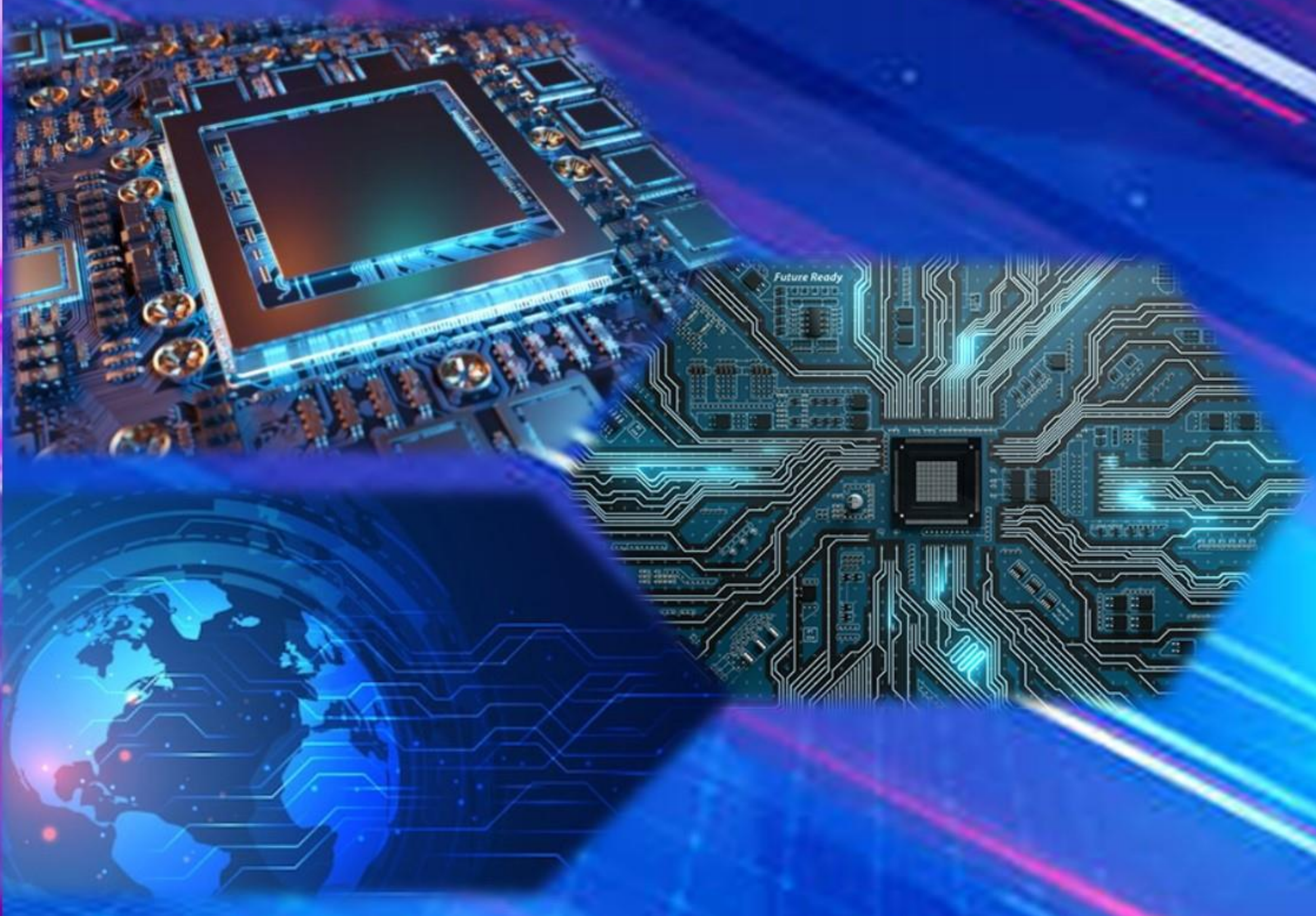
VERILOG

HARDWARE DESCRIPTION LANGUAGE



KSAM

Verilog HDL

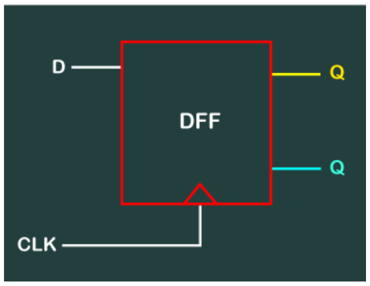
KSAM

VERILOG

Verilog is a Hardware Description Language (HDL). It is a language used for describing a digital system such as a network switch, a microprocessor, a memory, or a flip-flop. We can describe any digital hardware by using HDL at any level. Designs described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

What is Verilog?

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL), which is used to describe a digital system such as a network switch or a microprocessor or a memory a flip-flop.



Verilog was developed to simplify the process and make the HDL more robust and flexible. Today, Verilog is the most popular HDL used and practiced throughout the semiconductor industry.

HDL was developed to enhance the design process by allowing engineers to describe the desired hardware's functionality and let automation tools convert that behavior into actual hardware elements like combinational gates and sequential logic.

Verilog is like any other hardware description language. It permits the designers to design the designs in either Bottom-up or Top-down methodology.

- **Bottom-Up Design:** The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standards gates. This design gives a way to design new structural, hierarchical design methods.
- **Top-Down Design:** It allows early testing, easy change of different technologies, and structured system design and offers many other benefits.

Verilog Abstraction Levels

Verilog supports a design at many levels of abstraction, such as:

- Behavioral level

- Register-transfer level
- Gate level

Behavioral level

The behavioral level describes a system by concurrent algorithms behavioral. Every algorithm is sequential, which means it consists of a set of executed instructions one by one. Functions, tasks, and blocks are the main elements. There is no regard for the structural realization of the design.

Register-Transfer Level

Designs using the Register-Transfer Level specify a circuit's characteristics using operations and the transfer of data between the registers.

The modern definition of an RTL code is "Any code that is synthesizable is called RTL code".

Gate Level

The characteristics of a system are described by logical links and their timing properties within the logical level. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`).

The usable operations are predefined logic primitives (basic gates). Gate level modeling may not be the right idea for logic design. Gate level code is generated using tools such as synthesis tools, and his netlist is used for gate-level simulation and backend.

History of Verilog

- Verilog HDL's history goes back to the 1980s when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and a hardware description language.
- Cadence Design Systems acquired Gateway in 1989 and with it the rights to the language and the simulator. In 1990, Cadence put the language into the public domain, with the intention that it should become a standard, non-proprietary language.
- The Verilog HDL is now maintained by a nonprofit making organization, Accellera, formed from the merger of Open Verilog International (OVI) and VHDL International. OVI had the task of taking the language through the IEEE standardization procedure.
- In December 1995, Verilog HDL became IEEE Std. 1364-1995. A significantly revised version was published in 2001: IEEE Std. 1364-2001. There was a further revision in 2005, but this only added a few minor changes.
- Accellera has also developed a new standard, SystemVerilog, which extends Verilog.
- SystemVerilog became an IEEE standard (1800-2005) in 2005.

How is Verilog useful?

Verilog creates a level of abstraction that helps hide away the details of its implementation and technology.

For example, a D flip-flop design would require the knowledge of how the transistors need to be arranged to achieve a positive-edge triggered FF and what the rise, fall, and CLK-Q times required to latch the value onto a flop among much other technology-oriented details.

Power dissipation, timing, and the ability to drive nets and other flops would also require a more thorough understanding of a transistor's physical characteristics. Verilog helps us to focus on the behavior and leave the rest to be sorted out later.

Lexical Tokens

Lexical conventions in Verilog are similar to the C programming language. Verilog language source text files are a stream of lexical tokens.

A lexical token may consist of one or more characters, and every single character is in exactly one token.

The tokens can be keywords, comments, numbers, white space, or strings. All lines should be terminated by a semi-colon (;).

- Verilog HDL is a case-sensitive language.
- And all keywords are in lowercase.

White Space

White space can contain the characters for tabs, blanks, newlines, and form feeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

Comments

There are two types to represent the comments, such as:

1. Single line comments begin with the token // and end with a carriage return. For example, //this is the single-line syntax.
2. Multi-Line comments begin with the token /* and end with the token */ For example, /* this is multiline syntax*/

Numbers

We can specify constant numbers in binary, decimal, hexadecimal, or octal format. Negative numbers are represented in 2's complement form. The question mark (?) character is the Verilog alternative for the z character when used in a number. The underscore character (_) is legal anywhere in a number, but it is ignored as the first character.

1. Integer Number

Verilog HDL allows integer numbers to be specified as:

- Sized or unsized numbers (Unsized size is 32 bits).
- In a radix of decimal, hexadecimal, binary or octal.
- Radix and hex digits (a,b,c,d) are case insensitive.
- Spaces are allowed between the radix, size, and value.

Syntax

The syntax is given as:

1. <size>'<radix><value>

2. Real Numbers

- Verilog supports real constants and variables.
- Verilog converts real numbers to integers by rounding.
- Real Numbers can not contain 'A' and 'X'.
- Real numbers may be specified in either decimal or scientific notation.
- < value >.< value >
- < mantissa >E< exponent >
- Real numbers are rounded off to the nearest integer when assigning to an integer.

3. Signed and Unsigned Numbers

Verilog supports both the type of numbers, but with certain restrictions. In C language, we don't have int and unint types to say if a number is signed integer or unsigned integer.

Any number that does not have a negative sign prefix is positive. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers. Verilog internally represents negative numbers in 2's complement format. An optional signed specifier can be added for signed arithmetic.

4. Negative Numbers

Negative numbers are specified by placing a minus (-) sign before the size of a number. It is illegal to have a minus sign between base_format and number.

Identifiers

The identifier is the name used to define the object, such as a function, module, or register. Identifiers should begin with alphabetical characters or underscore characters.

For example, A_Z and a_z.

Identifiers are a combination of alphabetic, numeric, underscore, and $ characters. They can be up to 1024 characters long.

- Identifiers must begin with an alphabetic character or the underscore character (**a-z A-Z_**).
- Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (**a-z A-Z 0-9 _ $**).
- Identifiers can be up to 1024 characters long.

1. Escaped Identifiers

Verilog HDL allows any character to be used in an identifier by escaping the identifier.

Escaped identifiers are including any of the printable ASCII characters in an identifier.

- The decimal values 33 through 126, or 21 through 7E in hexadecimal.
- Escaped identifiers begin with the backslash (\). The backslash escapes the entire identifier.
- The escaped identifier is terminated by white space characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by white space.
- Terminate escaped identifiers with white space. Otherwise, characters that should follow the identifier are considered part of it.

Operators

Operators are special characters used to put conditions or to operate the variables. There are one, two, and sometimes three characters used to perform operations on variables.

1. Arithmetic Operators

These operators perform arithmetic operations. The + and -are used as either unary (x) or binary (z-y) operators.

The operators included in arithmetic operation are addition, subtraction, multiplication, division, and modulus.

2. Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0. The Operators included in relational operation are:

- == (equal to)
- != (not equal to)
- (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

3. Bit-wise Operators

Bit-wise operators do a bit-by-bit comparison between two operands. The Operators included in Bit-wise operation are:

- & (Bit-wise AND)
- | (Bit-wiseOR)
- ~ (Bit-wise NOT)
- ^ (Bit-wise XOR)
- ~^ or ^~(Bit-wise XNOR)

4. Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or groups of bits, expressions and treat all non-zero values as 1.

Logical operators are generally used in conditional statements since they work with expressions. The operators included in Logical operation are:

- ! (logical NOT)
- && (logical AND)
- || (logical OR)

5. Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value. The operators included in Reduction operation are:

- & (reduction AND)

- | (reduction OR)
- ~& (reduction NAND)
- ~| (reduction NOR)
- ^ (reduction XOR)
- ~^ or ^~(reduction XNOR)

6. Shift Operators

Shift operators are shifting the first operand by the number of bits specified by the second operand in the syntax.

Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension). The Operators included in Shift operation are:

- << (shift left)
- >> (shift right)

7. Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector. The operator included in Concatenation operation is:

- { }(concatenation)

8. Replication Operator

The replication operator is making multiple copies of an item. The operator used in Replication operation is:

- {n{item}} (n fold replication of an item)

9. Conditional Operator

Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition. The operator used in Conditional operation is:

- (Condition) ?

Operands

Operands are expressions or values on which an ***operator*** operates or works. All expressions have at least one operand.

1. Literals

Literals are constant-valued operands that are used in Verilog expressions. The two commonly used Verilog literals are:

- **String**: A literal string operand is a one-dimensional array of characters enclosed in double quotes (" ").
- **Numeric**: A constant number of the operand is specified in binary, octal, decimal, or hexadecimal number.

2. Wires, Regs, and Parameters

Wires, regs, and parameters are the data types used as operands in Verilog expressions. Bit-Selection "x[2]" and Part-Selection "x[4:2]"

Bit-selects and **part-selects** are used to select one bit and multiple bits, respectively, from a wire, regs or parameter vector using square brackets "[ ]".

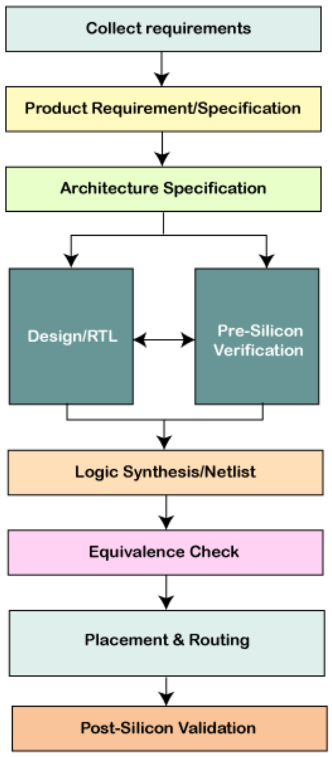
3. Function Calls

In the Function calls, the return value of a function is used directly in an expression without first assigning it to a register or wire.

It just places the function call as one of the types of operands. It is useful to know the bit width of the return value of the function call.

ASIC Design Flow

A typical design flow follows the below structure and can be broken down into multiple steps. Some of these phases happen in parallel and some in sequentially.



Requirements

A customer of a semiconductor firm is typically some other company who plans to use the chip in its systems or end products. So, the customer's requirements also play an important role in deciding how the chip should be designed.

The first step is to collect the requirements, estimate the end product's market value, and evaluate the number of resources required to do the project.

Specifications

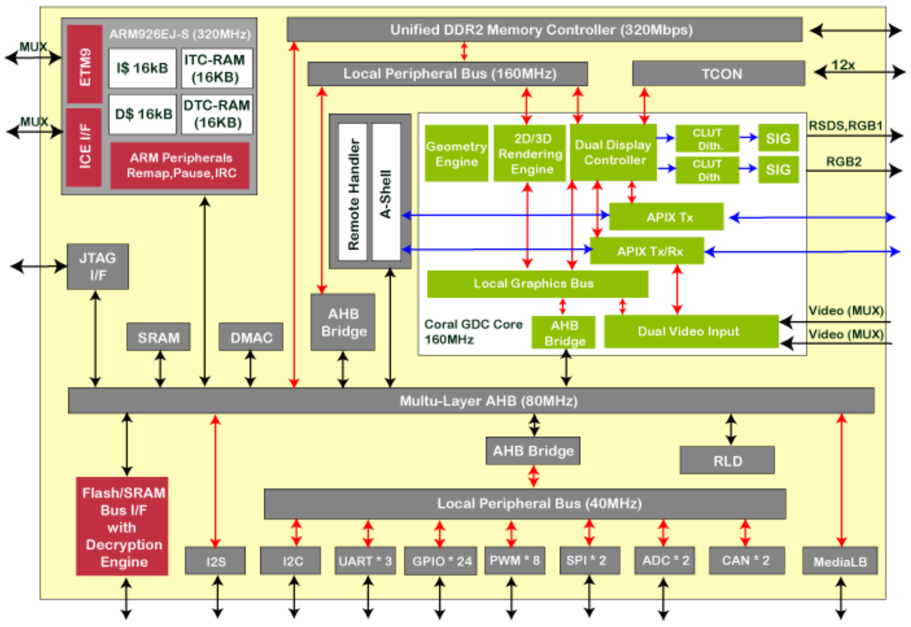
The next step is to collect specifications that describe the functionality, interface abstractly, and over all architecture of the chip to be designed. This can be something along the lines such as:uires computational power to run imaging algorithms to support virtual reality.

1. Requires two *ARM A53* processors with coherent interconnect and should run at 600 MHz.
2. Requires USB 3.0, Bluetooth, and *PCIe* 2nd gen interfaces.
3. It should support 1920x1080 pixel displays with an appropriate controller.

Architecture

Now, the architect gives a system-level view of how the chip should operate. They will decide what all other components are required, what clock frequencies they should run, and how to target power and performance requirements.

They also decide on how the data should flow inside the chip. An example would be the data flow when a processor fetches imaging data from the system ram and executes them. Meanwhile, the graphics engine will execute post-processed data from the previous batch dumped into another part of memory and so on.



Digital Design

Because of the complex nature of modern chips, it's impossible to build something from scratch, and in many cases, many components will be reused.

For example, company A requires a *FlexCAN* module to interact with other modules in an automobile. They can either buy the *FlexCAN* design from another company to save time and effort or spend resources to build one.

It's not practical to design such a system from basic building blocks such as flip-flops and *CMOS* transistors.

Instead, a behavioral description is developed to analyze the design in terms of functionality, performance, and other high-level issues using a Hardware Description Language such as Verilog or VHDL.

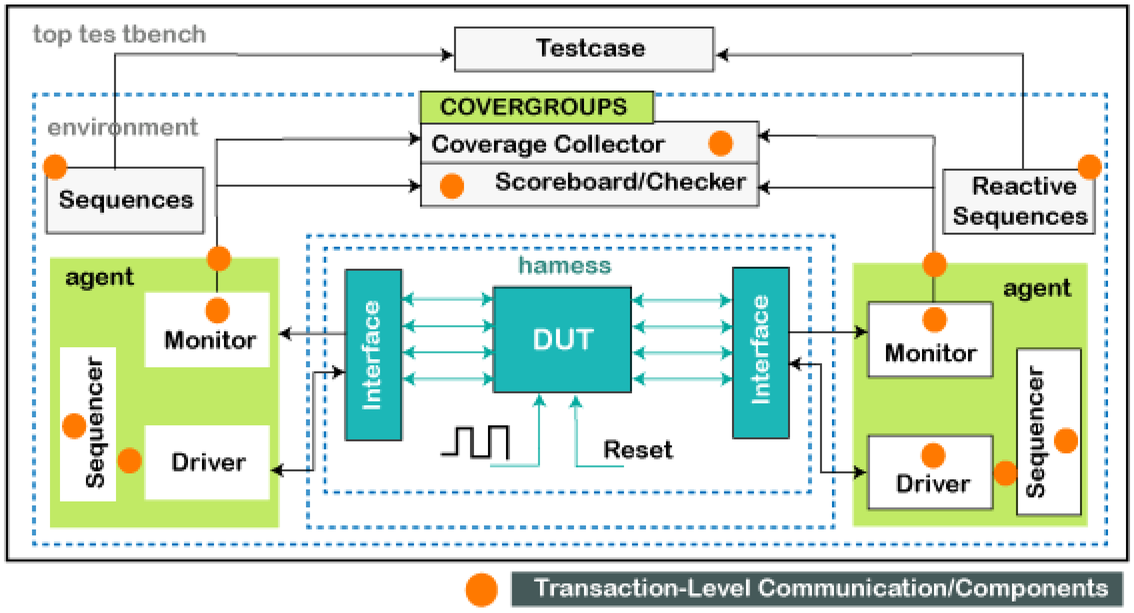
This is usually done by a digital designer and is similar to a high-level computer programmer equipped with digital electronics skills.

Verification

Once the RTL design is ready, it needs to be verified for functional correctness.

For example, a DSP processor is expected to issue bus transactions with fetching instructions from memory and know that this will happen as expected.

The functional verification is required at this point, which is done with *EDA* simulators' help that can model the design and apply a different stimulus to it. This is the job of a pre-silicon verification engineer.



To save time and reach functional closure, both the design and verification teams operate in parallel, where the designers release an *RTL* version. The verification team develops a *testbench* environment and test cases to test the functionality of that RTL version.

If any of these tests fail, it might indicate a problem with the design, and a "bug" will be raised on that design element. This bug will have to be fixed in the next version of the RTL release from the design team.

This process goes on until there is a good level of confidence in the design's functional correctness.

Logic Synthesis

Now we will convert this design into hardware schematic with real elements such as combinational gates and flip-flops. This step is called synthesis.

Logic synthesis tools enable the conversion of RTL description in HDL to a gate-level netlist. This netlist is a description of the circuit in terms of gates and connections between them.

Logic synthesis tools ensure that the netlist meets timing, area, and power specifications. Typically, they have access to different technology node processes and digital elements libraries and can make intelligent calculations to meet all these different criteria.

These libraries are obtained from semiconductor fabs that provide data characteristics for different components such as rise or fall times for flip-flops, input-output time for combinational gates, etc.

Logic Equivalence

The gate-level netlist is checked for logical equivalence with the RTL. Sometimes, a gate-level verification is performed where verification of certain elements is done once again, the difference being this time it is at the gate level and a lower level of abstraction.

Simulation times tend to be slower because of the huge number of elements involved in the design and back annotated delay information.

Placement and Routing

Then, the netlist is inputted to the physical design flow, where automatic place and the route are done with EDA tools' help. The *Cadence Encounter* and *Synopsys IC Compiler* are good examples of these kinds of tools.

This will select and place standard cells into rows, define ball maps for input and output, create different metal layers, and place buffers to meet timing.

Once this process is done, a layout is generated and usually sent for fabrication. This stage is usually handled by the physical design team, who are well familiar with the technology node and physical implementation details.

Validation

A sample chip will be made-up either by the same semiconductor firm or sent to a third-party such as *TSMC* or *Global Foundries*.

This sample now goes through a post-silicon validation process where another team of engineers runs different tester patterns. It is more difficult to debug in post-silicon validation than pre-silicon verification simply because the level of visibility into a chip's internal nodes is drastically reduced.

A million clock cycles would have finished in a second, and tracing back to the exact time of error will be time-consuming.

If there are any real issues or design bugs found at this stage, this will have to be fixed in RTL, re-verified, and all the steps that follow this will have to be performed.

Even though there are multiple steps in the design flow, a lot of the design activity is usually concentrated on the optimization and verification of the RTL description of the circuit.

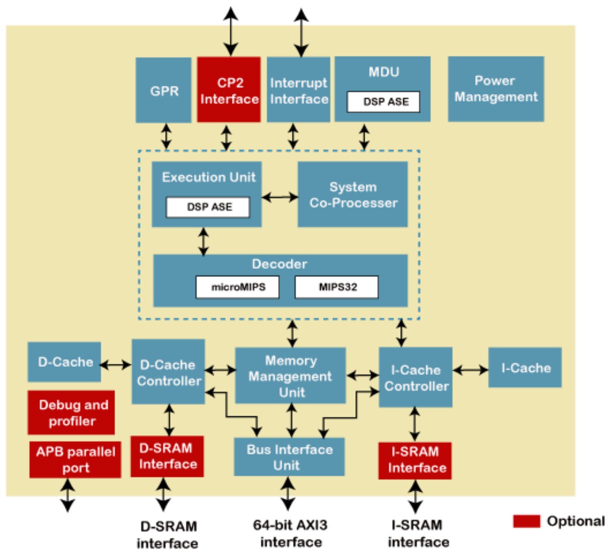
It is important to note that although EDA tools are available to automate the processes, improper usage will lead to inefficient designs. Hence, a designer has to make conscious choices during the design process.

Design Abstraction Layers

The Verilog language would be essential to understand the different layers of abstraction in chip design.

The top layer is the system-level architecture that defines the various sub-blocks and groups them based on functionality.

For example, a processor cluster can have multiple cache blocks, cores, and cache coherence logic. All of this will be represented as a single block with input and output signals.



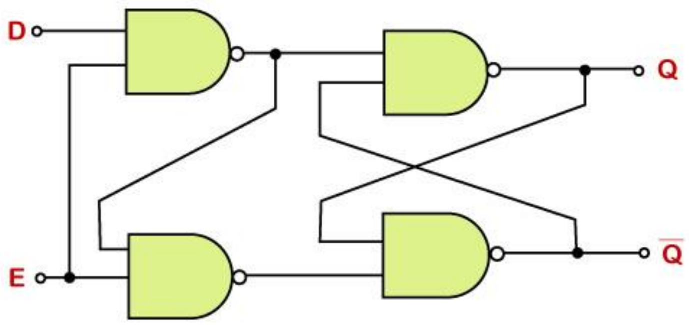
On the next level, each sub-block is written in a hardware description language to describe each block's functionality accurately.

Lower level implementation details such as circuit schematics, technology libraries are ignored at this stage.

For example, a controller block will have multiple Verilog files, each describing a smaller functionality component.

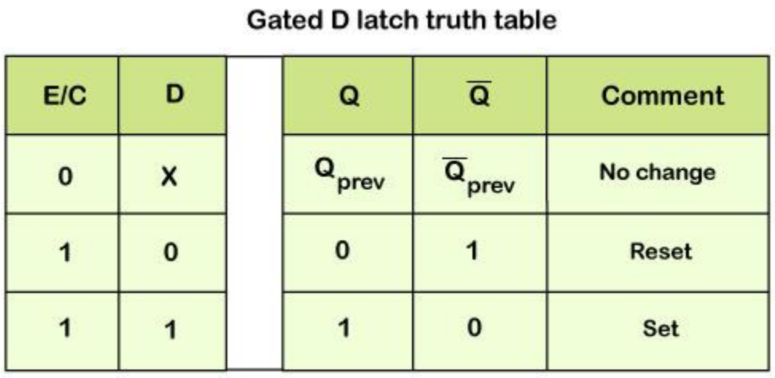
HDLs are then converted to gate-level schematics that involve technology libraries that characterize digital elements such as flip-flops.

For example, the digital circuit for a D latch contains NAND gates arranged in a certain manner such that all combinations of D and E inputs produce an output Q given by the truth table.



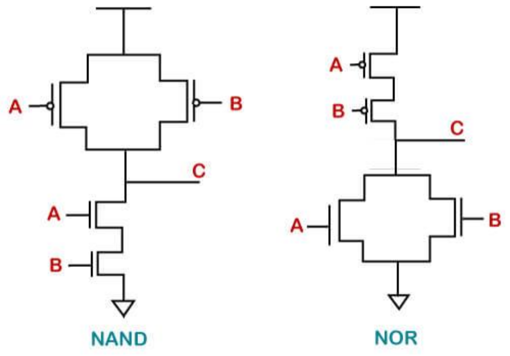
A truth table essentially gives permutation of all input signal levels and the resulting output level.

The hardware schematic can also be derived from the truth table using K-maps and Boolean logic. However, it is not useful to follow this method for more complex digital blocks *like controllers* and *processors*.


Gated D latch truth table

E/C	D		Q	$\overline{Q}$	Comment
0	X		Q_{prev}	$\overline{Q}_{prev}$	No change
1	0		0	1	Reset
1	1		1	0	Set

Implementation of a NAND gate is done by the connection of CMOS transistors in a particular format. At this level, the transistor channel widths, Vdd, and the ability to drive the output capacitative load are taken into account during the design process.



The final step is the layout of these transistors in silicon using EDA tools to be fabricated. Some device and technology knowledge would be required at this level because different layouts end up having different physical properties like resistance and capacitance, among other implications.

Design Styles

There are primarily two styles followed in the design of digital blocks, one is top-down, and another is bottom-up methodologies.

1. Top-Down
In this methodology, a top-level block is first defined along with identifying sub-modules required to build the top block.

Similarly, each sub-blocks is further divided into smaller components, and the process continues until we reach the leaf cell or a stage where it can't be further divided.

2. Bottom-up
The first task is to identify the available building blocks. Then put them together and connected in a certain way to build bigger cells and used to piece together the top-level block.

We can also use the combination of both flows. Architects define the system-level view of the design, and designers implement each of the functional blocks' logic and get synthesized into gates.

A top-down style is followed until this point. However, these gates have been built by following a bottom-up flow, starting with the smallest block's physical layout in the best possible area, power, and performance.

These standard cells also have a hardware schematic. And these can be used to obtain various information such as rise and fall in power, times, and other delays.

These cells are made available to the synthesis tool, which picks and instantiates them where required.

Verilog Data Types

Verilog introduces several new data types. These data types make RTL descriptions easier to write and understand.

The data storage and transmission elements found in digital hardware are represented using a set of Verilog Hardware Description Language (HDL) data types.

In Verilog, data types are divided into *NETS* and *Registers*. These data types differ in the way that they are assigned and hold values, and also they represent different hardware structures.

The Verilog HDL value set consists of four basic values:

Value	Description

0	Logic zero or false
1	Logic one or true
X	Unknown logical value
Z	The high impedance of the tri-state gate

Integer and Real Data Types

Many data types will be familiar to C programmers. The idea is that algorithms modeled in C can be converted to Verilog if the two languages have the same data types.

Verilog introduces new two-state data types, where each bit is 0 or 1 only. Using *two-state* variables in RTL models may enable simulators to be more efficient. And they are not affecting the synthesis results.

Types	**Description**
bit	user-defined size
byte	8 bits, signed
shortint	16 bits, signed
int	32 bits, signed
longint	64 bits, signed

- **Two-state integer types**

Unlike in C, Verilog specifies the number of bits for the fixed-width types.

Types	**Description**
reg	user-defined size
logic	identical to reg in every way

integer	32 bits, signed

- **Four-state integer types**

We preferred *logic* because it is better than *reg*. We can use logic where we have used reg or *wire*.

Type	Description
time	64-bit unsigned
shortreal	like a float in C
shortreal	like double in C
realtime	identical to real

Non-Integer Data Types

Arrays

In Verilog, we can define scalar and vector *nets* and *variables*. We can also define *memory arrays*, which are one-dimensional arrays of a variable type.

Verilog allowed multi-dimensioned arrays of both nets and variables and removed some of the restrictions on memory array usage.

Verilog takes this a stage further and refines the concept of arrays and permits more operations on arrays.

In Verilog, arrays may have either *packed* or *unpacked* dimensions, or both.

Packed dimensions

- Are guaranteed to be laid out contiguously in memory.
- It can be copied on to any other packed object.
- Can be sliced ("part-selects").
- Are restricted to the "bit" types (bit, logic, int, etc.), some of which (e.g., int) have a fixed size.

Unpacked dimensions

It can be arranged in memory in any way that the simulator chooses. We can reliably copy an array on to another array of the same type.

For arrays with different types, we have to use a cast, and there are rules for how an unpacked type is cast to a packed type.

Verilog permits several operations on complete unpacked arrays and slices of unpacked arrays.

For these, the arrays or slices involved must have the same type and shape, i.e., the same number and lengths of unpacked dimensions.

The packed dimensions may differ, as long as the array or slice elements have the same number of bits. The permitted operations are:

- Reading and writing the whole array.
- Reading and writing array slices.
- Reading and writing array elements.
- Equality relations on arrays, slices, and elements

Verilog also includes *dynamic* arrays (the number of elements may change during simulation) and *associative* arrays (which have a non-contiguous range).

Verilog includes several arrays of querying functions and methods to support all these array types.

Nets

Nets are used to connect between hardware entities like logic gates and hence do not store any value.

The net variables represent the physical connection between structural entities such as logic gates. These variables do not store values except trireg. These variables have the value of their drivers, which changes continuously by the driving circuit.

Some net data types are *wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1,* and *trireg*. A net data type must be used when a signal is:

- The output of some devices drives it.
- It is declared as an input or in-out port.
- On the left-hand side of a continuous assignment.

1. Wire

A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block.

A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

2. Wand (wired-AND)
The value of a wand depends on logical AND of all the drivers connected to it.

3. Wor (wired-OR)
The value of wor depends on the logical OR of all the drivers connected to it.

4. Tri (three-state)
All drivers connected to a tri must be z, except one that determines the tri's value.

5. Supply0 and Supply1
Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power).

Registers

A register is a data object that stores its value from one procedural assignment to the next. They are used only in functions and procedural blocks.

An assignment statement in a procedure acts as a trigger that changes the value of the data storage element.

Reg is a Verilog variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as unsigned numbers, and no sign extension is done for what the user might have thought were two's complement numbers.

Some register data types are *reg*, integer, time, and *real.reg* is the most frequently used type.

- **Reg** is used for describing logic.
- **An integer** is general-purpose variables. They are used mainly loops-indices, parameters, and constants. They store data as signed numbers, whereas explicitly declared reg types store them as unsigned. If they hold numbers that are not defined at compile-time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.
- **Real** in system modules.
- **Time** and **realtime** for storing simulation times in test benches. Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time.

Note: A reg need not always represent a flip-flop because it can also represent combinational logic.

- The reg variables are initialized to x at the start of the simulation. Any wire variable not connected to anything has the x value.
- The size of a register or wire may be specified during the declaration.

- When the reg or wire size is more than one bit, then register and wire are declared vectors.

Verilog String

Strings are stored in reg, and the width of the reg variable has to be large enough to hold the string.

Each character in a string represents an ASCII value and requires 1 byte. If the variable's size is smaller than the string, then Verilog truncates the leftmost bits of the string. If the variable's size is larger than the string, then Verilog adds zeros to the left of the string.

Behavioral Modelling and Timing

In Verilog, Behavioral models contain procedural statements, which control the simulation and manipulate variables of the data types.

These statements are contained within the procedures. Each of the procedures has an activity flow associated with it.

During the behavioral model simulation, all the flows defined by the *always* and *initial* statements start together at simulation time *zero*.

The *initial* statements are executed once, and the *always* statements are executed repetitively.

Example

The register variables **a** and **b** are initialized to binary 1 and 0 respectively at simulation time zero.

The initial statement is completed and not executed again during that simulation run. This initial statement is containing a begin-end block of statements. In this begin-end type block, **a** is initialized first, followed by **b**.

```
module behave;
reg [1:0]a,b;

initial
begin
  a = 'b1;
  b = 'b0;
end

always
begin
  #50 a = ~a;
```

- end
-
- always
- begin
-    #100 b = ~b;
- end
- End module

Procedural Assignments

Procedural assignments are for updating **integer, reg, time,** and **memory** variables. There is a significant difference between a procedural assignment and continuous assignment, such as:

1. Continuous assignments drive net variables, evaluated, and updated whenever an input operand changes value.

The procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

2. The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indices. The left-hand side indicates the variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

- Register, integer, real, or time variable: An assignment to the name reference of one of these data types.
- Bit-select of a register, integer, real, or time variable: An assignment to a single bit that leaves the other bits untouched.
- Part-select of a register, integer, real, or time variable: A part-select of two or more contiguous bits that leave the rest of the bits untouched. For the part-select form, only constant expressions are legal.
- Memory element: A single word of memory. Bit-selects and part-selects are illegal on memory element references.
- Concatenation of any of the above: A concatenation of any of the previous four forms can be specified, which effectively partitions the result of the right-hand side expression and then assigns the partition parts, in order, to the various parts of the concatenation.

Delay in Assignment

In a delayed assignment, **Δt** time units pass before the statement is executed, and the left-hand assignment is made. With an intra-assignment delay, the right side is evaluated immediately, but there is a delay of **Δt** before the result is placed in the left-hand assignment.

If another procedure changes a right-hand side signal during Δt, it does not affect the output. Synthesis tools do not support delays.

Syntax

An assignment has the following syntax, such as:

1. Procedural Assignmentvariable = expression
2. Delayed assignment#Δt variable = expression;
3. Intra-assignment delayvariable = #Δt expression;

Blocking Assignments

A blocking procedural assignment statement must be executed before executing the statements that follow it in a sequential block.

The statement does not prevent the execution of statements that follow it in a parallel block.

Syntax

The following syntax is for a blocking procedural assignment, such as:

1. <lvalue> = <timing_control> <expression>

- An lvalue is a data type that is valid for a procedural assignment statement.
- = is the assignment operator, and timing control is the optional intra -assignment delay. The timing control delay can either be a delay control or event control. The expression is the right-hand side value the simulator assigns to the left-hand side. Continuous procedural assignments and continuous assignments also use the = assignment operator used by blocking procedural assignments.

Non-blocking (RTL) Assignments

The non-blocking procedural assignment is used to schedule assignments without blocking the procedural flow.

We can use the non-blocking procedural statement whenever we want to make several register assignments within the same time step without regard to order or dependence upon each other.

Syntax

The following syntax is for a non-blocking procedural assignment:

1. <lvalue> <= <timing_control> <expression>

- An lvalue is a data type that is valid for a procedural assignment statement.

- <= is the non-blocking assignment operator, and timing control is the optional intra-assignment timing control.
  The timing control delay can be either a delay control or event control. The expression is the right-hand side value the simulator assigns to the left-hand side. The non-blocking assignment operator is the same operator the simulator uses for the less-than-or equal relational operator.
- The simulator interprets the <= operator as a relational operator when we use it in an expression and interprets the <= operator as an assignment operator when you use it in a non-blocking procedural assignment construct.

When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and executes the non-blocking procedural assignment in two steps:

Step 1: The simulator evaluates the right-hand side and schedules the new value assignment at a time specified by a procedural timing control.

Step 2: At the end of the time step, when the given delay has expired, or the appropriate event has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

Conditions

The conditional statement or if-else statement is used to decide whether a statement is executed.

Syntax

The syntax is as follows:

- <statement>
- ::= **if** (<expression>) <statement_or_null>
- ||= **if** (<expression>) <statement_or_null>
- **else** <statement_or_null>
- <statement_or_null>
-
- ::= <statement>
- ||= ;

- The <expression> is evaluated. If it is true (non-zero known value), then the first statement executes. If it is false (zero value or the value is x or z), then the first statement does not execute.
- If there is an else statement and <expression> is false, then the else statement executes.
- Since the numeric value of the, if expression is tested for being zero, specific shortcuts are possible.

Case Statement

The case statement is a unique multi-way decision statement that tests whether an expression matches several other expressions, and branches accordingly.

The case statement is useful for describing, for example, the decoding of a microprocessor instruction.

Syntax

The case statement has the following syntax:

- <statement>
- ::= **case** (<expression>) <case_item>+ endcase
- ||= casez (<expression>) <case_item>+ endcase
- ||= casex (<expression>) <case_item>+ endcase
- <case_item>
- ::= <expression> <,<expression>>* : <statement_or_null>
- ||= **default** : <statement_or_null>
- ||= **default** <statement_or_null>

- The case expressions are evaluated and compared in the exact order in which they are given.
- During the linear search, if one of the case item expressions matches the expression in parentheses, then the statement associated with that case item is executed.
- If all comparisons fail, and the default item is given, then the default item statement is executed.
- If the default statement is not given, and all of the comparisons fail, none of the case item statements are executed.

The case statement differs from the multi-way if-else-if construct in two essential ways, such as:

1. The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.

2. The case statement provides a definitive result when there are x and z values in an expression.

Looping Statements

There are four types of looping statements. They are used to controlling the execution of a statement zero, one, or more times.

1. Forever continuously executes a statement.

2. Repeat executes a statement a fixed number of times.

3. While executes a statement until expression becomes false, if the expression starts false, the statement is not executed at all.

4. For controls execution of its associated statements by a three-step process are:

Step 1: Executes an assignment normally used to initialize a variable that controls the number of loops executed.

Step 2: Evaluates an expression. Suppose the result is zero, then the for loop exits. And if it is not zero, for loop executes its associated statements and then performs step 3.

Step 3: Executes an assignment normally used to modify the loop control variable's value, then repeats step 2.

Syntax

The following are the syntax rules for the looping statements, such as:

- <statement>
- ::= forever <statement>
- ||=forever
- begin
-    <statement>+
- end
-
-
- <Statement>
- ::= repeat (<expression>) <statement>
- ||=repeat (<expression>)
- begin
-    <statement>+
- end
-
-
- <statement>
- ::= **while** (<expression>) <statement>
- ||=**while** (<expression>)
- begin
-    <statement>+
- end
-
-

- <statement>
- ::= **for** (<assignment> ; <expression> ; <assignment>)
- <statement>
- ||=**for** (<assignment> ; <expression> ; <assignment>)
- begin
-   <statement>+
- end

Delay Controls

Verilog handles the delay controls in the following ways, such as:

1. Delay Control

The execution of a procedural statement can be delay-controlled by using the following syntax:

- <statement>
- ::= <delay_control> <statement_or_null>
- <delay_control>
- ::= # <NUMBER>
- ||= # <identifier>
- ||= # (<mintypmax_expression>)

The following example delays the execution of the assignment by 10-time units.

1. #10 rega = regb;

Execution of the assignment delays by the amount of simulation time specified by the value of the expression.

2. Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register, or the occurrence of a declared event, by using the following event control syntax:

- <statement>
- ::= <event_control> <statement_or_null>
-
- <event_control>
- ::= @ <identifier>
- ||= @ (<event_expression>)
-
- <event_expression>

- ::= <expression>
- ||= posedge <SCALAR_EVENT_EXPRESSION>
- ||= negedge <SCALAR_EVENT_EXPRESSION>
- ||= <event_expression> <or <event_expression>>

*<SCALAR_EVENT_EXPRESSION> is an expression that resolves to a one-bit value.

Value changes on nets and registers can be used as events to trigger the execution of a statement. This is known as detecting an implicit event.

Verilog syntax also used to detect change based on the direction of the change, which is toward the value 1 (posedge) or the value 0 (negedge).

The behavior of posedge and negedge for unknown expression values are:

- A negedge is detected on the transition from 1 to unknown and from unknown to 0.
- And a posedge is detected on the transition from 0 to unknown and from unknown to 1.

Procedures

All procedures in Verilog are specified within one of the following four Blocks.

1. Initial blocks
2. Always blocks
3. Task
4. Function

Initial Blocks

The *initial* and *always* statements are enabled at the beginning of the simulation. The initial blocks execute only once, and its activity dies when the statement has finished.

Syntax

The following syntax is for the initial statement:

- <initial_statement>
- ::= initial <statement>

Example

The following example illustrates the use of the initial statement for the initialization of variables it the starting of simulation.

- Initial

- Begin
- Areg = 0; // initialize a register
- For (index = 0; index < size; index = index + 1)
- Memory [index] = 0; //initialize a memory
- Word
- End

Another usage of the initial Blocks is the specification of waveform descriptions that execute once to provide stimulus to the central part of the circuit being simulated.

- Initial
- Begin
- Inputs = 'b000000;
- // initialize at time zero
- #10 inputs = 'b011001; // first pattern
- #10 inputs = 'b011011; // second pattern
- #10 inputs = 'b011000; // third pattern
- #10 inputs = 'b001000; // last pattern
- End

Always Blocks

The always blocks repeatedly executes. Its activity dies only when the simulation is terminated. There is no limit to the number of initial and always blocks defined in a module.

Syntax

The always statement repeats continuously throughout the whole simulation run. The syntax for the always statement is given below

1. <always_statement>
2. ::= always <statement>

The always statement is only useful when used in conjunction with some form of timing control because of its looping nature.

Task and Function

Tasks and functions are procedures that are enabled by one or more places in other procedures.

Verilog Module

A module is a block of Verilog code that implements certain functionality. Modules can be embedded within other modules, and a higher level module can communicate with its lower-level modules using their input and output ports.

Syntax

A module should be enclosed within *a module* and *endmodule* keywords. The name of the module should be given right after the module keyword, and an optional list of ports may be declared as well.

Note: The ports declared in the list of port declarations cannot be re-declared within the module's body.

- module <name> ([port_list]);
-
-         // Contents of the module
- endmodule
-
-         // A module can have an empty portlist
-
- module name;
-         // Contents of the module
-
- endmodule

All variable declarations, functions, tasks, dataflow statements, and lower module instances must be defined within the module and endmodule keywords.

Purpose of a Module

A module represents a design unit that implements specific behavioral characteristics and will get converted into a digital circuit during synthesis.

Any combination of inputs can be given to the module, and it will provide a corresponding output.

