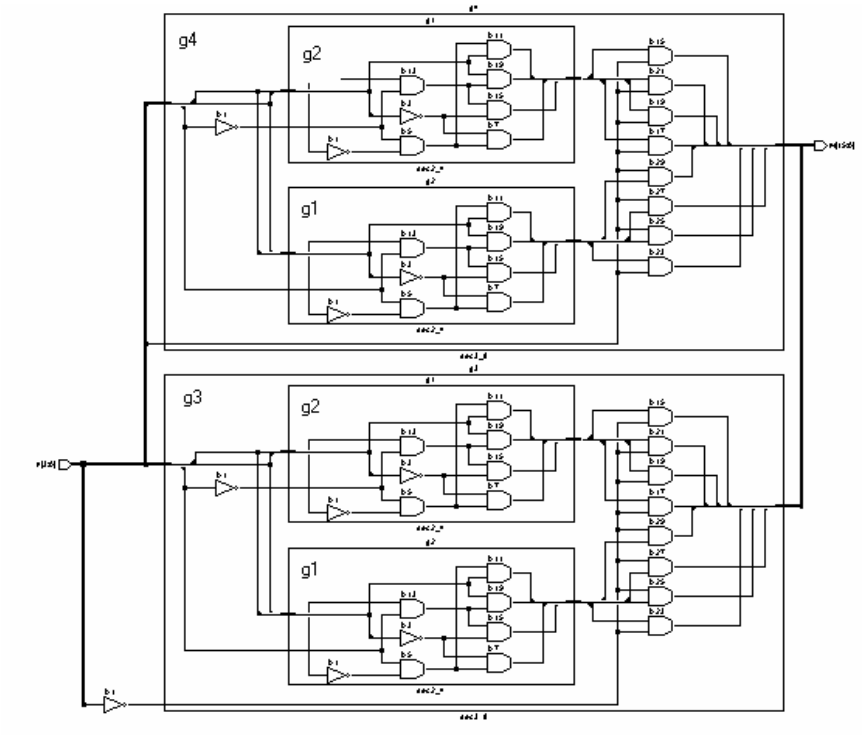


Figure 4.19 Four-to-sixteen decoder – hierarchy of instantiations.

4.6 TRI-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.

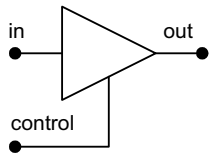
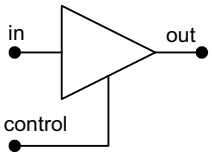
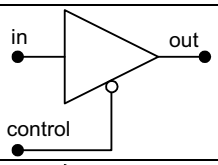
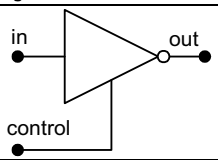
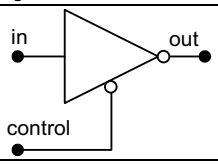


Figure 4.20 A tri-state buffer.

When
control = 0,
out is cut off from the input and tri-stated. The output, input and control signals should 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 4.4 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.

4.7 ARRAY 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.

4.7.1 Example 4.4 A Byte Comparator

A circuit to compare two variables each of one byte is given in Figure 4.21. 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 4.22 along with a test-bench. The simulated output is in Figure 4.23.

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.

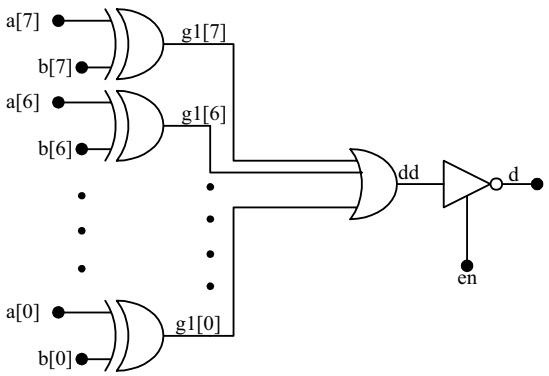


Figure 4.21 A byte comparator.

- 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 4.22 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 = 00000111, b = 00001100, d = 0
      #28 en = 1, a = 00000111, b = 00001110, d = 0
```

Figure 4.23 Results of the simulation run of the test bench in Figure 4.22.

4.8 ADDITIONAL 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 4.5

Table 4.5 Summary of the examples considered in Section 4.8

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 4.24 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 4.25 Results of running the test bench of the half-adder module in Figure 4.24.

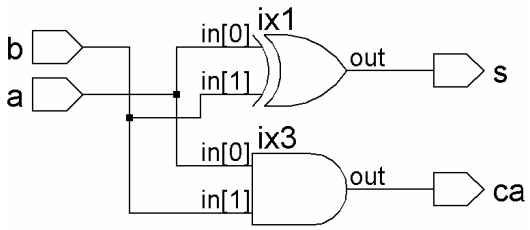


Figure 4.26 Synthesized output of the half-adder module of Figure 4.24.

```
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 4.27 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 = 0, cin = 1, 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 4.28 Results of running the test bench of the full-adder module in Figure 4.27.

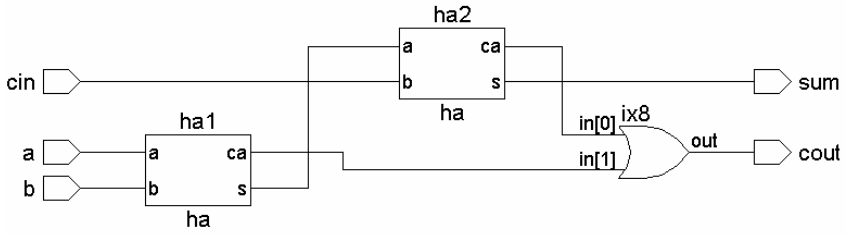


Figure 4.29 Synthesized output of the full-adder module of Figure 4.27.

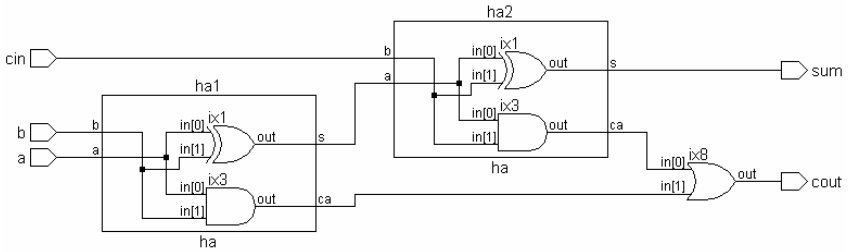


Figure 4.30 Synthesized circuit hierarchy of the full-adder module in Figure 4.27.

```
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 4.31 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 4.32 Results of running the test bench of the 4-to- mux module in Figure 4.31.

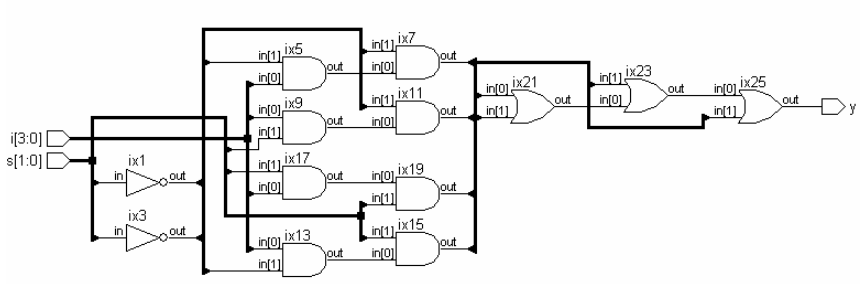


Figure 4.33 Synthesized output of the 4-to-1 Mux module of Figure 4.31.

```
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;
```

continued

continued

```
    #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 4.34 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 4.35 Results of running the test bench of the 4-to-1 mux module in Figure 4.34.

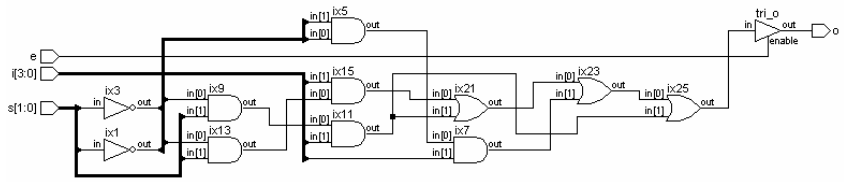


Figure 4.36 Synthesized output of the 4-to-1 mux module of Figure 4.34

```
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);
```

continued

continued

```
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 4.37 Design module and a test bench for a 2-to-1 mux module formed with tri-state buffers.

```
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 4.38 Results of running the test bench of the 2-to-1 mux module in Figure 4.37.

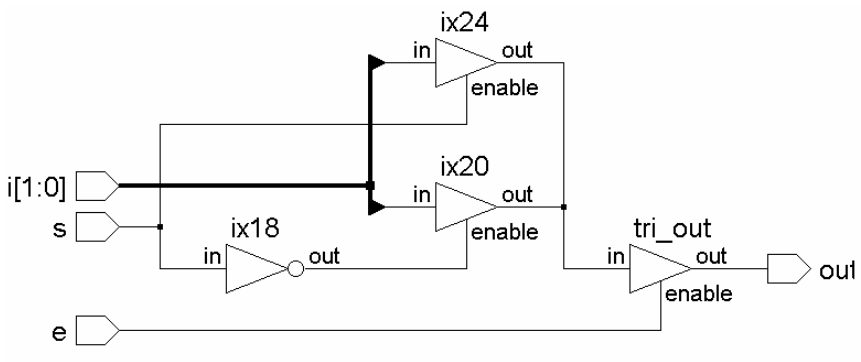


Figure 4.39 Synthesized output of the 2-to-1 mux module of Figure 4.37.

```
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
```

continued

continued

```
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 4.40 Design module and a test bench for a 4-to-1 mux module formed with tri-state buffers.

```
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 4.41 Results of running the test bench of the 4-to-1 mux module in Figure 4.40.

