## Unit - I

- Introduction to Verilog HDL
- Hierarchical Modeling Concepts
- Basic Concepts





## Introduction to VLSI Design

### Vacuum Tube:

- A Vacuum tube was first developed by John Ambrose Fleming in 1904 and is a glass tube that has had all gas has been removed creating a vacuum.
- Vacuum tubes contain electrodes for controlling electron flow in early computers that used them as a switch or an amplifier.



## First Transistor:

- A transistor is a semiconductor device used to amplify and switch electronic signals and electrical power.

- John Bardeen, William Shockley and Walter Brattain at Bell Labs, 1947.
- They were honored with the Nobel Prize in Physics "for their researches on semiconductors and their discovery of the transistor effect" in 1956.





## First Integrated Circuit :

- It is small chip that can function as an amplifier, oscillator, timer, microprocessor, or even computer memory.
- An IC is a small wafer, usually made of silicon, that can hold anywhere from hundreds to millions of transistors, resistors, and capacitors.

Jack Kilby's original integrated circuit, 1958

- Kilby won the 2000 Nobel Prize in Physics for his part of the invention of the integrated circuit.

## Integrated Circuits (ICs)

- Analog ICs process with analog waveform at input and output sides.
- Digital ICs process only on/off signals. These devices can be found in microprocessors, memory chips, and microcomputers.
- Digital logic is implemented using transistors in integrated circuits containing many gates.

  Small-Scale Integrated circuits (SSI) contain 10 gates or less
  Medium-Scale Integrated circuits (MSI) contain 10-1000 gates
  Large-Scale Integrated circuits (LSI) contain 1000-100000 gates
  Very Large-Scale Integrated circuits (VLSI) contain >1000000 gates

- Doubles Transistor count every 18 months or so Variety of logic families
  TTL - Transistor-Transistor Logic
  CMOS - Complementary Metal-Oxide Semiconductor
  ECL - Emitter-Coupled Logic





## Why VLSI?

- Integration improves the design
  Lower parasitic = higher speed
  Lower power consumption
  Physically smaller
- Integration reduces manufacturing cost - (almost) no manual assembly

## Advantages of VLSI

- Reduced Size
- Increased Speed
- Reduced Power Consumption
- Reduced Cost







3

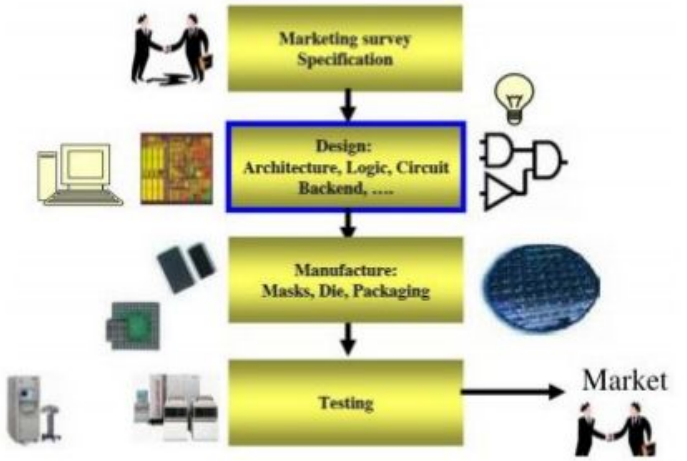
## VLSI Applications

- VLSI is an implementation technology for electronic circuitry - analogue or digital
- It is concerned with forming a pattern of interconnected switches and gates on the surface of a crystal of semiconductor
- Microprocessors
  - personal computers
  - microcontrollers
- Memory - DRAM / SRAM
- Special Purpose Processors - ASICS (CD players, DSP applications)
- Optical Switches
- Has made highly sophisticated control systems mass - producible at Low cost





## Flow

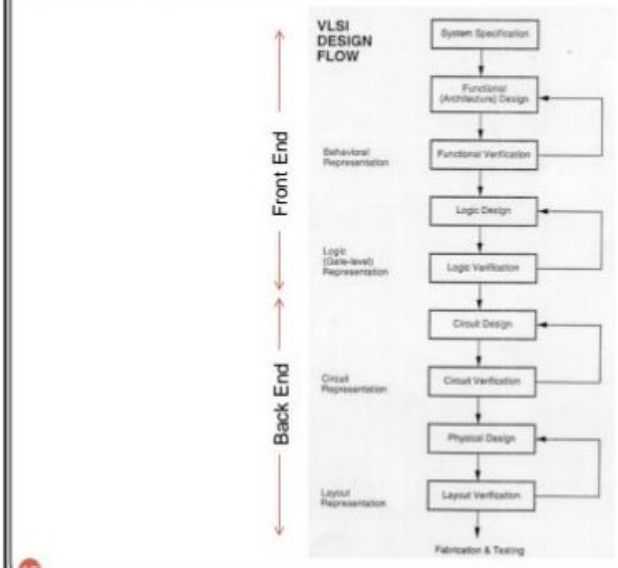









4

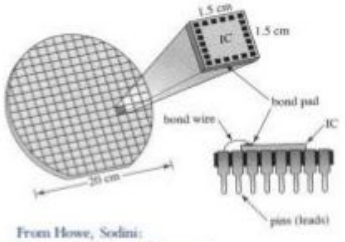







**After Front End & Back End processing -**

- Fabrication steps that go towards forming the devices and inter-connections between these devices to produce the functioning IC's.
- The end result are wafers each containing a regular array of the same IC chip or die.
- The wafer then has to be tested and the chips diced up and the good chips mounted and wire-bonded in different types of IC package and tested again before being shipped out.




From Howe, Sodini: Microelectronics:An Integrated Approach, Prentice Hall







5

## FPGA Design Flow






## ASIC Design Flow

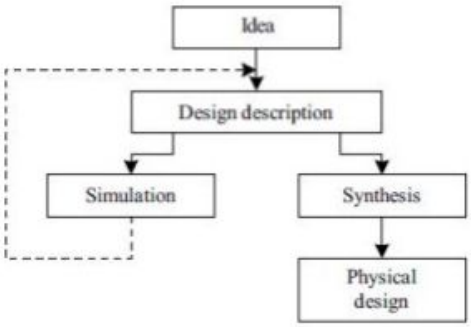



**Figure** Major activities in ASIC design.

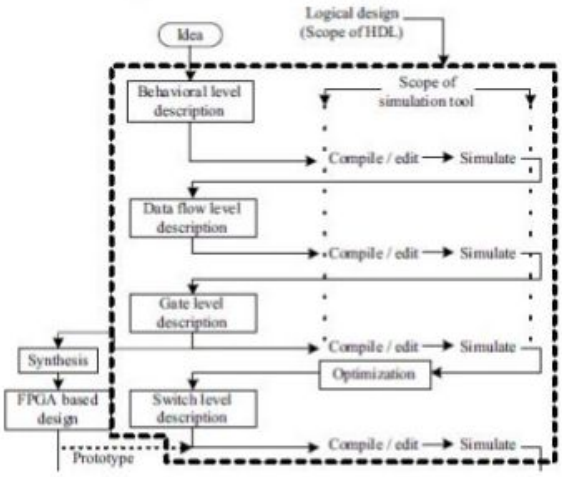






6

## ASIC design and development flow

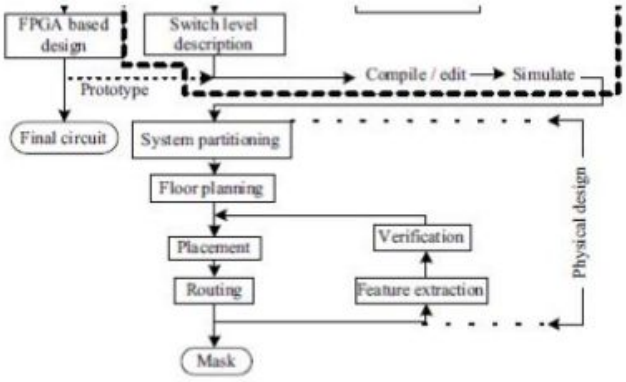



Contd..





## ASIC design and development flow










7

# Introduction to Verilog HDL

## 1.1. Evolution of Computer Aided Digital Design (CAD):

- The earliest digital circuits were designed with vacuum tubes and transistors. Integrated circuits were then invented where logic gates were placed on a single chip.
- The first integrated circuit (IC) chips were SSI (Small Scale Integration) chips where the gate count was very small.
- As technologies became sophisticated, designers were able to place circuits with hundreds of gates on a chip.
- These chips were called MSI (Medium Scale Integration) chips.







## Contd..

- With the advent of LSI (Large Scale Integration), designers could put thousands of gates on a single chip.
- At this point, design processes started getting very complicated, and designers felt the need to automate these processes.
- Computer Aided Design (CAD) techniques began to evolve.
- Chip designers began to use circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 100 transistors.
- The circuits were still tested on the breadboard, and the layout was done on paper or by hand on a graphic computer terminal.









8

## Contd..

- With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 100,000 transistors.
- Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard.
- Computer-aided techniques became critical for verification and design of VLSI digital circuits.
- Computer programs to do automatic placement and routing of circuit layouts also became popular.
- The designers were now building gate-level digital circuits manually on graphic terminals.





## Contd..

- They would build small building blocks and then derive higher-level blocks from them.
- This process would continue until they had built the top-level block.
- Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip.
- As designs got larger and more complex, logic simulation assumed an important role in the design process.
- Designers could iron out functional bugs in the architecture before the chip was designed further.

## 1.2. Emergence of HDLs:

- For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature.
- Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits.
- Hardware Description Languages (HDLs) came into existence.
- HDLs allowed the designers to model the concurrency of processes found in hardware elements.
- Hardware description languages such as Verilog HDL and VHDL became popular.
- Verilog HDL originated in 1983 at Gateway Design Automation.





## Contd..

- Later, VHDL was developed under contract from DARPA.
- Both verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.
- Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates.
- The advent of logic synthesis in the late 1980s changed the design methodology radically.
- Digital circuits could be described at a register transfer level (RTL) by use of an HDL.







10

## Contd..

- Thus, the designer had to specify how the data flows between registers and how the design processes the data.
- The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.
- Thus, logic synthesis pushed the HDLs into the forefront of digital design.
- Designers no longer had to manually place gates to build digital circuits.
- They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs.
- Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.





## Contd..

- HDLs also began to be used for system-level design.
- HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALS (Programmable Array Logic).
- A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.



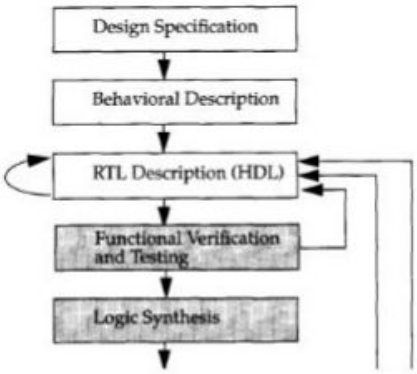




11

## 1.3. Typical Design Flow :

- Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.




Contd..





## Typical Design Flow :

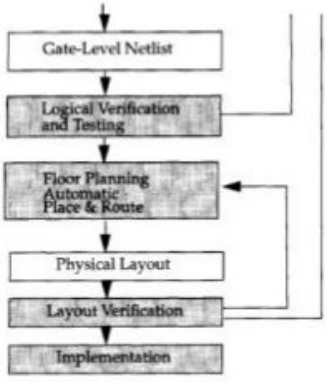



Figure I Typical Design Flow







12

## Introduction to Verilog HDL

- Verilog HDL is a Hardware Description Language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic-level to the gate level to the switch level.
- First developed by Gateway Design Automation in 1983.
- It was Proprietary Language
- Phil Moorby and Prabhu Goel
- Public Domain - 1990
- Open Verilog International (OVI) – Promote Verilog HDL
- IEEE standard – 1995
- IEEE Std 1364-1995
- Verilog HDL Language Reference Manual
- New Version in 2005 - IEEE Std 1364-2005





## Language Capabilities

1. Primitive logic gates such as ***and***, ***or*** and ***nand*** are built-in into the language.

2. Flexibility of creating a user-defined primitive (UDP). Such a primitive could either be a combinational logic primitive or a sequential logic primitive.

3. Switch level modeling primitive trasistors, such as ***pmos*** and ***nmos***, are also built-in into language.

4. A design can be modeled in three different styles or in a mixed style. These styles are:

i. **Behavioral style** – modeled using "**Procedural Constructs**"
Ex: always, initial – Sequential Execution

ii. **Dataflow style** – modeled using "**Continuous Assignments**"
Ex: assign – Concurrent (parallel) Execution

iii. **Structural style** – modeled using "**gate**" and "**module instantiations**".







13

## Language Capabilities

**Comparison among Different Modeling Levels:**

**Gate Level Modeling :**

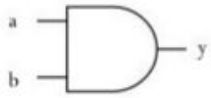
```
module AND_GATE(a,b,y);
input a;
input b;
output y;

and G1(y,a,b);

endmodule
```

**Data Flow Level Modeling :**

```
module AND_GATE(a,b,y);
input wire a;
input wire b;
output wire y;

assign y = a & b;

endmodule
```

**Behavioral Level Modeling :**

```
module AND_GATE(a,b,y);
input a;
input b;
output y;
reg y;

always@(a,b)
begin
y = a & b;
end

endmodule
```








## Language Capabilities Contd..

5. There are two data types in Verilog HDL
   a) The **net data type** represents a physical connection between structural elements.
   b) **Variable data type** may represent an abstract data storage element.
6. Hierarchical designs can be described, up to any level, using the module instantiation construct.
7. A design can be of arbitrary size; the language does not impose a limit.
8. Verilog HDL is non-proprietary and is an IEEE standard.
9. It is human and machine readable. Thus it can be used as an exchange language between tools and designers.





14

## Language Capabilities Contd..

10. Explicit language constructs are provided for specifying pin-to-pin delays, path delays and timing checks of a design.

11. The capabilities of the Verilog HDL language can be further extended by using the programming language interface (PLI) mechanism. PLI is a collection of routines that allow foreign functions to access information within a Verilog module and allows for designer interaction with the simulator.

12. A design can be described in a wide range of levels, ranging from switch level, gate level, register-transfer-level (RTL) to algorithmic level, including process and queuing level.

13. A design can be modeled entirely at the switch level using the built-in switch-level primitives.





## Language Capabilities Contd..

14. The same single language can be used to generate stimulus for the design and for specifying test constraints, such as specifying the values of inputs.

15. Verilog HDL can be used to perform response monitoring of the design under test, that is, the values of a design under test can be monitored and displayed. These values can also be compared with expected values, and in case of a mismatch, a report message can be printed.

16. At the behavioral-level, Verilog HDL can be used to describe a design not only at the RTL-level, but also at the architectural-level and its algorithmic-level behavior.

17. At the structural-level, gate and module instantiations can be used.

18. Verilog HDL also has built-in logic functions such as & (bitwise-and)|(bitwise-or).







15

## Language Capabilities Contd..

18. High-level programming language constructs such as conditionals, case statements, and loops are available in the language.
19. Notation of concurrency and time can be explicitly modeled.
20. Powerful file read and write capabilities are provided.
21. The language is non-deterministic under certain situations, that is, a model may produce different results on different simulators; for example, the ordering of events on an event queue is not defined by the standard.





## 2. Hierarchical Modeling Concepts

### 2.1. Design Methodologies or Design Hierarchy

- Top Down Deign Methodology
- Bottom Up Design Methodology







16

## Top-Down Design Methodology

- We define the top-level block and identify the sub-blocks necessary to build the top-level block.
- We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.

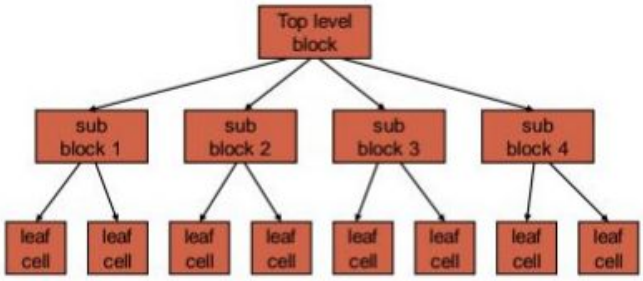



Figure 2-1 Top-down Design Methodology





## Bottom-up Design Methodology

- We first identify the building block that are available to us.
- We build bigger cells, using these building blocks.
- These cells are then used for higher-level blocks until we build the top-level block in the design.

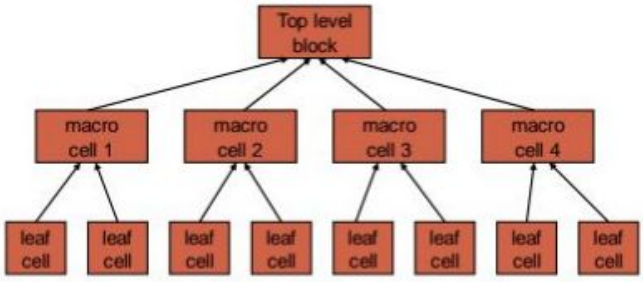



Figure 2-2 Bottom-up Design Methodology

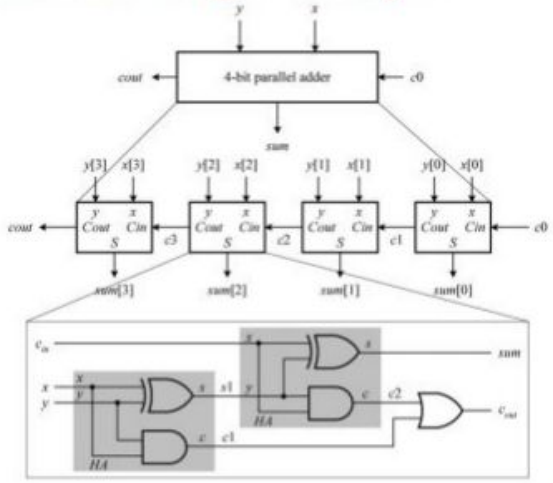






17

## Hierarchical Design - Example








## 2.2 4-bit Ripple Carry Counter:

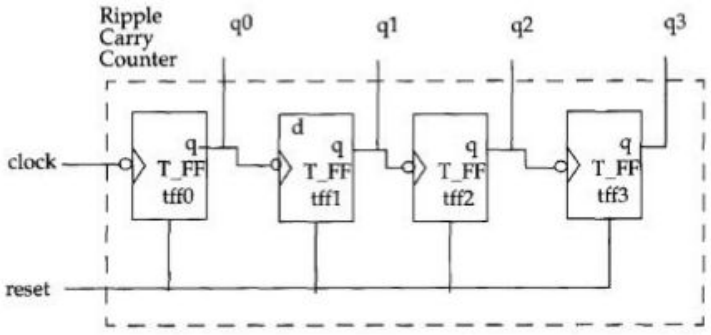



Figure 2-3 Ripple Carry Counter







18

**Ex: 4-bit Ripple Carry Counter**

| reset | $q_n$ | $q_{n+1}$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

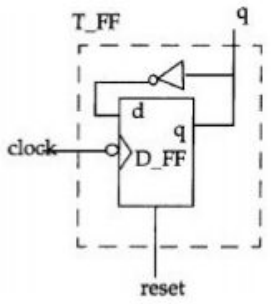



Figure 2-4 T-Flip Flop





**Ex: 4-bit Ripple Carry Counter**

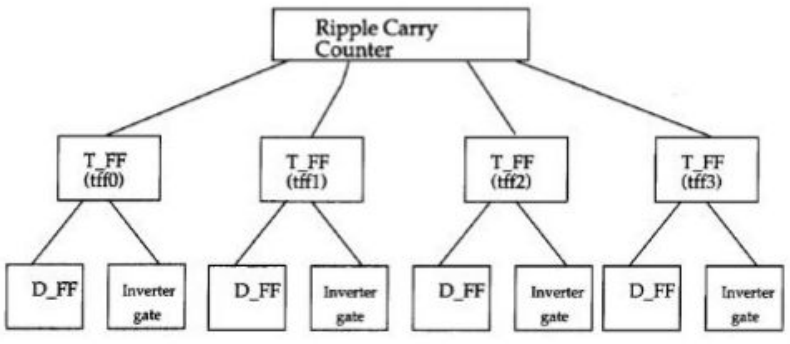



*Fig: Design Hierarchy*

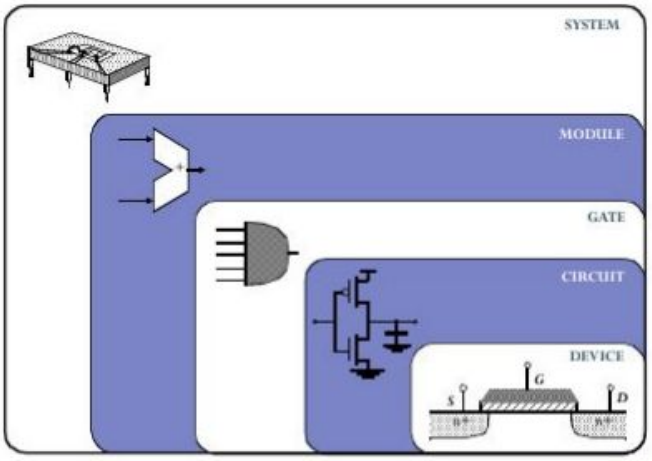






19

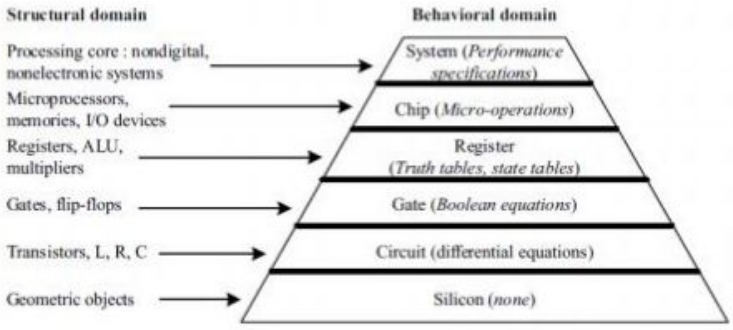
## Design Abstraction Levels








## Design Domains and Levels of Abstraction










20

## Levels of Abstraction:

**Behavioral or algorithmic level**

- This is the highest level of abstraction provided by Verilog HDL.
- A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details.
- Designing at this level is very similar to C programming.
- Modeled using "Procedural Constructs"

  Ex: always, initial – Sequential Execution

**Dataflow level**

- At this level the module is designed by specifying the data flow.
- The designer is aware of how data flows between hardware registers and how the data is processed in the design.
- Modeled using "Continuous Assignments"

  Ex: assign – Concurrent (parallel) Execution





## Contd..

**Gate level**

- The module is implemented in terms of logic gates and interconnections between these gates.
- Design at this level is similar to describing a design in terms of a gate-level logic diagram.
- Modeled using "gate" and "module instantiations".

**Switch level**

- This is the lowest level of abstraction provided by Verilog.
- A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.







21

## 2.3 Module:

- We now relate these hierarchical modeling concepts to Verilog.
- Verilog provides the concept of a module. A module is the basic building block in Verilog.
- A module can be an element or a collection of lower-level design blocks.
- Typically, elements are grouped into modules to provide common functionality that is used at many places in the design.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation.
- This allows the designer to modify module internals without affecting the rest of the design.





## Contd..

- In Verilog, a module is declared by the keyword **module**. A corresponding keyword **endmodule** must appear at the end of the module definition.
- Each module must have a **module-name**, which is the **identifier** for the module, and a **module-terminal-list**, which describes the **input** and **output** terminals of the module.

**Comparison Between C and Verilog HDL Languages**

```
#include <stdio.h>
main() {
int number1, number2, sum;

// calculating sum
sum = number1 + number2;

}
```

```
module AND_GATE(a,b,y);
input a;
input b;
output y;

and G1(y,a,b);

endmodule
```







22

## Contd..

```
module <module_name> (<module_terminal_list>);
…
<module internals>
…
…
endmodule
```

```
module T_FF (q, clock, reset);
.
.
<functionality of T-flipflop>
.
.
endmodule
```





## 2.4. Instances :

- A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.
- The process of creating objects from a module template is called instantiation, and the objects are called instances.
- In Example 2-1, the top-level block creates four instances from the T-flipflop (T-FF) template.
- Each T-FF instantiates a D-FF and an inverter gate.
- Each instance must be given a unique name. Note that // is used to denote single-line comments.