It allows the same *module* to be reused to form more significant modules that implement more complex hardware.

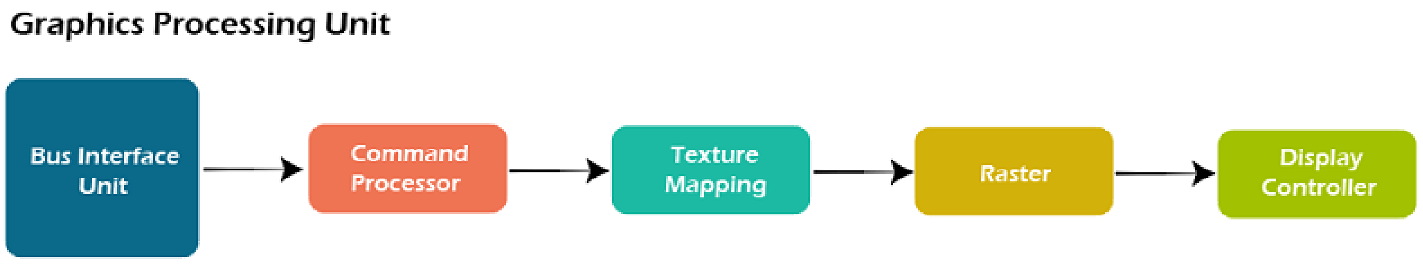
Hardware Schematic

Instead of building up smaller blocks to form bigger design blocks, the reverse process can also be done.

Consider the breakdown of a simple GPU engine into smaller components such that each can be represented as a module that implements a specific feature.

The below GPU engine is divided into five different sub-blocks where each performs a specific functionality.

The bus interface unit gets data from outside into the design, which gets processed by another unit to instructions extraction. Other units down the line process data provided by the previous unit.



Each sub-block can be represented as a module with a specific set of input and output signals for communication with other modules, and each sub-block can be further divided into more sub-sub-blocks as required.

Top-level Modules

A top-level module is one that contains all other modules. A top-level module is not instantiated within any other module.

For example, design modules are usually instantiated within top-level testbench modules so that simulation can be run by providing input stimulus.

But, the *testbench* is not instantiated within any other module because it is a block that encapsulates everything else.

1. Design Top Level

The design code shown below has a top-level module called design. It contains all other sub-modules required to make the design complete.

The sub-module can have a more nested sub-module, such as mod3 inside mod1 and mod4 inside mod2.

```
        //  Design code

module mod3 ( [port_list] );
    reg c;

        // Design code
```

```
endmodule

module mod4 ( [port_list] );
   wire a;
        // Design code
endmodule

module mod1 ( [port_list] );
   wire    y;


mod3    mod_inst1 ( );

   mod3    mod_inst2 ( );

endmodule

module mod2 ( [port_list] );

   mod4    mod_inst1 (  );

   mod4    mod_inst2 (  );

endmodule

      // Top-level module

module design ( [port_list]);

   wire    _net;
   mod1    mod_inst1   (  );

   mod2    mod_inst2   (  );

endmodule
```

2. Testbench Top Level

The testbench module contains a stimulus to check the functionality of the design and primarily used for functional verification by using simulation tools.

Hence the design is instantiated and called d0 inside the testbench module. The testbench is the top-level module from a simulator perspective.

```
//------------
// Testbench code
// this is the top-level module from simulation perspective
// because 'design' is instantiated within this module
//------------
module testbench;
    design d0 ( [port_list_connections] );
//-----------

endmodule
```

Hierarchical Names

A hierarchical structure is formed when modules can be instantiated inside one another, and hence the top-level module is called the *root*.

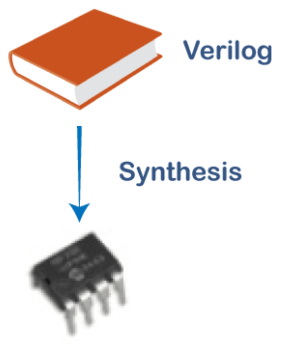
Since each lower module instantiates within a given module, which should have different identifier names, there will not be any ambiguity in accessing signals.

A hierarchical name is constructed by a list of these identifiers separated by dots (.) for each level of the hierarchy. Any signal can be accessed within any module using the hierarchical path to that particular signal.

RTL Verilog

In the digital circuit design, **register-transfer level (RTL)** is a design abstraction which models a synchronous digital circuit in terms of the data flow between hardware register, and the logical operations performed on those signals.

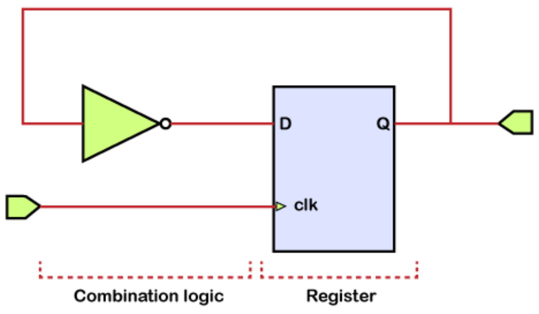
Register-transfer-level abstraction is used in HDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design at the RTL level is a typical practice in modern digital design.



A synchronous circuit consists of two elements, such as:

- **Registers (Sequential logic):** Registers synchronize the circuit's operation to the edges of the clock signal, and are the only elements in the circuit with memory properties. And they are usually implemented as D flip-flops.
- **Combinational logic:** Combinational logic performs all the logical functions in the circuit. And it consists of logic gates.

For example, a simple synchronous circuit is shown in the below image. The inverter is connected from the output Q to the register's input D to create a circuit. It changes its state on each rising edge of the CLK. In this circuit, the combinational logic consists of the inverter.



While designing digital integrated circuits with a hardware description language, the designs are usually arranged at a higher level of abstraction than the transistor level or logic gate level.

In HDLs, the designer declares the registers, which roughly correspond to variables in the programming languages and describes the combinational logic by using constructs such as if-then-else and arithmetic operations.

This level is called the *register-transfer level* or *RTL*. The term RTL focuses on describing the flow of signals between registers.

This description can usually be directly translated into an equivalent hardware implementation file using an EDA tool for synthesis. The synthesis tool also performs logic optimization.

At the register-transfer level, some types of circuits can be recognized. If there is a cyclic path of logic from a register's output to its input, then the circuit is called a *state machine* or sequential logic.

If there are logic paths from a register to another without a cycle, then it is called a *pipeline*.

RTL Circuit Design Cycle

RTL is used in the logic design phase of the integrated circuit design cycle. An RTL description is converted into a gate-level description of the circuit by a logic synthesis tool.

The synthesis results are then used by placement and routing tools to create a physical layout. Logic simulation tools may use a design's RTL description to verify its correctness.

Power Estimation Technique

The most accurate power analysis tools are available for the circuit level, but even with a switch rather than device-level modelling, tools at the circuit level have disadvantages. They are either too slow or require too much memory.

The majorities of these are simulators like SPICE and used by the designers for many years as performance analysis tools

Due to these disadvantages, gate-level power estimation tools have begun to gain some acceptance where faster, probabilistic techniques have begun to gain a foothold.

But it also has its trade-off as speedup is achieved on the cost of accuracy, especially in the presence of correlated signals.

Over the years, it has been realized that the low power design cannot come from the circuit- and gate-level optimizations. In contrast, system, architecture, and algorithm optimizations tend to have the largest impact on power consumption. Therefore, there has been a shift in the tool developers' incline towards high-level analysis and optimization tools for power.

Gate Equivalent Technique

It is a technique based on the concept of gate equivalents. The complexity of chip architecture can be described approximately in terms of gate equivalents, where the equivalent gate count specifies the average number of reference gates that are required to implement the particular function.

The total power required for the particular function is estimated by multiplying the approximated gate equivalents with the average power consumed per gate. The reference gate can be any gate, e.g., 2-input NAND gate. This technique is distributed in the following types, such as:

1. Class Independent Power Modeling: It is a technique which tries to estimate chip area, speed, and power dissipation based on information about the complexity of the design in terms of gate equivalents.

The functionality is divided among different blocks, but no distinction is made about the functionality of the blocks.

It is class independent. This technique is used by the *Chip Estimation System* (CES). This technique completes the following steps:

Step 1: Identify the functional blocks such as *counters, decoders, multipliers, memories*, etc.

Step 2: Assign a complexity in terms of Gate Equivalents. The number of GE's for each unit type are either taken directly as an input from the user or fed from a library.

2. Class Depedent Power Modeling: This approach is slightly better than the previous approach as it takes into account customized estimation techniques to the different types of functional blocks.

Therefore it is trying to increase the modelling accuracy, which wasn't in the case of previous techniques such as *logic, memory, interconnects*, and *clocks*.

The power estimation is done in a very similar manner to the independent case. The basic switching energy is based on a three-input AND gate and is calculated from technology parameters, e.g., gate width, tox, and metal width provided by the user.

Disadvantages

This approach also has the following disadvantages, such as:

1. The circuit activities are not modeled accurately as an overall activity factor is assumed for the entire chip, which is also not trustable as provided by the user.
2. The activity factors will vary throughout the chip; hence this is not very accurate and prone to error. This leads to the problem that even if the model gives a correct estimate for the chip's total power consumption, the module-wise power distribution is relatively inaccurate.
3. The chosen activity factor gives the correct total power, but the breakdown of power into logic, clock, memory, etc. is less accurate.
4. This tool is not much different or improved in comparison with CES.

Pre-characterized Cell Libraries Technique

This technique further customizes the power estimation of various functional blocks by having a separate power model for logic, memory, and interconnects. These suggest a *Power Factor Approximation* (PFA) method for individually characterizing an entire library of functional blocks such as multipliers, adders, etc. instead of a single gate-equivalent model for "logic" blocks.

Advantages

Pre-characterized cell libraries technique provides the following advantages:

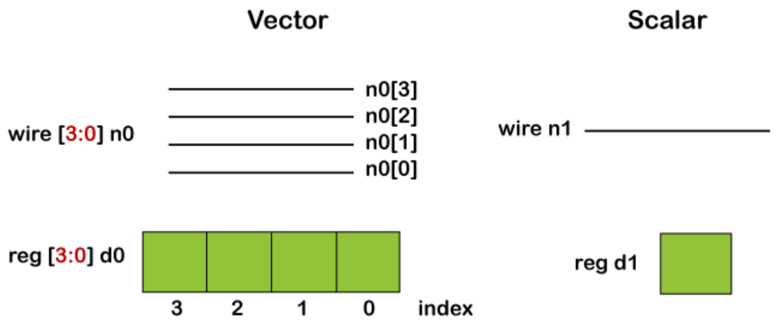
- Customization is possible in terms of whatever complexity parameters which are appropriate for that block. For a multiplier, the square of the word length was appropriate.
- The storage capacity is used in bits for memory, and the word length alone is adequate for the I/O drivers.

Verilog Scalar and Vector

Verilog needs to represent individual bits as well as groups of bits. A single bit sequential element is a flip-flop, and a 16-bit sequential element is a register. For these kinds of tasks, Verilog has *scalar* and *vector*.

Scalar and Vector

A *net* or *reg* declaration without a range specification is considered 1 bit wide and is a scalar. If a range is specified, the net or reg becomes a multibit entity known as a vector.



Vector range specification contains two constant expressions such as:

1. **MSB:** The most significant bit of constant expression, which is the left-hand value of the range.
2. **LSB:** The least significant bit of constant expression, which is the right-hand value of the range.

A colon should separate the MSB and LSB constant expressions.

The MSB constant expression and the LSB constant expression can be any value from positive, negative, and zero.

The LSB constant expression can be higher, equal, or less than the MSB constant expression.

Both the MSB and the LSB expressions should be constant expressions.

Vectors can be declared for all types of *net data types* and *reg* data types. Specifying vectors for *integer, real, realtime*, and *time* data types is illegal. Vector nets and registers are treated as unsigned values.

Syntax

The following is the simplified syntax of the vectors, such as:

- net_type [msb:lsb] list_of_net_identifiers;
- reg [msb:lsb] list_of_register_identifiers;

Examples

- wire   o_nor;   // single bit scalar net
- wire [7:0] o_flop;   // 8-bit vector net
- reg   parity;   // single bit scalar variable
- reg [31:0] addr;   // 32 bit vector variable to store address

The range gives the ability to address individual bits in a vector. The most significant bit of the vector should be specified as the left-hand value in the range. While the least significant bit of the vector should be specified on the right.

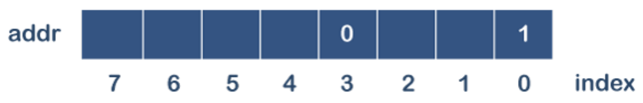
- wire [msb:lsb]   name;
- integer   my_msb;
- wire [15:0]   priority;   // MSB = 15, LSB = 0
- wire [my_msb: 2]   prior;   // illegal

The MSB and LSB should be a constant expression and cannot be substituted by a variable. But they can be any integer value such as positive, negative, or zero.

The LSB value can be higher than, less than, or equal to the MSB value.

Bit Selects

Any bit in a vectored variable can be individually selected and assigned a new value, as shown in the below image. This is called a bit select.



If the bit select is out of bounds or the bit select is x or z, then the value returned will be x.

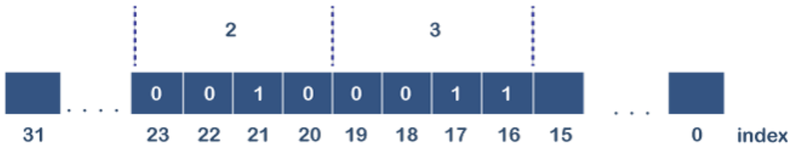
- reg [7:0]   addr;   // 8-bit reg variable [7, 6, 5, 4, 3, 2, 1, 0]
- addr [0] = 1;   // assign 1 to bit 0 of addr
- addr [3] = 0;   // assign 0 to bit 3 of addr
- addr [8] = 1;   // illegal : bit8 does not exist in addr

Part Selects

The selection of the range of contiguous bits is called the part selected. There are two types of part selects.

1. Constant part select

2. Indexed part select



- reg [31:0]   addr;
- addr [23:16] = 8'h23;   // bits 23 to 16 will be replaced by the new value 'h23 -> constant part-select.

A variable part select allows it to be used effectively in loops to select parts of the vector. Although the starting bit can be varied, the width has to be constant.

Syntax

- [<start_bit> +: <width>]   // part-select increments from start-bit
- [<start_bit> -: <width>]   // part-select decrements from start-bit

Example

```
module block;
        reg [31:0]  data;
        int         i;
    initial begin
        data = 32'hFACE_CAFE;
        for (i = 0; i < 4; i++) begin
                $display ("data[8*%0d +: 8] = 0x%0h", i, data[8*i +: 8]);
        end
        $display ("data[7:0]   = 0x%0h", data[7:0]);
        $display ("data[15:8]  = 0x%0h", data[15:8]);
        $display ("data[23:16] = 0x%0h", data[23:16]);
        $display ("data[31:24] = 0x%0h", data[31:24]);
    end
endmodule
```

Verilog Arrays

Verilog arrays are used to group elements into multi-dimensional objects to be manipulated more easily. The Verilog does not have user-defined types, and we are restricted to arrays of built-in Verilog types such as *nets, regs,* and other Verilog variable types.

An array is a collection of the same types of variables and accessed using the same name plus one or more indices.

Each array dimension is declared by having the *min* and *max* indices within the square brackets. Array indices can be written in either direction:

1. array_name[least_significant_index:most_significant_index]
2. array_name[most_significant_index:least_significant_index]

A multi-dimensional array can be declared by having multiple dimensions after the array declaration.

Any square brackets before the array identifier are part of the data type replicated in the array.

Verilog arrays are synthesizable so that we can use them in a synthesizable RTL code.

In C, arrays are indexed from 0 by integers, or converted to pointers. But the whole array can be initialized, and each element must be read or separately written in procedural statements.

In Verilog-2001, arrays are indexed from left-bound to right-bound. If they are vectors, they can be assigned as single units, but not if they are arrays. Verilog-2001 allows for multiple dimensions.

In Verilog-2001, all data types can be declared as arrays. The wire, reg, and all other net types can also have a vector width declared. A dimension declared before the object name is referred to as the *vector width* dimension.

The Verilog-2005 specification also calls a one-dimensional array with elements of type *reg* a memory. It is beneficial for modeling memory elements such as read-only memory (ROM), and random access memory (RAM).

The dimensions declared after the object name is referred to as the *array* dimensions. Arrays hold a fixed number of equally-sized data elements.

Individual elements are accessed by index using a consecutive range of integers. Some arrays allow access to individual elements using non-consecutive values of any data types.

Arrays can be classified as fixed-sized arrays, also known as static arrays whose size cannot change once their declaration is made, or dynamic arrays can be resized.

Verilog had only one type of array. Verilog arrays can be either *packed* or *unpacked*. Packed array refers to dimensions declared after the type and before the data identifier name. Unpacked array refers to the dimensions declared after the data identifier name.

Packed or Fixed Arrays

In Verilog, the term packed array refers to the dimensions declared before the object name.

A one-dimensional packed array is also called a vector. Packed array divides a vector into subfields, which can be accessed as array elements. A packed array is guaranteed to be represented as a contiguous set of bits in simulation and synthesis.

Packed arrays can be made of only the single-bit data types *bit, logic, reg*, enumerated types, and other packed arrays and packed structures. This also means we cannot have packed arrays of integer types with predefined widths.

The maximum size of a packed array can be limited but shall be at least 65536 (216) bits.

A packed array is guaranteed to be represented as a *contiguous* set of bits.

Unpacked Arrays

In Verilog, the term unpacked array is used to refer to the dimensions declared after the object name.

Unpacked arrays can be made of any data type. Each fixed-size dimension is represented by an address range, such as [0:1023].

Or a single positive number to specify the size of a fixed-size unpacked array, such as [1024]. The notation size is equivalent to [0:size-1].

An unpacked array may or may not be so represented as a *contiguous* set of bits.

Multi-dimensional Arrays

Multi-dimensional arrays can be declared with both packed and unpacked dimensions. Creating a multi-dimensional packed array is analogous to slicing up a continuous vector into multiple dimensions.

When an array has multiple dimensions that can be logically grouped, it is useful to use the *typedef* to define the multi-dimensional array in stages to enhance readability.

Verilog Arrays Indexing and Slicing

Verilog arrays could only be accessed one element at a time. In Verilog arrays, we can also select one or more contiguous elements of an array. This is called a *slice*.

An array slice can only apply to one dimension; other dimensions must have single index values in an expression.

Verilog Array Operations

Verilog arrays support many more operations than their traditional Verilog counterparts.

- **+: and -: Notation**

When accessing a range of a Verilog array slice, we can specify a variable slice by using the [start+: increment width] and [start-: decrement width] notations.

They are simpler than needing to calculate the exact start and end indices when selecting a variable slice. The increment or decrement width must be a constant.

- bit signed [31:0] car A [7:0]; // unpacked array of 8 32-bit vectors
- int car B [1:0]; // unpacked array of 2 integers
- car B = car A [7:6]; // select a 2-vector slice from car A
- car B = car A [6+:2]; // equivalent to car A[7:6]

  - **Assignments and Copying Operations**

Verilog arrays support many more operations. The following operations can be performed on both packed and unpacked arrays.

- A = B; // reading and writing the array
- A[i:j] = B[i:j]; // reading and writing a slice of the array
- A[x+:c] = B[y+:d]; // reading and writing a variable slice of the array
- A[i] = B[i]; // accessing an element of the array
- A == B; // equality operations on the array
- A[i:j] != B[i:j]; // equality operations on slice of the array

  - **Packed Array Assignment**

A Verilog packed array can be assigned at once, such as a multi-bit vector, an individual element or slice, and more.

```
• logic [1:0][1:0][7:0] packed_3d_array;
•
• always_ff @(posedge clk, negedge rst_n)
•   if (!rst_n) begin
•     packed_3d_array <= '0;                 // assign 0 to all elements of array
•   end
•
•   else begin
•     packed_3d_array[0][0][0]   <= 1'b0;            // assign one bit
•     packed_3d_array[0][0]      <= A0a;             // assign one element
•     packed_3d_array[0][0][3:0] <= 4'ha;            // assign part select
•     packed_3d_array[0]         <= 16'habcd;        // assign slice
•     packed_3d_array            <= 32'h01234567;    // assign entire array as vector
•   end
```

  - **Unpacked Array Assignment**

All or multiple elements of a Verilog unpacked array can be assigned to a list of values.

The list can contain values for individual array elements, or a default value for the entire array.

```
•  logic [7:0] a, b, c;
•  logic [7:0] d_array[0:3];
•  logic [7:0] e_array[3:0];          // note index of unpacked dimension is reversed
•
•  logic [7:0] mult_array_a[3:0][3:0];
•  logic [7:0] mult_array_b[3:0][3:0];
•
•  always_ff @(posedge clk, negedge rst_n)
•    if (!rst_n) begin
•      d_array <= '{default:0};                  // assign 0 to all elements of array
•    end
•
•    else begin
•      d_array      <= '{A00, c, b, a};           // d_array[0]=A00, d_array[1]=c,
•                                                      d_array[2]=b, d_array[3]=a
•
•      e_array      <= '{A00, c, b, a};            // e_array[3]=A00, e_array[2]=c,
•                                                   e_array[1]=b, d_array[0]=a
•
•      mult_array_a   <= '{'{A00, A01, A02, A03}, '{A04, A05, A06, A07},
•                               '{A08, A09, A0a, A0b},
•                              '{A0c, A0d, A0e, A0f}};        // assign to full array
•      mult_array_b[3] <= '{A00, A01, A02, A03};     // assign to slice of array
•    end
```

Verilog Ports

Port is an essential component of the Verilog module. Ports are used to communicate for a module with the external world through input and output.

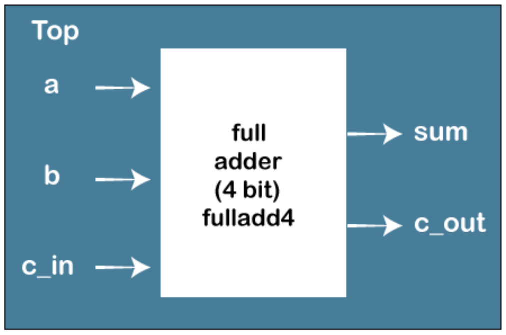
It communicates with the chip through its pins because of a module as a fabricated chip placed on a PCB.

Every port in the port list must be declared as *input, output* or *inout*. All ports declared as one of them is assumed to be wire by default to declare it, or else it is necessary to declare it again.

Ports, also referred to as pins or terminals, are used when wiring the module to other modules.

- 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.
- The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out.



I/O Ports for Top and Full Adder

Port Declaration

Each port in the port list is defined as input, output, or inout based on the port signal's direction.

If a port declaration includes the net or variable types, then that port is considered completely declared. It is illegal to declare the same port in a net or variable type declaration.

And if the port declaration does not include a net or variable type, then the port can be declared again in a net or variable type declaration.

For example, consider the ports for top and full adder shown in the above image.

- 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: By convention, outputs of the module are always first in the port list. This convention is also used in the predefined modules in Verilog.

Wire and Reg

In Verilog, all port declarations are implicitly declared as *wire*. If a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.

Input and inout ports are generally declared as wires. However, if output ports hold their value, they must be declared as *reg* as shown below:

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

NOTE: Ports of the type input ***and*** inout ***cannot be declared as reg.***

Port Connection Rules

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition.

1. Connecting by ordered list: It is the simple method for beginners. The signals to be connected must appear in the module instantiation in the same order as the ports in the module definition.

2. Connecting ports by name: For large designs where modules have approx 50 ports or above. In this situation, remembering the order of the ports in the module definition is complicated and impractical.

Verilog provides the capability to connect external signals to ports by the port names, rather than by position.

Another main reason for 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.

Ports Variations

- Verilog has undergone a few research, and the original IEEE version in *1995* had the following way for port declaration.

Here the module declaration had to first list of the names of ports within the brackets. And then the direction of those ports defined later within the body of the module.

```
module test (a, b, c);

   input   [3:0] a;          // inputs "a" and "b" are wires
   input   [3:0] b;
   output  [3:0] c;          // output "c" by default is a wire

   // Still, we declare them again as wires
   wire   [3:0] a;
   wire   [3:0] b;
   wire   [3:0] c;
endmodule
```

```
module test (a, b, c);

        input  [3:0] a, b;
         output [3:0] c;          // By default c is of type wire

        // port "c" is changed to a reg type

        reg    [3:0] c;

endmodule
```

- ANSI-C style port naming was introduced in *2001*. It allowed the type to be specified inside the port list.

Verilog assign Statement

Assign statements are used to drive values on the net. And it is also used in *Data Flow Modeling*.

Signals of type wire or a data type require the continuous assignment of a value. As long as the +5V battery is applied to one end of the wire, the component connected to the other end of the wire will get the required voltage.

This concept is realized by the assign statement where any wire or other similar wire (data-types) can be driven continuously with a value. The value can either be a constant or an expression comprising of a group of signals.

Syntax

The assignment syntax starts with the keyword assign, followed by the signal name, which can be either a signal or a combination of different signal nets.

The *drive strength* and *delay* are optional and mostly used for dataflow modeling than synthesizing into real hardware.

The signal on the right-hand side is evaluated and assigned to the net or expression of nets on the left-hand side.

```
1. assign <net_expression> = [drive_strength] [delay] <expression of different signals or cons
   tant value>
```

Delay values are useful for specifying delays for gates and are used to model timing behavior in real hardware. The value dictates when the net should be assigned with the evaluated value.

Rules

Some rules need to be followed during the use of an assign statement:

- LHS should always be a scalar, vector, or a combination of scalar and vector nets but never a scalar or vector register.
- RHS can contain scalar or vector registers and function calls.
- Whenever any operand on the RHS changes in value, LHS will be updated with the new value.
- Assign statements are also called continuous assignments.

Assign reg Variables

We cannot drive or assign *reg* type variables with an assign statement because a reg variable is capable of storing data and is not driven continuously.

Reg signals can only be driven in procedural blocks such as always and initial.

Implicit Continuous Assignment

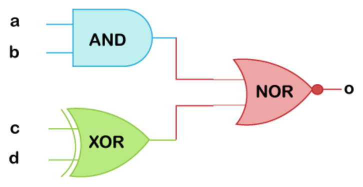
When an assign statement is used to assign the given net with some value, it is called an *explicit* assignment

If an assignment to be done during the net is declared, it is called an *implicit* assignment.

- wire [1:0] a;
- assign a = x & y;   // Explicit assignment
- wire [1:0] a = x & y;   // Implicit assignment

Combinational Logic Design

Consider the following digital circuit made from combinational gates and the corresponding Verilog code.



Combinational logic requires the inputs to be continuously driven to maintain the output, unlike sequential elements like flip flops where the value is captured and stored at the edge of a clock.

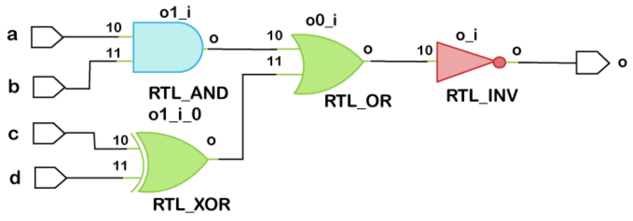
An assigned statement satisfies the purpose because the output o is updated whenever any of the inputs on the right-hand side change.

- // This module takes four inputs and performs a Boolean

- // operation and assigns output to o.
- // logic is realized using assign statement.
- module combo (input a, b, c, d, output o);
- assign o = ~((a & b) | c ^ d);
- endmodule

Hardware Schematic

After design elaboration and synthesis, a combinational circuit behaves the same way as modeled by the assign statement.



The signal o becomes 1 whenever the combinational expression on the RHS becomes true.

Similarly, o becomes 0 when RHS is false. Output o is X from 0ns to 10ns because inputs are X during the same time.

Verilog Operators

Operators perform an operation on one or more operands within an expression. An expression combines operands with appropriate operators to produce the desired functional expression.

1. Arithmetic Operators

For the FPGA, division and multiplication are very expensive, and sometimes we cannot synthesize division. If we use Z or X for values, the result is unknown. The operations treat the values as unsigned.

Character	Operation performed	Example
+	Add	b + c = 11
-	Subtrac	b - c = 9, -b=-10
/	Divide	b / a = 2
*	Multiply	a * b = 50

%	Modulus	b % a = 0

2. Bitwise Operators

Each bit is operated, the result is the size of the largest operand, and the smaller operand is left extended with zeroes to the bigger operand's size.

Character	Operation performed	Example
~	Invert each bit	~a = 3'b010
&	And each bit	b & c = 3'b010
|	Or each bit	a | b = 3'b111
^	Xor each bit	a ^ b = 3'b011
^~ or ~^	Xnor each bit	a ^~ b = 3'b100

3. Reduction Operators

These operators reduce the vectors to only one bit. If there are the characters z and x, the result can be a known value.

Character	Operation performed	Example
&	And all bits	&a = 1'b0, &d = 1'b0
~&	Nand all bits	~&a = 1'b1
|	Or all bits	|a = 1'b1, |c = 1'bX
~|	Nor all bits	~|a= 1'b0
^	Xor all bits	^a = 1'b1
^~ or ~^	Xnor all bits	~^a = 1'b0

4. Relational Operators

These operators compare operands and results in a 1-bit scalar Boolean value. The case equality and inequality operators can be used for unknown or high impedance values (z or x), and if the two operands are unknown, the result is a 1.

Character	Operation performed	Example
>	Greater than	a > b = 1'b0
<	Smaller than	a < b = 1'b1
>=	Greater than or equal	a >= d = 1'bX
<=	Smaller than or equal	a <= e = 1'bX
==	Equality	a == b = 1'b0
!=	Inequality	a != b = 1'b1
===	Case equality	e === e = 1'b1
!===	Case inequality	a !== d = 1'b1

5. Logical Operators

These operators compare operands and results in a 1-bit scalar Boolean value.

Character	Operation performed	Example
!	Not true	!(a && b) = 1'b1
&&	Both expressions true	a && b = 1'b0
| |	One ore both expressions true	a | | b = 1'b1

6. Shift Operators

These operators shift operands to the right or left, the size is kept constant, shifted bits are lost, and the vector is filled with zeroes.

Character	Operation performed	Example
>>	Shift right	b >> 1 results 4?b010X
<<	Shift left	a << 2 results 4?b1000

7. Assignment Operators

There are three assignment operators, each of which performs different tasks, and are used with different data types:

- assign (continuous assignment)
- <= (non-blocking assignment)
- = (blocking assignment)

8. Other Operators

These are operators used for condition testing and to create vectors.

Character	Operation performed	Example
?:	Conditions testing	test cond. ? if true do this or if not do this
{}	Concatenate	c = {a,b} = 8'101010x0
{{}}	Replicate	{3{2'b10}}= 6'b101010

9. Operators Precedence

The order of the table tells what operation is made first. The first one has the highest priority. The () can be used to override the default.

Operators precedence
+, -, !, ~ (Unary)
+,- (Binary)

<<, >>
<,>,<=,>=
==, !=
&
^, ^~ or ~^
|
&&
| |
?:

Verilog Always Block

In Verilog, the always block is one of the procedural blocks. Statements inside an always block are executed sequentially.

An always block always executes, unlike initial blocks that execute only once at the beginning of the simulation. The always block should have a sensitive list or a delay associated with it

The sensitive list is the one that tells the always block when to execute the block of code.

Syntax

The Verilog always block the following syntax

```
always @ (event)
    [statement]

always @ (event) begin
    [multiple statements]
end
```

Examples

The symbol @ after reserved word *always*, indicates that the block will be triggered *at* the condition in parenthesis after symbol @.

```
always  @ (x or y or sel)
begin
    m = 0;
  if (sel == 0) begin
    m = x;
   end else begin
    m = y;
   end
end
```

In the above example, we describe a 2:1 mux, with input x and y. The *sel* is the select input, and *m* is the mux output.

In any combinational logic, output changes whenever input changes. When this theory is applied to always blocks, then the code inside always blocks needs to be executed whenever the input or output variables change.

NOTE: It can drive reg and integer data types but cannot drive wire data types.

There are two types of sensitive list in the Verilog, such as:

1. Level sensitive (for combinational circuits).
2. Edge sensitive (for flip-flops).

The code below is the same 2:1 mux, but the output *m* is now a flip-flop output.

```
always  @ (posedge clk )
if (reset == 0) begin
    m <= 0;
end
    else if (sel == 0) begin
    m <= x;
   end
 else begin
    m <= y;
end
```

NOTE: The always block is executed at some particular event. A sensitivity list defines the event.

Sensitivity List

A sensitivity list is an expression that defines when the always block executed, and it is specified after the @ operator within the parentheses ( ). This list may contain either one or a group of signals whose value change will execute the always block.

In the code shown below, all statements inside the always block executed whenever the value of signals x or y change.

```
// execute always block whenever value of "x" or "y" change
always @ (x or y) begin
    [statements]
end
```

Need of Sensitivity List

The always block repeats continuously throughout a simulation. The sensitivity list brings a certain sense of timing, i.e., whenever any signal in the sensitivity list changes, the always block is triggered.

If there are no timing control statements within an always block, the simulation will hang because of a zero-delay infinite loop.

For example, always block attempts to invert the value of the signal clk. The statement is executed after every 0-time units. Hence, it executes forever because of the absence of a delay in the statement.

```
// always block started at time 0 units
// But when is it supposed to be repeated
// There is no time control, and hence it will stay and
// be repeated at 0-time units only and it continues
// in a loop and simulation will hang

   always clk = ~clk;
```

If the sensitivity list is empty, there should be some other form of time delay. Simulation time is advanced by a delay statement within the always construct.

```
always #10 clk = ~clk;
```

Now, the clock inversion is done after every 10-time units. That's why the real Verilog design code always requires a sensitivity list.

NOTE: Explicit delays are not synthesizable into logic gates.

Uses of always block

An always block can be used to realize combinational or sequential elements. A sequential element like flip flop becomes active when it is provided with a clock and reset.

Similarly, a combinational block becomes active when one of its input values change. These hardware blocks are all working concurrently independently of each other. The connection between each is what determines the flow of data.

An always block is made as a continuous process that gets triggered and performs some action when a signal within the sensitivity list becomes active.

In the following example, all statements within the always block executed at every positive edge of the signal clk

- // execute always block at the positive edge of signal "clk"
- always @ (posedge clk) begin
-     [statements]
- end

Sequential Element Design

The below code defines a module called *tff* that accepts a data input, clock, and active-low reset. Here, the always block is triggered either at the positive edge of the *clk* or the negative edge of *rstn*.

1. The positive edge of the clock

The following events happen at the positive edge of the clock and are repeated for all positive edge of the clock.

Step 1: First, if statement checks the value of active-low reset *rstn*.

- If *rstn* is zero, then output q should be reset to the default value of 0.
- If *rstn* is one, then it means reset is not applied and should follow default behavior.

Step 2: If the previous step is false, then

- Check the value of d, and if it is found to be one, then invert the value of q.
- If d is 0, then maintain value of q.

- module tff (input d, clk, rstn, output reg q);
- always @ (posedge clk or negedge rstn) begin
- **if** (!rstn)
-     q <= 0;

```
else
   if (d)
      q <= ~q;
   else
      q <= q;
end
endmodule
```

2. Negative edge of reset

The following events happen at the negative edge of *rstn*.

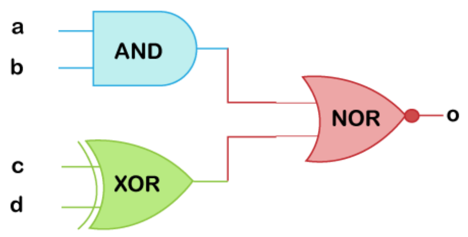
Step 1: First, if statement checks the value of active-low reset *rstn*. At the negative edge of the signal, its value is 0.

- If the value of *rstn* is 0, then it means reset is applied, and output should be reset to the default value of 0.
- And if the value of *rstn* is 1, then it is not considered because the current event is a negative edge of the *rstn*.

Combinational Element Design

An always block can also be used in the design of combinational blocks.

For example, the digital circuit below represents three different logic gates that provide a specific output at signal o.



The code shown below is a module with four input ports and a single output port called o. The always block is triggered whenever any of the signals in the sensitivity list changes in value.

The output signal is declared as type *reg* in the module port list because it is used in a procedural block. All signals used in a procedural block should be declared as type *reg*.

```
module combo (input a, input b, input c, input d, output reg o);
  always @ (a or b or c or d) begin
    o <= ~((a & b) | (c^d));
  end
```

- endmodule

The signal o becomes 1 whenever the combinational expression on the RHS becomes true. Similarly, o becomes 0 when RHS is false.

Verilog Initial Block

The *always* block indicates a free-running process, but the *initial* block indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block.

Initial blocks can be used in either synthesizable or non-synthesizable blocks. They are commonly used in test benches.

Initial blocks cause particular instructions to be performed at the beginning of the simulation before any other instructions operate. Initial blocks only operate once.

A synthesizable initial block is used to set the power-on value of registers, RAM, and ROM within FPGAs. However, initial blocks cannot be synthesized in ASICs or CPLDs.

Initial and always block describe independent processes, which means the statements in one process execute autonomously.

Both types of processes consist of procedural statements, and both start immediately as the simulator is started.

The main differences between them are:

- The initial processes execute once, whereas always process repeatedly execute forever.
- An always process must contain timing statements that will occasionally block execution and allow time to advance.

Syntax

Verilog initial block follows the following syntax:

- initial
-     [single statement]
-
- initial begin
-     [multiple statements]
-
- end

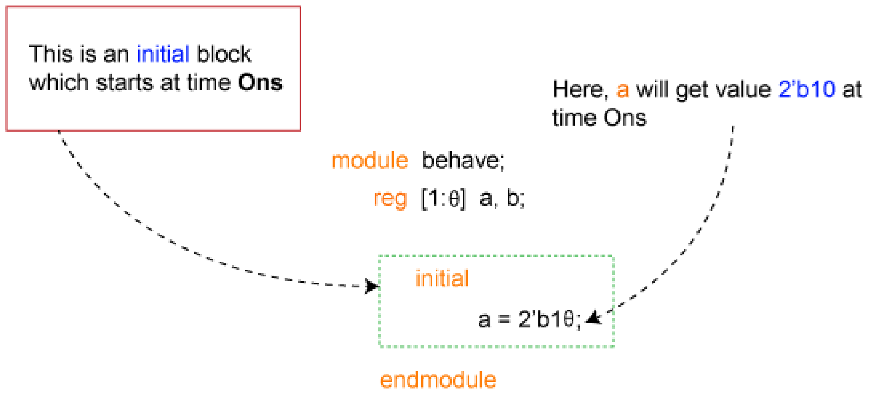
Initial block uses

An initial block is not synthesizable and cannot be converted into a hardware schematic with digital elements.

The initial blocks do not have more purpose than to be used in simulations. These blocks are primarily used to initialize variables and drive design ports with specific values.

Initial Block Execution

An initial block is started at the beginning of a simulation at time 0 unit. This block will be executed only once during the entire simulation. Execution of an initial block finishes once all the statements within the block are executed, as shown in the following image.


This is an initial block which starts at time 0ns
Here, a will get value 2'b10 at time 0ns

```
module  behave;
    reg  [1:0]  a, b;

        initial
              a = 2'b10;

        endmodule
```

The image shown above has a module called behave, which has a and b internal signals.

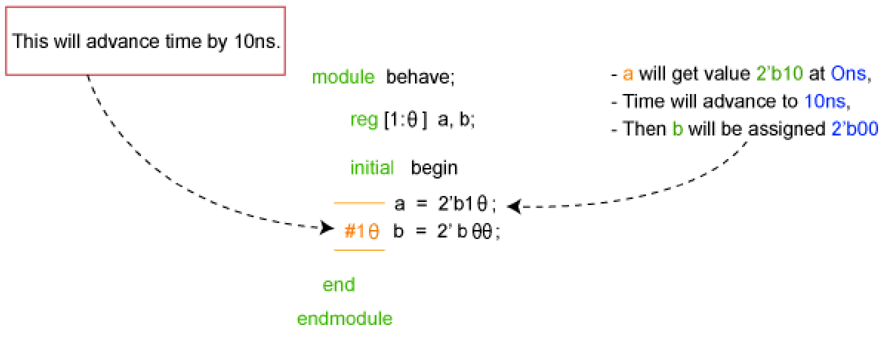
The initial block has only one statement, and hence it is not necessary to place the statement within begin and end.

This statement assigns the value 2'b10 to a when the initial block is started at time 0 units.

Initial Block Delay Element

The code shown below has an additional statement that assigns some value to the signal b. However, this happens only after 10-time units from the execution of the previous statement.

For example, If a is assigned first with the given value and then after 10-time units, b is assigned to 0.

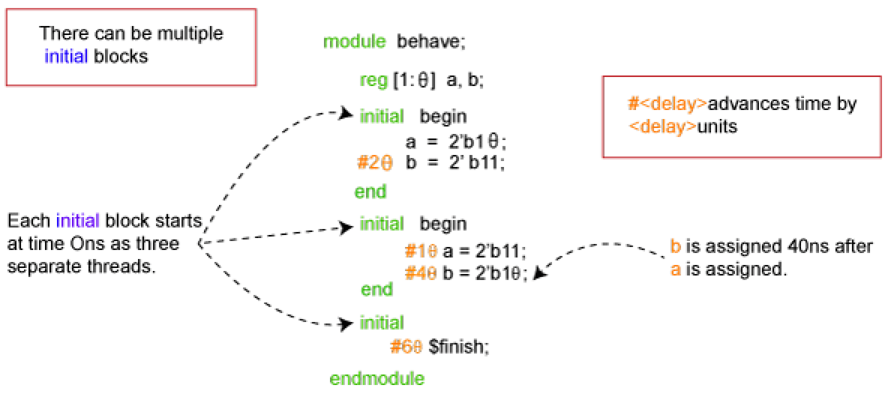

This will advance time by 10ns.
- a will get value 2'b10 at 0ns,
- Time will advance to 10ns,
- Then b will be assigned 2'b00

```
module  behave;
    reg [1:0]  a, b;
    initial  begin
         a = 2'b10;
     #10 b = 2'b00;
  end
endmodule
```

Initial Blocks in a module

There are no limits to the number of initial blocks that can be defined inside a module. The code shown below has three initial blocks, all of which are started at the same time and run in parallel.

However, depending on the statements and the delays within each initial block, the time taken to finish the block may vary.


There can be multiple initial blocks
#<delay>advances time by <delay>units
Each initial block starts at time 0ns as three separate threads.
b is assigned 40ns after a is assigned.

```
module  behave;
    reg [1:0]  a, b;
    initial  begin
             a = 2'b10;
        #20  b = 2'b11;
    end
    initial  begin
        #10 a = 2'b11;
        #40 b = 2'b10;
    end
    initial
        #60 $finish;
endmodule
```

NOTE: $finish is a Verilog system task that tells the simulator to terminate the current simulation.

In the above image, the first block has a delay of 20 units, while the second has a total delay of 50 units (10 + 40), and the last block has a delay of 60 units. Hence the simulation takes 60-time units to complete since there is atleast one initial block still running until 60-time units.

If the last block had a delay of 30-time units, as shown below, the simulation would have ended at 30-time units, thereby killing all the other initial blocks that are active at that time.

- initial begin
-     #30 $finish;
- end

Verilog Block Statements

The block statements are the grouping of two or more statements together, which act syntactically like a single statement. There are two types of blocks in the Verilog:

- Sequential block
- Parallel block

These blocks can be used if more than one statement should be executed. All statements in the *sequential* blocks will be executed sequentially in the given order.

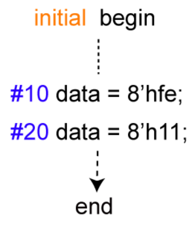
If a timing control statement appears within a block, then the next statement will be executed after that delay. The sequential block shall be delimited by the keywords begin and end.

