Video-1

Introduction to Verilog HDL

What is hardware description language?

HDL stands for hardware descriptive language. It is a specialized computer language which is used to describe the structure and behaviour of electronic circuits. When we talk about HDL, we directly or indirectly talk about a piece of hardware. So, we can say that it is related to hardware circuits. In early 70's when we don't have HDLs, pen paper and drafting papers were used to describe the specifications of circuit. Now we have HDLs to simply translate our ideas.

Properties of HDL

HDL supports concurrency and includes notion of time. Notion of time deals with the delays of the circuit. In real time we have delays in the circuit design and HDL consists of the delays. Concurrency is nothing but parallel execution of two events or statements. In C or C++ Language we don't have concurrency i.e., they are sequential in nature.

Verilog HDL

Verilog is a hardware descriptive language with the current standard of IEEE 1364-200. Verilog is used to describe the low-level language. Verilog is a flexible language where we have pre-defined basic gates and digital circuits which are already saved in digital library of Verilog which are known as primitives which we can use directly. Syntax of Verilog is similar to that of C language.

Differences between Verilog and C language

Verilog	C
A Hardware Description Language(HDL) used to model electronic system	A general purpose programming language that allows structured programming
HDLs have a notion of time	Do not have notion of time
Helps to design and describe digital system	Helps to build operating systems, databases, compilers, interpreters etc.
Files have .v file extension	Files have .c file extension

What is VHDL programming language?

VHDL stands for very high-speed integrated circuit hardware description language, which is used to model the behaviour and structure of digital system at multiple abstraction levels, VHDL is strongly typed and deterministic language. VHDL is an old language and is developed in old 80's and syntax is not similar to C. The advantage of VHDL is we can easily fix the bugs. In VHDL we say it as a strongly typed and the code in VHDL is long as compared to Verilog.

Difference between Verilog and VHDL

Verilog	VHDL
Used to model electronic system	Used in electronic design automation to describe digital and mixed signal system
Verilog is weakly typed	VHDL is strongly typed

Based on C language	Based on Pascal and Ada language
Case sensitive	Not Case sensitive
Simple data types	More complex data types
Verilog is newer than VHDL	Older than Verilog
Less complex	More complex

Video-2

Level of Abstraction

Verilog supports 4 levels of abstraction namely,
- Switch level
- Gate level
- Data flow level
- Behavioural level

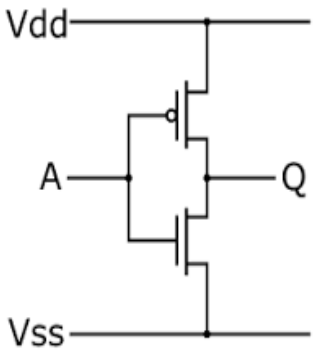
In switch level the implementation is done with the help of switches. In gate level we talk about gate interconnections. In data flow level design is done based on equations, which can be said that it is based on data flow, how the data is flowing in the circuit, based on that the equations have written. Behavioural level is based on behaviour of the circuit.

Switch Level: Module is implemented in terms of switches, we can represent the entire circuit as CMOS circuit. Here PMOS and NMOS are used as switches for the design.

Syntax: mos_name instance_name (output.data,control)

Example: CMOS inverter circuit

```
module inverter(Q,A);
        input A;
        output Q;
        supply1 vdd;
        supplt0 vss;
        pmos  p (Q,vdd,A);
        nmos  n (Q,vss,A);
endmodule
```



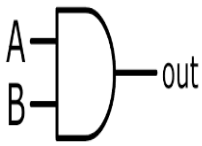
In syntax instance_name is optional. Consider the example of CMOS inverter, which has two supplies VDD and VSS; in the figure we can clearly state that PMOS is connected to Vdd and NMOS is connected to VSS. So based on the syntax PMOS input is A which is control, output is Q and supply is VDD so the line for pmos will be pmos (Q, VDD, A) Similarly, for NMOS as well.

Gate level: Module is implemented in terms of gates, which is lowest level of abstraction. Basic logic gates are available as predefined primitives. In digital library of Verilog these logic gates are already saved and can be used directly like and, or, xor, nand, nor etc.

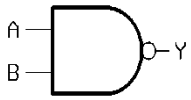
Syntax: Primitive_name instance_name (output, inputs)
Primitive_name is logic gate name and instance_name is optional.

Example:

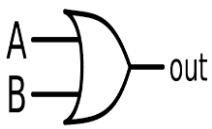
1. and G1(out,A,B);



2. nand G2(Y,A,B);



3. or G3(out,A,B);



Data flow level: Whenever RTL (Register transfer logic) is heard then data flow level comes into the action. At this level module is designed by specifying the data flow, like how the data is flowing in the circuit based on that the equation should be written. Signals are assigned by the data manipulating equations, design is implemented using continuous assignment i.e. assign statement in used in this level, the assignments present in it are concurrent in nature.

Keyword: “assign”

Example:

assign z = x & y; ----->x and y is assigned to z
assign p = q | r; -----> q or r is assigned to p

Behavioural level: This is the highest level of abstraction provided by HDL, this level describes the behaviour of the system. Different elements like functions, tasks and blocks can be used, two important constructors are initial and always.

Example: 2*1 MUX

```
always @(i0,i1,s)
begin
 if(s) // if select line is true then output is i1
        out = i1;

 else // if select line is false then output id i0

        out = i0;

end
```

Video-3

Modules and instantiation

Module: Module is the basic building block in Verilog. Consider an example of construction of house here the basic block is brick, with the help of brick we construct the house, similarly in Verilog module has the same purpose, it can be an element or a collection of lower-level design block. Module provides the required functionality to the higher level block through its port interface but hides the internal implementation. In other words modules provide the flexibility to the designers to modify the module internals without affecting the rest of the design.

