INTRODUCTION

→ What is Verilog ?

* It is a Hardware Description language.
* It is used to design & describe digital system's such as ALU's, Timers, Counters & microprocessors.
* Designs which are described in HDL are independent of circuit fabrication technology.
* IEEE Standard HDL.
* Verilog is used for Both synthesis & simulation.
* The first HDL was ISP (In-System Programming), invented By C. Gordon Bell and Alan Newell
* System Verilog is an extended version of verilog

→ History :-

* It was designed & first implimented by Prabhu Goel & Phil Moorby at Gateway Design Automation in 1984-1985.
* 1989 - Cadence Design System purchased Gateway Automation.
* 1990 - Open Verilog International formed.
* 1995 - IEEE standard 1364 adopted.

* It is similar in Syntax to the C programming language.
* Verilog allows different levels of abstraction to be mixed in the same model.
* Verilog is case sensitive language & most of the syntax is adopted from C language
* Extension used By Verilog files is ".v"
* Verilog have some advance features like Timing checks, User Defined Primitive (UDP) & Programming language Interface (PLI).
* Verilog is synthesizable as it is concurrent language whereas JAVA, C are sequential language.

→ Comments :-

* There are two ways to provide comment
  - Single line comment : //
  - Block comment : /* ... */

Ex) // example of single line comment
/* example of block comment statement 1
Statement 2 */

→ Whitespace :-

* Blank space (\b), Tabs (\t) & Newlines (\n) constitute whitespace in Verilog.
* White spaces are ignored by verilog except when it separates two identifiers.
* Whitespace are not ignored in strings.

→ Design flow of IC :-

* Design Specification
↓
* Behavioral Description
↓
* RTL description (IP)
↓
* Functional Verification (VIP)

* Implementation
↑
* Layout Verification
↑
* PAR Floor planning
↑
* Gate level netlist

→ Lexical Elements :-

i) Identifiers
ii) Comments
iii) Data Types
iv) Keywords
v) Number Representation
vi) Operator's

i) Identifiers :-

* Identifiers are name given to objects that can be referred in a design
* An identifier can contain alphabets (a-z, A-Z), digits (0-1) and special characters (_, $).
* An identifier must begin with alphabet or underscore.
* Identifiers are case sensitive
* * There are identifiers which allow use of any printable ASCII character They are called as escaped identifier. (\@appl, \#appl).
* An escaped identifier starts with a \ and ends with a white space
* An identifier can't be a reserved keyword of verilog library.
* Always use lowercase letters & should be relevant to the object.

	Valid	Invalid
Eg)	integer my_var ;	integer $my_var ;
	integer my$_var ;	integer 5my_var ;
	integer my_57var ;	integer 78 ;

ii) Keywords :-

* In verilog few identifier are reserved to define language constructs, they are called as keywords.
* * All keywords are in lower case.

Eg)	always	default	include	wait
	assign	disable	initial	time
	and	edge	input	realtime
	automatic	for	integer	reg
	begin	forever	module	wire
	case	fork	output	while
	casex	function	signed	primitive
	casez	generate	unsigned	endprimitive

iii) Number Specification :-

* Verilog simulator by default treat the number in decimal format.
* The numbers can be represented in binary, octal & hexadecimal formats too.
* There are two types of number specifications:
  - Sized numbers
  - Unsized numbers

* Sized Number :-

  Syntax => <size>'<base><number>

  - Size : decimal value specifying number of bits to represent number.
  - Base : Base represents the format of the number. (d or D, h or H, o or O, b or B).
  - Number : Specifying number in chosen base format.

Eg) 4'b1011; a = 32'b19; Syntax error
4'd11;
4'hB;

* Unsized Number :-
  - Numbers are written without size specification
  - Default no. of bits depends upon the machine & simulator, must be at least 32. (a = 100; // a is 32'd100;)
  - Decimal is the default base format if not specified.

Eg) integer a = 2500 // a stores the decimal value = 2500
integer b = 'h ABCD // a takes 32 bit value = 32'h0000-ABCD.

* Negative Number :-
  - To represent a negative number, add minus sign "-" before the size specification.

- Negative numbers are stored in 2's compliment in memory.

Eg) -8'd 7; // 8 Bit negative number stored as 2's compliment of 7.

* If declared variable has larger size than specified value

a -> 8 Bit => a = 8'b1010;

0	0	0	0	1	0	1	0

* If declared variable has smaller size than specified value

a -> 4 Bit

a = 4'b1011_1100;

1	1	0	0

- (_) used for separating & simplicity which is ignored by the compiler.

* x or z values :-

- x is used to represent unknown values.
- z is used to represent high impedance.
- If the MSB Bit of the number is x or z, then x or z is padded respectively to fill the remaining most significant bits.
- If the MSB Bit of the number is 1 or 0, then 0 is padded to fill the remaining most significant bits.

* "_" or "?" :-

- _ is allowed anywhere in a number except at first character
- ? is an alternative of z in Verilog.

Eg) 12'h14x // 12 Bit hex no with 4 LSB Bits x

11'o z32 // 11 Bit oct no with 5 MSB Bits z

5'b110 // 5 Bit Binary no with 2 MSB Bits 0

16'b 1011_0100_1100_1010 // 16 Bit Binary no

10'h9?? // 10 Bit hex no with padded

1	0	0	1	z	z	z	z	z	z

iv) **Data Types :-** Mainly 2 types Nets & Registers.

* Verilog supports following data types :
  - Value set
  - Nets
  - Registers
  - Vectors
  - Integers
  - Real
  - Time
  - Arrays
  - Strings.

* **Value Set :-** 4 State
  - 0 - logic zero
  - 1 - logic One
  - x - Unknown logic value
  - z - High impedance value
  - x & z are case insensitive.

** **Nets :-**
- Generally used to create connection between multiple elements of a module or to provide connection between various instances in top module.
- In net type signals, there is not storage capacity & requires continuous driving.
- Nets generally declared as "wire", "wand", "wor", "supply 1", "supply 0" etc. (tri, trior, triand, tri0, tri1, trireg). (Unsigned)
- If there is no value on net then by default it will take z.

Eg)
```
wire temp1;          // 1 Bit net
wire [3:0] temp2;    // 4 Bit net
```

- Supply is used for strengthening in switch level modelling.
- Nets are generally inputs, outputs in case of gate level data flow & structural modelling.
- Except for the trireg, they don't store values.

Date :
Page No.: 07

** Register :-

- Used when storage is required i.e they can hold the value until value is replaced.
- Registers don't require a constant driver
- Default value is x.
- Reg type signal assigned only inside always or initial statements.

Ex) reg temp1; // 1-bit unsigned reg
reg [3:0] temp2; // 4-bit unsigned reg
reg signed [11:0] temp3; // 12-bit signed reg.

- reg, integer, real, time, realtime etc.
- Incase of behavioural model it is always output only.

* Vectors :-

- It is used to represent a group of bits.
- net or reg data type can be declared as vectors.

Ex) wire temp1; // Scalar type : net
wire [0:7] temp2; // 8 bit vector type : net
reg [3:0] temp3; // 4 bit vector type : reg

- Selecting part of vector
temp 2[5]; // Selecting 5th bit of temp2
temp 2[3:6]; // selecting 3rd to 6th bit of temp2
temp3 [0:2]; // Invalid because order is different wrt declaration.
- wire [3:0] temp 2; wire [0:3] temp 3;

3	2	1	0
z	z	z	z

0	1	2	3
z	z	z	z

** No. of can be accessed in the same order of there declaration

* We can make a vector of datatypes which are by default of one bit.

* Intiger :-

- A type of register used to store intiger values.
- The default size depends upon the host machine but should be atleast 32 bits.
- Default value : x
- It is used to store signed quantities.
- Syntax => intiger variable_name ;

Ex) intiger int1, int2 ; // signed int1, int2
intiger [3:0] int3 ; // invalid syntax, size not allowed (can't be vector).

* Real :-

- A type of register used to store real values
- Values can be stored in decimal or scientific notation.
- Default value : 0 (0.0)
- Syntax => real variable_name ;

Ex) real a, b ; // assigned inside initial or always block
a = 3.14 ; // Assign a floating value 3.14
b = 5e12 ; // Scientific notation (5×10^{12}).

* Time & Realtime :-

- Time :- A type of register used to store simulation time
- Size depends upon implementation but should be atleast 64 bits.
- Realtime :- Store the time as floating point quantity.
- $time is system function which returns current simulation time.
- Default value : 0
- Syntax : time variable_name ;

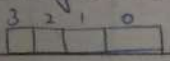
Ex) time a, b ; // time variable
a = $time ; // to be used inside the initial & always block.

★ Arrays :-

- Used to store the same type of data together
- Array declaration of reg or wire can be scalar of vector.
- A multidimensional array can be declared by mentioning the dimension ranges identifier name
- In verilog arrays are allowed for 'reg', 'wire', 'integer' & 'real' data type.
- Syntax => <data_type> array_name <array_size> ;

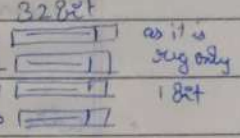
- Vector

reg [3:0] a ;



- array

reg a [3:0] =>



- Declaration =>

→ integer num [0:15] ; // num is scalar type integer array stores 16 values each 32 bit.

→ reg [7:0] data [31:0] ; // data is 8 bit vector of type reg size of array is 32

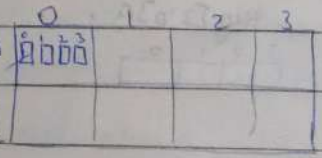
- integer a[10] ; Compact declaration (0 - - - 9)
- integer a[9:0] ; } Verbose declaration
  a[0:9] ; } Preferred

(rows column (RC))

→ wire temp [0:10] [0:5] ; // temp is 2-D net type array, row = 11 column = 6.

→ reg [3:0] mem [0:7] [0:7] ; // mem is 2-D reg type array. row = 8 (8 bits) col = 8, each 4 bit wide.

- Assigning & Accessing the array :-
  - → data[5]; // Accessing 8 bit data stored at index 5
  - → data[10][2]; // 2nd bit of data at location 10
  - → mem[3][2]; // Accessing the 3rd row & 2nd column data of mem
  - → num[3] = 45; // Assign integer value 45 at index = 3 of num
  - → num = 0; // Illegal - all elements can't be assigned without indexing
  - → data[20] = 8'hAA; // Assign hexadecimal value AA at index 20 of data.
  - → mem[3][3] = 4'b1011; // Assign 1011 at row = 3 & column 3 of mem
- wire arr[0:2][0:3][0:4]; (R, C, subC)



  - arr[0][0][0];

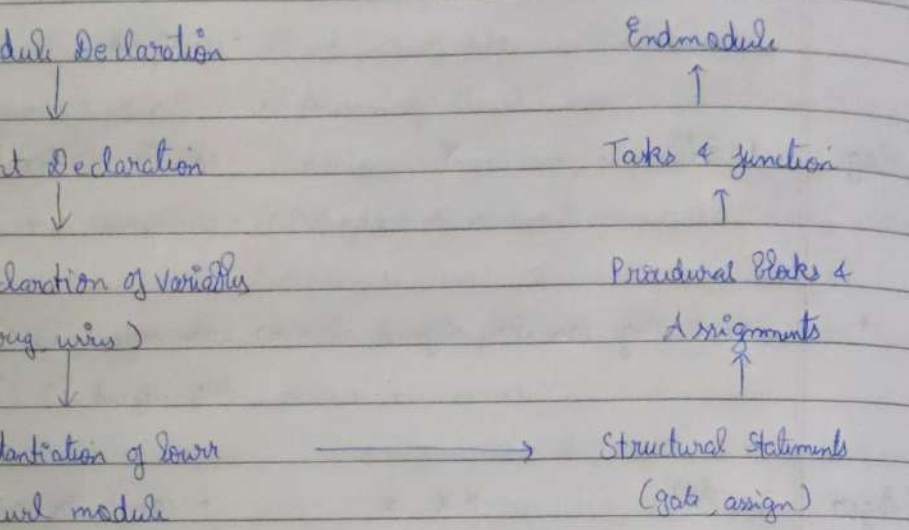
★ Strings :-

- reg data type is used to store string data
- Each character requires 8 bits for its storage
- If reg size is more than size of string, Blank space are padded on left.
- If reg size is less, the leftmost bits of strings are truncated.
- Syntax => reg [size of string] string_name ;
- Declaration =>
  - → reg [8*9:1] str1 = "FutureWiz"; // To store futurewiz 9 Bytes are required
    str1 = "FutureWiz"
  - → reg [8*3:1] str2 = "FutureWiz"; // str2 stores only 3 Bytes, others are truncated, str2 = "Wiz"
  - → reg [8*12:1] str3 = "FutureWiz"; // str3 can store 12 Bytes, extra Bytes are padded with Blank space/Zero
    // str3 = "__FutureWiz"

- 8 bit → ASCII value

Name	Size default	Default value	Note
Wire	1 - Bit	Z	• Can Be multi Bit or vector also • Unsigned , 4 - State
Reg	1 - Bit	X	• Can Be multi Bit or vector also • Unsigned , 4 - State
*	Wire & reg can Be signed also => wire signed [3:0] a ; reg signed [3:0] a ;		
Integer	32 Bit	XX...x	• Can't Be a vector • Signed , 4 State
Real	64 Bit	0 (%d) 0.0 (%f)	• Signed
time	64 Bit	xxxx...x	• Unsigned
real time	64 Bit	00....0	• Signed

⇒ Fundamental Blocks of Verilog :-



→ Module :-

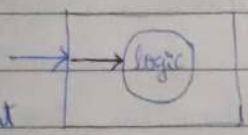
* A "module" is the basic building block in verilog
* A module can be an element or a collection of lower - level design blocks
* A module provide the necessary functionality to the higher - level block through its port interface.
* Module declaration comes on top of every verilog program
* Module is declared by the keyword module.
* A corresponding keyword endmodule must appear at the end of the module definition.
* Each module must have a module - name, which is the identifier for the module, and a port list, which describes the input & output terminals of the module
* Syntax =>
  - module module_name (port definition);
    // Body of module
    endmodule
  - module module_name;
    // Module may have empty port list
    endmodule

→ Ports :-

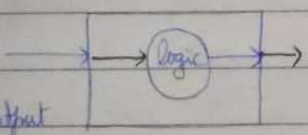
* Ports are the pins or signal which act as input or output of a specified module.
* Port list is an important component of verilog module.
* Ports provide a means for a module to communicate with the external world through input & output.
* There are 3 types of port available in Verilog :
  - input // module can only recieve info using its input ports
  - output // module can only send values using its output ports
  - inout // module can either send or recieve values using its inout ports.

** All ports declared as one of the above is assumed to be a wire by default, otherwise it is necessary to declare it again.

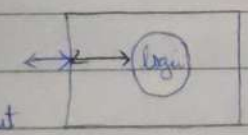
* Declaring multiple ports with same name is illigal.



- internally must always be type net, externally the inputs can be connected to variable reg or net type



- internally can be type net or reg, externally the output must be connected to a variable net type.



- inout port must be of type net both internally as well as externally.

* Syntax => <port-direction> <port-type> port_name ;

Ex) input a, b ;
output [2:0] y ;
output reg y ;
inout j ;

- input a; // wire By default
- output b; // wire By default
- output reg c; // output of type reg
- output integer e; // output of type integer
- input wire g; // input of type wire, default is wire
- input [31:0] h; // vector input, wire By default
- input integer i; // invalid as input must Be a net type
- reg for input & inout is not allowed.

↳ Alternative → input signed [31:0] i;

→ Parameters :-

* Parameter is Basically used for defining constants in writing code
* It can Be a Binary, integer, real or string.
* Syntax =>

Parameter variable_name = value;

* Example :

```
module adder(a,b,sum);
parameter N=4;
input [N-1:0] a;
input [N-1:0] b;
output [N-1:0] sum;
assign sum = a+b;
endmodule
```

* Parameter is a constant value which can't Be changed, But using "defparam" we can change the parameter value in the testbench. (overriding).
*

```
module adder #(Parameter WIDTH=8)(   //Parameterized module
input [WIDTH-1:0] a,b,
output [WIDTH-1:0]sum);
assign sum = a+b;
endmodule
```

Modelling Styles

=> Levels of Abstraction :-

* In verilog HDL a module can be defined using various levels of abstraction :
  i) Behavioral level
  ii) Register Transfer level (Data flow, Structural)
  iii) Gate level (Gate level simulation)
  iv) Switch level

ii) Gate level Modelling :-

* In this level of abstraction the system modeling is done at the gate level, i.e the properties of the gates
* For this style of modeling verilog provides in-built primitives.
* There are three classes for gate level modeling :-
  - logic
  - Buffer & Inverter
  - Tristate logic.

