CMR COLLEGE OF ENGINEERING & TECHNOLOGY
Kandlakoya (V), Medchal Road, HYDERABAD -501401
(**An Autonomous Institution under UGC & JNTUH , Approved by AICTE, Permanently Affiliated to JNTUH, Accredited by NAAC with 'A' Grade.**)

DEPARTMENT OF ECE

COURSE FILE

SUBJECT: DIGITAL DESIGN USING VERILOG HDL

PREPARED BY

Ms. T.SWAPNA RANI
Ms. NISHI CHANDRA

DEPARTMENT OF ECE

TIME TABLE

Faculty : Ms.T.Swapna Rani					CLASS:III ECE- A&B		
	1 9-10 - 10.10	**2 10.10 - 11.10**	**3 11.10 - 12.10**	**12.10 - 01.00**	**4 1.00 - 2.00**	**5 2.00 - 3.00**	**6 3.00 - 4.00**
MON	**III ECE D**				**M.TECH**		
TUE		**III ECE A**					
WED	**III ECE A**					**III ECE D**	
THU							**III ECE D**
FRI	**III ECE A**		**III ECE D**			**III ECE A**	
SAT			**III ECE A**		**III ECE D**		

Faculty : Ms. Nishi Chandra					CLASS:III ECE- C &D		
	1 9-10 - 10.10	**2 10.10 - 11.10**	**3 11.10 - 12.10**	**12.10 - 01.00**	**4 1.00 - 2.00**	**5 2.00 - 3.00**	**6 3.00 - 4.00**
MON		**III ECE C**					
TUE	**III ECE B**				**III ECE C**		
WED						**III ECE B**	
THU	**III ECE C**						**III ECE B**
FRI		**III ECE C**			**PA III ECE C**		
SAT		**III ECE B**				**III ECE C**	

SYLLABUS

Syllabus
CMR COLLEGE OF ENGINEERING & TECHNOLOGY, HYDERABAD

UNIT-I

INTRODUCTION TO VERILOG: Verilog as HDL, Levels of design Description, Concurrency, Simulation and Synthesis, Functional Verification, System Tasks, Programming Language Interface (PLI), Module, Simulation and Synthesis Tools

LANGUAGE CONSTRUCTS AND CONVENTIONS: Introduction, Keywords, Identifiers, White Space Characters, Comments, Numbers, Strings, Logic Values, Strengths, Data Types, Scalars and Vectors, Parameters, Operators.

UNIT-II:

GATE LEVEL MODELING: Introduction, AND Gate Primitive, Module Structure, Other Gate Primitives, Illustrative Examples, Tri-State Gates, Array of Instances of Primitives, Design of Flip – Flops with gate primitives, Delays, Strengths and Contention Resolution, Net Types, Design of Basic Circuits.

MODELING AT DATA FLOW LEVEL: Introduction, Continuous Assignment Structures, Delays and Continuous Assignments, Assignment to Vectors, Operators.

UNIT-III:

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

UNIT-IV:

SWITCH LEVEL MODELLING: Basic Transistor Switches, CMOS Switch, Bi – directional Gates, Time Delays with Switch Primitives, Instantiations with Strengths and Delays, Strength Contention with Trireg Nets.

SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES: Parameters, Path Delays, Module Parameters, System Tasks and Functions, File – Based Tasks and Functions, Compiler Directives, Hierarchical Directives, Hierarchical Access, User-defined Primitives (UDP).

UNIT-V:

SEQUENTIAL CIRCUIT DESCRIPTION: Sequential Models – Feedback Model, Capacitive Model, Implicit Model, Basic Memory Components, Functional Register, Static Machine Coding, Sequential Synthesis.

COMPONENT TEST AND VERIFICATION: Test Bench – Combinational Circuit Testing, Sequential Circuit Testing, Test Bench Techniques, Design Verification, Assertion Verification.

TEXT BOOKS:

1. Design through Verilog HDL – T. R. Padmanabhan and B. Bala Tripura Sundari, WSE, 2004, IEEE Press.
2. ZainalabdienNavabi, Verilog Digital System Design, TMH, 2nd Edition.

REFERENCES:

1. Fundamentals of Logic design with Verilog – Stephen Brown and ZvonkoVranesic, TMH, 2nd Edition, 2010.
2. Advanced Digital Logic Design using Verilog, State Machine & Synthesis for FPGA – Sunggu Lee, Cengage Learning, 2012.
3. Verilog HDL – Samir Palnitkar, 2nd Edition, Pearson Education, 2009.
4. Advanced Digital Design with Verilog HDL – Michael D. Ciletti, PHI, 2005.

LESSON PLAN

CMR COLLEGE OF ENGINEERING & TECHNOLOGY
Kandlakoya (V), Medchal Road, Hyderabad

DEPARTMENT OF ECE

LESSON PLAN

Name : T Swapna Rani/Nishi Chandra
Subject : Digital Design Through Verilog HDL
Desig. : Asst. Prof Branch : ECE
Department : E.C.E Year and Sem.: III – I

Unit / Topic	No. of periods required	Cumulative periods	Planned date
UNIT I			
Introduction, Verilog as HDL	1	1	13.06.2016
Levels of Design Description	1	3	14.06.2016
Concurrency	1	4	16.06.2016
Simulation and Synthesis	1	5	18.06.2016
Functional Verification	1	6	18.06.2016
System Tasks	1	7	20.06.2016
Programming Language Interface (PLI)	1	8	21.06.2016
Module	1	9	23.06.2016
Simulation and Synthesis Tools	1	10	23.06.2016
LANGUAGE CONSTRUCTS AND CONVENTIONS: Introduction	1	11	25.06.2016
Keywords, Identifiers	1	12	27.06.2016
White Space Characters, Comments	1	13	28.06.2016
Numbers, Strings	1	14	30.06.2016
Logic Values, Strengths,	1	15	31.06.2016
Data Types, Scalars and Vectors	1	16	02.07.2016
Parameters, Operators	1	17	03.07.2016
Tutorial	1	18	04.07.2016
UNIT II			
Introduction, AND Gate Primitive	1	19	05.07.2016
Module Structure	1	20	07.07.2016
Other Gate Primitives	1	21	11.07.2016
Illustrative Examples	1	22	12.07.2016
Tri-State Gates	1	23	16.07.2016
Array of Instances of Primitives	1	24	18.07.2016
Design of Flip-flops with Gate Primitives	1	25	19.07.2016
Delays	1	26	21.07.2016
Strengths and Contention Resolution	1	27	23.07.2016
Net Types	1	28	25.07.2016
Design of Basic Circuits	1	29	26.07.2016
DATA FLOW LEVEL: Introduction	1	30	26.07.2016
Continuous Assignment Structures	1	31	28.07.2016
Delays and Continuous Assignments	1	32	30.07.2016
Assignment to Vectors	1	33	01.08.2016
Operators	1	34	02.08.2016
Tutorial	1	35	03.08.2016
UNIT III			
BEHAVIORAL MODELING: Introduction	1	36	04.08.2016

Operations and Assignments	1	37	06.08.2016
Functional Bifurcation	1	38	08.08.2016
Initial Construct	1	39	09.08.2016
Always Construct, Examples	1	40	10.08.2016
Assignments with Delays	1	41	12.08.2016
Wait construct, Multiple Always Blocks	1	42	14.08.2016
Designs at Behavioral Level	1	43	16.08.2016
Blocking and Non-blocking Assignments	1	44	18.08.2016
The case statement, Simulation Flow	1	45	20.08.2016
if and if-else constructs	1	46	21.08.2016
assign-de assign construct	1	47	22.08.2016
repeat construct, for loop	1	48	23.08.2016
disable construct, while loop	1	49	25.08.2016
forever loop, parallel blocks	1	50	27.08.2016
force-release construct, Event	1	51	29.08.2016
Tutorial	1	52	30.08.2016
UNIT IV			
Introduction, Basic Transistor Switches	1	53	31.08.2016
CMOS Switches	1	54	01.09.2016
Bi-directional Gates	1	55	03.09.2016
Time Delays with Switch Primitives	1	56	06.09.2016
Instantiations with Strengths and Delays	1	57	13.09.2016
Strength Contention with Trireg Nets	1	58	14.09.2016
SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES: Introduction	1	59	15.09.2016
Parameters	1	60	17.09.2016
Path Delays	1	61	19.09.2016
Module Parameters	1	62	20.09.2016
System Tasks and Functions	1	63	22.09.2016
File-Based Tasks and Functions	1	64	24.09.2016
Compiler Directives	1	65	26.09.2016
User- Defined Primitives (UDP)	1	66	29.09.2016
Hierarchical Access	1	67	30.09.2016
Tutorial	1	68	01.10.2016
UNIT V			
Sequential models	1	69	03.10.2016
Basic Memory Components	1	70	04.10.2016
Functional Register	1	71	06.10.2016
State Machine Coding	1	72	13.10.2016
Testbench	1	73	15.10.2016
Testbench techniques	1	74	17.10.2016
Design Verification	1	75	18.10.2016
Assertion Verification	1	76	20.10.2016
Tutorial	1	77	22.10.2016

Initials of the Faculty Initials of the HOD

UNIT-I

INTRODUCTION TO VERILOG LANGUAGE CONSTRUCTS AND CONVENTIONS

Contents

- Verilog as HDL
- Levels of design Description
- Concurrency
- Simulation and Synthesis
- Functional Verification
- System Tasks
- Programming Language Interface (PLI)
- Module
- Simulation and Synthesis Tools
- Keywords, Identifiers, White Space Characters, Comments, Numbers, Strings, Operators.
- Logic Values
- Strengths
- Data Types
- Scalars and Vectors
- Parameters

Lecture Notes

UNIT – I

INTRODUCTION TO VERILOG

1. VERILOG AS 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.

History:

Beginning

Verilog was one of the first modern hardware description languages to be invented. It was created by Prabhu Goel and Phil Moorby during the winter of 1983/1984. The wording for this process was "Automated Integrated Design Systems" (later renamed to Gateway Design Automation in 1985) as a hardware modeling language. Gateway Design Automation was purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL, the HDL-simulator that would become the de facto standard (of Verilog logic simulators) for the next decade. Originally, Verilog was intended to describe and allow simulation; only afterwards was support for synthesis added.

Verilog-95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

In the same time frame Cadence initiated the creation of Verilog-A to put standards support behind its analog simulator Spectre. Verilog-A was never intended to be a standalone language and is a subset of Verilog-AMS which encompassed Verilog-95.

Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the Boolean algebra to determine its correct value). The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/end generate construct (similar to VHDL's generate/end generate) allows Verilog-2001 to control instance and statement instantiation through normal decision operators (case/if/else). Using generate/end generate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system tasks. And finally, a few syntax additions were introduced to improve code readability (e.g. always @*, named parameter override, C-style function/task/module header declaration).

Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.

Verilog 2005

Not to be confused with System Verilog, *Verilog 2005* (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features (such as the uwire keyword).

A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

System Verilog

System Verilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the System Verilog and Verilog language standards were merged into System Verilog 2009 (IEEE Standard 1800-2009).

The advent of hardware verification languages such as Open Vera, and Verisity'se language encouraged the development of Super log by Co-Design Automation Inc. Co-Design Automation Inc was later purchased by Synopsys. The foundations of Super log and Vera were donated to Accellera, which later became the IEEE standard P1800-2005: System Verilog.

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.

2.1 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. Fig.1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

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. Fig.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.

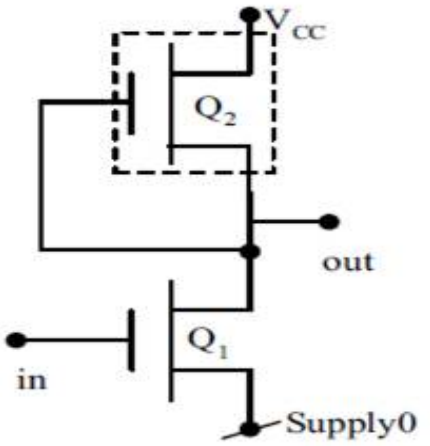


Fig.1 A simple Inverter circuit at the switch level

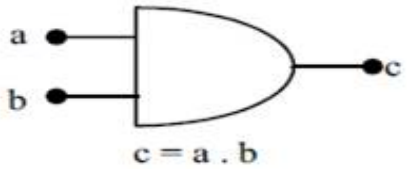


Fig.2 A simple AND gate represented at the gate level

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. Fig.3 shows an A-O-I relationship suitable for description with the Verilog constructs at the data flow level.

2.4 Behavioral Level

Behavioral level constitutes the highest level of design description; it is essentially at the

system level itself. 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. Fig. 4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of Verilog.

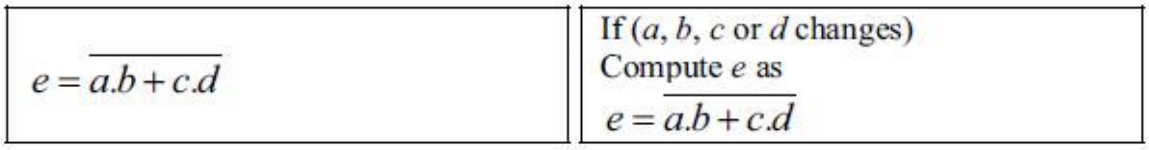


Fig.3 An A-O-I gate represented as a data flow type of relation

Fig.4 An A-O-I gate in pseudo code at behavior level

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.

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 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.

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").

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.

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.*

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.

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 Fig.5. The terminals of the module are referred to as 'ports'. The ports attached to a module can be of three types:

- **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 entity 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

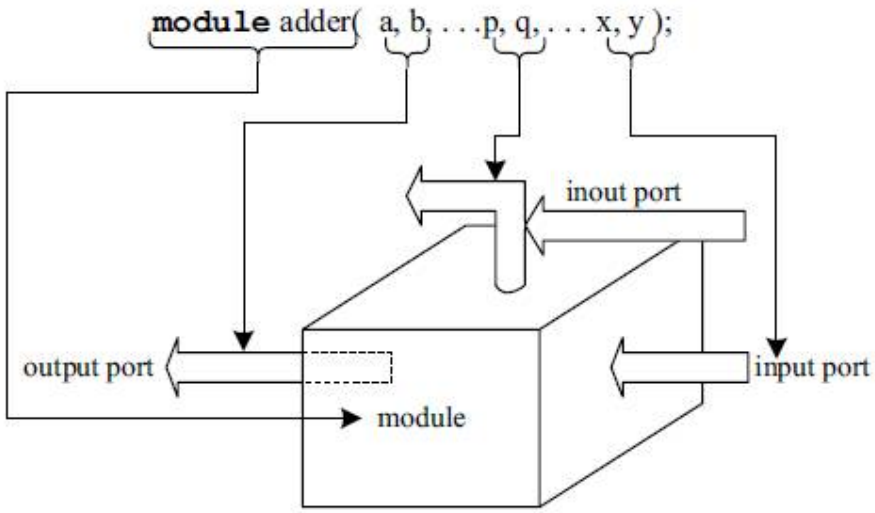


Fig.5 Representation of a module

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 Fig.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.

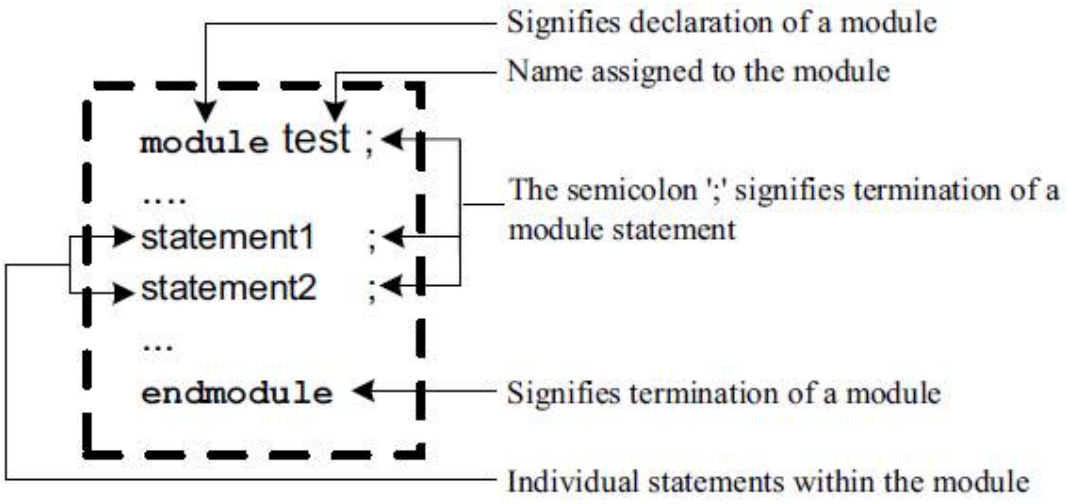


Fig.6 Structure of a typical simulation module

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 (Fig.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 mariner 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(fig.7b).

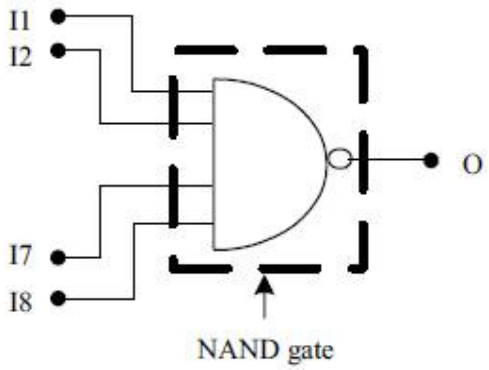


Fig. 7a Eight input NAND gate

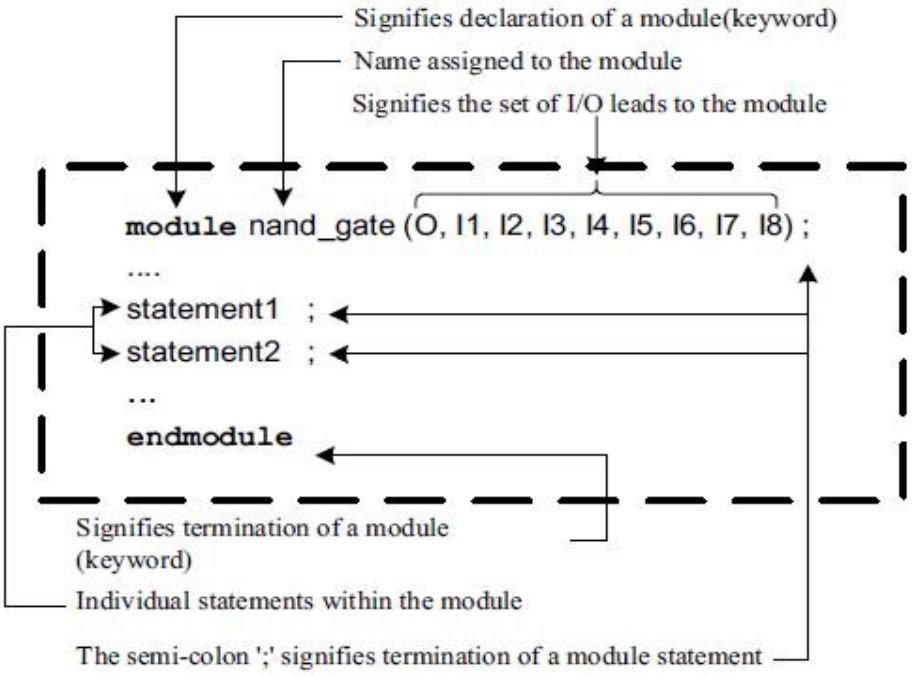


Fig. 7b Eight input NAND gate (Structure of the gate module)

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 Fig.8. The associated table (Table 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 Fig.8.

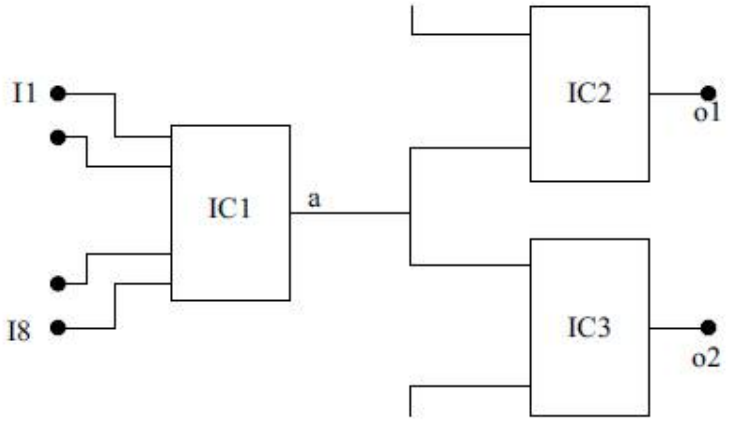


Fig.8 part of the circuit diagram of a digital circuit

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

Table 1 Partial list of IC numbers and their types

Each such use is an "instantiation." A typical instantiation of the module defined above has the form shown in Fig.9.

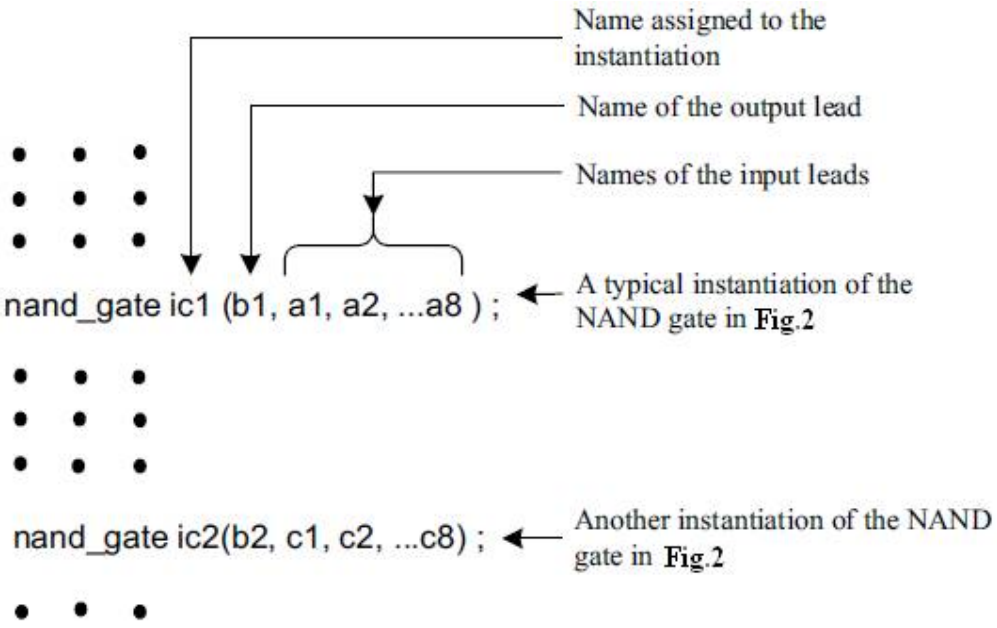


Fig.9 Instantiations of module nand_gate in another module

The following observations are in order here:

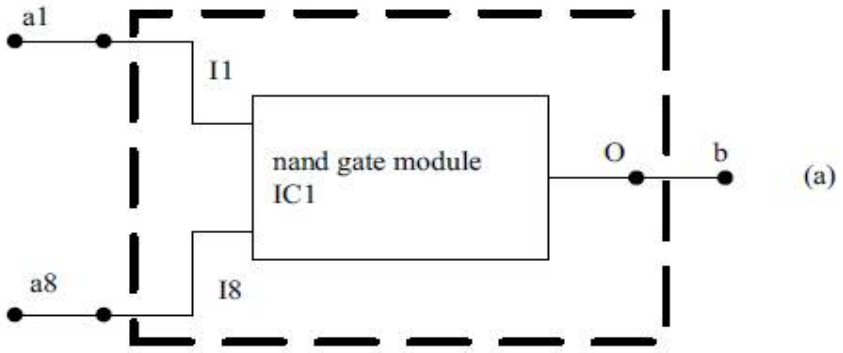
- The designer has defined a specific function within a module; the module is assigned the name "nand_gate."
- The nand_gate can be invoked (instantiated) by him in a design as many times as desired.
- Each instantiation has to be assigned a separate identifier name by him (called "IC1", "IC2",etc.).

- 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 Fig.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 Fig.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.




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

Fig.10 a) A typical circuit block b) Its instantiation

9. SIMULATION AND SYNTHESIS TOOLS

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.

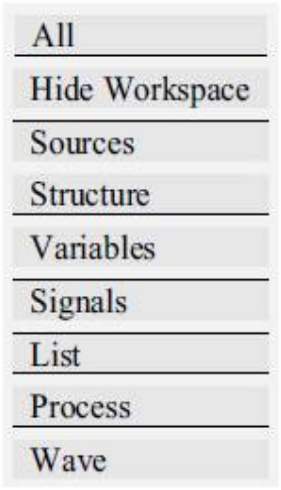
2.9.1 Use of Modelsim SE 5.5

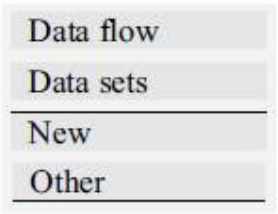
The procedure to invoke the tool and use it is briefly described here. The tool can be used to prepare a source file, edit and compile it, and simulate the compiled version.

Editing and Compilation

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

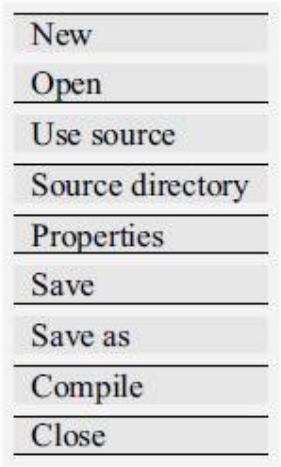
- Click on "View." We get the following menus





- 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.
- 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" pull down 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.

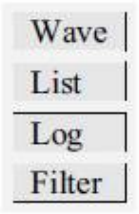
- The "Load" button below gets highlighted. Click on it. The design gets loaded and is ready for simulation run.
- Click the "Run" menu in the Modelsim main window. Select 100 ns runtime.
- The design runs for 100 ns (by default) and the output list appears in the main window. The listing can be selected, 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

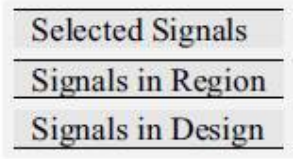
Simulation results can alternately be viewed as waveforms with the following procedure:

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

- Click on the "View" pull down 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.

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

pull down menu as follows

- Click on “File”. A pull down 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 Fig.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 design that was synthesized. It also generates and displays some timing information like "Critical Path Timing."

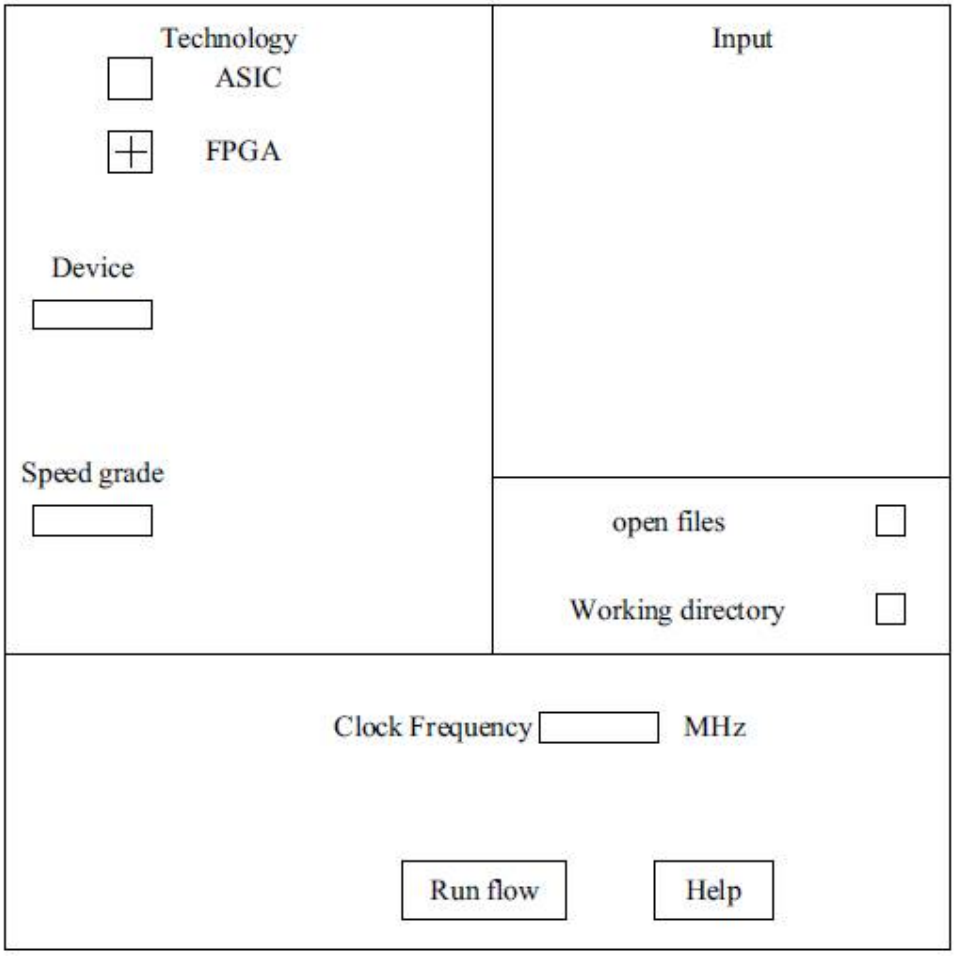


Fig.11 The window to do the settings for synthesis

LANGUAGE CONSTRUCTS AND CONVENTIONS

10. 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. In many respects they resemble those of C language

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.

10.1 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.

11. KEYWORDS

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

Examples

module<— signifies the beginning of a module definition.

endmodule<— signifies the end of a module definition.

begin<— signifies the beginning of a block of statements.

end<— signifies the end of a block of statements.

if<— signifies a conditional activity to be checked **while**<— signifies a conditional activity to be carried out.

12. 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, namel, namej, . . . <- 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.

13. WHITE SPACE CHARACTERS

Blanks (\b), tabs (\t), newlines (\n), and form feed 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.

14. 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, elk);

/* 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 elk 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.*/

15. 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.

15.1 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 non decimal 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. Fig.12 shows typical number representations with the significance of each field explained separately.

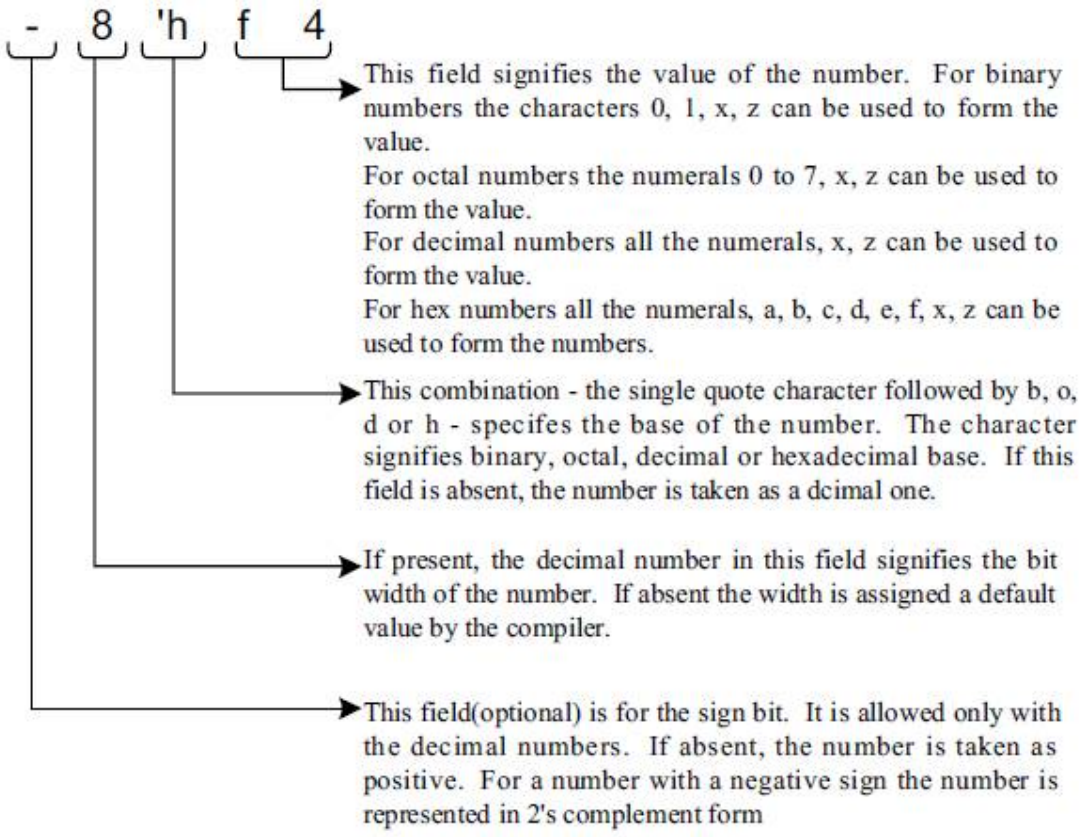


Fig.12 Representation of a number in verilog

Table 2 shows the format of specifications of the integer type numbers along with illustrative examples.

Representation	Remarks
33 'd33	Both of these represent decimal numbers of unspecified size - normally interpreted by Verilog as 32 bit wide, *i.e.*, 0000 0000 0000 0000 0000 0000 0010 0001
9'd439 9'D439 9'D4_39	All these represent 3 digit decimal numbers. D & d both specify decimal numbers. (underscore) is ignored
9'b1 1011 1x01 9'b11011x01 9'B11011x01	All these represent binary numbers of value 11011x01. B & b specify binary numbers. is ignored, x signifies the concerned bit to be of unknown value.
9'o123 9'0123 9'o1x3 9'o12z	All these represent 9-bit octal numbers. The binary equivalents are 001 010011, 001 010 011, 001 xxx 011,001010zzz respectively, z signifies the
'0213	An octal number of unspecified size having octal value 213.
8'ha5 8'HA58'hA5 8'ha_5	All these are 8 bit-wide-hex numbers of hex value a5h. The equivalent binary value is 1010 0101.
11'hb0	A 11 bit number with a hex assignment. Its value is 000 1011 0000. The number of bits specified is more than that indicated in the value field. Enough zeros are padded to the left as shown.
9'hza	A hex number of 9 bits. Its value is taken as zzzzz 1010.
5'hza	A 5-bit hex number. Its value is taken as z 1010.
5'h?a	A 5-bit hex number. Its value is taken as z 1010. '?' is another representation for 'z\
-5'h1a -3'b101	Negative numbers. Negative numbers are represented in 2's complement form.
-4'd7	A 4 bit negative number. Its value in 2's complement form is 7. Thus the number is actually - (16 - 7) = -9.

Table 2 Different ways of number representations in verilog

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 O'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.

15.2 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 fieldsa 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.

16. 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 acarriage 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 11100111 0101 0110 1101 01100010

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).

17. 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, it is equivalent to having the **x** value.

- The MOS switches 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** is an exception to this. It cannot store any value. The **trireg** cannot take on the **z** value. 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.

18. 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 types. Details are given in Table 3.

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 **"strongl"**and**C** at level **0** with strength **"pullO"-** a will take the value **1.**

- If a signal linea is driven by two sources — b at 1 level with strength **"pulll"**andC at level 0 with strength **"strongO,"** a will take the value 0.

- If a signal linea is driven by two sources — b at 1 level with strength **"strongl"**andC at level 0 with strength **"strongO,"** a will take the value x (indeterminate).

- If a signal linea is driven by two sources — b at 1 level with strength **"weakl"**andC at level 0 with strength **"largeO,"** a will take the value 0.

Strength name	Strength level (signifies inverse of source	Specification keyword	Abbreviation	Element modeled
Supply drive	7	**Supplyl** **Supply0**	**Sul** **Su0**	Power supply connection
Strong drive	6	**Strongl** **Strong0**	**Stl** St0	Default gate and assign output strength
Pull drive	5	**Pulll** **Pull0**	**Pul** **Pu0**	Gate and assign output strength
Large capacitor	4	**Largel** **Large0**	**La1** **La0**	Size of trireg net capacitor
Weak drive	3	**Weakl** **Weak0**	**We1** **We0**	Gate and assign output strength
Medium capacitor	2	**Mediuml** **Medium0**	**Mel** **Me0**	Size of trireg net capacitor
Small capacitor	1	**Smalll** **Small0**	**Sml** **Sm0**	Size of trireg net capacitor
High impedance	0	**Highzl** **Highz0**	**Hil** **Hi0**	Tri-stated line

Table 3 Details of strengths in verilog

19. DATATYPES

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.

19.1 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.

19.2 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.

20. 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 **regs** stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector." Fig.13 illustrates the difference between a scalar and a vector, wr and rd are two scalar nets connecting two circuit blocks circuit 1 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.

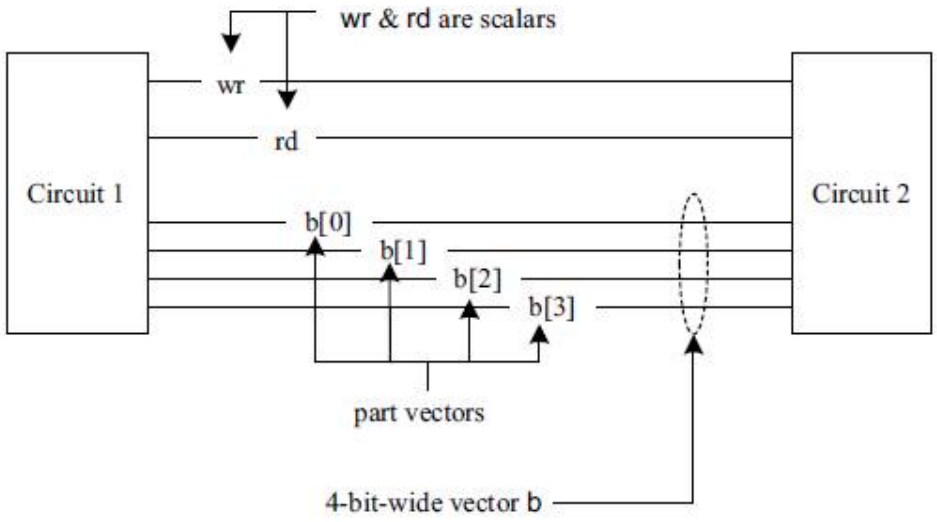


Fig.13 Illustration of Scalars and Vectors

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]anda[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]numl; // numl is a vector in the range -8 to +7.

21. 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,

parameterWOrd_size =16;

parameterwordsize = 16, memsize = 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 ofthe 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;

21. OPERATORS

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

a. Unary: the unary operator is associated with a single operand. The operator precedes the operand - for example, ~a.

b. Binary: the binary operator is associated with two operands. The operator appears between the two operands - for example,a&b.

c. 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.
```

UNIT-II

GATE LEVEL MODELING
MODELING AT DATA FLOW LEVEL

Contents

- AND Gate Primitive
- Module Structure
- Other Gate Primitives
- Illustrative Examples
- Tri-State Gates
- Array of Instances of Primitives
- Design of Flip – Flops with gate primitives
- Delays
- Strengths and Contention Resolution
- Net Types
- Design of Basic Circuits
- Introduction
- Continuous Assignment Structures
- Delays and Continuous Assignments
- Assignment to Vectors
- Operators

UNIT – II

GATE LEVEL MODELING

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. They can be instantiated directly in other modules. Further design description using gate primitives is quite close to the actual circuits.

2.AND Gate Primitive

The AND gate primitive in Verilog is instantiated with the following statement:

***and g1 (O, 11, 12, . . .,* In);**

Here '**and**' is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. **O** is the gate output; **11, 12,** *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.

2.1 Example 1

Fig.1 shows the stimulus program for testing the AND gate g1. The outputobtained by stimulating the program is shown in Fig.2. Some explanationregarding the simulation program is in order here.

The moduletest_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
```

Fig.1 A Module to instantiate AND gate Primitive

```
 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
```

Fig.2 Output of the Module shown in fig.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 Fig.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.

2.2 Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table 1. It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

		Input 1			
		0	1	*x*	*z*
Input 2	0	0	0	0	0
	1	0	1	*x*	*x*
	x	0	*x*	*x*	*x*
	z	0	*x*	*x*	*x*

Table 1 Truth Table of AND gate Promitive

- 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, l,xor **z** state.
- The output is at 1 state if and only if every one of the inputs is at 1 state.
- For all other cases the output is at the **x** state.
- Note that the output is never at the **z** state - the high impedance state. This is true of all other gate primitives as well.

3. MODULE STRUCTURE

Fig.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.
- All the variables in the ports-list are to be identified as **inputs, outputs,** or**inouts.** The corresponding declarations have the form shown below:
    - **Inputa1,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;

  regb1, 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. 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 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.

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 2 Basic gate primitives in Verilog with details

4.1 Truth Table

Extending the concepts of truth table of the AND gate primitive, one can form the truth tables of all other gate primitives. The basic features of each are given in Table 3.

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	

Table 3 Rules for deciding the output values of gate primitives for different input combinations

5. ILLUSTRATIVE EXAMPLES

5.1 Example 2

The commonly used A-O-I gate is shown in Fig.3 for a simple case. The module and the test bench for the same are given in Fig.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 Fig.4 remain the same as those in the circuit of Fig.3.

The moduleaoi_gate in the figure has input and output ports since it describes a circuit with signal inputs and an output. The moduleaoi_st is a stimulus module. It generates inputs to the aoi_gate module and gets its output. It has no input or output ports.

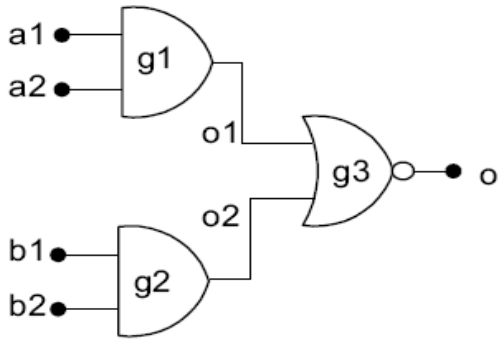


Fig.3 A typical A-O-I gate circuit

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 AOI gate in Fig.3. Within theaoi_gate module, all signals are of type net. Theaoi_ gate module in Fig.4 is instantiated once in the moduleaoi_st for testing.

Any such instantiation of a user-defined module in another module has to be assigned a name. The instantiation is given the name gg here. Note that all the inputs to the instantiation ofaoi_gate in the test bench are fed through**regs.**

Theaoi_gateandaoi_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 Fig.5, which confirms this.

```
/*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
```

Fig.4 Module for A-O-I gate of Fig.3

```
#    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
```

Fig.5 Results of aoi_st test bench of fig.3

The moduleaoi_gate has been synthesized and the synthesized circuit shown in Fig.6; the figure does not warrant any detailed explanation.

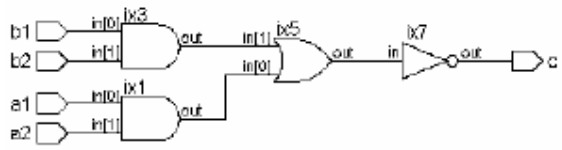


Fig.6 Synthesized version of the moduleaoi_gate of fig.4

```
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
```

Fig.7 Another realization of the A-O-I gate with test bench

Both the modules can do with some elegant simplification. First consider the stimulus moduleaoi_st in Fig.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 Fig.4 is recast in Fig.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 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 inputsto 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.

The stimulus module need not necessarily have a port;*aoi_st* in Fig.4 and *aoi_st2* in Fig.7 are typical examples. The results of running the test bench *aoi_st2* of Fig.7 are shown in Fig.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. Thus the instantiation of the *aoi_gate2* in the test bench of Fig.7 can be described alternately as

```
aoigate2 gg (.0(0), .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.

```
#          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
```

Fig.8 Results of aoi_st test bench

5.2 Example 3: 4-to-16 Decoder

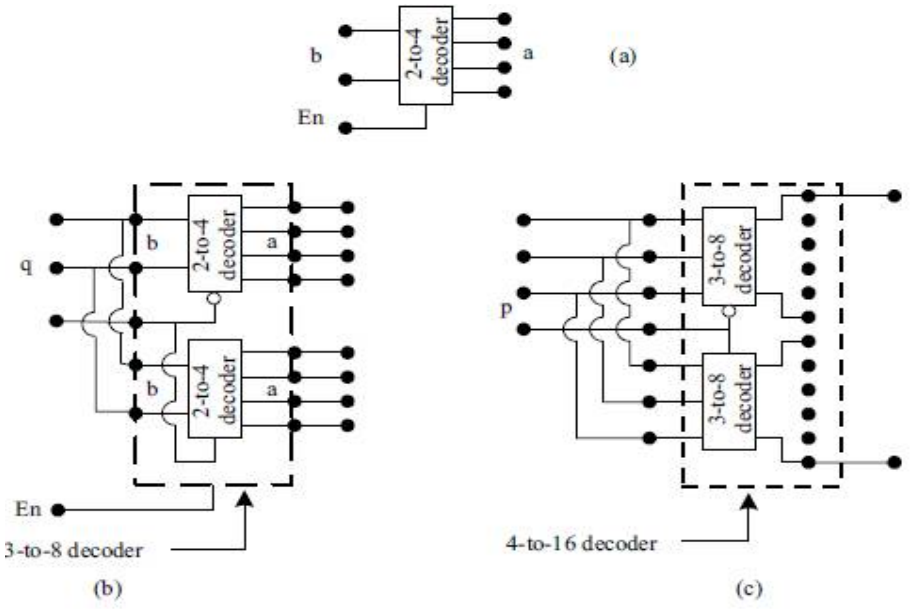


Fig.9 Design of 4–to–16 decoder using smaller decoders

(a) 2–to–4 decoder (b) 3–to–8 decoder (c) 4–to–16 decoder using two 3–to–8 decoders

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.

Fig.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 Fig.9(b). The 2-to-4 decoder block used here is shown in Fig.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.

```
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
```

Fig.10 Design Description of a 2-to-4 decoder circuit with test bench

Fig.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 Fig.11.