Verilog framework/syntax: The basic format of the Verilog code

```
module module_name (x,y,z);

        input x,y;

        output z;

        statements;

        ……..;
```

…….;

endmodule

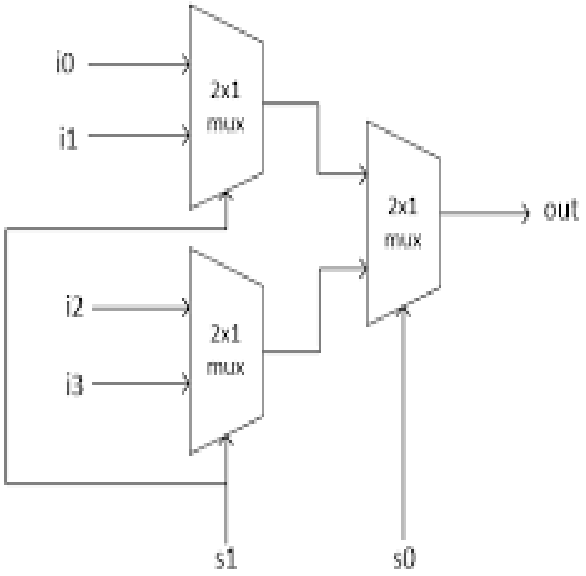
Verilog code starts with module and ends with endmodule. Here module signifies the declaration of the module and endmodule signifies the termination of the module. After module, module name is written based upon the circuit with any name. In parenthesis inputs and outputs are written i.e., list of ports, then declare the ports whether it is input, output or inout port, after declaring the ports, statements should be written depending upon the level, which states the functionality of the circuit. Semi-colon represents the termination of the line.

Instantiation: Instantiation allows the creation of hierarchy in Verilog description. With the help of sub-modules main module can be built i.e. top block with the help of sub-modules which is hierarchy, which can be done using instantiation. The process of creating object from a module template and the objects are called instances. Consider an example where full adder is implemented using half adder. First module for half adder will be written and then we instantiate the half adder module two times to create a copy of half adder circuit. In Verilog nesting of module is illegal i.e. defining one module in another module.

Example: 4*1 MUX using 2*1 MUX

Verilog code for 2*1 MUX

```
module mu(i1,i2,s0,out);
        input i1,i2,s0;
        output out;
always@(i2,i1,s0)
begin
        if(s0)
                out = i1;
        else
                out = i0;
end
endmodule
```



Now 4*1 mux using 2*1 Mux

```
module mux_4(i0,i1,i2,i3,s0,s1,out);
        input i0,i1,i2,i3,s0,s1;
        output out;
        wire a0,a1;
        mu m1(i0,i1,s0,a0);
        mu m2(i2,i3,s0,a0);
        mu m3(a0,a1,s1,out);
endmodule
```

Verilog Keywords

Verilog is case sensitive in nature, “$”, “_”, alphanumeric characters can be used, The proper format for writing keyword is having a word starting with lower case like module_new, module_new1, module$new. The incorrect format is stating key word with “$”, “_” or with numbers.

- new_block
- new_block1
- new_block$

- 1new_block
- $new_block

Comments

Comments are basically used for documentation purpose, which improves the readability of the code. It is recommended to use comments to know why we are writing the particular statement.

There are two types of comments:

• Single-line comment – It starts with "//" . Verilog skips from that point to end of the line

• Multiline comment– It starts with "/*" and ends with "*/", Multiline cannot be nested.

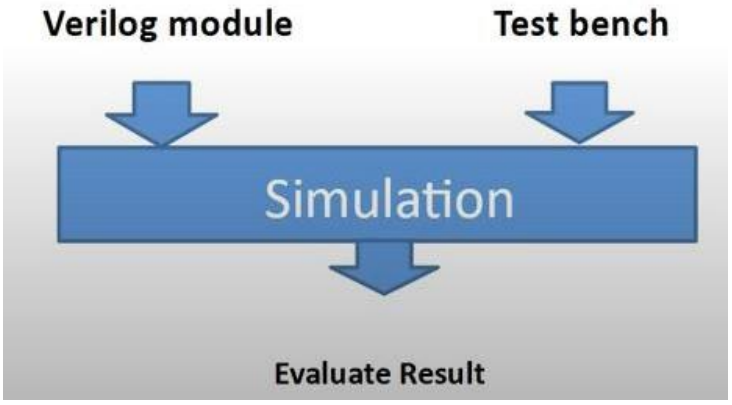
Example:

```
assign z = x ^ y; // z is equal to x^y

assign r = p & q; /* This is multiline
                     comment */
```

Video-4

Simulation, Synthesis and design methodologies

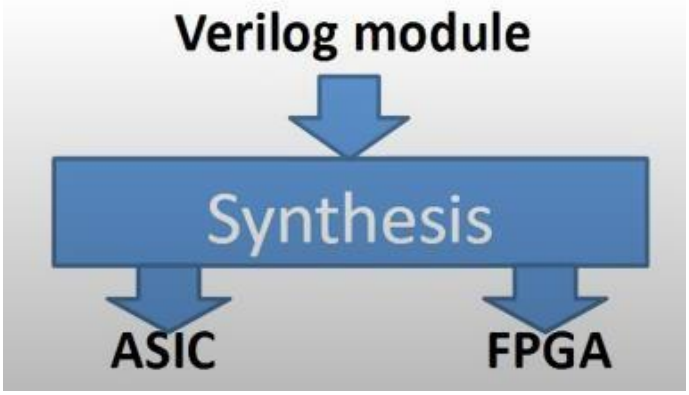
Simulation



It is used to verify the functionality of the digital design that is modelled using HDL like Verilog. If we have an RTL, then with the help of simulation we can check whether the code is working correctly or not and for that simulation we need to write

the testbenches. In testbench some custom inputs are provided namely, stimulus, the output that is generated with the custom inputs, evaluation is done based on that output waveforms and the complete process is called as simulation. For simulation purpose we apply the different inputs to the design at different times, so that to check whether the RTL code behaves in intended way or not. Verilog module and testbenches are required and based on the outputs, for the inputs results are evaluated.

