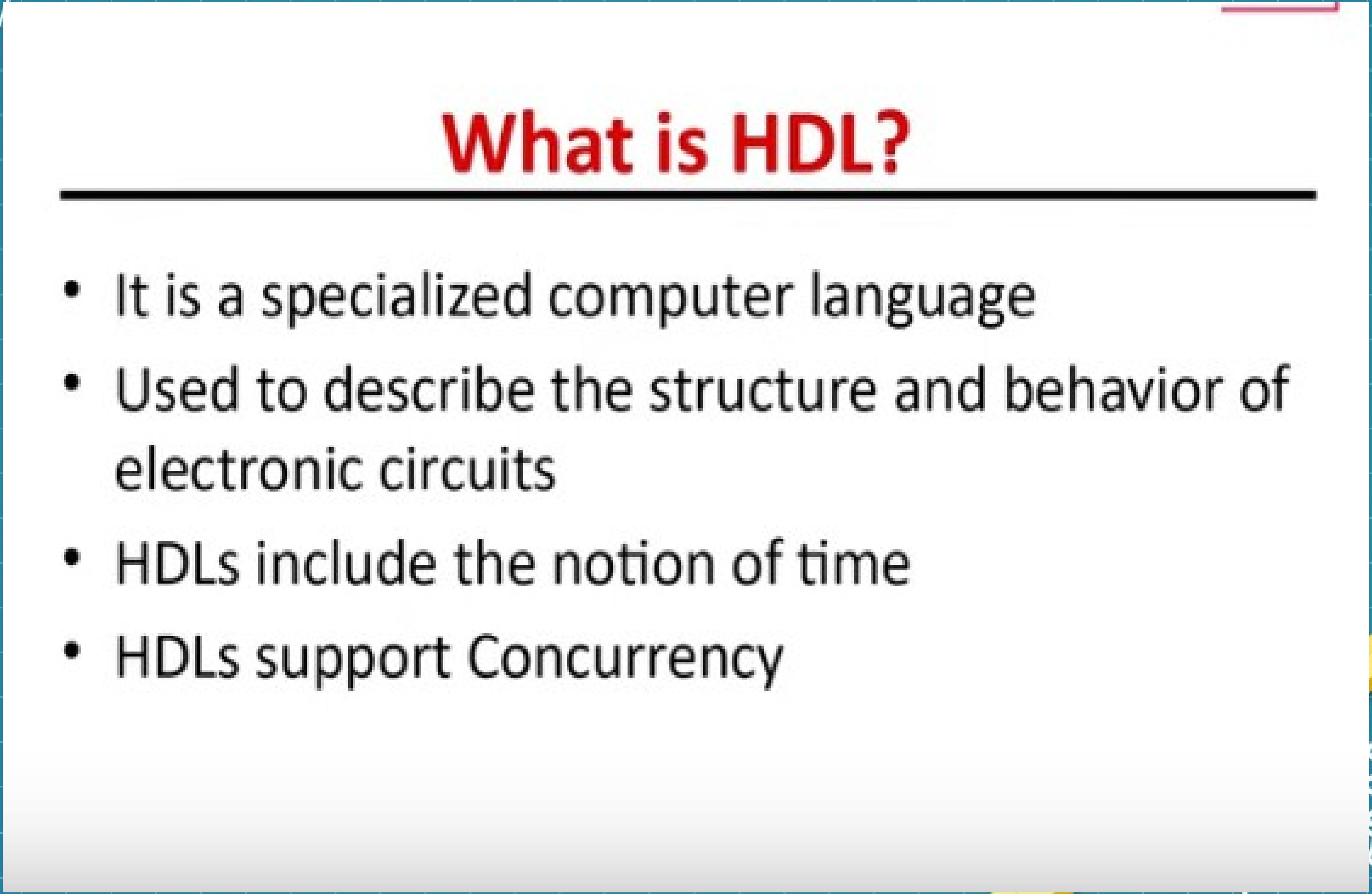
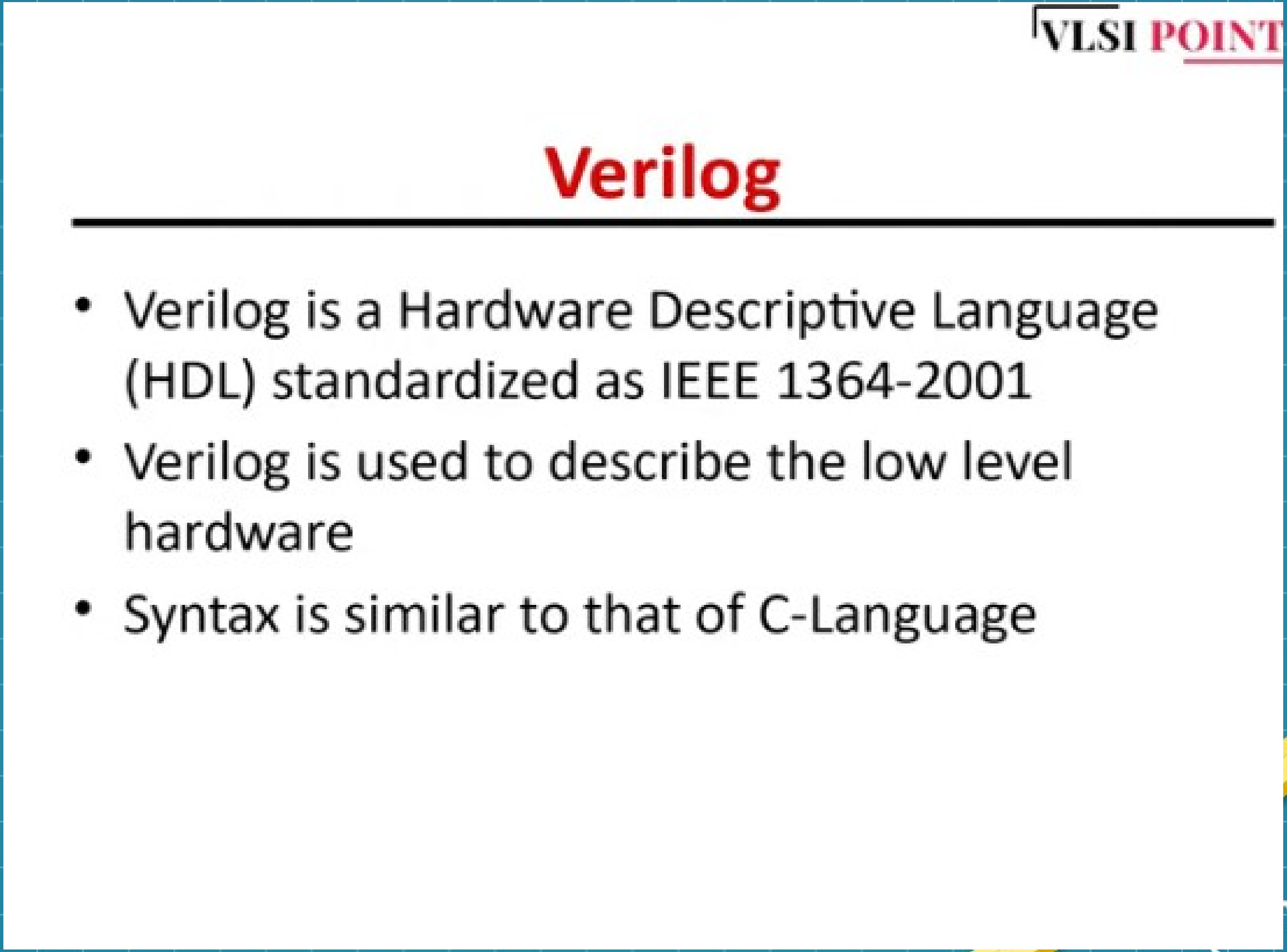
Introduction to Hardware Description Language (HDL)



What is HDL?

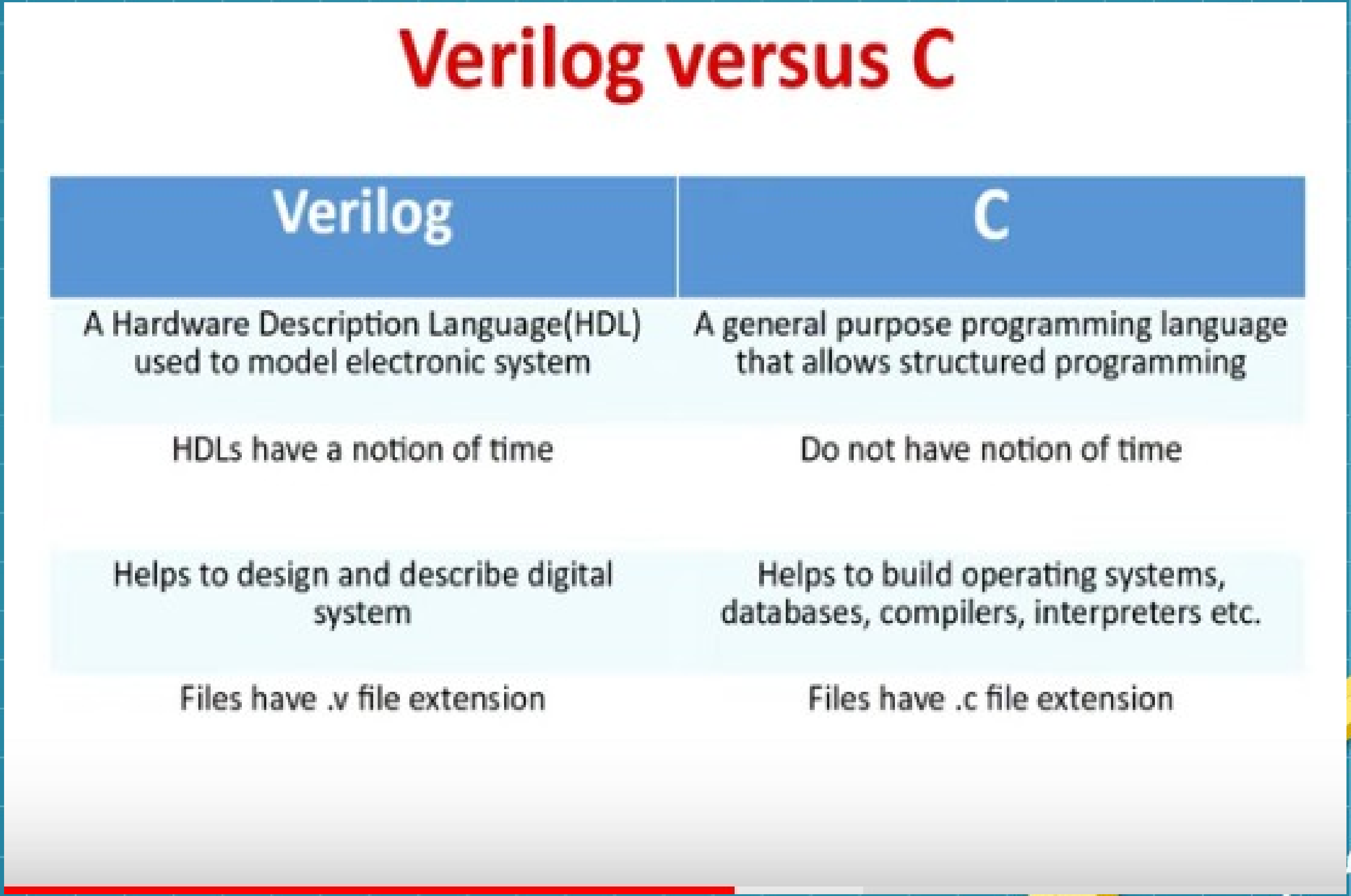
- It is a specialized computer language
- Used to describe the structure and behavior of electronic circuits
- HDLs include the notion of time
- HDLs support Concurrency



VLSI POINT

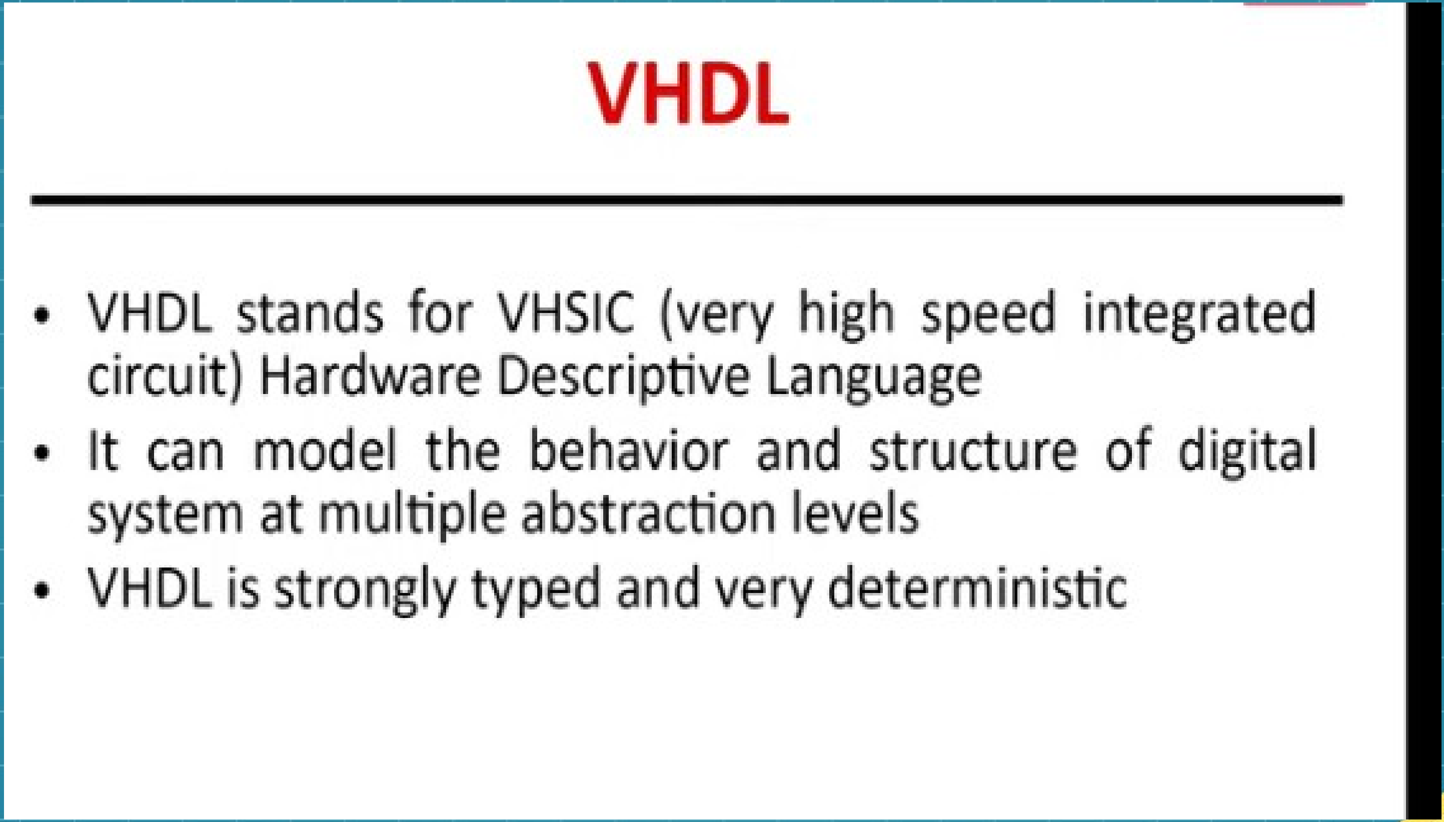
Verilog

- Verilog is a Hardware Descriptive Language (HDL) standardized as IEEE 1364-2001
- Verilog is used to describe the low level hardware
- Syntax is similar to that of C-Language



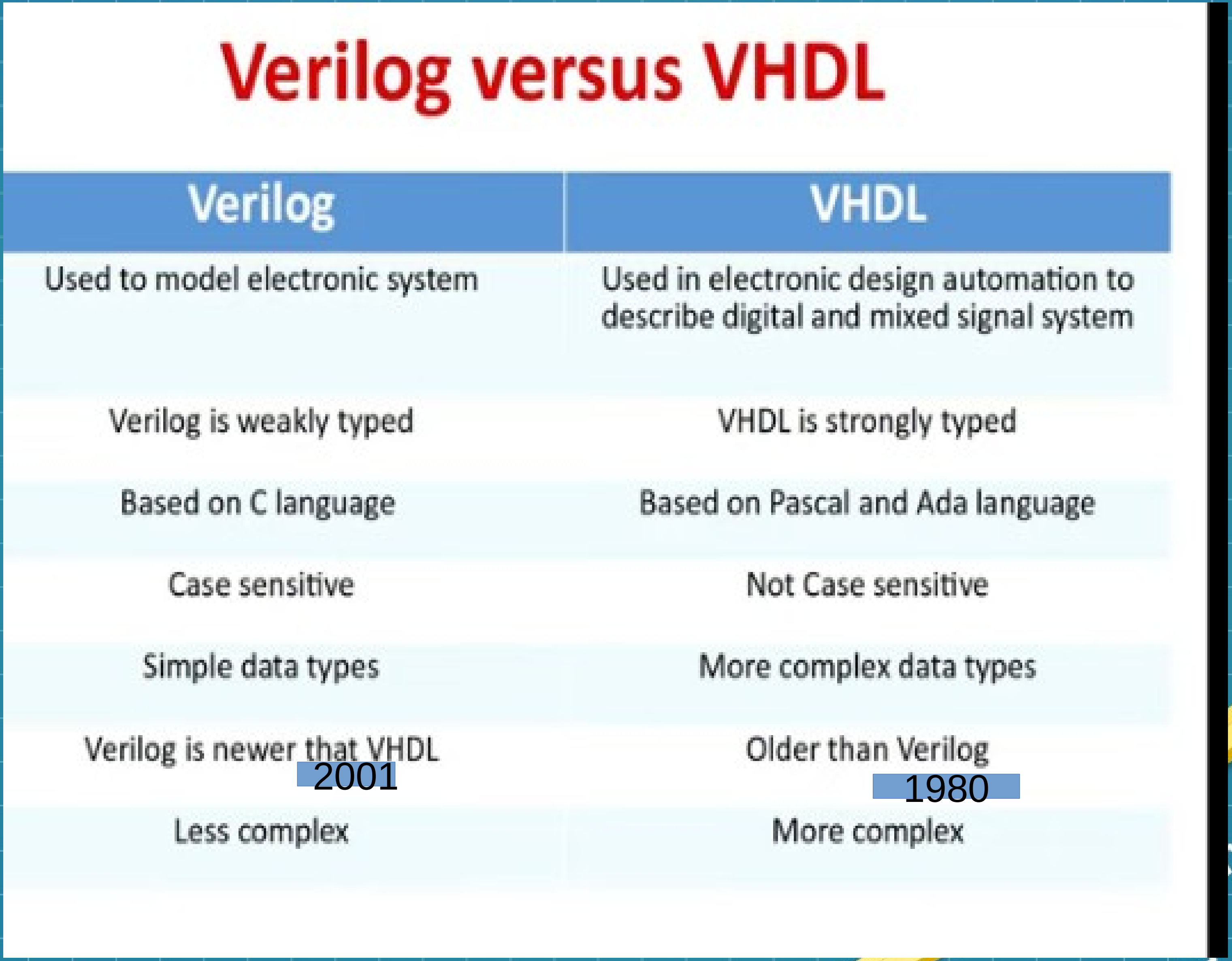
Verilog versus C

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



VHDL

- VHDL stands for VHSIC (very high speed integrated circuit) Hardware Descriptive Language
- It can model the behavior and structure of digital system at multiple abstraction levels
- VHDL is strongly typed and very deterministic



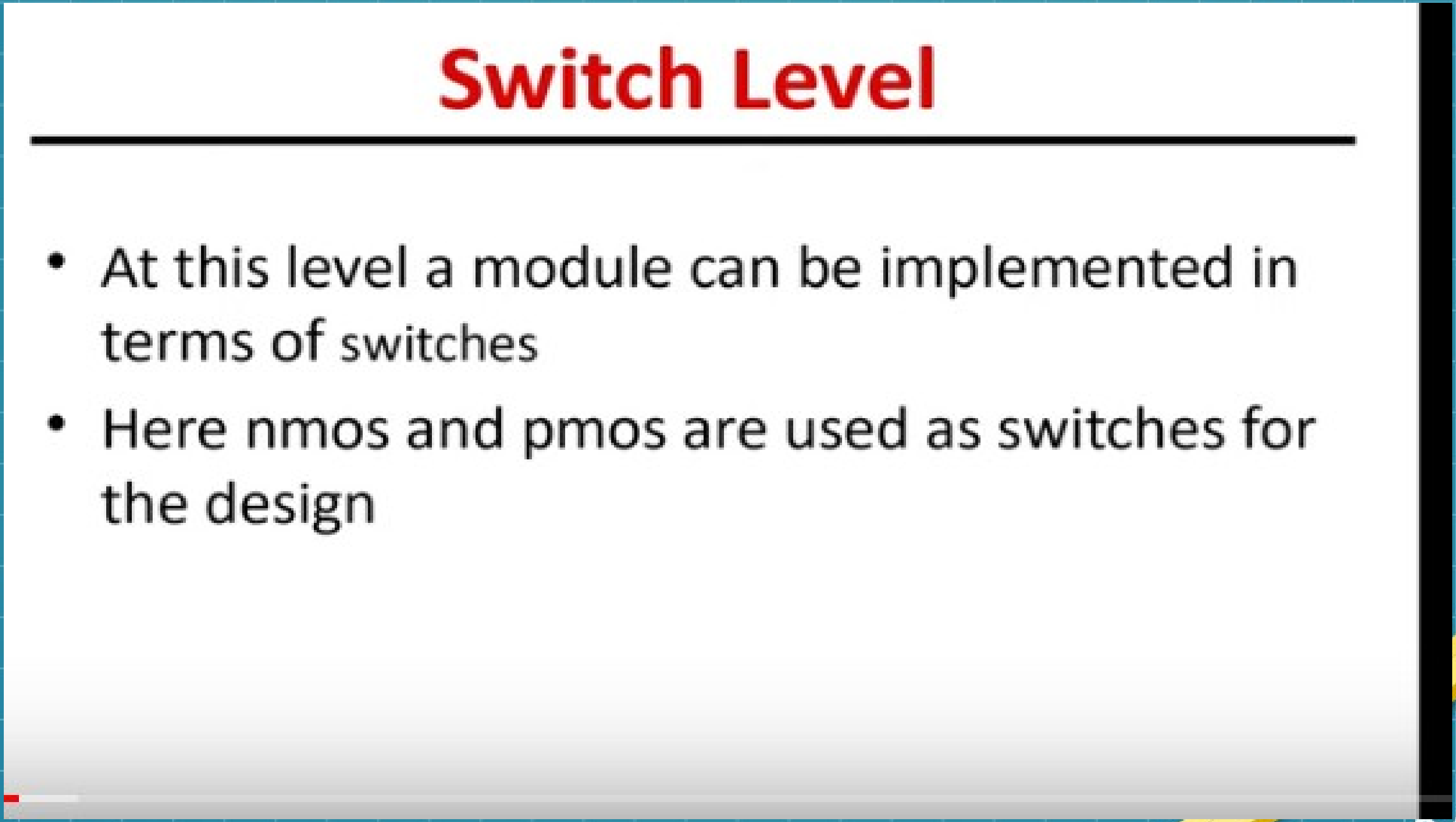
Verilog versus VHDL

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 that VHDL 2001	Older than Verilog 1980
Less complex	More complex

Verilog
Level of abstraction

Objective

- Switch Level — Implementation is done by Switches
- Gate Level — Logi gates interconnection
- Data Flow Level — How the data flowing in the ckt, design in term of Eq.
- Behavioral Level — Behavior of the citcuit



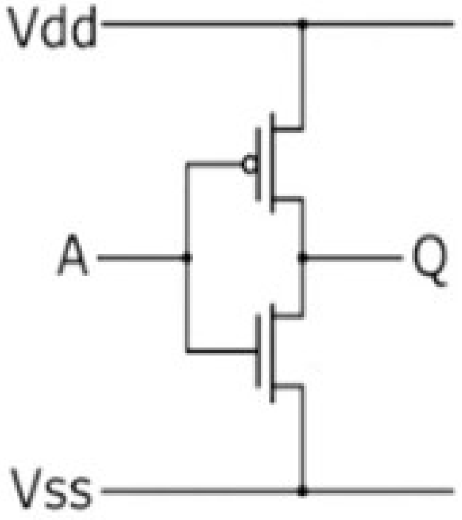
Switch Level

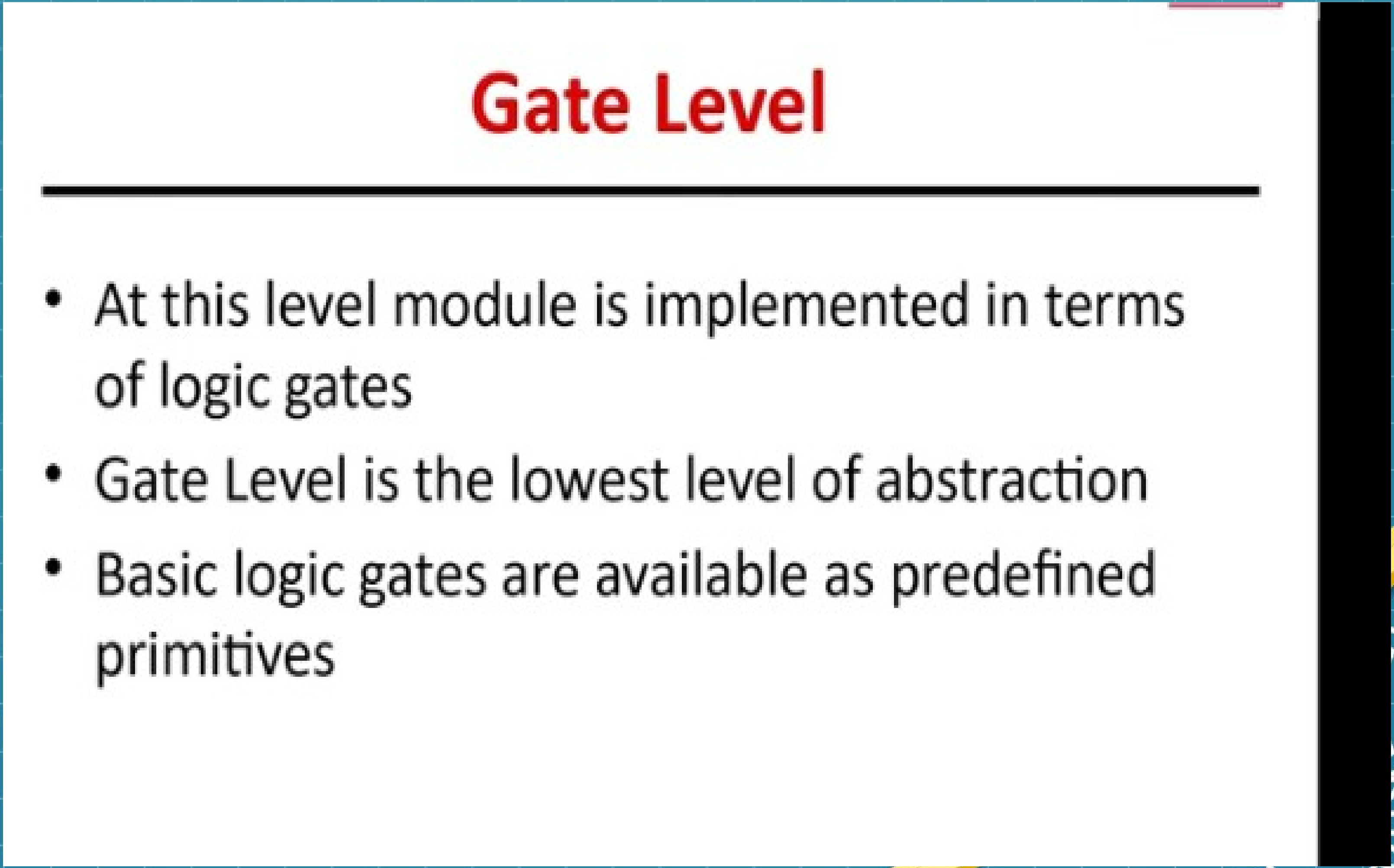
- At this level a module can be implemented in terms of switches
- Here nmos and pmos are used as switches for the design

Syntax: mos_name instance_name (output, data , control)

Example:

```
module inverter(Q,A);
     input A;
     output Q;
     supply1 vdd;
     supplt0 vss;

     pmos  p (Q,vdd,A);
     nmos  n (Q,vss,A);

endmodule
```





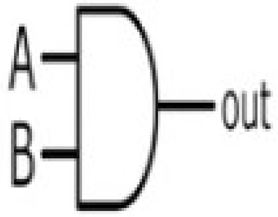
Gate Level

- At this level module is implemented in terms of logic gates
- Gate Level is the lowest level of abstraction
- Basic logic gates are available as predefined primitives

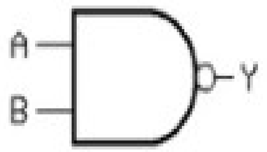
Syntax: Primitive_name instance_name(output , inputs)

Example:

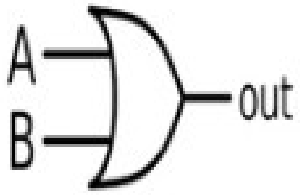
1. and G1(out,A,B);

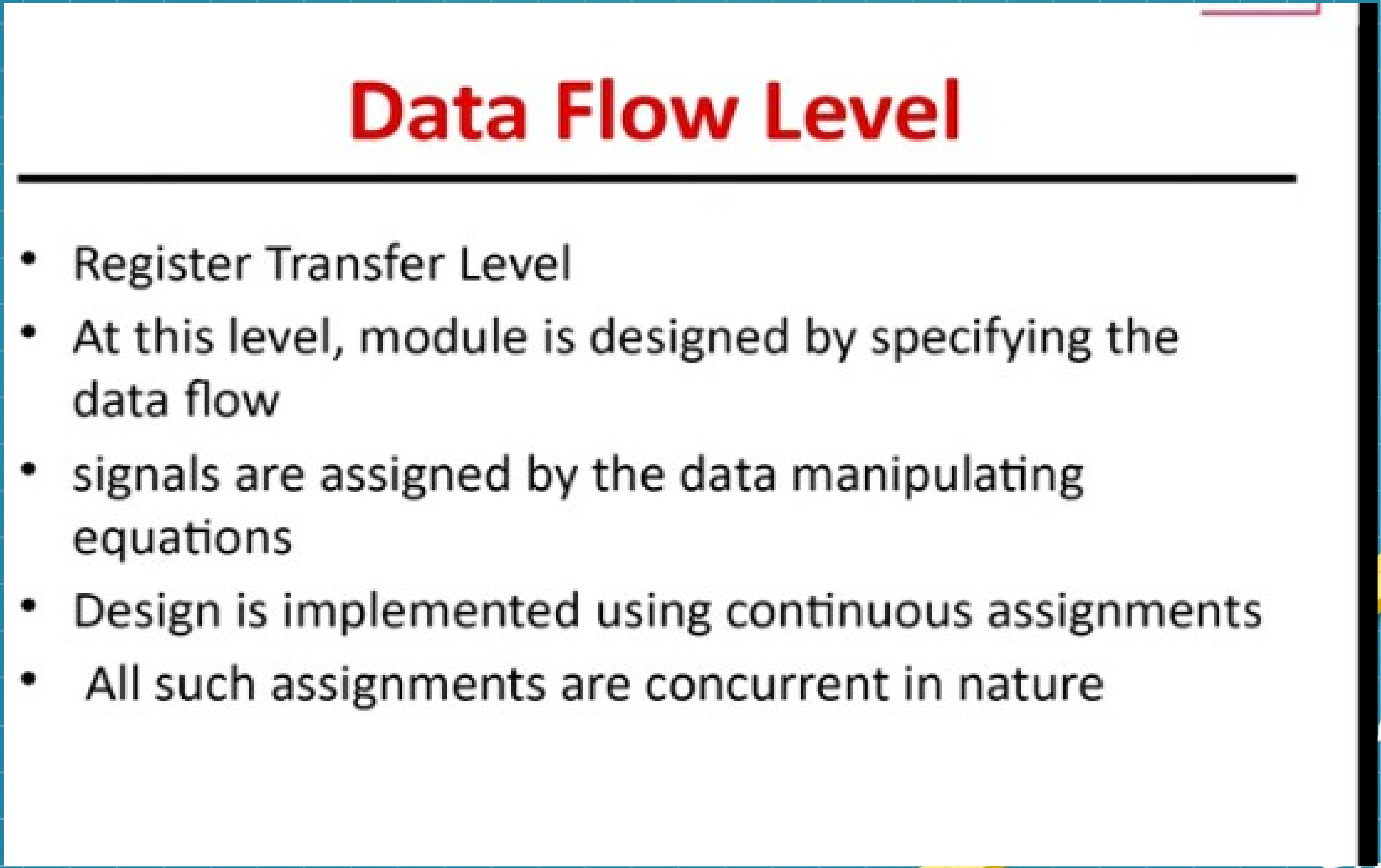


2. nand G2(Y,A,B);