```
//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
```

Fig.11 Results of the test bench of Fig.10

Fig.12 shows a 3-to-8 decoder module formed by repeated instantiation of the 2-to-4

decoder of Fig.10. The eight AND gate instantiations ensure that the outputs are enabled only whenenn — a separate "Enable" signal — goes active.

```
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
```

Fig.12 3-to-8 decoder module formed by repeated instantiation of the 2-to-4 decoder module in fig.10

Following the same logic, the module for the 4-to-16 decoder is described in Fig.13. A test bench to test the module through all the possible input states is also included in the figure. Fig.14 shows the results of running the test- bench.

```
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
```

Fig.13 A 4-to-16 decoder formed by the repeated instantiation of 3-to8 decoder module with test bench

```
//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
```

Fig.14 Results of the test bench of test bench for the 4-to-16 decoder

- Two signals within the two nested modules are monitored in dec4_16_stimulus. Formation of their hierarchical addresses is also shown in fig.15.
- 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.
-

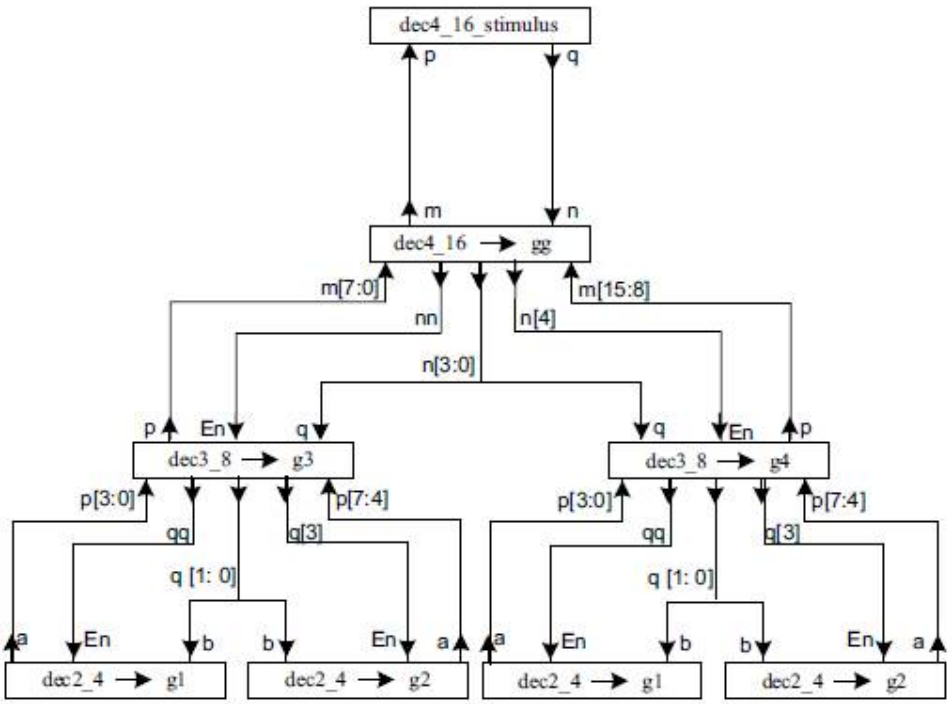


Fig.15 Block diagram representation of the module instantiations and signal assignments for the module of Fig.10

5.2.1 Decoder Synthesis

The synthesized circuit of the 2-to-4 decoder module of Fig.10 (dec2_4) is shown in Fig.16. The AND gate cells available in the library are all of the two-input type; hence six such cells (designated as ix5, ix7, ixll, ixl3, ixl5, and ixl9) 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 ixl and ix3). The wider lines in the figure signify bus- type interconnections.

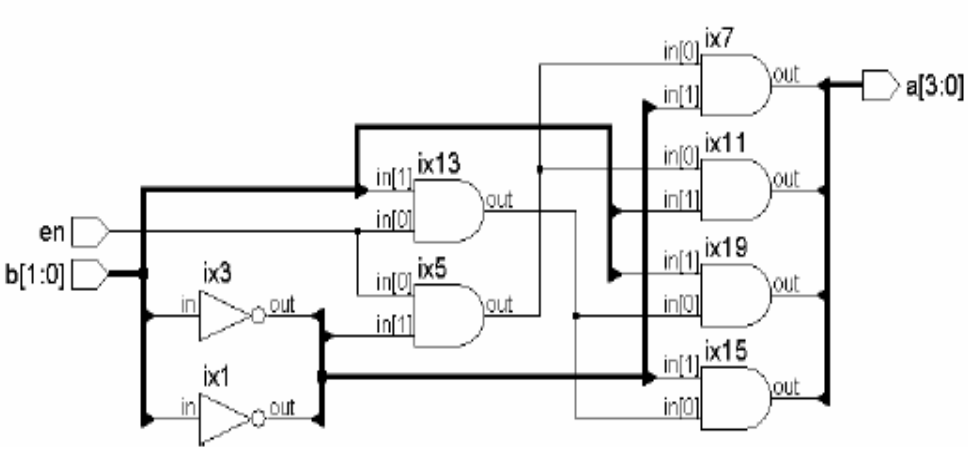


Fig.16 The synthesized circuit of the 2-to-4 decoder of Fig.10

The synthesized circuit of the 3-to-8 decoder module of Fig.12 (dec3_8) is shown in Fig.17. The two instantiations of the dec2_*4* module *(g1* and *g2)* are shown as black boxes.

Similarly, Fig.18 shows the synthesized circuit of the 4-to-16 decoder module of Fig.13 (dec4_16). The two instantiations of the dec3_8 module *(g3* and *g4)* appear as black boxes inside.

Fig.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.

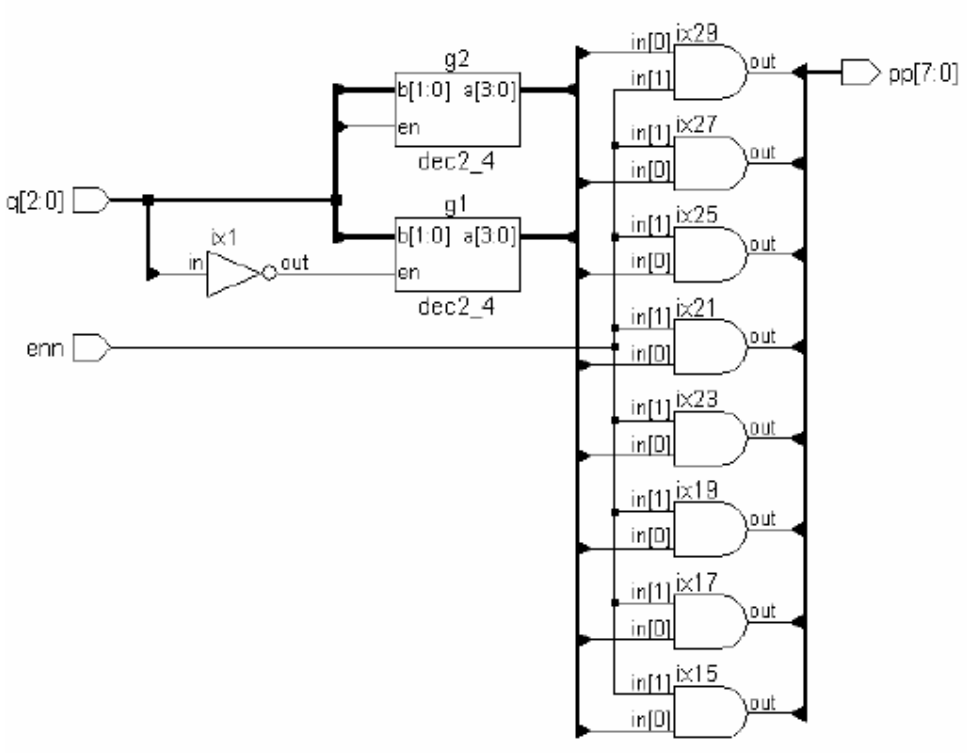


Fig.17 The Synthesized circuit of 3-to-8 decoder of Fig.12

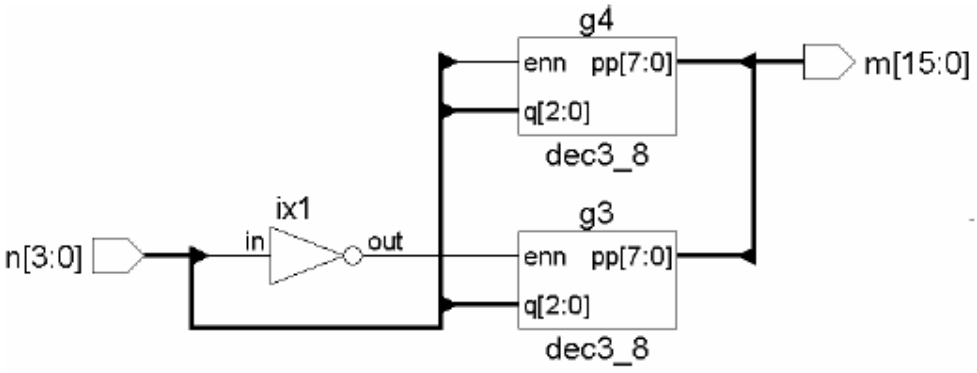


Fig.18 The Synthesized circuit of 4-to-16 decoder of Fig.13

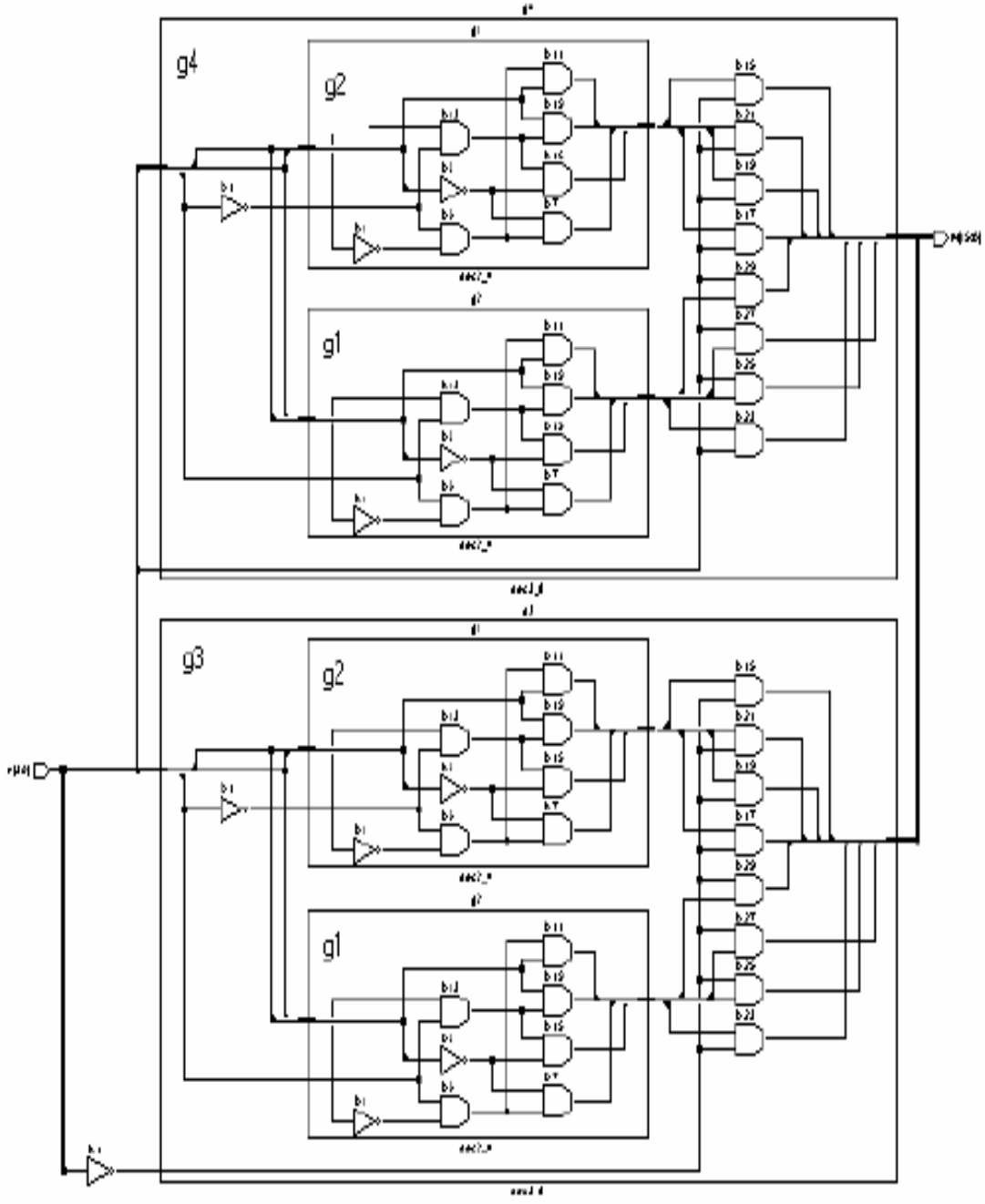


Fig.19 4-to-16 decoder – hierarchy of instantiations

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

Bufiflnn (out, in, control);

The symbol of the buffer is shown in Fig.20.

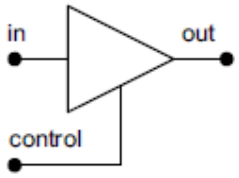


Fig.20 A tri–state Buffer

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.

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 bufifl as well as the other tri-state type primitives are shown in Table 4. In all the cases shown in Table 4, *out* is the output, *in* is the input, and *control*, the control variable.

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 bufifl and bufifO. It is at **H** for notifl and notifO.
  - If the input is at 1 state, the output is **H** for bufifl and bufifO. It is at **L** for notifl and notifO.

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

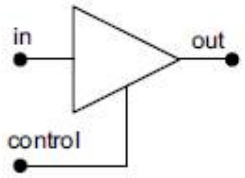
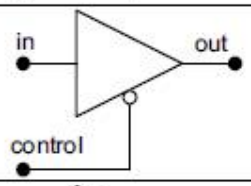
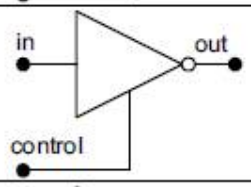
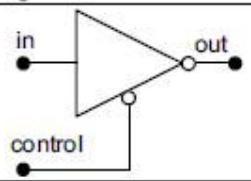
Typical instantiation	Functional representation	Functional description
bufif1 (out, in, control);		Out = in if control = 1; else out = z
bufif0 (out, in, control);		Out = in if control = 0; else out = z
notif1 (out, in, control);		Out = complement of in if control = 1; else out = z
notif0 (out, in, control);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

Table 4 Instantiation and functional details of tri-state buffer primitives

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.

7.1 Example 4: A Byte Comparator

A circuit to compare two variables each of one byte is given in Fig.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 Fig.22 along with a test-bench. The simulated output is in Fig.23.

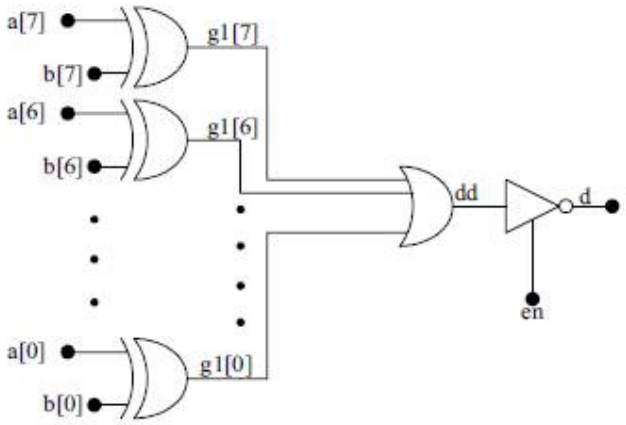


Fig.21 A byte Comparator

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.
- 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.

```
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
```

Fig.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
```

Fig.23 Results of test bench for 8-bit comparator

8. 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.

Example 5: A Simple Latch

Fig.24 shows the design description of a simple latch formed with two NAND gates.

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

Fig.24 A module to instantiate AND gate primitive and test it

A test bench for the same is shown in Fig.25 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 torbandsb 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 Fig.26.

```
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
```

Fig.25. A test bench for the flip-flop and its Simulation results

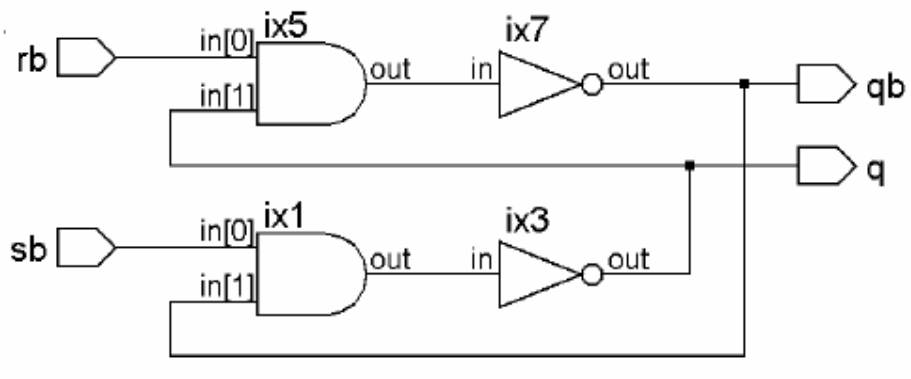


Fig.26 Synthesized circuit of the flip-flop module

Example 6: An RS Flip-Flop

The design module of an RS flip-flop along with a test bench for the same is shown in Fig.27. The module is a slight modification of the flip-flop of Fig.24. The simulation results are shown in Fig.28.

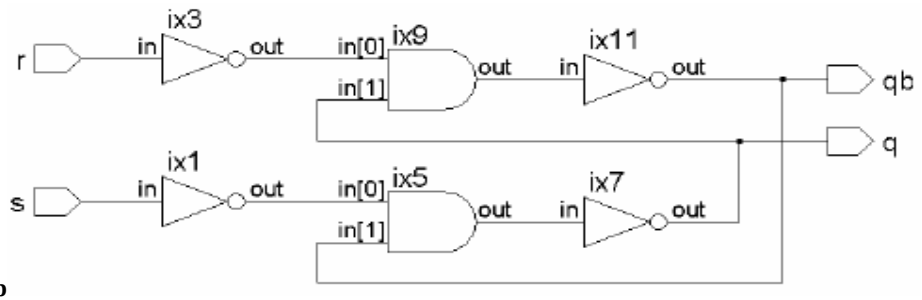
The synthesized circuit is shown in Fig.29. One can easily relate the difference between this circuit and that of Fig.26 to the corresponding difference between the respective design modules.

```
module srff(s,r,q,qb);
input s,r;
output q,qb;
wire ss,rr;
not(ss,s),(rr,r);
nand(q,ss,qb);
nand(qb,rr,q);
endmodule

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

Fig.27 Module of an RS flip-flop with NAND gates and its test bench

```
#  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
```

Fig.28 Results of test bench for RS flip-



flop

Fig.29 Synthesized circuit of the RS flip-flop module

Example 7: A Clocked RS Flip-Flop

The module in Fig.30 is for a clocked RS flip-flop. It is the RS flip-flop of Fig.27 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. The simulation results are shown in Fig.31. Fig.32 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
```

Fig.30 Module of a clocked RS flip-flop using NAND gates and its test bench

```
#  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
```

Fig.31 Results of test bench of clocked RS flip-flop

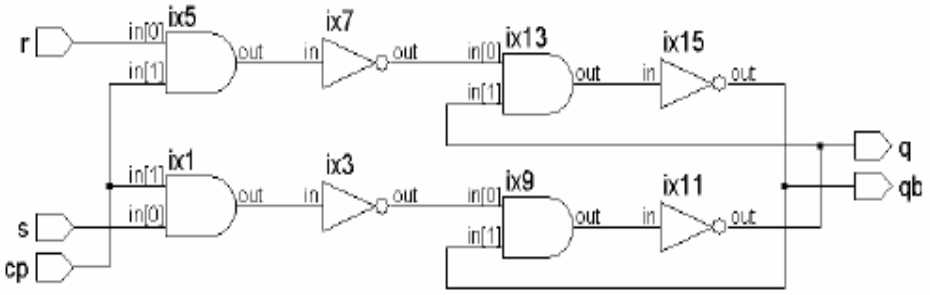


Fig.32 Synthesized circuit of the clocked RS flip-flop module

Example 8: A D-Latch

The design description of a D latch is given in Fig.33. It has one instantiation of the basic flip-flop of Fig.24. A test bench for the latch is also included in the figure. The simulation results are shown in Fig.34. Two versions of the synthesized circuit are shown in Fig.35 and Fig.36, respectively. The basic latch [sbrbff] - which was instantiated in the module of Fig.33 — is shown as a black box in Fig.35. The internals of the latch are shown in Fig.36, 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);
```

```
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
```

Fig.33 Module of D latch and its test bench

```
#  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
```

Fig.34 Results of the test bench for D latch

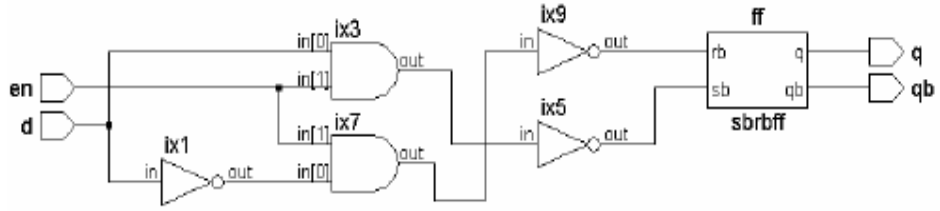


Fig.35 Synthesized circuit of the D latch module

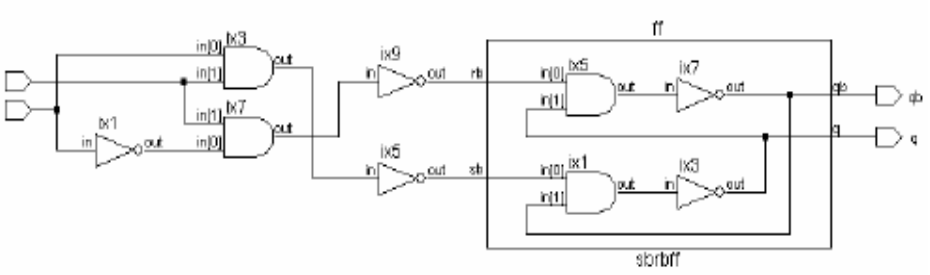


Fig.36 Synthesized circuit of the D latch module showing hierarchy

Example 9: An Edge-Triggered Flip-Flop

Fig.37 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. FF1 and FF2 are minor modifications of FF3. The design modules for FF1 and FF2 are given in Fig.38. 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 Fig.39; the respective waveforms are shown in Fig.40. 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.

Synthesized circuits of the latches FF1 (sbrbffdff) and FF2 (sbrbff 1) are shown in Fig.41 and Fig.42, respectively. The synthesized circuit for the overall flip-flop is shown in Fig.43. 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.The flip-flop of Fig.37can be made comprehensive with slight modifications. It can be replicated and with suitable additions, expanded substantially into register files and full-fledged memory.

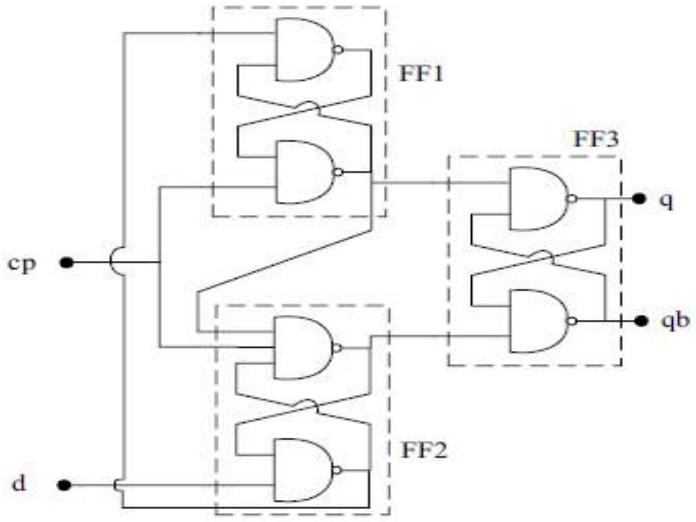


Fig.37 Circuit of a skeletal edge triggered flip-flop

```
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
```

Fig.38 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
```

Fig.39 Results of test bench for edge triggered flip-flop

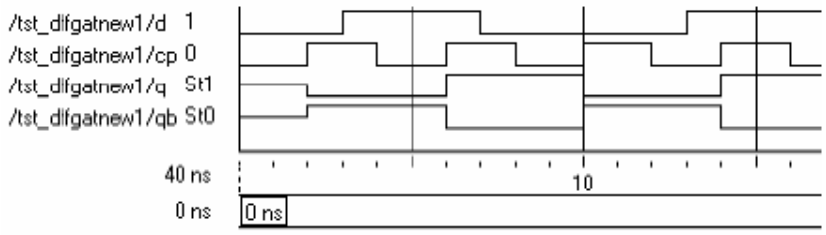


Fig.40 Input and output waveforms for the edge triggered flip-flop module

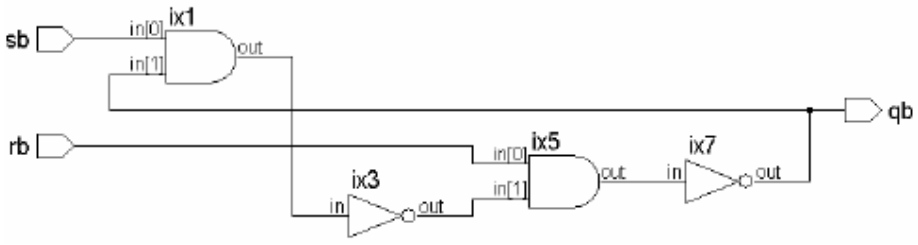


Fig.41 synthesized circuit of the flip-flops brbffdff module

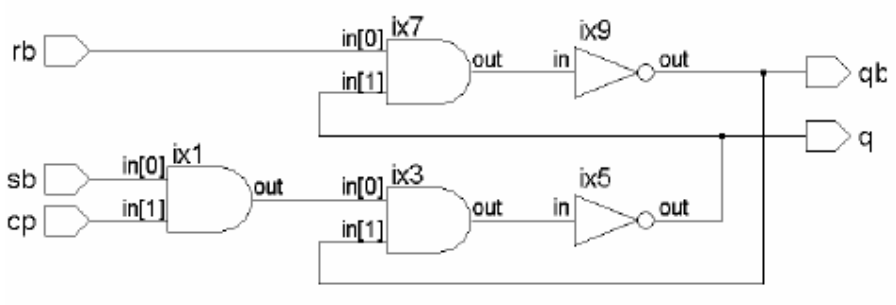


Fig.42 synthesized circuit of the flip-flop sbrbff1 module

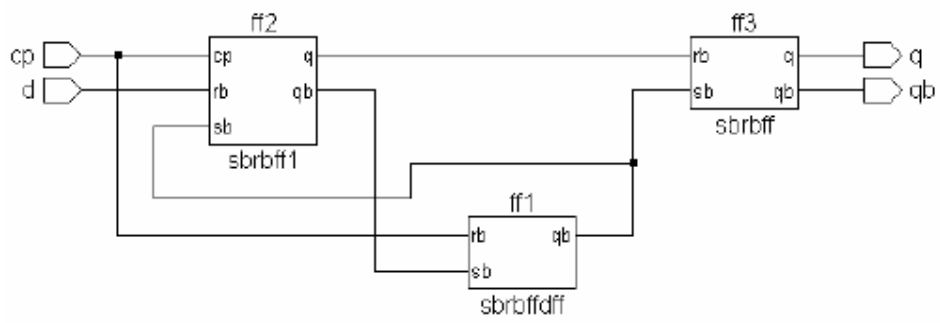


Fig43. Synthesized circuit of the flip-flop dffgatnew1 module

9. 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. 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.

9.1 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 Fig.44 (a), which connects two circuit blocks through a net nn with a delay of 2 time steps associated with it. The module in Fig.45 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 Fig.44 (b), which brings 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(1 to 0) transition

has a delay of 1 time step. The delays are explained in Fig.46. The module of Fig.45 has been modified and shown in Fig.47; the propagation delays are different for rise and fall here.

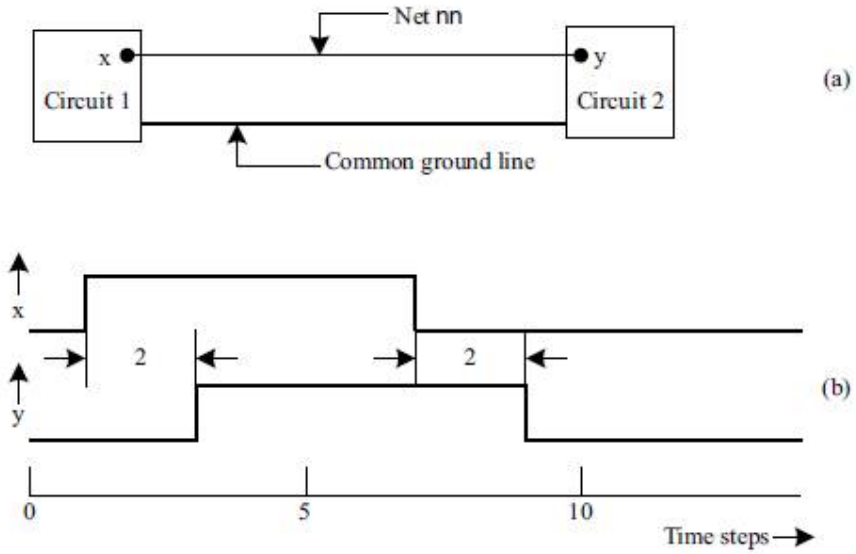


Fig.44A net connecting two circuit blocks and the delay through it
(a) Connection diagram (b) Input and output waveforms

```
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
```

Fig.45 A module to illustrate net delay and its test bench

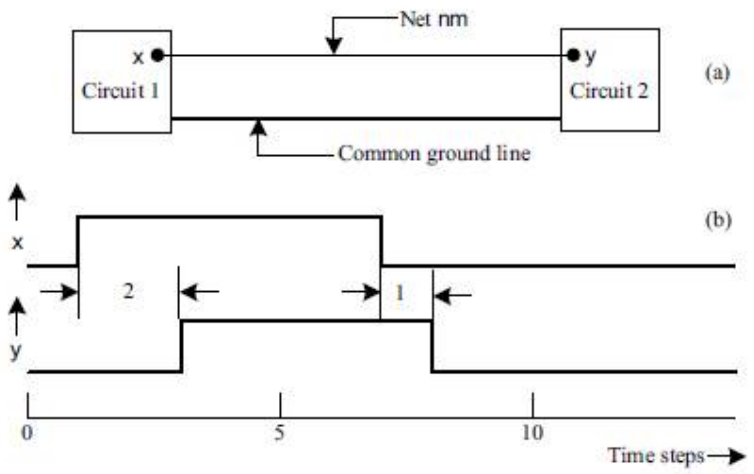


Fig.46 A net connecting two circuit blocks and the delay through it
(a) Connection diagram (b) Input and output waveforms

```
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
```

Fig.47 A module to demonstrate different delays for rise and fall times on net

9.2 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

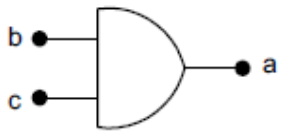
transitions from input to output. A more detailed description can be as follows:

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

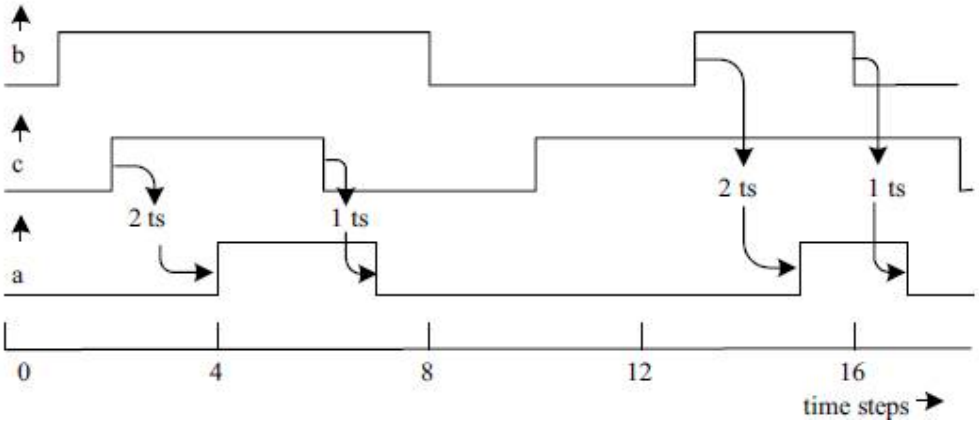
With the above statement the positive (0 to 1) transition at the output has a delay of 2 time steps while the negative (1 to 0) transition has a delay of 1 time step. Fig.48 shows a module to illustrate the delays associated with gate primitives. A test bench for the same is also shown in the figure. The results of running the test bench are shown in Fig.50. The AND gate instantiation in Fig.48 has different delays for the output transitions; respective waveforms are shown in Fig.49.

```
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
```

Fig.48 Module to demonstrate the delays using gates



(a)



(b)

Fig.49 AND gate instantiations with different delays for the positive and negative transitions
(a) Gate instantiated (b) Associated waveforms

```
#  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
```

Fig.50 Results of test bench

In a more detailed design description, delays can be associated with nets as well as gates. Consider the design description shown in Fig.51 (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 Fig.51 (b). Typical waveforms of input signals as well as other signals are shown in Fig.52, to illustrate the different delays in the design description.

Fig.52 (a) and Fig.52 (b) illustrate how changes in one of the inputs - b1 - affect the other signals; the signals and gates affected are shownhighlighted in Fig.52 (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 Fig.52 (b). These conform to the delays specified in the design segment of Fig.51 (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 Fig.52 (c). The waveforms of signals

affected and the associated propagation designs are shown in Fig.52 (d). These too conform to the program segment of Fig.51 (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
```

Fig.51 (a) A design having eight different time delay values

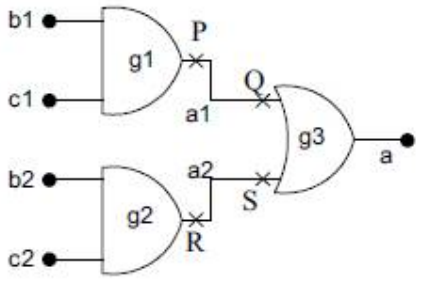
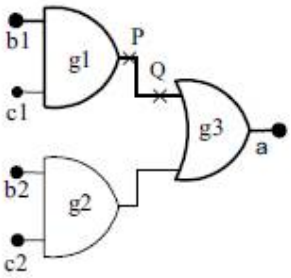
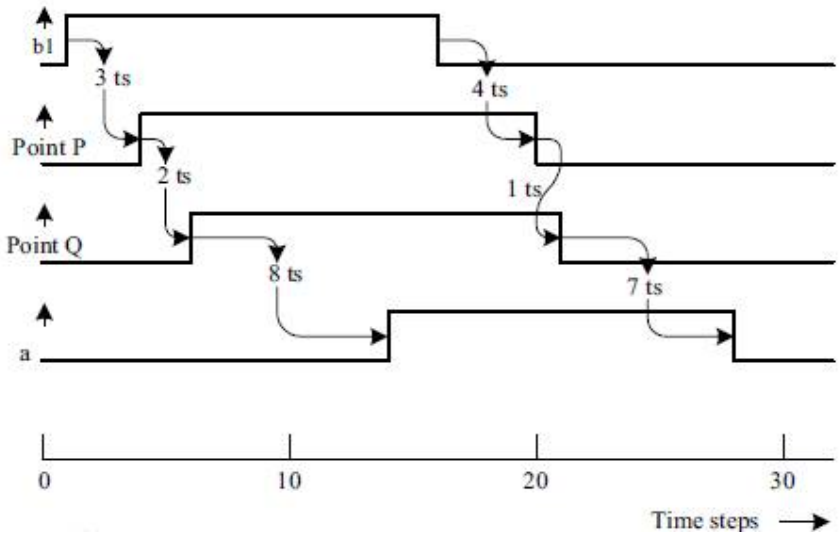


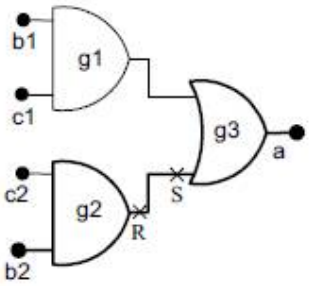
Fig.51 (b) The circuit for the above module


b1
g1
P
Q
c1
g3
a
b2
g2
c2

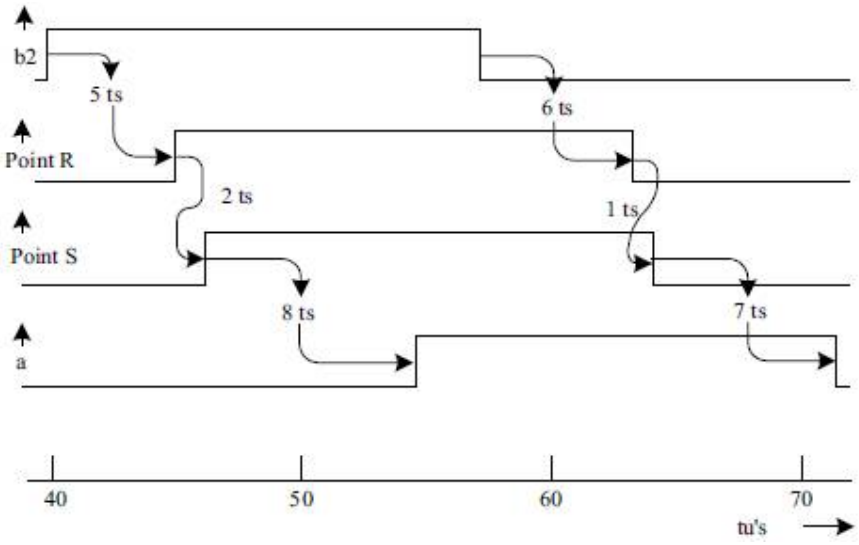
(a)


b1
3 ts
4 ts
Point P
2 ts
1 ts
Point Q
8 ts
7 ts
a
0
10
20
30
Time steps

(b)


b1
g1
c1
g3
a
c2
S
g2
R
b2

(c)



(d)

Fig.52 Illustration of signal delays in the design description segment in Fig.51
(a) The circuit portion active during changes to signal b1
(b) Signal waveforms following changes to signal bl
(c) The circuit portion active during changes to signal b2
(d) Signal waveforms following changes to signal b2

9.3 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 Fig.53 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).

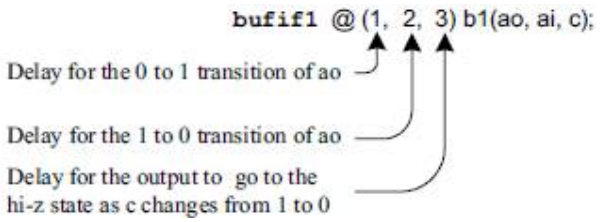


Fig.53 Delays associated with typical tri-state gate

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; /1 The net a has a propagation delay whose minimum, typical and maximum values are 1 ns, 2 ns, and 3 ns, respectively*/

buf if 1 #(1:2:3, 2:4:6, 3:6:9) g3 (aO, bO, cO);

/* 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: The cumulative delay for a signal in a path puts an upper limit on the maximum operating frequency *vis-a-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 reroute selected signals or redo selected design segments to reduce critical path delays.

9.4 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).

10. 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. Signal strength declarations are of two types - those associated with outputs of gate primitives and those with nets.

10.1 Strengths of Gate Primitives

Gate output strengths can be specified separately. Table 5 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.

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

Table 5 Strength levels associated with outputs of gate primitives

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 Fig.54 for a specific case; the format remains the same for all types of gate primitives.

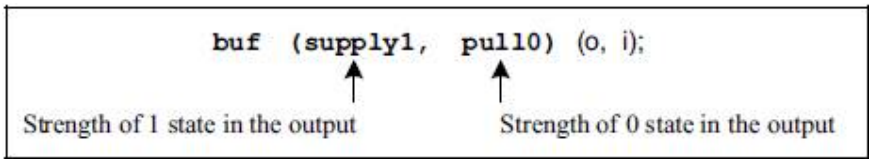


Fig.54 Format for specifying strengths in the instantiation of a gate primitive

10.2 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 10: Strength Contention

Consider the module in Fig.55. The logic levels taken by the signal O for different combinations of inputs to the two buffers g1 and g2 are shown in Table 6. Contentions of signals with other combinations of levels can be resolved in the same manner.

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

Table 6 Outputs for different inputs for the example of Fig.55

```
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
```

Fig.55 (a) A module to illustrate strength contention

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.

```
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
```

Fig.55 (b) Output of module in Fig.55 (a)

Consider the Example in Fig.56, which is a slightly modified version of that in Fig.55. The output logic values for different input combinations are given in Table 7. 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
```

Fig.56 Illustration of strength contention resulting in x-type output and simulation results

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

Table 7 Outputs for different inputs in the example of Fig.56

10.3 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 8. 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.

Name	Large	Medium	Small
Strength	Strongest		Weakest

Table 8 Capacitive storage strengths on nets

Example 11: Net Storage

Consider the design in Fig.57. 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
```

Fig.57 Illustration of net storage and simulation results

10.4 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 9 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.

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 9 Signal strength names and their relative weights

10.5 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 10 gives details. As can be seen from the table, whenever the two net types lead to a logical clash, the external data type prevails.

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

Table 10 Net assignments with port connections

11. 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.

11.1 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 Fig.56 for which the input-output values are shown in Table 7. 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.

Example 12: Illustration of wand type net

Fig.58 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 11 also. Contention can be seen to be resolved in two possible ways:

- When i 1 = 1 and i 2 = 0, the stronger signal i 1 at the 1 level prevails and o = 1. The contention is resolved according to the strengths.
- When il = 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 Fig.59; the circuit translates into an AND gate which is erroneous.

```
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
```

(a)

```
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
```

(b)

Fig.58 Illustration of use of the wand-type net and simulation results

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

Table 11 Output values for different inputs of the design in Fig.58

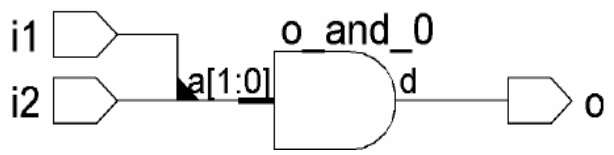


Fig.59 Synthesized version of the module with the wand-type net

Example 13: Illustration of wor-type net

Consider the design segment in Fig.58 with o being declared as a **wor** type of net instead of a **wand** type. The corresponding design module is shown in Fig.60. A test bench and simulation results are also shown in the figure. The outputs for all possible combinations of inputs are given in Table 12. Contention can be seen to be resolved in two possible ways:

- When il = 1 and i2 = 0, the stronger signal il at the 1 level prevails and o = 1. The contention is resolved according to the strengths.
- When il = 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 Fig.61; the circuit translates into an OR gate; this is consistent with the desired input-output relationship shown in Table 12.

```
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
```

(a)

```
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
```

(b)

Fig. 60 Illustration of use of the wor-type net and simulation results

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

Table 12 Output values for different inputs of the design in Fig.60

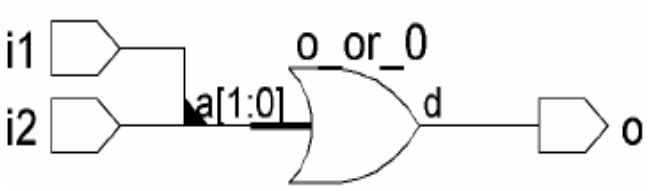


Fig.59 Synthesized version of the module with the wor-type net

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, triO,** and **tril** (discussed below) may not be supported by some.

11.2 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 14: Illustration of tri-type net

Consider the design segment in Fig.62. 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);
...
```

Fig.62 A segment of a design to illustrate tri type of net

11.3 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 Fig.63, which shows a design segment, illustrates an application. Table 13 lists the output values of signals considered in the design segment of Fig.63.

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, g 1, 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);
...
```

Fig. 63 A segment of a design to illustrate tri0 and tri1 types of net

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

Table 13 Output values for different inputs of the segment in Fig.63

11.4 supply0 and supply1

Supply0 and **supply1** are the keywords signifying the high- and low-side supplies. Nets to be connected to the V_{cc} supply are declared as **supply1,** and those to be grounded are declared as **supply0.**

11.5 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.

11.6 Combining Delays & Strengths

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.

12. 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. 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 15: 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.

```
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;
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
```

Fig.64 a 4-bit adder module and its test bench

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. Fig.64 shows a 4-bit adder module and a test-bench for it. The simulation results are given in Fig.65. The adder module is built up by repeated instantiation of the full-adder module. The synthesized version of the adder is shown in Fig.66. The full-adder module instantiations appear here as black boxes with respective inputs and outputs.

