Dataflow Modeling

DATAFLOW MODELING - LEARNING OBJECTIVES

- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements
- Define expressions, operators, and operands
- List operator types for all possible operations Arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, and conditional
- Use dataflow constructs to model practical digital circuits in Verilog

DATAFLOW MODELING - INTRODUCTION

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually
- Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design
- However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level
- Dataflow modeling provides a powerful way to implement a design
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates

DATAFLOW MODELING – INTRODUCTION –Cont’d

- With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance
- No longer can companies devote engineering resources to handcrafting entire designs with gates
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description
- This process is called logic synthesis
- Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated
- This approach allows the designer to concentrate on optimizing the circuit in terms of data flow
- For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design
- In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling

CONTINUOUS ASSIGNMENTS

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net
- This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction
- The assignment statement starts with the keyword assign
- The syntax of an assign statement is as follows

CONTINUOUS ASSIGNMENTS - CHARACTERISTICS

- The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
- The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
- Delay values can be specified for assignments in terms of time units.
- Delay values are used to control the time when a net is assigned the evaluated value.
- This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

DELAYS

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side

- Three ways of specifying delays in continuous assignment statements are
  - Regular assignment delay
  - Implicit continuous assignment delay and
  - Net declaration delay

REGULAR ASSIGNMENT DELAY

- The first method is to assign a delay value in a continuous assignment statement.

- The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.

- If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay.

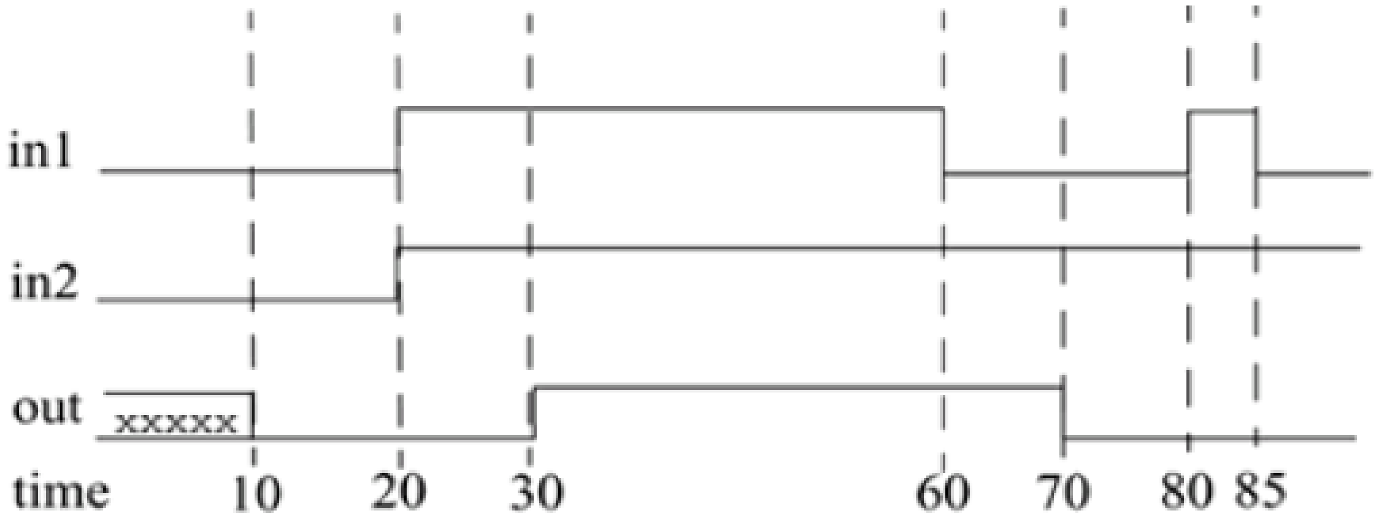
- An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

- assign #10 out = in1 & in2; // Delay in a continuous assign

REGULAR ASSIGNMENT DELAY –Cont’d

- When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- When in1 goes low at 60, out changes to low at 70.
- However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0.
- A pulse of width less than the specified assignment delay is not propagated to the output.