** In gate level modelling inputs & outputs are both net/wire type.

* Syntax => primitive_name instance_name (optional) [instance range]
  (output port(s), input port(s), control port);

- instance_range when we want to call the same primitive multiple times.

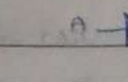
- logic :-
  - The first port is output, other ports are input.
  - These primitives has only one output & any number of inputs
  - The following gates are available :
    "and" "or" "nand" "nor" "xor" "xnor"
  - Max input values can go upto 16.
  - Syntax => primitive_name instance_name (output port, input1, input2 ...);
  - Ex => and (X, A, B) / and a1(X, A, B);
    or (X, A, B) / or o1(X, A, B);

A	B	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
0	x	0	x	1	x	x	x
0	z	0	x	1	x	x	x
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1
1	x	x	1	x	0	x	x
1	z	x	1	x	0	x	x
x	0	0	x	1	x	x	x
x	1	x	1	x	0	x	x
x	x	x	x	x	x	x	x
x	z	x	x	x	x	x	x
z	0	0	x	1	x	x	x
z	1	x	1	x	0	x	x
z	x	x	x	x	x	x	x
z	z	x	x	x	x	x	x

- **Buffer and Inverter :-**
  - The last port is input, other ports are output.
  - These primitives can have any number of o/p & only one i/p.
  - The following gates are available
    "buf" "not"
  - Ex :
    ```
    buf (o1, inp1);
    buf (o1, o2, inp1);
    not (o1, inp1);
    not (o2, o2, inp1);
    ```

A	Buf	not
0	0	1
1	1	0
x	x	x
z	x	x

- **Tristate logic :-**
  - There are buf/not gates with control input
  - The last port is control input, middle port is input & first port is output.
  - The following gates are available
    "bufif1" "notif1" "bufif0" "notif0"
  - Ex :
    ```
    bufif0 (op, inp, cntrl);
    notif0 (op, inp, cntrl);
    ```



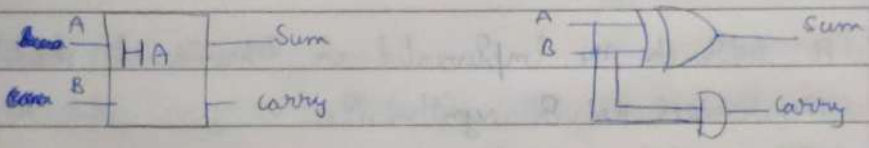
A	control	Y
0	0	0
1	0	1
x	1	z

★ Properties of In Built Primitives :

- These have one scalar output & multiple scalar inputs. The output of the gate is evaluated as soon as the input changes.
- Since they are predefined in verilog, they don't require module definition.
- Primitives are instantiated like module.
- If you are using the same primitive more than one time then label is must for each primitive for better readability.
- Array instantiation is also possible with primitives.

```
and a0 (out[0], inp1[0], inp2[0]);        // Allowed
and a1 (out[1], inp1[1], inp2[1]);        and a[1:0] (out, inp1, inp2);
```

—o Write a gate level code for Half adder



RTL :-

```
★ module HA(a, b, S, C);
    input a, b;
    output S, C;
    xor x1(S, a, b);
    and a1(C, a, b);
endmodule
```

Test Bench :-

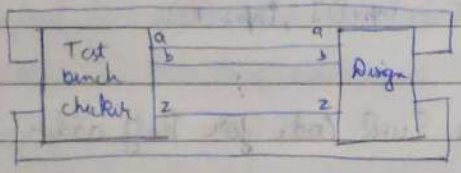
```
★. module HA_test;
  . reg a, b;
    wire S, C;
  . HA dut(a, b, S, C);
  . initial Begin
      a=0; b=0;
      #2 a=0; b=1;
      #2 a=1; b=0;
      #2 a=1; b=1;
    end
  . initial Begin
      $monitor("a=%b, b=%b, S
      =%b, C=%b", a, b, S, C);
    end
```

//VCD file

```
• initial Begin
    $dumpfile("dump.vcd");
    $dumpvars(0, a, b, S, C);
  end
• endmodule
```

⇒ Testbench :-

* Generating waveforms could be a complex task as, instead of testing few specific cases, we sometimes need to test a set of cases. In all such cases, we write stimulus files for our design known as test bench files.
* Used to verify the functionality of the design
* It involves four major steps i) Generating stimulous
  ii) Driving these values on DUT
  iii) Capturing or monitoring DUT o/p's.
  iv) Matching the expected with obtained results.



* All testbench are implemented in behavioural modelling & the testbench need not be synthesizable.
* Write module without ports.
- Inputs are reg & outputs are wire
- DUT instantiation
- Initial begin
  = { Generation of stimulus of different combination of DUT i/p's
- Delays if required
- end
- initial begin — capturing the results & printing it.
  $monitor ( );
  end
- initial begin — To print the waveforms
  $dumpfile ("dump.vcd");
  $dumpvars (0, list of variables); → level of hierarchy
  end
  VCD → Value change dump
  • whenever the entries of o/p's & i/p's are changed these details are kept into "VCD" file
- endmodule

→ module instantiation :-

* Complex systems can be designed by combining sub-systems.
* So the modules describing the sub system will be called by the top module.
* Two approaches :- i) Port Connection by ordered list (Positional mapping)
  ii) Port Connection by name.
* Syntax => module_name inst_name (Port mapping) ;
  (inst_name : mandatory)

i) Port Connection by ordered list :-

- We use ordered list inside the parent module to instantiate the lower modules.
- We must know the exact order of ports in lower module to make the right connections.

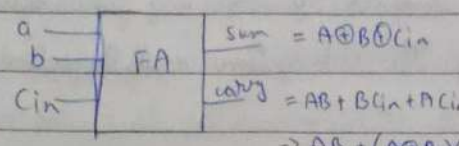
ii) Port Connection by name :-

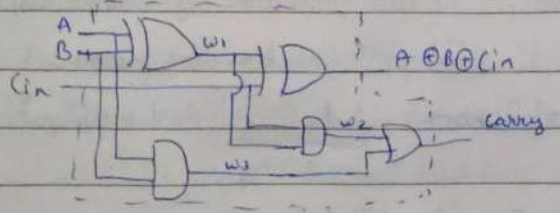
- A better way is to link by calling the names of each port one by one.
- In this approach we use the dot (.) symbol to indicate the port mapping.
- The parent port is written after the dot & then inside the brackets we write the lower module port name.

Ex) sub1 d0 (.a(in1), .b(in2), .c(out1));

* Ports that are not connected with any wire in instantiated module attains the High-impedance value.

→ Write a gate level code for Full adder





RTL :-

```
* module FA(a,b,cin,S,C);
  input a,b,cin;
  output S,C;
  wire w1,w2,w3;
  xor x1(w1,a,b);
  xor x2(S,w1,cin);
  and a1(w2,w1,cin);
  and a2(w3,a,b);
  or o1(C,w2,w3);
endmodule
```

Testbench :

```
* module FA_test;
  reg a,b,cin;
  wire S,C;
  FA dut(a,b,cin,S,C);
  initial begin
  a=0; b=0; cin=0;
  #2 a=0; b=0; cin=1;
  #2 a=0; b=1; cin=0;
  #2 a=0; b=1; cin=1;
  #2 a=1; b=0; cin=0;
  #2 a=1; b=0; cin=1;
  #2 a=1; b=1; cin=0;
  #2 a=1; b=1; cin=1;
  end
  initial begin
  $monitor("a=%b, b=%b, cin=%b,
  S=%b, C=%b", a,b,cin,S,C);
  end
  initial begin
  $dumpfile("dump.vcd");
  $dumpvars(0,a,b,cin,S,C);
  end
endmodule
```

iii) Register Transfer level Modelling :-

* It is a design abstraction that describes how data moves between registers and what operations are performed on that data.
* In this we have two types - Data flow modelling
  - Structural modelling.

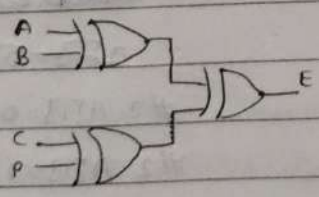
⇒ Data flow modelling :-

* Implementation of logic circuit is done with the help of operators.
* We use "assign" keyword i.e Continuous assignment.
* In data flow modelling inputs & outputs both are wire type
* Data flow modelling is used to implement combinational circuits.

→ Continuous Assignment :-

* Used to assign value to a net. (LHS)
* This assignment are always active, i.e any change on RHS side leads to assignment of the target. (Since no triggering required)
* An expression can be made up of a net, register or a function call(RHS)
* Target has to be of type net.
* Syntax => assign target = expression or const value ;

Eg => input a,b;
output c;
assign c = a & b;




```
* module my_design (A,B,C,P,E);
  input A,B,C,P;
  output E;
  assign E = ((A^B)^(C^P));
endmodule
```

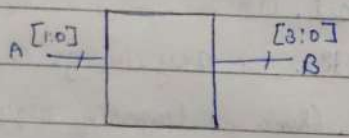
* How assignment operator works?
  - Every assignment operator works in two phase
    - Evaluation of RHS
    - Assign the evaluated value in LHS

* wire x, y, a;
  assign a = x & y;
  assign a = x | y;

  Multiple driving condition => Allowed when target is wire & not allowed for reg...

  x = 1 y = 0 | x = 0 y = 0
  o/p => x | o/p => 0

→ Write a data flow code for 2:4 decoder




```
* module Dec2to4(A,B);
  input [1:0]A;
  output [3:0]B;
  assign B[0] = (~A[1])&(~A[0]);
  assign B[1] = (~A[1])&(A[0]);
  assign B[2] = (A[1])&(~A[0]);
  assign B[3] = (A[1])&(A[0]);
endmodule
```

Testbench

```
* module dec_test;
  reg [1:0] A;
  wire [3:0] B;
  Dec2to4 d1(A,B);
  initial begin
  [redacted]
  A[1] = 0; A[0] = 0;
  #2 A[1] = 0; A[0] = 1;
  #2 A[1] = 1; A[0] = 0;
  #2 A[1] = 1; A[1] = 1;
  end
endmodule
```

OPERATORS

i) Arithmetic Operators
ii) Logical Operators
iii) Shift Operators
iv) Concatination Operators
v) Replication Operators
vi) Reduction Operator
vii) Conditional Operator
viii) Relational operator
ix) Equality Operator
x) Bitwise Operator

i) Arithmetic Operators :-

* Operator	Expression	Discription
+	a+b	Addition
-	a-b	Subtraction
*	a*b	Multiplication
/	a/b	Division
**	a**b	Exponent
%	a%b	Modulus

* Key Points :-

- If any operand has X or Z value then result of entire expression would be X.
- If the second operand of division or modulus is zero, the result will be X.
- If the second operand of exponent operator is zero, then result → 1.
- If either operand of exponent operator is real, then result will also be real.

* Modulus (Takes sign of numerator).

5%3 => 2
-5%3 => -2
5%-3 => 2
-5%-3 => -2

→ Arithmitic Operations:

* module arithop;
  reg [15:0] A,B;
initial Begin
  A = 8'd25;
  B = 8'd15;
$ display ("Addition of %d & %d = %d", A, B, A+B);
$display ("subtraction of %d & %d = %d", A, B, A-B);
$display ("multiplication of %d & %d = %d", A, B, A*B);
$display ("Division of %d & %d = %d", A, B, A/B);
$display ("Modulus of %d & %d = %d", A, B, A%B);
$display ("Square of %d = %d", A, A**2);
end
endmodule

O/P =>

Addition of 25 & 15 = 40
Subtraction of 25 & 15 = 10
multiplication of 25 & 15 = 375
Division of 25 & 15 = 1
Modulus of 25 & 15 = 10
Square of 25 = 625

* $monitor → will print when ever there is a change } printing task
  $display → will print only once

iii) Logical Operators :-

* They are Binary Operators

Operator	Expression	Description
!	!A	Logical NOT
&&	A && B	Logical AND
||	A || B	Logical OR

* Key Points :-

- It returns single bit 1, 0 or x.
- If operand is non zero, it is equivalent to logic 1.
- If operand is zero, it is equivalent to logic 0.
- If operand is neither zero nor non zero, it is equivalent to X.
- It is treated as false condition by simulators. (if(x) → false)

*

A	B	&&	||
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
0	x	0	x
x	0	0	x
1	x	x	1
x	1	x	1
x	x	x	x

! => 0 → 1
1 → 0
x → x

* • A = 4'd10 ; B = 4'd15;
A && B => 1
• A = 4'd0 ; B = 4'd15;
A && B => 0

*

Operand	logic 0/1/x ?
4'b1100	logic 1
4'bx100	logic 1
4'b0xz0	logic x
4'b0000	logic 0

• A = 4'dx ; B = 4'd15;
A && B => x
• A = 4'd10; B = 4'd15;
A || B = 1
• A = 4'd0; B = 4'd15;
A || B = 1
• A = 4'dx ; B = 4'd15;
A || B = 1
• A = 4'd10; B = 4'd0;
!A = 0 !B = 1

iii) Shift Operators:-

* They are Binary Operators (means 2 operands are needed).

* Operator	Expression	Description
>>	A >> B	Logical Right Shift
<<	A << B	Logical left Shift
>>>	A >>> B	Arithmetic Right Shift
<<<	A <<< B	Arithmetic left Shift

* Key Points

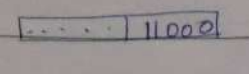
- Logical shift, operand 1 must be unsigned.
- Arithmetic shift, operands must be signed
- In logical shift, the vacant position is filled by zero.
- Arithmetic right shift pads MSB bits
- Arithmetic left shift works same as logical left shift.
- Shift by 1'b x or 1'b z result is X. (operand 2)

* • A = 8'b0000_0001
A << 1 => 00000010
• A << 3 => 00001000

• A = 8'b1000_0000 (it should be signed)
A >>> 1 => 11000000
A >>> 2 => 11100000
A <<< 1 => 00000000

* module tb;
reg [3:0] a;
integer b;
initial begin
a = 4'b0011;
b = 3 >>> a; // Right shift
b = 3 <<< a; // Left shift
end

• 32 bit & decimal



iv) Concatination Operators :-

* Concatination operators can accept any number of operands
* This operation provides mechanism to append multiple operands.
* Syntax => { op1, op2, ..., opn };

* Example :

```
module concat;
  reg [3:0] A,B,C;
  initial begin
    A=4'b0001; B=4'b1001; C=4'b1111;
    $display ("%b", {A,B});          // Result => 00011001
    $display ("%b", {A,B,C});        // Result => 000110011111
    $display ("%b", {A,4'b0011,B[2],C}); // Result => 000100111 0 1111
  end
endmodule
```

* { 4'b0100, 1, B, A[3], B[3:2] }
  (32 bit default)
  0100_1_1001_0_10

v) Replication Operator :-

* It can accept any number of operands
* It is used to replicate an operand specified no. of times.
* Syntax => { replicate_time { operand } };
* Example :
  {2{A}} => 0001_0001
  {A[0], {2{B}}} => 1_1001_1001
  {A, 4'b0011, {2{C}}} 0001_0011_1111_1111

vi) Reduction Operator :-

* They are unary operators
* It performs bitwise operation on a single vector
* Reduction operators return 1 bit result.

Operator	Expression	Description	O/P when A = 0001
&	&A	Reduct^n AND	0
~&	~&A	Reduct^n NAND	1
|	|A	Reduct^n OR	1
~|	~|A	Reduct^n NOR	0
^	^A	Reduct^n XOR	1
~^ or ^~	~^A	Reduct^n XNOR	0

vii) Conditional Operators :-

* They are ternary operators
* If condition evaluate to true, then true_expression is returned else false_expression is returned.
* Nesting of conditional operator is also allowed.
* Syntax => condition ? true_exp : false_exp ;
* Example

```
assign y2 = (sel==2) ? a : b;        //if sel = 2, the return a, else b
assign y1 = sel ? I1 : I0;           //2:1 mux using ternary operator
```

- If sel = x/z

  y1 = x

```
assign y3 = sel1 ? (sel0 ? I3 : I2) : (sel0 ? I1 : I0);    //4:1 mux
```

viii) Relational Operator :-

* They are Binary Operators.
* It returns logic 1, if expression is true & return logic 0 if expression is false.
* It return X, if any operand contains X or Z.

* Operator	Expression	Description	o/p
>	A(1001) > B(1100)	Greater than	0
<	A(1001) < B(1100)	Less than	1
>=	A(1100) >= B(1100)	Greater than Equal	1
<=	A(100X) <= B(1100)	Less than Equal	X

ix) Equality Operator :-

* They are Binary Operator
** Logical equality (==, !=) returns 0, 1 or X
** Case equality (===, !==) returns 0 or 1
* Logical equality returns X if either operands has X or Z data.
* Case equality compares all bits including X & Z.
* Verilog provide following equality operators.

