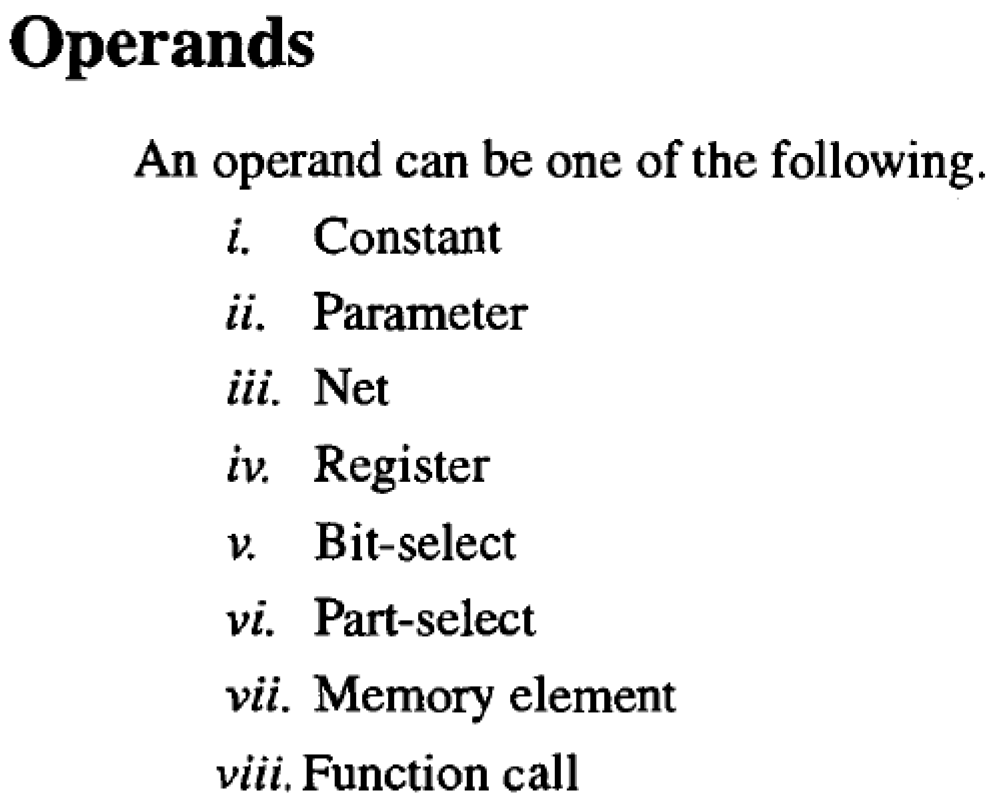
OPERATORS

Operands and Operators

- An expression is formed using operands and operators



Operands

An operand can be one of the following.

i. Constant

ii. Parameter

iii. Net

iv. Register

v. Bit-select

vi. Part-select

vii. Memory element

viii. Function call

Operators

Operators in Verilog HDL are classified into the following categories.