REGULAR ASSIGNMENT DELAY



IMPLICIT CONTINUOUS ASSIGNMENT DELAY

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.
- //implicit continuous assignment delay
- wire #10 out = in1 & in2;
- //same as
- wire out;
- assign #10 out = in1 & in2;
- The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

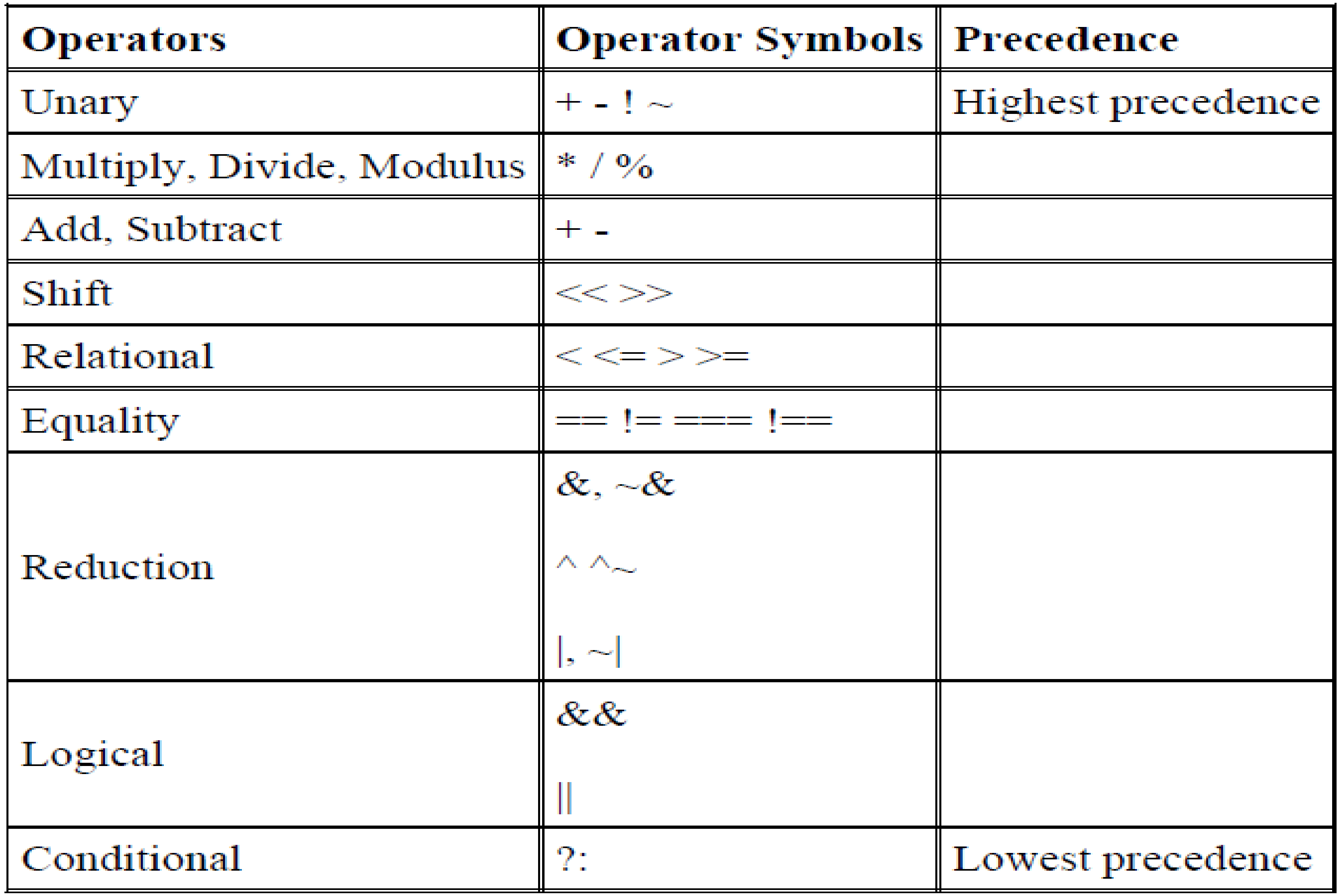
NET DECLARATION DELAY

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.

- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.