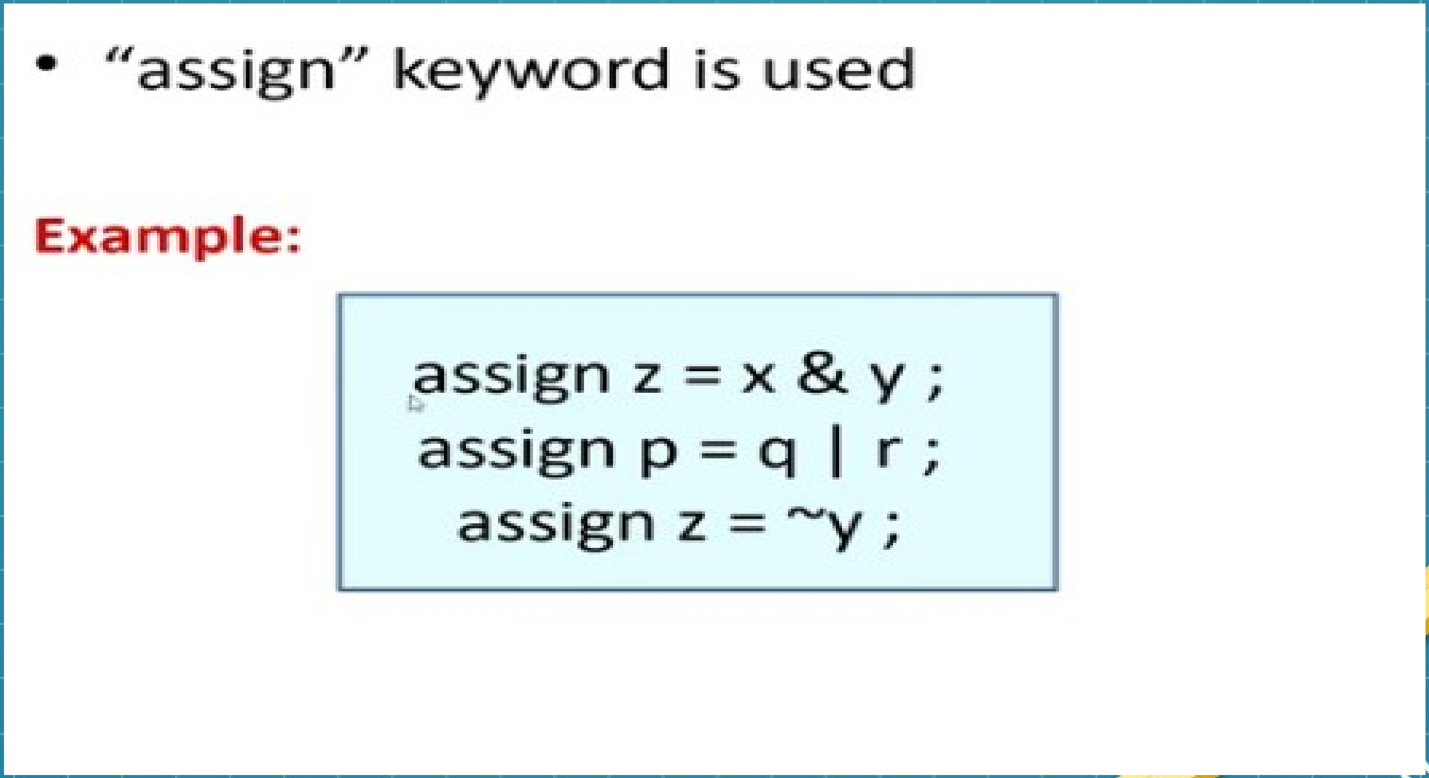
3. or G3(out,A,B);





Data Flow Level

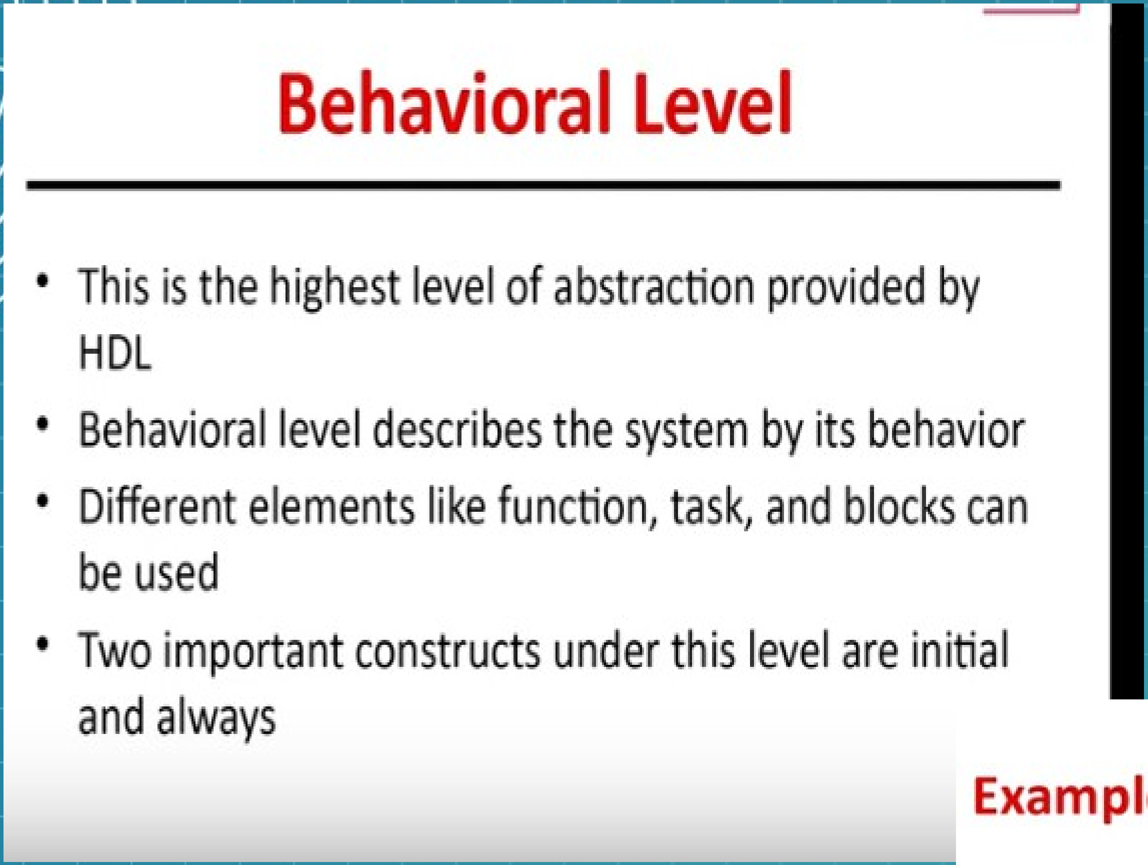
- Register Transfer Level
- At this level, module is designed by specifying the data flow
- signals are assigned by the data manipulating equations
- Design is implemented using continuous assignments
- All such assignments are concurrent in nature



- "assign" keyword is used

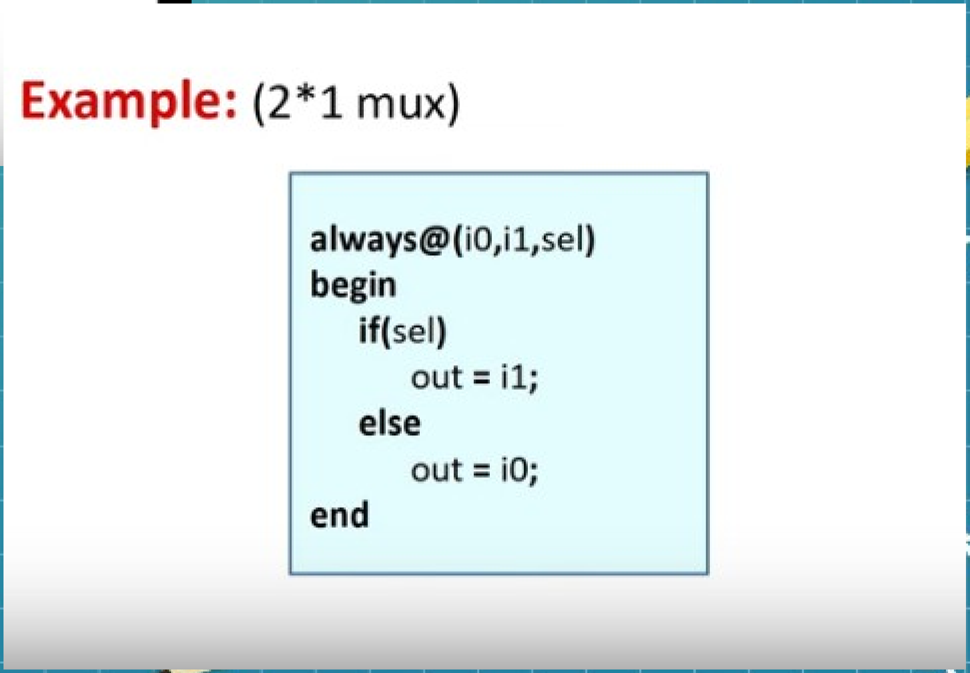
Example:

```
assign z = x & y ;
assign p = q | r ;
assign z = ~y ;
```



Behavioral Level

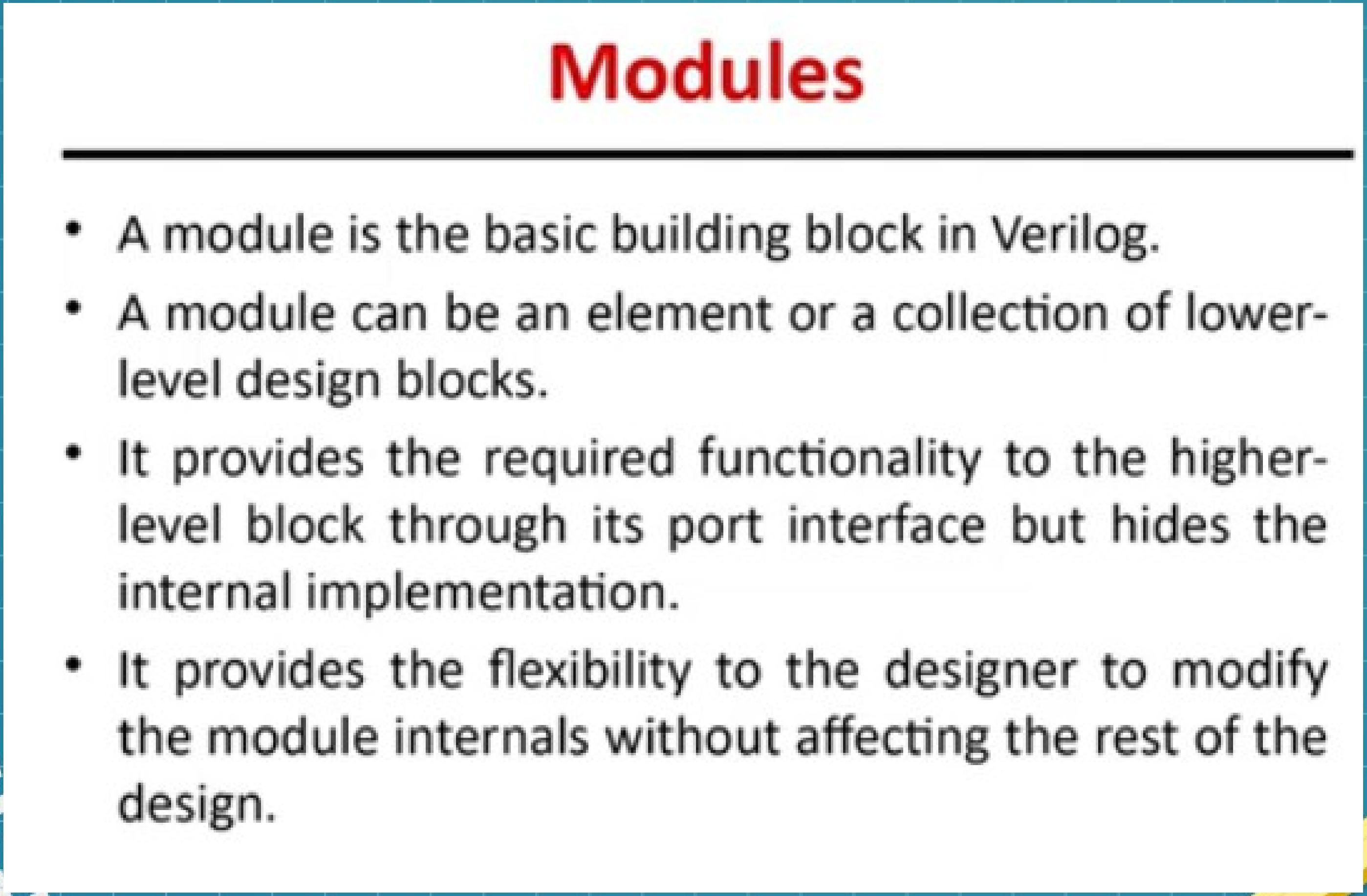
- This is the highest level of abstraction provided by HDL
- Behavioral level describes the system by its behavior
- Different elements like function, task, and blocks can be used
- Two important constructs under this level are initial and always



Example: (2*1 mux)

```
always@(i0,i1,sel)
begin
    if(sel)
        out = i1;
    else
        out = i0;
end
```

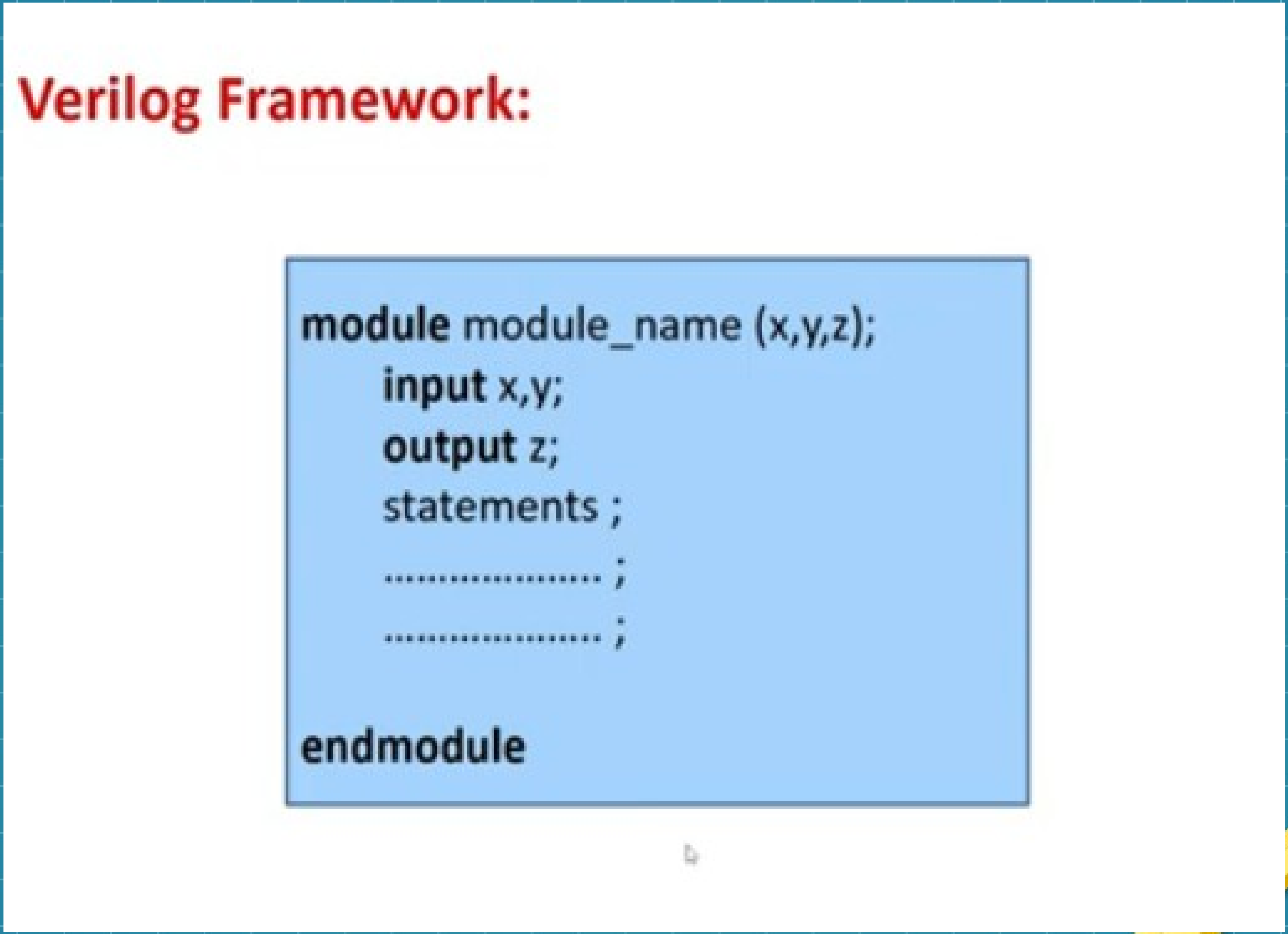
Modules and Instantiation



Modules

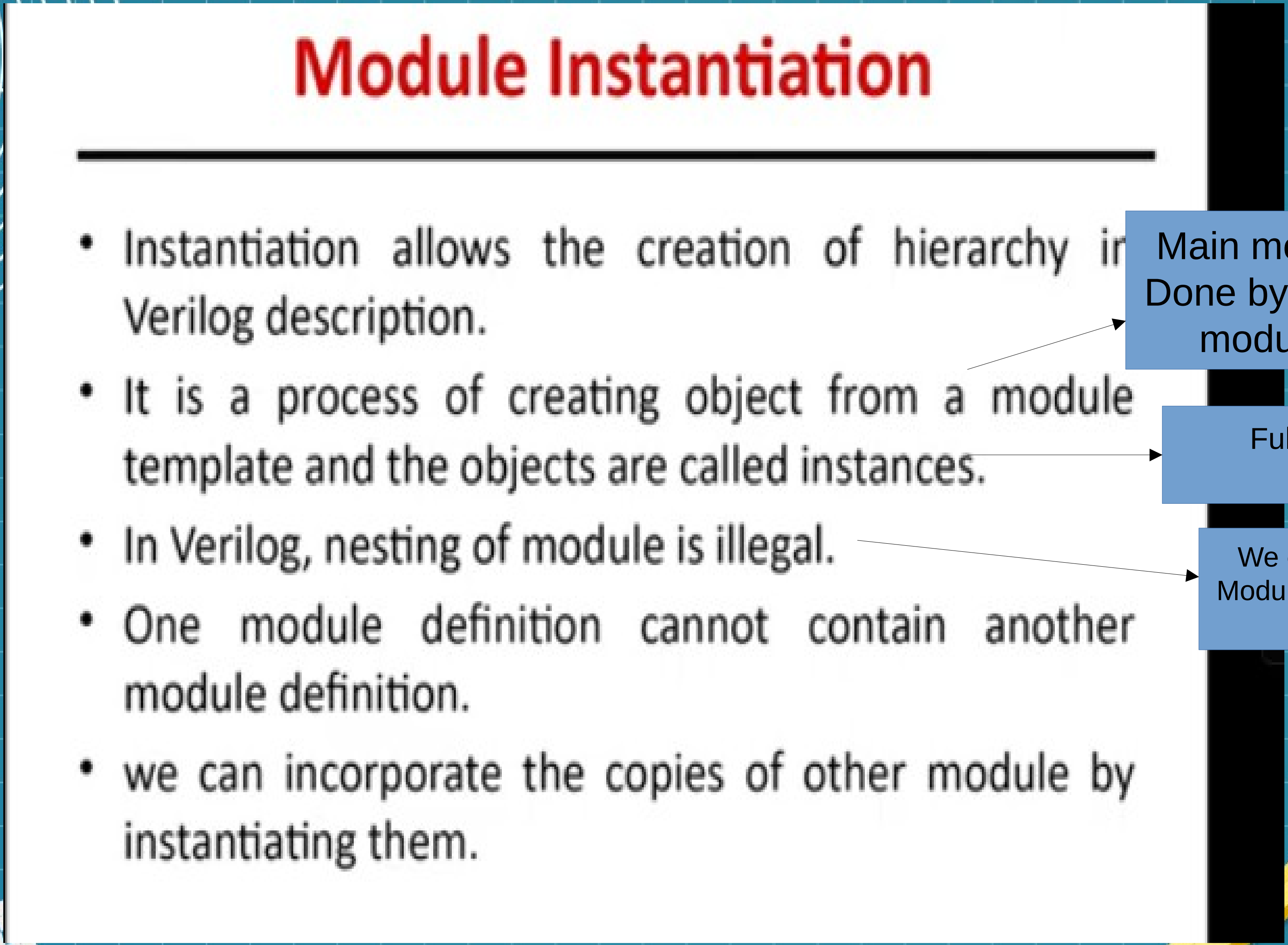
- A module is the basic building block in Verilog.
- A module can be an element or a collection of lower-level design blocks.
- It provides the required functionality to the higher-level block through its port interface but hides the internal implementation.
- It provides the flexibility to the designer to modify the module internals without affecting the rest of the design.

Provides the flexibilty to designers to modify the Module internals w/0 affecting Resy of the designs



Verilog Framework:

```
module module_name (x,y,z);
       input x,y;
       output z;
       statements ;
       ..................... ;
       ..................... ;

endmodule
```



Module Instantiation

- Instantiation allows the creation of hierarchy i Verilog description.
- It is a process of creating object from a module template and the objects are called instances.
- In Verilog, nesting of module is illegal.
- One module definition cannot contain another module definition.
- we can incorporate the copies of other module by instantiating them.

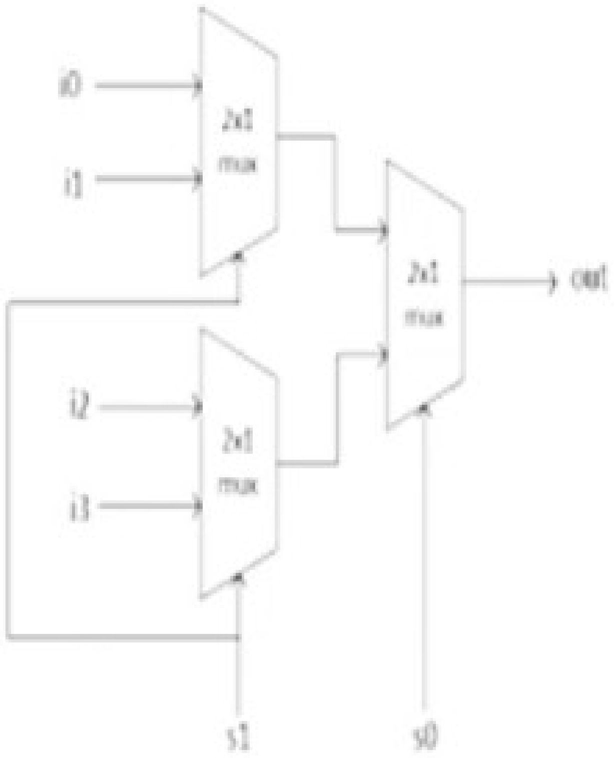
Main module Done by small modules

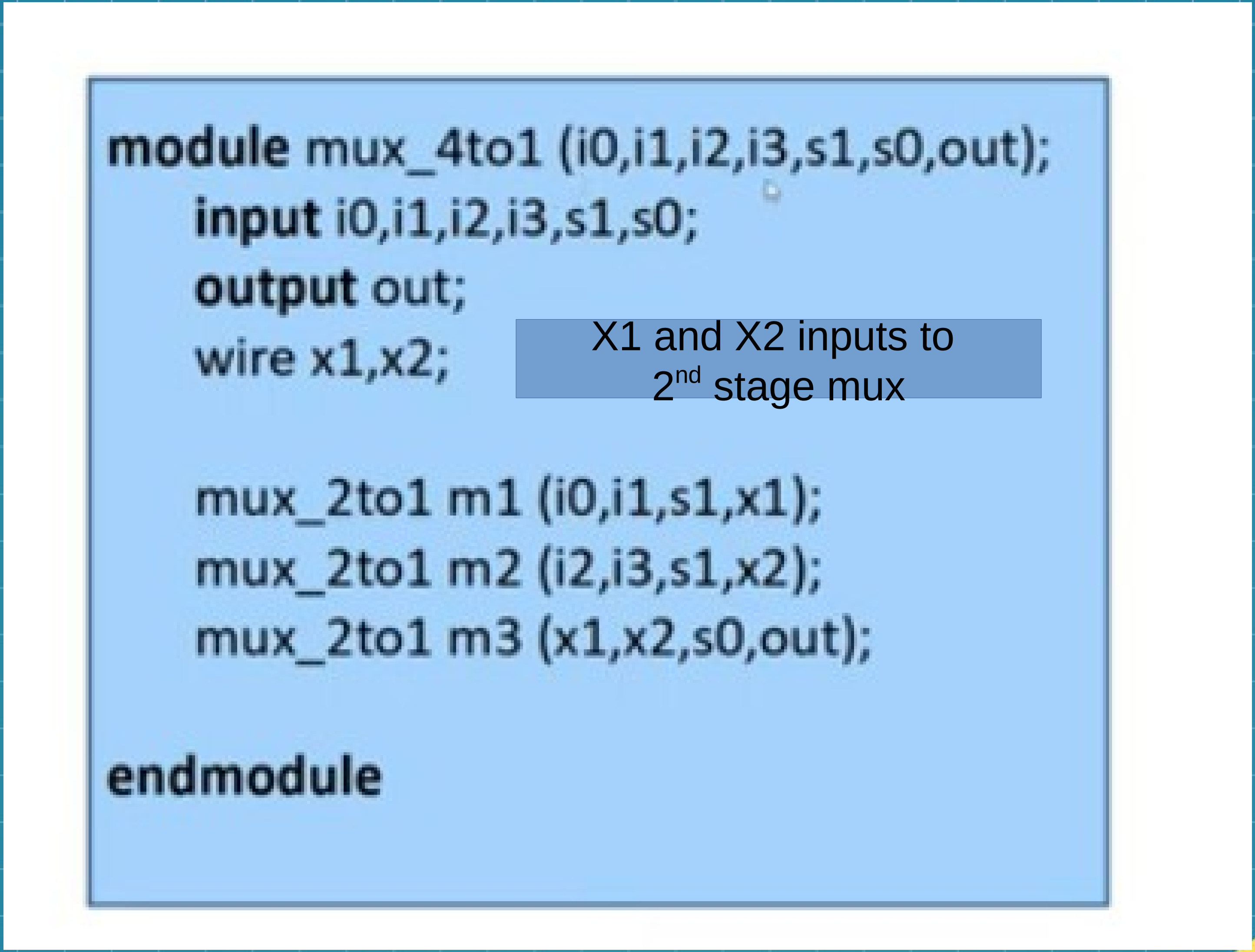
Full adder using Half adder

We can define one Module inside another module

Example: 4*1 mux using 2*1 mux

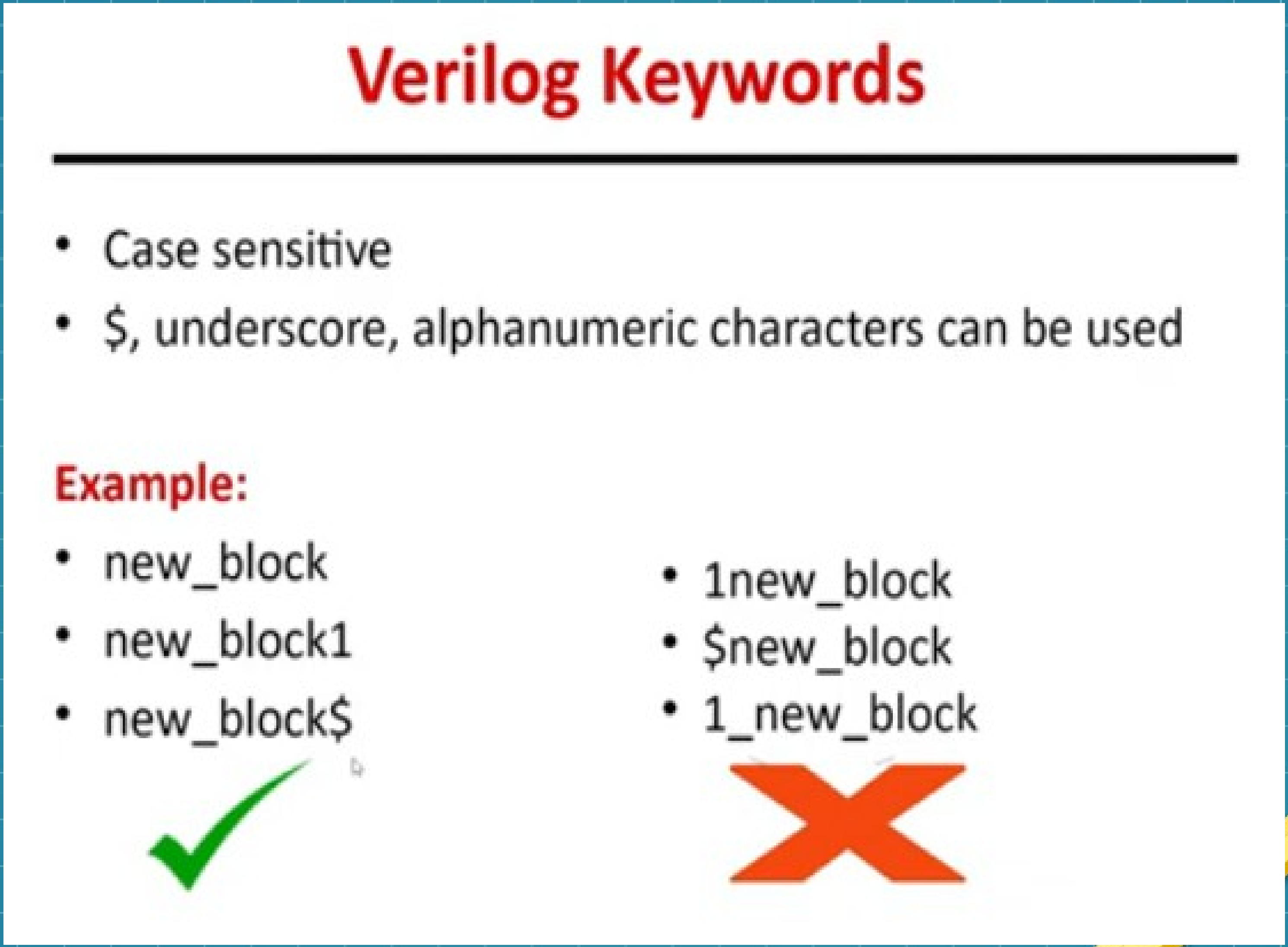
```
module mux_2to1 (i0,i1,sel,out);
    input i0,i1,sel;
    output out;
    always@(i0,i1,sel)
    begin
        if(sel)
          out = i1;
        else
          out = i0;
    end
endmodule
```





```
module mux_4to1 (i0,i1,i2,i3,s1,s0,out);
    input i0,i1,i2,i3,s1,s0;
    output out;
    wire x1,x2;

    mux_2to1 m1 (i0,i1,s1,x1);
    mux_2to1 m2 (i2,i3,s1,x2);
    mux_2to1 m3 (x1,x2,s0,out);

endmodule
```

X1 and X2 inputs to 2nd stage mux



Verilog Keywords

- Case sensitive
- $, underscore, alphanumeric characters can be used

Example:

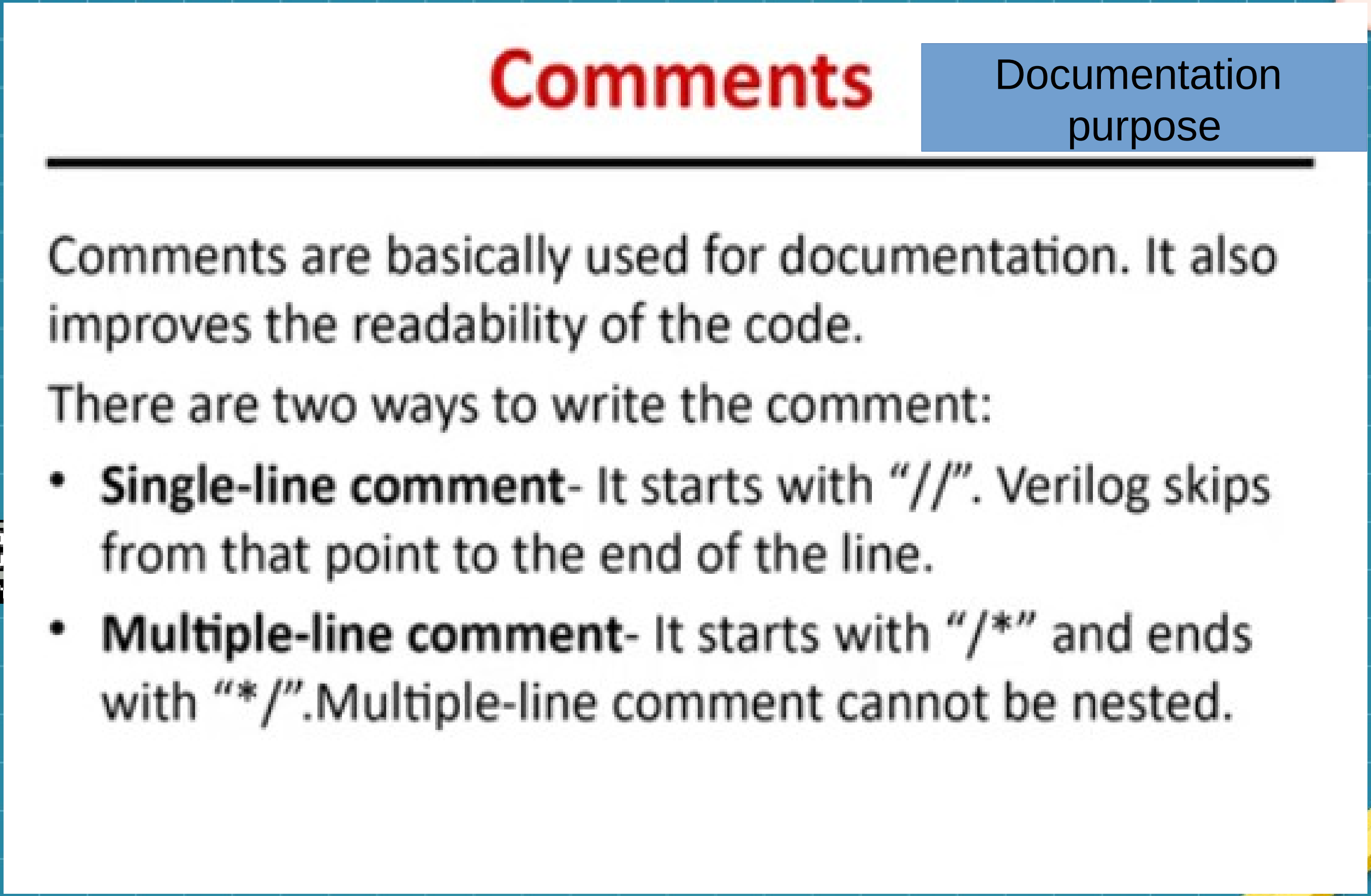
- new_block
- new_block1
- new_block$

✓

- 1new_block
- $new_block
- 1_new_block

✗

Documentation purpose

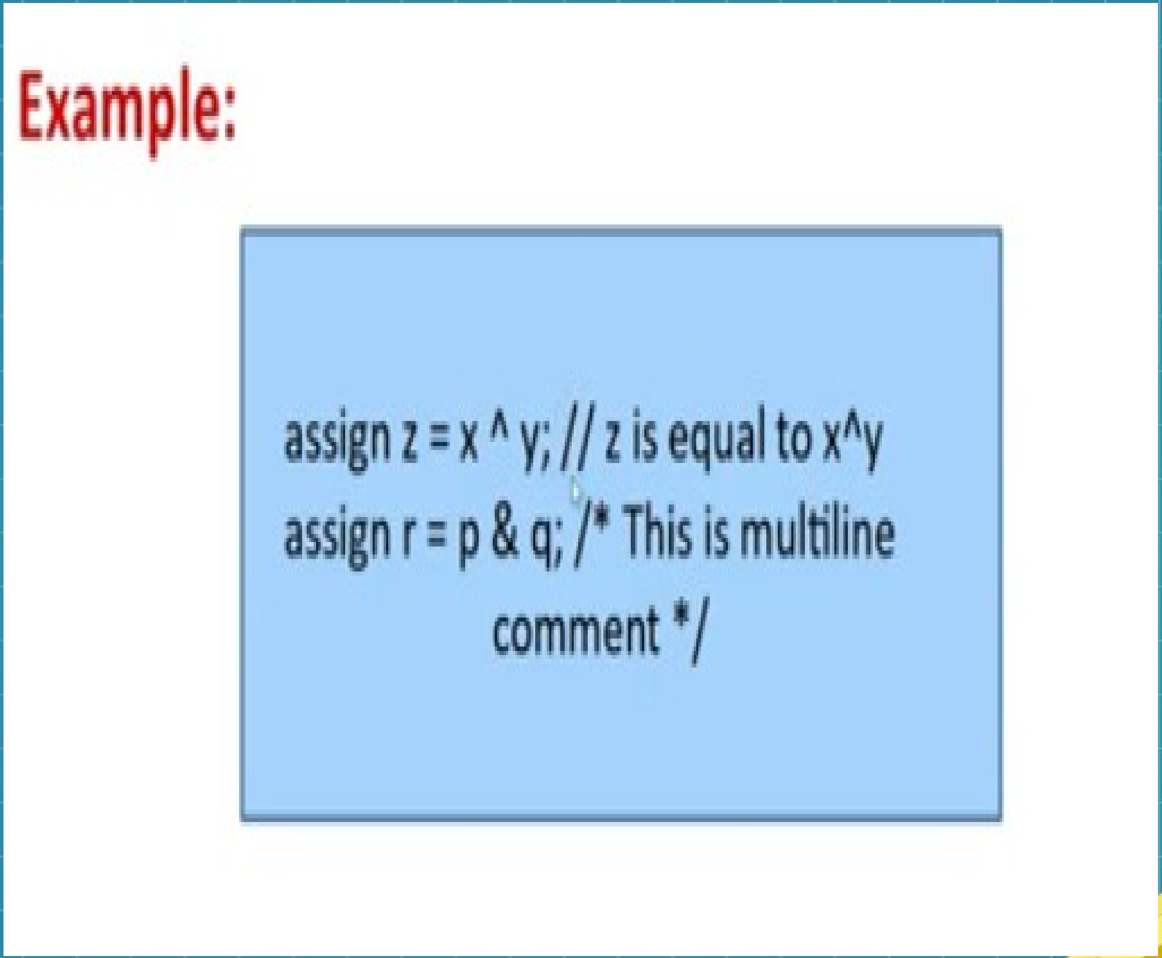


Comments

Comments are basically used for documentation. It also improves the readability of the code.