Operator	Expression	Description	o/p A=1001, B=1100	o/p A=100X, B=100X
==	A == B	Equality	0	X
!=	A != B	Inequality	1	X
===	A === B	Case equality	0	1
!==	A !== B	Case Inequality	1	0

x) Bitwise Operator:

* They are binary operators
* It performs bit by bit operation on two operands
* If one operand is shorter than other, zeros are padded to match the length of larger operand.
* Return size is equal to that of larger operand.

Operator	Expression	Description	A=1001, B=1100	A=100x B=100x
~	~A	Bitwise Negate	0110	011x
&	A&B	Bitwise AND	1000	100x
|	A|B	Bitwise OR	1101	100x
^	A^B	Bitwise XOR	0101	000x
~^ or ^~	A~^B	Bitwise XNOR	1010	111x

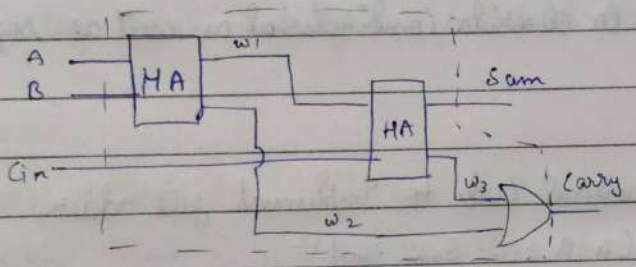
* In XOR if either of the i/p is x/z then o/p is x.

* A = 1011
  - A && 1'b0 => 0
  - A & 1'b0 => 0000
  - &A => 0

⇒ Hierarchy in Verilog / Structural Modelling :

* Verilog provides a concept of code reusability
* This reusability concept is implemented by taking instances of one code & use that instance in another code.
* With the help of this reusability concept a hierarchical structure can be created either top-to-Bottom or Bottom-to-top.
* This reusability concept reduced the length of design also provides the modularity in code.
* This kind of modelling is known as Structural style of modelling.

-o Write Structural code for Full adder using Half adder




```
* module HA(a,b,S,C);
  input a,b;
  output S,C;
  assign S = a^b;
  assign C = a&b;
  endmodule
```

```
* module FA(a,b,Cin,S,C);
  input a,b,Cin;
  output S,C;
  wire w1,w2,w3;
  HA h1(.a(a),.b(b),.S(w1),.C(w2));
  HA h2(.a(w1),.b(Cin),.S(S),.C(w3));
  assign C = w2|w3;
  endmodule
```

⇒ Behavioral Modelling :-

* It defines the behaviour of digital circuit. It doesn't describe how the circuit will be implemented into actual hardware.
* This type of modelling is like writing a algorithm.
* Describes the behaviour at higher level.
* It uses higher level constructs such as : always, for, while, function, task etc.
* Mostly used in RTL verification or writing a testbench.
* As the code written inside the procedural block are executed sequentially, hence avoided in designs where speed is one of the factor.
* Can be used to describe combinational as well as sequential logic circuits.

→ Write a Behavioral code to implement full adder

```
module FA(A,B,Cin, Sum, Cout);
   input A, B, Cin;
   output Sum, Cout;
   reg Sum, Cout;
  always @(*) begin
    {Cout, Sum} = A + B + Cin;
  end
endmodule
```

=> Procedural Blocks :-

* It represent that section of verilog where statements are executed in a sequential manner.
* There can be multiple procedure blocks, each block starts at 0 simulation time.
* In this blocks output must be of type reg.
* Nesting of procedural blocks is not allowed.
* Code inside the procedural block is executed continuously unless its stopped by certain timing control (#, @ or wait).
* Assignment statment inside procedural blocks are : Blocking or Non Blocking.
* Verilog offers two types of procedural statments :- initial block
  - Always block.

-> Initial Block :-

* Initial block is used for one time execution during the whole simulation.
* There may be any numbers of initial block inside a module.
* If there are more than one initial block, every block starts to execute at 0 simulation time.
* If there are multiple statments within an initial block, they must be grouped using begin & end keyword.
* Initial block are typically used for initialization, monitoring, providing test inputs & other operations which must be executed only once during the entire simulation.
* Syntax => initial begin
  // procedural assign
  // delay statment ;
  end

* Begin & end required only if there are more than one statement inside the initial Block.
* Timing control is used to provide delay in execution of assignment by specified time.
* '#' followed by time value to provide delay.
** initial Block is non-synthesizable.

* module gate (input a,b, output reg y1, y2);
  initial Begin
    y1 = a & b; } → Both assignments will execute
    y2 = a | b; } at '0' simulation time.
  end
endmodule

* module tb;
  reg b, c, out;
  reg a = 1; // reg can be initialized during declaration.
  initial Begin
  a = 0; b = 0; // a & b will get value '0' at 0ns
  #5 b = 1; // b will get value '1' at 5ns
  #13 a = 1; // a will get value '1' at 18ns
  end
endmodule

⇒ Always Block :-

* Always Block execute continuously during simulation just like a looping fashion.
* This statement is used to model Block of hardware that requires continuous execution.
* If there are multiple always Blocks, then all Blocks start at 0 time & executes simultaneously.
* If there are multiple statements within an always Block, they should be grouped using begin and end keyword.
* Syntax => always @ (sensitivity list)

```
begin
    // timing control
    // procedural assignment
end
```

- always Block can be controlled with help of sensitivity list
- If sensitivity list is not provided, then always Block runs continuously.
- always Block is executed whenever event occurs on any of the variable present in sensitivity list.
- @ is event control directive :- which holds the execution of statement until the event occur on any of the variable listed in sensitivity list.
- begin & end required only if there are more than one statement.

* Statement inside the always Block are executed in sequential manner.
* Timing control is used to provide delay in the execution of a assignment by specified time.
* Always Block is synthesizable provided the coding guidelines for * synthesis are applied.
* always Block can be used for both combinational & sequential logic design.

- always @(a,b)
  begin
  ≡
  end
  #2 a=0; b=0; execute
  #2 a=0; b=1; execute

- always @(a)
  begin
  ≡
  end
  #2 a=0; b=0; execute
  #2 a=0; b=1; Not execute (no change)

→ Always Block for Combinational circuit :-

* As in combinational circuits, the output depends only on the input signals, hence all the input signals must be provided in the sensitivity list.
* Hence, the statements written inside the always block will be executed when event occur on the any signal present in sensitivity list.
* reg variable is used inside the always block (target).
* Primarily Blocking assignments ("=") are used.

Ex) module xor (out, a, b);
input a, b;
output reg out;
always @(a,b) // always @(a or b) // always @(*) (not used in program)
↳ includes all the inputs
out = a^b;
endmodule

→ Always Block for Sequential circuits :-

* To infer synchronous logic in RTL coding, non Blocking assignments (<=) are primarily used.

Ex) module dff (input clk, d, output reg q);
always @(posedge / negedge clk) begin
q <= d;
end
endmodule

→ Key Points of always & initial Block :-

* Both always & initial Block are procedural Blocks
* All the statements inside these Blocks are executed sequentially
* If more than one statements, then they must be grouped using Begin and end.
* Multiple number of always & initial Blocks are allowed in one module
* * If module has both initial & always Block, then execution of initial Block will start before always Block.
* Initial Block executes only once at t=0, while always Block execution starts at t=0 and repeats its execution thereafter.
* Initial Block is not synthesizable, hence used for testbench or simulation not for designs.
* Always Block is synthesizable provided the RTL guidelines for synthesizable code is incorporated.
* always @ (posedge clk or rst) logically incorrect.

→ Write a code which makes initial Block work or function like a always Block & vice versa.

```
• module initial_as_always;
    reg clk = 0;
    initial begin
    forever #5 clk = ~clk;
    end
endmodule
```

```
• module always_as_initial;
    reg done = 0;
    always begin
      if (!done) begin
      $display ("RT=%0t", $time);
      done = 1;
    end
    disable always_as_initial;
  end
endmodule
```

* This runs only once by using disable, acting like initial.

=> Clock Generator :-

* Write verilog module to generate clock.

```
module clock_gn;
 reg clk;
 initial
  clk = 1'b0;
 always
  #5 clk = ~clk;
endmodule
```

(Continuous running)

=>

```
module clock_gn;
 reg clk;
 initial begin
  clk = 0;
  #100 $finish;
 end
 always
  #5 clk = ~clk;
endmodule
```

* Write Verilog module to generate clock of specified frequency

```
module clock_gn;
 reg clk;
 real freq = 100;   // freq in KHz
 real tp = 1/(freq*1e3)*1e9;  // Convert in ns
initial begin
 clk = 1'b0;
 #(tp*4) $finish;
 end
always begin
 #(tp/2) clk = ~clk;
end
endmodule
```

* Write Verilog module to generate clock of specified freq & Duty Cycle

```
module clock_gen;
 reg clk;
 real freq = 100;  // freq in KHz
 real tp = 1/(freq*1e3)*1e9;  // Convert to ns
 real duty_cycle = 50;  // In percentage
 real on_time = duty_cycle * tp/100;
initial begin
 clk = 1'b0;  // Initialize the clock with 0
 #(tp*4) $finish
end
always begin
 #(tp - on_time) clk = 1'b1;
 #(on_time) clk = 1'b0;
end
endmodule
```

Order of Execution

* We have 5 regions
  i) Prepond region :- Initialization
  ii) Active region
  iii) Inactive region
  iv) NBA region :- LHS updation of NBA.
  v) Postpond region :- $monitor & $strobe

ii) Active Region :- • Blocking assignments
  • Continuous assignments
  • RHS of Non Blocking assignments
  • $display & $write

iii) Inactive Region :- • LHS update for NBA
  • #0 Blocking Assignments
    a = b ; executed first
    #0 a = b ; This later

System Tasks

* In verilog HDL some system task are available by which we can perform specific task & make any simulation more understandable.
* By using system task one can perform following task :
  • Displaying & monitoring of input & output
  • Control of simulation
  • Timing checks (setup & hold violation)
* Name of system task begins with "$" followed by name of the task
* All system task are not synthesizable
* System tasks can be used in code as well as in test Bench.
* They are always used in procedural Blocks

→ Display System tasks :-

* The display system tasks are primarily used to display the information or debug messages to track the simulation flow.
* Verilog provides following display system tasks:
  - $display
  - $write
  - $monitor
  - $strobe

* $display :-

- Display strings, variables, & expressions immediately & append newline at the end of the message.
* Syntax is like printf in C.
* $display inserts a newline by default.
& $display ("Helloworld");
  $display ("value of count is %d", counter);
  $display (a,b,c);
  $display ($time); // print current time
- $display uses various format specifiers to display the values
  - %d or %D - Print value in decimal
  - %b or %B - Print value in Binary
  - %h or %H - Print value in Hexadecimal
  - %o or %O - Print value in Octal
  - %c or %C - Print's ASCII character
  - %s or %S - Prints String
  - %f or %F - Print's Real number in decimal format
  - %t or %T - Prints current simulation time

* $write :-

- Display strings, variables, & expressions without appending the newline at the end of the message.
- It's functionality is similar to $display except it doesn't add a new line automatically.

Ex>
```
module display_test;
  reg [3:0] a;
  initial begin
  a = 4'b1010;
  $display("display task");
  $display("Count value = %b", a);
  $write("write task");
  $write("Count value = %b", a);
  end
endmodule
```

O/P =>
display task
Count value = 1010

O/P =>
write task Count value = 1010

* $monitor :-

- $monitor continuously monitors list of variables & executes whenever any of the arguments changes it's value.
- $monitor automatically adds a new line.
- Only one $monitor can be active at a time.
- If there are multiple $monitor in the code, the new $monitor will override any previous $monitor in effect.
- "$monitor on" & "$monitor off" are used to enable & disable monitoring respectively.
- Monitoring is by default ON
- Syntax => $monitor("values are a = %d & b = %d", a, b);

Ex
```
module monitor_test;
reg [3:0] a,b;
initial begin
a = 4'hA;
b = 4'h9;
$monitor ("Sim Time = %t, a = %h, b = %h", $time, a, b);
#5 a = 4'h1; b = 4'h1;
#2 a = 4'h2; b = 4'h2;
end
endmodule
```

** // $monitor ("ST = %t, a = %h", $time, a);
// $monitor ("ST = %t, b = %h", $time, b);

Output =>
ST = 0, b = 9
ST = 5, b = 1
ST = 7, b = 2

O/P => Sim Time = 0, a = A, b = 9
Sim Time = 5, a = 1, b = 1
Sim Time = 7, a = 2, b = 2

★ $Strobe :-

- It is similar to $display
- It automatically adds a new line
- Displays data at the end of current simulation time
- Syntax => $ strobe (" strobe task - a = %b b = % b", a, b);

Ex>
```
module strobe_test;
reg [3:0] a,b;
initial begin
a = 4'hA; b = 4'h9;
#5 a = 4'h1;
$ display (" Display : Sim Time = %t, a = %h b = %h", $time, a, b);
$strobe (" Strobe : Sim Time = %t, a = %h b = %h", $time, a, b);
b = 4'h1;
end
endmodule
```

O/P =>
Display : Sim Time = 5, a = 1, b = 9
Strobe : Sim Time = 5, a = 1, b = 1

```
module tb;
reg [2:0] a=3'b000, b=3'b000;
initial begin
a=3'b101;
$display("display1: Sim time=%0t, a=%0b, b=%0b", $time, a, b);
$strobe("strobe1: Sim time=%0t, a=%0b, b=%0b", $time, a, b);
b=3'b011; #10 $strobe("strobe2: ...."); b=3'b100;
$display("display2: Sim time=%0t, a=%0b b=%0b", $time, a, b);
a=3'b111;
end
endmodule
```

O/P =>

display1 : Sim time = 0, a = 101 b = 000
strobe1 : Sim time = 0, a = 101 b = 011
display2 : Sim time = 10, a = 101, b = 100
strobe2 : Sim time = 10, a = 111 b = 100

→ Time System task :-

* Verilog offers following functions to access current simulation time
  - $time
  - $realtime
  - $stime

- $time - Returns 64 bit integer representing the current simulation time in terms of timescale unit.
- $stime - Returns 32-bit Unsigned integer representing the current simulation time in terms of timescale unit.
- $realtime - Returns real number representing the current simulation time in terms of timescale unit.

→ Simulation control tasks :-

* Verilog provides following simulation control system tasks.
  - $Stop - is used to suspend simulation
  - $finish - is used to exit simulation
  - $reset - is used to reset simulation to time 0.

Syntax	Description
$Stop or $Stop(N);	N=0 Prints nothing
$finish or $finish(N);	N=1 Prints simulation time & location
$reset	N=2 Prints simulation time, location, memory & CPU stats used during simulation.

- Default N = 1

Blocking & Non Blocking Assignments

→ Continuous Assignment :-

- Continuous assignment are used to assign values to a net
- Any change on RHS side leads to assignment of the target
- An expression can be made up of a net, register or a function call
- Target has to be of net type.
- Syntax ⇒ assign target = expression;

Eg) module my_design;

```
input a,b;          // a & b are of type net (default).
output c;           // c: net type
assign c= a & b;    // Continuous assignment
endmodule
```

→ Procedural Assignments :-

- Procedural assignments are used for assigning the values to reg, integer, real, or time variable in a sequential manner.
- The assignments are always used inside the procedures such as: initial, always, tasks & functions.
- Output must be of reg type
- The output will hold the value until next assignment to same variable
- In verilog there are two types of procedural assignments available
  i) Blocking (=)
  ii) Non Blocking (<=)

i) Blocking :- (=)

- Blocking statements are basically preferred for combinational design.
- During simulation it blocks the execution of statements until the assignment occurs.
- Blocking assignments are executed one after other in procedural block.
- In blocking statements, assignments occur at the very same time.