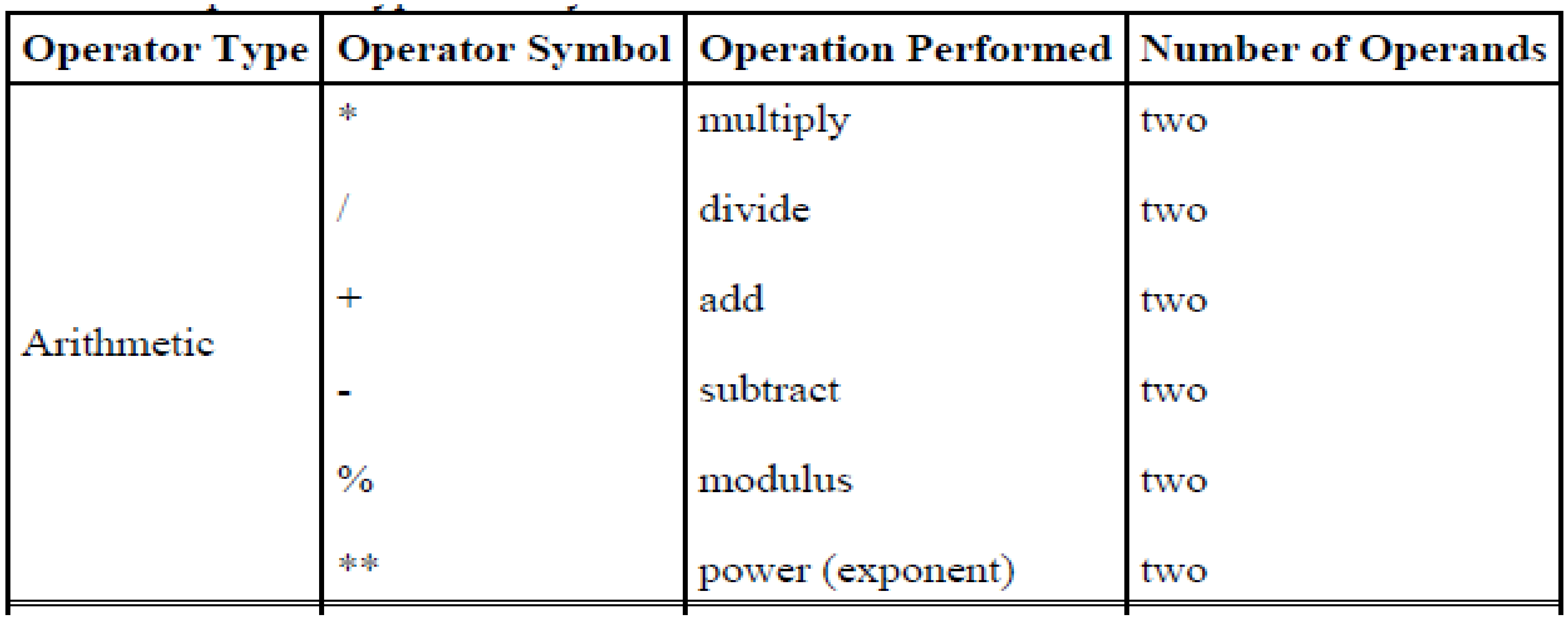
- Net declaration delays can also be used in gate-level modeling.
- //Net Delays
- wire # 10 out;
- assign out = in1 & in2;
- //The above statement has the same effect as the following.
- wire out;
- assign #10 out = in1 & in2;

Operator Precedence



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

OPERATOR TYPES



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two

Arithmetic Operators

There are two types of arithmetic operators: binary and unary.

Binary operators

Binary arithmetic operators are
multiply (*),
divide (/),
add (+),
subtract (-),
power (**),
and modulus (%).

Binary operators take two operands.

NOTE :

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

Arithmetic Operators

Unary operators

The operators + and - can also work as unary operators.