```
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
```

Fig.65 Simulation results of Fig.64

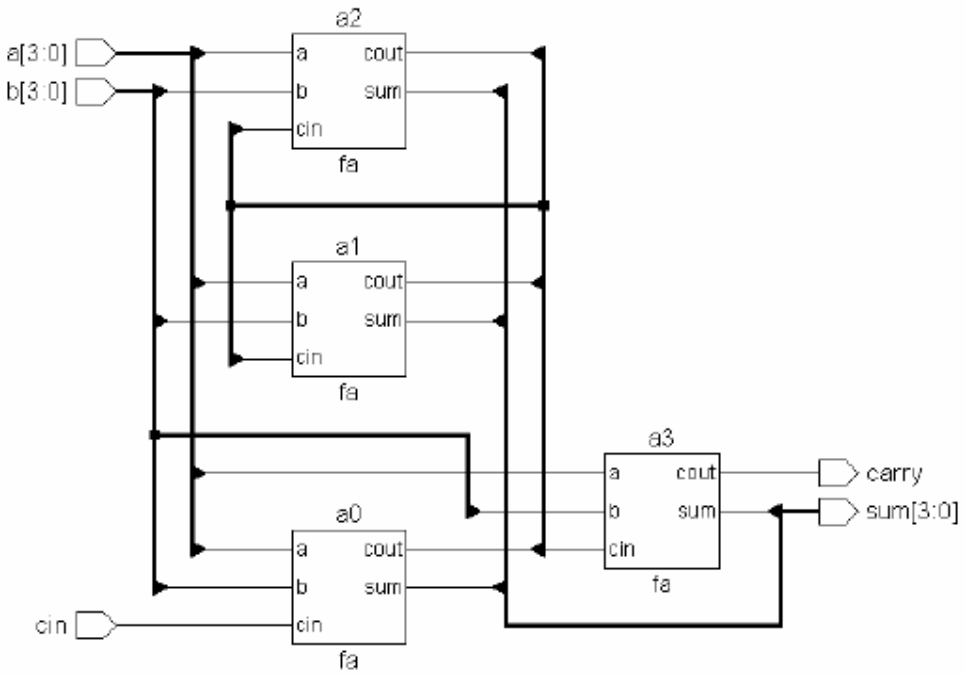


Fig.66 Synthesized circuit of the adder module

Fig.67 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 Fig.68.

```
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
```

Fig.67 A 4-bit Adder Module

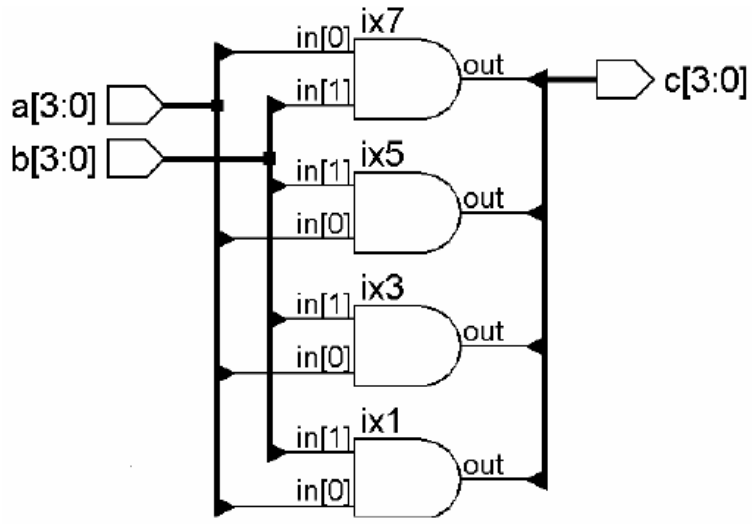


Fig.68 Synthesized circuit of the andg4 module

The module in Fig.69 carries out the bit-wise XOR operation on 2 nibbles. Its synthesized circuit is shown in Fig.70. Similarly, the module in Fig.71 complements 2 nibbles in a bit-wise manner. The corresponding synthesized circuit is shown in Fig.72.

```
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
```

Fig.69 A 4-bit XOR module

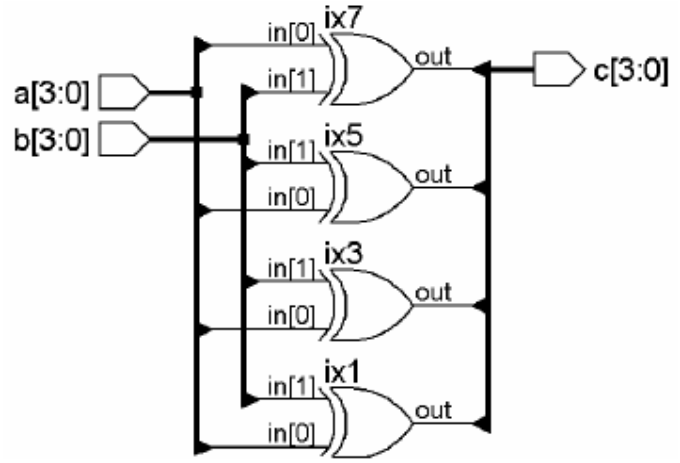


Fig.70 Synthesized circuit of the XOR module

```
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
```

Fig.71 A module to complement a 4-bit vector

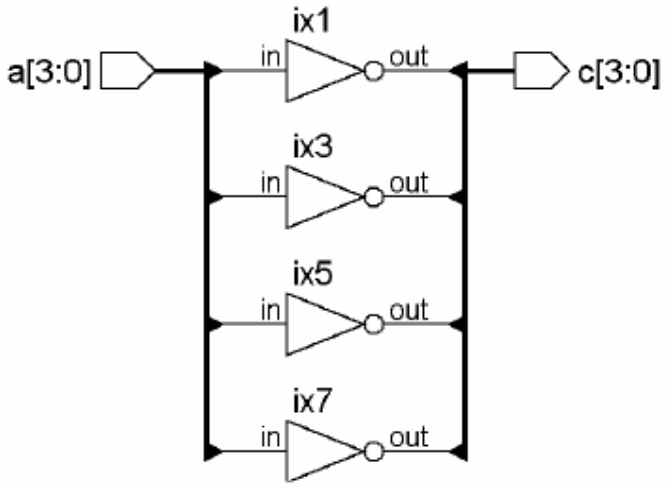


Fig.72 Synthesized circuit of the module in Fig.71

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 Fig.73, and its synthesized circuit is shown in Fig.74.

```
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
```

Fig.73 A 2-to-4 decoder module

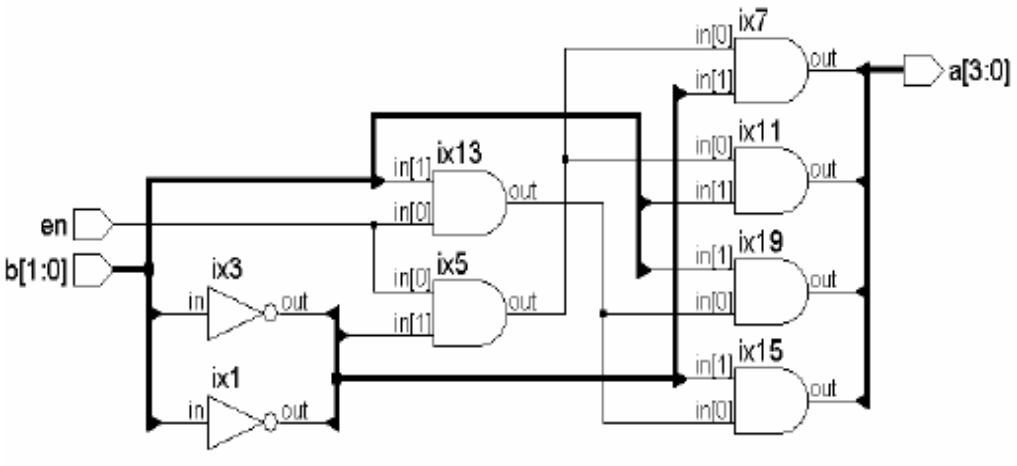


Fig.74 Synthesized circuit of the decoder module

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-l mux serves this purpose. The mux module is shown in Fig.75; its synthesized circuit is in Fig.76.

```
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
```

Fig.75 A 4-to-1 mux module

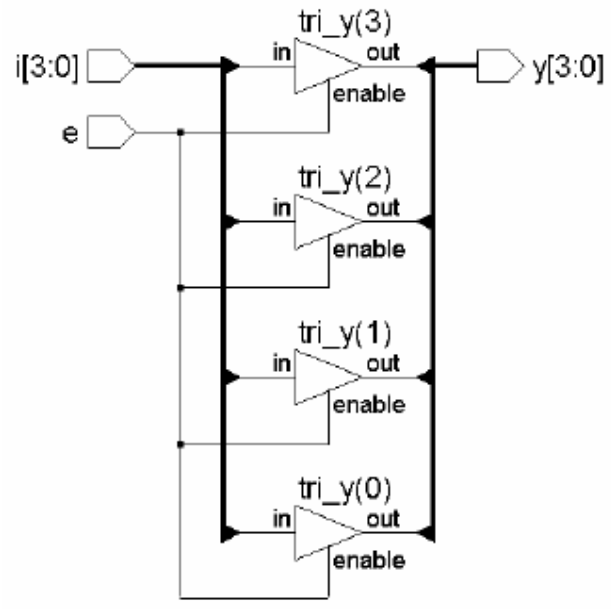


Fig.76 Synthesized circuit of the mux module

The overall ALU module is shown in Fig.77. 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-l mux. The ALU module here has been synthesized and shown in Fig.78. Each functional block instantiated in Fig.77 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 of the 4-bit module in a more comprehensive module.

```
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);
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
```

Fig.77 A 4-bit ALU module

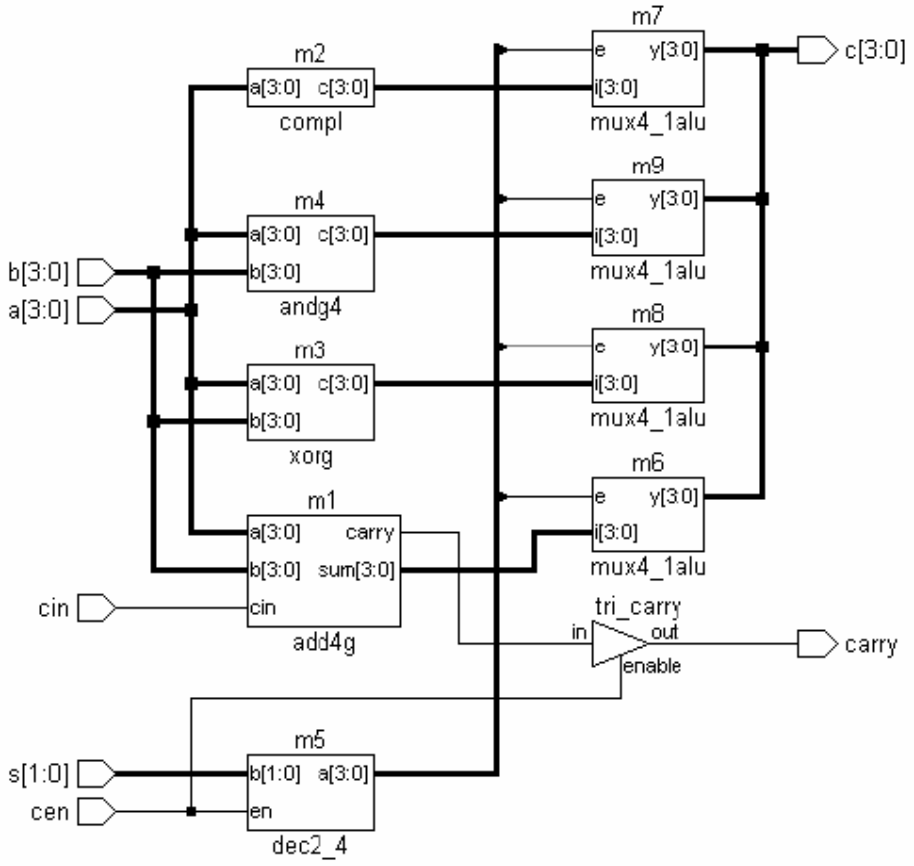


Fig.78 Synthesized circuit of the ALU module

MODELING AT DATAFLOW LEVEL

13. INTRODUCTION

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

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

13. CONTINUOUS ASSIGNMENT STRUCTURES

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

assign C = a && b;

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

- a and b are operands - typically single-bit logic variables.
- "**&&**" is a logic operator. It does the bit-wise AND operation on the two operands a and b.
- "**=**" is an assignment activity carried out.
- **C** is a net representing the signal which is the result of the assignment. In general, an operand can be of any one of the following types:
- A constant number [including real].
- Net of a scalar or vector type including part of a vector.
- Register variable of a scalar or vector type including part of a vector.
- Memory element.
- A call to a function that returns any of the above. The function itself can be a user-defined or of a system type.

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

An AND gate module which uses the above assignment is shown in Fig.79. The test bench for the same is shown in Fig.80, and the waveforms of nets a, b, and c obtained with the simulation are shown in Fig.81. The simulation software used has the facility to capture the waveforms of selected signals in the "ran" phase; this has been invoked to get the waveforms in Figure 6.3. No separate **$monitor** command is included in the test bench of Fig.80.

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

Fig.79 A module with an AND gate at the data flow level

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

Fig.80 Test bench of Fig.79

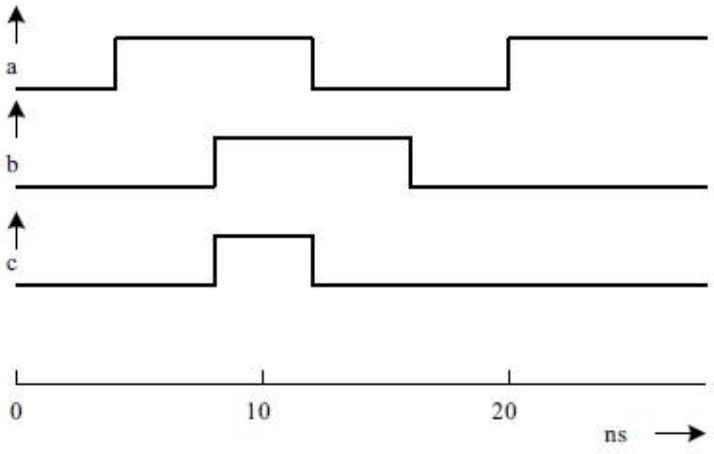


Fig.81 Input and Output Waveforms

Multiple assignments can be carried out through a direct extension of the structure adopted in the above case. Consider the AOI gate in Fig.82. A few patterns of the assignments for the circuit are given in Fig.83 to Fig.84.

Multiple assignments can be carried out through a direct extension of the structure adopted in the above case. Consider the AOI gate in Fig.82. A few patterns of the assignments for the circuit are given in Fig.83 to Fig.85.

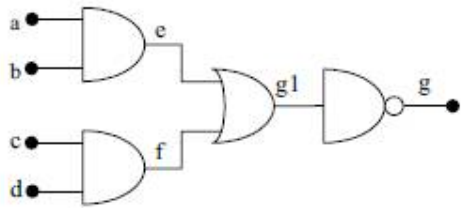


Fig.82 An A-O-I Gate

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

Fig.83 Data flow level assignment statements for A-O-I gate

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

Fig.84 Another set of data flow level assignments for A-O-I gate

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

Fig.85 Another set of data flow level assignment for A-O-I gate

Observations:

- The semicolon terminates an assignment statement. Commas separate different assignments in an assignment statement.
- "|" is the bit-wise OR operator and the bit-wise negation operator in Verilog.
- All the quantities in the left-hand side of a continuous assignment have to be of net type. Thus e, f, g, and gl have to be declared as nets.
- All the operations in an assignment are evaluated whenever any of the operands in the assignment changes value. Further, all the assignments are carried out concurrently. Hence the order of the assignments or the statements is immaterial.
- The right-hand sides of assignment statements can be nets, legs, or function calls. Here a, b, c, and d can be nets or legs. All other variables have to be nets.

The module for the A-O-I gate of Fig.79 is given in Fig.86 - it is formed around the assignment statement of Fig.83. The same can be tested through a test bench.

13.1 Combining Assignment and Net Declarations

The assignment statement can be combined with the net declaration itself making the assignment implicit in the net declaration itself. Thus the two statements

wire C;

assign C **= *a & b;***

Can be combined as **wire** C = a & b;

The above simplification cannot be carried over to multiple declarations. With this proviso, the module of Fig.86 can be modified as shown in Fig.87. In the modules of Fig.86 and Fig.87, a, b, c, and d are declared as **input** and g as **output.** These would be taken as nets if there are no separate declarations concerning their types. However, the intermediate quantities - e, f, and g1— should be declared as **wire.** Synthesized version of the A-O-I circuit is shown in Fig.88.

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

Fig.86 **A compact description of the AOI module at the data flow level**

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

Fig.87 **Alternate design module to realize the A-O-I gate**

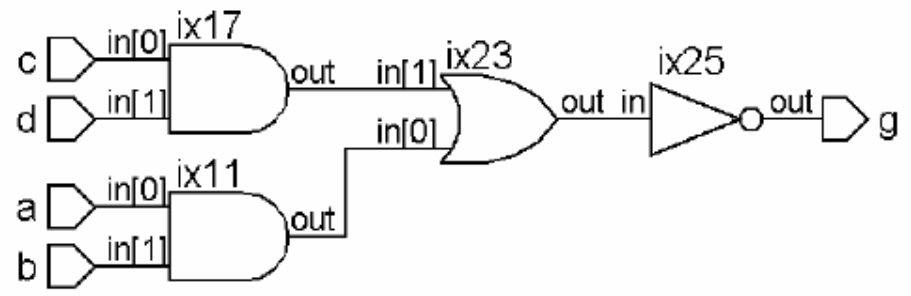


Fig.88 Synthesized **circuit of the A-O-I gate module**

13.2 Continuous Assignments and Strengths

A net to which a continuous assignment is being made can be assigned strengths for its logic levels. The procedure is akin to the strength allocation to the outputs of primitives. The AOI gate of Fig.87 is modified with strength allocations to the output and is shown in Fig.89. The assignment to g can be combined with the wire declaration into a single statement as

wire (pulll, strongO)g = ~g1;

As mentioned earlier, one can have only one assignment in the statement here. In a bigger design, g in Fig.89 can be assigned to other expressions or primitives also.

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

Fig.89 The A-O-I gate with strength allocation

14. DELAYS AND CONTINUOUS ASSIGNMENTS

Delays can be incorporated at the data flow level in different ways. Consider the combination of statements in Fig.90. The assignment takes effect with a time delay of 2 time steps. If a or b changes in value, the program waits for 2 time steps, computes the value of c based on the values of a and b at the time of computation, and assigns it to c. In the interim period, a or b may change further, but c changes and takes the new value only 2 time steps after the change in a or b initiates it. Typical waveforms for a, b, and c are shown in Fig.91. Note that the changes in a and b of duration less than 2 time steps are ignored *vis-a-vis* assignment to the net c. The following may be noted with respect to the waveforms:

- a changes at 0 ns, 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 0 ns, 2 ns, 6 ns, 8 ns and 13 ns. All these trigger changes to c.
- In every case change to C comes into effect with a time delay of 2 time steps - that is, at the 2nd, 4th, 7th, 8th, 10th, 11th, 14th and 15th ns, respectively.
- Whenever C changes, its new value is decided by the values of a and b at that instant of time. In effect, c changes at 2nd, 4th and 7th ns only.

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

Fig.90 Delays with assignments

The program segment in Fig.91 also gives the same output as shown in Fig.92.

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

Fig.91 Alternate design description for delays with assignments

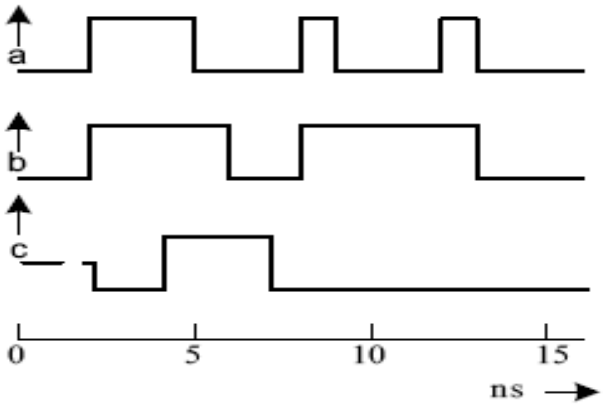


Fig.92 Input and Output Waveforms

If the time delay is in the net and not in the assignment proper, its effect is not any different. Consider the program segment in Fig.93. Here the changes in the values of **d** are computed immediately following those in **a** and **b.** The assignment takes effect immediately. The delay in the net **C** causes a delay of 2 time steps in the assignment to **C.** Such a delay is not present for **d.** Typical waveforms for the program segment are shown in Fig.94. Note the following:

- **a** changes at 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 2 ns, 6 ns, 8 ns and 13 ns. All these trigger changes to **C** and **d** also.
- In every case, change to **C** comes into effect with a time delay of 2 time steps - that is, in effect, c changes at 2nd, 4th and 7th ns only.
- Whenever **C** changes, its new value is decided by the values of **a** and **b** at that instant of time.
- In every case, changes to d come into effect immediately.

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

Fig.93 Combining delays with assignments

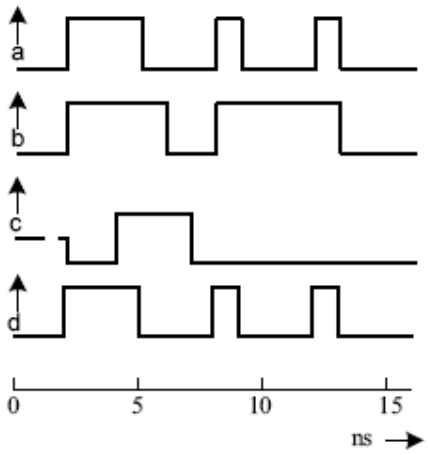


Fig.94 Input and output waveforms

15. ASSIGNMENT TO VECTORS

The continuous assignments are equally applicable to vectors. A single statement can describe operations involving vectors wherever possible. This is illustrated in the adder module in Fig.95, which adds two 8-bit numbers. Here it is assumed that the sum is also of 8 bits. However to account for the possibility of a carry bit being generated in the course of the addition process, it is desirable to increase the vector size of c by one bit.

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

Fig.95 An Adder module at data flow level

15.1 Concatenation of Vectors

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

{a, b, c]

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

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

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

Example 16: Eight-Bit Adder

Fig.96 shows the design description of an 8-bit adder, where the output vector is formed directly by concatenation. The adder takes a carry input and gives out a carry output. The adder module here can form the "seed" adder block in a multi- byte adder chain.

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

Fig.96 A complete 8-bit adder module at data flow level

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

{2{p}} represents the concatenated vector **{P,P}**

{2{p}, q} represents the concatenated vector ***{p, p, q}***

The two statements

assign GND=supply0;

p={8{GND}};

together ground the 8 bits of the vector p.

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

{a, 3 {2{b , c}, d}}

is equivalent to the vector

{a, b, c, b, c, d, b, c, b, c, d, b, c, b, c, d }

16. OPERATORS

A set of operators is available in Verilog. The operator symbols are similar to those in C language. With these operators we can carry out specified operations on the operands and assign the results to a net or a vector set of nets as the case may be.

16.1 Unary Operators

Unary operators do an operation on a single operand and assign the result to the specified net. The unary operators in Verilog are given in Table 6.1. All unary operators get precedence over binary and ternary operators. The operators "+" and preceding an integer or a real number change its sign. These are also unary operators, though not separately listed in Table 14.

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

Table 14 Unary Operators and their Symbols

16.2 Binary Operators

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

- The arithmetic operators treat both the operands as numbers and return the result as a number.
- All net and **reg** operand values are treated as unsigned numbers.
- Real and integer operands may be signed quantities.
- If either of the operand values has a zero value, the entire result has a zero value (?).

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

16.2.1 Arithmetic Operators

The arithmetic operators of the binary type are given in Table 15. The modulus operand is similar to that in C language - It provides the remainder of the division of two numbers. The module in Fig.95 is an example of the illustration of the use of the arithmetic binary operator "+" (for addition). Other arithmetic operators are also used in a similar manner.

Operand type	Symbol	Remarks
Multiplication	*	
Division	/	The result is x if the denominator is zero
Modulus	%	
Addition	+	
Subtraction	–	

Table15 Arithmetic Operators and their Symbols

Observations:

- In integer division the fractional part of the result is truncated and ignored.
- If any bit of an operand is **x** or **z** in an arithmetic operation, the result takes the **x** value.
- If the first operand of a modulus operator is negative, the result is also a negative number.

Depending on the type of definition of a number, a modulus operation can lead to different results. Typical examples are given in Table 16.

Expressions involving modulus operator	Result of the operation	Remarks
15 % 5	0	Results are obvious
14 % 5	4	
4'hf % 5	0	The numbers 4'hf and 4'he are in hex format with decimal values of 15 and 14, respectively. But the denominator 5 is in decimal form.
4'he % 5	4	
6'o15 % 5	3	6'o15 is an octal number with a decimal value of 13.
–4 % 3	–1	
4 % –3		Illegal form

Table 16 Typical modulus operations and their results

16.2.2 Logical Operators

There are two logical operators involving two operands. The operands concerned can be variables or expressions involving variables. In both cases the result of the operation is a single bit of value 1 (true) or 0 (false). If a bit in one of the operands is x or z, the result of evaluation of the expression has an **x** value. The operator details are shown in Table 17. The modules in Fig.86 and Fig.87 are examples of the illustration of the use of logical binary operators.

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

Table 17 Binary Logical Operators and their Symbols

16.2.3 Relational Operators

There are four relational operators; their details are shown in Table 18. A relational operator treats both the operands as binary numbers and compares them. The result is a 1 (true) bit or a 0 (false) bit. If a bit in either operand is **x** or **z,** the result has **x** (unknown) value. The operands can be variables or expressions involving variables. Operands of net or **reg** type are treated as unsigned numbers. Real and integers can be positive or negative *(i.e.,* signed) numbers.

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

Table 18 Relational Operators and their Symbols

16.2.4 Equality Operators

The equality operator makes a bit-by-bit comparison of the two operands and produces a result bit. The result bit is a 1 (true) if the operand condition is satisfied; otherwise it is 0 (false). The operands can be variables or expressions involving variables. If the operands are of unequal length, the shorter one is zero filled to match the larger operand. The operators in this category are only of two types - those to test the equality and those to test inequality. The four operators in this category are given in Table 19.

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

Table 19 Equality Operators and their Symbols

16.2.5 Bit-wise Logical Operators

The operator does a specified bit-by-bit operation on the two operands and produces a set of result bits. The result is (bit-wise) as wide as the wider operand. If the width of one of the operands is less than that of the other, it is bit-extended by filling zero bits and the widths are matched. Subsequently, the specified operation is carried out. If one of the operands has an **x** or **z** bit in it, the corresponding result bit is **x.** Either operand can be a single variable or an expression involving variables. Table 20 gives the four operators of this category.

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

Table 20 Logical Operators and their Symbols

16.2.6 Operator Truth Table

The truth tables for different types of bit-wise operators are given in Table 21. Note that an **z** input is treated as an **x** value.

AND

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

OR

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

XOR

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

XNOR

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

Negation

Input	0	1	x
Output	1	0	x

Table 21 Truth Tables for Bit-wise Operators

16.2.7 Shift Operators

Table 22 shows the two operators of this category. The « operator executes left shift operation, while the » operator executes the right shift operation. In either case the operand specified on the left is shifted by the number of bits specified on the right. The shifting is done irrespective of whether the bits are 0, 1, **x,** or **z.** The bits shifted out are lost. The vacated positions created as a result of the shifting are filled with zeroes. If the right operand is **x** or **z,** the result has an x value. If the right operand is negative, the left operand remains unchanged.

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

Table 22 Shift type Operators and their Symbols

16.3 Ternary Operator

Verilog has only one ternary operator - the conditional operator. It checks a condition and does a branching. It is a versatile and powerful operator. It enhances the potential of design description substantially. The general form is

A?b:c

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

- "A" is evaluated first.
- If A is true, **b** is evaluated.
- If A is false, **c** is evaluated.

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

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

As an example, consider the assignment statement

assign y = W ? X : z;

where **W, X, y** and **Z** are binary bits. If the bit **W** is true **(1), y** is assigned the value of **X**: otherwise - that is, if **W** is false (0) - **y** is assigned the value of **Z**. The assignment statement here multiplexes **X** and **Z** onto **y**; **W** is the control signal here. Consider the assignment

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

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

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

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

assign C = S ? a: b;

The net **C** is connected to **a** if **S=1**; else it is connected to **b**

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

The conditional operator can be nested. Nesting gives rise to a variety of uses of the operator.

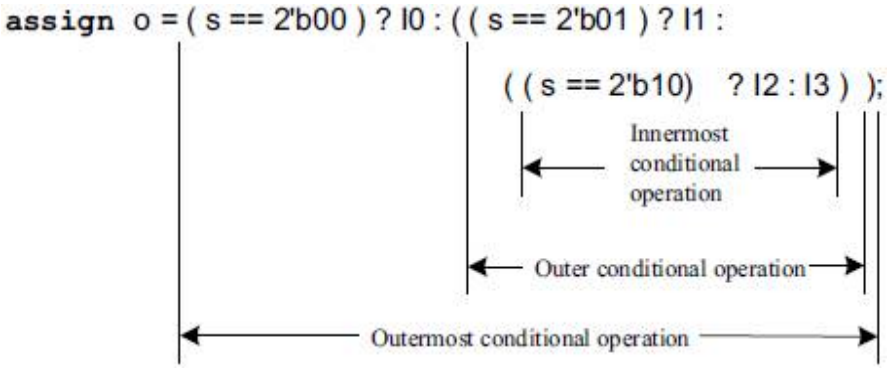


Fig.97 Illustration of nested conditional operations

As an example, consider the formation of an ALU. ALU can be defined in a compact manner using the ternary operator.

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

In the example here, f is taken as a control word. If it is equal to the number add, d is to be equal to the sum of a and b. If f is equal to the number subtract, d is to be equal to the difference between a and b. If it is equal to the number compl, d is to be the complement of a. Otherwise (i.e., f = 3) d is taken as the complement of b.

16.4 Operator Priority

A clear understanding of the operator precedence makes room for a compact design description. But it may lead to ambiguity and to inadvertent errors. Whenever one is not sure of the operator priorities, it is better to resort to the use of parentheses and ensure clarity and accuracy of expressions. Further, some synthesizers may not interpret the operator precedence properly. These too call for the apt use of parentheses.

The operators are arranged in tabular form and shown in Table 23. The table brings out the order of precedence. The order of precedence decides the priority for sequence of execution and circuit realization in any assignment statement. The following form the basic rules for the same:

- Unary operators have the highest priority and execute first.
- Subsequently the binary operators execute. Amongst these the algebraic operators have the highest precedence. Amongst the algebraic operators *, / and % have precedence over + and - operators.
- Subsequent precedence amongst the binary operators is as shown in the table.
- Conditional operator has the lowest precedence and hence is executed last.
- In any expression, operators associate from left to right. Ternary operator is the only exception to this; it associates from right to left.

Unary operators	! & ~& | ~| ^ ~^ + –	Highest precedence
Binary operator	* ? /	
	+ -	
	<< >>	
	< <= > >=	
	== != === !==	
	& ^ ~^	
	|	
	&&	
	||	
Ternary operators	? :	Lowest precedence

Table 23 Operator Precedence Details

Example 16: BCD Adder

```
module bcd(co,sumd,a,b);
input [3:0]a,b;
output [3:0]sumd;
output co;
wire [3:0]sumb;
assign sumb = a + b;
assign{co,sumd}=(sumb<=4'b1001)?{1'b0,sumb}:(sumb+4'b01
10);
endmodule

module tst_bcd;//Test bench
reg [3:0]a,b;
wire co;
wire [3:0]sumd;
bcd bcc(co,sumd,a,b);
initial
    begin
        a = 4'h0 ; b = 4'h0;
    #2  a = 4'h1 ; b = 4'h0;
    #2  a = 4'h2 ; b = 4'h1;
    #2  a = 4'h4 ; b = 4'h5;
    #2  a = 4'h6 ; b = 4'h6;
    #2  a = 4'hd ; b = 4'h1;
    #2  a = 4'hf ; b = 4'h0;
    end
initial $monitor($time,"a = %b, b = %b, co = %b, sumd =
%b",a,b,co,sumd);
initial #16 $stop;
endmodule
```

Fig.98 BCD Adder Module at the Data flow module

A BCD adder can be formed through a compact assignment using a ternary operator. The assignment statement has the form

assign {co, sumd} = (sumb<=4'b1001)?{1 'b0,sumb}: (sumb + 4'b0110;

The adder module using the above assignment and a test-bench for the same are shown in Fig.98. The synthesized version of the circuit is shown in Fig.99. The results of running the test bench are given in Fig.100.

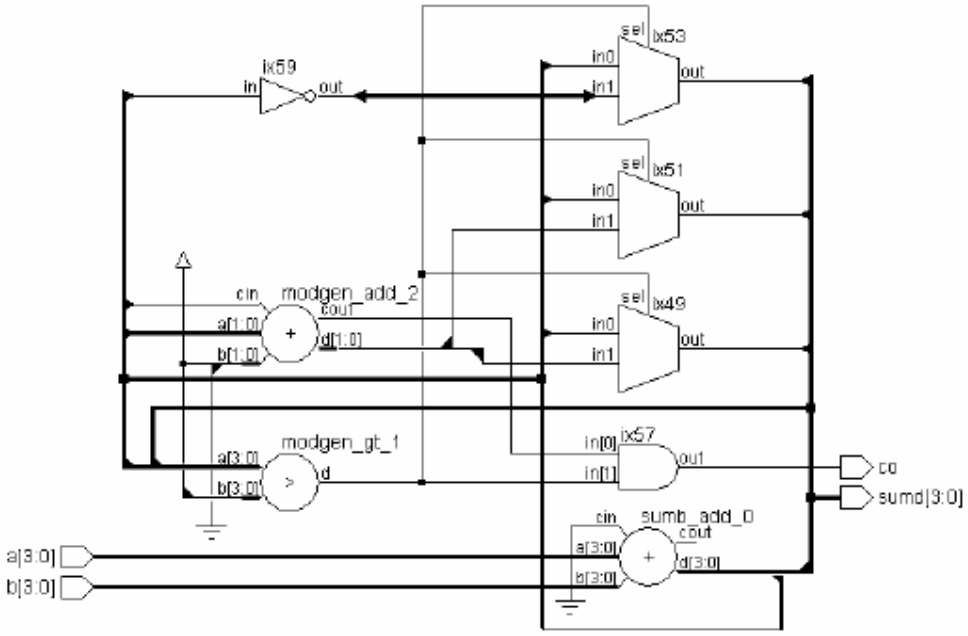


Fig.99 Synthesized circuit of the BCD Adder

```
# 0 a = 0000 , b = 0000 ,  co = 0 , sumd = 0000
# 2 a = 0001 , b = 0000 ,  co = 0 , sumd = 0001
# 4 a = 0010 , b = 0001 ,  co = 0 , sumd = 0011
# 6 a = 0100 , b = 0101 ,  co = 0 , sumd = 1001
# 8 a = 0110 , b = 0110 ,  co = 1 , sumd = 0010
#10 a = 1101 , b = 0001 ,  co = 1 , sumd = 0100
#12 a = 1111 , b = 0000 ,  co = 1 , sumd = 0101
```

Fig.100 Results of BCD Adder Module

UNIT-III

BEHAVIORAL MODELING

Contents

- Introduction
- Operations and Assignments
- Functional Bifurcation
- 'Initial' Construct
- 'Always' Construct
- Examples
- Assignments with Delays
- 'Wait' Construct
- Multiple Always Blocks
- Designs at Behavioral Level
- Blocking and Non-Blocking Assignments
- The case statement
- Simulation Flow
- 'if' and 'if-else' constructs
- 'assign – de-assign' construct
- 'repeat' construct
- 'for' loop
- the 'disable' construct
- 'while' loop
- 'forever' loop
- parallel blocks

- ‘force-release’ construct
- Event

UNIT – III

BEHAVIORAL MODELING

1. INTRODUCTION

Design descriptions at data flow level and gate level are close to the circuit. At every stage of the design description process, one can relate the modules and the instantiations with the corresponding logic or sequential blocks and their interconnections. The approach is practical and effective as long as the gate count remains within a few hundred. An increase in gate count may still be accommodated, if it is due to an increase in vector size -for example, when a system designed and tested at the 8-bit level is being scaled up to a 16- or 32-bit level. But with many of the VLSI's of today, one has to work at a different dimension - the circuit can have a million gates. The increase in vector size may still be accommodated at the data flow level (e.g., 32- or 64-bit systems), since it calls only for scaling of a smaller design. But increase in terms of functional complexity makes the approach almost intractable for many designs.

Behavioral level modeling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behavior; it can be described at a functional level itself instead of getting bogged down with implementation details. The description is carried out essentially with constructs similar to those in "C" language; the design itself is similar to programming in "C", For example, one can describe an FFT or a digital filter routine in terms of these constructs. The design can be simulated, debugged, and finalized. This completes the system level structure for the design. Subsequently, one can expand the design by describing the modules in terms of components closer to the data flow and gate level models. One can simulate and debug each such component module, check it for its functionality, integrate it with the main design and test conformity. Constructs for such layered expansion of design are available in behavioral modeling. Proceeding with the layered expansion of design, one can have the final design description at the RTL level itself. However, we may add here that such a top-down activity is more in the realm of design.

The constructs available in behavioral modeling aim at the system level description. Here direct description of the design is not a primary consideration in the Verilog standard. Rather, flexibility and versatility in describing the design are in focus. One should be able to describe the design and simulate it for its functionality. Hence the constructs aim essentially at these two aspects of the design. Synthesis tools available from different vendors can synthesize most of the constructs at the data flow as well as the gate levels, but not all constructs or combinations possible at the behavioral level can be synthesized. The extent to which the constructs at the behavioral level are accommodated in synthesis varies with vendors. The synthesized circuit need not guarantee optimum or near- optimum realization either. These limitations are in line with the basic purpose of behavioral level modeling mentioned above - that is, to complete an error or bug- free description and identify the functional modules required. Their synthesis is more often done following a more detailed design description at the RTL level.

2. OPERATIONS AND ASSIGNMENTS

The design description at the behavioral level is done through a sequence of assignments. These are called 'procedural assignments' - in contrast to the continuous assignments at the data flow level.

The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "=" sign - for example,

  N= ~N;

  Here the content of **reg *N*** is complemented and assigned to the reg *N* itself. The assignment is essentially an updating activity.
- The operation on the right can involve operands and operators. The operands can be of different types - logical variables, numbers - real or integer and so on.
- The operands on the right side can be of the net or variable type. They can be scalars or vectors.
- It is necessary to maintain consistency of the operands in the operation expression - e.g.,

  N= m | l;

  Here ***m*** and ***l*** have to be same types of quantities - specifically a **reg, integer, time, real, realtime,** or memory type of data - declared in advance.
- The operand to the left of the "=" operator has to be of the variable (e.g., **reg)** type. It has to be specifically declared accordingly. It can be a scalar, a vector, a part vector, or a concatenated vector.
- Procedural assignments are very much like sequential statements in C. Normally they are carried out one at a time sequentially. As soon as a specified operation on the right is carried out, the result is assigned to the quantity on the left - for example

  N = m +l;

  N1=N* N;
- The above form a set of two procedures placed within an **always** block. Generally they are carried out sequentially in the order specified; that is, first m and **l** are added and the result assigned to N. Then the square of N is assigned to **N1**.
- The sequential nature of the assignments requires the operands on the left of the assignment to be of **reg** (variable) type. The basic sequential nature of assignments here is in direct contrast to the concurrent nature of assignments at the data flow level.

3. FUNCTIONAL BIFURCATION

Design description at the behavioral level is done in terms of procedures of two types; one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an "**always**" banner. The second concerns simulation - its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process; all these can be carried out under the "**always**" banner, an "**initial**" banner, or their combinations.

However, each **always** and each **initial** block initiates an activity flow during simulation. In general, the activity with all such blocks starts at the simulation time and flows concurrently during the whole simulation process. The concurrent flow of activity with all processes is characteristic of any behavioral level module. A procedure-block of either type - **initial** or **always** - can have a structure shown in Fig.101**.**

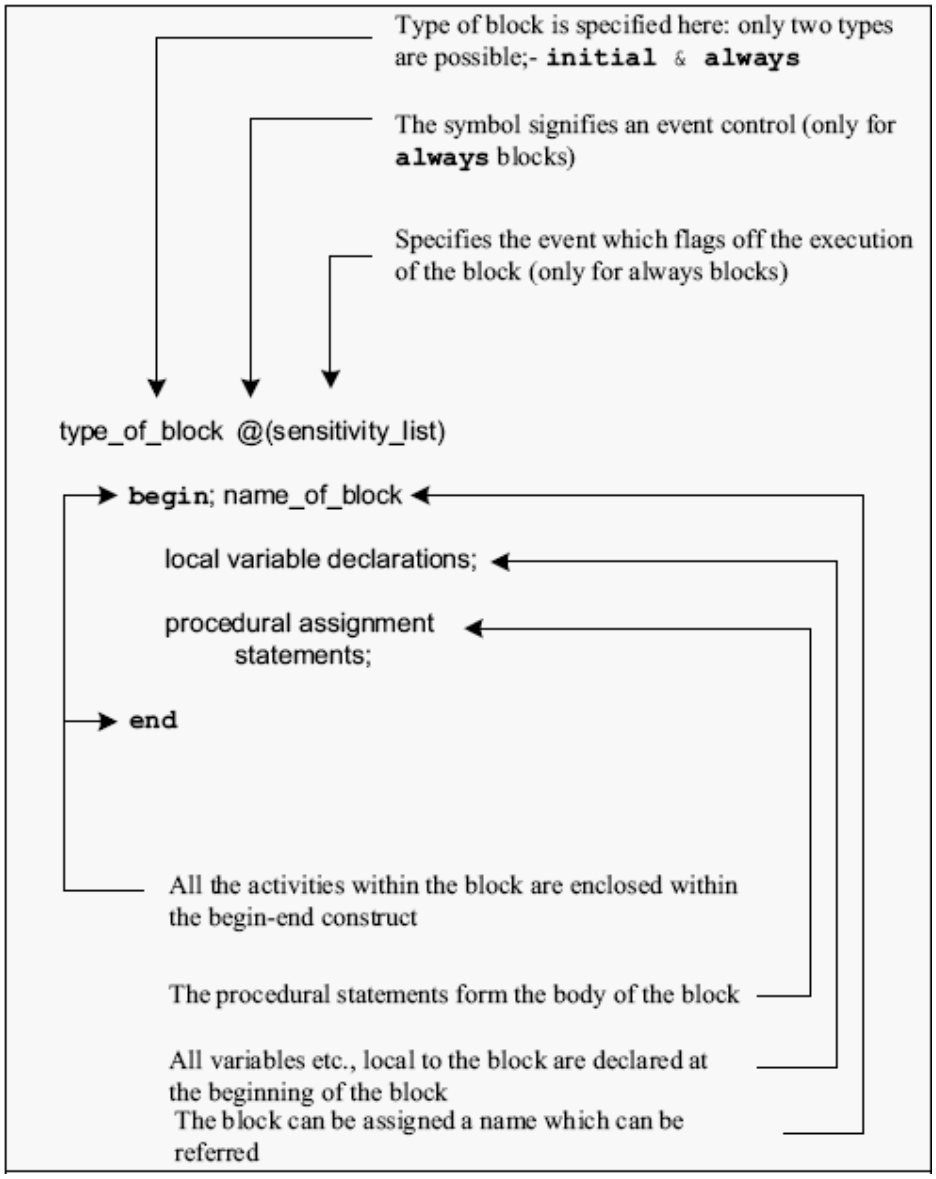


Fig. 101 Structure of a typical procedural block

A block starts with the declaration of the type of block - that is, **initial** or **always.** It may be followed by the definition of a triggering activity and then the body of the block. The body may be a single procedural assignment or a group of procedural assignments. In the latter case the block appears within a **"begin - end"** or similar blocks. The **initial** and **always** blocks have distinct characteristics.

3.1. begin — end Construct

If a procedural block has only one assignment to be carried out, it can be specified as below:

initial ***#2 a=0;***

The above statement assigns the value 0 to variable a at the simulation time of 2 ns. It is possibly the simplest initial block. More often more than one procedural assignment is to be carried out in an **initial** block. All such assignments are grouped together between **"begin"** and **"end"** declarations. Functionally, the construct is similar to the **begin end** construct in Pascal or the { } construct in C language. The following are to be noted here:

- Every **begin** declaration must have its associated **end** declaration.
- **begin - end** constructs can be nested as many times as desired.
- For clarity in description and to avoid mistakes, nested **begin - end** blocks are separated suitably as shown in Fig.102.

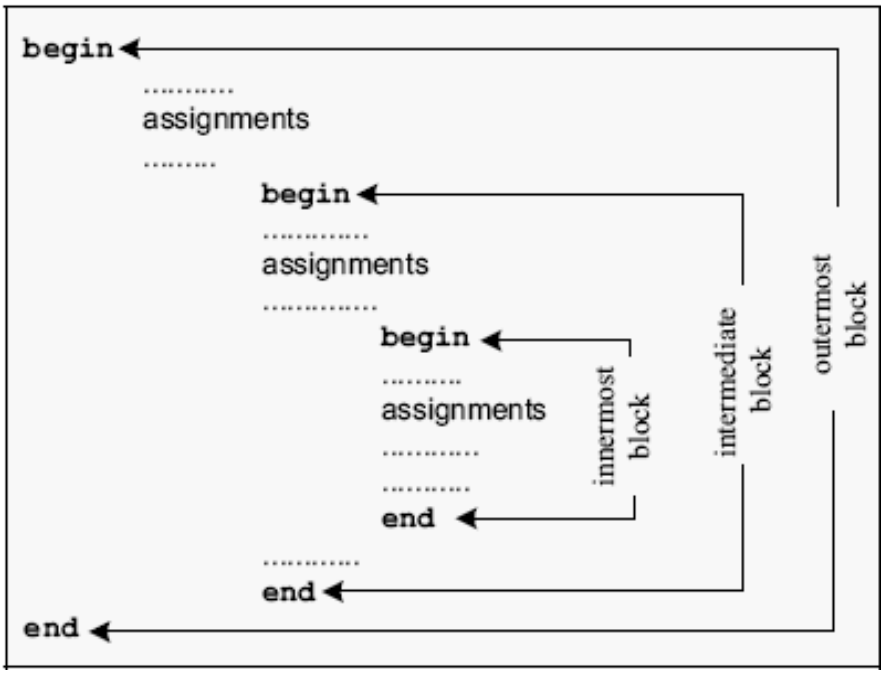


Fig.102 Nesting of begin – end Blocks

3.2. Name of the Block

Any block can be assigned a name, but it is not mandatory. Only the blocks which are to be identified and referred by the simulator need be named. Needless to say the names assigned to different blocks have to be different. Names chosen should conform to the rules for the selection of names to variables. Assigning names to blocks serves different purposes:

- Registers declared within a block are local to it and are not available outside. However, during simulation they can be accessed for simulation, *etc.*, by proper dereferencing.
- Named blocks can be disabled selectively when desired.

3.3. Local Variables

Variables used exclusively within a block can be declared within it. Such a variable need not be declared outside, in the module encompassing the block. Such local declarations conserve memory and offer other benefits too. Regs declared and used within a block are static by nature. They retain their values at the time of leaving the block. The values are modified only at the next entry to the block.

4. INITIAL CONSTRUCT

A set of procedural assignments within an **initial** construct are executed only once - and, that too, at the times specified for the respective assignments.

