DIGITAL SYSTEM DESIGN USING VERILOG

COURSE CODE: 19EC5DCDSV (3 CREDITS)

MODULE 1C

Faculty In-Charge: Dr. Dinesha P

drdinesh-ece@dayanandasagar.edu

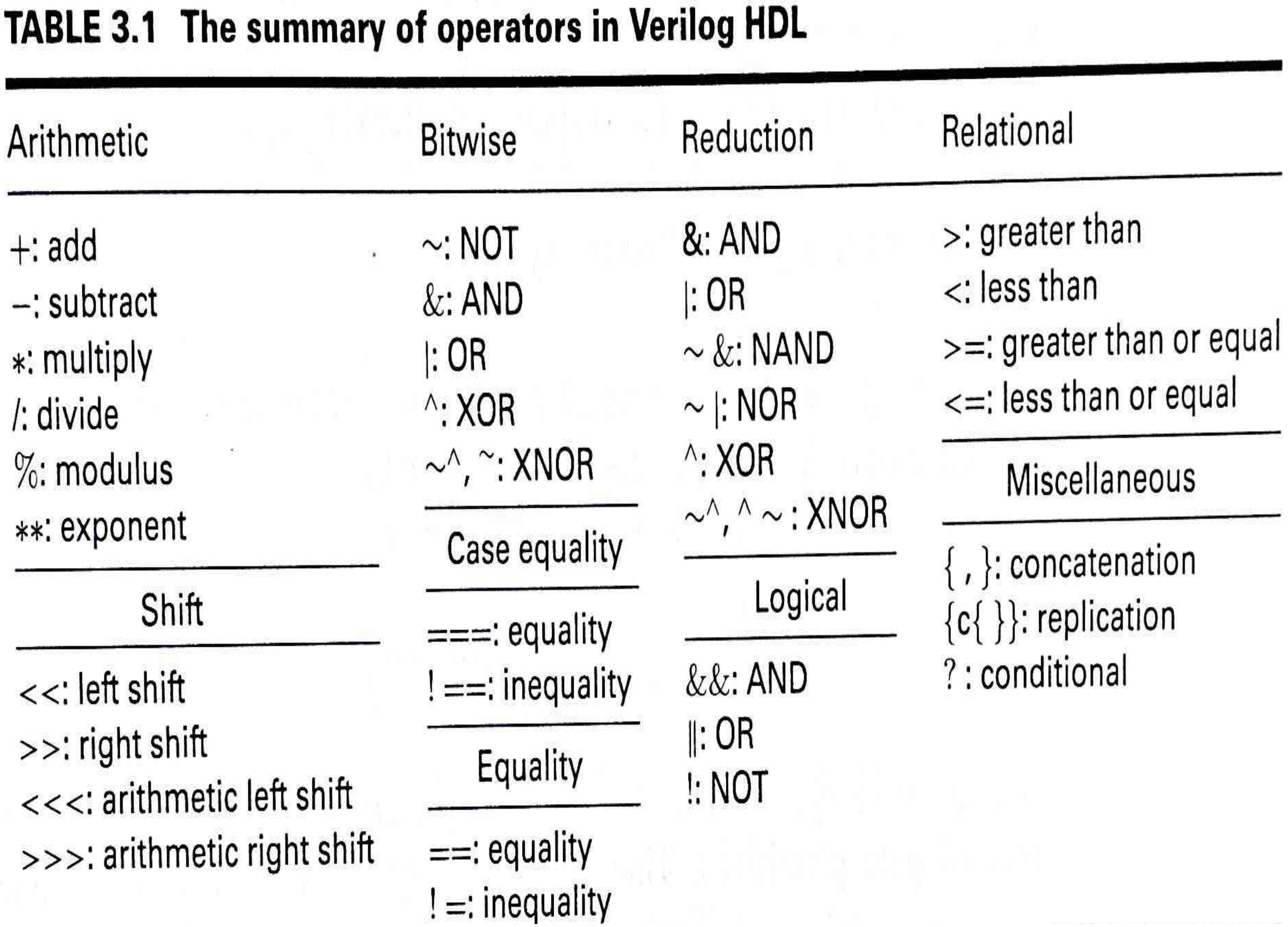


TABLE 3.1 The summary of operators in Verilog HDL

Arithmetic	Bitwise	Reduction	Relational
+: add	~: NOT	&: AND	>: greater than
–: subtract	&: AND	|: OR	<: less than
*: multiply	|: OR	~ &: NAND	>=: greater than or equal
/: divide	^: XOR	~ |: NOR	<=: less than or equal
%: modulus	~^, ^~: XNOR	^: XOR	**Miscellaneous**
**: exponent	**Case equality**	~^, ^ ~ : XNOR	{ , }: concatenation
Shift	===: equality	**Logical**	{c{ }}: replication
<<: left shift	! ==: inequality	&&: AND	? : conditional
>>: right shift	**Equality**	||: OR	
<<<: arithmetic left shift	==: equality	!: NOT	
>>>: arithmetic right shift	! =: inequality		