i. Arithmetic operators

ii. Relational operators

iii. Equality operators

iv. Logical operators

v. Bitwise operators

vi. Reduction operators

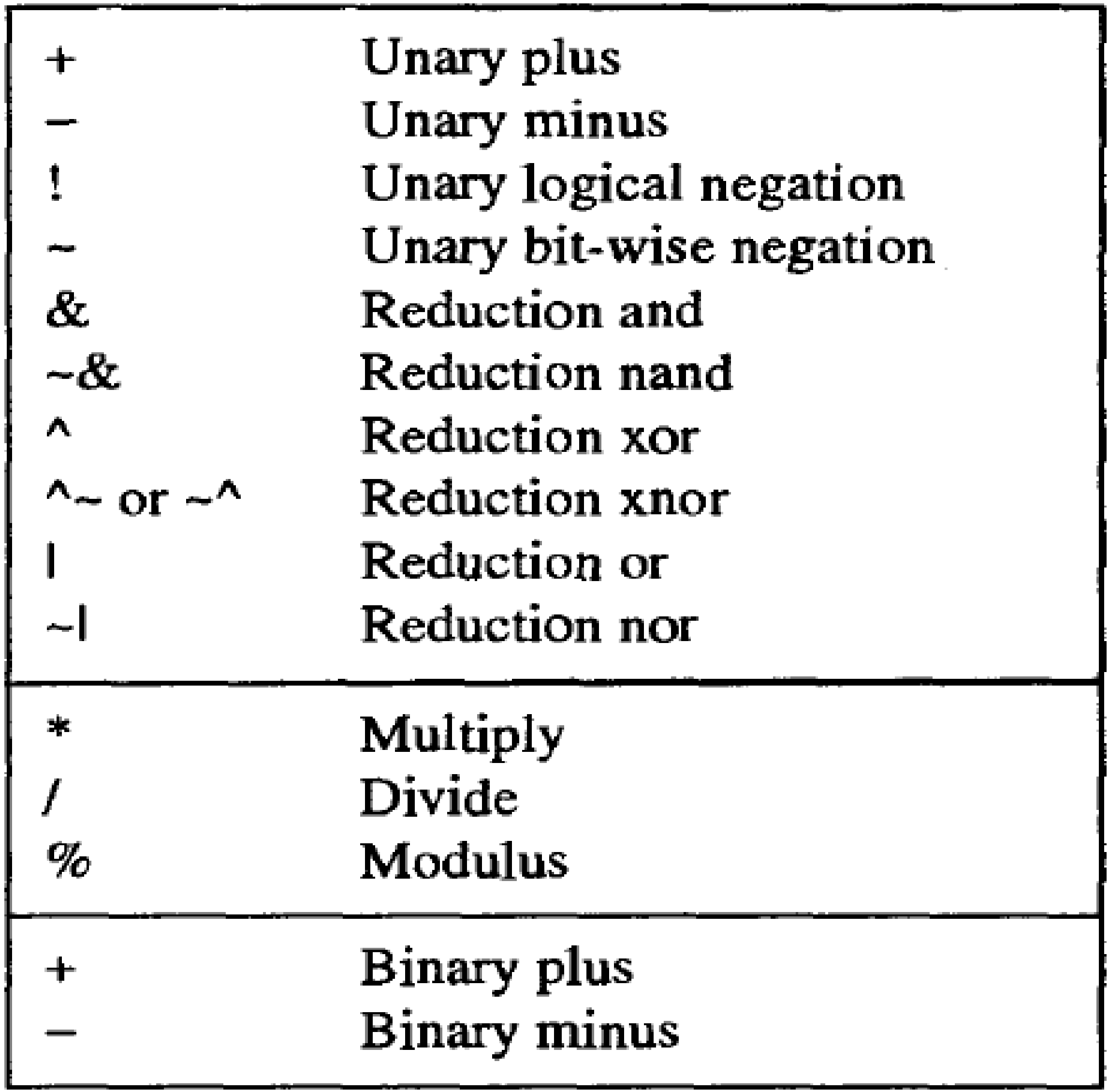
vii. Shift operators

viii. Conditional operators

ix. Concatenation and replication operators

Table showing precedence and names of all the operators

- The operators are listed from highest precedence(top row) to the lowest precedence (bottom row).
- Operators in the same row have identical precedence.



+	Unary plus
−	Unary minus
!	Unary logical negation
~	Unary bit-wise negation
&	Reduction and
~&	Reduction nand
^	Reduction xor
^~ or ~^	Reduction xnor
|	Reduction or
~|	Reduction nor
*	Multiply
/	Divide
%	Modulus
+	Binary plus
−	Binary minus

- All operators associate left to right except for the conditional operator that associates right to left.
- The expression:

A + B- C

is evaluated as:

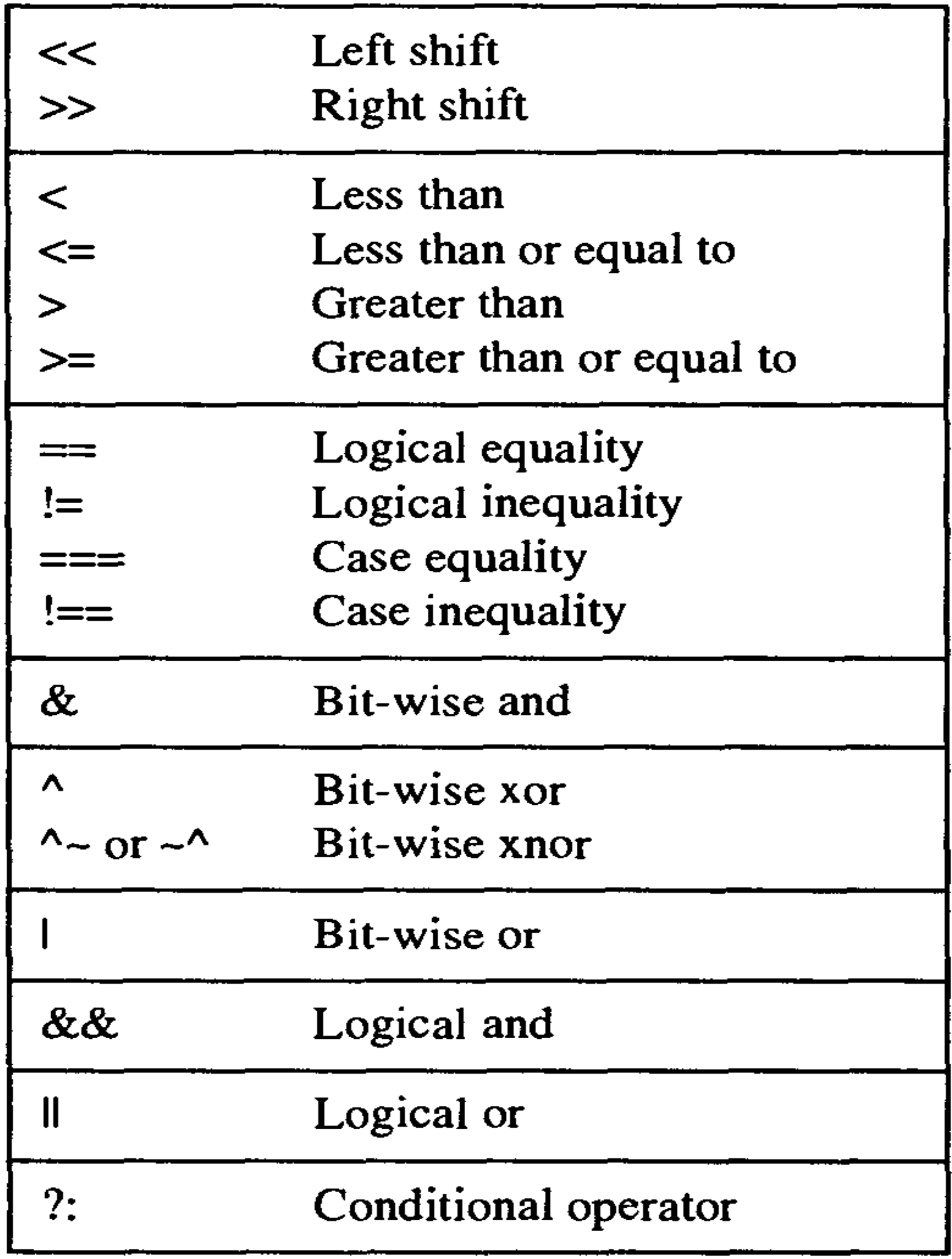
(A + B) - C *(Left to right)*

- while the expression:

A ? B : C ? D : F

is evaluated as:

- A ? B : (C ? D : F) *(Right to left)*



<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality
&	Bit-wise and
^	Bit-wise xor
^~ or ~^	Bit-wise xnor
|	Bit-wise or
&&	Logical and
||	Logical or
?:	Conditional operator