- Race condition may occur.
- Syntax => register_type variable = expression;

Ex) module Block_test;
reg [1:0] a=2'b01, b=2'b10, c=2'b11;
initial Begin
a=2'b00; b=a; c=b;
$display ("a=%b\n, b=%b\n, c=%b\n", a, b, c);
end
endmodule

o/p =>
a=00
b=00
c=00

iii) Non-Blocking Assignments: (<=)

- Non-Blocking statements doesn't Blocks the execution of other statements
- In case of non-Blocking, the assignment is schduled to occur at the end of current simulation time or at the end of procedural Block
- Verilog recommends to use non-Blocking statements to model a sequential circuit.
- Syntax => register_type variable <= expression;

Ex)
,,
a<=2'b00; b<=a; c<=b;
,,

o/p
a=01
b=10
c=11

Q) module ex1;
integer a;
initial Begin
#5 a=6; // #5 a<=6;
$Strobe ($time, "strobe", a);
$display ($time, "display", a);
a=12; // a<=12;
end
endmodule

o/p => (=)
5 display 6
5 Strobe 12

o/p => (<=)
5 display x
5 Strobe 12

Q) module ex2;
integer a,b,c,d;
initial begin
a=6; $display ($time,a,b,c,d); //a<=6;
b=12; $display ($time,a,b,c,d); //b<=12;
end
initial begin
c=20; $display ($time,a,b,c,d); //c<=20;
d=25; $display ($time,a,b,c,d); //d<=25;
end
endmodule

o/p => (=)

0 6 x x x
0 6 12 x x
0 6 12 20 x
0 6 12 20 25

o/p => (<=)

0 x x x x
0 x x x x
0 x x x x
0 x x x x

$display (ex2) refer to the structure

Q
a=6;
#5 b=12;
#5 c=20
#5 d=25

o/p =>

0 6 x x x
5 6 12 x x
5 6 12 20 x
10 6 12 20 25

Q) a<=6;
#5 b<=12;
#5 c<=20;
#5 d<=25;

o/p =>

0 x x x x
5 6 x x x
5 6 x x x
10 6 12 20 x

Q) $Strobe (ex 2) ryer for the structure

```
a = 6;
#5 b=12;
#5 c=20
#5 d=25
```

$time	a	b	c	d
0	6	x	x	x
5	6	12	20	x
5	6	12	20	x
16	6	12	20	25

```
a<=6;
#5 b<=12;
#5 C<=20;
#5 d<=25;
```

$time	a	b	c	d
0	6	x	x	x
5	6	12	20	x
5	6	12	20	x
10	6	12	20	25

Q) integer a,b,c,d;

```
initial begin
$monitor ($time, a, b, c, d);
  a<=5;
  #3 b=6;
begin
#2 c<=10;
#5 d<=6;
end
#5 a=3;
end
```

$time	a	b	c	d
0	5	x	x	x
3	5	6	x	x
5	5	6	10	x
10	5	6	10	6
15	3	6	10	6

Conditional Statements

* Conditional statements are statements in which assignments occurs when certain condition will be true or false.
* 'if-else' are used as conditional statements
* Nesting of conditional statements is also allowed.
* Expression is considered as true if it evaluates to a non zero value
* Expression is considered as false if it evaluates to a zero or ambigous (x) value
* Syntax : if without else

```
if (<expression>)
begin
  true-statements;
end
```

Ex : D latch

```
always@(enable, d)
  if(enable)
    begin
      q=d;
    end.
```

- If 'else' part is not written it will call the previous value (latch inferring).

* Syntax : if with else

```
if (<expression>) begin
  true_statements;
end
else begin
  false_statements;
end
```

Ex: module test (a,b,c);

```
input a,b;
output c;
always@(a,b) begin
  if(a==b)
    c=0;
  else
    c=1;
end
endmodule
```

* Nested if-else :-

```
if (<condition>) begin
  true_statement;
end
else if (<condition2>) begin
  true_statement;
end
else begin
  default_statements;
end.
```

** Diffrence between if-else & ternary operator

- In if-else → Condition - 1 True statement
  Condition - '0' or 'x' False statement

- In ternary → Condition - 1 True statement
  Condition - 0 False statement
  Condition - x/z Matching the true & false statements

Ex: assign y = (1'bx) ? 8'b1010_0100 : 8'b0101_0101;

y = xxxx_010x //

```
1010 0100
0101 0101
xxxx 010x
```

Ex: if (1'bx) y = 8'b1010_0100;
else y = 8'b0101_0101;

y => 8'b01010101 //

Multiway Branching

* Verilog provides case statement which tests whether expression matches one of the multiple alternatives.
* The expression is compared to the alternatives in the order they are written.
* Statements corresponding to first alternative that matches the expression are executed.
* In this if no match is found, default statement is executed.
* default statement is optional.
* default statement can be written in any where, it is always the last alternative expression is to be compared to.
* Nesting of case statement is allowed
* Begin and end Keyword is optional if there is only one statement to be executed for given alternative then Begin & end need not to be written

** The case statement is different from if-else in following ways:
- Expression given in if else is more general while a single expression is matched with multiple item in case statement.
- case provides the definitive results when x & z are present in the expression.

* Syntax:

```
case (<expression>)
  alter1: begin
    multiple statements 1; end
  alter2: begin
    single statement 2; end
  default: begin
    default statements; end
endcase
```

* Ex:

```
4x1 mux
always@(*)
case(sel)
  2'b00 : out=a;
  2'b01 : out=b;
  2'b10 : out=c;
  2'b11 : out=d;
  default: $display("invalid statement");
endcase
```

* In 'case' latch inferring will be occuring when all the alternatives are not evaluated & there is no default statement also.

→ case:
- In a normal case statement, valid inputs are 0, 1, x & z
- The expression & alternatives are compared bit wise bit including x & z.

→ casez:
- In casez, valid inputs are 0, 1 & x.
- It treats 'z or ?' present in expression or alternation as don't care

→ casex:
- In casex, valid inputs are 0 & 1
- It treats x, z or ? present in the expression or alternative as don't care.

Ex) always @(*)

```
casex (sel)
  2'b0x : out=a;
  2'b0z : out=b;
  2'b10 : out=c;
  2'b11 : out=d;
  default : out=0;
endcase
```

```
always
// casez (sel)
```

```
// casex (sel)
```

		casex	casez	casex
O/P =>	sel = 00	casex : out = b	casez : out = b	casex : out = a
	sel = 0x	casex : out = a	casez : out = a	casex : out = a
	sel = 0z	casex : out = b	casez : out = a	casex : out = a
	sel = xx	casex : out = 0	casez : out = 0	casex : out = a

* Reverse case :

00 - default
01 - s[0]
10 - s[1]
11 - s[1]

```
case (1)
  s[1] : Statmnt;
  s[0] : Statmnt;
  default : statmnt;
endcase
```

* Parallel case : It only checks only one case at a time

```
case (s)
  2'b00 : x = 10;
endcase
```

only one alternative is written in a case
writing multiple case will make it parallel case.

⇒ Race Condition :-

* In verilog sometimes more than one assignments are scheduled for execution at same time.
* In this situations non-deterministic condition occurs. that is known as race around condition.
* In race around conditions order of execution dependent upon simulator
* To avoid this situations one can use #0 modling.

Ex) module race_around1;
reg [1:0] a=2'b01, b=2'b10;
initial a=b;
initial b=a;
endmodule

- Definitely both a & b will be same
- Depends upon which initial block executes first that depends upon simulator

1st → a=2'b10 & b=2'b10
2nd → a=2'b01 & b=2'b01

module race_around_sol;
reg [1:0] a=2'b01, b=2'b00;
initial a<=b;
initial b<=a;
endmodule

- a=2'b10 & b=2'b01 (swap)

LOOPS IN VERILOG

* loops in verilog are used to execute the same piece of code a number of times.
* For iteration functionality verilog basically provides four different loops: for, while, repeat, forever.
* loops can have delay statements
* We can combine more than one statements using begin-end block in an looping instruction.
* looping statements should be used within a procedural block i.e inside the initial block or always block.
* Synthesizable loop: for, repeat (to design HW we require fixed no of iterations).

→ For loop:-

* for loop is most widely used loop in verilog to replicate the hardware
* for loop executes till the condition is true
* Syntax:

```
for (<initialization>; <condition_check>; <updation_in_condition>)
begin
  statment 1;
  statment 2;
end
```

Ex

```
module for_test;
 integer i;
 initial begin
  for(i=1; i<=8; i=i+1)
  begin
   $display("The for loop count = %0d", i);
  end
 end
endmodule
```

O/P =>

```
The for loop count = 1
                   = 2
                   = 3
                   = 4
                   = 5
                   = 6
                   = 7
                   = 8
```

0/p

- reg [2:0] a;
  for(a=0; a<=7; a=a+1)
  $display("The value of a is = %d", a);

  0
  1
  2
  3
  4
  5
  6
  7
  0
  :

  Infinite loop.

- ~~for loop~~

* In verilog each part of the for loop is optional, but semicolon(;) must still be present to separate them.

i) for(; i<5; i=i+1)

- If not initialized, it may use a default or undefined value, leading to unpredictable behaviour.

ii) for(i=0; ; i=i+1)

- Treated as an infinite loop because there is no condition to stop.

iii) for(i=0; i<5;){

- No automatic change in i, so if not manually updated inside the loop, this may lead to infinite loop

iv) for(;;) // infinite loop

- Equivalent to while(1)

→ While loop :-

* While loops are used in verilog often to run some code for an intermidate amount of time
* while loop executes until the condition is not true.
* Syntax ⇒ while (condition)
  begin
  multiple statements;
  end
* while loops are not synthesizable, as when synthesis tool tries to convert while loop in gates & registers it must know the exact number of times loop will run. In case of while loop it is unknown
* Very useful in testbenches & verification.

Ex) module while_test;
integer count = 0;
initial begin
while (count != 7)
#1 count = count + 1;
$display ("The count value is = %d", count);
end
endmodule

O/P ⇒ The count value is = 7

module clk_gen;
reg clk;
initial begin
clk = 1'b0;
while (1)
#5 clk = ~ clk;
end
endmodule

O/P → It will generate clock continuously which flip its value after every 5 time units.

- If 'begin' & 'end' was given multiple statements would have displayed (7)

→ Repeat loop :-

* repeat loop executes the loop for a fixed number of times
* repeat (<no. of time>)
  begin
    Statement 1;
    Statement n;
  end

Ex > module repeat_test;
integer i = 0;
initial begin
repeat (10)
#1 i = i + 1;
$display ("The value of i is = %d", i);
end
endmodule

O/P =>
The value of i is = 10

→ forever loop :-

* forever is an infinite loop that executes without any condition
* forever is equivalent to while(1)
** The forever loop don't contain any expression, executes forever until "$finish" task is encountered.
* forever can be exited by use of disable statement
* primarily used in testbench

* Ex :- module clk_gen;
reg clk;
initial begin
forever #5 clk = ~clk;
end
endmodule

* Syntax =>
forever
begin
multiple statements;
end

Q Write a code which makes for loop work like a while loop & vice versa.

- module for_with_while;
  integer i;
  initial begin
  i = 0;
  while (i<5) begin
  $display("while loop : i = %0d", i);
  i = i+1;
  end
  end
  endmodule

- module while_with_for;
  integer i=0;
  initial begin
  for (; i<5;) begin
  $display("For loop : i = %0d", i);
  i = i+1;
  end
  end
  endmodule

Generate

⇒ Generate Statement :-

* Generate statement are used to generate replication of hardware which is provided in generate statement.
* It works in looping manner that's why also known as generate loop.
* All generate instantiations (hardware to be replicated) are written inside generate & endgenerate keywords.
* Generate instantiations can be combinations of any of the following:
  - module instance
  - User defined primitives
  - gate primitives
  - Continuous assignments
* Verilog permits declaration of nets, registers & events in generate scope.
* ** Declaration of parameters, local parameters, input, output, inout port & specify block is not allowed.

* There are three types of generate statements
  i) Generate loop
  ii) Generate condition
  iii) Generate case
* "genvar" is the keyword used to declare variables that will be used only in evaluation of generate block.
* Value of genvar variable can only be modified by generate loop.
* Nesting of generate loop is allowed.
* Block identifiers are required for generate loop in most cases.

* Syntax =>

- generate - for :-

```
genvar i;
generate
for(i=0; i<8; i=i+1)
begin: label
  S1;
  S2;
end
endgenerate
```

- generate - conditional

```
generate
if (condition)
  ≡
else
  ≡
endgenerate
```

- generate - case

```
generate
case (variable) begin
  c1;
  c2;
  default statement;
endcase
end
endgenerate
```

→ 4- Bit adder using generate for

* // Full adder
module FA(input a,b,cin, output sum,carry);
assign {carry, sum} = a+b+cin;
endmodule

• module adder (input [3:0] op1,op2, output cout, output [3:0] sum);
wire [4:0] carry;
assign carry[0] = 1'b0;
assign cout = carry[4];
genvar i
generate for (i=0; i<4; i=i+1)
begin : ripple
FA fa (op1[i], op2[i], carry[i], sum[i], carry[i+1]);
end
endgenerate
endmodule

* module adder #(parameter WIDTH=4)(input [WIDTH-1:0] op1,op2,
output cout, output [WIDTH-1:0] sum);
wire [WIDTH:0] carry;
assign carry[0] = 1'b0;
assign cout = carry[WIDTH];
genvar i;
generate for (i=0; i<WIDTH; i=i+1)
begin : ripple
assign {carry[i+1], sum[i]} = op1[i] + op2[i] + carry[i];
end
endgenerate
endmodule

→ Generate Conditional Example:-

```
* module andor #(parameter WIDTH = 4)(input [WIDTH-1:0] op1, op2,
  output [WIDTH-1:0] result);
  parameter logics = "and1";
  genvar i;
  generate
  if (logics == "and1")
   for (i=0; i<WIDTH; i=i+1)
   begin : andlogic
    assign result[i] = op1[i] & op2[i]; end
  else
   for (i=0; i<WIDTH; i=i+1)
   begin : orlogic
   assign result[i] = op1[i] | op2[i]; end
  endgenerate
  endmodule
```

→ Generate Case Example:-

```
* module adder (sum, carry, a, b, c);
  parameter N=3;
  input [N:0] a,b; input c;
  output carry; output [N:0] sum;
  generate
  case (N)
  1: adder_1bit a1(sum, carry, a, b, c);
  2: adder_2bit a2(sum, carry, a, b, c);
  default: adder_custom a3 #(N)
          (sum, carry, a, b, c);
  endcase
  endgenerate
  endmodule
```

Procedural Block Statements

* Block statements are used to group two or more statements together so that they belong to single logic element.
* Procedural Block statements basically of two types:
  - Begin - end
  - fork - join
* Begin - end is known as sequential block as all the statements inside this are execute in sequential manner
* fork - join is known as parallel block as all the statements inside this are executed at same simulation time manner.

→ Begin - end
* It is a sequential block
* Begin - end is synthesizable statement
* Delay statements are allowed
* Nesting is allowed
* Syntax => initial / always

```
begin
  S1;
  S2;
  S3;
end
```

Ex: integer a=0, b=0, c=0, d=0, e=0;
initial begin

a=5;	// a=5	0
#3 b=7;	// b=7	3
#5 c=4;	// c=4	8
#8 d=10;	// d=10	16
#2 e=9;	// e=9	18

end

→ Fork - join :-

* It is a concurrent Block
* fork join is non-synthesizable statement
* Delay statements are allowed
* Nesting is allowed.
* Syntax => initial / always

```
fork
  S1;
  S2;
  Sn;
join
```

Ex) integer a=0, b=0, c=0, d=0, e=0;

```
initial fork
  a=5;        // a=5  0
  #3 b=7;     // b=7  3
  #5 c=4;     // c=4  5
  #8 d=10;    // d=10 8
  #2 e=9;     // e=9  2
join
```

** If we are not providing any delay then the execution of fork join & begin-end is same.

* module test;

```
integer a=0, b=0, c=0, d=0, e=0;
initial begin
$monitor ($time, a, b, c, d, e);
a=5; #3 b=7; #6 c=4;
fork
#1 a=9; #4 d=3; #5 c=8;
join
#8 d=10; #2 e=9;
end endmodule
```

Time	a	b	c	d	e
0	5	0	0	0	0
5	5	7	0	0	0
11	5	7	4	0	0
12	9	7	4	0	0
15	9	7	4	3	0
16	9	7	8	3	0
24	9	7	8	10	0
26	9	7	8	10	9

```
* module tb;
integer a=0, b=0, c=0, d=0, e=0;
initial fork
a=2; #6 b=7; #3 c=4;
begin
$monitor ($time, a, b, c, d, e);
#1 b=9; #4 d=3; #5 c=1;
end
#9 d=10; #2 e=9;
join
endmodule
```

Time	a	b	c	d	e
0	2	0	0	0	0
1	2	9	0	0	0
2	2	9	0	0	9
3	2	9	4	0	9
5	2	9	4	3	9
6	2	7	4	3	9
9	2	7	4	10	9
10	2	7	1	10	9

Named Block

* Verilog allows giving names to block statements. Thus blocks are called named blocks.
* Local variables can be declared for named blocks.
** Variables in a named block can be accessed by using hierarchical name referencing.
* It is possible to stop execution (disable) of named block from within the same block or different block.
* Syntax =>

```
initial
begin : Block 1
integer count;
// count variable local to Block1
statements;
end
```

```
initial
begin : Block 2
reg a; // a variable local to Block2
Statements;
end
```

* module tb;

initial

begin : Block 1

integer count;

count = 0;

$monitor ($time, count);

forever begin

#5 count = count + 1;

if (count == 8)

disable block 1;

end end

endmodule

o/p

Sim time	count
0	0
5	1
10	2
15	3
20	4
25	5
30	6
35	7 ← Printing values
40	8

* module tb;

integer count = 0, a = 0, b = 0;

initial begin : Block 2

b = 2;

begin : Block 1

forever

#1 count = count + 1;

end

end

initial begin

$monitor ($time, a, b, count);

a = 3; #5 a = 7; #4 a = 2;

#2 disable Block 2, Block 1;

#3 a = 9; (Block 1 will be disabled)

end

endmodule

Time	a	b	count
0	3	2	0
1	3	2	1
2	3	2	2
3	3	2	3
4	3	2	4
5	7	2	5
6	7	2	6
7	7	2	7
8	7	2	8
9	2	2	9
10	2	2	10
14	9	2	10

Functions & Tasks

* Verilog lets you define sub-programs using tasks & functions.
* They are used to improve the readability & to exploit re-usability code.
* Definition of task & function must be in a module
* Task or functions can only contain sequential statements.
** Functions can accept any number of inputs, do some processing on inputs & return only one output.
* Tasks is more general, that can calculate multiple result values & return them in output or inout type arguments.
* Tasks may contain @, posedge & other simulation time consuming elements.