All statements in the *parallel* blocks are executed at the same time or concurrently. It means that the next statement's execution will not be delayed even if the previous statement contains a timing control statement. The parallel block shall be delimited by the keywords fork and join.

Sequential Block

Statements are wrapped using *begin* and *end* keywords and executed sequentially in the given order. Delay values are treated relative to the time of execution of the previous statement.

After all the statements within the block are executed, control may be passed to some other place.



```
initial  begin

#10 data = 8'hfe;
#20 data = 8'h11;

end
```

Syntax

Sequential block statement follows the following syntax:

- begin: name
-     statement1;
-     …………..
- end

Characteristics

The sequential block has the following characteristics, such as:

- Statements will be executed in the sequence, one after another.
- Delay values for each statement are treated relative to the simulation time of the previous statement's execution.
- Control can pass out of the block after the last statement executes.

Example

```
module design0;
   bit [31:0] data;

      // initial block starts at time 0

         initial begin

      // After 10 time units, data becomes 0xfe
       #10   data = 8'hfe;
      $display ("[Time=%0t] data=0x%0h", $time, data);

      // After 20 time units, data becomes 0x11
       #20   data = 8'h11;
      $display ("[Time=%0t] data=0x%0h", $time, data);
   end
endmodule
```

In the above example, first statement in begin and end block will be executed at 10 time units, and the second statement at 30 time units because of the relative nature. It is 20 time units after execution of the previous statement.

```
ncsim> run
[Time=10] data=0xfe
[Time=30] data=0x11
ncsim: *W,RNQUIE: Simulation is complete.
```

Parallel Block

A **parallel** block can execute statements concurrently, and delay control can be used to provide the assignments' time-ordering. Statements are launched in parallel by wrapping them within the *fork* and *join* keywords.

Syntax

The parallel block has the following simplified syntax:

```
fork: name
      statement;
       …………
join
```

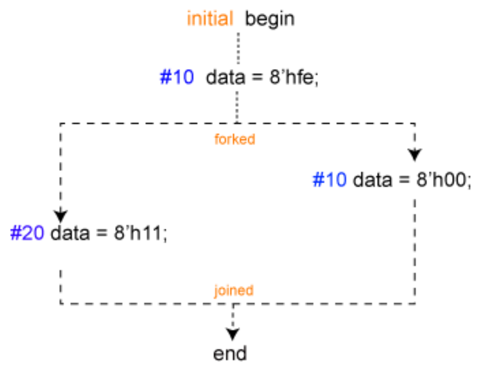
Characteristics

A parallel block has the following characteristics:

- Statements will execute concurrently.
- Delay values for each statement are considered relative to the simulation time of entering the block.
- Delay control can be used to provide time-ordering for assignments.
- Control can pass out of the block when the last time-ordered statement executes.

Example

- initial begin
-    #10   data = 8'hfe;
-     fork
-       #20 data = 8'h11;
-        #10 data = 8'h00;
-     join
- end



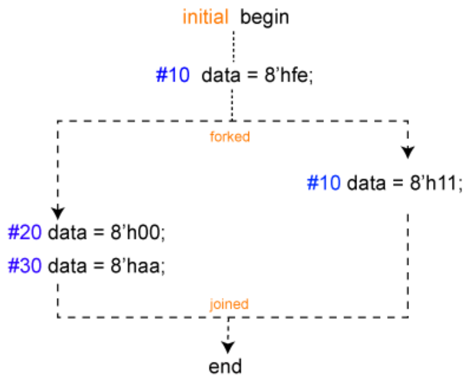
In the above example, fork and join block will be launched after executing the statement at 10-time units.

Statements within this block will be executed in parallel, and the first statement will be the one where data is assigned a value of 8'h00 since the delay for that is 10-time units after the fork-join launch.

After 10 more time units, the first statement will be launched and data will get the value 8'h11.

- initial begin
-    #10 data = 8'hfe;
-     fork
-        #10 data = 8'h11;

- begin
- #20 data = 8'h00;
- #30 data = 8'haa;
- end
- join
- end
-



There is a begin-end block in the above example, and all statements within the begin-end block will be executed sequentially. Still, the block itself will be launched in parallel, along with the other statements. Data will get 8'h11 at 20-time units, 8'h00 at 30-time units, and 8'haa at 60-time units.

Naming of blocks

Both sequential and parallel blocks can be named by adding *name_of_block* after the *begin* and *fork* keywords. By doing this, the block can be referenced in a *disable* statement.

- begin: name_seq
- [statements]
- end
- fork: name_fork
- [statements]
- join

Verilog Assignments

Placing values onto variables and nets are called assignments. There are three necessary forms:

1. Procedural
2. Continuous
3. Procedural continuous

Legal LHS values

An assignment has two parts, right-hand side (RHS) and left-hand side (LHS) with an equal symbol (=) or a less than-equal symbol (<=) in between.

Assignment Type	**Left-hand Side**
Procedural	○ Variables (vector or scalar) ○ Bit-select or part-select of an integer, vector reg, or time variable. ○ Memory word. ○ Concatenation of any of the above.
Continuous	○ Net (vector or scalar) ○ Bit-select or part-select of a vector net. ○ Concatenation of bit-selects and part-selects.
Procedural Continuous	○ Variable or net (scalar/vector) ○ Part-select or bit-select of a vector net.

The RHS can contain any expression that evaluates to a final value while the LHS indicates a variable or net to which RHS's value is being assigned.

Procedural Assignment

Procedural assignments occur within procedures such as *initial, always, task*, and *functions* are used to place values onto variables. The variable will hold the value until the next assignment to the same variable.

The value will be placed onto the variable when the simulation executes this statement during simulation time. This can be modified and controlled the way we want by using control flow statements such as *if-else-if, looping*, and *case statement* mechanisms.

```
• reg [7:0]  data;
• integer    count;
• real       period;
•
• initial begin
•     data = 8'h3e;
```

- period = 4.23;
- count = 0;
- end
-
- always @ (posedge clk)
- count++;

Variable Declaration Assignment

An initial value can be placed onto a variable at the time of its declaration. The assignment does not have the duration and holds the value until the next assignment to the same variable happens.

NOTE: The variable declaration assignments to an array are not allowed.

- module my_book;
- reg [31:0] data = 32'hdead_cafe;
- initial begin
- #20 data = 32'h1234_5678; // data will have dead_cafe from time 0 to 20
- end
- endmodule
- reg [3:0] a = 4'b4;
- reg [3:0] a;
- initial a = 4'b4;

If the variable is initialized during declaration and at 0 times in an initial block as shown below, the order of evaluation is not guaranteed, and hence can have either 8'h05 or 8'hee.

- module my_book;
- reg [7:0] addr = 8'h05;
- initial
- addr = 8'hee;
- endmodule
- reg [3:0] array [3:0] = 0; // illegal
- integer i = 0, j; // declares two integers i,j and 0 is assigned to i
- real r2 = 4.5, r3 = 8; // declares two real numbers r2,r3 and are assigned 4.5, 8 resp.
- time startTime = 40; // declares time variable with initial value 40

Continuous Assignment

This is used to assign values onto scalar and vector nets. And it happens whenever there is a change in the RHS.

It provides a way to model combinational logic without specifying an interconnection of gates and makes it easier to drive the net with logical expressions.

- // Example model of an AND gate
- wire a, b, c;
- assign a = b & c;

Whenever b or c changes its value, the whole expression in RHS will be evaluated and updated with the new value.

Net Declaration Assignment

This allows us to place a continuous assignment on the same statement that declares the net.

NOTE: Only one declaration assignment is possible because a net can be declared only once.

1. wire penable = 1;

Procedural Continuous Assignment

These are procedural statements that allow expressions to be continuously assigned to variables or nets. And these are the two types.

1. Assign deassign: It will override all procedural assignments to a variable and deactivate it using the same signal with *deassign*.

The value of the variable will remain the same until the variable gets a new value through a procedural or procedural continuous assignment.

The LHS of an *assign* statement cannot be a part-select, bit-select, or an array reference, but it can be a variable or a combination of the variables.

- reg q;
- initial begin
-     assign q = 0;
-     #10 deassign q;
-    end

2. Force release: These are similar to the *assign deassign*statements but can also be applied to nets and variables.

The LHS can be a bit-select of a net, part-select of a net, variable, or a net but cannot be the reference to an array and bit or part select of a variable.

The *force* statement will override all other assignments made to the variable until it is released using the *release* keyword.

```
•   reg o, a, b;
•   initial begin
•           force o = a & b;
•           ……….
•            release o;
•   end
```

Verilog Blocking and Non-blocking

Verilog supports blocking and non-blocking assignments statements within the always block with their different behaviors.

The blocking assignment is similar to software assignment statements found in most popular programming languages. The non-blocking assignment is the more natural assignment statement to describe many hardware systems, especially for synthesis.

The blocking assignments can only be used in a few situations, such as modeling combinational logic, defining functions, or implementing testbench algorithms. All IEEE P1364.1 compliant synthesis tools are required to support both blocking and non-blocking assignments in explicit-style code, with the restriction that each variable and each block may use only one or the other kind of assignment.

Blocking Assignment

Blocking assignment statements are assigned using (=) operator and are executed one after the other in a procedural block. But, it will not prevent the execution of statements that run in a parallel block.

```
•  module Block;
•    reg [7:0] a, b, c, d, e;
•
•    initial begin
•      a = 8'hDA;
•      $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
•      b = 8'hF1;
•      $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
•      c = 8'h30;
•      $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
•    end
•
•    initial begin
•      d = 8'hAA;
•      $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
```

```
    e = 8'h55;
    $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
  end
endmodule
```

There are two *initial* blocks which are executed in parallel. Statements are executed sequentially in each block and both blocks finish at time 0ns.

To be more specific, variable is assigned first, that followed by the display statement which is then followed by all other statements.

This is visible in the output where variable *b* and *c* are 8'hxx in the first display statement. This is because variable *b* and *c* assignments have not been executed yet when the first $*display* is called.

```
ncsim> run
[0] a=0xda b=0xx c=0xx
[0] a=0xda b=0xf1 c=0xx
[0] a=0xda b=0xf1 c=0x30
[0] d=0xaa e=0xx
[0] d=0xaa e=0x55
ncsim: *W,RNQUIE: Simulation is complete.
```

In the below example, we'll add a few delays into the same set of statements to see how it reacts and behaves.

```
module Block;
  reg [7:0] a, b, c, d, e;
  initial begin
    a = 8'hDA;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
    #10 b = 8'hF1;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
    c = 8'h30;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
  end
  initial begin
    #5 d = 8'hAA;
    $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
    #5 e = 8'h55;
    $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
  end
endmodule
```

After execution, it gives the following data.

```
ncsim> run
[0] a=0xda b=0xx c=0xx
[5] d=0xaa e=0xx
[10] a=0xda b=0xf1 c=0xx
[10] a=0xda b=0xf1 c=0x30
[10] d=0xaa e=0x55
ncsim: *W,RNQUIE: Simulation is complete.
```

Non-blocking Assignment

Non-blocking assignment statements are allowed to be scheduled without blocking the execution of the following statements and is specified by a (<=) symbol.

The same symbol is used as a relational operator in expressions, and as an assignment operator in the context of a non-blocking assignment.

Take the same example as above, replace all (=) symbols with a non-blocking assignment operator (<=), we'll get the difference in the output.

```
module Block;
  reg [7:0] a, b, c, d, e;
  initial begin
    a <= 8'hDA;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
    b <= 8'hF1;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
    c <= 8'h30;
    $display ("[%0t] a=0x%0h b=0x%0h c=0x%0h", $time, a, b, c);
  end
initial begin
    d <= 8'hAA;
    $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
    e <= 8'h55;
    $display ("[%0t] d=0x%0h e=0x%0h", $time, d, e);
  end
endmodule
```

Now, all the $*display* statements printed *'h'x*. The reason for this behavior is the execution of the non-blocking assignment statements.

The RHS of every non-blocking statement of a particular time-step is captured and moves onto the next statement.

The captured RHS value is assigned to the LHS variable only at the end of the time-step.

```
ncsim> run
[0] a=0xx b=0xx c=0xx
[0] a=0xx b=0xx c=0xx
[0] a=0xx b=0xx c=0xx
[0] d=0xx e=0xx
[0] d=0xx e=0xx
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Control Blocks

Hardware behavior cannot be implemented without conditional statements and other ways to control the flow of logic. Verilog has a set of mechanisms and control flow blocks.

if-else-if

This conditional statement is used to decide whether certain statements will be executed or not. It is very similar to the if-else-if statements in C language. If the expression evaluates to true, then the first statement will be executed.

If the expression evaluates to false and if an else part exists, the else part will be executed.

Syntax

Following is the most simplified syntax of the if-else-if conditional statement:

```
// if statement without else part
if (expression)
   [statement]

// if statement with an else part
if (expression)
   [statement]
else
   [statement]

// if else for multiple statements should be
```

- // enclosed within "begin" and "end"
-
- **if** (expression) begin
-     [multiple statements]
- end **else** begin
-     [multiple statements]
- end
-
- // if-else-if statement
- **if** (expression)
-     [statement]
- **else if** (expression)
-     [statement]
-
- **else**
-     [statement]

The else part of an if-else is optional, and it can create confusion. To avoid this confusion, it's easier to always associate the else to the previous if that lacks an else. Another way is to enclose statements within a begin-end block. The last else part handles the none-of-the-above or default case where none of the other conditions were satisfied.

Loops provide a way of executing single or multiple statements within a block one or more number of times. In Verilog, there are four different types of looping statements.

1. Forever loop

This loop will continuously execute the statements within the block.

Syntax

- Forever
-     [statement]
- forever begin
-     [multiple statements]
- end

Example

- module my_block;
- initial begin
-     forever begin

```
        $display ("This will be printed forever, simulation can hang ...");
    end
end
endmodule
```

After the execution of the above example, it produces the following data.

```
ncsim> run
This will be printed forever, simulation can hang
This will be printed forever, simulation can hang
...
...
This will be printed forever, simulation can hang
This will be printed forever, simulation can hang
This will be printed forever, simulation can hang
This will be printed forever, simulation can hang
...
...
Result reached a maximum of 5000 lines.
Killing process.
```

2. Repeat loop

This will execute statements a fixed number of times. If the expression evaluates to an X or Z, it will be treated as zero and not executed.

Syntax

```
repeat ([num_of_times]) begin
   [statements]
end
repeat ([num_of_times]) @ ([some_event]) begin
   [statements]
end
```

Example

```
module my_block;
   initial begin
       repeat(5) begin
          $display("This is a new iteration ...");
       end
```

```
    end
endmodule
```

The above code generates the following outcome.

```
ncsim> run
This is a new iteration
This is a new iteration
This is a new iteration
This is a new iteration
This is a new iteration

ncsim: *W,RNQUIE: Simulation is complete.
```

3. ile loop

This will execute statements as long as an expression is true and will exit once the condition becomes false. If the condition is false from the start, statements will not be executed at all.

Syntax

```
while (expression) begin
    [statements]
end
```

Example

```
module my_block;
    integer i = 5;
    initial begin
      while (i > 0) begin
         $display ("Iteration#%0d", i);
        i = i - 1;
      end
    end
endmodule
```

Run the above code, and we will get the following output.

```
ncsim> run
Iteration#5
Iteration#4
```

- Iteration#3
- Iteration#2
- Iteration#1
- ncsim: *W,RNQUIE: Simulation is complete.

4. For loop

For loop controls the statements using a three-step process:

- Initialize a loop counter variable.
- Evaluate the expression, usually involving the loop counter variable.
- Increment loop counter variable so that the expression will become false at a later time, and the loop will exit.

Syntax

```
for ( initial_assignment; condition; increment_variable) begin
   [statements]
end
```

Example

```
module my_block;
   integer i = 5;
   initial begin
     for (i = 0; i < 5; i = i + 1) begin
        $display ("Loop #%0d", i);
     end
   end
endmodule
```

After execution the for loop code, the output looks like

```
ncsim> run
Loop #0
Loop #1
Loop #2
Loop #3
Loop #4
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Functions

The purpose of a function is to return a value that is to be used in an expression. A function definition always starts with the *function* keyword followed by the return type, name, and a port list enclosed in parentheses. And it ends with the *endfunction* keyword.

Functions should have at least one input declaration and a statement that assigns a value to the register with the same name as the function. And the return type can be *void* if the function does not return anything.

Functions can only be declared inside a module declaration and can be called through always blocks, continuous assignments, or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked.

Functions describe combinational logic. Functions are an excellent way to reuse procedural code since modules cannot be invoked from a procedure.

The returned type or range declaration followed by a function identifier and semicolon should appear after the function keyword. A function can contain declarations of returned type, parameters, range, registers, events, and input arguments. These declarations are similar to module items declaration.

Net declarations are illegal. Declaration of registers, parameters, events, range, and returned type is not required. A function without a range or return type declaration returns a one-bit value.

Any expression can be used as a function call argument. Functions cannot contain any time-controlled statements, and they cannot enable tasks. Functions can return only one value.

When we find specific pieces of code to be repetitive and called multiple times within the RTL, they mostly do not consume simulation time. They might involve complex calculations that need to be done with different data values.

In this situation, we can declare a function and place the repetitive code inside the function and allow it to return the result.

It will reduce the number of lines in the RTL. Now we need to do a function call and pass data on which the computation needs to be performed.

Syntax

Following is the syntax to use a function in the Verilog:

- function [automatic] [return_type] name ([port_list]);
-     [statements]
-
- endfunction

The keyword automatic will make the *reentrant* function and items declared within the task that are dynamically allocated rather than shared between different invocations of the task. This will be useful for recursive functions and when the same function is executed concurrently by N processes.

Function declarations

A function declaration specifies the function's name, the function input arguments, the variables (reg) used within the function, the width of the function return value, and the function local parameters and integers.

Syntax

Following is the specified syntax to declare function in the Verilog:

- function [msb: lsb] function_name;
- input [msb: lsb] input_arguments;
- reg [msb: lsb] reg_variable_list;
- parameter [msb: lsb] parameter_list;
- integer [msb: lsb] integer_list;
- [statements]
- Endfunction

Example

- function [7:0] sum;
- input [7:0] a, b;
- begin
- sum = a + b;
- end

Function Return Value

The function definition will implicitly create an internal variable of the same name as that of the function.

Hence it is illegal to declare another variable of the same name inside the scope of the function. The return value is initialized by assigning the function result to the internal variable.

1. sum = a + b;

Calling a Function

A function call is an operand with an expression. A function call must specify in its terminal list all the input parameters.

Example

```
reg [7:0] result;
reg [7:0] a, b;
initial begin
   a = 4;
    b = 5;
   #10 result = sum (a, b);
end
```

Rules

The following are some of the general rules for the Verilog functions:

- A function cannot contain any time-controlled statements such as *#, @, wait, posedge*, and *negedge*.
- A function cannot start a task because it may consume simulation time but can call other functions.
- A function should have atleast one input argument.
- A function cannot have non-blocking assignments or force-release or assign-deassign.
- A function cannot have any triggers.
- A function cannot have an *inout* or *output* declaration.
- Functions must contain a statement that assigns the return value to the implicit function name register.

Verilog Task

A function is meant to do some processing on the input and return a single value. In contrast, a task is more general and can calculate multiple result values and return them using *output* and *inout* type arguments.

Tasks can contain time-consuming simulation elements such as **@, posedge**, and others. Tasks are used in all programming languages, generally known as procedures or subroutines.

Data is passed to the task, the processing is done, and the result returned. They have to be specifically called, with the data ins and outs, rather than just wired into the general netlist.

Included in the main body of code, they can be called many times, reducing code repetition.

- Tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compiled directive, including the task in the file, which instantiates the task.
- Tasks can include timing delays, such as *posedge, negedge, # delay*, and *wait*.
- Tasks can have any number of inputs and outputs.

- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- Tasks can take, drive, and source global variables when no local variables are used. When local variables are used, the output is assigned only at the end of task execution.
- Tasks can call another task or function.
- Tasks can be used for modeling both combinational and sequential logic.

A task must be specifically called with a statement. It cannot be used within an expression as a function can.

Syntax

A task begins with keyword task and ends with keyword endtask. Inputs and outputs are declared after the keyword task.

Local variables are declared after input and output declaration.

```
• // Style 1
•
• task [name];
•    input  [port_list];
•    inout  [port_list];
•    output [port_list];
•    begin
•       [statements]
•    end
• endtask
•
•
• // Style 2
•
• task [name] (input [port_list], inout [port_list], output [port_list]);
•    begin
•       [statements]
•    end
• endtask
```

The keyword *automatic* will make the *reentrant* task. Otherwise, it will be static by default. If a task is static, all its member variables will be shared across different invocations of the same task that has been launched concurrently.

NOTE: Hierarchical references cannot access an automatic task item.

Calling a task

If the task does not need any arguments, then a list of arguments can be avoided. If the task needs arguments, they can be provided in the same statement as its invocation.

```
//Style 1
task sum (input [7:0] a, b, output [7:0] c);
       begin
          c = a + b;
       end
   endtask

// Style 2
   task sum;
      input  [7:0] a, b;
       output [7:0] c;
      begin
          c = a + b;
      end
   endtask
        initial begin
       reg [7:0] x, y , z;
      sum (x, y, z);
   end
```

The task-enabling arguments (x, y, z) correspond to the arguments (a, b, c) defined by the task.

Since **a** and **b** are inputs, values of **x** and **y** will be placed in **a** and **b**, respectively. Because **c** is declared output and connected with **z** during invocation, the sum will automatically be passed to the variable **z** from **c**.

Global tasks

Tasks that are declared outside all modules are called global tasks as they have a global scope and can be called within any module.

```
// This task is outside all modules
task display();
  $display("Hello");
endtask
module des;
```

- initial begin
-   display();
- end
- endmodule

After executes the above code, it produces the following output.

1. xcelium> run
2. Hello
3. xmsim: *W,RNQUIE: Simulation is complete.

Difference between Function and Task

Function	**Task**
It cannot have time-controlling statements/delay and hence executes in the same simulation time unit.	It can contain time-controlling statements/delay and may only complete at some other time.
It cannot enable a task.	It can enable other tasks and functions.
The function should have atleast one input argument and cannot have output or inout arguments.	Tasks can have zero or more arguments of any type.
A function can return only a single value.	It cannot return a value but can achieve the same effect using output arguments.

Verilog Case Statement

The case statement checks if the given expression matches one among the other expressions inside the list and branches. It's typically accustomed implement a device.

The if-else construct may not be applicable if there unit of measurement many conditions to be checked and would synthesize into a priority encoder instead of a device.

In Verilog, a case statement includes all of the code between the Verilog keywords, case ("casez", "casex"), and endcase. A case statement can be a select-one-of-many construct that is roughly like Associate in nursing if-else-if statement.

Syntax

A Verilog case statement starts with the case keyword and ends with the endcase keyword.

The expression within parentheses area unit aiming to be evaluated specifically once and is compared with the list of alternatives inside the order they are written.

And the statements that the selection matches the given expression unit of measurement dead. A block of multiple statements ought to be sorted and be within begin and end.

```
• case ()
•    case_item1 :
•    case_item2,
•    case_item3 :
•    case_item4 :
•      begin
•
•      end
•    default:
•
• endcase
```

If none of the case things match the given expression, statements within the default item unit of measurement dead. The default statement is non mandatory, and there's only one default statement throughout a case statement. Case statements are nested.

Execution will exit the case block whereas not doing one thing if none of the items match the expression, and a default statement is not given.

Example

The following vogue module includes a 2-bit opt for signal to route one among the three different 3-bit inputs to the sign stated as out.

A case statement is used to assign the correct input to output supported the value of sel. Since sel can be a 2-bit signal, it'll have twenty 2 combos, zero through 3. The default statement helps to line output to zero if sel is 3.

```
module my_mux (input [2:0] a, b, c,        // three 3-bit inputs
                     [1:0]sel,             // 2-bit opt for signal to choose on from a, b, c
                 output reg  [2:0] out);        // Output 3-bit signal
 // invariably block is dead whenever a, b, c or sel changes in value
 invariably @ (a, b, c, sel) begin
   case(sel)
     2'b00    : out = a;       // If sel=0, output can be a
     2'b01    : out = b;       // If sel=1, output is b
     2'b10    : out = c;       // If sel=2, output is c
     default  : out = 0;       // If sel is something, out is commonly zero
   endcase
 end
```

- endmodule

Case Statement Header

A case statement header consists of the case ("casez", "casex") keyword followed by the case expression, usually all on one line of code.

When adding full_case or parallel_case directives to a case statement, the directives unit of measurement added as a comment in real time following the case expression at the tip of the case statement header and before any of the case things on ensuing code lines.

Case item

The case item is that the bit, vector, or Verilog expression accustomed compare against the case expression.

Unlike different high-level programming languages like 'C', the Verilog case statement includes implicit break statements.

The first case item that matches this case expression causes the corresponding case item statement to be dead, thus all of the rest of the case things unit of measurement skipped for this undergo the case statement.

Case item statement

A case item statement is one or plenty of Verilog statements dead if the case item matches this case expression. Not like VHDL, Verilog case things can themselves be expressions.

To alter the parsing of Verilog code document, Verilog case item statements ought to be enclosed between the keywords *"begin"* and *"end"* if over one statement is to be dead for a specific case item.

Casez

In Verilog, there is a casez statement, a variation of the case statement that enables "z" and "?" values to be treated throughout case-comparison as "don't care" values.

"Z" and "?" unit of measurement treated as a don't care if they are inside the case expression or if they are inside the case item.

When secret writing a case statement with "don't care," use a casez statement and use "?" characters instead of "z" characters inside the case things to purpose "don't care" bits.

Casex

In Verilog, there is a casex statement, a variation of the case statement that enables "z", "?", and "x" values to be treated throughout comparison as "don't care" values.

"x", "z" and "?" unit of measurement treated as a don't care if they are inside the case expression or if they are inside the case item.

Full Case Statement

A full case statement can be a case statement inside that all getable case-expression binary patterns are matched to a case item or a case default.

If a case statement does not embrace a case default, and it's getable to go looking out a binary case expression that does not match any of the printed case things, the case statement is not full.

A full case statement can be a case statement inside that every getable binary, non-binary, and mixture of binary and non-binary patterns is boxed in as a case item inside the case statement.

Verilog does not would like case statements to be either synthesis or high-density lipoprotein simulation full, but Verilog case statements is made full by adding a case default. VHDL desires case statements to be high-density lipoprotein simulation full, that usually desires Associate in Nursing "others" clause.

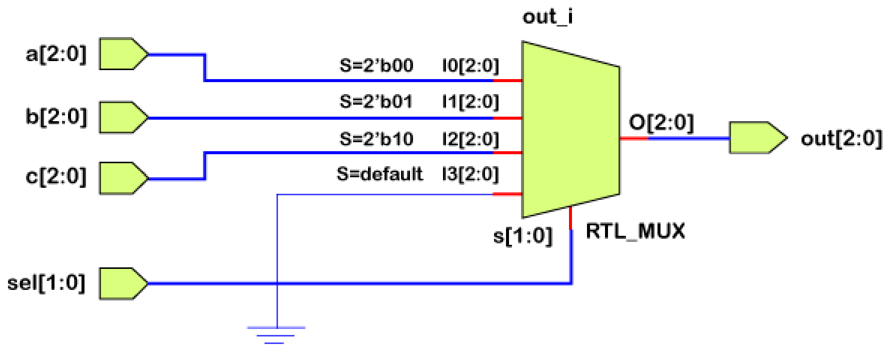
Parallel Case Statement

A parallel case statement can be a case statement inside that it's only getable to match a case expression to only one case item.

If it's getable to go looking out a case expression which may match over one case item, the matching case things unit of measurement stated as overlapping case things, and so the case statement is not parallel.

Hardware Schematic

The RTL code is elaborated to get a hardware schematic that represents a 4 to 1 multiplexer.



After executes the above design, the output is zero when sel is 3 and corresponds to the assigned inputs for other values.

- ncsim> run
- [0] a=0x4 b=0x1 c=0x1 sel=0b11 out=0x0
- [10] a=0x5 b=0x5 c=0x5 sel=0b10 out=0x5

```
[20] a=0x1 b=0x5 c=0x6 sel=0b01 out=0x5
[30] a=0x5 b=0x4 c=0x1 sel=0b10 out=0x1
[40] a=0x5 b=0x2 c=0x5 sel=0b11 out=0x0
ncsim: *W,RNQUIE: Simulation is complete.
```

In a case statement, the comparison only succeeds when each bit of the expression matches one of the alternatives including 0, 1, x and z. In the above example, if any of the bits in sel is either x or z, the *default* statement will be executed because none of the other alternatives matched. In such a case, output will be all zeros.

```
ncsim> run
[0] a=0x4 b=0x1 c=0x1 sel=0bxx out=0x0
[10] a=0x3 b=0x5 c=0x5 sel=0bzx out=0x0
[20] a=0x5 b=0x2 c=0x1 sel=0bxx out=0x0
[30] a=0x5 b=0x6 c=0x5 sel=0bzx out=0x0
[40] a=0x5 b=0x4 c=0x1 sel=0bxz out=0x0
[50] a=0x6 b=0x5 c=0x2 sel=0bxz out=0x0
[60] a=0x5 b=0x7 c=0x2 sel=0bzx out=0x0
[70] a=0x7 b=0x2 c=0x6 sel=0bzz out=0x0
[80] a=0x0 b=0x5 c=0x4 sel=0bxx out=0x0
[90] a=0x5 b=0x5 c=0x5 sel=0bxz out=0x0
ncsim: *W,RNQUIE: Simulation is complete.
```

If the case statement in design has x and z in the case item alternatives, the results will differ.

```
module mux (input [2:0] a, b, c, output reg [2:0] out);

  // Case items have x and z, and sel has to match the exact value for
  // output to be assigned with the corresponding input

 always @ (a, b, c, sel) begin
    case(sel)
      2'bxz: out = a;
      2'bzx: out = b;
      2'bxx: out = c;
      default: out = 0;
    endcase
  end

endmodule
```

Differentiation between the case and if-else

The case statement is different from if-else-if in two ways, such as:

- Expressions given in an *if-else* block are more general, while in a case block, a single expression is matched with multiple items.
- *The case* will provide a definitive result when there are X and Z values in an expression.

Verilog Parameters

In Verilog, parameters are constants and do not belong to any other data type such as register or net data types.

A constant expression refers to a constant number or previously defined parameter. We cannot modify parameter values at runtime, but we can modify a parameter value using the *defparam* statement.

The *defparam* statement can modify parameters only at the compilation time. Parameter values can be modified using # delay specification with module instantiation.

In Verilog, there are two methods to override a module parameter value during a module instantiation.

1. By using the defparam keyword.
2. And module instance parameter value assignment.

After the defparam keyword, the hierarchical path is specified to the parameter and the parameter's new value. This new value should be a constant expression. If the right-hand side expression references any parameters, it should be declared within the module where defparam is invoked.

The module instance parameter value assignment method seems like an assignment of delay to gate instance. This method overrides parameters inside instantiated modules as they appear in the module. Using this format, parameters cannot be skipped.

Constant expressions can contain previously declared parameters. When changes are detected on the previously declared parameters, all parameters that depend on this value are updated automatically.

Consider, a 4-bit adder can be parameterized to accept a value for the number of bits, and new parameter values can be passed during module instantiation. So, an N-bit adder converts into a 4-bit, 8-bit or 16-bit adder. They are like arguments to a function that is passed during a function call.

- parameter MSB = 7;   // MSB is a parameter with the constant value 7
- parameter REAL = 4.5;   // REAL holds the real number

```
parameter FIFO_DEPTH = 256,
          MAX_WIDTH = 32;          // Declares two parameters
parameter [7:0] f_const = 2'b3;     // 2 bit value is converted into 8 bits; 8'b3
```

There are two types of parameters, *module* and *specify*, and both accept a range specification. But, they are made as wide as the value to be stored them, and hence a range specification is not necessary.

Module parameters

It can be used to override parameter definitions within a module and makes the module have a different set of parameters at compile time. A parameter can be modified with the *defparam* statement. It is common to use uppercase letters in names for the parameter to notice them instantly.

The below module uses parameters to specify the bus width, data width and the depth of FIFO within the design, and can be overridden with new values when the module is instantiated or by using defparam statements.

```
module design_ip  ( addr, wdata, write, sel, rdata);
    parameter  BUS_WIDTH    = 32,
               DATA_WIDTH   = 64,
               FIFO_DEPTH   = 512;
    input addr;
    input wdata;
    input write;
    input sel;
    output rdata;
    wire [BUS_WIDTH-1:0] addr;
    wire [DATA_WIDTH-1:0] wdata;
    reg  [DATA_WIDTH-1:0] rdata;
    reg [7:0] fifo [FIFO_DEPTH];
endmodule
```

In the new ANSI style of Verilog port declaration, we may declare parameters such as:

```
module design_ip
   #(parameter BUS_WIDTH=32,
       parameter DATA_WIDTH=64)
         (input [BUS_WIDTH-1:0] addr,
// other port declarations
  );
```

Overriding parameters

Parameters can be overridden with new values during module instantiation. The first part is the module called *design_ip* by the name d0 where new parameters are passed within # ( ).

The second part is use a Verilog construct called *defparam* to set the new parameter values. The first method is commonly used to pass new parameters in RTL designs. And the second method is used in testbench simulations to quickly update the design parameters without having to reinstantiate the module.

```
module tb;
    // Module instantiation override
    design_ip  #(BUS_WIDTH = 64, DATA_WIDTH = 128) d0 ( [port list]);
   // Use of defparam to override
    defparam d0.FIFO_DEPTH = 128;
endmodule
```

The module counter has two parameters *N* and *DOWN*, which is declared to have a default value of 2 and 0.

N controls the number of bits in the output, effectively controlling the width of the counter. It is a 2-bit counter by default.

Parameter *DOWN* controls whether the counter should increment or decrement. The counter will decrement because the parameter is set to 0.

2-bit up Counter

```
module counter
  # ( parameter N = 2, parameter DOWN = 0)

  (input clk, input rstn, input en, output reg [N-1:0] out);

  always @ (posedge clk) begin
    if (!rstn) begin
      out <= 0;
    end else begin
      if (en)
        if (DOWN)
          out <= out - 1;
        else
          out <= out + 1;
      else
```

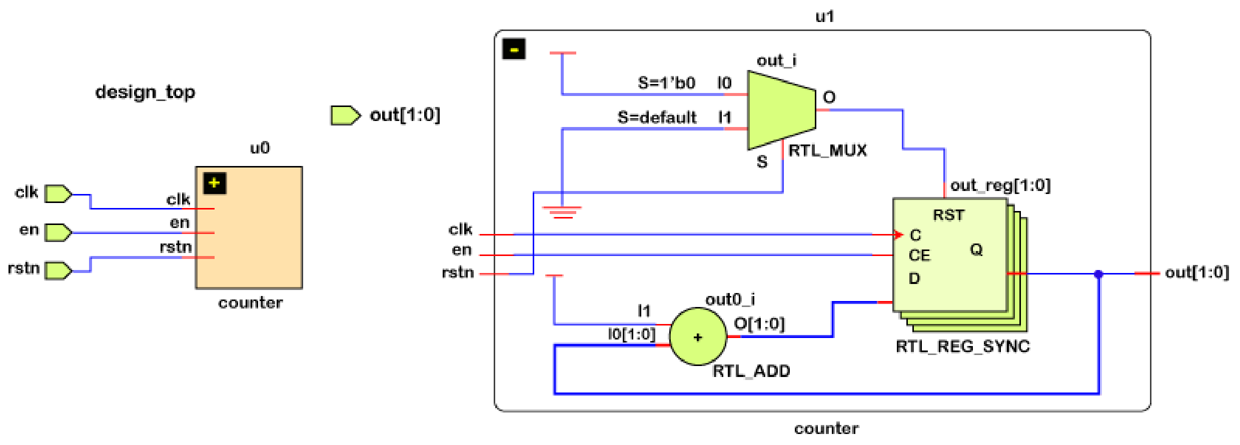
```
        out <= out;
    end
  end
endmodule
```

The module counter is instantiated with *N* as 2 even though it is not required because the default value is anyway 2.

DOWN is not passed during module instantiation. And it takes the default value of 0 making it an up-counter.

```
module design_top (input clk, input rstn, input en, output [1:0] out);
  counter #(.N(2)) u0 (.clk(clk), .rstn(rstn), .en(en));
endmodule
```

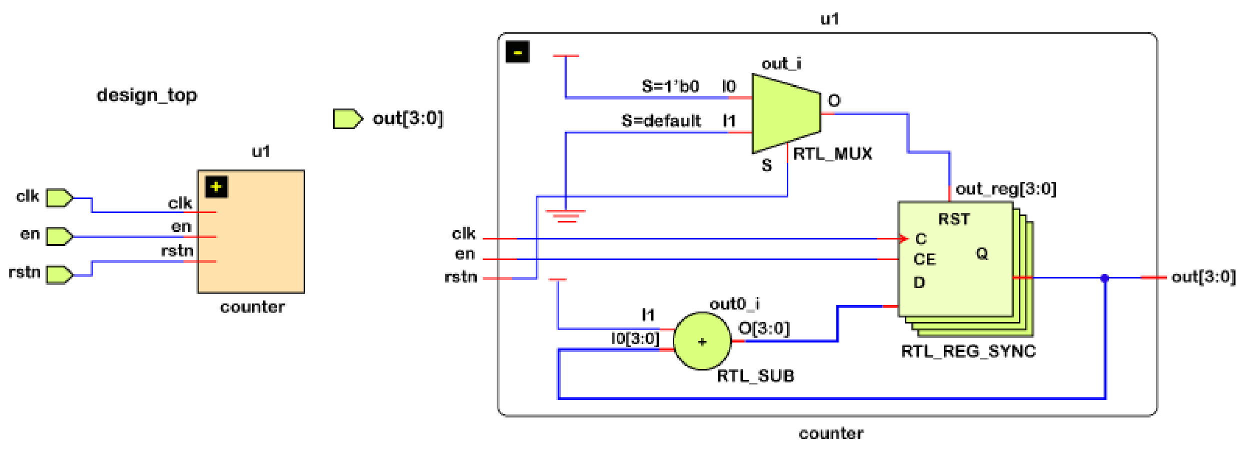
The default parameters are used to implement the counter where *N* equals two, making it a 2-bit counter, and *DOWN* equals zero, making it an up-counter. The output from the counter is left unconnected at the top level.



4-bit down Counter

In this case, the module counter is instantiated with N as 4 making it a 4-bit counter. DOWN is passed a value of 1 during the module instantiation and hence a down-counter is implemented.

```
module design_top (input clk, input rstn, input en, output [3:0] out);
  counter #(.N(4), .DOWN(1))
           u1 (.clk(clk), .rstn(rstn), .en(en));
endmodule
```



Specify Parameters

These parameters are used to provide time and delay values and declared using the *specparam* keyword. It is allowed to use both within the specified block and the main module body.

```
// Use of specify block
Specify
   specparam  t_rise = 200, t_fall = 150;
   specparam  clk_to_q = 70, d_to_q = 100;
endspecify
// Within main module
module  my_block ( );
   specparam  dhold = 2.0;
   specparam  ddly  = 1.5;
   parameter  WIDTH = 32;
endmodule
```

Difference between Specify and Module Parameters

Specify parameter	Module parameter
Specify the specparam keyword declares parameter.	The module parameter is declared by parameter.
It can be declared inside a specific block or within the main module.	It can only be declared within the main module.
This parameter may be assigned specparams and parameters.	This may not be assigned specparams.

SDF can be used to override values.	Instance declaration parameter values or defparam can be used to override.

Notes

Here are some important notes for the Verilog parameters, such as:

- If we are using the *defparam* statement, we must specify a hierarchical path to the parameter.
- We cannot skip over a parameter in a *module instance parameter value assignment*. If we need to do this, use the initial value for a not overwritten parameter.
- When one parameter depends on the other, then the second will automatically be updated if we change the first one.

Verilog Timing Control

Timing control statements are required in simulation to advance time. The time at which procedural statements will get executed shall be specified using timing controls.

There are the following types of timing controls in Verilog:

- Delay control
- Edge sensitive event control
- Level sensitive event control
- Named events

The *delay* control is a way of adding a delay between when the simulator encounters the statement and when it executes.

The *event* expression allows the statement to be delayed until the occurrence of some simulation event, which can change of value on a net or variable or an explicitly named event triggered in another procedure.

Simulation time can be advanced by one of the following methods. Nets and gates have been modeled to have internal delays, also advance simulation time.

Delay Control

Delay control is achieved by specifying the waiting time to execution when the statement is encountered. The symbol # is used to specify the delay.

We can specify the delay based timing control in three ways:

1. **Regular delay control:** It will be specified on the procedural assignment left as a non-zero number.
2. **Intra- assignment delay control:** In this case, delays will be specified on the assignment operator's right-hand side. The right-hand side expression will be evaluated at the current time, and the assignment will occur only after the delay.
3. **Zero delay control:** Zero delay control statement specifies zero delay value to the left-hand side of a procedural assignment. This method is used to ensure the statement is executed at the end of the simulation time. It means, zero delay control statement is executed after all other statements in that simulation time are executed.

Event Control

An event controls the execution of a statement or a block of a statement. Value changes on variables and nets can be used as a synchronization event to trigger the execution of other procedural statements and is an *implicit* event.

The event is based on the direction of change like towards 0, which makes it a *negedge* and change towards 1 make it a *posedge*.

- A negedge is a transition from 1 to X, Z or 0 and from X or Z to 0
- A posedge is a transition from 0 to X, Z or 1 and from X or Z to 1

A transition from the same state to the same state does not suppose to be an edge. The edge event can be detected only on the LSB of a vector signal or variable.

If an expression evaluates to the same result, then it cannot be considered as an event. There are different types of event-based controls.

1. Regular event control: Execution of statement will happen on signal changes or at positive or negative transitions of signals. For example, posedge of a clock, and the negedge of reset, etc.

1. @(posedge clock) out = in;
2. @(clock) z = n << 2;

2. Named Event Control: The event keyword can be used to declare a named event that can be triggered explicitly.

An event cannot hold any data, no time duration, and can be made to occur at any particular time.

A named event is triggered by the -> operator by prefixing it before the named event handle. A named event can be waited upon through the @ operator.

- module tb;

```
event a_event;
event b_event[5];

initial begin
  #20 -> a_event;

  #30;
  ->a_event;

  #50 ->a_event;
  #10 ->b_event[3];
end

always @ (a_event) $display ("T=%0t [always] a_event is triggered", $time);

initial begin
  #25;
  @(a_event) $display ("T=%0t [initial] a_event is triggered", $time);

  #10 @(b_event[3]) $display ("T=%0t [initial] b_event is triggered", $time);
end

endmodule
```

Named events are used to synchronize two or more concurrently running processes. Events can be declared as arrays.

For example, the *always* block and the second *initial* block are synchronized by a_event. Events can be declared as arrays such as in the case of b_event, which is an array of size 5, and index 3 is used for trigger and wait for purpose.

3. Event OR control: The transitions of signal or event can trigger statements' execution, as shown below.

The or operator can wait until any one of the listed events is triggered in an expression. The comma (,) can be used instead of the or operator.

- always @(clock or in)   //Wait for the clock or in to change
- OR
- always @(clock, in)

Edge Sensitive Event Control

In Verilog, the @ character specifies an edge-sensitive event control that blocks until there is a transition in value (an edge) for one of the event's identifiers.

The edge events are queued and then serviced by @(...) guards, rather than @(?) being a guard that waits on edge events, blocking until an edge event happens.

The only edge events that matter to an @(...) guard are those that happen while it is waiting. Edge events that happen before reaching the guard are irrelevant to the guard.

Level Sensitive Event Control

A procedural statement's execution can be delayed until a condition becomes true and accomplished with the *wait* keyword. And it is a level-sensitive control.