23

## Contd..

*Example 2-1* *Module Instantiation*

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations
                //will be explained later.
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```





## Contd..

```
// Define the module T_FF. It instantiates a D-flipflop. We assumed
// that module D-flipflop is defined elsewhere in the design. Refer
// to Figure 2-4 for interconnections.
module T_FF(q, clk, reset);

//Declarations to be explained later
output q;
```

```
input clk, reset;
wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive. Explained later.

endmodule
```







24

## Contd..

- In Verilog, it is illegal to nest modules.
- One module definition cannot contain another module definition within the module and endmodule statements.
- Instead, a module definition can incorporate copies of other modules by instantiating them.
- It is important not to confuse **module definitions** and **instances of a module**.
- Module definitions simply specify how the module will work, its internals, and its interface.
- Modules must be instantiated for use in the design.





## Contd..

- Example 2-2 shows an illegal module nesting where the module T-FF is defined inside the module definition of the ripple carry counter.

*Example 2-2* *Illegal Module Nesting*

```
// Define the top-level module called ripple carry counter.
// It is illegal to define the module T_FF inside this module.
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;

    module T_FF(q, clock, reset);// ILLEGAL MODULE NESTING
    …
    <module T_FF internals>
    …
    endmodule // END OF ILLEGAL MODULE NESTING

endmodule
```







25

## Components of a Simulation:

- Once a design block is completed, it must be tested.
- The functionality of the design block can be tested by applying stimulus and checking results.
- We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate.
- The stimulus block can be written in Verilog.
- A separate language is not required to describe stimulus.
- The stimulus block is also commonly called a test bench.
- Different test benches can be used to thoroughly test the design block.





- Two styles of stimulus application are possible. **Contd..**
- In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block.
- In Figure 2-6, the stimulus block becomes the top-level block.
- It manipulates signals clk and reset, and it checks and displays output signal q.

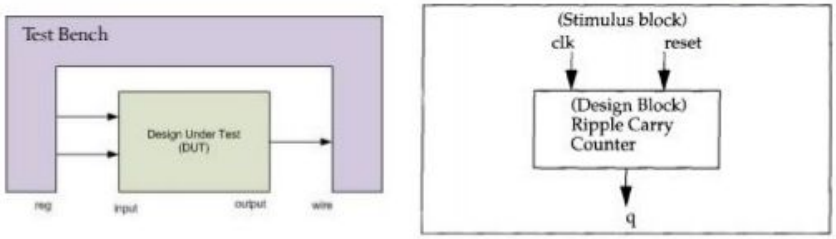



*Figure 2-6 Stimulus Block Instantiates Design Block*





26

## Contd..

- The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module.
- The stimulus block interacts with the design block only through the interface.
- This style of applying stimulus is shown in Figure 2-7.
- The stimulus module drives the signals d-clk and d-reset, which are connected to the signals clk and reset in the design block.
- It also checks and displays signal c-q, which is connected to the signal q in the design block.
- The function of top-level block is simply to instantiate the design and stimulus blocks.





## Contd..

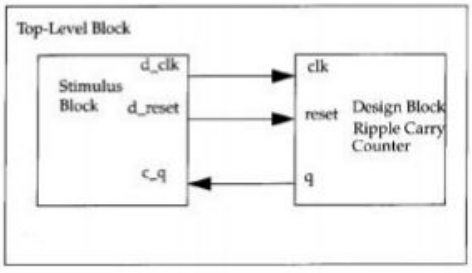



*Figure 2-7 Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module*

- Either stimulus style can be used effectively



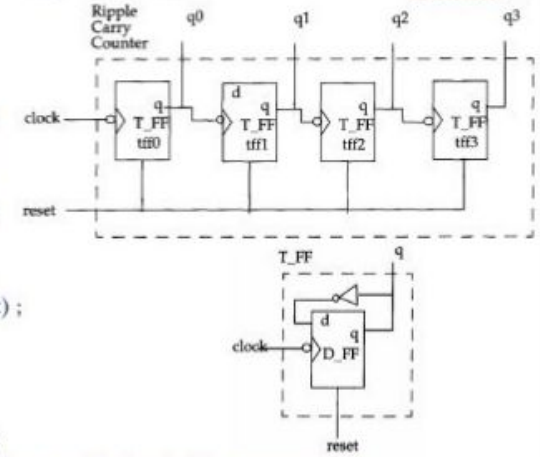




27

Contd..

```
module ripple_carry_counter(q, clk, reset) ;
output [3:0] q;
input clk, reset;

T-FF tff0(q[0] ,clk, reset);
T-FF tff1(q[1] ,q[0], reset);
T-FF tff2 (q[2] ,q[1], reset);
T-FF tff3 (q[3] ,q[2], reset);
endmodule

module T_FF (q, clk, reset) ;
output q;
input clk, reset;
wire d;
D-FF dff0 (q, d, clk, reset) ;
not n1(d, q) ; // not is a Verilog-provided primitive. case sensitive
endmodule
```








Contd..

```
// module D-FF with synchronous reset
module D_FF(q, d, clk, reset) ;
output q;
input d, clk, reset;
reg q;
// Lots of new constructs. Ignore the functionality of the constructs.
// Concentrate on how the design block is built in a top-down fashion.
always @(posedge reset or negedge clk)
if (reset)
q = 1'b0;
// module D-FF with synchronous reset
else
q = d;
endmodule
```







28

## Stimulus Block

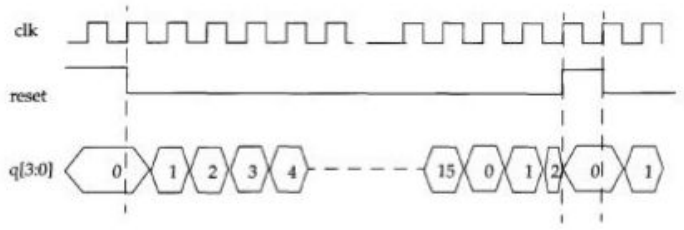



*Figure 2-8 Stimulus and Output Waveforms*





Contd..

```
`timescale 1ns/100ps
module stimulus;
reg clk, reset;
wire [3:0] q;
// instantiate the design block
ripple-carry-counter rl(q, clk, reset);
initial  clk = 1'bO; //set clk to 0
always  #5 clk = ~clk; //toggle clk every 5 time units
// Control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220.
initial
begin
reset = 1'b1;  #15 reset = 1'bO; #180 reset = 1'b1; #10 reset = 1'bO;
#20 $finish; //terminate the simulation
end
// Monitor the outputs
initial  $monitor ($time, " Output q = %d" , q) ;
endmodule
```







29

*Example 2-6* *Output of the Simulation* Contd..

```
  0 Output q =  0
 20 Output q =  1
 30 Output q =  2
 40 Output q =  3
 50 Output q =  4
 60 Output q =  5
 70 Output q =  6
 80 Output q =  7
 90 Output q =  8
100 Output q =  9
110 Output q = 10
120 Output q = 11
130 Output q = 12
140 Output q = 13
150 Output q = 14
160 Output q = 15
170 Output q =  0
180 Output q =  1
190 Output q =  2
195 Output q =  0
210 Output q =  1
220 Output q =  2
```



## Summary Contd..

- Two kinds of design methodologies are used for digital design: top-down and bottom-up. A combination of these two methodologies is used in today's digital designs. As designs become very complex, it is important to follow these structured approaches to manage the design process.
- Modules are the basic building blocks in Verilog. Modules are used in a design by instantiation. An instance of a module has a unique identity and is different from other instances of the same module. Each instance has an independent copy of the internals of the module. It is important to understand the difference between modules and instances.
- There are two distinct components in a simulation: a design block and a stimulus block. A stimulus block is used to test the design block. The stimulus block is usually the top-level block. There are two different styles of applying stimulus to a design block.







30

## 3.1 Lexical Conventions :

- Whitespace
- Comments
- Operators
- Number Specification
- Strings
- Identifiers and Keywords
- Escaped Identifiers





## 3. Basic Concepts:

**Learning Objectives:**

- Understand lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers.
- Define the logic value set and data types such as nets, registers, vectors, numbers, simulation time, arrays, parameters, memories, and strings.
- Identify useful system tasks for displaying and monitoring information, and for stopping and finishing the simulation.
- Learn basic compiler directives to define macros and include files.







31

### 3.1.1 Whitespace

- Blank spaces (\b) , tabs (\t) and newlines (\n) comprise the whitespace.
- Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

### 3.1.2 Comments

- Comments can be inserted in the code for readability and documentation.
- There are two ways to write comments.
- A one-line comment starts with "//".
- Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested.





```
a = b && c; // This is a one-line comment

/* This is a multiple line
   comment */

/* This is /* an illegal */ comment */
```

### 3.1.3 Operators:

- Operators are of three types, unary, binary, and ternary. Unary operators precede the operand.
- Binary operators appear between two operands.
- Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```





32

## 3.1.4. Number Specification: Contd..

- There are two types of number specification in Verilog: sized and unsized.

**Sized numbers**

- Sized numbers are represented as **<size>' <base format> <number>**.
- <size> is written only in decimal and specifies the number of bits in the number.
- Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('0 or '0). The number is specified as consecutive digits from O,1,2,3,4,5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

```
4'b1111 // This is a 4-bit   binary number
12'habc // This is a 12-bit  hexadecimal number
16'd255 // This is a 16-bit  decimal number.
```





## Unsized numbers Contd..

- Numbers that are specified without a <base format> specification are decimal numbers by default.
- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

```
23456 // This is a 32-bit   decimal number by default
'hc3 // This is a 32-bit   hexadecimal number
'o21 // This is a 32-bit   octal number
```

- **X or Z values**
- Verilog has two symbols for unknown and high impedance values.
- These values are very important for modeling real circuits. An unknown value is denoted by an X. A high impedance value is denoted by z.







33

Contd..

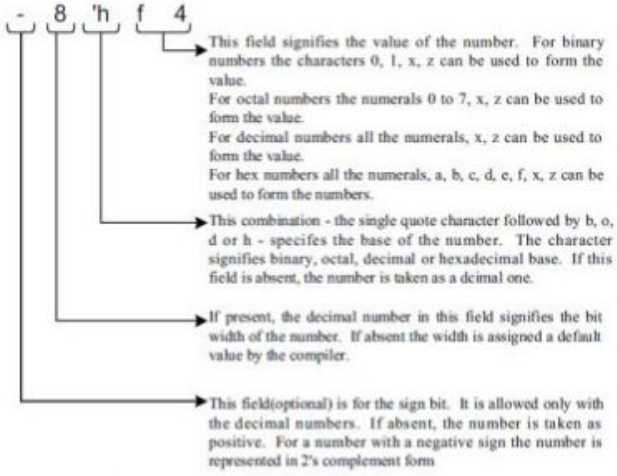



**Figure 3.1** Representation of a number in Verilog: One can use capital letters instead of small letters in the last two fields.







Contd..

| Representation | Remarks |
|---|---|
| 33<br>'d33 | Both of these represent decimal numbers of unspecified size – normally interpreted by Verilog as 32 bitwide, *i.e.*, 0000 0000 0000 0000 0000 0000 0010 0001 |
| 9'd439<br>9'D439<br>9'D4_39 | All these represent 3 digit decimal numbers. D & d both specify decimal numbers. "_" (underscore) is ignored |
| 9'b1_1011__1x01<br>9'b11011x01<br>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<br>9'O123<br>9'o1x3<br>9'o12z | All these represent 9-bit octal numbers. The binary equivalents are 001 010 011,<br>001 010 011, 001 xxx 011, 001 010 zzz respectively. z signifies the concerned bits to be in the high impedance state. |
| 'o213 | An octal number of unspecified size having octal value 213. |
| 8'ha5<br>8'HA5<br>8'hA5<br>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<br>-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. |









34

- An X or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base.
- If the most significant bit of a number is of X, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, X, or z. This makes it easy to assign X or z to whole vector. If the most significant digit is I, then it is also zero extended.

```
12'h13x // This is a 12-bit hex number; 4 least significant bits unknown
6'hx // This is a 6-bit hex number
32'bz // This is a 32-bit high impedance number
```

- Negative numbers
- Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>.

```
-6'd3 // 8-bit   negative number stored as 2's complement of 3
4'd-2 // Illegal specification
```

Contd..





Underscore characters and question marks

Contd..

- An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.
- A question mark "?" is the Verilog HDL alternative for z in the context of numbers. The ? is used to enhance readability in the **casex** and **casez** statements.

```
12'b1111_0000_1010 // Use of underline characters for readability
4'b10?? // Equivalent of a 4'b10zz
```

### 3.1.5. Strings :

- A string is a sequence of characters that are enclosed by double quotes.
- The restriction on a string is that it must be contained on a single line, that is, without a carriage return.







35

- It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

### 3.1.6. Identifiers and Keywords:

- Keywords are special identifiers reserved to define the language constructs.
- Keywords are in lowercase.
- Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore ( - ) and the dollar sign ( $ ) and are case sensitive.
- Identifiers start with an alphabetic character or an underscore.
- They cannot start with a number or a $ sign (The $ sign as the first character is reserved for system tasks).





```
reg value; // reg is a keyword; value is an identifier
input clk; // input is a keyword, clk is an identifier
```

### 3.1.7. Escaped Identifiers :

- Escaped identifiers begin with the backslash ( \ ) character and end with whitespace (space, tab, or newline).
- All characters between backslash and whitespace are processed literally.
- Any printable ASCII character can be included in escaped identifiers.
- The backslash or whitespace is not considered a part of the identifier.

```
\a+b-c
\**my_name**
```







36

## 3.2. Data Types

1. Value Set
2. Nets
3. Registers
4. Vectors
5. Integer, Real, and Time Register Data Types
6. Arrays
7. Memories
8. Parameters
9. Strings





### 3.2.1. Value Set:

Verilog supports four values and eight strengths to model the functionality of real hardware.

The four value levels are listed in Table 3-1.

*Table 3-1 Value Levels*

| Value Level | Condition in Hardware Circuits |
|---|---|
| 0 | Logic zero, false condition |
| 1 | Logic one, true condition |
| x | Unknown value |
| z | High impedance, floating state |

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits.

Value levels o and 1 can have the strength levels listed in Table 3-2.







37

## Contd..

*Table 3-2 Strength Levels*

| Strength Level | Type | Degree |
|---|---|---|
| supply | Driving | strongest |
| strong | Driving | |
| pull | Driving | |
| large | Storage | |
| weak | Driving | |
| medium | Storage | |
| small | Storage | |
| highz | High Impedance | weakest |





## Contd..

- If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
- For example, if two signals of strength strongl and weak0 contend, the result is resolved as a strongl.
- If two signals of equal strengths are driven on a wire, the result is unknown.
- If two signals of strength strongl and strong0 conflict, the result is an X.
- Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.
- Only trireg nets can have storage strengths large, medium, and small.







38

## 3.2.2. Nets :

- Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.
- In Figure 3-1 net a is connected to the output of and gate gl. Net a will continuously assume the value computed at the output of gate gl, which is b & C.

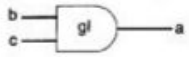



*Figure 3-1 Example of Nets*

- Nets are declared primarily with the keyword wire.
- Nets are one-bit values by default unless they are declared explicitly as vectors.
- The terms w i r e and net are often used interchangeably.





## Contd..

- The default value of a net is z (except the trireg net, which defaults to X ).
- Nets get the output value of their drivers. If a net has no driver, it gets the value z.

```
wire a; // Declare net a for the above circuit
wire b,c; // Declare two wires b,c for the above circuit
wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.
```

- Note that net is not a keyword but represents a class of data types such as wire , wand, wor, tri, triand, trior, trireg, etc.
- The w i r e declaration is used most frequently.







39

### 3.2.3. Registers:

- Registers represent data storage elements. Registers retain value until another value is placed onto them.
- Do not confuse the term registers in Verilog with hardware registers built from edge-triggered flip-flops in real circuits. In Verilog, the term register merely means a variable that can hold a value.
- Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do.
- Values of registers can be changed anytime in a simulation by assigning a new value to the register. Register data types are commonly declared by the keyword reg.
- The default value for a reg data type is X. An example of how registers are used is shown Example 3-1.





*Example 3-1* *Example of Register*

Contd..

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
  reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
  #100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

### 3.2.4. Vectors:

- Nets or reg data types can be declared as vectors (multiple bit widths).
- If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit   bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; //Vector register,virtual address 41 bits wide
```







40

## Contd..

- Vectors can be declared at [high# : low#] or [low# : high#], but the left number in the squared brackets is always the most significant bit of the vector.
- In the example shown above, bit 0 is the most significant bit of vector virtual-addr.
- For the vector declarations shown above, it is possible to address bits or parts of vectors.

```
busA[7] // bit # 7 of vector busA
bus[2:0] // Three least significant bits of vector bus.
 // using bus[0:2] is illegal because the significant bit should
 // always be on the left of a range specification
virtual_addr[0:1] // Two most significant bits of vector virtual_addr
```





## 3.2.5. Integer, Real, and Time Register Data Types:

- Integer, real, and time register data types are supported in Verilog.

### Integer

- An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting.
- The default width for an integer is the host-machine word size, which is implementation specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial
    counter = -1; // A negative one is stored in the counter
```





41

## Real Contd..

- Real number constants and real register data types are declared with the keyword real.
- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 X $10^6$ ).
- Real numbers cannot have a range declaration, and their default value is 0. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
real delta; // Define a real variable called delta
initial
begin
   delta = 4e10; // delta is assigned in scientific notation
   delta = 2.13; // delta is assigned a value 2.13
end
integer i; // Define an integer i
initial
   i = delta; // i gets the value 2 (rounded value of 2.13)
```





## Time Contd..

- Verilog simulation is done with respect to simulation time.
- A special time register data type is used in Verilog to store simulation time.
- A time variable is declared with the keyword **time**.
- The width for time register data types is implementation specific but is at least 64 bits. The system function $time is invoked to get the current simulation time.
- Simulation time is measured in terms of simulation seconds. The unit is denoted by S, the same as real time.

```
time save_sim_time; // Define a time variable save_sim_time
initial
   save_sim_time = $time; // Save the current simulation time
```







42

### 3.2.6. Arrays:

- Arrays are allowed in Verilog for reg, integer, time, and vector register data types.
- Arrays are not allowed for real variables.
- Arrays are accessed by <array-name> [<subscript>l. Multidimensional arrays are not permitted in Verilog.

```
integer count[0:7]; // An array of 8 count variables
reg bool[31:0]; // Array of 32 one-bit boolean register variables
time chk_point[1:100]; // Array of 100 time checkpoint variables
reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide
integer matrix[4:0][4:0]; // Illegal declaration. Multidimensional
array

count[5] // 5th element of array of count variables
chk_point[100] // 100th time check point value
port_id[3] // 3rd element of port_id array. This is a 5-bit value.
```

- It is important not to confuse arrays with net or register vectors.





- A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are l-bit or n-bits wide.

### 3.2.7. Memories:

- In digital simulation, one often needs to model register files, RAMS, and ROMs.
- Memories are modeled in Verilog simply as an array of registers. Each element of the array is known as a word. Each word can be one or more bits.
- It is important to differentiate between n l-bit registers and one n-bit register.
- A particular word in memory is obtained by using the address as a memory array subscript.

```
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words (bytes)
membyte[511] // Fetches 1 byte word whose address is 511.
```







43

### 3.2.8. Parameters:

- Verilog allows constants to be defined in a module by the keyword parameter.
- Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time.
- This allows the module instances to be customized. Module definitions may be written in terms of parameters. Hardcoded numbers should be avoided.
- Parameters can be changed at module instantiation or by using the defparam statement, Useful Modeling Techniques.
- Thus, use of parameters makes the module definition flexible.
- Module behavior can be altered simply by changing the value of a parameter.





```
parameter port_id = 5; // Defines a constant port_id
parameter cache_line_width=256; // Constant defines width of cache line
```

### 3.2.9. Strings:

- Strings can be stored in reg.
- The width of the register variables must be large enough to hold the string.
- Each character in the string takes up 8 bits (1 byte). If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros.
- If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string.
- It is always safe to declare a string that is slightly wider than necessary.







44

## Contd..

```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes
wide
initial
   string_value = "Hello Verilog World"; // String can be stored
                                         // in variable
```

- Special characters serve a special purpose in displaying strings, such as newline, tabs and displaying argument values.
- Special characters can be displayed in strings only when they are preceded by escape characters, as shown in Table 3-3.

*Table 3-3 Special Characters*

| Escaped Characters | Character Displayed |
|---|---|
| \n | newline |
| \t | tab |
| %% | % |
| \\ | \ |
| \" | " |
| \ooo | Character written in 1–3 octal digits |





## 3.3. System Tasks and Compiler Directives :

- In this section we introduce two special concepts used in Verilog: system tasks and compiler directives.

### 3.3.1. System Tasks :

- Verilog provides standard system tasks to do certain routine operations.
- All system tasks appear in the form $<keyword>.
- Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks. We will discuss only the most useful system tasks.
- Other tasks are listed in Verilog manuals provided by your simulator vendor or in the Verilog HDL Language Reference Manual.







45

## Displaying information:

- $display is the main system task for displaying values of variables or strings or expressions.
- This is one of the most useful tasks in Verilog.
- Usage: $display(p1, p2, p3 ,....., pn);
- p1, p2, p3, ..., pn can be quoted strings or variables or expressions. The format of $display is very similar to printf in C.
- A $display inserts a newline at the end of the string by default. A $display without any arguments produces a newline. Strings can be formatted by using the format specifications listed in Table 3-4.
- For more detailed format specifications, see Verilog HDL Language Reference Manual.





## Contd..

*Table 3-4 String Format Specifications*

| Format | Display |
|---|---|
| %d or %D | Display variable in decimal |
| %b or %B | Display variable in binary |
| %s or %S | Display string |
| %h or %H | Display variable in hex |
| %c or %C | Display ASCII character |
| %m or %M | Display hierarchical name (no argument required) |
| %v or %V | Display strength |
| %o or %O | Display variable in octal |
| %t or %T | Display in current time format |
| %e or %E | Display real number in scientific format (e.g., 3e10) |
| %f or %F | Display real number in decimal format (e.g., 2.13) |
| %g or %G | Display real number in scientific or decimal, whichever is shorter |







46

## Contd..

- Example 3-2 shows some examples of the $display task. If variables contain X or z values they are printed in the displayed string as X or z.

*Example 3-2 $display Task*

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World

//Display value of current simulation time 230
$display($time);
-- 230

//Display value of 41-bit virtual address 1fe0000001c and time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c

//Display value of port_id 5 in binary
reg [4:0] port_id;
```





*Example 3-2 $display Task (Continued)*

## Contd..

```
$display("ID of the port is %b", port_id);
-- ID of the port is 00101

//Display x characters
//Display value of 4-bit bus 10xx (signal contention) in binary
reg [3:0] bus;
$display("Bus value is %b", bus);
-- Bus value is 10xx

//Display the hierarchical name of instance p1 instantiated under
//the highest-level module called top. No argument is required.This
//is a useful feature)
$display("This string is displayed from %m level of hierarchy");
-- This string is displayed from top.p1 level of hierarchy
```

- Special characters are discussed in Section 3.2.9, Strings. Examples of displaying special characters in strings as discussed are shown in Example 3-3.

*Example 3-3 Special Characters*

```
//Display special characters, newline and %
$display("This is a \n multiline string with a %% sign");
-- This is a
-- multiline string with a % sign

//Display other special characters
```







47

## Monitoring information:

- Verilog provides a mechanism to monitor a signal when its value changes.
- This facility is provided by the $monitor task.
- Usage: $monitor(p1 ,p2,p3 ,...., pn);
- The parameters pl, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the $display task is used in the $monitor task.
- $monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes.
- Unlike $display, $monitor needs to be invoked only once.





- Only one monitoring list can be active at a time. **Contd..**
- If there is more than one $monitor statement in your simulation, the last $monitor statement will be the active statement. The earlier $monitor statements will be overridden. Two tasks are used to switch monitoring on and off.
- Usage: $monitoron;
- $monitoroff;
- The $monitoron tasks enables monitoring, and the $monitoroff task disables
- monitoring during a simulation.
- Monitoring is turned on by default at the beginning of the simulation and can be controlled during the simulation with the $monitoron and $monitoroff tasks. Examples of monitoring statements are given in Example 3-4. Note the use of $time in the $monitor statement.







48

*Example 3-4* *Monitor Statement* Contd..

```
//Monitor time and value of the signals clock and reset
//Clock toggles every 5 time units and reset goes down at 10 time units
initial
begin
   $monitor($time,
             " Value of signals clock = %b reset = %b", clock,reset);
end
```

*Partial output of the monitor statement:*

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

- Stopping and finishing in a simulation
- The task $stop is provided to stop during a simulation.
- Usage: $stop;
- The $stop task puts the simulation in an interactive mode.





- The designer can then debug the design from the interactive mode. Contd..
- The $stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.
- The $finish task terminates the simulation.
- Usage: $finish;
- Examples of $stop and $finish are shown in Example 3-5.