They are used to specify the positive or negative sign of the operand. Unary + or - operators have higher precedence than the binary + or - operators.

```
-4 // Negative 4
+5 // Positive 5
```

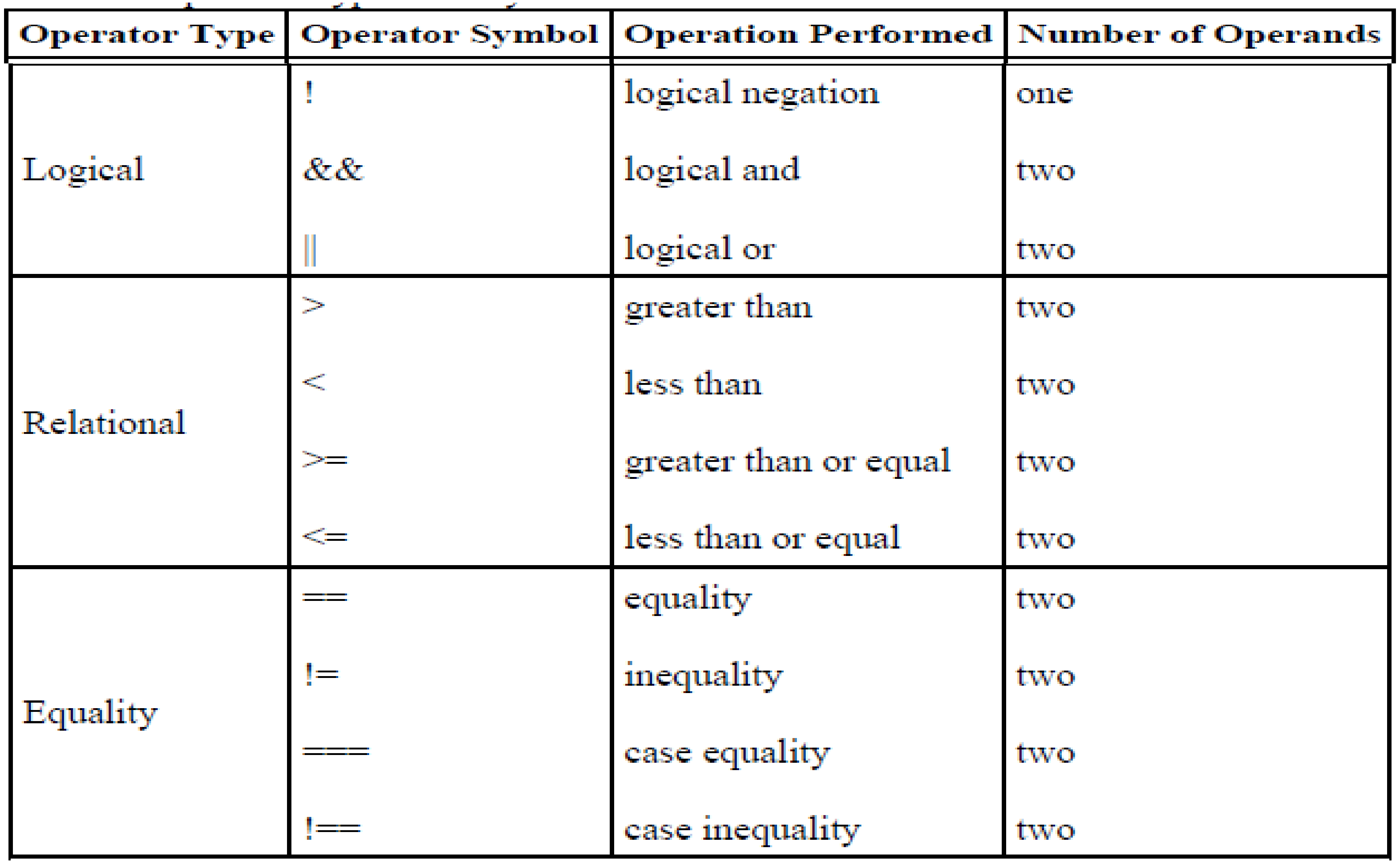
Negative numbers are represented as 2's complement internally in Verilog.

It is advisable to use negative numbers only of the type integer or real in expressions.

Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

```
//Advisable to use integer or real numbers
-10 / 5// Evaluates to -2
//Do not use numbers of type <sss> '<base> <nnn>
-'d10 / 5// Is equivalent (2's complement of 10)/5 = (232 - 10)/5
// where 32 is the default machine word width.
// This evaluates to an incorrect and unexpected result
```

OPERATOR TYPES –Cont’d



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Logical Operators

Logical operators are logical-and (&&), logical-or (||) and logical-not (!).

Operators && and || are binary operators. Operator ! is a unary operator.