Synthesis

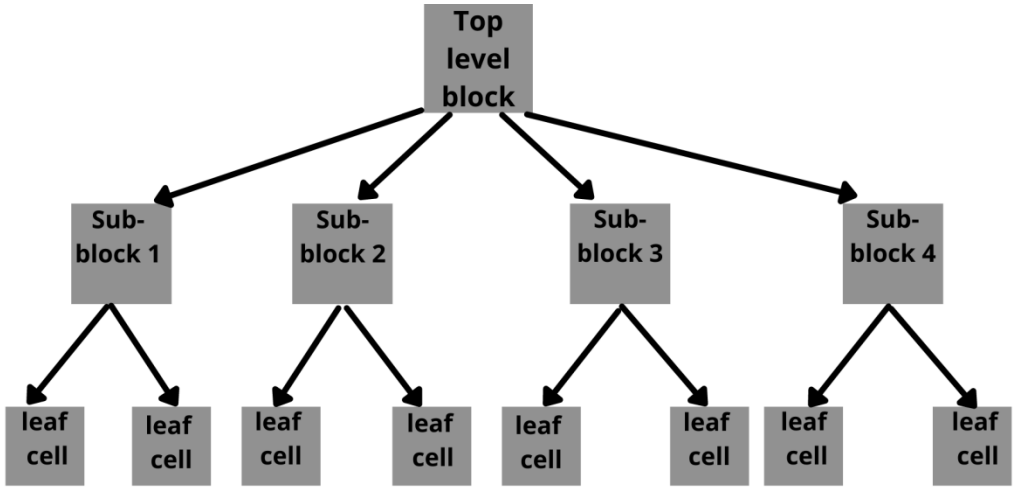


Synthesis is a process in which the digital design that is modelled using HDL is translated into an implementation consisting logic gates or in other word synthesis converts RTL into netlist. It will just make an optimal design based on the working strategy which are using and also give the consumption of resources are available. For synthesis we only need Verilog module and no testbench is required. Either ASIC or FPGA used to perform the synthesis operation. Since there are no testbench, we don't have any concept of time delay in synthesis as already the functioning on hardware so time delay is only considered in simulation and not in synthesis process.

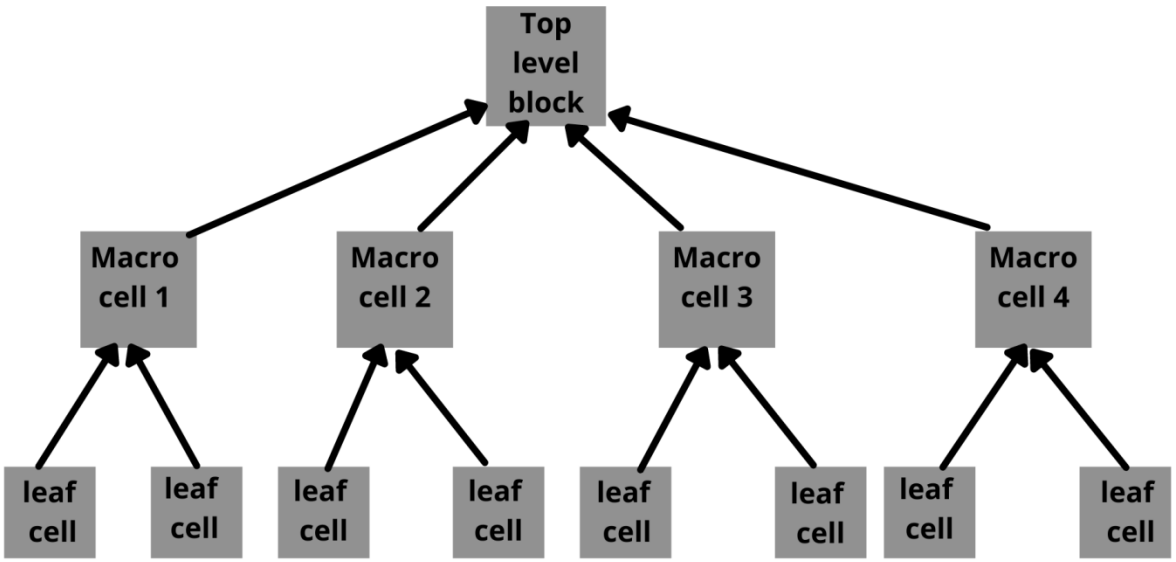
Design Methodologies

In Verilog we have two types of methodologies:

• Top-down design methodology



• Bottom-up design methodology



In top-down the flow of design from top to bottom, starts from top level block and then dividing each and every block into sub blocks until we reach to the leaf cell i.e., the smallest possible block. In bottom up design, start is initiated from bottom level i.e., from the lowest possible block and then combine those blocks to make the macros these macros are again combined to make top level block so as seen the flow is from bottom to top.

Video-5

Data Types

Data type is a classification that specifies which type of value a variable has, what type of mathematical operation i.e., logical or arithmetical is applied to that variable so that it doesn't cause any error in the system.

VERILOG supports two types of data types:

• Register

• Net

Register Data type

Register data types are used to store the values, it only holds the value but it is not a hardware register. Keyword used for register data type is "reg". The default value of reg is X (unknown). In register we have different classes like reg, integer, real, time etc.

Net Data type

Net represents the connection between hardware elements and must be continuously driven i.e., cannot be used to store the values. Keyword used for net data type is "wire" and default value is Z. Net represents different classes such as wire, wand, wor, tri, triand etc.

Values and signal strength

Verilog supports 4 different value levels and these are 0, 1, X, Z.

0 ----------logic zero/false condition

1 ----------logic one/true condition

X ---------- Unknown Logic value

Z ---------- High impedance, floating state

8 different strength levels

The strength level increase from bottom level to top level. When two signals are applied to the input of wire then the higher signal level is reflected at the output. When signal strength of the same level is applied then the output is unknown.