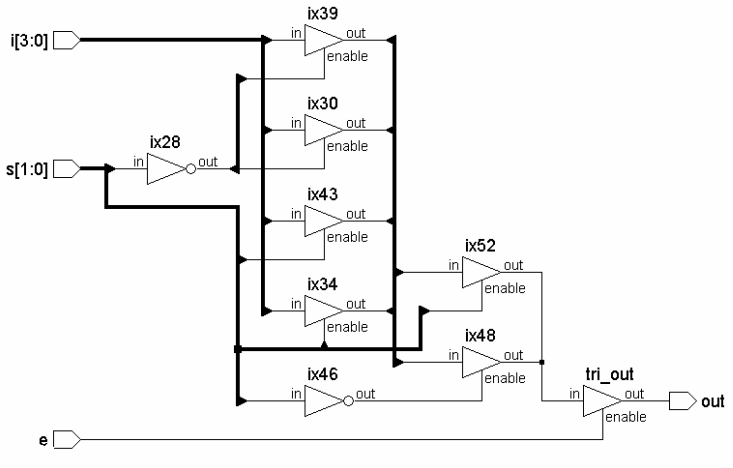


Figure 4.42 Synthesized output of the 4-to-1 mux module of Figure 4.40.

4.9 EXERCISES

1. Modify the test bench of Figure 4.1 and test the functionality of each of the basic gate primitives namely, OR, NOR, NAND, EXOR, EXNOR, NOT, and BUF.

For all the Exercises below prepare test benches and run the same.

2. Draw the half-adder circuit in terms of EX-OR gates and AND gate. Prepare a half-adder module in terms of EX-OR and AND gate primitive.
3. Prepare a full-adder module using half-adder module and OR gate Primitive.
4. Prepare a 4-bit adder module in terms of full-adder and half-adder modules. Treat the two 4-bit numbers as vectors for all input combination.
5. Prepare a module to generate a look-ahead-carry bit for the above problem.
6. Prepare modules for addition of 16 bit words and 32 bit words.
7. Prepare a module for conversion of an 8-bit number into its respective BCDs.
8. Prepare a module to add 2 BCDs
9. Prepare a module for the conversion of a pair of BCDs into the corresponding byte.
10. Prepare a module to generate Excess-3 code type of 4-bit output from a BCD.
11. Prepare a module to generate a BCD from an Excess-3 code digit.
12. Prepare an adder module to add Excess-3 coded digits.

13. Prepare a module to convert a set of 8 bits in gray code into an equivalent binary number.
14. Prepare an adder module to convert an 8-bit binary number into gray code.
15. Prepare a half-subtractor module and use it to form a 4-bit subtractor module.
16. Prepare a module to generate the 1's complement of a 4-bit number.
17. Prepare a module to generate 2's complement of a 4-bit number.
18. A set of 5-bit numbers is available as vectors – b [4:0]; b[4] is the sign bit. b [3:0] represent the number in 1's complement form. Prepare
    a) a module to add two such numbers
    b) a module to subtract one such number from the other
19. Repeat the above problem when the numbers are in 2's complement form.
20. Prepare a module to multiplex two input bits into one output bit.
21. Prepare a module to demultiplex one bit into 2 bits.
22. Use the 2 to 4 decoder module and prepare
    a) a 4 to 1 multiplexer module
    b) a 1 to 4 demultiplexer module
23. A is an 8-bit vector. Prepare a module to form another 8-bit vector B with its bits forming the mirror image of A.
24. A 16-bit barcode driver output is available. Generate the corresponding 4 bit output from these (Priority Encoder)
25. Prepare a module to generate 16-bit barcode driver outputs from a 4-bit binary number.
26. Prepare a module to generate 7-segment driver outputs from a 4-bit number.
27. Two 4-bit binary numbers *a* and *b* are available. Prepare a comparator module. The comparator module will generate 2 output bits. One bit is 0 if a > b and 1 if a < b. The second bit is 1 if a = b and 0 otherwise.
28. Prepare a 2-bit ALU module and its test bench. Let the module inputs – A and B – be 2-bit wide. D is the 2-bit output. Ci is the carry input and Co is the carry output. F is the function select vector. If F = 1, D = A + B; if F = 2, D = A + B + Ci; if F = 3, D =A - B; if F = 4, D = A – B - Ci; if F = 5, D = A OR B; if F = 6, D = A AND B; if F = 7, D = A XOR B.
29. Prepare a module for addition of bytes, instantiating the nibble adder of Exercise 4.4 repeatedly. Use the look-ahead-carry output of Exercise 4.5 to generate the carry bit from bit position 3 to bit position 4.
30. Use arrays of instances and redo the 4-to-16 decoder module of Figure 4.13.

5

GATE LEVEL MODELING – 2

5.1 INTRODUCTION

Design of combinational circuits was discussed in detail in the last chapter. Flip-flops too can be designed in a similar manner - that is, in terms of gate primitives. The same can be extended to registers, register files, memory, and so on. These can be combined with combinational circuits to form designs at the MSI level. Design of different types of flip-flops is discussed here through a series of examples. Subsequently, constructs available to account for different types of propagation delays are discussed. Constructs to represent source and load impedances and their use along with propagation delays are dealt with subsequently [IEEE].

5.2 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 5.1 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 step intervals. 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 5.3.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

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

Figure 5.1 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
         #  0 sb = 1 , rb = 0 , q = 0 , qb  = 1
         #  2 sb = 1 , rb = 1 , q = 0 , qb  = 1
         #  4 sb = 0 , rb = 1 , q = 1 , qb  = 0
         #  6 sb = 1 , rb = 1 , q = 1 , qb  = 0
         #  8 sb = 1 , rb = 0 , q = 0 , qb  = 1
         # 10 sb = 1 , rb = 1 , q = 0 , qb  = 1
         # 14 sb = 0 , rb = 1 , q = 1 , qb  = 0
         # 16 sb = 1 , rb = 1 , q = 1 , qb  = 0
         # 18 sb = 1 , rb = 0 , q = 0 , qb  = 1
```

Figure 5.2 A test bench for the flip-flop of Figure 5.1 and results of running the test bench.

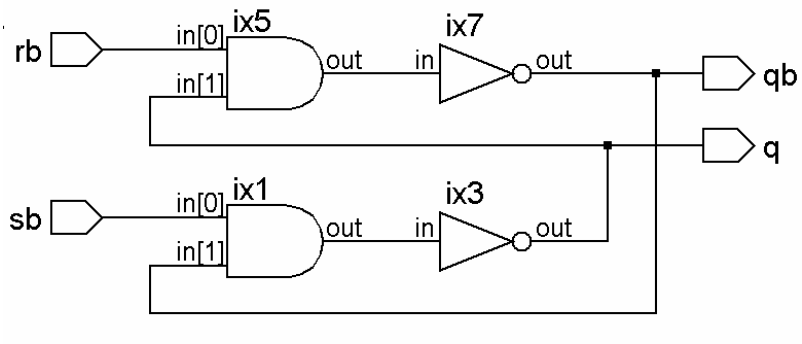


Figure 5.3 Synthesized circuit of the flip-flop module of Figure 5.1.

Example 5.2 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
```

continued

continued

```
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 5.5 Results of running the test bench for the flip-flop of Figure 5.4.

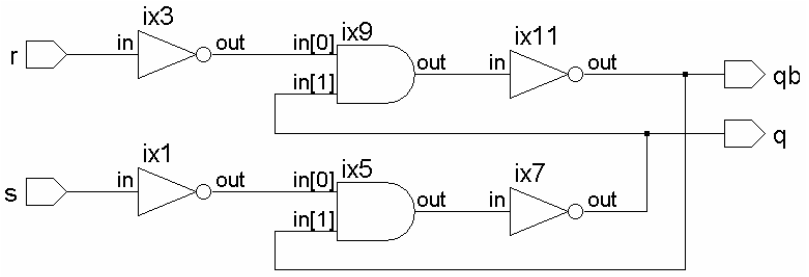


Figure 5.6 Synthesized circuit of the flip-flop module of Figure 5.4.

Example 5.3 A Clocked RS Flip-Flop

The module in Figure 5.7 is for a clocked RS flip-flop. It is the RS flip-flop of Figure 5.4 with the clock signal gating the R and S inputs. A test bench for the flip-flop is also shown in the figure. The clock waveform in the test bench is a square wave with a period of 4 ns [see Example 7.5 for details]. The simulation results are shown in Figure 5.8. Figure 5.9 shows the synthesized circuit of the flip-flop.

```
module srffcplev(cp,s,r,q,qb);
input cp,s,r;
output q,qb;
wire ss,rr;
nand(ss,s,cp),(rr,r,cp),(q,ss,qb),(qb,rr,q);
endmodule

module srffcplev_tst;// test-bench
reg cp,s,r;
wire q,qb;
srffcplev ff(cp,s,r,q,qb);
initial
begin
      cp=1'b0;
      s =1'b1;
      r =1'b0;
end
always #2cp=~cp;
always
begin
      #4 s =1'b0;r =1'b0;
      #4 s =1'b0;r =1'b1;
      #4 s =1'b0;r =1'b0;
      #4 s =1'b1;r =1'b0;
      #4 s =1'b0;r =1'b0;
end
initial $monitor($time,"cp = %b ,s = %b , r = %b , q =
%b , qb  = %b " ,cp,s,r,q,qb);
initial #20 $stop;
endmodule
```

Figure 5.7 Module of a clocked RS flip-flop with NAND gates and a test bench for the same.

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

Figure 5.8 Results of running the test bench for the flip-flop of Figure 5.7.

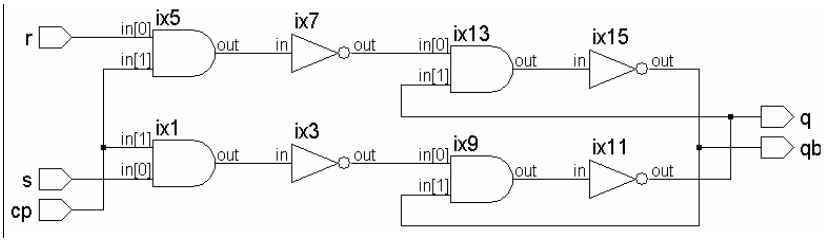


Figure 5.9 Synthesized circuit of the flip-flop module of Figure 5.7.

Example 5.4 A D-Latch

The design description of a D latch is given in Figure 5.10. It has one instantiation of the basic flip-flop of Figure 5.1. A test bench for the latch is also included in the figure. The simulation results are shown in Figure .5.11. Two versions of the synthesized circuit are shown in Figure 5.12 and Figure 5.13, respectively. The basic latch [sbrbff] — which was instantiated in the module of Figure 5.10 — is shown as a black box in Figure 5.12. The internals of the latch are shown in Figure 5.13, which brings out the hierarchy clearly.