The wait statement evaluates a condition, and if it is false, the procedural statements remain blocked until the condition becomes true.

```
module tb;
  reg [3:0] ctr;
  reg clk;

  initial begin
    {ctr, clk} <= 0;
    wait (ctr);
    $display ("T=%0t Counter reached non-zero value 0x%0h", $time, ctr);
    wait (ctr == 4) $display ("T=%0t Counter reached 0x%0h", $time, ctr);
    $finish;
  end
  always #10 clk = ~clk;
always @ (posedge clk)
    ctr <= ctr + 1;
endmodule
```

After completion the execution, it produces the following output:

```
ncsim> run
T=10 Counter reached non-zero value 0x1
T=70 Counter reached 0x4
T=90 Counter reached 0x5
T=170 Counter reached 0x9
Simulation complete via $finish(1) at time 170 NS + 1
```

Implicit Event Expression List

The event expression list or sensitivity list is often a common cause for many functional errors in the RTL because the user may forget to update the sensitivity list after introducing a new signal in the procedural block.

```
module tb;
   reg a, b, c, d;
   reg x, y;

 // signals inside () after @ operator
   // it is a, b, c or d

   always @ (a, b, c, d) begin
      x = a | b;
      y = c ^ d;
   end

   initial begin
         $monitor ("T=%0t a=%0b b=%0b c=%0b d=%0b x=%0b y=%0b", $time, a, b, c, d, x, y);
      {a, b, c, d} = 0;

      #10 {a, b, c, d} = $random;
      #10 {a, b, c, d} = $random;
      #10 {a, b, c, d} = $random;
   end

endmodule
```

Now execute the above code and we will get the following output:

```
ncsim> run
T=0 a=0 b=0 c=0 d=0 x=0 y=0
T=10 a=0 b=1 c=0 d=0 x=1 y=0
T=20 a=0 b=0 c=0 d=1 x=0 y=1
T=30 a=1 b=0 c=0 d=1 x=1 y=1
ncsim: *W,RNQUIE: Simulation is complete.
```

If the user decides to add new signal e and capture the inverse into z, then special care must be taken to add e also into the sensitivity list.

```
module tb;
   reg a, b, c, d, e;
```

```
  reg x, y, z;

// Add "e" also into sensitivity list
always @ (a, b, c, d, e) begin
     x = a | b;
     y = c ^ d;
     z = ~e;
  end

  initial begin
    $monitor ("T=%0t a=%0b b=%0b c=%0b d=%0b e=%0b x=%0b y=%0b z=%0b",
                         $time, a, b, c, d, e, x, y, z);
    {a, b, c, d, e} = 0;

    #10 {a, b, c, d, e} <= $random;
    #10 {a, b, c, d, e} <= $random;
    #10 {a, b, c, d, e} <= $random;
  end

endmodule
```

And the output is given below:

```
ncsim> run
T=0 a=0 b=0 c=0 d=0 e=0 x=0 y=0 z=1
T=10 a=0 b=0 c=1 d=0 e=0 x=0 y=1 z=1
T=20 a=0 b=0 c=0 d=0 e=1 x=0 y=0 z=0
T=30 a=0 b=1 c=0 d=0 e=1 x=1 y=0 z=0
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog allows the sensitivity list to be replaced by * which is a convenient shorthand that eliminates these problems by adding all nets and variables read by the statement as shown in the below code.

```
module tb;
  reg a, b, c, d, e;
  reg x, y, z;

// Use @* or @(*)
always @ * begin
     x = a | b;
     y = c ^ d;
```

```
        z = ~e;
    end

    initial begin
      $monitor ("T=%0t a=%0b b=%0b c=%0b d=%0b e=%0b x=%0b y=%0b z=%0b",
                      $time, a, b, c, d, e, x, y, z);
      {a, b, c, d, e} = 0;

      #10 {a, b, c, d, e} <= $random;
      #10 {a, b, c, d, e} <= $random;
      #10 {a, b, c, d, e} <= $random;
    end

endmodule
```

The output looks as below:

```
ncsim> run
T=0 a=0 b=0 c=0 d=0 e=0 x=0 y=0 z=1
T=10 a=0 b=0 c=1 d=0 e=0 x=0 y=1 z=1
T=20 a=0 b=0 c=0 d=0 e=1 x=0 y=0 z=0
T=30 a=0 b=1 c=0 d=0 e=1 x=1 y=0 z=0
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Inter and Intra Delay

Verilog delay statements can have delays specified either on the left-hand side of the assignment operator's right-hand side.

Inter Assignment Delays

An inter-assignment delay statement has delay value on the left-hand side of the assignment operator.

Inter assignment are those delay statements where the execution of the entire statement or assignment got delayed.

In Verilog, Inter assignment delays often correspond to the inertial delay or the VHDL's regular delay statements.

- // Delay is specified on the left side
- #<delay> <LHS> = <RHS>

It indicates that the statement itself is executed *after* the delay expires, and is the most commonly used form of delay control.

Example

```
module vd;
  reg  a, b, c, q;

  initial begin
    $monitor("[%0t] a=%0b b=%0b c=%0b q=%0b", $time, a, b, c, q);

    // Now initialize all signals to 0 at time 0
    a <= 0;
    b <= 0;
    c <= 0;
    q <= 0;

    // Inter-assignment delay. Wait for #5 time units
    // and then assign a and c to 1. Note that 'a' and 'c'
    // gets updated at the end of current timestep
    #5  a <= 1;
        c <= 1;

    // Inter-assignment delay. Wait for #5 time units
  // and then assign 'q' with whatever value RHS gets
    // evaluated to
    #5 q <= a & b | c;
    #20;
  end
endmodule
```

Here, q becomes 1 at time 10 units because the statement gets evaluated at 10 time units and RHS which is a combination of a, b and c evaluates to 1. After completion the execution, it gives the following output.

```
xcelium> run
[0] a=0 b=0 c=0 q=0
[5] a=1 b=0 c=1 q=0
[10] a=1 b=0 c=1 q=1
xmsim: *W,RNQUIE: Simulation is complete.
```

Intra Assignment Delays

Intra assignment delay indicates that the statement itself is executed *after* the delay expires, and it is the most commonly used form of delay control.

They can be used with blocking and non-blocking assignments. If a statement with intra-assignment timing controls is encountered during simulation, then the expression will be evaluated, and its value will be stored.

Then, the statement's execution will be suspended until the time specified by the delay control expires. Changes in the expression value up to the time of the event will be ignored.

```
// Delay is specified on the right side
<LHS> = #<delay> <RHS>
```

An intra-assignment delay is declared to the right-hand side of the assignment operator. This indicates that the statement is evaluated, and the values of all signals on RHS are captured first.

Example

```
module vd;
  reg  a, b, c, q;
  initial begin
    $monitor("[%0t] a=%0b b=%0b c=%0b q=%0b", $time, a, b, c, q);
    // Initialize all signals to 0 at time 0
    a <= 0;
    b <= 0;
    c <= 0;
    q <= 0;
    // Inter-assignment delay: Wait for #5 time units
    // and then assign a and c to 1. Note that 'a' and 'c'
    // gets updated at the end of current timestep
    #5  a <= 1;
        c <= 1;
    // Intra-assignment delay: First execute the statement
    // then wait for 5 time units and then assign the evaluated
    // value to q
    q <= #5 a & b | c;
    #20;
  end
endmodule
```

The above code gives the below output:

```
xcelium> run
[0] a=0 b=0 c=0 q=0
[5] a=1 b=0 c=1 q=0
xmsim: *W,RNQUIE: Simulation is complete.
```

At 5 time units, a and c are assigned using non-blocking statements. And the behavior of non-blocking statements is evaluated but gets assigned to the variable only at the end of the time step.

So the value of a and c is evaluated to 1 but not assigned when the next non-blocking statement q is executed. So when RHS of q is evaluated, a and c still has an old value of 0, and hence $monitor does not detect a change to display the statement.

To observe the change, let us change assignment statements to a and c from non-blocking to blocking.

```
// Non-blocking changed to blocking
// rest of the code remains the same
        #5  a = 1;
    c = 1;
     q <= #5 a & b | c;
```

And the output looks like as:

```
xcelium> run
[0] a=0 b=0 c=0 q=0
[5] a=1 b=0 c=1 q=0
[10] a=1 b=0 c=1 q=1
xmsim: *W,RNQUIE: Simulation is complete.
```

Verilog Gate Delay

Verilog gate delays specify how values propagate through nets or gates. The gate delay declaration specifies a time needed to propagate a signal change from the gate input to its output.

The gate delay declaration can be used in gate instantiations. The delays can also be used for delay control in procedural statements.

Digital elements are binary entities and only hold either of the two values, 0 and 1. The transition from 0 to 1 and 1 to 0 has a transitional delay, and therefor each gate element propagates the value from input to its output.

For example, a two-input AND gate has to switch the output to 1 if both inputs become 1 and back to 0 when inputs become 0.

The net delay declaration specifies a time needed to propagate values from drivers through the net. It can be used in continuous assignments and net declarations.

This gate and pin to pin delays can be specified in Verilog when instantiating logic primitives.

Rise, Fall, and Turn-Off Delays

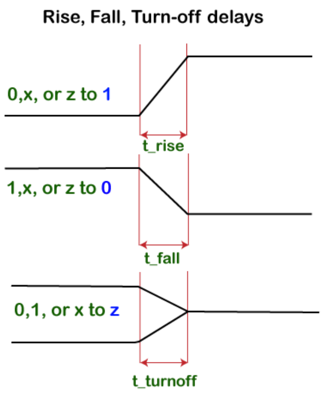
The delays declaration can contain up to three values, such as *rise, fall,* and *turn-off* delays.

- The time taken for the output of a gate to change from some value to 1 is called a *rise* delay.
- The time taken for the output of a gate to change form some value to 0 is called a *fall* delay.
- The time taken for the output of a gate to change from some value to high impedance is called *turn-off* delay.

If only one delay value is specified, then it is used for all signal changes. The default delay is zero.

If two delays are specified, then the first delay specifies the rise delay, and the second delay specifies the fall delay.

If the signal changes to high-impedance or unknown, then the smaller value will be used.



If three values are given, then the first value specifies the rise delay, the second specifies the *fall* delay, and the third specifies *the turn-off* delay. If the signal changes to an unknown value, then the smallest of these three values will be used.

These delays apply to any signal as they all can rise or fall anytime in real circuits and are not restricted to only outputs of gates. There are three ways to represent gate delays.

1. One delay format
2. Two delay format
3. Three delay format

The two delay format can be applied to most primitives whose outputs do not transition to high impedance.

A three delay format cannot be applied to an AND gate because the output will not go to Z for any input combination.

- // Single delay specified - used for all three types of transition delays

- or #(<delay>) o1 (out, a, b);
-
- // Two delays specified - used for Rise and Fall transitions
- or #(<rise>, <fall>) o1 (out, a, b);
- // Three delays specified - used for Rise, Fall and Turn-off transitions
- or #(<rise>, <fall>, <turn_off>) o1 (out, a, b);

If only a single delay is specified, all three types of delays will use the same given value.

If there are two delays specified, the first one represents *the rise*, and the second one represents the *fall* delay.

If there are three delays specified, they represent *rise, fall*, and *turn-off* delays, respectively.

1. One Delay Format

- module des (input a, b, output out1, out2);
- // AND gate has 2 time unit gate delay
-     and #(2) o1 (out1, a, b);
-   // BUFIF0 gate has 3 time unit gate delay
- bufif0 #(3) b1 (out2, a, b);
- endmodule

Now, See that the output of *AND* gates change 2 time units after one of its inputs change.

- module tb;
-   reg a, b;
- wire out1, out2;
-   des d0 (.out1(out1), .out2(out2), .a(a), .b(b));
-   initial begin
- {a, b} <= 0;
-     $monitor ("T=%0t a=%0b b=%0b and=%0b bufif0=%0b", $time, a, b, out1, out2);
-     #10 a <= 1;
-     #10 b <= 1;
-     #10 a <= 0;
-     #10 b <= 0;
- end
- endmodule

For example, b becomes 1 while a is already 1 at T=20. But the output becomes 1 only at T=22. Similarly, a goes back to zero at T=30, and the output gets the new value at T=32.

Gate delay is specified as 3-time units for *BUFIF0*. If b changes from 0 to 1 while a is already at 1, then the output takes 3-time units to get updated to Z and finally does so at T=23.

Output

```
ncsim> run
T=0 a=0 b=0 and=x bufif0=x
T=2 a=0 b=0 and=0 bufif0=x
T=3 a=0 b=0 and=0 bufif0=0
T=10 a=1 b=0 and=0 bufif0=0
T=13 a=1 b=0 and=0 bufif0=1
T=20 a=1 b=1 and=0 bufif0=1
T=22 a=1 b=1 and=1 bufif0=1
T=23 a=1 b=1 and=1 bufif0=z
T=30 a=0 b=1 and=1 bufif0=z
T=32 a=0 b=1 and=0 bufif0=z
T=40 a=0 b=0 and=0 bufif0=z
T=43 a=0 b=0 and=0 bufif0=0
ncsim: *W,RNQUIE: Simulation is complete.
```

2. Two Delay Format

Let's apply the same *testbench* shown above to a different Verilog model shown below where *rise* and *fall* delays are explicitly mentioned.

```
module des (input a, b, output out1, out2);
  and #(2, 3) o1 (out1, a, b);
  bufif0 #(4, 5) b1 (out2, a, b);
endmodule
```

And it produces the following output, such as:

```
ncsim> run
T=0 a=0 b=0 and=x bufif0=x
T=3 a=0 b=0 and=0 bufif0=x
T=5 a=0 b=0 and=0 bufif0=0
T=10 a=1 b=0 and=0 bufif0=0
T=14 a=1 b=0 and=0 bufif0=1
T=20 a=1 b=1 and=0 bufif0=1
T=22 a=1 b=1 and=1 bufif0=1
T=24 a=1 b=1 and=1 bufif0=z
T=30 a=0 b=1 and=1 bufif0=z
T=33 a=0 b=1 and=0 bufif0=z
T=40 a=0 b=0 and=0 bufif0=z
T=45 a=0 b=0 and=0 bufif0=0
ncsim: *W,RNQUIE: Simulation is complete.
```

3. Three Delay Format

```
module des (    input   a, b, output out1, out2);
and #(2, 3) o1 (out1, a, b);
bufif0 #(5, 6, 7) b1 (out2, a, b);
endmodule
```

The three delay format code gives the following output:

```
ncsim> run
T=0 a=0 b=0 and=x bufif0=x
T=3 a=0 b=0 and=0 bufif0=x
T=6 a=0 b=0 and=0 bufif0=0
T=10 a=1 b=0 and=0 bufif0=0
T=15 a=1 b=0 and=0 bufif0=1
T=20 a=1 b=1 and=0 bufif0=1
T=22 a=1 b=1 and=1 bufif0=1
T=27 a=1 b=1 and=1 bufif0=z
T=30 a=0 b=1 and=1 bufif0=z
T=33 a=0 b=1 and=0 bufif0=z
T=40 a=0 b=0 and=0 bufif0=z
T=46 a=0 b=0 and=0 bufif0=0
ncsim: *W,RNQUIE: Simulation is complete.
```

Min, Typ, and Max Delays

Delays are neither the same in different parts of the fabricated chip nor the same for different temperatures and other variations. So Verilog also provides an extra level of control for each of the delay types mentioned above.

Every digital gate and transistor cell has a minimum, typical, and maximum delay specified based on process node and is typically provided by libraries from fabrication foundry.

For rise, fall, and turn-off delays, the three values *min, typ,* and *max* can be specified and stand for minimum, typical and maximum delays.

This is another level of delay control in Verilog. Only one of the min, typ, and max values can be used in the entire simulation run.

It is specified at the start of the simulation and depends on the simulator used. The typ is the default value.

The *min* value is the minimum delay value that the gate is expected to have.

The *typ* value is the typical delay value that the gate is expected to have.

The *max* value is the maximum delay value that the gate is expected to have.

```
module des (input a, b, output out1, out2);
  and #(2:3:4, 3:4:5) o1 (out1, a, b);
  bufif0 #(5:6:7, 6:7:8, 7:8:9) b1 (out2, a, b);
endmodule
```

The output looks like:

```
ncsim> run
T=0 a=0 b=0 and=x bufif0=x
T=4 a=0 b=0 and=0 bufif0=x
T=7 a=0 b=0 and=0 bufif0=0
T=10 a=1 b=0 and=0 bufif0=0
T=16 a=1 b=0 and=0 bufif0=1
T=20 a=1 b=1 and=0 bufif0=1
T=23 a=1 b=1 and=1 bufif0=1
T=28 a=1 b=1 and=1 bufif0=z
T=30 a=0 b=1 and=1 bufif0=z
T=34 a=0 b=1 and=0 bufif0=z
T=40 a=0 b=0 and=0 bufif0=z
T=47 a=0 b=0 and=0 bufif0=0
ncsim: *W,RNQUIE: Simulation is complete.
```

Data Flow Modeling

Dataflow modeling makes use of the functions that define the working of the circuit instead of its gate structure.

Dataflow modeling has become a popular design approach, as logic synthesis tools became sophisticated. This approach allows the designer to focus on optimizing the circuit in terms of the flow of data.

Dataflow modeling uses several operators that act on operands to produce the desired results. Verilog provides about 30 operator types. Dataflow modeling describes hardware in terms of the flow of data from input to output.

The dataflow modeling style is mainly used to describe combinational circuits. The primary mechanism used is a continuous assignment.

Continuous Assignments

A value is assigned to a data type called *net*, which is used to represent a physical connection between circuit elements in a continuous assignment. The value assigned to the net is specified by an expression that uses operands and operators.

A continuous assignment replaces gates in the circuit's description and describes the circuit at a higher level of abstraction. A continuous assignment statement starts with the keyword *assign*.

Syntax

The syntax of a continuous assignment is

1. assign [delay] LHS_net = RHS_expression;

LHS_net is a destination net of one or more bit, and RHS_expression is an expression of various operators.

The statement is evaluated at any time any of the source operand value changes, and the result is assigned to the destination net after the delay unit.

- The LHS of the assign statement must always be a scalar or vector net or a concatenation. It cannot be a register.
- Continuous statements are always active statements, which means that if any value on the RHS changes, LHS changes automatically.
- Registers or nets or function calls can come in the RHS of the assignment.
- The RHS expression is evaluated whenever one of its operands changes. Then the result is assigned to the LHS.
- Delays can be specified in the assign statement.

Example

1. assign out1 = in1 & in2; // perform and function on in1 and in2 and assign the result to out 1
2. assign out2 = not in1;
3. assign #2 z[0] = ~(ABAR & BBAR & EN); // perform the desired function and assign the re sult after 2 units

The target in the continuous assignment expression can be one of the following:

1. A scalar net
2. Vector net
3. Constant bit-select of a vector

4. Constant part-select of a vector
5. Concatenation of any of the above

Let us take another set of examples in which a scalar and vector nets are declared and used

- wire COUNT, CIN; // scalar net declaration
- wire [3:0] SUM, A, B; // vector nets declaration
- assign {COUT,SUM} = A + B + CIN; // A and B vectors are added with CIN

NOTE: Multiple continuous assignment statements are not allowed on the same destination net.

Continuous Assignment on Vectors

As described in the characteristics, the continuous assignment can be performed on vector nets.

```
module adder(a,b,sum);
input [2:0] a,b;
output [3:0] sum;

assign sum = a + b;
$display("a = %b, b = %b, sum=%b", a,b,sum);
endmodule
```

The above code describes a 3-bit adder. The MSB of the sum is dedicated to carry in the above module. It generates the following output:

```
a = 100, b = 111, sum = 1011                    // (a = 4, b = 7, sum = 011, carry = 1)
```

The concatenation of vector and scalar nets is also possible. The same example for 3-bit adder is shown by using concatenation:

```
module adder(a,b,sum);
input [2:0] a,b;
output [2:0] sum; //sum is a vector
output carry; // carry is a scalar

assign {carry,sum} = a + b; //assigning result to a concatenation of scalar and vector
$display("a = %b, b = %b, sum=%b, carry = %b", a,b,sum,carry);
endmodule
```

The output is:

```
a = 100, b = 111, sum = 011, carry = 1
```

1. Regular Continuous Assignment

It follows the following steps, such as:

Step 1: Declare net.

Step 2: Write a continuous assignment on the net.

The below code follows Regular continuous assignment:

- wire out; // net 'out' is declared
- assign out = a&b; //continuous assignment on declared net

2. Implicit Continuous Assignment

We can also place a continuous assignment on a net when it is declared. The format will look like the below:

- wire out = a & b; // net declaration and assignment together

3. Implicit Net Declaration

In Verilog, during an implicit assignment, if LHS is declared, it will assign the RHS to the declared *net*, but if the LHS is not defined, it will automatically create a net for the signal name.

- Wire in0, in1;
- Assign out = in0 ^ in1;

In the above example, *out* is undeclared, but Verilog makes an implicit net declaration for out.

Delays

In real-world hardware, there is a time gap between change in inputs and the corresponding output.

For example, a delay of 2 ns in an *AND* gate implies that the output will change after 2 ns from the time input has changed.

Delay values control the time between the change in an RHS operand and when the new value is assigned to LHS. It is similar to specifying delays for gates. Adding delays helps in modeling the timing behavior in simple circuits.

It is getting us closer to simulating the practical reality of a functioning circuit. There are different ways to specify a delay in continuous assignment statements, such as:

1. Regular Assignment Delay

We assign a delay value in the continuous assignment statement. The delay value is specified after the *assign* keyword.

This delay is applicable when the signal in LHS is already defined, and this delay represents the delay in changing the value of the already declared net. For example,

1. Assign #10 out = in0 | in1;

If there is any change in the RHS operands, then RHS expression will be evaluated after 10 units of time and the evaluated expression will be assigned to LHS.

At time t, if there is a change in one of the operands in the above example, then the expression is calculated at t+10 units of time.

It means that if in0 or in1 changes value before 10-time units, then the values of in1 and in2 at the time of re-computation (t+10) are considered.

2. Implicit Continuous Assignment Delay

Here, we use an implicit continuous assignment to specify both a delay and an assignment on the net.

- wire #10 out = in0 ^ in1;   //implicit Continuous Assignment Delay.

Is same as

- wire out
- assign #10 out = in0 ^ in1;

3. Net Declaration Delay

In this case, the delay is associated with the net instead of the assignment.

Here, a delay is added when the net is declared without putting continuous assignment.

- wire #10 out;
- assign out = in;

This code has the same effect as the following:

- wire out
- assign #10 out = in;

Gate Level Modeling

In Verilog, most of the digital designs are done at a higher level of abstraction like RTL. However, it becomes natural to build smaller deterministic circuits at a lower level by using combinational elements such as *AND* and *OR*.

Modeling done at this level is called gate-level modeling as it involves *gates* and has a one to one relationship between a hardware schematic and the Verilog code.

Verilog supports a few basic logic gates known as *primitives*, as they can be instantiated, such as modules, and they are already predefined.

Gate level modeling is virtually the lowest level of abstraction because the switch-level abstraction is rarely used. Gate level modeling is used to implement the lowest-level modules in a design, such as *multiplexers, full-adder*, etc. Verilog has gate primitives for all basic gates

Verilog supports built-in primitive gates modeling. The gates supported are *multiple-input, multiple-output, tri-state*, and *pull gates*.

The multiple-input gates are *and, nand, or, nor, xor,* and *xnor* whose number of inputs are two or more, and has only one output.

The multiple-output gates are *buf* and *not* whose output is one or more and has only one input.

The language also supports the modeling of tri-state gates, including *bufif0, bufif1, notif0,* and *notif1*. These gates have one input, one control signal, and one output.

The pull gates are *pullup* and *pulldown* with a single output only.

Syntax

Following is the basic syntax for each type of gates with zero delays, such as:

- and | nand | or | nor | xor | xnor [instance name] (out, in1, ..., inN);   // [] is optional an d | is selection
- buf | not [instance name] (out1, out2, ..., out2, input);
- bufif0 | bufif1 | notif0 | notif1 [instance name] (outputA, inputB, controlC);
- pullup | pulldown [instance name] (output A);

One can also have multiple instances of the same type of gate in one construct separated by a comma:

- and [inst1] (out11, in11, in12), [inst2] (out21, in21, in22, in23), [inst3] (out31, in31, in32, i n33);

The gate-level modeling is useful when a circuit is a simple combinational, such as a *multiplexer*. A multiplexer is a simple circuit that connects one of many inputs to an output.

Gate Primitives

Gate primitives are predefined modules in Verilog, which are ready to use. There are two classes of gate primitives:

1. Single input gate primitives
2. Multiple input gate primitives

1. Single input gate primitives

Single input gate primitives have a single input and one or more outputs. The gate primitive are *not, buf, notif*, and *bufif* also have a control signal.

The gates propagate only if the control signal is asserted, else the output is high impedance state.

1.1 Not, buf Gates

These gates have only one scalar input but may have multiple outputs.

buf stands for a buffer and transfer the value from input to the output without any change in polarity.

not stands for an inverter which inverts the polarity of the signal at its input. So a 0 at its input will yield a 1 and vice versa.

Syntax

```
module gates (  input a, b,
                output c, d);

    buf (c, a, b);  // c is the output, a and b are inputs
     not (d, a, b);  // d is the output, a and b are inputs

endmodule
```

Example

```
module tb;
   reg a, b;
   wire c, d;
   integer i;
gates u0 ( .a(a), .b(b), .c(c), .d(d));
   initial begin
```

```
     {a, b} = 0;
    $monitor ("[T=%0t a=%0b b=%0b c(buf)=%0b d(not)=%0b", $time, a, b, c, d);
     for (i = 0; i < 10; i = i+1) begin
        #1  a <= $random;
            b <= $random;
    end
end
endmodule
```

1.2 Bufif, Notif Gates

Bufif and notif primitives are buffers and inverters, respectively, with an additional control signal to enable the output is available through bufif and notif primitives.

These gates have a valid output only if the control signal is enabled else, and the output will be in high impedance.

There are two versions of these, one with the normal polarity of control indicated by a 1 such as bufif1 and notif1. And second with the inverted polarity of control indicated by a 0 such as bufif0 and notif0.

Syntax

```
module bufif_notif_gates (output c, d, input a, b);
bufif (c, a, b);   // c is the output, a and b are inputs
notif (d, a, b);  // d is the output, a and b are inputs
endmodule
```

Example

```
module bufif_notif_gates_tb;
  reg a, b;
  wire c, d;
  bufif_notif_gates Instance0 (c, d, a, b);
  initial begin
    a = 0; b = 0;
 #1 a = 0; b = 1;
 #1 a = 1; b = 0;
 #1 a = 1; b = 1;
  end
  initial begin
    $monitor ("T=%t| a=%b |b=%b| c(bufif)=%b |d(notif)=%b", $time, a, b, c, d);
  end
endmodule
```

2. Multiple Input Gate Primitives

Multiple input gate primitives include *AND, OR, XOR, NAND, NOR*, and *XNOR*. They may have multiple inputs and a single output.

2.1 AND, OR, XOR Gates

An AND, OR, and an XOR gate need multiple scalar inputs and produce a single scalar output.

The first terminal in the argument list to these primitives is the output, which changed as any inputs shift.

Syntax

```
module and_or_xor_gates (output c, d, e, input a, b);
  and (c, a, b);   // c is the output, a and b are inputs
  or  (d, a, b);  // d is the output, a and b are inputs
  xor (e, a, b);   // e is the output, a and b are inputs
endmodule
```

Example

```
module and_or_xor_gates_tb;
reg a, b;
wire c, d, e;
and_or_xor_gates Instance0 (c, d, e, a, b);
 initial begin
   a = 0; b = 0;
   #1 a = 0; b = 1;
   #1 a = 1; b = 0;
   #1 a = 1; b = 1;
 end
 initial begin
   $monitor ("T=%t |a=%b |b=%b |c(and)=%b |d(or)=%b |e(xor)=%b", $time, a, b, c, d, e);
 end
endmodule
```

1.2 NAND, NOR, XNOR Gates

The inverse of all the above gates is NAND, NOR, and XNOR. The same design from above is reused only that the primitives are interchanged with their inverse versions.

Syntax

- module nand_nor_xnor_gates (output c, d, e, input a, b);
- nand (c, a, b); // c is the output, a and b are inputs
- nor (d, a, b); // d is the output, a and b are inputs
- xnor (e, a, b); // e is the output, a and b are inputs
- endmodule

Example

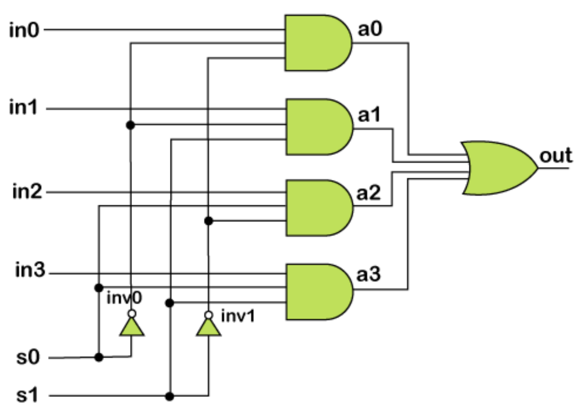
- module nand_nor_xnor_gates_tb;
- reg a, b;
- wire c, d, e;
- nand_nor_xnor_gates Instance0 (c, d, e, a, b);
- initial begin
- a = 0; b = 0;
- #1 a = 0; b = 1;
- #1 a = 1; b = 0;
- #1 a = 1; b = 1;
- end
- initial begin
- $monitor ("T=%t |a=%b |b=%b |c(nand)=%b |d(nor)=%b |e(xnor)=%b", $time, a, b, c, d, e);
- end
- endmodule

All these gates may also have more than two inputs.

- module all_gates (output x1, y1, z1, x2, y2, z2 , input a, b, c, d);
- and (x1, a, b, c, d); // x1 is the output, a, b, c, d are inputs
- or (y1, a, b, c, d); // y1 is the output, a, b, c, d are inputs
- xor (z1, a, b, c, d); // z1 is the output, a, b, c, d are inputs
- nand (x2, a, b, c, d); // x2 is the output, a, b, c, d are inputs
- nor (y2, a, b, c, d); // y2 is the output, a, b, c, d are inputs
- xnor (z2, a, b, c, d); // z2 is the output, a, b, c, d are inputs
- endmodule

Gate Level Modeling of a Multiplexer

The gate-level circuit diagram of 4x1 mux is shown below. It is used to write a module for 4x1 mux.

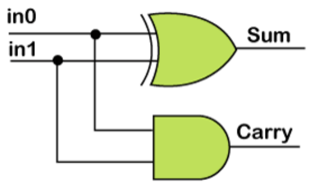


```
module 4x1_mux (out, in0, in1, in2, in3, s0, s1);

// port declarations
output out; // Output port.
input in0, in1, in2. in3; // Input ports.
input s0, s1; // Input ports: select lines.

// intermediate wires
wire inv0, inv1; // Inverter outputs.
wire a0, a1, a2, a3; // AND gates outputs.

// Inverters.
not not_0 (inv0, s0);
not not_1 (inv1, s1);

// 3-input AND gates.
and and_0 (a0, in0, inv0, inv1);
and and_1 (a1, in1, inv0, s1);
and and_2 (a2, in2, s0, inv1);
and and_3 (a3, in3, s0, s1);

// 4-input OR gate.
or or_0 (out, a0, a1, a2, a3);

endmodule
```

Gate Level Modeling of Full-adder

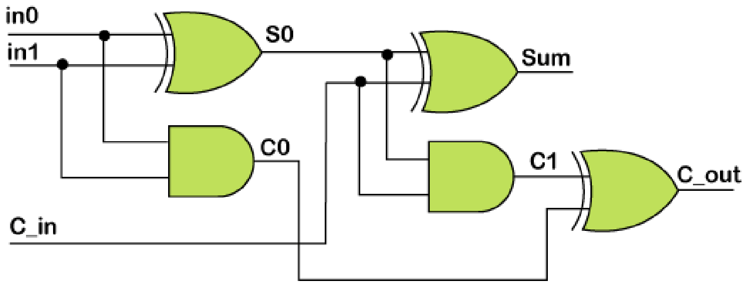
Here is the implementation of a full adder using the half adder.

1. Half adder



```
module half_adder (sum, carry, in0, in1);

output sum, carry;
input in0, in1;

// 2-input XOR gate.
xor xor_1 (sum, in0, in1);

// 2-input AND gate.
and and_1 (carry, in0, in1);

endmodule
```

2. Full adder



```
module full_adder (sum, c_out, ino, in1, c_in);

output sum, c_out;
input in0, in1, c_in;

wire s0, c0, c1;
// Half adder: port connecting by order.
```

```
half_adder ha_0 (s0, c0, in0, in1);

// Half adder : port connecting by name.
half_adder ha_1 (.sum(sum),
            .in0(s0),
             .in1(c_in),
            .carry(c1));

// 2-input XOR gate, to get c_out.
xor xor_1 (c_out, c0, c1);

endmodule
```

Switch Level Modeling

The switch level of modeling provides a level of abstraction between the logic and analog-transistor levels of abstraction. It describes the interconnection of transmission gates, which are abstractions of individual MOS and CMOS transistors.

The switch level transistors are modeled as being either on or off, conducting or not conducting. The values carried by the interconnections are abstracted from the whole range of analog voltages or currents to a small number of discrete values. These values are referred to as signal *strengths*.

Verilog also provides support for transistor level modeling. However, designers rarely use these days as the complexity of circuits has required them to move to higher levels of abstractions rather than use switch level modeling.

Usually, the transistor level modeling is referred to model in hardware structures using transistor models with analog input and output signal values

On the other hand, gate-level modeling refers to modeling hardware structures wing gate models with digital input and output signal values between these two modeling schemes are referred to as switch level modeling.

The transistors only exhibit digital behavior and their input at the transistor level, and output signal values are only limited to digital values.

Verilog uses a 4 value logic value system, so Verilog switch input and output signals can take any of the four *0, 1, Z,* and *X* logic values.

Switch Level Primitives

Switches are unidirectional or bidirectional and resistive or non-resistive for each group, those primitives that switch on with a positive gate such as *NMOS* transistor and those that switch on with a negative gate such as *PMOS* transistor.

- **Switching On:** It means that logic values flow from the input transistor to its input.
- **Switching off:** It means that the transistor's output is at Z level regardless of its input value.
- A unidirectional transistor passes its input value to its output when it is switched on.
- A bidirectional transistor conducts both ways.
- A resistive structure reduces the strength of its input logic when passing it to its output.

MOS switches

Two types of MOS switches can be defined with the keywords *nmos* and *pmos*.

- nmos keyword is used to model NMOS transistors.
- And pmos keyword is used to model PMOS transistors.

In Verilog, nmos and pmos switches are shown as the following:

- nmos n1(out, data, control); // instantiate a nmos switch
- pmos p1(out, data, control); // instantiate a pmos switch

Since switches are Verilog primitives such as logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name.

- nmos (out, data , control); // instantiate nmos switch ; no instance name
- pmos (out, data, control); // instantiate pmos switch; no instance name

The value of the *out* signal is determined from the values of data and control signals.

Some combinations of data and control signals cause the gates to output to either a 1 or 0 or a z value without a preference for either value. The symbol L stands for 0 or Z; H stands for 1 or z.

Thus, the nmos switch conducts when its control signal is 1. If the control signal is 0, the output assumes a high impedance value. Similarly, a pmos switch conducts if the control signal is 0.

CMOS Switches

CMOS switches are declared with the **cmos** keyword. A cmos device can be modeled with a nmos and a pmos device. A CMOS switch is instantiated as shown below, such as:

- cmos cl(out, data, ncontrol, pcontrol); //instantiate cmos gate
- or
- cmos (out, data, ncontrol, pcontrol); //no instance name given

The *ncontrol* and *pcontrol* usually are complements of each other. When the ncontrol signal is 1, and the pcontrol signal is 0, the switch conducts. If ncontrol is 0 and pcontrol is 1, the output of the switch is high impedance value.

The cmos gate is essentially a combination of two gates: one *nmos* and one *pmos*. Thus the cmos instantiation shown above is equivalent to the following.

- nmos (out, data, ncontrol); //instantiate a nmos switch
- pmos (out, data, pcontrol); //instantiate a pmos switch

Bidirectional Switches

NMOS, PMOS and CMOS gates conduct from drain to source. It is important to have devices that conduct in both directions.

In such cases, signals on either side of the device can be the driver signal. Bidirectional switches are provided for this purpose.

Three keywords are used to define bidirectional switches, such as:

1. tran: The tran switch acts as a buffer between the two signals *inout1* and *inout2*. Either inoutl or inout2 can be the driver signal.

- module des (input io1, ctrl, output io2);
-
-   tran (io1, io2);
-
- endmodule

2. tranif0: The tranif0 switch connects the two signals *inout1* and *inout2* only if the *control* signal is logical 0. If the control signal is a logical 1, then the nondriver signal gets a high impedance value z. The driver signal retains the value from its driver.

- module des (input io1, ctrl, output io2);
-
-   tranif0 (io1, io2, ctrl);
-
- endmodule

3. Tranif1: The tranif1 switch conducts if the *control* signal is a logical 1.

- module des (input io1, ctrl, output io2);
-
-   tranif1 (io1, io2, ctrl);
-
- endmodule

Resistive switches reduce signal strengths when signals pass through them. Regular switches retain strength levels of signals from input to output.

Verilog User Defined Primitives

A modeling technique whereby the user can virtually argument predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs). These primitives are self-contained and do not instantiate other primitives or modules.

Verilog provides a standard set of primitives, such as *AND, NAND,* NOT, *OR,* and *NOR*, as a part of the language. These are also known as built-in primitives.

Instances of these new UDPs can be used in the same manner as the gate primitives to represent the circuit being modeled. This technique can reduce the amount of memory and improve simulation performance. The Verilog-XL algorithm accelerates the evaluation of these UDPs.

However, designers occasionally like to use their custom-built primitives when developing a design.

Each UDP has exactly one output, which can be in one of these states: 0, 1, or x. The tri-state value z is not supported. Any input that has the value Z will be treated as X.

These two types of behavior can be represented in user-defined primitives:

1. Combinational UDP
2. Sequential UDP

A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of output.

Sequential UDPs provide an efficient and easy way to model sequential circuits such as *latches and flip-flops*.

A sequential UDP can model both level-sensitive and edge-sensitive behavior. The maximum number of inputs to a combinational UDP is 10. The maximum number of inputs to a sequential UDP is limited to 9 because the internal state counts as an input.

Syntax

UDP begins with the reserved word *primitive* and ends with *endprimitive*. Ports/terminals of primitive should follow. UDPs should be defined outside the *module* and *endmodule*.

- primitive UDP_name (output, input, ...);
- port_declaration
- [reg output;]
- [initial output = initial_value;]
- table
-   truth_table
- endtable

- endprimitive

UDP Rules

- UDP takes only scalar input terminals (1 bit).
- UDP can have only one scalar output. The output terminal must always appear first in the terminal list.
- The state in a sequential UDP is initialized with an initial statement.
- State table entries can contain values of 0, 1 or X. Z values passed to a UDP are treated as X values.
- UDPs are defined at the same level as modules.
- UDPs are instantiated exactly like gate primitives.
- UDPs do not support inout ports.

Verilog UDP Symbols

Verilog user-defined primitives can be written at the same level as module definitions, but never between *module* and *endmodule*. They can have many input ports but always one output port, and bi-directional ports are not valid. All port signals have to be scalar, which means they have to be 1-bit wide.

Hardware behavior is described as a primitive state table that lists out a different possible combination of inputs and their corresponding output within the *table* and *endtable*. Values of input and output signals are indicated using the following symbols.

Symbol	Comments
0	Logic 0
1	Logic 1
x	Unknown, can be either logic 0 or 1. It can be used as input/output or current state of sequential UDPs
?	Logic 0, 1 or x. It cannot be the output of any UDP
-	No change, only allowed in the output of a UDP
ab	Change in value from a to b where a or b is either 0, 1, or x

*	Same as ??, indicates any change in the input value
r	Same as 01 -> rising edge on input
f	Same as 10 -> falling edge on input
p	Potential positive edge on input; either 0->1, 0->x, or x->1
n	Potential falling edge on input; either 1->0, x->0, 1->x

Combinational UDP

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated, and one of the state table rows is matched. The output state is set to the value indicated by that row. The maximum number of inputs to a Combinational UDP is 10.

Consider the following example, which defines a multiplexer with two data inputs, a control input. But there can only be a single output.

```
// Output should always be the first signal in the port list

primitive mux (out, sel, a, b);
   output out;
   input sel, a, b;

   table
         //  sel   a   b          out
          0   1   ?   :   1;
         0   0   ?   :   0;
          1   ?   0   :   0;
         1   ?   1   :   1;
          x   0   0   :   0;
         x   1   1   :   1;
   endtable

endprimitive
```

A ? indicates that the signal can be either 0, 1 or x and does not matter in deciding the final output.

Example

Below is a testbench module that instantiates the UDP and applies input stimuli to it.

```
module tb;
  reg   sel, a, b;
  reg [2:0] dly;
  wire  out;
  integer i;

  // Instantiate the UDP
  // UDPs cannot be instantiated with port name connection
  mux u_mux ( out, sel, a, b);

  initial begin
    a <= 0;
    b <= 0;

    $monitor("[T=%0t] a=%0b b=%0b sel=%0b out=%0b", $time, a, b, sel, out);

 // Drive a, b, and sel after different random delays
    for (i = 0; i < 10; i = i + 1) begin
        dly = $random;
      #(dly) a <= $random;
        dly = $random;
      #(dly) b <= $random;
        dly = $random;
      #(dly) sel <= $random;
    end
  end
endmodule
```

Sequential UDP

Sequential UDP allows the mixing of the level-sensitive and edge-sensitive constructs in the same description. The output port should also be declared as *reg* type within the UDP definition and can be optionally initialized within an *initial* statement.

Sequential UDP takes the value of its inputs and the current value of its output to determine the next value of its output. The value of the output is also the internal state of the UDP.

Sequential UDPs have an additional field in between the input and output field, which is delimited by a ":" representing the current state.

Sequential UDP provides an easy and efficient way to model sequential circuits such as latches and flip-flops. The maximum number of inputs to a Sequential UDP is limited to 9 because the internal state counts as an input. There are two kinds of sequential UDPs.

1. Level-Sensitive UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be *reg* type, and there is an additional field in each table entry.

This new field represents the current state of the UDP. The output field in a sequential UDP represents the next state.

```
primitive d_latch (q, clk, d);
   output  q;
   input   clk, d;
   reg     q;

   table
               // clk     d       q   q+
         1   1   :   ? : 1;
          1   0   :   ? : 0;
         0   ?   :   ? : -;
   endtable

endprimitive
```

In the above code, a hyphen "-" on the last row of the table indicates no change in value for q+.

```
module tb;
  reg clk, d;
  reg [1:0] dly;
  wire q;
  integer i;

  d_latch u_latch (q, clk, d);

  always #10 clk = ~clk;

  initial begin
   clk = 0;

   $monitor ("[T=%0t] clk=%0b d=%0b q=%0b", $time, clk, d, q);

   #10;                              // To see the effect of X

   for (i = 0; i < 50; i = i+1) begin
     dly = $random;
    #(dly) d <= $random;
   end

   #20 $finish;
  end
endmodule
```

2. Edge-Sensitive UDPs

In level-sensitive behavior, the inputs and the current state's values are sufficient to determine the output value.

Edge sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs.