1. Arithmetic operator

The arithmetic operators are:

- + (unary and binary plus)
- – (unary and binary minus)
- * (multiply)
- / (divide)
- % (modulus)

```
//suppose that: a = 4'b0011;
//              b = 4'b0100;
//              d = 6; e = 4; f = 2;
//then,
a + b  //add a and b; evaluates to  4'b0111
b - a  //subtract a from b; evaluates to  4'b0001
a * b  //multiply a and b; evaluates to  4'b1100
d / e  //divide d by e, evaluates to  4'b0001. Truncates fractional part
e ** f //raises e to the power f, evaluates to 4'b1111
       //power operator is most likely not synthesible
```

If any operand bit has a value "x", the result of the expression is all "x"
If an operand is not fully known the result cannot be either.

```
3  %  2;  //evaluates to  1
16 %  4;  //evaluates to  0
-7 %  2;  //evaluates to  -1, takes sign of first operand
 7 % -2;  //evaluates to  1, takes sign of first operand
```

Integer division truncates any fractional part. For example,

```
    7 / 4            is 1
```

The % (modulus) operator gives the remainder with the sign of the first operand.

```
    7 % 4            is 3
```

while:

```
    -7 % 4           is -3
```

If any bit of an operand in an arithmetic operation is an **x** or a **z**, the entire result is an **x**. For example,

```
    'b10x1 + 'b01111   is  'bxxxxx
```

- Unary operators
  - Operators "+" and "-" can act as unary operators
  - They indicate the sign of an operand

```
i.e., -4  // negative four
      +5  // positive five
```

!!! Negative numbers are represented as 2's compliment numbers !!!

!!! Use negative numbers only as type integer or real !!!

!!! Avoid the use of <sss>'<base><number >in expressions !!!

!!! These are converted to unsigned 2's compliment numbers !!!

!!! This yields unexpected results in simulation and synthesis !!!

2. Relational Operators

The relational operators are:

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

- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.

```
//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101, z = 4'b1xxx
a <= b //evaluates to logical zero
a >  b //evaluates to logical one
y >= x //evaluates to logical 1
y <  z //evaluates to x
```

!!! Note: These are expensive and slow operators at gate level !!!

Example:

```
23 > 45
```

is false (value 0), while:

```
52 < 8'hxFF
```

is x. If operands are not of the same size, the smaller operand is zero-filled on the most significant bit side (the left). For example,

```
'b1000 >= 'b01110
```

is equivalent to:

```
'b01000 >= 'b01110
```

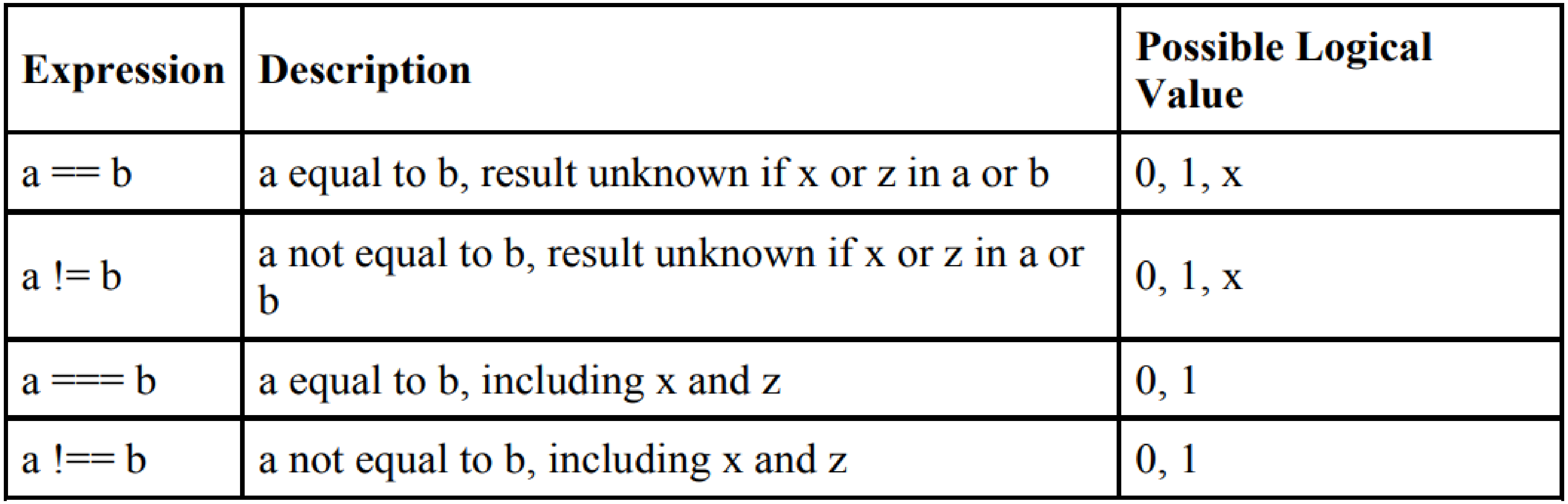
which is false (value 0).