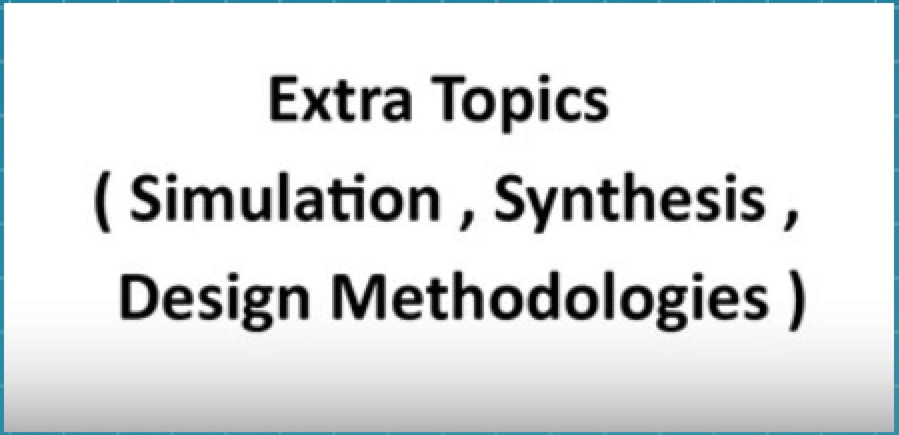
There are two ways to write the comment:

- **Single-line comment**- It starts with "//". Verilog skips from that point to the end of the line.
- **Multiple-line comment**- It starts with "/*" and ends with "*/".Multiple-line comment cannot be nested.

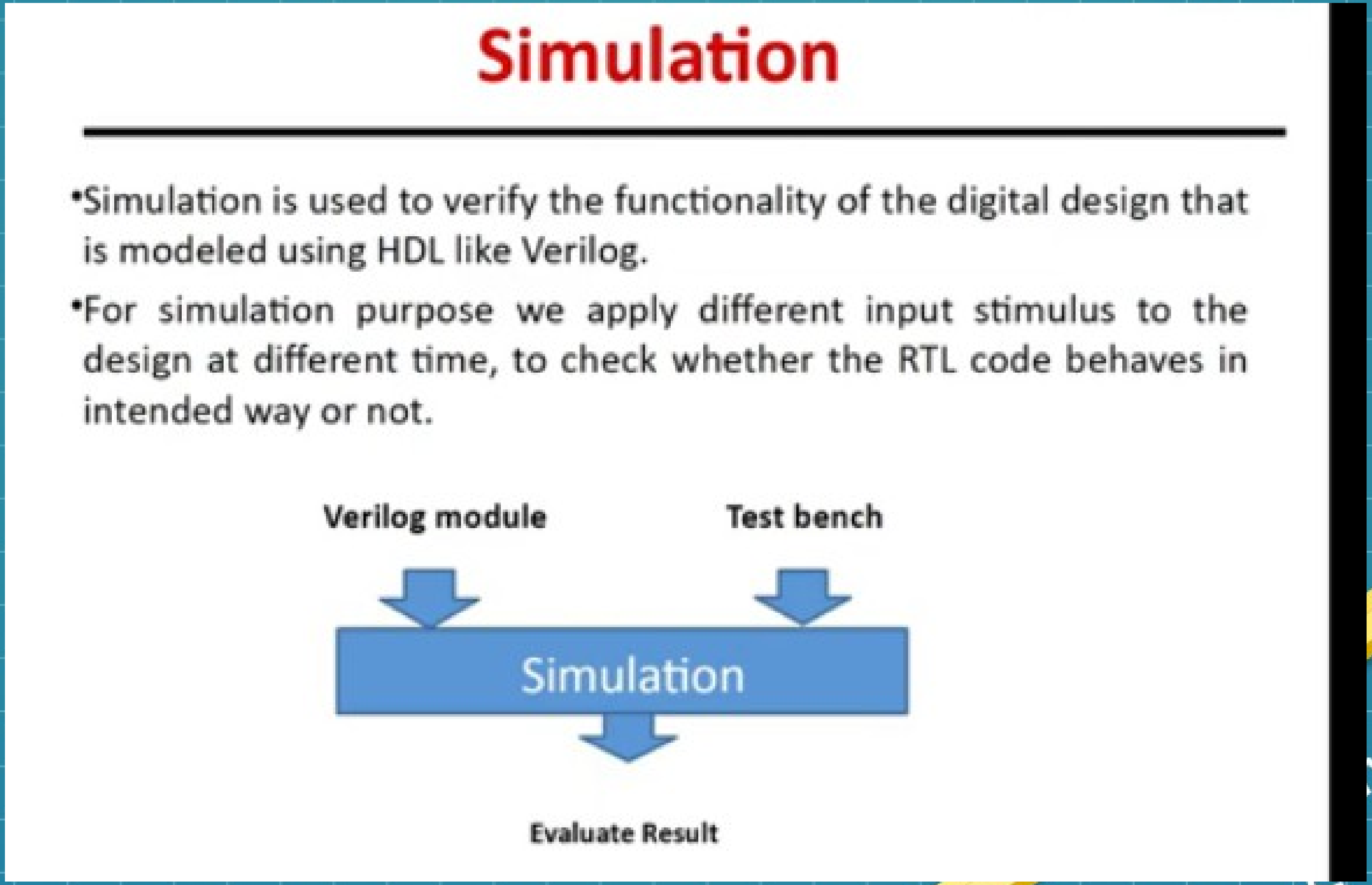


Example:

```
assign z = x ^ y; // z is equal to x^y
assign r = p & q; /* This is multiline
                 comment */
```

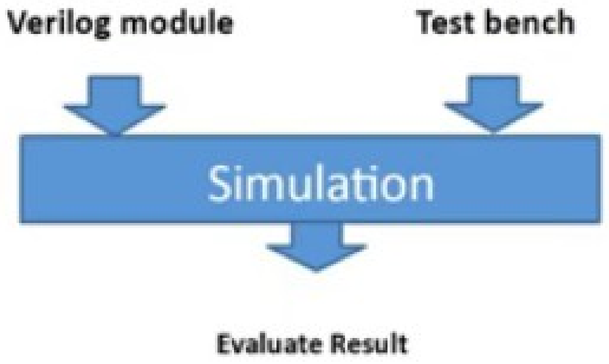


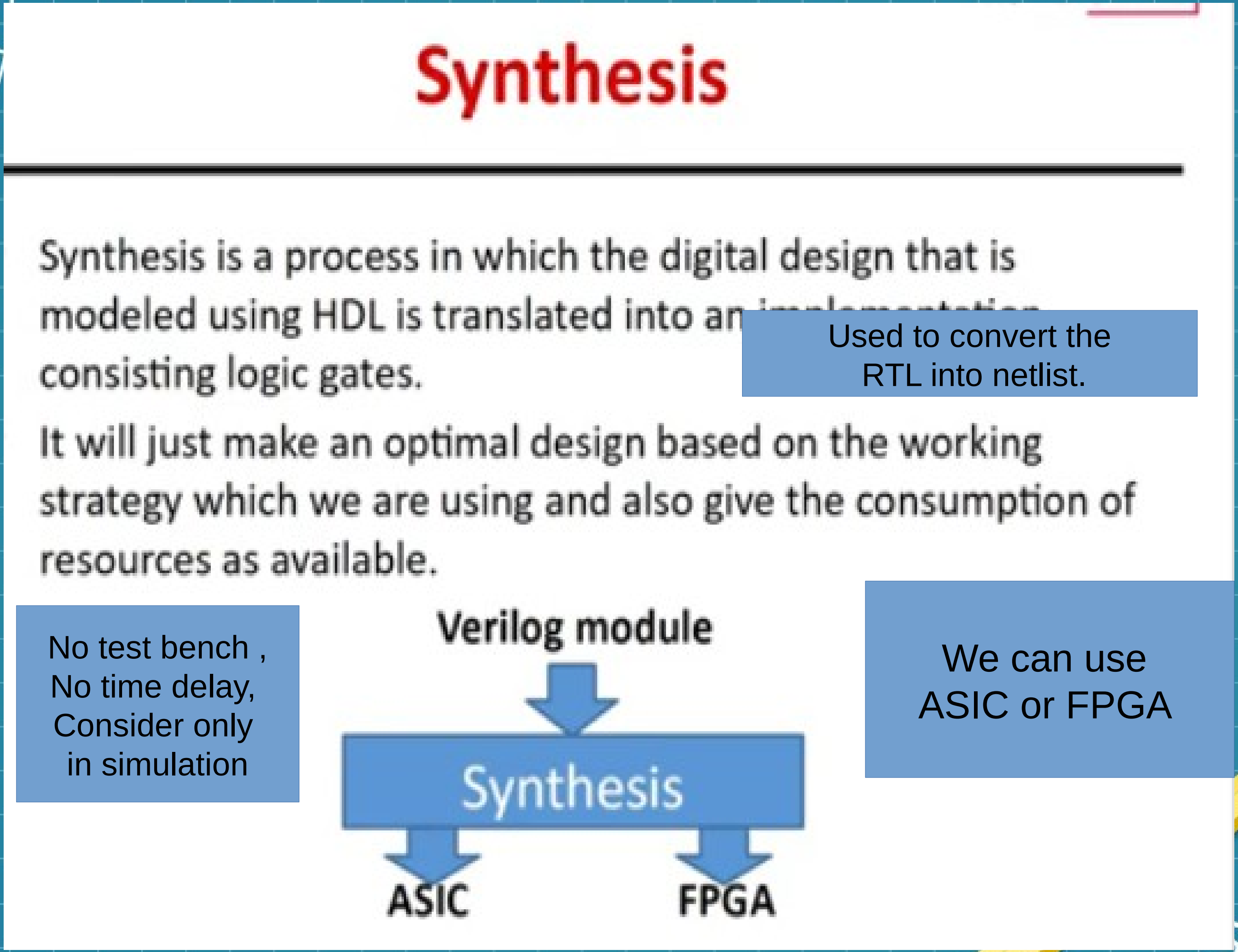
Extra Topics (Simulation , Synthesis , Design Methodologies)



Simulation

- Simulation is used to verify the functionality of the digital design that is modeled using HDL like Verilog.
- For simulation purpose we apply different input stimulus to the design at different time, to check whether the RTL code behaves in intended way or not.





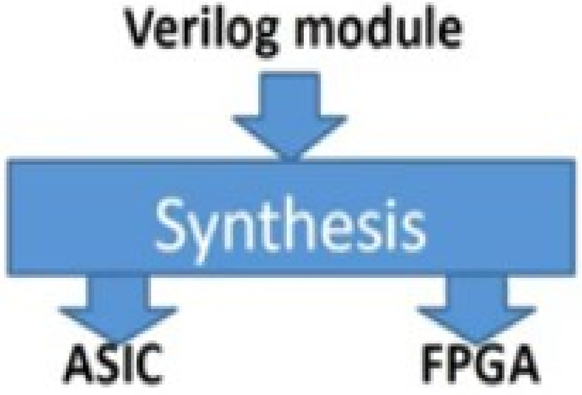
Synthesis

Synthesis is a process in which the digital design that is modeled using HDL is translated into an [illegible] consisting logic gates.

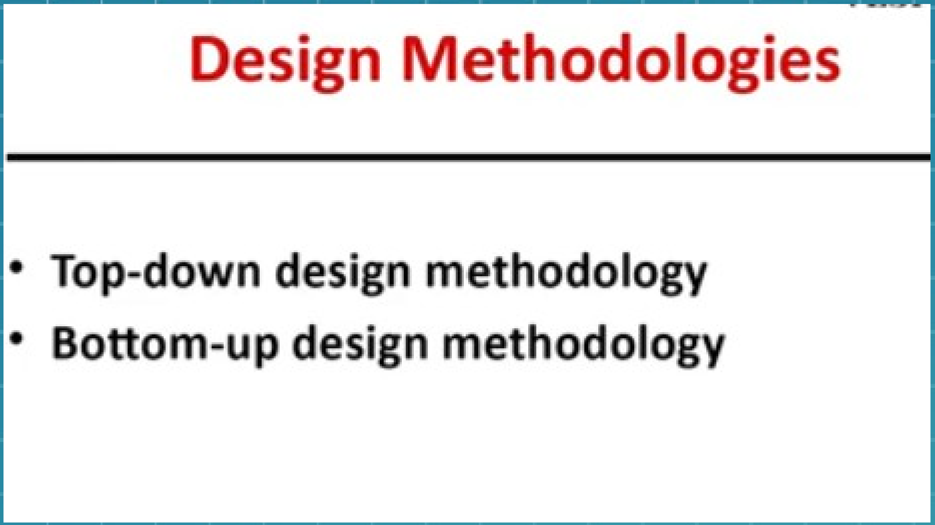
Used to convert the RTL into netlist.

It will just make an optimal design based on the working strategy which we are using and also give the consumption of resources as available.

No test bench , No time delay, Consider only in simulation

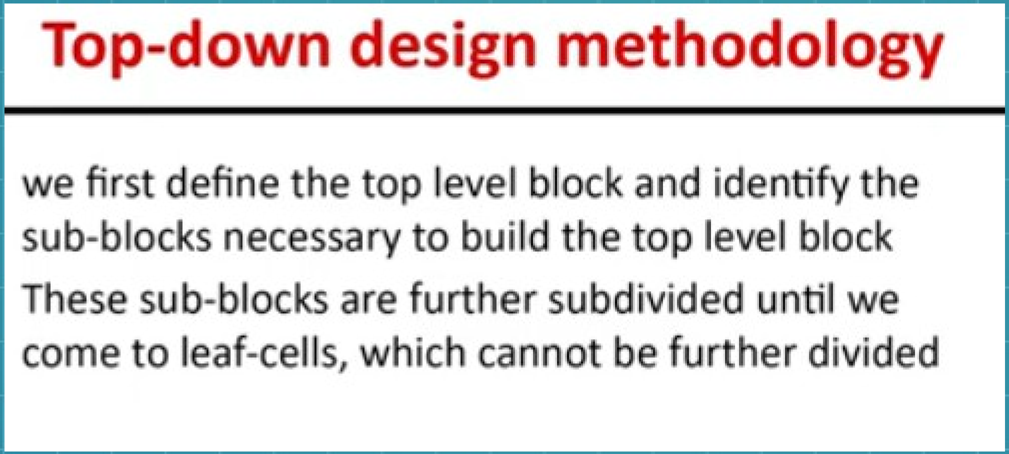


We can use ASIC or FPGA



Design Methodologies

- Top-down design methodology
- Bottom-up design methodology

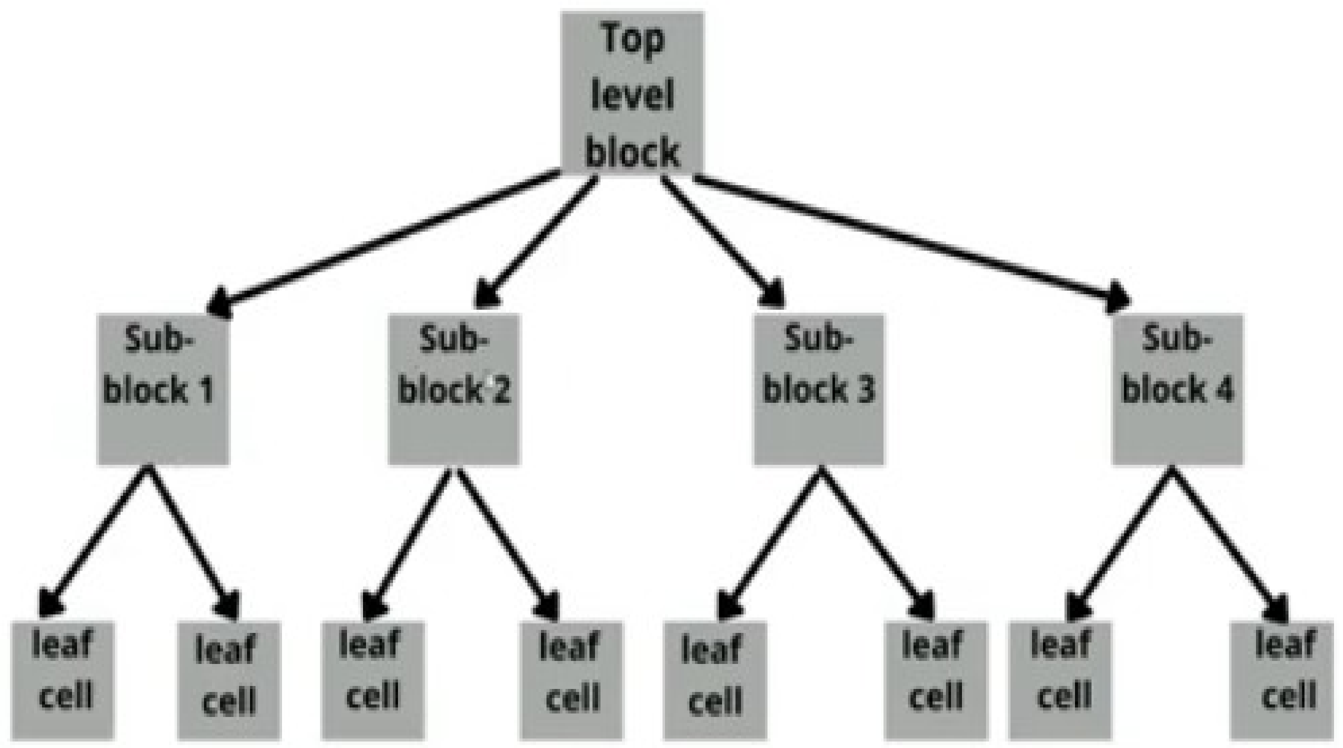


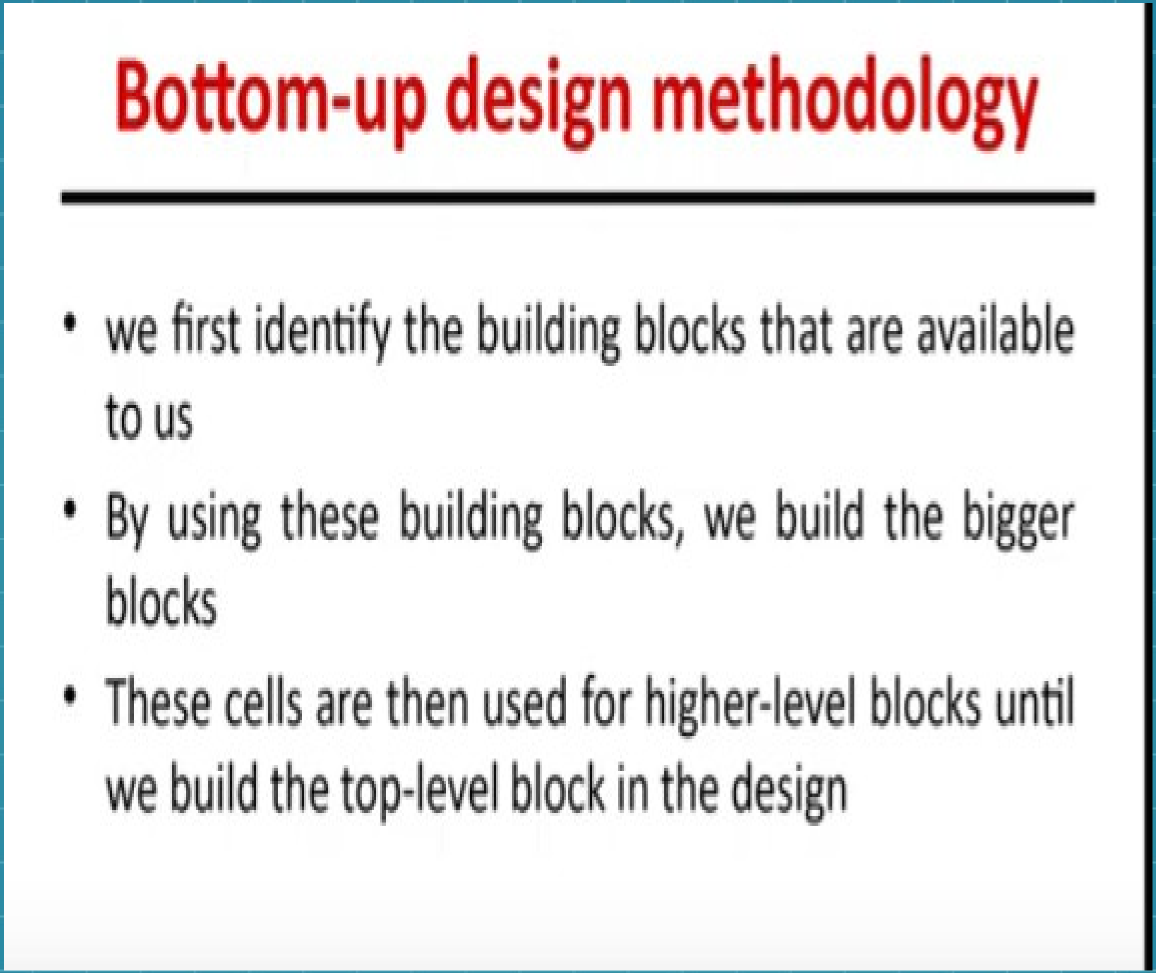
Top-down design methodology

we first define the top level block and identify the sub-blocks necessary to build the top level block

These sub-blocks are further subdivided until we come to leaf-cells, which cannot be further divided

Top-down design methodology

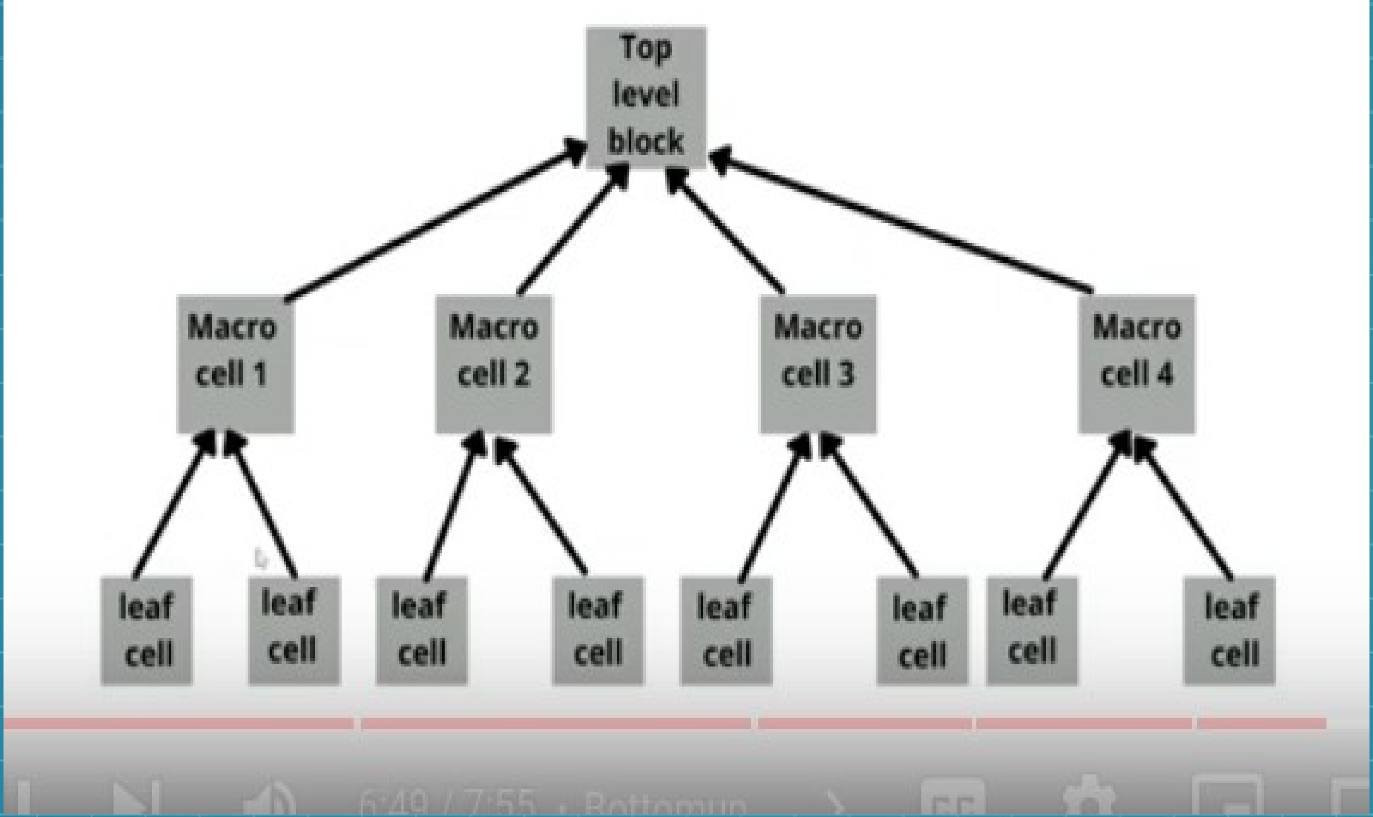


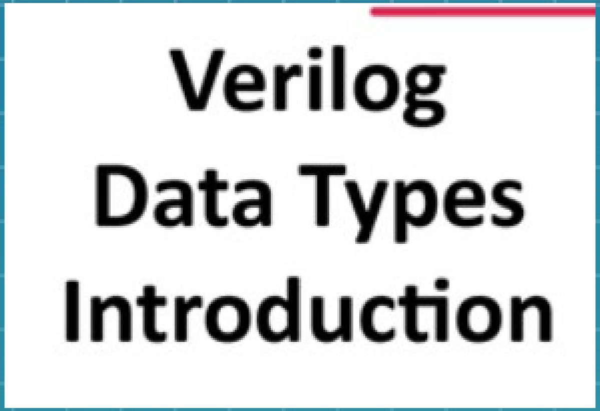


Bottom-up design methodology

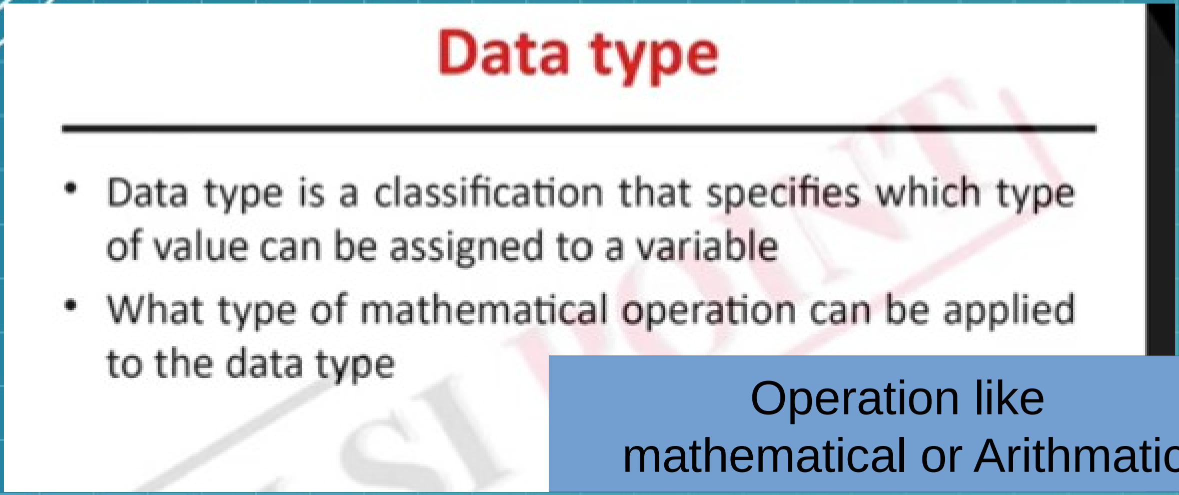
- we first identify the building blocks that are available to us
- By using these building blocks, we build the bigger blocks
- These cells are then used for higher-level blocks until we build the top-level block in the design

Bottom-up design methodology





Verilog Data Types Introduction

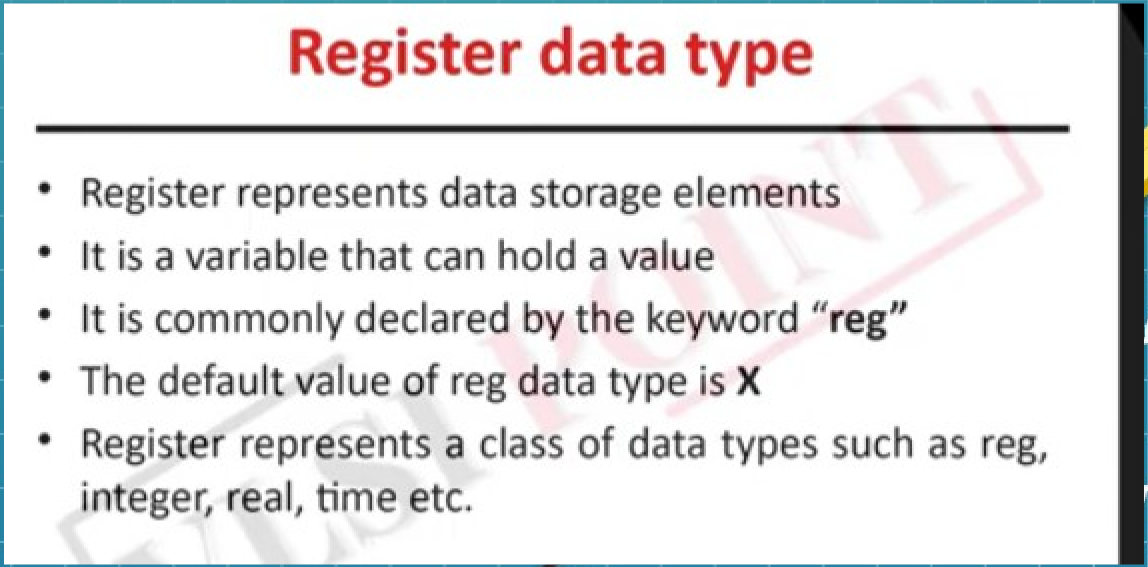


Data type

- Data type is a classification that specifies which type of value can be assigned to a variable
- What type of mathematical operation can be applied to the data type

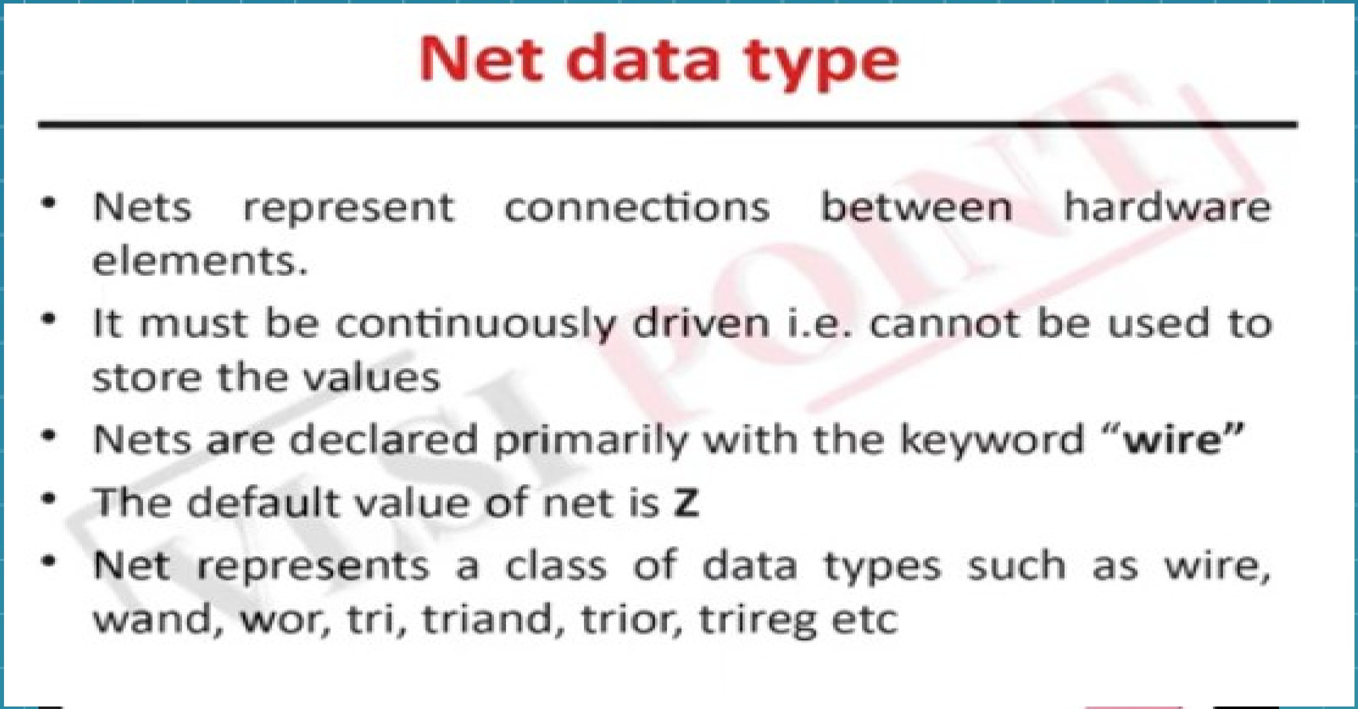
Operation like mathematical or Arithmatic

Two Data types Register and Net



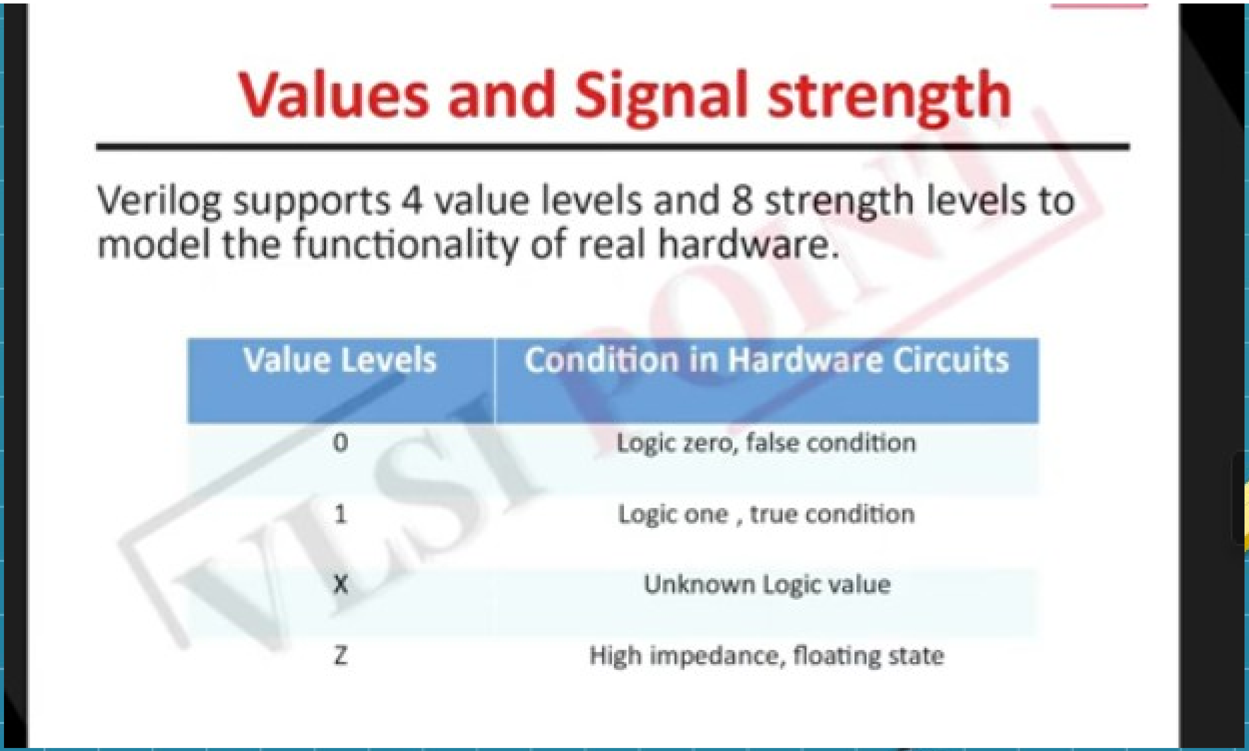
Register data type

- Register represents data storage elements
- It is a variable that can hold a value
- It is commonly declared by the keyword "**reg**"
- The default value of reg data type is **X**
- Register represents a class of data types such as reg, integer, real, time etc.