A D flip-flop is modeled as a Verilog user-defined primitive in the example shown below. Note that the rising edge of the clock is specified by 01 or 0?

```
primitive d_flop (q, clk, d);
   output  q;
   input   clk, d;
   reg     q;

   table
                          // clk        d        q       q+
         // obtain output on rising edge of clk
          (01)    0   :   ?   :   0;
         (01)    1   :   ?   :   1;
          (0?)    1   :   1   :   1;
         (0?)    0   :   0   :   0;

         // ignore negative edge of clk
```

```
      (?0)    ?   :   ?   :   -;

      // ignore data changes on steady clk
     ?      (??):   ?   :   -;
   endtable

endprimitive
```

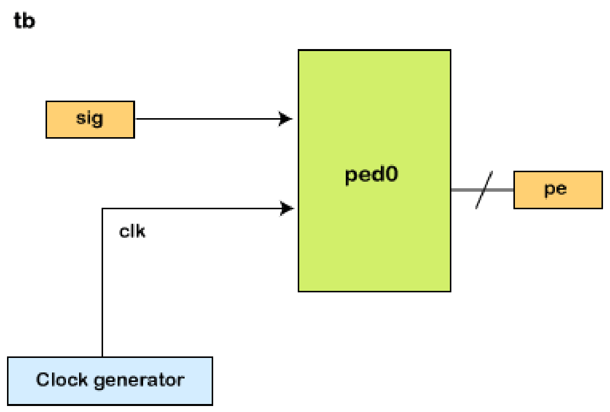
The UDP is instantiated and driven with random d input values in the testbench after a random number of clocks.

```
module tb;
  reg clk, d;
  reg [1:0] dly;
  wire q;
  integer i;

  d_flop u_flop (q, clk, d);

  always #10 clk = ~clk;

  initial begin
    clk = 0;

    $monitor ("[T=%0t] clk=%0b d=%0b q=%0b", $time, clk, d, q);

    #10;  // To see the effect of X

    for (i = 0; i < 20; i = i+1) begin
      dly = $random;
      repeat(dly) @(posedge clk);
      d <= $random;
    end

    #20 $finish;
  end
endmodule
```

The output q follows the input d after 1 clock delay, which is the D flip-flop's desired behavior.

Verilog Simulation Basics

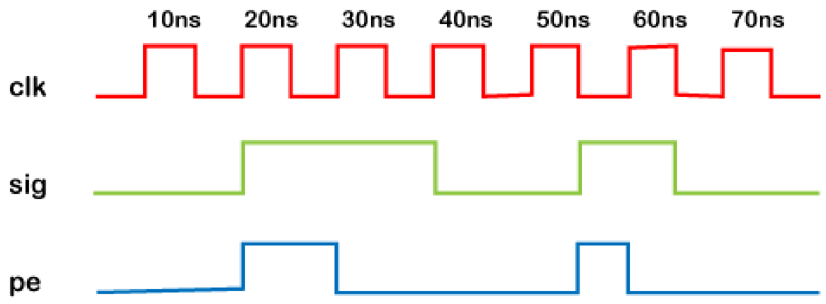
Verilog is a hardware description language, and there is no requirement for designers to simulate their RTL designs for converting them into logic gates.



Simulation is a technique for applying different input stimulus to the design at different times to check if the RTL code behaves in an intended way.

Simulation is a well-familiar technique to verify the robustness of the design. It is similar to how a fabricated chip will be used in the real world and how it reacts to different inputs.

For example, the above design represents a positive edge detector with inputs *clock* and *signal*, which are evaluated at periodic intervals to determine the output. Simulation allows us to view the timing diagram of related signals to understand how the design description behaves in Verilog.



Many EDA companies develop *simulators* capable of figuring out the outputs for various inputs to the design. Verilog is defined in terms of a *discrete event* execution model. And different simulators are free to use different algorithms to provide the user with a consistent set of results.

The Verilog code is divided into multiple processes and threads and may be evaluated at different times in a simulation.

Example

In this example, the *tb* (testbench) is a container to hold a design module. There are two *signals* or *variables* that can be assigned individual values at specific times. *clk* represents a clock which is generated within the testbench.

This is done using the *always* statement by alternating the clock's value after every 5ns. The *initial* block contains a set of statements that assign different values to both the signals.

```
module tb;
  reg clk;
  reg sig;

  // Clock generation
  // Process starts at time 0ns and loops after every 5ns
  always #5 clk = ~clk;

  // Process starts at time 0ns
  initial begin
    // This system task will print out the signal values everytime they change
    $monitor("Time = %0t clk = %0d sig = %0d", $time, clk, sig);

    // Also called stimulus, we simply assign different values to the variables
    // after some simulation "delay"
    sig = 0;
    #5 clk = 0;         // Assign clk to 0 at time 5ns
    #15  sig = 1;      // Assign sig to 1 at time 20ns (#5 + #15)
    #20  sig = 0;      // Assign sig to 0 at time 40ns (#5 + #15 + #20)
    #15  sig = 1;      // Assign sig to 1 at time 55ns (#5 + #15 + #20 + #15)
    #10  sig = 0;      // Assign sig to 0 at time 65ns (#5 + #15 + #20 + #15 + #10)
    #20 $finish;       // Finish simulation at time 85ns
  end
endmodule
```

The simulator provides the following output after execution of the above testbench.

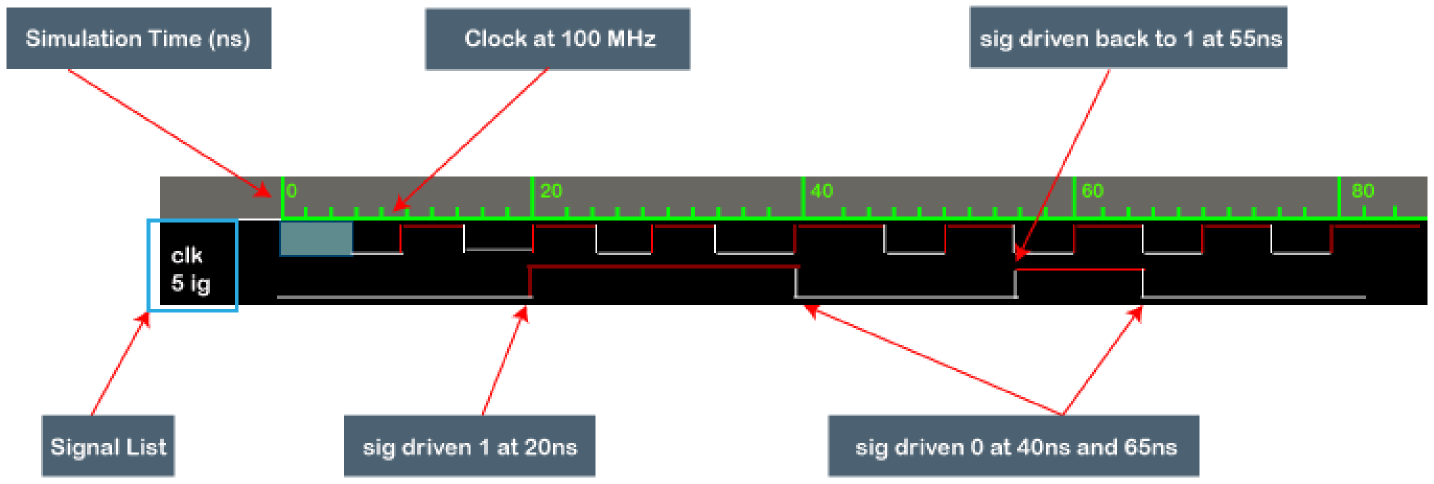
```
ncsim> run
Time = 0 clk = x sig = 0
Time = 5 clk = 0 sig = 0
Time = 10 clk = 1 sig = 0
Time = 15 clk = 0 sig = 0
```

- Time = 20 clk = 1 sig = 1
- Time = 25 clk = 0 sig = 1
- Time = 30 clk = 1 sig = 1
- Time = 35 clk = 0 sig = 1
- Time = 40 clk = 1 sig = 0
- Time = 45 clk = 0 sig = 0
- Time = 50 clk = 1 sig = 0
- Time = 55 clk = 0 sig = 1
- Time = 60 clk = 1 sig = 1
- Time = 65 clk = 0 sig = 0
- Time = 70 clk = 1 sig = 0
- Time = 75 clk = 0 sig = 0
- Time = 80 clk = 1 sig = 0
- Simulation complete via $finish(1) at time 85 NS + 0

Simulation Waveform

Simulations allow dumping design and testbench signals into a waveform that can be graphically represented to analyze and debug the RTL design functionality.

The waveform shown below is obtained from an EDA tool and shows each signal's progress for time and is the same as the timing diagram.

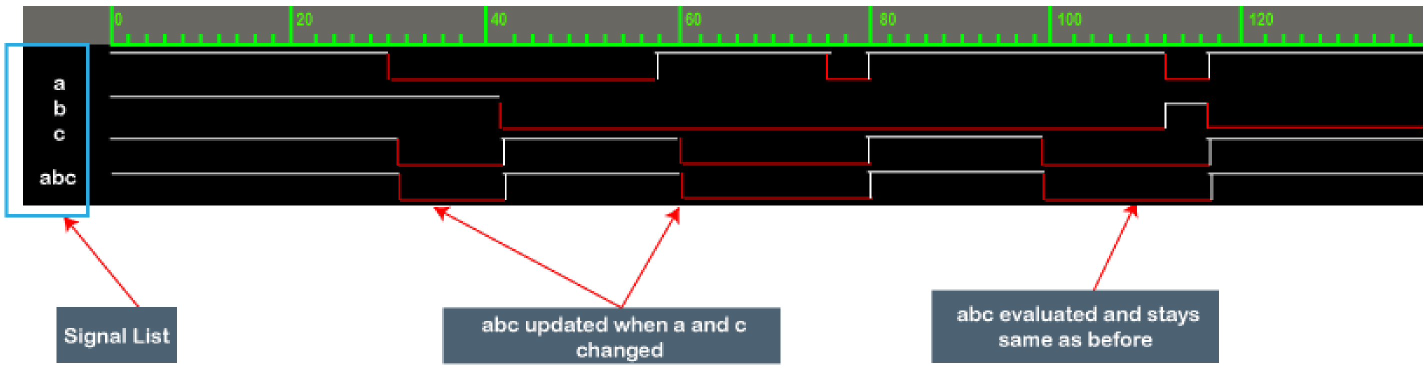


Every change in the value of a net or variable is called an *update event*. And processes are sensitive to *update events* such that these processes are evaluated whenever the update event happens and is called an *evaluation event*. Because of having the possibility of multiple processes being evaluated arbitrarily, the order of changes has to be tracked in something called an *event queue*.

They are ordered by the simulation time. Placement of a new event in the queue is called *scheduling*.

Simulation time refers to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated.

```
module des;
   wire abc;
    wire a, b, c;

    assign abc = a & b | c;          // abc is updated via the assign statement (process)

                                      //whenever a, b or c change -> update event

endmodule
```



Regions in an Event queue

The Verilog event queue is divided into five regions, and events can be added to any of them. However, it can be removed only from the active region.

Events	Description
Active	It occurs at the current simulation time and can be processed in any order
Inactive	It occurs at the current simulation time but is processed after all active events are done
Non-blocking	It evaluated at some previous time, but the assignment is done in the current simulation time after active and inactive events are done
Monitor	It processed after all the active, inactive and non-blocking events are done
Future	It occurs at some future simulation time

A simulation cycle is where all active events are processed. The standard guarantees a particular scheduling order except for a few cases and.

For example, statements inside a begin-end block will only be executed in the order in which they appear.

```
module tb;
   reg [3:0] a;
    reg [3:0] b;
   initial begin      // Statements are executed one after the other at appropriate simulation times
        a = 5;          // At time 0ns, a is assigned 5
       b = 2;  // In the same simulation step (time 0ns), b is assigned 2
        #10 a = 7;  // When simulation advances to 10ns, a is assigned 7    end

endmodule
```

The event queue defines that assignment to *b* should happen after assignment to *a*.

Verilog Timescale

Verilog simulation depends on how time is defined because the simulator needs to know what a #1 means in terms of time. The `timescale compiler directive specifies the time unit and precision for the modules that follow it.

Syntax

```
`timescale <time_unit>/<time_precision>
// for example
`timescale 1ns/1ps
`timescale 10us/100ns
`timescale 10ns/1ns
```

The *time_unit* is the measurement of delays and simulation time, while the *time_precision* specifies how delay values are rounded before being used in the simulation.

Use the following timescale constructs to use different time units in the same design. The delay specifications in the design are not synthesizable and cannot be converted to hardware logic

- *'timescale* for the base unit of measurement and precision of time.
- *$printtimescale* system task to display time unit and precision.
- *$time* and *$realtime* system functions return the current time, and the default reporting format can be changed with another system task *$timeformat*.

Character	Unit
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	Femtoseconds

The integers in these specifications can be either 1, 10 or 100 and the character string that specifies the unit can take any value mentioned in the table above.

Example 1: 1ns/1ns

```
// Declare the timescale where time_unit is 1ns
// and time_precision is also 1ns
`timescale 1ns/1ns

module tb;
  // To understand the effect of timescale, let us
  // drive a signal with some values after some delay
 reg val;

 initial begin
  // Initialize the signal to 0 at time 0 units
  val <= 0;

  // Advance by 1 time unit, display a message and toggle val
  #1     $display ("T=%0t At time #1", $realtime);
  val <= 1;

  // Advance by 0.49 time unit and toggle val
  #0.49  $display ("T=%0t At time #0.49", $realtime);
```

```
    val <= 0;

    // Advance by 0.50 time unit and toggle val
    #0.50   $display ("T=%0t At time #0.50", $realtime);
    val <= 1;

    // Advance by 0.51 time unit and toggle val
    #0.51   $display ("T=%0t At time #0.51", $realtime);
    val <= 0;

     // Let simulation run for another 5-time units and exit
    #5 $display ("T=%0t End of simulation", $realtime);
  end
endmodule
```

The first delay statement uses #1, making the simulator wait for the exactly 1-time unit, specified to be 1ns with a `timescale directive. The second delay statement uses 0.49, which is less than half a time unit.

However, the time precision is specified to be 1ns, and the simulator cannot go smaller than 1 ns, which makes it to round the given delay statement and yields 0ns. So the second delay fails to advance the simulation time.

The third delay statement uses exactly half the time unit [hl]#0.5[/lh], and again the simulator will round the value to get #1, which represents one whole time unit. So this gets printed at T=2ns.

The fourth delay statement uses a value more than half the time unit and gets rounded as well, making the display statement to be printed at T=3ns. After the execution, it gives the following output:

```
ncsim> run
T=1 At time #1
T=1 At time #0.49
T=2 At time #0.50
T=3 At time #0.51
T=8 End of simulation
ncsim: *W,RNQUIE: Simulation is complete.
```

The simulation runs for 8ns as expected, but notice that the waveform does not have smaller divisions between each nanosecond. This is because the precision of time is the same as the time unit.

Example 2: 10ns/1ns

The only change made in this example compared to the previous one is that the timescale has been changed from 1ns/1ns to 10ns/1ns. So the time unit is 10ns, and precision is at 1ns.

```
// Declare the timescale where time_unit is 10ns
// and time_precision is 1ns
`timescale 10ns/1ns

// Testbench is the same as in the previous example
module tb;
   // To understand the effect of timescale, let us
   // drive a signal with some values after some delay
  reg val;

  initial begin
   // Initialize the signal to 0 at time 0 units
   val <= 0;

   // Advance by 1 time unit, display a message and toggle val
   #1     $display ("T=%0t At time #1", $realtime);
   val <= 1;

   // Advance by 0.49 time unit and toggle val
   #0.49   $display ("T=%0t At time #0.49", $realtime);
   val <= 0;

   // Advance by 0.50 time unit and toggle val
   #0.50   $display ("T=%0t At time #0.50", $realtime);
   val <= 1;

   // Advance by 0.51 time unit and toggle val
   #0.51   $display ("T=%0t At time #0.51", $realtime);
   val <= 0;

    // Let simulation run for another 5-time units and exit
   #5 $display ("T=%0t End of simulation", $realtime);
  end
endmodule
```

Actual simulation time is obtained by multiplying the delay specified using # with the time unit, and then it is rounded off based on precision. The first delay statement will then yield 10ns, and the second one gives 14.9, which gets rounded to become 15ns.

The third statement similarly adds 5ns (0.5 * 10ns), and the total time becomes 20ns. The fourth one adds another 5ns (0.51 * 10) to advance the total time to 25ns.

```
ncsim> run
T=10 At time #1
T=15 At time #0.49
T=20 At time #0.50
T=25 At time #0.51
T=75 End of simulation
ncsim: *W,RNQUIE: Simulation is complete.
```

NOTE: The base unit in the waveform is in tens of nanoseconds with a precision of 1ns.

Example 3: 1ns/1ps

The only change made in this example compared to the previous one is that the timescale has been changed from 1ns/1ns to 1ns/1ps. So the time unit is 1ns, and precision is at 1ps.

```
// Declare the timescale where time_unit is 1ns
// and time_precision is 1ps
`timescale 1ns/1ps

// Testbench is the same as in the previous example
module tb;
   // To understand the effect of timescale, let us
   // drive a signal with some values after some delay
  reg val;

  initial begin
   // Initialize the signal to 0 at time 0 units
    val <= 0;

    // Advance by 1 time unit, display a message and toggle val
   #1      $display ("T=%0t At time #1", $realtime);
    val <= 1;

    // Advance by 0.49 time unit and toggle val
   #0.49   $display ("T=%0t At time #0.49", $realtime);
```

```
    val <= 0;

    // Advance by 0.50 time unit and toggle val
   #0.50   $display ("T=%0t At time #0.50", $realtime);
    val <= 1;

    // Advance by 0.51 time unit and toggle val
   #0.51   $display ("T=%0t At time #0.51", $realtime);
    val <= 0;

     // Let simulation run for another 5-time units and exit
   #5 $display ("T=%0t End of simulation", $realtime);
  end
endmodule
```

See that the time units scaled to match the new precision value of 1ps. And time is represented in the smallest resolution, which in this case is picoseconds.

```
ncsim> run
T=1000 At time #1
T=1490 At time #0.49
T=1990 At time #0.50
T=2500 At time #0.51
T=7500 End of simulation
ncsim: *W,RNQUIE: Simulation is complete.
```

Default Timescale

Although Verilog modules are expected to have a timescale defined before the module, simulators may insert a default timescale.

The actual timescale that gets applied at any scope in a Verilog elaborated hierarchy can be printed using the system task $printtimescale, which accepts the scope as an argument.

```
module tb;
   initial begin
       // Print timescale of this module
       $printtimescale(tb);
       // $printtimescale($root);
   end
endmodule
```

NOTE: A timescale directive was not placed before this module. The simulator ended up applying a 1ns/1ns timescale value.

```
xcelium> run
Time scale of (tb) is  1ns /  1ns
xmsim: *W,RNQUIE: Simulation is complete.
```

Standard Timescale Scope

By default, a timescale directive placed in a file is applied to all modules that follow the directive until the definition of another timescale directive.

```
`timescale 1ns/1ps

module tb;
  des m_des();
  alu m_alu();

  initial begin
    $printtimescale(tb);
    $printtimescale(tb.m_alu);
    $printtimescale(tb.m_des);
  end
endmodule

module alu;

endmodule

`timescale 1ns/10ps

module des;

endmodule
```

In the above example, *tb* and *alu* end up with a timescale of 1ns/1ns while *des* get a timescale of 1ns/10ps because of the directive placement before the module definition of *des*.

```
xcelium> run
Time scale of (tb) is  1ns /  1ps
Time scale of (tb.m_alu) is  1ns /  1ps
Time scale of (tb.m_des) is  1ns /  10ps
```

```
xmsim: *W,RNQUIE: Simulation is complete.
```

Verilog Files

Other files can be included in the current file using an `include directive, which is a pre-processor directive and makes the compiler place contents of the included file before compilation.

This is equivalent to simply pasting the entire contents of the other file in this main file.

```
// main.v
`timescale 1ns/1ps

module tb;
  des m_des();
  alu m_alu();

  initial begin
    $printtimescale(tb);
    $printtimescale(tb.m_alu);
    $printtimescale(tb.m_des);
  end
endmodule

`include "file_alu.v"
`include "file_des.v"

// file_alu.v
module alu;
endmodule

// file_des.v
`timescale 1ns/10ps

module des;

endmodule
```

See that results are precisely the same as in the previous example. *alu* gets a timescale of 1ns/1ps because it was the last directive that stayed valid until the compiler found *alu* definition insisted of placing it in a different file.

The *des* gets a timescale of 1ns/10ps because the directive was replaced before its definition.

```
xcelium> run
Time scale of (tb) is  1ns /  1ps
Time scale of (tb.m_alu) is  1ns /  1ps
Time scale of (tb.m_des) is  1ns /  10ps
xmsim: *W,RNQUIE: Simulation is complete.
```

Swapping Files can Change Timescale.

The order of inclusion of files plays an important role in the redefinition of timescale directives, which is evident in the example below.

```
// main.v
`timescale 1ns/1ps

module tb;
  des m_des();
  alu m_alu();

  initial begin
    $printtimescale(tb);
    $printtimescale(tb.m_alu);
    $printtimescale(tb.m_des);
  end
endmodule

// Swapped order of inclusion
`include "file_des.v"
`include "file_alu.v"

// file_alu.v
module alu;
endmodule

// file_des.v
`timescale 1ns/10ps

module des;

endmodule
```

See that the module *alu* now gets a timescale of 1ns/10ps.

- xcelium> run
- Time scale of (tb) is 1ns / 1ps
- Time scale of (tb.m_alu) is 1ns / 10ps
- Time scale of (tb.m_des) is 1ns / 10ps
- xmsim: *W,RNQUIE: Simulation is complete.

This is one reason for having a timescale directive at the top of files so that all modules in that file assume the correct timescale irrespective of file inclusion.

However, this approach may make it difficult to compile with a different timescale precision without altering each file.

Many compilers and simulators also provide an option to override default timescale values, which will be applied to all modules.

Verilog Timeformat

Verilog timescale directive specifies time unit and precision for simulations.

Verilog *$timeformat* system function specifies *%t* format specifier reporting style in display statements such as *$display* and *$strobe*.

Syntax

1. $timeformat(<unit_number>, <precision>, <suffix_string>, <minimum field width>);

- *unit_number* is the smallest time precision out of all `timescale directives used in the source code.
- *precision* represents the number of fractional digits for the current timescale.
- *suffix_string* is an option to display the scale alongside real time values.

Unit Number	Time Unit
-3	1ms
-6	1us
-9	1ns
-12	1ps

-15	1fs

Example 1: 1ns/1ps

Below is an example of how *$timeformat* affects the format of the time unit display.

```
`timescale 1ns/1ps

module tb;
  bit a;

  initial begin

    // Wait for some time
    // precision is 1/1000 of
 the main scale (1ns), the 3rd position will truncate
 this delay
    #10.512351;

    // Display current time with default timeformat parameters
    $display("[T=%0t] a=%0b", $realtime, a);

    // Change timeformat parameters and display again
    $timeformat(-9, 2, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Remove the space in suffix, and extend fractional digits to 5
    $timeformat(-9, 5, "ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Here suffix is wrong, it should not be "ns" because we are
    // setting display in "ps" (-12)
    $timeformat(-12, 3, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Correct the suffix to ps
    $timeformat(-12, 2, " ps");
    $display("[T=%0t] a=%0b", $realtime, a);
  end
endmodule
```

Now executes the above code, and it will give the following output, such as

```
xcelium> run
[T=10512] a=0
[T=10.51 ns] a=0
[T=10.51200ns] a=0
[T=10512.000 ns] a=0
[T=10512.00 ps] a=0
xmsim: *W,RNQUIE: Simulation is complete.
```

Example 2: 1ns/100ps

We consider the same example from above with a different timescale.

```
`timescale 1ns/100ps

module tb;
  bit a;

  initial begin

    // Wait for some time
    // precision is 1/1000 of
 the main scale (1ns), the 3rd position will truncate
 this delay
    #10.512351;

    // Display current time with default timeformat parameters
    $display("[T=%0t] a=%0b", $realtime, a);

    // Change timeformat parameters and display again
    $timeformat(-9, 2, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Remove the space in suffix, and extend fractional digits to 5
    $timeformat(-9, 5, "ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Here suffix is wrong, it should not be "ns" because we are
    // setting display in "ps" (-12)
    $timeformat(-12, 3, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Correct the suffix to ps
    $timeformat(-12, 2, " ps");
    $display("[T=%0t] a=%0b", $realtime, a);
  end
endmodule
```

Above code produce the following output:

```
xcelium> run
[T=105] a=0
[T=10.50 ns] a=0
[T=10.50000ns] a=0
[T=10500.000 ns] a=0
[T=10500.00 ps] a=0
xmsim: *W,RNQUIE: Simulation is complete.
```

Example 3: 100ns/1ns

```
`timescale 100ns/1ns

module tb;
  bit a;

  initial begin

    // Wait for some time - note that because precision is 1/1000 of
    // the main scale (1ns), this delay will be truncated by the 3rd
    // position
    #10.512351;

    // Display current time with default timeformat parameters
    $display("[T=%0t] a=%0b", $realtime, a);

    // Change timeformat parameters and display again
    $timeformat(-9, 2, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Remove the space in suffix, and extend fractional digits to 5
    $timeformat(-9, 5, "ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Here suffix is wrong, it should not be "ns" because we are
    // setting display in "ps" (-12)
    $timeformat(-12, 3, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);

    // Correct the suffix to ps
    $timeformat(-12, 2, " ps");
    $display("[T=%0t] a=%0b", $realtime, a);
  end
endmodule
```

NOTE: #1 represents 100ns and hence #10 yields 1000ns.

And the output looks like:

```
xcelium> run
[T=1051] a=0
[T=1051.00 ns] a=0
[T=1051.00000ns] a=0
[T=1051000.000 ns] a=0
[T=1051000.00 ps] a=0
xmsim: *W,RNQUIE: Simulation is complete.
```

Verilog Scheduling Semantics

Verilog design and testbench typically have many code lines comprising of *always* or *initial* blocks, continuous assignments, and other procedural statements that become active at different times in the course of a simulation.

Every change in the value of a signal in the Verilog model is considered an *update event*. And processes such as always and assign blocks sensitive to these update events are evaluated in an arbitrary order and called an *evaluation* **event**.

Since these events can happen at different times, they are better managed and ensured their correct order of execution by scheduling them into *event queues* arranged by simulation time.

```
module tb;
    reg a, b, c;
    wire d;

    // 'always' is a process that gets evaluated when either 'a' or 'b' is updated.
    // When 'a' or 'b' changes in value it is called an 'update event'.
    // When 'always' block is triggered because of a change in 'a' or 'b' it is called an evaluati
on event.
```

```

   always @ (a or b) begin
     c = a & b;
   end

 // Here 'assign' is a process that is evaluated when either 'a' or 'b' or 'c' gets updated.

         assign d = a | b ^ c;
endmodule
```

Event Queue

A simulation step can be segmented into four different regions. An active event queue is just a set of processes that need to execute at the current time, resulting in more processes to be scheduled into active or other event queues. Events can be added to any of the regions, but always removed from the *active* region.

- *Active* events occur at the current simulation time and can be processed in any order.
- *Inactive* events occur at the current simulation time but are processed after all active events are processed.
- *Non-blocking assign* events that were evaluated previously will be assigned after all active and inactive events are processed.
- *Monitor* events are processed after all active, inactive and non-blocking assignments are done.

When all events in the active queue for the current time step has been executed. The simulator advances time to the next time step and executes its active queue.

```
module tb;
   reg x, y, z

   initial begin
      #1  x = 1;
          y = 1;
      #1  z = 0;
   end
endmodule
```

Simulation starts at time 0, and the first statement is scheduled to be executed when simulation time reaches 1 time unit at which it assigns x and y to 1.

This is the active queue for the current time, which is a 1-time unit. The simulator then schedules the next statement after 1 more time unit at which z is assigned 0.

Non-deterministic

During the simulation, there can be race conditions that end up giving different outputs for the same design and testbench. One of the reasons for non-deterministic behavior is because *active* events can be removed from the queue and processed in any order.

When multiple processes are triggered simultaneously, then the order in which the processes are executed is not specified by the Institute of Electrical and Electronics Engineers (*IEEE*) standards. It is arbitrary, and it varies from the simulator to the simulator. This is called the *non-determinism*.

There are two common cases of non-determinism that are caused by the same root cause but that manifest in different ways.

Case 1: When multiple statements execute in zero time, then the order they execute affects the results. Therefore, a different order of execution gives different but correct results. Execution in zero time means that the statements are evaluated without advancing simulation time.

```
always @(d)
  q = d;
assign q = ~d;
```

These two processes, one procedural block and one continuous assignment are scheduled to execute at the same time when variable d changes.

If the always block is evaluated first, variable q is assigned the new value of d by the always block. Then the continuous assignment is executed. It assigns the complement of the new value of d to variable q.

If the continuous assignment is evaluated first, q gets the complement of the new value of d. Then the procedural assignment assigns the new value (non-complemented) to q. Therefore, the two orders produce the opposite results.

Case 2: This case is considered when the interleaving procedural statements in blocks are executed simultaneously. When two procedural blocks are scheduled simultaneously, there is no guarantee that all statements in a block finish before the statements in the other block begin. The statements from the two blocks can execute in an interleaving order.

```
always @(posedge clk) // always block1
begin
  x = 1'b0;
  y = x;
end

always @(posedge clk) // always block2
begin
```

- x = 1'b1;
- end

Both always blocks are triggered when a positive edge of the clk arrives. One interleaving order is

- x = 1'b0;
- y = x;
- x = 1'b1;

In this case, y gets 0. And another interleaving order is

- x = 1'b0;
- x = 1'b1;
- y = x;

In this case, y gets 1.

Verilog Display Tasks

Display system tasks are mainly used to display informational and debug messages to track the simulation flow from log files. There are different groups of display tasks and formats in which they can print values.

Generally, display system tasks are grouped into three categories, such as:

1. Display and Write tasks
2. Verilog strobe
3. Continuous monitoring tasks

When one of these tasks is invoked, it simply prints its arguments. The order of printed arguments is the same as the order that the x appears in the argument list. If no argument is specified, it can be declared a null argument, and when the display task is invoked, it simply prints a single space character. An argument can be an expression that returns a value and a quoted string.

Display and Write Tasks

The first group of displaying tasks is very similar to print the function in the ANSI C language. The $write and the $display tasks work in the same way, and the only difference is that the $display task adds a new line character at the end of the output, while the $write task does not.

Syntax

Both *$display* and *$write* display arguments in the order they appear in the argument list.

- $display(<list_of_arguments>);
- $write(<list_of_arguments>);

$write does not append the newline character to the end of its string, while *$display* can be seen from the example shown below.

Example

```
module tb;
  initial begin
    $display ("This ends with a new line ");
    $write ("This does not,");
    $write ("like this. To start new line, use newline char");
    $display ("This always starts on a new line!");
  end
endmodule
```

Now, executes the above code, and we will get the following output.

```
ncsim> run
This ends with a new line
This does not, like this. To start a new line, use newline char
Hello!
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Strobes

$strobe prints the final values of variables at the end of the current delta time-step and has a similar format like *$display*. A newline is automatically added to the text.

Example

```
module tb;
  initial begin
    reg [7:0] a;
    reg [7:0] b;

    a = 8'h2D;
    b = 8'h2D;

    #10;                  // Wait till simulation reaches 10ns
    b <= a + 1;           // Assign a+1 value to b

    $display ("[$display] time=%0t a=0x%0h b=0x%0h", $time, a, b);
    $strobe  ("[$strobe]  time=%0t a=0x%0h b=0x%0h", $time, a, b);

    #1;
    $display ("[$display] time=%0t a=0x%0h b=0x%0h", $time, a, b);
    $strobe  ("[$strobe]  time=%0t a=0x%0h b=0x%0h", $time, a, b);

  end
endmodule
```

NOTE: $strobe shows the final updated value of the variable b at time 10ns, 0x2E, and $display picks that up only in the next simulation delta at 11ns.

And the output looks like:

```
ncsim> run
[$display] time=10 a=0x2d b=0x2d
[$strobe]  time=10 a=0x2d b=0x2e
[$display] time=11 a=0x2d b=0x2e
[$strobe]  time=11 a=0x2d b=0x2e
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Verilog Continuous Monitors

$monitor helps to automatically print out variable or expression values whenever the variable or expression in its argument list changes.

It achieves a similar effect of calling $display after every time any of its arguments get updated. A newline is automatically added to the text.

Example

```
module tb;
initial begin
    reg [7:0] a;
    reg [7:0] b;

    a = 8'h2D;
    b = 8'h2D;

    #10;                  // Wait till simulation reaches 10ns
    b <= a + 1;           // Assign a+1 value to b

    $monitor ("[$monitor] time=%0t a=0x%0h b=0x%0h", $time, a, b);

    #1 b <= 8'hA4;
    #5 b <= a - 8'h33;
    #10 b <= 8'h1;

  end
endmodule
```

$monitor is like a task that is spawned to run in the background of the main thread, which monitors and displays value changes of its argument variables. A new $monitor task can be issued any number of times during the simulation.

```
ncsim> run
[$monitor] time=10 a=0x2d b=0x2e
[$monitor] time=11 a=0x2d b=0xa4
[$monitor] time=16 a=0x2d b=0xfa
[$monitor] time=26 a=0x2d b=0x1
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Format Specifiers

To print variables inside display functions, appropriate *format specifiers* have to be given for each variable.

These tasks have a special character (%) to indicate that the information about signal value is needed. When using a string, the compiler recognizes the % character and knows that the next character is a format specification.

If the format specification sign is used, a corresponding argument should always be followed (exception is the %m argument).

Argument	**Description**
%h, %H	Display in hexadecimal format
%d, %D	Display in decimal format
%b, %B	Display in binary format
%o or %O	Display octal format
%m, %M	Display hierarchical name

%s, %S	Display as a string
%t, %T	Display in time format
%f, %F	Display 'real' in a decimal format
%e, %E	Display 'real' in an exponential format

These system tasks can be invoked with "o", "h" and "b" extensions. For example *$writeb, $writeo*, and *$displayh*. When invoked, they inform the simulator that there are some arguments without corresponding format specifications, and the default display format should be changed. By default, $display and $write system tasks use the decimal format to change display formats.

The size of the displayed data is essential. Generally, it depends on the format specification. If we are using a hexadecimal format, the data will be displayed as four characters, each of them representing four bits of the value (a single hexadecimal value can be represented as four bits).

Similarly, octal values will be displayed as a group of characters representing three bits. The result of an expression is automatically sized. However, we can change default settings by adding 0 (zero) after the % character.

Another very useful display task feature is ruled applying to the result of an expression that has an unknown or high impedance value. If we are using the decimal format (%d), then we follow the following rules:

- Single lowercase "x" character will be displayed when all bits are of an unknown value.
- Single uppercase "X" character will be displayed when some bits are of an unknown value.
- Single lowercase "z" character will be displayed when all bits are of a high impedance value.
- A single uppercase "Z" character will be displayed when some bits are of high impedance.

And if we are using hexadecimal (%h) and octal (%o) formats, we follow the following rules:

- Single lowercase "x" will be displayed when all bits in the group are of an unknown value.
- Single uppercase "X" will be displayed when some group bits are of an unknown value.
- Single lowercase "z" will be displayed when all bits in the group are of a high impedance value.
- Single uppercase "Z" will be displayed when some bits in the group are of high impedance.

NOTE: In the octal format, a group represents three bits that can be represented as one digit within the range is 0 to 7. In the hexadecimal format, four bits can be represented as one character within the range 0 to 9 and characters in range a to f.

Example

```
module tb;
  initial begin
    reg [7:0]  a;
    reg [39:0] str = "Hello";
    time       cur_time;
    real       float_pt;

    a = 8'h0E;
    float_pt = 3.142;

    $display ("a = %h", a);
    $display ("a = %d", a);
    $display ("a = %b", a);
    $display ("str = %s", str);
    #200 cur_time = $time;
    $display ("time = %t", cur_time);
    $display ("float_pt = %f", float_pt);
    $display ("float_pt = %e", float_pt);
  end
endmodule
```

Above code gives the following output after the execution, such as:

```
ncsim> run
a = 0e
a =  14
a = 00001110
str = Hello
time =                   200
float_pt = 3.142000
float_pt = 3.142000e+00
ncsim: *W,RNQUIE: Simulation is complete.
```

Verilog Escape Sequences

Some characters are considered unique since they stand for other display purposes such as new-line, tabs, and form feeds.

To print these special characters, each occurrence of such characters has to be *escaped*.

Argument	**Description**
\n	Newline character

\t	Tab character
\\	Backslash
\ddd	Octal code
%%	Percent sign

Example

```
module tb;
  initial begin
    $write ("Newline character
");
    $display ("Tab character    stop");
    $display ("Escaping  " %%");

/*
    // Compilation errors
    $display ("Without escaping ");       // ERROR : Unterminated string
    $display ("Without escaping "");       // ERROR : Unterminated string
*/
  end
endmodule
```

And the output looks like:

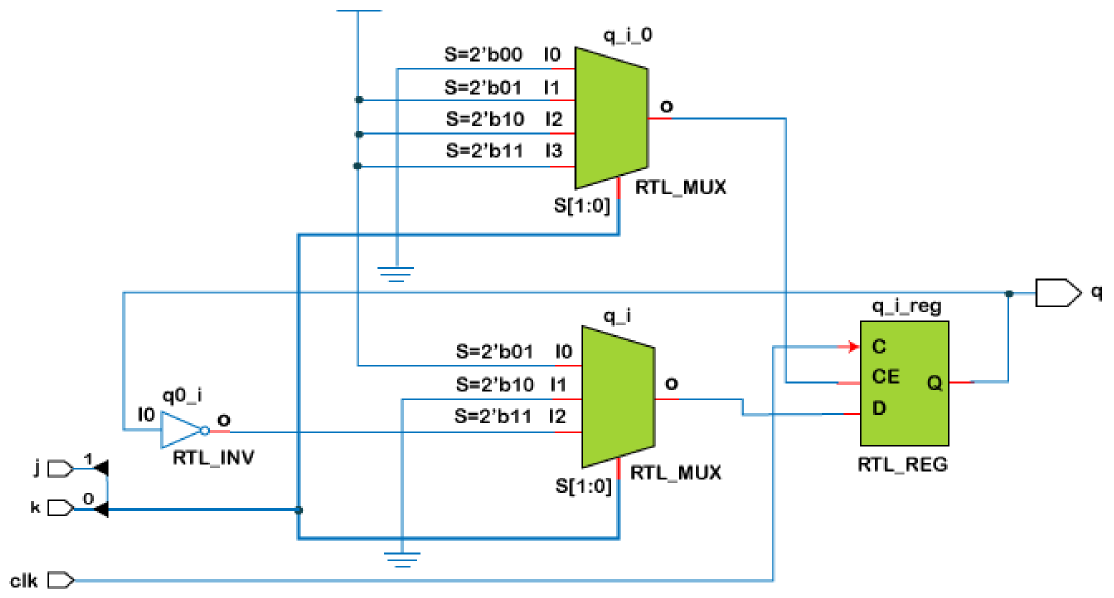
```
ncsim> run
Newline character

Tab character         stop
Escaping  " %
ncsim: *W,RNQUIE: Simulation is complete.
```

JK Flip Flop

The JK flip-flop is the most versatile of the basic flip flops. A JK flip-flop is used in clocked *sequential* logic circuits to store one bit of data.

It is almost identical in function to an SR flip flop. The only difference is eliminating the undefined state where both S and R are 1. Due to this additional clocked input, a JK flip-flop has four possible input combinations, such as "logic 1", "logic 0", "no change" and "toggle".

Hardware Schematic



Example

We will program JK Flip Flop in Verilog and write a testbench for the same code.

```
module jk_ff ( input j, input k, input clk, output q);

  reg q;

  always @ (posedge clk)
    case ({j,k})
       2'b00 :  q <= q;
       2'b01 :  q <= 0;
       2'b10 :  q <= 1;
       2'b11 :  q <= ~q;
     endcase
endmodule
```

Testbench

```
module tb_jk;
  reg j;
  reg k;
  reg clk;

  always #5 clk = ~clk;
```

```

jk_ff   jk0 ( .j(j),
                 .k(k),
               .clk(clk),
                 .q(q));

initial begin
   j <= 0;
    k <= 0;

   #5 j <= 0;
      k <= 1;
   #20 j <= 1;
       k <= 0;
   #20 j <= 1;
       k <= 1;
   #20 $finish;
end

initial
    $monitor ("j=%0d k=%0d q=%0d", j, k, q);
endmodule
```

Edge-triggered JK Flip-flop

The type of JK flip-flop described here is an edge-triggered JK flip-flop. It is built from two gated latches: one a master gated D latch and a slave gated SR latch.

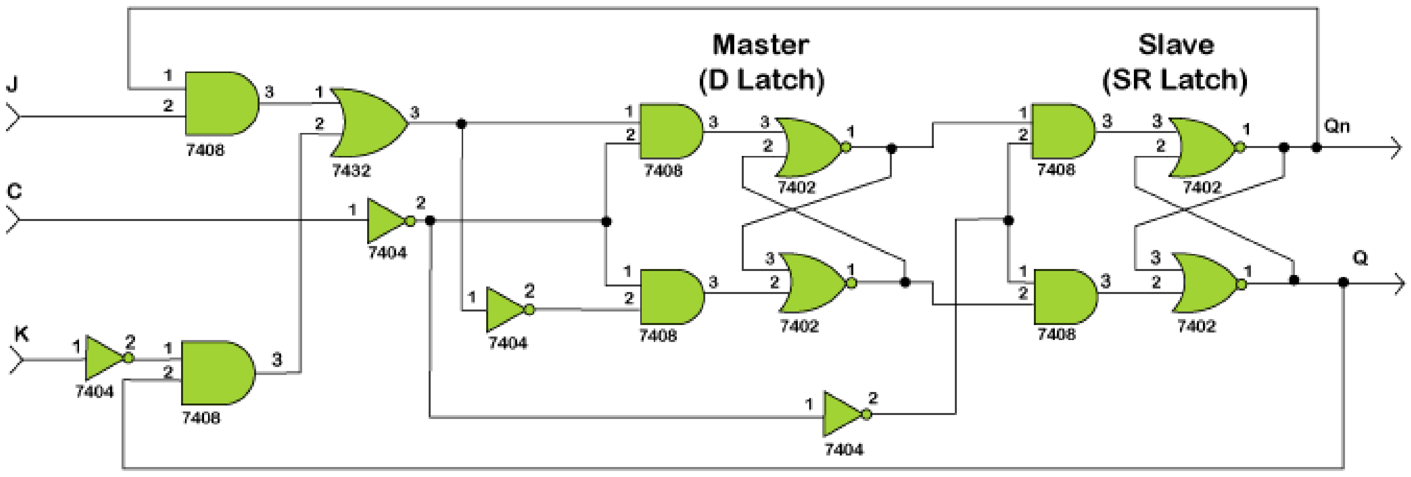
This is a modified version of the *edge-triggered D flip flop*. The flip-flop's outputs are fed back and combined with the inputs. The master takes the flip-flop's inputs, such as J (set), K (reset), and C (clock).

The clock input is inverted and fed into the D latch's gate input. The slave takes the master's outputs as inputs (Q to S and Qn to R) and complements the master's clock input. The slave's outputs are the flip-flop's outputs. This difference in clock inputs between the two latches disconnects them and eliminates the transparency between the flip-flop's inputs and outputs.

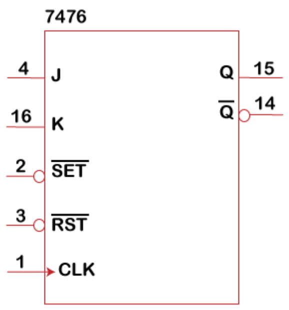
The below schematic image shows a positive edge-triggered JK flip-flop. The two inputs J and K, are used to set and reset the data, respectively. They can also be used to toggle the data. The clock input C is used to control both the master and slave latches, making sure only one of the latches can set its data at any given time.

When C has the value 0, the master latch can set its data, and the slave latch cannot. When C has the value 1, the slave can set its data, and the master cannot. When C transitions from 0 to 1, the master has its outputs set, which reflect the flip-flop's inputs when the transition occurred.

The outputs Q and Qn are the flip-flop's stored data and the complement of the flip-flop's stored data.



The schematic symbol for a 7476 edge-triggered JK flip-flop is shown below. This chip has inputs to set and reset the flip-flop's data asynchronously.



Example

Below is the Verilog code for a positive edge-triggered JK flip-flop. An active-low reset input has been added to asynchronously clear the flip-flop.