Logical operators follow these conditions:

1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x(ambiguous).

2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition).

If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.

3. Logical operators take variables or expressions as operands.

NOTE: Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Logical Operators

- // 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.

Relational Operators

- Relational operators are
- greater-than (>)
- less-than (<)
- greater-than-or-equal-to (>=) and
- 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
- These operators function exactly as the corresponding operators in the C programming language

Relational Operators

- // 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

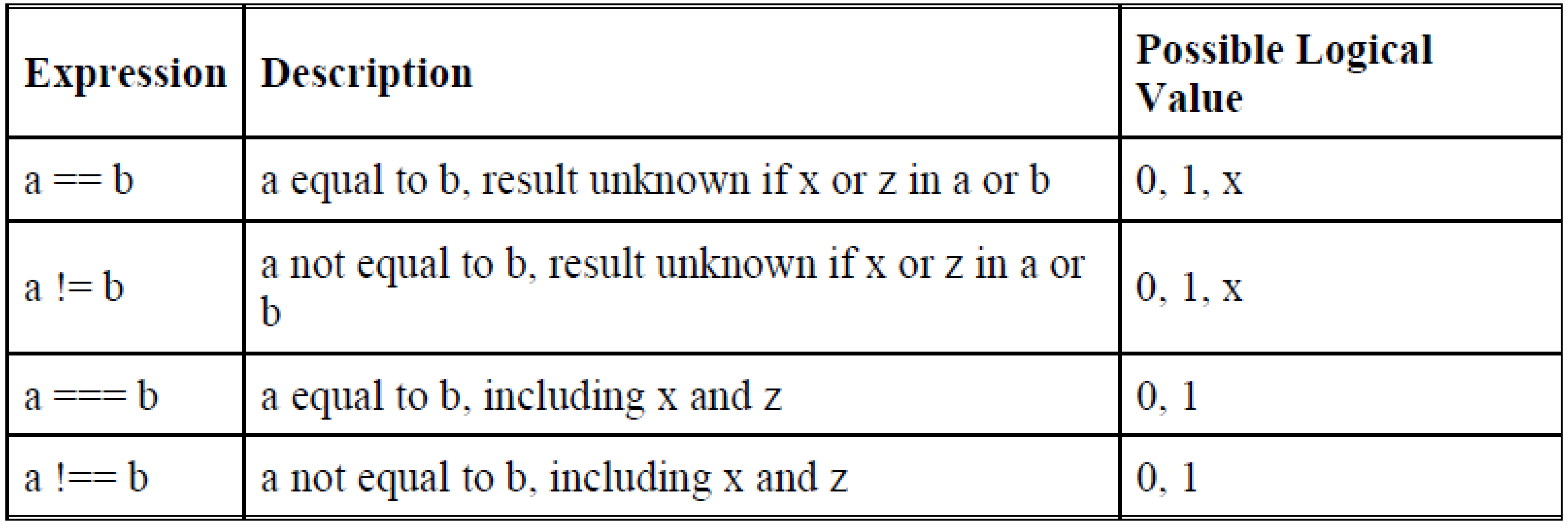
Equality Operators

- Equality operators are
- logical equality (==),
- logical inequality (!=),
- case equality (===),
- and case inequality (!==).

- When used in an expression, equality operators return logical value 1 if true, 0 if false.

- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.