Net data type

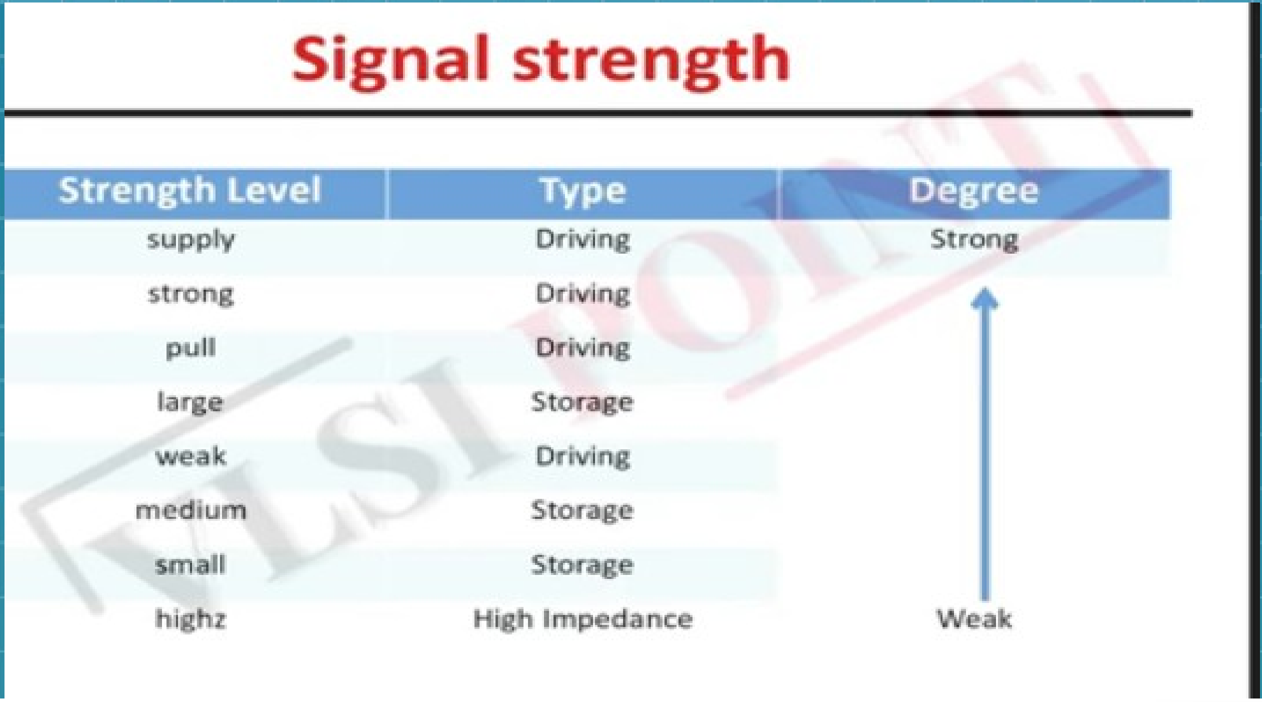
- Nets represent connections between hardware elements.
- It must be continuously driven i.e. cannot be used to store the values
- Nets are declared primarily with the keyword “**wire**”
- The default value of net is **Z**
- Net represents a class of data types such as wire, wand, wor, tri, triand, trior, trireg etc



Values and Signal strength

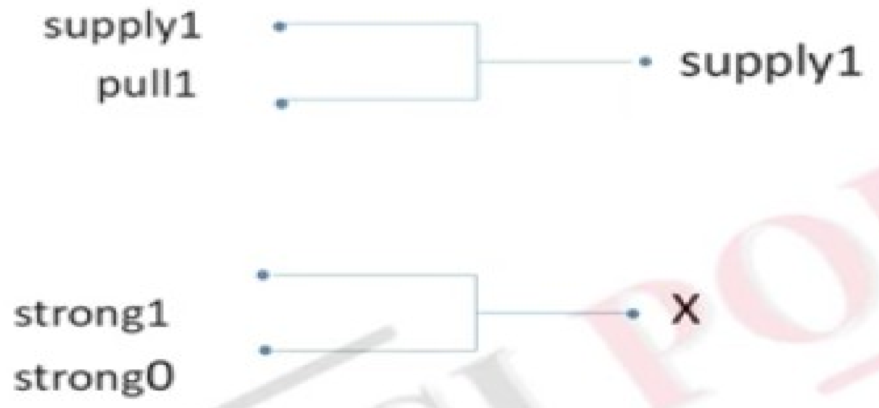
Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.

Value Levels	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one , true condition
X	Unknown Logic value
Z	High impedance, floating state



Signal strength

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

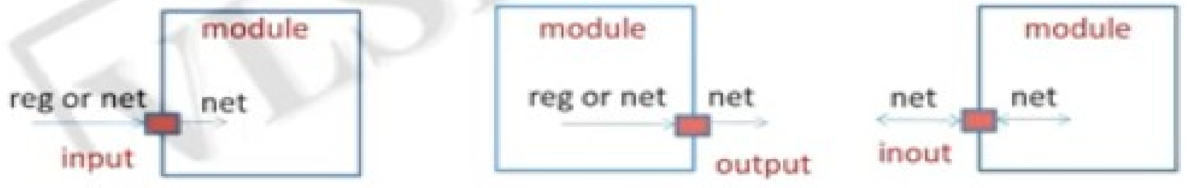


- If two signals of unequal strength are driven on a wire, the stronger signal prevails.
- If two signals of equal strengths are driven on a wire, the result is unknown.

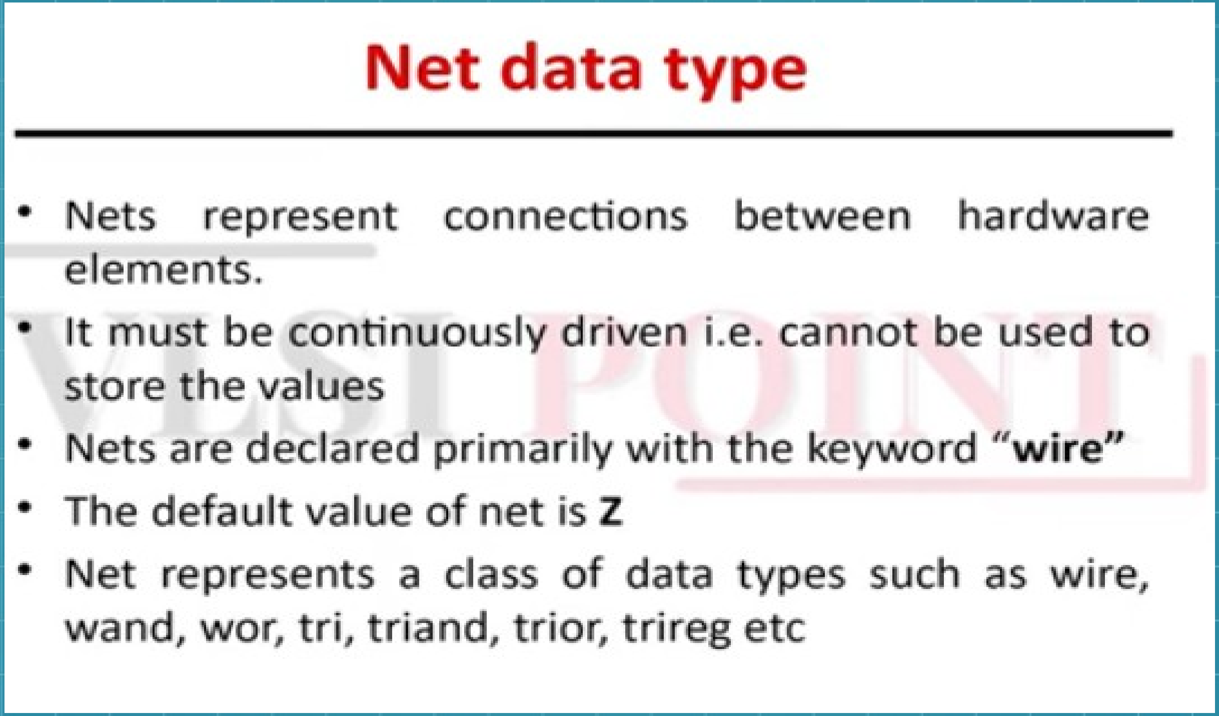
Port Assignments

Inputs, Outputs and Inouts

- **Input**- internally net , externally reg or net
- **Output**- internally reg or net , externally net
- **Inout**- only wire data type



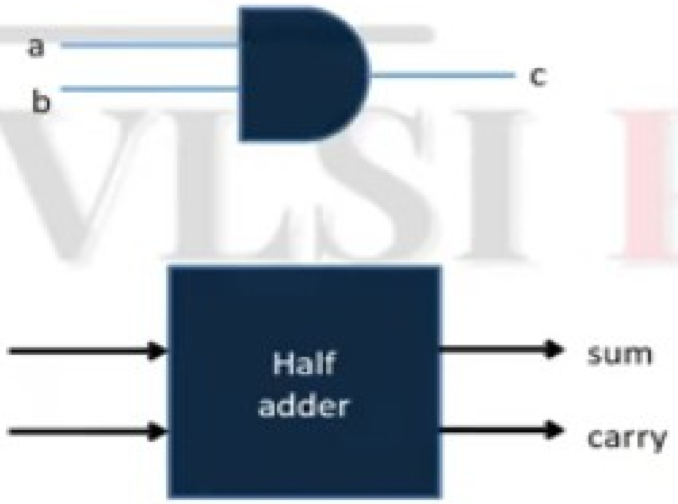
Verilog
Net data Type



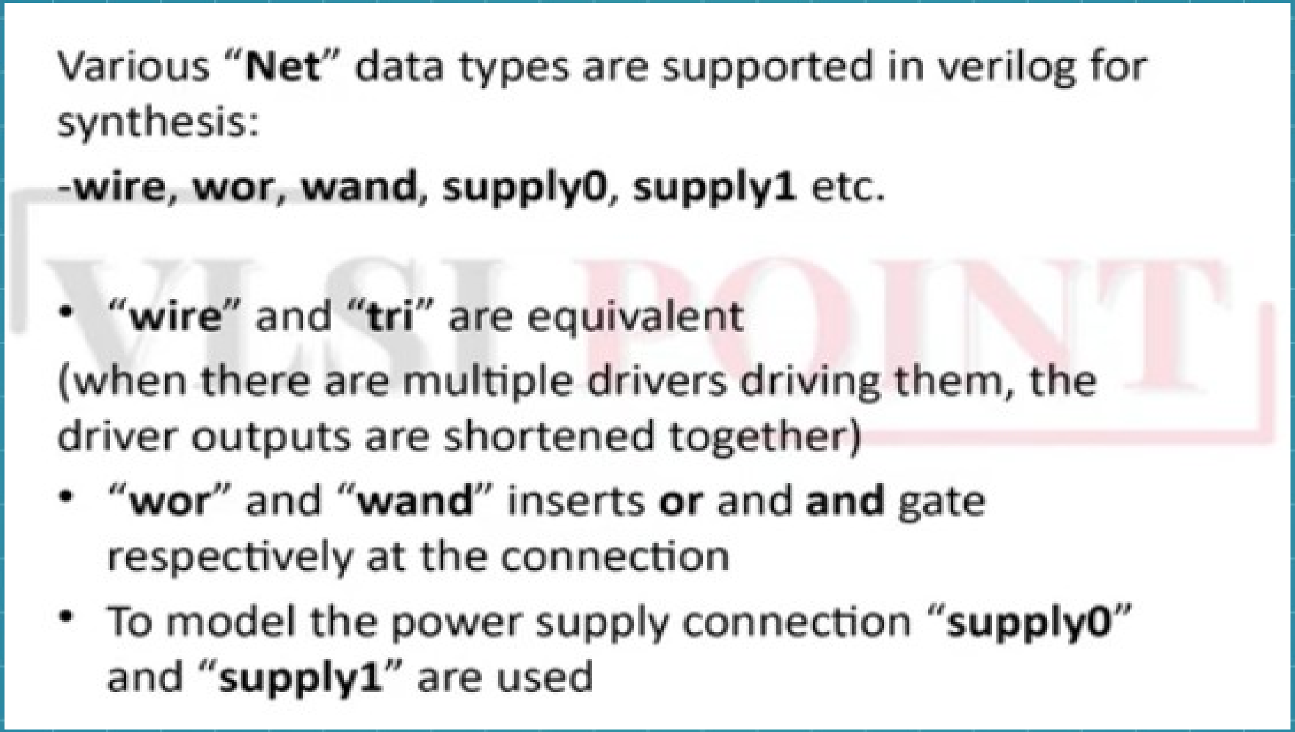
Net data type

- Nets represent connections between hardware elements.
- It must be continuously driven i.e. cannot be used to store the values
- Nets are declared primarily with the keyword “**wire**”
- The default value of net is **Z**
- Net represents a class of data types such as wire, wand, wor, tri, triand, trior, trireg etc

- Consider the below figure, Net c is connected to the output of and gate.



```
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
```



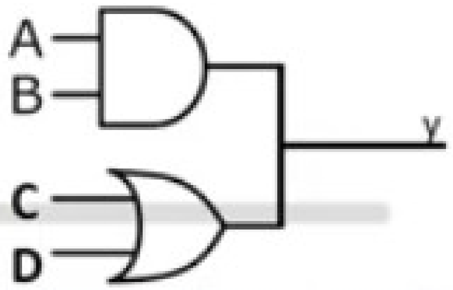
Various “**Net**” data types are supported in verilog for synthesis:

-**wire, wor, wand**, **supply0**, **supply1** etc.

- “**wire**” and “**tri**” are equivalent

(when there are multiple drivers driving them, the driver outputs are shortened together)

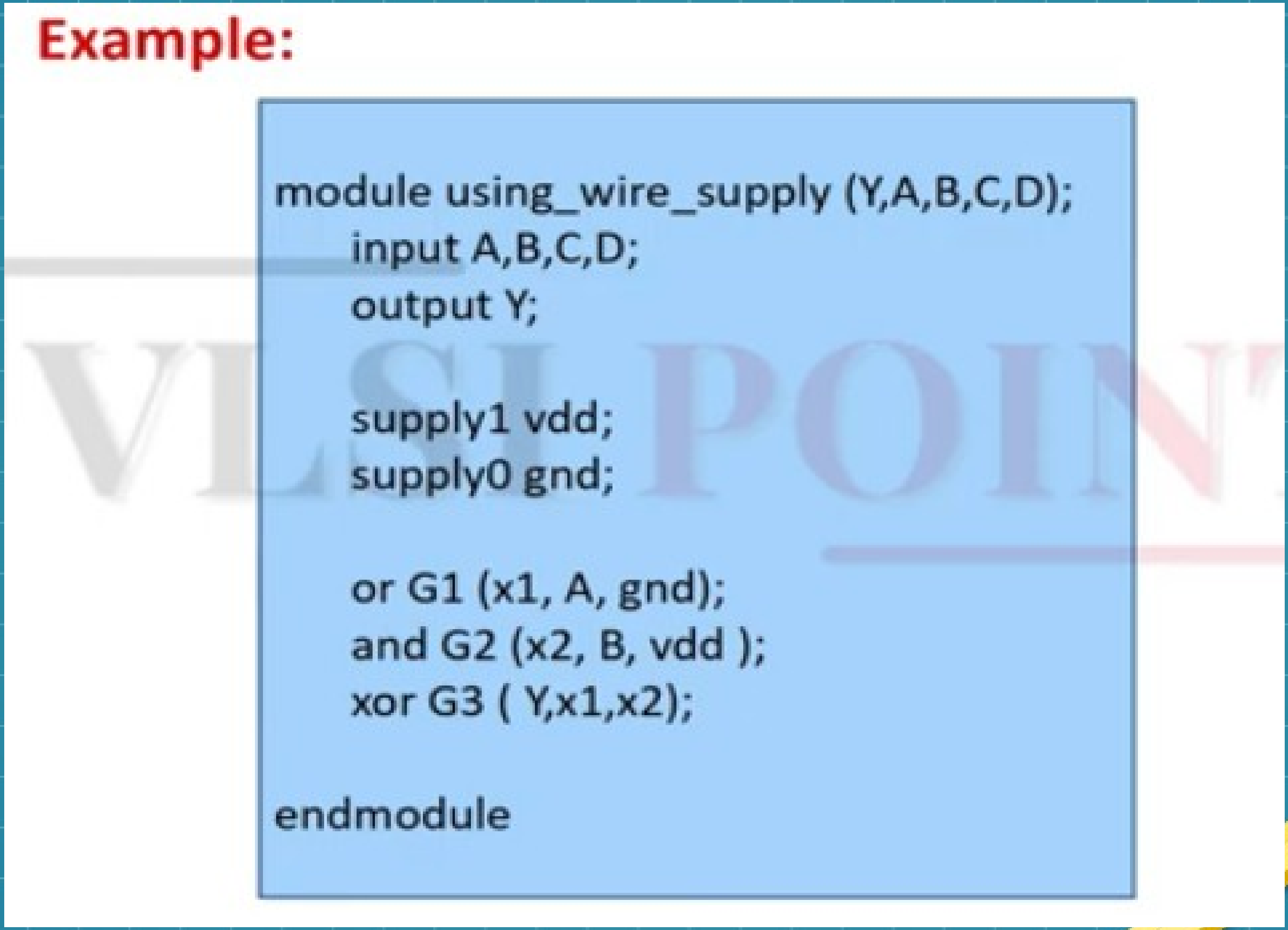
- “**wor**” and “**wand**” inserts **or** and **and** gate respectively at the connection
- To model the power supply connection “**supply0**” and “**supply1**” are used



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

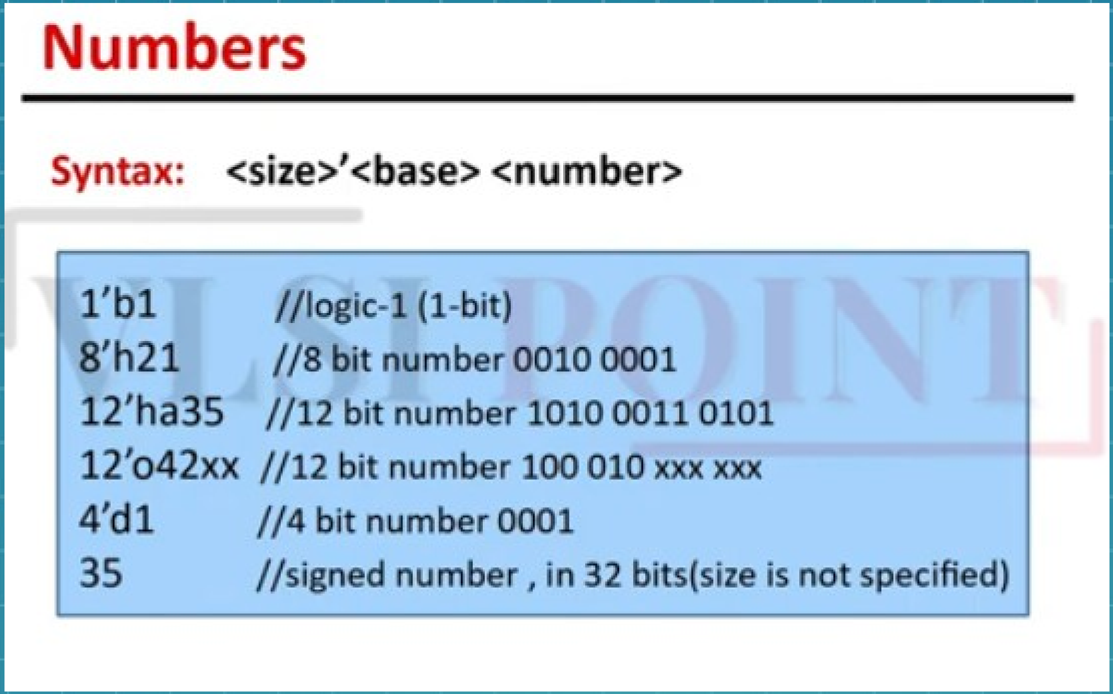
```
module use_of_wire (y,A,B,C,D);
      input A,B,C,D;
      output y;
      wire y;

      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
```



Example:

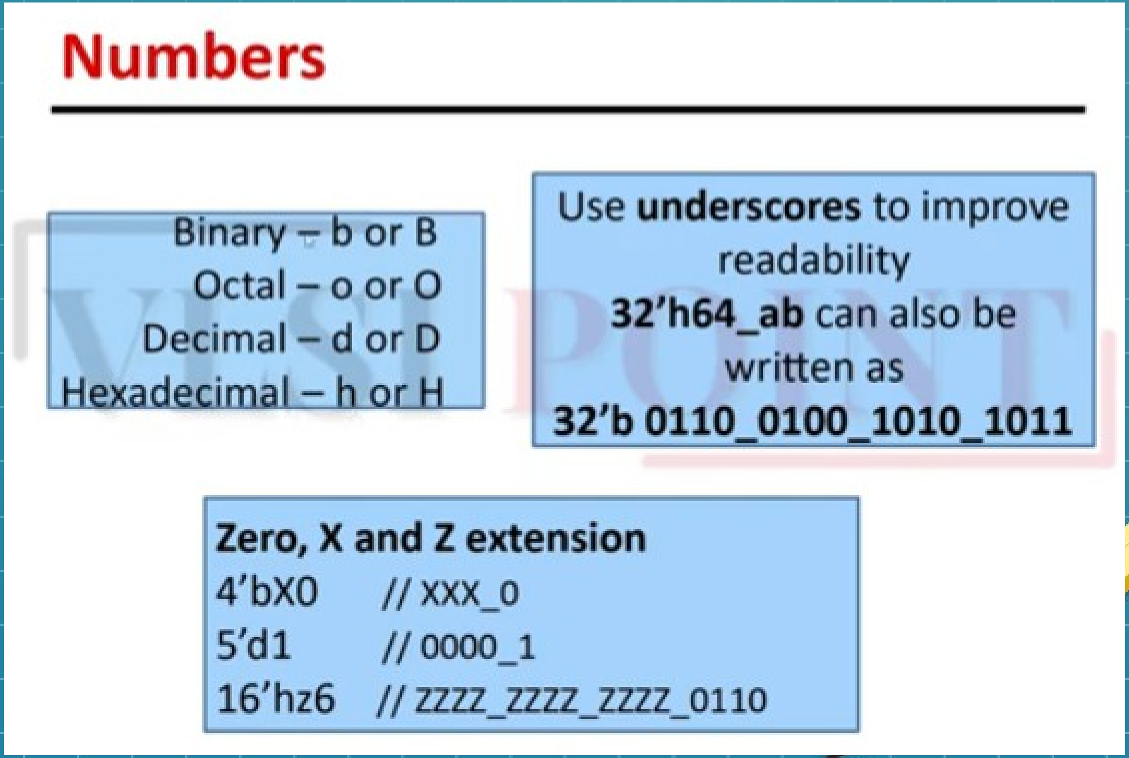
```
module using_wire_supply (Y,A,B,C,D);
    input A,B,C,D;
    output Y;

    supply1 vdd;
    supply0 gnd;

    or G1 (x1, A, gnd);
    and G2 (x2, B, vdd );
    xor G3 ( Y,x1,x2);

endmodule
```



Numbers

Syntax: <size>'<base> <number>

```
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)
```



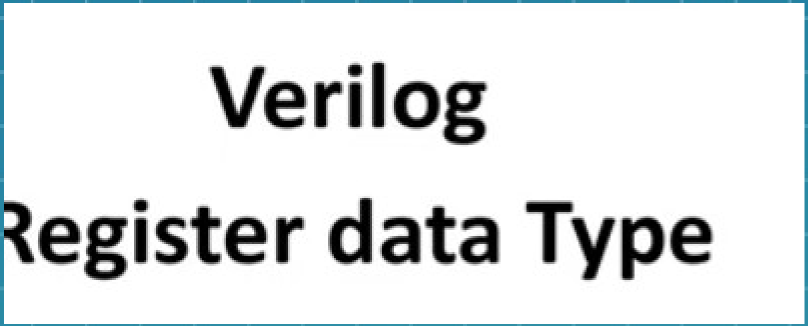
Numbers

Binary – b or B
Octal – o or O
Decimal – d or D
Hexadecimal – h or H

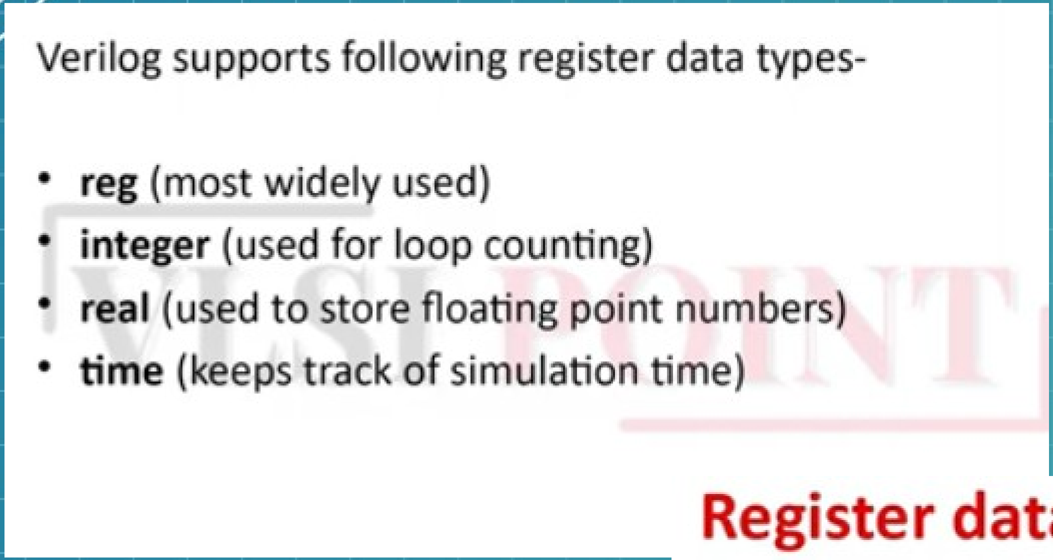
Use **underscores** to improve readability
32'h64_ab can also be written as
32'b 0110_0100_1010_1011

Zero, X and Z extension

```
4'bX0      // XXX_0
5'd1       // 0000_1
16'hz6     // ZZZZ_ZZZZ_ZZZZ_0110
```

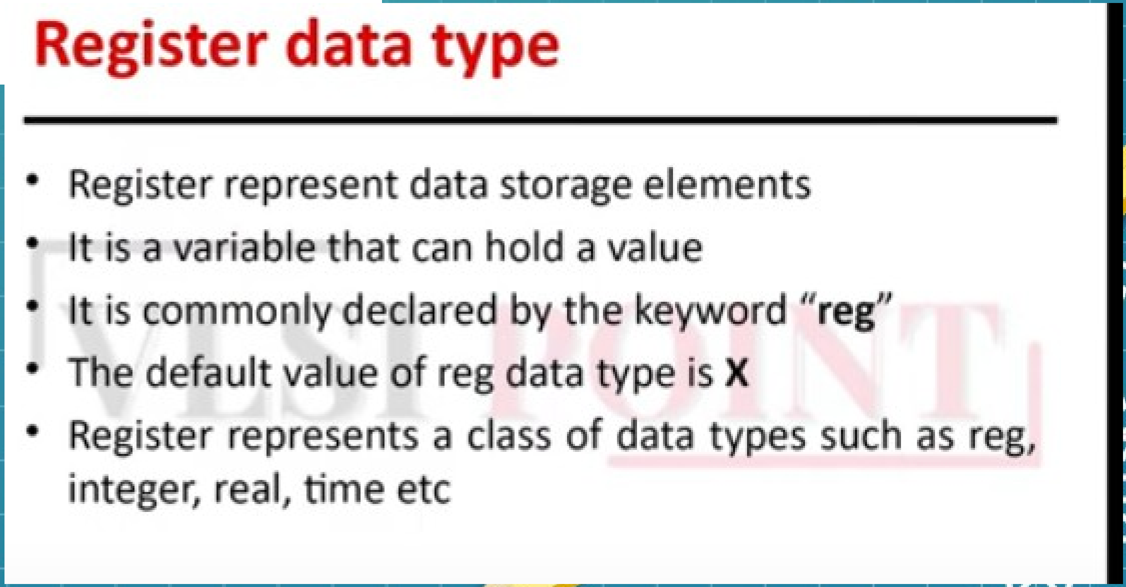


Verilog Register data Type



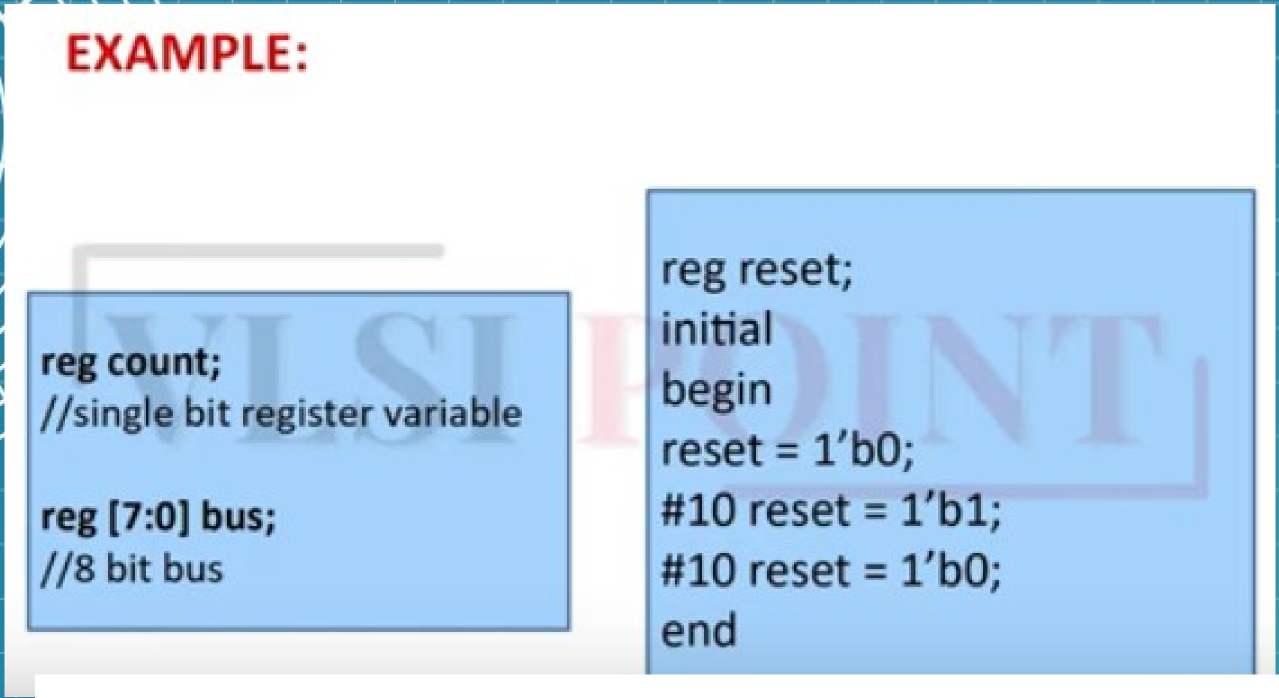
Verilog supports following register data types-