```
module jk_ff_edge_triggered(Q, Qn, C, J, K, RESETn);
  output Q;
  output Qn;
  input  C;
  input  J;
  input  K;
  input  RESETn;

  wire   Kn;                    // The complement of the K input.
```

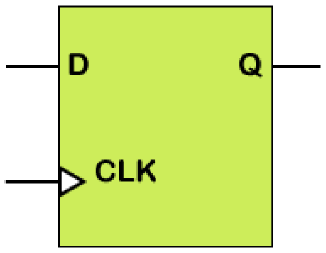
```
    wire   D;
    wire   D1;                    // Data input to the D latch.
    wire   Cn;                    // Control input to the D latch.
    wire   Cnn;                   // Control input to the SR latch.
    wire   DQ;                    // Output from the D latch, inputs to the gated SR latch (S).
    wire   DQn;                   // Output from the D latch, inputs to the gated SR latch (R).

    assign D1 = !RESETn ? 0 : D;                // Upon reset force D1 = 0

    not(Kn, K);
    and(J1, J, Qn);
    and(K1, Kn, Q);
    or(D, J1, K1);
    not(Cn, C);
    not(Cnn, Cn);
    d_latch dl(DQ, DQn, Cn, D1);
    sr_latch_gated sr(Q, Qn, Cnn, DQ, DQn);
  endmodule                                // jk_flip_flop_edge_triggered

  module d_latch(Q, Qn, G, D);
    output Q;
    output Qn;
    input  G;
    input  D;

    wire   Dn;
    wire   D1;
    wire   Dn1;

    not(Dn, D);
    and(D1, G, D);
    and(Dn1, G, Dn);
    nor(Qn, D1, Q);
    nor(Q, Dn1, Qn);
  endmodule                                // d_latch

  module sr_latch_gated(Q, Qn, G, S, R);
    output Q;
```

```
output Qn;
input  G;
input  S;
input  R;

wire   S1;
wire   R1;

and(S1, G, S);
and(R1, G, R);
nor(Qn, S1, Q);
nor(Q, R1, Qn);
endmodule                    // sr_latch_gated
```

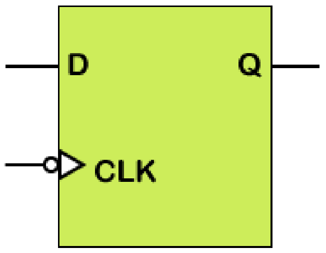
D Flip-Flop

A *D* flip-flop is a sequential element that follows the input pin *d* at the clock's given edge. D flip-flop is a fundamental component in digital logic circuits.

There are two types of D Flip-Flops being implemented: Rising-Edge D Flip Flop and Falling-Edge D Flip Flop.



Rising-Edge D Flip-Flop



Falling-Edge D Flip-Flop

D flip flop is an edge-triggered memory device that transfers a signal's value on its D input to its Q output when an active edge transition occurs on its clock input. Then, the output value is held until the next active clock cycle.

Flip flops are inferred using the edge triggered *always* statements. The *always* statement is edge-triggered by including either a *posedge* or *negedge* clause in the event list. Here are some examples of sequential *always* statements, such as:

- always @(posedge Clock)
- always @(negedge Clock)
- always @(posedge Clock or posedge Reset)
- always @(posedge Clock or negedge Reset)
- always @(negedge Clock or posedge Reset)

- always @(negedge Clock or negedge Reset)

If an asynchronously reset flip flop is being modeled, a second *posedge* or *negedge* clause is needed in the event list of the *always* statement. Also, most synthesis tools require that the reset must be used in *if* statement directly following the always statement, or after *begin* if it is in a sequential *begin-end* block.

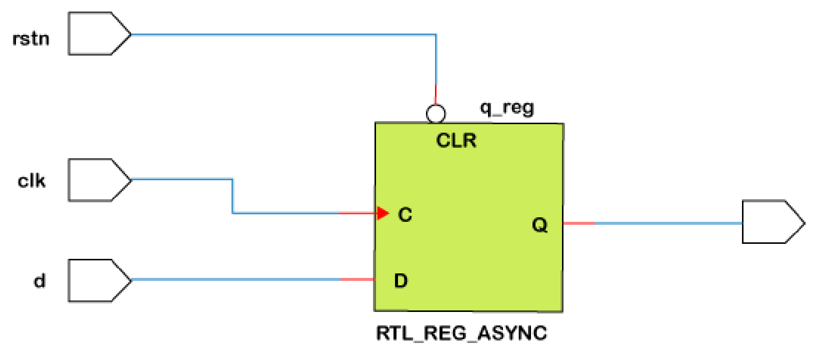
Example

```
//Active low asynchronous reset
    always @(posedge Clock or negedge Reset)
    begin
       if (!Reset)
        …….
       ……..
end
```

Design 1: With the async active-low reset

```
module dff (input d,
          input rstn,
           input clk,
          output reg q);

   always @ (posedge clk or negedge rstn)
      if (!rstn)
        q <= 0;
      else
        q <= d;
endmodule
```

Hardware Schematic



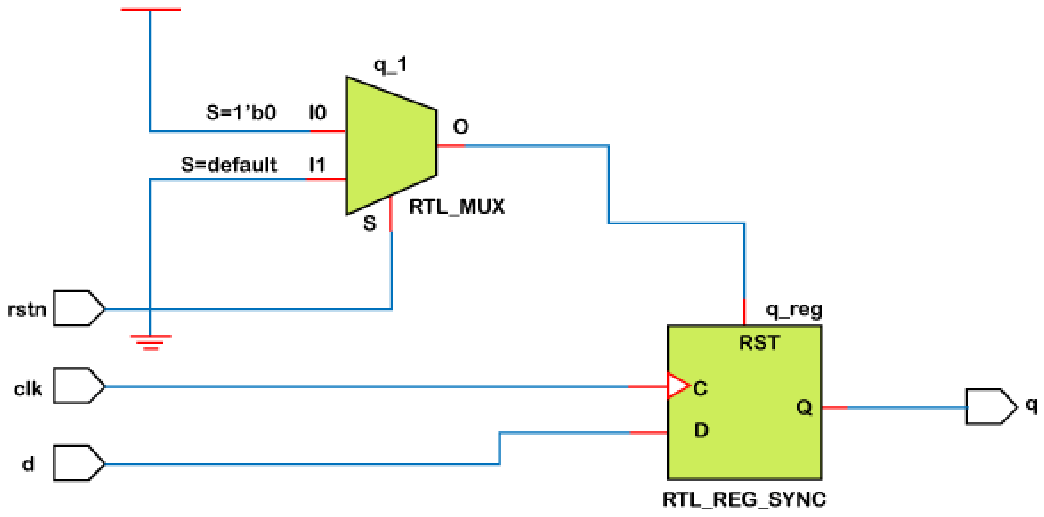
Testbench

```
module tb_dff;
   reg clk;
   reg d;
   reg rstn;
   reg [2:0] delay;

   dff  dff0 ( .d(d),
              .rsnt (rstn),
               .clk (clk),
              .q (q));

   // Generate clock
   always #10 clk = ~clk;

   // Testcase
   initial begin
      clk <= 0;
      d <= 0;
      rstn <= 0;

      #15 d <= 1;
      #10 rstn <= 1;
      for (int i = 0; i < 5; i=i+1) begin
         delay = $random;
          #(delay) d <= i;
      end
   end
endmodule
```

Design 2: With sync active-low reset

```
module dff (input d,
            input rstn,
             input clk,
            output reg q);

    always @ (posedge clk)
       if (!rstn)
         q <= 0;
       else
         q <= d;
endmodule
```

Hardware Schematic



Testbench

```
module tb_dff;
    reg clk;
    reg d;
    reg rstn;
    reg [2:0] delay;

    dff  dff0 ( .d(d),
              .rsnt (rstn),
               .clk (clk),
              .q (q));

    // Generate clock
    always #10 clk = ~clk;

    // Testcase
    initial begin
        clk <= 0;
        d <= 0;
        rstn <= 0;

        #15 d <= 1;
        #10 rstn <= 1;
        for (int i = 0; i < 5; i=i+1) begin
            delay = $random;
            #(delay) d <= i;
        end
    end
endmodule
```

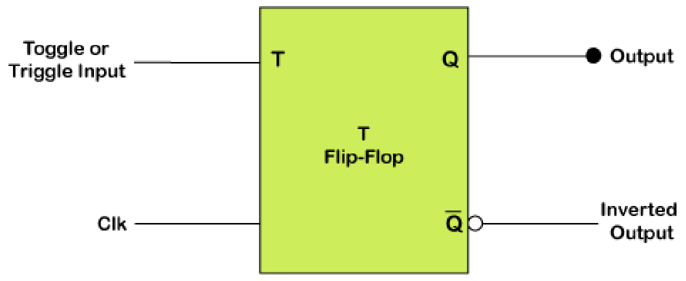
T Flip Flop

T stands for ("toggle") flip-flop to avoid an intermediate state in SR flip-flop. We should provide only one input to the flip-flop called Trigger input Toggle input to avoid an intermediate state occurrence.

Then the flip - flop acts as a Toggle switch. The next output state is changed with the complement of the present state output. This process is known as *Toggling*.

We can construct the T flip-flop by making changes in the ***JK flip-flop***. The T flip-flop has only one input, which is constructed by connecting the input of JK flip-flop. This single input is called T.

The Block diagram of the T flip-flop is given below where T defines the "Toggle" input, and CLK defines the "clock signal" input.



T Flip Flop Circuit

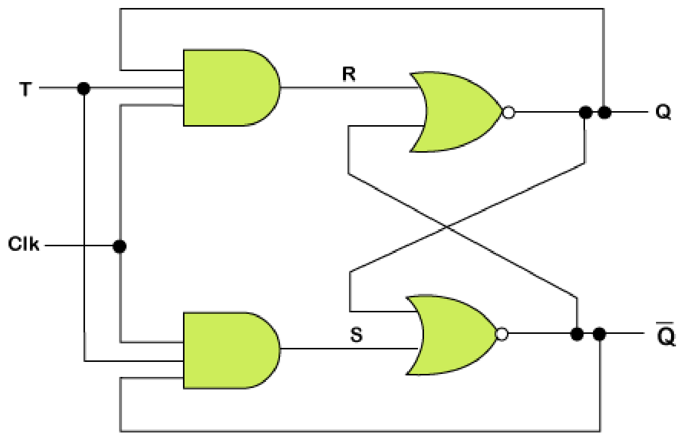
There are two methods which are used to form the T flip-flop:

- By connecting the output feedback to the input in "SR Flip Flop".
- We pass the output that we get after performing the XOR operation of T and Q_{PREV}output as the D input in D Flip Flop.

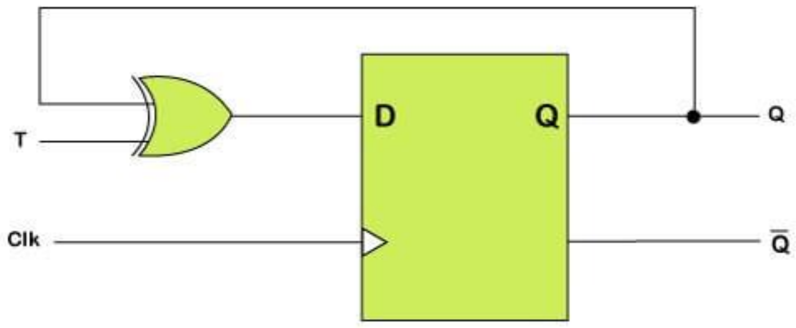
Construction

The T flip-flop is designed bypassing the AND gate's output as input to the NOR gate of the SR flip-flop. The inputs of the "AND" gates, the present output state Q, and its complement Q' are sent back to each AND gate.

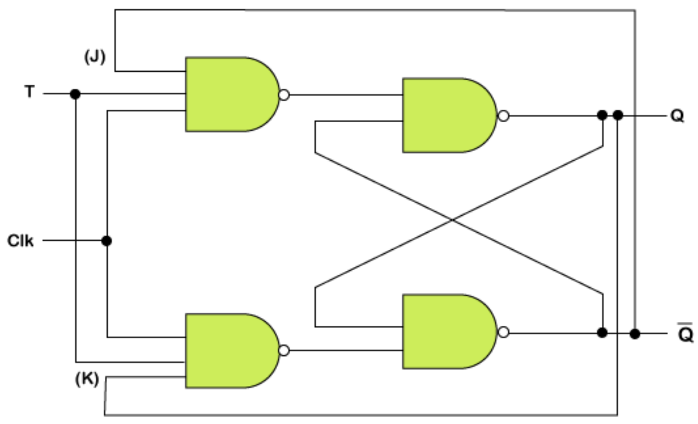
The toggle input is passed to the AND gates as input. These gates are connected to the Clock (CLK) signal. In the T flip-flop, a pulse train of little triggers is passed as the toggle input, which changes the flip flop's output state. The circuit diagram of the T flip flop using SR flip flop is given below:



The T flip flop is formed using the D flip flop. In D flip flop, the output after performing the XOR operation of the T input with the output "QPREV" is passed as the D input. The logical circuit of the T flip flop by using the D flip flop is given below:



The simplest construction of a D flip flop is with JK flip flop. Both the JK flip flop inputs are connected as a single input T. Below is the logical circuit of the T flip flop, which is formed from the JK flip flop:



Truth Table of T flip flop

T	Q	Q'	Q	Q'
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

The upper NAND gate is enabled, and the lower NAND gate is disabled when the output Q To is set to 0. Make the flip flop in "set state (Q=1)", the trigger passes the S input in the flip flop.

The upper NAND gate is disabled, and the lower NAND gate is enabled when the output Q is set to 1. The trigger passes the R input in the flip flop to make the flip flop in the "reset state (Q=0)".

Operations of T-Flip Flop

The T flip flop's next sate is similar to the current state when the T input is set to false or 0.

- If the toggle input is set to 0 and the present state is also 0, the next state will be 0.
- If toggle input is set to 0 and the present state is 1, the next state will be 1.

The next state of the flip flop is opposite to the current state when the toggle input is set to 1.

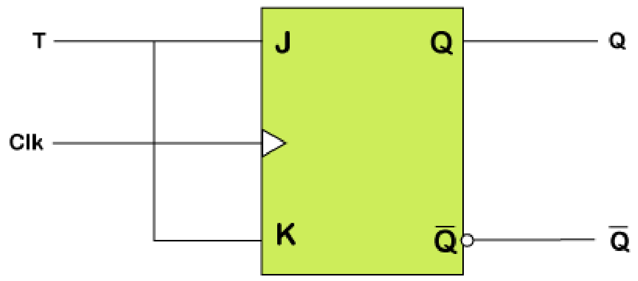
- If toggle input is set to 1 and the present state is 0, the next state will be 1.
- If toggle input is set to 1 and the present state is 1, the next state will be 0.

The T flip flop is toggled when the incoming trigger alternatively changes the set and reset inputs. The T flip flop requires two triggers to complete a full cycle of the output waveform.

The frequency of the output produced by the T flip flop is half of the input frequency. The T flip flop works as the "Frequency Divider Circuit."

In T flip flop, the state at an applied trigger pulse is defined only when the previous state is defined. It is the main drawback of the T flip flop.

The T flip flop can be designed from "JK Flip Flop", "SR Flip Flop", and "D Flip Flop" because the T flip flop is not available as ICs. The block diagram of T flip flop using "JK Flip Flop" is given below:



Example

```
module tff ( input clk, input rstn, input t, output reg q);

  always @ (posedge clk) begin
    if (!rstn)
      q <= 0;
    else
        if (t)
          q <= ~q;
        else
          q <= q;
  end
endmodule
```

Testbench

```
module tb;
  reg clk;
  reg rstn;
  reg t;

  tff u0 (  .clk(clk),
            .rstn(rstn),
           .t(t),
          .q(q));

  always #5 clk = ~clk;
  initial begin
    {rstn, clk, t} <= 0;

    $monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time, rstn, t, q);
    repeat(2) @(posedge clk);
    rstn <= 1;

    for (integer i = 0; i < 20; i = i+1) begin
      reg [4:0] dly = $random;
     #(dly) t <= $random;
    end
    #20 $finish;
  end
endmodule
```

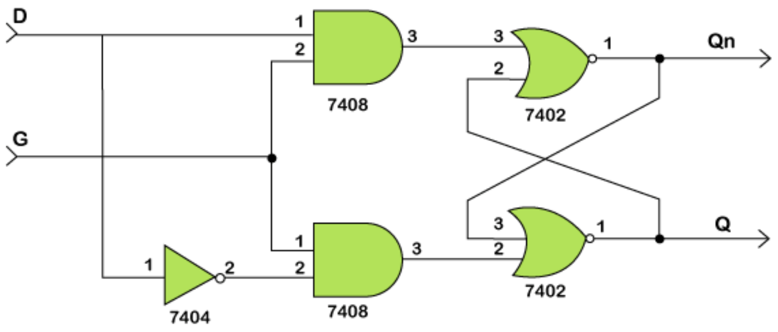
Verilog D Latch

A flip-flop captures data at its input at the negative or positive edge of a clock. The important thing is that whatever happens to data after the clock edge until the next clock edge will not be reflected in the output.

A *latch* does not capture at the edge of a clock; instead, the output follows input as long as it is asserted.

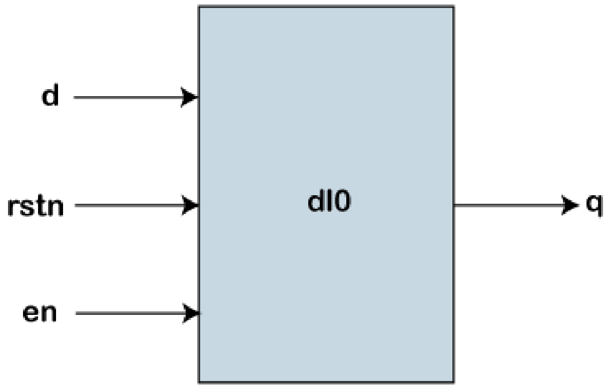
The D latch is used to store one bit of data. The D latch is essentially a modification of the gated SR latch.

The following image shows the parameters of the D latch in Verilog. The input D is the data to be stored. The input G is used to control the storing. The outputs Q and Qn are the stored data and the complement of the stored data, respectively



Example

In this example, we have a latch with three inputs and one output. The input *d* stands for data, which can be either 0 or 1, *rstn* stands for active-low reset, and *en* stands for enabling, which is used to make the input data latch to the output.



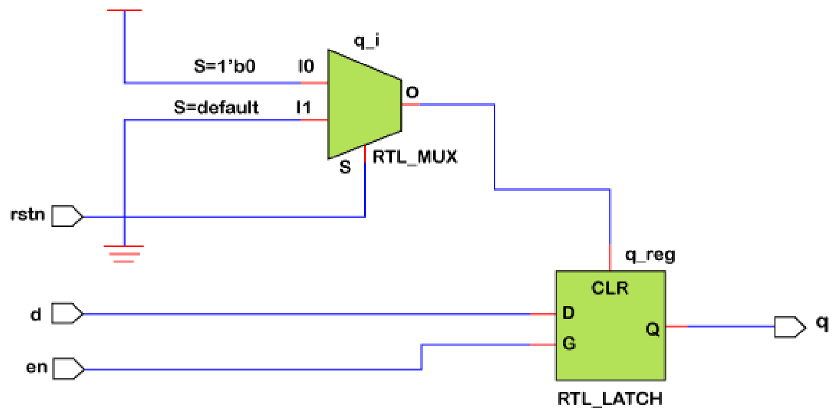
Reset being active-low means that the design element will be reset when this input goes to 0 or reset is active when its value is low. The value of output *q* is dictated by the inputs *d, en* and *rstn*.

```
module d_latch (  input d,           // 1-bit input pin for data
                  input en,          // 1-bit input pin for enabling the latch
                  input rstn,        // 1-bit input pin for active-low reset
                  output reg q);     // 1-bit output pin for data output

  // This always block is "always" triggered whenever en/rstn/d changes
  // If reset is asserted, then the output will be zero
  // Else as long as enable is high, output q follows input d

  always @ (en or rstn or d)
     if (!rstn)
       q <= 0;
     else
       if (en)
          q <= d;
endmodule
```

NOTE: The sensitivity list to the always block contains all the signals required to update the output.

This block will be triggered whenever any of the signals in the sensitivity list changes its value. Also, *q* will get the value of d only when *en* is high and hence is a *positive* latch.

Hardware Schematic



Testbench

```
module tb_latch;
  // Declare variables that can be used to drive values to the design
  reg d;
  reg en;
  reg rstn;
  reg [2:0] delay;
  reg [1:0] delay2;
  integer i;

  // Instantiate design and connect design ports with TB signals
  d_latch  dl0 ( .d (d),
                 .en (en),
                  .rstn (rstn),
                 .q (q));

  // This initial block forms the stimulus to test the design
  initial begin
    $monitor ("[%0t] en=%0b d=%0b q=%0b", $time, en, d, q);

    // 1. Initialize testbench variables
    d <= 0;
    en <= 0;
    rstn <= 0;

    // 2. Release reset
    #10 rstn <= 1;

```

```
      // 3. Randomly change d and enable
      for (i = 0; i < 5; i=i+1) begin
        delay = $random;
        delay2 = $random;
        #(delay2) en <= ~en;
        #(delay) d <= i;
      end
    end
endmodule
```

To make our testbench assert and deassert signals in a more random manner, we have declared a reg variable called *delay* of size 3 bits so that it can take any value from 0 to 7. Then the *delay* variable is used to delay the assignment of *d* and *en* to get different patterns in every loop.

The above code produce the following output, such as:

```
ncsim> run
[0] en=0 d=0 q=0
[11] en=1 d=0 q=0
[18] en=0 d=0 q=0
[19] en=0 d=1 q=0
[20] en=1 d=1 q=1
[25] en=1 d=0 q=0
[27] en=0 d=0 q=0
[32] en=0 d=1 q=0
[33] en=1 d=1 q=1
[34] en=1 d=0 q=0
ncsim: *W,RNQUIE: Simulation is complete.
```

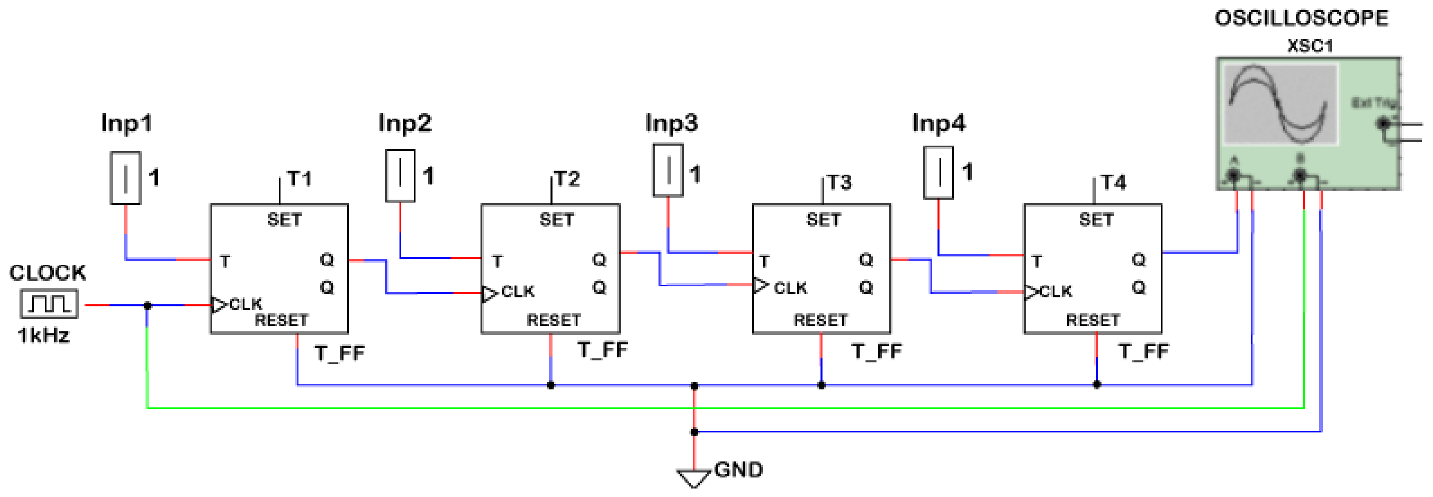
Verilog Ripple Counter

A ripple counter is an asynchronous counter in which the preceding flop's output clocks all the flops except the first.

Asynchronous means all the elements of the circuits do not have a common clock. For example, a 4 bit counter will count from 0000 to 1111.

Design

We will supply a 1Khz clock signal to the first T Flip Flop, and the rest of the three Flip Flops will have their clocks from the output (Q) of the previous Flip Flop. See the schematic below:



The above circuit contains 4 T Flip Flops because we need 4 bit Ripple Counter. T1 has its clock supplied by a Digital source of 1Khz, and the rest of Flip Flops used previous Flip Flop output as the clock

Input T of all T flip flops is HIGH (1) so that T Flip Flop toggles input on every clock edge.

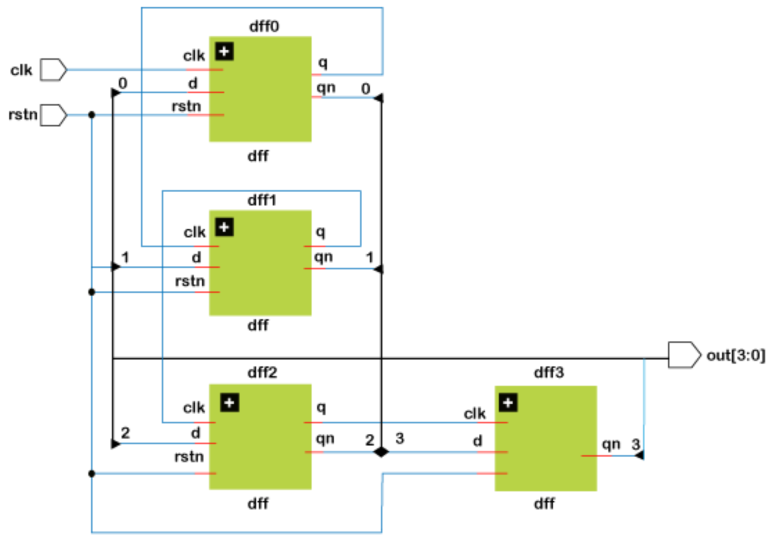
Example

We will do three modules to implement this counter. The first module is to implement the main program. The second module is used to implement T flip flop logic and the third to implement D Flip Flop logic.

```
module dff (input d,
            input clk,
             input rstn,
            output reg q,
             output qn);
  always @ (posedge clk or negedge rstn)
    if (!rstn)
      q <= 0;
    else
      q <= d;

  assign qn = ~q;
endmodule

module ripple ( input clk,
             input rstn,
              output [3:0] out);
  wire  q0;
```

```
 wire  qn0;
 wire  q1;
 wire  qn1;
 wire  q2;
 wire  qn2;
 wire  q3;
 wire  qn3;

 dff   dff0 ( .d (qn0),
              .clk (clk),
               .rstn (rstn),
              .q (q0),
               .qn (qn0));

 dff   dff1 ( .d (qn1),
              .clk (q0),
               .rstn (rstn),
              .q (q1),
               .qn (qn1));
 dff   dff2 ( .d (qn2),
               .clk (q1),
              .rstn (rstn),
               .q (q2),
              .qn (qn2));
 dff   dff3 ( .d (qn3),
              .clk (q2),
               .rstn (rstn),
              .q (q3),
               .qn (qn3));
 assign out = {qn3, qn2, qn1, qn0};
endmodule
```

Hardware Schematic



Testbench

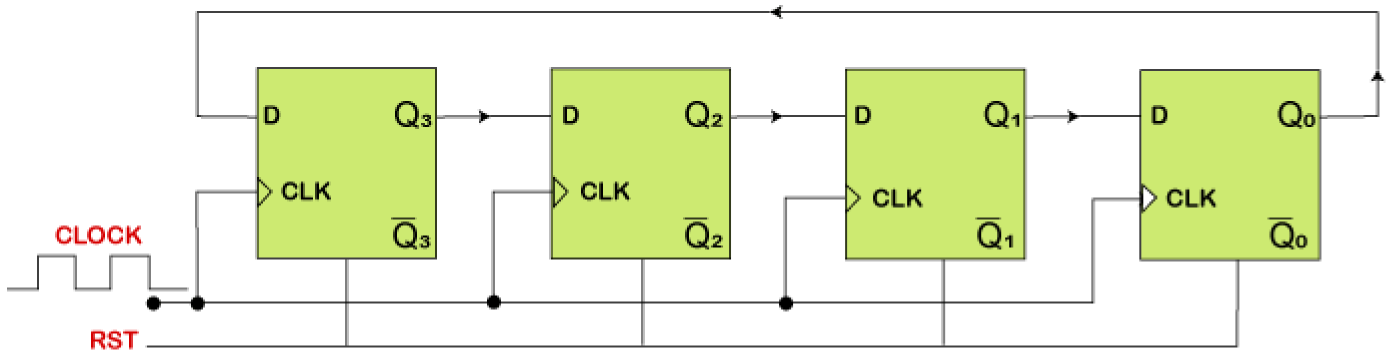
```
module tb_ripple;
  reg clk;
  reg rstn;
  wire [3:0] out;

  ripple r0   (  .clk (clk),
                  .rstn (rstn),
                 .out (out));

  always #5 clk = ~clk;

  initial begin
     rstn <= 0;
    clk <= 0;

    repeat (4) @ (posedge clk);
     rstn <= 1;

     repeat (25) @ (posedge clk);
    $finish;
  end
endmodule
```

Verilog Ring Counter

A ring counter is a digital circuit with a series of flip flops connected in a feedback manner. Ring Counter is composed of Shift Registers. The data pattern will re-circulate as long as clock pulses are applied.

The circuit is a special type of shift register where the last flip flop's output is fed back to the input of the first flip flop. When the circuit is reset, except one of the flip flop output, all others are made zero. For the n-flip flop ring counter, we have a *MOD-n* counter. That means the counter has n different states.

For example, if we take a 4-bit Ring Counter, then the data pattern will repeat every four clock pulses. If the pattern is 1000, it will generate 0100, 0010, 0001, 1000, etc.



Example

```
module ring_ctr  #(parameter WIDTH=4)
 (
   input clk,
   input rstn,
   output reg [WIDTH-1:0] out
 );

 always @ (posedge clk) begin
     if (!rstn)
       out <= 1;
     else begin
      out[WIDTH-1] <= out[0];
       for (int i = 0; i < WIDTH-1; i=i+1) begin
        out[i] <= out[i+1];
       end
     end
```

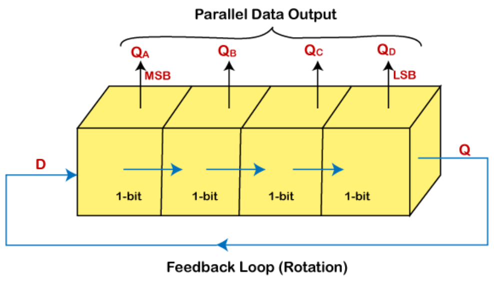
```
  end
endmodule
```

Testbench

```
module tb;
  parameter WIDTH = 4;

  reg clk;
  reg rstn;
  wire [WIDTH-1:0] out;

  ring_ctr  u0 (.clk (clk),
                .rstn (rstn),
               .out (out));

  always #10 clk = ~clk;

  initial begin
    {clk, rstn} <= 0;

    $monitor ("T=%0t out=%b", $time, out);
   repeat (2) @(posedge clk);
    rstn <= 1;
   repeat (15) @(posedge clk);
    $finish;
  end
endmodule
```

Rotational Movement of a Ring Counter

Since the above example has four distinct states, it is also known as a *modulo-4* or *mod-4* counter, with each flip-flop output having a frequency value equal to one-fourth of the main clock frequency.



The *MODULO* or *MODULUS* of a counter is the number of states the counter counts or sequences through before repeating itself, and a ring counter can be made to output any modulo number.

A *mod-n* ring counter will require *n* number of flip-flops connected to circulate a single data bit providing *n* different output states.

In the above example, only four of the possible sixteen states are used, making ring counters very inefficient in their output state usage.

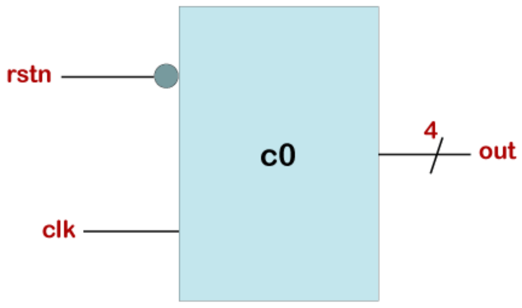
4-bit Counter

The 4-bit counter starts incrementing from 4'b0000 to 4'h1111 and come back to 4'b0000. It will keep counting as long as it is provided with a running clock, and reset is held high.

The rollover happens when the most significant bit of the final addition gets discarded. When the counter is at a maximum value of 4'h1111 and gets one more count request, the counter tries to reach 5'b10000, but since it can support only 4-bits, the MSB will be discarded, resulting in 0.

- 0000
- 0001
- 0010
- ...
- 1110
- 1111
- rolls over
- 0000
- 0001
- ...

The design contains two inputs, one for the clock and second for an active-low reset. An active-low reset is where the design is reset when the reset pin's value is 0. There is a 4-bit output called *out*, which essentially provides the counter values.



Electronic Counter Example

```
module counter (input clk,      // Declare input port for the clock to allow counter to count up
                input rstn,            // Declare input port for the reset to allow the counter to be reset to 0 when required
                 output reg[3:0] out);    // Declare 4-bit output port to get the counter values

 // This always block will be triggered at the rising edge of clk (0->1)
  // Once inside this block, it checks if the reset is 0, then change out to zero
  // If reset is 1, then the design should be allowed to count up, so increment the counter

  always @ (posedge clk) begin
    if (! rstn)
      out <= 0;
    else
      out <= out + 1;
  end
endmodule
```

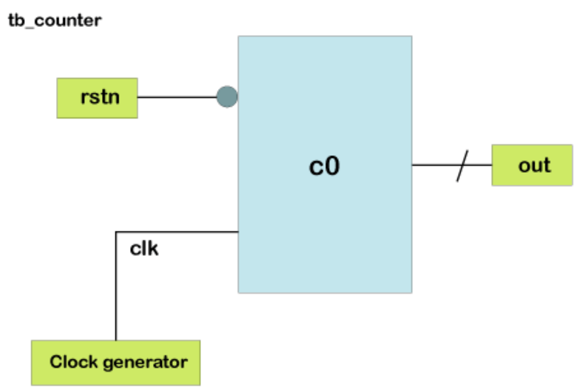
The *module* counter has a clock and active-low reset (*n*) as inputs and the counter value as a 4-bit output.

The *always* block is executed whenever the clock transitions from 0 to 1, which signifies a positive edge or a rising edge.

The output is incremented only if reset is held high or 1, achieved by the *if-else* block. If reset is low at the clock's positive edge, then output is reset to a default value of 4'b0000.

Testbench

We can instantiate the design into our testbench module to verify that the counter is counting as expected.



The testbench module is named *tb_counter*, and ports are not required since this is the top-module in simulation. However, we need to have internal variables to generate, store and drive the clock and reset.

For this purpose, we have declared two variables of type *reg* for clock and reset. We also need a *wire* type net to make the connection with the design's output; else, it will default to a 1-bit scalar net.

The clock is generated via *always* block, which will give a period of 10-time units. The *initial* block is used to set initial values to our internal variables and drive the design's reset value.

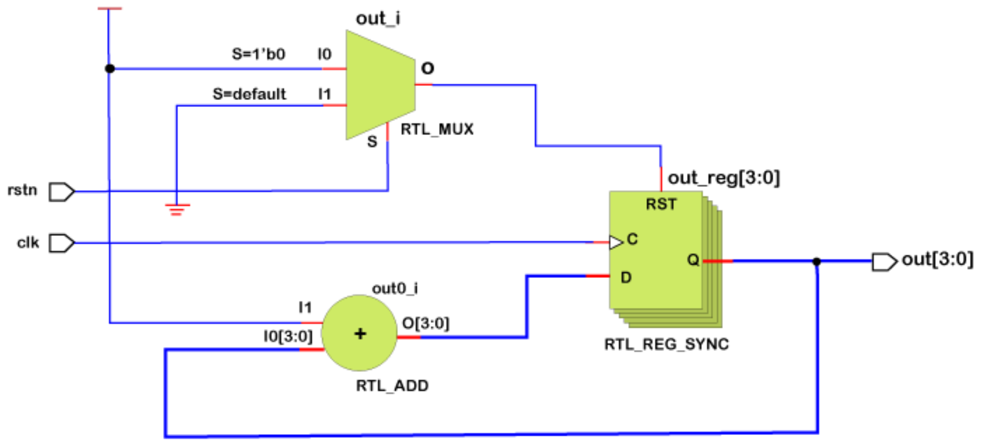
The design is instantiated in the testbench and connected to our internal variables to get the values when we drive them from the testbench.

We don't have any *$display* statements in our testbench, and hence we will not see any message in the console.

```
module tb_counter;
  reg clk;                   // Declare an internal TB variable called clk to drive clock to the design
  reg rstn;                  // Declare an internal TB variable called rstn to drive active low rese
t to design
  wire [3:0] out;            // Declare a wire to connect to design output

  // Instantiate counter design and connect with Testbench variables
  counter   c0 ( .clk (clk),
                 .rstn (rstn),
                  .out (out));

  // Generate a clock that should be driven to design
  // This clock will flip its value every 5ns -> time period = 10ns -> freq = 100 MHz

 always #5 clk = ~clk;

  // This initial block forms the stimulus of the testbench
  initial begin
    // Initialize testbench variables to 0 at start of simulation
    clk <= 0;
    rstn <= 0;

    // Drive rest of the stimulus, reset is asserted in between
    #20   rstn <= 1;
    #80   rstn <= 0;
    #50   rstn <= 1;

    // Finish the stimulus after 200ns
    #20 $finish;
  end
endmodule
```

Note that the counter resets to 0 when the active-low reset becomes 0, and when to reset is de-asserted at around 150ns, the counter starts counting from the next occurrence of the clock's positive edge.

Hardware Schematic



Verilog Mod-N Counter

Counters are sequential logic devices that follow a predetermined sequence of counting states triggered by an external clock (CLK) signal. The number of states or counting sequences through which a particular counter advances before returning to its original first state is called the *modulus* (MOD). In other words, the modulus (or modulo) is the number of states the counter counts and is the dividing number of the counter.

Modulus Counters, or ***MOD*** counters, are defined based on the number of states that the counter will sequence before returning to its original value.

For example, a 2-bit counter that counts from 00_2 to 11_2 in binary, 0 to 3 in decimal, has a modulus value of 4 (00 → 1 → 10 → 11, and return to 00); therefore, be called a modulo-4, or mod-4, counter. Note also that it has taken four clock pulses to get from 00 to 11.

In this example, there are only two bits (n = 2) then the maximum number of possible output states (maximum modulus) for the counter is $2^n = 2^2$ or 4. However, counters can be designed to count to any 2^n states in their sequence by cascading together multiple counting stages to produce a single modulus or MOD-N counter

Therefore, a "Mod-N" counter will require the "N" number of flip-flops connected to count a single data bit while providing 2^n different output states (n is the number of bits). Note that N is always a whole integer value.

Then we can see that MOD counters have a modulus value that is an integral power of 2, that is, 2, 4, 8, 16 and so on to produce an n-bit counter depending on the number of flip-flops used, and how they are connected, determining the type and modulus of the counter.

Example

```
module modN_ctr
  # (parameter N = 10,
     parameter WIDTH = 4)

  ( input   clk,
    input   rstn,
    output  reg[WIDTH-1:0] out);

  always @ (posedge clk) begin
    if (!rstn) begin
      out <= 0;
    end else begin
      if (out == N-1)
        out <= 0;
      else
        out <= out + 1;
    end
  end
endmodule
```

Testbench

Following is the testbench for the same example given above, such as:

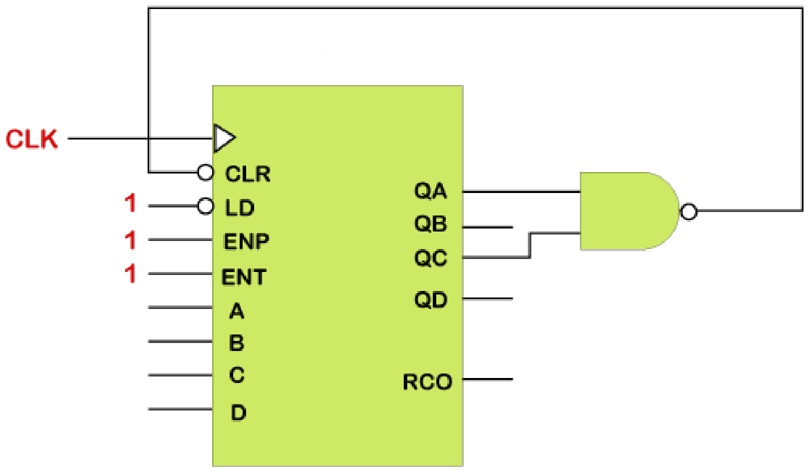
```
module tb;
  parameter N = 10;
  parameter WIDTH = 4;

  reg clk;
  reg rstn;
  wire [WIDTH-1:0] out;

  modN_ctr u0  (    .clk(clk),
                    .rstn(rstn),
                    .out(out));

  always #10 clk = ~clk;

  initial begin
    {clk, rstn} <= 0;