=> Functions :-

* Declaration of functions started with keyword "function" & ended with "endfunction".
* Function always return a single value.
* They can't have output & inout ports.
* A function must have at least one input argument. It may contain more than one input.
** Delays, Event & timing control statements are not supported by functions.
* Functions always executes in zero simulation time.
* When a function is declared, a register with the same name is implicitly (By default) declared in verilog.
* A value is returned back by assigning appropriate value to the implicit register (function name).
* Functions can't have any triggers.
* The function is invoked by specifying functions name followed by input arguments.
* A function can be called from a function, task, port declaration, continuous assignment & procedural assignment.

* A function can invoke other functions but it can't invoke other tasks.
* Syntax =>
  - function [return type] function_name ;
    input declarations ;
    // local registers declaration ;
    begin
    sequential statements ;
    end
    endfunction

  - function [return type] function_name (input declarations) ;
    // local register's declarations ;
    begin
    sequential statements ;
    end
    endfunction.

* Return type & all inputs are 1-bit reg by default
* Return variable name is same as function_name
* * Only blocking sequential statements are allowed.
* Functions have access to variables declared outside its scope

→ How to call function ??
- assign net_name = function_name (input_list) ;
- reg_name1 = function_name (input_list) ;
- reg_name2 <= function_name (input_list) ;

* Function always executes in Active region.

Q Write a function which accepts 8 bit input din & returns parity. Use for loop instead of reduction operator.

```
function parity ( input [7:0] din );
  integer i;
  reg temp;
  begin
  temp = 0;
  for (i=0; i<=7; i=i+1)
  begin
  temp = temp ^ din[i];
  parity = temp;
  end
endfunction
```

```
module parity_call (din, dout);
  input [7:0] din;
  output dout;
  assign dout = parity (din);
  endmodule
```

Q Write a function to return maximum out of three integers a,b&c.

```
function integer maximum;
  input signed [31:0] a,b,c;
  begin
  if (a > b && a > c)
    maximum = a;
  else if (b > c)
    maximum = b;
  else
    maximum = c;
  end
endfunction
```

```
module max_val (a,b,c, max_val);
  input signed [31:0] a,b,c;
  output integer max_val;
  assign max_val = maximum (a,b,c);
  endmodule.
```

→ Automatic Function :-

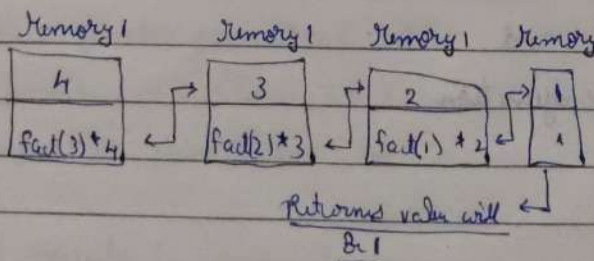
* A function call allocates static memory to variables used in its operations.
* If the same function is called again, same set of allocated memories are used for its operation.
* In case of recursive or re-entrant functions (function called within a function) using static memories may give wrong result.
* "automatic" keyword is added after function keyword to target a recursive function.
** An automatic function allocates a new set of memory to the variables every time a function is called.

Q Write a function which accepts positive integer & returns factorial of that number

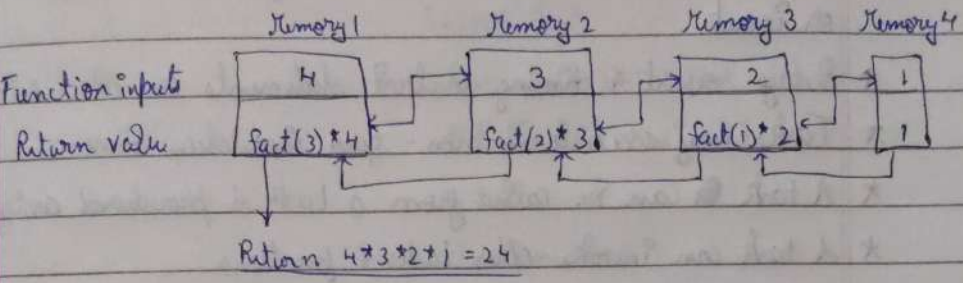
```
function [63:0] fact (input [31:0] din);
begin
if (din)
  fact = fact(din - 1) * din;
else
  fact = 1;
end
endfunction
```

```
* module fact_test (din, dout);
input [31:0] din;
output reg [63:0] dout;
always @ (*)
  dout = fact (din);
endmodule
```

* Static function Problem



* To solve this issue or will we add automatic keyword because we intend to design a recursive function.
  - function automatic [63:0] fact (input [31:0] din);



- Since automatic keyword is added to the function, for every function call a new set of memory is created & the result is returned to location where the function call occurs.

* Verilog supports some other type of functions:
  - Signed function
  - Constant function

- function signed my_func1 (input list);
  begin
  sequential statements;
  end
  endfunction

* function [31:0] log2 (input [31:0] data);
  reg [31:0] temp;
  begin
  temp = 0;
  while (data > 1) begin
  data = data >> 1;
  temp = temp + 1;
  end
  log2 = temp;
  end
  endfunction

⇒ Tasks :-

* Tasks are declared with keyword task and endtask.
* Tasks can pass multiple values through output & inout ports.
* A task may have zero or more arguments of type input, output or inout.
* Delay, event & timing control statements are allowed inside tasks.
* Tasks may execute in non-zero simulation time.
* A task can be called from a task & procedural assignment.
* A task can invoke other tasks & functions.
* A task is invoked by specifying task name followed by input, output or inout agruments.
* * The argument should be specified in the same order they are declared
* Tasks can be disabled by "disable" keyword.

* task task_name ; • task task_name (port declaration) ;
input, output or inout declaration ;
local registers declaration ;
begin
// begin-end if multiple statements
[timing control]
sequential statements ;
end
endtask

* By default the size of ports & local arguments is 1 bit reg
* Both blocking & non-blocking sequential statements are allowed
* Call -> task_name (input, output or inout list) ;

Ex>

```
task inoument                    module task_inout;                    o/p =>
inout integer a;                 integer x=0;                          x=1
  a=a+1;                         // Task declaration                   x=7
endtask                          initial begin
                                   increment(x); // Task call
                                   $display("x=%d",x);
                                   x=6;
                                   increment(x);
                                   $display("x=%d",x);
                                   end endmodule
```

-> Static & Automatic Tasks :-

* Static task is the default task in verilog.
* In static tasks all the member variables will be shared across diffent invocation of same task.
* Automatic Task : "automatic" keyword is placed after task to make task automatic.
* In automatic tasks, for each invocations of same task a new memory is allocated to its member variables.

Ex>

```
module task_demo;                    module task_demo;
  task display();                      task automatic display();
  integer x=0;                         integer x=0;
    x= x+2;                              x= x+2;
  $display("x=%d",x);                  $display("x=%d",x);
endtask                              endtask
  initial display();                   initial display();
  initial display();                   initial display();
endmodule                            endmodule
o/p => x=2                           o/p => x=2
       x=4                                  x=2
```

→ Global Task :-

* The task that are declared outside the module are known as global tasks
* The scope these tasks is global, they can be called within any module.

```
• task display();
$display("Welcome to Futurewiz");
endtask
• module task_demo1;
initial begin
$ display("The task_demo1:");
display();
end
endmodule
```

```
• module task_demo2;
initial begin
$display("The task_demo2:");
display();
end
endmodule
```

O/P => The task_demo1
Welcome to Futurewiz
The task_demo2
Welcome to Futurewiz

→ How to disable a Task :-

* Any task can be disabled using the keyword "disable".

Ex) module task_disable_demo;
task display();
begin : Message_display1
$display("At Sim Time = %0t Welcome to Futurewiz", $time);
#50 $display("At Sim Time = %0t This is situated in Noida", $time);
end
begin : Message_display2

```
#10 $display("At sim time =%t Futurewiz is a VLSI training institute",$time);
#10 $display("At sim time = %t it is the venture of Truechip",$time);
end endtask
initial display();
initial begin
  #40 disable display.messages-display1;
end endmodule
```

O/P => At sim time=0 Welcome to Futurewiz
At sim time = 50 Futurewiz is a VLSI training institute
At sim time = 60 it is the venture of Truechip

```
• initial display();
initial begin
  #40 disable display.messages-display2;
end endmodule
```

O/P => At sim time = 0 Welcom to Futurewiz
At sim time = 50 This is situated in Noida
At sim time = 60 Futurewiz is a VLSI training institute
At sim time = 70 it is the venture of Truechip.

→ Difference between Task & function :-

Function	Task
• Functions are used to return only single value.	• Task can provide multiple values
• Function must have at least one input argument.	• Task may have zero or more arguments of type input, output or inout.
• Timing control statements are not allowed	• Timing control statements are allowed
• Always executes at zero simulation time	• May or may not executes at zero simulation time.

- Only Blocking assignments are allowed
- Functions can invoke other function only

- Both Blocking & non-Blocking assignments are allowed
- Tasks can invoke other tasks as well as functions.

=> Paramters :-

* Paramters are used to declare constants in verilog.
* Thes constants can be overridden during compilation time (tb)
* "defparam" keyword is used to overrid a paramter.

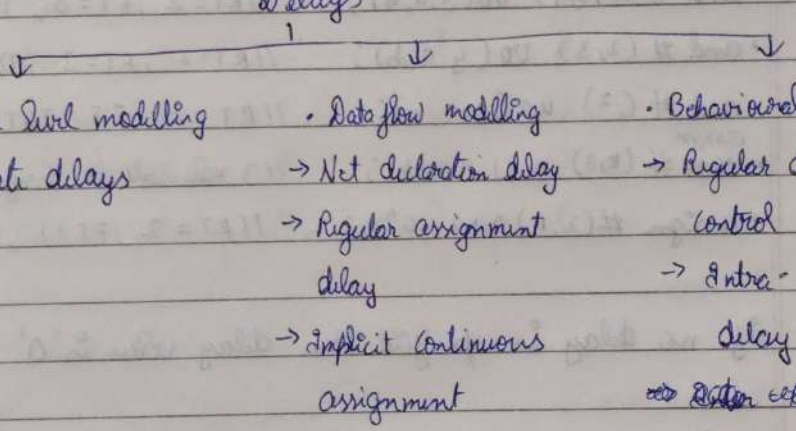
* Syntax => defparam instancelabl.paramter_name = value;

Ex) module custom_adder (a,b,cin, sum, carry);
parameter size = 2;
input [size-1:0] a,b;
input cin;
output [size-1:0] sum;
output carry;
assign {carry, sum} = a+b+cin;
endmodule

- module adder_test;
defparam m0. size = 4;
reg cin; reg [m0. size-1:0] a,b;
wire [m0. size-1:0] sum; wire carry;
custom_adder m0(a,b,cin, sum, carry);
Initial begin repeat(5) begin
a = $random; b = $random; c = $random;
#10; $display(a,b,cin, sum, carry);
end end endmodule

DELAY's

* Delays specify time in which assigned values propagate through nets or from inputs to outputs of ports or from one port to another port.
* Aim of any designer may be to produce a circuit that functions correctly, it is equally important that the circuit also conforms to any timing constraints required of it.
** Delay & timing control statement are not synthesizable
* Delays can be modelled in a variety of ways, depending on the overall design approach which are mentioned below:
  - Gate level modelling
  - Data flow modelling
  - Behavioral modelling



⇒ Gate delays :-

* These delays basically provided with in-built gate primitives, continuous assignment & user define primitives.
* Gate delays can be classified as following:
  - Rise delay
  - Fall delay
  - Turn-off delay

* Rise delay encountered only when output changes from any state (0,x,z) to 1.
* Fall delay encountered only when output changes from any state (1,x,z) to 0.
* Turn-off delay encountered only when output changes from any state (0,1,x) to z.
* Unknown delay: In case when output goes from (1,0,z) to x then minimum out of rise, fall & turn-off delay will be assigned.

* Syntax =>

primitive_name / assgn / udp # (rise, fall, turn-off) label (o/p, i/p1, i/p2);

Ex: • and #(2,3,4) u0(y,a,b); //RT=2, FT=3, TOT=4
• and #(2,3) u0(y,a,b); //RT=2, FT=3, TOT=2 (min)
• and #(2) u0(y,a,b); //RT=2=FT=TOT
• assign #(3) out1 = a|b; //3 value will be assigned to all delays
• assign #(2,1) out2 = a^b; //RT=2, FT=1, TOT=1

* If no delay is specified then delay value is '0'.

```
• module my_design (input x,y, output o1,o2);
  and #(1) a1(o1,x,y);
  buf#0 #(2) b1(o2,x,y);
  endmodule
  module tb;
  reg a,b; wire o1,o2;
  my_design dut (a,b,o1,o2)
  initial begin
  a=0; b=0;
```

```
$monitor ("T=%0t a=%0b b=%0b out1=%0b out2=%0b", $time, a, b, o1, o2);
#10 a<=1; #10 b<=1; #10 a<=0; #10 b<=0;
end
endmodule
```

O/P =>

T	a	b	out1	out 2
0	0	0	z	z
1	0	0	0	z
2	0	0	0	0
10	1	0	0	0
12	1	0	0	1
26	1	1	0	1
21	1	1	1	1
22	1	1	1	z
30	0	1	1	z
31	0	1	0	z
40	0	0	0	z
42	0	0	0	0

→ min, typ & max delays :-

* Delays are neither the same in different parts of the fabricated chip nor the same for different temperatures & other variations.
* Every digital gate & transistor cell has a minimum, typical & maximum delay specified based on process node & is typically provided by libraries from fabrication foundry.
* For rise, fall, & turn-off delays, the three values min, typ & max can be specified.
  - min : minimum, typ : typical & max : maximum.
* This is another level of delay control in verilog. Only one of the min, typ & max values can be used in the entire simulation run.
* It is specified at the start of the simulation & depends on the simulator used.
  - 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 (default).
  - The max value is the maximum delay value the gate is expected to have.

* Syntax =>

primitive_name / assign / udp #(min:typ:max, min:typ:max, min:typ:max) inst (output, input1, i/p2)
(min:typ:max → Rise; min:typ:max → Fall; min:typ:max → Turn off)

Ex) and #(2:3:4) U0(out, ip1, ip2); // for all transition Rise delay = 2 Typ = 3 & max = 4

and #(2:3:4, 3:5:7) U1(out, ip1, ip2); // for Rise delay - Min = 2 T = 3 max = 4
Fall delay - Min = 3 T = 5 max = 7

bufif0 #(2:3:4, 3:4:6, 4:5:7) U0(out, ip1, ctrl); RD - Min = 2 T = 3 max = 4
FD - Min = 3 T = 4 max = 6
TO - Min = 4 T = 5 max = 7

⇒ Data flow delays :- / Net delays :-

→ Net declaration delay :-

* In this delay assignment, we assign the delay value at the time of declaration.

Ex) wire #10 c; //At the time of declaration delay is assigned
assign c = a & b;

→ Implicit continuous assignment delay :-

* Delay value is assigned at the time of implicit declaration.