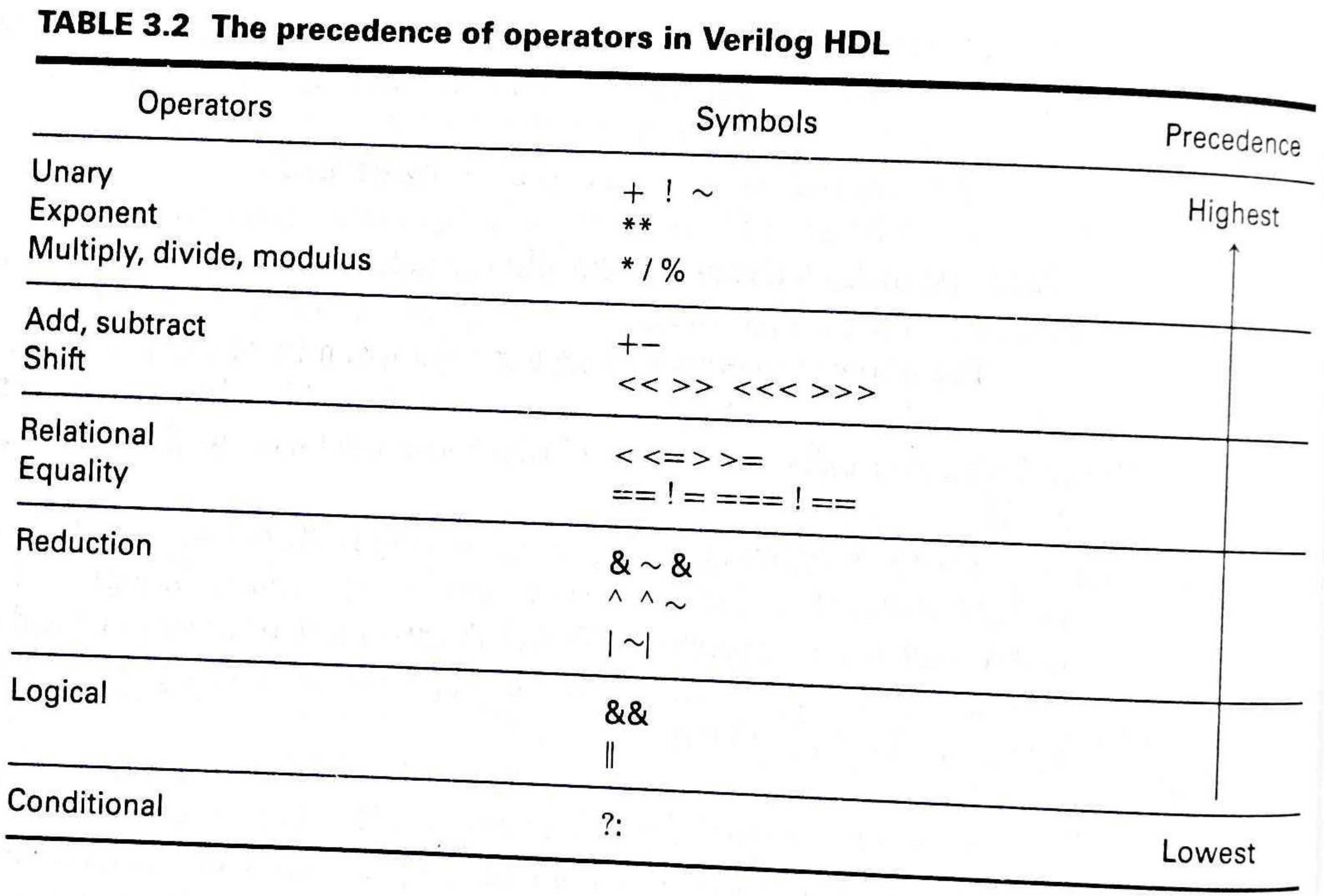


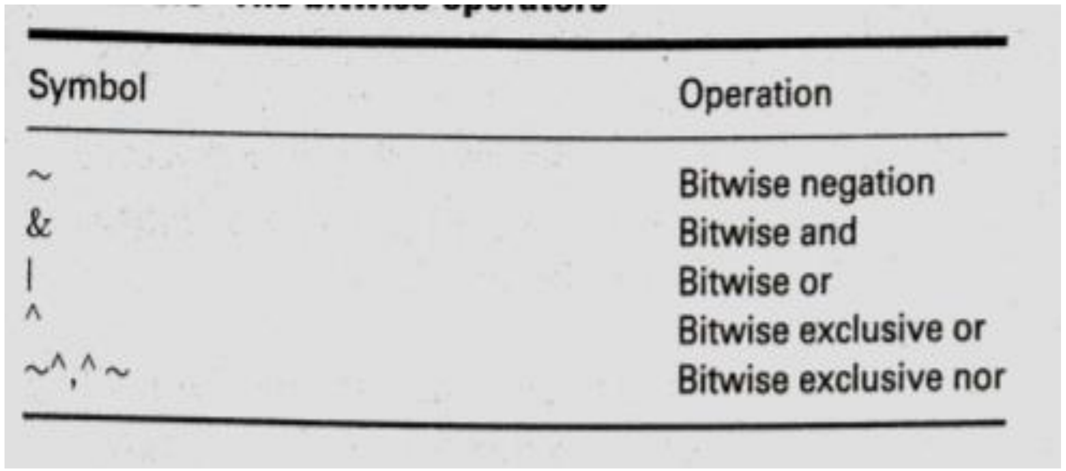
TABLE 3.2 The precedence of operators in Verilog HDL

Operators	Symbols	Precedence
Unary Exponent Multiply, divide, modulus	+ ! ~ ** * / %	Highest
Add, subtract Shift	+− << >> <<< >>>	
Relational Equality	< <= > >= == != === !==	
Reduction	& ~& ^ ^~ | ~|	
Logical	&& ||	
Conditional	?:	Lowest