```
module dlatch(en,d,q,qb);
input d,en;
output q,qb;
wire dd;
wire s,r;
not n1(dd,d);
nand (sb,d,en);
nand g2(rb,dd,en);
```

continued

continued

```
sbrbff ff(sb,rb,q,qb);//Instantiation of the sbrbff
endmodule

module tstdlatch; //test-bench
reg d,en;
wire q,qb;
dlatch ff(en,d,q,qb);
initial
begin
      d  = 1'b0;
      en = 1'b0;
end
always #4 en =~en;
always #8 d=~d;
initial $monitor($time," en = %b , d = %b , q = %b , qb
= %b " , en,d,q,qb);
initial #40 $stop;
endmodule
```

Figure 5.10 Module of a D latch and a test bench for the same.

```
#  0 en = 0, d = 0, q = x, qb  = x
#  4 en = 1, d = 0, q = 0, qb  = 1
#  8 en = 0, d = 1, q = 0, qb  = 1
# 12 en = 1, d = 1, q = 1, qb  = 0
# 16 en = 0, d = 0, q = 1, qb  = 0
# 20 en = 1, d = 0, q = 0, qb  = 1
# 24 en = 0, d = 1, q = 0, qb  = 1
# 28 en = 1, d = 1, q = 1, qb  = 0
# 32 en = 0, d = 0, q = 1, qb  = 0
# 36 en = 1, d = 0, q = 0, qb  = 1
```

Figure 5.11 Results of running the test bench for the D latch of Figure 5.10.

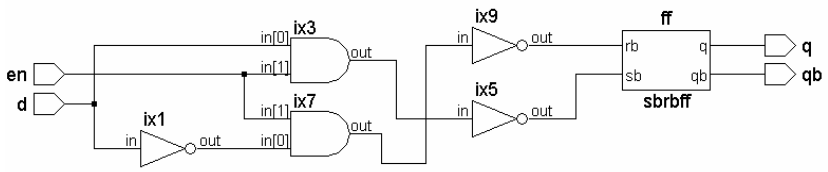


Figure 5.12 Synthesized circuit of the D latch module of Figure 5.10.

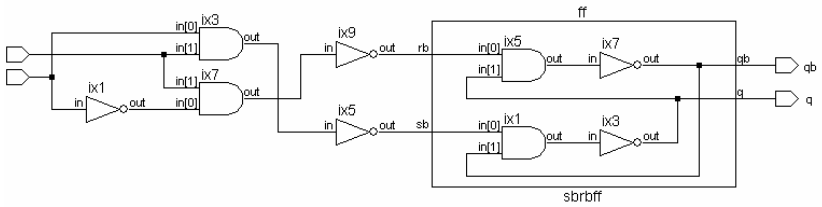


Figure 5.13 Synthesized circuit of the D latch module of Figure 5.10 showing hierarchy.

Example 5.5 An Edge-Triggered Flip-Flop

Figure 5.14 shows the circuit of an edge-triggered flip-flop. It is a simplified version of the 7474 IC. The circuit is a combination of three latches – designated as FF1, FF2, and FF3 in the figure. FF3 is similar to the latch considered in Example 5.1. FF1 and FF2 are minor modifications of FF3. The design modules for FF1 and FF2 are given in Figure 5.15. All three latches are instantiated to form the edge-triggered flip-flop. A test bench for the flip-flop is also included in the figure. With a square waveform for the clock – cp – the waveform for the d input is chosen to bring out the edge-triggered nature of operation of the flip-flop. The output obtained by running the test bench is shown in Figure 5.16; the respective waveforms are shown in Figure 5.17. One can see that the output changes only at the positive edges of the clock, and it assumes the value of the input at that instant of time.

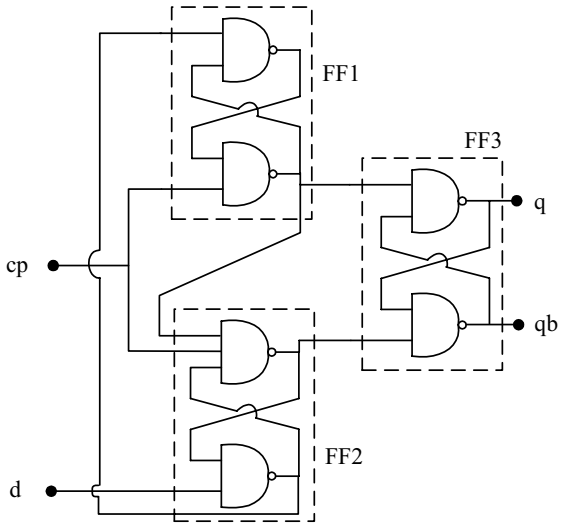


Figure 5.14 Circuit of a skeletal edge-triggered flip-flop.

```
module dffgatnew1(cp,d,q,qb);
input d,cp;
output q,qb;
wire sb,rb;
wire s,r;
sbrbffdff ff1(rb,cp,s);
sbrbff1 ff2(s,d,cp,r,rb);
sbrbff ff3(s,r,q,qb);
endmodule

module tst_dffgatnew1; //test-bench
reg d,cp;
wire q,qb;
dffgatnew1 ff(cp,d,q,qb);
initial
begin
     d =1'b0;cp =1'b0;
     #2 cp =1'b1;#2 cp =1'b0;#2 cp =1'b1;#2 cp =1'b0;
     #2 cp =1'b1;#2 cp =1'b0;#2 cp =1'b1;#2 cp =1'b0;
end
initial
begin
     #3 d=1'b1;#2d=1'b1;#2d=1'b0;#3d=1'b0;#3d=1'b1;
end
initial $monitor($time," cp = %b , d = %b , q = %b , qb
= %b " , cp,d,q,qb);
initial #40 $stop;
endmodule

module sbrbffdff(sb,rb,qb);
input sb,rb;
output qb;
wire q;
nand(q,sb,qb);
nand(qb,rb,q);
endmodule

module sbrbff1(sb,rb,cp,q,qb); //test-bench
input sb,rb,cp;
output q,qb;
nand(q,sb,cp,qb);
nand(qb,rb,q);
endmodule
```

Figure 5.15 Module of a positive edge-triggered flip-flop and its test bench.

```
#   0 cp = 0 , d = 0 , q = x , qb  = x
#   2 cp = 1 , d = 0 , q = 0 , qb  = 1
#   3 cp = 1 , d = 1 , q = 0 , qb  = 1
#   4 cp = 0 , d = 1 , q = 0 , qb  = 1
#   6 cp = 1 , d = 1 , q = 1 , qb  = 0
#   7 cp = 1 , d = 0 , q = 1 , qb  = 0
#   8 cp = 0 , d = 0 , q = 1 , qb  = 0
#  10 cp = 1 , d = 0 , q = 0 , qb  = 1
#  12 cp = 0 , d = 0 , q = 0 , qb  = 1
#  13 cp = 0 , d = 1 , q = 0 , qb  = 1
#  14 cp = 1 , d = 1 , q = 1 , qb  = 0
#  16 cp = 0 , d = 1 , q = 1 , qb  = 0
```

Figure 5.16 Results of running the test bench for the flip-flop of Figure 5.15.

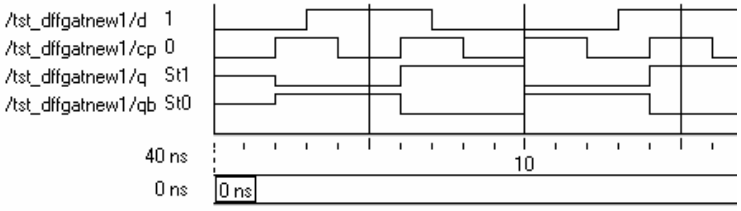


Figure 5.17 Clock (cp), data input (d), and output waveforms for the edge-triggered flip-flop with the test bench in Figure 5.15.

Synthesized circuits of the latches FF1 (sbrbffdff) and FF2 (sbrbff1) are shown in Figure 5.18 and Figure 5.19, respectively. The synthesized circuit for the overall flip-flop is shown in Figure 5.20. FF1, FF2, and FF3 are represented as boxes there; only their interconnections are shown. The comprehensive circuit in terms of the elementary gates is not shown.

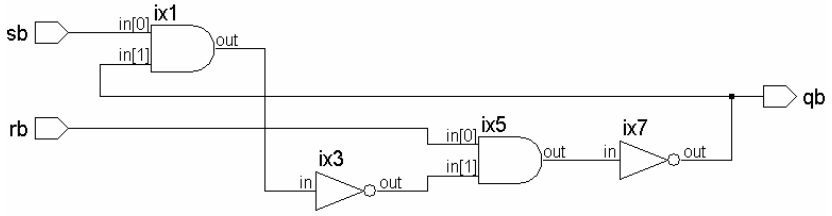


Figure 5.18 Synthesized circuit of the flip-flop sbrbffdff of Figure 5.15.

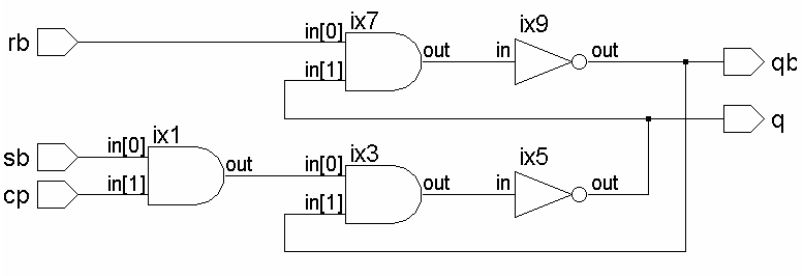


Figure 5.19 Synthesized circuit of the flip-flop sbrbff1 of Figure 5.15.

The flip-flop of Figure 5.14 can be made comprehensive with slight modifications. It can be replicated and with suitable additions, expanded substantially into register files and full-fledged memory [see the Exercises at the end of the chapter].

5.3 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].

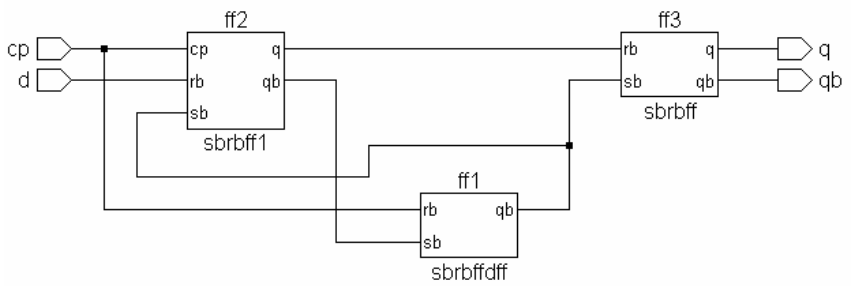


Figure 5.20 Synthesized circuit of the flip-flop dffgatnew1 in Figure 5.15.

5.3.1 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; // nn is 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

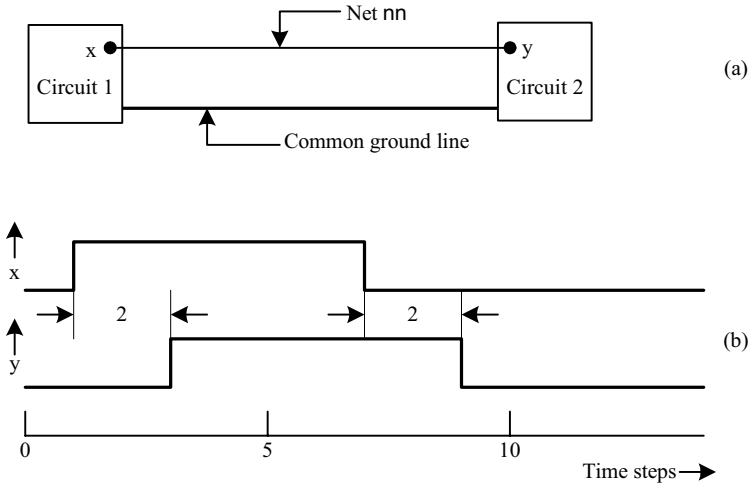


Figure 5.21 A net connecting two circuit blocks and the delay through it: (a) Connection diagram (b) Typical signal waveforms at the input and output ends of the net.