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



Register data type

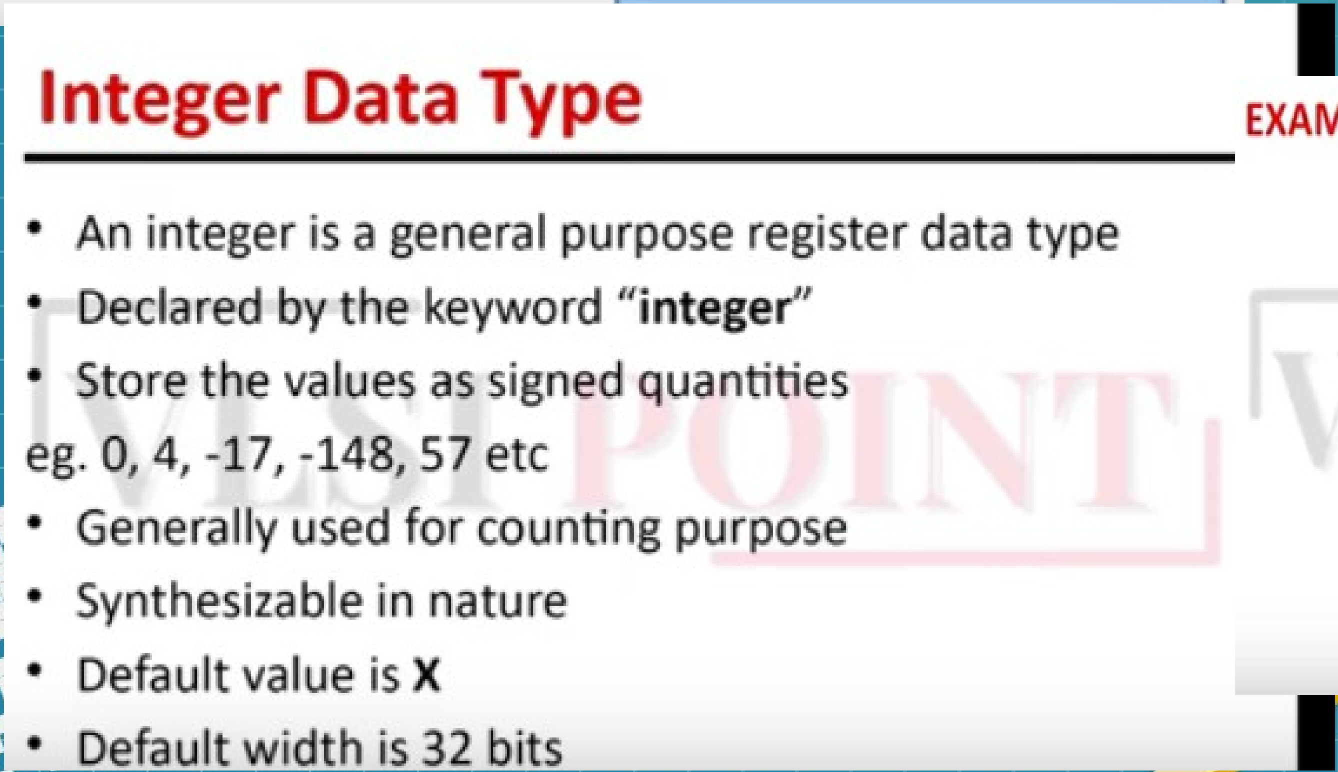
- Register represent data storage elements
- It is a variable that can hold a value
- It is commonly declared by the keyword "**reg**"
- The default value of reg data type is **X**
- Register represents a class of data types such as reg, integer, real, time etc



EXAMPLE:

```
reg count;
//single bit register variable

reg [7:0] bus;
//8 bit bus
```

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

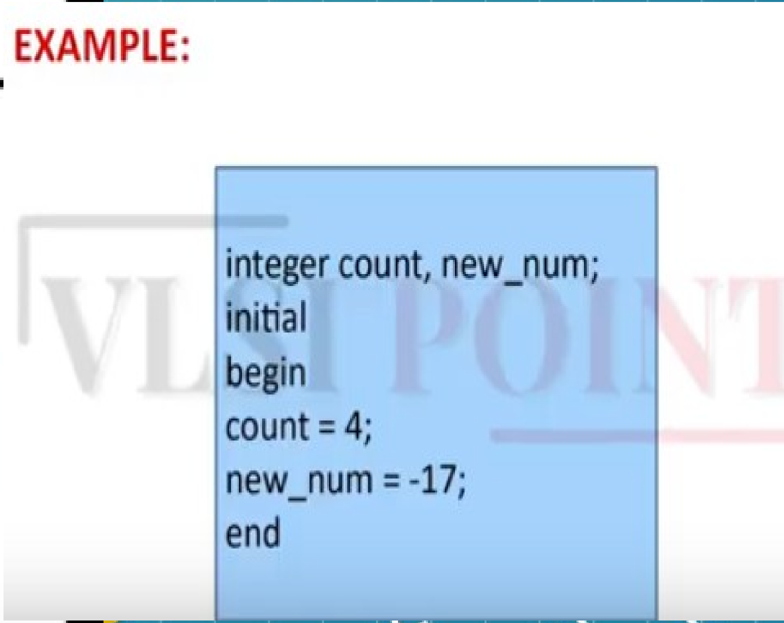


Integer Data Type

- An integer is a general purpose register data type
- Declared by the keyword "**integer**"
- Store the values as signed quantities

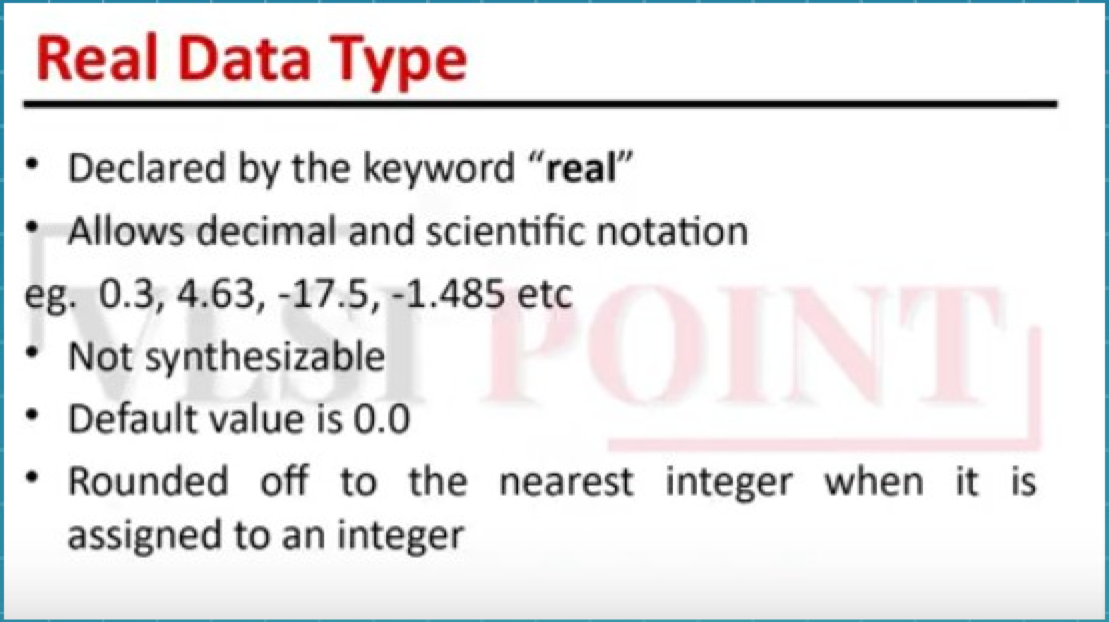
eg. 0, 4, -17, -148, 57 etc

- Generally used for counting purpose
- Synthesizable in nature
- Default value is **X**
- Default width is 32 bits



EXAMPLE:

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

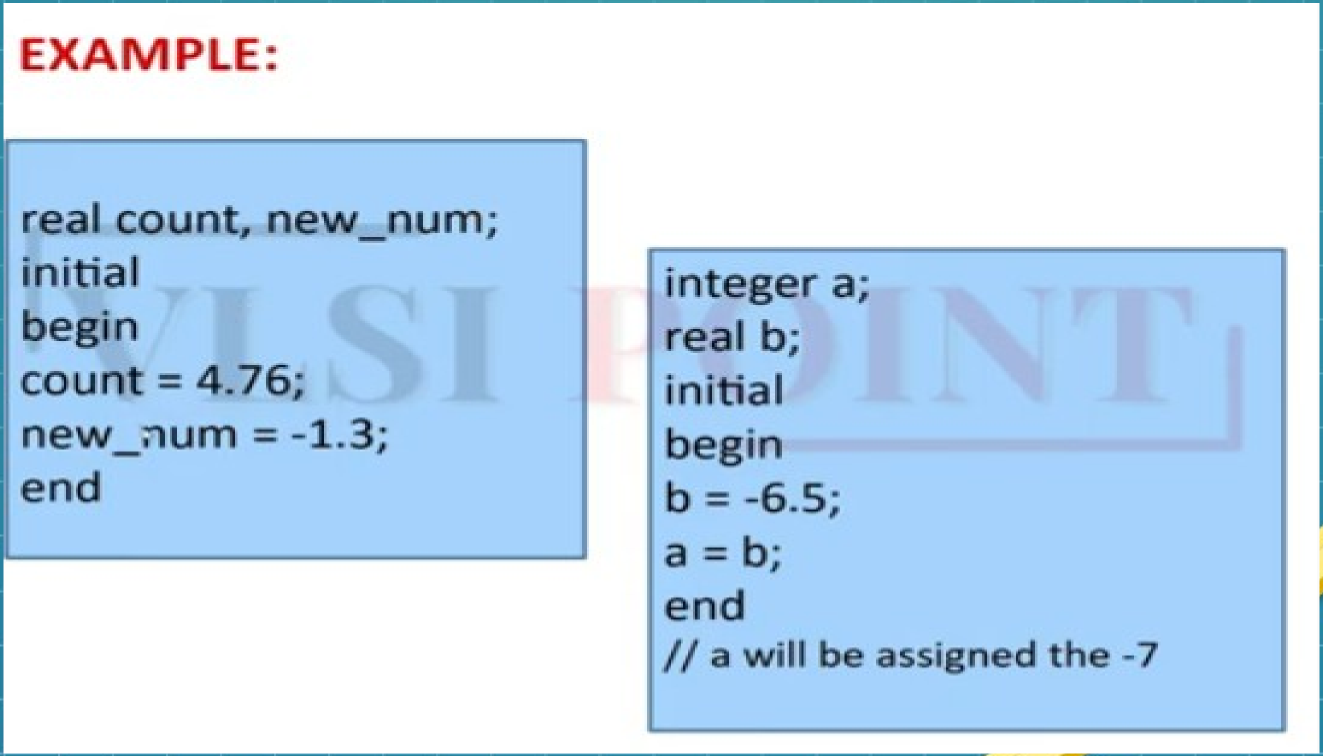


Real Data Type

- Declared by the keyword "**real**"
- Allows decimal and scientific notation

eg. 0.3, 4.63, -17.5, -1.485 etc

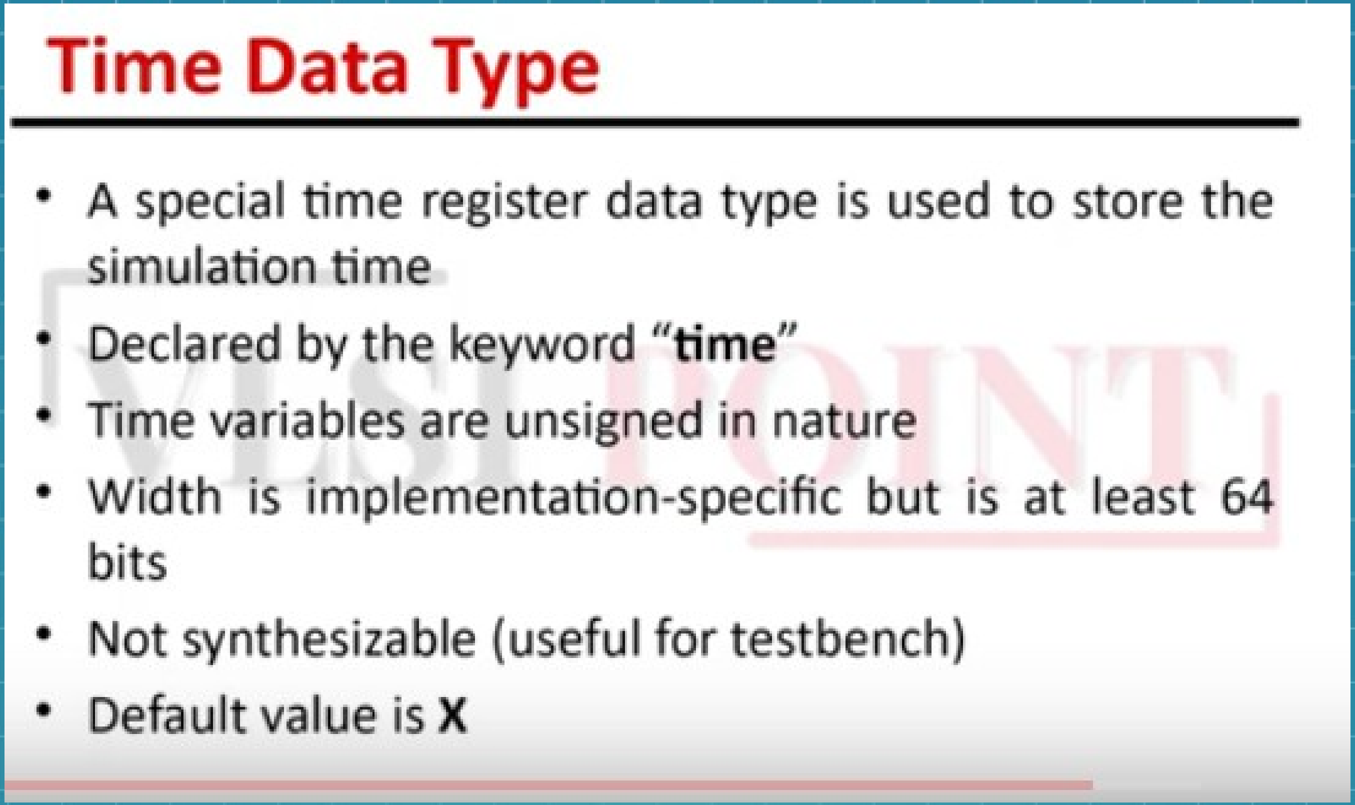
- Not synthesizable
- Default value is 0.0
- Rounded off to the nearest integer when it is assigned to an integer



EXAMPLE:

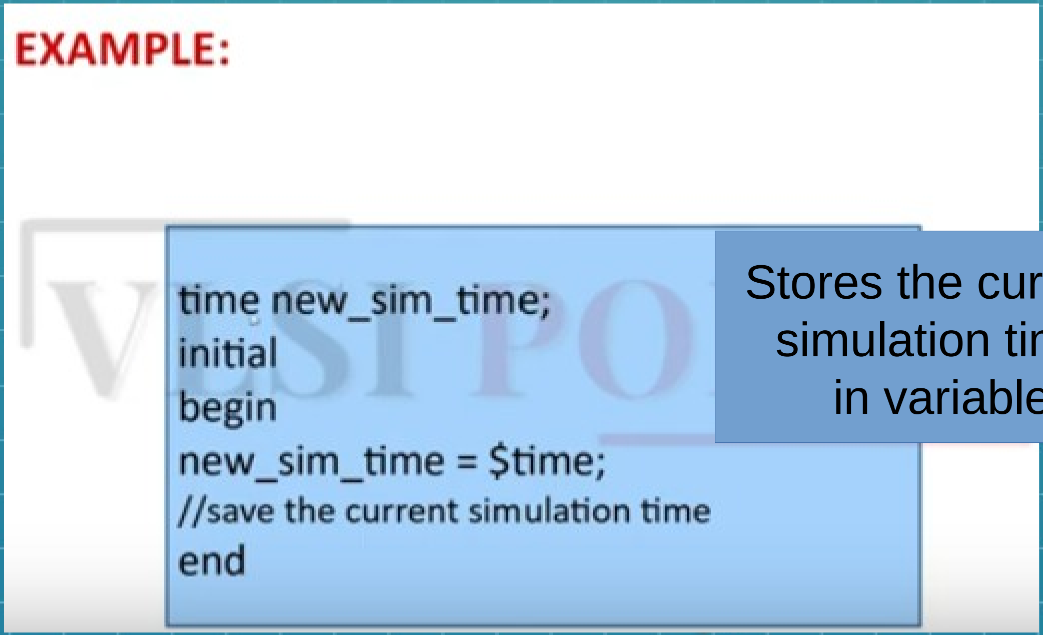
```
real count, new_num;
initial
begin
count = 4.76;
new_num = -1.3;
end
```

```
integer a;
real b;
initial
begin
b = -6.5;
a = b;
end
// a will be assigned the -7
```



Time Data Type

- A special time register data type is used to store the simulation time
- Declared by the keyword "**time**"
- Time variables are unsigned in nature
- Width is implementation-specific but is at least 64 bits
- Not synthesizable (useful for testbench)
- Default value is **X**

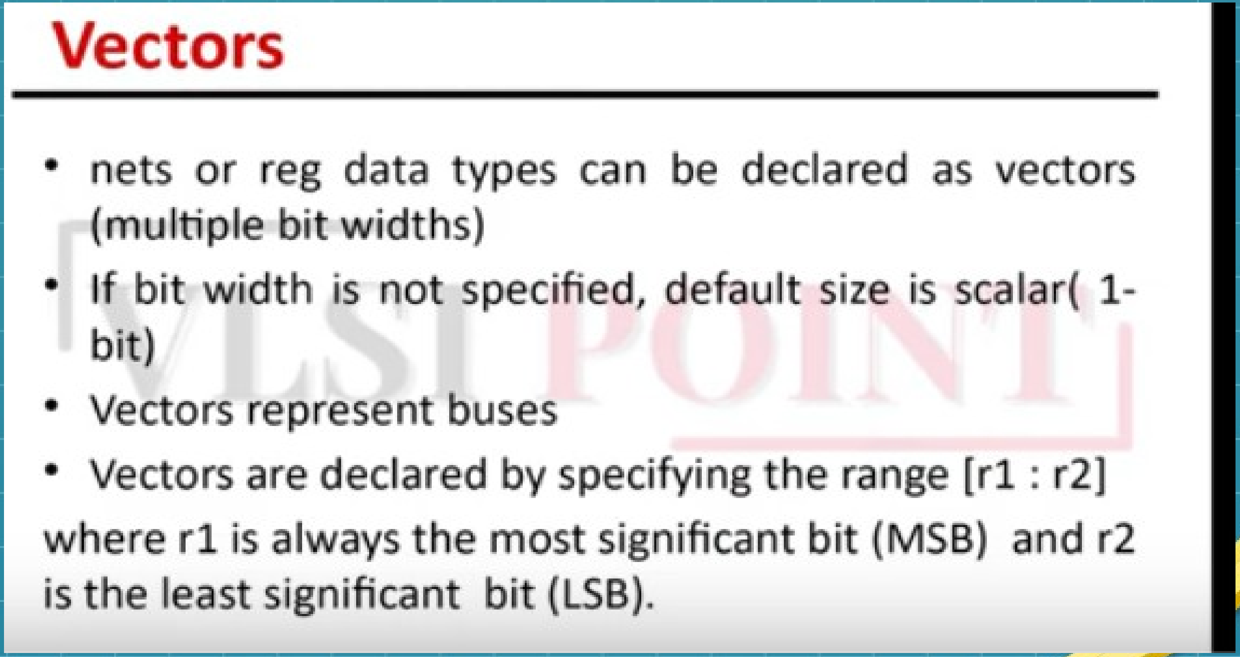


EXAMPLE:

```
time new_sim_time;
initial
begin
new_sim_time = $time;
//save the current simulation time
end
```

Stores the current simulation time in variable

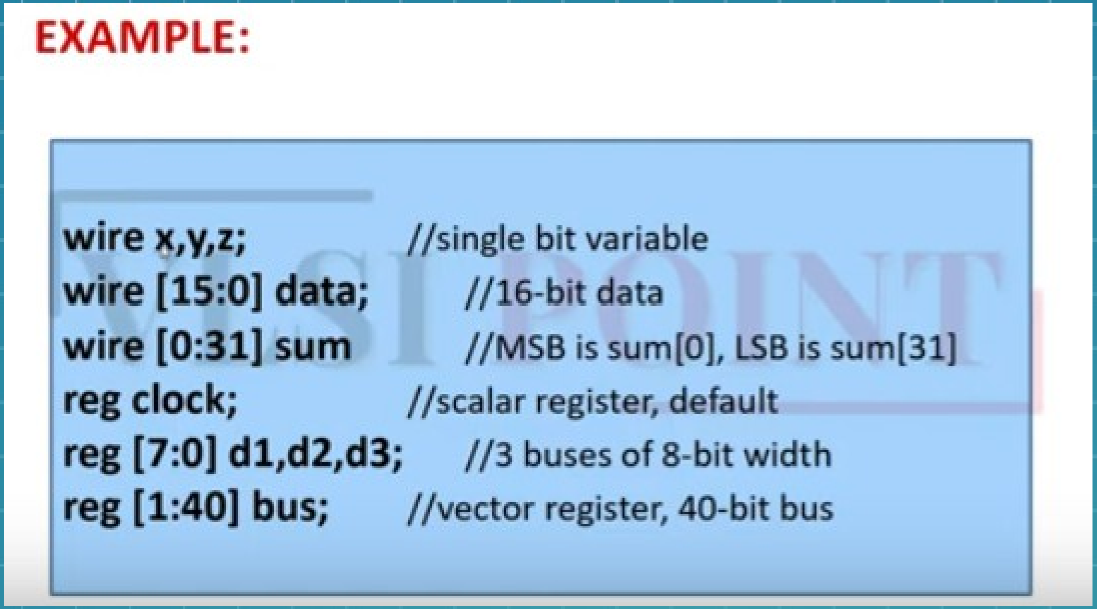
Verilog (Vectors, Arrays, Memories, Parameters, Strings)



Vectors

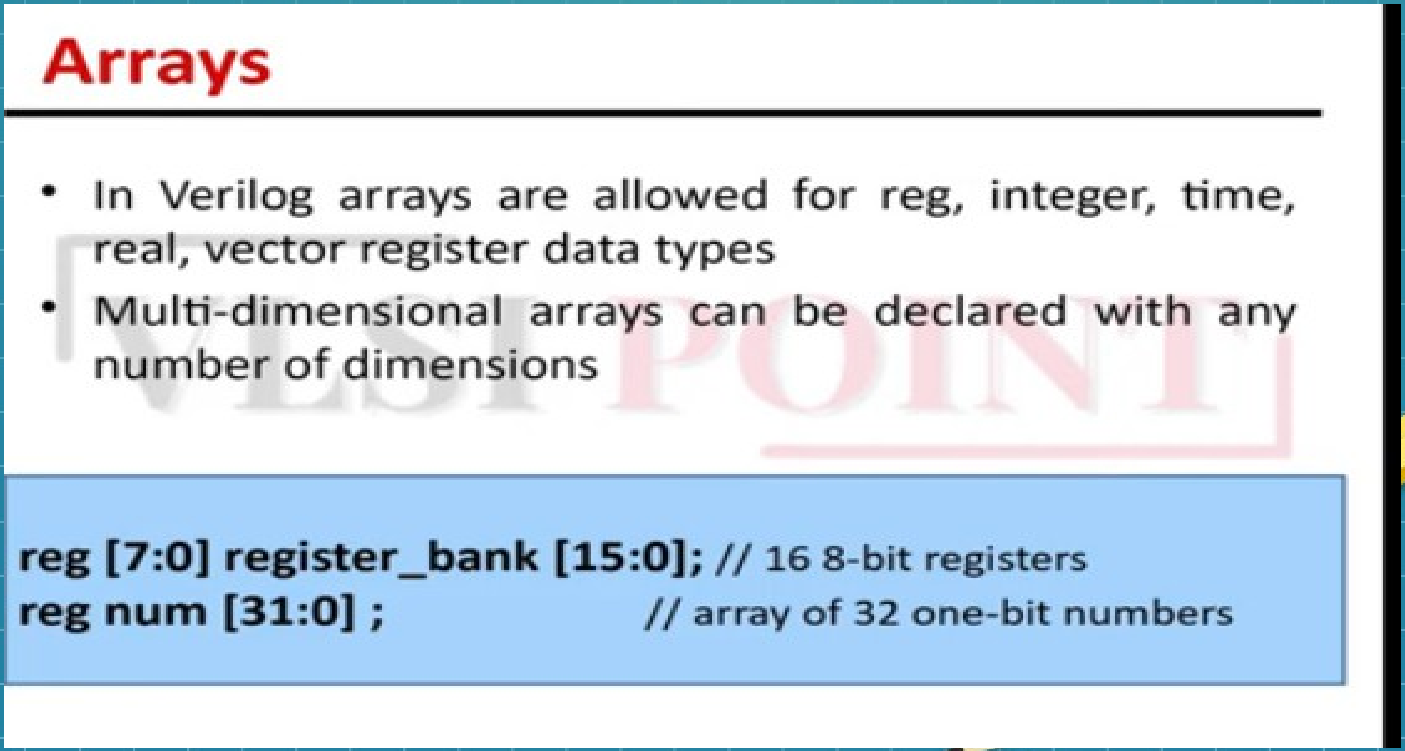
- nets or reg data types can be declared as vectors (multiple bit widths)
- If bit width is not specified, default size is scalar(1-bit)
- Vectors represent buses
- Vectors are declared by specifying the range [r1 : r2]

where r1 is always the most significant bit (MSB) and r2 is the least significant bit (LSB).



EXAMPLE:

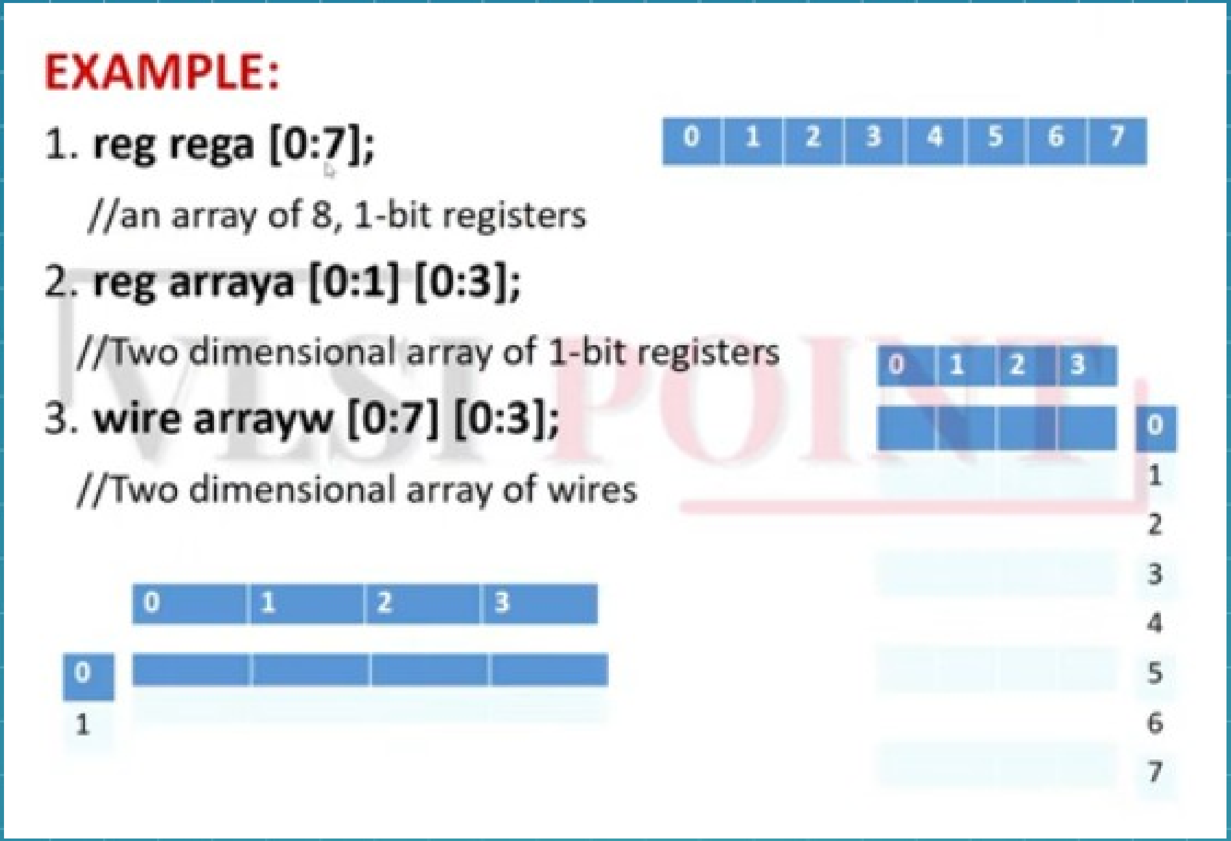
```
wire x,y,z;              //single bit variable
wire [15:0] data;        //16-bit data
wire [0:31] sum          //MSB is sum[0], LSB is sum[31]
reg clock;               //scalar register, default
reg [7:0] d1,d2,d3;      //3 buses of 8-bit width
reg [1:40] bus;          //vector register, 40-bit bus
```



Arrays

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

```
reg [7:0] register_bank [15:0]; // 16 8-bit registers
reg num [31:0] ;                // array of 32 one-bit numbers
```



EXAMPLE:

1. **reg rega [0:7];**

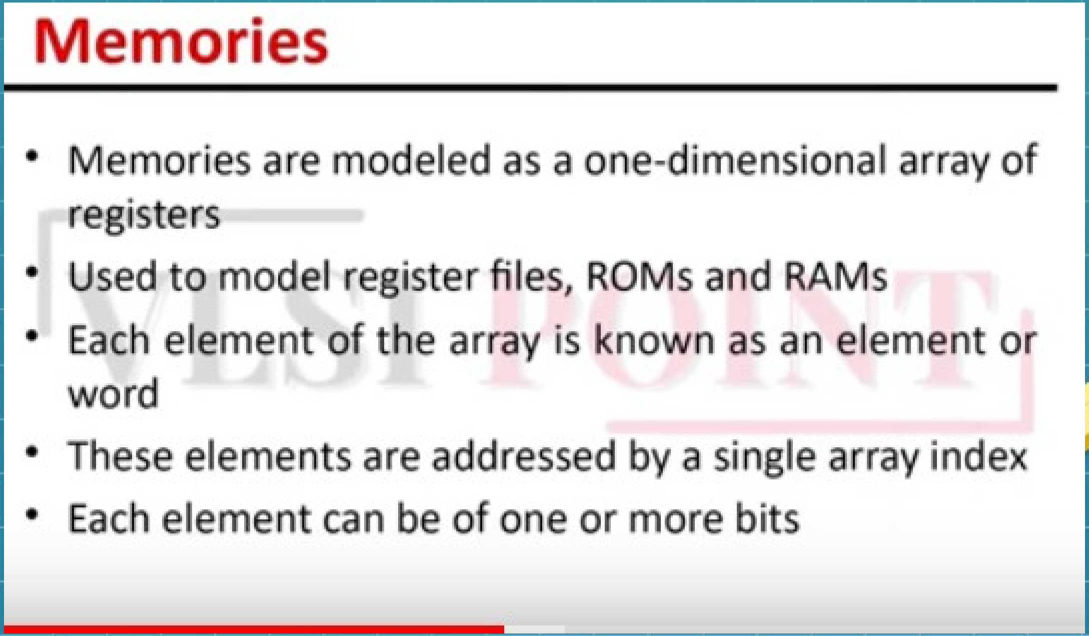
   //an array of 8, 1-bit registers

2. **reg arraya [0:1] [0:3];**

   //Two dimensional array of 1-bit registers

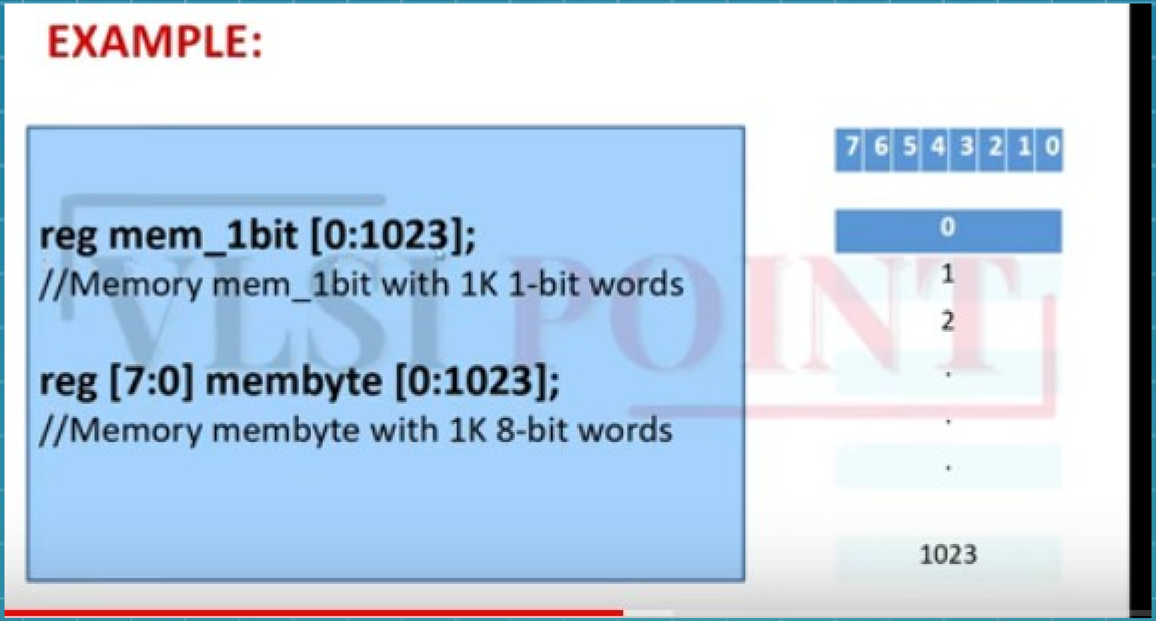
3. **wire arrayw [0:7] [0:3];**

   //Two dimensional array of wires



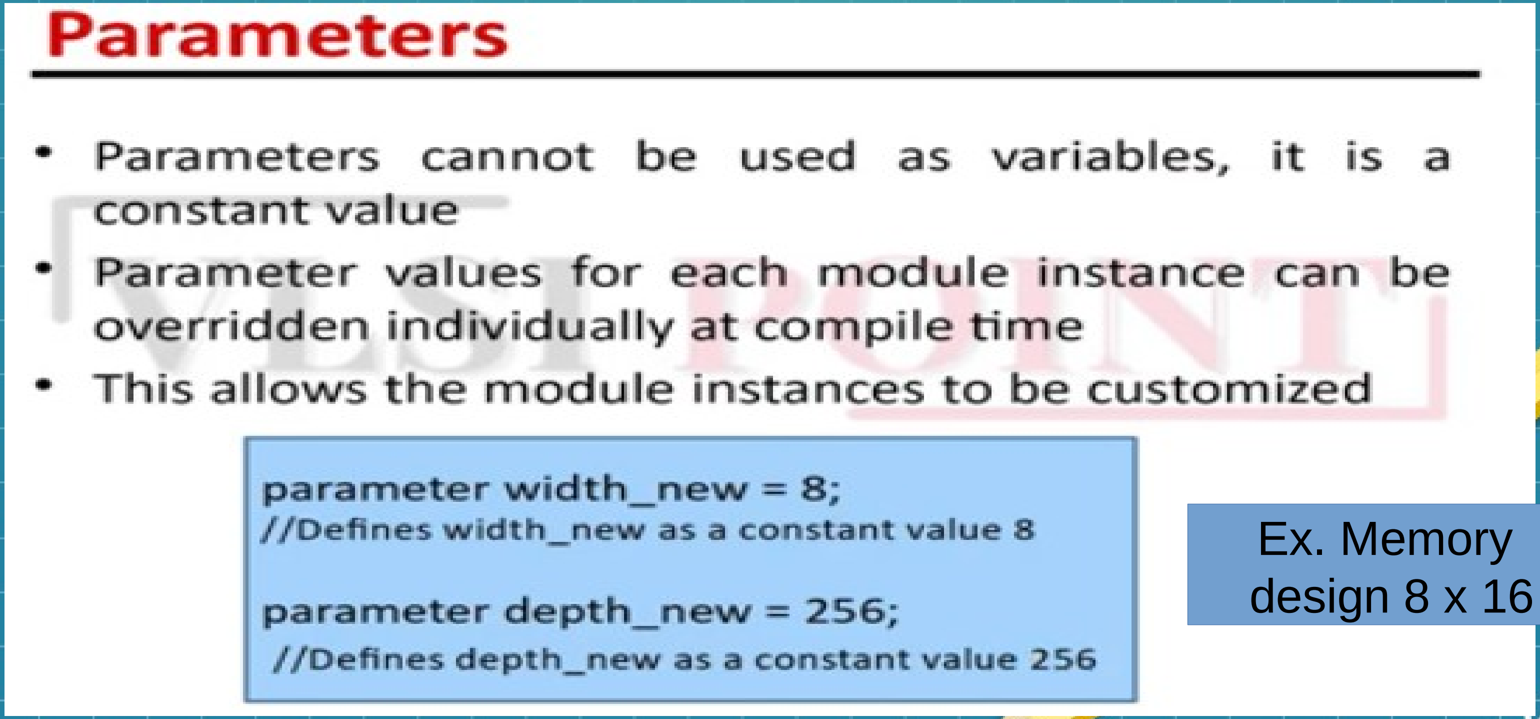
Memories

- Memories are modeled as a one-dimensional array of registers
- Used to model register files, ROMs and RAMs
- Each element of the array is known as an element or word
- These elements are addressed by a single array index
- Each element can be of one or more bits