*Example 3-5* *Stop and Finish Tasks*

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time.
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```







49

## 3.3.2. Compiler Directives :

- Compiler directives are in Verilog. All compiler directives are defined by using the ' <keyword> construct. We deal with the two most useful compiler directives.

### `define

- The 'define directive is used to define text macros in Verilog (see Example 3-6).
- This is similar to the #define construct in C.
- The defined constants or text macros are used in the Verilog code by preceding them with a ' (back tick). The Verilog compiler substitutes the text of the macro wherever it encounters a ' <macro-name>.





*Example 3-6* *'define Directive*

Contd..

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
`define WORD_SIZE 32

//define an alias. A $stop will be substituted wherever `S appears
`define S $stop;

//define a frequently used text string
`define WORD_REG reg [31:0]
// you can then define a 32-bit register as `WORD_REG reg32;
```

### `include

- The ' include directive allows you to include entire contents of a Verilog source
- file in another Verilog file during compilation. This works similarly to the #include in the C programming language.
- This directive is typically used to include header files, which typically contain global or commonly used definitions (see Example 3-7).Two other directives, ' ifdef and ' timescale, are used frequently.







50

## Contd..

*Example 3-7* *`include Directive*

```
// Include the file header.v, which contains declarations in the
// main verilog file design.v.
`include header.v
..
..
<Verilog code in file design.v>
..
..
```





## Summary:

- We discussed the basic concepts of Verilog in this chapter. These concepts lay the foundation for the material discussed in the further chapters.
- Verilog is similar in syntax to the C programming language . Hardware designers with previous C programming experience will find Verilog easy to learn.
- Lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers were discussed.
- Various data types are available in Verilog. There are four logic values, each with different strength levels.
- Available data types include nets, registers, vectors, numbers, simulation time, arrays, memories, parameters, and strings.







51

## Contd..

- Data types represent actual hardware elements very closely.
- Verilog provides useful system tasks to do functions like displaying, monitoring, suspending, and finishing a simulation.
- Compiler directive 'define is used to define text macros, and ' include is used to include other Verilog files.





# 4. Modules and Ports

## Learning Objectives:

- Identify the components of a Verilog module definition, such as module names, port lists, parameters, variable declarations, dataflow statements, behavioral statements, instantiation of other modules, and tasks or functions.
- Understand how to define the port list for a module and declare it in Verilog.
- Describe the port connection rules in a module instantiation.
- Understand how to connect ports to external signals, by ordered list, and by name.
- Explain hierarchical name referencing of Verilog identifiers.







52

## 4.1. Modules:

- We ignored the internals of modules and concentrated on how modules are defined and instantiated. A module in Verilog consists of distinct parts, as shown in Figure 4-1.

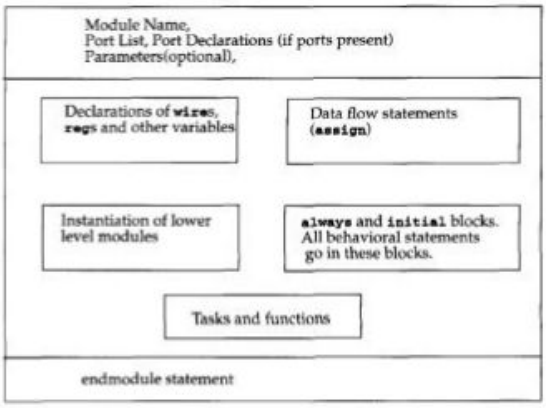



*Figure 4-1 Components of a Verilog Module*



## Contd..

- A module definition always begins with the keyword **module**. The module name, port list, port declarations, and optional parameters must come first in a module definition.
- Port list and port declarations are present only if the module has any ports to interact with the external environment. The five components within a module are - variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions.
- These components can be in any order and at any place in the module definition. The endmodule statement must always come last in a module definition.
- All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

28 August 2021



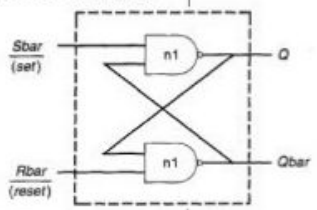


53

*Example 4-1* *Components of SR Latch*

Contd..

```
// This example illustrates the different components of a module

// Module name and port list
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);

//Port declarations
output Q, Qbar;
input Sbar, Rbar;

// Instantiate lower-level modules
// In this case, instantiate Verilog primitive nand gates
// Note, how the wires are connected in a cross-coupled fashion.
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

// endmodule statement
endmodule
```








Contd..

```
// Module name and port list
// Stimulus module
module Top;

// Declarations of wire, reg, and other variables

wire q, qbar;
reg set, reset;

// Instantiate lower-level modules
// In this case, instantiate SR_latch
// Feed inverted set and reset signals to the SR latch
SR_latch m1(q, qbar, ~set, ~reset);

// Behavioral block, initial
initial
begin
  $monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
  set = 0; reset = 0;
  #5 reset = 1;
  #5 reset = 0;
  #5 set = 1;
end

// endmodule statement
endmodule
```







54

## Contd..

- Notice the following characteristics about the modules defined above.
- In the SR latch definition above , notice that all components described in Figure 4-1 need not be present in a module.
- We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial ) .
- However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block ( initial ) , and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.
- Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.





## 4.2. Ports:

- Ports provide the interface by which a module can communicate with its environment.
- For example, the input/output pins of an IC chip are its ports.
- The environment can interact with the module only through its ports.
- The internals of the module are not visible to the environment.
- This provides a very powerful flexibility to the designer.
- The internals of the module can be changed without affecting the environment as long as the interface is not modified.
- Ports are also referred to as terminals.







55

## 4.2.1. List of Ports: Contd..

- A module definition contains an optional list of ports.
- If the module does not exchange any signals with the environment, there are no ports in the list.
- Consider a 4-bit full adder that is instantiated inside a top-level module Top.
- The diagram for the input/ output ports is shown in Figure 4-3.

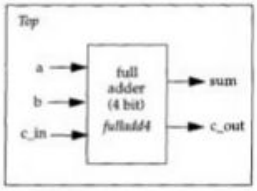



*Figure 4-3 I/O Ports for Top and Full Adder*





- Notice that in the above figure, the module Top is a top-level module. Contd..
- The module fulladd4 is instantiated below Top.
- The module fulladd4 takes input on ports a, b, and c-in and produces an output on ports sum and c-out. Thus, module fulladd4 performs an addition for its environment.
- The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment.
- Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in Example 4-2.

*Example 4-2 List of Ports*

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```







56

## 4.2. 2. Port Declaration: Contd..

- All ports in the list of ports must be declared in the module. Ports can be declared as follows:

| Verilog Keyword | Type of Port |
|---|---|
| `input` | Input port |
| `output` | Output port |
| `inout` | Bidirectional port |

- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.
- Thus, for the example of the fulladd4 in Example 4-2, the port declarations will be as shown in Example 4-3.





## Contd.. *Example 4-3 Port Declarations*

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_out;

input [3:0] a, b;
input c_in;
//End port declarations section
…
<module internals>
…
endmodule
```

- Note that all port declarations are implicitly declared as wire in Verilog.
- Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout. Input or inout ports are normally declared as wires.
- However, if output ports hold their value, they must be declared as reg.







57

## Contd..

- For example, in the definition of DFF, in Example 2-5, we wanted the output q to retain its value until the next clock edge. The port declarations for DFF will look as shown in Example 4-4.

*Example 4-4* *Port Declarations for DFF*

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value; therefore it is declared as reg.
input d, clk, reset;
-
-
endmodule
```

- Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.





## 4.2.3. Port Connection Rules: Contd..

- One can visualize a port as consisting of two units, one unit that is internal to the module another that is external to the module.
- The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules.
- The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure 4-4

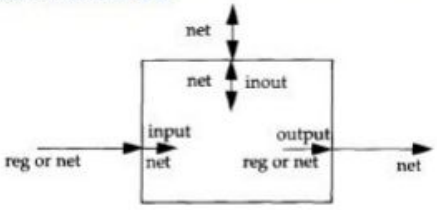



*Figure 4-4* *Port Connection Rules*







58

## Inputs: Contd..

- Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

## Outputs:

- Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

## Inouts:

- Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.





## Width matching: Contd..

- It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.

## Unconnected ports:

- Verilog allows ports to remain unconnected.
- For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module as shown below.

```
fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected
```

- Example of illegal port connection. To illustrate port connection rules, assume that the module fulladd4 in Example 4-3 is instantiated in the stimulus block Top.
- An example of an illegal port connection is shown in Example 4-5.







59

*Example 4-5* *Illegal Port Connection* Contd..

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa0
    fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
    //Illegal connection because output port sum in module fulladd4
    //is connected to a register variable SUM in module Top.

    .
    .
    <stimulus>
    .
    .
endmodule
```

- This problem is rectified if the variable SUM is declared as a net (wire).
- A similar problem would occur if an input port were declared as a rep.





## 4.2.4. Connecting Ports to External Signals: Contd..

- There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition.
- The two methods cannot be mixed.

### Connecting by ordered list:

- Connecting by ordered list is the most intuitive method for most beginners. The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition. Once again, consider the module fulladd4 defined in Example 4-3.
- To connect signals in module Top by ordered list, the Verilog code is shown in Example 4-6. Notice that the external signals SUM, C-OUT, A, B, and CJN appear in exactly the same order as the ports sum, c-out, a, b, and c-in in module definition of fulladd4.







60

Contd..

*Example 4-6 Connection by Ordered List*

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

   //Instantiate fulladd4, call it fa_ordered.
   //Signals are connected to ports in order (by position)
   fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
   …
   <stimulus>
   …
endmodule


module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
   …
   <module internals>
   …
endmodule
```





## Connecting ports by name:

Contd..

- For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error prone.
- Verilog provides the capability to connect external signals to ports by the port names, rather than by position.
- We could connect the ports by name in Example 4-6 above by instantiating the module fulladd4, as follows.
- Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),
.a(A),);
```







61

## Contd..

- Note that only those ports that are to be connected to external signals must be specified in port connection by name.
- Unconnected ports can be dropped.
- For example, if the port c-out were to be kept unconnected, the instantiation of filladd4 would look as follows.
- The port c-out is simply dropped from the port list.

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

- Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.





## 4.3. Hierarchical Names:

- We described earlier that Verilog supports a hierarchical design methodology. Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy.
- Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (" .") for each level of hierarchy.
- Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier. The top-level module is called the root module because it is not instantiated anywhere.
- It is the starting point. To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.







62

## Contd..

- To clarify this process, let us consider the simulation of SR latch in Example 4-1.
- The design hierarchy is shown in Figure 4-5.

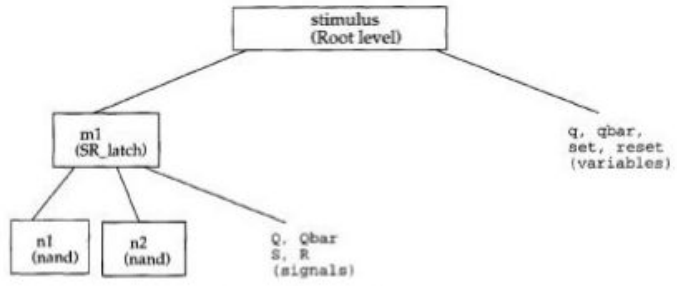



*Figure 4-5 Design Hierarchy for SR Latch Simulation*





## Contd..

- For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module.
- The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates ml, which is a module of type SR-latch.
- The module m1 instantiates nand gates nl and 722. Q, Qbar, S, and R are port signals in instance ml. Hierarchical name referencing assigns a unique name to each identifier.
- To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module. Example 4-7 shows hierarchical names for all identifiers in the above simulation.
- Notice that there is a dot ( .) for each level of hierarchy from the root module to the desired identifier.







63

## Contd..

*Example 4-7 Hierarchical Names*

```
stimulus                    stimulus.q
stimulus.qbar               stimulus.set
stimulus.reset              stimulus.m1
stimulus.m1.Q               stimulus.m1.Qbar
stimulus.m1.S               stimulus.m1.R
stimulus.n1                 stimulus.n2
```

- Each identifier in the design is uniquely specified by its hierarchical path name.
- To display the level of hierarchy, use the special character %m in the $display task.
- See Table 3-4, String Format Specifications, for details.





## Summary:

- Module definitions contain various components. Keywords module and endmodule are mandatory. Other components-port list, port declarations, variable and signal declarations, dataflow statements, behavioral blocks, lower-level module instantiations, and tasks or functions-are optional and can be added as needed.
- Ports provide the module with a means to communicate with other modules or its environment. A module can have a port list. Ports in the port list must be declared as input, output, or inout. When instantiating a module, port connection rules are enforced by the Verilog simulator.
- Ports can be connected by name or by ordered list. Each identifier in the design has a unique hierarchical name. Hierarchical names allow us to address any identifier in the design from any other level of hierarchy in the design.







64


Digital Design and Implementation using Verilog HDL
28 August 2021

## 5.1 Gate Types

- A logic circuit can be designed by use of logic gates.
- Verilog supports basic logic gates as predefined *primitives*.
- These primitives are instantiated like *modules* except that they are predefined in Verilog and do not need a module definition.
- All logic circuits can be designed by using basic gates.
- There are two classes of basic gates: *and/or gates and buf/not gates.*





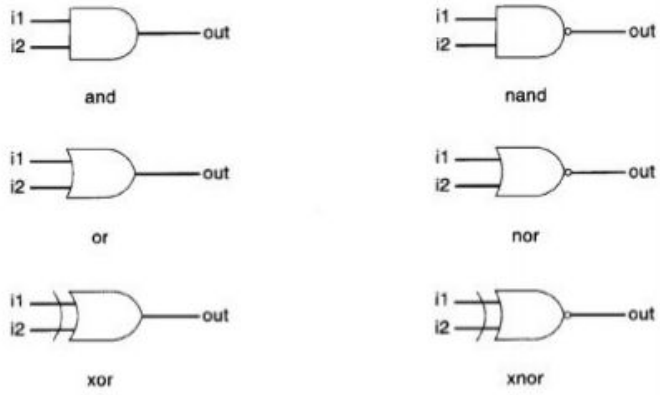
## 5.1.1 and/or Gates

- **and/or** gates have **one scalar output** and **multiple scalar inputs**.
- *The* **first terminal** in the list of gate terminals is an **output** and the **other** terminals are **inputs**.
- The output of a gate is evaluated as soon as one of the inputs changes.
- The **and/or** gates available in Verilog are shown below

```
and     or      xor
nand    nor     xnor
```

- The corresponding logic symbols for these gates are shown in Figure 5-1.
- We consider gates with two inputs.
- The output terminal is denoted by *out*. *Input* terminals are denoted by **il** and **i2**.

## Contd..




*Fieure 5-1 Basic Gates*





## Contd..

- These gates are instantiated to build logic circuits in Verilog.
- Examples of gate instantiations are shown below.
- In Example 5-1, for all instances, *OUT is* connected to the output out, and *IN1 and IN2 are connected to the two inputs* **il** and i2 of the gate primitives.
- Note that the instance name does not need to be specified for primitives.
- This lets the designer instantiate hundreds of gates without giving them a name.
- More than two inputs can be specified in a gate instantiation.
- Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation (see Example 5-1).
- Verilog automatically instantiates the appropriate gate.

## Contd..

*Example 5-1* *Gate Instantiation of And/Or Gates*

```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation
```





## Contd..

- The truth tables for these gates define how outputs for the gates are computed from the inputs.
- Truth tables are defined assuming two inputs.
- The truth tables for these gates are shown in Table 5-1.
- Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

| and | i1: 0 | 1 | x | z |
|---|---|---|---|---|
| i2: 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

| nand | i1: 0 | 1 | x | z |
|---|---|---|---|---|
| i2: 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| x | 1 | x | x | x |
| z | 1 | x | x | x |

## Contd..

| or | i1=0 | i1=1 | i1=x | i1=z |
|---|---|---|---|---|
| i2=0 | 0 | 1 | x | x |
| i2=1 | 1 | 1 | 1 | 1 |
| i2=x | x | 1 | x | x |
| i2=z | x | 1 | x | x |

| nor | i1=0 | i1=1 | i1=x | i1=z |
|---|---|---|---|---|
| i2=0 | 1 | 0 | x | x |
| i2=1 | 0 | 0 | 0 | 0 |
| i2=x | x | 0 | x | x |
| i2=z | x | 0 | x | x |

| xor | i1=0 | i1=1 | i1=x | i1=z |
|---|---|---|---|---|
| i2=0 | 0 | 1 | x | x |
| i2=1 | 1 | 0 | x | x |
| i2=x | x | x | x | x |
| i2=z | x | x | x | x |

| xnor | i1=0 | i1=1 | i1=x | i1=z |
|---|---|---|---|---|
| i2=0 | 1 | 0 | x | x |
| i2=1 | 0 | 1 | x | x |
| i2=x | x | x | x | x |
| i2=z | x | x | x | x |





## 5.1.2 buf/not Gates

- *Buf/not gates have one scalar input and one or more scalar outputs.*
- *The last* terminal in the port list is connected to the input.
- Other terminals are connected to the outputs.
- We will discuss gates that have one input and one output.
- Two basic *buf/not gate primitives are provided in Verilog.*

```
buf        not
```

The symbols for these logic gates are shown in Figure 5-2.

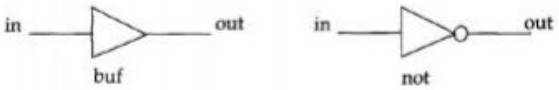



*Figure 5-2 Buf and Not gates*

## Contd..

- These gates are instantiated in Verilog as shown Example 5-2.
- Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.

*Example 5-2* *Gate Instantiations of Buf/Not Gates*

```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```





## Contd..

- The truth tables for these gates are very simple.
- Truth tables for gates with one input and one output are shown in Table 5-2.

*Table 5-2* *Truth Tables for Buf/Not gates*

| buf | in | out |
|---|---|---|
| | 0 | 0 |
| | 1 | 1 |
| | x | x |
| | z | x |

| not | in | out |
|---|---|---|
| | 0 | 1 |
| | 1 | 0 |
| | x | x |
| | z | x |

## bufif/notif

- Gates with an additional control signal on **buf and not gates are also available.**

```
bufif1        notif1
bufif0        notif0
```

- These gates propagate only if their control signal is asserted.
- They propagate **z if** their control signal is deasserted.
- Symbols for *bufiflnotif are shown in Figure 5-3.*







## Contd..

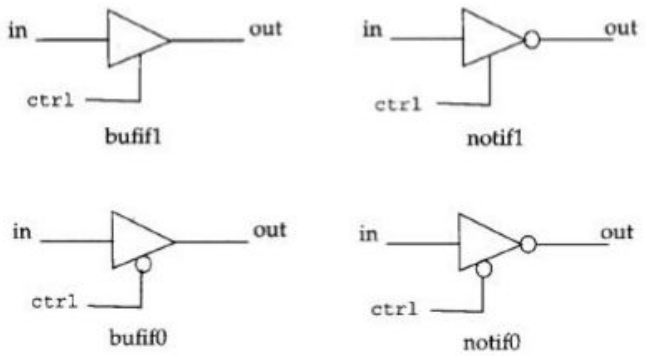



*Figure 5-3 Gates Bufif and Notif*

- The truth tables for these gates are shown in Table 5-3. **Contd..**

*Table 5-3 Truth Tables for Bufif/Notif Gates*

| bufif1 in \ ctrl | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | z | 0 | L | L |
| 1 | z | 1 | H | H |
| x | z | x | x | x |
| z | z | x | x | x |

| bufif0 in \ ctrl | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | z | L | L |
| 1 | 1 | z | H | H |
| x | x | z | x | x |
| z | x | z | x | x |

| notif1 in \ ctrl | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | z | 1 | H | H |
| 1 | z | 0 | L | L |
| x | z | x | x | x |
| z | z | x | x | x |

| notif0 in \ ctrl | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | z | H | H |
| 1 | 0 | z | L | L |
| x | x | z | x | x |
| z | x | z | x | x |





## Contd..

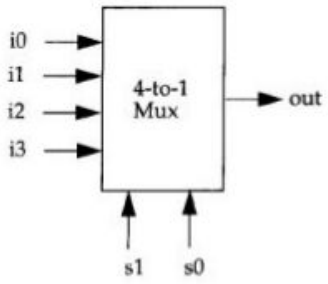
- These gates are used when a signal is to be driven only when the control signal is asserted.
- Such a situation is applicable when multiple drivers drive the signal.
- These drivers are designed to drive the signal on mutually exclusive control signals.
- Example 5-3 shows examples of instantiation of bufif and notif gates.

*Example 5-3 Gate Instantiations of Bufif/Notif Gates*

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);

//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

## 5.1.3 Examples

- **Gate-level multiplexer:**
- We will assume for this example that signals *sl and SO do not get the value X or z.*




| s1 | s0 | out |
|---|---|---|
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |

*Figure 5-4 4-to-1 Multiplexer*





## Contd..

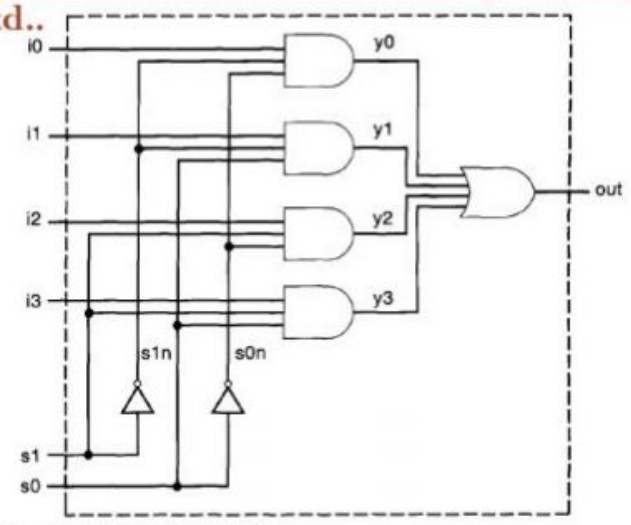



*Figure 5-5 Logic Diagram for Multiplexer*

## Contd..

Example 5-4 Verilog Description of Multiplexer

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the 1/0 diagram.
module mux4-to-l (out, i0, i1, i2, i3, s1, S0);
// Port declarations from the 1/0 diagram
output out;
input i0, i1, i2, i3;
input sl, S0;
// Internal wire declarations
wire sln, son;
wire y0, y1, y2, y3;
// Gate instantiations
// Create sln and son signals.
not (s1n, s1) ; not (son, S0);
```





## Contd..

```
// 3-input and gates instantiated
and (y0, i0, s1n, son); and (y1, i1, s1n, so); and (y2, i2, s1, son); and (y3,
i3, s1, S0);
// 4-input or gate instantiated
or (out, y0, yl, y2, y3)
endmodule
```

*Example 5-5 Stimulus for Multiplexer*

```
// Define the stimulus module (no ports)
module stimulus;
// Declare variables to be connected // to inputs
reg IN0, IN1, IN2, IN3;  reg S1, S0;
// Declare output wire
wire OUTPUT;
```

Contd..

```
// Instantiate the multiplexer
mux4-to-1 mymux (OUTPUT, IN0, IN1, IN2, IN3, S1, S0) ;
// define the stimulus module (no ports)  // Stimulate the inputs
initial
begin
// set input lines
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#l $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",  IN0, IN1,
IN2, I N3);
// choose IN0
S1 = 0; so = 0;  #l $display("S1 = %b, S0 = %b, OUTPUT = %b \n",
S1, S0, OUTPUT);
// choose IN1
S1 = 0; S0 = 1;  #l $display("S1 = %b, S0 = %b, OUTPUT = %b \n",
S1, S0, OUTPUT);
```





Contd..

```
// choose IN2
S1 = 1; S0 = 0;  #l $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1,
S0, OUTPUT);
// choose IN3
S1 = 1; S0 = 1;  #l $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1,
S0, OUTPUT);
end
endmodule
```

The output of the simulation is shown below.
Each combination of the select signals is tested.

```
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0
```

## 4-bit full adder

$$sum = (a \oplus b \oplus cin)$$

$$cout = (a \cdot b) + cin \cdot (a \oplus b)$$

The logic diagram for a 1-bit full adder is shown in Figure 5-6.