3. Equality Operators

The equality operators are:

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

- The result is 0 if the comparison is false, else the result is a 1.
- In **case comparisons**, values x and z are compared strictly as values, that is, with no interpretations, and the result can never be an unknown,
- While in **logical comparisons**, values x and z have their usual meaning and the result may be unknown; that is, for logical comparisons if either operand contains an x or a z, the result is the unknown value (x).



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

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

```
//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101,
//z = 4'b1xxz, m = 4'b1xxz, n = 4'b1xxx

a   ==  b  //evaluates to logical 0
x   !=  y  //evaluates to logical 1
x   ==  z  //evaluates to x
z  ===  m  //evaluates to logical 1
z  ===  n  //evaluates to logical 0
m !==   n  //evaluates to logical 1
```

Here is an example. Given:

```
Data = 'b11x0;
Addr = 'b11x0;
```

then:

```
Data == Addr
```

is unknown, that is, the value x, and:

```
Data === Addr
```

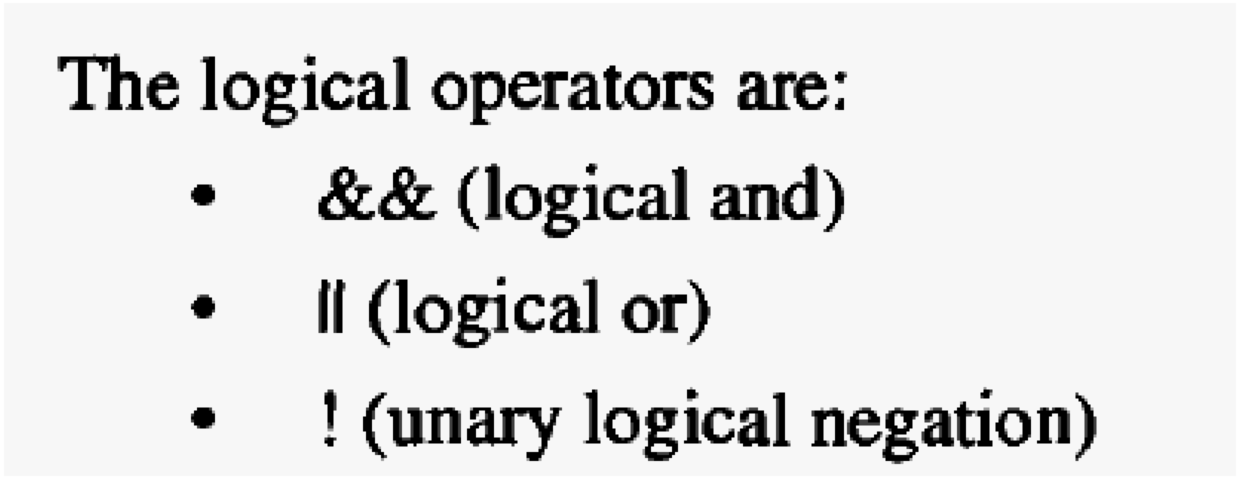
is true, that is, the value 1.