EXAMPLE:

```
reg mem_1bit [0:1023];
//Memory mem_1bit with 1K 1-bit words

reg [7:0] membyte [0:1023];
//Memory membyte with 1K 8-bit words
```

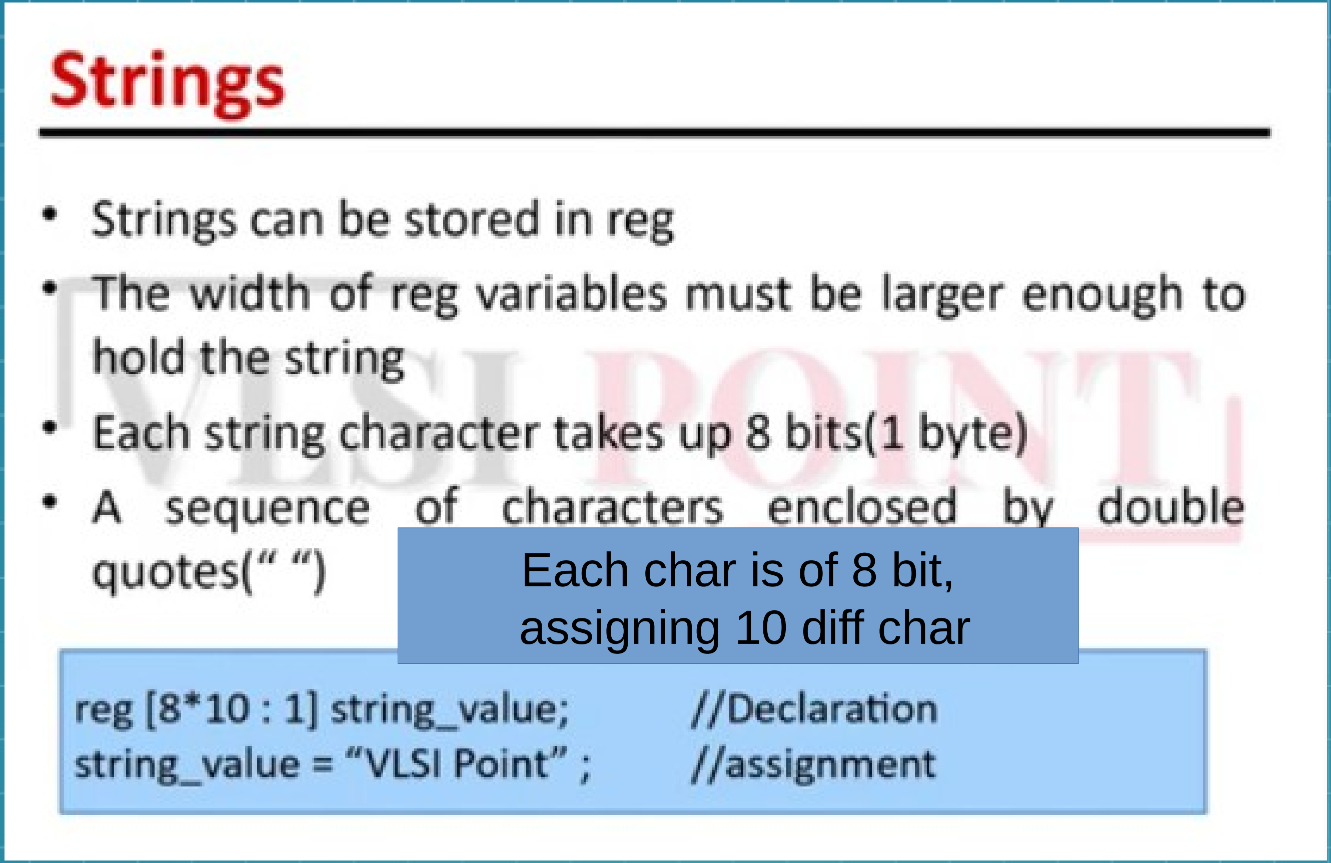


Parameters

- Parameters cannot be used as variables, it is a constant value
- Parameter values for each module instance can be overridden individually at compile time
- This allows the module instances to be customized

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

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

Ex. Memory design 8 x 16

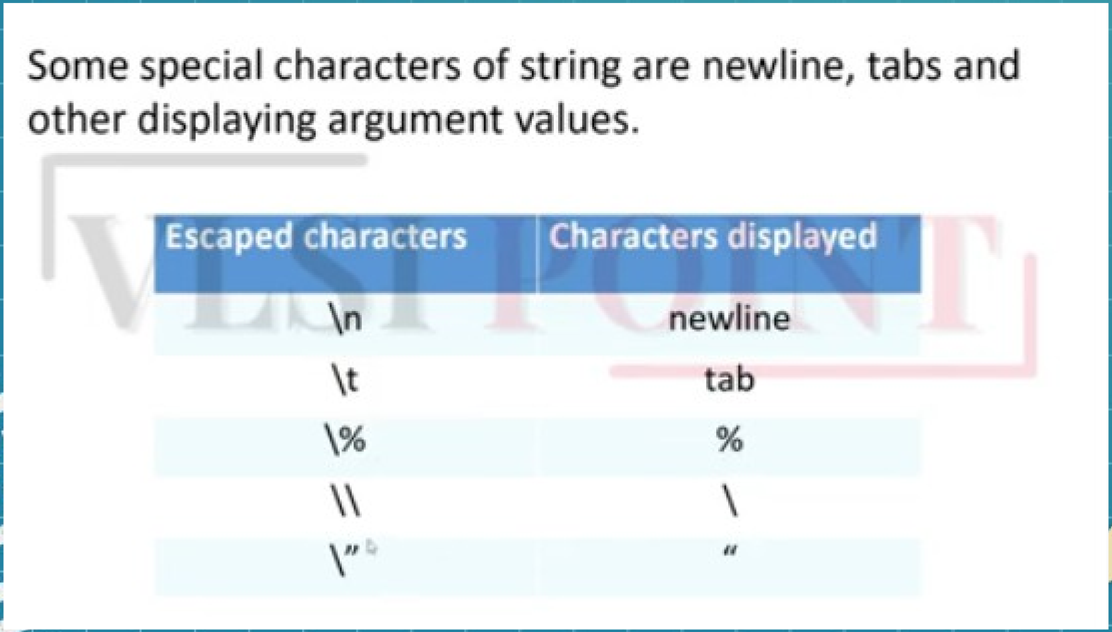


Strings

- Strings can be stored in reg
- The width of reg variables must be larger enough to hold the string
- Each string character takes up 8 bits(1 byte)
- A sequence of characters enclosed by double quotes(“ “)

```
reg [8*10 : 1] string_value;          //Declaration
string_value = “VLSI Point” ;         //assignment
```

Each char is of 8 bit,
assigning 10 diff char



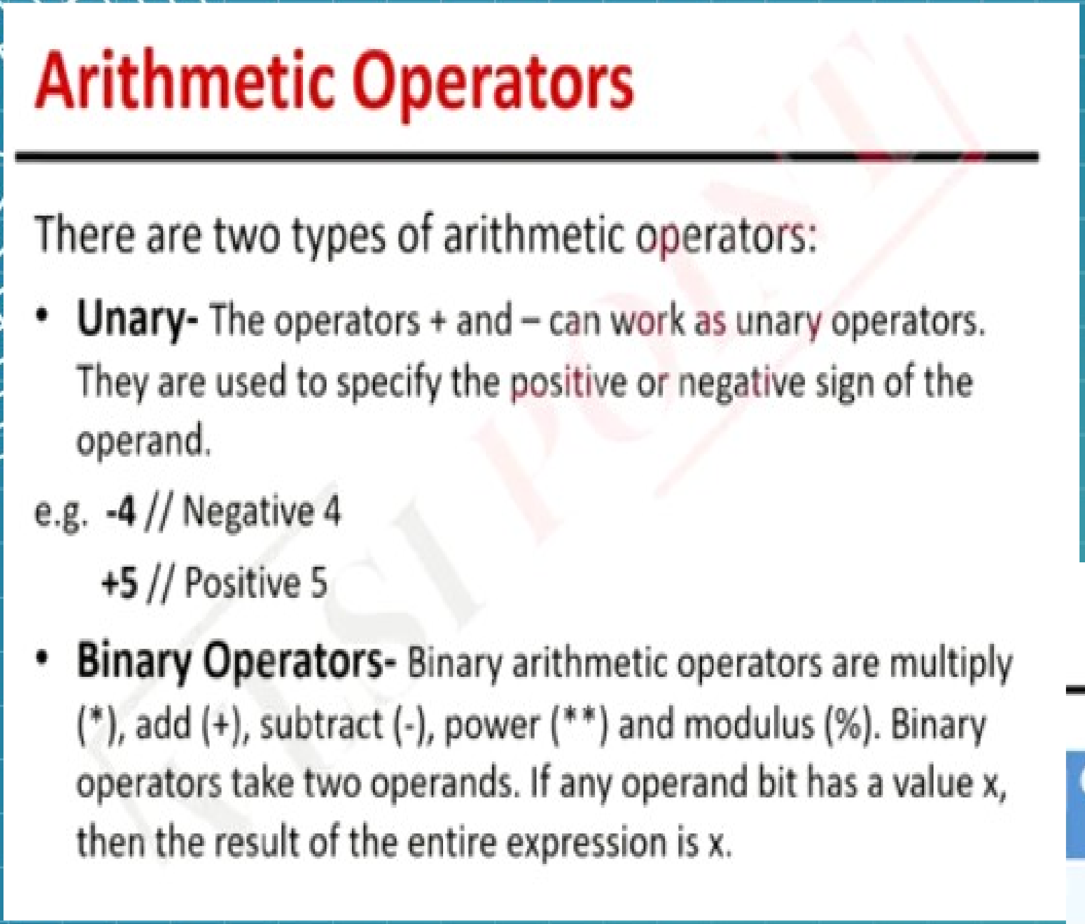
Some special characters of string are newline, tabs and other displaying argument values.

Escaped characters	Characters displayed
\n	newline
\t	tab
\%	%
\\	\
\”	“

Operators In Verilog

Verilog provides various types of operators

- Arithmetic Operators
- Logical Operators
- Bitwise Operators
- Equality Operators
- Relational Operator
- Reduction Operator
- Shift Operator
- Concatenation
- Conditional



Arithmetic Operators

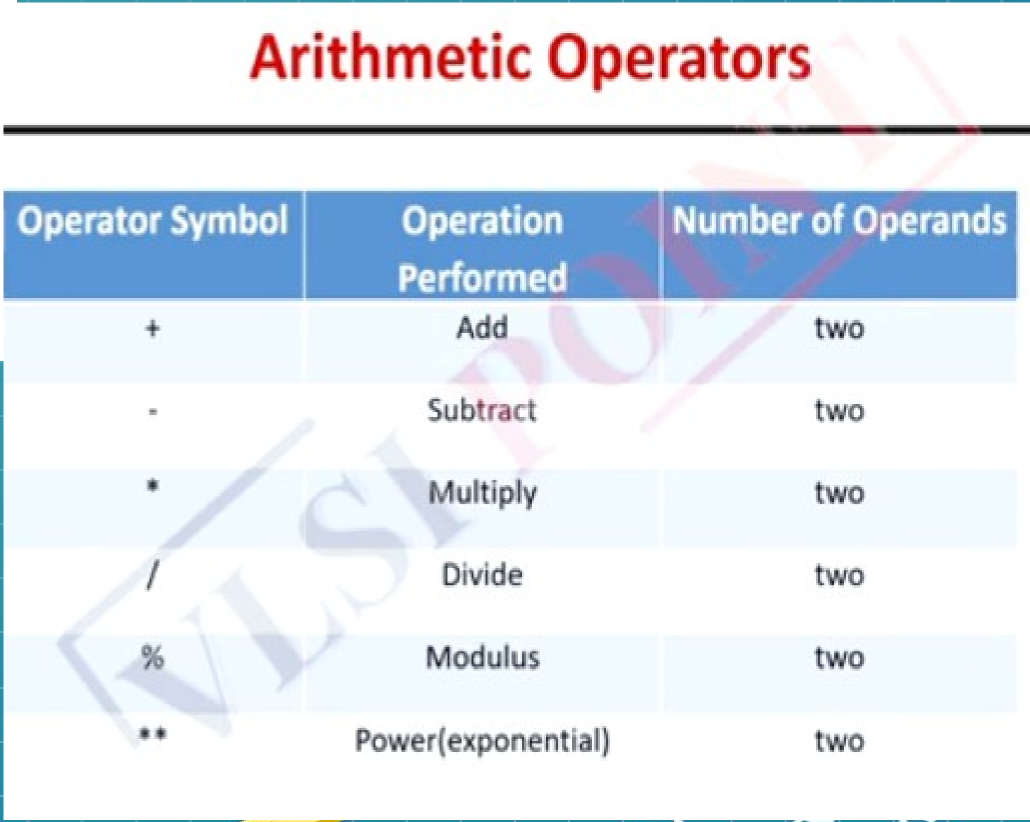
There are two types of arithmetic operators:

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

e.g. **-4** // Negative 4

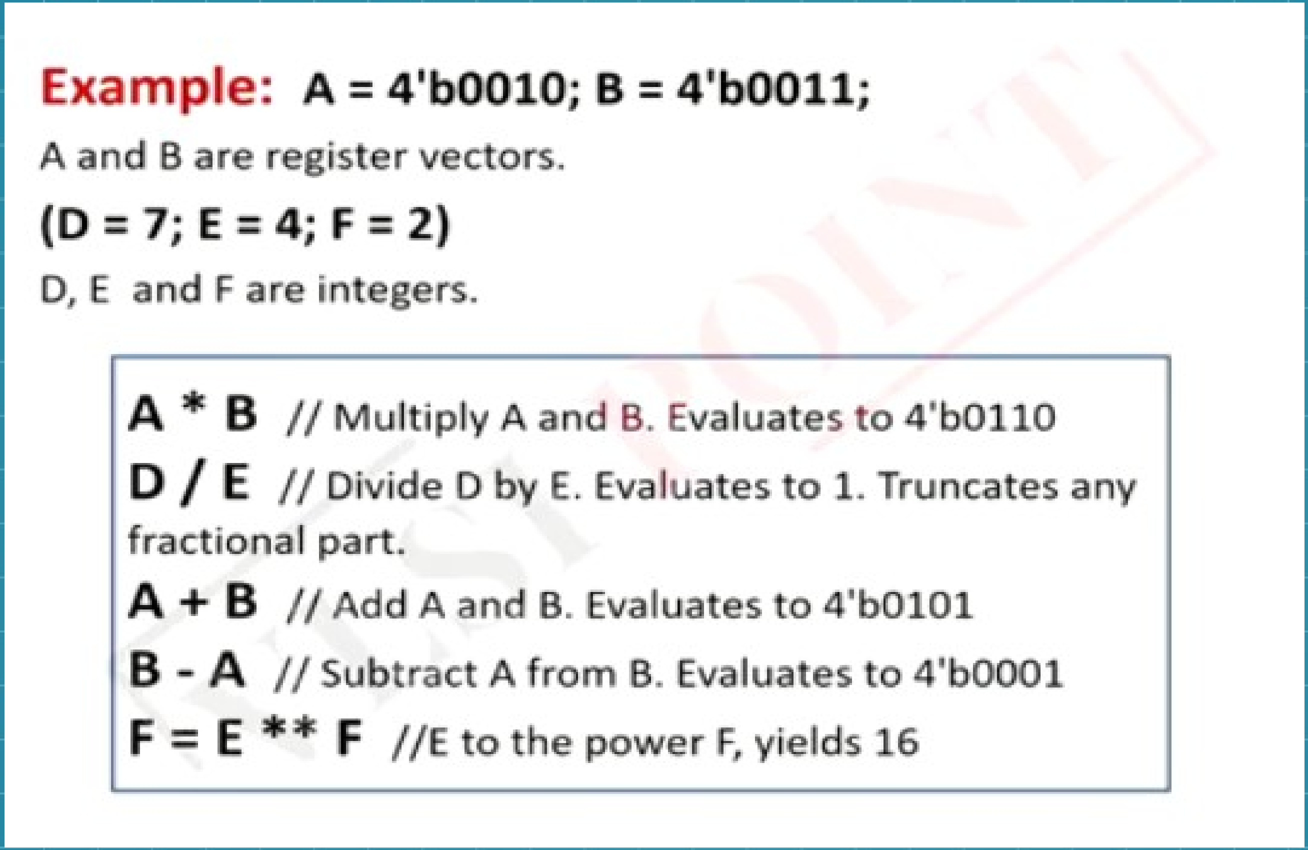
+5 // Positive 5

- **Binary Operators-** Binary arithmetic operators are multiply (*), add (+), subtract (-), power (**) and modulus (%). Binary operators take two operands. If any operand bit has a value x, then the result of the entire expression is x.



Arithmetic Operators

Operator Symbol	Operation Performed	Number of Operands
+	Add	two
-	Subtract	two
*	Multiply	two
/	Divide	two
%	Modulus	two
**	Power(exponential)	two



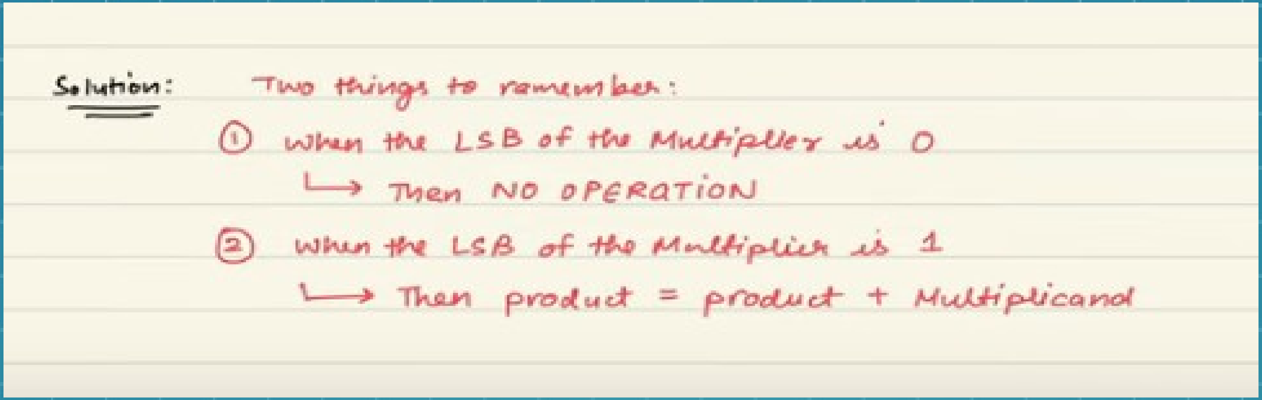
Example: A = 4'b0010; B = 4'b0011;

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
```



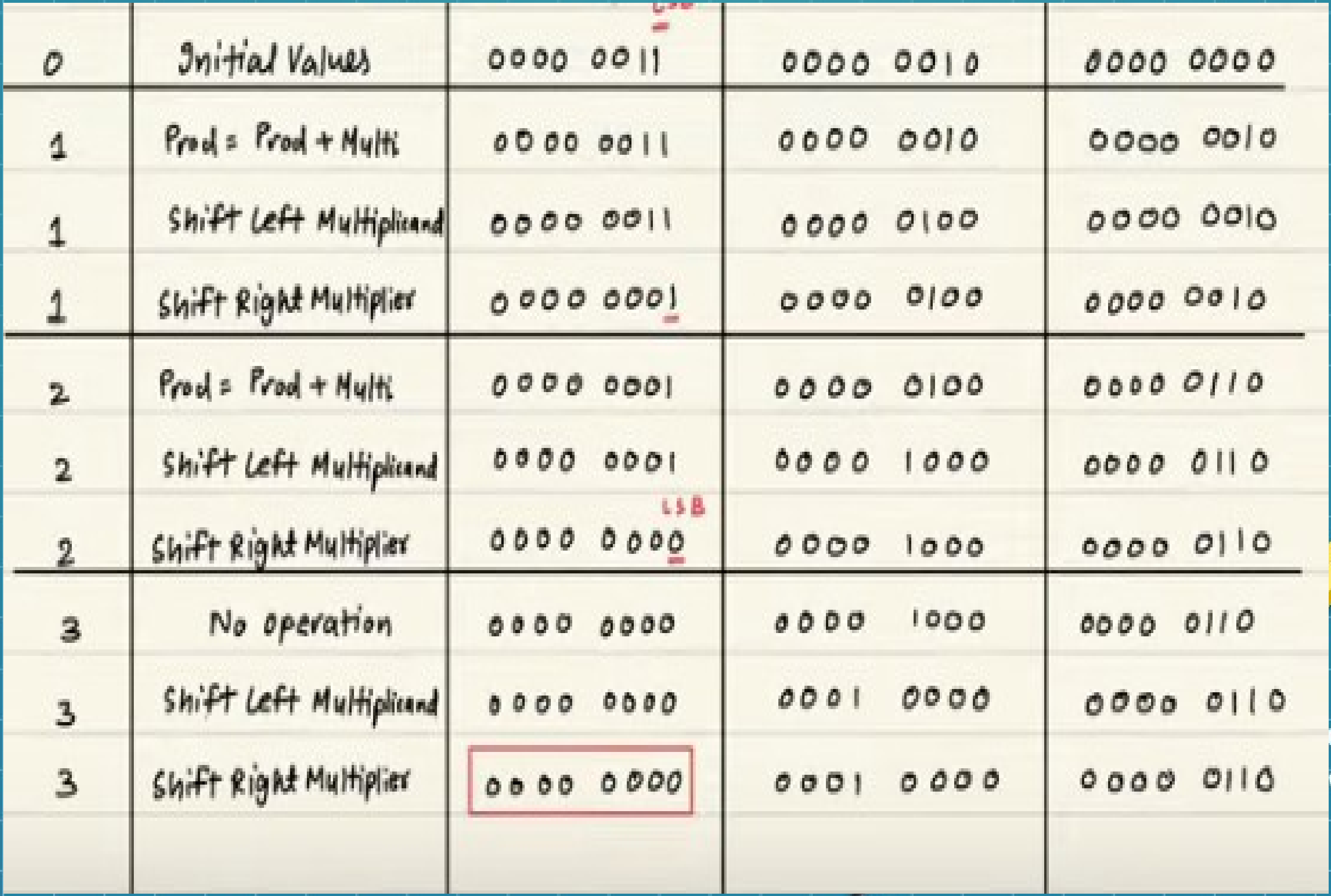
Solution: Two things to remember:

① when the LSB of the Multiplier is 0

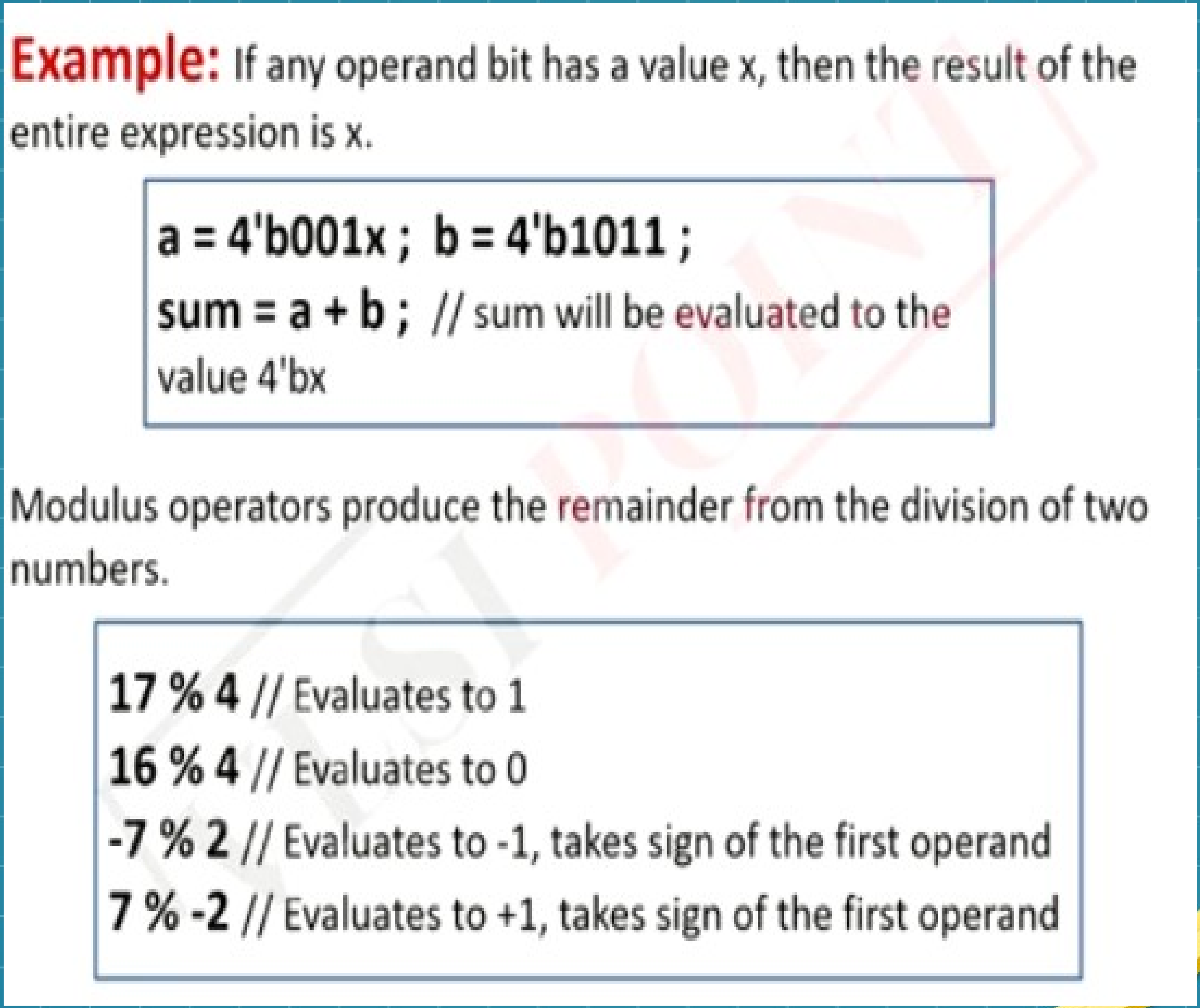
$\hookrightarrow$ Then NO OPERATION

② when the LSB of the Multiplier is 1

$\hookrightarrow$ Then product = product + Multiplicand



0	Initial Values	0000 001$\underline{1}$	0000 0010	0000 0000
1	Prod = Prod + Multi	0000 0011	0000 0010	0000 0010
1	Shift Left Multiplicand	0000 0011	0000 0100	0000 0010
1	Shift Right Multiplier	0000 000$\underline{1}$	0000 0100	0000 0010
2	Prod = Prod + Multi	0000 0001	0000 0100	0000 0110
2	Shift Left Multiplicand	0000 0001	0000 1000	0000 0110
2	Shift Right Multiplier	0000 000$\overset{\text{LSB}}{\underline{0}}$	0000 1000	0000 0110
3	No operation	0000 0000	0000 1000	0000 0110
3	Shift Left Multiplicand	0000 0000	0001 0000	0000 0110
3	Shift Right Multiplier	$\boxed{0000\ 0000}$	0001 0000	0000 0110

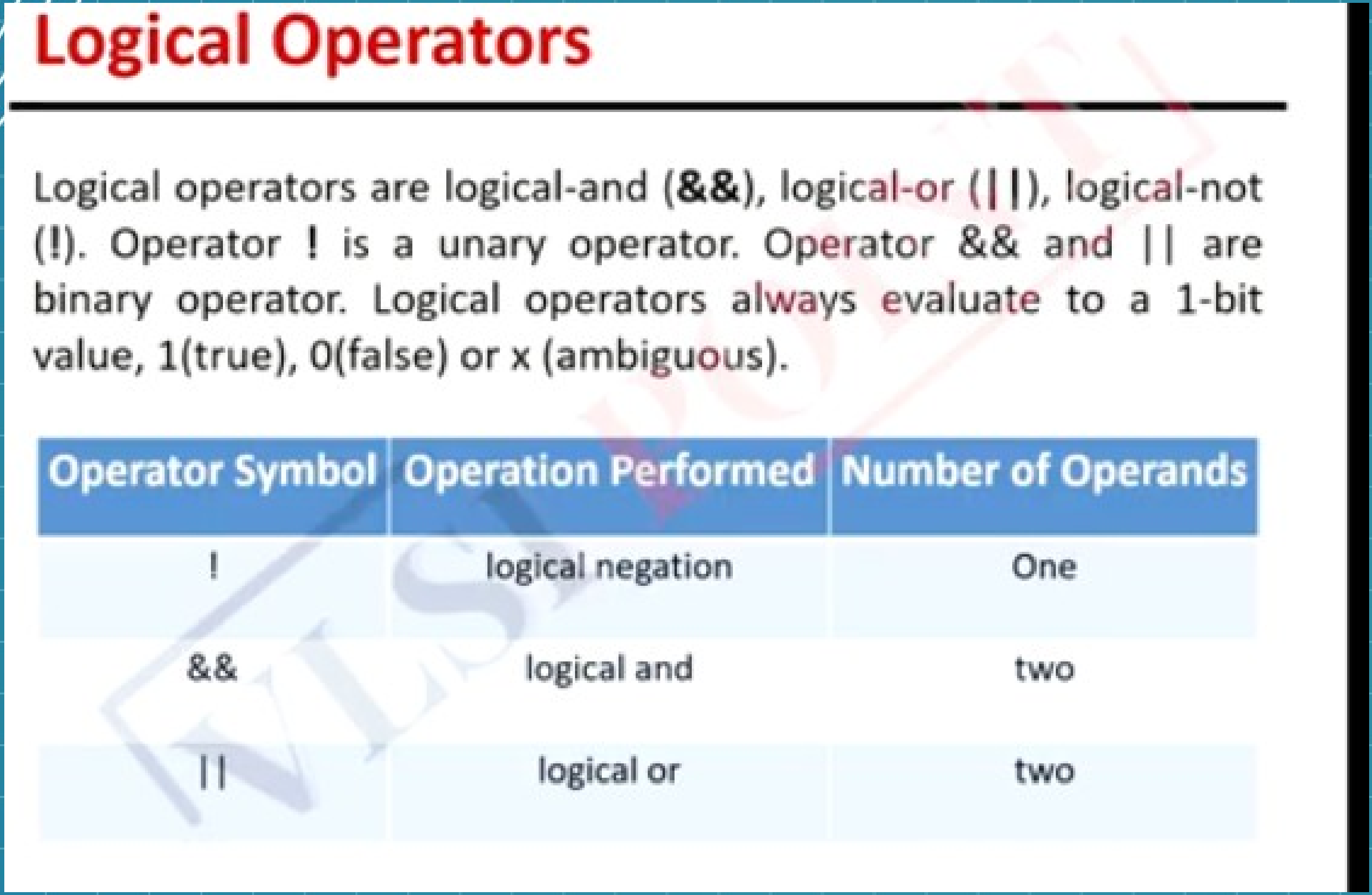


Example: If any operand bit has a value x, then the result of the entire expression is x.