Strength Level	Type	Degree
supply	Driving	Strong
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	Weak

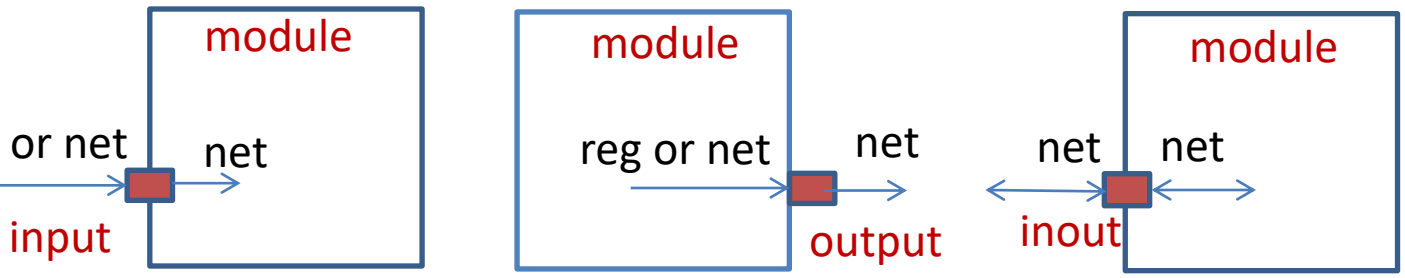
Port Assignments

Three different types of ports

Input: Unidirectional, within the module it is of wire data type and externally when it is assigned to this port then it can be reg or net data type

Output: Unidirectional, internally it is of reg or net data type, externally it is of net data type

Inout: Bidirectional, only wire data type.



Video-6

Net data type

Net represents the connection between hardware elements and must be continuously driven i.e., cannot be used to store the values. Keyword is wire and default value Z. Net represents different classes such as wire, wand, wor, tri, triand etc. Consider an AND gate with inputs a and b, output c. The data type for output c is NET i.e., it is continuously driven, as the input value changes the output value also changes accordingly and doesn't store the value. Consider a HALF-ADDER circuit with inputs a and b and sum as sum and carry as carry, the Verilog code for Half-adder is as follows:

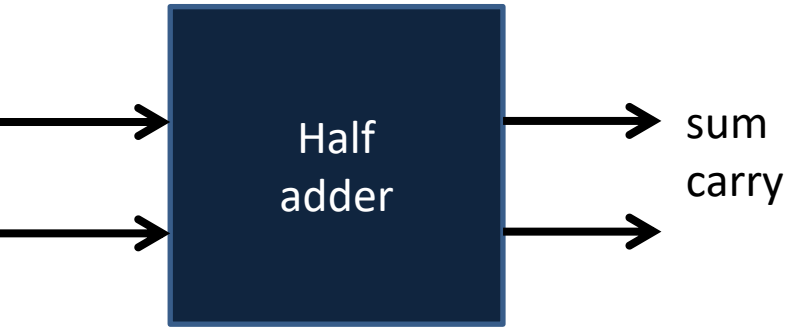
```
module half_adder(a,b,sum,carry);
        input a,b;
        output sum,carry;
        wire sum,carry;
        assign sum = a^b;
        assign carry = a&b;
endmodule
```

Sum and carry are taken as net data type because these are continuously driven and as stated previously for assign statement the left variable should be a net data type.

Use of wand class: Consider the following circuit

In the above circuit the final output is Y, if we use simple wire data type the output will then it will give indeterministic value as the output value y is assigned by two different values i.e., A & B and C | B where simulator will get confused that which value it needs to take for assigning the output Y. SO when wand or wor is used then there will be no problem. In the above example the if the ouput is used as wand then the output will be of and of A&B and C|D the output will be more accurate. Similarly wor can be used.

Example: Consider a HALF-ADDER circuit with inputs a and b and sum as sum and carry as carry, the Verilog code for Half-adder is as follows:

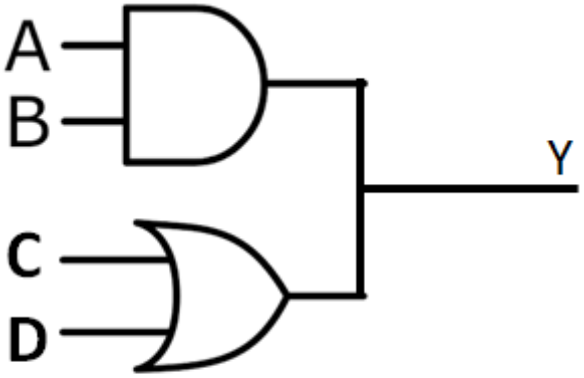



```
module half_add (a,b,sum,carry);
        input a, b;
        output sum, carry;
```

```
    wire sum, carry;
    assign sum = a ^ b ;
    assign carry = a & b ;
endmodule
```

Sum and carry are taken as net data type because these are continuously driven and as stated previously for assign statement the left variable should be a net data type.

Use of wandclass: Consider the following circuit:



Y = (A & B) & (C | D)

Example:

```
module use_of_wire (y,A,B,C,D);
    input A,B,C,D;
    output y;
    wire y;
    assign y = A & B;
    assign y = C | D;
endmodule
module use_of_wire (y,A,B,C,D);
    input A,B,C,D;
    output y;
    wand y;
    assign y = A & B;
    assign y = C | D;
endmodule
```

Supply1 and supply0: Consider the following Verilog code

```
module using_wire_supply (Y,A,B,C,D);
    input A,B,C,D;
    output Y;
    supply1 vdd;
    supply0 gnd;
    or G1 (x1, A, gnd);
    and G2 (x2, B, vdd );
    xor G3 ( Y,x1,x2);
endmodule
```

Here inputs are A,B,C,D and output is Y. supply1 vdd and supply0 gnd, different logic gates or, and, xor.

Supply1 and supply0: Consider the following Verilog code

```
module using_wire_supply (Y,A,B,C,D);
    input A,B,C,D;
    output Y;
    supply1 vdd;
    supply0 gnd;
    or G1 (x1, A, gnd);
    and G2 (x2, B, vdd );
    xor G3 ( Y,x1,x2);
endmodule
```

Here inputs are A,B,C,D and output is Y. supply1 vdd and supply0 gnd, different logic gates are or, and, xor.

Numbers

Representation of numbers (Syntax) in Verilog code: **<size>'<base><number>**

Size ---------- Total number of bits

Base ---------- Base of the number

Number ---------- Actual number

Examples:

```
1'b1          //logic-1 (1-bit)
8'h21         //8 bit number 0010 0001
12'ha35       //12 bit number 1010 0011 0101
12'o42xx      //12 bit number 100 010 xxx xxx
4'd1          //4 bit number 0001
35            //signed number , in 32 bits(size is not specified)
```

Representation of bases:

Binary – b or B

Octal – o or O

Decimal – d or D

Hexadecimal – h or H

Use of "_" (underscore) is used to improve the readability of the number it is mainly used for higher bit numbers such as 32 bit or 64-bit numbers.