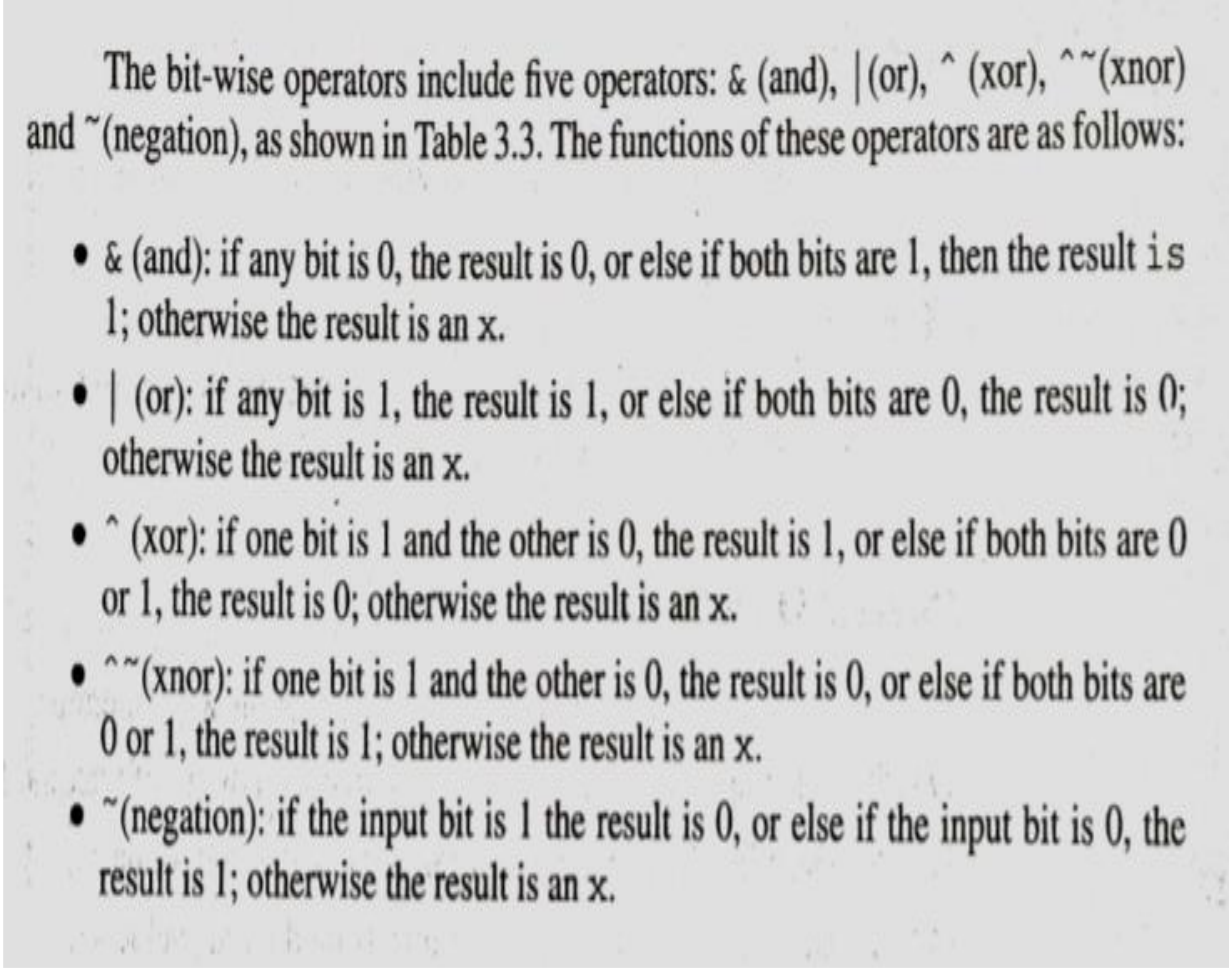
Verilog Operators:

Verilog HDL has a rich set of operators, including arithmetic operators, bit wise operators, logical operators, concatenation and replication operators, relational operators, equality operators, shift operators, and conditional operators.

Bit wise operators: Bit wise operators perform a bit by bit operation on two operands and produce a vector results. In bit wise operators, a z is treated as unknown x,. When two operands are not equal length, the shorter operand is zero-extended to match of the loner operand.



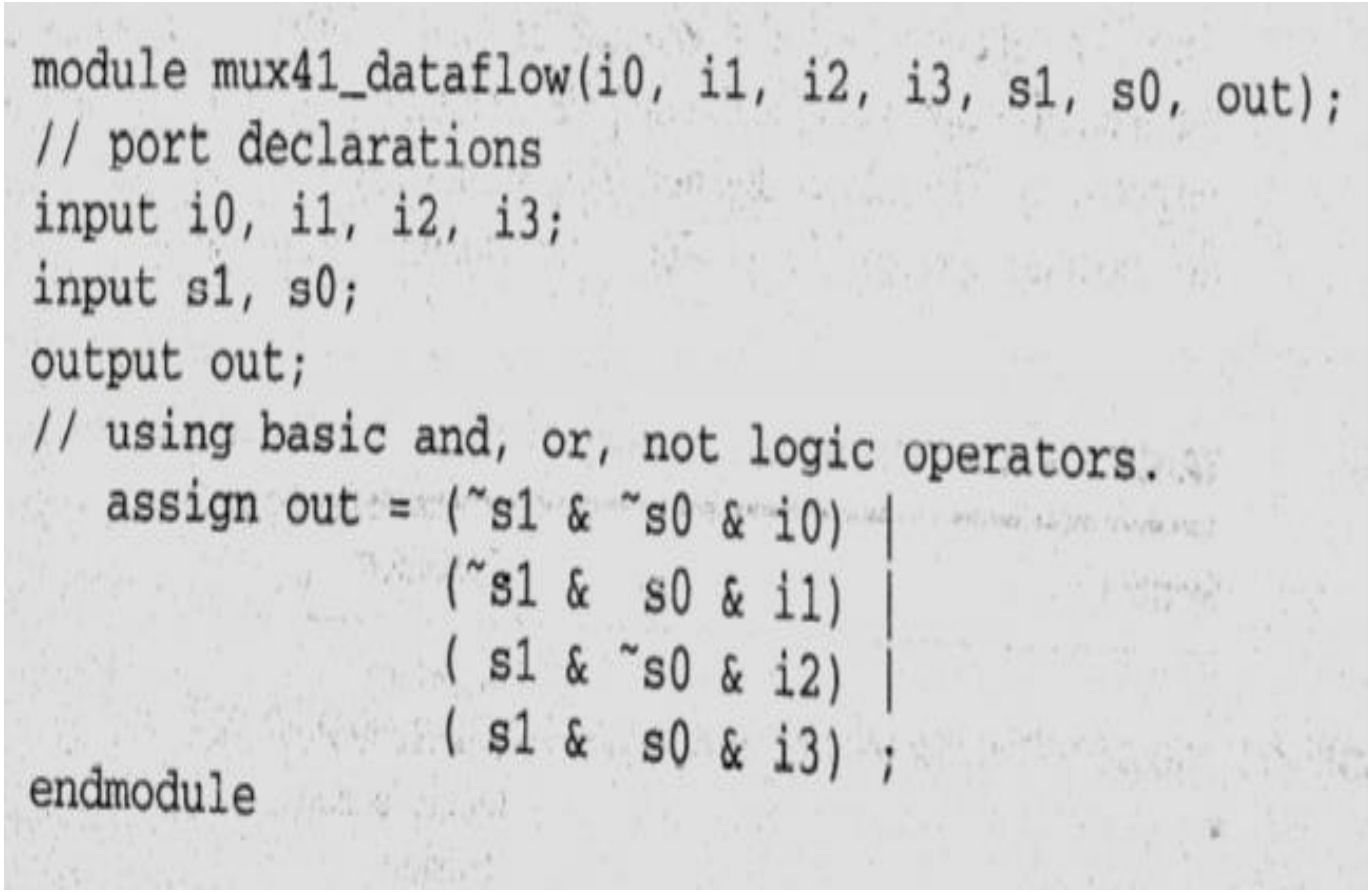
Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor



The bit-wise operators include five operators: & (and), | (or), ^ (xor), ^~(xnor) and ~(negation), as shown in Table 3.3. The functions of these operators are as follows:

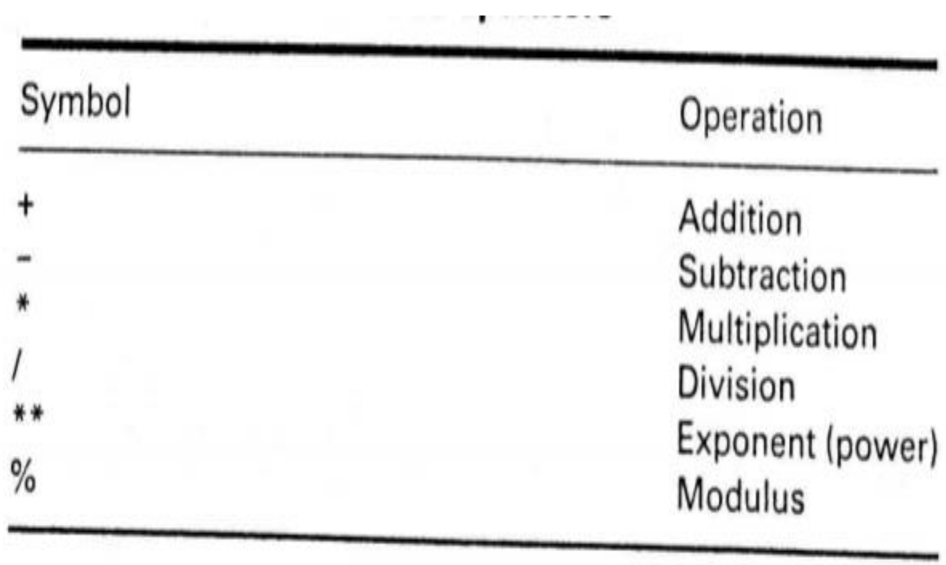
- & (and): if any bit is 0, the result is 0, or else if both bits are 1, then the result is 1; otherwise the result is an x.
- | (or): if any bit is 1, the result is 1, or else if both bits are 0, the result is 0; otherwise the result is an x.
- ^ (xor): if one bit is 1 and the other is 0, the result is 1, or else if both bits are 0 or 1, the result is 0; otherwise the result is an x.
- ^~(xnor): if one bit is 1 and the other is 0, the result is 0, or else if both bits are 0 or 1, the result is 1; otherwise the result is an x.
- ~(negation): if the input bit is 1 the result is 0, or else if the input bit is 0, the result is 1; otherwise the result is an x.

Dataflow model for a 4:1 multiplexer

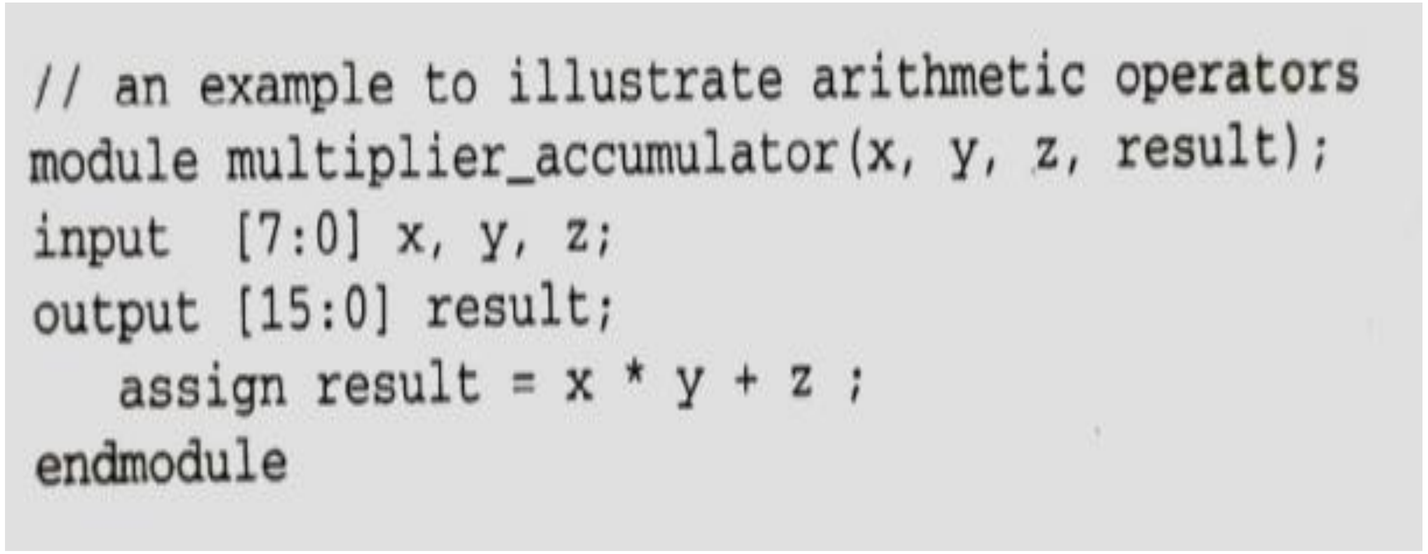


```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// using basic and, or, not logic operators.
   assign out = (~s1 & ~s0 & i0) |
                (~s1 &  s0 & i1) |
                ( s1 & ~s0 & i2) |
                ( s1 &  s0 & i3) ;
endmodule
```