```
reg a,b;
initial
      begin
            a = 1'b0;
            b = 1'b0;
       #2   a = 1'b1;
       #3   b = 1'b1;
       #1   a = 1'b0;
       #100$stop;
      end
```

Fig.103 Typical initial Block

Consider the **initial** process shown in Fig.103. It is characterized by the following:

- In any assignment statement the left-hand side has to be a storage type of element. It can be a **reg, integer,** or **real** type of variable. The right-hand side can be a storage type of variable or a net.
- The context decides whether the assignment is of a continuous type or procedural type. In the latter case it is present within an **always** or an **initial** construct.

- All the procedural assignments are executed sequentially - in the same order as they appear in the design description. The waveforms of a and b conforming to the assignments in the block are shown in Fig.104.
-

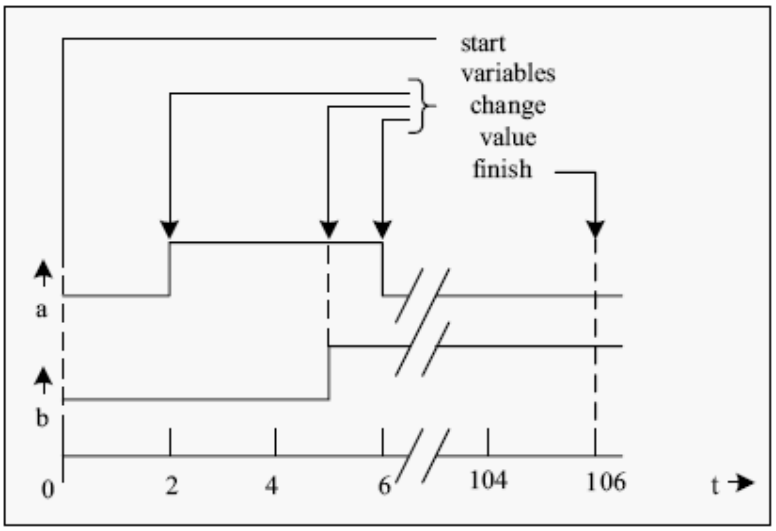


Fig.104 Output waveforms for the given initial block

- Initially (at time t = 0 ns), **a** and b are set equal to zero. At time 2 ns a is made equal to 1. After 3 more nanoseconds - that is, at the 5th ns — b is made equal to 1.
- After one more ns - that is, at the 6th ns - a is made equal to 0.
- **$stop** is a system task. 100 ns later - that is, at the 106th ns - the simulation comes to an end.

Integer values have been used here to decide time delay values. In a more general case the delay value can be a constant expression. It is evaluated and decided dynamically as the simulation proceeds.

The **initial** block above does three controlling activities during the simulation run.

- Initialize the selected set of **reg**'s at the start.
- Change values of **reg**'s at predetermined instances of time. These form the inputs to the module under test and test it for a desired test sequence.
- Stop simulation at the specified time.

Specific system tasks available in Verilog can be used to tabulate the values of selected variables. Providing such output display in a desired or preferred format is the activity of the simulation run. Two system tasks are useful here - **$display & $monitor**.

By way of illustration consider the simulation routine in Fig.105. It incorporates the block Fig.103 and two system tasks. The result of the simulation is shown in Fig.106.

```
module nil;
reg a, b;
initial
begin
        a = 1'b0;
        b = 1'b0;
        $display ("display: a = %b, b = %b", a, b);
  #2    a = 1'b1;
  #3    b = 1'b1;
  #1    a = 1'b0;
  #100  $stop;
end
initial
$monitor("monitor: a = %b, b = %b", a, b);
endmodule
```

Fig. 105 Typical module with an initial block

```
output
        # display : a = 0 ,b = 0
        # monitor : a = 0 ,b = 0
        # monitor : a = 1 ,b = 0
        # monitor : a = 1 ,b = 1
        # monitor : a = 0 ,b = 1
```

Fig. 106 Output for the test bench

The **$display** task is a one-time activity. It is executed when encountered. At that instant in simulation the values of *a* and *b* are zero and the same are displayed. In contrast, **Smonitor** is a repeated activity. It need be present only once in a simulation routine - all the specified variables will be monitored. If multiple **Smonitor** tasks are present in the routine, only the last one will be active. All others will be ignored. In contrast, the **$display** task may appear any number of times in a module. It is executed every time it is encountered.

Simulators have the facility to observe the waveforms and changes in the magnitudes of different variables with simulation time. The necessary facility is provided with the help of user-friendly menus and icons. Waveforms of *a* and *b* obtained with the test bench of Fig.105 are shown in Fig.107; they can be seen to be consistent with their values shown in Fig.106.

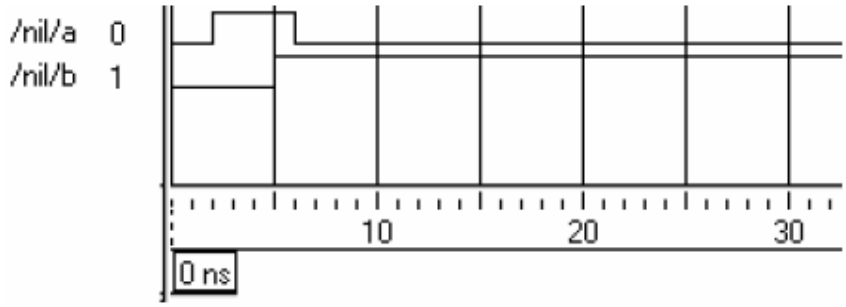


Fig. 107 Output waveforms

4.1. Multiple initial Blocks

A module can have as many **initial** blocks as desired. All of them are activated at the start of simulation. The time delays specified in one **initial** block are exclusive of those in any other block. Consider the module in Fig.108 which is a modified version of that in Fig.105. It has four **initial** blocks. The **Smonitor** task is declared separately. The simulated results are shown in Fig.109.

```
module nil1;
initial
reg a, b;
begin
        a = 1'b0;
        b = 1'b0;
        $display ($time,"display:  a = %b,  b = %b", a, b);
  #2    a = 1'b1;
  #3    b = 1'b1;
  #1    a = 1'b0;
end
initial  #100$stop;
initial $monitor ($time, "monitor:  a = %b,  b = %b", a, b);
initial
        begin
   #2   b = 1'b1;
        end
endmodule
```

Fig. 108 A Typical module with multiple initial blocks

```
output
      # display : a = 0 , b = 0
      # monitor : a = 0 , b = 0
      # monitor : a = 0 , b = 1
      # monitor : a = 1 , b = 1
      # monitor : a = 1 , b = 0
      # monitor : a = 1 , b = 1
      # monitor : a = 0 , b = 1
```

Fig. 109 Output for multiple initial blocks

Observations:

- All changes in ***a*** are brought about in one initial block.
- Changes to ***b*** are specified in two blocks, and both these blocks are executed concurrently.
- The progress of simulation time in different blocks is concurrent. However, those in one block are sequential. Changes in *b* are consistent with this.
- The **$stop** task is in an independent **initial** block. Hence simulation is terminated at 100 ns.
- More than one activity may be scheduled for execution at one time instant. Those in one **initial** block are executed in the same order as they appear - that is, sequentially.

  Thus, the two events **a = 1'b0; b = 1'b0;**

  are executed in the same sequential order - that is, ***b*** is set to 0 after **a** is set to 0, although both the activities are scheduled for execution at the same time.
- At 2 ns ***a*** changes to 1 and ***b*** changes to 0. These two activities are to be done concurrently. They are in different **initial** blocks. The order of their execution depends upon the implementation. This does not cause any anomaly in the present case. But it can be a potential source of problem in more involved designs and their simulation.

7.5 ALWAYS CONSTRUCT

The **always** process signifies activities to be executed on an "always basis." It's essential characteristics are:

- Any behavioral level design description is done using an always block.
- The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.
- The process can have one assignment statement or multiple assignment statements. In the latter case all the assignments are grouped together within a **"begin - end"**

construct.

- Normally the statements are executed sequentially in the order they appear.

5.1 Event Control

The **always** block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol "0". The event can be a change in the variable specified in either direction or a change in a specified direction. For example,

- **@(negedge elk) :**

  executes the following block at the negative edge of the **reg** (variable) elk.

- **@(posedge elk) :**

  executes the following block at the positive edge of the **reg** (variable) elk.

- **@clk:**

  executes the following block at both the edges of elk.

The event can be a combination as well.

- **@(prt or clr) :**

  With the above event the block is executed whenever either of the variables prt or clr undergoes a change.

- **@(posedge clk1 or negedge clk2) :**

  With the above event the block is executed in two cases - whenever the clock clkl changes from 0 to 1 state or the clock clk2 changes from 1 to 0. One can specify more elaborate events by OR'ing individual ones. The following are to be noted:

- The events can be changes in **reg, integer, real** or a signal on a net. These should be declared beforehand.

- No algebra or logic operation is permitted as an event. The OR'ing signifies "execute the block if any one of the events takes place."

- The edge transition on each event is to be specified separately

- Note the difference between the following:

  - **(posedge** clk1 or clk2): means "execute the block following if clk1 goes to 1 state or clk2 changes state (whether 0 to 1 or 1 to 0)."

  - **(posedge** clk1 or **posedge** clk2): means "execute the block following if clk1 goes to 1 state or clk2 goes to 1 state."

- The positive transition for a reg type single bit variable is a change from 0 tol. For a logic

variable it is a transition from false to true.

- The **"posedge"** transition for a signal on a net can be of three different types:
  - 0 to 1
  - 0 to **x** or **z**
  - **x** or **z** to 1
- The **"negedge"** transition for a signal on a net can be of three different types :-
  - 1 to 0
  - 1 to **x** or **z**
  - **x** or **z** to 0
- If the event specified is in terms of a multibit **reg,** only its least significant bit is considered for the transition. Changes in the other bits are ignored.
- The event-based flagging-off of a block is applicable only to the **always** block.
- According to the recent version of the LRM, the comma operator (,) plays the same role as the keyword **or.** The two can be used interchangeably or in a mixed form. Thus the following are identical:

  @ (a **or** b **or** C)

  @ (a **or** b, C)

  @ (a, b, c)

  @ (a, b **or** c)

6. EXAMPLES

Example 17 A Versatile Counter

We consider a versatile up-down counter module with the following facilities:

- ***Clear input*** If it goes high, the counter is cleared and reset to zero.
- ***U/D input*** If it goes high, the counter counts up; if it goes down, the counter counts down.
- The counter counts at the **negative edge** of the clock.

The counter counts up or down between 0 and *N* where *N* is any 4-bit hex number. The above counter design specifications are implemented in stages. The module in Fig.110 is an up counter which counts up repeatedly from 0 to a preset number *N*. A test-bench for the counter is also shown in the figure. *N* is an input to the module. The count advances at every negative edge of the clock. When the count reaches the value *N,* the count value *a* is reset to

0. The simulation results are shown as waveforms in Fig.111.

The periodic clock waveform, the incrementing of *a* at every negative edge of the clock and counting of *a* from 0 to the set value of *N* can be seen from the figure. The synthesized circuit of the counter is shown in Fig.112. It has a versatile counter block and a comparator. The comparator compares the value of *a* with the set value of *N* and resets the counter when the two are equal.

```
module counterup(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b0000;
always@(negedge clk) a=(a==N)?4'b0000:a+1'b1;
endmodule

module tst_counterup;//TEST_BENCH
reg clk;
reg[3:0]N;
wire[3:0]a;
counterup c1(a,clk,N);
initial
begin
     clk = 0;
     N   = 4'b1011;
end
always #2 clk=~clk;
initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N);
endmodule
```

Fig.110 An UP Counter Module

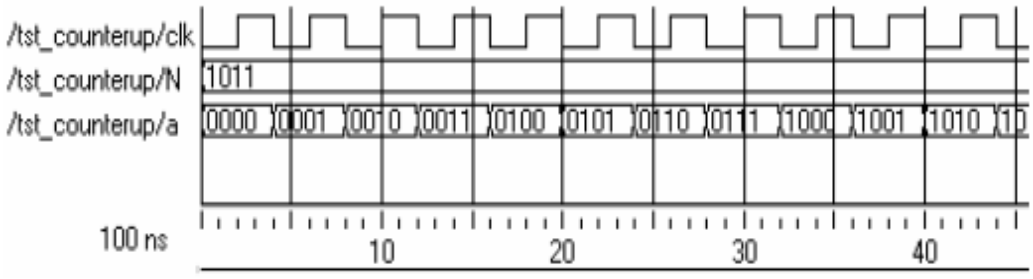


Fig.111 Simulation waveforms

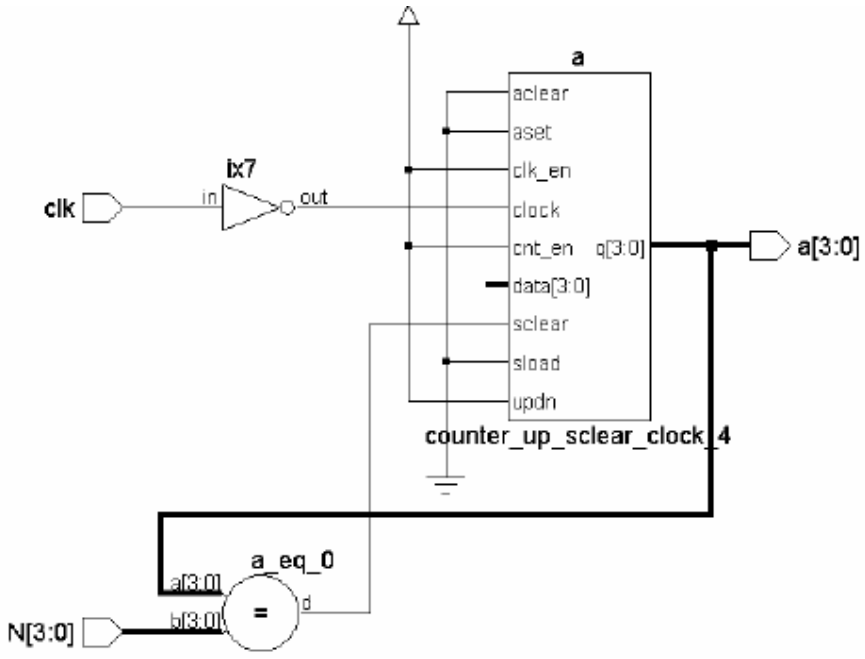


Fig.112 Synthesized Circuit for UP Counter

```
module counterdn(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk) a=(a==4'b0000)?N:a-1'b1;
endmodule

module tst_counterdn();//TEST_BENCH
reg clk;
reg[3:0]N;
wire[3:0]a;
counterdn cc(a,clk,N);
initial
begin
    N   = 4'b1010;
    Clk = 0;
end
always #2 clk=~clk;
initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N);
initial #55 $stop;
endmodule
```

Fig.113 DOWN Counter Module

The module of Fig.113 is a down counter. The count **a** decrements at the negative edge of the clock - clk. The counter counts down from *N* to zero. As soon as the count reaches the value 0, it is set back to *N*. The simulation results are shown tabulated in Fig.114 and as waveforms in Fig.115; these can be seen to be consistent with the design module. The synthesized circuit is shown in Fig.116. The basic blocks - namely versatile counter, comparator and buffer for the clock - are the same as those for the up counter of Fig.112. The comparator output loads the value of *N* back into the counter every time a reaches the set value of *N*.

```
Output
     #  0a=1010,clk=0,N=1010
     #  2a=1010,clk=1,N=1010
     #  4a=1001,clk=0,N=1010
     #  6a=1001,clk=1,N=1010
     #  8a=1000,clk=0,N=1010
     # 10a=1000,clk=1,N=1010
     # 12a=0111,clk=0,N=1010
     # 14a=0111,clk=1,N=1010
     # 16a=0110,clk=0,N=1010
     # 18a=0110,clk=1,N=1010
     # 20a=0101,clk=0,N=1010
     # 22a=0101,clk=1,N=1010
     # 24a=0100,clk=0,N=1010
     # 26a=0100,clk=1,N=1010
     # 28a=0011,clk=0,N=1010
     # 30a=0011,clk=1,N=1010
     # 32a=0010,clk=0,N=1010
     # 34a=0010,clk=1,N=1010
     # 36a=0001,clk=0,N=1010
     # 38a=0001,clk=1,N=1010
     # 40a=0000,clk=0,N=1010
     # 42a=0000,clk=1,N=1010
     # 44a=1010,clk=0,N=1010
     # 46a=1010,clk=1,N=1010
     # 48a=1001,clk=0,N=1010
     # 50a=1001,clk=1,N=1010
     # 52a=1000,clk=0,N=1010
     # 54a=1000,clk=1,N=1010
```

Fig. 114 Results for DOWN Counter

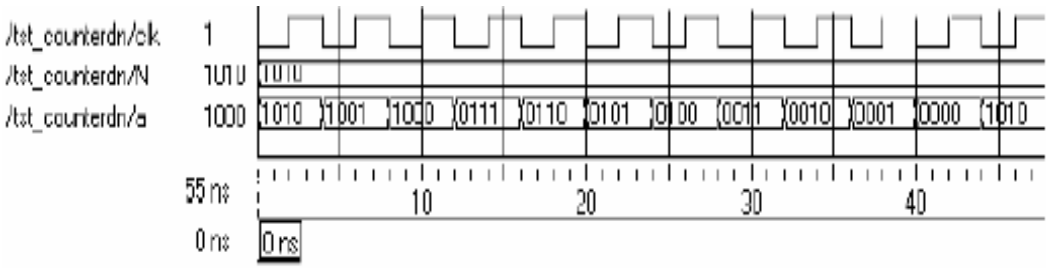


Fig.115 Simulation Results

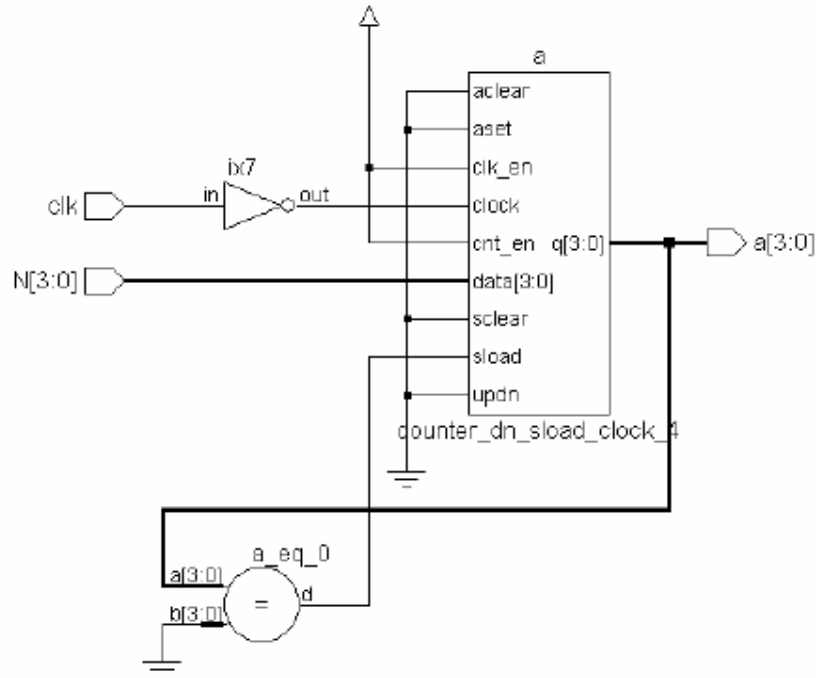


Fig.116 Synthesized Circuit for DOWN Counter

The up and down modes of counting have been combined in the up down counter of Fig.117. A test bench is also shown in the figure. The test results are tabulated in Fig.119 and also shown as waveforms in Fig.118.

Fig.120 shows the synthesized circuit; the counter block remains the same as in the last two cases; the mode control part of the circuit has been changed to meet the enhanced needs. The counting can be seen to be changing from "up" to the "down" type, when the mode control input u_d changes.

```
module updcounter(a,clk,N,u_d);
input clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk)
a=(u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-
1'b1);
endmodule

module tst_updcounter();//TEST_BENCH
reg clk,u_d;
reg[3:0]N;
wire[3:0]a;
updcounter c2(a,clk,N,u_d);
initial
begin
    N   = 4'b0111;
    u_d = 1'b0;
    clk = 0;
end
always #2 clk=~clk;
always #34u_d=~u_d;
initial $monitor
($time,"clk=%b,N=%b,u_d=%b,a=%b",clk,N,u_d,a);
initial #64 $stop;
endmodule
```

Fig. 117 An UP / DOWN Counter

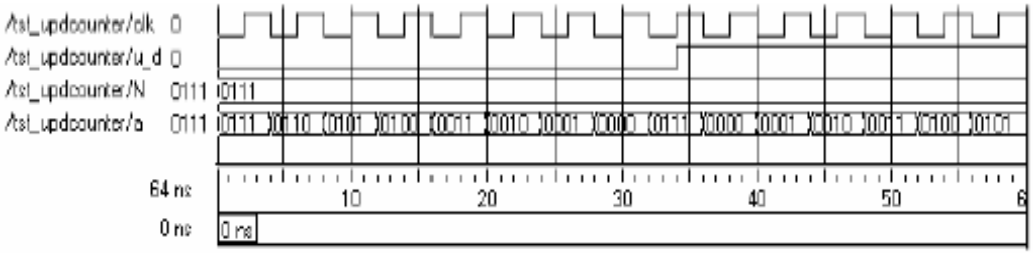


Fig. 118 Simulation Results

```
#                  0clk=0,N=0111,u_d=0,a=0111
#                  2clk=1,N=0111,u_d=0,a=0111
#                  4clk=0,N=0111,u_d=0,a=0110
#                  6clk=1,N=0111,u_d=0,a=0110
#                  8clk=0,N=0111,u_d=0,a=0101
#                 10clk=1,N=0111,u_d=0,a=0101
#                 12clk=0,N=0111,u_d=0,a=0100
#                 14clk=1,N=0111,u_d=0,a=0100
#                 16clk=0,N=0111,u_d=0,a=0011
#                 18clk=1,N=0111,u_d=0,a=0011
#                 20clk=0,N=0111,u_d=0,a=0010
#                 22clk=1,N=0111,u_d=0,a=0010
#                 24clk=0,N=0111,u_d=0,a=0001
#                 26clk=1,N=0111,u_d=0,a=0001
#                 28clk=0,N=0111,u_d=0,a=0000
#                 30clk=1,N=0111,u_d=0,a=0000
#                 32clk=0,N=0111,u_d=0,a=0111
#                 34clk=1,N=0111,u_d=1,a=0111
#                 36clk=0,N=0111,u_d=1,a=0000
#                 38clk=1,N=0111,u_d=1,a=0000
#                 40clk=0,N=0111,u_d=1,a=0001
#                 42clk=1,N=0111,u_d=1,a=0001
#                 44clk=0,N=0111,u_d=1,a=0010
#                 46clk=1,N=0111,u_d=1,a=0010
#                 48clk=0,N=0111,u_d=1,a=0011
#                 50clk=1,N=0111,u_d=1,a=0011
#                 52clk=0,N=0111,u_d=1,a=0100
#                 54clk=1,N=0111,u_d=1,a=0100
#                 56clk=0,N=0111,u_d=1,a=0101
#                 58clk=1,N=0111,u_d=1,a=0101
#                 60clk=0,N=0111,u_d=1,a=0110
#                 62clk=1,N=0111,u_d=1,a=0110
```

Fig. 119 Test Bench Results

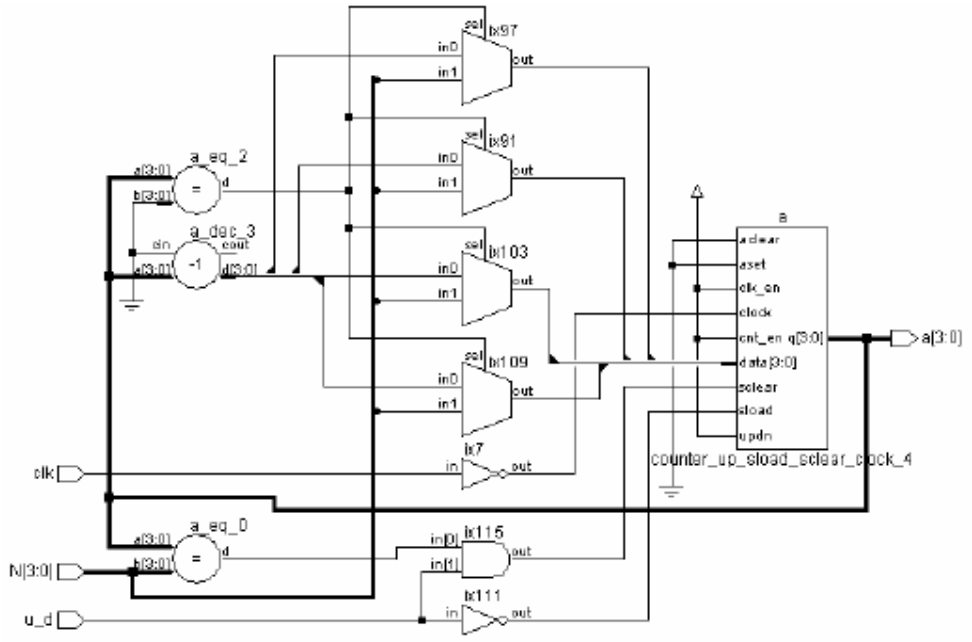


Fig. 120 Synthesized Circuit for UP / DOWN Counter

Example 18 Shift Register

Figure 121 shows an 8-bit shift register module along with a test bench for the same. The register shifts by one bit to the right if r_l = 1 and to the left by one bit otherwise. The whole shift register is described in a single line of procedural assignment, namely

always@(negedge clk) a=(r_l)?(a>>1'b1):(a<< 1'b1);

The simulation results are given in tabular form in Fig.122 and as waveforms in Fig.123.

```
module shifrlter(a,clk,r_l);
input clk,r_l;
output [7:0]a;
reg[7:0]a;
initial a= 8'h01;
always@(negedge clk)
```

```
begin
     a=(r_l)?(a>>1'b1):(a<<1'b1);
end
endmodule

module tst_shifrlter;//test-bench
reg clk,r_l;
wire [7:0]a;
shifrlter shrr(a,clk,r_l);
initial
begin
     clk =1'b1;
     r_l = 0;
end
always #2 clk =~clk;
initial #16 r_l =~r_l;
initial
$monitor($time,"clk=%b,r l = %b,a =%b ",clk,r l,a);
initial #30 $stop;
endmodule
```

Fig.121 8-bit Shift Register

```
Output
  #     0 clk=1, r_l = 0 , a = 00000001
  #     2 clk=0, r_l = 0 , a = 00000010
  #     4 clk=1, r_l = 0 , a = 00000010
  #     6 clk=0, r_l = 0 , a = 00000100
  #     8 clk=1, r_l = 0 , a = 00000100
  #    10 clk=0, r_l = 0 , a = 00001000
  #    12 clk=1, r_l = 0 , a = 00001000
  #    14 clk=0, r_l = 0 , a = 00010000
  #    16 clk=1, r_l = 1 , a = 00010000
  #    18 clk=0, r_l = 1 , a = 00001000
  #    20 clk=1, r_l = 1 , a = 00001000
  #    22 clk=0, r_l = 1 , a = 00000100
  #    24 clk=1, r_l = 1 , a = 00000100
  #    26 clk=0, r_l = 1 , a = 00000010
  #    28 clk=1, r_l = 1 , a = 00000010
```

Fig.122 Test Bench Results

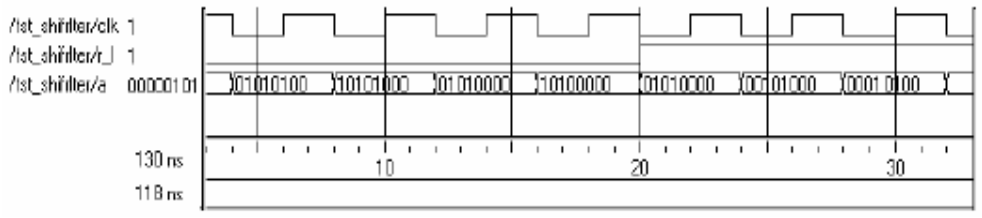


Fig.123 Simulation Results

Example 19 Clocked Flip-Flop

The module for a clocked flip-flop is shown in Fig.124. A test bench for the flip-flop is also included in the figure. The test results are shown in Fig.125 and Fig.126 as waveforms and in tabular form, respectively. The input can be seen to be sensed, latched, and presented as output at every negative edge of the clock. Otherwise the output remains frozen at the last latched value. The synthesized circuit of the flip-flop is shown in Fig.127.

```
module dff(do,di,clk);
output do;
input di,clk;
reg do;
initial
do=1'b0;
always@(negedge clk) do=di;
endmodule

module tst_dffbeh();//test-bench
reg di,clk;
wire do;
dff d1(do,di,clk);
initial
begin
     clk=0;
     di=1'b0;
end
always #3clk=~clk;
always #5 di=~di;
initial
$monitor($time,"clk=%b,di=%b,do=%b",clk,di,do);
initial #35 $stop;
endmodule
```

Fig.124 A D Flip-Flop Module

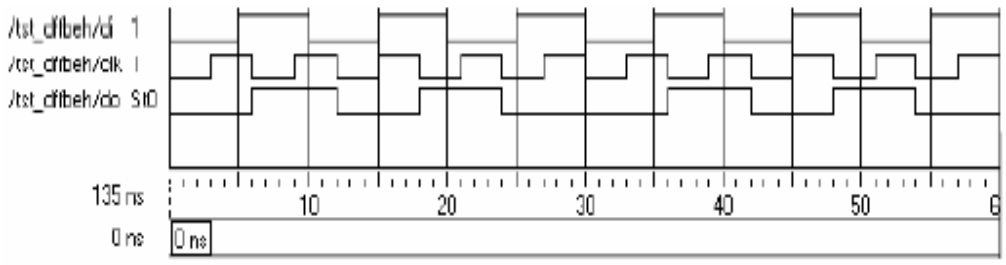


Fig.125 Simulation Results

```
Output
  #                  0clk=0,di=0,do=0
  #                  3clk=1,di=0,do=0
  #                  5clk=1,di=1,do=0
  #                  6clk=0,di=1,do=1
  #                  9clk=1,di=1,do=1
  #                 10clk=1,di=0,do=1
  #                 12clk=0,di=0,do=0
  #                 15clk=1,di=1,do=0
  #                 18clk=0,di=1,do=1
  #                 20clk=0,di=0,do=1
  #                 21clk=1,di=0,do=1
  #                 24clk=0,di=0,do=0
  #                 25clk=0,di=1,do=0
  #                 27clk=1,di=1,do=0
  #                 30clk=0,di=0,do=0
  #                 33clk=1,di=0,do=0
```

Fig.126 Test Bench Results

7. ASSIGNMENTS WITH DELAYS

Specific delays can be associated with procedural assignments. The delay refers to the specific activity it qualifies. A variety of possibilities of specifying delays to assignments exist.

Consider the assignment **always** #3 b = a;

simulator encounters this at zero time and posts the entire activity to be done 3 ns later. Further, by virtue of the always nature of the activity, the assignment is scheduled to be repeated every 3 ns, irrespective of whether a changes in the meantime. Values of a at the 3rd, 6th, 9th, *etc.*, ns are sampled and assigned to b. Fig.127 shows the waveforms of a and b with the above assignment and execution of the module in Fig.128. Changes in the values of a lasting less than 3 ns may be ignored. Specifically, in this case, a took the value of 1 during the interval 4th ns to the 5th ns which is not passed on to **b.**

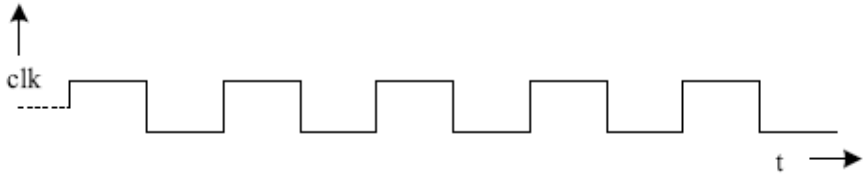


Fig.127 Clock Waveform using always block

```
module del1;
reg a,b;
always #3 b=a;
Initial

begin
                a = 1'b1;
                b = 1'b0;
          #1    a = 1'b0;
          #3    a = 1'b1;
          #1    a = 1'b0;
          #2    a = 1'b1;
          #3    a = 1'b0;
end
initial $monitor($time," a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Fig.128 A module to illustrate delayed assignment

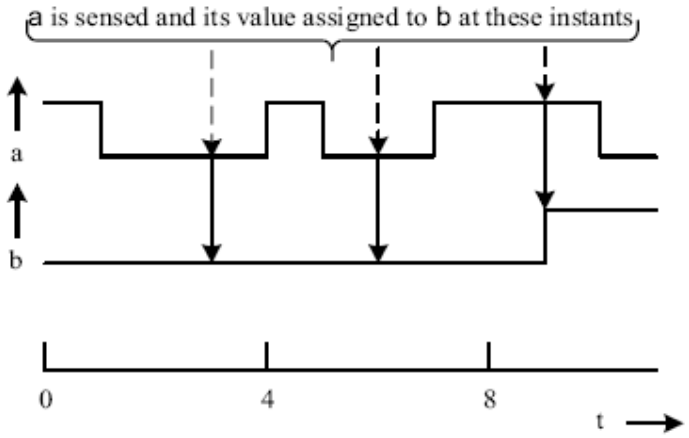


Fig.129 Waveforms of a & b

The module of fig.130 is a modified version of that in Fig.128. The activities within the always block (of a single statement) are carried out whenever the value of a changes. The sole activity is that of assigning the value of a to b with a delay of 2 ns - that is, 2 ns after a changes sign. The waveform assigned to a as well as the resulting waveform of b is shown in Fig.131. If a were to remain invariant, b will have no assignment here.

```
module del2;
reg a,b;
always @(a) #2 b=a;

Initial
begin
                a = 1'b1;
                b = 1'b0;
        #1      a = 1'b0;
        #3      a = 1'b1;
        #1      a = 1'b0;
        #2      a = 1'b1;
        #3      a = 1'b0;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Fig.130 A module to illustrate delayed assignments

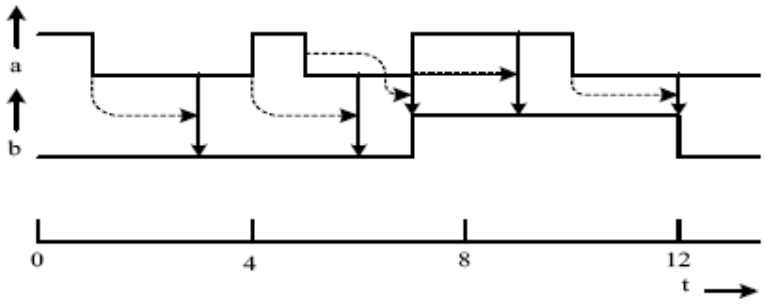


Fig.131 Waveforms of a & b

7.1 Intra-assignment Delays

An assignment delay of the type discussed above, delays execution of the whole assignment by the specified time duration. In contrast, the "intra-assignment" delay carries out the assignment in two parts. An assignment with an intra- assignment has the form

A = # dl expression;

Here the expression is scheduled to be evaluated as soon as it is encountered. However, the result of the evaluation is assigned to the right-hand side quantity a after a delay specified by **dl. dl** can be an integer or a constant expression. Consider the example in Fig.132 **b** is assigned the value of **a** with an intra-assignment delay of 2 ns. The value of a is sensed at zero ns and assigned to b after 2 ns. Until that time, b retains its old value. Again at the 2nd ns, a is sensed and b is assigned the new value of a at the 4th ns, and so on. Partial results of simulation are shown in Table 24. The following points are to be noted here:

- The value of **a** is sensed at time instants 2, 4, 6, *etc.*

- Values at other instants of time are not sensed.
- All assignments are carried out with a delay of 2 ns.
- Changes in **a** which do not last for **2** ns may be ignored.

```
Module del4;
Integer a, b;
Always b = #2 a;
Initial
begin
        a = 0;     b = 0;  #2   a =1;  #2   a =2;  #2 a =3;
  #2    a =4;  #2 a = 5;  #2   a =6;  #2   a =7;  #2 a =8;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Fig.132 A module to illustrate delayed assignment

t	a	b	Remarks
0	0	0	There are two assignment statements to a at 2 ns intervals – namely the one in the always block and the other one in the initial block; both are concurrent. The simulator decides the precedence. The output here shows that the assignment in the always block has the precedence.
2	1	x	
4	2	1	
6	3	2	
8	4	2	

Table 24 Simulation Results

7.2 Delay Assignments

In all the illustrations above, delay was specified as a number. It may be a variable or a constant expression. In case it is an expression, it is evaluated and execution delayed by the number of time steps. If the number evaluates to a negative quantity, the same is interpreted as a 2's complement value. In the statement

always* #b *a = a + 1 ;

a and b are variables. The execution incrementing a is scheduled at b ns. If b changes, the execution time also changes accordingly. As another example consider the procedural assignment

always #(b + c) *a = a + 1 ;*

Here a, b, and c are variables. The algebraic addition of variables b and c is to be done. The scheduler schedules the incrementing of 3 and reassigning the incremented values back to 3 with a time delay of (b + c) ns. As an additional example consider the assignment below with an intra-assignment delay.

always # (a + b) α **= # (b + c) a +1;**

Here the simulator evaluates **(a** + **b)** during simulation. After a lapse of **(a** + **b)** ns, execution of the statement is taken up; (a + 1**)** is evaluated and assigned as the new value of 3 - but the assignment is delayed by **(b + c)** ns.

7.3 Zero Delay

A delay of 0 ns does not really cause any delay. However, it ensures that the assignment following is executed last in the concerned time slot. Often it is used to avoid indecision in the precedence of execution of assignments.

8. wait CONSTRUCT

The **wait** construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments. Its syntax has the form

wait (alpha) assignmentl;

alpha can be a variable, the value on a net, or an expression involving them. If alpha is an expression, it is evaluated; if true, assignmentl is carried out. One can also have a group of assignments within a block in place of assignmentl. The activity is level-sensitive in nature, in contrast to the edge-sensitive nature of event specified through @. Specifically the procedural assignment

@ clk a = b;

assigns the value of b to a when elk changes; if the value of b changes when elk is steady, the value of a remains unaltered. In contrast, with

wait (clk) #2 a = b;

the simulator waits for the clock to be high and then assigns b **to** a with a delay of 2 ns. The assignment will be refreshed as long as the elk remains high.

Example 20: **A rudimentary serial transmitter module**

Fig.133 shows a rudimentary and crude version of a serial receiver module and its test bench. Simulation results are shown in Fig.134. The module receives serial data on the **di** line. The data are synchronized to the clock **clk.** The sequence of operations carried out by the module is as follows:

- Wait for recv input to go high.
- Once recv=l, latch the next 4 successive bits of incoming data into respective bit positions of the do register.

- Once the above nibble receipt is accomplished, set acknowledgment flag high.
- If recv continues to remain high, the subsequent serial bits will be loaded into the do nibble, again and again in groups of 4 bits.
- If at any time recv goes low, the receipt and the serial to parallel conversion will come to a stop.

```
module sr_rec(do, ack, clk, di, recv);
output [3:0] do; output ack;
input clk, recv, di;
reg [3:0] do; reg ack;
initial ack = 1'b0;
always   begin
                    wait(recv)
                    @(negedge clk) do[0]=di;
                    @(negedge clk) do[1]=di;
                    @(negedge clk) do[2]=di;
                    @(negedge clk) do[3]=di;
                    @(negedge clk) ack = 1'b1;
         end
endmodule

module tst_sr_rec;
reg clk, di, recv;
wire [3:0]do; wire ack;
initial    begin
         clk=1'b0; recv=1'b0; di=1'b0;  #5 recv=1'b1;
         end
always #2clk = ~clk;
initial    begin
         #7di=1'b1; #4di=1'b0; #8di=1'b1; #8di=1'b0;
         end
initial $monitor($time, "clk=%d, recv=%b, di=%b, do=%b, ack=%b",
         clk, recv, di, do, ack);
sr_rec rrcc(do, ack, clk, di, recv);

initial #25 $stop;
endmodule
```

Fig.133 A rudimentary serial transmitter module

```
//output
#                0clk=0, recv=0, di=0, do=xxxx, ack=0
#                2clk=1, recv=0, di=0, do=xxxx, ack=0
#                4clk=0, recv=0, di=0, do=xxxx, ack=0
#                5clk=0, recv=1, di=0, do=xxxx, ack=0
#                6clk=1, recv=1, di=0, do=xxxx, ack=0
#                7clk=1, recv=1, di=1, do=xxxx, ack=0
#                8clk=0, recv=1, di=1, do=xxx1, ack=0
#               10clk=1, recv=1, di=1, do=xxx1, ack=0
#               11clk=1, recv=1, di=0, do=xxx1, ack=0
#               12clk=0, recv=1, di=0, do=xx01, ack=0
#               14clk=1, recv=1, di=0, do=xx01, ack=0
#               16clk=0, recv=1, di=0, do=x001, ack=0
#               18clk=1, recv=1, di=0, do=x001, ack=0
#               19clk=1, recv=1, di=1, do=x001, ack=0
#               20clk=0, recv=1, di=1, do=1001, ack=0
#               22clk=1, recv=1, di=1, do=1001, ack=0
#               24clk=0, recv=1, di=1, do=1001, ack=1
```

Fig.134 Simulation Results

9. MULTIPLE ALWAYS BLOCKS

All the activities within an always block are scheduled for sequential execution. The activities can be of a combinational nature, a clocked sequential nature, or a combination of these two. A design description involving such combinations is conventionally called the 'Register Transfer Level' description. Basically, any circuit block whose end-to-end operation can be described as a continuous sequence can be described within an **always** block. A typical circuit block conforming to the above description is shown in Fig.135. It has three activities termed Al, A2, and A3. These three are to be done in that order. Activity A1 accepts x as input and it generates output B and p. p and y form inputs to activity A2. Similarly activity A2 generates outputs c and q after activity Al is completed. *q* and z form outputs of A2. After activity A2 is completed, activity A3 is scheduled. It accepts z and *q* as inputs and generates *D* as output. Here if Al, A2, and A3 are logical activities, the whole block can be synthesized as a combinational logic unit. If one or more of these are clocked events, execution may be sequential. The design examples considered so far are broadly of this category.

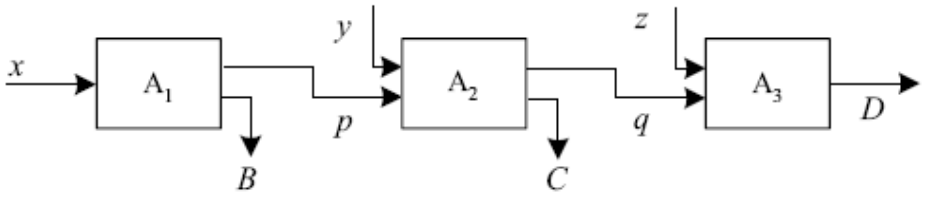


Fig.135 Sequentially connected always blocks

In a comparatively bigger IC, the activity flow can be more complex. One with an additional level of complexity is shown in Fig.136. The activities are marked A1-A2-A3 and B1-B2-B3, These are the two streams in the circuit. It is possible that the intermediate results of one may affect the flow of the other. Functioning of two timers - dependent on each other -is a typical example. A processor servicing serial reception and serial transmission simultaneously is another example. In all these cases, each sequential activity is described in a separate always block.

A design of the type in Fig.136 can be described with two always blocks. In some others, three or more always blocks may be called for.

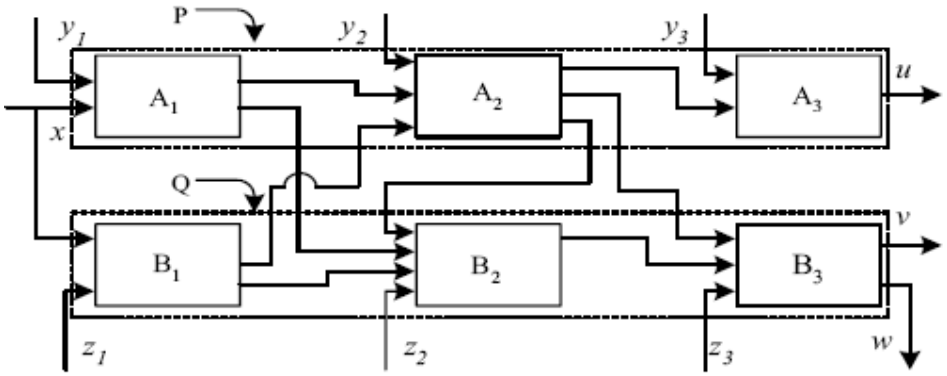


Fig.136 Sequentially connected always blocks for concurrent execution

Activities within one always block are normally sequential. If necessary, they can be made selectively concurrent. But when designs are spread out in two or more always blocks, they are necessarily concurrent. Thus the blocks P and Q in Fig.136 are concurrent while the "sub-blocks" within each (namely Al, A2, and A3 within block P and Bl, B2, and B3 within block Q) are sequential. In short, with behavioral level descriptions, one can organize the activities to be in concurrent form, in sequential form, or in combinations. In contrast, all design descriptions involving constructs at gate and data flow levels are necessarily concurrent.