Ex) wire #10 c = a & b;

⇒ Behavioural modling delays :-

→ Regular / Intra assignment delay :-

* In regular / intra assignment delay, the execution of statement is delayed by a specified time
* First delay is performed & then register is updated with the value at current time.
* Any event on the sensitivity list is ignored till all the delays are executed.
* Both Blocking & non-Blocking shows the similar behaviour with regular delay.
* Syntax :

```
# delay_value output = expression ;  or
# delay_value output <= expression
```

Ex) module delay (input signed [3:0] a, output integer b);
always @ (a)
#3 b=a; // #3 b<=a ; will produce some results
endmodule

→ Intra Assignment delay :-

* In intra assignment delay control, the assignment of statement is delayed by a specified time.
* Current values are read & assigned to the register after specified delay.
** Any event on the sensitivity list is ignored till all the delays are executed for Blocking statements.
* Blocking & non-Blocking statements don't show same behaviour with this type of delay.
* Syntax =>

output = #delay_value expression; or
output <= # delay_value expression;

Ex) module delay (input signed [31:0] a, output integer b);
always @(a)
#3 b=a; // #3 b<=a;
endmodule

o/p =>
Simtim = 0, a=10, b=x
Simtim = 3, a=10, b=10
Simtim = 4, a=15, b=10
Simtim = 7 a=15, b=15

module delay_tb;
reg signed [31:0] a;
wire integer b;
delay d1(a,b);
initial begin
a=10;
$monitor ("simtim =%t, a=%d, b=%d", $time, a,b);
#4 a=15;
end
endmodule

* Assignment Statement	Description
. #5 z = x & y;	• x & y is evaluated after 5 time units & assigned to z • x or y changes within 5 time units, ignored • If input pulse is shorter than 5 time units then also changes is ignored.
• #5 z <= x & y;	• same as #5 z = x & y
• z = #5 x & y;	• x & y evaluated immediately but assigned to z after 5 time unit • if x & y changes its value with in 5 time unit, produces no effect on z.
• z <= #5 x & y;	• x & y evaluated immediately & scheduled assignment to z after 5 time unit. • Further changes in x or y are taken in consideration • This statement behaves according to hardware behaviour.

```
Q2 module tb;
reg a=10, c=15 b=5;
initial begin
#5 a=5;                    // Sim tim=5  a=5
$ display (a);
b = #2 11 ;
$ display (b);             // Sim time =7  b=11
c <= #5 20;
$ display (c);             // Sim tim = 12  c=15
$ strobe (c);              // Sim tim = 12  c=20
#5 c <= 30;
$ display (c);             // Sime time = 17  c=20
$ strobe (c);              // Sim tim = 17  c=30
end
```

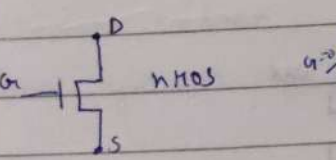
Q) integer a=5, b=7, c=9;
initial begin
#5 b=c-a;
#5 c=a*2;
#5 a=b+c;
$display(a,b,c); 14 4 10
end
endmodule

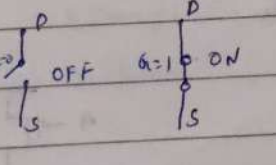
Switch Level Modeling

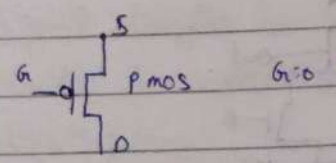
* Verilog supports another level of abstraction in which CMOS circuits can be designed known as switch level modelling.
* The switch level of modelling shows the abstraction level between the digital logic & analog transistor levels of abstraction.
* At switch level, transistor works only in two regions (0,1) but verilog uses a 4 value logic value system. so verilog switch input and output signals can take any of the four (0,1, z & x) logic values.
* Switch level of modeling is rarely used by RTL designers now a days, because of the higher level of complexity at switch level.

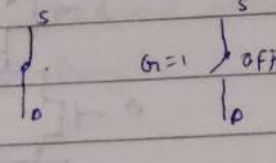
→ Basics of MOS switches :-

* Metal oxide semiconductor field effect transistor (MOSFET) is generally known as MOS.
* MOSFET's are voltage controlled switches.
* Two type of MOSFET's :
  - NMOS (N-channel MOSFET)
  - PMOS (P-channel MOSFET)
* It has three connection points : a source, a drain, & a gate.



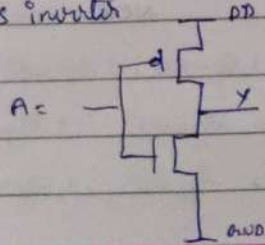




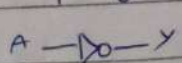


- Combination of both nmos & pmos switches is known as CMOS technology.
- CMOS stands for Complimentary metal oxide semiconductor.

CMOS inverter



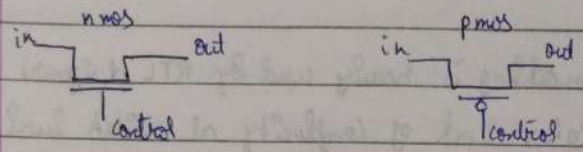
A	Y
0	1
1	0



→ Primitives of switch level modelling :-

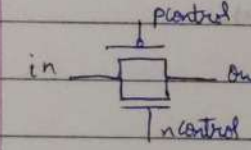
* For switch level design verilog supports switch primitives.
* These primitives can be used like the normal gate primitives.
* These switch primitives are not synthesizable.
* Verilog provides following switch level primitives.

- pmos
- nmos
- cmos
- tran
- tranif0
- tranif1



* Syntax :-

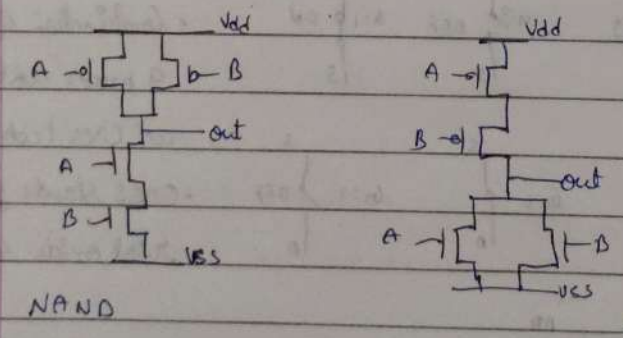
- nmos (out, in, control);
  control = 0 out = z
  control = 1 out = in
- pmos (out, in, control);
  control = 0 out = in
  control = 1 out = z



- cmos (out, in, ncontrol, pcontrol);

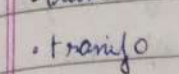
P & n control are usually compliment of each other

→ NAND & NOR gate :-

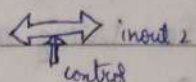


→ Bidirectional switches :-

* Verilog provides following bidirectional switches:
  - tran
  - tranif1
  - tranif0



tran(inout1, inout2)



tranif1 (inout1, inout2, control)

tranif0 (inout1, inout2, control)

control	tranif0	tranif1
0	inout2 = inout1	inout2 = z
1	inout2 = z	inout2 = inout1

→ Resistive Switches :-

* Resistive switches have higher source to drain impedance than regular switches & reduce the strength of signals passing through them.
* nmos, pmos, cmos & bidirectional switches can be modified as resistive devices in Verilog.
* Resistive switches are declared with keyword having an "r" prefixed to the corresponding regular switch keyword.
* Verilog provides following resistive switches:
  - rnmos
  - rpmos
  - rtranif0
  - rcmos
  - rtran
  - rtranif1
* Same syntax as regular switches.

→ Signal Strengths in switch level modeling :-

* Verilog has 4 driving strengths, 3 - capacitive / storage strengths & high impedance.
* Syntax =>

  (Strength0, Strength1) or (Strength1, Strength0) or capacitive_strength

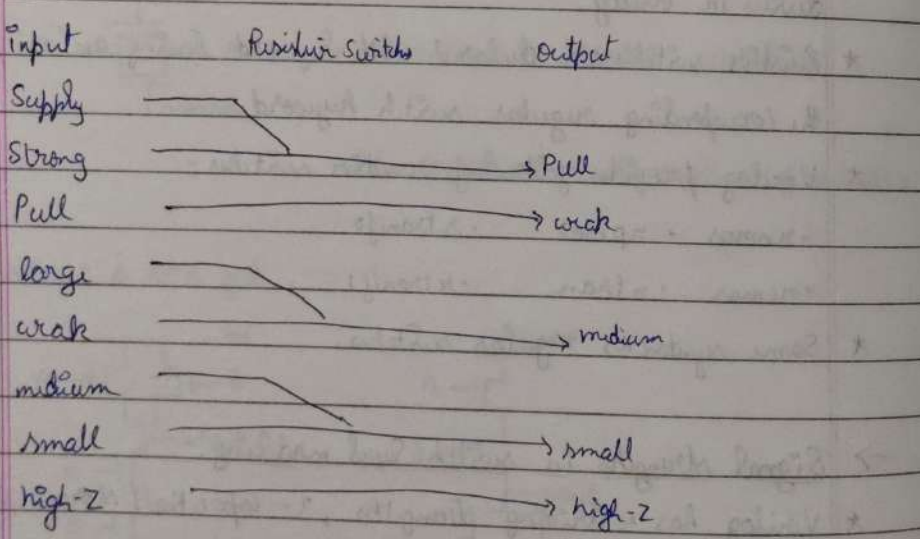
* Strength0 : Strength when the net drives the value 0.
* Strength1 : Strength when the net drive the value 1.

* Capacitive strength is only for the trireg net only.
* The default strength is strong drive.
* For pullup & pulldown gate, the default strength is pull drive.
* For trireg the default strength is medium capacitive
* For supply nets, the default strength is supply drive.

*

Strength	Strength 0	Strength 1	Strength type	Degree
Supply	supply0	supply1	Driving	Highest
Strong	strong0	strong1	Driving	↑
Pull	pull0	pull1	Driving	
large	large0	large1	Storage/capacitive	
weak	weak0	weak1	Driving	
medium	medium0	medium1	Capacitive/storage	
Small	Small0	small1	Capacitive/storage	
High Z	High Z	High1	High Z	Lowest

*



Q) Write a Switch level code to implement NAND & NOR gate.

* //NAND

```
module nand_switch(output out, input a, b);
  wire n;
  supply1 Vdd;
  supply0 gnd;

  pmos p1(out, Vdd, a);          // series connection
  pmos p2(out, Vdd, b);

  nmos n1a(n1, gnd, a);          // Parallel connection
  nmos n1b(out, n1, b);
endmodule
```

* //NOR

```
module nor_switch(output out, input a,b);
  wire p1_out;
  supply1 Vdd;
  supply0 gnd;

  pmos p1(out, Vdd, a);          // parallel connection
  pmos p2(out, Vdd, b);

  nmos n1(p1_out, gnd, a);       // Series connection
  nmos n2(out, p1_out, b);
endmodule
```

User Defined Primitives

* User defined primitives define new primitives, which functions exactly the same as built in primitives in verilog.
* Instances of these new UDPs can then be used in exactly the same manner as the gate primitives to represent the circuit being modled
* Evaluations of these UDPs is accelerated by the verilog-XL algorithm
* This technique can reduce the amount of memory that a description needs & can improve simulation performance
* UDP definitions are independent of a module. They are at same level as that of module definition in syntax hierarchy.
* UDP don't contain any module or primitive instantiation inside it. They are instantiated the same way other modules & gate level primitives are instantiated.

→ Properties :-

* One of the advanced concepts of verilog
* Non Synthesizable
* Single output many input
* Consume very less memory
* I/O's must be scalar type only.
* 'z' value is not supported by UDPs

→ Types of UDPs :-

* 2 categories :-
  - Combinational UDPs
  - Sequential UDPs

→ Combinational UDPs :-

* Output has to be first port in the port list.
* Output state is function of current input states. Whenever input

state changes, UDP is evaluated & output state is set to a value as indicated by entry in the state table for current input state.

* The output is 'x' for a combinational of input state not present in the state table.

* primitive primitive_name (output, input, input, ...);
  output terminal declaration;
  input terminal declaration;
  table // state table entries
    table_entry;
    table_entry;
    :
  endtable
endprimitive

→ State Table :-

* State table is specified with help of 'table' & 'endtable' keyword.
* Each entry in the state table of a combinational UDP has following syntax ⇒ input1_val input2_val .... inputN_val : output_val;
* Input values in the state table should be specified in the same order as they appear in port (terminal) list.
* ":" is used to separate in inputs & output
* ";" is used to end a state table entry.
* All possible combinations of inputs where output produces a known value, must be explicitly covered.
* Combinations of the inputs that are not explicitly specified will drive the output to the unknown value 'x'.
* "?" is used to represent don't care condition
  - "?" covers 0, 1, x values.
* There should not be any conflicting entries in the state table.

* State table for OR operation is written below:

```
table
// a   b   : c;
   0   0   : 0;
   0   1   : 1;
   1   0   : 1;
   1   1   : 1;
   1   x   : 1;
   x   1   : 1;
endtable
```

```
primitive my_or(Y,A,B);
  output Y;
  input A,B;
  table
    0 0 : 0;
    0 1 : 1;
    1 ? : 1;
    ? 1 : 1;
  endtable
endprimitive
```

AND:-

```
primitive my_AND(Y,A,B);
  output x;
  input A, B;
  table
    0 ? : 0;
    1 1 : 1;
    ? 0 : 0;
  endtable
endprimitive
```

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1
0	x	0
x	0	0

* XOR :-

```
00 : 0;
01 : 1;
10 : 1;
11 : 0;
```

* XNOR

```
00 : 1;
01 : 0;
10 : 0;
11 : 1;
```

→ Sequential UDPs :-

** Output has to be the first port & has to be declared as "reg".

* An initial statement can be used to initialize output of sequential UDP.

* The input specification of state table entries can be in terms of input levels or edge transitions.

* All possible combinations of input must be specified to avoid unknown output values.

* primitive primitive_name (output, input, input, ...);
  output reg output_terminal;
  input input_terminals;
  initial output_terminal = logic_value;
  table // State table entries
    table_entry;
    table_entry;
  endtable
endprimitive

* Each entry in the state table of a sequential UDP has following syntax :
  input_values : previous_output_state : output_value;

* Output_value is evaluated based on inputs & previous_output state.
* This value is specified in output_value.
* "-" is used to represent no change in state.

• primitive d_latch (q, d, clear, en);
  output q;
  reg q;
  input d, clear, en;
  initial q=0;

table

// d clear en : q : q+ ;

? 1 ? : ? : 0 ; // clear = 1 , op = 0

? 0 0 : ? : - ; // clear = 0, en = 0 op = latch

0 0 1 : ? : 0 ; // clear = 0, en = 1 op = d

1 0 * : ? : 1 ; // clear = 0, en = 1 op = d

endtable

endprimitive

- primitive T_latch (q, T, clear, en);

output q ;

reg q ;

input T, clear, en ;

initial q = 0 ;

table

// T clear en : q : q+ ;

? 1 ? : ? : 0 ;

? 0 0 : ? : - ;

0 0 1 : 0 : 0 ;

1 0 * : 0 : 1 ;

0 0 1 : 1 : 1 ;

1 0 1 : 1 : 0 ;

endtable

endprimitive

* Sequential State Table :-

State table Symbols	Definition
0	• logic 0 on input or output
1	• logic 1 on input or output
x or X	• unknown on input or output
-	• no change on output (may be only be used with sequential UDPs)
?	• don't care if an input is 0, 1, or x
b or B	• don't care if an input is 0 or 1
• (vw)	• input transition from logic v to logic w eg : (01) represents transition from 0 to 1
r or R	• rising input transition : same as (01)
f or F	• falling input transition : same as (10)

* For edge sensitive UDP transition are denoted by writing old value followed by a new value inside ( ).

eg) (01) → 0 to 1 (0?) → 0 to 0, 1 or x
(10) → 1 to 0 (?1) → 0, 1 or x to 1

* Only one edge specifications are allowed per state entry in a state table.
* Cover all possible combinations of transactions & levels in the state table for which output has known value.

• // State table for positive edge triggered d-ff with clear table

```
D   clear  clk  :  q  :  q+ ;
?     1     ?   :  ?  :  0 ;     // clear output
?    (10)   ?   :  ?  :  - ;     // Hold data at negedge clear
0     0    (01) :  ?  :  0 ;     // capture data at posedge clk
```

D	clear	clk	: q	: q^+ ;	
1	0	(01)	: ?	: 1 ;	// capture data at posedge clk
0	0	(x1)	: ?	: 0 ;	// capture data at posedge clk
1	0	(x1)	: ?	: 1 ;	// capture data at posedge clk
?	0	(1?)	: ?	: - ;	// ignore negedge of clk
?	0	(x0)	: ?	: - ;	// ignore negedge of clk
?	0	0	: ?	: - ;	// ignore data when clock is steady
?	0	(0x)	: ?	: - ;	// ignore clock going to unknown state

endtable

Compiler Directives

* Verilog supports some special keywords for controlling the compilation these keywords are known as compiler directives
* Compiler directive begin with "`" [back-quote].
* Direction can be declared with the syntax as : `name of directive
* Compiler directive can be declared anywhere in the verilog code but it's recommended that few of them should be used outside the module
* Compiler directives are very useful commands in verilog, they are effective from the point they are declared to the point they are overridden by other directive.
* Compiler directive helps in conditional compilation as well as in conditional execution.
* Compiler directives apply globally to the design & also can save significant compilation time.
* Verilog supports following compiler directives :
  - `include
  - `timescale
  - `define & `undef
  - `ifdef, `else, `endif

-> `include :-

* Verilog models can be organized into different files & then compiled together as one unit. One of the useful features in this regard is the `include compiler directive.
* The compiler assumes that content of included source file appears in place of `include compiler directive.
* `include can be used to include globally used function, tasks & macros without enclosing them inside module & endmodule
* Syntax => `include "file_name.v" // relative path
  `include "C:/folder/src/file_name" // absolute path
  module xyz (ip & op);
  //statements
  endmodule

→ 'define & 'undef :-

* 'define is the second most used directive in verilog & very useful one. It's just like parameter, this one is used to add a MACRO to your design.
* The 'undef compiler directives lets you remove definitions of text macros created by the 'define compiler directive.
* 'define can be defined with arguments, so every 'define call can be followed by actual parameters. The compiler recognizes a 'define by its name preceded by accent grave (`) character.
* Syntax =>

```
'define SIZE 8
'define xor_b (x,y) (x & !y) | (!x & y)
// Thus text macros can be used as follow:
module abc(--);
reg ['SIZE -1 : 0] data_out;
c = 'xor_b(a,b);
'undef SIZE
....
endmodule
```

→ 'timescale :-

* This directive specifies time unit & time precision for simulation of modules that follow it.
* The time unit specifies the unit of measurement for times & delays.
* The time precision value should be equal or smaller than the time unit value (PU ≤ tu).
* The time precision specifies how delay values should be rounded during the simulation.
* Both time precision & time unit should be made up of an integer number & a character string that represent a unit of time.