Equality Operators

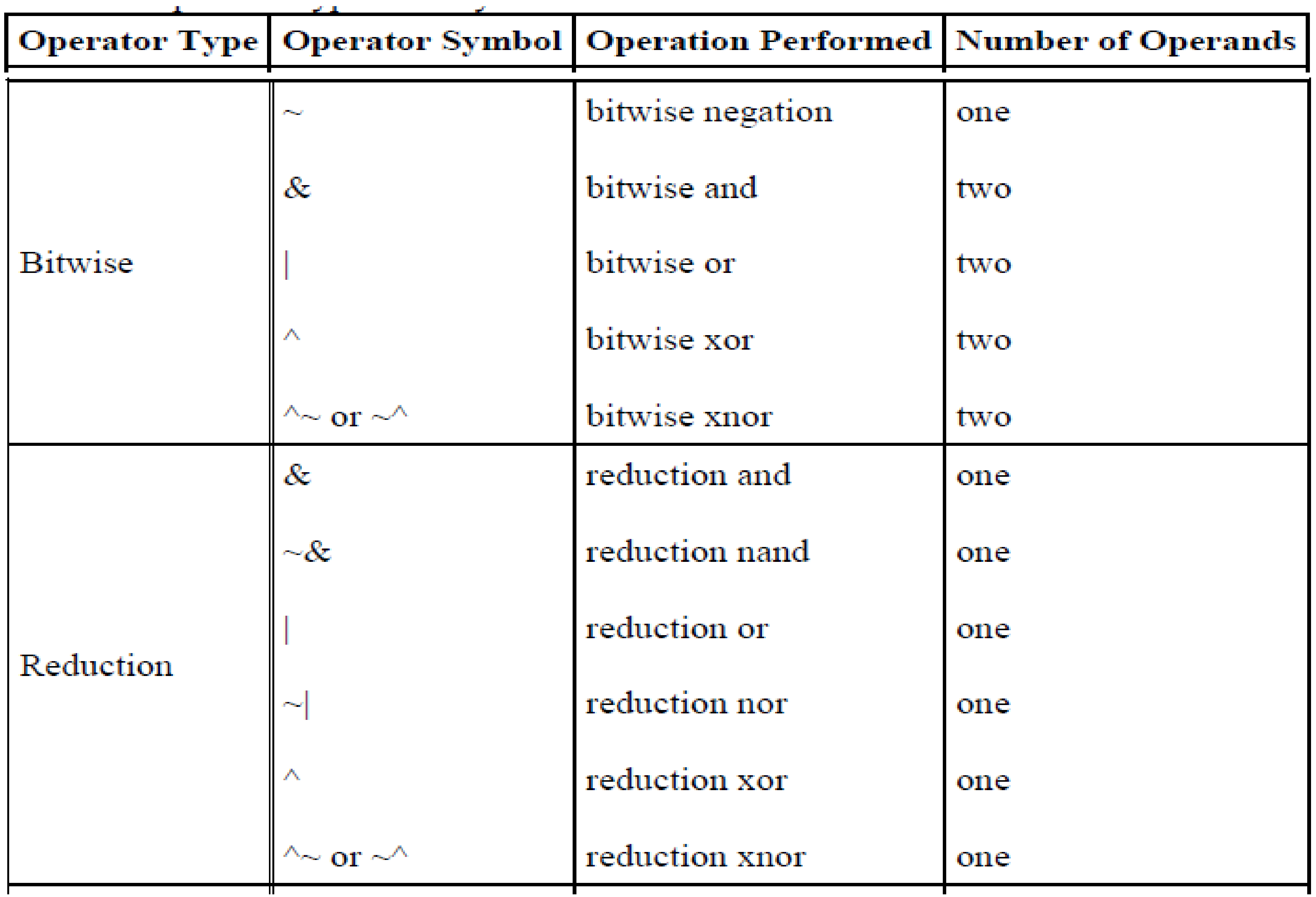


Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

Equality Operators

- It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==).
- 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.
- // 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