10. DESIGNS AT BEHAVIORAL LEVEL

All simple algebraic as well as logical expressions can be described at the behavioral level. One can also mix them with blocks at the gate level as well as the data flow level to form composite as well as more involved modules.

```
module aoibeh(o,a,b);
output o;
input[1:0]a,b;
reg o,a1,b1,o1;
always@(a[1] or a[0]or b[1]or b[0])
begin
      a1=&a;
      b1=&b;
      o1=a1||b1;
      o=~o1;
end
endmodule

module tst_aoibeh;
reg [1:0]a,b; /*  specicific 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
     a[0]=1'b0;a[1] =1'b0;b[0]=1'b0;b[1] =1'b0;
     #3 a[0] =1'b1;
     #3 a[1] =1'b1;
     #3 b[0] =1'b1;
     #3 b[1] =1'b0;
     #3 a[0] =1'b1;
     #3 a[1] =1'b0;
     #3 b[0] =1'b0;
end
initial #100 $stop;//the simulation ends after running
for 100 tu's.
initial $monitor($time, "o =%b,a[0]=%b,a[1]=%b, b[0] =
%b ,b[1] = %b ",o,a[0],a[1],b[0],b[1]);
aoibeh gg(o,a,b);
endmodule
```

Fig.137 A-O-I gate at behavioral level modeling and its test bench

```
 # 0  o = 1,a[0]=0,a[1]=0,b[0]=0,b[1]=0
 # 3  o = 1,a[0]=1,a[1]=0,b[0]=0,b[1]=0
 # 6  o = 0,a[0]=1,a[1]=1,b[0]=0,b[1]=0
 # 9  o = 0,a[0]=1,a[1]=1,b[0]=1,b[1]=0
 #18  o = 1,a[0]=1,a[1]=0,b[0]=1,b[1]=0
 #21  o = 1,a[0]=1,a[1]=0,b[0]=0,b[1]=0
```

Fig.138 Simulation Results

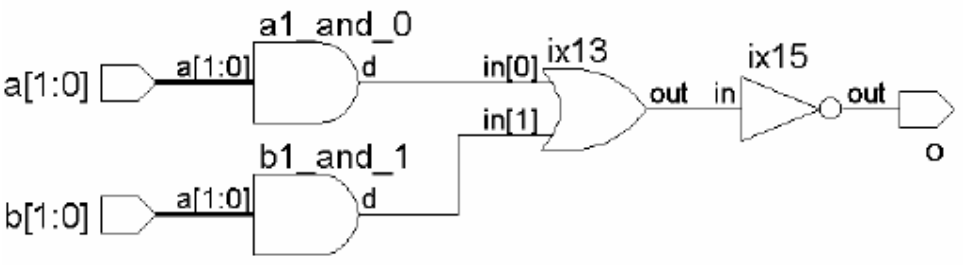


Fig.139 Synthesized circuit

Alternative methods for A-O-I gate design:

```
module aoibeh1(o,a,b);
output o;
input[1:0]a,b;
reg o;
always@(a[1]ora[0]or b[1]orb[0]) o=~((&a)||(&b));
endmodule
```

Fig.140 Alternative 1

```
module aoibeh2(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
and g1(a1,a[1],a[0]),g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1||b1);
endmodule
```

Fig.141 Alternative 2

```
module aoibeh3(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a,b1=&b;
always@(a1 or b1)o=~(a1||b1);
endmodule
```

Fig.142 Alternative 3

```
module aoibeh4(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a;
and g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1||b1);
endmodule
```

Fig.143 Alternative 4

11. BLOCKING AND NONBLOCKING ASSIGNMENTS

All assignment within an initial or an always block considered so far are done through an equality ("=") operator. These are executed sequentially - that is, one statement is executed, and only then the following one is executed. Such assignments block the execution of the following lot of assignments at any time step. Hence they are called "blocking assignments". Further, when such a blocking assignment has time delays associated with it, the delay is applicable to the following assignment or activity also.

One comes across situations where assignments are to be effected concurrently. A facility called the "nonblocking assignment" is available for such situations. The symbol "<=" signifies a non-blocking assignment. The same symbol signifies the "less than or equal to" operator in the context of an operation. The context decides the role of the symbol. The main characteristic of a non- blocking assignment is that its execution is concurrent with that of the following assignment or activity.

Consider the set of nonblocking assignments in Fig.144. All three assignments are executed concurrently - that is, A, B, and C are assigned the values 00 01 and 11 concurrently and not sequentially. Fig.145 shows the same non-blocking assignments with time delays. All three assignments are taken up for execution concurrently.

```
A <= 2'b00;
B <= 2'b01;
C <= 2'b11;
```

Fig. 144 Non Blocking Assignments

```
      A <= 2'b00;
#2    B <= 2'b01;
#1    C <= 2'b11;
```

Fig. 145 Non Blocking Assignments with delays

Nonblocking assignments are essentially two-step affairs. For all the non-blocking assignments in a block, the right-hand sides are evaluated first. Subsequently the specified assignments are scheduled. Consider the block of assignments in Fig.146. First A is assigned the binary value 00, and then B is assigned the value 01. These two assignments are sequential. The subsequent two assignments are concurrent.

The assignment **A <= b** "reads" the value of B, stores it separately, and then assigns it to A. The new value of a is 01. The assignment takes the value of A- i.e., 00 - stores it separately and assigns it to B. Thus the new value of B is 00. After the block is executed, A has the value 01 while B has the value 00. Contrast this with the set of blocking assignments in Fig.147. All four assignments here are sequential in nature. The third one, namely

A= B;

assigns the value 01 to a; subsequently the fourth and following assignment

B = A;

assigns the present value of A (i.e., 01) to b; the value of b remains at 01 itself.

```
A = 2'b00;
B = 2'b01;
A <= B;
B <= A;
```

Fig.146 Non Blocking Assignments

```
A = 2'b00;
B = 2'b01;
A = B;
B = A;
```

Fig. 145 Blocking Assignments

```
initial
begin
    A= 1'b0;
    B= 1'b1;
    C = 1'b0;
end
always @(posedge clk)
        begin
        A <= B;
        @(negedge clk) C <= B &(~c);
        #2 B< = C;
        end
```

Fig.148 Module with blocking and non blocking assignments

Consider the block of Fig.148. It has three nonblocking assignments. The sequence of execution of the three assignments is as follows:

- At the positive edge of the clock, values of A, B, and C are read and stored and B &(~C) are computed.
- A is assigned the stored value of B (=1); this and the activity in (1) above are carried out concurrently in the same time step.
- At the next negative elk edge, C is assigned the value of B & (~C) evaluated and stored earlier (=1) — mentioned in (1) above.

- Two nanoseconds after the positive edge of elk (i.e., after the entiy to the block), B is assigned the value of C stored earlier (=0).

In the segment in Fig.149, two always blocks do assignments concurrently; both of these are of the blocking variety. The values assigned to A and B are decided by the structure of the simulator. The block has the potential to create a race condition. In contrast, in the segment of Fig.150, the two assignments are of the nonblocking type; A is assigned the previous value of B, while B is assigned the previous value of A. The race condition is avoided here.

```
always @(posedge clk)
A = B;
always @(posedge clk)
B = A;
```

Fig.149 A set of assignments with potential race condition

```
always @(posedge clk)
A <= B;
always @(posedge clk)
B <= A;
```

Fig.150 A set of assignments to avoid race condition

Although blocking and nonblocking assignment can be mixed in a block, many synthesis tools may not support such combinations.

11.1. Nonblocking Assignments and Delays

Delays - of the assignment type and the intra-assignment type - can be associated with nonblocking assignments also. The principle of their operation is similar to that with blocking assignments. The delay values can be constant expressions. Blocking and nonblocking assignments, together with assignment and intra-assignment delays, open up a variety of possibilities. They can be used individually and in combinations to suit different situations.

12. THE case STATEMENT

The **case** statement is an elegant and simple construct for multiple branching in a module. The keywords **case, endcase,** and **default** are associated with the **case** construct. Format of the **case** construct is shown in Fig.151. First expression is evaluated. If the evaluated value matches refl, statementl is executed; and the simulator exits the block; else expression is compared with ref2 and in case of a match, Statement2 is executed, and so on. If none of the refl, ref2, t'/c., matches the value of expression, the **default** statement is executed.

```
Case (expression)
Ref1 : statement1;
Ref2 : statement2;
Ref3 : statement3;
. . .
. . .
default: statementd;
endcase
```

Fig. 151 Syntax of case statement

Observations:

- A statement or a group of statements is executed if and only if there is an exact - bit by bit - match between the evaluated expression and the specified refl, ref2, etc.
- For any of the matches, one can have a block of statements defined for execution. The block should appear within the **begin-end** construct.
- There can be only one **default** statement or **default** block. It can appear anywhere in the case statement.
- One can have multiple signal combination values specified for the same statement for execution. Commas separate all of them.

Example 20: 4×1 MUX

```
module mux4_1(a,s,y);
input [3:0] a;
input [1:0] s;
output y;
reg y;
always @ (s)
begin
    case (s)
        2'b00: y = a[0];
        2'b01: y = a[1];
        2'b10: y = a[2];
        2'b11: y = a[3];
        default: y = 1'bz;
    endcase
end
endmodule

module mux4_1_test;
reg [3:0] a;
reg [1:0] s;
wire y;
mux4_1(a,s,y);
```

```
initial
begin
        a = 4'b0000;
        s = 2'b00;
end
always
begin
#2      a = 4'b1101;
#2      s = 2'b01;
#2      s = 2'b10;
#2      s = 2'b11;
end
initial $monitor ($time, "a=%b, s=%b, y=%b",a,s,y);
initial #10 $stop;
endmodule
```

Simulation Results:

```
0       a=0000,     s=00,       y=0
2       a=0010,     s=00,       y=0
4       a=0010,     s=01,       y=1
6       a=0010,     s=10,       y=0
8       a=0010,     s=11,       y=0
```

Example 21: ALU

```
module alu (a,b,m,ci,y,co);
input [3:0] a,b;
input [1:0] m;
input ci;
output [3:0] y;
output co;
reg [3:0] y;
reg co;
always @ (m)
begin
        case (m)
        2'b00: {co,y} = (a+b+ci);
        2'b01: y = ~a;
        2'b10: y = a&b;
        2'b11: y = a^b;;
        default: begin
                        $display ("No Operation");
                        y = 4'bzzzz;
                 end
        endcase
end
endmodule
```

```
module alu_test;
reg [3:0] a,b;
reg [1:0] m;
reg ci;
wire [3:0] y;
wire co;
alu (a,b,m,ci,y,co);
initial
begin
        a = 4'b1011;
        b = 4'b1101;
        ci = 1'b0;
        m = 2'b00;
end
always
begin
        #2      m = 2'b01;
        #2      m = 2'b10;
        #2      m = 2'b11;
end
initial $monitor ($time, "a=%b, b=%b, ci=%b, m=%b, co=%b, y=%b",
a,b,ci,m,co,y);
initial #11 $stop;
endmodule
```

Simulation Results:

```
0   a=1011,   b=1101,      ci=0,        m=00,        co=1,        y=1000
2   a=1011,   b=1101,      ci=0,         m=01,       co=1,        y=0100
4   a=1011,   b=1101,      ci=0,        m=10,        co=1,        y=1001
6   a=1011,   b=1101,      ci=0,         m=11,       co=1,        y=0110
No Operation
8   a=1011,    b=1101,     ci=0,        m=1x,        co=1,        y=zzzz
No Operation
10  a=1011,   b=1101,      ci=0,        m=z0,        co=1,        y=zzzz
```

12.1. Casex and Casez

The **case** statement executes a multiway branching where every bit of the **case** expression contributes to the branching decision. The statement has two variants where some of the bits of the **case** expression can be selectively treated as don't cares - that is, ignored. **Casez** allows **z** to be treated as a don't care. "?" character also can be used in place of **z. casex** treats **x** or **z** as a don't care.

13. SIMULATION FLOW

Different constructs for design description and simulation have been dealt with so far. These can be at different levels of abstraction - gate, data flow, or behavioral level. The constructs to be discussed in the following chapters add to the variety and flexibility. Such elements in different combinations make up the design and simulation modules in Verilog. Further, as an HDL, Verilog has to be an inherently parallel processing language. The fact that all the elements of a digital circuit function and interact continuously conforming to their interconnections demands parallel processing.

- Simulation is carried out in simulation time. The simulator functions with simulation time advancing in (equal) discrete steps.
- At every simulation step a number of active events are sequentially carried out.
- The simulator maintains an event queue - called the "Stratified Event Queue" - with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.
- The active event can be of an update type or evaluation type.

  The evaluation event can be for evaluation of variables, values on nets, expressions, *etc.*

  Refreshing the queue and rearranging it constitutes the update event.
- Any updating can call for a subsequent evaluation and *vice versa*.
- Only after all the active events in a time step are executed, the simulation advances to the next time step.

Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL.

A number of active events can be present for execution at any simulation time step; all may vie for "attention". Amongst these, an event specified at #0 time is scheduled for execution at the end - that is, before simulation advances to the next time step. The order, in which the other events are executed, is essentially simulator-dependent.

13.1. Stratified Event Queue

The events being carried out at any instant give rise to other events - inherent in the execution process.

All such events can be grouped into the following 5 types:

- Active events
- Inactive events - The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.
- Blocking Assignment Events - Operations and processes carried out at previous time steps with results to be updated at the current time step are of this category.
- Monitor Events - The Monitor events at the current time step - **$monitor** and **$strobe** -

are to be processed after the processing of the active events, inactive events, and nonblocking assignment events.

- Future events - Events scheduled to occur at some future simulation time are the future events.

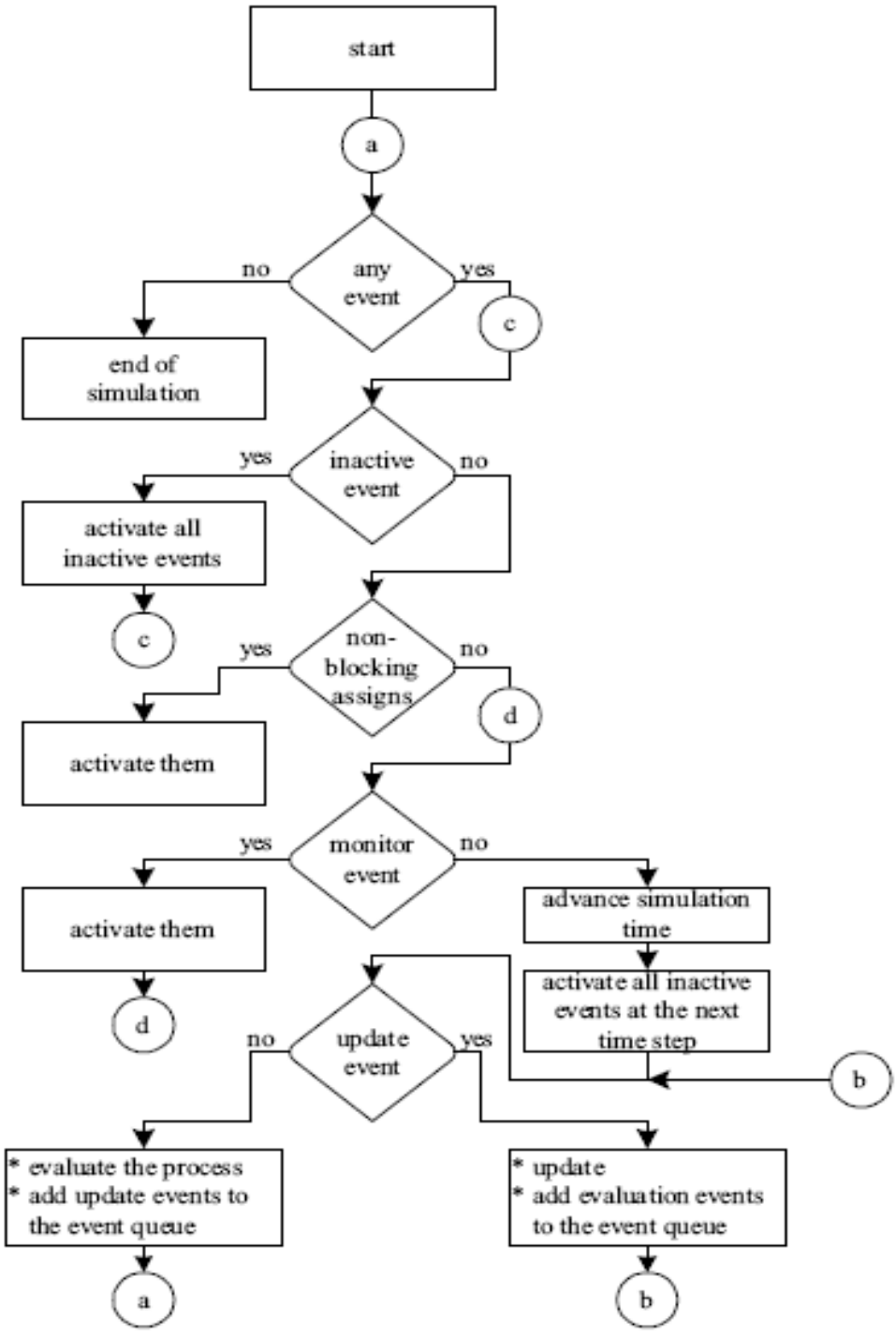


Fig. 152 Simulation Flow

14. if AND **if-else CONSTRUCTS**

```
…….
assignment1;
If (condition) assignment2;
assignment3;
assignment4;
……..
```

The **if** construct checks a specific condition and decides execution based on the result.. After execution of assignmentl, the condition specified is checked. If it is satisfied, assignment2 is executed; if not, it is skipped. In either case the execution continues through assignment3, assignments etc. Execution of assignment2 alone is dependent on the condition. The rest of the sequence remains.

The flowchart equivalent of the execution is shown in Fig.153. If the number of assignments associated with the **if** condition is more than 1, the whole set of them can be grouped within a **begin-end** block.

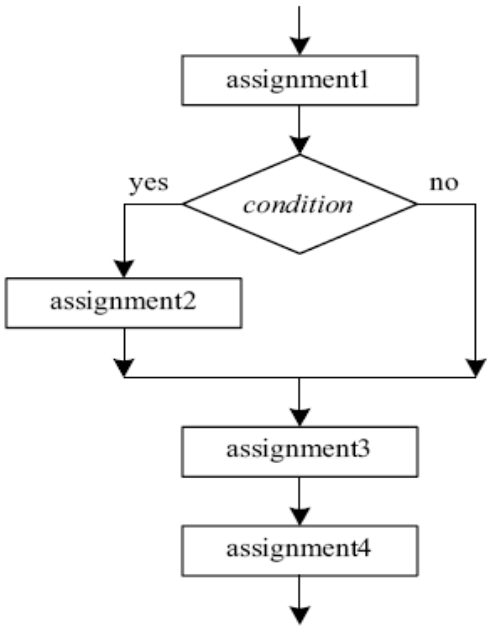


Fig.153 Flow chart of *if* statement

The **if- else** construct is more common and turns out to be more useful than the **if** construct taken alone. Fig.154 shows the same in flowchart form. The design description has two branches; the alternative taken is decided by the condition.

- After the execution of assignmentl, if the *condition* is satisfied, alternative1 is followed and assignment2 and assignments are executed. Assignment4 and assignment 5 are skipped and execution proceeds with assignment6.

- If the *condition* is not satisfied, assignment2 and assignments are skipped and assignment4 and assignments are executed. Then execution continues with assignment6.

```
…….
assignment1;
If (condition)
begin
        assignment2;
        assignment3;
end
else
begin
        assignment4;
        assignment5;
end
……..
```

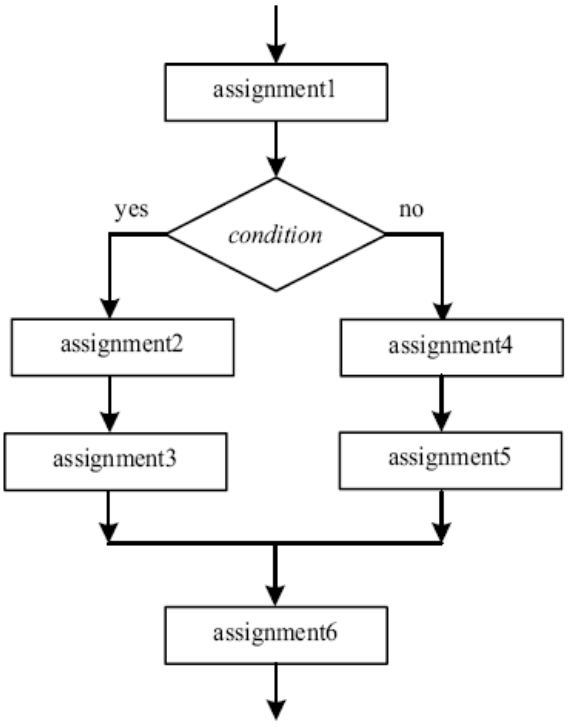


Fig. 154 Flow chart of *if-else* statement

Example 22: 1X4 Demultiplexer

```
module demux1_4 (a,y,s);
input a;
input [1:0] s;
output [3:0] y;
reg [3:0] y;
always@ (a or s)
begin
        if(s=2'b00)
        begin
                y[0] = a;
                y[3:1] = 3'bzzz;
        end
        else if (s=2'b01)
        begin
                y[1] = a;
                {y[3],y[2],y[0]} = 3'bzzz;
        end
        else if (s=2'b10)
        begin
                y[2] = a;
                {y[3],y[1],y[0]} = 3'bzzz;
        else (s=2'b11)
        begin
                y[3] = a;
                y[2:0] = 3'bzzz;
        end
end
endmodule

// Test Bench
module demux1_4_test;
reg a;
reg [1:0] s;
wire [3:0] y;
initial
begin
        a=1'b1; s=2'b00; y=4'b0000;
end
always
begin
        #2 s=2'b01;
        #2 s=2'b10;
        #2 s=2'b11;
end
demux1_4 M1 (a,y,s);
initial $monitor ($time, "s = %b, a = %b, y = %b", s,a,y);
initial #10 $stop;
endmodule
```

15. assign-deassign CONSTRUCT

A behavior block is activated by the event at the beginning. A proper operation demands that all variables with assignments within the block are to be included in the sensitivity list.

The assign — deassign constructs allow continuous assignments within a behavioral block. By way of illustration, consider the following simple block:

always@(posedge clk) a = b;

By way of execution, at the positive edge of elk the value of b is assigned to variable a, and a remains frozen at that value until the next positive edge of elk. Changes in b in the interval are ignored.

As an alternative, consider the block

always@(posedge clk) assign C = d;

Here at the positive edge of elk, C is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as changes in variable C: The assignment here is akin to a direct (one way) electrical connection to C from d established at the positive edge of clk.

Again consider an enhanced version of the above block as

```
always
begin
    @(posedge clk) assign C = d;
    @(negedge clk) deassign C;
end
```

The above block signifies two activities:

(i) At the positive edge of clk, C is assigned the value of d in a continuous manner.

(ii) At the following negative edge of clk, the continuous assignment to C is removed; subsequent changes to d are not passed on to C; it is as though C is electrically disconnected from d.

The above sequence of twin activities is repeated cyclically. In short, assign allows a variable or a net change in the sensitivity list to mandate a subsequent continuous assignment within, **deassign** terminates the assignment done through the **assign**-based statement. The assignment to C in the above two cases is referred to as a "Procedural Continuous Assignment".

Example 23: D Latch

```
module dlatch (d,clr,pr,q,en);
input d,clr,pr,en;
output q;
reg q;
always @ (clr or pr or en or d)
begin
        if (clr) assign q = 1'b0;
        else if (pr) assign q = 1'b1;
        else if (en) assign q = d;
        else deassign q;
end
endmodule
```

16. repeat CONSTRUCT

The repeat construct is used to repeat a specified block a specified number of times. The typical format is

```
……….
repeat (a)
begin
        assignment1;
        asignement2;
        ……
end
……..
```

The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is encountered, a is evaluated. The following block is executed "a" times. If "a" evaluates to **0** or **x** or **z,** the block is not executed.

Example 24: Design of a Memory

```
module memory ();
reg [7:0] mem [9:0];
integer i;
reg clk;
always
begin
        repeat (10)
        begin
                @negedge clk
                mem[i] = i*4;
                i = i+1;
        end
        repeat (10)
```

```
begin
        @negedge clk
        i = i-1;
        $display ("t = %0d, mem[%0d] = %0d", $time, i, mem[i]);
end
initial
begin
        clk = 1'b0;
        i=0;
        #42 $stop;
end
always #2 clk = ~clk;
endmodule
```

17. for LOOP

The **for** loop in Verilog is quite similar to the **for** loop in C. The typical format is

```
…….
for (assignment1; expression; assignment2)
loop statements;
…….
```

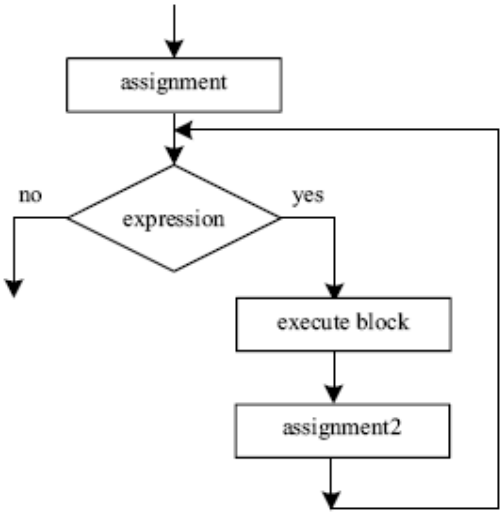


Fig. 155 Flow chart of *for* loop

It has four parts; the sequence of execution is as follows:

- Execute assignment1.
- **Evaluate expression.**
- If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
- If *expression* evaluates to the false state (0), exit the loop.
- Execute assignment2. Go to step 2.

Operation of the loop is shown in Fig.155 in flowchart form. In general, whenever one has to accommodate alternatives for execution, the **if** and **if-else** constructs are preferred. Whenever a sequence of assignments is to be done repeatedly with an index for termination, the **for** construct is preferred.

Example 25: Design of a Memory

```
module memory ();
reg [7:0] mem [9:0];
integer i;
reg clk;
always
begin
        for (i=0; i<10;i=i+1)
        begin
                @negedge clk
                mem[i] = i*4;
        end
        for (i=0; i<10;i=i+1)
        begin
                @negedge clk
                $display ("t = %0d, mem[%0d] = %0d", $time, i, mem[i]);
        end
        initial
        begin
                clk = 1'b0;
                #42 $stop;
        end
        always #2 clk = ~clk;
        endmodule
```

18. THE disable CONSTRUCT

There can be situations where one has to break out of a block or loop. The **disable** statement terminates a named block or task. Control is transferred to the statement immediately following the block. Conditional termination of a loop, interrupt servicing, etc., are typical contexts for its use. Often the disabling is carried out from within the block itself. The **disable** construct is functionally similar to the break in C.

Observations:

- The **disable** statement has to have a block (or task) identifier tagged to it - in this respect it differs from "break" in C.
- Once encountered, it terminates execution of the block; the following statements within the block are not executed.
- Typically it can be used to handle exceptions to regularly assigned activities for example, Interrupt, Hold, Reset, *etc.*

19. while LOOP

The typical format for while loop is

while (expression) statement;

The Boolean expression is evaluated. If it is **true,** the statement (or block of statements) is executed and expression evaluated and checked. If the expression evaluates to false, the loop is terminated and the following statement is taken for execution. If the expression evaluates to **true,** execution of statement (block of statements) is repeated. Thus the loop is terminated and broken only if the expression evaluates to false. The flowchart for the while loop is shown in Fig.156.

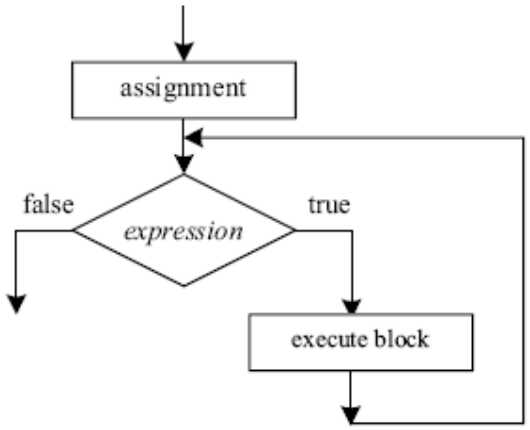


Fig.156 Flow chart of *while* loop

Observations:

- Whenever the **while** construct is used, event or time-based activity flow within the block has to be ensured.
- With the **while** construct the expression associated with the keyword **while** must become **false** through the execution of assignments inside the block. Otherwise we end up with an endless looping within the block, causing a deadlock.
- There may be situations where we have to **wait** in a loop while an **event** external to it changes to trigger an activity. The **wait** construct is to be used for such situations and not **while.** With the **wait** construct the activity is scheduled and execution continued with the other activities. With the **while** construct until the associated loop is not complete, other activities are not taken up.

Example 26: Design of a Memory

```
module memory ();
reg [7:0] mem [9:0];
integer i;
reg clk;
always
begin
```

```
while (i<10)
begin
        @negedge clk
        mem[i] = i*4;
        i = i+1;
end
while (i<10)
begin
        @negedge clk
        i = i-1;
        $display ("t = %0d, mem[%0d] = %0d", $time, i, mem[i]);
end
initial
begin
        clk = 1'b0;
        i = 0;
        #42 $stop;
end
always #2 clk = ~clk;
endmodule
```

20. forever LOOP

Repeated execution of a block in an endless manner is best done with the **forever** loop (compare with repeat where the repetition is for a fixed number of times).

Example 27: Design of a Memory

```
module memory ();
reg [7:0] mem [9:0];
integer i;
reg clk;
always
begin: mem_write
        forever@negedge clk
        begin
                if (i>=10) disable mem_write;
                mem[i] = i*4;
                i = i+1;
        end
end
always
begin: mem_read
        forever@negedge clk
        begin
                if (i>=20) disable mem_read;
                $display ("t = %0d, mem[%0d] = %0d", $time, i, mem[i]);
                i = i+1;
```

```
        end
end
initial
begin
        clk = 1'b0;
        i = 0;
        #42 $stop;
end
forever #2 clk = ~clk;
endmodule
```

21. PARALLEL BLOCKS

All the procedural assignments within a **begin-end** block are executed sequentially. The **fork-join** block is an alternate one where all the assignments are carried out concurrently (The nonblocking assignments too can be used for the purpose.). One can use a **fork-join** block within a **begin-end** block or vice versa.

```
module fk_jn_a;
integer a;
initial
begin
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
end
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulatiom results
# a=0, t=0
# a=1, t=1
# a=2, t=3
# a=3, t=6
```

(a)

```
module fk_jn_b;
integer a;
initial
fork
        a=0;
 #1     a=1;
 #2     a=2;
 #3     a=3;
 #4     $stop;
join
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=2
# a=3, t=3
```

(b)

Fig. 157 (a) begin-end block and simulation results (b) fork-join block and simulation results

```
module fk_jn_c;
integer a;
initial
begin
 #5        a=5;
           fork
            #1        a=0;
            #2        a=1;
            #3        a=2;
            #4        a=3;
            #5        $stop;
           join
end
initial $monitor ("a=%0d, t=%0d",a,$time);
endmodule
```

```
//Simulation results
# a=x, t=0
# a=5, t=5
# a=0, t=6
# a=1, t=7
# a=2, t=8
# a=3, t=9
```

(a)

```
module fk_jn_d;
integer a;
initial
fork
 #5        a=5;
           begin
            #1        a=0;
            #2        a=1;
            #3        a=2;
            #4        a=3;
            #5        $stop;
           end
join
initial $monitor ("a=%0d, t=%0d",a,$time);
endmodule
```

```
//Simulation results
# a=x, t=0
# a=0, t=1
# a=1, t=3
# a=5, t=5
# a=2, t=6
# a=3, t=10
```

(b)

Fig.158 (a) fork-join block within begin-end block
(b) begin-end block within fork-join block

22. Force-release CONSTRUCT

When debugging a design with a number of instantiations, one may be stuck with an unexpected behavior in a localized area. Tracing the paths of individual signals and debugging the design may prove to be too tedious or difficult. In such cases suspect blocks may be isolated, tested, and debugged and *status quo ante* established. The **force-release** construct is for such a localized isolation for a limited period. Figure 8.53 shows the use of a **force-release** construct in a test bench. The assignment

force a = 1'bO;

forces the variable **a** to take the value **0.**

force b = c&d;

forces the variable **b** to the value obtained by evaluating the expression **c&d.** Subsequently a few assignments are made in the test bench. At a later part of the test bench, **a** and **b** are released that is, their original assignments are restored. The assignments here have specific characteristics:

force a = 1 'bO;

force b = c&d;

assignmentl;

assignment2;

release a;

release b;

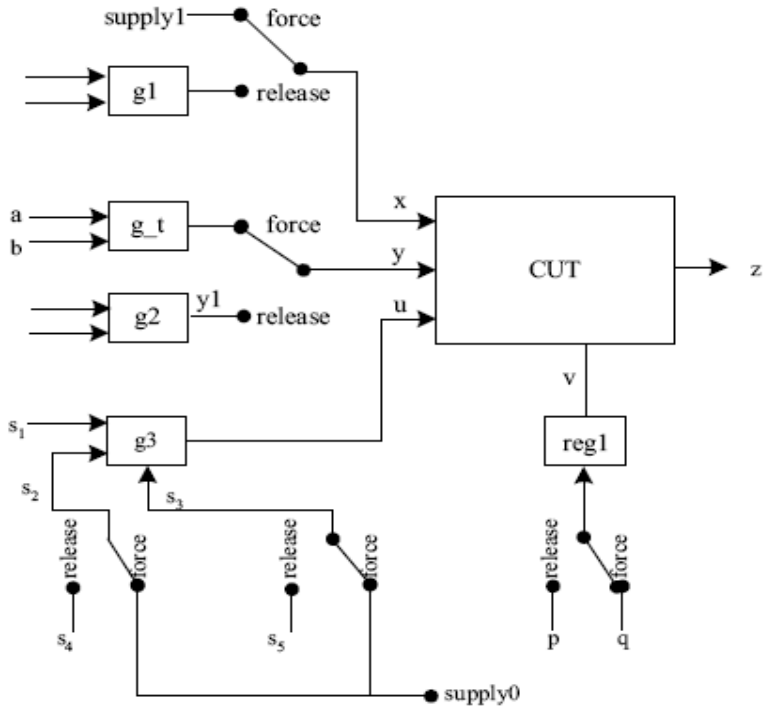


Fig.159 An example circuit to illustrate force-release construct

Observations:

- The force-release construct is similar to the assign-deassign construct. The latter construct is for conditional assignment in a design description. The **force-release** construct is for "**short time**" assignments in a test-bench. Synthesis tools will not support the force-release constructs.
- The **force-release** construct is equally valid for net-type variables and **reg**-type variables. The net-type variables revert to their normal values on release. With **reg**-type variables the value forced remains until another assignment to the reg.
- The variable, on which the values are forced during testing, must be properly dereferenced.
- In the illustration above, each variable was forced one at a time. It was done only to simplify the illustration sequence and focus attention on the possible use of the construct. In practice, different variables can be forced together before the special debug sequence. Their release too can be together.

23. EVENT

The keyword **event** allows an abstract event to be declared. The event is not a data type with any specific values; it is not a variable **(reg)** or a net. It signifies a change that can be used as a trigger to communicate between modules or to synchronize events in different modules. One typical example for event is

……
event change
……
always
……..
……→ change
…….
always @ change
…….....

change has been declared as an **event.** In the course of execution of an **always** block, the event is triggered. The operator signifies the triggering. Subsequently, another activity can be started in the module by the event change. The **always**@(change) block activates this. The event change can be used in other modules also by proper dereferencing; with such usage an activity in a module can be synchronized to an event in another module.

The **event** construct is quite useful, especially in the early stages of a design. It can be used to establish the functionality of a design at the behavioral level; it allows communication amongst different instantiated modules without associated inputs or outputs.

```
module rec_tst;
reg clk,di; integer n,i;
reg[8:1] aa;wire [8:1] a;
always #2 clk = ~clk;
rec rrcc(a,di,clk);
always @(rrcc.buf_ful) $display("t=%0d, aa=%h, a=%h",$time,aa,a);
initial
        for (n=1;n<3000;n=n+113) begin
                                aa=n;i=0;
                                repeat(8)@(posedge clk)
                                        begin
                                                i=i+1;
                                                di=aa[i];
                                                //$write("bb=%b",aa[i]);
                                         end
                                #3      i=0;
                                end //Why '#3'?
initial clk=1'b0; initial #400  $stop;
endmodule
```

Fig. 160 A module to illustrate *event* construct

UNIT-IV

SWITCH LEVEL MODELLING SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES

Contents

- Basic Transistor Switches
- CMOS Switch
- Bi – directional Gates
- Time Delays with Switch Primitives
- Instantiations with Strengths and Delays
- Strength Contention with Trireg Nets
- Parameters
- Path Delays
- Module Parameters
- System Tasks and Functions
- File – Based Tasks and Functions
- Compiler Directives
- Hierarchical Access
- User-defined Primitives (UDP)

UNIT – IV

SWITCH LEVEL MODELLING

In today's environment the MOS transistor is the basic element around which a VLSI is built. Designers familiar with logic gates and their configurations at the circuit level may choose to do their designs using MOS transistors. Verilog has the provision to do the design description at the switch level using such MOS transistors. Switch level modeling forms the basic level of modeling digital circuits. The switches are available as primitives in Verilog; they are central to design description at this level. Basic gates can be defined in terms of such switches. By repeated and successive instantiation of such switches, more involved circuits can be modeled.

Designers familiar with logic gates, digital functional blocks, and their interplay can successfully carry out a complete VLSI design without any involvement at the switch level.

1. BASIC TRANSISTOR SWITCHES

Consider an NMOS transistor of the depletion type. When used in a digital circuit, it can be in one of three modes:

- $V_G < V_S$ **where** V_G and V_S **are the gate and source voltages with respect to the drain:** The transistor is OFF and offers very high impedance across the source and the drain. It is in the z state.
- $V_G \equiv V_S$**:** The transistor is in the active region. It presents a resistance between the source and the drain. The value depends on the technology. Such a resistive state of the transistor can be modeled in Verilog. A transistor in this mode can be represented as a resistance in Verilog - as **pull1** or **pull0** depending on whether the drain is connected to **supply1** or source is connected to **supply0.**
- $V_G > V_S$**:** The transistor is fully turned on. It presents very low resistance (~0 Ω) between the source and drain.

An enhanced-mode NMOS transistor also has the above three modes of operation. Similar modes are possible for the PMOS transistor also.

1.1. Basic Switch Primitives

Different switch primitives are available in Verilog. Consider an **nmos** switch. A typical instantiation has the form

nmos (out, in, control);

nmos - a keyword - represents an NMOS transistor functioning as a switch. The switch has three terminals - in, out and control. When the control input is at 1 (high) state, the switch is on. It connects the input lead to the output side and offers zero impedance. When

the control input is low, the switch is OFF and output is left floating (**z** state). If the control is in the **z** or the **x** state, output may take corresponding values.

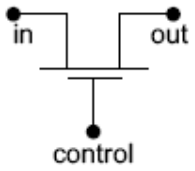
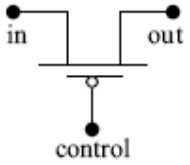


Fig. 161 (a) nmos switch



(b) pmos switch

The keyword **pmos** represents a PMOS transistor functioning as a switch. The PMOS switch has three terminals. A typical instantiation of the switch has the form

pmos (out, in, control);

When the control is at 1 (high) state, the switch is off. Output is left floating. When control is at 0 (low) state, the switch is on, input is connected to output, and output is at the same state as input. For other input values, output is at other values.

When instantiating an **nmos** or a **pmos** switch, a name can be assigned to the switch. But the name is not essential. The **nmos** and **pmos** switches function as unidirectional switches.

1.2. Resistive Switches

nmos and pmos represent switches of low impedance in the on-state, rnmos and rpmos represent the resistive counterparts of these respectively. Typical instantiations have the form

rnmos (output, input, control);

rpmos (output, input, control);

With rnmos if the control 1 input is at 1 (high) state, the switch is ON and functions as a definite resistance. It connects input to output through a resistance. When control 1 is at the 0 (low) state, the switch is OFF and leaves output floating.

The rpmos switch is ON when control is at 0 (low) state. It inserts a definite resistance between the input and the output signals but retains the signal value.

Because rpmos and rnmos are resistive switches, they reduce the signal strength when in the on state. The reduced strength is mostly one level below the original strength. The only exceptions are small and hi-z. For these the strength and the state remain unaltered. The

rpmos and rnmos switches function as unidirectional switches; the signal flow is from the input to the output side.

Input strength	Output strength
Supply – drive	Pull – drive
Strong – drive	Pull – drive
Pull – drive	Weak – drive
Weak – drive	Medium – capacitive
Large – capacitive	Medium – capacitive
Medium – capacitive	Small – capacitive
Small – capacitive	Small – capacitive
High impedance	High impedance

Table 25 Input and Output strengths for Resistive Switches

1.3. pullup and pulldown

A MOS transistor functions as a resistive element when in the active state. Realization of resistance in this form takes less silicon area in the IC as compared to a resistance realized directly, **pullup** and **pulldown** represent such resistive elements. A typical instantiation here has the form

pullup (X);

Here the net X is pulled up to the **supply1** through a resistance. Similarly, the instantiation

pulldown (y);

pulls y down to the **supply0** level through a resistance. The **pullup** and **pulldown** primitives can be used as loads for switches or to connect the unused input ports to V_{CC} or GND, respectively. They can also form loads of switches in logic circuits.

The default strengths for **pullup** and **pulldown** are **pull1** and **pull0** respectively. One can also specify strength values for the respective nets. For example,

pullup (**strong1**) (X);

specifies a resistive **pullup** of net X to **supply1.** One can also assign names to the **pullup** and **pulldown** primitives. Thus

pullup (strong1) rs (X)

represents an instantiation of **pullup** designated rs having strength **strong1.**

Difference between **tri** and **pullup** or **pulldown** is to be understood clearly, **pullup** is a functional element; it represents a resistive connection to **supply1.** In contrast **tri1** is a type of net; in the absence of an assignment, it remains connected to **supply1.** A similar difference exists between **pulldown** and **tri0.**

Example 28: CMOS Inverter

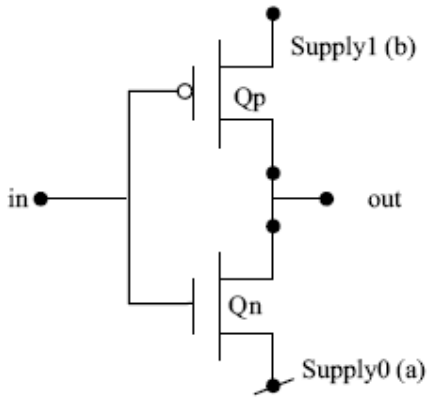


Fig.162 CMOS Inverter

A CMOS inverter is formed by connecting an **nmos** and a **pmos** switch in series across the supply. The output terminals are joined together to form the common output. Similarly, the input is used as the common control input to both the switches.

```
module cmosinv (out,in);
        input in;
        output out;
        supply0 a;
        supply1 b;
        nmos(out,a,in);
        pmos(out,b,in);
endmodule
```

Example 29: CMOS NOR gate

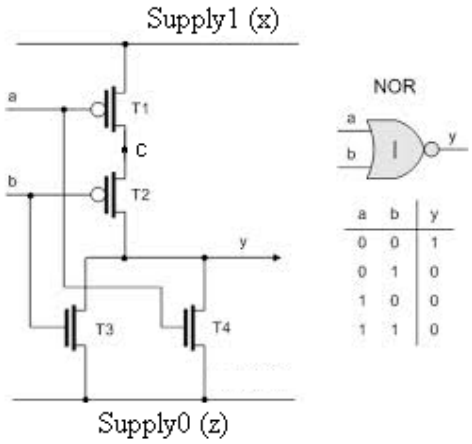


Fig.163 CMOS NOR Gate

```
module cmosnor (y,a,b);
        input a,b;
        output y;
        wire c;
        supply0 z;
        supply1 x;
        pmos(c,x,a);
        pmos(y,c,b);
        nmos(y,z,a);
        nmos(y,z,b);
endmodule
```

Example 30: CMOS NAND Gate

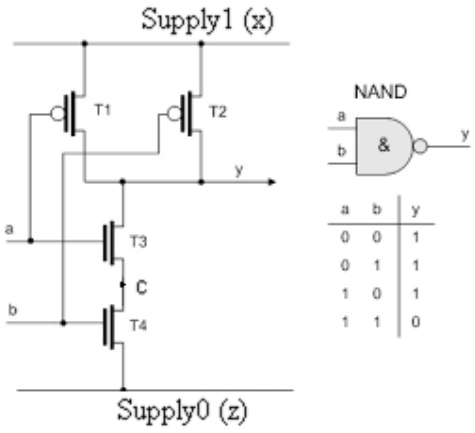


Fig.164 CMOS NAND Gate

2. CMOS SWITCH

A CMOS switch is formed by connecting a PMOS and an NMOS switch in parallel - the input leads are connected together on the one side and the output leads are connected together on the other side. Fig.165 shows the switch so formed. It has two control inputs:

N_Control turns ON the NMOS transistor and keeps it ON when it is in the 1 state.

P_Control turns ON the PMOS transistor and keeps it ON when it is in the 0 state.