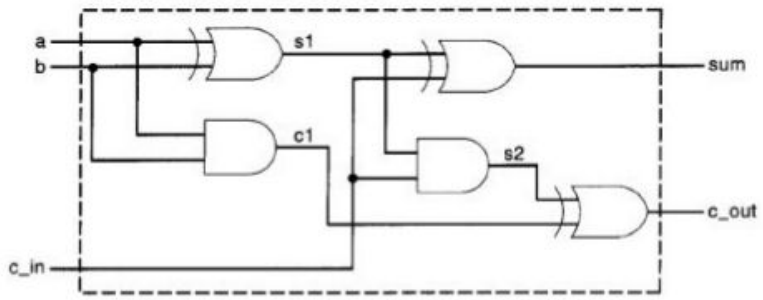



*Figure 5-6 1-bit Full Adder*





## Contd..

Example 5-6 Verilog Description for 1-bit Full Adder

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in) ;
// I/O port declarations
output sum, c_out;
input a, b, c_in;
// Internal nets
wire s1, c1, c2;
// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);
xor (sum, s1, c_in) ;
and (c2, s1, c_in);
or (c_out, c2, c1);
endmodule
```

## Contd..

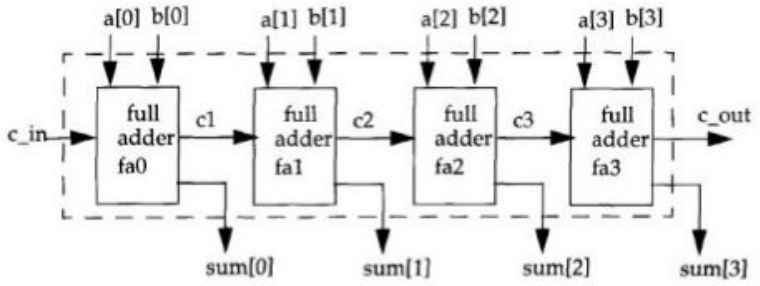



*Figure 5-7 4-bit Full Adder*





Example 5-7 Verilog Description for 4-bit Full Adder

## Contd..

```
// Define a 4-bit full adder
module fulladd4(sum, c-out, a, b, c-in);
// I/O port declarations
output [3:01 sum; output c-out;
input [3 : 01 a, b;
input c-in;
// Internal nets
wire cl, c2, c3;
// Instantiate four l-bit full adders
fulladd fa0 (sum[Ol, cl, a[Ol, b[Ol, c-in);
fulladd fa1(sum[ll, c2, a[ll, b[ll, cl);
fulladd fa2(surn[21, c3, a[21, b[21, c2);
fulladd fa3 (sumt31, c-out, at31, b[31, c3);
endmodule
```

Example 5-8 Stimulus for 4-bit Full Adder

Contd..

```
// Define the stimulus (top level module)
module stimulus;
// Set up variables
reg [3:0] A, B;      reg C-IN;
wire [3:01 SUM;  wire C-OUT;
// Instantiate the 4-bit full adder. call it FA1-4
fulladd4 FA1_4(SUM, C-OUT, A, B, C-IN) ;
// Setup the monitorins for the sisnal values
initial
begin
$monitor($time," A= %b, B=%b, C-IN= %b, --- C-OUT= %b, SUM=
%b\nl', A, B, C-IN, C-OUT, SUM);
end
```







Contd..

```
// Stimulate inputs
initial  begin
A = 4'dO; B = 4'dO; C-IN = l'bO; #5 A = 4'd3; B = 4'd4;
#5 A = 4'd2; B = 4'd5;  #5 A = 4'd9; B = 4'd9;
#5 A = 4'dlO; B = 4'd15; #5 A = 4'dlO; B = 4'd5; C-IN = l'bl;
end
endmodule
```

The output of the simulation is shown below.

```
0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000
```

## 5.2 Gate Delays

- In real circuits, logic gates have delays associated with them.
- Gate delays allow the Verilog user to specify delays through the logic circuits.
- Pin-to-pin delays can also be specified in Verilog.

### 5.2.1 Rise, Fall, and Turn-off Delays

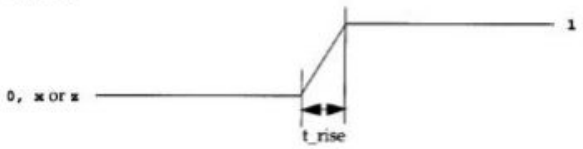
- There are three types of delays from the inputs to the output of a primitive gate.





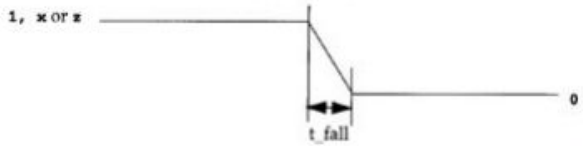
## Rise delay Contd..

- The rise delay is associated with a gate output transition to a **1 from another** value.




## Fall delay

- The fall delay is associated with a gate output transition to a 0 from another value.



## Turn-off delay

Contd..

- The turn-off delay is associated with a gate output transition to the high impedance value **(Z) from another value.**
- If the value changes to X, the minimum of the three delays is considered.

- Three types of delay specifications are allowed.
- If only one delay is specified, this value is used for all transitions.
- If two delays are specified, they refer to the rise and fall delay values.
- The turn-off delay is the minimum of the two delays.
- If all three delays are specified, they refer to rise, fall, and turn-off delay values.
- If no delays are specified, the default value is zero. Examples of delay specification are shown in example 5-9.





Contd..

*Example 5-9* *Types of Delay Specification*

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions
and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6
bufif0 #(3,4,5) b1 (out, in, control); //Rise = 3, Fall = 4, Turn-off = 5
```

## 5.2.2 Min/Typ/Max Values

### Min value

- The min value is the minimum delay value that the designer expects the gate to have.

### Typ val

- The typ value is the typical delay value that the designer expects the gate to have.

### Max value

- The max value is the maximum delay value that the designer expects the gate to have.

- Min, typ, or max values can be chosen at Verilog run time.
- Method of choosing a min/typ/max value may vary for different simulators or operating systems.





## Contd..

- This allows the designers the flexibility of building three delay values for each transition into their design.
- The designer can experiment with delay values without modifying the design.
- Example 5-1 0 Min, Max and Typical Delay Values

```
// One delay
// if +mindelays, delay= 4
// if +typdelays, delay= 5
// if +maxdelays, delay= 6
and #(4:5:6) al(out, il, i2);
```

## Contd..

- //Two delays

```
// if +mindelays, rise= 3, fall= 5, turn-off = rnin
// if +typdelays, rise= 4, fall= 6, turn-off = rnin
// if +maxdelays, rise= 5, fall= 7, turn-off = rnin
and #(3:4:5, 5:6:7) a2(out, il, i2);
```

- //Three delays

```
// if +mindelays, rise= 2 fall= 3 turn-off = 4
// if +typdelays, rise= 3 fall= 4 turn-off = 5
// if +maxdelays, rise= 4 fall= 5 turn-off = 6
and #(2:3:4, 3:4:5, 4:5:6) a3(out, il,i2);
```







## Contd..

- Examples of invoking the Verilog-XL simulator with the command-line options are shown below.
- Assume that the module with delays is declared in the file *test.v*.

```
//invoke simulation with maximum delay
> verilog test.v +maxdelays

//invoke simulation with minimum delay
```

```
> verilog test.v +mindelays

//invoke simulation with typical delay
> verilog test.v +typdelays
```

## 5.2.3 Delay Example

- Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits.
- A simple module called D implements the following logic equations:
- out = (a.b)+c
- The gate-level implementation is shown in Module D (Figure 5-8). The module contains two gates with delays of 5 and 4 time units.

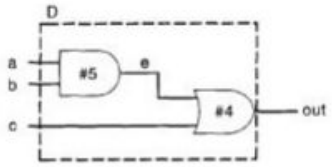



*Figure 5-8 Module D*





## Contd..

- The module D is defined in Verilog as shown in Example 5-11.

Example 5-11 Verilog Definition for Module D with Delay

```
// Define a simple combination module called D
module D (out, a, b, c);
// I/Oport declarations
output out;
input a,b,c;
// Internal nets
wire e;
// Instantiate primitive gates to build the circuit
and # (5) a1 (e, a, b) ; //Delay of 5 on gate a1
or #(4) o1(out, e,c); //Delay of 4 on gate o1
endmodule
```

- This module is tested by the stimulus file shown in Example 5-12. Contd..

Example 5-12 Stimulus for Module D with Delay

```
// Stimulus (top-level module)
module stimulus;
// Declare variables
reg A, B, C; wire OUT;
// Instantiate the module D
D dl( OUT, A, B, C);
// Stimulate the inputs. Finish the simulation at 40 time units
initial
begin
A= l'b0; B= l'b0; C= l'b0; #l0 A= l'bl; B= l'bl; C= l'bl;
#l0 A= l'bl; B= l'bO; C= l'bO; #20 $finish;
end
endmodule
```





## Contd..

- The waveforms from the simulation are shown in Figure 5-9 to illustrate the effect of specifying delays on gates.
- The waveforms are not drawn to scale.
- However, simulation time at each transition is specified below the transition.

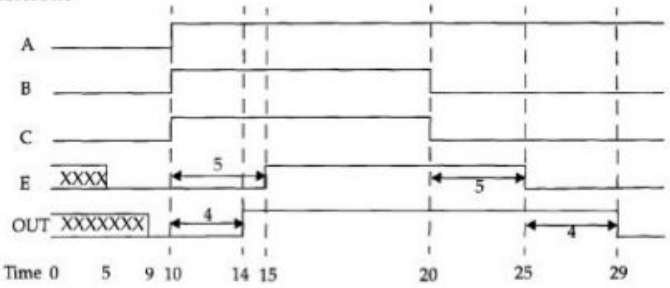



*Figure 5-9 Waveforms for Delay Simulation*

## Contd..

- 1. The outputs E and OUT are initially unknown.
- 2. At time 10, after A, B, and C all transition to I, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
- 3. At time 20, B and C transition to 0. E changes value to o after 5 time units, and OUT transitions to 0, 4 time units after E changes.

- It is a useful exercise to understand how the timing for each transition in the above waveform corresponds to the gate delays shown in Module D.





## Summary

- Basic types of gates are and, or, xor, buf, and not. Each gate has a logic symbol, truth table, and a corresponding Verilog primitive. Primitives are instantiated like modules except that they are predefined in Verilog. Output of a gate is evaluated as soon as one of its inputs changes.
- For gate-level design, start with the logic diagram, write the Verilog description for the logic by using gate primitives, provide stimulus, and look at the output. Two design examples, a 4-to-1 multiplexer and a 4-bit full adder, were discussed. Each step of the design process was explained.
- Three types of delays are associated with gates, rise, fall, and turn-off. Verilog allows specification of one, two, or three delays for each gate. Values of rise, fall, and turn-off delays are computed by Verilog, based on the one, two, or three delays specified

## Contd..

- For each type of delay, a minimum, typical, and maximum value can be specified. The user can choose which value to apply at simulation time. This provides the flexibility to experiment with three delay values without changing the Verilog code.
- The effect of propagation delay on waveforms was explained by the simple, two-gate logic example. For each gate with a delay of t, the output changes t time units after any of the inputs change.

## Objectives

After completing this chapter, you will be able to:

- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements.
- Define expressions, operators, and operands.
- List operator types for all possible operations.
- Use dataflow constructs to model practical digital circuits in Verilog HDL.





## Continuous Assignments

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a **net**.
- A continuous assignment replaces gates in the description of the circuit and describes the circuit at a **higher level of abstraction**.
- A continuous assignment statement starts with the keyword **assign**.
- The syntax of an **assign** statement is as follows.

```
//Syntax of assign statement in the simplest form
<continuous_assign>
      ::= assign <drive_strength>?<delay>? <list_of_assignments>;
```

## Contd..

- Notice that **drive strength is optional** and can be specified in terms of strength Levels Value Set.
- The **default** value for drive strength is **strong1** and **strong0**.
- The **delay value is also optional** and can be used to specify delay on the assign statement.
- This is like specifying delays for gates.





## Contd..

Continuous assignments have the following characteristics.

1. The **left hand side** of an assignment must always be a **scalar** or **vector net** or a concatenation of scalar and vector nets. It **cannot** be a scalar or vector **Register**.

2. Continuous assignments are **always active**. The assignment expression is evaluated as soon as one of the **right-hand-side operands changes** and the **value is assigned to the left-hand-side net**.

3. The operands on the **right-hand side can be registers or nets or function calls**. Registers or nets can be scalars or vectors.

4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits

## Contd..

Examples of continuous assignments are shown below. Operators such as &, *, | , {,} and + used in the examples.

At this point, concentrate on how the assign statements are specified.

*Example 6-1* *Examples of Continuous Assignment*

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```





## 6.1.1 Implicit Continuous Assignment

- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.
- There can be only one implicit declaration assignment per net because a net is declared only once.
- In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment
wire out = in1 & in2;
```

## 6.2 Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.
- Three ways of specifying delays in continuous assignment statements are *regular assignment delay, implicit continuous assignment delay, and net declaration delay.*

### 6.2.1 Regular Assignment Delay

- The first method is to assign a delay value in a continuous assignment statement.
- The delay value is specified after the keyword **assign.**





## Contd..

- Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.
- If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered.
- This property is called inertial delay.
- An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

## Contd..

- The waveform in Figure 6-1 is generated by simulating the above assign statement. It shows the delay on signal out. Note the following changes.
- 1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- 2. When in1 goes low at 60, out changes to low at 70.
- 3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- 4. Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.







C

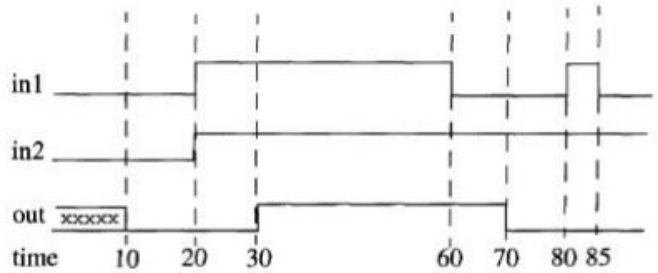



*Figure 6-1 Delays*

Inertial delays also apply to gate delays, discussed in *Gate-Level Modeling*.

## Contd..

### 6.2.2 Implicit Continuous Assignment Delay

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay
wire #10 out = in1 & in2;

//same as
wire out;
assign #10 out = in1 & in2;
```

- The declaration above has the same effect as defining a *wire out and declaring a* continuous assignment on *out*.







## Contd..

### 6.2.3 Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net *out, then any* value change applied to the net *out is delayed accordingly.*
- *Net declaration delays* can also be used in gate-level modeling.

```
//Net Delays
wire # 10 out;
assign out = in1 & in2;

//The above statement has the same effect as the following.
wire out;
assign #10 out = in1 & in2;
```

## 6.3 Expressions, Operators, and Operands

- Dataflow modeling describes the design in terms of expressions instead of primitive gates. *Expressions, operators, and operands form the basis of dataflow* modeling.

### 6.3.1 Expressions

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```





## 6.3.2 Operands

- Operands can be any one of the data types defined in *Data Types*. Some constructs will take only certain types of operands. Operands can be *constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net* or a vector register), *part-select (selected bits of the vector net or register vector), memories or function calls.*

```
integer count, final_count;
final_count = count + 1;//count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are
                                //part-select register operands

reg ret_value;
ret_value = calculate_parity(A, B);//calculate_parity is a
                                //function type operand
```

## 6.3.3 Operators

- Operators act on the operands to produce desired results.
- Verilog provides various types of operators.
- Operator types are discussed in detail in Section 6.4, *Operator Types*.

```
d1 && d2 // && is an operator on operands d1 and d2
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```





## 6.4 Operator Types

- Verilog provides many different operator types.
- Operators can be *arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional.*
- *Some* of these operators are similar to the operators used in the C programming language.
- Each operator type is denoted by a symbol. Table 6-1 shows the complete listing of operator symbols classified by category.

## Contd..

*Table 6-1 Operator Types and Symbols*

| Operator Type | Operator Symbol | Operation Performed | Number of Operands |
|---|---|---|---|
| Arithmetic | * | multiply | two |
| | / | divide | two |
| | + | add | two |
| | - | subtract | two |
| | % | modulus | two |
| Logical | ! | logical negation | one |
| | && | logical and | two |
| | \|\| | logical or | two |
| Relational | > | greater than | two |
| | < | less than | two |
| | >= | greater than or equal | two |
| | <= | less than or equal | two |
| Equality | == | equality | two |
| | != | inequality | two |
| | === | case equality | two |
| | !== | case inequality | two |

19



## Contd..

| Operator Type | Operator Symbol | Operation Performed | Number of Operands |
|---|---|---|---|
| Bitwise | ~ | bitwise negation | one |
| | & | bitwise and | two |
| | \| | bitwise or | two |
| | ^ | bitwise xor | two |
| | ^~ or ~^ | bitwise xnor | two |
| Reduction | & | reduction and | one |
| | ~& | reduction nand | one |
| | \| | reduction or | one |
| | ~\| | reduction nor | one |
| | ^ | reduction xor | one |
| | ^~ or ~^ | reduction xnor | one |
| Shift | >> | Right shift | two |
| | << | Left shift | two |
| Concatenation | { } | Concatenation | any number |
| Replication | { { } } | Replication | any number |
| Conditional | ?: | Conditional | three |

20

## 6.4.1 Arithmetic Operators

- There are two types of arithmetic operators: binary and unary.

**Binary operators**

- Binary arithmetic operators are *multiply (*), divide (/), add (+), subfract (-) and modulus (%). Binary operators take two operands.*

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; // D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E.Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
```





## Contd..

- If any operand bit has a value X, then the result of the entire expression is X. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

- Modulus operators produce the *remainder from the division of two numbers. They* operate similarly to the modulus operator in the C programming language.

```
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first operand
```

## Contd..

Unary operators

- The operators + and - can also work as *unary operators. They are used to specify* the positive or negative sign of the operand.
- Unary + or - operators have higher precedence than the binary + or – operators.

```
-4 // Negative 4
+5 // Positive 5
```





## Contd..

- Negative numbers are represented as 2's complement internally in Verilog.
- It is advisable to use negative numbers only of the type integer or real in expressions.
- Designers should avoid negative numbers of the type <*sss*> '<*base*> <*nnn*> *in* expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

```
//Advisable to use integer or real numbers
-10 / 5// Evaluates to -2

//Do not use numbers of type <sss> '<base> <nnn>
-'d10 / 5// Is equivalent (2's complement of 10)/5 = (2^32 - 10)/5
// where 32 is the default machine word width.
// This evaluates to an incorrect and unexpected result
```

## 6.4.2 Logical Operators

- Logical operators are *logical-and (&&), logical-or (| | ) and logical-not (! ).*
- *Operators* && and | | are binary operators.
- Operator ! is a unary operator. Logical operators follow these conditions:

1. Logical operators always evaluate to a l-bit value, o (false), 1 (true), or *X* (ambiguous).
2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical o (false condition). If any operand bit is X or z, it is equivalent to X (ambiguous condition) and is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.





## Contd..

- Use of parentheses to group logical operations is highly recommended to improve readability.
- Also, the user does not have to remember the precedence of operators.

```
// Logical operations
A = 3; B = 0;
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A// Evaluates to 0. Equivalent to not(logical-1)
!B// Evaluates to 1. Equivalent to not(logical-0)

// Unknowns
A = 2'b0x; B = 2'b10;
A && B // Evaluates to x. Equivalent to (x && logical 1)

// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3
are true.
 // Evaluates to 0 if either is false.
```

## 6.4.3 Relational Operators

- Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=) , and less-than-or-equal-to (<=).
- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and o if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value X.
- These operators function exactly as the corresponding operators in the C programming language.







## Contd..

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
```

```
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

## 6.4.4 Equality Operators

- Equality operators are *logical equality (==), logical inequality (!=), case equality (===),* and *case inequality (!==).*
- *When used in an expression, equality operators return* logical value 1 if true, 0 if false.
- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table 6-2 lists the operators.

*Table 6-2 Equality Operators*

| Expression | Description | Possible Logical Value |
|---|---|---|
| a == b | a equal to b, result unknown if x or z in a or b | 0, 1, x |
| a != b | a not equal to b, result unknown if x or z in a or b | 0, 1, x |
| a === b | a equal to b, including x and z | 0, 1 |
| a !== b | a not equal to b, including x and z | 0, 1 |

29



## Contd..

- It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==).
- The logical equality operators (==, !=) will yield an X if either operand has X or Z in its bits.
- However, the case equality operators ( ===, !== ) compare both operands bit by bit and compare all bits, including X and Z.
- The result is 1 if the operands match exactly, including X and Z bits.
- The result is 0 if the operands do not match exactly.
- Case equality operators never result in an X.

11 August 2021

30

## Contd..

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```





## 6.4.5 Bitwise Operators

- ABitwise operators are *negation (~), and(&), or (|), xor (^), xnor (^~, ~^).*
- *Bitwise* operators perform a bit-by-bit operation on two operands.
- They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
- If one operand is shorter than the other, it will be bit extended with zeros to match the length of the longer operand.
- Logic tables for the bit-by-bit computation are shown in Table 6-3. A *z is treated as an X in a bitwise operation.*
- The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

## Contd..

*Table 6-3 Truth Tables for Bitwise Operators*

| bitwise and | 0 | 1 | x |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x |
| x | 0 | x | x |

| bitwise or | 0 | 1 | x |
|---|---|---|---|
| 0 | 0 | 1 | x |
| 1 | 1 | 1 | 1 |
| x | x | 1 | x |

| bitwise xor | 0 | 1 | x |
|---|---|---|---|
| 0 | 0 | 1 | x |
| 1 | 1 | 0 | x |
| x | x | x | x |

| bitwise xnor | 0 | 1 | x |
|---|---|---|---|
| 0 | 1 | 0 | x |
| 1 | 0 | 1 | x |
| x | x | x | x |

| bitwise negation | Result |
|---|---|
| 0 | 1 |
| 1 | 0 |
| x | x |

33



## Contd..

- Examples of bitwise operators are shown below.

```
// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1

~X     // Negation. Result is 4'b0101
X & Y  // Bitwise and. Result is 4'b1000
X | Y  // Bitwise or. Result is 4'b1111
X ^ Y  // Bitwise xor. Result is 4'b0111
X ^~ Y // Bitwise xnor. Result is 4'b1000
X & Z  // Result is 4'b10x0
```

- It is important to distinguish bitwise operators ~, &, *and* | *from logical operators* !, &&, ||. *Logical operators always yield a logical value of 0, 1, X, whereas bitwise* operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

11 August 2021

34

## Contd..

```
// X = 4'b1010, Y = 4'b0000

X | Y // bitwise operation. Result is 4'b1010
X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

### 6.4.6 Reduction Operators

- Reduction operators are *and (&), nand (~&), or ( | ), **nor (~| ), xor (^), and xnor (~^, ^~)**. Reduction operators take only one operand. Reduction operators perform a* bitwise operation on a single vector operand and yield a 1-bit result. The logic tables for the operators are the same as shown in Section 6.4.5, *Bitwise Operators.*







## Contd..

- The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand.
- Reduction operators work bit by bit from right to left.
- Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

```
// X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.
```

## Contd..

- The use of a similar set of symbols for logical (!, &&, || ), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially.
- The difference lies in the number of operands each operator takes and also the value of result computed.







## 6.4.7 Shift Operators

- Shift operators are right shift ( >>) and left shift (<<). These operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around.