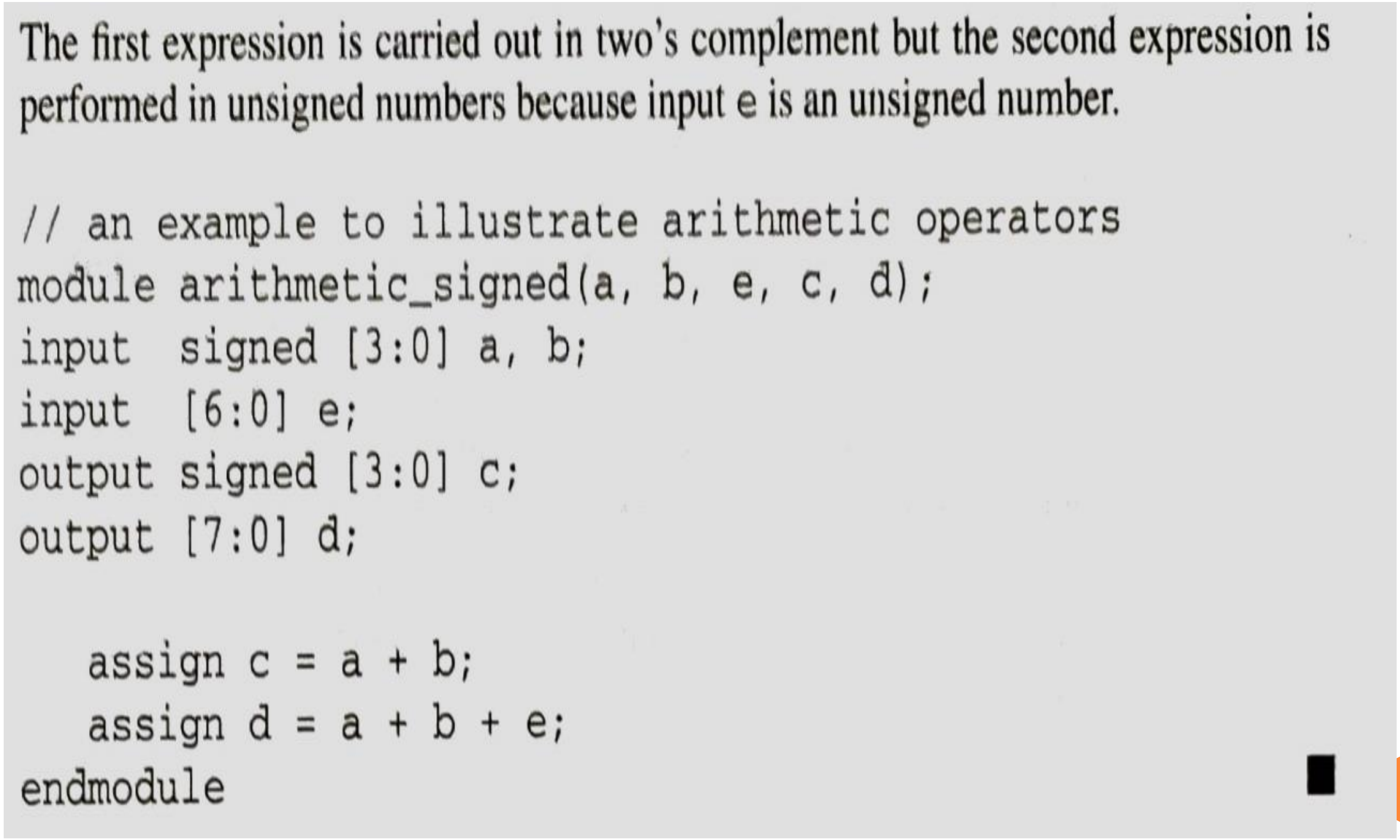
Arithmetic Operators:



Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus



```
// an example to illustrate arithmetic operators
module multiplier_accumulator(x, y, z, result);
input  [7:0] x, y, z;
output [15:0] result;
   assign result = x * y + z ;
endmodule
```