```
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.

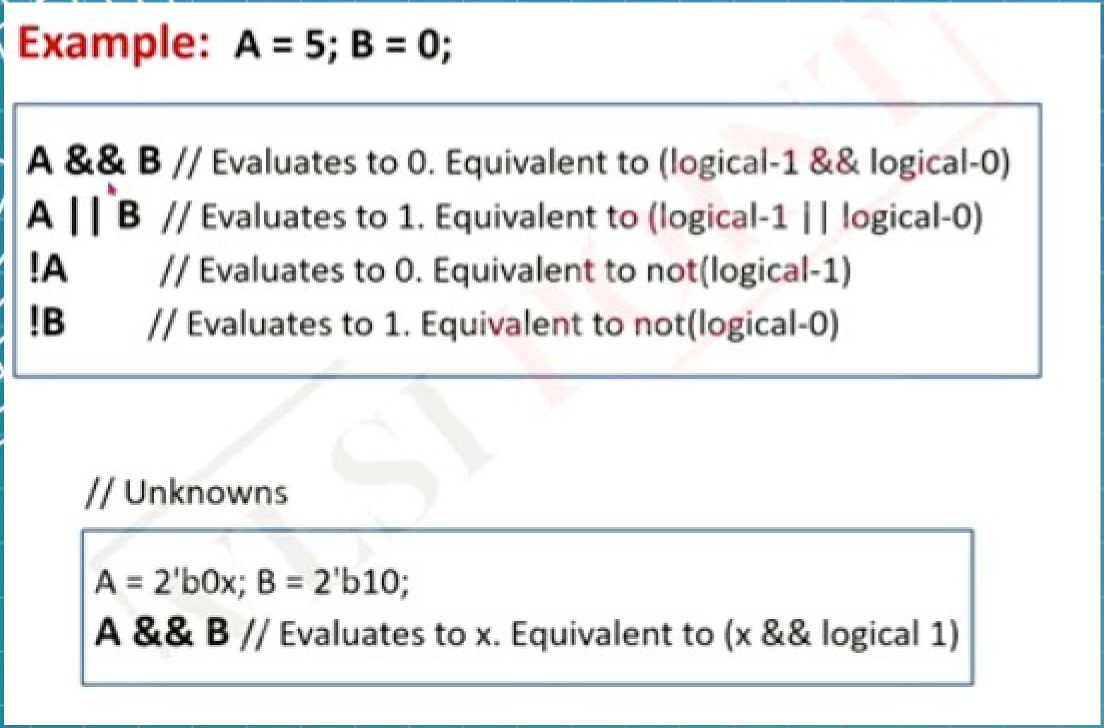
```
17 % 4 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first operand
```



Logical Operators

Logical operators are logical-and (**&&**), logical-or (||), logical-not (!). Operator **!** is a unary operator. Operator && and || are binary operator. Logical operators always evaluate to a 1-bit value, 1(true), 0(false) or x (ambiguous).

Operator Symbol	Operation Performed	Number of Operands
!	logical negation	One
&&	logical and	two
||	logical or	two

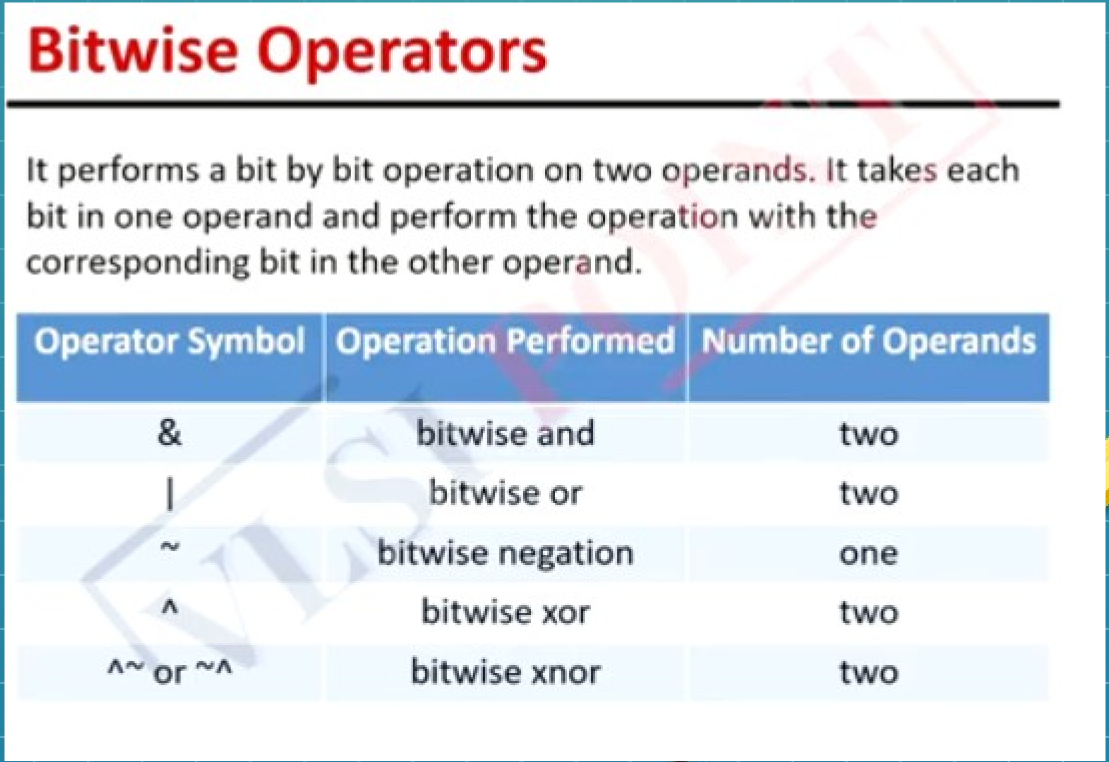


Example: **A = 5; B = 0;**

```
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B  // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A       // Evaluates to 0. Equivalent to not(logical-1)
!B      // Evaluates to 1. Equivalent to not(logical-0)
```

```
// Unknowns
```

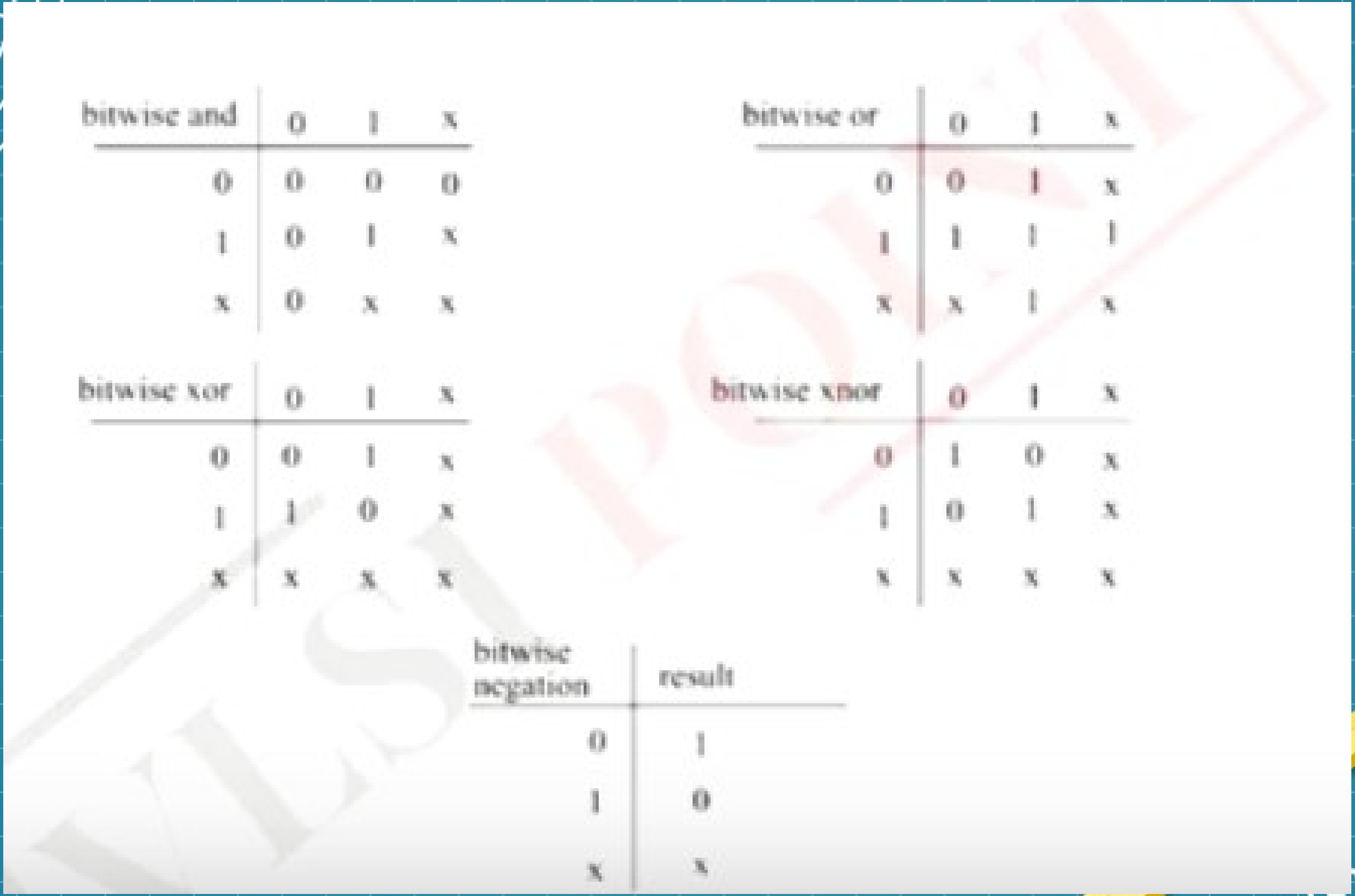
```
A = 2'b0x; B = 2'b10;
A && B // Evaluates to x. Equivalent to (x && logical 1)
```



Bitwise Operators

It performs a bit by bit operation on two operands. It takes each bit in one operand and perform the operation with the corresponding bit in the other operand.

Operator Symbol	Operation Performed	Number of Operands
&	bitwise and	two
|	bitwise or	two
~	bitwise negation	one
^	bitwise xor	two
^~ or ~^	bitwise xnor	two



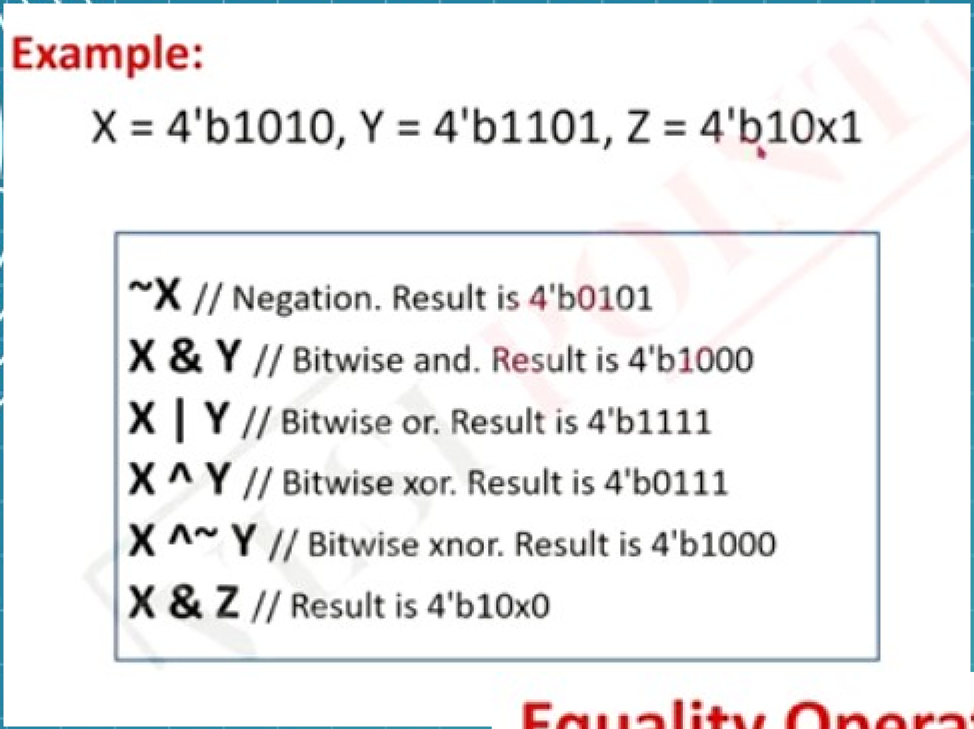
bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

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

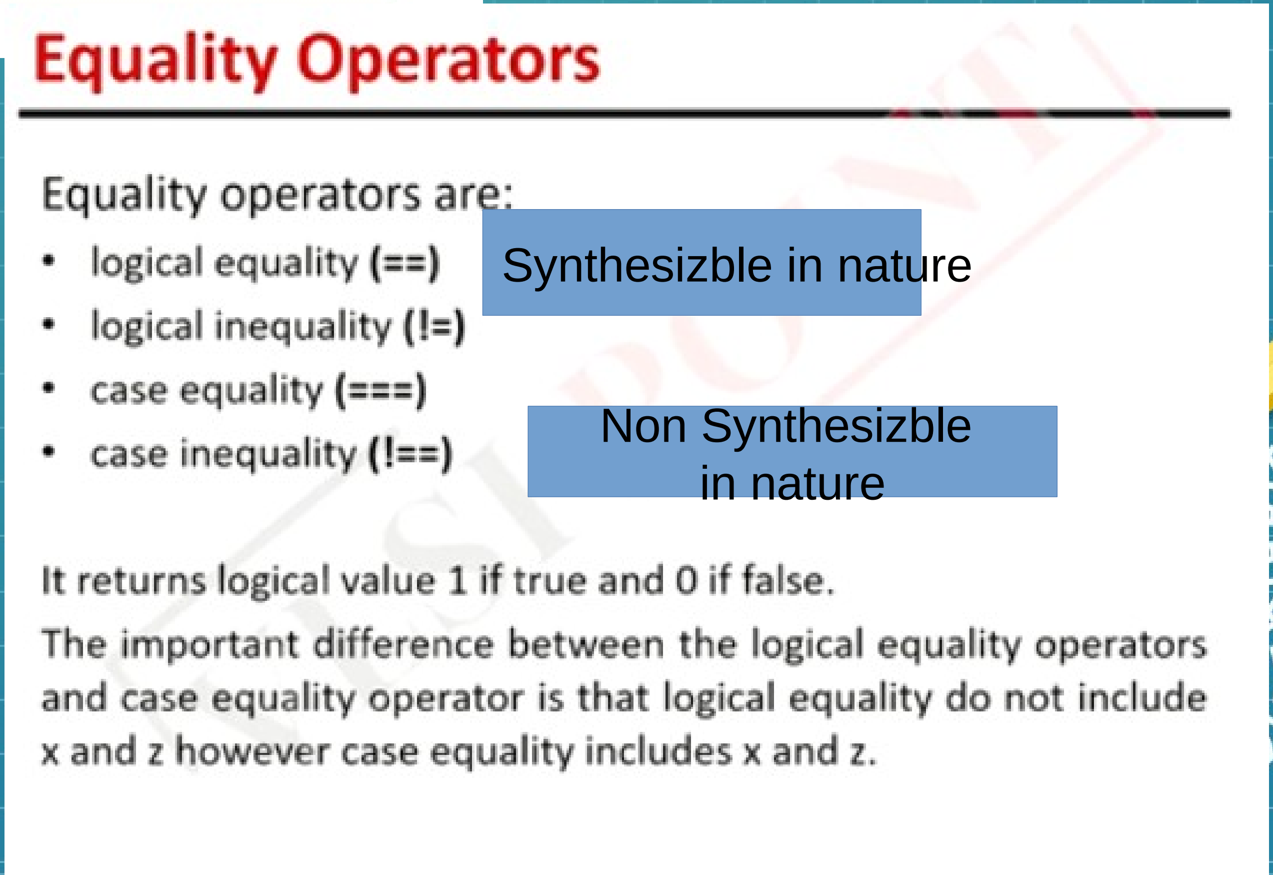
bitwise negation	result
0	1
1	0
x	x



Example:

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

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



Equality Operators

Equality operators are:

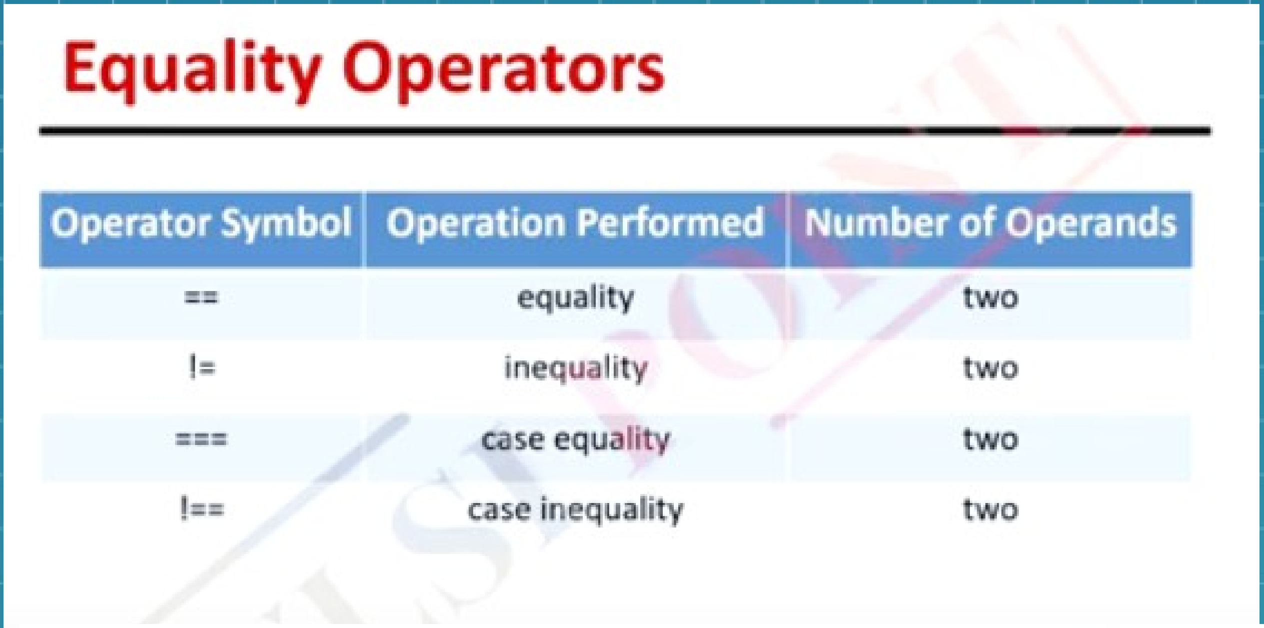
- logical equality (==)
- logical inequality (!=)
- case equality (===)
- case inequality (!==)

Synthesizble in nature

Non Synthesizble in nature

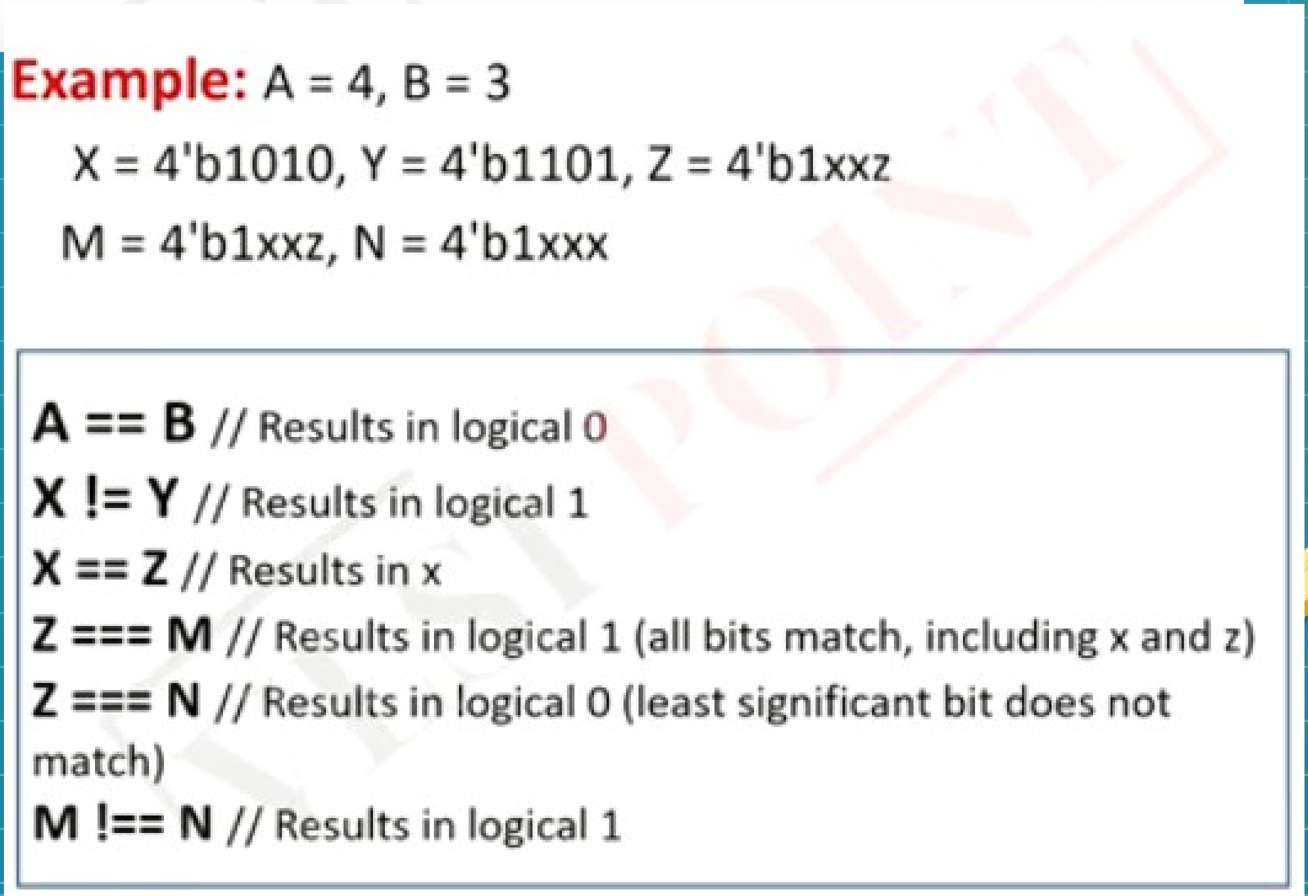
It returns logical value 1 if true and 0 if false.

The important difference between the logical equality operators and case equality operator is that logical equality do not include x and z however case equality includes x and z.



Equality Operators

Operator Symbol	Operation Performed	Number of Operands
==	equality	two
!=	inequality	two
===	case equality	two
!==	case inequality	two

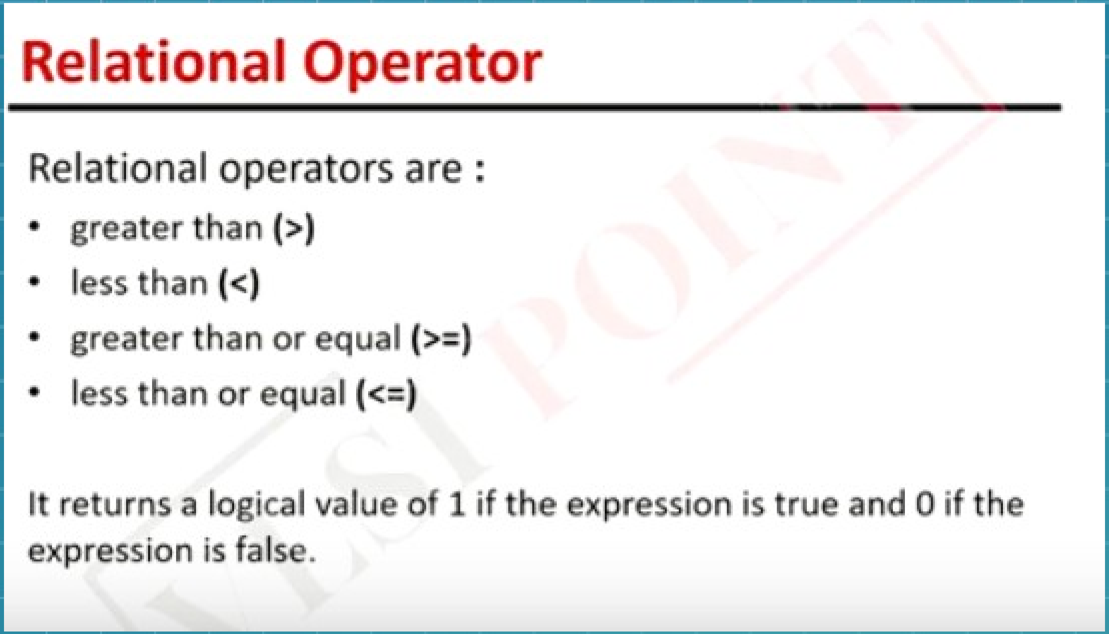


Example: A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxz

M = 4'b1xxz, N = 4'b1xxx

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

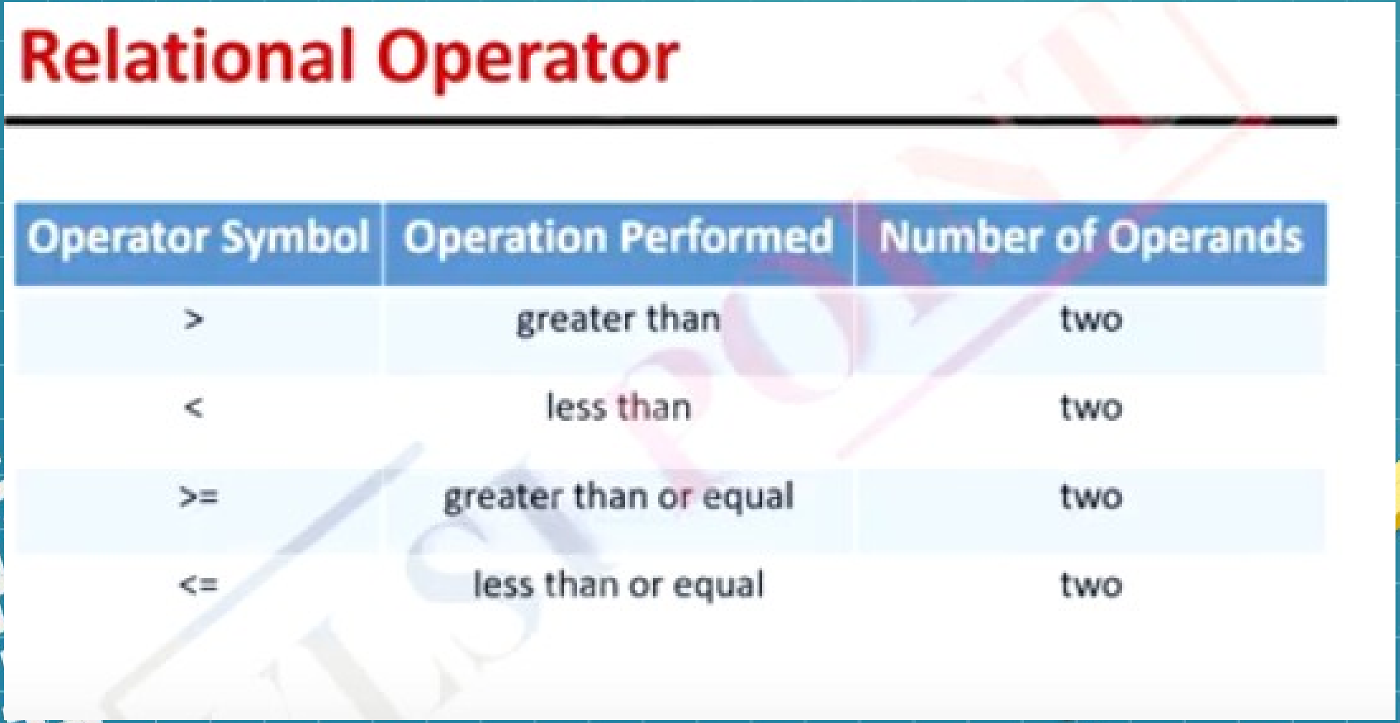


Relational Operator

Relational operators are :

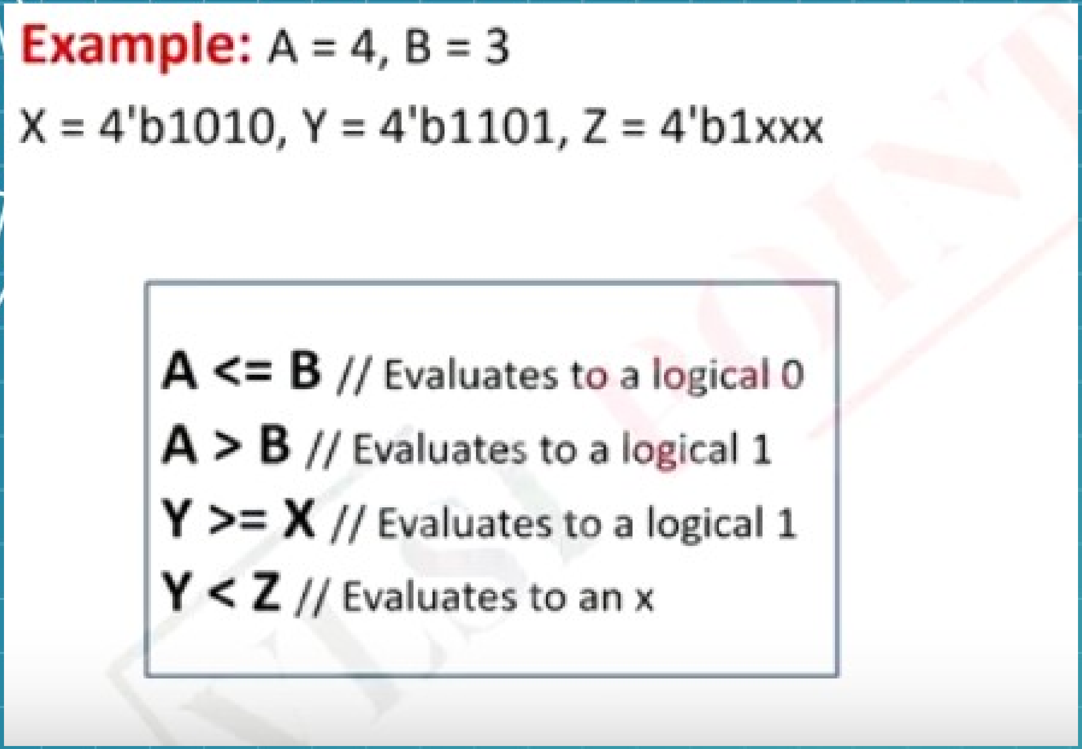
- greater than **(>)**
- less than **(<)**
- greater than or equal **(>=)**
- less than or equal **(<=)**

It returns a logical value of 1 if the expression is true and 0 if the expression is false.