```
// X = 4'b1100

Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
```

- Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

## 6.4.8 Concatenation Operator

- The *concatenation operator ( {, }) provides a mechanism to append multiple* operands.
- The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
- Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

39



## 6.4.9 Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ( { }).

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```



40

## 6.4.10 Conditional Operator

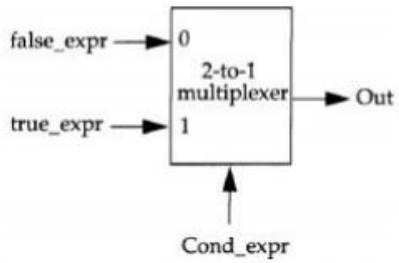
- The conditional operator(? :) takes three operands.
- Usage: condition_expr ? true_expr : false_expr ;
- The condition expression (condition_expr) is first evaluated.
- If the result is true (logical 1), then the true-expr is evaluated. If the result is false (logical o), then the false-expr is evaluated.
- If the result is X (ambiguous), then both true-expr and false-expr are evaluated and their results are compared, bit by bit, to return for each bit position an X if the bits are different and the value of the bits if they are the same.





## Contd..

- The action of a conditional operator is similar to a multiplexer.
- Alternately, it can be compared to the if-else expression.




- Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control

## Contd..

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

- Conditional operations can be nested. Each *true-expr or false-expr can itself be a* conditional operation.
- In the example that follows, convince yourself that ***(A==3)*** and control are the two select signals of 4-to-1 multiplexer with n, m, y, *x as the* inputs and out as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```





## 6.4.11 Operator Precedence

- If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence.
- Operators listed in Table 6-4 are in order from highest precedence to lowest precedence.
- It is recommended that parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.

## Contd..

*Table 6-4 Operator Precedence*

| Operators | Operator Symbols | Precedence |
|---|---|---|
| Unary | + - ! ~ | Highest precedence |
| Multiply, Divide, Modulus | * / % | |
| Add, Subtract | + - | |
| Shift | << >> | |
| Relational | < <= > >= | |
| Equality | == != === !== | |
| Reduction | &, ~&<br>^ ^~<br>\|, ~\| | |
| Logical | &&<br>\|\| | |
| Conditional | ?: | Lowest precedence |







## 6.5 Examples

### 6.5.1 4-to-1 Multiplexer Method 1: logic equation

*Example 6-2 4-to-1 Multiplexer, Using Logic Equations*

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =   (~s1 & ~s0 & i0)|
               (~s1 & s0 & i1) |
               (s1 & ~s0 & i2) |
               (s1 & s0 & i3) ;

endmodule
```

## Contd..

- Method 2: conditional operator

*Example 6-3* *4-to-1 Multiplexer, Using Conditional Operators*

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator

assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```





## 6.5.2 4-bit Full Adder

- **Method 1: dataflow operators**

*Example 6-4* *4-bit Full Adder, Using Dataflow Operators*

```
// Define a 4-bit full adder by using dataflow statements.
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;

endmodule
```

## Contd..

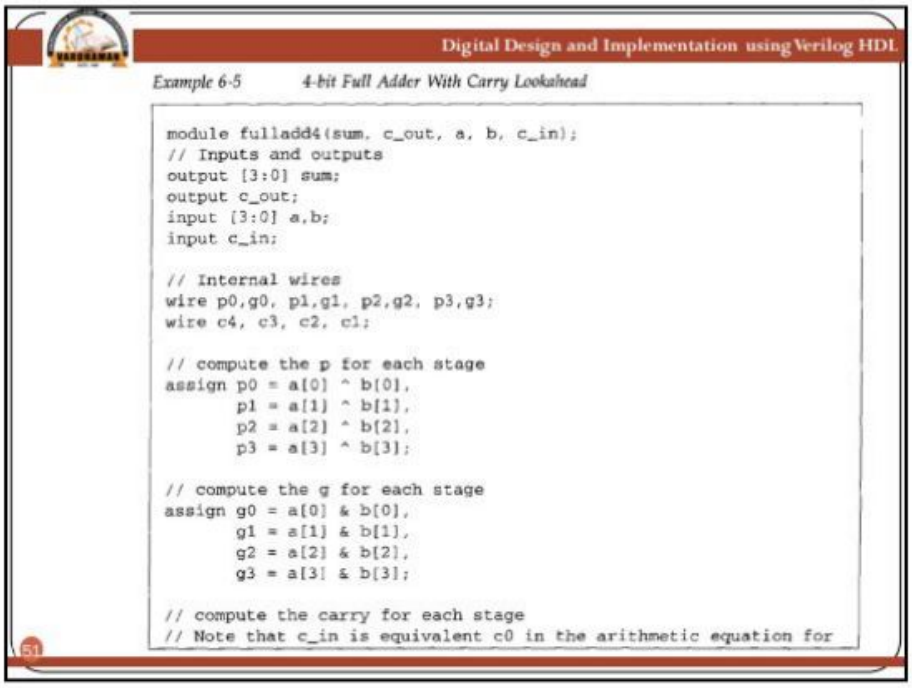
- **Method 2: full adder with carry look ahead**
- In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals.
- An n-bit ripple carry adder will have *2n gate levels. The propagation time can be a limiting factor on the speed of the* circuit.
- One of the most popular methods to reduce delay is to use a *carry look ahead mechanism.*
- *Logic equations for implementing the carry look ahead* mechanism can be found in any logic design book.

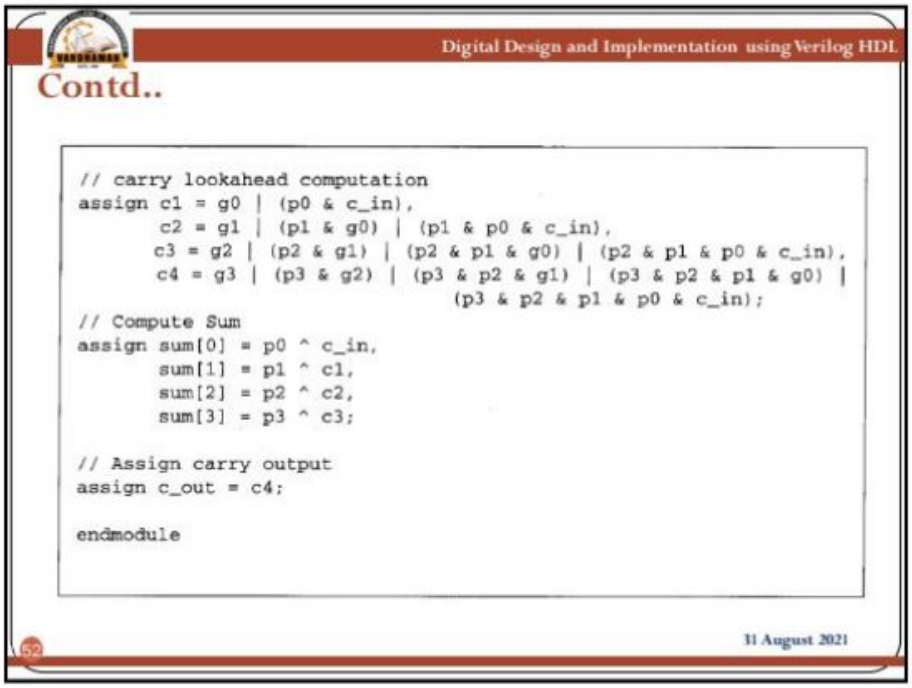




## Contd..

- The propagation delay is reduced to *four gate levels, irrespective of the number of bits in the adder.*
- *The* Verilog description for a carry look ahead adder is shown in Example 6-5.
- This module can be substituted in place of the full adder modules described before without changing any other component of the simulation.
- The simulation results will be unchanged.


Digital Design and Implementation using Verilog HDL
Example 6-5 4-bit Full Adder With Carry Lookahead
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;
// compute the p for each stage
assign p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
// compute the g for each stage
assign g0 = a[0] & b[0],
g1 = a[1] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];
// compute the carry for each stage
// Note that c_in is equivalent c0 in the arithmetic equation for
51


Digital Design and Implementation using Verilog HDL
Contd..
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in,
sum[1] = p1 ^ c1,
sum[2] = p2 ^ c2,
sum[3] = p3 ^ c3;
// Assign carry output
assign c_out = c4;
endmodule
11 August 2021
52

## 6.5.3 Ripple Counter

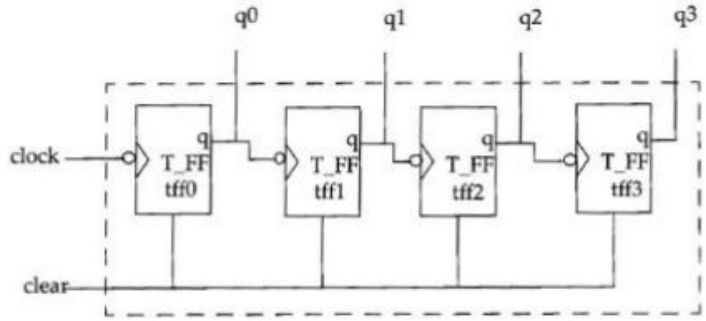



*Figure 6-2 4-bit Ripple Carry Counter*





## Contd..

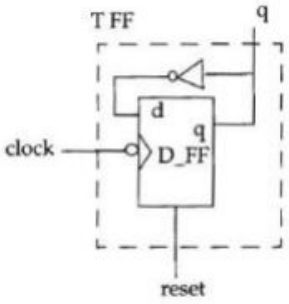



*Figure 6-3 T-flipflop*

## Contd..

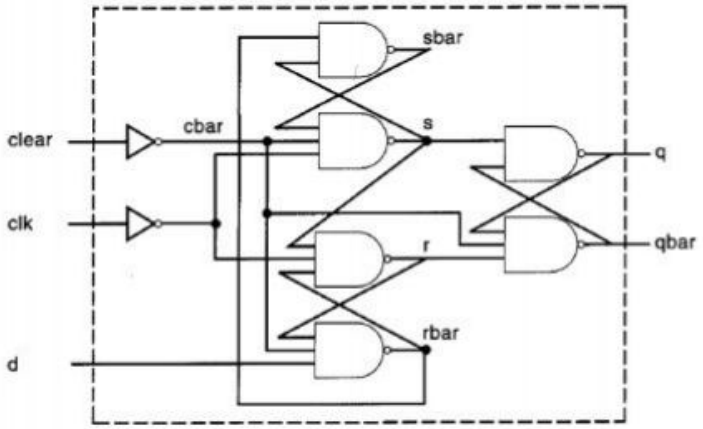



*Figure 6-4 Negative Edge-Triggered D-flipflop with clear*

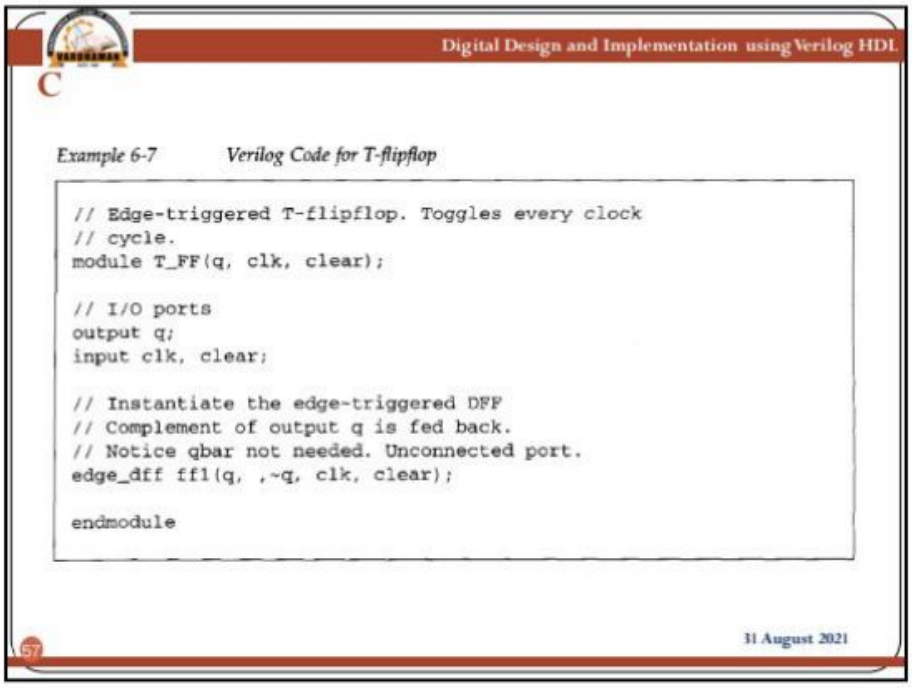


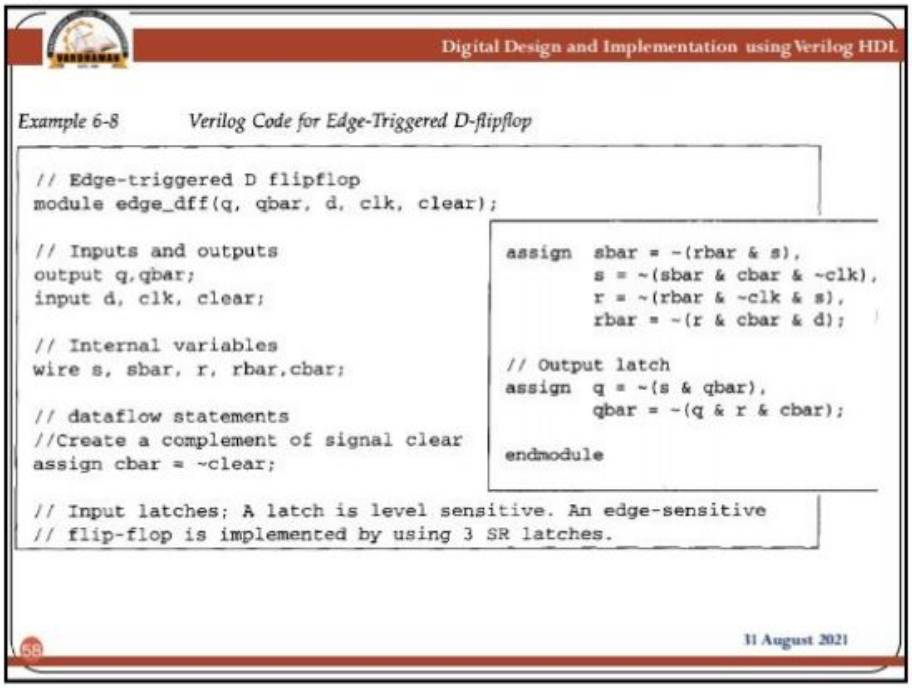


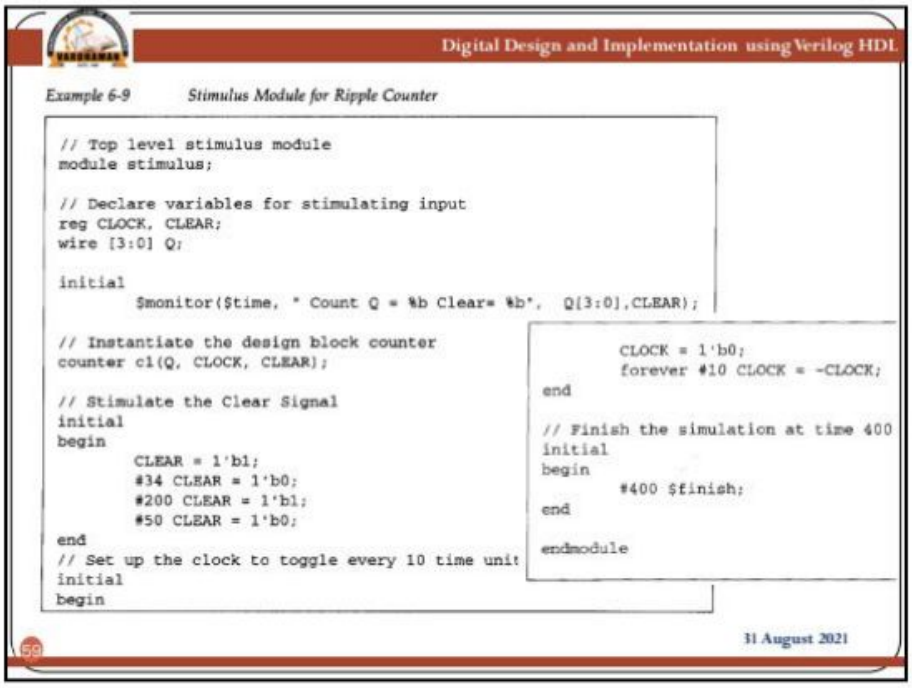
## Contd..

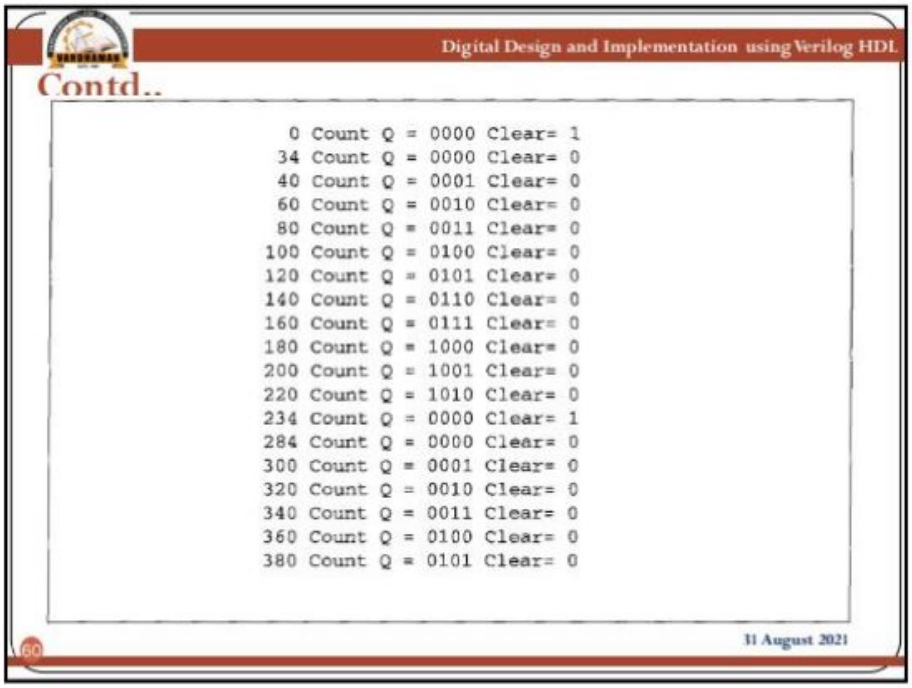
*Example 6-6 Verilog Code for Ripple Counter*

```
// Ripple counter
module counter(Q , clock, clear);

// I/O ports
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```


Digital Design and Implementation using Verilog HDL
C
Example 6-7 Verilog Code for T-flipflop
// Edge-triggered T-flipflop. Toggles every clock
// cycle.
module T_FF(q, clk, clear);
// I/O ports
output q;
input clk, clear;
// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);
endmodule
11 August 2021
57


Digital Design and Implementation using Verilog HDL
Example 6-8 Verilog Code for Edge-Triggered D-flipflop
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);
// Inputs and outputs
output q,qbar;
input d, clk, clear;
// Internal variables
wire s, sbar, r, rbar,cbar;
// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;
// Input latches; A latch is level sensitive. An edge-sensitive
// flip-flop is implemented by using 3 SR latches.
assign sbar = ~(rbar & s),
s = ~(sbar & cbar & ~clk),
r = ~(rbar & ~clk & s),
rbar = ~(r & cbar & d);
// Output latch
assign q = ~(s & qbar),
qbar = ~(q & r & cbar);
endmodule
11 August 2021
58


Digital Design and Implementation using Verilog HDL
Example 6-9 Stimulus Module for Ripple Counter
// Top level stimulus module
module stimulus;
// Declare variables for stimulating input
reg CLOCK, CLEAR;
wire [3:0] Q;
initial
$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);
// Instantiate the design block counter
counter c1(Q, CLOCK, CLEAR);
// Stimulate the Clear Signal
initial
begin
CLEAR = 1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
end
// Set up the clock to toggle every 10 time unit
initial
begin
CLOCK = 1'b0;
forever #10 CLOCK = ~CLOCK;
end
// Finish the simulation at time 400
initial
begin
#400 $finish;
end
endmodule
11 August 2021
59


Digital Design and Implementation using Verilog HDL
Contd..
0 Count Q = 0000 Clear= 1
34 Count Q = 0000 Clear= 0
40 Count Q = 0001 Clear= 0
60 Count Q = 0010 Clear= 0
80 Count Q = 0011 Clear= 0
100 Count Q = 0100 Clear= 0
120 Count Q = 0101 Clear= 0
140 Count Q = 0110 Clear= 0
160 Count Q = 0111 Clear= 0
180 Count Q = 1000 Clear= 0
200 Count Q = 1001 Clear= 0
220 Count Q = 1010 Clear= 0
234 Count Q = 0000 Clear= 1
284 Count Q = 0000 Clear= 0
300 Count Q = 0001 Clear= 0
320 Count Q = 0010 Clear= 0
340 Count Q = 0011 Clear= 0
360 Count Q = 0100 Clear= 0
380 Count Q = 0101 Clear= 0
11 August 2021
60

## 6.6 Summary

- Continuous assignment is one of the main constructs used in dataflow modeling. A continuous assignment is always active and the assignment expression is evaluated as soon as one, of the right-hand-side variables changes. The left-hand side of a continuous assignment must be a net. Any logic function can be realized with continuous assignments.
- Delay values control the time between the change in a right-hand-side variable and when the new value is assigned to the left-hand side. Delays on a net can be defined in the **assign statement, implicit continuous** assignment, or net declaration.
- Assignment statements contain expressions, operators, and operands.





## Contd..

- The operator types are *arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, replication, and conditional. Unary operators require one* operand, binary operators require two operands, and ternary require three operands. The concatenation operator can take any number of operands.
- The *conditional operator behaves like a multiplexer in hardware or like the ifthen-*else statement in programming languages.
- Dataflow description of a circuit is more concise than a gate-level description. The 4-to-1 multiplexer and the 4-bit full adder discussed in the gate-level modeling chapter can also be designed by use of dataflow statements. Two dataflow implementations for both circuits were discussed. A 4-bit ripple counter using negative edge-triggered D-flipflops was designed.

## Objectives

After completing this chapter, you will be able to:

- Describe the behavioral modeling structures
- Describe procedural constructs
- Understand the features of initial blocks
- Understand the features of always blocks
- Distinguish the differences between blocking and nonblocking assignments
- Understand the features of timing controls
- Understand the features of selection constructs
- Understand the features of loop constructs





## Introduction to Behavioral Modeling

- Behavioral modeling enables you to describe a system at a high level of abstraction.
- Use behavioral modeling to describe the functionality of the system.
- Verilog uses high-level language constructs much like a traditional programming language. Example: C-Language
- The constructs available in behavioral modeling aim at the system level description.
- Behavioral modeling in Verilog is described by specifying a set of concurrently active procedural blocks.
- RTL code is a subset of behavioral modeling. Some behavioral constructs are synthesizable.

## Operations and Assignments

- The assignment is done through the "=" symbol (or the "<=" symbol).

- 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,

  $N = \sim N;$

  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.





## Operations and Assignments

- The operands on the right side can be of the net or variable type. They can be scalars or vectors.

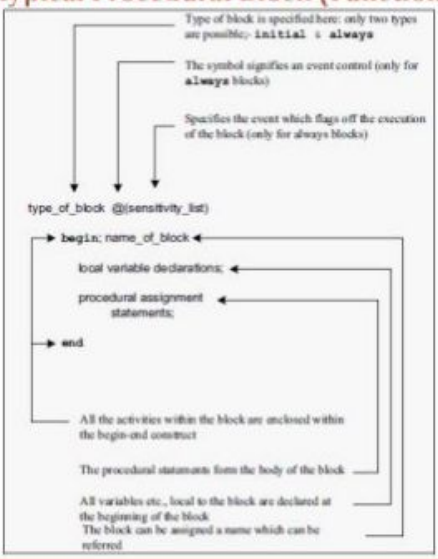
It is necessary to maintain consistency of the operands in the operation expression – *e.g.*,

$N = m / l;$

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 can be a scalar, a vector, a part vector, or a concatenated vector.

## Structure of a Typical Procedural Block (Functional Bifurcation)

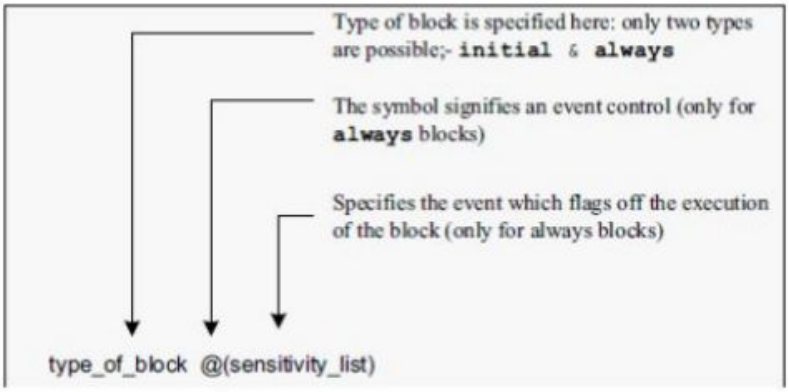




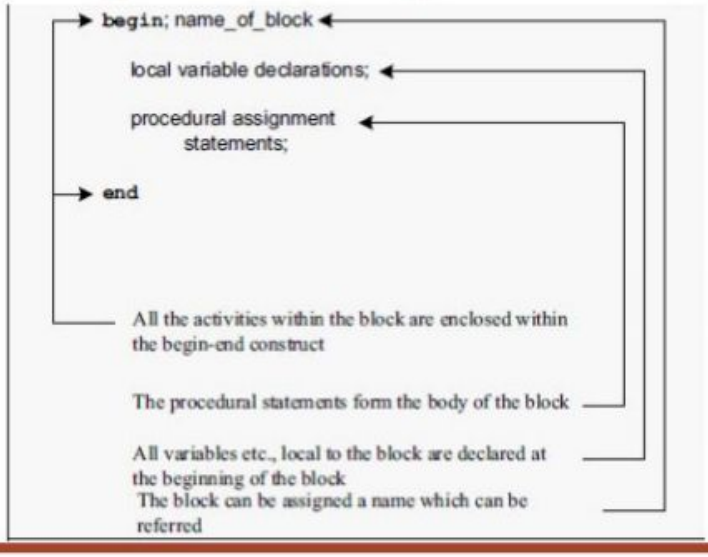





## Structure of a Typical Procedural Block (Functional Bifurcation)



## Structure of a Typical Procedural Block (Functional Bifurcation)

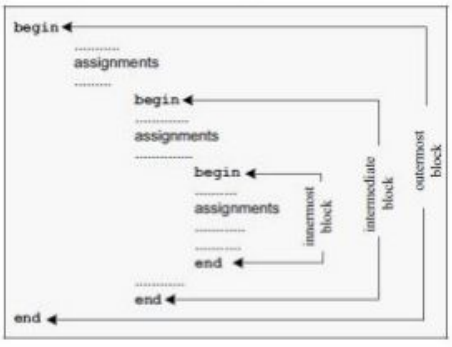







## begin-end Constructs

- 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.



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





## 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.

## Procedural Constructs

- Two Procedural Constructs:
  - **initial** block: executes only once
  - **always** block: executes in a loop
- Block execution is triggered based on user specified conditions.
- All procedural blocks are automatically activated at time 0
- All procedural blocks are executed concurrently.
- **reg** is the main data type that is manipulated within a procedural block
  - It holds its value until assigned a new value.






## Procedural Constructs Cont...

**Initial:**

- Executes only once at the beginning of simulation

**initial**

*statements*

**Always:**

- Executes continuously; must be used with some form of timing control

**always** ***(timing_control)***

*statements*

## Initial Construct