Extensions:

The basic concept of extension is that whatever the total bits are present we have to complete the number of bits i.e., size so when the actual number doesn't consist of those number of bits then extension will be used.

Video-7

Reg Data Type

Verilog supports following register data types-

- **reg** (most widely used)
- **integer** (used for loop counting)
- **real** (used to store floating point numbers)
- **time** (keeps track of simulation time)

These reg data types are used to store the values. It only holds the value but it is not a hardware register,

keyword used is reg. The default value of reg is X (unknown). In register we have different classes like reg, integer, real, time etc.

Example of reg data type:

Declaration of reg can be of single bit or multiple bits

Syntax:

reg count; // single bit reg variable

reg [7:0] bus // multi bit (8-bits)

Consider the following Verilog code

```
reg reset;
initial
begin
        reset = 1'b0;
        #10 reset = 1'b1;
end
```

In the above Verilog code we have a register data type reset, initially the reset value is 0 and then for 10 time units it changes it values, the variable holds the value 1 until it will get the another value at additional time units.

Integer data type

It is a general-purpose data type, where the key word is "integer". It stores the values as signed quantities i.e., we can assign both the positive and negative value numbers to the integer data type. It is used for counting in nature and is synthesizable in nature. Default value is X (unknown) and default bit size is 32 bits.

Example:

```
integer count, new_num;
initial
begin
        count = 4; n
        new_num = -17;
end
```

From the above example, integer can be declared in a given syntax and assignment of a value to that variable is also shown.

Real Data type

Keyword used to declare a real data type is real. It allows decimal and scientific notation. It is mainly used to store the floating-point numbers. It is non-synthesizable in nature and the default value is 0.0. When integer is assigned to real variable in that case the value is rounded off to a nearest integer.

EXAMPLE:

```
real count, new_num;
initial
```

```
begin
        count = 4.5;
        new_num = -1.7;
end
integer count,num;
initial
begin
        count = -4.5;
        num = count; // num value will be -5
end
```

Time Data type

It is a special time register data type which is used to store the simulation time, keyword used is “time”. Time variables are unsigned in nature and width is generally 64 bits, it is non synthesizable in nature and default value is X.

EXAMPLE:

```
time new_time;
initial
begin
new_time = $time; // current simulation time will be stored
end
```

Video-8

Vectors, Arrays, Parameters and strings

Vectors

Nets or reg data types are declared as vectors i.e., we can declare these two data types with single bit or multiple bits. If bit width is not defined then the default size is 1 bit which is a scalar. Vector represents the buses. Vectors are declared by specifying the range [r1 : r2] where r1 is MSB and r2 is LSB.

EXAMPLE:

```
wire x, y, z; // single bit variable
wire [15:0] data; // 16-bit data
wire [0:31] sum; // 32-bit data where 0 is MSB and 31 is LSB
reg [7:0] d1,d2,d3; //3 buses of 8 bit each
```

Arrays

In Verilog arrays are allowed for reg, integer, time, real, vector register data types. Multi-dimensional arrays are declared with any number of dimensions.

Example:

```
reg [7:0] a[15:0]; // here there are 16 reg with 8 bits in each reg
```

reg b[31:0]; // there are 32 one bit numbers

reg c[0:1][0:3] /*Two dimensional array where left vector represents the rows and the right vector represents the column */

Memories

Memories are modelled as one-dimensional of registers, these are basically used to model register files, ROMs and RAMs. Each element of the array is known as an element or word. We can address these elements by the index number which can be of one bit or multiple bit.

Example:

reg mem [0:1023]; // memory mem with 1k 1-bit words

reg [7:0] mem [0:1023]; // memory mem with 1k 8 bit words

Parameters

Parameters cannot be used as a variable as it is constant value, advantage of using parameter is that, if a variable is declared as a parameter then it will help to customize the code.

Example:

parameter width_new = 8; //Defines width_new as a constant value 8

parameter depth_new = 256; //Defines depth_new as a constant value 256

String

Strings are stored in reg, strings are all about characters i.e., sequence of characters. In string each character is represented by 8 bits and these sequence will be enclosed by **" "** .

Example:

reg [8*10:1] str; //Declaration

Str = "VLSI POINT"; //assignment

Some special characters of string are newline, tabs and other displaying argument values.

Special Characters	Characters displayed
\n	New line
\t	Tab
\%	%
\\	\
\"	"

Video-9

Operators in Verilog

In any programming languages like C, C++, Java there are some operators which are built-in. In that language it can operate

on data items like number, character and some values. In Verilog there are some set of operators which are similar to high level languages. Some operators in Verilog are more specific which have direct connection with hardware that Verilog supports to model. Some of the operators are:

- Arithmetic operators
- Logical operators
- Bitwise operators
- Equality operators
- Relation operators
- Reduction operators
- Shift operators
- Concatenation
- Conditional

Arithmetic Operators

There are two types of arithmetic operators:

➢ **Unary**: The operators + and – can work as unary operators. It is used to determine whether the given operand is positive or negative.

Eg: -4 // negative 4
5 // positive 5

➢ **Binary**: In binary 2 operands are considered and then perform different binary operations.

Operator symbol	Operation performed	Number of operands
+	Add	2
-	Subtract	2
*	Multiply	2
/	Divide	2
%	Modulus	2
**	Power	2

A = 4'b0010; B = 4'b0011; A and B are register vectors.

(D = 7; E = 4; F = 2) D, E and F are integers.

A * B // Multiply A and B. Evaluates to 4'b0110

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.

A + B // Add A and B. Evaluates to 4'b0101

B - A // Subtract A from B. Evaluates to 4'b0001

F = E ** F //E to the power F, yields 16

Example:

If any operand bit has a value x, then the result of the entire expression is x.

a = 4'b001x ; b = 4'b1011 ;

sum = a + b ; // sum will be evaluated to the value 4'bx

Modulus operators produce the remainder from the division of two numbers.