If the operands are of unequal lengths, the smaller operand is zero-filled on the most significant side, that is, on the left. For example,

```
2'b10 == 4'b0010
```

is same as: 4'b0010 == 4'b0010, which is true (the value 1)

4. Logical Operators



The logical operators are:

- && (logical and)
- || (logical or)
- ! (unary logical negation)

- These operators operate on logical values 0 or 1.
- Logical operators evaluate to a 1 bit value - 0 (false), 1 (true), or x (ambiguous)
- Operands not equal to zero are equivalent to one
- Logical operators take variables or expressions as operators

If a bit in any of the operands is an x, the result is also an x.

```
    !x      is  x
```

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

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

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

If

```
Crd = 'b0;    // 0 is false.
Dgs = 'b1;    // 1 is true.
```

then:

```
Crd && Dgs        is 0 (false)
Crd || Dgs        is 1 (true)
! Dgs             is 0 (false)
```

For vector operands, a non-zero vector is treated as a 1.

For example,

```
A_Bus = 'b0110;
B_Bus = 'b0100;
```

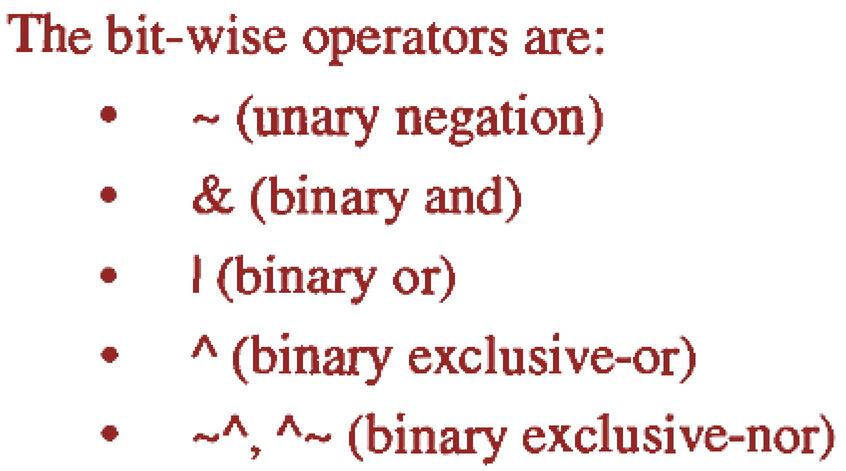
then:

```
A_Bus || B_Bus        is 1
A_Bus && B_Bus        is also 1
```

and:

`! A_Bus` `is same as` `! B_Bus` , which is 0.

5. Bit-wise Operators



The bit-wise operators are:

- ~ (unary negation)
- & (binary and)
- | (binary or)
- ^ (binary exclusive-or)
- ~^, ^~ (binary exclusive-nor)

- These operators operate bit-by-bit, on corresponding bits of the input operands and produce a vector result.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.
- A z is treated as an x in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Tables to show the result of the bit-by-bit operation for the various operators

& (and)	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

| (or)	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

^ (xor)	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

^~ (xnor)	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

~ (negation)	0	1	x	z
	1	0	x	x

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

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

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

```
// 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
```

6. Reduction Operator

The reduction operators operate on all bits of a single operand and produce a 1-bit result. The operators are:

- & (reduction and):
  If any bit is 0, the result is 0, else if any bit is an **x** or a **z**, the result is an **x**, else the result is a 1.
- ~& (reduction nand):
  Invert of & reduction operator.
- | (reduction or):
  If any bit is a 1, the result is 1, else if any bit is an **x** or a **z**, the result is an **x**, else the result is 0.
- ~| (reduction nor):
  Invert of | reduction operator.
- ^ (reduction xor):
  If any bit is an **x** or a **z**, the result is an **x**, else if there are even number of 1's in the operand, the result is 0, else the result is 1.
- ~^ (reduction xnor):
  Invert of ^ reduction operator.