```
//general syntax of the initial sequential block
//containing more than one statement
initial
begin
   //sequential statement 1
   //sequential statement 2
  ...
end

//general syntax of the initial sequential block
//containing one statement (no need for begin...end)
initial
   //sequential statement
```





## Always Constructs

(a)

```
1  always @(event_expression)
2  begin
3   //sequential statement 1
4   //sequential statement 2
5   ...
6  end
```

(b)

```
1  always @(input1 or input2 or input3...)
2  begin
3   //sequential statement 1
4   //sequential statement 2
5   ...
6  end
```

(c)

```
1  always @(input1, input2, input3...)
2  begin
3   //sequential statement 1
4   //sequential statement 2
5   ...
6  end
```

(d)

```
1  always @( * )
2  begin
3      //sequential statement 1
4      //sequential statement 2
5   ...
6  end
```

(e)

```
1  always @(a)
2      y = a * a;
```

Alternative formats for the always sequential block: (a) General form of the always sequential block; (b) always sequential block with or-separated list; (c) always sequential block with comma-separated list; (d) always sequential block with wildcard event expression; (e) always sequential block containing a single sequential statement.

## Example of Always Construct

```
reg  clock;
initial  clock = 1`b0;

always  #5 clock = ~clock;
```





## Examples of Initial Construct

```
reg  x, y, z;
initial  begin      // complex statement
         x = 1`b0;  y = 1`b1;  z = 1`b0;
#10      x = 1`b1;  y = 1`b1;  z = 1`b1;
end
initial x = 1`b0;  // single statement
```

## Initial Construct

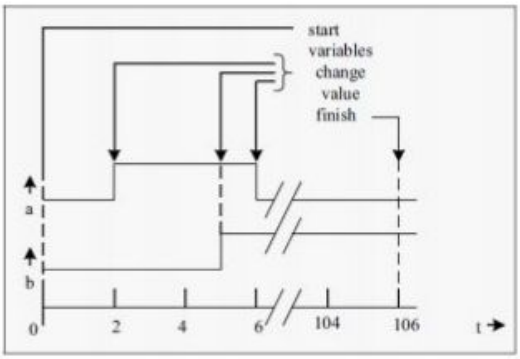
- A set of procedural assignments within an **initial** constructs are executed only once.
- In any assignment statement the **left-hand** side has to be a **storage** type of element (and not a net). It can be a **reg, integer, or real** type of variable. The right hand side can be a **storage** type of variable (**reg, integer,** or **real** type of variable) or a **net**.
- All the procedural assignments appear within a **begin–end** block.
- 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 Figure.





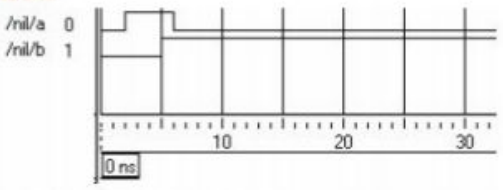
## initial Construct

```
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
```



## initial Construct

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




OUTPUT:

0ns display: a = 0, b = 0
0ns monitor: a = 0, b = 0
2ns monitor: a = 1, b = 0
5ns monitor: a = 1, b = 1
6ns monitor: a = 0, b = 1





## Multiple Initial Construct

- 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.

## Initial Construct

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

```
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
```

OUTPUT

0ns display: a = 0, b = 0
0ns monitor: a = 0, b = 0
2ns monitor: a = 1, b = 1
6ns monitor: a = 0, b = 1





## Always Construct

- Any behavioural 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.
- The process can have one assignment statement or multiple assignment statements. All the assignments are grouped together within a "**begin – end**" construct.
- Normally the statements are executed sequentially in the order they appear.

## 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 the **reg** (variable) clk.
- @**(posedge clk)**
  executes the following block at the positive edge of the **reg** (variable) clk.
- @clk
  executes the following block at both the edges of clk.





## Event Control cont...

- @(prt or clr) :
  With the above event the block is executed whenever either of the variables prt or clr under goes a change.
- @**(posedge** clk1 **or negedge** clk2**) :**
  With the above event the block is executed in two cases – whenever the clock clk1 changes from 0 to 1 state or the clock clk2 changes from 1 to 0.
- 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."
- @**(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)."

## Event Control

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



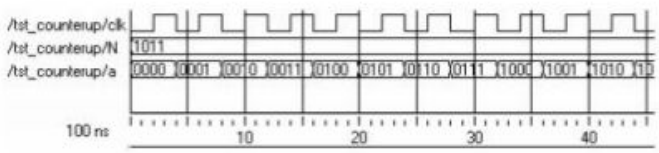


## Up Counter Example:

```
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
```

## Up-Counter Example cont…

```
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
```





## Down counter Example:

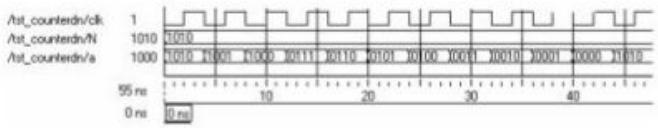
```
module counterdn(a,clk,N);
input clk;
input [3:0]N;
output [3:0]a;
reg [3:0]a;

initial a =4'b0000;

always@(negedge clk)
a=(a==4'b0000)?N:a-1'b1;

endmodule
```

```
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
```

## Down Counter Example:

```
module tst_counterdn();
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
```

```
# 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
```








## Up-Down Counter:

```
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
```

## Up-Down Counter:

```
module tst_updcounter();
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 #34 u_d=~u_d;
initial $monitor ($time,"clk=%b,N=%b,u_d=%b,a=%b",clk,N,u_d,a);
initial #64 $stop;
endmodule
```



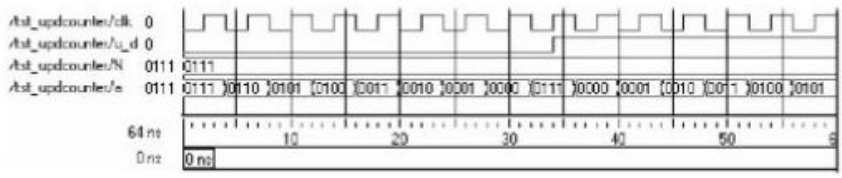




## Up-Down Counter:

```
#                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
```

## Up-Down Counter:








## Clear Up-Down Counter:

```
module clrupdcou(a,clr,clk,N,u_d);
input clr,clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk or posedge clr)
a=(clr)?4'h0:((u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-1'b1));
/*signals having priority over clk have to be included in the sensitivity list*/
endmodule
```

## Clear Up-Down Counter Testbench:

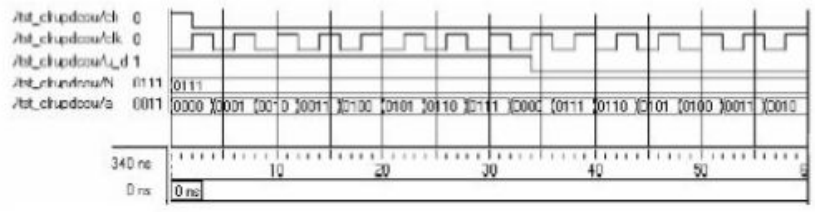
```
module tst_clrupdcou; //TEST_BENCH
reg        clr,clk,u_d;
reg [3:0]  N;
wire [3:0] a;
clrupdcou cc11(a,clr,clk,N,u_d);
initial
begin
N = 4'b0111; Clr = 1'b1; u_d=1'b1; Clk = 0;
end
always
begin
#2 clk = ~clk; clr = 1'b0;
end
always #34 u_d<=~u_d;
initial $monitor($time,"clk=%b,clr=%b,u_d=%b,N=%b,a=%b",clk,clr,u_d,N,a);
initial #60 $stop;
endmodule
```







## Clear Up-Down Counter Output Waveform:



## Shift Register:

```
module rl_shift(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
```





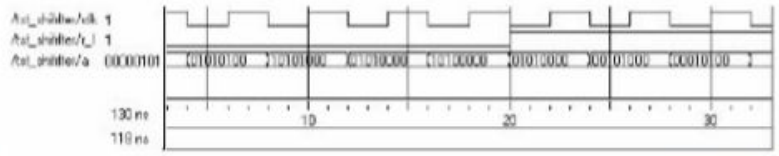


## Shift Register Testbench:

```
module tst_rl_shift;//test-bench
reg clk,r_l;
wire [7:0]a;
rl_shift shrr(a,clk,r_l);
initial
begin
clk =1'b1;  r_l = 0;
end
always #2 clk =~clk;
initial #32 r_l =~r_l;
initial $monitor($time,"clk=%b,r_l = %b,a =%b ",clk,r_l,a);
initial #100 $stop;
endmodule
```

## Shift Register Output:

```
Output
  #     0 clk=1, r_1 = 0 , a = 00000001
  #     2 clk=0, r_1 = 0 , a = 00000010
  #     4 clk=1, r_1 = 0 , a = 00000010
  #     6 clk=0, r_1 = 0 , a = 00000100
  #     8 clk=1, r_1 = 0 , a = 00000100
  #    10 clk=0, r_1 = 0 , a = 00001000
  #    12 clk=1, r_1 = 0 , a = 00001000
  #    14 clk=0, r_1 = 0 , a = 00010000
  #    16 clk=1, r_1 = 1 , a = 00010000
  #    18 clk=0, r_1 = 1 , a = 00001000
  #    20 clk=1, r_1 = 1 , a = 00001000
  #    22 clk=0, r_1 = 1 , a = 00000100
  #    24 clk=1, r_1 = 1 , a = 00000100
  #    26 clk=0, r_1 = 1 , a = 00000010
  #    28 clk=1, r_1 = 1 , a = 00000010
```







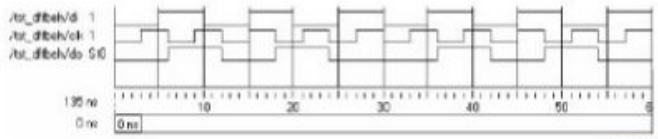

## Clocked Flip-Flop:

```
module tst_DFF();
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
```

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

## Flip Flop Output:

```
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
```








## Ex: D-Latch

```
module DFF_EN(do,di,en); // d-latch
output do;
input di,en;
reg do;
initial
do=1'b0;
always@(di or en)
if(en)
do=di;
endmodule
```

## D-Latch Testbench

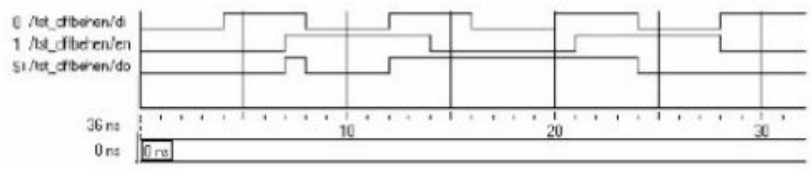
```
module tst_DFF_EN;//test-bench
reg di,en;
wire do;
DFF_EN d1(do,di,en);
initial
begin
en=0; di=1'b0;
end
always#7 en =~en;
always#4 di=~di;
initial $monitor($time,"en=%b,di=%b,do=%b",en,di,do);
initial #50 $stop;
endmodule
```







## D-Latch Output Waveform:



## 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.
- Syntax: **wait** (alpha) assignment1;
- 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, assignment1 is carried out.
- One can also have a group of assignments within a block in place of assignment1.
- The activity is **level-sensitive** in nature, in contrast to the edge-sensitive nature of event specified through @.





## Ex-1: Counter using Wait construct

```
module ctr_wt(a,clk,N,En);
input clk,En;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b1111;
always
begin
     wait(En)
     @(negedge clk)
      a=(a==N)?4'b0000:a+1'b1;
end
endmodule
```

**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 clk remains high.

## Ex-1: cont..

```
module tst_ctr_wt;
reg         clk,En;
reg [3:0]  N;
wire[3:0]  a;
ctr_wt c1(a,clk,N,En);
initial
begin
clk=0;N=4'b1111;En=1'b0;#5 En=1'b1;#20 En=1'b0;
end
always #2 clk=~clk;
initial #35 $stop;
initial $monitor($time,"clk=%h,En=%b,N=%b,a=%b",clk,En,N,a,);
endmodule
```





## Wait Construct Ex-1 Output

```
//output
#          0clk=0,En=0,N=1111,a=1111
#          2clk=1,En=0,N=1111,a=1111
#          4clk=0,En=0,N=1111,a=1111
#          5clk=0,En=1,N=1111,a=1111
#          6clk=1,En=1,N=1111,a=1111
#          8clk=0,En=1,N=1111,a=0000
#         10clk=1,En=1,N=1111,a=0000
#         12clk=0,En=1,N=1111,a=0001
#         14clk=1,En=1,N=1111,a=0001
#         16clk=0,En=1,N=1111,a=0010
#         18clk=1,En=1,N=1111,a=0010
#         20clk=0,En=1,N=1111,a=0011
#         22clk=1,En=1,N=1111,a=0011
#         24clk=0,En=1,N=1111,a=0100
#         25clk=0,En=0,N=1111,a=0100
#         26clk=1,En=0,N=1111,a=0100
#         28clk=0,En=0,N=1111,a=0101
#         30clk=1,En=0,N=1111,a=0101
#         32clk=0,En=0,N=1111,a=0101
#         34clk=1,En=0,N=1111,a=0101
```

## Example-2: Serial receiver Wait Construct

- Wait for recv input to go high.
- Once recv=1, latch the next 4 successive bits of incoming data into respective bit positions of the do register.

```
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
```







## Wait Construct

```
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 #2 clk = ~clk;
initial begin
#7 di=1'b1; #4 di=1'b0; #8 di=1'b1; #8 di=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
```

## Example on Wait Construct output

```
//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
```





## Multiple Always Blocks

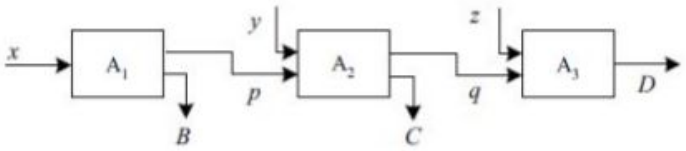



Figure: A module where execution proceeds through three blocks sequentially

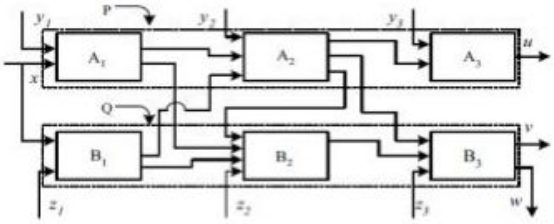
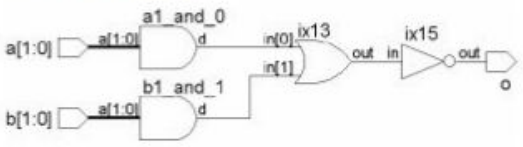



Figure :A module where execution proceeds concurrently through two groups of blocks.

## Design at Behavioral Level

```
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
```




```
# 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
```





## Example-1:

```
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 instantiations; 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
```

## Example-2:

```
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
```

```
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
```





## Example

```
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
```

```
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
```

## Assignments

- The assignment is the basic mechanism for placing values into nets and variables.
- There are two basic forms of assignments:
- **Continuous assignments,** which assigns values to nets.
- **Procedural assignments,** which assigns values to registers.
- There are two additional forms of assignments, **assign/deassign** and **force/release**, which are called **procedural continuous assignments**.







## Assignments Cont...

- An assignment consists of two parts, a left-hand side and a right-hand side.
- The right-hand side can be any expression that evaluates to a value.
- The left-hand side indicates the variable data type (reg) to which the right-hand side value is to be assigned.

## Assignments Cont...

- The left-hand side can take one of the forms given in Table below, depending on whether the assignment is a continuous assignment or a procedural assignment.

| Statement type | Left-hand side |
|---|---|
| Continuous assignment | Net (vector or scalar)<br>Constant bit-select of a vector net<br>Constant part-select of a vector net<br>Constant indexed part-select of a vector net<br>Concatenation or nested concatenation of any of the above left-hand side |
| Procedural assignment | Variables (vector or scalar)<br>Bit-select of a vector reg, integer, or time variable<br>Constant part-select of a vector reg, integer, or time variable<br>Indexed part-select of a vector reg, integer, or time variable<br>Memory word<br>Concatenation or nested concatenation of any of the above left-hand side |





## Continuous Assignments (Used in Dataflow Modeling)

- Continuous assignments drive values onto nets (vector and scalar).
- The assignment occurs whenever simulation causes the value of the right-hand side to change.
- The right-hand side expression is not restricted in any way.
- Whenever an operand in the right hand side changes value during simulation, the whole right-hand side expression is evaluated and assigned to the left-hand side.
- Modeling a 16-bit adder with a continuous assignment

```
wire [15:0] sum, a, b; // declaration of 16-bit vector nets
wire cin, cout; // declaration of 1-bit (scalar) nets
assign {cout, sum} = a + b + cin;
```

## Some Continuous Assignments

```
//some continuous assignment statements
   assign A = q[0] , B = q[1] , C = q [2] ;

   assign out = (~s1 & ~s0 & i0) |
     (~s1 & s0 & i1) |
     (s1 & ~s0 & i2) |
     (s1 & s0 & i3);

   assign #15{ c_out, sum} = a + b + c_in;
```





## Continuous Assignments

```
module latch (output q, input data, en);

assign q = en ? data : q;

endmodule
```

## Procedural Assignments

- Procedural assignments drive values onto registers (vector and scalar).
- The target must be a register or integer type.
- They do not have duration (the register data type holds the value of the assignment until the next procedural assignment to the register).
- They occur within procedures such as ***always*** and ***initial.***
- They are triggered when the flow of execution in the simulation reaches them.
- Two kinds of procedural assignments
  - **Blocking assignment**
  - **Non-blocking assignment**





## Procedural Assignments

- **Syntax**

  variable_value = [timing_control] expression

  variable_value <= [timing_control] expression

  [timing_control] variable_value = expression

  [timing_control] variable_value <= expression

## Blocking Assignments

- These statements are executed in the specified order
- Use the "=" operator

```
reg t1,t2,t3;
initial
  begin
  t1=a&b;
  t2=b&cin;
  t3=a&cin;
  cout=t1|t2|t3;
  end
```

- The t1 assignment occurs first, t1 is computed, then the second statement executes, t2 is assigned and the third statement is executed and t3 is assigned, and so on.





## Blocking Assignments

```
// blocking assignments
initial begin
  x = #5 1'b0;   // at time 5
  y = #3 1'b1;   // at time 8
  z = #6 1'b0;   // at time 14
end
```

## Non-blocking Assignments

- Use the <= operator
- Used to model several concurrent data transfers
- The assignment to the target is not blocked (due to delays).

```
reg  x, y, z;
// nonblocking assignments
initial begin
   x <= #5 1'b0;  // at time 5
   y <= #3 1'b1;  // at time 3
   z <= #6 1'b0;  // at time 6
end
```







## Blocking and Non-Blocking Assignments

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

Fig 1: Swapping variable values through nonblocking assignments.

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

Fig 2: Another group of blocking assignments.

- From Fig-1, 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.

## Blocking and Non-Blocking Assignments Cont…

- The assignment B <= A ; takes the value of A– i.e., 00 – stores it separately and assigns it to B.
- 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 Figure-2.
- 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.





## Blocking and Non-Blocking Assignments Cont…

```
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
```

**Figure:** Segment of a module involving blocking and non-blocking assignments.

## Blocking and Non-Blocking Assignments Cont…

- 1. At the positive edge of the clock, values of A, B, and C are read and stored and B &(~C) are computed.
- 2. 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.
- 3. At the next negative clk edge, C is assigned the value of B & (~C) evaluated and stored earlier (=1) which is mentioned in (1) above.
- 4. Two nanoseconds after the positive edge of clk (*i.e., after the entry to the* block), B is assigned the value of C stored earlier (=0).





## Blocking and Non-Blocking Assignments Cont…

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

Figure -1: A set of assignments with a potential race condition.

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

Figure-2: The assignments of Figure-1 modified to avoid race condition.

- In the segment in Figure-1, **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 Figure-2, the two assignments are of the **non-blocking** type; **A** is assigned the **previous** value of **B**, while **B** is assigned the **previous** value of **A**. The **race condition** is **avoided** here.

## Blocking and Non-Blocking Assignments Observations

- In a design whenever a number of **concurrent** data transfers take place after a common event, **non-blocking** assignments are **preferred**. The common event forms the sensitivity list followed by the non-blocking assignments.
- All non-blocking assignments in a block are executed concurrently. The scheduling is done in the same order as the specified statements. If two assignments are done to a **reg** in a time step, the latter prevails. example

  A <= 1;

  A <= 0;

  A is assigned the value of **zero**.
- Although blocking and non-blocking assignment can be mixed in a block, many **synthesis tools may not support** such combinations.





## Non-blocking Assignments and Delays

- **Delays** – of the assignment type and the **intra-assignment** type – can be associated with non-blocking assignments also.
- The principle of their operation is similar to that with blocking assignments. As explained earlier, the **delay** values can be **constant** expressions.
- Blocking and non-blocking 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.

## Non-blocking Assignments and Delays

```
module nil1 (c1, a, b);
output c1;
input a, b;
reg c1;
always @(a or b)
        #3      c1 = a&b;
endmodule
```

Fig: 1 A time delay in an evaluation.

```
module nil2 (c2, a, b);
output c2;
input a, b;
reg c2;
always  @(a or b)
                c2 = #3  a&b;
endmodule
```

Fig: 2 An intra-assignment delay.

Consider the module of Figure1, 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 period, the block is not activated. Hence the module does not depict the desired output.





## Non-blocking Assignments and Delays

- Consider the module of Figure-2 with an intra-assignment delay of 3 ns to the assignment to c2. The always block is activated if a or b changes. (a & b) is evaluated immediately but assigned to c2 only after 3 ns. However, the behavior is not acceptable on two counts:
- The output assignment has to wait for 3 ns after the change.
- Only after the delayed assignment to c2, the event (a or b) checked for change. If a or b changes in period, the block is not activated.
- The module in Figure-3 has a blocking delay of 3 ns; but the assignment is of the non-blocking type.
- 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 3 ns. If a or b changes in period, the block is not activated.

## Non-blocking Assignments and Delays

```
module nil3 (c3, a, b);
output c3;
input a, b;
reg c3;
always @(a or b)
        #3      c3 <= a&b;
endmodule
```

Fig -3 A time delay in a non-blocking assignment.