17 % 4 // Evaluates to 1

16 % 4 // Evaluates to 0

-7 % 2 // Evaluates to -1, takes sign of the first operand

7 % -2 // Evaluates to +1, takes sign of the first operand

Logical Operators

Logical operators are used to define the conditional statement where the result will be true or false. Different types of logical operators.

Operator symbol	Operation performed	Number of operands
!	Logical negation	1
&&	Logical and	2
||	Logical or	2

Example:

Consider **A =5, B=0**

A && B // Evaluates 0 as one of the operands is low

A || B // Evaluates 1 as one of the operandsis high

!A // Evaluates 0 as A is high Consider operands with unknown values.

!B // Evaluates to 1. Equivalent to not(logical-0)

Unknowns: **A = 2'b0x; B = 2'b10;**

A && B // Evaluates to x. Equivalent to (x && logical 1)

So, if one or both the operands (even one of the bit) consists of unknown values then the output will be unknown.

Bit wise operator

Bit wise operators performs the operation between the bits i.e., it takes each bit from operands and then performs the operation.

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Different types of bitwise operators:

Operator Symbol	Operation performed	Number of operands
&	Bitwise and	2
|	Bitwise or	2
~	Bitwise negation	1
^	Bitwise xor	2
~^ or ^~	Bitwise xnor	2

Example:

Consider **X = 4'b1010; Y = 4'b1101; Z = 4'b10x1**

~X // negation of X which results in 4'b0101

X&Y // bitwise and which results in 4'b1000

X&Z // bitwise and which results in 4'b10x0

Equality Operators

Two different types of logical operators are there Logical and case.

- Logical equality (==)
- Logical inequality (!=)
- Case equality (===)
- Case Inequality (!==)

Logical equality is synthesizable in nature and case equality are non synthesizable in nature.

The result of equality operator is either logic 0 or logic 1 i.e., of single bit only. The major difference between logical and case equality is that logical equality doesn't include X or Z whereas case equality includes X or Z.

Example:

Consider A=4; B=3; X= 4'b1010; Y=4'b1101;
Z=4'b1xxz; M = 4'b1xxz; N = 4'b1xxx;
A==B // Results in logic0.
X==Z // Results in x

Z === N //results in logical 0 (LSB doesn't match)

M !== N // results in logical 1.

Relational Operators

There are 4 types of relational operators:

- Greater than (>)
- Less than (<)
- Greater than or equal (>=)
- Less than or equal (<=)

It returns a logical value either true or false i.e. a single bit output value.

Example:

Consider A=4; B=3; X=4'b1010; Y=4'b1101;
Z= 4'b1xxx

A<=B //Results in logical0

Y>=X //Results in logical1

Y<Z // Results in x.

Consider operands with unknown values. So, if one or both the operands (even one of the bit) consists of unknown values then the output will be unknown only.

Reduction operator

Different types of reduction operatorsare:

- And (&)
- Or (|)
- Nand (~&)
- Nor (~|)
- Xor (^)
- Xnor (~^)

It takes only one operand, reduction operator performs a bitwise operation on a single vector operand and yield a 1- bit result

Example:

Consider X =4'b1010

&X // Equivalent to 1 & 0 & 1 & 0. Results in 1'b0

|**X** // Equivalent to 1 | 0 | 1 | 0. Results in 1'b1

^**X** // Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0

A reduction xor or xnor can be used for even or odd parity generation of a vector.

Shift Operator

There are 4 different types of shift operators.

- Shift right(>>)
- Left shift (<<)
- Arithmetic left shift(<<<)
- Arithmetic rightshift(>>>)

Regular shift operators shift a vector operand to the right or to the left by a specified number of bits. When the bits are shifted, the vacant bits are filled with zeros.

Arithmetic shift operator uses the context of the expression to determine the value with which we can fill the vacated bit i.e., signed bit is present or not is verified according to that vacant bit is filled.

Example:

Consider X=4'b1100;

Y = X>>1 // Y is 4'b0110 shift right by 1 bit MSB is filled by bit 0

Y = X<<2 // Y is 4'b0000. Shift left by 2 bits.

Y = X>>>1 // Y is 4'b1110

Concatenation operator

The concatenate operator is ({,}). It provides the mechanism to append the multiple operands, which are sized, unsized operands cannot be used. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select or sized constants.

Example:

Consider A= 1'b1; B=2'b00; C=2'b10; D=3'b110

Y = {B,C} // Y results in 4'b0010 Y={A,B,3'b001}//Y results in 6'b100001

Y = {A,B[0],C[1]} //Y results in 3'b101

Y = {A,2{B},3{C}} // Y results in 11'b10000101010

Conditional Operator

Conditional operator consists of 3 operands and the syntax used is:

Conditional_expression ? true_expression : false_expression

First the conditional expression is evaluated, if the evaluated conditional expression is true then the true expression is evaluated else the false expression is evaluated.

Conditional operator can be nested.

E.g. assign out = (A==3) ? (control ? (x : y) : (control ? m:n))

Conditional operator is similar to that of multiplexer.

Consider a 2*1 Mux with select line as conditional expression then if the select line is high i.e., the conditional expression is true then the high output will be evaluated i.e., true statement will be the output else output will be low i.e., false expression.

Operator precedence

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

Video-11

Gate level Modelling

In Verilog there are some pre-defined gate primitives. Gate level modeling is the lowest level of abstraction. All logic gates can be implemented using the gates.

There are two different types of gates:

1. Basic Gates
2. Bufif/ Notif Gates

Basic Gates

In these gates one scalar output and multiple scalar inputs are present. Output is evaluated as soon as the input changes.