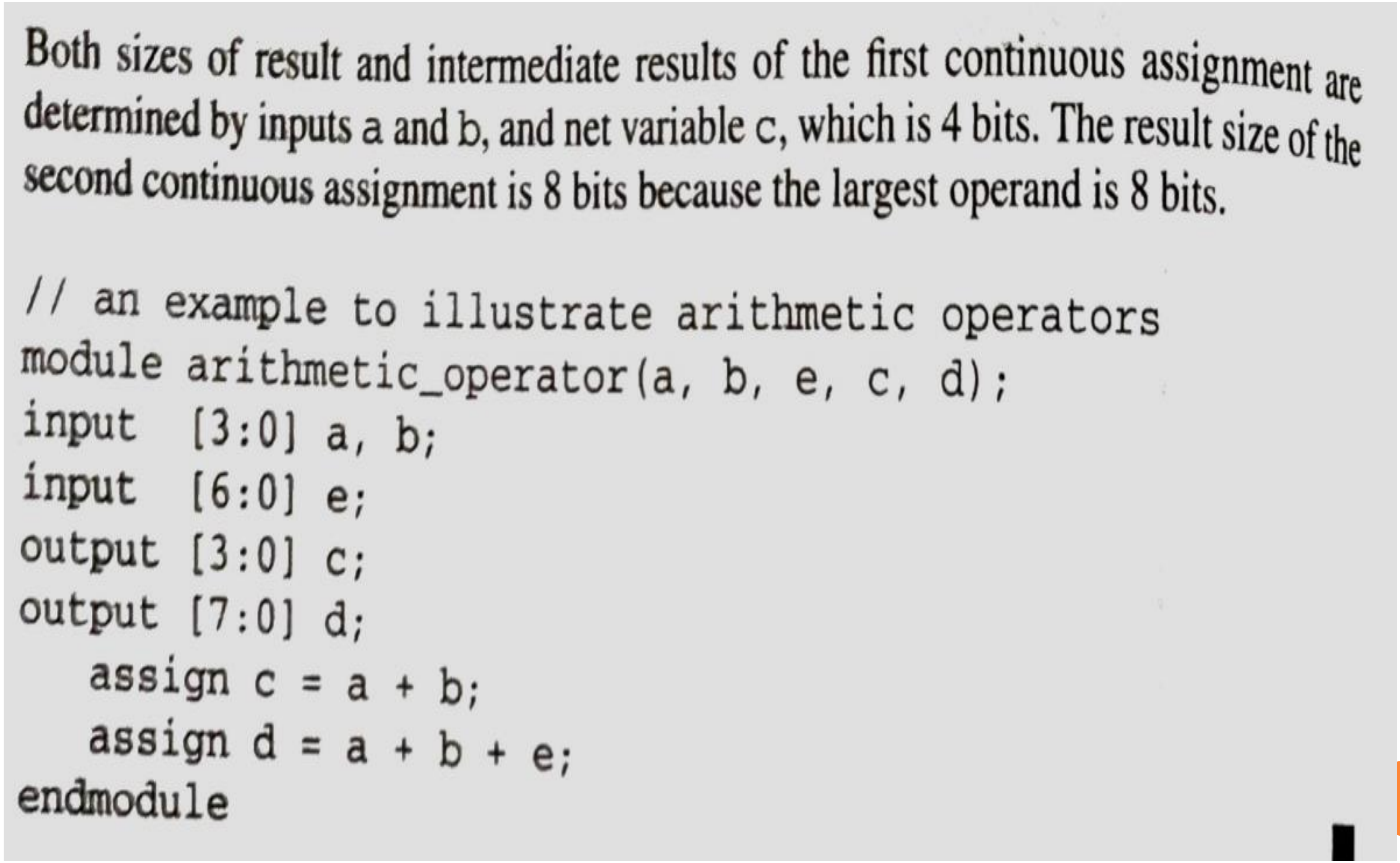


The first expression is carried out in two's complement but the second expression is performed in unsigned numbers because input e is an unsigned number.

```
// an example to illustrate arithmetic operators
module arithmetic_signed(a, b, e, c, d);
input  signed [3:0] a, b;
input  [6:0] e;
output signed [3:0] c;
output [7:0] d;

   assign c = a + b;
   assign d = a + b + e;
endmodule
```

■

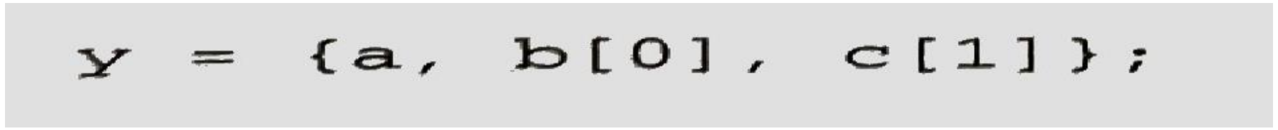


Both sizes of result and intermediate results of the first continuous assignment are determined by inputs a and b, and net variable c, which is 4 bits. The result size of the second continuous assignment is 8 bits because the largest operand is 8 bits.

```
// an example to illustrate arithmetic operators
module arithmetic_operator(a, b, e, c, d);
input  [3:0] a, b;
input  [6:0] e;
output [3:0] c;
output [7:0] d;
   assign c = a + b;
   assign d = a + b + e;
endmodule
```

Concatenation and Replication operations:

Concatenation operator is expressed by { and }, with commas separating the expression as shown in the table.



```
y = {a, b[0], c[1]};
```

Another form of concatenation is replication as shown the table

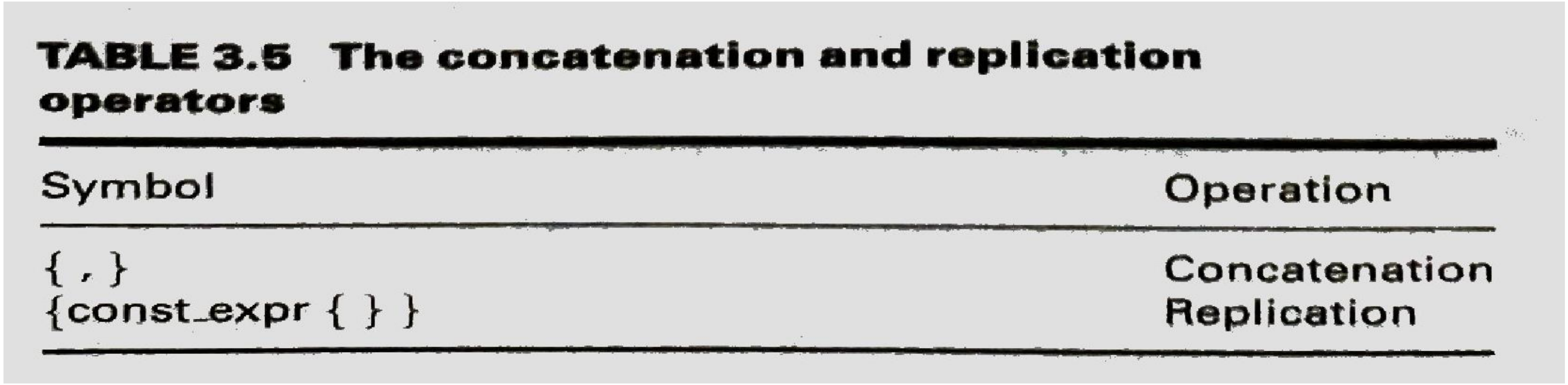
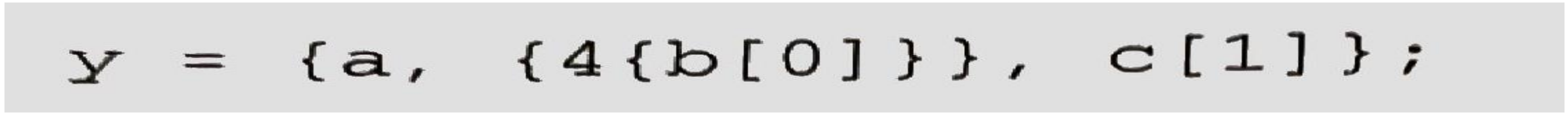