The CMOS switch is instantiated as shown below.

cmos csw (out, in, Ncontrol, P_control);

Significance of the different terms is as follows:

cmos : The keyword for the switch instantiation

CSW: Name assigned to the switch in the instantiation

out: Name assigned to the output variable in the instantiation

in: Name assigned to the input variable in the instantiation

N_Control: Name assigned to the control variable of the NMOS transistor in the instantiation

P_Control: Name assigned to the control variable of the PMOS transistor in the instantiation

Switch – 1:

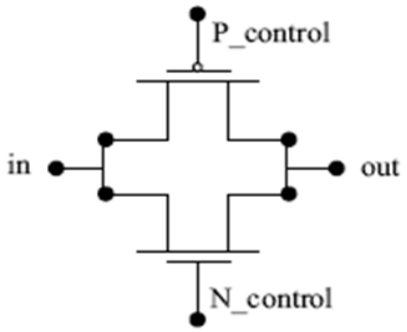


Fig. 165 CMOS Switch

```
module cmos(out,in,nctr,pctr);
input in,nctr,pctr;
output out;
nmos gn(out,in,nctr);
pmos(out,in,pctr);
endmodule
```

Switch – 2:

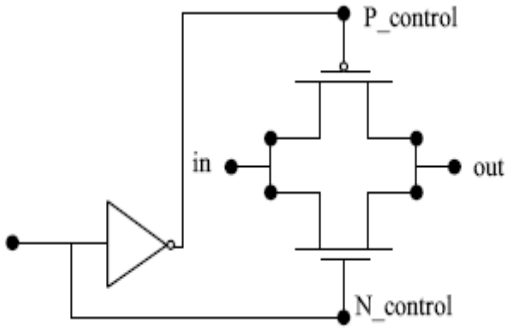


Fig. 166 CMOS Switch

```
module cmos(out,in,con);
input in,con;
output out;
wire pctr;
not gn(pctr,con)
nmos gn(out,in,nctr);
cmos gc(out,in,con,pctr);
endmodule
```

Example 31: A RAM Cell

Fig.167 shows a basic ram cell with facilities for writing data, storing data, and reading data. When switch sw2 is on, qb - the output of inverter g1 - forms the input to the inverter g2 and vice versa. The g1-g2 combination functions as a latch and freezes the last state entry before SW2 turns on.

The step-by-step function of the cell is as follows:

- When WSb (write/store) is high, switch SW1 is ON, and switch SW2 OFF. With Sw1 on, input Din is connected to the input of gate g1 and remains so connected.
- When WSb goes low, din is isolated, since SW1 is OFF. But SW2 is ON and the data remains latched in the latch formed by g1-g2. In other words the data Din is stored in the RAM cell formed by g1-g2.
- When RD (Read) goes active (=1), the latched state is available as output Do. Reading is normally done when the latch is in the stored state.

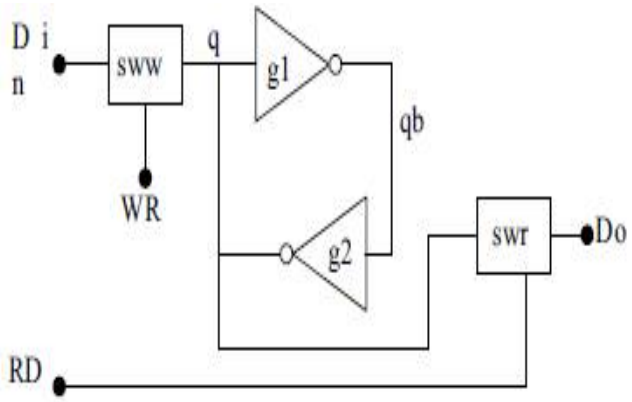


Fig. 167 A Dynamic RAM Cell

```
module ram1(do,din,wr,rd);
output do; input din,wr,rd;
wire qb,q;
tri do;
scw sww(q,din,wr), swr(do,q,rd);
not (pull1,pull0) n1(qb,q), n2(q,qb);
endmodule

module scw(out,in,n_ctr);
output out; input in,n_ctr;
wire p_ctr;
assign p_ctr = ~n_ctr;
cmos sw(out,in,n_ctr,p_ctr);
endmodule
```

3. BI-DIRECTIONAL GATES

The gates discussed so far **(nmos, pmos, rnmos, rpmos, rcmos)** are all unidirectional gates. When turned ON, the gate establishes a connection and makes the signal at the input side available at the output side. Verilog has a set of primitives for bi-directional switches as well. They connect the nets on either side when ON and isolate them when OFF. The signal flow can be in either direction. None of the continuous-type assignments at higher levels dealt with so far has a functionality equivalent to the bi-directional gates. There are six types of bidirectional gates.

3.1 **tran** and **rtran**

The **tran** gate is a bi-directional gate of two ports. When instantiated, it connects the two ports directly. Thus the instantiation

tran (s1, s2);

connects the signal lines s1 and s2. Either line can be **input, inout** or **output, rtran** is the resistive counterpart of **tran.**

3.2 tranif1 and rtranif1

tranif1 is a bi-directional switch turned ON/OFF through a control line. It is in the ON-state when the control signal is at 1 (high) state. When the control line is at state 0 (low), the switch is in the OFF state. A typical instantiation has the form

tranif1 ***(s1, s2, c);***

Here C is the control line. If c=l, s1 and s2 are connected and signal transmission can be in either direction, **rtranif1** is the resistive counterpart of **tranif1.** It is instantiated in an identical manner.

3.3 tranif0 and rtranif0

tranif0 and **rtranif0** are again bi-directional switches. The switch is OFF if the control line is in the 1 (high) state, and it is ON when the control line is in the 0 (low) state. A typical instantiation has the form

tranif0 ***(s1, s2, C);***

With the above instantiation, if C = 0, Si and S2 are connected and signal transmission can be in either direction. If C = 1, the switch is OFF and Si and S2 are isolated from each other. **rtranif0** is the resistive counterpart of **tranif0.**

Observations:

- Any instantiation of a bi-directional switch of the above types can be given a name. But a name is not essential. It is true of the other switches also.
- With the bi-directional switches the signal on either side can be of **input, output,** or **inout** type. They can be nets or appearing as ports in the module. But the type declaration on the two sides has to be consistent.
- The connections to the bi-directional terminals of each of the bi-directional switches have to be scalars or individual bits of vectors and not vector themselves.
- In the above instantiation s1 can be an input port in a module. In that case, s2 has to be a net forming an input to another instantiated module or circuit block. s2 can be of **output** or **inout** type also. But it cannot be another input port.
  - s1 and s2 - both cannot be output ports.
  - s1 and s2 - both can be **inout** ports.
- With **tran, tranif1,** and **tranif0** bi-directional switches if the input signal has strength **supply1 (supply0),** the output side signal has strength **strong1 (strong0).** For all other strength values of the input signal, the strength value of the output side signal retains the strength of the input side signal.
- With **rtran, rtranif1** and **rtranif** switches the output side signal strength is less than that of the input side signal. The strength reduction is on the lines shown in Table **26** for **rnmos, rpmos,** and **rcmos** switches.

Type of Bi-directional switch	Typical instantiation	Condition to be ON	Remarks
2 port	`tran` (a, b);	Always ON (if instantiated)	Acts essentially as a buffer
	`rtran` (a, b);	– do –	Acts essentially as a buffer with reduction in the strength of the signal
3 port	`tranif1` (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation
	`tranif0` (a, b, c);	ON if c = 0	– do –
	`rtranif1` (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation; signal strength on the output side is lower than that on the input side
	`rtranif0` (a, b, c);	ON if c = 0	– do –

Table 26 Bidirectional Gates

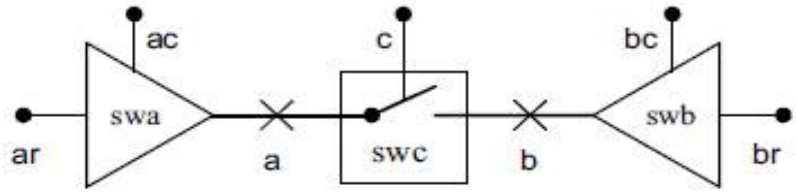


Fig. 168 Bus switching using Bidirectional Gates

Example 32: RAM Cell

Fig.169 shows a single RAM cell. It can be instantiated in vector form to form a full-fledged ram. a_d is the decoded address line. When active, it turns on the bi-directional switch g3 and establishes a two-way connection between net ddd and net q. g1 and g2 together form a latch in feedback fashion. When g3 is OFF, the latch stores the state it was last in. It is connected to ddd through g3 by activating a_d for writing and reading. The following are possible after such selection and connection:

- When **wr = 1, cmos** gate **g4** turns **ON;** the data at the input port **di** (with strength **strongO / strongl)** are connected to **q** through **ddd.** It forces the latch to its state - since **q** has strength **pullO / pulll** only - **di** prevails here. This constitutes the write operation.
- When **rd = 1, cmos** gate **g5** turns ON. The net **ddd** is connected to the output net **do.** The data stored in the latch are made available at the output port **do.** This constitutes the read operation.

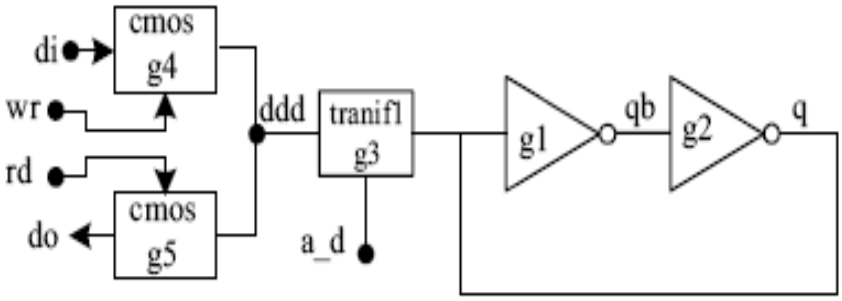


Fig. 169 A RAM Cell using Bidirectional Gates

```
module ram_cell1(do,di,wr,rd,a_d);
output do; input di,wr,rd,a_d;
wire ddd,q,qb,wrb,rdb;
not (rdb,rd), (wrb,wr);
not(pull1,pull0) (q,qb), (qb,q);
```

tranif1 g3(ddd,q,a_d);
cmos g4(ddd,di,wr,wrb), g5(do,ddd,rd,rdb);
endmodule

4. TIME DELAYS WITH SWITCH PRIMITIVES

For example, an NMOS switch instantiated as

nmos g1 (out, in, **Ctrl);**

has no delay associated with it. The instantiation

nmos (delayl) g2 (out, in, Ctrl);

has delayl as the delay for the output to rise, fall, and turn OFF. The instantiation

nmos (delay_r, delay_f) g3 (out, in, ctrl);

has delay_r as the rise-time for the output. delay_f is the fall-time for the output. The turn-off time is zero. The instantiation

nmos (delay_r, delay**_f,** delay_o) g4 (out, in, ctrl);

has delay_r as the rise-time for the output. delay_f is the fall-time for the output delay_0 is the time to turn OFF when the control signal ctrl goes from 0 to 1. Delays can be assigned to the other uni-directional gates **(rcmos, pmos, rpmos, cmos,** and **rcmos)** in a similar manner. Bi-directional switches do not delay transmission - their rise- and fall-times are zero. They can have only turn-on and turn-off delays associated with them, **tran** has no delay associated with it.

trainf1 (delay_r, delay_f) g5 (out, in, ctrl);

represents an instantiation of the controlled bi-directional switch. When control changes from 0 to 1, the switch turns on with a delay of delay_r. When control changes from 1 to 0, the switch turns off with a delay of delay_f.

transif1 (delayO) g2 (out, in, ctrl);

represents an instantiation with delayO as the delay for the switch to turn on when control changes from 0 to 1, with the same delay for it to turn off when control changes from 1 to 0. When a delay value is not specified in an instantiation, the turn-on and turn-off are considered to be ideal that is, instantaneous. Delay values similar to the above illustrations can be associated with **rtranif1, tranif0,** and **rtranif0** as well.

5. INSTANTIATIONS WITH STRENGTHS AND DELAYS

In the most general form of instantiation, strength values and delay values can be combined. For example, the instantiation

nmos (strong1, strong0) (delay_r, delay_f, delay_o) gg (s1, s2, ctrl) ;

means the following:

- It has strength **strong0** when in the low state and strength **strong1 when** in the high state. When output changes state from low to high, it has a delay time of delay_r.
- When the output changes state from high to low, it has a delay time of delay_f.
  - When output turns-off it has a turn-off delay time of delay _o.

rnmos, pmos, and **rpmos** switches too can be instantiated in the general form in the same manner. The general instantiation for the bi-directional gates too can be done similarly.

6. STRENGTH CONTENTION WITH TRIREG NETS

Strength contention in transistor switches can be resolved by using **trireg** nets. Such storage can be assigned one of three strengths - **large, medium,** or **small.** Driving such a net from different sources can lead to contention; the relative strength levels of the sources also have a say in the signal level taken by the net.

EXAMPLE 33:

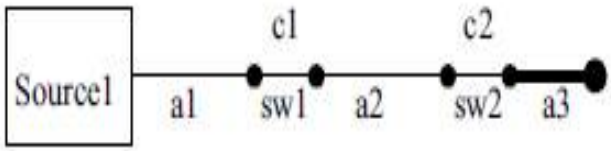


Fig. 170 An Example circuit to illustrate strength contention in switch primitives

```
module demo_1;
trireg(large)a3; trireg(small)a2; wire a1; reg c1,c2,b;
buf(strong1,strong0) source1(a1,b);
tranif1 sw1(a2,a1,c1), sw2(a3,a2,c2);
initial begin
        $display("t\ta1\tc1\ta2\tc2\ta3");
    #0  {c1,c2,b}=3'b111; #1 {c1,c2,b}=3'b011; #1 {c1,c2,b}=3'b001;
    #1  {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100; #1 {c1,c2,b}=3'b000;
    #1  {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100;
    #1  {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000;
    #1  {c1,c2,b}=3'b001; #1 {c1,c2,b}=3'b101; #1 {c1,c2,b}=3'b111;
    #1  $stop;
      end
initial $monitor("%0d\t%b\t%b\t%b\t%b\t%b",$time,a1,c1,a2,c2,a3);
endmodule
```

Fig. 171 Module for the fig. 170

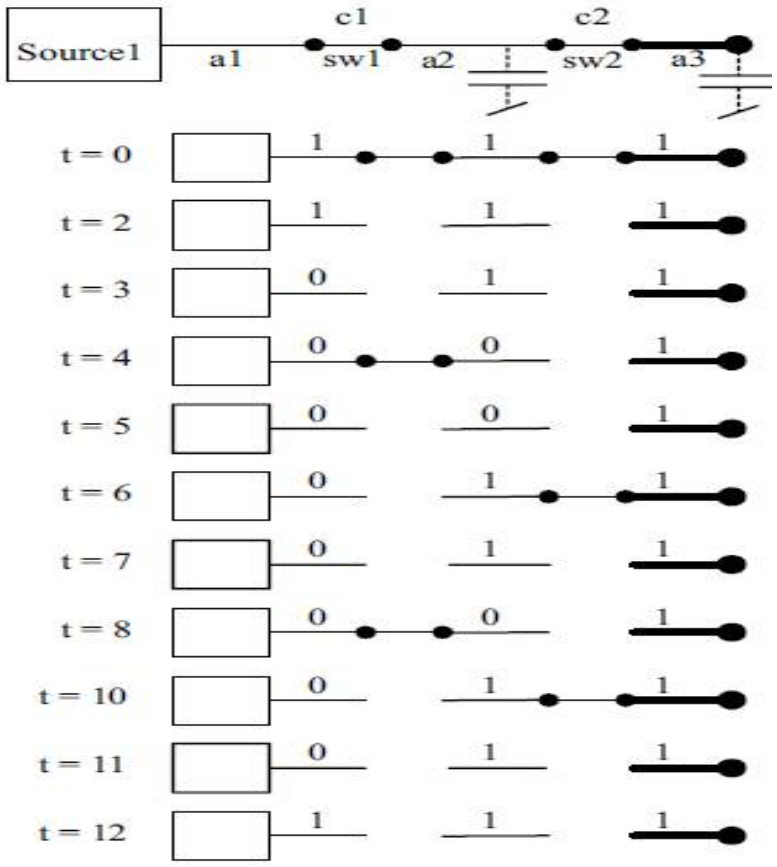


Fig. 172 Changes in signal values at different times

#t	a1	c1	a2	c	a3
#0	1	1	1	1	1
#1	1	0	1	1	1
#2	1	0	1	0	1
#3	0	0	1	0	1
#4	0	1	0	0	1
#5	0	0	0	0	1
#6	0	0	1	1	1
#7	0	0	1	0	1
#8	0	1	0	0	1
#9	0	0	0	0	1
#10	0	0	1	1	1
#11	0	0	1	0	1
#12	1	0	1	0	1
#13	1	1	1	0	1
#14	1	1	1	1	1

Fig. 173 Simulation Results

SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES

7. PARAMETERS

The parameter constructs facilitate such flexibility. Constants signifying timing values, ranges of variables, wires, etc., can be specified in terms of assigned names. Such assigned names are called parameters. The parameter values can be specified and changed to suit the design environment or test environment. Such changes are effected and frozen at instantiation. The assigned values cannot change during testing or synthesis. In this respect a parameter is different from a net or a variable.

Two types of parameters are of use in modules:

- Parameters related to timings, time delays, rise and fall times, etc., are technology-specific and used during simulation. Parameter values can be assigned or overridden with the keyword **"specparam"** preceding the assignments.
- Parameters related to design, bus width, and register size are of a different category. They are related to the size or dimension of a specific design; they are technology-independent. Assignment or overriding is with assignments following the keyword **"defparam".**

7.1 Timing-Related Parameter

Example 34: Half Adder Module

```
module ha_l(s,ca,a,b);
input a,b; output s,ca;
xor #(1,2) (s,a,b);
and #(3,4) (ca,a,b);
endmodule

//test-bench module tstha_l();
reg a,b; wire s,ca;
ha_l hh(s,ca,a,b);
initial begin a=0;b=0; end
always begin #5 a=l;b=0; #5 a=0;b=l; #5 a=l;b=l; #5 a=0;b=0; end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum = %b "
,a,b,ca,s);
initial #30 Sstop;
endmodule
```

7.2 Parameter Declarations and Assignments

Declaration of parameters in a design as well as assignments to them can be effected using the keyword **"Parameter."** A declaration has the form

parameter alpha = *a*, beta = *b*

where

- **parameter** is the keyword,
- alpha and beta are the names assigned to two parameters and
- *a, b* are values assigned to alpha and beta, respectively.

In general *a* and *b* can be constant expressions. The parameter values can be overridden during instantiation but cannot be changed during the run-time. If a parameter assignment is made through the keyword **"localparam,"** its value cannot be overridden.

Observations:

- *As* mentioned earlier, **parameters** are constants which can be altered during compilation but not during runtime.
- A **Parameter** can be signed or unsigned in nature; it can be an integer or a real number.
- Its nature - signed or not, real or integral type as well as range - can be specified at the time of declaration or decided by default based on assignment.

Examples

parameter a = 3; // a is a positive integer

parameter b = - 3; // b is a signed integer

parameter c = 3.0, d = 3.0e2; //c and d are unsigned real numbers.

In all the above cases the **parameter** type and range are decided by default.

parameter integer e = 3; /* e is declared to be an integer type of parameter and assigned the value 3. */

parameter real f = 3.0; /* f is declared to be a real unsigned real number and assigned the value 3. */

In the last two cases the **parameter** type is declared explicitly and remains so.

Whenever a **parameter** value is overridden during instantiation, type, signed/unsigned, *etc.*, remain unchanged.

Example 35: Half Adder Module

```
module ha_2(s,ca,a,b);
input a,b; output s,ea;
parameter dllr=l,dl2f=2,dl3r=3,dl4f=4;
xor #(dllr,dl2f) (s,a,b);
and #(dl3r,dl4f) (ca,a,b);
endmodule

//test-bench module tstha_2();
reg a,b; wire s,ea;
ha_2 hh(s,ca,a,b);
initial begin a=0;b=0; end
always begin #5 a=l;b=0; #5 a=0;b=l; #5 a=l;b=l; #5 a=0;b=0; end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum = %b "
,a,b,ca,s);
initial #30 Sstop;
endmodule
```

7.3 Type Declarations for Parameters

The above examples do not have any type declaration statements for the parameters dllr, dl2f, dl3r, and dl4f. However, integer value assignments are made to each of them; implicitly they are taken as integers by the simulator. But in general one can use constant expressions on the right-hand side of the assignments. For example,

parameter dllr **=1,** dl2f=dllr+ **1,** dl3r=3 , dl4f= dl2f*2;

As mentioned earlier, all four parameters are automatically taken as integers by the simulator. If the above statement is modified as

parameter dllr **=1,** dl2f =dllr + **1.0,** dl3r =3 , dl4f = dl2P2;

the parameter types will be radically different, dllr and dl3r will be treated as integers but dl2f and hence dl4f will be treated as real. However, the numerical values assigned will remain unaltered and hence the simulation results too will be the same.

8. PATH DELAYS

The time delays discussed so far are all delays associated with individual operations or activities in a module. They refer to basic circuit elements in a design - at the microlevel itself. These are called "distributed delays" in LRM. Verilog has the provision to specify and check delays associated with total paths - from any input to any output of a module. Such paths and delays are at the chip or system level. They are referred to as "module path delays." Constructs available make room for specifying their paths and assigning delay values to them

- separately or together.

8.1 Specify Blocks

Module paths are specified and values assigned to their delays through **specify** blocks. They are used to specify rise time, fall time, path delays pulse widths, and the like.

specify

specparam rise_time = 5, fall_time = 6; (a =>b) = (rise_time, fall_time);

(c => d) = (6, 7);

endspecify

The block starts with the keyword "specify" and ends with the keyword "endspecify". Specify blocks can appear anywhere within a module. The block can have two types of statements:

- One type starts with the keyword **specparam** and assigns numerical values to timing parameters declared elsewhere. The **specparam** statements can appear within a module or within a specify block. The right sides of the assignments can be constants or constant expressions involving such parameters already assigned.
- The second type specifies paths and assigns values to time delays to them.

A specify block can have only the above types of assignments. Circuit function assignments, assignments to module parameters, etc., are not permitted within it.

8.2 Module Paths

Module paths can be specified in different ways inside a specify block. The simplest has the form

A*>B

Here "A" is the source and "B" the destination. The source can be an input or an inout port. The destination can be output or an inout port. The symbol combination "*>" specifies the path from the source to the destination. It encompasses all the possible paths from A to B. If A and B are scalars, it signifies a single path.

- If A is a vector and B is a scalar, it signifies all the paths from every bit of A to the scalar B. Thus if A is a 4-bit-wide vector, 4 paths are specified.
- If A is a scalar and B is a vector, it signifies all the paths from A to every bit of the vector B. Thus if B is an 8-bit vector, it signifies all 8 possible paths.
- If both A and B are vectors, it signifies all the possible paths from eveiy bit of the vector A to every bit of the vector B; thus if A is a 4-bit vector and B is an 8-bit vector, it signifies 4 * 8 = 32 possible paths; a total of 32 delay values (all being equal to each other) are implied here.

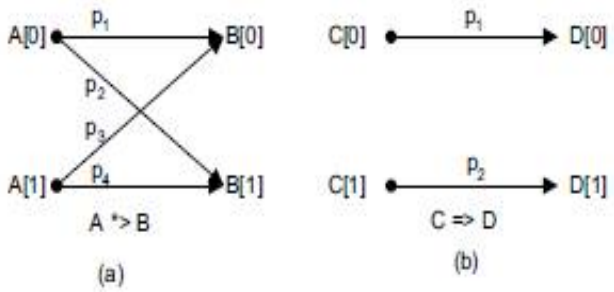


Fig. 174 Illustration of the difference between the operations *> and =>

Fig.174 (a) illustrates a case of all possible paths from a 2-bit vector A to another 2-bit vector B; the specification implies 4 paths. A statement of the type

C => D

signifies only all the parallel paths. Here C and D have to be vectors of the same size. The path specified signifies transmission from every bit of vector C to the corresponding bit of vector D. In this sense the path description is more restrictive than that of A*>B above. Fig.174 (b) illustrates a case of all possible parallel paths from a 2-bit vector C to another 2-bit vector D; the specification implies a total of 2 paths only.

The assignment

(a,b *> S) =1;

implies that

- The propagation delay from input a to output S is 1 ns and
- The propagation delay from input b to the output S is also 1 ns.
- Further the delay value is 1 ns for the change in the state of s from 0 to 1 as well as from 1 to 0.

Similarly the statement

(a,b *> ca) = 2;

implies that the delay from a to ca as well as that from b to ca is 2 ns; further, it holds for any transition in ca.

Example 36: Half Adder Module with path delays

```
module ha_7(s,ca,a,b);
input a,b; output s,ca;
specify
(a,b*>s)=l;
(a,b*>ca)=2;
endspecify xor (s,a,b);
and (ca,a,b);
endmodule
```

```
//test-bench module tstha_7();
reg a,b; wire s,ca;
ha_7 hh(s,ca,a,b);
initial begin a=0;b=0; end
always begin #5 a=l;b=0; #5 a=0;b=l; #5 a=l;b=l; #5 a=0;b=0; end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum = %b "
,a,b,ca,s);
initial #30 Sstop;
endmodule
```

8.3 Conditional Pin-to-Pin Delays

The pin to pin path of a signal may change depending on the value of another signal; in turn the number of circuit elements in the alternate path may differ.

8.4 Edge-Sensitive Paths

Behavior level modules can have signal paths activated following an edge in a different signal. Verilog has the provision to specify such delays during simulation. They can be specified in a variety of ways. The path may get activated following a positive edge or a negative edge in a signal. The path delay may be specified for rise or fall in the output or for positive or negative polarity transitions separately. The delay assignment can be made conditional on an expression; such a path specification is an "edge sensitive state dependent path".

8.5 Pulse Filtering and its Control

All transitions on an input pin with less than a specified module path delay are termed "pulses." Normally, when a module path delay is specified, all pulses are ignored; that is, the simulator does not take cognizance of such narrow transitions. However, response to such narrow pulses can be specified through specparam in a specify block. A statement

specparam PATHPULSE$ (X , y) = **(a,** b);

implies the following concerning the module pulse path from X to ***y:***

- Ignore all pulses of width less than **a** ns. **a** is referred to as the "rejection limit" for the pulse path.
- Take cognizance of all the pulses wider than **b** ns. Note that the specification has relevance only if the delay value for the pulse path (specified in the specify block) is larger than **b.**
- For all pulses of width value between **a** and **b,** the output is in error and in x state.

The PATHPULSE$ specification is governed by the following:

- It has to appear within a specify block as a specparam assignment as shown above.

- It specifies the limits for the path pulse-error limit as well as reject limit for the specified path.

- A statement as

specparam PATHPULSE$ = (a, b);

implies that **a** and **b** are the error and reject limits for the pulse widths for all the paths specified within the specify block; the simulator checks for the pulse width and if it is between **a** and **b** values, the output goes to x state.

- A set of statements

specparam PATHPULSE$ (X, Y) = (a, b);
specparam PATHPULSE$ = (c, d);

implies that for the path from **X** to **Y, a** and **b** are the error and reject limits, respectively; further, for all other pulse paths within the specify block, the limits for error and rejection are c and d, respectively. If only one limit is specified as

specparam PATHPULSE$ =a;

a is taken as the error limit as well as reject limit for the concerned paths.

9. MODULE PARAMETERS

Module parameters are associated with size of bus, register, memory, ALU, and so on. They can be specified within the concerned module but their value can be altered during instantiation. The alterations can be brought about through assignments made with **defparam.** Such **defparam** assignments can appear anywhere in a module.The rules of assigning values for the module parameters, deciding their size, type, *etc.*, are all similar to those of **specify** parameters.

```
module alu_6 (d, co, a, b, f,cci);
parameter msb=3;
output [msb:0] d; output co; wire[msb:0]d; input cci;
input [msb : 0 ] a, b; input [1 : 0] f;
specify  (a,b=>d)=(1,2);   (a,b,cci*>co)=1; endspecify
assign {co,d}=(f==2'b00)?(a+b+cci):((f==2'b01)?(a-b):((f==2'b10)?
            {1'bz,a^b}:{1'bz,~a}));
endmodule
//test-bench
module tst_alu7();
defparam aa.msb=7; parameter nl=7;
reg [nl:0]a,b; reg[1:0] f; reg cci; wire[nl:0]d; wire co;
```

```
alu_6 aa(d,co,a,b,f,cci);
initial begin cci=1'b0; f=2'b00;a=8'h00;b=8'h00; #30 $stop;end
always   begin
#3 cci =1'b0;f=2'b00;a=8'h01;b=8'h00; #3 cci =1'b1;f=2'b00;a=8'h08;b=8'h0f;
#3 cci =1'b1;f=2'b01;a=8'h02;b=8'h01; #3 cci =1'b0;f=2'b01;a=8'h23;b=8'h27;
#3 cci =1'b1;f=2'b01;a=8'h23;b=8'h23; #3 cci =1'b1;f=2'b10;a=8'h23;b=4'h23;
#3 cci =1'b1;f=2'b11;a=8'h2f;b=8'h2c;
          end
initial $monitor($time, " cci = %b , a= %b ,b = %b ,f = %b ,d =%b ,co= %b ",cci
,a,b,f,d,co);
endmodule
```

Fig.175 ALU Module

10. SYSTEM TASKS AND FUNCTIONS

Verilog has a number of System Tasks and Functions defined in the LRM. They are for taking output from simulation, control simulation, debugging design modules, testing modules for specifications, *etc.* A "$" sign preceding a word or a word group signifies a system task or a system function. Some of the system tasks and functions have been extensively used in the earlier chapters. Some others with the potential for common use are described and illustrated here. The complete list is available in the LRM.

10.1 Output Tasks

A number of system tasks are available to output values of variables and selected messages, *etc.*, on the monitor. Out of these **$monitor** and **$display** tasks have been extensively used in the preceding chapters. These and related tasks are discussed below.

- $display → Used to display the arguments in the desired format.
  - $displayb
  - $displayo
  - $displayh
  - $displayd

- $write → Used to display the argument with specified format but does not advance to the new line
  - $writeb
  - $writeo
  - $writeh
  - $writed

- %m → Used to display instantiated modules in another module.

- $strobe → Used to display sampled version of a variable or a set of variables.

10.1 *Display Tasks*

The **$display** task, whenever encountered, displays the arguments in the desired format; and the display advances to a new line. $ **write** task carries out the desired display but does not advance to the new line. For both the format is identical to that of **scanf** and **printf** in C language. The features are briefly outlined here:

- The arguments are displayed in the same order as they appear in the display statement.
- The arguments can be variables, an expression involving variables, or quoted strings.
- The strings are output as such except the escape sequences. An escape sequence starts with the character \ or the character %.
- "\" signifies one of a set of special characters.
- "%m" signifies that the hierarchical name of the particular argument is to be displayed "%" followed by a character specifies the format for display of the following argument or an aspect of the following argument.
- If the format for the display of an argument is not specified, a default format is assumed. It is binary for **$displayb** and **$writeb,** octal for **$displayo** and **$writeo,** decimal for **$displayd** and **$writed,** hex for **$displayh** and **$writeh.**
- If any argument is in the form of an expression, it is evaluated and the result displayed or written; it is sized automatically. With decimal numbers the leading zeros are suppressed. Insertion of a "0" character (zero digit) between the "%" symbol and the radix overrides the automatic sizing.

11. FILE-BASED TASKS AND FUNCTIONS

LRM has the provision to accommodate and integrate design and test modules kept in different files. It makes room for structuring the design in an elegant manner and developing it with a "cross-functional" approach. Different facilities are specified in the LRM. That to output results to a file is discussed here as a specific case. To carry out any file-based task, the file has to be opened, reading, writing, *etc.*, completed and the file closed. The keywords for all file-based tasks start with the letter f to distinguish them from the other tasks.

Writing out to a file

- $fdisplay
- $fwrite
- $fstrobe
- $fmonitor
- $fflush

reading from a file

- $readmemb
- $readmemh
- $fscanf

String formatting functions

- $swrite
- $sformat
- $sscanf

Example 37: Half Adder Test Bench

```
module ha_test;
    reg a,b; wire s,co; integer filen;
    ha hh(s,co,a,b);
    initial
    begin
            a=0;
            b=0;
            filen=$fopen("ha_f.txt");
    end
    always
    begin
            #5      a=0; b=1;
            #5      a=1; b=0;
            #5      a=1; b=1;
            #5      a=0; b=0;
    end
    initial $fmonitor(filen, $time"a=%b, b=%b, s=%b, co=%b", a,b,s,co);
    initial
    begin
            $fdisplay (filen);
            #30 $stop;
            $fclose(filen);
    end
    endmodule
```

12. COMPILER DIRECTIVES

A number of compiler directives are available in Verilog. They allow for macros, inclusion of files, and timescale-related parameters for simulation. All compiler directives are preceded by the ' (accent grave) character. Representative compiler directives are discussed here with illustrations.

12.1 'define Directive

The **"define** directive is for macro substitution. It substitutes the macro by a defined text. Hence a macro name can be used in place of such a group of characters in the listing wherever the group is to appear. Subsequently, the macro name can be substituted during

compilation by the actual text. The **‘define** directive is used to define and associate the desired text with the macro name.

The ‘**define** compiler directive can also be used to substitute a number by a macro name. It allows for deciding bus-width, specific delay values, *etc*., at compilation time.

```
module alu_a (d, co, a, b, f,cci);

`define  add 2'b00
`define subtract 2'b01
`define exor 2'b10

output [3:0] d; utput co; wire[3:0]d;
input cci; input [3 : 0 ] a, b; input [1 : 0] f;
assign {co,d}=(f==`add)?(a+b+cci):((f==`subtract)?(a-b):((f==`exor)?
{1'bz,a^b}:{1'bz,~a}));
endmodule
```

Fig.176 ALU Module

12.2 Time-Related Tasks

A set of compiler directives and system tasks relate to the running time of simulation as well as the delays in the concerned modules. A wide range of timescales as well as precision levels are available for selection during simulation.

‘timescale

The ‘ **timescale** compiler directive allows the time scale to be specified for the design. When a ‘**timescale** directive is encountered in a file, the same is valid for all subsequent modules within the file. The ‘ **timescale** directive has two components.

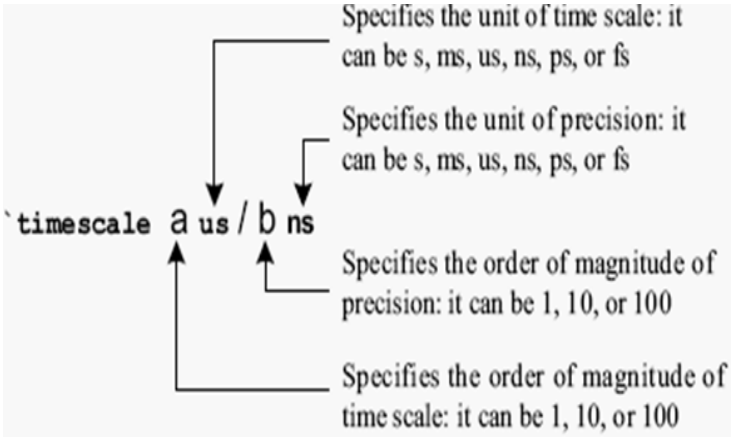


Fig.177 Timescale structure

A few examples are given below:

- **‘timescale 1** ms/100 ps

implies that in the following design all the time values specified are in ms and they have a precision of 100 ps. Thus

3, 3.0, 3.022 are all inteipreted as 3 ms;
3.1, 3.12,3.199 are all inteipreted as 3.1 ms; and
0.1, 0.12 are interpreted as 100 ps .

- **‘timescale** 1 ms/1 ms

implies that in the following design all the time values specified are in ms and they have a precision of 1 ms. Thus

3, 3.0, 3.022, 3.1, 3.12, 3.199 are all interpreted as 3 ms and

0.1, 0.12 are interpreted as 0 ms .

$timeformat

The timescale and the format for display can be changed during simulation with the help of **$timef ormat** task. The syntax for the task is explained in Figure 11.58. Whenever "p,s" (microsecond) is to be specified for defining or changing time scale, it is specified as "us." Conventions for all other timescale values (s, ms, ns, ps, and fs) remain unaltered.

A negative number in the 0 to -15 range signifying time unit: 0 stands for s, -1 for 0.1s and so on; -15 implies fem to second. any convenient s t r i n g t o b e displayed as such

$timeformat (-aa)

An integer specifying precision: it represents the number of digits to the right of the decimal point "cc", dd

An integer specifying the field width for the display

Simulation Time

Simulation time value can be obtained, displayed or used in specific expressions; a limited amount of flexibility is available here: -

- **$time** returns the value of simulation time as an integer.
- **$realtime** returns the value of simulation time as a real number.

Default Timescale

If the time scale values are not specified in the source file, simulation is carried out with the default values specified in the tool used for simulation. The default value of time unit is taken as nanosecond

13. HIERARCHICAL ACCESS

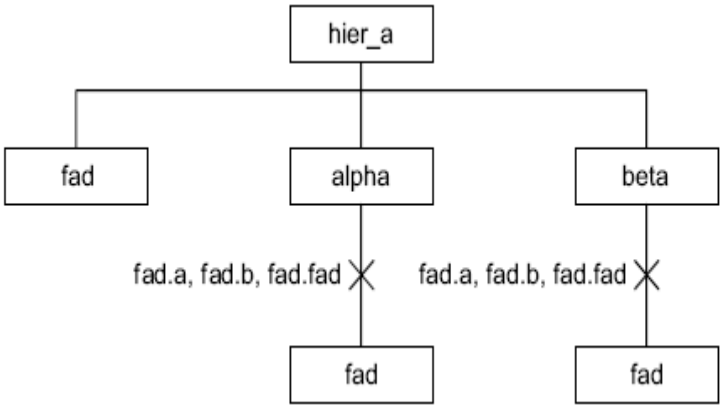


Fig. 178 Hierarchical Access

```
module hier_a;
integer aa, bb, cc, pp, qq, rr;
initial
begin: alpha
        aa = 2; bb = 3;
        cc = fad(aa,bb);
        $display("fad.a = %0d, fad.b = %0d, fad.fad = %0d", fad.a,fad.b,fad.fad);
end
initial
begin: beta
        pp = 4;qq =6;
        rr = fad(pp,qq);
        $display("fad.a = %0d, fad.b = %0d, fad.fad = %0d", fad.a,fad.b,fad.fad);
end
function integer fad;
input [7:0] a, b;
fad = a + b;
endfunction
endmodule

# fad.a = 2, fad.b = 3, fad.fad = 5
# fad.a = 4, fad.b = 6, fad.fad = 10
```

```
module hier_d;
integer aa, bb, cc, pp, qq, rr;
initial
begin
        aa = 2; bb = 3;
        tad(aa,bb,cc);
        $display("tad.a = %0d, tad.b = %0d, tad.c = %0d", tad.a,tad.b,tad.c);
end
initial
begin
        pp = 4;qq =6;
        tad(pp,qq,rr);
        $display("tad.a = %0d, tad.b = %0d, tad.c = %0d", tad.a,tad.b,tad.c);
end
task tad;
input a, b;
output c;
integer a,b,c;
c = a + b;
endtask
endmodule

# tad.a = 2, tad.b = 3, tad.c = 5
# tad.a = 4, tad.b = 6, tad.c = 10
```

Fig.179 Hierarchical Access using functions

15.USER DEFINED PRIMITIVES (UDP)

The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives - called "user defined primitive (UDP)" and use them. A UDP can be defined anywhere in a source text and instantiated in any of the modules. Their definition is in the form of a table in a specific format. It makes the UDP types of functions simple, elegant, and attractive.

UDPs are basically of two types - combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

15.1 Combinational UDPs

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An inout declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module. The definition cannot be within any other module.

```
primitive udp_and (out, in1, in2);
output out;
input in1, in2;
table
//          In1         In2         Out
            0           0:          0;
            0           1:          0;
            1           0:          0;
            1           1:          1;
endtable
endprimitive
```

Fig.180 and gate primitive

- The first statement starts with the keyword "primitive", it is followed by the name assigned to the primitive and the port declarations.
- A UDP can have only one output port. It has to be the first in the port list.
- All the ports following the first are input ports and are all scalars.
- inout ports are not permitted in a UDP definition.
- Output and input are declared in the body of the UDP.
- The behavior block of the primitive is given in the form of a table. It is specified between keywords **table** and **endtable.**
- The combinational function is defined as a set of rows (akin to the truth table).
- All the input values are specified first - each in a separate field in the same order as they appear in the port declaration.
- A colon and then the output value follow the set of input values. The statement ends with a semicolon - as with every statement in Verilog.
- A comment line is inserted in the example following the **"table"** entry. It facilitates understanding the tabular entries.
- All the inputs are nets - **wire**-type. Hence there is no need for a separate type definition.

- Output can be of the net or **reg** type depending upon the type of primitive - explained later.
- The last keyword statement - **"endprimitive"** — signifies the end of the definition.

15.2 Sequential UDPs

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state [Wakerly], A positive or a negative going edge or a simple change in a logic variable can trigger the transition from the present state of the circuit to the next state. A sequential UDP can accommodate all these. The definition still remains tabular as with the combinational UDP. The next state can be specified in terms of the present state, the values of input logic variables and their transitions. The definition differs from that of a combinational UDP in two respects:

- The output has to be defined as a **reg.** If a change in any of the inputs so demands, the output can change.
- Values of all the input variables as well as the present state of the output can affect the next state of the output. In each row the input values are entered in different fields in the same sequence as they are specified in the input port list. It is followed by a colon (:). The value of the present state is entered in the next field which is again followed by a colon (:). The next state value of the output occupies the last field. A semicolon (;) signifies the end of a row definition.

15.3 Sequential UDPs and Tasks

Sequential UDPs and tasks are functionally similar. Tasks are defined inside modules and used inside the module of definition. They are not accessible to other modules. In contrast, sequential UDPs are like other primitives and modules. They can be instantiated in any other module of a design.

15.4 UDP Instantiation with Delays

Outputs of UDPs also can take on values with time delays. The delays can be specified separately for the rising and falling transitions on the output. For example, an instantiation as

udp_and_b #(1,2) g1 (out, inl, in2);

can be used to instantiate the UDF of Figure 9.25 for carry output generation. Here the output transition to 1 (rising edge) takes effect with a time delay of 1 ns. The output transition to 0 (falling edge) takes effect with a time delay of 2 ns. If only one time delay were specified, the same holds good for the rising as well as the falling edges of the output transition.

15.5 Vector-Type Instantiation of UDP

UDP definitions are scalar in nature. They can be used with vectors with proper declarations. For example, the full-adder module fa in Figure 9.26 can be instantiated as an 8-bit vector to form an 8-bit adder. The instantiation statement can be

```
fa [7:0] aa(co, s, a, b, {co[6:0],1'b0});
```

s (sum), co (carry output), a (first input), and b(second input) are all 8-bit vectors here. The vector type of instantiation makes the design description compact; however, it may not be supported by some simulators.

UNIT-V

Sequential Circuit Description
Component test and Verification

Contents

- Sequential models – Feedback model, Capacitive model, Implicit Model
- Basic Memory Components
- Functional Register
- Static Machine Coding
- Sequential Synthesis
- Test bench – Combinational circuit testing, sequential circuit testing
- Test bench techniques
- Design verification
- Assertion verification

UNIT – V

SEQUENTIAL CIRCUIT DESCRIPTION

1. SEQUENTIAL MODELS

In digital circuits, storage of data is done either by feedback, or by gate capacitances that are refreshed frequently.

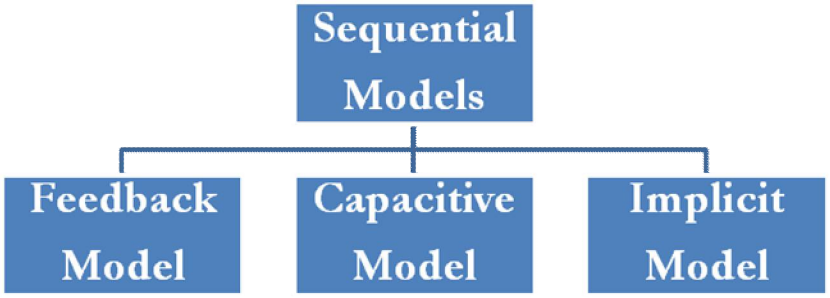


Fig.181 Sequential models

1.1 FEEDBACK MODEL:

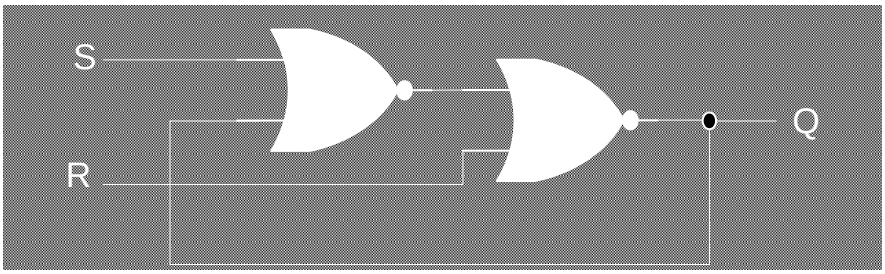


Fig.182 Feedback Model

1.2 CAPACITIVE MODEL

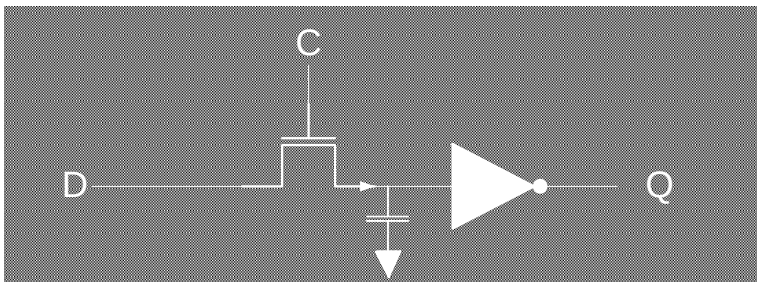


Fig.183Capacitive model

When c becomes 1 the value of D is saved in the input gate of the inverter and when c becomes 0 this value will be saved until the next time that c becomes 1 again.

1.3 IMPLICIT MODEL

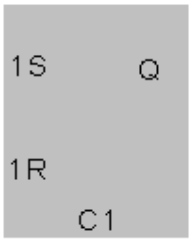


Fig.184 Implicit model

Feedback and capacitive models are technology dependent and have the problem of being too detailed and too slow to simulate. Verilog offers language constructs that are technology independent and allow much more efficient simulation of circuits with a large number of storage elements.