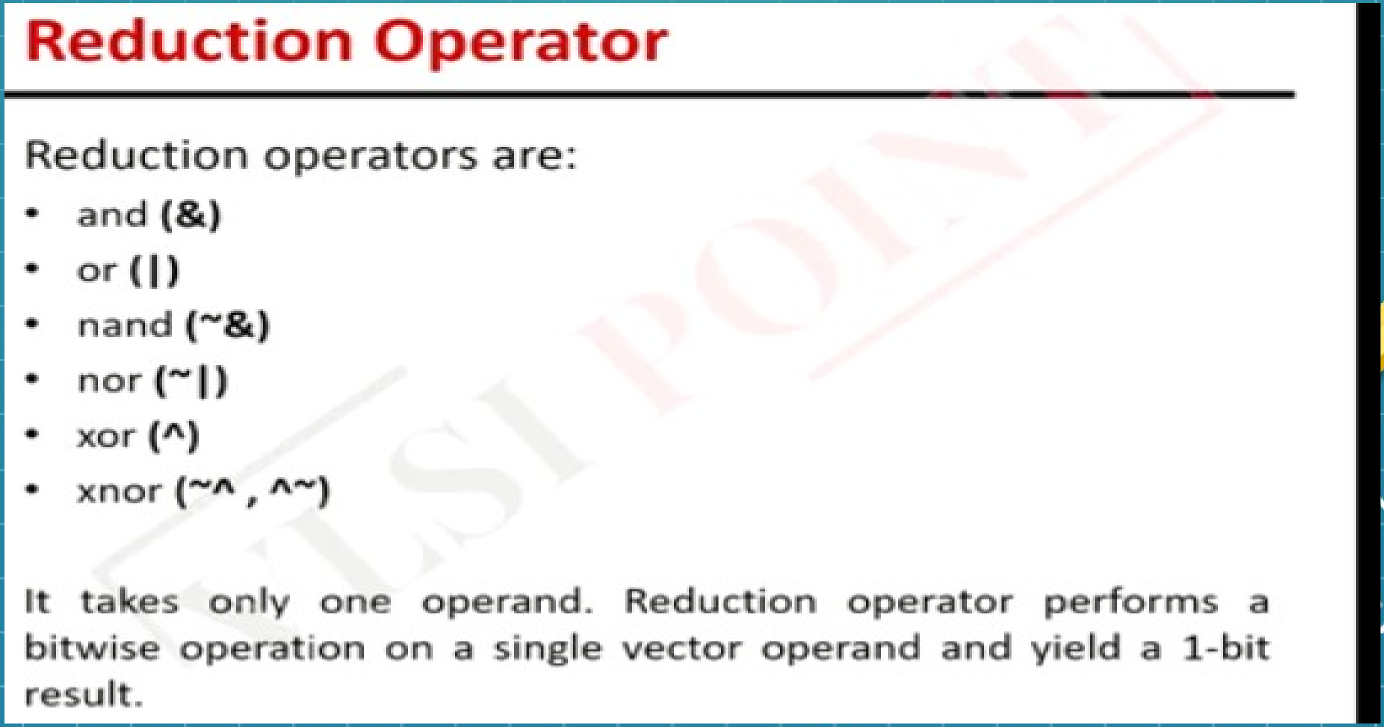
Relational Operator

Operator Symbol	Operation Performed	Number of Operands
>	greater than	two
<	less than	two
>=	greater than or equal	two
<=	less than or equal	two


Example: A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

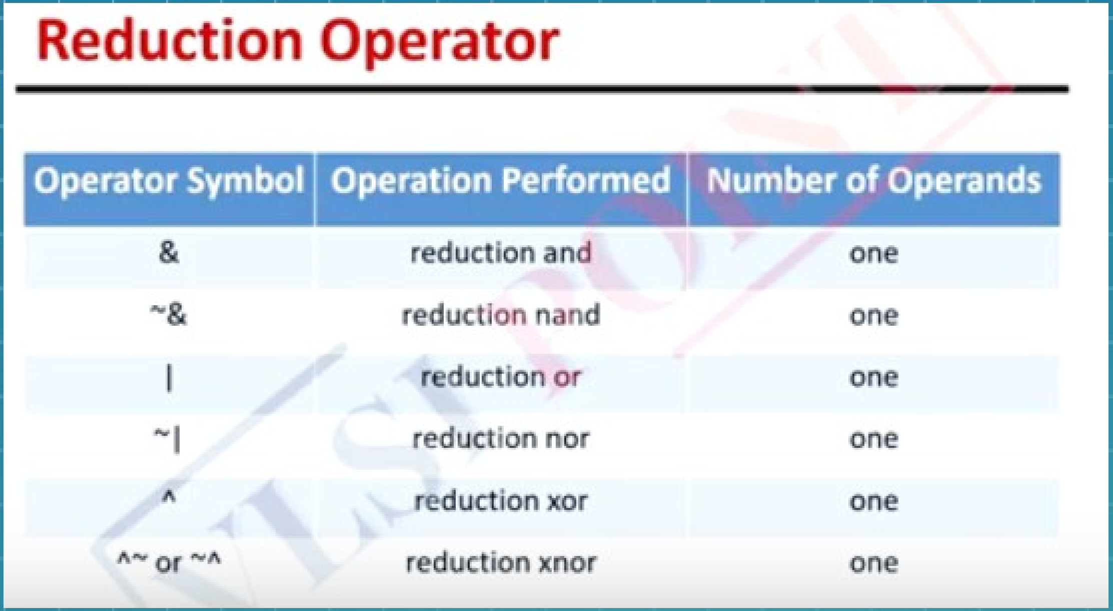
```
A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```


Reduction Operator

Reduction operators are:

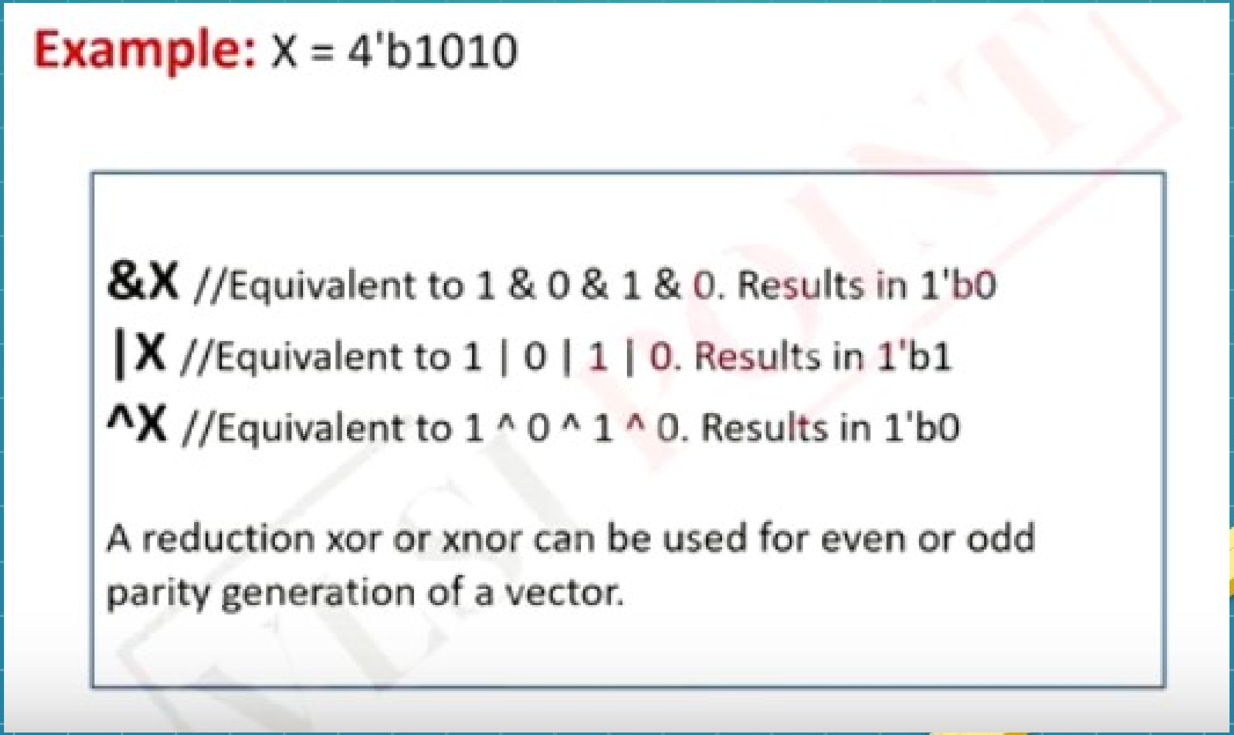
- and **(&)**
- or **(|)**
- nand **(~&)**
- nor **(~|)**
- xor **(^)**
- xnor **(~^ , ^~)**

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



Reduction Operator

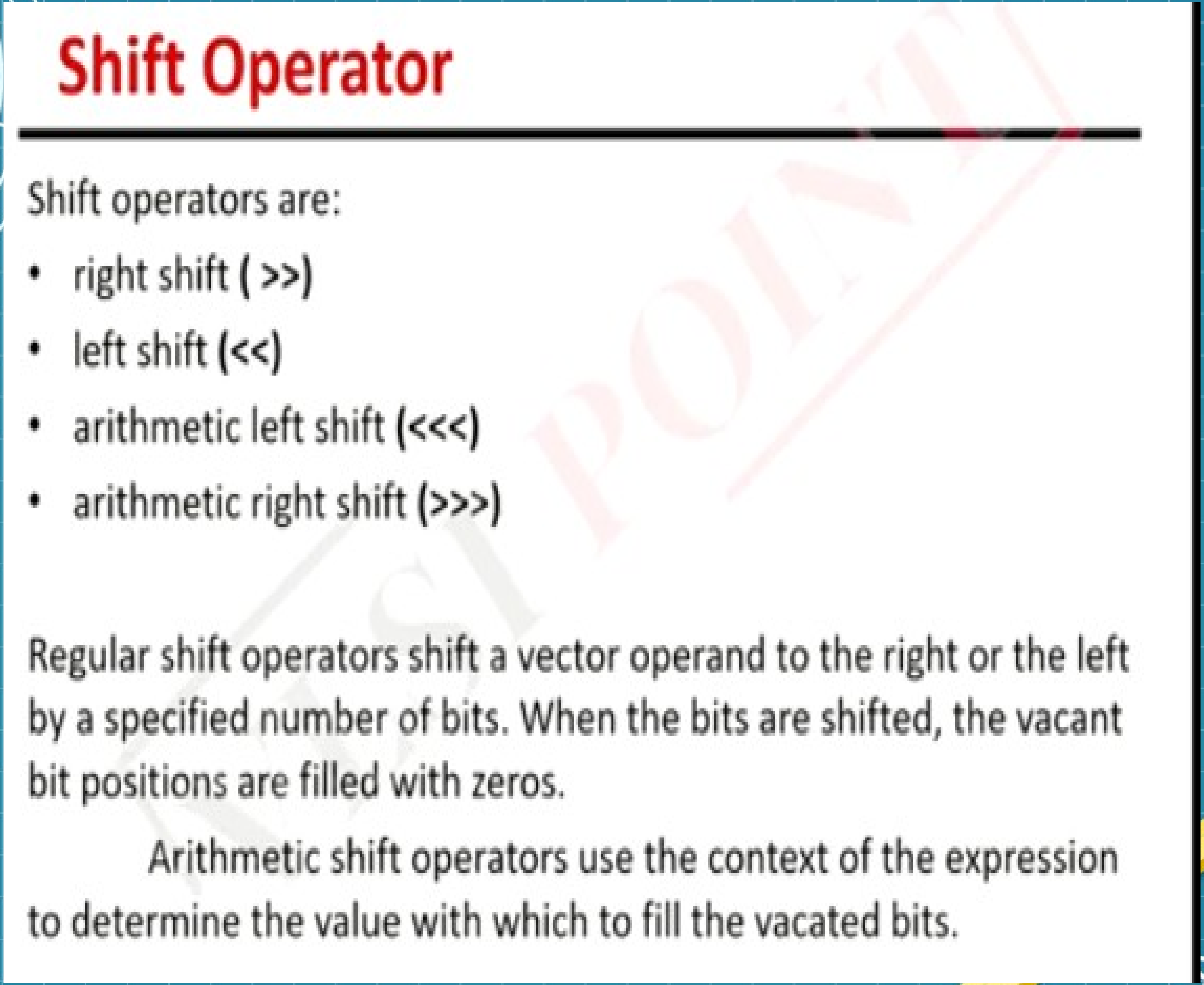
Operator Symbol	Operation Performed	Number of Operands
&	reduction and	one
~&	reduction nand	one
|	reduction or	one
~|	reduction nor	one
^	reduction xor	one
^~ or ~^	reduction xnor	one



Example: 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.



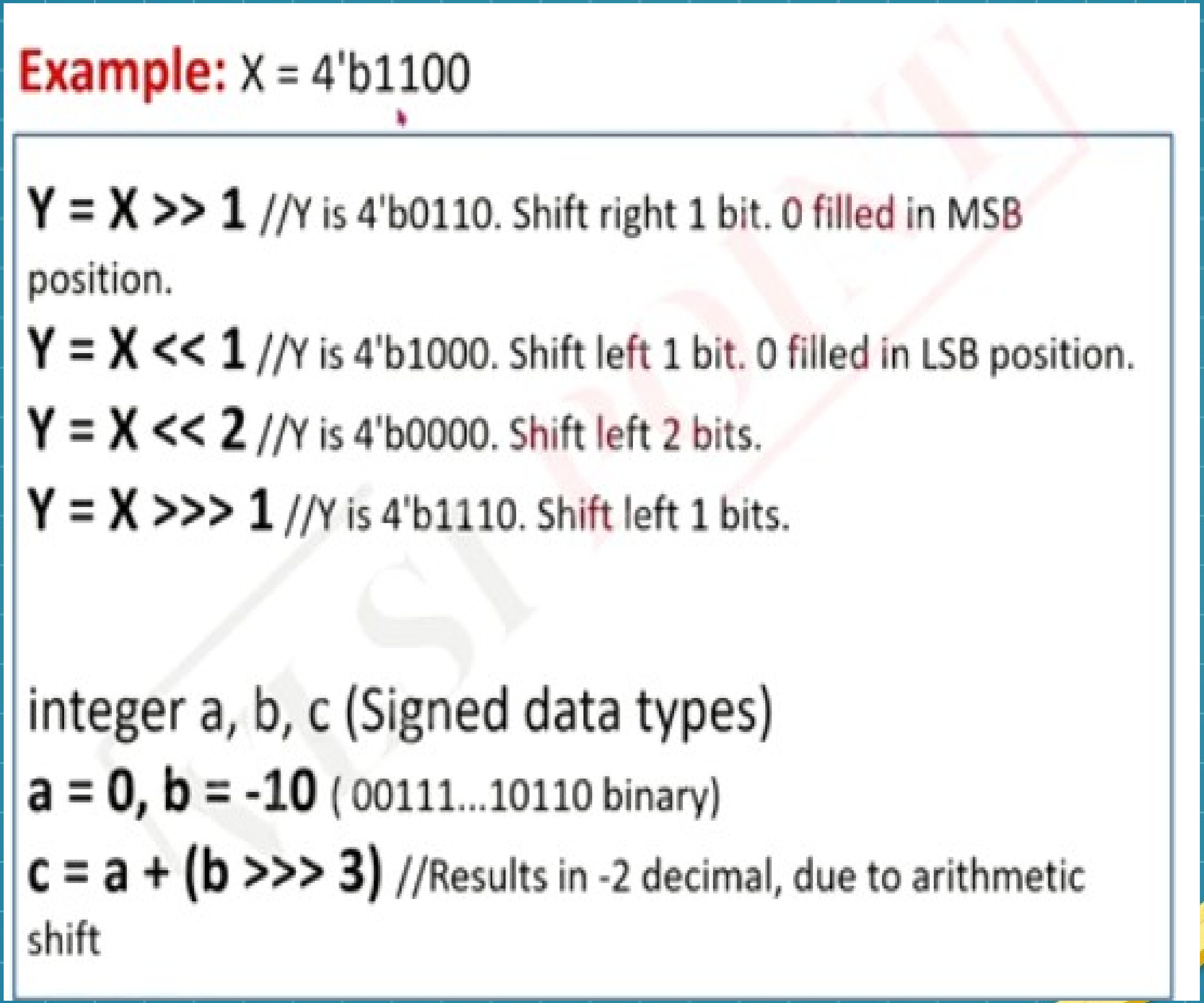
Shift Operator

Shift operators are:

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

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

Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.



Example: X = 4'b1100

Y = X >> 1 //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.

Y = X << 1 //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.

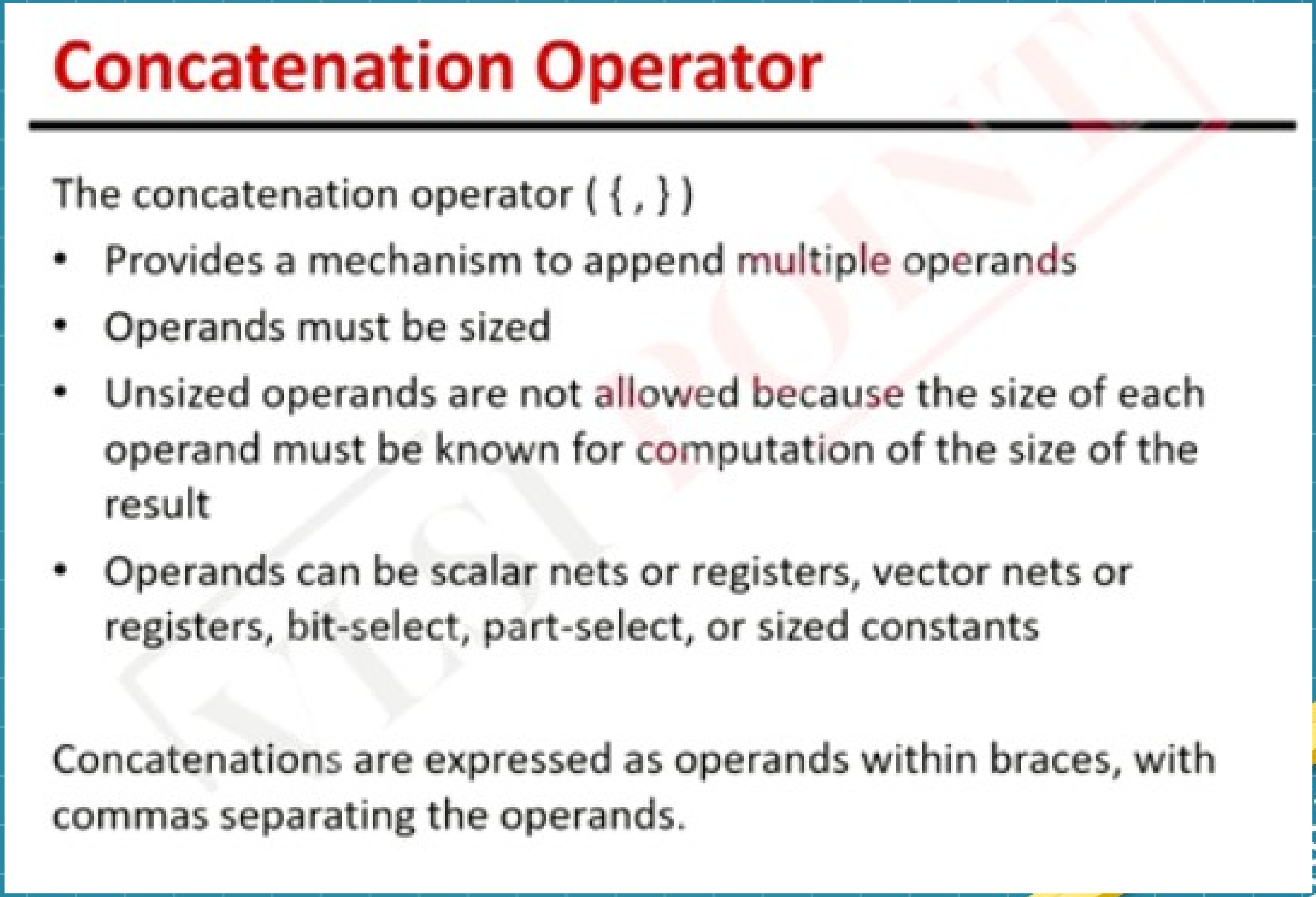
Y = X << 2 //Y is 4'b0000. Shift left 2 bits.

Y = X >>> 1 //Y is 4'b1110. Shift left 1 bits.

integer a, b, c (Signed data types)

a = 0, b = -10 (00111...10110 binary)

c = a + (b >>> 3) //Results in -2 decimal, due to arithmetic shift

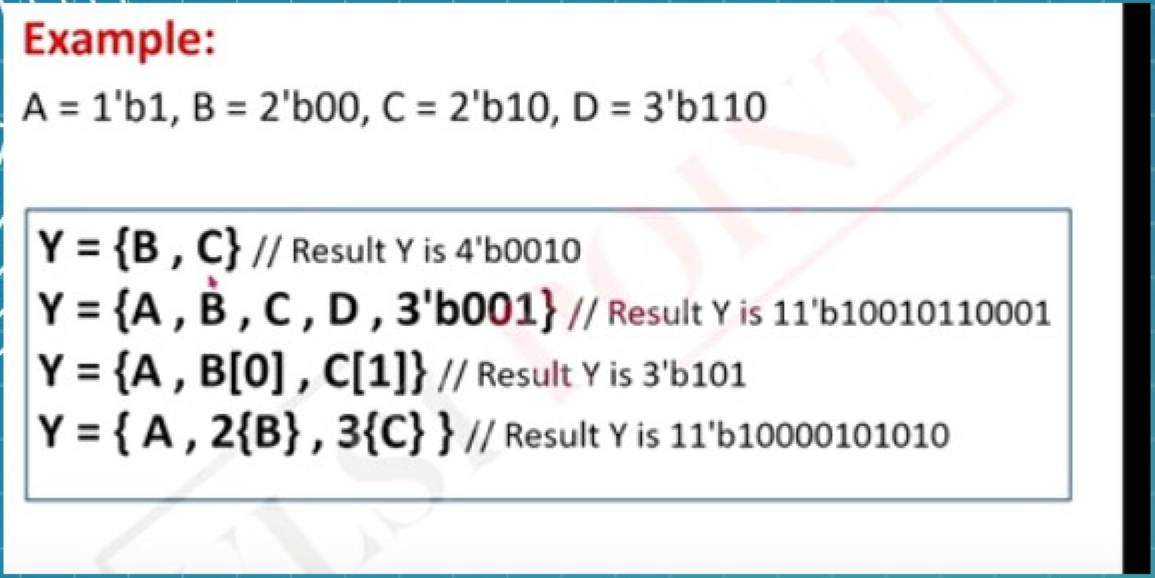


Concatenation Operator

The concatenation operator ({ , })

- Provides a mechanism to append multiple operands
- Operands must be sized
- Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result
- Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants

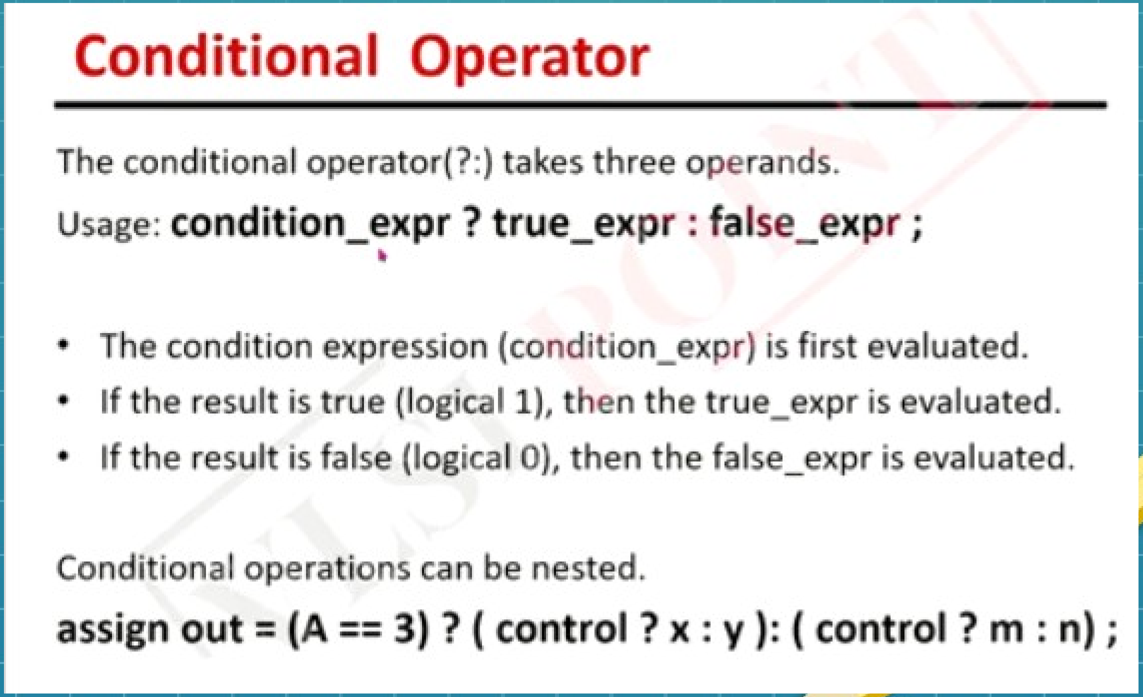
Concatenations are expressed as operands within braces, with commas separating the operands.



Example:

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

```
Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0] , C[1]} // Result Y is 3'b101
Y = { A , 2{B} , 3{C} } // Result Y is 11'b10000101010
```



Conditional Operator

The conditional operator(?:) takes three operands.

Usage: **condition_expr ? true_expr : false_expr ;**

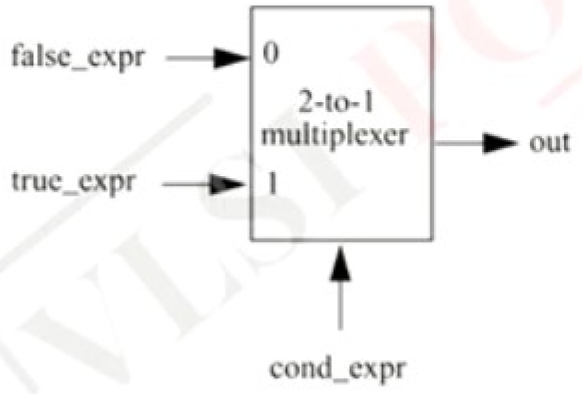
- The condition expression (condition_expr) is first evaluated.
- If the result is true (logical 1), then the true_expr is evaluated.
- If the result is false (logical 0), then the false_expr is evaluated.

Conditional operations can be nested.

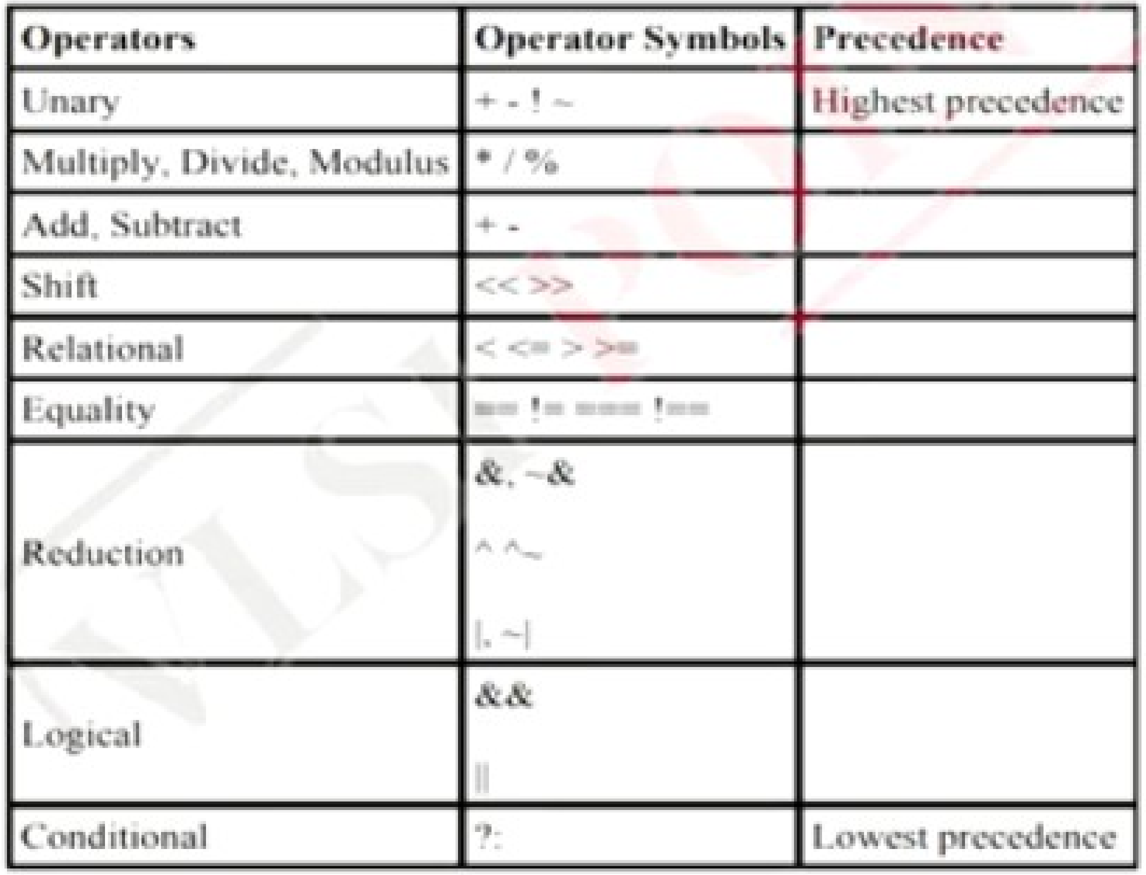
```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Conditional Operator

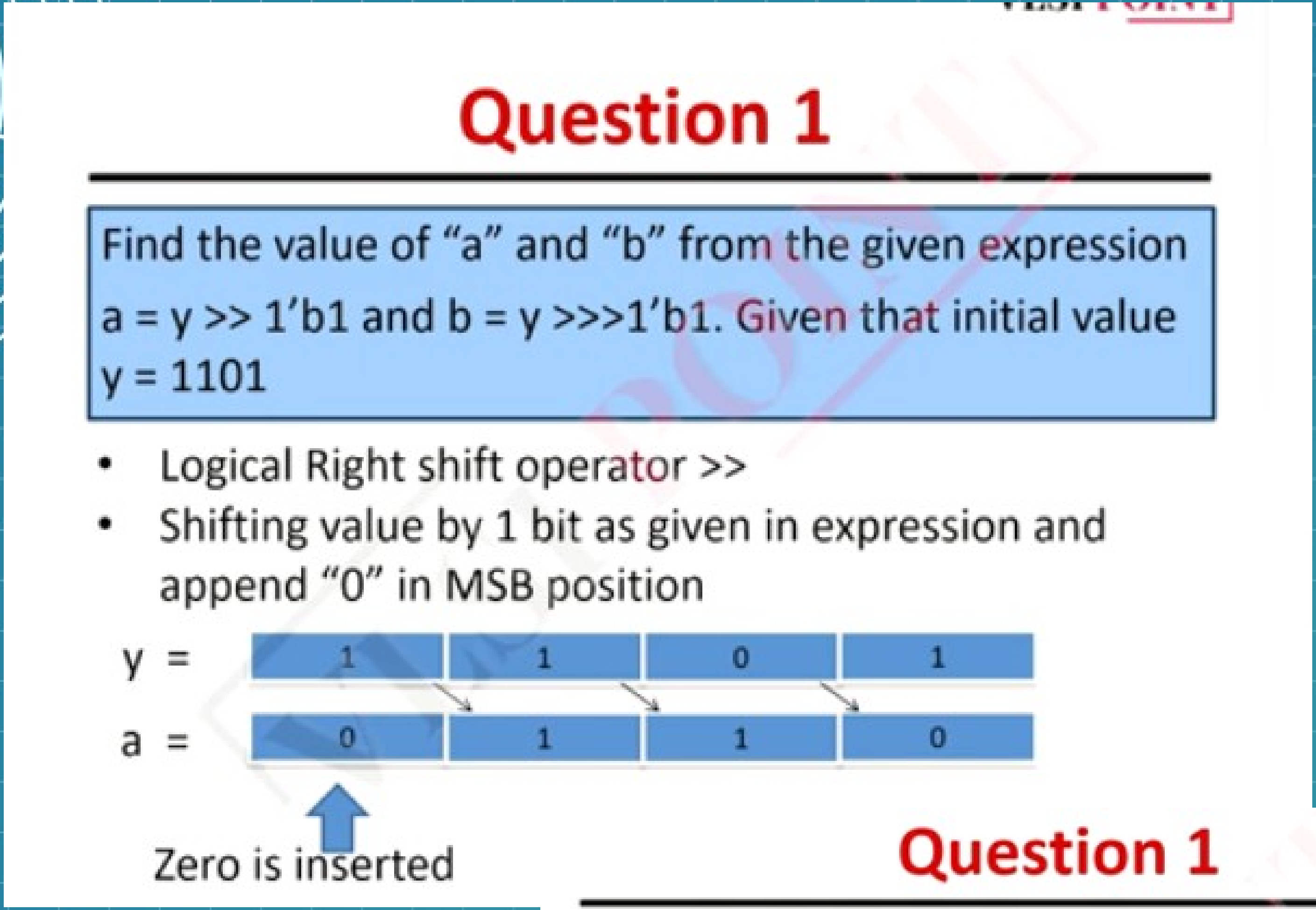
The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Operator Precedence



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



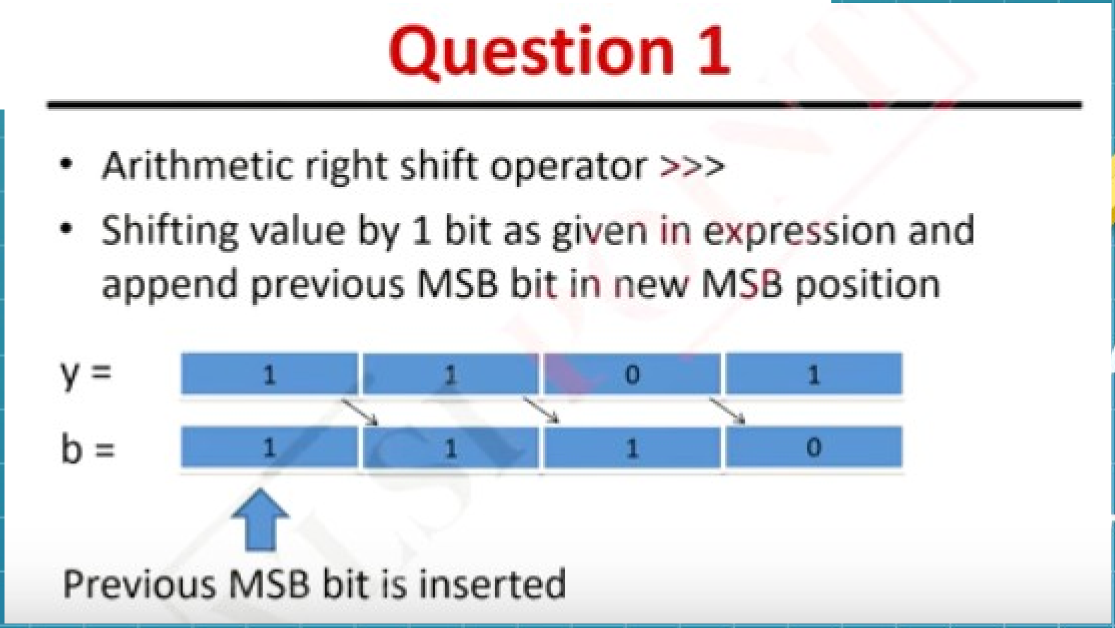
Question 1

Find the value of "a" and "b" from the given expression
a = y >> 1'b1 and b = y >>>1'b1. Given that initial value
y = 1101

- Logical Right shift operator >>
- Shifting value by 1 bit as given in expression and append "0" in MSB position

y =	1	1	0	1
a =	0	1	1	0

Zero is inserted



Question 1

- Arithmetic right shift operator >>>
- Shifting value by 1 bit as given in expression and append previous MSB bit in new MSB position

y =	1	1	0	1
b =	1	1	1	0

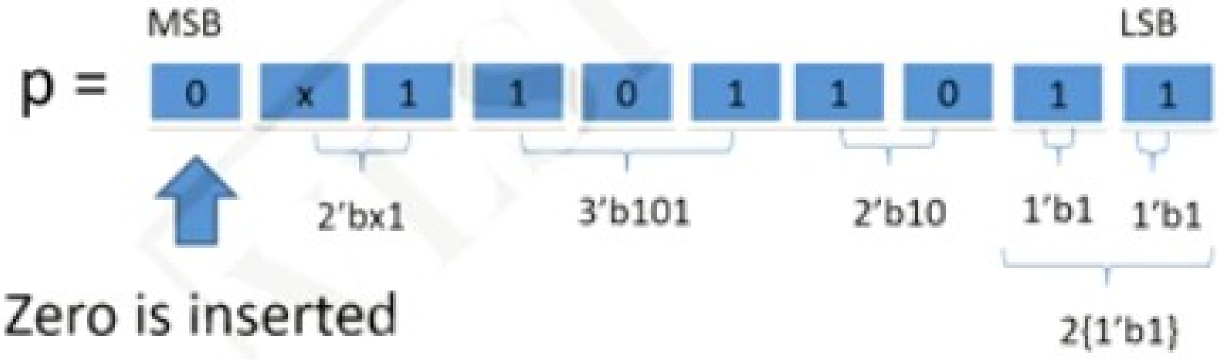
Previous MSB bit is inserted

Question 2

If “p” is a vector of length 10 bit then, find the value of p for the given expression

p = {2’bx1,3’b101,2’b10,2{1’b1}}

- concatenation operator {}



Zero is inserted