    $monitor ("T=%0t rstn=%0b out=0x%0h", $time, rstn, out);
    repeat(2) @ (posedge clk);
    rstn <= 1;

    repeat(20) @ (posedge clk);
    $finish;
  end
endmodule
```

The output looks like:

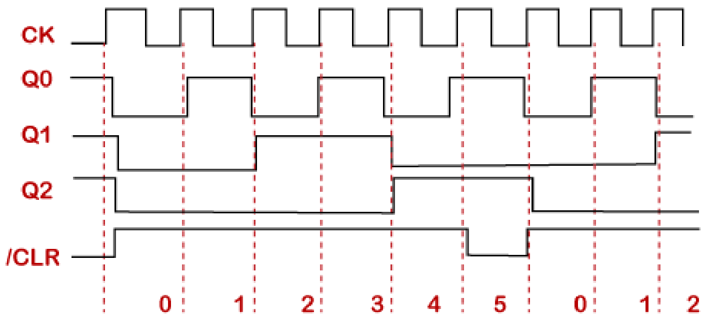
```
ncsim> run
T=0 rstn=0 out=0xx
T=10 rstn=0 out=0x0
T=30 rstn=1 out=0x0
T=50 rstn=1 out=0x1
T=70 rstn=1 out=0x2
T=90 rstn=1 out=0x3
T=110 rstn=1 out=0x4
T=130 rstn=1 out=0x5
T=150 rstn=1 out=0x6
T=170 rstn=1 out=0x7
T=190 rstn=1 out=0x8
T=210 rstn=1 out=0x9
T=230 rstn=1 out=0xa
T=250 rstn=1 out=0x0
T=270 rstn=1 out=0x1
T=290 rstn=1 out=0x2
T=310 rstn=1 out=0x3
T=330 rstn=1 out=0x4
T=350 rstn=1 out=0x5
T=370 rstn=1 out=0x6
T=390 rstn=1 out=0x7
T=410 rstn=1 out=0x8
Simulation complete via $finish(1) at time 430 NS + 0
```

Mod 6 Up Counter

This is a counter that resets at a chosen number. For example, a two-digit decimal counter, left to its own devices, will count from 00 to 99. This is not much use for a clock unless you have 100 seconds minutes.

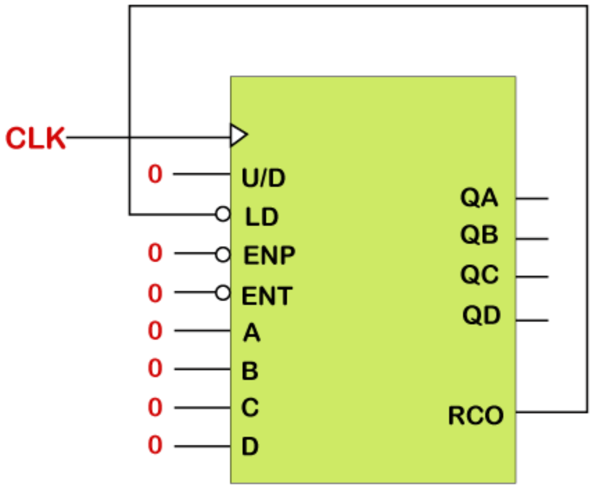


To fix the problem, the counter must go from 00 to 59. This is achieved by detecting a 6 in the left-hand digit and using it to reset the counter to zero. This would be a Modulo 6 Counter or 60 if we included both digits.

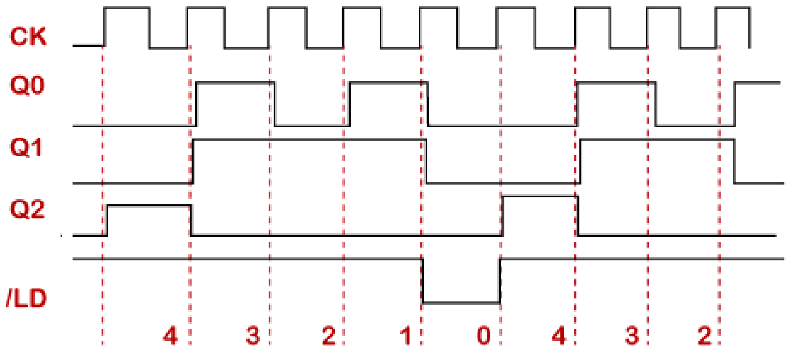


Mod 5 Down Counter

Suppose we want to design a MOD-5 counter. First, we know that "m = 5", so 2^n must be greater than 5.



As $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and 8 is greater than 5, then we need a counter with three flip-flops (N = 3) giving us a natural count of 000 to 111 in binary (0 to 7 decimal).



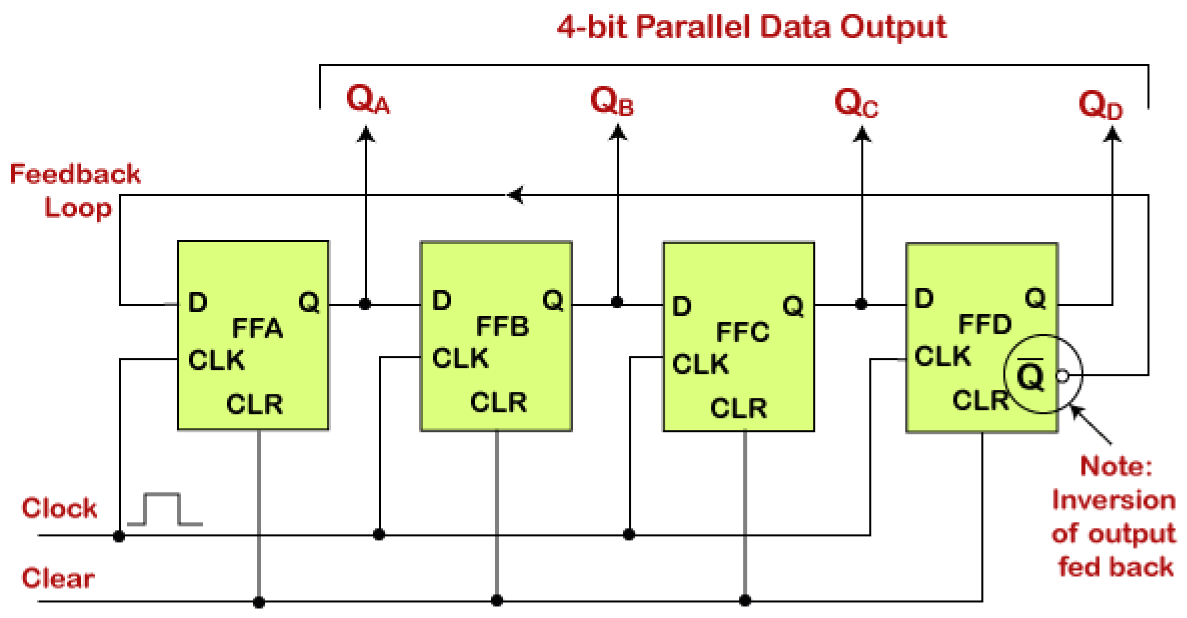
Verilog Johnson Counter

A Johnson counter is a digital circuit with a series of flip flops connected in a feedback manner. Verilog Johnson counter is a counter that counts 2N states if the number of bits is N.

The circuit is a special type of shift register where the last flip flop's complement output is fed back to the first flip flop's input. This is almost similar to the ring counter with a few extra advantages.

The Johnson counter's main advantage is that it only needs half the number of flip-flops compared to the standard ring counter, and then it's modulo number is halved. So an *n-stage* Johnson counter will circulate a single data bit, giving a sequence of 2n different states and can therefore be considered a *mod-2n* counter.

The inverted output Q of the last flip-flop is connected back to the input D of the first flip-flop. Below is the circuit diagram for a 4-bit Johnson counter:



This inversion of Q before it is fed back to input D causes the counter to *count* differently. Instead of counting through a fixed set of patterns just like the normal ring counter such as for a 4-bit counter, "0001"(1), "0010"(2), "0100"(4), "1000"(8) and repeat. The Johnson counter counts up and then down as the initial logic "1" passes through it to the right replacing the preceding logic "0".

A 4-bit Johnson ring counter passes blocks of four logic "0" and then four logic "1" thereby producing an 8-bit pattern.

As the inverted output Q is connected to the input D, this 8-bit pattern continually repeats. For example, "1000", "1100", "1110", "1111", "0111", "0011", "0001", "0000". This is demonstrated in the following table:

Clock Pulse No	**FFA**	**FFB**	**FFC**	**FFD**
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1

6	0	0	1	1
7	0	0	0	1

As well as counting or rotating data around a continuous loop, ring counters can also be used to detect or recognize various patterns or number values within a set of data. By connecting simple logic gates such as the ***OR*** gates to the flip-flops' outputs, the circuit can be made to detect a set number or value.

Standard 2, 3 or 4-stage **Johnson Ring Counters** can also be used to divide the clock signal frequency by varying their feedback connections, and divide-by-3 or divide-by-5 outputs are also available.

For example, a 3-stage Johnson Ring Counter could be used as a 3-phase, 120-degree phase shift square wave generator by connecting to the data outputs at A, B and NOT-B.

The standard 5-stage Johnson counter, such as the commonly available CD4017, is generally used as a synchronous decade counter/divider circuit.

Other combinations, such as the smaller 2-stage circuit, which is also called a *Quadrature* Oscillator or Generator, can be used to produce four individual outputs that are each 90 degrees out-of-phase for each other to produce a 4-phase timing signal.

Example

```
module johnson_ctr #(parameter WIDTH=4)
  (
    input clk,
    input rstn,
    output reg [WIDTH-1:0] out
  );

  always @ (posedge clk) begin
      if (!rstn)
         out <= 1;
      else begin
        out[WIDTH-1] <= ~out[0];
        for (int i = 0; i < WIDTH-1; i=i+1) begin
          out[i] <= out[i+1];
        end
      end
  end
endmodule
```

Testbench

```
module tb;
  parameter WIDTH = 4;

  reg clk;
  reg rstn;
  wire [WIDTH-1:0] out;

  johnson_ctr   u0 (.clk (clk),
                 .rstn (rstn),
                .out (out));

  always #10 clk = ~clk;

  initial begin
    {clk, rstn} <= 0;

    $monitor ("T=%0t out=%b", $time, out);
    repeat (2) @(posedge clk);
    rstn <= 1;
    repeat (15) @(posedge clk);
    $finish;
  end
endmodule
```

And the output looks like:

```
ncsim> run
T=0 out=xxxx
T=10 out=0001
T=50 out=0000
T=70 out=1000
T=90 out=1100
T=110 out=1110
T=130 out=1111
T=150 out=0111
T=170 out=0011
T=190 out=0001
T=210 out=0000
T=230 out=1000
T=250 out=1100
T=270 out=1110
T=290 out=1111
T=310 out=0111
Simulation complete via $finish(1) at time 330 NS + 0
```

Bidirectional Shift Register

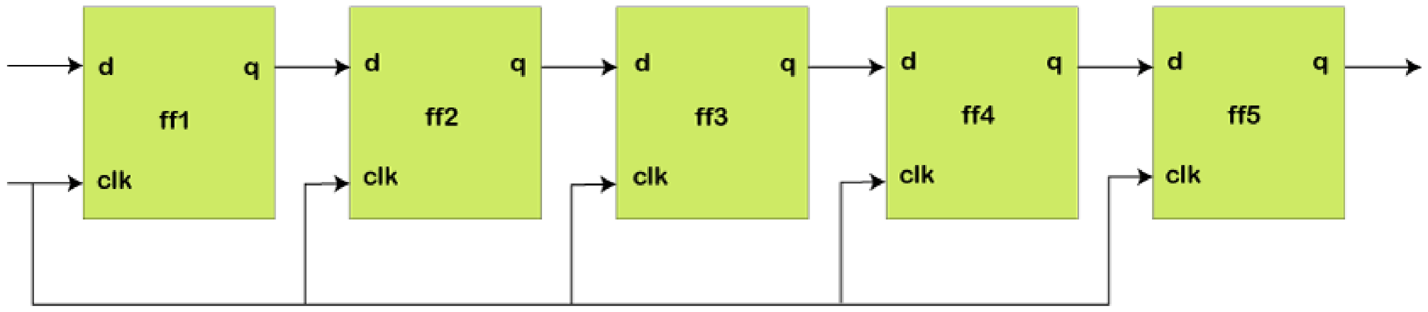
Flip flops can be used to store a single bit of binary data (1or 0). However, to store multiple bits of data, we need multiple flip flops. N flip flops are to be connected to store n bits of data.

A *Register* is a device that is used to store such information. It is a group of flip flops connected in series used to store multiple bits of data.

The information stored within these registers can be transferred using *shift registers*.

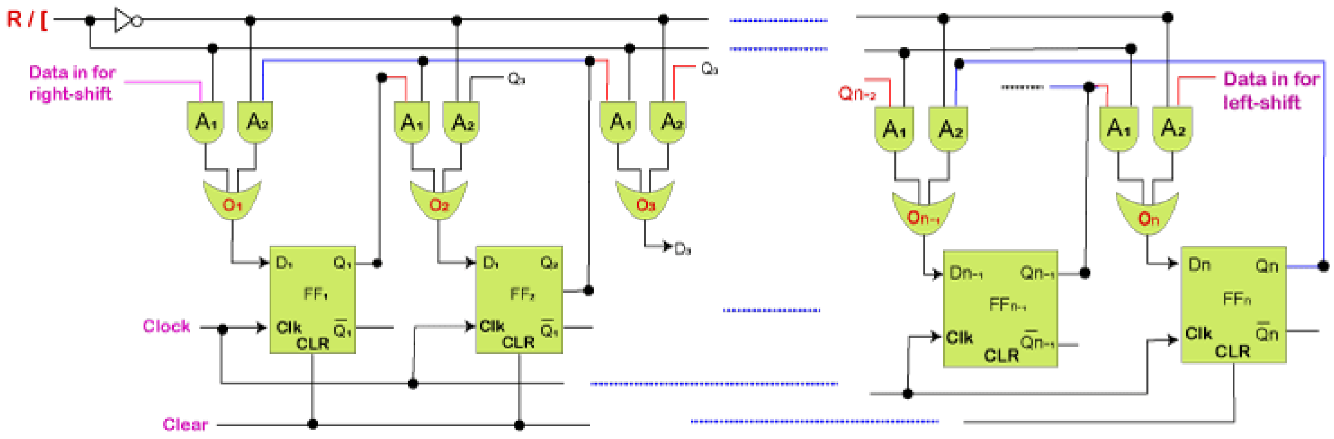
A *shift register* is a cascade of flip-flops where one flop's output pin *q* is connected to the next data input pin (d). Because all flops work on the same clock, the bit array stored in the shift register will shift by one position.

For example, if a 5-bit right shift register has an initial value of 10110 and the input to the shift register is tied to 0, then the next pattern will be 01011 and the next 00101.



Bidirectional shift registers are the storage devices capable of shifting the data either right or left, depending on the mode selected.

The following image shows an n-bit bidirectional shift register with serial data loading and retrieval capacity. Initially, all the flip flops in the register are reset by driving their clear pins high.



n-bit Bidirecional Shift Register

$R/\boxed{L}$ control line is made either low or high to opt for either left-shift or right-shift of the data bits, respectively.

An n-bit shift register can be formed by connecting n flip-flops where each flip flop stores a single bit of data.

The registers which will shift the bits to the left are called *Shift left registers*. The registers which will shift the bits to the right are called *Shift right registers*. Shift registers are basically of four types, such as:

1. Serial In Serial Out shift register
2. Serial In parallel Out shift register
3. Parallel In Serial Out shift register
4. Parallel In parallel Out shift register

Example

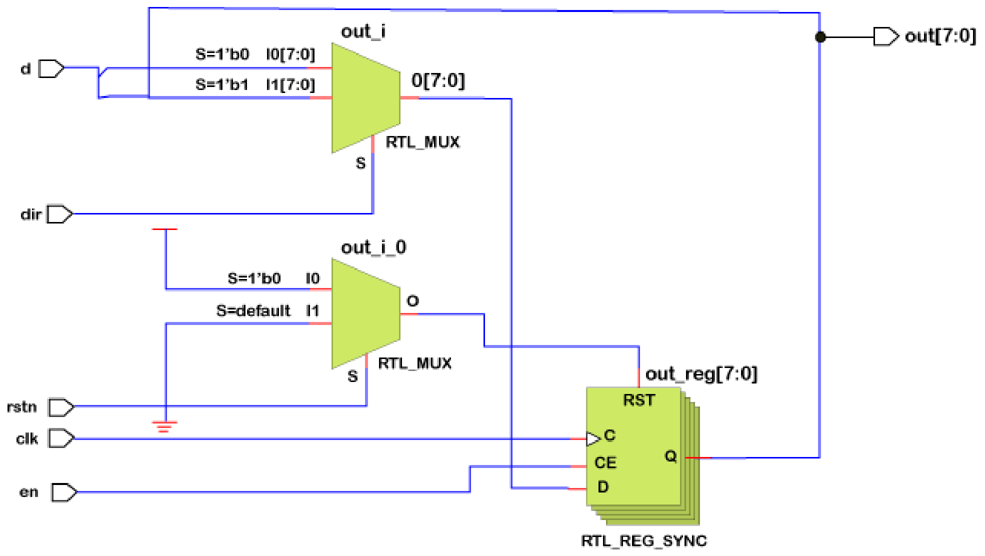
In this shift register example, we take five inputs and one n-bit output and the design is parameterized using parameter MSB to signify the width of the shift register.

If n is 4, then it becomes a 4-bit shift register. If n is 8, then it becomes an 8-bit shift register. This shift register has a few key features:

- It can be enabled or disabled by driving *en* pin.
- It can shift to the left as well as right when *dir* is driven.
- If *rstn* is pulled low, it will reset the shift register, and output will become 0.
- The input data value of the shift register can be controlled by *d* pin.

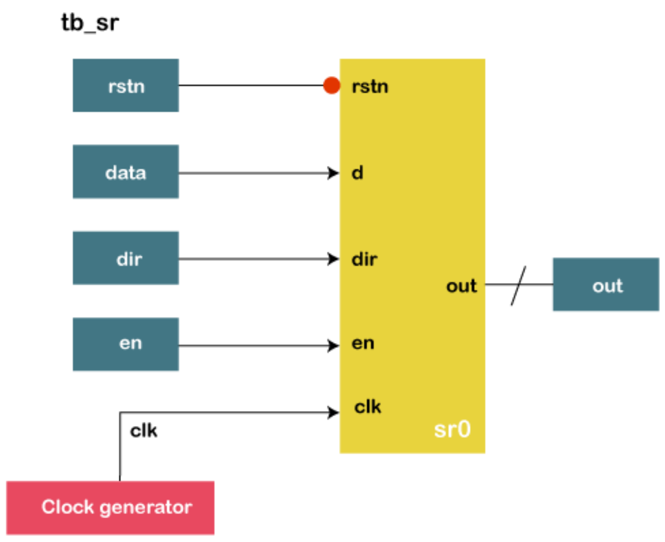
```
module shift_reg  #(parameter MSB=8) (  input d,              // Declare input for data to the f
irst flop in the shift register
                              input clk,          // Declare input for the clock to all flops in the shift register

                              input en,    // Declare input for enable to switch the shift register on/off

                              input dir,                 // Declare input to shift in either left or right direction
                              input rstn,       // Declare input to reset the register to a default value
                              output reg [MSB-1:0] out);
 // Declare output to read out the current value of all flops in this register



  // This always block will "always" be triggered on the rising edge of the clock
// Once it enters the block, it will first check to see if reset is 0 and if yes, then reset register
// If no, then check to see if the shift register is enabled
// If no => maintain previous output. If yes, then shift based on the requested direction

always @ (posedge clk)
  if (!rstn)
    out <= 0;
  else begin
    if (en)
      case (dir)
        0 :  out <= {out[MSB-2:0], d};
        1 :  out <= {d, out[MSB-1:1]};
      endcase
    else
      out <= out;
  end
endmodule
```

Hardware Schematic



Testbench

The testbench is used to verify the functionality of this shift register. The example is instantiated into the top *module*, and the inputs are driven with different values. The behavior for each of the inputs can be observed at the output *out* pin.



```
module tb_sr;
  parameter MSB = 16;       // [Optional] Declare a parameter to represent number of bits in shift r
egister

  reg data;                 // Declare a variable to drive d-input of design
  reg clk;                  // Declare a variable to drive clock to the design
  reg en;                   // Declare a variable to drive enable to the design
  reg dir;                  // Declare a variable to drive direction of shift register
  reg rstn;                 // Declare a variable to drive reset to the design
  wire [MSB-1:0] out;       // Declare a wire to capture output from the design

  // Instantiate design (16-bit shift register) by passing MSB and connect with TB signals
  shift_reg  #(MSB) sr0  (  .d (data),
                            .clk (clk),
                          .en (en),
                            .dir (dir),
                          .rstn (rstn),
                            .out (out));

  // Generate clock time period = 20ns, freq => 50MHz
  always #10 clk = ~clk;

  // Initialize variables to default values at time 0
  initial begin
    clk <= 0;
     en <= 0;
```

```
      dir <= 0;
      rstn <= 0;
      data <= 'h1;
    end

    // Drive main stimulus to the design to verify if this works
    initial begin

      // 1. Apply reset and deassert reset after some time
      rstn <= 0;
      #20 rstn <= 1;
          en <= 1;

      // 2. For 7 clocks, drive alternate values to data pin
      repeat (7) @ (posedge clk)
         data <= ~data;

      // 3. Shift direction and drive alternate value to data pin for another 7 clocks
      #10 dir <= 1;
      repeat (7) @ (posedge clk)
         data <= ~data;

      // 4. Drive nothing for the next 7 clocks, allow shift register to shift based on dir simply
      repeat (7) @ (posedge clk);

      // 5. Finish the simulation
      $finish;
    end

    // Monitor values of these variables and print them into the log file for debug
    Initial
      $monitor ("rstn=%0b data=%b, en=%0b, dir=%0b, out=%b", rstn, data, en, dir, out);
  endmodule
```

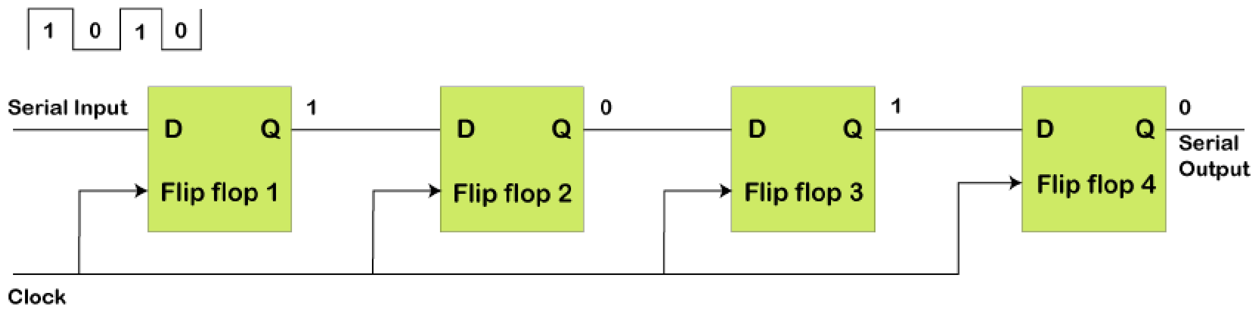
Serial-In Serial-Out Shift Register (SISO)

The shift register, which allows serial input (one bit after the other through a single data line) and produces a serial output, is known as the Serial-In Serial-Out shift register.

Since there is only one output, the data leaves the shift register one bit simultaneously in a serial pattern, thus the name Serial-In Serial-Out Shift Register.

The logic circuit given below shows a serial-in serial-out shift register. The circuit consists of four D flip-flops which are connected serially.

All these flip-flops are synchronous since the same clock signal is applied on each flip flop.



The above circuit is an example of a shift right register, taking the serial data input from the flip flop's left side. The main use of a SISO is to act as a delay element.

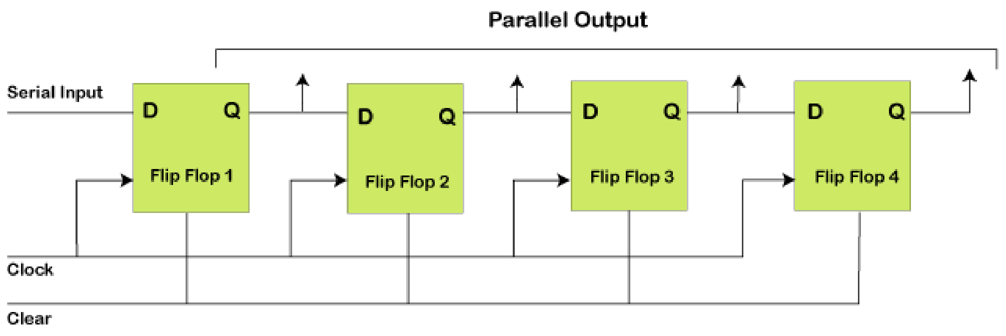
Serial-In Parallel-Out shift Register (SIPO)

The shift register, which allows serial input (one bit after the other through a single data line) and produces a parallel output, is known as the Serial-In Parallel-Out shift register.

The logic circuit given below shows a serial-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected.

The clear (CLR) signal is connected, and the clock signal to all the 4 flip flops to RESET them. The first flip flop's output is connected to the next flip flop's input and so on.

All these flip-flops are synchronous since the same clock signal is applied on each flip flop.



The above circuit is an example of a shift right register, taking the serial data input from the flip flop's left side and producing a parallel output.

They are used in communication lines where a data line's demultiplexing into several parallel lines is required because the main use of the SIPO register is to convert serial data into parallel data.

Parallel-In Serial-Out Shift Register (PISO)

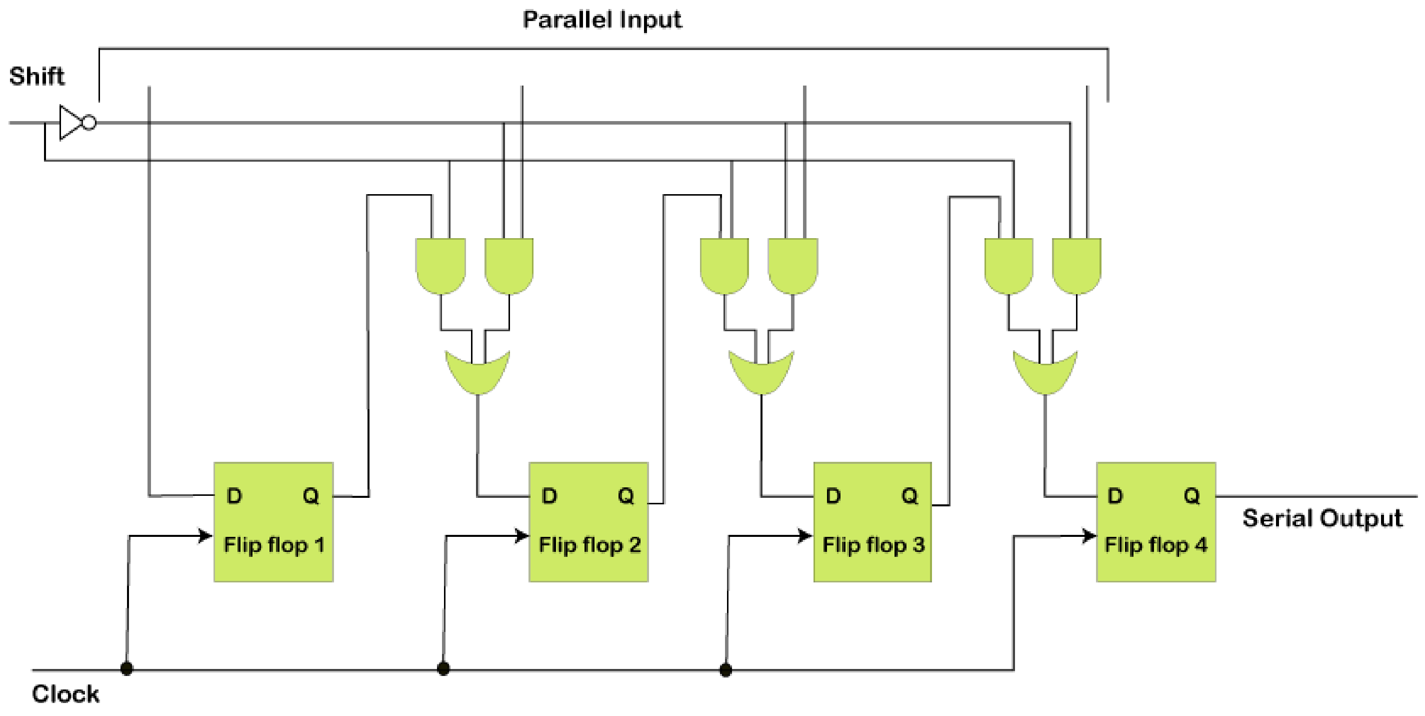
The shift register, which allows parallel input (data is given separately to each flip flop and simultaneously) and produces a serial output, is known as the Parallel-In Serial-Out shift register.

The logic circuit given below shows a parallel-in-serial-out shift register. The circuit consists of four D flip-flops which are connected.

The clock input is directly connected to all the flip flops. The input data is still connected individually to each flip flop through a multiplexer at every flip flop's input.

The output of the previous flip flop and parallel data input is connected to the MUX input, and the output of MUX is connected to the next flip flop.

All these flip-flops are synchronous since the same clock signal is applied to each flip flop.



A Parallel in Serial out (PISO) shift registers used to convert parallel data to serial data.

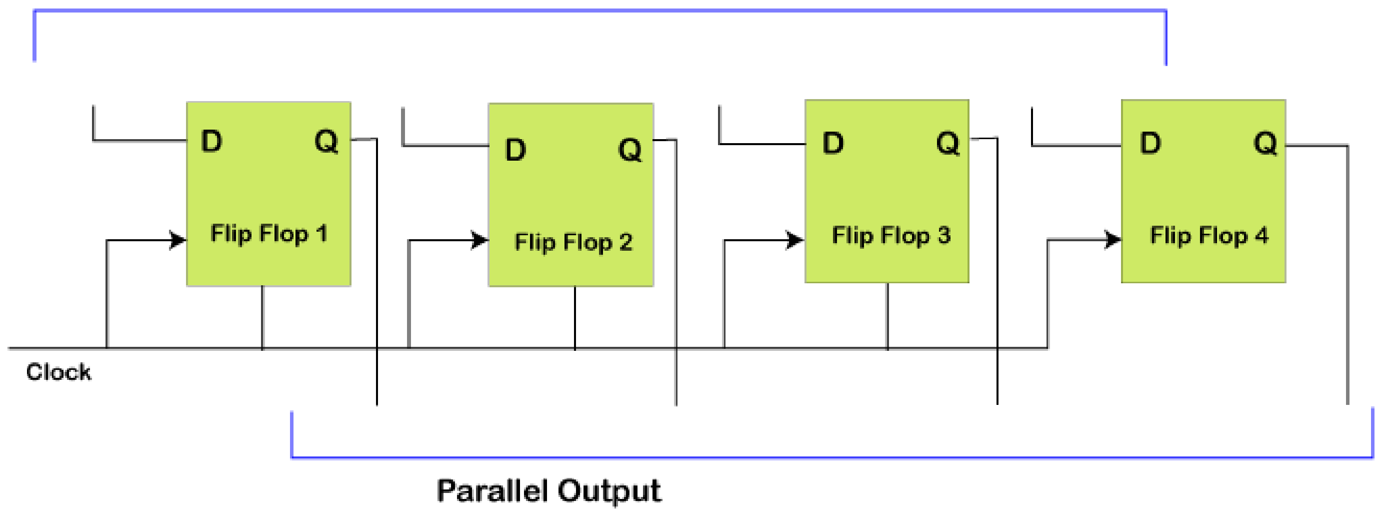
Parallel-In Parallel-Out Shift Register (PIPO)

The shift register allows parallel input (data is given separately to each flip flop and simultaneously) and produces a parallel output, known as the Parallel-In parallel-Out shift register.

The logic circuit given below shows a parallel-in-parallel-out shift register. The circuit consists of four D flip-flops which are connected. The clear (CLR) signal and clock signals are connected to all the 4 flip flops.

There are no interconnections between the individual flip-flops in this type of register since no serial shifting of the data is required.

Data is given as input separately for each flip flop and in the same way, and output is also collected individually from each flip flop.



A Parallel in Parallel out (PIPO) shift register is used as a temporary storage device, and like the SISO Shift register, it acts as a delay element.

Bidirectional Shift Register

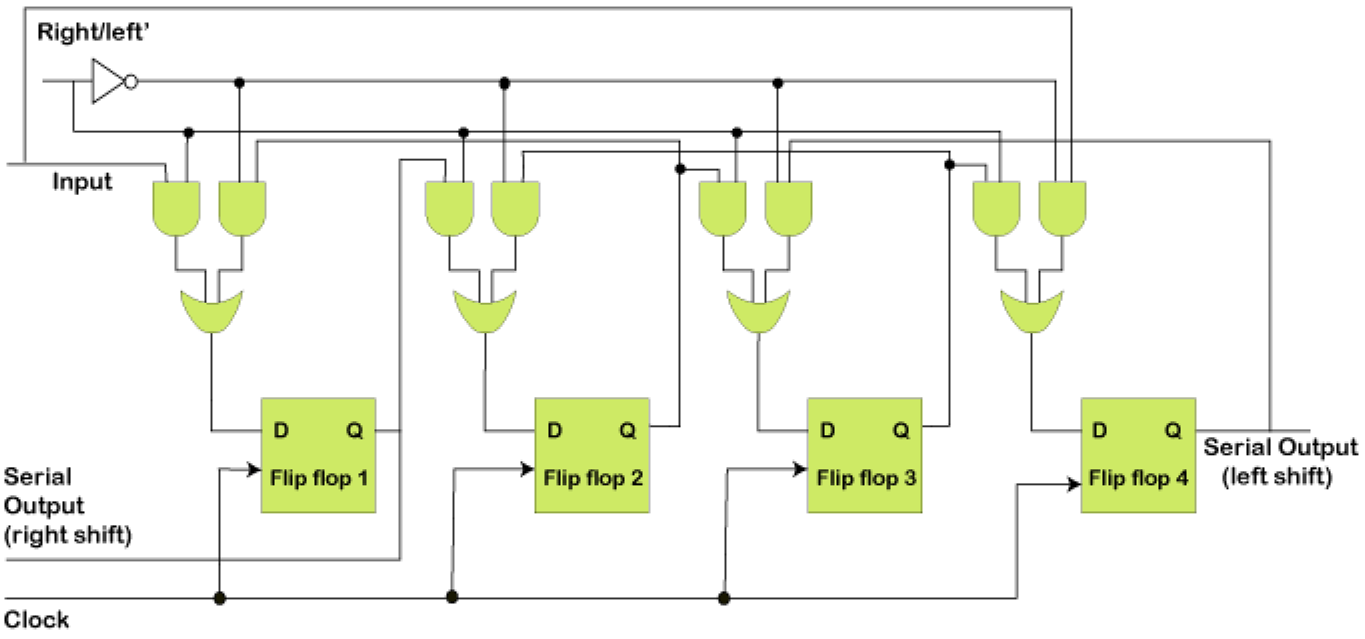
If we shift a binary number to the left by one position, it is equivalent to multiplying the number by 2. If we shift a binary number to the right by one position, it is equivalent to dividing the number by 2.To perform these operations; we need a register that can shift the data in either direction.

Bidirectional shift registers are the registers capable of shifting the data either right or left, depending on the mode selected.

If the mode selected is 1(high), then the data will be shifted towards the right direction, and if the mode selected is 0(low), then the data will be shifted towards the left direction.

The logic circuit given below shows a Bidirectional shift register. The circuit consists of four D flip-flops which are connected.

The input data is connected at two ends of the circuit, and depending on the mode selected, only one and gate are in the active state.



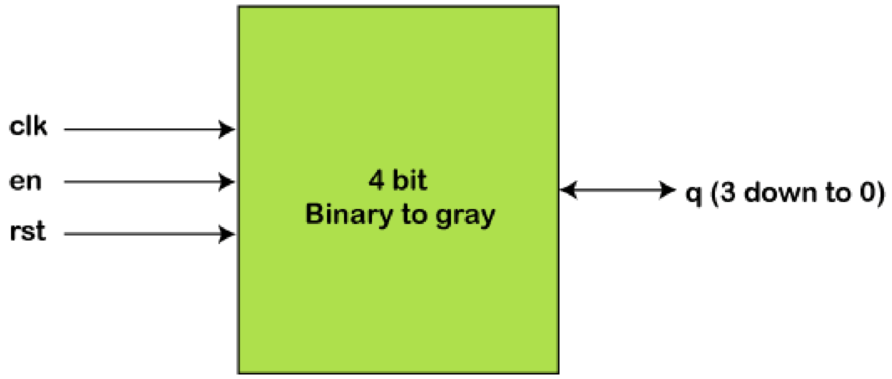
Applications of shift Registers

Here are the following applications of the shift registers, such as:

- The shift registers are used for temporary data storage.
- The shift registers are also used for data transfer and data manipulation.
- The serial-in serial-out and parallel-in parallel-out shift registers are used to produce time delay to digital circuits.
- The serial-in parallel-out shift register is used to convert serial data into parallel data. They are used in communication lines where demultiplexing of a data line into several parallel lines is required.
- A Parallel in Serial out shift register is used to convert parallel data to serial data.

Verilog Gray Counter

Gray code is a kind of binary number system where only one bit will change at a time. Today gray code is widely used in the digital world. It will be helpful for error correction and signal transmission. The Gray counter is also useful in design and verification in the VLSI domain.



A Gray Code encodes integers as sequences of bits with the property that the representations of adjacent integers differ in exactly one binary position.

There are different types of gray codes, such as Balanced, Binary reflected, Maximum Gap, and Antipodal Gray code.

Counters have a primary function of producing a specified output sequence and are sometimes referred to as pattern generators

Design

In a gray code, only one bit changes at one time. This design code has two inputs, clock and reset signals and one 4 bit output that will generate gray code.

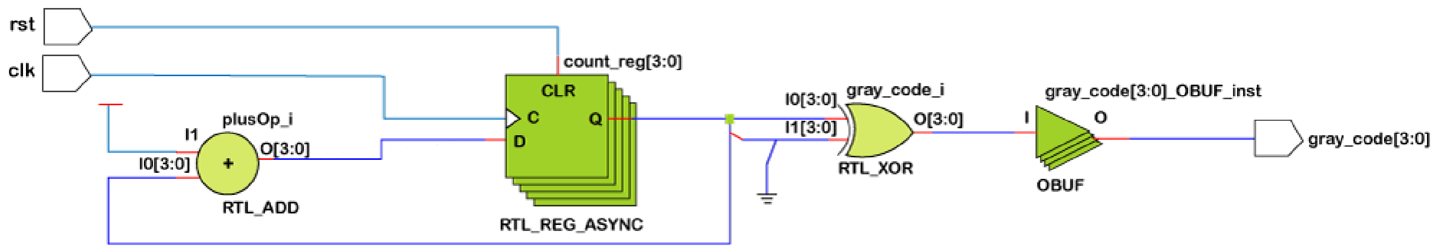
First, if the *rstn* signal is high, then the output will be zero, and as soon as *rstn* goes low, on the rising edge of *clk*, the design will generate a four-bit gray code and continue to generate at every rising edge of *clk* signal.

This design code can be upgraded and put binary numbers as input, and this design will work as a binary to gray code converter.

```
module gray_ctr
  # (parameter N = 4)

  ( input   clk,
    input   rstn,
    output reg [N-1:0] out);

    reg [N-1:0] q;

    always @ (posedge clk) begin
        if (!rstn) begin
          q <= 0;
```

```
            out <= 0;
        end else begin
          q <= q + 1;
`ifdef FOR_LOOP
          for (int i = 0; i < N-1; i= i+1) begin
          out[i] <= q[i+1] ^ q[i];
          end
          out[N-1] <= q[N-1];
`else
            out <= {q[N-1], q[N-1:1] ^ q[N-2:0]};
`endif
    end
    end
endmodule
```

Hardware Schematic



Testbench

```
module tb;
  parameter N = 4;

  reg clk;
  reg rstn;
  wire [N-1:0] out;

  gray_ctr u0 ( .clk(clk),
                .rstn(rstn),
               .out(out));

  always #10 clk = ~clk;

  initial begin
    {clk, rstn} <= 0;

    $monitor ("T=%0t rstn=%0b out=0x%0h", $time, rstn, out);

    repeat(2) @ (posedge clk);
    rstn <= 1;
    repeat(20) @ (posedge clk);
    $finish;
  end
endmodule
```

And it produces the following output, such as:

```
ncsim> run
T=0 rstn=0 out=0xx
T=10 rstn=0 out=0x0
T=30 rstn=1 out=0x0
T=50 rstn=1 out=0x1
T=70 rstn=1 out=0x3
T=90 rstn=1 out=0x2
T=110 rstn=1 out=0x6
T=130 rstn=1 out=0x7
T=150 rstn=1 out=0x5
T=170 rstn=1 out=0x4
T=190 rstn=1 out=0xc
T=210 rstn=1 out=0xd
T=230 rstn=1 out=0xf
T=250 rstn=1 out=0xe
T=270 rstn=1 out=0xa
T=290 rstn=1 out=0xb
T=310 rstn=1 out=0x9
T=330 rstn=1 out=0x8
T=350 rstn=1 out=0x0
T=370 rstn=1 out=0x1
T=390 rstn=1 out=0x3
T=410 rstn=1 out=0x2
Simulation complete via $finish(1) at time 430 NS + 0
```

Balanced Gray Code

In balanced Gray codes, the number of changes in different coordinate positions is as close as possible.

A Gray code is *uniform* or *uniformly* balanced if its transition counts are all equal.

Gray codes can also be *exponentially* balanced if all of their transition counts are adjacent powers of two, and such codes exist for every power of two.

For example, a balanced 4-bit Gray code has 16 transitions, which can be evenly distributed among all four positions (four transitions per position), making it uniformly balanced.

1. 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 0

2. 0 0 1 1 1 1 0 0 1 1 1 1 0 0 0 0
3. 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0
4. 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1 1

n-ary Gray Code

There are many specialized types of Gray codes other than the binary-reflected Gray code. One such type of Gray code is the n-ary Gray code, also known as a *non-Boolean* Gray code. As the name implies, this type of Gray code uses non-Boolean values in its encodings.

For example, a 3-ary ternary Gray code would use the values {0, 1, and 2}. The (n, k)-Gray code is the n-ary Gray code with k digits. The sequence of elements in the (3, 2)-Gray code is: {00, 01, 02, 12, 11, 10, 20, 21, and 22}.

The (n, k)-Gray code may be constructed recursively, as the BRGC, or may be constructed iteratively.

Monotonic Gray Codes

Monotonic codes are useful in interconnection networks theory, especially for minimizing dilation for linear arrays of processors.

If we define the weight of a binary string to be the number of 1s in the string, then although we clearly cannot have a Gray code with strictly increasing weight, we may want to approximate this by having the code run through two adjacent weights before reaching the next one.

Beckett-Gray Code

Another type of Gray code, the Beckett-Gray code, is named for Irish playwright *Samuel Beckett*, who was interested in *symmetry*. His play *Quad* features four actors and is divided into sixteen time periods. Each period ends with one of the four actors entering or leaving the stage.

The play begins with an empty stage, and Beckett wanted each subset of actors to appear on stage exactly once. A 4-bit binary Gray code can represent the set of actors currently on stage.

However,

Beckett placed an additional restriction on the script: he wished the actors to enter and exit so that the actor who had been on stage the longest would always be the one to exit.

The actors could then be represented by a first-in, first-out (FIFO) queue so that the actor being dequeued is always the one who was enqueued first.

Beckett was unable to find a Beckett-Gray code for his play, and indeed, an exhaustive listing of all possible sequences reveals that no such code exists for n = 4. It is known today that such codes do exist for n = 2, 5, 6, 7, and 8, and do not exist for n = 3 or 4.

Snake-in-the-box Codes

Snake-in-the-box codes, or snakes, are the sequences of nodes of induced paths in an n-dimensional *hypercube* graph, and coil-in-the-box codes, or coils, are the sequences of nodes of induced cycles in a hypercube.

Viewed as Gray codes, these sequences have the property of detecting any single-bit coding error.

Single-track Gray Code

Another kind of Gray code is the single-track Gray code (STGC) developed by *Norman B. Spedding* and refined by *Hiltgen, Paterson* and *Brandestini* in "Single-track Gray codes" (1996).

The STGC is a cyclical list of P unique binary encodings of length n such that two consecutive words differ in exactly one position. When the list is examined as a P × n matrix, each column is a cyclic shift of the first column.

The name comes from their use with rotary encoders, where many tracks are being sensed by contacts, resulting in each in an output of 0 or 1. To reduce noise due to different contacts not switching the same moment in time, one preferably sets up the tracks so that the contacts' data output is in Gray code.

To get high angular accuracy, one needs lots of contacts; to achieve at least 1-degree accuracy, one needs at least 360 distinct positions per revolution, which requires a minimum of 9 bits of data and the same number of contacts.

If all contacts are placed at the same angular position, then 9 tracks are needed to get a standard BRGC with at least 1-degree accuracy. However, if the manufacturer moves a contact to a different angular position but at the same distance from the center shaft, then the corresponding "ring pattern" needs to be rotated the same angle to give the same output.

Two-dimensions Gray Code

Two-dimensional Gray codes are used in communication to minimize the number of bit errors in quadrature amplitude modulation adjacent points in the constellation.

In a standard encoding, the horizontal and vertical adjacent constellation points differ by a single bit, and adjacent diagonal points differ by 2 bits.

Verilog File Operations

Verilog has system tasks and functions that can open files, output values into files, read values from files and load into other variables and close files.

This application describes how the Verilog model or testbench can read text and binary files to load memories, apply a stimulus, and control simulation. The file I/O functions format is based on the C stdio routines, such as *fopen, fgetc, fprintf,* and *fscanf*.

The Verilog language has a set of system functions to write files ($fdisplay, $fwrite, etc.) but only reads files with a single, fixed format ($readmem).

In the past, if we wanted to read a file that was not in *$readmem* format, we would have to learn the Programming Language Interface (PLI) and the C language, write C code to read the file and pass values into Verilog, then debug the combined C and Verilog code. Also, the Verilog is limited to 32 open files at a time.

However, using the new file I/O system functions, we can perform the file I/O directly from Verilog. We can write Verilog HDL to:

- Read stimulus files to apply patterns to the inputs of a model.
- Read a file of expected values for comparison with your model.
- Read a script of commands to drive simulation.
- Read either ASCII or binary files into Verilog registers and memories.
- Have hundreds of log files open simultaneously (though they are written to one at a time).

The code for all the examples in this file is included in the examples directory for the file I/O functions.

NOTE: These system tasks behave the same as the equivalent stdio routines. For example, $fscanf will skip over white-space, including blank lines, just like fscanf(). We can prototype code in C then convert it to Verilog.

Opening and Closing Files

The *$fopen* function opens a file and returns a multi-channel descriptor in an unsized integer format. This is unique for each file. All communications between the simulator and the file take place through the file descriptor. Users can specify only the name of a file as an argument. It will create a file in the default folder or a folder given in the full path description.

We use the *$fclose* function to close an opened file. This function is called without any arguments. It simply closes all open files. If an argument is specified, it will close only a file in which the descriptor is given. By default, before the simulator terminates, all open files are closed. It means that the user does not have to close any files, and closing is done automatically by the simulator.

All file output tasks work in the same way as their corresponding display tasks. The only difference is a file descriptor that appears as the first argument in the function argument list. These functions only can append data to a file and cannot read data from files.