2. BASIC MEMORY COMPONENTS

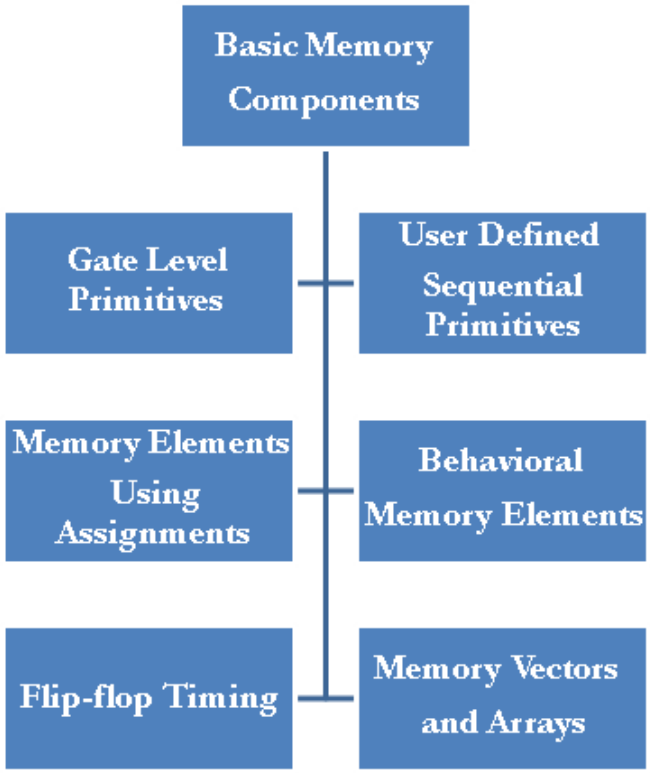


Fig.185 Basic Memory components

2.1GATE LEVEL PRIMITIVES

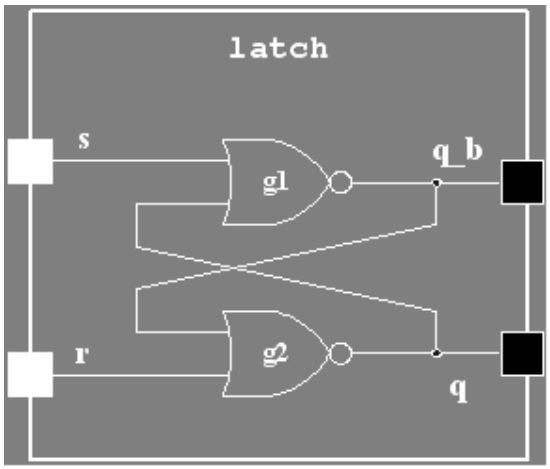


Fig.186 Basic Latch

Example 38: Basic Latch

```
`timescale 1ns/100ps
module latch (input s, r, output q, q_b );
   nor #(4)
            g1 ( q_b, s, q ),
            g2 ( q, r, q_b );
endmodule
```

q and q_b outputs are initially X and remain at this ambiguous state for as long as s and r remain 0. Simultaneous assertion of both inputs results in loss of memory.

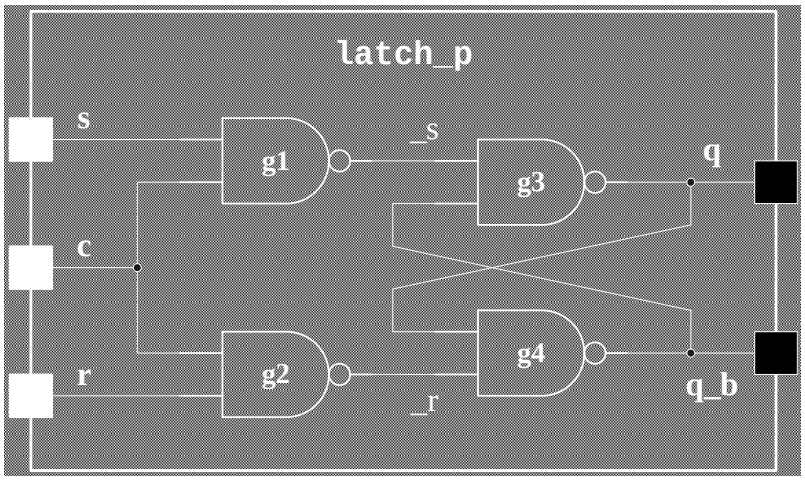


Fig.187 Latch with parameters

Example 39: Latch with Parameters

```
`timescale 1ns/100ps
module latch_p #(parameter tplh=3, tphl=5) (input s, r, c, output q, q_b );
  wire _s, _r;
  nand #(tplh,tphl)
    g1 ( _s, s, c ),
    g2 ( _r, r, c ),
    g3 ( q, _s, q_b ),
    g4 ( q_b, _r, q );
endmodule
```

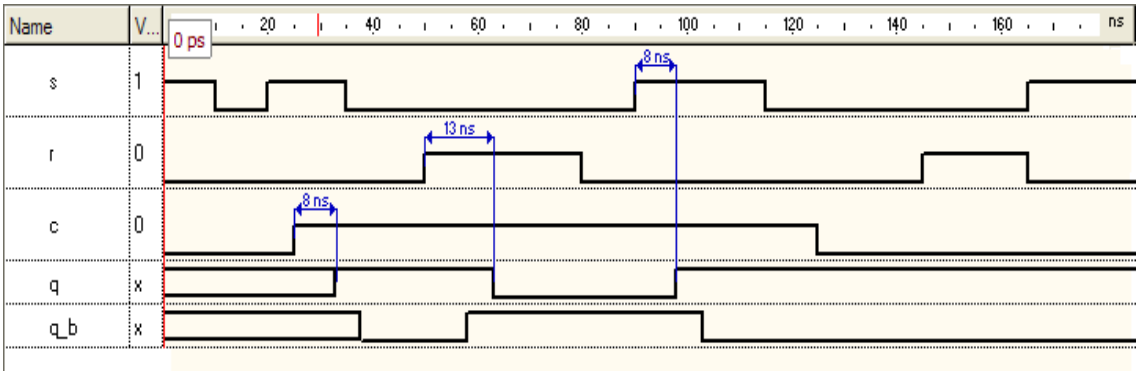


Fig.188 Output waveforms for latch

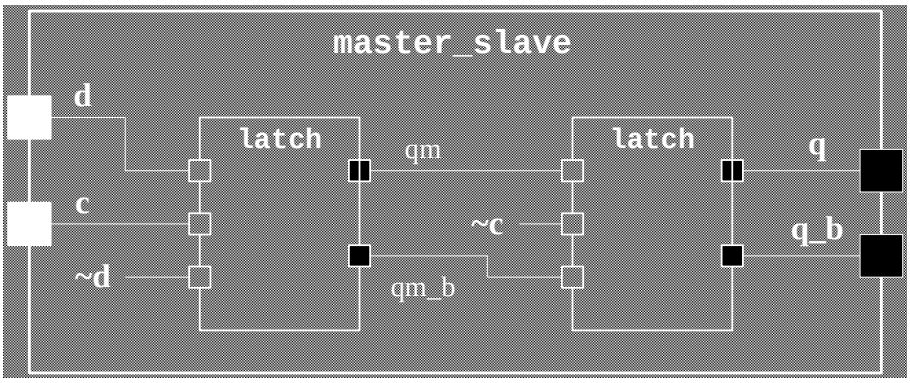


Fig.189 Master slave flip-flop

Example 40: Master slave flip-flop

```
`timescale 1ns/100ps
module master_slave (input d, c, output q, q_b );
  wire qm, qm_b;
  defparam master.tplh=4, master.tphl=4, slave.tplh=4, slave.tphl=4;
  latch_p
        master ( d, ~d, c, qm, qm_b ),
        slave  ( qm, qm_b, ~c, q, q_b );
endmodule
```

2.2 USER DEFINED SEQUENTIAL PRIMITIVES

- Verilog provides language constructs for defining sequential UDPs:
- Faster Simulation of memory elements
- Correspondence to specific component libraries

Example 41: Latch primitive

```
primitive latch( q, s, r, c );
  output q;
  reg q;
  input s, r, c;
  initial q=1'b0;
  table
// s r c  q  q+ ;
// ------:---:----;
    ? ? 0 : ? : - ;
    0 0 1 : ? : - ;
    0 1 1 : ? : 0 ;
    1 0 1 : ? : 1 ;
  endtable
endprimitive
```

```
primitive latch( q, s, r, c );
table
// s r c  q  q+ ;
// ------:---:----;
    ? ? 0 : ? : - ;
    0 0 1 : ? : - ;
    0 1 1 : ? : 0 ;
    1 0 1 : ? : 1 ;
  endtable
endprimitive
```

2.3 MEMORY ELEMENTS USING ASSIGNMENTS

When a block's clock input is 0, it puts its output back to itself (feedback), and when its clock is 1it puts its data input into its output.

Example 42:

```
`timescale 1ns/100ps
module master_slave_p #(parameter delay=3)
                              (input d, c, output q);
  wire qm;
  assign #(delay) qm =  c ? d : qm;
  assign #(delay) q  = ~c ? qm : q;
endmodule
```

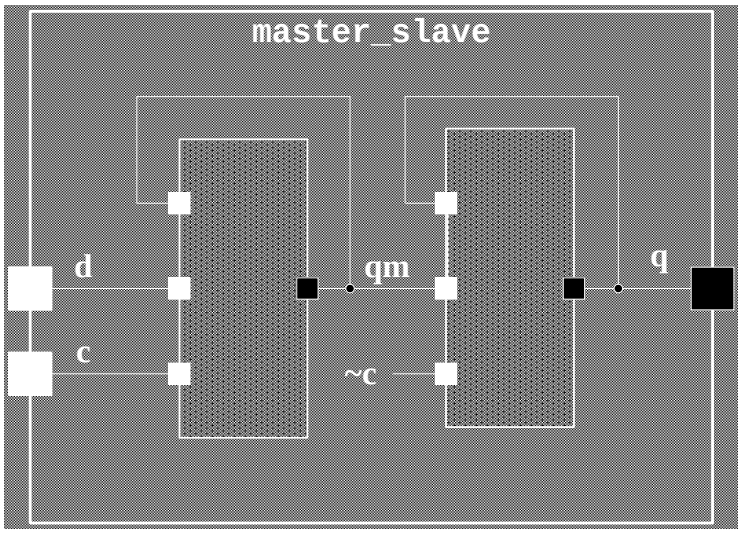


Fig.190 Master Slave Flip-flop

2.4 BEHAVIORAL MEMORY ELEMENTS

Behavioral Coding:

- A more abstract and easier way of writing Verilog code for a latch or flip-flop.
- The storage of data and its sensitivity to its clock and other control inputs will be implied in the way model is written.

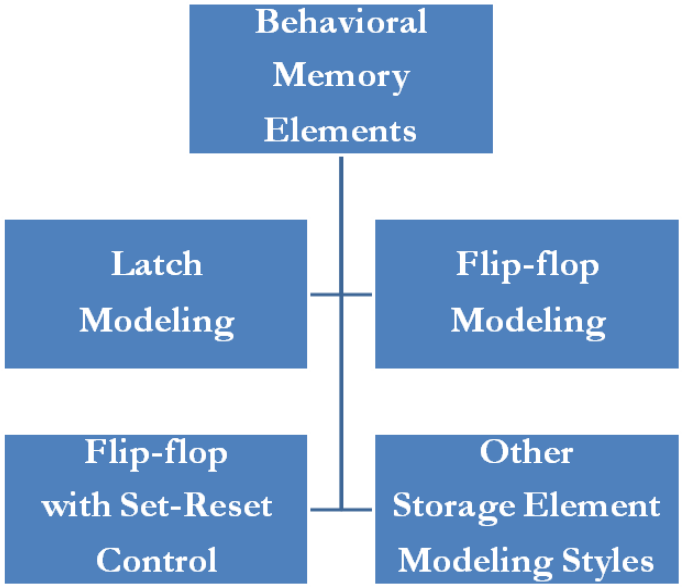


Fig.191 Behavioral Memory Elements

2.4.1 LATCH MODELING

Example 43: A latch

```
`timescale 1ns/100ps
module latch (input d, c, output reg q, q_b );
  always @( c or d )
    if ( c )
    begin
                #4 q = d;
                #3 q_b = ~d;
    end
endmodule
```

```
`timescale 1ns/100ps
module latch (input d, c, output reg q, q_b );
  always @( c or d )
    if ( c )
    begin
      q   <= #4  d;
      q_b <= #3 ~d;
    end
endmodule
```

2.4.2 FLIP-FLOP MODELING

Example 44:

```
`timescale 1ns/100ps
module d_ff (input d, clk, output reg q, q_b );
  always @( posedge clk )
  begin
    q   <= #4  d;
    q_b <= #3 ~d;
  end
endmodule
```

2.4.3 FLIP-FLOP WITH SET-RESET CONTROL

Example 45:

```
`timescale 1ns/100ps
module d_ff_sr_Synch (input d, s, r, clk, output reg q, q_b );
  always @(posedge clk) begin
    if( s ) begin
      q <= #4 1'b1;
      q_b <= #3 1'b0;
    end else if( r ) begin
      q <= #4 1'b0;
      q_b <= #3 1'b1;
```

```
    end else begin
      q <= #4 d;
      q_b <= #3 ~d;
    end
  end
endmodule


module d_ff_sr_Synch (input d, s, r, clk,
                                    output reg q, q_b );
  always @(posedge clk) begin
    if( s ) begin
                  .................
                end else if( r ) begin
     .................
    end else begin
     .................
                end
        end
endmodule


.................
    if( s ) begin
      q <= #4 1'b1;
      q_b <= #3 1'b0;
    end else if( r ) begin
      q <= #4 1'b0;
      q_b <= #3 1'b1;
    end else begin
      q <= #4 d;
      q_b <= #3 ~d;
    end
    .................

`timescale 1ns/100ps
module d_ff_sr_Asynch (input d, s, r, clk, output reg q, q_b );
 always @( posedge clk, posedge s, posedge r )
 begin
    if( s ) begin
      q <= #4 1'b1;
      q_b <= #3 1'b0;
    end else if( r ) begin
      q <= #4 1'b0;
      q_b <= #3 1'b1;
    end else begin
      q <= #4 d;
      q_b <= #3 ~d;
    end
  end
endmodule
```

```
module d_ff_sr_Asynch (input d, s, r, clk,
                                    output reg q, q_b );
  always @( posedge clk, posedge s, posedge r ) begin
     if( s ) begin
                  .....................
     end else if( r ) begin
                  .....................
     end else begin
                  .....................
     end
   end
endmodule
```

```
          .....................
             if( s ) begin
        q   <= #4 1'b1;
        q_b <= #3 1'b0;
     end else if( r ) begin
        q   <= #4 1'b0;
        q_b <= #3 1'b1;
     end else begin
        q   <= #4  d;
        q_b <= #3 ~d;
     end
          .....................
```

2.4.4 OTHER STORAGE ELEMENT MODELING STYLES

Example 46:

```
`timescale 1ns/100ps
module latch (input d, c, output reg q, q_b );
  always begin
     wait ( c );
     #4 q <= d;
     #3 q_b <= ~d;
  end
endmodule
```

2.5 FLIP-FLOP TIMING

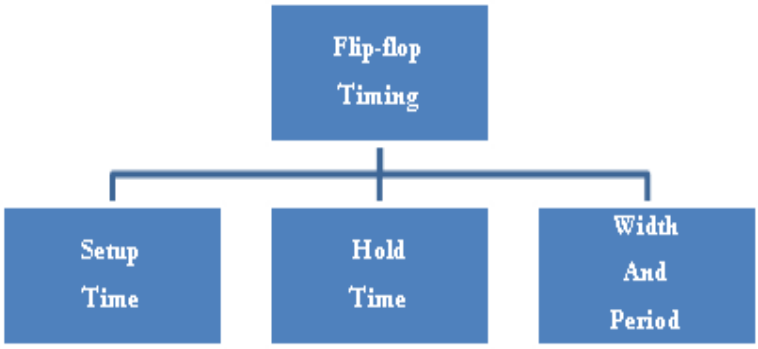


Fig.192 Flip-flop Timing

Setup Time

- The Minimum necessary time that a data input requires to setup before it is clocked into a flip-flop.
- Verilog construct for checking the setup time: $setup task
- The $setup task:
- Takes flip-flop data input, active clock edge and the setup time as its parameters.
  Is used within a specify block.

Ex:
```
specify
$setup (d, posedge clk, 5);
endspecify
```

Hold Time

- The Minimum necessary time a flip-flop data input must stay stable (holds its value) after it is clocked.
- Verilog construct for checking the setup time: $hold task
- The $setup task:
- Takes flip-flop data input, active clock edge and the required hold time as its parameters.
- Is used within a specify block.

Ex:
```
specify
$hold ( posedge clk, d, 3 );
endspecify
```

- The Verilog $setuphold task combines setup and hold timing checks.

Ex: $setuphold (posedge clk, d, 5, 3);

Width And Period

- Verilog $width and $period check for minimum pulse width and period.
- **Pulse Width:** Checks the time from a specified edge of a reference signal to its opposite edge.
- **Period:** Checks the time from a specified edge of a reference signal to the same edge.

EX:
```
specify
        $setuphold ( posedge clk, d, 5, 3 );
        $width  (posedge r, 4);
        $width  (posedge s, 4);
        $period (negedge clk, 43);
endspecify
```

3. FUNCTIONAL REGISTER

3.1 SHIFT REGISTERS

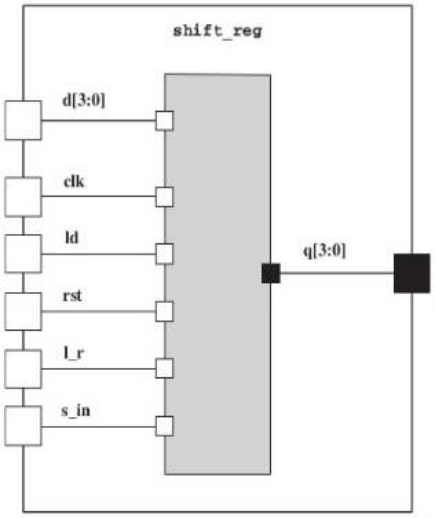


Fig. 193 Basic Shift Register

3.2 COUNTERS

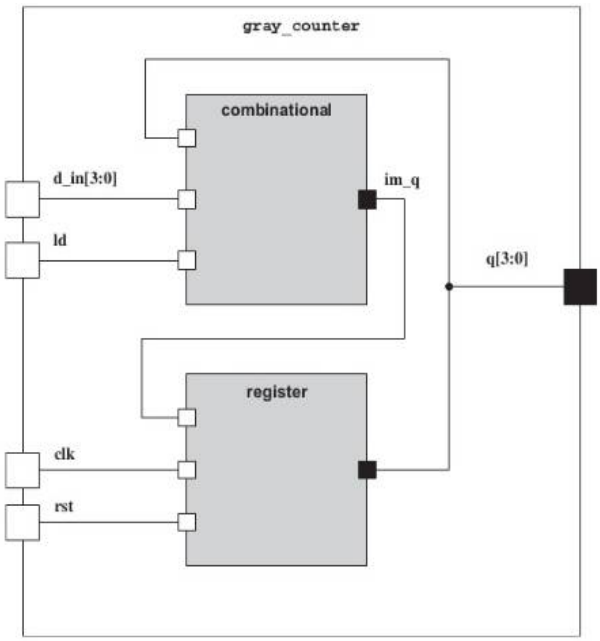


Fig.194 Grey Code Counter

4. STATE MACHINE CODING

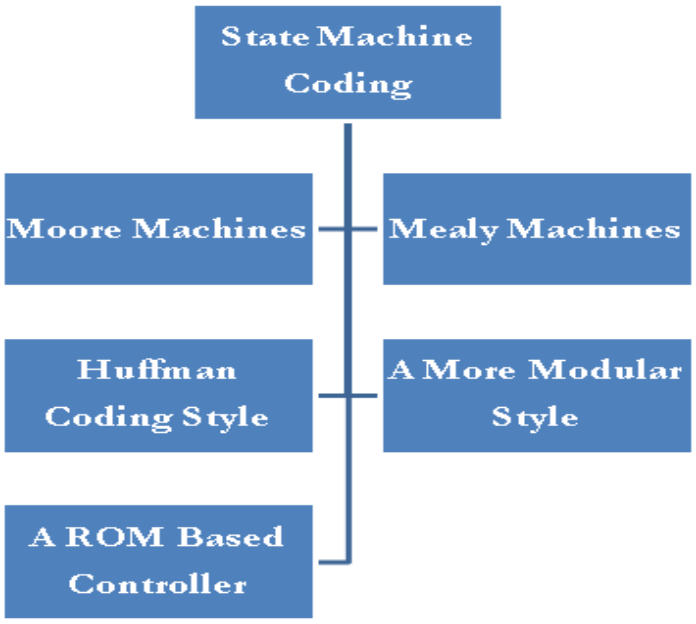


Fig.194 State Machine Coding

Moore Machine :

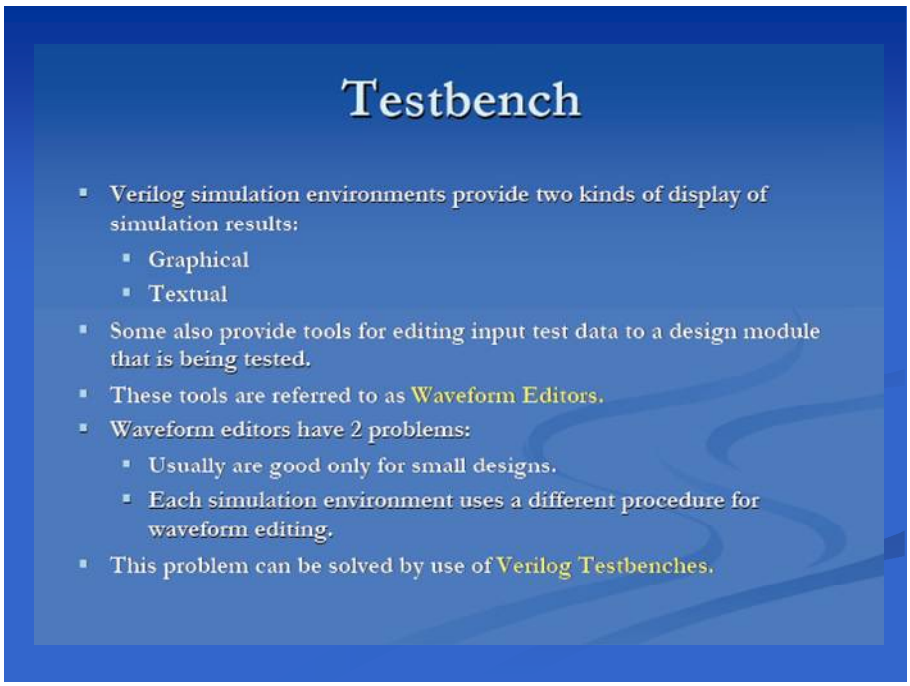
- A state machine in which all outputs are carefully synchronized with the circuit clock.
- In the state diagram form, each state of the machine specifies its outputs independent of circuit inputs.
- In Verilog code of a state machine, only circuit state variables participate in the output expression of the circuit.

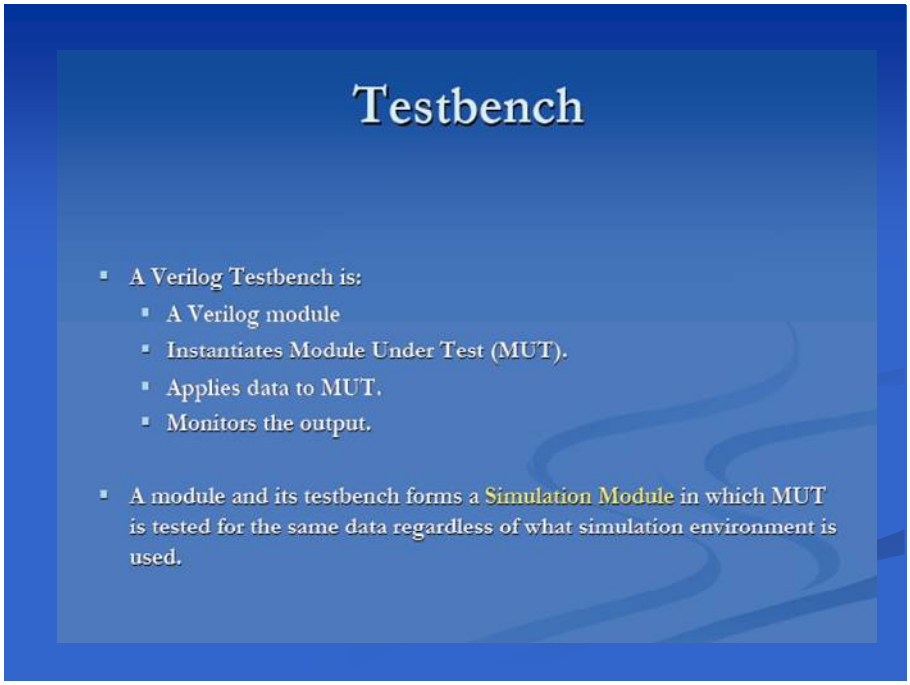
Mealy Machine :

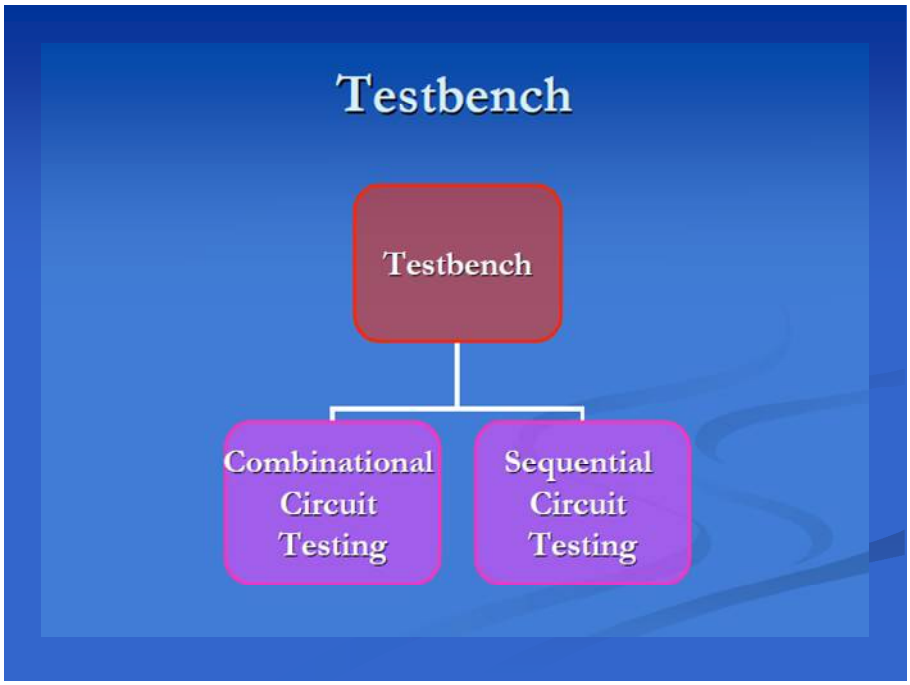
- Is different from a Moore machine in that its output depends on its current state and inputs while in that state.
- State transitions and clocking and resetting the machine are no different from those of a Moore machine. The same coding techniques are used.

5. COMPONENT TEST AND VERIFICATION

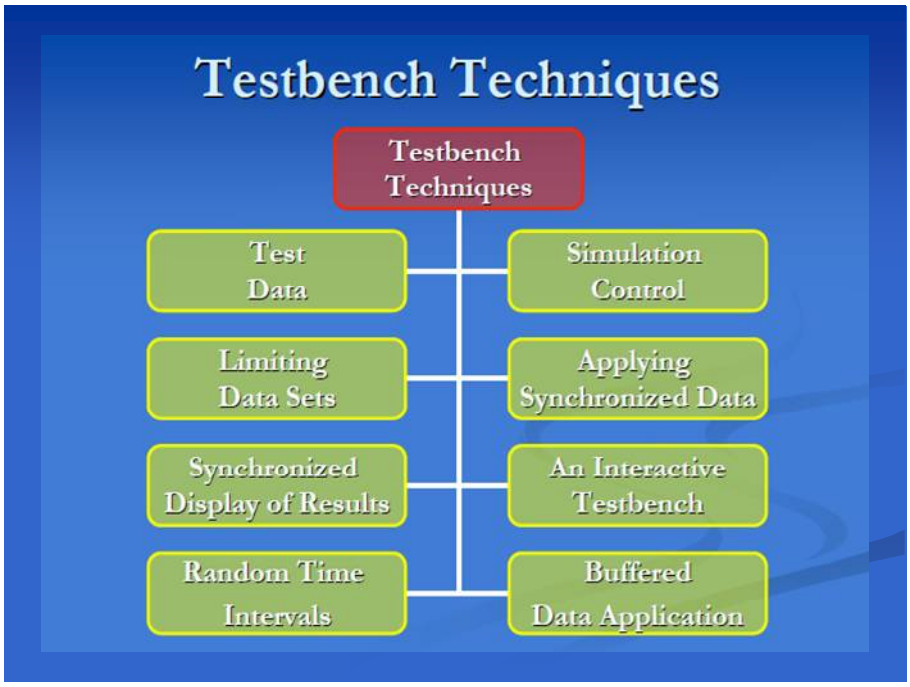
5.1 TEST BENCH





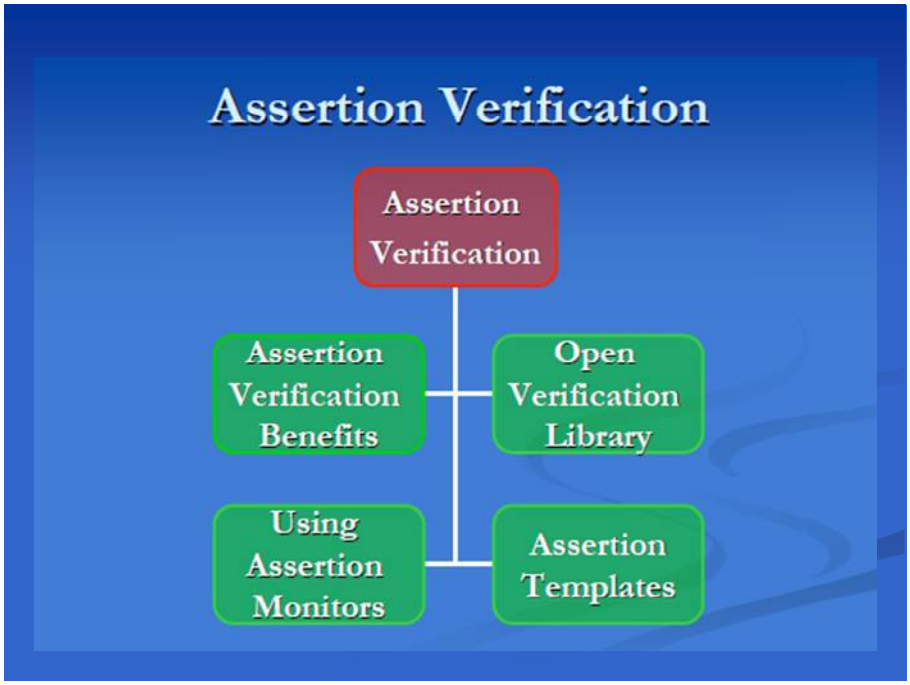


5.2 TEST BENCH TECHNIQUES:



5. DESIGN VERIFICATION:

6. ASSERTION VERIFICATION:

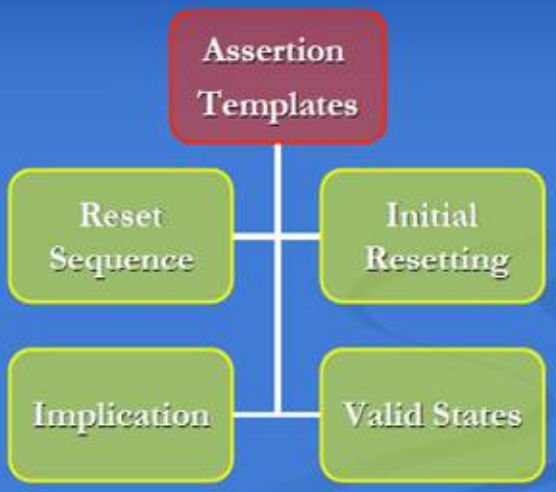


Assertion Verification Benefits

- Ways in which assertion monitors are helpful:
  - Designer Discipline: With placing an assertion in a design, a designer is disciplining him/her-self to look into the design more carefully and extract properties.
  - Observability: Assertions add monitoring points to a design that make it more observable.
  - Formal Verification Ready: Having inserted assertion monitors to a design, readies it for verification by a formal verification tool.
  - Executable Comments: Assertion monitors can be regarded as comments that explain some features or behavior of a design.
  - Self Contained Designs: A design with assertion monitors has the design description and its test procedure all in one Verilog module.

Using Assertion Monitors

Assertion Templates



Text Based Testbenches

- Verilog has an extensive set of tasks for reading and writing external files:
  - Opening and closing files,
  - Positioning a pointer in a file,
  - Writing or appending a file
  - Reading files

SOLVED EXAMPLES

QUESTION BANK

Question Bank

(a) Short Questions with Answers

1. **What is the difference between a function and a task?**
   Answer

Functions	Tasks
Can enable another function but not another task.	Can enable other tasks and functions.
Executes in 0 simulation time.	May execute in non-zero simulation time.
Must not contain any delay, event, or timing control statements.	May contain delay, event, or timing control statements.
Must have at least one input argument. They can have more than one input.	May have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

2. **What is the difference between $display and $monitor?**
   Answer
   The syntax of both statements is same. $monitor continuously monitors the values of the variables or signals specified in the parameter list and executes the statement whenever the value of any one of the variable/parameter changes. Unlike $display, $monitor needs to be invoked only once.

3. **What is the difference between wire and reg?**
   Answer

   Wire is a net data type, represents connections between hardware elements. It's default value is z. Where as reg is a register data type, which represent data storage elements. Registers retain value until another value is placed onto them. It's default value is x.

4. **What is the difference between blocking and non-blocking assignments?**
   Answer
   Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. The " = " operator is used to specify blocking assignments. Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A " <= " operator is used to specify nonblocking assignments.

5. **What is the difference between casex, casez and case statements?**
   Answer
   casez treats all z values in the case expression as don't cares. casex treats all x and z values in the case expression as don't cares.

6. **What is the difference between transport delay and inertial delay?**
   Answer
   Transport delay is the delay caused by the wires connecting the gates. Wire do delay the signal they carry, this is due to the wire resistance, capacitance, and inductance. Simply transport delay is propagation delay on a wire. In verilog transport delay is modeled as follows:
   a <= #10 b; Inertial delay is the time taken by a gate to change its output. It is the gate delay. In verilog inertial delay is modeled as follows: assign #10 a = b;

7. **What is the difference between unary and logical operators?**
   Answer
   Unary operators have only one operand, where as logical operators are of two operands.

8. **What is the difference between (= = , ! =) and (= = = , ! = =)?**
   Answer
   The equality operators (= = , ! =) will yield an x if either operand has x or z in its bits. Where as the case equality operators (= = = , ! = =) compare both operands bit by bit and compare all bits, including x and z.

9. **What are the difference between Verilog and VHDL?**
   Answer
   Verilog is similar to C programming language and VHDL is similar to ADA. Verilog is simple to learn and simple to write code where as VHDL takes longer time to learn and is bit complicated when it comes to write codes.

10. **Tell me how blocking and non blocking statements get executed?**
    Answer
    Execution of blocking assignments can be viewed as a one-step process: 1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement. A blocking assignment "blocks" trailing assignments in the same always block from occurring until after the current assignment has been completed

    Execution of nonblocking assignments can be viewed as a two-step process: 1. Evaluate the RHS of nonblocking statements at the beginning of the time step. 2. Update the LHS of nonblocking statements at the end of the time step.

11. **Variable and signal which will be Updated first?**
    Signals

12. **What is sensitivity list?**
    The sensitivity list indicates that when a change occurs to any one of elements in the list change, begin…end statement inside that always block will get executed.

13. **In a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk? if yes, why?**
    Yes in a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk other wise it will result in pre and post synthesis mismatch.

14. **What is pli?why is it used?**
    Programming Language Interface (PLI) of Verilog HDL is a mechanism to interface Verilog programs with programs written in C language. It also provides mechanism to access internal databases of the simulator from the C program.
    PLI is used for implementing system calls which would have been hard to do otherwise (or impossible) using Verilog syntax. Or, in other words, you can take advantage of both the paradigms - parallel and hardware related features of Verilog and sequential flow of C - using PLI.

15. **There is a triangle and on it there are 3 ants one on each corner and are free to move along sides of triangle what is probability that they will collide?**
    Ants can move only along edges of triangle in either of direction, let's say one is represented by 1 and another by 0, since there are 3 sides eight combinations are possible, when all ants are going in same direction they won't collide that is 111 or 000 so probability of not collision is 2/8=1/4 or collision probability is 6/8=3/4
    Verilog interview Questions

16. **How to write FSM is verilog?**
    there r mainly 4 ways 2 write fsm code
    1) using 1 process where all input decoder, present state, and output decoder r combine in one process.
    2) using 2 process where all comb ckt and sequential ckt separated in different process
    3) using 2 process where input decoder and persent state r combine and output decoder seperated in other process
    4) using 3 process where all three, input decoder, present state and output decoder r separated in 3 process.

17. **What is difference between freeze deposit and force?**
    $deposit(variable, value);
    This system task sets a Verilog register or net to the specified value. variable is the register or net to be changed; value is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim force -deposit command.
    The force command has -freeze, -drive, and -deposit options. When none of these is specified, then -freeze is assumed for unresolved signals and -drive is assumed for resolved signals. This is designed to provide compatibility with force files. But if you prefer -freeze as the default for both resolved and unresolved signals.

18. **What is the difference between the following two lines of Verilog code?**
    #5 a = b;
    a = #5 b;
    #5 a = b; Wait five time units before doing the action for "a = b;".
    a = #5 b; The value of b is calculated and stored in an internal temp register,After five time units, assign this stored value to a.

19. **What is the difference between:**
    c = foo ? a : b and
    if (foo) c = a; else c = b;
    The ? merges answers if the condition is "x", so for instance if foo = 1'bx, a = 'b10, and b = 'b11, you'd get c = 'b1x. On the other hand, if treats Xs or Zs as FALSE, so you'd always get c = b.

20. **What does `timescale 1 ns/ 1 ps signify in a verilog code?**
    'timescale directive is a compiler directive.It is used to measure simulation time or delay time.

Usage : `timescale / reference_time_unit : Specifies the unit of measurement for times and delays. time_precision: specifies the precision to which the delays are rounded off.

21. What is the difference between === and == ?

output of "==" can be 1, 0 or X.
output of "===" can only be 0 or 1.
When you are comparing 2 nos using "==" and if one/both the numbers have one or more bits as "x" then the output would be "X" . But if use "===" outpout would be 0 or 1.
e.g A = 3'b1x0
B = 3'b10x
A == B will give X as output.
A === B will give 0 as output.
"==" is used for comparison of only 1's and 0's .It can't compare Xs. If any bit of the input is X output will be X
"===" is used for comparison of X also.

22. How to generate sine wav using verilog coding style?

A: The easiest and efficient way to generate sine wave is using CORDIC Algorithm.

23. What is the difference between wire and reg?

Net types: (wire,tri)Physical connection between structural elements. Value assigned by a continuous assignment or a gate output. Register type: (reg, integer, time, real, real time) represents abstract data storage element. Assigned values only within an always statement or an initial statement. The main difference between wire and reg is wire cannot hold (store) the value when there no connection between a and b like a->b, if there is no connection in a and b, wire loose value. But reg can hold the value even if there in no connection. Default values:wire is Z,reg is x.

24. what is verilog case (1) ?

```
wire [3:0] x;
always @(...) begin
case (1'b1)
x[0]: SOMETHING1;
x[1]: SOMETHING2;
x[2]: SOMETHING3;
x[3]: SOMETHING4;
endcase
end
```

The case statement walks down the list of cases and executes the first one that matches. So here, if the lowest 1-bit of x is bit 2, then something3 is the statement that will get executed (or selected by the logic).

25. Why is it that "if (2'b01 & 2'b10)..." doesn't run the true case?

This is a popular coding error. You used the bit wise AND operator (&) where you meant to use the logical AND operator (&&).

26. What is difference between Verilog full case and parallel case?

A "full" case statement is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default. If a case statement does not include a case default and if it is possible to find a binary case expression that does not match any of the defined case items, the case statement is not "full."
A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel."

27. Write a verilog code to swap contents of two registers with and without a temporary register?

With temp reg :

```
always @ (posedge clock)
begin
temp=b;
b=a;
a=temp;
end
```

Without temp reg:

```
always @ (posedge clock)
begin
a <= b;
b <= a;
end
```

28. Difference between blocking and non-blocking?

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the = and <= assignment operators. The blocking assignment statement (= operator) acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit. For example, the following Verilog program

```
// testing blocking and non-blocking assignment

module blocking;
reg [0:7] A, B;
initial begin: init1
A = 3;
#1 A = A + 1; // blocking procedural assignment
B = A + 1;

$display("Blocking: A= %b B= %b", A, B ); A = 3;
#1 A <= A + 1; // non-blocking procedural assignment
B <= A + 1;
#1 $display("Non-blocking: A= %b B= %b", A, B );
end
endmodule
```

produces the following output:

```
Blocking: A= 00000100 B= 00000101
Non-blocking: A= 00000100 B= 00000100
```

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current time unit and to assign the registers new values at the end of the current time unit. This reflects how register transfers occur in some hardware systems.

blocking procedural assignment is used for combinational logic and non-blocking procedural assignment for sequential

29. Tell me about verilog file I/O?

OPEN A FILE

integer file;
file = $fopenr("filename");
file = $fopenw("filename");
file = $fopena("filename");

The function $fopenr opens an existing file for reading. $fopenw opens a new file for writing, and $fopena opens a new file for writing where any data will be appended to the end of the file. The file name can be either a quoted string or a reg holding the file name. If the file was successfully opened, it returns an integer containing the file number (1..MAX_FILES) or NULL (0) if there was an error. Note that these functions are not the same as the built-in system function $fopen which opens a file for writing by $fdisplay. The files are opened in C with 'rb', 'wb', and 'ab' which allows reading and writing binary data on the PC. The 'b' is ignored on Unix.

CLOSE A FILE

integer file, r;
r = $fcloser(file);
r = $fclosew(file);

The function $fcloser closes a file for input. $fclosew closes a file for output. It returns EOF if there was an error, otherwise 0. Note that these are not the same as $fclose which closes files for writing.

30. Difference between task and function?

Function:
A function is unable to enable a task however functions can enable other functions.
A function will carry out its required duty in zero simulation time. (The program time will not be incremented during the function routine)
Within a function, no event, delay or timing control statements are permitted
In the invocation of a function their must be at least one argument to be passed.
Functions will only return a single value and can not use either output or inout statements.

Tasks:
Tasks are capable of enabling a function as well as enabling other versions of a Task
Tasks also run with a zero simulation however they can if required be executed in a non zero simulation time.
Tasks are allowed to contain any of these statements.
A task is allowed to use zero or more arguments which are of type output, input or inout.
A Task is unable to return a value but has the facility to pass multiple values via the output and inout statements .

31. Difference between inter statement and intra statement delay?

```
//define register variables
reg a, b, c;

//intra assignment delays
initial
begin
a = 0; c = 0;
b = #5 a + c; //Take value of a and c at the time=0, evaluate
//a + c and then wait 5 time units to assign value
```

```
//to b.
end

//Equivalent method with temporary variables and regular delay control
initial
begin
a = 0; c = 0;
temp_ac = a + c;
#5 b = temp_ac; //Take value of a + c at the current time and
//store it in a temporary variable. Even though a and c
//might change between 0 and 5,
//the value assigned to b at time 5 is unaffected.
end
```

32. Difference between $monitor,$display & $strobe?

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like cwaves?. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes.

The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly where it is executed. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (hierarchy level). %5d, %5b etc. would give exactly 5 spaces for the number instead of the space needed. Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax:

```
$display ("format_string", par_1, par_2, ... );
$strobe ("format_string", par_1, par_2, ... );
$monitor ("format_string", par_1, par_2, ... );
```

33. What is meant by inferring latches,how to avoid it?

Consider the following :

```
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
endcase
```

in a case statement if all the possible combinations are not compared and default is also not specified like in example above a latch will be inferred ,a latch is inferred because to reproduce the previous value when unknown branch is specified.

For example in above case if {s1,s0}=3 , the previous stored value is reproduced for this storing a latch is inferred.

The same may be observed in IF statement in case an ELSE IF is not specified.

To avoid inferring latches make sure that all the cases are mentioned if not default condition is provided.

34. Tell me structure of Verilog code you follow?

A good template for your Verilog file is shown below.

```
// timescale directive tells the simulator the base units and precision of the simulation
`timescale 1 ns / 10 ps
module name (input and outputs);
```

```
// parameter declarations
parameter parameter_name = parameter value;
// Input output declarations
input in1;
input in2; // single bit inputs
output [msb:lsb] out; // a bus output
// internal signal register type declaration - register types (only assigned within always
statements). reg register variable 1;
reg [msb:lsb] register variable 2;
// internal signal. net type declaration - (only assigned outside always statements) wire net
variable 1;
// hierarchy - instantiating another module
reference name instance name (
.pin1 (net1),
.pin2 (net2),
.
.pinn (netn)
);
// synchronous procedures
always @ (posedge clock)
begin
.
end
// combinatinal procedures
always @ (signal1 or signal2 or signal3)
begin
end
assign net variable = combinational logic;
endmodule
```

35. Difference between Verilog and vhdl?

Compilation

VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue.

Verilog. The Verilog language is still rooted in it's native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

Data types

VHDL. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog. Compared to VHDL, Verilog data types a re very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical

wire in the implied modeled circuit. Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of it's simplicity.

Design reusability
VHDL. Procedures and functions may be placed in a package so that they are avail able to any design-unit that wishes to use them.

Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive.