```
module nil4 (c4, a, b);
output c4;
input a, b;
reg c4;
always @(a or b)
        c4 <= #3 a&b;
endmodule
```

Fig-4 An intra-assignment delay in a non-blocking assignment.

- The module in Figure-4 possibly 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 assignment is essentially a delay operation.





## Non-blocking Assignments and Delays

- Figure-5 shows the waveforms for c1, c2, c3, and c4 in the modules of Figures 1 to 4 for representative waveforms of a and b. One can clearly see that c4 has a representation of a & b, which is the most acceptable of the lot.

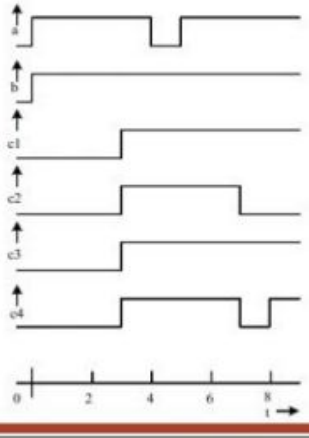
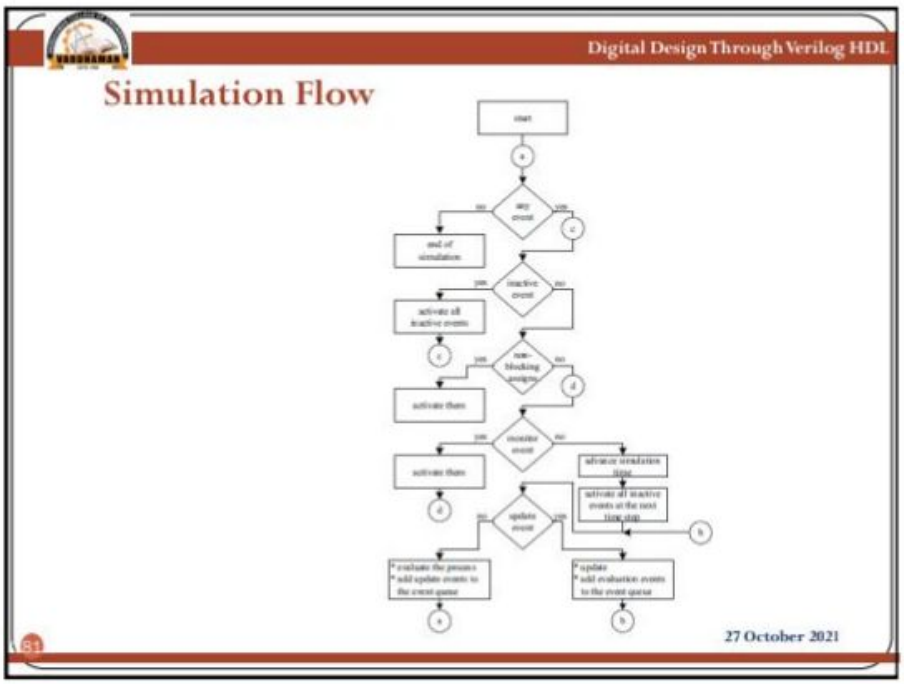
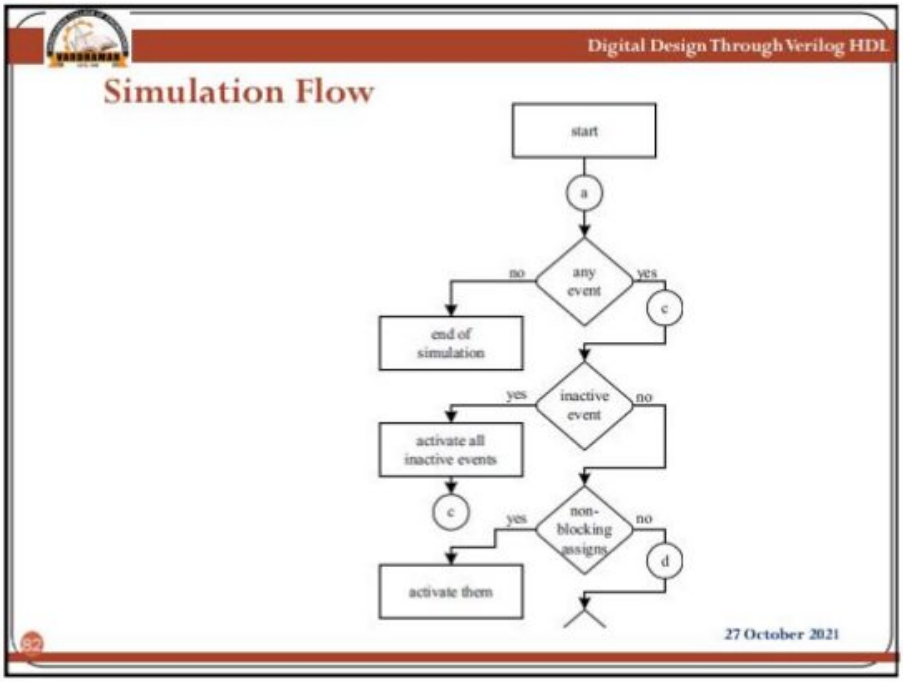
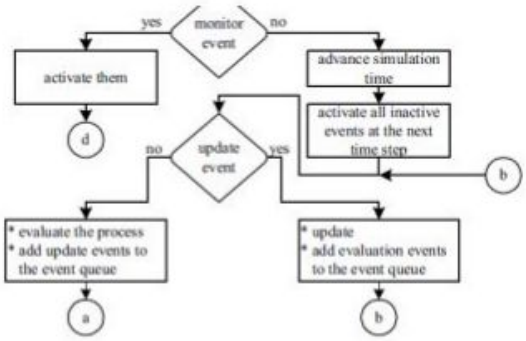



Fig 5: Waveforms of c1, c2, c3, and c4 of the modules in Figures-1 to 4 for representative values of a and b.


Digital Design Through Verilog HDL
Simulation Flow
27 October 2021


Digital Design Through Verilog HDL
Simulation Flow
start
a
any event
no
yes
c
end of simulation
inactive event
yes
no
activate all inactive events
c
non-blocking assigns
yes
no
d
activate them
27 October 2021

## Simulation Flow








## Simulation Flow:

- Simulation time: 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.

## Simulation Flow: 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 –A number of active events can be present for execution at any simulation time step.
- 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.
- Non 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 non-blocking assignment events.
- Future events – Events scheduled to occur at some future simulation time are the future events.





## Coding Styles

- Blocking assignments (=) are used to implement the combinational circuits.
- Non-blocking assignments (<=) are used to implement the sequential circuits.

## Timing Control

- **Delay timing control**
  - Regular delay control
  - Intra assignment delay control
- **Event Timing Control**
  - Edge-triggered event control
    - Named event control
    - Event or control
  - Level-sensitive event control





## Delay Timing Control

- **Regular delay control**
  - Defers the execution of the entire statement

```
reg  x, y;
integer count;

  #25       y <= ~x;              // at time 25
  #15  count <= count  + 1;       // at time 40
```

## Delay Timing Control

- **Intra-assignment delay control**
  - Defers the assignment to the left-hand-side variable

```
    y = #25  ~x;          // assign to y at time 25
count = #15 count  + 1;   // assign to count at time 40
```

```
    y <= #25  ~x;         // assign to y at time 25
count <= #15 count  + 1;  // assign to count at time 15
```





## Event Timing Control

- **Edge-triggered event control**
  - @(posedge clock)
  - @(negedge clock)

```
always @(posedge clock)  begin
    reg1 <= #25  in_1;
    reg2 <= @(negedge clock) in_2 ^ in_3;
    reg3 <= in_1;
end
```

## Event Timing Control

- **A named event**

```
...
event change;
...
always
...
... → change;
...
always@change
...
```

```
event  received_data;                  // declare
always @(posedge clock)
    if (last_byte) -> received_data;  // trigger
always @(received_data)                // recognize
    begin ... end
```





## Event Timing Control

- **Event 'or' control**
  - or ; events can be or'ed to indicate "if any of the events occur".
  - , ; comma separated event list.
  - * or (*) means any signals

```
always @(posedge clock or negedge reset_n)
    if (!reset_n) q <= 1`b0;
    else          q <= d;
```

## Event Timing Control

- **Level-sensitive event control**

```
wait (condition)
  procedural_statement;
```

```
always
    wait (count_enable) count = count –1 ;

always
    wait (count_enable) #10 count = count –1 ;
```







## Selection Statements

- **Case/Casex/Casez Statements**

  A case statement is a multi-way conditional branch.

- 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.

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

## 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 ref1, 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.





## Case statement Examples:

```
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'h1;
2'b01:o=4'h2;
2'b10:o=4'h4;
2'b11:o=4'h8;
default: begin
        $display ("error");
        o=4'h0;
         end
endcase
end
endmodule
```

```
//test bench
module tst_dec2_4beh();
wire [3:0]o; reg[1:0] i; //reg en;
dec2_4beh dec(o,i);
initial
begin
i =2'b00;
#2i =2'b01;
#2i =2'b10;
#2i =2'b11;
#2i =2'b11;
#2i =2'b0x;
end
initial $monitor ($time , " output o = %b , input i
= %b " , o ,i);
endmodule
```

```
output
# 0 output o =  0000  , input i  = 00
# 2 output o =  0001  , input i  = 01
# 4 output o =  0010  , input i  = 10
# 6 output o =  0100  , input i  = 11
# error
# 10 output o =  0000  , input i = 0x
```

## Case Statement Example 2:

```
module dec2_4beh1(o,i);
output[3:0]o; input[1:0]i; reg[3:0]o;
always@(i)
begin
case(i)
2'b00:o[0]=1'b1;
2'b01:o[1]=1'b1;
2'b10:o[2]=1'b1;
2'b11:o[3]=1'b1;
2'b0x,2'b1x,2'bx0,2'bx1:o=4'b0000;
default: begin
           $display ("error");
           o=4'h0;
         end
endcase
end
endmodule
```

```
# 0 output o =  xxx1  , input i  = 00
# 2 output o =  xx11  , input i  = 01
# 4 output o =  x111  , input i  = 10
# 6 output o =  1111  , input i  = 11
# 10 output o = 0000  , input i  = 1x
# 12 output o = 0000  , input i  = 0x
# 14 output o = 0000  , input i  = x0
# 16 output o = 0000  , input i  = x1
# error
# 18 output o = 0000  , input i  = xx
# error
# 20 output o = 0000  , input i  = 0z
```





## Case Statement Example 2:

```
module tst_dec2_4beh1;//test bench
wire [3:0]o; reg[1:0] i;
dec2_4beh1 dec(o,i);
initial
begin
i =2'b00;
#2 i =2'b01;
#2 i =2'b10;
#2 i =2'b11;
#2 i =2'b11;
#2 i =2'b1x;
#2 i =2'b0x;
#2 i =2'bx0;
#2 i =2'bx1;
#2 i =2'bxx;
#2 i =2'b0z;
end
initial $monitor ($time , " output o = %b , input i= %b " , o ,i);
endmodule
```

## ALU using CASE statement

```
module alubeh(c,a,b,f);
output [3:0] c;
input  [3:0] a,b;
input [1:0] f;
reg [3:0] c;
always@(a or b or f)
begin
case(f)
2'b00: c=a+b;
2'b01: c=a-b;
2'b10: c=a&b;
2'b11: c=a|b;
endcase
end
endmodule
```

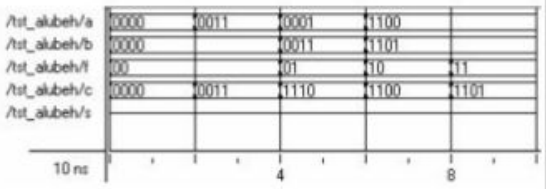
```
0f=00,a=0000,b=0000,c=0000
#2f=00,a=0011,b=0000,c=0011
#4f=01,a=0001,b=0011,c=1110
#6f=10,a=1100,b=1101,c=1100
#8f=11,a=1100,b=1101,c=1101
#10f=00,a=0011,b=0000,c=0011
```





## ALU using CASE statement

```
module tst_alubeh;//test-bench
reg [3:0] a,b; reg[1:0]f; wire[3:0]c;
alubeh aa(c,a,b,f);
initial
begin
f=2'b00;a=2'b00;b=2'b00;
end
always
begin
#2 f=2'b00;a=4'b0011;b=4'b0000;
#2 f=2'b01;a=4'b0001;b=4'b0011;
#2 f=2'b10;a=4'b1100;b=4'b1101;
#2 f=2'b11;a=4'b1100;b=4'b1101;
end
initial
$monitor($time,"f=%b,a=%b,b=%b,c=%b",f,a,b,c);
initial #10 $stop;
endmodule
```



## Selection Statements

- **Don't Cares in Case**

  casex – both the values x and z are considered as don't cares.

  casez – the value z is considered as don't care.

Ex: Priority Encoder

```
module pri_enc(a,b);
output [1:0] a;
input [3:0] b;
reg [1:0] a;
always@(b)
casez(b)
4'bzzz1:a=2'b00;
4'bzz10:a=2'b01;
4'bz100:a=2'b10;
4'b1000:a=2'b11;
endcase
endmodule
```





## Selection Statements

- **Don't Cares in Case**

Ex: Priority Encoder TB

```
module pri_enc_tst;//test-bench
reg [3:0]b; wire[1:0]a;
pri_enc pp(a,b);
initial b=4'bzzz0;
always
begin
#2 b=4'bzzz1;
#2 b=4'bzzz1;
#2 b=4'bzz10;
#2 b=4'bz100;
#2 b=4'b1000;
end
initial $monitor($time, "input b =%b,a =%b ",b,a);
initial #40 $stop;
endmodule
```

output:

```
 0input  b =  zzz0  ,a =  01
 2input  b =  zzz1  ,a =  00
 6input  b =  zz10  ,a =  01
 8input  b =  z100  ,a =  10
10input  b =  1000  ,a =  11
12input  b =  zzz1  ,a =  00
16input  b =  zz10  ,a =  01
18input  b =  z100  ,a =  10
20input  b =  1000  ,a =  11
22input  b =  zzz1  ,a =  00
26input  b =  zz10  ,a =  01
28input  b =  z100  ,a =  10
30input  b =  1000  ,a =  11
32input  b =  zzz1  ,a =  00
36input  b =  zz10  ,a =  01
38input  b =  z100  ,a =  10
```

## Selection Statements

- **Conditional Statements**

The syntax of an if statement is:

```
if (condition_1)
procedural_statement_1;
else if(condition_2)
procedural_statement_2;
else
procedural_statement_3;
```



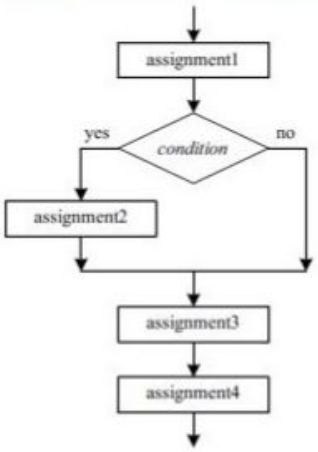


## Selection Statements

**If Construct:**

```
...
assignment1;
if (condition) assignment2;
assignment3;
assignment4;
...
```

## Selection Statements – Flowchart of 'if'








## Selection Statements – Flowchart of 'if'

Ex: Ring Counter

```
Reg[7:0] a;
Reg c;
always@(posedge clk)
begin
        c = a[0];
        a = a>>1'b1; // Since the vacated bit of a is filled with a zero, it need be
        if( c ) a[7] = c;// set only if a[0] =1
end
```

Figure 8.3 A Ring counter description using the `if` construct.

## Selection Statements – Flowchart of 'if-else'

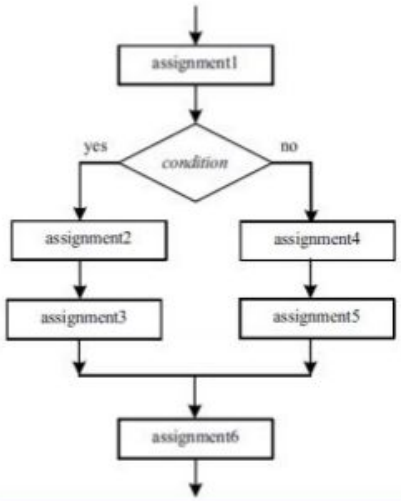
**If – else Construct:**

```
...
assignment1;
if(condition)
        begin           // Alternative 1
                assignment2;
                assignment3;
        end
else
        begin           //alternative 2
                assignment4;
                assignment5;
        end
assignment6;
...
...
```





## Selection Statements – Flowchart of 'if-else'



## Selection Statements – Flowchart of 'if'

Ex: Demux

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





## Selection Statements – Flowchart of 'if'

Ex: Demux TB

```
//tst_bench
module tst_demux();
reg b; reg[1:0]s; wire[3:0]a;
demux d1(a,b,s);
initial
b=1'b0;
always
begin
#2 s=2'b00;b=1'b1; #2 s=2'b00;b=1'b0; #2 s=2'b01;b=1'b0;
#2 s=2'b10;b=1'b1; #2 s=2'b11;b=1'b0;
end
initial  $monitor("t=%0d, s=%b,b=%b,output =%b",$time,s,b,a);
initial #30 $stop;
endmodule
```

Demux output

```
# t=0, s=xx,b=0,output a=0zzz
# t=2, s=00,b=1,output a=zzz1
# t=4, s=00,b=0,output a=zzz0
# t=6, s=01,b=0,output a=zz0z
# t=8, s=10,b=1,output a=z1zz
# t=10, s=11,b=0,output a=0zzz
```

## Selection Statements –

Observations:

- The **$write** is a system task; it is similar to the **$display** task except in one respect: When $**write** is executed, the simulator **does not advance to the new line** after the specified display

- The character set **'%0d'** within the $write statement ensures that the concerned quantity is displayed in decimal form with the **minimum number of digits** necessary for it. It makes the display elegant.





## Selection Statements –

```
//counter using if else if;
module countif(a,clk);
output [7:0]a; input clk; reg[7:0] a,n;
initial
begin
n=8'h0a;
a=8'b00000000;
#45 n=8'h23;
end
always@(posedge clk)
begin
$write ("time=%0d ",$time);
if(a==n)
a=8'h00;
else a=a+1'b1;
end
endmodule
```

```
module tst_countif();//test-bench
reg clk;
wire[7:0]a;
countif c1(a,clk);
initial clk =1'b0;
always
#2clk=~clk;
initial
$monitor(" n=%h, a=%h",c1.n,a);
initial #100 $stop;
endmodule
```

## Selection Statements –

```
 n=0a, a=00
time=2   n=0a, a=01
time=6   n=0a, a=02
time=10  n=0a, a=03
time=14  n=0a, a=04
time=18  n=0a, a=05
time=22  n=0a, a=06
time=26  n=0a, a=07
time=30  n=0a, a=08
time=34  n=0a, a=09
time=38  n=0a, a=0a
time=42  n=0a, a=00

 n=23, a=00
time=46  n=23, a=01
```

```
time=50  n=23, a=02
time=54  n=23, a=03
time=58  n=23, a=04
time=62  n=23, a=05
time=66  n=23, a=06
time=70  n=23, a=07
time=74  n=23, a=08
time=78  n=23, a=09
time=82  n=23, a=0a
time=86  n=23, a=0b
time=90  n=23, a=0c
time=94  n=23, a=0d
time=98  n=23, a=0e
```





## Iterative or Loop Statements

- For Loop
- While Loop
- Repeat Loop
- Forever Loop

## For Loop

- Syntax

```
for (init_expr; condition_expr; update_expr)
statement;
```

- The **for** loop in verilog is quite similar to **for** loop in c.
- The structure of the for loop

```
....
for(assignment1; expression; assignment 2)
statement;
...
```

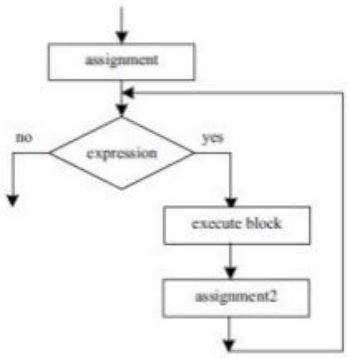
1. Execute assignment1.
2. Evaluate *expression*.
3. If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
4. If *expression* evaluates to the false state (0), exit the loop.
5. Execute assignment2. Go to step 2.





## For Loop

- Flow chart of execution of the for loop



## For Loop

- Ex: Memory Load

```
module MEM_LOAD;
  reg [7:0] m [15:0];
  integer i;
  reg clk;
  always @(negedge clk)
  begin
     for(i=0;i<8;i=i+1)
     begin
     m[i] = i*8;
     $display("t = %0d, i = %0d, m[i] = %0d", $time, i, m[i]);
     end //for
  end //always
  initial clk = 1'b0;
  always #2 clk = ~clk;
  initial #70 $stop;
  endmodule
```

```
//Simulation results
# t=0, i=0, m[0]=0
# t=0, i=1, m[1]=8
# t=0, i=2, m[2]=16
# t=0, i=3, m[3]=24
# t=0, i=4, m[4]=32
# t=0, i=5, m[5]=40
# t=0, i=6, m[6]=48
# t=0, i=7, m[7]=56
```

**Note**: Same data is stored on memory repeatedly on following time instants: 4,8,12,16,20,24,28 32,36,40,44,48,52,56,60,64,68

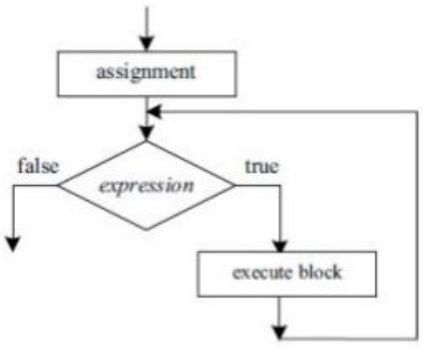




## While Loop

- Syntax

```
while (condition_expr) statement;
```



## 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.



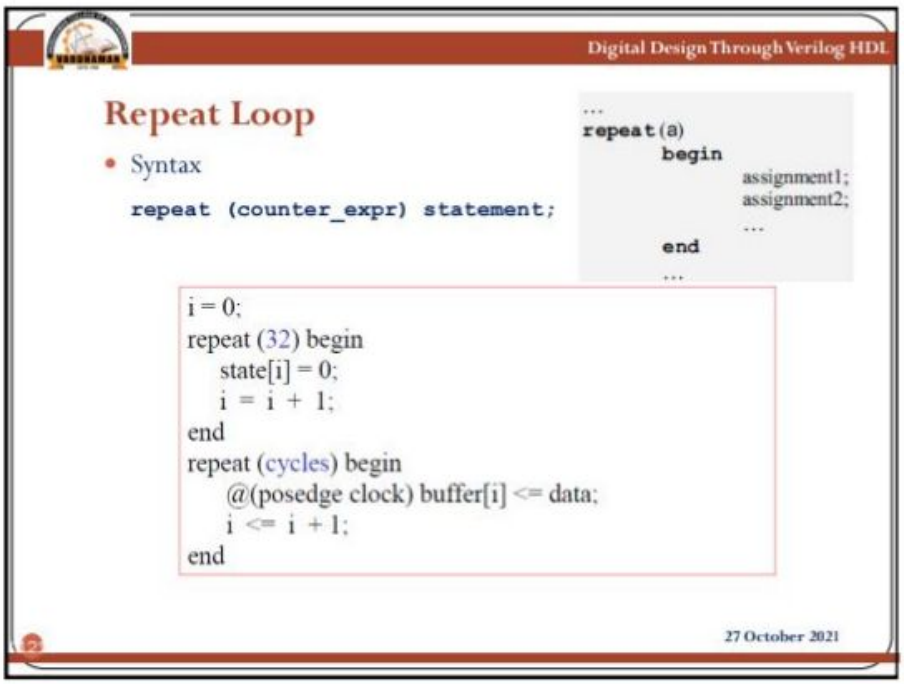


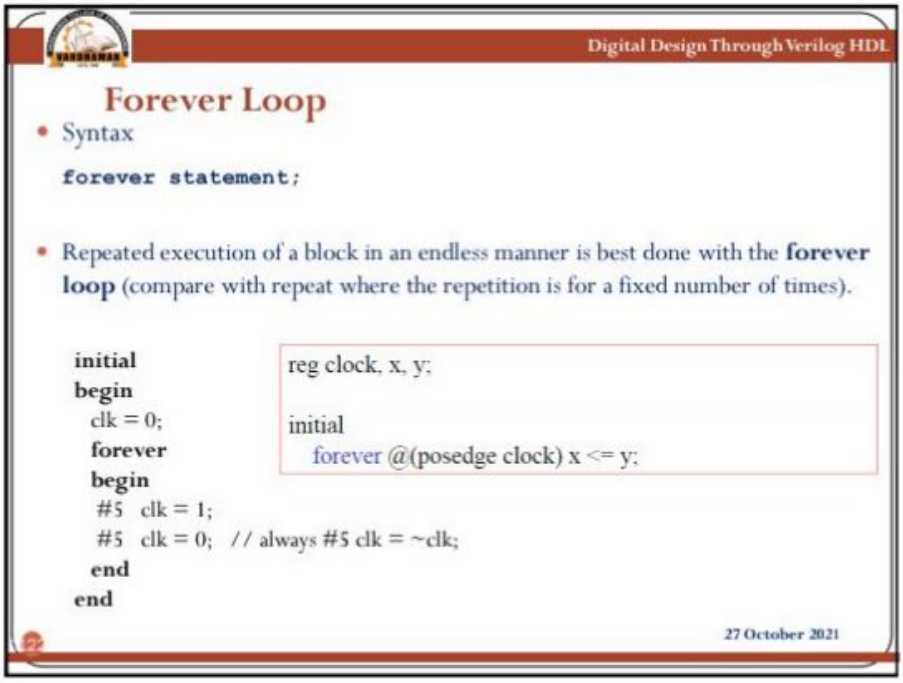
## While Loop

- Example:

```
while (count < 12) count <= count + 1;
while (count <= 100 && flag) begin
   // put statements wanted to be carried out here
end
```

```
// count the zeros in a byte
integer i;
always @(data) begin
  out = 0; i = 0;
  while (i <= 7) begin    // simple condition
     if (data[i] == 0) out = out + 1;
     i = i + 1;
  end
end
```


Digital Design Through Verilog HDL
Repeat Loop
Syntax
repeat (counter_expr) statement;
...
repeat(a)
begin
assignment1;
assignment2;
...
end
...
i = 0;
repeat (32) begin
state[i] = 0;
i = i + 1;
end
repeat (cycles) begin
@(posedge clock) buffer[i] <= data;
i <= i + 1;
end
27 October 2021


Digital Design Through Verilog HDL
Forever Loop
Syntax
forever statement;
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).
initial
begin
clk = 0;
forever
begin
#5 clk = 1;
#5 clk = 0; // always #5 clk = ~clk;
end
end
reg clock, x, y;
initial
forever @(posedge clock) x <= y;
27 October 2021

## 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.
- The disable construct is functionally similar to the **break** in **C**





## disable Construct

```
module or_gate(b,a,en);
input [3:0]a; input en;
output b; reg b;
integer i;
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
endmodule
```

```
# t=0, en = 0, a = 0000, b = x
# t=2, en = 1, a = 0000, b = 0
# t=4, en = 1, a = 0001, b = 0
# t=6, en = 0, a = 0001, b = 0
# t=8, en = 1, a = 0001, b = 1
# t=10, en = 1, a = 0010, b = 1
# t=12, en = 0, a = 0010, b = 1
# t=14, en = 1, a = 0010, b = 1
# t=16, en = 1, a = 0000, b = 1
# t=18, en = 0, a = 0000, b = 1
# t=20, en = 1, a = 0000, b = 0
# t=22, en = 1, a = 0011, b = 0
# t=24, en = 0, a = 0011, b = 0
# t=26, en = 1, a = 0011, b = 1
# t=28, en = 1, a = 0100, b = 1
# t=30, en = 0, a = 0100, b = 1
# t=32, en = 1, a = 0100, b = 1
# t=34, en = 1, a = 1111, b = 1
```

## disable Construct

```
//test-bench
module tst_or_gate();
reg[3:0]a; reg en;  wire b;
or_gate gg(b,a,en);
initial
begin  a = 4'h0;  en = 1'b0; end
initial begin
#2 en=1'b1; #2 a =4'h1; #2 en=1'b0;
#2 en=1'b1; #2 a =4'h2; #2 en=1'b0;
#2 en=1'b1; #2 a =4'h0; #2 en=1'b0;
#2 en=1'b1; #2 a =4'h3; #2 en=1'b0;
#2 en=1'b1; #2 a= 4'h4; #2 en=1'b0;
#2 en=1'b1; #2 a=4'hf;
end
initial $monitor("t=%0d, en = %b, a = %b, b = %b",$time,en,a,b);
initial #60 $stop;
endmodule
```





## Block Statement

- A block statement provides a mechanism to group two or more statements to act syntactically like a single statement.
- There are two kinds of blocks in Verilog HDL. They are:
  - i) Sequential block (begin…end): Statements are executed sequentially in the given order.
  - ii) Parallel block (fork…join): Statements in this block execute concurrently.
- A block can be labeled optionally.

## Sequential block

- Statements in a sequential block execute in sequence.
- Sequential blocks have the following characteristics:
  - The statement in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for non-blocking assignments with intra-assignment timing control).
  - If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.





## Sequential block without delay

```
//Illustration 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;

initial
begin
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};
end
```

## Sequential block with delay

```
//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
begin
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 15
        #20 w = {y, x}; //completes at simulation time 35
end
```





## Parallel block

- Statements in a parallel block execute concurrently.
- Parallel blocks have the following characteristics:
  - Statements in a parallel block are executed concurrently
  - Ordering of statements is controlled by the delay or event control assigned to each statement.
  - If delay or event control is specified, it is relative to the time the block was entered.

## Parallel blocks with delay

```
//Example 1: Parallel blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
fork
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 10
        #20 w = {y, x}; //completes at simulation time 20
join
```





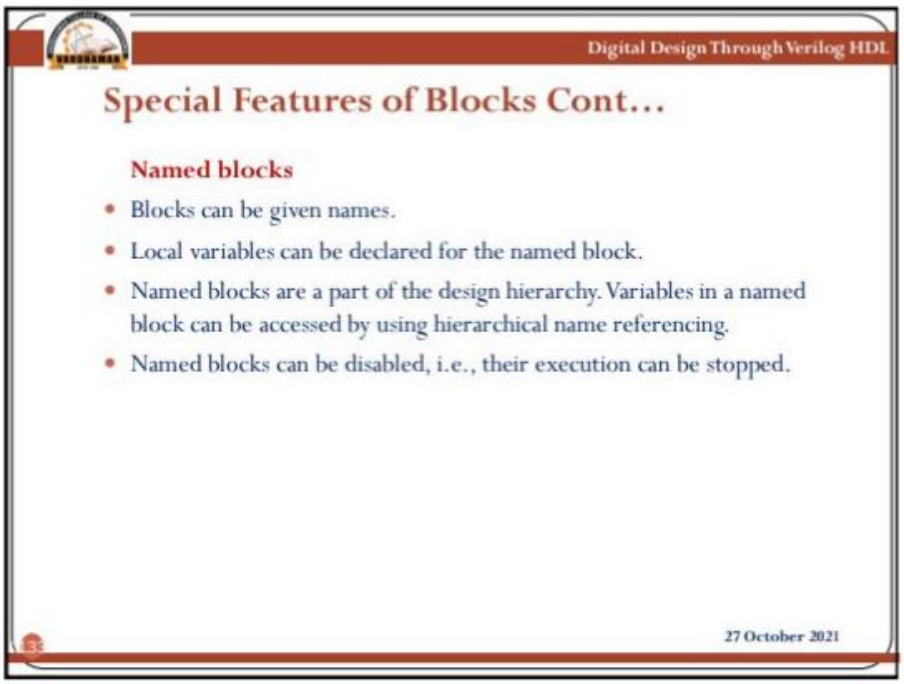
## Special Features of Blocks

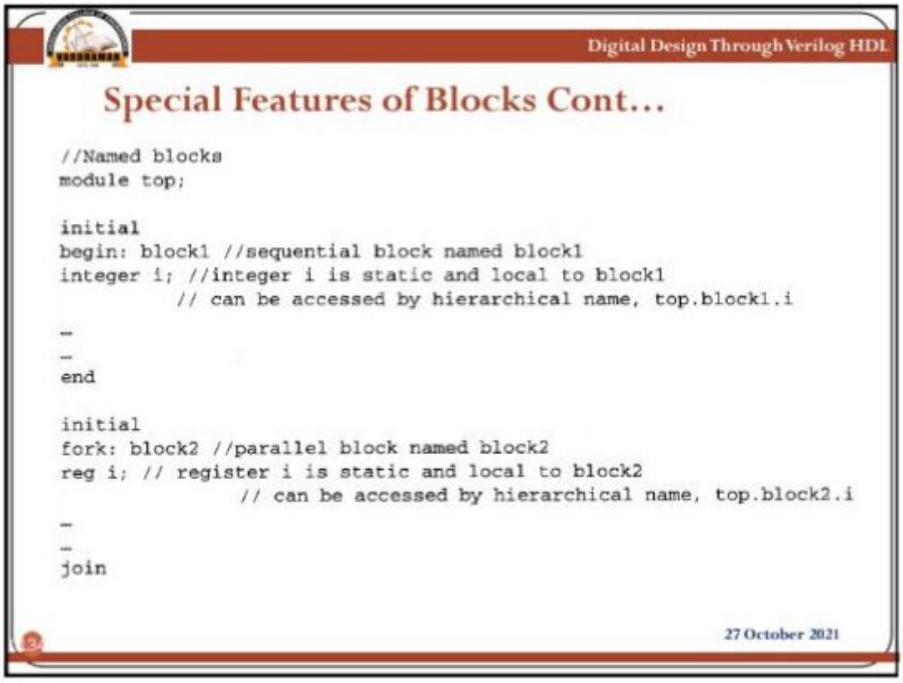
- Three special features available with block statements: **nested blocks, named blocks, disabling of named blocks.**

**Nested blocks**

- Blocks can be nested.
- Sequential and parallel blocks can be mixed.

```
//Nested blocks
initial
begin
        x = 1'b0;
        fork
                #5 y = 1'b1;
                #10 z = {x, y};
        join
        #20 w = {y, x};
end
```


Digital Design Through Verilog HDL
Special Features of Blocks Cont…
Named blocks
• Blocks can be given names.
• Local variables can be declared for the named block.
• Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.
• Named blocks can be disabled, i.e., their execution can be stopped.
27 October 2021


Digital Design Through Verilog HDL
Special Features of Blocks Cont…
//Named blocks
module top;
initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
// can be accessed by hierarchical name, top.block1.i
…
…
end
initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
// can be accessed by hierarchical name, top.block2.i
…
…
join
27 October 2021

## Special Features of Blocks Cont…

**Disabling named blocks**

- The keyword disable provides a way to terminate the execution of a named block.
- Disable can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal.
- Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.





## Generate Blocks

- Generate statements allow verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parameterized models.
- All generate instantiations are coded with a module scope and require the keyword **generate – endgenerate.**
- Generated instantiations can be one or more of the following types:
  - Modules
  - User Defined Primitives
  - Verilog Gate Primitives
  - Continuous Assignments
  - Initial and always blocks

## Generate Blocks Cont…

- There are three methods to create generate statements:
  - Generate Loop
  - Generate Conditional
  - Generate Case
- A generate statement is of the form:

```
generate
//generate loop statements
//generate conditional statements
//generate case statements
//Nested generate statements
endgenerate
```





## Generate Loop

- A generate loop permits one or more of the following to be instantiated multiple times using a for loop:
  - Variable declarations
  - Modules
  - User defined primitives, gate primitives
  - Continuous assignments
  - Initial and always blocks

## Generate Loop Example

- **Bit-wise XOR of two N-bit buses:**

```
module bitwise_xor(out, i0, i1);
parameter n=32;
output [n-1:0] out;
input [n-1:0] i0, i1;
genvar j;
generate for (j=0;j<n;j=j+1)
begin: xor_loop
xor g1(out[j], i0[j], i1[j]);
end
endgenerate
endmodule
```





## Generate Loop Example

- **N-bit Full Adder (Ripple Adder) using Module Instantiation:**

```
module generate_fa(a,b,ci,s,co);
parameter N=4;
input [N-1:0]a,b;
input ci;
output [N-1:0]s;
output co;
wire [N:0]c;
genvar j;
assign c[0]=ci;
generate for(j=0;j<N;j=j+1)
        begin: fa_generate
        fa f1(a[j],b[j],c[j],s[j],c[j+1]);
        end
endgenerate
assign co=c[N];
endmodule
```

## Generate Loop Example

- N-bit Full Adder (Ripple Adder) using Gate Primitives:

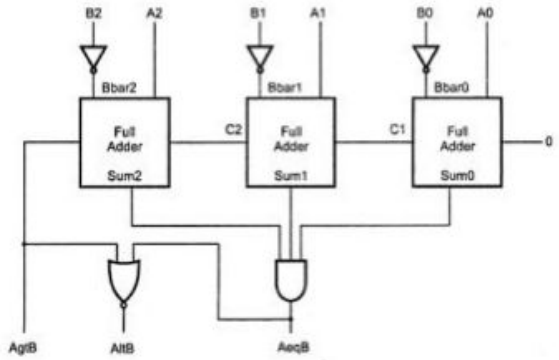
```
module generate_fa(a,b,ci,s,co);
parameter N=4;
input [N-1:0]a,b;
input ci;
output [N-1:0]s;
output co;
wire [N:0]c;
genvar j;
assign c[0]=ci;
generate for(j=0;j<N;j=j+1)
        begin: fa_generate
        wire w1,w2,w3;
        xor x1(w1,a[j],b[j]),x2(s[j],w1,c[j]);
        and a1(w2,a[j],b[j]),a2(w3,w1,c[j]);
        or o1(c[j+1],w2,w3);
        end
endgenerate
assign co=c[N];
endmodule
```





## Generate Loop Example

- Comparator using adders




3 bit comparator using adders

## Generate Loop Example

- **N-bit Comparator:**

```
module gen_comp(a,b,agtb,aeqb,altb);
parameter N=4;
input [N-1:0]a,b;
output agtb,aeqb,altb;
wire [N-1:0]s,bbar;
wire [N:0]c,eq;
assign c[0]=1'b0;
assign eq[0]=1'b1;
generate
genvar j;
for(j=0;j<N;j=j+1)
        begin: comp_block
        not(bbar[j], b[j]);
        fa f1(a[j],bbar[j],c[j],s[j],c[j+1]);
        and a1(eq[j+1],s[j],eq[j]);
        end
        endgenerate
        assign agtb=c[N];
        assign aeqb=eq[N];
        nor n1(altb,aeqb,agtb);
endmodule
```





## Generate Conditional

- A generate conditional is like an if-else-if generate construct that permits the following Verilog constructs to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated:
  - Module
  - User defined primitives, gate primitives
  - Continuous assignments
  - Initial and always blocks

## Generate Conditional Example

- Serial In Serial Out Shift Register:

```
module shift_register(ser_in, clk, rst, ser_out);
parameter N=4;
input ser_in, rst, clk;
output ser_out;
wire [N-1:1]s;
genvar j;
generate for(j=0;j<N;j=j+1)
        begin: block1
        if (j==N-1)
        dff d0(ser_in, rst, clk, s[j]);
        else
        if (j==0)
        dff d1(s[j+1],rst,clk,ser_out);
        else
        dff d2(s[j+1],rst,clk,s[j]);
        end
endgenerate
endmodule
```





## Generate Conditional Example

- Generic Full Adder without input carry:

```
module adder(a,b,sum,cout);
parameter N=4;
input  [N-1:0]a,b;
output [N-1:0]sum;
output cout;
wire [N-1:0]c;
genvar j;
generate for(j=0;j<N;j=j+1)
        begin: block1
        if (j==0)
        ha h1(a[j],b[j],sum[j],c[j]);
        else
        fa f1(a[j],b[j],c[j-1],sum[j],c[j]);
        end
endgenerate
assign cout=c[N-1];
endmodule
```

## Generate Case

- A generate case permits the following Verilog constructs to be conditionally instantiated into another module based on a select-one-of-many case construct that is deterministic at the time the design is elaborated:
  - Modules
  - User defined primitives, gate primitives
  - Continuous assignments
  - Initial and always blocks





## Generate Case Example

- Generic Full Adder without input carry:

```
module adder(a,b,sum,cout);
parameter N=4;
input  [N-1:0]a,b;
output [N-1:0]sum;
output cout;
wire [N-1:0]c;
genvar j;
generate for(j=0;j<N;j=j+1)
        begin: block1
        case (j)
        0: ha h1(a[j],b[j],sum[j],c[j]);
        default:
        fa f1(a[j],b[j],c[j-1],sum[j],c[j]);
        endcase
        end
endgenerate
assign cout=c[N-1];
endmodule
```

## Procedural Continuous Statements

- A procedural continuous assignment is a procedural assignment, that is, it can appear within an always statement or an initial statement.
- This assignment can override all the other assignments to a net or a variable.
- There are two kinds of procedural continuous assignments:
  - **assign** and **deassign** procedural statements
  - **force** and **release** procedural statements





## Procedural Continuous Statements cont...

- An assign procedural statement overrides all procedural assignments to a variable.
- The deassign procedural statement ends the continuous assignment to a variable.
- The value in the variable is retained until assigned again.

## assign – deassign Construct

- It allows continuous assignments within a behavioral block.

Example:

- always@(posedge clk) a = b;
- always@(posedge clk) assign c = d;
- always
  begin
  @(posedge clk) assign c = d;
  @(negedge clk) deassign c;
  end





## assign – deassign Construct

```
module demux1(a0,a1,a2,a3,b,s);
output a0,a1,a2,a3;
input b;
input [1:0]s;
reg a0,a1,a2,a3;
always@(s)
if(s==2'b00)
assign {a0,a1,a2,a3}={b,3'oz};
else if(s==2'b01)
assign {a0,a1,a2,a3}={1'bz,b,2'bz};
else if(s==2'b10)
assign {a0,a1,a2,a3}={2'bz,b,1'bz};
else if(s==2'b11)
assign {a0,a1,a2,a3}={3'oz,b};
endmodule
```

```
# t=1, s=01, b=0,  {a0,a1,a2,a3} =0zzz
# t=2, s=01, b=1,  {a0,a1,a2,a3} =z0zz
# t=3, s=10, b=1,  {a0,a1,a2,a3} =z1zz
# t=4, s=10, b=0,  {a0,a1,a2,a3} =zz1z
# t=5, s=11, b=0,  {a0,a1,a2,a3} =zz0z
# t=6, s=11, b=1,  {a0,a1,a2,a3} =zzz0
# t=7, s=00, b=1,  {a0,a1,a2,a3} =zzz1
# t=8, s=00, b=0,  {a0,a1,a2,a3} =1zzz
# t=9, s=01, b=0,  {a0,a1,a2,a3} =0zzz
# t=10, s=01, b=1,  {a0,a1,a2,a3} =z0zz
# t=11, s=10, b=1,  {a0,a1,a2,a3} =z1zz
# t=12, s=10, b=0,  {a0,a1,a2,a3} =zz1z
# t=13, s=11, b=0,  {a0,a1,a2,a3} =zz0z
```

## assign – deassign Construct

```
module tst_demux1();
reg b;
reg[1:0]s;
demux1 d2(a0,a1,a2,a3,b,s);
initial begin b=1'b0;s=2'b0; end
always
begin
#1 s=s+1'b1;
$display("t=%0d, s=%b, b=%b, {a0,a1,a2,a3}=%b",$time,s,b,{a0,a1,a2,a3});
#1b=~b;
$display("t=%0d, s=%b, b=%b, {a0,a1,a2,a3}=%b",$time,s,b,{a0,a1,a2,a3});
end
initial #14 $stop;
endmodule
```





## assign – deassign Construct

```
module dffassign(q,qb,di,clk,clr,pr);
output q,qb;
input di,clk,clr,pr;
reg q;
assign qb=~q;
always@(clr or pr)
begin
if(clr)assign q = 1'b0;//asynchronous clear and
else if(pr) assign q = 1'b1;// preset of FF overrides
else deassign q;// the synchronous behaviour
end
always@(posedge clk)
q = di;//synchronous (clocked)value assigned to q
endmodule
```

```
# t=0, clk=0, clr=1, pr=0,  di=0, q=0
# t=2, clk=1, clr=0, pr=0,  di=0, q=0
# t=4, clk=0, clr=0, pr=0,  di=1, q=0
# t=6, clk=1, clr=0, pr=0,  di=1, q=1
# t=8, clk=0, clr=0, pr=0,  di=0, q=1
# t=10, clk=1, clr=0, pr=0,  di=0, q=0
# t=12, clk=0, clr=0, pr=0,  di=1, q=0
# t=14, clk=1, clr=0, pr=0,  di=1, q=1
# t=16, clk=0, clr=0, pr=1,  di=0, q=1
# t=18, clk=1, clr=0, pr=1,  di=0, q=1
# t=20, clk=0, clr=0, pr=0,  di=1, q=1
# t=22, clk=1, clr=0, pr=0,  di=1, q=1
# t=24, clk=0, clr=0, pr=0,  di=0, q=1
# t=26, clk=1, clr=0, pr=0,  di=0, q=0
# t=28, clk=0, clr=0, pr=0,  di=1, q=0
# t=30, clk=1, clr=0, pr=0,  di=1, q=1
# t=32, clk=0, clr=0, pr=1,  di=0, q=1
# t=34, clk=1, clr=0, pr=1,  di=0, q=1
# t=36, clk=0, clr=0, pr=1,  di=1, q=1
# t=38, clk=1, clr=0, pr=1,  di=1, q=1
# t=40, clk=0, clr=0, pr=0,  di=0, q=1
# t=42, clk=1, clr=0, pr=0,  di=0, q=0
# t=44, clk=0, clr=0, pr=0,  di=1, q=0
```

## assign – deassign Construct

```
module dffassign_tst();
reg di,clk,clr,pr;
wire q,qb;
dffassign dd(q,qb,di,clk,clr,pr);
initial
begin
clr=1'b1;pr=1'b0;clk=1'b0;di=1'b0;
end
always #2 clk=~clk;clr=1'b0;
always # 4 di =~di;
always #16 pr=1'b1;
always #20 pr =1'b0;
initial
$monitor("t=%0d, clk=%b, clr=%b, pr=%b,di=%b, q=%b",$time,clk,clr,pr,di,q);
initial #46 $stop;
endmodule
```





## Procedural Continuous Statements cont…

**Ex: assign and deassign procedural statements**

```
module dff(d,clear,clk,q);
input d, clear, clk;
output reg q;
always @(clear)
if (!clear)
assign  q=0;
else
deassign q;
always @(posedge clk)
q<=d;
endmodule
```

## Procedural Continuous Statements cont…

**Ex. assign and deassign procedural statements**

```
module dff(q,qbar,d,clk,rst);
input d,clk,rst;
output q,qbar;
reg q,qbar;
always @(negedge clk)
begin q<=d;
qbar<=~d;
end
always @(rst)
if (rst)
begin
assign q<=1'b0;
assign qbar<=1'b1;
end
else
begin
deassign q;
deassign qbar;
end
endmodule
```





## Force – release Statements

- The force and release procedural statements can be used to override assignments on both registers and nets.
- **Ex: force and release on registers.**

```
module
…
dff d1(q,qbar,d,clk,rst);
…
initial
begin
#50 force d1.q=1'b1;
#50 release d1.q;
end
…
endmodule
```

```
…
force a = 1'b0;
force b = c&d;
assignment1;
assignment2;
…
release a;
release b;
…
```

## Force – release Statements

- **Ex: force and release on nets.**

```
module
…
assign out=a&b&c;
…
initial
begin
#50 force out=a|b&c;
#50 release out;
end
…
endmodule
```





## Force – release Statements

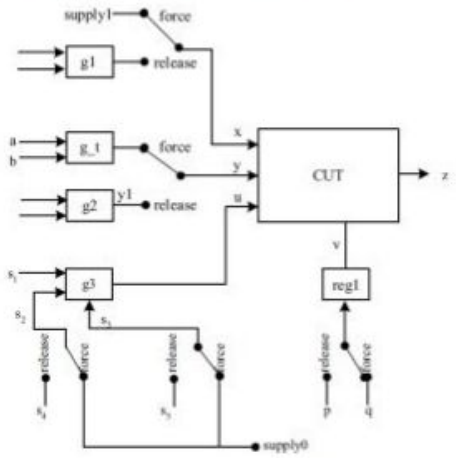



Figure 8.54 A circuit CUT under test with different possibilities of forcing test signals on it.

## Force – release Statements

- They are temporary, for a limited time and for test purposes only.
- Both nets and **regs** can be forced in this manner; that is, their regular values can be overridden.
- When a net is forced to a value, it takes the new value assigned. On release, its previous assignment comes back into effect.
- When a **reg** is forced to a value, it takes the newly assigned value. Even after the release, the newly assigned value continues to hold good until another procedural assignment changes its value.
- Figure 8.54 illustrates a test case for different uses of the **force–release** construct. CUT is a circuit block under test. The design has the following input connections.
- Input x connected to combinational circuit g1
- Input y connected to combinational circuit g2
- Input u connected to combinational circuit g3
- Input v connected to **reg1**





## Force – release Statements

(a)

```
. . .
assignment1;
#10     force cut.x = supply1;
        assignment2;
        assignment3;
#20     release cut.x;
. . .
```

(b)

```
. . .
assignment4;
#10     force cut.y = a & b;
        assignment5;
        assignment6;
#20     release cut.y;
. . .
```

(c)

```
. . .
assignment7;
#10     force cut.s2 = supply0;
        force cut.s3 = supply0;
        assignment8;
        assignment9;
#20     release cut.s2;
        release cut.s3;
        . . .
```

(d)

```
. . .
assignment10;
#10 force cut.v = q;
assignment11;
assignment12;
#20 release cut.v;
#5 assignment13;
cut.v = p;
. . .
```