```
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 5.22 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.

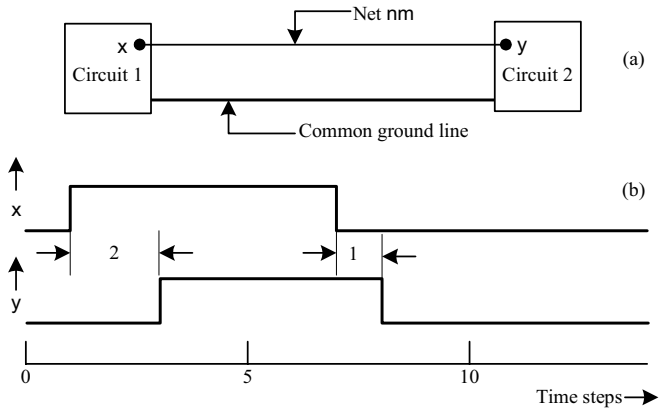


Figure 5.23 A net connecting two circuit blocks and the delays through it: (a) Connection diagram (b) Typical signal waveforms at the input and output ends of the net.

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

5.3.2 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);
```

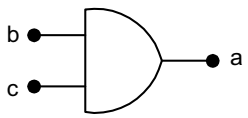
The above represents an AND gate description with a uniform delay of 3 ns for all transitions from input to output. A more detailed description can be as follows:

```
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 5.25 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 5.25 has different delays for the output transitions; respective waveforms are shown in Figure 5.26.

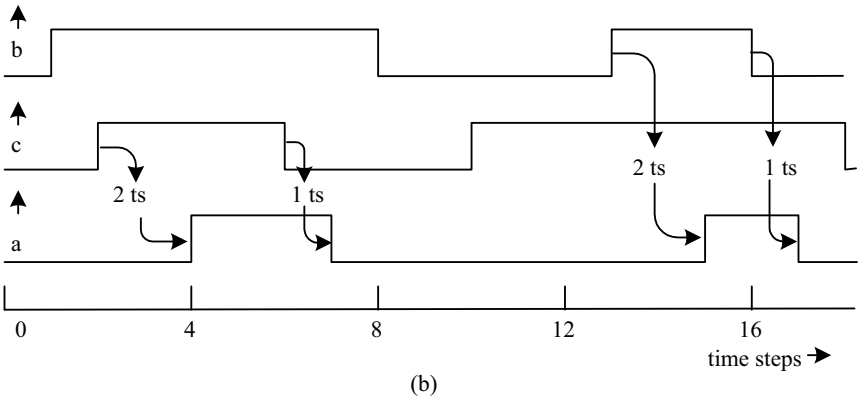
```
module gade(a,a1,b,c,b1,c1);
input b,c,b1,c1;
output a,a1;
or #3gg1(a1,c1,b1);
and #(2,1)gg2(a,c,b);
endmodule

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 5.26 AND gate instantiation with different delays for the positive and negative transitions and associated waveforms: (a) Gate instantiated.



(b)

Figure 5.26 (cont'd**) (b)** associated waveforms (time step has been abbreviated to "ts" in the 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 5.27 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 5.28(a) A design having eight different time delay values.

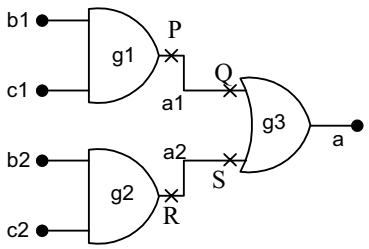
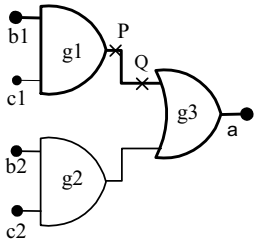
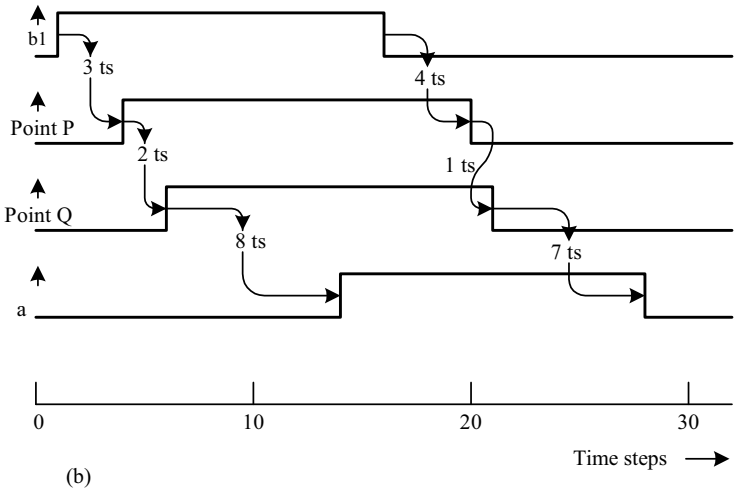


Figure 5.28(b) The circuit for the module considered in Figure 5.28(a).



(a)



(b)

Figure 5.29 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).

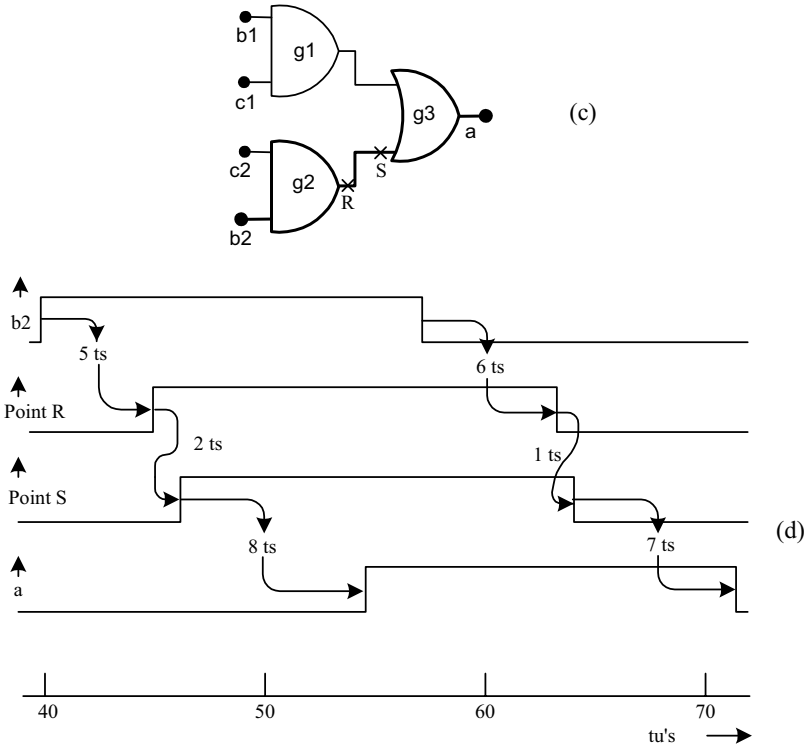


Figure 5.29 (cont'd) (c) The circuit portion active during changes to signal b2. (d) Signal waveforms following changes to signal b2 (time step has been abbreviated as ts in tbe diagrams).

5.3.3 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).

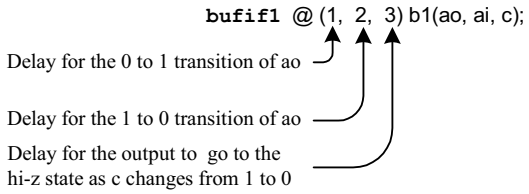


Figure 5.30 Delays associated with a typical tri-state gate.

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, typical and 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 have their 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.

- The cumulative delay for a signal in a path puts an upper limit on the maximum operating frequency *vis-à-vis* the signal.
- A signal may go through multiple paths in a design to arrive at one gate. It is necessary to match the delays within specified tolerances for reliable operation of the device.
- In larger designs, one has to identify the longest signal path (critical path). This puts an upper limit on the operating frequency apart from causing mal-operation in a worst-case scenario. One of the practices in design is to re-route selected signals or redo selected design segments to reduce critical path delays.

5.3.4 General Definitions for Delays

Specific numerical values have been used for all the delays in the examples so far. However, Verilog LRM allows constant expressions to be used for any of the delay values. The expressions used may involve simple algebra in terms of integers and known quantities (but not variables).

5.4 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.

5.4.1 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 5.1 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	

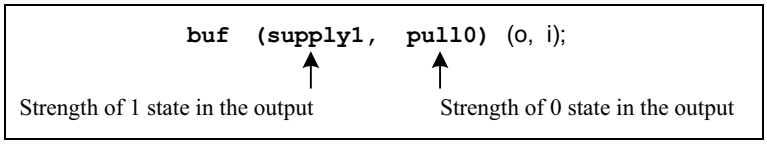


Figure 5.31 Format for specifying strengths in the instantiation of a gate primitive.

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.

5.4.2 Strength Contention in Gate Primitives

When two signals of opposite polarity and differing strengths drive a line, the output status is decided by the stronger signal. However, if the signals are of equal strength, the output is indeterminate. Different contention possibilities arise here. The variety is brought out through examples.

Example 5.6 Strength Contention

Consider the module in Figure 5.32. The logic levels taken by the signal `o` for different combinations of inputs to the two buffers `g1` and `g2` are shown in Table 5.2. Contentions of signals with other combinations of levels can be resolved in the same manner.

Table 5.2 Outputs for different inputs for the example of Figure 5.32

Logic value of input i1	Logic value of input i2	Logic value of output o	Remarks
0	0	0	No contention
0	1	1	Contention; the stronger signal – i2 – prevails
1	0	1	Contention; the stronger signal – i1 – prevails
1	1	1	No contention