TABLE 3.5 The concatenation and replication operators

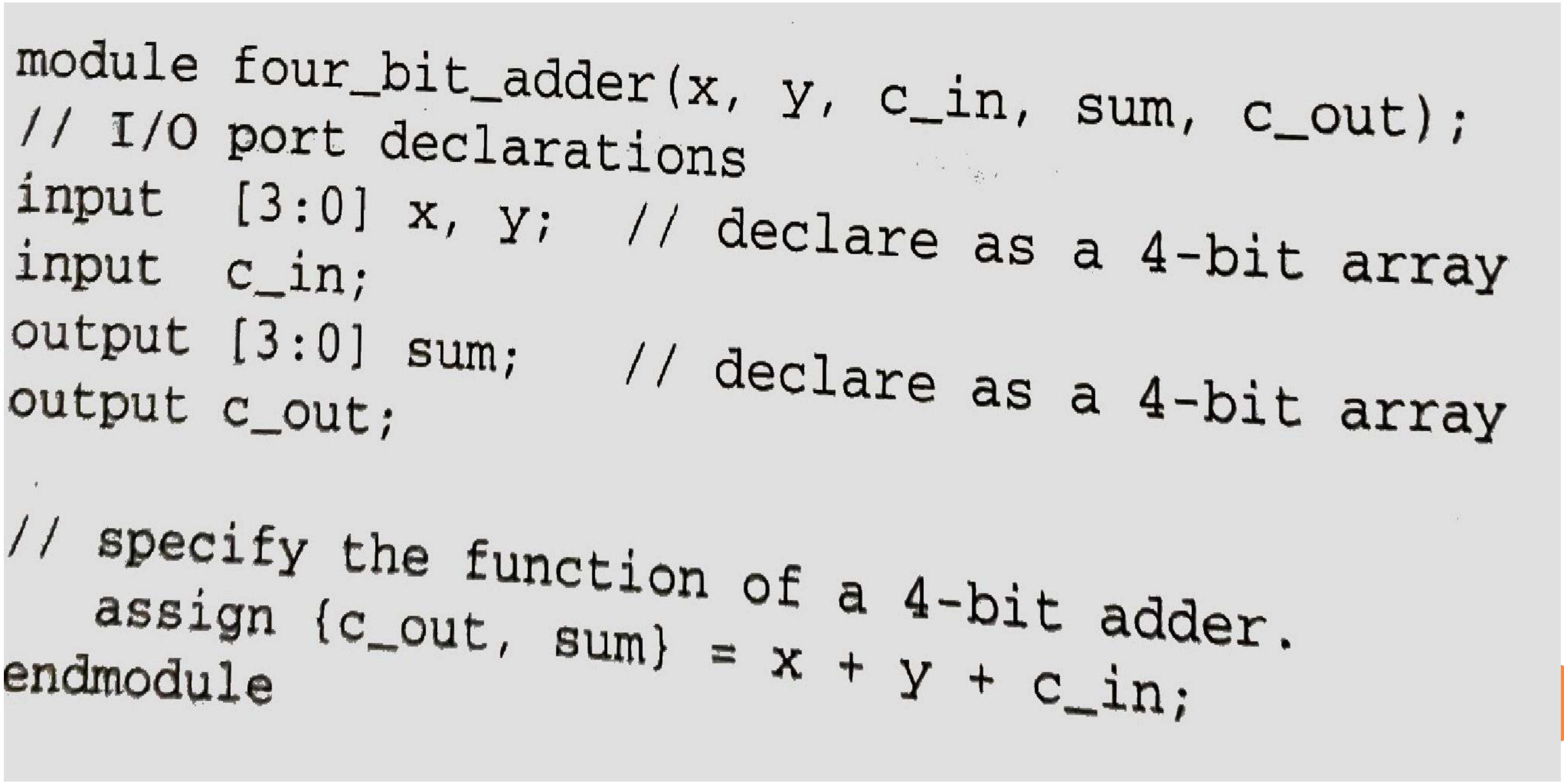
Symbol	Operation
{ , }	Concatenation
{const_expr { } }	Replication



```
y = {a, {4{b[0]}}, c[1]};
```