36. Can you tell me some of system tasks and their purpose?

$display, $displayb, $displayh, $displayo, $write, $writeb, $writeh, $writeo.
The most useful of these is $display.This can be used for displaying strings, expression or values of variables.
Here are some examples of usage.

```
$display("Hello oni");
--- output: Hello oni
$display($time) // current simulation time.
--- output: 460
counter = 4'b10;
$display(" The count is %b", counter);
--- output: The count is 0010
```

$reset resets the simulation back to time 0; $stop halts the simulator and puts it in interactive mode where the
user can enter commands; $finish exits the simulator back to the operating system

37. Can you list out some of enhancements in Verilog 2001?

In earlier version of Verilog ,we use 'or' to specify more than one element in sensitivity list . In Verilog 2001, we can use comma as shown in the example below.

```
// Verilog 2k example for usage of comma
always @ (i1,i2,i3,i4)
```

Verilog 2001 allows us to use star in sensitive list instead of listing all the variables in RHS of combo logics . This removes typo mistakes and thus avoids simulation and synthesis mismatches,
Verilog 2001 allows port direction and data type in the port list of modules as shown in the example below

```
module memory (
input r,
input wr,
input [7:0] data_in,
input [3:0] addr,
output [7:0] data_out
);
```

38. Write a Verilog code for synchronous and asynchronous reset?

Synchronous reset, synchronous means clock dependent so reset must not be present in sensitivity disk eg:

```
always @ (posedge clk )

begin if (reset)
. . . end
```

Asynchronous means clock independent so reset must be present in sensitivity list.
Eg
Always @(posedge clock or posedge reset)
begin
if (reset)
. . . end

39. Will case infer priority register if yes how give an example?

yes case can infer priority register depending on coding style

```
reg r;
// Priority encoded mux,
always @ (a or b or c or select2)
begin
r = c;
case (select2)
2'b00: r = a;
2'b01: r = b;
endcase
end
```

40. Casex,z difference,which is preferable,why?

CASEZ :
Special version of the case statement which uses a Z logic value to represent don't-care bits.
CASEX :
Special version of the case statement which uses Z or X logic values to represent don't-care bits.

CASEZ should be used for case statements with wildcard don't cares, otherwise use of CASE is required; CASEX should never be used.
This is because:
Don't cares are not allowed in the "case" statement. Therefore casex or casez are required. Casex will automatically match any x or z with anything in the case statement. Casez will only match z's -- x's require an absolute match.

41. Given the following Verilog code, what value of "a" is displayed?

```
always @(clk) begin
a = 0;
a <= 1;
$display(a);
end
```

This is a tricky one! Verilog scheduling semantics basically imply a
four-level deep queue for the current simulation time:
1: Active Events (blocking statements)
2: Inactive Events (#0 delays, etc)
3: Non-Blocking Assign Updates (non-blocking statements)
4: Monitor Events ($display, $monitor, etc).

Since the "a = 0" is an active event, it is scheduled into the 1st "queue". The "a <= 1" is a non-blocking event, so it's placed into the 3rd queue. Finally, the display statement is placed into the 4th queue. Only events in the active queue are completed this sim cycle, so the "a = 0" happens, and then the display shows a = 0. If we were to look at the value of a in the next sim cycle, it would show 1.

42. How do you implement the bi-directional ports in Verilog HDL?

```
module bidirec (oe, clk, inp, outp, bidir);

// Port Declaration
input oe;
input clk;
input [7:0] inp;
output [7:0] outp;
inout [7:0] bidir;
reg [7:0] a;
reg [7:0] b;
assign bidir = oe ? a : 8'bZ ;
assign outp = b;
// Always Construct
always @ (posedge clk)
begin
b <= bidir;
a <= inp;
end
endmodule
```

43. What are Different types of Verilog Simulators ?

There are mainly two types of simulators available.
Event Driven
Cycle Based
Event-based Simulator:

This Digital Logic Simulation method sacrifices performance for rich functionality: every active signal is calculated for every device it propagates through during a clock cycle. Full Event-based simulators support 4-28 states; simulation of Behavioral HDL, RTL HDL, gate, and transistor representations; full timing calculations for all devices; and the full HDL standard. Event-based simulators are like a Swiss Army knife with many different features but none are particularly fast.

Cycle Based Simulator:

This is a Digital Logic Simulation method that eliminates unnecessary calculations to achieve huge performance gains in verifying Boolean logic:

1.) Results are only examined at the end of every clock cycle; and
2.) The digital logic is the only part of the design simulated (no timing calculations). By limiting the calculations, Cycle based Simulators can provide huge increases in performance over conventional Event-based simulators.
Cycle based simulators are more like a high speed electric carving knife in comparison because they focus on a subset of the biggest problem: logic verification.
Cycle based simulators are almost invariably used along with Static Timing verifier to compensate for the lost timing information coverage.

44. What is Constrained-Random Verification ?

Introduction

As ASIC and system-on-chip (SoC) designs continue to increase in size and complexity, there is an equal or greater increase in the size of the verification effort required to achieve

functional coverage goals. This has created a trend in RTL verification techniques to employ constrained-random verification, which shifts the emphasis from hand-authored tests to utilization of compute resources. With the corresponding emergence of faster, more complex bus standards to handle the massive volume of data traffic there has also been a renewed significance for verification IP to speed the time taken to develop advanced testbench environments that include randomization of bus traffic.

Directed-Test Methodology

Building a directed verification environment with a comprehensive set of directed tests is extremely time-consuming and difficult. Since directed tests only cover conditions that have been anticipated by the verification team, they do a poor job of covering corner cases. This can lead to costly re-spins or, worse still, missed market windows. Traditionally verification IP works in a directed-test environment by acting on specific testbench commands such as read, write or burst to generate transactions for whichever protocol is being tested. This directed traffic is used to verify that an interface behaves as expected in response to valid transactions and error conditions. The drawback is that, in this directed methodology, the task of writing the command code and checking the responses across the full breadth of a protocol is an overwhelming task. The verification team frequently runs out of time before a mandated tape-out date, leading to poorly tested interfaces. However, the bigger issue is that directed tests only test for predicted behavior and it is typically the unforeseen that trips up design teams and leads to extremely costly bugs found in silicon.

Constrained-Random Verification Methodology

The advent of constrained-random verification gives verification engineers an effective method to achieve coverage goals faster and also help find corner-case problems. It shifts the emphasis from writing an enormous number of directed tests to writing a smaller set of constrained-random scenarios that let the compute resources do the work. Coverage goals are achieved not by the sheer weight of manual labor required to hand-write directed tests but by the number of processors that can be utilized to run random seeds. This significantly reduces the time required to achieve the coverage goals.

Scoreboards are used to verify that data has successfully reached its destination, while monitors snoop the interfaces to provide coverage information. New or revised constraints focus verification on the uncovered parts of the design under test. As verification progresses, the simulation tool identifies the best seeds, which are then retained as regression tests to create a set of scenarios, constraints, and seeds that provide high coverage of the design.

(b) Unit Wise Important Questions in DDVHD

UNIT – I

Short:

1. What is Concurrency in Verilog?
2. Explain briefly about PLI.
3. Explain Simulation and Synthesis processes.
4. List out Lexical tokens.
5. What are the advantages of Verilog over VHDL?
6. Explain the role of Verilog as HDL.

Long:

1. What are different types of modules in verilog? Explain in detail?
2. Explain about different levels of design descriptions in detail.
3. Briefly explain about the following
   a) Keywords b) Numbers c) Identifiers
4. Briefly explain about the following
   a) Logic Values b) Operators c) Strings
5. Explain in detail about the following
   a) Comments b) Data types c) Scalars and Vectors
6. Explain about simulation and synthesis tools in detail with proper diagrams.

UNIT – II

Short:

1. List out Unary Operators.
2. Give instantiations of basic and universal logic gates in verilog.
3. What are tri-state gates? Differentiate them with basic logic gates.
4. Define delay. Give different types of delays in gate level modeling.
5. What is meant by concatenation of vectors? Explain its significance in verilog with example.
6. Give different binary operators in verilog.
7. Differentiate logical operators and bit wise logical operators with examples.

Long:

1. Explain all gate primitives with their instantiations and truth tables.
2. a) Explain the design of Flip-flops with gate primitives.
   b) Design edge triggered master slave flip-flop using the universal logic gates.
3. What is strength contention in verilog and how it can be resolved by using wand and wor. Explain with examples.
4. Explain net types with examples.
5. a) Design 2-to-4 line decoder using logic gates.
   b) Design 4-bit ALU using data flow modeling.
6. a) Design 4X1 mux using logic gates and data flow modeling.
7. Explain the different types of operators in detail.

UNIT – III

Short:

1. Differentiate 'initial' and 'always' constructs.
2. What is meant by functional bifurcation?
3. Explain blocking and non-blocking assignments with one example.
4. Differentiate conditional statements and loop statements with examples.
5. Explain about 'disable' construct.
6. Define event in verilog. Give one example.
7. Differentiate sequential blocks and parallel blocks.
8. Explain 'wait' construct with one example.
9. Differentiate force-release and assign-deassign constructs.

Long:

1. Explain the following with examples.
   a) Assignment with delays b) Intra – Assignment delays
   c) Delay Assignments d) Zero delay
2. Explain the design of digital circuit at behavioral level with proper examples.
3. Design a verilog module and its test bench using 'initial' and 'always' constructs.
4. a) Explain about simulation flow in detail.
   b) Explain about multiple always blocks.
5. a) Design a 4X1 mux using case statement.
   b) Design an ALU using if-else construct.
6. Design a priority encoder using case statement. Give its test bench and simulation results.
7. Design a 15X8 memory to store multiples of 5 using
   (i) repeat (ii) While
8. Design a BCD counter with reset and preset using conditional statements. Give its test bench and simulation results.

UNIT – IV

Short:

1. What are applications of CMOS switch over basic transistor switches?
2. What is the significance of resistive switches in verilog? Give the instantiations of resistive switches?
3. Give the systems tasks used for display of output?
4. Define UDPs? What is the significance of UDPs in verilog? Give the instantiation of UDP using one example.
5. Define Compiler directives. Explain its significance in verilog.
6. What is meant by Hierarchical Access in verilog?
7. Define parameters? Give the different parameter used in verilog.
8. Explain the instantiations of switch primitives with strengths and delays.
9. Define strength contention in switch level modeling. Give the method to resolve it.
10. Differentiate pin-to-pin delays and parallel delays.

Long:

1. Design the following modules in verilog.
   (i) CMOS Inverter (ii) CMOS NOR gate
   (iii) CMOS NAND gate (iv) CMOS switch
2. Design an EX-OR gate using universal logic gates. Give its verilog module and test bench in switch level modeling.
3. Explain the following terms in detail with examples.
   a) Bidirectional Switches b) Strength contention with trireg nets
4. a) Explain How specparam can be used in specify block using one example.
   b) What is the significance of pathpulse$ task?
5. a) Design full adder with path delays.
   b) Design a half adder with defparam construct.
6. Explain file based tasks and functions with examples.
7. Explain the following in detail using appropriate examples.
   a) $timeformat b) `define c) `timescale
8. a) Design an UDP for 3X1 mux and give its test bench.
   b) Design an UDP for edge triggered JK flip-flop. Instantiate the UDP in the verilog module and test it through test bench. Give the simulation results.

UNIT – V

Short:

1. List the advantages of test benches over waveform editors?
2. Differentiate sequential circuit testing and combinational circuit testing?
3. Explain about flip-flop timing.
4. Explain about Bidirectional memory?
5. Explain sequential synthesis? What are the different sequential synthesis models in verilog?
6. Explain Design verification in detail?
7. Give the benefits of Assertion verification.
8. What are assertion templates?

Long:

1. Explain about test bench techniques in detail.
2. Explain about Assertion verification in detail.
3. Explain assert_change and assert_one_hot using a verilog module.
4. Explain about state machine coding in detail.
5. Explain about sequential synthesis in detail with appropriate examples.
6. What are functional registers? Design universal shift register and up/down counter.
7. Design LFSR and MISR using verilog.
8. Design FIFO queue using verilog with read and write signals.

INTERNAL PAPERS
(MID & ASSIGNMENT)

Internal Examination Question Papers

Hall Ticket No. Question | | | | | | | | | | | **Paper Code:**

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)

B.TECH V Semester- II mid Examinations APRIL – 2016

(Regulation: CMRCET-R14)

Subject Name: DIGITAL DESIGN THROUGH VERILOG HDL

Date: .08.2016 **Time:** **Max.Marks:25**

PART-A

Answer all TEN questions (Compulsory)

Each question carries ONE mark. **10x1=10M**

1. Define module with one example.
2. What is concurrency?
3. What is the difference between simulation and synthesis?
4. Give an example for number representation in octal and hexadecimal.
5. Write the format for buffer instantiation
6. What are the different net types?
7. What the different signal strengths associated with nets.
8. Differentiate between bufif1 and bufif0.
9. What is behavioral modelling?
10. What is the difference between wire and reg.?

PART-B

Answer any THREE questions. Each question carries FIVE Marks. **3x5=15M**

11. Explain in brief, the lexical tokens of language constructs in Verilog.
12. Give the example for 8 input nand gate instantiation.
13. Design a 4X16 decoder using repeated instantiation of 3X8 decoder.
14. Explain how to resolve the contention using wand and wor type nets with appropriate examples.
15. Give the structure of typical procedural block and explain.

Hall | | | | | | | | | | | **Ticket No.**

Question Paper Code:

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)

B.TECH V Semester- I mid Examinations February 2016

(Regulation: CMRCET-R14)

Subject Name: DIGITAL DESIGN THROUGH VERILOG HDL

Date: .08.2016 **Time:** **Max.Marks:5**

Answer all questions. Each question carries one Marks. 01x5=5M

1. Explain PLI, system tasks and functional verification
2. Explain different data types in verilog
3. Design SR, D and clocked SR flip flop and write the test bench for the same.
4. Design an ALU using Data flow modelling style
5. Design a 4 bit up counter in behavioural modelling and write the test bench for same

Hall Ticket No.

Question Paper Code:

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)

B.TECH V Semester- I mid Examinations February 2016

(Regulation: CMRCET-R14)

Subject Name: DIGITAL DESIGN THROUGH VERILOG HDL

Date: .08.2016 **Time:** **Max.Marks:25**

PART-A

Answer all TEN questions (Compulsory)

Each question carries ONE mark. **10x1=10M**

1. What is continuous assignment.
2. Define tri0 and tri1 nets
3. Design a simple OR gate in Data flow level.
4. What is functional bifurcation?
5. Explain about casex and casez statements.
6. Explain about if and else if construct with example.
7. Write a short notes on bidirectional gates
8. Differentiate between rtranif1 and rtranif0.
9. Explain about feedback model.
10. Write a short notes on combinational circuit testing.

PART-B

Answer any THREE questions. Each question carries FIVE Marks.

03x5=15M

11. Design BCD adder using data flow modeling style and write its test bench
12. Design clocked D flipflop in behavioural modelling style and write its testbench
13. Design 3 input CMOS nand gate using Switch level modeling.
14. Explain path delay with an example and write its test bench.
15. Explain testbench techniques

Hall | | | | | | | | | | | **Ticket No.**

Question Paper Code:

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS)

B.TECH V Semester- I mid Examinations February 2016

(Regulation: CMRCET-R14)

Subject Name: DIGITAL DESIGN THROUGH VERILOG HDL

Date: .08.2016 **Time:** **Max.Marks:5**

Answer all questions. Each question carries one Marks. 01x5=5M

1.(a) Design BCD adder and 4-bit Ring Counter using Data flow modelling .

(b) Differentiate between Inter-assignment and Intra-assignment delays using two different examples.

2.(a) Design priority encoder using **casez** statement. Write short note on Simulation flow.

(b)Design D-flip flop with clear and preset facility using assign and de-assign statement .design an Or-gate using disable construct.

3.(a) Explain forever ,while and force-release construct with examples for each.

(b) Design CMOS NAND ,NOR and NOT gate using Switch level modelling.

4.(a) Design basic Ram cell with neat diagrams. Define bidirectional gates with examples.

(b) Explain module and time related parameters with example for each.

5 (a) Explain about Basic memory concepts.

(b) What are the different Testbench techniques.

PPT's

UNIT - I

Digital Design Using Verilog

Adopted from:
T. R. Padmanabhan and B. Bala Tripura Sundari,
Design through Verilog HDL –Wiley, 2009.

OBJECTIVES AND OUTCOMES

Objective: To make the student learn and understand

- Acquire a basic knowledge of the Verilog HDL
- Language constructs and conventions in Verilog
- Basic Concepts of Verilog HDL like Data Types, System Tasks and Compiler Directives.

Outcomes: The student will be able to,

- Define basic terms in HDL
- Knows Syntax and lexical conventions
- Remembers Data types, operators
- Remember testbenches for simulation and verification

UNIT - I

INTRODUCTION TO VERILOG:

- Verilog as HDL
- Levels of design Description
- Concurrency
- Simulation and Synthesis
- Functional Verification
- System Tasks
- Programming Language Interface (PLI)
- Module
- Simulation and Synthesis Tools
- Test Benches.

LANGUAGE CONSTRUCTS AND CONVENTIONS:

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

VERILOG AS AN HDL

- Verilog aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC.

LEVELS OF DESIGN DESCRIPTION

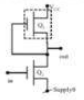
- Gate Level
- Data Flow
- Behavioral Level
- Circuit Level or switch level

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*".

Circuit Level or switch 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.



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

y = (ab+cd)

BEHAVIORAL LEVEL

- Behavioral level constitutes the highest level of design description; it is essentially at the system level itself.
- With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a "C" program.

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.
- All the activities scheduled at one time step are completed and then the simulator.

Verilog Language Concepts

- Concurrency
- Simulation and Synthesis
- Functional Verification
- System Tasks
- Programming Language Interface (PLI)

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 circuits realized from them are essentially direct translations of functions into circuit elements.

FUNCTIONAL VERIFICATION

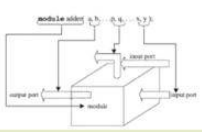
- Testing is an essential ingredient of the VLSI design process as with any hardware circuit.
- It has two dimensions to it – *functional tests* and *timing tests*.
- Testing or functional verification is carried out by setting up a "test bench" for the design.

PROGRAMMING LANGUAGE INTERFACE (PLI)

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.
- 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

MODULE



CONT...

- The ports attached to a module can be of three types:
  - **input** ports through which one gets entry into the module
  - **output** ports through which one exits the module.
  - **inout** ports: These represent ports through which one gets entry into the module or exits the module
- All the constructs in Verilog are centred on the module.

SIMULATION AND SYNTHESIS TOOLS

MODULE SYNTAX

- module module_name (port_list);

  Input, output, inout declaration
  Intermediate variable declarations

  Functional Description
  (gate / switch / data flow / Behv.)

  endmodule

TEST BENCH SYNTAX

- A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli.
- module tb_module_name ;

Input, output, inout declaration
Intermediate variable declarations

Stimulus (initial / always) endmodule

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

- **CASE SENSITIVITY**

  Verilog is a case-sensitive language like C
- **KEYWORDS**
  - The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated
  - All keywords in Verilog are in small letters

WHITE SPACE CHARACTERS , COMMENTS

- **WHITE SPACE CHARACTERS**
- Blanks (\b), tabs (\t), newlines (\n), and form feed form the white space characters in Verilog
- **COMMENTS**
- A single line comment begins with "//"
- multiline comments "/*" signifies the beginning of a comment and "*/" its end.

IDENTIFIERS

NUMBERS, STRINGS

- **NUMBERS**

Integer Numbers : the number is taken as 32 bits wide.

- 25, 253, –253
- - 8 'h f 4

Real Numbers: Real numbers can be specified in decimal or scientific notation

4.3, 4.3e2

- **STRINGS :** A string is a sequence of characters enclosed within double quotes
- "This is a string"

LOGIC VALUES

- 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.**
- **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

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.

Strength Name	Strength Level	Element Modelled	Declaration Abbreviation
Supply Drive	7 Strong Drive	Power supply connections.	supply
6		Default gate & assign output strength.	strong
Pull Drive	5 Large	Gate & assign output strength.	pull
Capacitor	4 Weak	Size of trireg net capacitor.	large
Capacitor	3	Gate & assign output strength.	weak
Medium Capacitor	2	Size of trireg net capacitor.	medium
Small Capacitor	1	Size of trireg net capacitor.	small
High Impedence	0	Not Applicable.	highz

NET DATA TYPE

WIRE / TRI WOR / TRIOR TRI0	WAND / TRIAND TRI1
SUPPLY1 -- For Vdd	TRIREG -- Infers a capacitance
	SUPPLY0 -- For Vss

DIFFERENCES BETWEEN *WIRE* AND *TRI*

- **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.

VARIABLE DATA TYPE

- A variable is an abstraction for a storage device
  - reg
  - time
  - integer
  - real
  - Realtime
- **MEMORY**
  - Reg [15:0] memory[511:0];
    - an array called "memory"; it has 512 locations.
    - Each location is 16 bits wide

CONTENTION

WIRE /TRI	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

WAND /TRIAND	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

WOR /TRIOR	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

TRI1(0)	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	1(0)

SCALARS AND VECTORS

PARAMETERS, OPERATORS.

PARAMETERS

All constants can be declared as parameters at the outset in a Verilog module

- parameter word_size = 16;
- parameter word_size = 16, mem_size = 256;

OPERATORS

- Unary: – for example, ~a.
- Binary: – for example, a&b.
- Ternary: – for example, a?b:c

Unit - II

GATE LEVEL MODELING:

- Introduction
- AND Gate Primitive
- Module Structure
- Other Gate Primitives
- Illustrative Examples
- Tri-State Gates
- Array of Instances of Primitives
- Design of Flip – Flops with gate primitives
- Delays
- Strengths and Contention Resolution,
- Net Types
- Design of Basic Circuits

MODELING AT DATA FLOW LEVEL:

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

Digital Design Through Verilog

Adopted from
T. R. Padmanabhan and B. Bala Tripura Sundari,
Design through Verilog HDL –Wiley, 2009.

UNIT - II

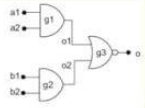
GATE LEVEL MODELING

- All the basic gates are available as "Primitives" in Verilog.

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

VERILOG MODULE FOR AOI LOGIC

```
module aoi_gate(o,a1,a2,b1,b2);
    input a1,a2,b1,b2; output o;
    wire o1,o2;
and g1(o1,a1,a2);
and g2(o2,b1,b2);
nor g3(o,o1,o2); endmodule

module aoi_st;
reg a1,a2,b1,b2; wire o;
initial begin

    a1 = 0;      a2 = 0;   b1 = 0;   b2 = 0;
    #3 a1 = 1;   a2 = 1;   b1 = 1; b2 = 0;
    end
    initial #100
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
```



ARRAY OF INSTANCES OF PRIMITIVES

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

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

TRI-STATE GATES

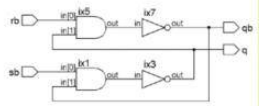
- Four types of tri-state buffers are available in Verilog as primitives

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

DESIGN OF FLIP-FLOPS WITH GATE PRIMITIVES

- ***Simple Latch***

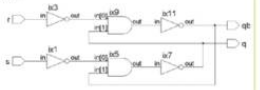
```
module sbrbff(sb,rb,q,qb);
input sb,rb;
output q,qb;

nand(q,sb,qb);
nand(qb,rb,q);

endmodule
```



RS FLIP-FLOP MODULE

- module srff(s,r,q,qb);

input s,r; output q,q
wire ss,rr;



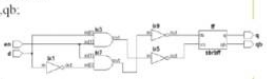
not(ss,s),(rr,r);
nand(q,ss,qb);
nand(qb,rr,q);

endmodule

D-LATCH MODULE

- module dlatch(en,d,q,qb);

input d,en; output q,qb;
wire dd; wire s,r;

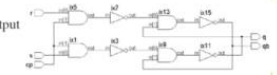


not n1(dd,d);
nand (sb,d,en); nand
g2(rb,dd,en);
sbrbff ff(sb,rb,q,qb);
endmodule

A CLOCKED RS FLIP-FLOP MODULE

- module srffcplev(cp,s,r,q,qb);

input cp,s,r; output
ss,rr;



nand (ss,s,cp),
(rr,r,cp),
(q,ss,qb),
(qb,rr,q);
endmodule

DELAYS

DELAYS WITH TRI-STATE GATES

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

Delay for the 0 to 1 transition of ao

Delay for the 1 to 0 transition of ao

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

STRENGTHS AND CONTENTION RESOLUTION

Name	supply	strong	pull	weak	High impedance
Abbreviations	sul su0	st1 st0	pul pu0	wel we0	HiZ1 HiZ0
Strength	Strongest			Weakest	

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

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

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

MIN, TYPICAL, MAX DELAYS

- and #(2:3:4) g1(a0, a1, a2);
  // min, typical, max delays
- and #(1:2:3, 2:4:6) g2(b0, b1, b2);
- bufif1 #(1:2:3, 2:4:6, 3:6:9) g3 (a0, b0, c0);
- wire #(1:2:3) a;

NET CHARGES

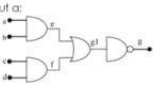
Name	large	medium	small
Strength	Strongest		Weakest

SIGNAL STRENGTH NAMES AND 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

DATA FLOW MODULE FOR AOI

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

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



MODELING AT DATA FLOW LEVEL

- **CONTINUOUS ASSIGNMENT STRUCTURES**

  assign c = a && b;

- **Combining Assignment and Net Declarations**

  wire c;
  assign c = a & b;
  can be combined as
  wire c = a & b;

- **Continuous Assignments and Strengths**

  wire (pull1, strong0)g = ~g1;

DELAYS AND CONCATENATION

OPERATORS

BINARY OPERATORS

- Arithmetic operators and their symbols

Operand type	Symbol	Remarks
Multiplication	*	
Division	/	The result is x if the denominator is zero
Modulus	%	
Addition	+	
Subtraction	-	

- Binary logical operators and their symbols

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

- Relational operators and their symbols

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

UNARY OPERATORS

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

CONT.

- Equality operators and their symbols

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

CONT…

- Bit-wise logical operators and their symbols

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

- Shift type operators and their symbols

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

OPERATOR PRIORITY

- The table brings out the order of precedence. The order of precedence decides the priority for sequence of execution and circuit realization in any assignment

Unary operators	! & ~& | ~| ^ ~^ + -	Highest precedence
Binary operator	* / %	
	+ -	
	<< >>	
	< <= > >=	
	== != === !==	
	&	
	^ ~^	
	|	
	&&	
	||	
Ternary operators	? :	Lowest precedence

TERNARY OPERATOR

- A ? B : C
- assign y = w ? x : z;
- Assign d = (f == add) ? (a+b) : ((f = sub) ? (a-b) : ((f==compl) ? ~a : ~b;

DIGITAL DESIGN Using Verilog

UNIT - III

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

OPERATIONS AND ASSIGNMENTS

- The design description at the behavioral level is done through a sequence of assignments.
- These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level.
- All the procedural assignments are executed sequentially in the same order as they appear in the design description.

BEHAVIORAL MODELING

BEHAVIORAL MODELING

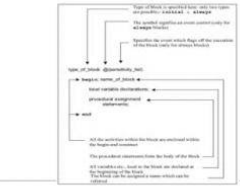
- Behavioral level modeling constitutes design description at an abstract level.
- One can visualize the circuit in terms of its key modular functions and their behavior; it can be described at a functional level itself instead of getting bogged down with implementation details.

FUNCTIONAL BIFURCATION

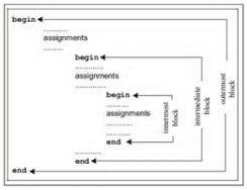
- Design description at the behavioral level is done in terms of procedures of two types;
- one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an "always".
- The second concerns simulation – its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process; all these can be carried out under the "always" banner, an "initial" banner, or their combinations.

procedure-block structure

- A procedure-block of either type – initial or always
  - can have a structure shown in Figure



NESTED BEGIN – END BLOCKS



BEGIN – END CONSTRUCT

- If a procedural block has only one assignment to be carried out, it can be specified
- as initial #2 a=0;
- More than one procedural assignment is to be carried out in an initial block. All such
- assignments are grouped together between "begin" and "end" declarations.
- Every begin declaration must have its associated end declaration.
- begin – end constructs can be nested as many times as desired.

INITIAL CONSTRUCT

- A set of procedural assignments within an initial construct are executed only Once
- In any assignment statement the left-hand side has to be a storage type of element (and not a net). It can be a reg, integer, or real type of variable. The right-hand side can be a storage type of variable or a net.

```
initial
  begin
        a = 1'b0; b
        = 1'b0;
        #2 a = 1'b1;
        #3 b = 1'b1;
        #100$stop;
  end
```


MULTIPLE INITIAL BLOCKS

```
module nil1;
        initial
        reg a, b; begin
        a = 1'b0;
                                b = 1'b0;
$display ($time, "display :  a = %b, b = %b", a, b);
#2 a = 1'b1;
end
initial #100$stop; initial
begin                           #2 b = 1'b1; end


endmodule
```

EVENT CONTROL

- The always block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol "@".
- @(negedge clk) :executes the following block at the negative edge of clk.
- @(posedge clk) : executes the following block at the positive edge of the clk.
- @clk : executes the following block at both the edges of clk.
- @(prt or clr) :
- @(posedge clk1 or negedge clk2) :
- @ (a or b or c) can also write as @ (a or b or c) @ (a, b, c) @ (a, b or c)

ALWAYS CONSTRUCT

- The always process signifies activities to be executed on an "always basis."
- Its essential characteristics are:

·Any behavioral level design description is done using an always block.

·The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.

·The process can have one assignment statement or multiple assignment statements.

·Normally the statements are executed sequentially in the order they appear.

EXAMPLE *COUNTER*

- module counterup(a,clk,N); input clk;

input[3:0]N;

output[3:0]a;

reg[3:0]a;

initial a=4'b0000;

always@(negedge clk) a=(a==N)?4'b0000:a+1'b1;

- endmodule

ASSIGNMENTS WITH DELAYS

- always #3 b = a;
  - Values of a at the 3rd, 6th, 9th, *etc.*, ns are sampled and assigned to b.
- Initial
- begin
  - a = 1'b1;
  - b = 1'b0;
  - #1 a = 1'b0;
  - #3 a = 1'b1;
  - #1 a = 1'b0;
  - #2 a = 1'b1;
  - #3 a = 1'b0;
- end

ZERO DELAY

- A delay of 0 ns does not really cause any delay.
- However, it ensures that the assignment following is executed last in the concerned time slot.
- always
  - begin a = 1;
  - #0 a = 0;
  - end

INTRA-ASSIGNMENT DELAYS

- The "intra-assignment" delay carries out the assignment in two parts.
- An assignment with an intra-assignment has the form
  - A = # dl expression;
- Here the expression is scheduled to be evaluated as soon as it is encountered.
- However, the result of the evaluation is assigned to the right-hand side quantity a after a delay specified by dl.
- dl can be an integer or a constant expression
  - always #2 a = a + 1;
  - always #b a = a + 1;

WAIT CONSTRUCT

- The wait construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments.
- Its syntax has the form
  - wait (alpha) assignment1;
- alpha can be a variable, the value on a net, or an expression involving them.
- @clk a = b; assigns the value of b to a when clk changes;
- wait (clk) #2 a = b; the simulator waits for the clock to be high and then assigns b to a

BLOCKING AND NONBLOCKING

- All assignment within an initial or an always block done through an equality ("=") operator. These are executed sequentially. Such assignments block the execution of the following lot of assignments at any time step. Hence they are called "blocking assignments".
- If the assignments are to be effected concurrently A facility [illegible] "nonblocking assignment" is available. [illegible] assignment. The main characteristic of a nonblocking assignment is that its execution is concurrent

NONBLOCKING ASSIGNMENTS AND DELAYS

- The principle of Delays of the intra-assignment type operation is similar to that with blocking assignments.
- always @(a or b)
  - #3 c1 = a&b;
- which has a delay of 3 ns for the blocking assignment to c1. If a or b changes, the always block is activated. Three ns later, (a&b) is evaluated and assigned to c1. The event "(a or b)" will be checked for change or trigger again. If a or b changes, all the activities are frozen for 3 ns. If a or b changes in the interim period, the block is not activated. Hence the module does not depict the desired output.
- always @(a or b)
- c2 = #3 a&b;
- The always block is activated if a or b changes. (a & b) is evaluated immediately but assigned to c2 only after 3 ns. Only after the delayed assignment to c2, the event (a or b) checked for change. If a or b changes in the interim period, the block is not activated.

CONT...

- For all the non-blocking assignments in a block, the right-hand sides are evaluated first. Subsequently the specified assignments are scheduled.
- What will happen if the following statements are executed
- A <= B; // A, B will swapped
- B <= A ;
- And
- A = B;
- B = A; // A, B will have same value

- always @(a or b)
- #3 c3 <= a&b;
- The block is entered if the value of a or b changes but the evaluation of a&b and the assignment to c3 take place with a time delay of 3ns. If a or b changes in the interim period, the block is not activated.
- always @(a or b)
- c4 <= #3 a&b;
  - represents the best alternative with time delay. The always block is activated if a or b changes. (a&b) is evaluated immediately and scheduled for assignment to c4 with a delay of 3 ns. Without waiting for the assignment to take effect (*i.e.*, at the same time step as the entry to the block), control is returned to the event control operator. Further changes to a or b – if any – are again taken cognizance of.

THE *CASE* STATEMENT

- simple construct for multiple branching in a module. The keywords case, endcase, and default are associated with the case construct.
- Format of the case construct is

```
Case (expression)
Ref1 : statement1; Ref2 :
statement2; Ref3 :
statement3;
...
...
default: statementd;
endcase
```

CASEX AND CASEZ

- The case statement executes a multiway branching where every bit of the case expression contributes to the branching decision. The statement has two variants where some of the bits of the case expression can be selectively treated as don't cares – that is, ignored.
- Casez allows z to be treated as a don't care. "?" character also can be used in place of z.
- casex treats x or z as a don't care.

EXAMPLE

```
module dec2_4beh(o,i);
    output[3:0]o;
    input[1:0]i;
    reg[3:0]o;
    always@(i)
    begin
    case(i)
                2'b00:o=4'h0;
                2'b01:o=4'h1;
                2'b10:o=4'h2;
                2'b11:o=4'h4;
    default:    begin       $display ("error");
                            o=4'h0;
```

SIMULATION FLOW

- In Verilog the parallel processing is structured through the following [IEEE]:
  - Simulation time: Simulation is carried out in simulation time.
  - At every simulation step a number of active events are sequentially carried out.
  - The simulator maintains an event queue – called the "Stratified Event Queue" – with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.
  - The active event can be of an update type or evaluation type. The evaluation event can be for evaluation of variables, values on nets, expressions, *etc.* Refreshing the queue and rearranging it constitutes the update event.
  - Any updating can call for a subsequent evaluation and *vice versa*.
  - Only after all the active events in a time step are executed, the simulation advances to the next time step.
  - Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL.

STRATIFIED EVENT QUEUE

- The events being carried out at any instant give rise to other events – inherent in
- the execution process. All such events can be grouped into the following 5 types:
  - Active events –
  - Inactive events – The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.
  - Blocking Assignment Events – Operations and processes carried out at previous time steps with results to be updated at the current time step are of this category.
  - Monitor Events – The Monitor events at the current time step – Smonitor and Strobe – are to be processed after the processing of the active events, inactive events, and nonblocking assignment events.
  - Future events – Events scheduled to occur at some future simulation time are the future events.

IF AND IF-ELSE CONSTRUCTS

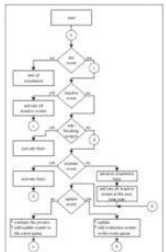
- The if construct checks a specific condition and decides execution based on the result.

```
assignment1;
if (condition) assignment2;
assignment3;
```

- Use of the if–else construct.

```
assignment1; if(condition)
begin // Alternative 1
assignment2;
end
else
begin //alternative 2
assignment3;
end assignment4;
```

FLOWCHART FOR THE SIMULATION FLOW.



EXAMPLE

```
module demux(a,b,s);
    output [3:0]a;
    input b, [1:0]s;
    reg[3:0]a;
    always@(b or s)
            begin  if(s==2'b00)
                        begin  a[2'b0]=b;
    a[3:1]=3'bZZZ;      end
                   else if(s==2'b01)
                        begin  a[2'd1]=b;
    {a[3],a[2],a[0]}=3'bZZZ;    end
                   else if(s==2'b10)
                        begin  a[2'd2]=b;
```

ASSIGN–DEASSIGN CONSTRUCT

- The assign – deassign constructs allow continuous assignments within a behavioral block.
- always@(posedge clk) a = b;
- At the positive edge of clk the value of b is assigned to a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored.
- As an alternative, consider the block
- always@(posedge clk) assign c = d;
- Here at the positive edge of clk, c is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as changes in variable c.

REPEAT CONSTRUCT

- The repeat construct is used to repeat a specified block a specified number of times:
- …
- repeat (a)
- begin
- assignment1;
- assignment2;
- …
- end
- …
- The quantity a can be a number or an expression evaluated to a number.
- The following block is executed "a" times. If "a" evaluates to 0 or x or z, the block is not executed.

- Always
- Begin
- @(posedge clk) assign c = d;
- @(negedge clk) deassign c;
- end
- The above block signifies two activities:
- 1. At the positive edge of clk, c is assigned the value of d in a continuous manner.
- 2. At the following negative edge of clk, the continuous assignment to c is removed; subsequent changes to d are not passed on to c; it is as though c is electrically disconnected from d.

FOR LOOP

- The for loop in Verilog is quite similar to the for loop in C
- It has four parts; the sequence of execution is as follows:
  - 1. Execute assignment1.
  - 2. Evaluate *expression*.
  - 3. If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
  - 4. If *expression* evaluates to the false state (0), exit the loop.
  - 5. Execute assignment2. Go to step 2
- . . . .
- for(assignment1; *expression*; assignment 2)
- statement;
- . . .

THE DISABLE CONSTRUCT

- To break out of a block or loop. The disable statement terminates a named block or task. Control is transferred to the statement immediately following the block
- The disable construct is functionally similar to the *break* in C

```
always@(posedge en)
begin:OR_gate
b=1'b0;
for(i=0;i<=3;i=i+1) if(a[i]==1'b1)

                begin    b=1'b1;
                         disable OR_gate;
                end
        end
```

FOREVER LOOP

- Repeated execution of a block in an endless manner is best done with the forever loop (compare with repeat where the repetition is for a fixed number of times).

```
always @(posedge en)
forever#2 clk=~clk;
```

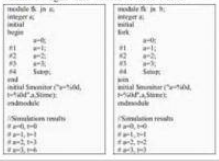
WHILE LOOP

- The Boolean *expression* is evaluated. If it is true, the statement s are executed and expression evaluated and checked. If the *expression* evaluates to false, the loop terminated and the following statement is taken for execution
- while(|a) begin

```
        b=1'b1;
        @(posedge clk)
        a=a-1'b1;
end
b=1'b0;
```

PARALLEL BLOCKS

- All the procedural assignments within a begin–end block are executed sequentially. The fork–join block is an alternate one where all the assignments are carried out concurrently (The non-blocking assignments too can be used for the purpose.). One can use a fork-join block within a begin–end block or vice versa.



CONTENTS BEYOND THE SYLLABUS

TOPICS :

1. Open Verification Library
2. Assertion Monitors
3. Assertion Templates

INTRODUCTION :

Verification with assertions refers to the use of an assertion language to specify expected behaviour in a design, and of tools that evaluate the assertions relative to the design under verification.

Assertion-based verification is mostly useful to design and verification engineers who are responsible for the RTL design of digital blocks and systems. ABV lets design engineers capture verification information during design. It also enables internal state, data path, and error precondition coverage analysis.

Simple example of assertion could be a FIFO: whenever a FIFO is full and a write happens, it is illegal. So a FIFO designer can write an assertion which checks for this condition and asserts failure.

ASSERTION LANGUAGES:

Currently there are multiple ways available for writing assertions as shown below.

- Open Verification Library (OVL).
- Formal Property Language Sugar
- System Verilog Assertions

Most assertions can be written in HDL, but HDL assertions can be lengthy and complicated. This defeats the purpose of assertions, which is to ensure the correctness of the design. Lengthy, complex HDL assertions can be hard to create and subject to bugs themselves.

Advantages of using assertions:

- Testing internal points of the design, thus increasing observability of the design.
- Simplifying the diagnosis and detection of bugs by constraining the occurrence of a bug to the assertion monitor being checked.
- Allowing designers to use the same assertions for both simulation and formal verification.

Assertions in Digital Logic Design - RTL (Verilog, SystemVerilog etc.)

An assertion is a logical state defined to monitor the occurrence of certain events in the logic design during behavioural simulations. Defining logical states for assertions are implemented as properties (or rules). Each property can be visualized as a Boolean Expression.

As long as an assertion holds true no messages are populated in the logic simulation's. Whenever assertions fails, simulator produces 'Error messages'.

Types of Assertions:-

1) Immediate assertion.
2) Concurrent assertions.

Immediate Assertions -

Immediate assertions are executed only once and are mostly implemented within 'initial blocks'. Due to limited use cases its not widely used and limited to simulations.

Example of immediate assertion below:-

```
assert (A==B) $display("Pass");
  else $error("Fail, reporting Error");
```

Failure of assertion is reported by else statement. In 'else' branch we can also include severity of the failure. The level of severity varies from $info, $warning, $error or $fatal. The $error is the default severity in SystemVerilog.

Concurrent Assertions –

The concurrent assertions are tied closely to the RTL design to behave inline with the implementation logic. These assertions are most valuable and widely used. Its useful for both formal verification and behavioral simulations.

There are two types of concurrent assertions :-

Assertions checking the property only with rising edges of the clock. Assertions which are always active in time and properties are constantly validated.

Assertions in Digital Logic Design - RTL (Verilog, SystemVerilog etc.)

Example of concurrent assertions:-

```
assert property (@posedge clk) (fifo_full && fifo_wr);
```

Implementing assertion monitors:

Assertion monitors address design verification concerns and can be used as follows to increase design confidence.

Combine assertion monitors to increase the coverage of the design (for example, in interface circuits and corner cases).

Include assertion monitors when a module has an external interface. In this case, assumptions on the correct input and output behaviour should be guarded and verified.

Include assertion monitors when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores), or may not completely understand the module. In these cases, guarding the module with assertion monitors may prevent incorrect use of the module.

Normally assertions are implemented by the designers to safeguard their design, so they code the assertions into their RTL. A simple example of an assertion would be: writing into FIFO, when it is full. Traditionally verification engineers have been using assertions in their verification environments without knowing that they are assertions. For verification a simple application of assertions would be checking protocols. Example: expecting the grant of an arbiter to be asserted after one clock cycle and before two cycles after the assertion of request.

For using Open Verification Library examples you need Open Verification Library from Accellera. For running PSL examples you need a simulator that can support PSL

Assertion with OVL :

We need to include the assertion file that we need to use. If in our example we are using assert_fifo_index.vlib, we use synopsys translate_off to prevent the synthesis tools from reading the code within **synopsys translate_off** and **synopsys translate_on**. We want to do this, as this is simulation code not meant for synthesis. Next we need to enable assertions by `define OVL_ASSERT_ON. There are many other defines that we can use to control the OVL assertion; details of each option can be found in the OVL manual.

Assertion in RTL

- **assert_fifo_index :** Prints error whenever there is overflow or underflow error.
- **assert_always :** Prints error whenever a write happens with the fifo full flag set
- **assert_never :** Prints error whenever a read happens with the fifo empty flag set
- **assert_increment :** Prints error whenever the write pointer increments by a value > 1

OVL Assertion List

- **assert_always**

The assert_always assertion checker checks the single-bit expression test_expr at each rising edge of clk to verify whether it evaluates to TRUE.

- **assert_always_on_edge**

The assert_always_on_edge assertion checker checks the single-bit expression sampling_event for a particular type of transition.

- **assert_change**

The assert_change assertion checker checks the expression start_event at each rising edge of clk to determine if it should check for a change in the value of test_expr. If start_event is sampled TRUE, the checker evaluates test_expr and re-evaluates test_expr at each of the subsequent num_cks rising edges of clk. If the value of test_expr has not been sampled changed from its start value by the last of the num_cks cycles, the assertion fails.

- **assert_cycle_sequence**

The assert_cycle_sequence assertion checker checks the expression event_sequence at the rising edges of clk to identify whether or not the bits in event_sequence assert sequentially on successive rising edges of clk.

- **assert_decrement**

The assert_decrement assertion checker checks the expression test_expr at each rising edge of clk to determine if its value has changed from the one at the previous rising edge of clk. If so, the checker verifies that the new value equals the previous one decremented by value. The checker allows the value of test_expr to wrap, if the total change equals the decrement value.

- **assert_even_parity**

The assert_even_parity assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a value that has even parity. A value has even parity if it is 0 or if the number of bits set to 1 is even.

- **assert_fifo_index**

The assert_fifo_index assertion checker tracks the numbers of pushes (writes) and pops (reads) that occur for a FIFO or queue memory structure. This checker does permit simultaneous pushes/ pops on the queue within the same clock cycle. It ensures the FIFO never overflows (i.e., too many pushes occur without enough pops) and never underflows (i.e., too many pops occur without enough pushes).

- **assert_increment**

The assert_increment assertion checker checks the expression test_expr at each rising edge of clk to determine if its value has changed from the one at the previous rising edge of clk. If so, the checker verifies that the new value equals the previous one incremented by value. The checker allows the value of test_expr to wrap, if the total change equals the increment value.

- **assert_never**

The assert_never assertion checker checks the single-bit expression test_expr at each rising edge of clk to verify the expression does not evaluate to TRUE.

- **assert_one_hot**

The assert_one_hot assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a one-hot value. A one-hot value has exactly one bit set to 1.

- **assert_range**

The assert_range assertion checker checks the expression test_expr at each rising edge of clk to verify the expression falls in the range from min to max, inclusive. The assertion fails if test_expr< min or max < test_expr.

- **assert_one_cold**

The assert_one_cold assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a one-cold or inactive state value. A one-cold value has exactly one bit set to 0.