```
module contres(o,i1,i2);
input i1,i2;
output o;
buf(supply1,pull0)g1(o,i1), g2(o,i2);//note that the
endmodule// same net is driven by both the gates.

module tst_contres; //TEST BENCH
reg i1,i2;
contres cc(o,i1,i2);
initial
begin
      i1 =0;
      i2 =0;
end   //no contention
always
begin
     #4 i1 =0; i2 = 1;// contention; the stronger
     #4 i1 =1; i2 = 0;// signal prevails.
     #4 i1 =1; i2 = 1;//no contention.
end
initial $monitor($time,"i1=%b,i2=%b,o=%b",i1,i2,o);
initial #40$stop;
endmodule
```

```
output
    #                    0 i1 = 0 , i2 = 0 , o = 0
    #                    4 i1 = 0 , i2 = 1 , o = 1
    #                    8 i1 = 1 , i2 = 0 , o = 1
    #                   12 i1 = 1 , i2 = 1 , o = 1
    #                   16 i1 = 0 , i2 = 1 , o = 1
    #                   20 i1 = 1 , i2 = 0 , o = 1
    #                   24 i1 = 1 , i2 = 1 , o = 1
    #                   28 i1 = 0 , i2 = 1 , o = 1
    #                   32 i1 = 1 , i2 = 0 , o = 1
    #                   36 i1 = 1 , i2 = 1 , o = 1
    #                   40 i1 = 0 , i2 = 1 , o = 1
```

Figure 5.32 A module to illustrate strength contention; the test bench and simulation results are also shown in the figure.

The outputs for the four input combinations are given in the table. Whenever there is a contention, the logic value of the output is decided by the stronger signal. In fact the design description here realizes an OR gate at the output side without additional hardware. It does not lead to any ambiguity.

Consider the Example in Figure 5.33, which is a slightly modified version of that in Figure 5.32. The output logic values for different input combinations are given in Table 5.3. The gate outputs are decided by following the same logic as in the last case. However, in one case — when both gates "drag" the output with equal strength in opposite directions — the output logic level is indeterminate — that is, **x**.

```
module contres1(o,i1,i2);
input i1,i2;
output o;
buf(strong1 ,pull0)g1(o,i1); buf(pull1,pull0)g2(o,i2);
endmodule

module tst contres1; //TEST BENCH
reg i1,i2;
contres1 cc(o,i1,i2);
initial
begin
i1 =0;i2 =0;end    //no contention
always
begin
#4 i1 = 0; i2 = 1; //contention between pull0 due to
//i1 and pull1 due to i2; output is x
#4 i1 =1; i2 =0;  //contention; output is 1 since
//strong1 of i1 prevails.
#4 i1 =1 ;i2 = 1; //no contention.
end
initial $monitor($time  ," i1 = %b , i2 = %b ,o = %b "
,i1,i2,o);
initial #40 $stop;
endmodule
```

```
output
     #                    0 i1 = 0, i2= 0 ,o = 0
     #                    4 i1 = 0, i2= 1 ,o = x
     #                    8 i1 = 1, i2= 0 ,o = 1
     #                   12 i1 = 1, i2= 1 ,o = 1
     #                   16 i1 = 0, i2= 1 ,o = x
     #                   20 i1 = 1, i2= 0 ,o = 1
     #                   24 i1 = 1, i2= 1 ,o = 1
     #                   28 i1 = 0, i2= 1 ,o = x
     #                   32 i1 = 1, i2= 0 ,o = 1
     #                   36 i1 = 1, i2= 1 ,o = 1
```

Figure 5.33 Illustration of strength contention resulting in **x**-type output; the test bench and simulation results are also shown in the figure.

Table 5.3 Outputs for different inputs in the example of Figure 5.33

Logic value of input i1	Logic value of input i2	Logic value of output o	Remarks
0	0	0	No contention
0	1	x	Contention; both signals being of equal strength, the output is indeterminate
1	0	1	Contention; the stronger signal i1 prevails and forces the output to logic state 1
1	1	1	No contention

5.4.3 Net Charges

Whenever a net is driven by a signal, it takes the logic value of the signal. When the signal source is tri-stated, the net too gets tri-stated. In practice the net can have a capacitor associated with it, which can store the signal level even after the signal source dries up (*i.e*., tri-stated). To account for this situation, a charge storage capacity is associated with the net. Such nets are declared with the keyword **`trireg`**. By virtue of the inherent capacitance associated with them, trireg nets can never be in the high impedance state – that is, they can assume 0, 1, or **`x`** value only. A **`trireg`** net can be in one of two possible states only:

- *Driven state*: When driven by a source or multiple sources, the net assumes the strength of the source. It can be any of the strengths specified in Table 5.1 except the high impedance value.
- *Capacitive state*: When the driven source (sources) is (are) tri-stated, the net retains the last value it was in – by virtue of the capacitance associated with it. The value can be 0, 1 or **`x`** (but not the high impedance value).

When in the capacitive state, a net can have a storage strength associated with it. Three such storage strengths are possible – namely **`large`**, **`medium`**, and **`small`**. Their details are shown in Table 5.4. When a storage strength is not specified, it is assigned the default value – **`medium`**. For a **`trireg`** net one cannot assign storage strength capacity separately for the 0 and the 1 states.

A **`trireg`** net can be driven with possibilities of contention from two or more sources; such cases are considered in Chapter 10.

Table 5.4 Capacitive storage strengths on nets

Name	**`large`**	**`medium`**	**`small`**
Strength	Strongest		Weakest

Example 5.7 Net Storage

Consider the design in Figure 5.34. As long as the signal control = 1, the signal out follows the signal in. When control goes to 0, out is disconnected from the input and it ''floats.'' It retains the last value due to the capacitance storage capacity. The storage strength is **`medium`**, signifying a medium value of capacitance.

```
module charge(out,in,control);
output out;
trireg(medium)out;
input in,control;
bufif1 g1(out,in,control);
endmodule

module tst_charge; //TESTBENCH
reg in, control;
charge c1(out,in,control);
initial
    begin
    in =0;control =0;//when control=0 output is x
    #2 control =0;in =0;
    #2 control =1;in =0;
    #2 control =1;in =1;
    #2 control =0;in =0; // output is retained at
    end // the last value
initial $monitor($time ," in= %b ,control = %b , out=
%b " ,in,control,out);
initial #40$stop;
endmodule
```