The and/or gates available in Verilog are: **And, or, xor, nand, nor, xnor**

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

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

or	i1: 0	1	x	z
i2: 0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	i1: 0	1	x	z
i2: 0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	i1: 0	1	x	z
i2: 0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	i1: 0	1	x	z
i2: 0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Synatx:

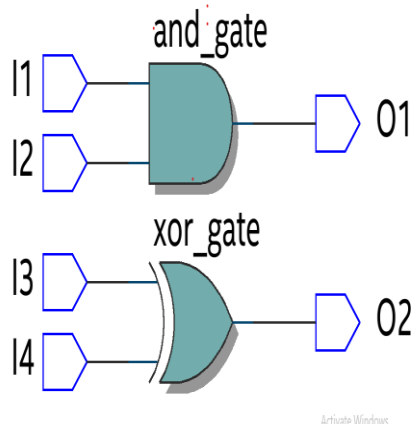
Primitive_name instance_name(output,inputs);

Example:

and G1(out,A,B); // A,B are inputs and out is output

nand G2(out,A,B); // A,B are inputs and out is output

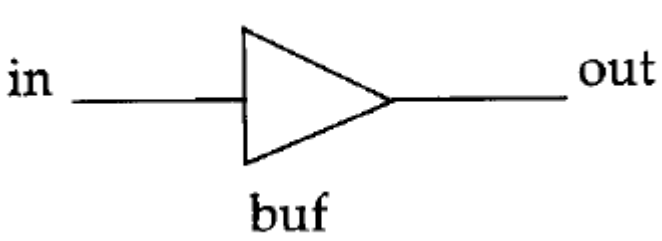
Consider AND gate with inputs I1,I2 and output O1 and OR gate with inputs I3,I4 and output O2 then the Verilog code would be written in this manner:



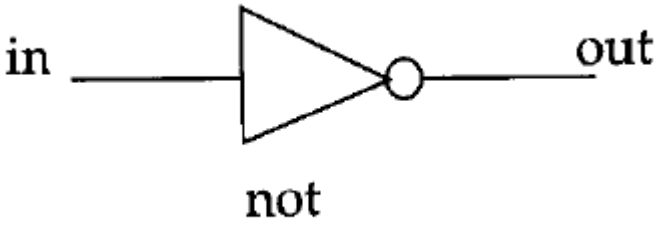

```
module gate_ex(O1,O2,I1,I2,I3,I4);

        input I1,I2,I3,I4;
        output O1,O2;
        and a (O1,I1,I2);
        or b (O2,I3,I4);
endmodule
```

Buf/Not Gates

These gates have one scalar input and one or more scalar outputs. Below image shows the buffer and not gates



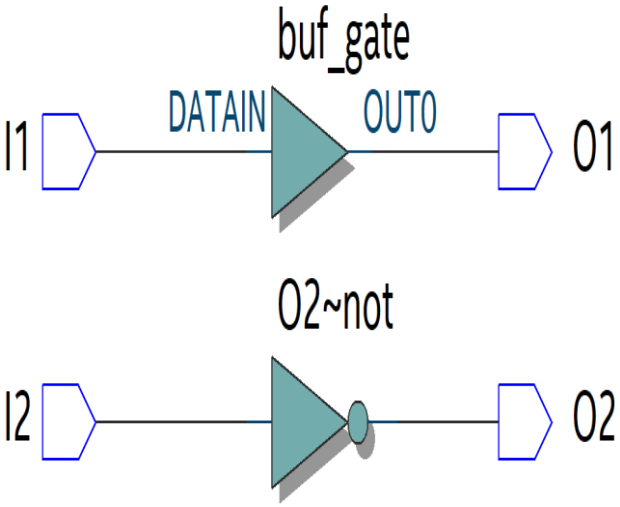
buf	in	out
	0	0
	1	1
	x	x
	z	x



not	in	out
	0	1
	1	0
	x	x
	z	x

Syntax: gate_name (output, input)

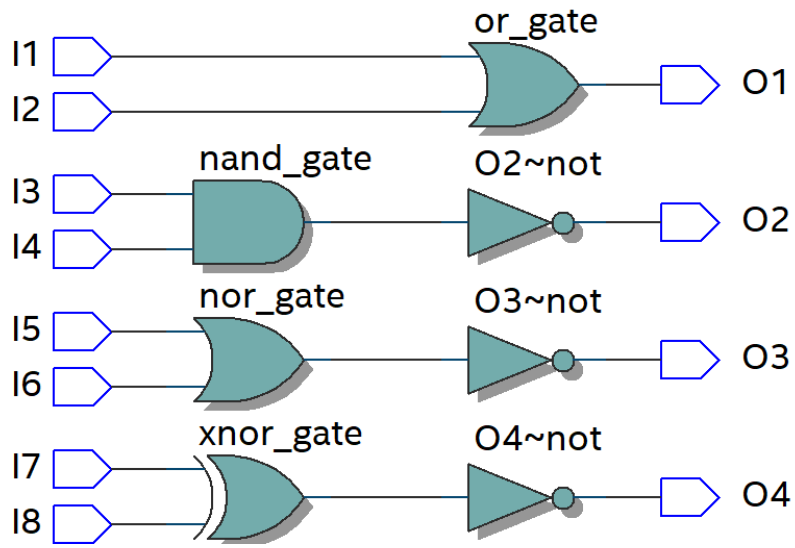
Example:




```
module gate_eg(O1,O2,I1,I2);
        input I1,I2;
        output O1,O2;
        buf buf_gate(O1,I1);
        not not_gate(O2,I2);
```

endmodule

Example:

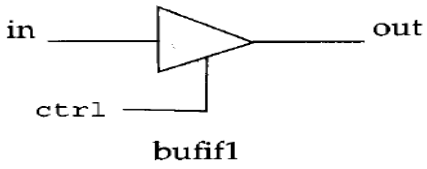



```
module gate_eg (O1,O2,O3,O4,I1,I2,I3,I4,I5,I6,I7,I8);

        input I1,I2,I3,I4, I5,I6,I7,I8;

        output O1,O2,O3,O4;

        or(O1,I1,I2);

        nand(O2,I3,I4);

        nor(O3,I5,I6);

        xnor(O4,I7,I8);

endmodule
```

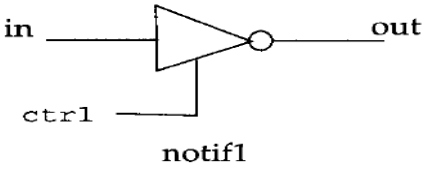
Bufif/Notif gates

These gates have additional control signals, The input propagate if the control signal is given otherwise the output will be in high impedance state. There are 4 different types of bufif notif gates bufif1, bufif0, notif1, notif1.