** The valid numbers are : 1, 10, & 100. The valid characters are : s, ms, us, ns, ps & fs.

* Actual time units & time precision values can be displayed by the "$printtimescale" system task.

* Syntax =>

'timescale time_unit unit / time_precision unit

- time_unit is the unit of measurement for time values, such as simulation time & delay values.
- time_precision is the minimum value (least count) for time. All time values are rounded off to the time_precision values.
- $printtimescale system task is used to display the time unit & precision for a module.

Eg> 'timescale 1ns/1ps

```
module timechk;
integer a;
initial begin a=1; #5.1 a=2, #7.23 a=3;
#1.514 a=4;
end
initial $monitor(a, $realtime); endmodule
```

o/p =>

a	$realtime	$time
1	0	0
2	5.1	5
3	12.33	12
4	13.84	14

- 1ns → 1000ps
- 'timescale 1ns/1ns ✓
- 'timescale 1ps/1ns ✗
- 'timescale 100ps/1000fs ✓
- 'timescale 10ns/110ps ✗

- `timescale 1ns/1ns

#0.3 // 0 ns 0.3 x 1ns => 0.3ns → 0ns

#2.3567 // 2ns 2.3567 x 1ns => 2.3567ns → 2ns

#0.5 // 1ns 0.5 x 1ns => 0.5ns → 1ns

- `timescale 1ns/1ps

#2.35 // 2.35ns 2.35 x 1ns => 2.35ns

// 2 ($time) 2.35 x 1000 => 2350ps (round)

2350ps => 2.35ns // (real time)

#0.3 // 0.3ns 0.3 x 1ns => 0.3ns x 100 => 300ps

// 0 ($time) 300ps => 0.3ns // (realtime)

- $time

#0.2 => integer x time unit & precision unit (0)

- $realtime

#0.2 => 0.2 x time unit & precision unit => integer (200ps)

#0.7325 // 0.733ns 0.7325 x 1ns => .7325ns x 1000

// 1ns ($time) 732.5ps => 733ps

733 / 1000 => 0.733

- % of returns value in the lowest unit mentioned.

- `timescale 1ns/100ps

#2.3567 // 2.4ns 2.3567 x 1ns => 2.3567ns

// 2 ($time) 2.3567 x 10 => 23.567ps

24ps => 2.4ns

100ps

→ Conditional Compilators :-

* In verilog few compiler directives are available to provide the conditional compilation. They allow the specified part of program text that will be compiled only if a specified condition is true.
* Nesting of these compiler directive is allowed
* The available compiler directives for conditional compilation are :
  - `ifdef  - `else - `endif
  - `ifndef  - `elsif
* `ifdef & `ifndef is always closed by a corresponding `endif.

Ex) `define AND
module logic_gate (input a, b, output c);

`ifdef AND
assign c = a & b;
`elsif OR
or (c, a, b);
`else
assign c = a ^ b;
`endif
endmodule

- `define upcount
module counter (input clk, output integer count = 0);
always @ (posedge clk)
`ifndef upcount
count <= count + 1;
`else
count <= count - 1;
`endif
endmodule

→ Conditional execution :-

* Conditional execution is simply control the execution of code at run time.
* $test $ plusargs & $value $plusargs system function is used for conditional execution.

* Syntax =>

$test$ plusargs ("string")
$value $plusargs ("user_string", value)

* $test $plusargs :-

- This system function is used when a value for the argument is not required.
- It searches the list of plusargs for a user defined string.
- The function will return non zero value if the supplied plusargs matches its all characters in the provided string. else it will return zero
- module sample ;
initial begin
if ($test $plusargs ("HELLO")) $display ("Hello executed");
if ($test $plusargs ("HE")) $display ("Hi executed");
if ($test $plusargs ("HELL")) $display ("Help executed");
end
endmodule

- Testing of $test $plusargs
Vsim -novopt sample +HELL
→ Hc executed
→ Hel executed

* $value$plusargs :-

- This system function also searches the list of plusargs like $test$ plusargs but it can take the value for a user defined string too.
- The function will return non-zero value if the supplied plusargs matches its all characters in the provided string, else it will return zero.
- module sample ;

```
integer dec ; reg [8*6:1] str;
initial begin
if ($value$plusargs("HELLO=%d", dec))
 $display("Hello executed dec=%d", dec);
if ($value$plusargs("HE=%s", str))
 $display("He executed str=%s", str); end
endmodule
```

- Testing of $value$plusargs

```
Vsim -novopt sample +HELLO=9
-> Hello executed dec=9
```

Timing Checks Tasks

* Verilog has certain constructs to perform common timing checks.
* Timing checks tasks are for verification of timing properties of designs & for reporting timing violations.
  Ex setup violation, hold violation etc.
** System timing checks only be used in specify blocks.
* All timing checks begin with $ sign they are not categorized as system tasks.
* No system task is allowed inside specify block.
* Timing check performs the following operations.
  - Records occurance time of a data or reference event
  - Waits for occurance of second data or reference event.
  - Compares the elapsed time to the specified limit.
  - Reports violation in design if any.
* Timing check tasks are invoked every time critical events occur within given time limits.

* Argument	Description	Type
• Reference_event	The transition of a control signal that establishes the reference time for tracking timing violations on the data_event	Module input or inout that is scalar or vector net.
• Data_event	The signal change that initiates the timing check & is monitored for violations	Module i/p or inout that is scalar or vector net.
• Limit	A time limit used to detect timing violations on the data_event	Constant expression or specparam.

* Verilog supports following timing checks:
  - $ setup
  - $ hold
  - $ setuphold
  - $ recovery
  - $ removal
  - $ recrem
  - $ width
  - $ period
  - $ nochange
  - $ skew
  - $ timeskew
  - $ fullskew

* $setup - checks setup time violation
  - $ setup (data_event, reference_event, limit);
  - (time of reference_event) - (time of data event) < limit
- Data event is a data signal
- Reference event is a clock signal
- limit is setup time.

Ex) module setup (din, clk, q);
  input din, clk;
  output q;
  always@ (posedge clk) q <= din
  • specify
    specparam tsetup = 7, delay = 10;
    (data 1 => q) = delay;
    $setup (din, posedge clk, tsetup);
  • endspecify
endmodule

* $hold - check the hold timing violation
  - $hold (reference_event, data_event, limit);
  - (time of data event) - (time of reference event) < limit.
- Data event is data signal, reference event is clock signal
- limit is hold time.

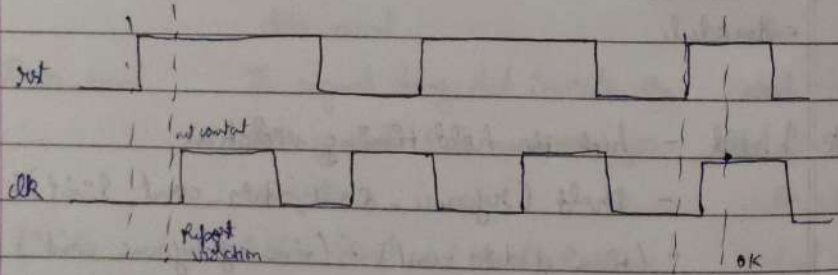
Ex) $hold (posedge clk, din, thold);

* $setuphold - $setuphold (reference_event, data_event, setup_limit, hold_limit);

Ex) specparam tsetup = 6, thold = 7, ddelay = 10;
(din => q) = ddelay;
$setuphold (posedge clk, din, tsetup, thold);

* $recovery - checks the violation for control signal like clear, reset, set etc.
- $recovery (reference_event, data_event, limit)
- Reference is a clear, reset or set
- data -> clock signal
- limit -> recovery time

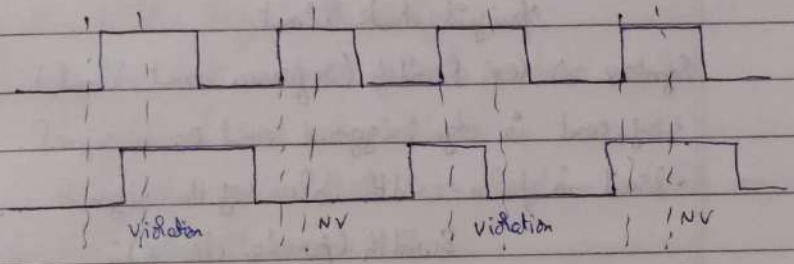
Ex) module recovery (in1, out1);
input in1;
output out1;
assign out1 = in1 ? 1'b1 : 1'bz;
specify
specparam trecovery = 10;
$recovery (posedge in1, out1, trecovery);
endspecify
endmodule



$recovery (posedge clear, posedge clk, 3)

* $removal - checks the violations for control signals like clear, reset
  - $removal (reference_event, data_event, limit);

ex) module dff (din, clk, clr, dout);
input din, clr, clk;
output reg dout;
always@ (posedge clk, posedge clr)
if (clr) dout <= 0 else dout <= din;
specify
$removal (negedge clk, posedge clk, 3);
endspecify
endmodule

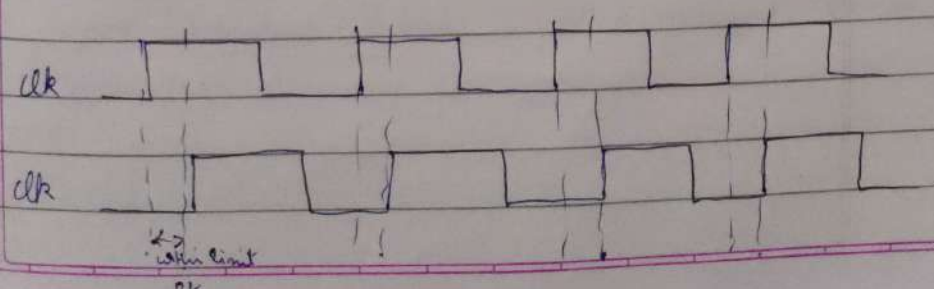


$removal (posedge clear, posedge clk, 3);

* $recrem - recovery & removal
  - $recrem (ref_event, data_event, recovery_limit, removal_limit);

* $skew - it is the time delta between actual & expected arrival time of a signal.
  - it checks that skew is not more than the specified limit w.r.t a signal

Syntax => $skew (ref_event, data_event, limit);

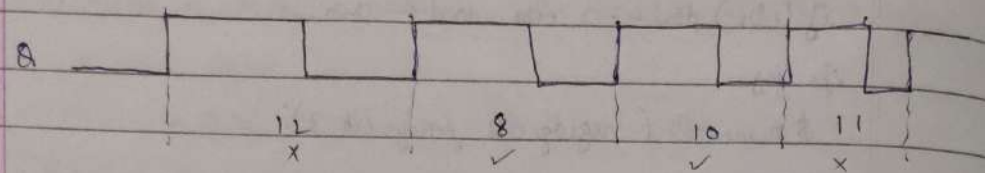


* $period - checks whether reference signal time period less than timing check limit

Syntax => $period (ref_event, limit);

- ref_event is edge triggered event on a signal
- limit is time period of the signal

$period (posedge clk, 10)

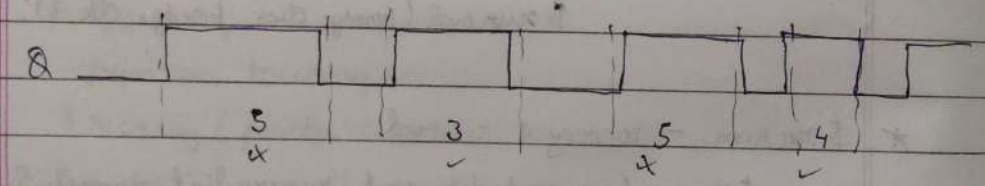


* $width - checks whether reference signal has width less than the specified check limit.

Syntax => (or $width (reference_event, limit);

- ref_event is edge triggered event on a signal
- limit is pulse width value of the signal

$width (posedge clk, 5);



Random Number Generator

* There are 3 approaches for verification
  - Directed verification (a=0; b=0; c=0; ...)
  - Randomized Verification ($random)
  - Constraint randomized verification ({$random}%4)

* Directed verification is a time consuming process
* Randomized verification will generate random value & we can't specify
* Therefore constraint randomized verification is widely used.

* Random numbers are used to provide random test inputs which helps in finding hidden bugs in the design.
* "$random" is system task used to generate random numbers.
** Syntax : $ random [(seed_value)];
  - seed_value is used to ensure that same random number sequence is generated each time it runs.
  - seed parameter can be reg, integer or time
  - * $random returns 32 bit signed integer

Ex) module random_gen;
  integer seq1, rg2, rg3, rg4, rg5;
  always begin
    rg1 = $random; // Generate a 32 bit signed integer
    rg2 = $unsigned ($random); // Generate unsigned integer
    rg3 = $random %50; // Generate signed integer in rang -49 to 49
    rg4 = $unsigned ($random)%100; // signed integer (0-99)
    rg5 = {$random}%35; // {3-unsigned (0-34)}
    #10;
  end
endmodule

- To generate b/w 5 to 15
  a = 5 + {$random}%11;
- -20 to 0
  a = -20 + {$random}%21;
- -20 to 5
  a = -20 + {$random}%26;

File Operations

* This is one of the good features that was added to verilog 2001.
* Using this features external files can be called in test-bench for reading in vectors & storing results of simulations for further analysis.
* The verilog has system tasks or functions that can open a file, save the values to file, read values from file & load values to other variables, close the file.
* The file operations majorly provide following features:
  - Reading characters from file from a fixed location
  - Reading lines from a file
  - Writing lines into a file.

* $fopen(file_name);
  $fclose(file_name);
  $fdisplay(arguments);
  $fwrite(arguments);
  $fstrobe(arguments);
  $fmonitor(arguments);
  $readmemb("file", memory_identifier [, begin_address [, end_address]]);
  $readmemh("file", memory_identifier [, begin_address [, end_address]]);

→ $fopen :-

* $fopen system task is used to open a file
* Syntax => file_handler = $fopen("name of the file.txt", "mode");
* $fopen returns a 32 bit value called multichannel descriptor.
* Only one bit of a multichannel descriptor is set to 1.
* Each bit of multichannel descriptor is used to represent different channel, except bit 0 (reserved).
* Standard output has a multichannel descriptor with bit 0 set to 1 (default condition).
* Standard output is also called as channel 0, which is open by default.
* Each successive $fopen call opens a new channel & returns 32-bit multichannel descriptor with next MSB bit set to 1.
* Multichannel descriptor offer advantage of writing multiple files at same time.

* File Opening modes :-
  - "r" - open the file in read mode
  - "w" - create the file for writing or over write the existing file
  - "a" - Open the file for appending
  - "r+" - open for both reading & writing
  - "w+" - Truncate or create the file
  - "a+" - Append or create new file for updating at EOF.

Ex > module file_test ;
initial begin
fp = $fopen ("test_file.txt", "r"); • open the file in read mode, store the file pointer in fp.
$fclose (fp);
end endmodule

-> Writing to file :-
- $fdisplay - similar function as $display, it writes to a file
- $fmonitor - similar function as $monitor, it writes to a file
- $fwrite - similar function as $write, it writes to a file
- $fstrobe - similar function as $strobe, it writes to a file

* All the above mentioned functions by default print the values in decimal, but they have three more versions to print the values in different radix.
  - $displayb - Binary
  - $displayo - octal
  - $displayh - hexadecimal

* Syntax => $fmonitor (file_descriptor, p1, p2 ... pn);
  - file descriptor specifies which file are to be written
  - p1, p2, ..., pn can be variables, constants or strings.