```
output
         #     0 in = 0 , control = x , out=x
         #     2 in = 0 , control = 0 , out=x
         #     4 in = 0 , control = 1 , out=0
         #     6 in = 1 , control = 1 , out=1
         #     8 in = 0 , control = 0 , out=1
```

Figure 5.34 Illustration of net storage; the test bench and simulation results are also shown in the figure.

5.4.4 Contention Between Net and Gate Primitive Outputs

In case of a contention between a signal output from a gate and the charge on a net, the contention is decided by the relative strengths of the signals on each. Table 5.5 combines all the strengths – those of the gate outputs as well as those of tri-stated nets and – lists them in the order of their relative strengths. The abbreviations associated with the strengths are not repeated here.

5.4.5 Net Types and Port Assignments

All input ports of modules have to accept inputs from outside when instantiated and respond to changes in them. Hence they have to be of net type. Note that input ports cannot be of reg type since their values cannot be changed from outside. The output ports of instantiated modules can be of net or reg types. **`Inout`** ports have to function as input or output ports; hence they too have to be of net types.

The port assignments in an instantiation can be to scalars, vectors, part vectors, or concatenated vectors. However, their sizes should match those of the ports in the module definitions. Further, the type restrictions mentioned above have to be complied with.

In many situations the net types in the module definition and its instantiation may differ in the case of input and **`inout`** ports. In such cases the resulting net type can be of only one type. Either the net type declared in the module definition or that in the instantiation (external type) dominates. The choice is decided by a specific protocol in the LRM. Table 5.6 gives details. As can be seen from the table, whenever the two net types lead to a logical clash, the external data type prevails (identified by an asterisk in the table).

Table 5.5 Signal strength names and their relative weights

Signal strength name	Strength level
Supply (drive)	Strongest 7
Strong (drive)	6
Pull (drive)	5
Large (capacitance)	4
Weak (drive)	3
Medium (capacitance)	2
Small (capacitance)	Weakest 1
High impedance	0

Table 5.6 Net assignments with port connections

Internal net	External net							
	Wire & tri	Wand & triand	Wor & trior	Trireg	Tri0	Tri1	Su0	Su1
Wire & tri	E	E	E	E	E	E	E	E
Wand & triand	I	E	*	*	*	*	E	E
Wor & trior	I	I	E	*	*	*	E	E
Trireg	I	I	*	E	E	E	E	E
Tri0	I	I	*	I	E	*	E	E
Tri1	I	I	*	I	*	E	E	E
Su0	I	I	I	I	I	I	E	*
Su1	I	I	I	I	I	I	*	E

Notes "E" signifies that the external net prevails, and "I" that the internal net prevails.
"*" signifies a logical clash; the external net prevails.

5.5 NET TYPES

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

5.5.1 `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 5.35 A design module to illustrate use of the wand-type net; a test bench and the results of simulation are also shown.

Table 5.7 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.

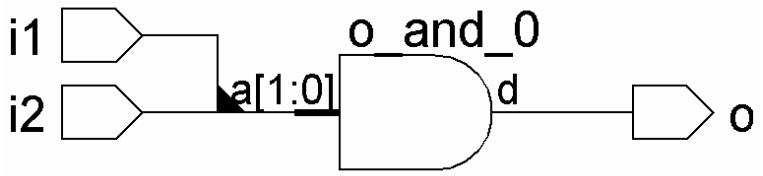


Figure 5.36 Synthesized version of the module with the wand-type net in Figure 5.35 above.

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

Table 5.8 Output values for different inputs of the design in Figure 5.37

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

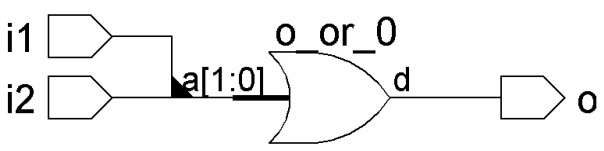


Figure 5.38 Synthesized version of the module with the w**or**-type net in Figure 5.37.

One can see that **wand** and **wor** are keywords to implement wired-or type logic. Observations:

- Many synthesizers do not support wired-or logic. **wand** and **wor** may be used to advantage when supported by the synthesizer.
- The net **triand** is functionally identical to the net **wireand**. Similarly, the net **trior** is functionally identical to the net **wireor**.
- All synthesizers support **wire**. **Triand**, **trior**, **tri0**, and **tri1** (discussed below) may not be supported by some.

5.5.2 Tri

The keyword **tri** has a function identical to that of **wire**. When a net is driven by more than one tri-state gate, it is declared as **tri** rather than as **wire**. The distinction is for better clarity. Similarly, **Triand** and **trior** are the counterparts of **wand** and **wor,** respectively.

Example 5.10 Illustration of tri-type net

Consider the design segment in Figure 5.39. Here the signal on net out is controlled by the control signal En. If En = 1, signal a is steered to the net out and the output of gate g2 is tri-stated. On the other hand, if En = 0, signal b is steered to the net out and the gate g1 is tri-stated. If the buffers are controlled by independent Enable signals, the output is resolved according to the respective strengths.

```
. . .
tri out;
wire a, b, En;
bufif1 g1(out, a, En);
bufif0 g2(out, b, En);
. . .
```

Figure 5.39 A segment of a design to illustrate **tri** type of net.

5.5.3 `Tri0` and `tri1`

If the output of a tri-state buffer is to be pulled up to the 1 state when tri-stated, it is declared as net **`tri1`**. Similarly, it is declared as **`tri0`** if it is to be pulled down to 0 state when tri-stated. **`Tri0`** and **`tri1`** provide respective default outputs and avoid any following circuit having a tri-stated input. In turn, it may manifest as an added load at the concerned gate output. The example in Figure 5.40, which shows a design segment, illustrates an application. Table 5.9 lists the output values of signals considered in the design segment of Figure 5.40.

Referring to the figure (and the table), one can see that when En = 0, all three buffers g0, g1, and g2 are off. Net o3, being a wire is tri-stated and is in **`z`** state. However, net o1, being of **`tri0`** type, is pulled down to 0 state irrespective of the input value. Net 02, being of **`tri1`** type, is pulled up to 1 state. When En = 1, all three buffers are ON and the respective outputs follow the input. Thus though g0, g1, and g2 are functionally identical, they behave differently due to the difference in the type of the respective output nets.

Reset, *Chip Enable* and similar signals can be pulled up or down as required with **`tri0`** or **`tri1`**; this signifies the normal status –that is, the chip is disabled or the reset is disabled. As and when the chip is to be enabled, the same is done by enabling the buffer for the required period. Similarly, the *reset* can be activated for a specified period to reset the chip; subsequently, the reset can be deactivated to restore normal operation of the chip.

```
. . .
tri0 o1;
tri1 o2;
wire o3;
bufif1 g0 (o1, I, En), g3 (o2, I, En);
buif1 g1(o3, I, En);
. . .
```

Figure 5.40 A segment of a design to illustrate **`tri0`** and **`tri1`** types of net.

Table 5.9 Output values for different inputs of the segment in Figure 5.40

Logic value of I	Logic value of En	Logic value of o1	Logic value of o2	Logic value of o3
0	0	0	1	Z
0	1	0	0	0
1	0	0	1	Z
1	1	1	1	1

5.5.4 supply0 and supply1

supply0 and **supply1** are the keywords signifying the high- and low-side supplies. Nets to be connected to the Vcc supply are declared as **supply1**, and those to be grounded are declared as **supply0**. Their use is illustrated in Chapter 10.

5.5.5 Ambiguous Strengths

Certain **x** or **z** type of input port values of gate primitives can lead to outputs of apparently ambiguous strengths. A number of such situations can arise. Such cases are brought out and illustrated in the LRM. Nevertheless, such ambiguous situations may be avoided in practice.

5.5.6 Combining Delays & Strengths

So far we have discussed incorporation of strengths in net declarations and instantiations of primitives. Incorporation of a variety of delays and specifying tolerances on them were dealt with in the previous sections. One can combine delays and strengths in net declarations as well as in instantiation of gate primitives. The formats for the same are illustrated below

Wire (drive_strength_1, drive_strength_0) # (delay_0_1, delay_1_0, turn_off_delay) signal1, signal2;

Gate_type (drive_strength_1, drive_strength_0) # (delay_0_1, delay_1_0, turn_off_delay) instance_1(signal1, signal2);

For each of the delays above, one can also specify the minimum, typical, and maximum values. Such values can be specified in terms of constant expressions also. All these have been dealt with separately in detail earlier. Hence combining them and illustrating through examples is not done again here.

5.6 DESIGN OF BASIC CIRCUITS

Elementary gates are the basic building blocks of all digital circuits – whether combinational, sequential, or involved versions combining both. Conversely, any digital circuit can be split up into constituent elementary gates. The variety of examples of combinational circuits considered in the last chapter, and the sequential circuit examples at the beginning of this chapter are testimony to this. Any digital circuit however involved it may be, can be realized in terms of gate primitives. The step-by-step procedure to be adopted may be summarized as follows:

1. Draw the circuit in terms of the gates.
2. Name gates and signals.
3. Using the same nomenclature as above, do the design description.
4. As the functional blocks like encoder, decoder, half-adder, full-adder, *etc*., get more and more involved, treat each as a building block with corresponding inputs and outputs.
5. Make more involved circuits in terms of the building blocks – as far as possible. Each block within another block manifests as an instantiation of one module within another.

Example 5.11 ALU

We consider the design of an ALU as an example of a relatively complex design. The ALU considered carries out four functions:

- Addition of two 4-bit numbers.
- Complementing all the bits of a 4-bit vector.
- Bit-by-bit AND operation on two nibbles.
- Bit-by-bit XOR operation on two nibbles.

A set of 2 mode select bits selects the function to be carried out from amongst the above four. The design has been evolved in a step-by-step manner. Figure 5.41 shows a 4-bit adder module and a test-bench for it. The simulation results are given in Figure 5.42. The adder module is built up by repeated instantiation of the full-adder module considered in Section 4.8. The synthesized version of the adder is shown in Figure 5.43. The full-adder module instantiations appear here as black boxes with respective inputs and outputs.

```
module add4g(sum,carry,a,b,cin);
input[3:0]a,b;
input cin;
output[3:0]sum;
output carry;
wire [2:0]cc;
fa a0(sum[0],cc[0],a[0],b[0],cin);
fa a1(sum[1],cc[1],a[1],b[1],cc[0]);
fa a2(sum[2],cc[2],a[2],b[2],cc[1]);
fa a3(sum[3],carry,a[3],b[3],cc[2]);
endmodule

module tstadd4g; //Test bench
reg[3:0]a,b;
reg cin;
wire[3:0]sum;
```

continued

continued

```
wire carry;
add4g gg(sum,carry,a,b,cin);
initial
begin
      a =4'h0;b=4'h0;cin=0;
end
always
begin
     #2 a=4'h0;b=4'h0;cin=1'b0;
     #2 a=4'h1;b=4'h0;cin=1'b1;
     #2 a=4'h1;b=4'h0;cin=1'b1;
     #2 a=4'h5;b=4'h3;cin=1'b0;
     #2 a=4'h7;b=4'h0;cin=1'b1;
     #2 a=4'h8;b=4'h9;cin=1'b1;
     #2 a=4'h0;b=4'h0;cin=1'b0;
     #2 a=4'hb;b=4'h7;cin=1'b0;
     #2 a=4'h0;b=4'h0;cin=1'b0;
     #2 a=4'hf;b=4'hf;cin=1'b0;
     #2 a=4'hf;b=4'hf;cin=1'b1;
end
initial $monitor($time," a = %b, b = %b, cin = %b,
outsum = %b, outcar = %b ", a, b, cin, sum, carry);
initial #30 $stop ;
endmodule
```

Figure 5.41 A 4-bit adder module and its test bench