- Reduction operators take only one operand.
- Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- The logic tables for the operators are the same as, Bitwise Operators. The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand.
- Reduction operators work bit by bit from right to left.
- Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively
- The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially. The difference lies in the number of operands each operator takes and also the value of results computed.

```
//let 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

- Note that the logical equality (==) operator cannot be used for comparison; the logical equality operator comparison will only yield the result x. The case equality operator yields the value 1 which is the desired result

Here are some examples. Given,

```
A = 'b0110;
B = 'b0100;
```

then:

```
| B          is 1
& B          is 0
~^ A         is 1
```

The reduction xor operator can be used to determine if any bit of a vector is an x. Given,

```
MyReg = 4'b01x0;
```

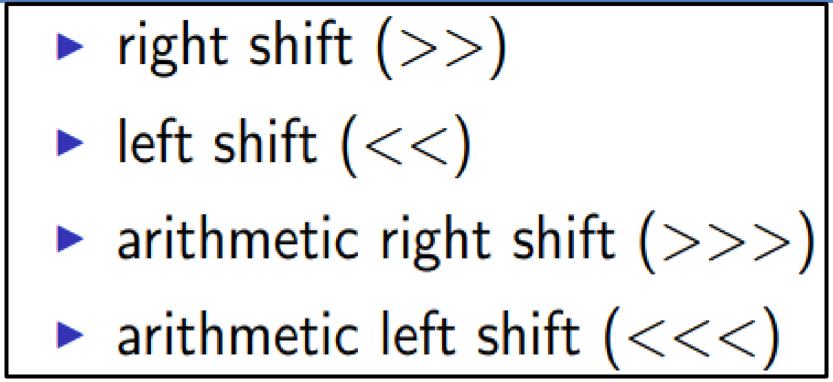
then:

```
^MyReg is an x
```

This can be checked using an if statement such as:

```
if (^MyReg === 1'bx)
  $display ("There is an unknown in the vector MyReg!");
```

7. Shift Operator



- right shift (>>)
- left shift (<<)
- arithmetic right shift (>>>)
- arithmetic left 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.
- ***Left Arithmetic Shift*** -moves each bit to the left. The vacant LSB is filled with zero and MSB is discarded. It is identical to Left Logical Shift.
- ***Right Arithmetic Shift*** -moves each bit to the right. The LSB is discarded and the vacant MSB is filled with the value of the previous (now shifted one position to the right) MSB.

```
// X = 4'b1100

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

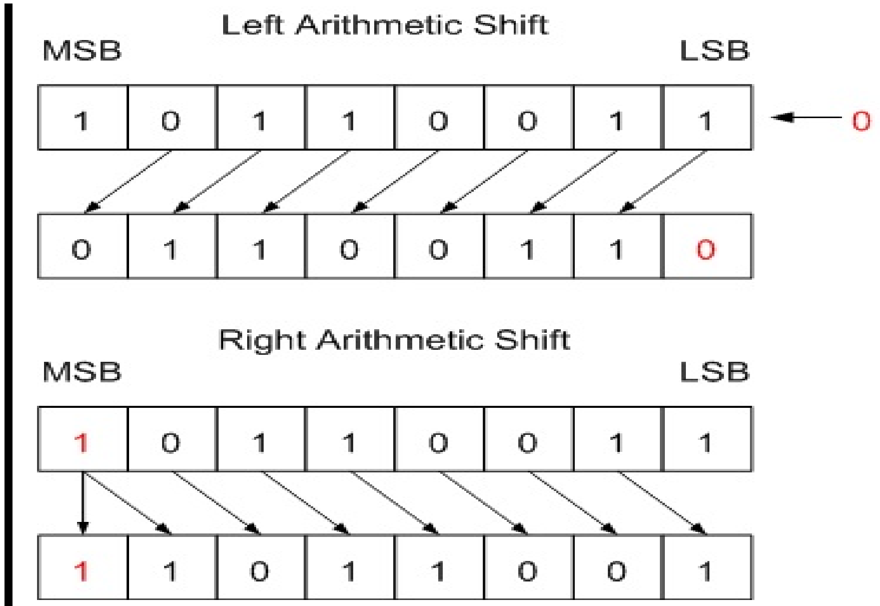
```
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.

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
```