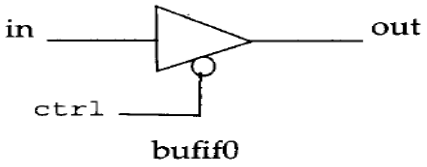
Below image shows the following 4 gates:



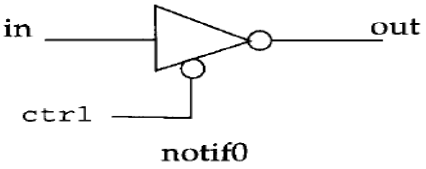
bufif1



notif1



bufif0



notif0

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

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

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

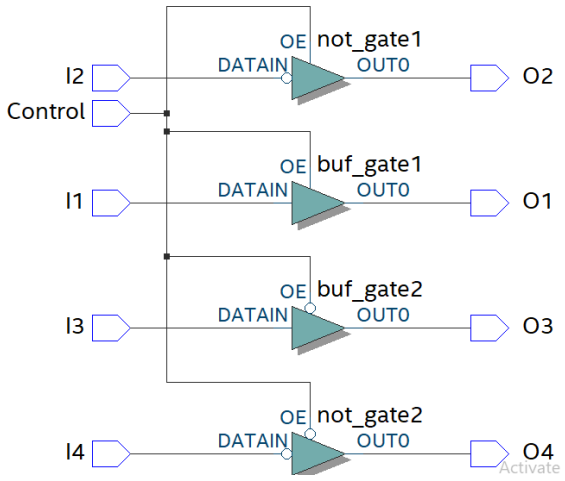
in \ notif0	ctrl 0	1	x	z
0	1	z	H	H
1	0	z	L	L
x	x	z	x	x
z	x	z	x	x

Syntax: gate_name (output, input,control)

Example:

```
module gate_eg(O1,O2,O3,O4,,I1,I2,I3,I4, Control);

        input I1,I2,I3,I4,Control;

        output O1,O2,O3,O4;

        bufif1 buf_gate1(O1,I1,Control);

        notif1 not_gate1(O2,I2,Control);

        bufif0 buf_gate2(O3,I3,Control);
```

```
    notif0 not_gate2(O4,I4,Control);
endmodule
```



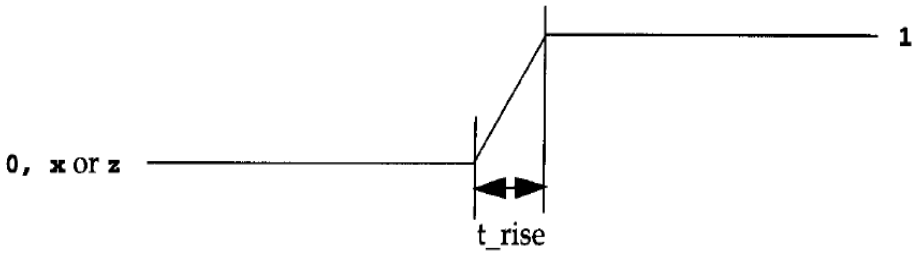
Gate Delays

In Verilog we introduce gate delays in the logic circuits, there are three different delays.

- Rise delay
- Fall delay
- Turn-offdelay.

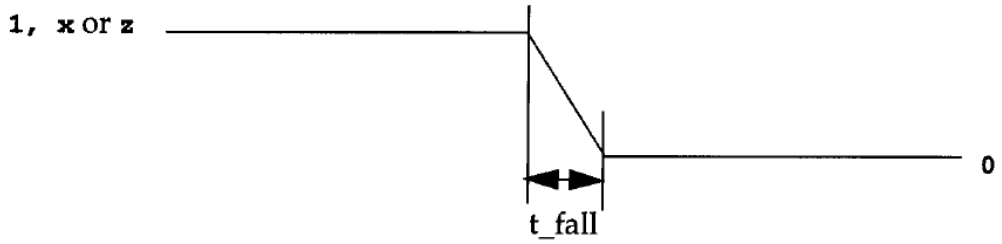
Rise Delay:

In rise delay the gate output transmits from 0, x or z to logic 1.



Fall Delay:

In rise delay the gate output transmits from 1, x or z to logic 0.



Turn-off delay:

When there is a transition in gate output to high impedance value from any other logic value. If the gate output transmits to don't care then minimum of three delays is considered.

Syntax:

For all transistion

gate_name #(delay_time) a1(output, inputs)

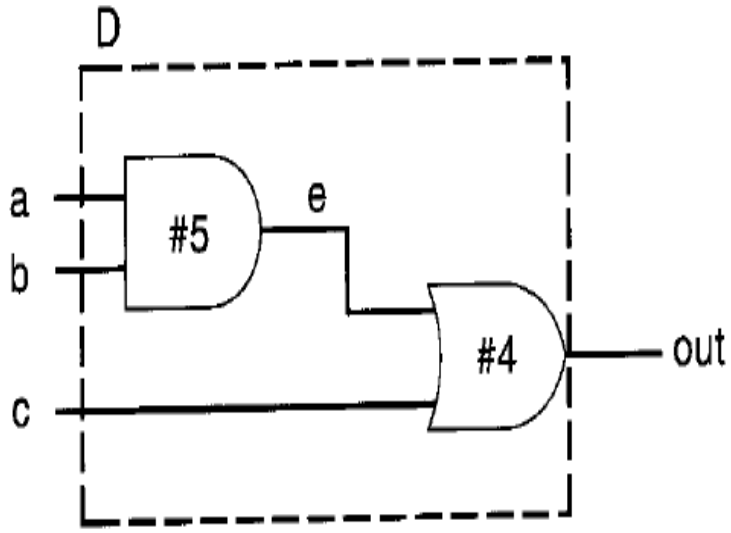
For rise and fall delay specification

gate_name #(rise_val,fall_val) a1(output, inputs)

For rise fall and turnoff delay specification

gate_name #(rise_val,fall_vall,turnoff_value) a1(output, inputs)

Example:



```
Module D(a,b,c,out);
        input a,b,c;
        output out;
        wire e;
        and  #(5) a1(e,a,b);
        or #(4) o1(out,e,c);
endmodule
```