```
output
# 0 a =0000,b =0000,cin = 0,outsum =0000,outcar =0
# 2 a =0001,b =0000,cin = 0,outsum =0001,outcar =0
# 4 a =0001,b =0000,cin = 1,outsum =0010,outcar =0
# 6 a =0001,b =0001,cin = 1,outsum =0011,outcar =0
# 8 a =0101,b =0011,cin = 0,outsum =1000,outcar =0
#10 a =0111,b =0110,cin = 1,outsum =1110,outcar =0
#12 a =1000,b =1001,cin = 1,outsum =0010,outcar =1
#14 a =1010,b =0001,cin = 1,outsum =1100,outcar =0
#16 a =1011,b =0111,cin = 0,outsum =0010,outcar =1
#18 a =1000,b =1000,cin = 0,outsum =0000,outcar =1
#20 a =1111,b =1111,cin = 0,outsum =1110,outcar =1
#22 a =1111,b =1111,cin = 1,outsum =1111,outcar =1
#24 a =0001,b =0000,cin = 0,outsum =0001,outcar =0
#26 a =0001,b =0000,cin = 1,outsum =0010,outcar =0
#28 a =0001,b =0001,cin = 1,outsum =0011,outcar =0
```

Figure 5.42 Simulation results of running the test bench in Figure 5.41.

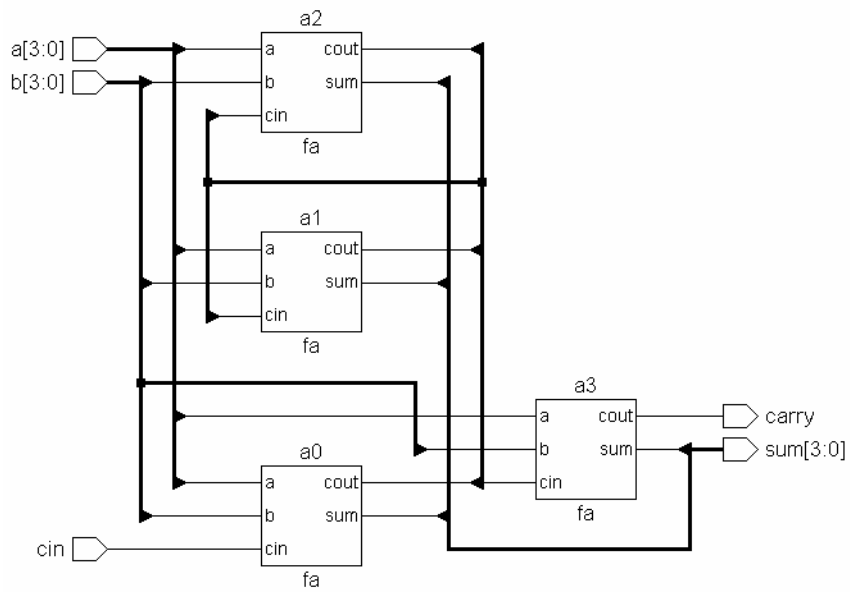


Figure 5.43 Synthesized circuit of the adder module of Figure 5.41.

Figure 5.44 shows a module to AND two nibbles. It is done through direct instantiation of AND gate primitives for two inputs. The corresponding synthesized circuit is shown in Figure 5.45.

```
module andg4(c,a,b);
input[3:0]a,b;

output[3:0]c;
and(c[0],a[0],b[0]);
and(c[1],a[1],b[1]);
and(c[2],a[2],b[2]);
and(c[3],a[3],b[3]);
endmodule
```

Figure 5.44 A 4-bit adder module.

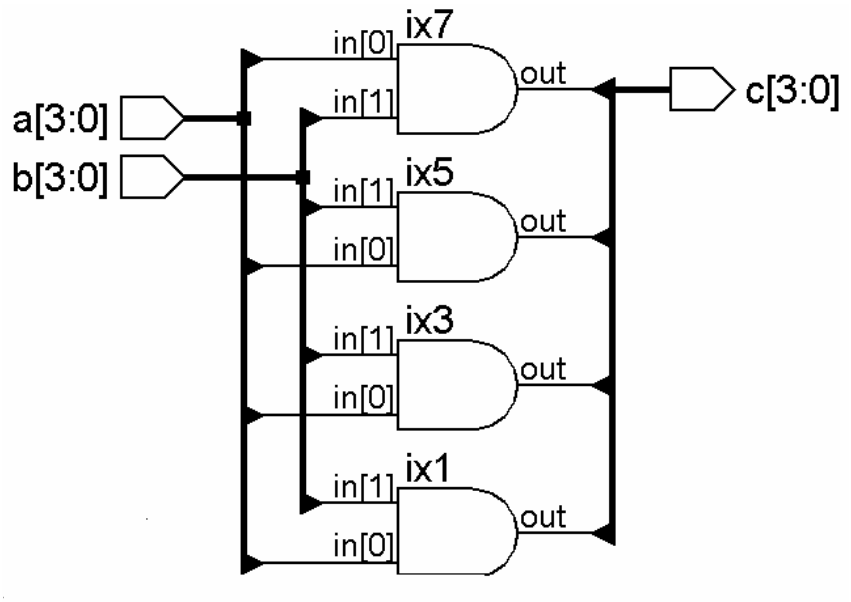


Figure 5.45 Synthesized circuit of the AND module of Figure 5.44 andg4.

The module in Figure 5.46 carries out the bit-wise XOR operation on 2 nibbles. Its synthesized circuit is shown in Figure 5.47. Similarly, the module in Figure 5.48 complements 2 nibbles in a bit-wise manner. The corresponding synthesized circuit is shown in Figure 5.49.

```
module xorg(c,a,b);
input[3:0]a,b;
//input cen;
output[3:0]c;
wire [3:0]cc;
xor x0(c[0],a[0],b[0]);
xor x1(c[1],a[1],b[1]);
xor x2(c[2],a[2],b[2]);
xor x3(c[3],a[3],b[3]);
endmodule
```

Figure 5.46 A 4-bit XOR module.

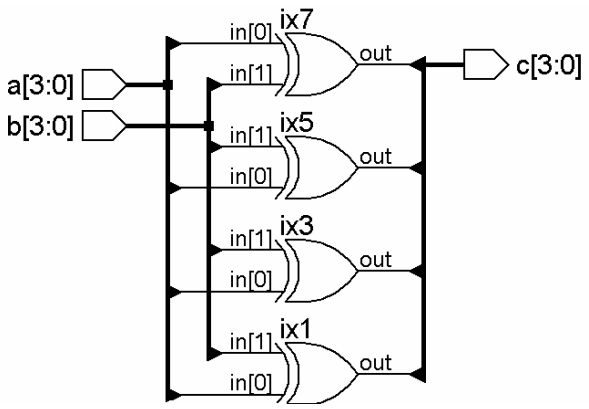


Figure 5.47 Synthesized circuit of the XOR module of Figure 5.46.

```
module compl(c,a);
input[3:0]a;
output[3:0]c;
not(c[0],a[0]);
not(c[1],a[1]);
not(c[2],a[2]);
not(c[3],a[3]);
endmodule
```

Figure 5.48 A module to complement a 4-bit vector.

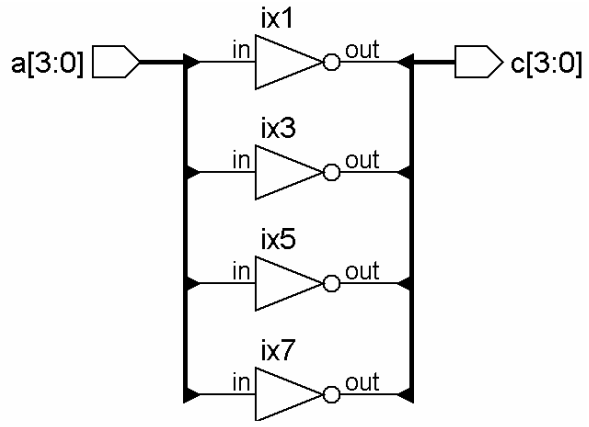


Figure 5.49 Synthesized circuit of the module in Figure 5.48.

```
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
```

Figure 5.50 A 2-to-4 decoder module.

A 2-bit binary number with its 4 distinct states can be used to select any one of the 4 desired functions; it calls for the use of a 2-to-4 decoder. Such a module is shown in Figure 5.50, and its synthesized circuit is shown in Figure 5.51.

As explained above, the decoder outputs can be used to select anyone of the 4 functional outputs and steer it to the final output; a 4-to-1 mux serves this purpose. The mux module is shown in Figure 5.52; its synthesized circuit is in Figure 5.53.

The overall ALU module is shown in Figure 5.54. It instantiates all the above modules. Depending on the mode specified, one of the four functions is selected by the 2-to-4 decoder; its output is multiplexed on to the output by the 4-to-1 mux. The ALU module here has been synthesized and shown in Figure 5.55. Each functional block instantiated in Figure 5.54 appears here as a corresponding distinct black box.

More functions can be added, if desired, to make the ALU more comprehensive. The ALU size can be increased to 16 or 32 bits by repeated instantiation (after some minor modifications) of the 4-bit module in a more comprehensive module.

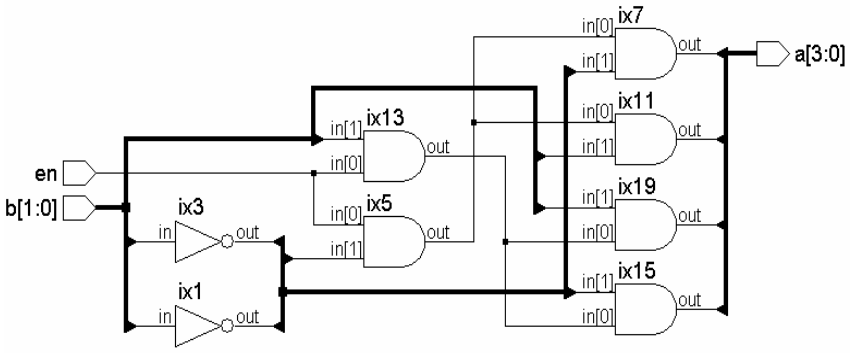


Figure 5.51 Synthesized circuit of the decoder module of Figure 5.50.