- $fmonitor(fil1, "value of a = %d & b = %d", a, b);
- $fmonitor((fil1 | fil2), "value of a = %d & b = %d", a, b); (Both files)
- fild1 = fil1 | 1; $fdisplay(fild1, "Hello");
  fild2 = fil2 | fil3; $fdisplay(fild2, "word");

Ex> module fil(a, b, c);
  input a, b;
  output c;
  integer f1;
  assign c = a + b;
  initial begin
  f1 = $fopen("fil1.txt");
  $fmonitor(f1, a, b, c);
  end endmodule

- module hhh;
  integer fil1, fil2, fil3;
  integer fild1, fild2, fild3, fild4, fild5
  initial begin
  fil1 = $fopen("fil1.txt"); // fil1 = 32'h0000_0002 (bit 1 set)
  fil2 = $fopen("fil2.txt"); // fil2 = 32'h0000_0004 (bit 2 set)
  fil3 = $fopen("fil3.txt"); // fil3 = 32'h0000_0008 (bit 3 set)
  end
  initial begin // 2 | 1 => 32'h0000_003
  fild1 = fil1 | 2; $display(fild1, "hello"); // writes to fil1.txt (Transcript
  fild2 = fil2 | fil3; $fdisplay(fild2, "world"); // 4 | 8 => 32'h0000_000c
  fild3 = 8; $fdisplay(fild3, "are"); // 4 => 32'h0000_0004
  fild4 = fil1 | fil2 | 1; $fdisplay(fild4, "you"); // writes to fil3
  fild5 = fil3 & 1; $fdisplay(fild5, "hello again"); // 2 | 4 | 1 => 32'h0000_0007
  end endmodule // writes to fil.txt, fil2.txt & transcript.

=> Memory Initialization:-

* Verilog provides a system task to initialize memories from a file.
* "$readmemb" & "$readmemh" are system tasks used to initialize memories.
* Syntax => • $readmemb("file_name", memory_name);
  • $readmemb("filename", memory_name, start_address);
  • $readmemb("filename", memory_name, start_add, end_address);

* $readmemb & $readmemh are used if the values present in file are in Binary or Hexadecimal format respectively.
* In initialization file, data is separated by whitespace
* Data can contain X & Z.
* Optionally address can be specified with @address. Here address has to be in hexadecimal format.
** If there are multiple entries for same address, the last entries will be considered
* If there is conflict between (start_add, end_add) & address provided in initialization file tool will report warning.

• module example:

```
integer i;                                  // whitespaces are used to separate data
reg [7:0] mem [0:7];                        // Data specified in Binary [$readmemb]
initial $readmemb ("file.txt", mem);
initial for (i=0; i<=7; i=i+1)
  $display ("value at location %d is = %b", i, mem[i]);
endmodule
```

file.txt

```
1001_0110   //L-0
1011_0100   //L-1
1010_0001   //L-2
1111_0010   //L-3
0111_1110   //L-4
1101_1110   //L-5
1001_1101   //L-6
0101_0111   //L-7
```

VCD System Tasks

* The VCD (Value change Dump) file format is specified in the IEEE 1364 standard.
* The VCD file is an ASCII file containing header information, variable definitions, & variable value changes.
* VCD is commonly used in Verilog designs, & is controlled by VCD system task call in the Verilog source code.
* The VCD files store information about value changes during simulation for nets & registers.
* The value change dumping function is more efficient than the $monitor task both in performance & in storage space.
* The files saved in the VCD format contain several sections that describe:
  - The file creation date & time.
  - Version number, $timescale directive & signals specification
  - $dumpvars task which defines which variable to dump into a file
  - Following syntaxes are used for VCD system tasks:
    - $dumpfile (filename)
    - $dumpvars
    - $dumpvars (level, list of variables)
    - $dumpoff
    - $dumpon
    - $dumpall
    - $dumplimit
    - $dumpflush

* $dumpfile - Used to create a VCD file.
* $dumpvars - Used to specify variables to be dumped.

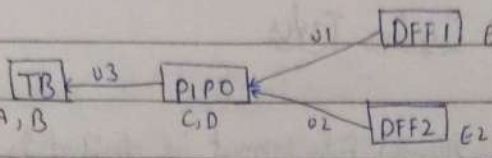
level → how many levels of hierarchy of modules has to be dumped to the VCD file.

1 → will dump all variables in current specified module, not module instantiated inside current specified module

0 → will dump all variables in current specified module & modules instantiated inside current specified module

2 → will dump current module & one level up.

Ex)



Dumps values of

- $dumpvars(1,TB); // A&B
- $dumpvars(1,TB.U3); // C&D
- $dumpvars(2,TB); // A B C D
- $dumpvars(0,TB); // All variables
- $dumpvars(1,TB.U3,A); // A, C & D
- $dumpvars(1,TB.U3.U1,TB.U3); // E1 & C D

→ Keywords inside VCD file :-

* $dumpall - is used to write current value of all selected variables.
* $dumpon / $dumpoff - is used to resume or stop dumping process.

  Syntax =>
  - $dumpall ; // write current value of all variables
  - $dumpoff ; // stop dumping
  - $dumpon ; // resume dumping
* $dumpflush - is used to empty VCD buffer & ensure all data in buffer is written to VCD file.
  - It is used to update the dump file while simulation is running.
  - $dumpflush ;
* $comment - Represents comment inside a VCD file.
  - $comment // This is single line comment $end
* $timescale - Specifies time scale which was used during simulation
  - $timescale time_number time_unit $end
* $date - specifies the date on which VCD file was created
  - $date date_in_text_format $end.
* $upscope - indicates change of scope to next higher level in design hierarchy.
  - $upscope $end

-> QuestaSim VCD features :-

* creating VCD file without using verilog system tasks :

```
• vsim -novopt testbench          // Start TB simulation
• vcd file vcd_filename           // create VCD file
• vcd add var1 var2               // Add variables to be dumped
• add wave *                      // Add wave window
• run time_value                  // Run simulation
```

* converting a VCD file to a waveform file

```
• vcd2wlf    vcd_filename    wlf_filename.wlf
```

Procedural Continuous Assignment

* A procedural continuous assignment assigns a value to a register continuously for specified duration.
* A procedural continuous assignment overrides any other assignment.
* A procedural continuous assignment is not the same as a continuous assignment.
* Procedural continuous assignment are declared inside procedural blocks.
* Used only for simulation
* Verilog provides two types of procedural continuous assignments:
  - Assign & Deassign
  - force & release

→ Assign & Deassign :-

* Assign & deassign works according to their name i.e assign simply overrides regular procedural assignments & deassign re-enables regular procedural assignments.
** It works only for register data type.
* After deassign last value remains on the register until a new procedural assignment changes it.

Ex)
```
module dff(input din, rst, clk, output reg q);
 always@(posedge clk)
  q <= din;
 always @(rust)
 if(rust) assign q = 1'b0;
 else deassign q;      //Next assignment at posedge clk.
endmodule
```

→ Force & Release :-

* The keyword "force" & "release" have the same effect as "assign" & "deassign" i.e force simply overrides regular procedural assignments & release re-enables regular procedural assignments.
* force & release can be used for nets, registers, bit-or part select of a net (not register), or a concatenation.
* Assignments in priority order
  - force
  - assign (procedural continuous)
  - Procedural / continuous assignments

Ex) Main module

```
module dff (din, clk, q);
  input din, clk;
  output reg q;
  always @(posedge clk)
    q <= din;
endmodule
```

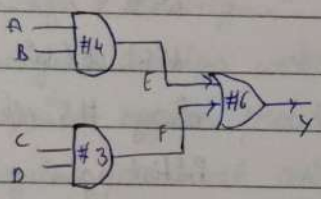
Test bench

```
module dff_test;
  reg din, clk=1;
  wire q;
  dff U0 (din, clk, q);
  always #5 clk = ~clk;
  initial begin
    force U0.q=0;
    #9 release U0.q;
    forever #10 din = $random;
  end endmodule
```

Modelling delays

* Functional verification of hardware only tells whether the functionality of design is correct or not, but in real hardware, logic elements & path have delays associated with them.
* Therefore, it is also required that design meets timing requirements.
* Verilog provides the following modeling delays:
  - Distributed delays - are specified for individual logic elements
  - lumped delays - are specified on per module basis
  - Pin to Pin delays - are assigned individually to paths from each input to output.

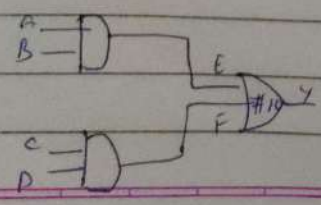
→ Distributed Delays:-




```
module dist_ddlay (g,a,b,c,d);
output g;
input a,b,c,d;
wire e,f;
// delay is applied to each gate
and #4 a1(e,a,b);
and #3 a2(f,c,d);
or #6 a3(g,e,f);
endmodule
```

→ Lumped Delays:- They are specified as a single delay on the output gate of the module

* The lumped delay takes into consideration the maximum delay from an input to output.



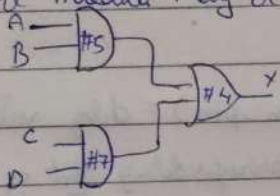

```
and a1(e,a,b);
and a2(f,c,d);
or #10 a3(g,e,f);
endmodule
```

-> Difference between Distributed & dumped delays:-

Distributed delays	dumped delays
• Difficult to implement	• Most easy to implement
• Delays are accurate	• Delays are less accurate
• All path may or may not have same amount of delay	• All path have same amount of delay.

-> Pin to Pin delays:-

* In pin-to-pin delay model, delays are assigned individually to paths from each input to output.
* Although pin-to-pin delays may look very detailed, but they are easier for designers, as it requires no knowledge of internal model.
** The model may be coded in Behavioral statements, dataflow, gates or mixed.



* These delays are also known as path delays.

path a-e-y, delay = 9
path b-e-y, delay = 9
path c-f-y, delay = 11
path d-f-y, delay = 11

* For bigger circuits this delay model is easier to implement as compared to distributed delay model.
* Designer needs to know the I/O pins of the module rather than the internals of the module so easier to model, even though it is very detailed
* In Verilog, path delays are specified using "Specify blocks".

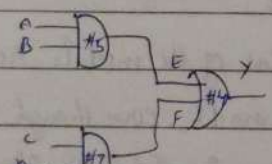
-> Specify Block:-

* The specify block describes paths across the module & assigns delays to those paths.
* It performs timing checks like setup & hold times.
** The specify block is declared inside the module declaration & bounded by the keyword specify & endspecify.

* Syntax => module xyz(a,b,c);
input a,b; output c;
always @(a or b)
end
specify
:
endspecify
endmodule

* There are two ways to describe pin 2 pin delays:
- Parallel connection
- Full connection

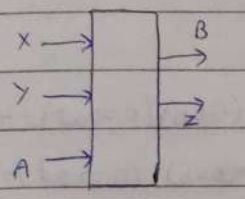
* Parallel connection :-
- Syntax : (source pin => destination_pin) = delay_value;
- Each bit of source connects to corresponding bit of destination.
- Parallel connection can be created between source & destination only if they are of same size.
- They only connect one source to one destination.



module parallel (a,b,c,d,y);
input a,b,c,d; output y; wire e,f;
and U1 (e,a,b); and U2 (f,c,d); or (y,e,f);
specify
(a => y) = 9; (b => y) = 9; (c => y) = 11;
(d => y) = 11;
endspecify
endmodule

* Full Connection :-
  - Syntax : (source_pin *> destination_pin) = delay_value;
* Each bit of source connects to every bit of destination
  - Source & destination need not be of same size
  - They can be used to describe path between :
    - a vector & a scalar
    - Vector of different sizes
    - multiple sources or multiple destinations

Ex>




```
module full(x,y,A,z,B);
input x,y; input [1:0] A; output z,B;
always block begin
 end
specify
(x, y *> z) = 7;        // (x=> z)=7; (y=> z)=7;
(A *> B) = 3;           // (A[0] => B)=3; (A[1]=>B)3;
endspecify endmodule
```

→ Specparam :-
* "specparam" keyword is used to declare special parameters inside specify blocks.
* They are used to specify pin to pin delay values & then these parameters are used instead of hardcoded delay values.
* These parameters remain local to a specify block
* Syntax => specparam parameter_name = delay_value;

```
module design (input a,b,c, output out);
  assign out = (a&b)|c;
specify
specparam aout = 2, bout = 4, cout = 3;
(a => out) = aout; (b => out) = bout; (c => out) = cout;
endspecify endmodule
```

* Min, Max & Typical delay can also be specified for pin to pin delays.
* Syntax => min : Typical : max
  Ex =>

```
specparam rise = 1:2:3, fall = 2:4:5, turnoff = 3:5:7;
(a=>b) = (rise, fall, turnoff);
```

-
```
module design (input a,b, output out);
assign out = a&b;
specify
if(a) (a=> out)=4;          // if (a&b) (a=>out)=4;
if (~a) (a=>out)=3;         // if (~(a&b)) (a=>out)=3;
endspecify endmodule
```

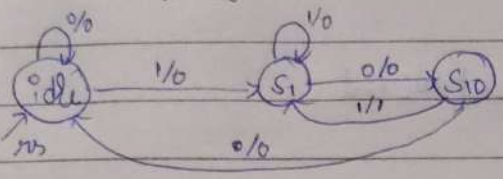
-
```
module design (input a,b,c, output out);
assign out = a^b;
specify
if( {a,b} == 2'b00) (c=>out)=4;
if( {a,b} != 2'b00) (c=>out)=1;
endspecify endmodule
```

-
```
module dff (input clk, din, output reg dout);
always @ (posedge clk)
 dout <= din;
 specify
 (posedge clk => (dout : din)) = (1,2);
endspecify
endmodule
```

Finite State Machine

→ 101 overlapping mealy :-

*



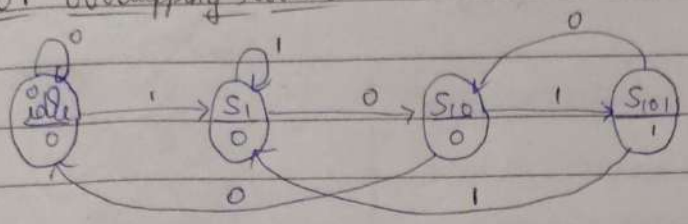
* module mealy_seq_101_OV (in_seq, clk, rst, dut_out);
input in_seq, clk, rst;
output reg dut_out;
reg [1:0] ps, ns; // Declaration for state variable
// State Encoding
parameter idle = 2'b00;
parameter S1 = 2'b01;
parameter S10 = 2'b10;
// Implement State Register
always @ (posedge clk or negedge rst) begin
if (!rst)
ps <= idle;
else ps <= ns;
end
// always block for NSD and OD
always @ (in_seq, ps) begin
case (ps)
idle : if (in_seq) begin
ns = S1; dut_out = 0; end
else begin
ns = idle; dut_out = 0; end
S1 : if (in_seq) begin
ns = S1; dut_out = 0; end
else begin
ns = S10; dut_out = 0; end

```
S10 : if(in_seq) begin
      ns = S1; dt_out = 1; end
      else begin
        ns = idle; dt_out = 0; end
default : begin
      ns = idle; dt_out = 0; end
endcase
end
endmodule
```

* //TESTBENCH

```
module seq_101_mealy_ov_test;
  reg in_seq, clk, rst;
  wire dt_out;
mealy_seq_101_ov dut(in_seq, clk, rst, dt_out);
 initial begin
 clk = 0; rst = 0;
 #12 rst = 1;
 in_seq = 1;
 #10 in_seq = 0;
 #10 in_seq = 1;
 #10 in_seq = 0;
 #10 in_seq = 1;
 #20 $finish;
 end
 always #5 clk = ~clk;
endmodule
```

→ 101 overlapping moore :-




```
module moore_seq_101_OV (in_seq, clk, rst, dut_out);
  input in_seq, clk, rst;
  output reg dut_out;
  reg [1:0] ps, ns;
  parameter idle = 2'b00;
  parameter S1 = 2'b01;
  parameter S10 = 2'b10;
  parameter S101 = 2'b11;
  always @ (posedge clk) begin
  if(!rst) ps <= idle;
  else ps <= ns; end
  always @ (in_seq, ps) begin
  case (ps)
  idle : if(in_seq) begin
    ns = S1; dut_out = 0; end
    else begin
    ns = idle ; dut_out = 0; end
  S1 : if (in_seq) begin
    ns = S1; dut_out = 0; end
    else begin
    ns = S10; dut_out = 0; end
  S10 : if (in_seq) begin
    ns = S101; dut_out = 0; end
    else begin
    ns = idle ; dut_out = 0; end
  S101 : if (in_seq) begin
    ns = S1; dut_out = 1; end
    else begin
    ns = S10; dut_out = 1; end
  default : begin
    ns = idle; dut_out = 0; end
  endmodule
```