OPERATOR TYPES –Cont’d



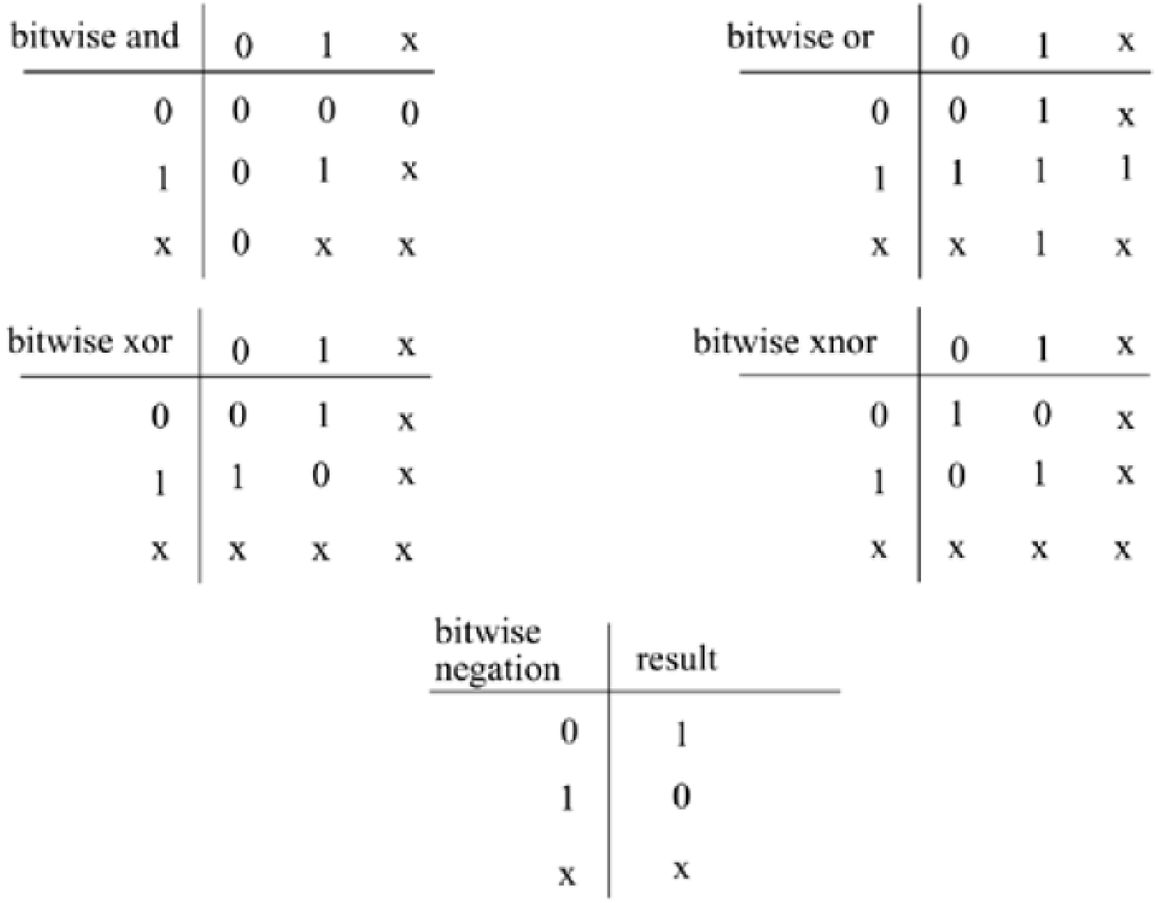
Operator Type	Operator Symbol	Operation Performed	Number of Operands
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Bitwise Operators

- Bitwise operators are
- negation (~)
- and(&)
- or (|)
- xor (^)
- xnor (^~, ~^)
- Bitwise operators perform a bit-by-bit operation on two operands
- They take each bit in one operand and perform the operation with the corresponding bit in the other operand
- 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

Bitwise Operators

Table 6-3. Truth Tables for Bitwise Operators



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

Bitwise Operators

- Examples of bitwise operators are shown below.
- // 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
- 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.

Reduction Operators

- Reduction operators are
- and (&)
- nand (~&)
- or (|)
- nor (~|)
- xor (^), and
- xnor (~^, ^~)

- Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result

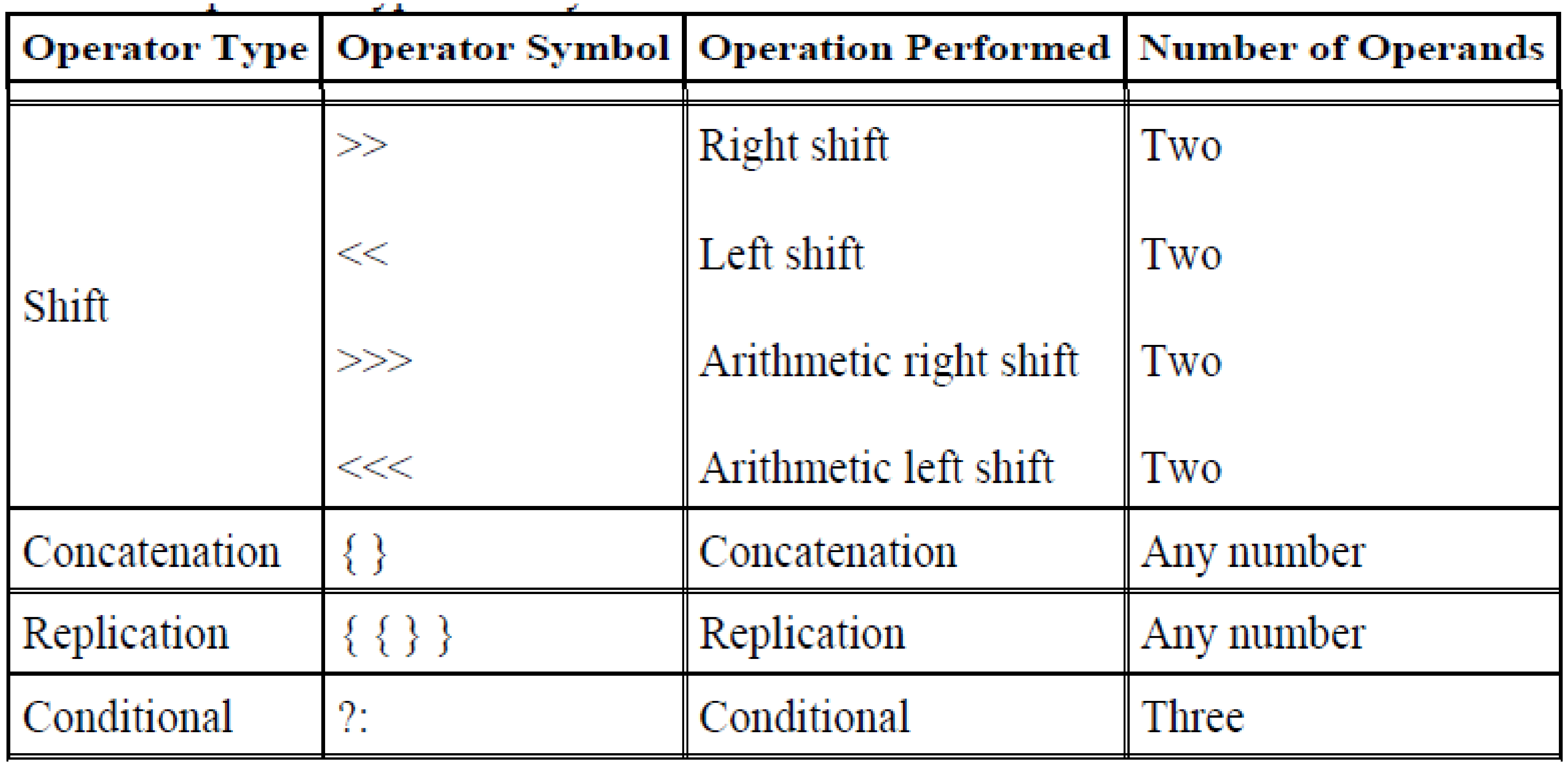
- The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand

- Reduction operators work bit by bit from right to left

Reduction Operators

- Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.
- // 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.
- 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.

OPERATOR TYPES –Cont’d



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Shift Operators

- Shift operators are
- right shift (>>),
- left shift (<<),
- arithmetic right shift (>>>), and
- arithmetic left shift (<<<).

- Regular shift operators shift a vector operand to the right or the left by a specified number of bits.
- The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros.
- Shift operations do not wrap around.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

Shift Operators

- // 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
- Shift operators are useful because they allow the designer to model shift operations, shiftand-
- add algorithms for multiplication, and other useful operations.

Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple 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.
- Concatenations are expressed as operands within braces, with commas separating the operands.

- 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

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 ({ }).
- 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

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.

- If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.
- Alternately, it can be compared to the if-else expression.

Conditional Operator

- Conditional operators are frequently used in dataflow modeling to model conditional assignments.
- The conditional expression acts as a switching control.
- //model functionality of a tristate buffer
- assign addr_bus = drive_enable ? addr_out : 36'bz;
- //model functionality of a 2-to-1 mux
- assign out = control ? in1 : in0;
- Conditional operations can be nested. Each true_expr or false_expr can itself be a conditional operation.
- In the example that follows, convince yourself that (A==3) and
- control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and
- out as the output signal.
- assign out = (A == 3) ? (control ? x : y): (control ? m : n) ;