if

```
reg [0:7] Qreg;
. . .
Qreg = 4'b0111;
```

then:

```
 Qreg >> 2          is  8'b0000_0001
```



- Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication
- Verilog HDL has no exponentiation operator. However, the shift operator can be used to support this partly
- For example, if we are interested in computing $2^{NumBits}$, this can be achieved by using the shift operator, such as:

```
32'b1 << NumBits            // NumBits must be less than 32
```

In a similar vein, a 2-to-4 decoder can be modeled using a shift operator.

```
wire [0:3] DecodeOut = 4'd1 << Address[0:1];
```

A	B	Y0	y1	y2	y3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Address[0:1]can have values 0, 1,2, and 3. Correspondingly, DecodeOut has the values 4'b0001,4'b0010,4'b0100, and 4'b1000, thereby modeling a decoder.

8. Conditional Operator

- The conditional operator selects an expression based on the value of the condition expression and it is of the form:

  cond_expr ? expr 1 : expr2

- If cond_expr is true (that is, has value 1), expr1 is selected, if cond_expr is false (value 0), expr2 is selected.
- If cond_expr is an x or a z, both true and false expressions are evaluated and their results compared bit by bit (bitwise operation) with the logic: **0 with 0 gives 0, 1 with 1 gives 1, rest are x**

Here is an example.

```
wire [0:2] Student = Marks > 18 ? Grade_A : Grade_C;
```

The expression *Marks* > 18 is computed; if true, *Grade_A* is assigned to *Student*, if *Marks* is <= 18, *Grade_C* is assigned to *Student*.

Here is another example.

```
always
  #5 Ctr = (Ctr != 25) ? (Ctr + 1) : 5;
```

The expression in the procedural assignment says that if *Ctr* is not equal to 25, increment *Ctr*, else if *Ctr* becomes 25, reset it to 5.

2:1 MUX

```
module mux ( m_out, A, B, select);
   output  m_out;
   input   A, B, select;

   assign  m_out = select ? A: B ;
endmodule
```

tri-state buffer

```
//8-bit wide,
//active-low enabled tri-state buffer
module ts_buff8(
  input  [7:0]  d_in,
  input         en_n,
  output [7:0]  d_out
  );
  assign d_out = ~en_n ? d_in : 8'bz;
endmodule
```

9. Concatenation operator

- Concatenation is the operation of joining bits from smaller expressions to form larger expressions. It is of the form:

  { expr1 , expr2 , . . . , exprN }

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

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

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

```
 wire [7:0] Dbus;
 wire [11:0] Abus;
```

```
 assign Dbus[7:4] = {Dbus[0], Dbus[1], Dbus[2], Dbus[3]};
        // Assign lower four bits in reverse order to upper
        // four bits.
 assign Dbus = {Dbus[3:0], Dbus[7:4]};
        // Swap lower and upper four bits.
```

```
{Dbus, 5}      // Unsized constant in concatenation is not
               // allowed.
```

10. Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({ })
- Replication is performed by specifying a repetition number. It is of the form:

```
{ repetition_number { expr1 , expr2 ,. . . , exprN }}
```

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

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

Here are some examples.

```
Abus = {3{4'b1011}};   // The bit-vector 12'b1011_1011_1011
Abus = {{4 {Dbus[7]}}, Dbus}; /* Sign extension */

{3{1'b1}}    is 111
{3{Ack}}     is same as {Ack, Ack, Ack}
```

THANK YOU