```
module tb;
  // Declare a variable to store the file handler
   integer fd;

   initial begin
```

```
        // Open a new file by the name "my_file.txt"
        // with "write" permissions, and store the file
        // handler pointer in variable "fd"
        fd = $fopen("my_file.txt", "w");

        // Close the file handle pointed to by "fd"
        $fclose(fd);
    end
endmodule
```

Opening File Modes

Argument	Description
"r" or "rb"	Open for reading.
"w" or "wb"	Create a new file for writing. If the file exists, truncate it to zero length and overwrite it.
"a" or "ab"	If the file exists, append (open for writing at EOF), else create a new file.
"r+", "r+b" or "rb+"	Open for both reading and writing.
"w+", "w+b" or "wb+"	Truncate or create for an update.
"a+", "a+b", or "ab+"	Append, or create a new file for an update at EOF.

How to Write Files

Function	Description
$fdisplay	Similar to $display, write out to file instead
$fwrite	Similar to $write, write out to file instead

$fstrobe	Similar to $strobe, write out to file instead
$fmonitor	Similar to $monitor, write out to file instead

Each of the above system's functions prints values in radix decimal. They also have three other versions to print values in binary, octal and hexadecimal.

Function	Description
$fdisplay()	Prints in decimal by default
$fdisplayb()	Prints in binary
$fdisplayo()	Prints in octal
$fdisplayh()	Prints in hexadecimal

```
module tb;
   integer    fd;
   integer    i;
   reg [7:0]  my_var;

   initial begin
      // Create a new file
      fd = $fopen("my_file.txt", "w");
      my_var = 0;

     $fdisplay(fd, "Value displayed with $fdisplay");
      #10 my_var = 8'h1A;
      $fdisplay(fd, my_var);      // Displays in decimal
      $fdisplayb(fd, my_var);     // Displays in binary
      $fdisplayo(fd, my_var);     // Displays in octal
      $fdisplayh(fd, my_var);     // Displays in hex

     // $fwrite does not print the newline char '\n' automatically at
     // the end of each line; So we can predict all the values printed
     // below to appear on the same line
     $fdisplay(fd, "Value displayed with $fwrite");
      #10 my_var = 8'h2B;
      $fwrite(fd, my_var);
      $fwriteb(fd, my_var);
      $fwriteo(fd, my_var);
      $fwriteh(fd, my_var);


     // Jump to new line with '\n', and print with strobe which takes
     // the final value of the variable after non-blocking assignments
     // are done
     $fdisplay(fd, "\nValue displayed with $fstrobe");
      #10 my_var <= 8'h3C;
      $fstrobe(fd, my_var);
      $fstrobeb(fd, my_var);
      $fstrobeo(fd, my_var);
      $fstrobeh(fd, my_var);
     #10 $fdisplay(fd, "Value displayed with $fmonitor");
     $fmonitor(fd, my_var);
      for(i = 0; i < 5; i= i+1) begin
          #5 my_var <= i;
      end
     #10 $fclose(fd);
   end
endmodule
```

The above code gives the following outputs, such as:

```
Value displayed with $fdisplay
26
00011010
032
1a
The value displayed with $fwrite
43001010110532b
The value displayed with $fstrobe
60
00111100
074
3c
The value displayed with $fmonitor
60
0
```

Read Files

To read and store data from a memory file, we use the *$readmemh* and *$readmemb* functions.

The **$readmemb** task reads binary data, and **$readmemh** reads hexadecimal data. Data has to exist in a text file. White space is allowed to improve readability, as well as comments in both single-line and block. The numbers have to be stored as binary or hexadecimal values. The basic form of a memory file contains numbers separated by new line characters loaded into the memory.

When a function is invoked without starting and finishing addresses, it loads data into memory starting from the first cell. Start and finish addresses have to be used to load data only into a specific part of memory.

The address can be explicit, given in the file with the @ character, followed by a hexadecimal address with data separated by a space. It is essential to remember the start and finish addresses range given in the file. The argument in function calls has to match each other. Otherwise, an error message will be displayed, and the loading process will be terminated.

Reading a Line

The system function *$fgets* reads characters from the file specified by [hl]fd[/hd] into the variable str until *str* is filled, or a newline character is read and transferred to str, or an EOF condition is encountered.

If an error occurs during the read, it returns code zero. Otherwise, it returns the number of characters read.

Detecting EOF

The system function *$feof* returns a non-zero value when EOF is found and returns zero otherwise for a given file descriptor as an argument.

```
module tb;
   reg[8*45:1] str;
   integer    fd;

   initial begin
     fd = $fopen("my_file.txt", "r");

     // Keep reading lines until EOF is found
     while (! $feof(fd)) begin

```

```
        // Get current line into the variable 'str'
        $fgets(str, fd);

        // Display contents of the variable
        $display("%0s", str);
      end
      $fclose(fd);
    end
endmodule
```

Multiple Arguments to fdisplay

When multiple variables are given to *$fdisplay*, it simply prints all variables in the given order one after another without space.

```
  reg [3:0] a, b, c, d;
  reg [8*30:0] str;
  integer fd;

  initial begin
    a = 4'ha;
    b = 4'hb;
    c = 4'hc;
    d = 4'hd;

    fd = $fopen("my_file.txt", "w");
    $fdisplay(fd, a, b, c, d);
    $fclose(fd);
  end
endmodule
```

Formatting Data to String

The first argument in the *$sformat* system function is the variable name into which the result is placed.

The second argument is the *format_string*, which tells how the following arguments should be formatted into a string.

```
module tb;
    reg [8*19:0] str;
    reg [3:0] a, b;
    initial begin
        a = 4'hA;
        b = 4'hB;

        // Format 'a' and 'b' into a string given
        // by the format, and store into 'str' variable
        $sformat(str, "a=%0d b=0x%0h", a, b);
        $display("%0s", str);
    end
endmodule
```

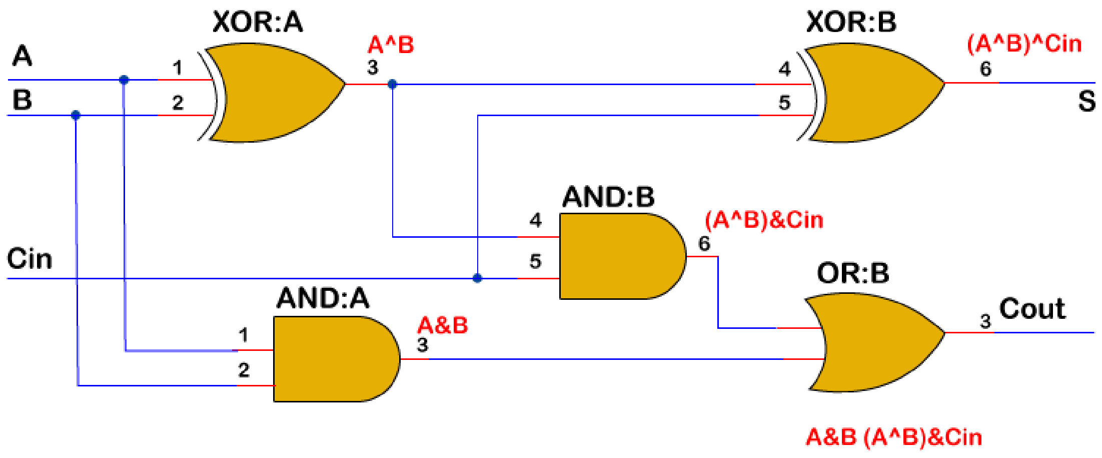
Verilog Full Adder

The full adder is a digital component that performs three numbers an implemented using the logic gates. It is the main component inside an ALU of a processor and is used to increment addresses, table indices, buffer pointers, and other places where addition is required.

A one-bit full adder adds three one-bit binary numbers, two input bits, one carry bit, and outputs a sum and a carry bit.

A full adder is formed by using two half adders and *ORing* their final outputs. A half adder adds two binary numbers. The full adder is a combinational circuit so that it can be modeled in Verilog language.

The logical expression for the two outputs *sum* and *carry* are given below. A, B are the input variables for two-bit binary numbers, Cin is the carry input, and Cout is the output variables for Sum and Carry.



Truth Table

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Example

An example of a 4-bit adder is shown below, which accepts two binary numbers through the signals a and b.

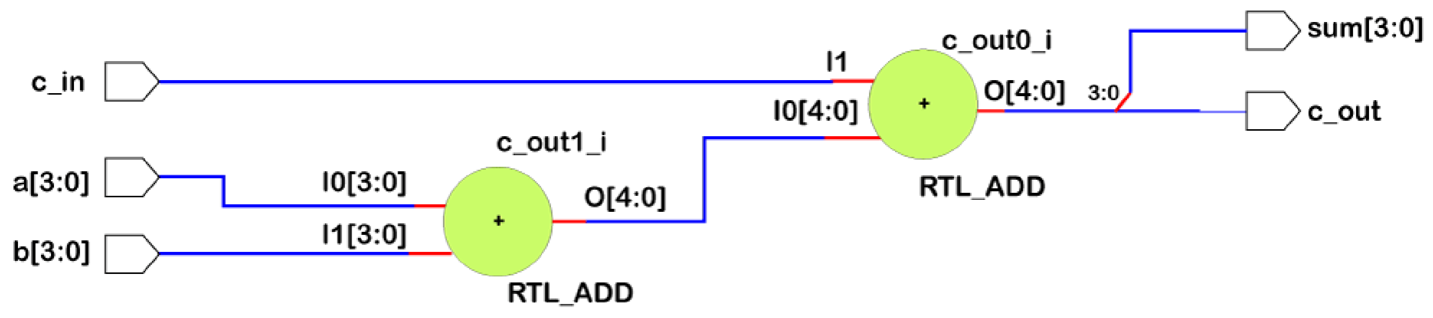
An adder is a combinational circuit. Therefore Verilog can model it using a continuous assignment with *assign* or an *always* block with a sensitivity list that comprises all inputs.

```
module fulladder (  input [3:0] a,
                    input [3:0] b,
                     input c_in,
                    output c_out,
                     output [3:0] sum);

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

Below code shows the uses an *always* block which gets executed whenever any of its inputs change value.

```
module fulladder (  input [3:0] a,
                input [3:0] b,
                 input c_in,
                output reg c_out,
                 output reg [3:0] sum);

   always @ (a or b or c_in) begin
   {c_out, sum} = a + b + c_in;
  end
endmodule
```

Hardware Schematic



Testbench

First, add the timescale directive. It starts with a grave accent ` but does not end with a semicolon. Timescale directive is used for specifying the unit of time used in further modules and the time resolution (one picosecond). The time resolution is the precision factor that determines the degree of accuracy of the time unit in the modules.

Next are the module and variable declaration.

- The register (reg) type holds the value until the next value is driven by the clock pulse onto it and is always under *initial* or *always* block. It is used to apply a stimulus to the input.
- Wires (wire) are declared for the passive variables. Their values don't change and can't be assigned them inside, always an initial block.

Then comes the module instantiation.

- The test bench applies stimulus to the Device Under Test (DUT). The DUT must be instantiated under the testbench. Port mapping is the linking of testbench's modules with that of the design modules.
- Now we'll give an initial stimulus to the input variables. This is done under the *initial* block.
- We can also stop the simulation in a pre-mentioned delay time using $finish.

The different thing is the use of two system tasks:

- *$dumpfile* is used to dump the changes in net and registers' values in a VCD file (value change dump file).
- *$dumpvars* is used to specify which variables should be dumped in the file name specified by the filename argument.
- Now, it depends on the user whether they want to display the simulation result on the TCL console or not. We used a *$monitor*, which displays the value of the signal whenever its value changes.
- It is executed inside *always* block, and the sensitivity list remains the same as explained in the above section.
- The format specifier *%t* gives us the current simulation time, and *%d* is used to display the value of the variable in decimal.

```
module tb_fulladd;
   // 1. Declare testbench variables
  reg [3:0] a;
  reg [3:0] b;
  reg c_in;
  wire [3:0] sum;
  integer i;

   // 2. Instantiate the design and connect to testbench variables
  fulladd  fa0 ( .a (a),
                 .b (b),
                .c_in (c_in),
                 .c_out (c_out),
                .sum (sum));

   // 3. Provide stimulus to test the design
  initial begin
    a <= 0;
    b <= 0;
    c_in <= 0;

    $monitor ("a=0x%0h b=0x%0h c_in=0x%0h c_out=0x%0h sum=0x%0h", a, b, c_in, c_out, sum);

     // Use a for loop to apply random values to the input
    for (i = 0; i < 5; i = i+1) begin
      #10 a <= $random;
          b <= $random;
            c_in <= $random;
    end
  end
endmodule
```

When a and b add up to give a number more than 4 bits wide, the sum rolls over to zero and c_out becomes 1. For example, the line highlighted in yellow adds up to give 0x11 and the lower 4 bits get assigned to sum and bit#4 to c_out.

Verilog Priority Encoder

An encoder is a combinational circuit. It has 2^n input lines and n output lines. It takes up these 2^n input data and encodes them into n-bit data. And it produces the binary code equivalent of the input line, which is active high.

But, a normal encoder has a problem. If there is more than one input line with logic value 1, it will encode the wrong output. It only works when only one of the inputs is high. It malfunctions in the case of multiple high inputs.

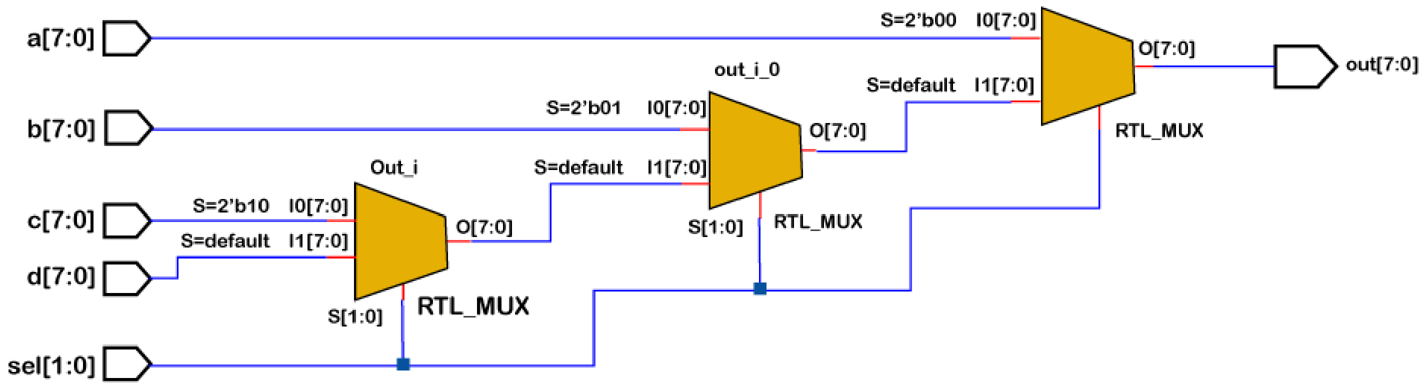
Thus, to solve the above disadvantage, we "prioritize" the level of each input. Hence, if multiple input lines are selected, the output code will correspond to the input with the highest designated priority. This type of encoder is called the *Priority Encoder*.

Example

```
module pr_en ( input [7:0] a,
           input [7:0] b,
            input [7:0] c,
           input [7:0] d,
            input [1:0] sel,
           output reg [7:0] out);

  always @ (a or b or c or d or sel) begin
```

```
        if (sel == 2'b00)
          out <= a;
        else if (sel == 2'b01)
          out <= b;
        else if (sel == 2'b10)
          out <= c;
        else
          out <= d;
    end
endmodule
```

Hardware Schematic



Testbench

A testbench is an HDL module used to test another module, called the *device under test (DUT)*. The test bench contains statements to apply inputs to the DUT and check that the correct outputs are produced.

The input and desired output patterns are called *test vectors*. Below is the test bench for the priority encoder:

```
module tb_4to1_mux;
  reg [7:0] a;
  reg [7:0] b;
  reg [7:0] c;
  reg [7:0] d;
  wire [7:0] out;
  reg [1:0] sel;
  integer i;

```

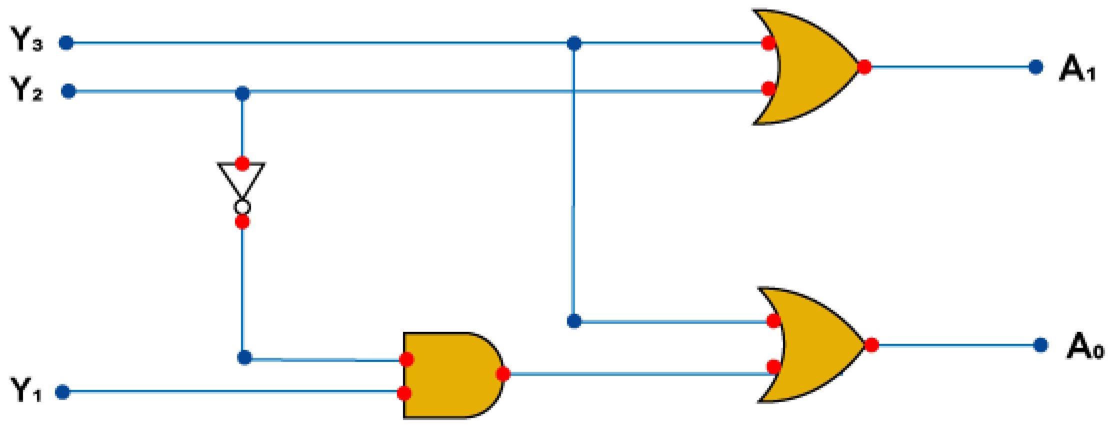
```
    pr_en    pr_en0 (   .a (a),
                        .b (b),
                        .c (c),
                        .d (d),
                        .sel (sel),
                        .out (out));

    initial begin
      sel <= 0;
      a <= $random;
      b <= $random;
      c <= $random;
      d <= $random;

      for (i = 1; i < 4; i=i+1) begin
         #5 sel <= i;
      end

      #5 $finish;
    end
  endmodule
```

Now, we will see how to design a 4:2 Priority Encoder using different modeling styles in Verilog.

1. Gate Level Modeling

This is virtually the lowest abstraction layer used by designers for implementing the lowest level modules, as the switch level modeling isn't that common. As the name suggests, gate-level modeling makes use of the gate primitives available in Verilog.

Below we are describing a Priority Encoder using Gate-Level modeling:



From the circuit, we can observe that one AND, two OR and one NOT gates are required for designing. Let's start coding.

Gate level Modeling for 4:2 priority encoder

As any Verilog code, we start by declaring the module and terminal ports.

- module priority_encoder_42(A0,A1,Y0,Y1,Y2,Y3);
- ....
- endmodule

Note that we declare outputs first followed by inputs as the built-in gates also follow the same pattern. Let's declare the input and output ports.

- input Y3, Y2, Y1, Y0;
- output A0, A1;

Now, we can declare the intermediate signals. These are signals that are not the terminal ports. From the above circuit, the signals from NOT and AND gates are treated as intermediate signals.

- wire y2bar; //not of y2
- wire and_out; // and of y2bar and y1

Now we define the logic gates. We use the gate (*<outputs>,<inputs>*) syntax to use the in-built gates in Verilog.

- not(y2bar, y2);
- and(and_out, y2bar, y1);
- or(A1, Y3, Y2);
- or(A0, and_out, Y3);

So, our final code looks like:

- module priority_encoder_42(A0,A1,Y0,Y1,Y2,Y3);
- input Y3, Y2, Y1, Y0;
- output A0, A1;
- wire y2bar; //not of y2
- wire and_out; // and of y2bar and y1
- not(y2bar, y2);
- and(and_out, y2bar, y1);
- or(A1, Y3, Y2);
- or(A0, and_out, Y3);
- endmodule

2. Dataflow Modeling

In this modeling technique, we use logic equations to describe data flow from input to output. We need not bother about the gates that make up the circuit.

Hence, it is much easier to construct complicated circuits using this abstraction level since there is no need to know the actual physical layout.

It uses the assign keyword to describe the circuit by using the logic equation.

The logic equation for the priority encoder is:

- A1 = Y3 + Y2
- A0 = Y3 + Y1Y2?

Dataflow modeling of 4:2 Priority Encoder

As always, we start with the module and port declarations:

- module priority_encoder_datafloe(A0,A1,Y0,Y1,Y2,Y3);
- input Y0,Y1,Y2,Y3;
- output A0,A1;

Now, we have to describe the flow of data to the outputs using *assign*.

- assign A1 = Y3 + Y2;
- assign A0 = Y3 + ((~Y2)&Y1);

Hence, our final code:

- module priority_encoder_datafloe(A0,A1,Y0,Y1,Y2,Y3);
- input Y0,Y1,Y2,Y3;
- output A0,A1;

-
- assign A1 = Y3 + Y2;
- assign A0 = Y3 + ((~Y2)&Y1);
- endmodule

3. Behavioral Modeling

Behavioral Modeling is the highest level of abstraction in Verilog HDL. We can describe the circuit by just knowing how it works. And we do not need to know the logic circuit or logic equation. A truth table is given below, such as:

Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

With this truth table, we can design our priority Encoder using Verilog.

Behavioral Modeling of 4:2 Priority Encoder

let's starts with the module and the port declaration.

- module priority_encoderbehave(A, Y);
- input [3:0]Y;
- output reg [1:0]A;

We have to mention output as *reg* in behavioral modeling. As we use *procedural assignments* in this modeling style, we have to ensure the outputs retain their value until the next value is given to them.

- always@(Y)
- begin
- ....
- end

What we have declared in brackets is the sensitivity list. Here, depending on the value of *Y*. The *always* keyword will make sure that the statements get executed every time the sensitivity list is triggered.

In between *begin* and *end*, we write the procedure for how the system works:

```
casex(Y)
4'b0001:A = 2'b00;
4'b001x:A = 2'b01;
4'b01xx:A = 2'b10;
4'b1xxx:A = 2'b11;
default:$display("Error!");
endcase
```

The *case* compares an expression to a series of cases and executes the statement or statement group associated with the first matching case. We have used *casex*, which is a special version of the *case*. This will treat the x and z values as don't cares.

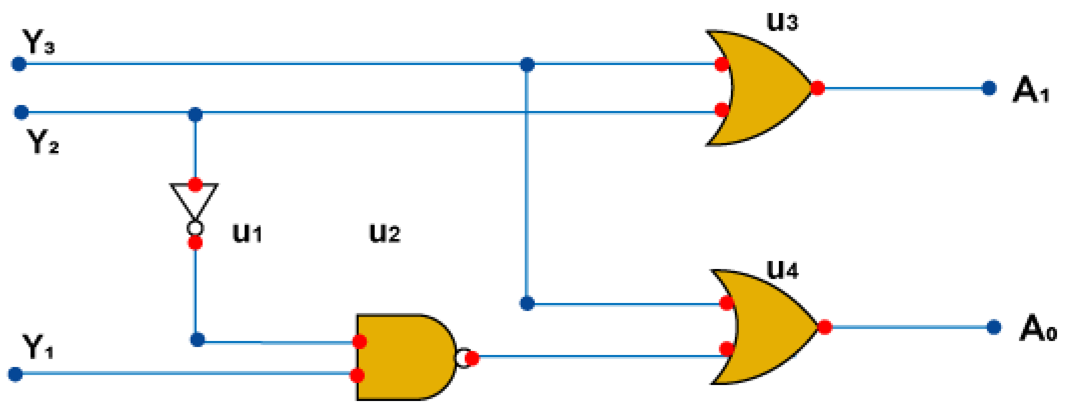
```
module priority_encoderbehave(A, Y);
input [3:0]Y;
output reg [1:0]A;
always@(Y)
begin
casex(Y)
4'b0001:A = 2'b00;
4'b001x:A = 2'b01;
4'b01xx:A = 2'b10;
4'b1xxx:A = 2'b11;
default:$display("Error!");
endcase
end
endmodule
```

4. Structural Modeling

Structural modeling describes the hardware structure of a digital system. It is somewhat similar to gate-level modeling. The only difference is it doesn't include any built-in gates. We create separate modules for each gate and then integrate to form the whole circuit.

Logic Circuit

In the case of 4:2 priority encoder, we require two OR, an AND and a NOT gates.



Structural Modeling of 4:2 Priority Encoder

To start with code, we will first structurize the OR gate.

We declare the *module* as or_gate. Then, we declare input and output ports

- module or_gate(c,a,b);
- input a,b
- output c;

Then, we use *assign* statement to write the logical expression for OR.

- assign c = a | b;

Thus our OR gate module will be:

- module or_gate(c,a,b);
- input a,b;
- output c;
- assign c = a | b;
- endmodule

Similarly, we do for AND gate:

- module and_gate(z,x,y);
- input x,y;
- output z;
- assign z = x&y;
- endmodule

And NOT gate:

- module not_gate(f,e);
- input e;
- output f;
- assign f = ~e;
- endmodule

Note: We keep variables for assigning inputs and outputs in one module different from others. It ensures mixing up of signals does not happen during a simulation.

Now we can proceed describing the Priority Encoder as the top-level module.

As usual, start with the module and port declarations.

- module priority_encoder_struct(A0,A1,Y0,Y1,Y2,Y3);
- input Y0, Y1, Y2, Y3;
- output A0,A1;

Now combine these individual modules for logic gates into one single module for the top module. It's done with the help of a module instantiation concept in which top modules are build using lower modules.

Using the logic circuit, we will instantiate the lower modules in this top using *instantiation by port name*.

- not_gate u1(.f(y2bar), .e(y2));
- and_gate u2(.z(w1), .x(y2bar), .y(y1));
- or_gate(.c(A1), .a(Y3), .b(Y2));
- or_gate(.c(A0), .a(Y1), .b(w1));

Hence, the Verilog code for the priority encoder in structural style is:

- module or_gate(c,a,b);
- input a,b;
- output c;
- assign c = a|b;
- endmodule
- module not_gate(f,e);
- input e;
- output f;
- assign f = ~e;
- endmodule
- module and_gate(z,x,y);
- input x,y;

- output z;
- assign z = x&y;
- endmodule
- module priority_encoder_struct(A0,A1,Y0,Y1,Y2,Y3);
- input Y0, Y1, Y2, Y3;
- output A0,A1;
- not_gate u1(.f(y2bar), .e(y2));
- and_gate u2(.z(w1), .x(y2bar), .y(y1));
- or_gate(.c(A1), .a(Y3), .b(Y2));
- or_gate(.c(A0), .a(Y1), .b(w1));
- endmodoule

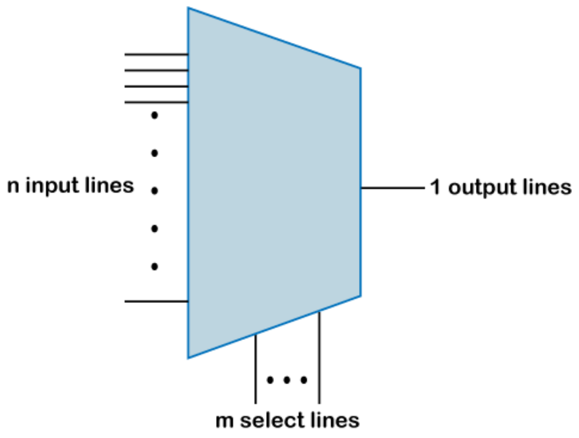
Verilog Multiplexer

A multiplexer is a device that selects one output from multiple inputs. It is also known as a data selector. We refer to a multiplexer with the terms *MUX* and *MPX*.

Multiplexers are used in communication systems to increase the amount of data sent over a network within a certain amount of time and bandwidth. It allows us to squeeze multiple data lines into one data line.

It switches between one of the many input lines and combines them one by one to the output. It decides which input line to switch using a control signal.

Physically, a multiplexer has n input pins, one output pin, and m control pins. n = 2^m. Since a multiplexer's job is to select one of the data input lines and send it to the output, it is also known as a *data selector*.



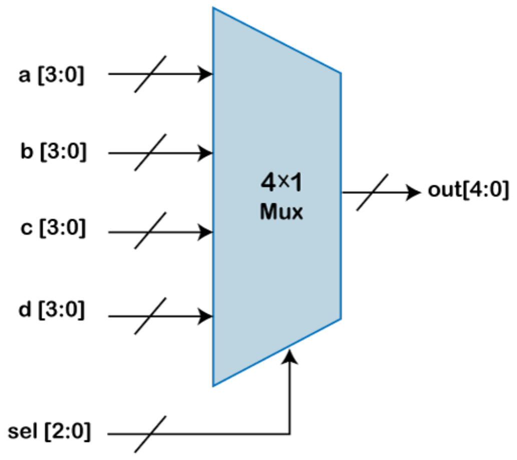
There are three main ways of constructing a multiplexer.

- Digital multiplexers are made up of logic gates.
- Analog multiplexers are made using transistors.

- Mechanical switches or rotary switches are made using rotating shafts.

The *MUX* itself acts like a digitally controlled multi-position switch where the binary code applied to the select inputs controls the data input, which will be switched to the output.

For example, it transfers data from one of the N inputs to the output based on the select signal. A 4-bit multiplexer would have N inputs each of 4 bits where each input can be transferred to the output by using a select signal.



Sel is a 2-bit input and can have four values. Each value on the select line will allow one of the inputs to be sent to the output.

A 4x1 multiplexer can be implemented in multiple ways and here we show two of the most common ways:

1. Using assign statement

```
module mux_4to1_assign ( input [3:0] a,           // 4-bit input called a
                         input [3:0] b,           // 4-bit input called b
                         input [3:0] c,           // 4-bit input called c
                         input [3:0] d,           // 4-bit input called d
                         input [1:0] sel,         // input sel used to select between a,b,c,d
                         output [3:0] out);       // 4-bit output based on input sel

  // When sel[1] is 0, (sel[0]? b:a) is selected and sel[1] is 1, (sel[0] ? d:c) is taken
  // If sel[0] is 0, a is sent to output, else b and if sel[0] is 0, c is sent to output, else d
  assign out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);

endmodule
```

The module called *mux_4x1_assign* has four 4-bit data inputs, one 2-bit select input and one 4-bit data output. The multiplexer will select either a, b, c, or d based on the select signal sel using the assign statement.

2. Using case statement

When we use case statement, then the signal *out* is declared as a *reg* type because it is used in a *procedural* block.

```
module mux_4to1_case ( input [3:0] a,            // 4-bit input called a
                       input [3:0] b,            // 4-bit input called b
                       input [3:0] c,            // 4-bit input called c
                       input [3:0] d,            // 4-bit input called d
                       input [1:0] sel,          // input sel used to select between a,b,c,d
                       output reg [3:0] out);    // 4-bit output based on input sel

  // This always block gets executed whenever a/b/c/d/sel changes value
  // When it happens, output is assigned to either a/b/c/d
  always @ (a or b or c or d or sel) begin
     case (sel)
       2'b00 : out <= a;
       2'b01 : out <= b;
       2'b10 : out <= c;
       2'b11 : out <= d;
     endcase
  end
endmodule
```

The module called *mux_4x1_case* has four 4-bit data inputs, one 2-bit select input and one 4-bit data output. The multiplexer will select either a, b, c, or d based on the select signal sel using the case statement.

Uses of Multiplexer

The multiplexer includes the following useful points, such as:

1. In a communication system where we have a communication network, a multiplexer increases the system's efficiency by allowing audio and video data transmission on a single channel.
2. In optical fiber communication, a multiplexer does the same job to combine multiple fiber cables onto one fiber cable using a technique called *Dense wavelength division multiplexing*.
3. In satellite communication, multiplexers transfer data from the satellite's computer system to the ground segment using *GSM* communication.

4. It also works as a parallel to serial data converter.
5. A computer decreases the number of copper lines necessary to connect the memory to other computer parts.

How to join multiplexers?

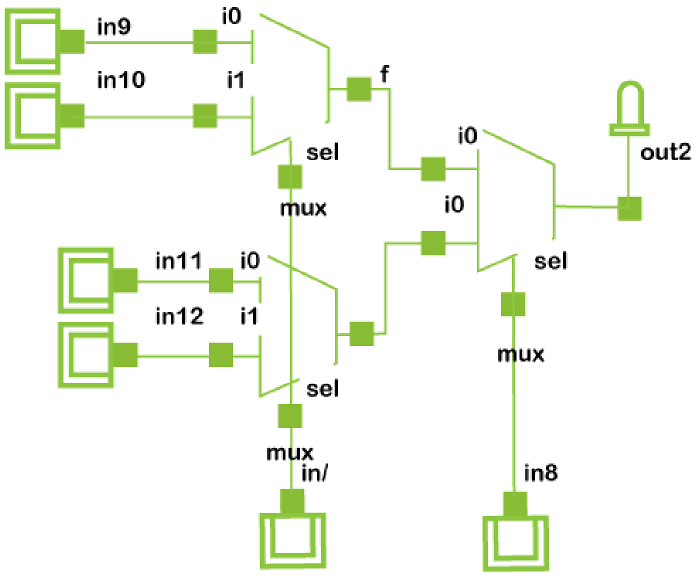
If we have small multiplexers, but we want to increase their functionality, we can join them to obtain a MUX with more inputs. The cascading of multiplexers is easy. Ensure that we connect to give the same number of inputs and control lines as the target MUX.

Let's make 4:1 MUX using 2:1 multiplexers. We know that a 2:1 MUX has two inputs and one select line. So joining two 2:1 multiplexers will give us four inputs and two select lines.

We can reduce the outputs to one, so we use another 2:1 MUX to combine two lines to get a single line.

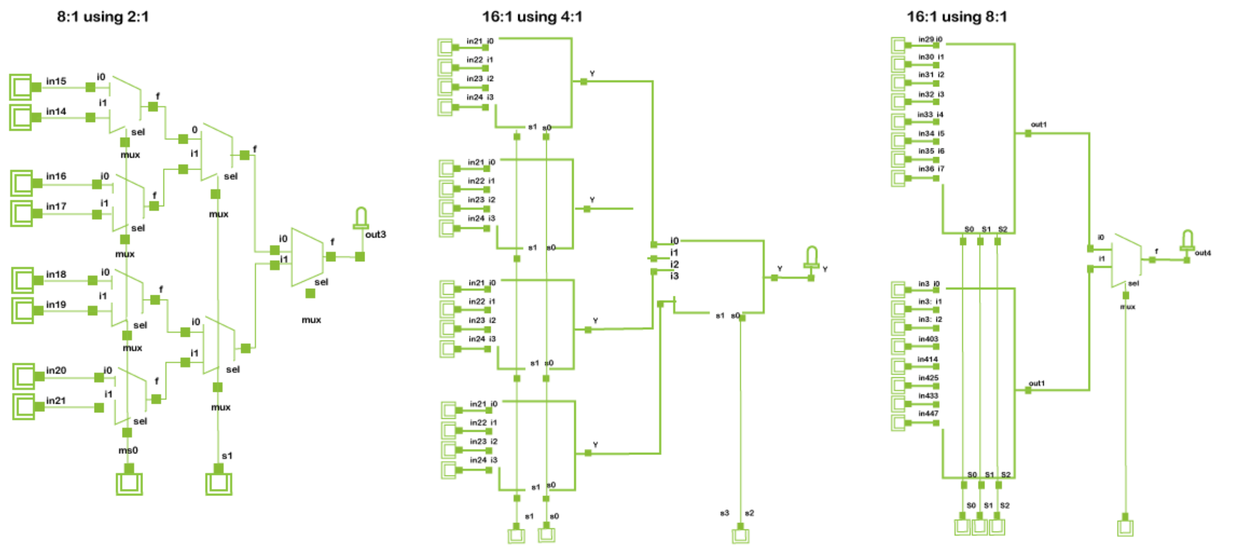
However, though that gives us the one output that we require, it gives us an additional select line. So now we have three select lines.

Now we reduce three select lines to two select lines. We can do that by joining two select lines. That would essentially reduce the two lines to one single line. The following image shows the result we get by applying our logic.

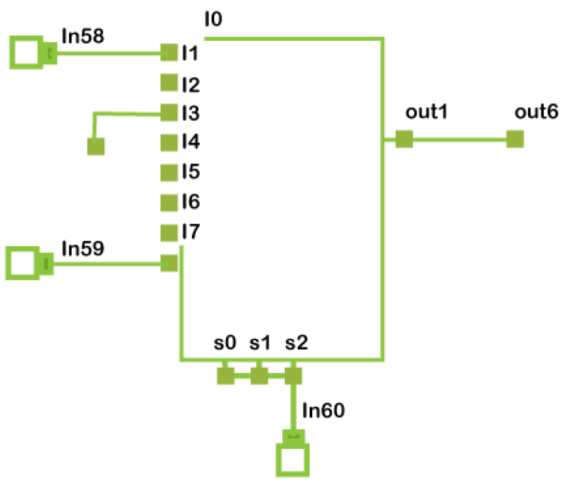


8:1 and 16:1 Multiplexers

Similar to the process we saw above, we can design an 8 to 1 multiplexer using 2:1 multiplexers, 16:1 MUX using 4:1 MUX, or 16:1 MUX using 8:1 multiplexer.



We can also go the opposite way and use a multiplexer with more inputs than required as a smaller MUX. Here's an 8:1 multiplexer being used as a 2:1 multiplexer.



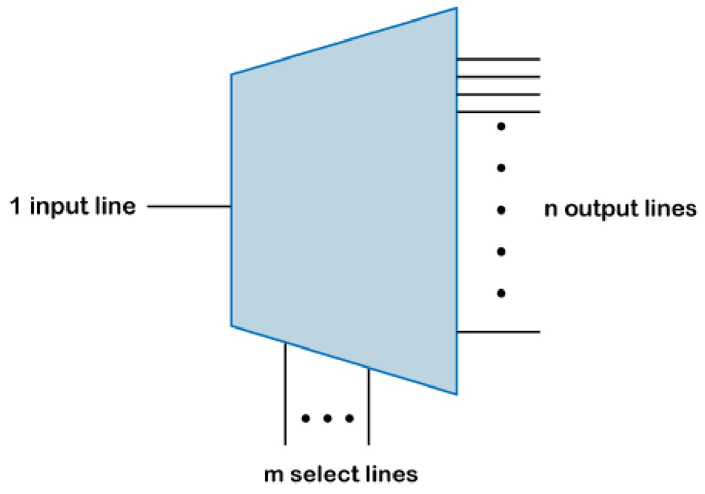
Demultiplexer

A demultiplexer is a combinational logic circuit that performs the opposite function as that of a multiplexer. A demultiplexer is alternatively referred to as a *demux*.

In a *demux*, we have n output lines, one input line, and m select lines. The relation between the number of output lines and the number of select lines is the same as we saw in a multiplexer. That is, 2^m = n. Depending on the value of the binary number formed by the select lines, any one of the output lines connects to the input line.

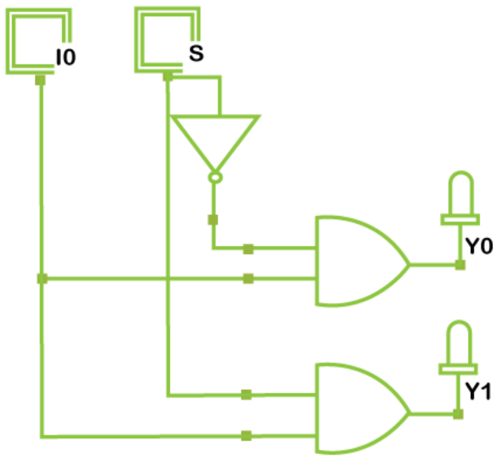
The rest of the output lines at this point go to an OFF state. That is, the value of the remaining lines is 0.

In this way, a demultiplexer converts serial data to parallel data and acts as a serial-parallel converter. Moreover, since it connects one data line to multiple data lines and switches between them, a demultiplexer is also known as a *data distributor*. The general symbol of a demultiplexer is shown in the following image.



How does a demultiplexer work?

To understand the working of a demultiplexer, we will straight away design one. The 1:2 demux is the simplest of all demultiplexers. We have one input, two outputs, and one select line (2^m = 2, therefore m=1).



As we can see, depending on the select line's value, one of the output connects to the input line. When S is 0, the first output line connects to the input. When S is 1, the second output line connects to the input.

In this way, a demultiplexer distributes data from one data line to multiple data lines.

Next, we will design a 1:4 demultiplexer. From the formula for select lines we saw above, a 1:4 demux will have two select lines. Let's draw the truth table for a 1:4 demux.

Truth Table

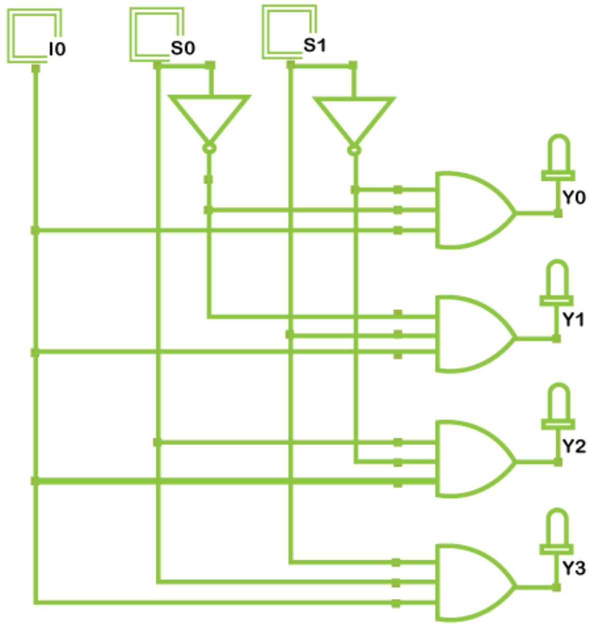
I0	S0	S1	Y0	Y1	Y2	Y3

I	0	0	I	0	0	0
I	0	1	0	I	0	0
I	1	0	0	0	I	0
I	1	1	0	0	0	I

As we can see, this truth table is shorter than the one for the 4:1 mux because instead of taking both the possible values of the input, we just took it as I. The resulting equations will be the same. So from the truth table, we get,

Y0 = S0'S1'
Y1 = S0'S1
Y2 = S0S1'
Y3 = S0S1

The resultant circuit for the above equations is shown below.



Uses of Demultiplexer

Here are some important uses of demultiplexer, such as:

1. A demultiplexer can receive serial data from a multiplexer present at the transmission end in a communication system. The *demux* then converts the data into its original form.
2. We can store an ALU's output in multiple memory registers using a demultiplexer.
3. The demultiplexer also acts as a serial to parallel converter.