```
module mux4_1alu(y,i,e);
input [3:0] i;
input e;
output [3:0]y;
bufif1 g1(y[3],i[3],e);
bufif1 g2(y[2],i[2],e);
bufif1 g3(y[1],i[1],e);
bufif1 g4(y[0],i[0],e);
endmodule
```

Figure 5.52 A 4-to-1 mux module.

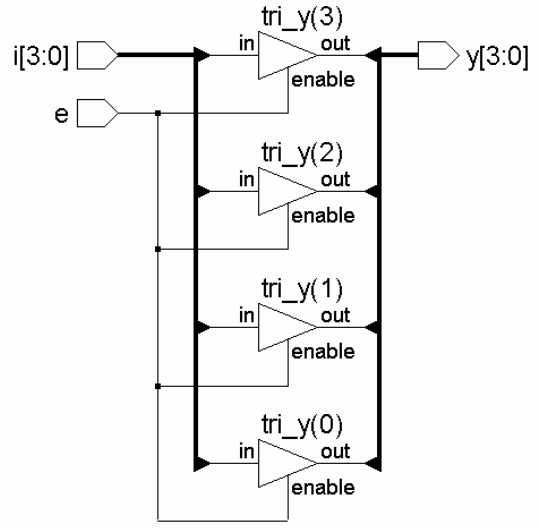


Figure 5.53 Synthesized circuit of the mux module of Figure 5.52.

```
module alu_4g(a,b,c,carry,cin,cen,s);
input [3:0]a,b;
input[1:0]s;
input cen,cin;
output [3:0]c;
output carry;
wire [3:0] data0,data1,data2,data3,e;
wire carry1 ;
dec2_4 m5(e,s,cen);
add4g m1(data0,carry1,a,b,cin);
```

continued

continued

```
compl m2(data1,a);
xorg  m3(data2,a,b);
andg4 m4(data3,a,b);
bufif1 g5(carry,carry1,cen);
mux4_1alu m6(c,data0,e[0]);
mux4_1alu m7(c,data1,e[1]);
mux4_1alu m8(c,data2,e[2]);
mux4_1alu m9(c,data3,e[3]);
endmodule
```

Figure 5.54 A 4-bit ALU module.

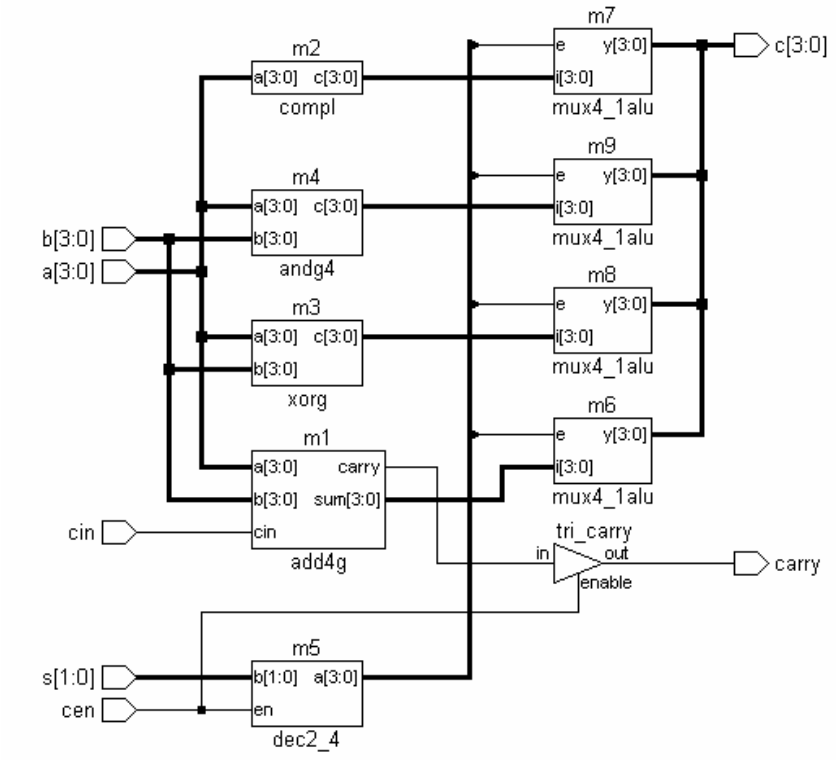


Figure 5.55 Synthesized circuit of the ALU module of Figure 5.54.

5.7 EXERCISES

In each of the cases below, prepare the test bench and test the design

1. Realize each of the flip-flops below using NOR gates.
   RS flip-flop; D-latch; Clocked RS flip-flop; Edge-triggered D flip-flop; Master-slave flip-flop.
2. Figure 5.56 shows the circuit of a flip-flop. Prepare the design module and test it. Explain why it does not work.
3. Modify the flip-flop in Figure 5.56 above with 2 ns delay for sb. Test the flip-flop with different waveforms for d and clk; in each case ensure that the clock does not remain high continuously for more than 1 ns. Explain the need for this restriction.
4. Figure 5.57 shows the basic memory cell built around a d-latch. One can write data into it or read data from it.
   a. When rd/wrb input is low, the flip-flop is in write mode; data are an input line; data on data line are written into the latch, when clk is given a positive pulse.
   b. When rd/wrb input is high, the flip-flop is in read mode; data stored in the latch are made available on the data line.

   Build a module around the d-latch to realize the memory cell.
5. Expand the above to form a byte-wide memory cell.

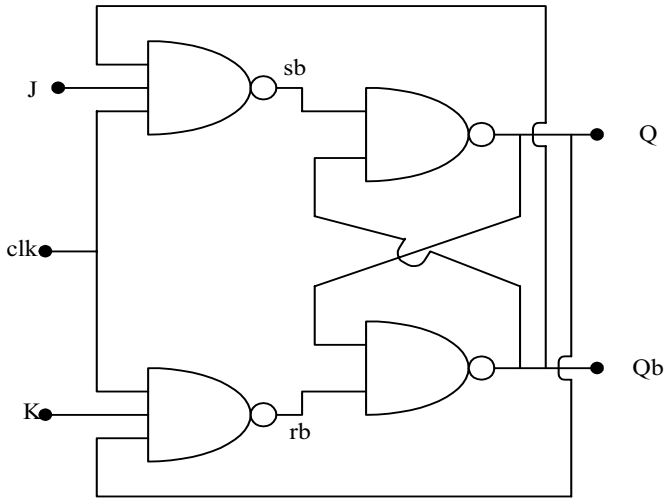


Figure 5.56 A conventional JK flip-flop.

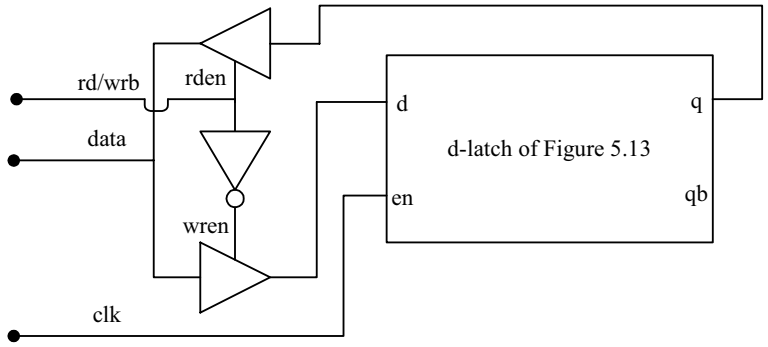


Figure 5.57 A d-latch with necessary additional circuitry to form a memory cell.

6. Replicating the memory element above, one can form a memory. Consider a memory of 16 locations addressed by a 4-bit-wide address bus. The memory will have a 4-to-16 address decoder. It will have an `rd/wrb` input for reading from it and writing data into it. The decoded address can be used to gate the rden and wren inputs to the respective tri-state buffers. Prepare the design module for the memory.
7. Consider the 4-to-1 mux module in Figure 4.31 and its synthesized circuit in Figure 4.33. Identify the signal paths in which maximum number of gates is involved. What is the number of gates in the path here?

   Identify the signal paths in which the number of gates involved is a minimum. What is the number of gates here and which are these?
8. For each of the gate primitives in Exercise 7 above, take the minimum, typical, and maximum delays to be 1 ns, 2 ns and 3 ns respectively. With the typical delay values, estimate the minimum and maximum delays of transmission. Verify by simulation. Repeat the exercise with minimum and minimum delay values.
9. In Exercise 8 above, assign the minimum delay values for the shortest paths and maximum for the longest paths. Using these, estimate the minimum and maximum time delays for the mux (see also the pin-to-pin delay specifications in Chapter 11).
10. Identify the ALU functions in the 8085 processor. Design an ALU module to carry out these.
11. Identify the ALU functions in the 8088 processor. Design an ALU module to carry out these (ignore the instructions for multiplication and division).
12. a[1:0] and b[1:0] are two 2-bit numbers. Their product – designated as m[3:0] – is in general a 4-bit number; it is formed as follows:

    Form m[0] by AND operation on a[0] and b[0].

Through a half-adder add the bits a[1]&b[0] and a[0]&b[1]. The sum bit is m[1]. Let the carry bt be c.

Through a half-adder add the bits a[1]&b[1] and c obtained above. The sum bit is m[2] and the carry bit m[3].

Design a 2-bit multiplier following the above steps and test it for all possible input value combinations.

13. Let abcd and efgh be two 4-bit numbers where a, b,…., g, h represent the respective bit values. The 4-bit numbers are multiplied as follows:

    Form the four 4-bit numbers 00cd, 00gh, ab00, and ef00.

    Form the following four intermediate products using 2-bit multipliers:

    00cd with 00gh
    00cd with ef00
    ab00 with 00gh
    ab00 with ef00

    Add all the above four intermediate products to get the final 7-bit result.

    Design a 4-bit multiplier module following the above steps. Instiantiate 2-bit multiplier module, half- and full-adder modules, *etc*., wherever necessary.

14. Following steps analogous to the above, design an 8-bit multiplier.
15. Write down the Boolean logic expressions for all the product bits of a 4-bit multiplier; using these, design an 8-bit multiplier.