The right hand side has six bits in total

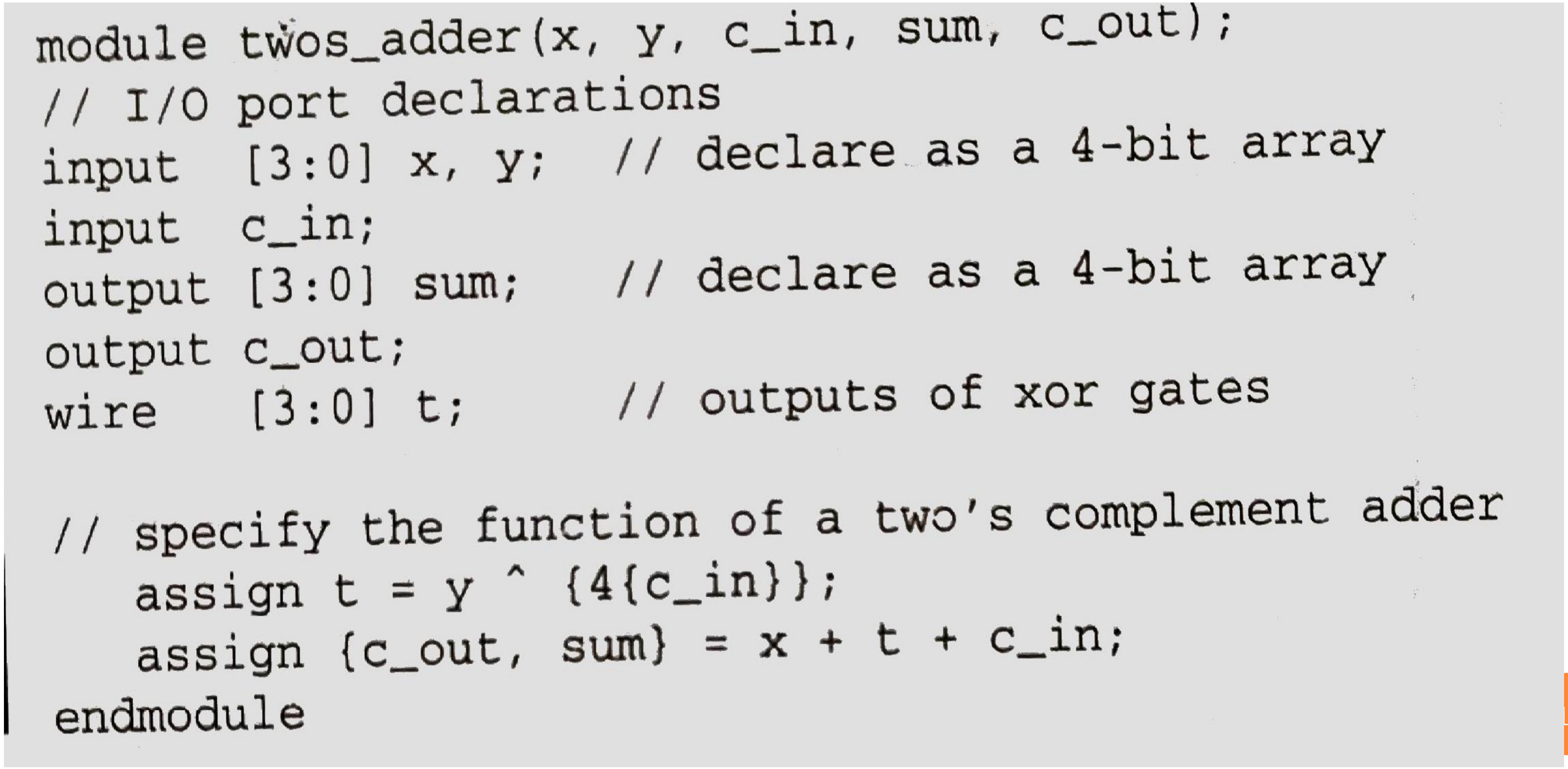
In this example, a 4 bit adder is described using dataflow description, illustrate the use of cancatenation



```
module four_bit_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  [3:0] x, y;  // declare as a 4-bit array
input  c_in;
output [3:0] sum;   // declare as a 4-bit array
output c_out;

// specify the function of a 4-bit adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```

The following example illustrate the use of replication statement
Example: 4 bit two’s compliment adder

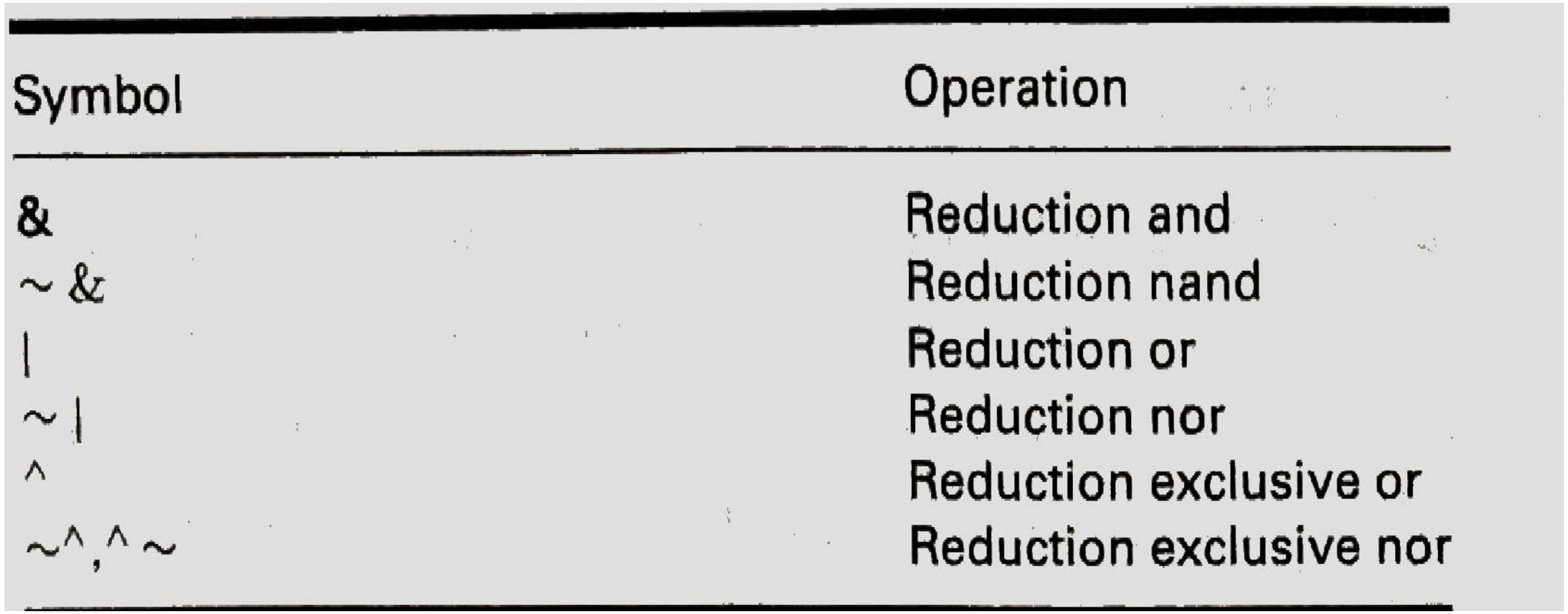


```
module twos_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  [3:0] x, y;  // declare as a 4-bit array
input  c_in;
output [3:0] sum;   // declare as a 4-bit array
output c_out;
wire   [3:0] t;     // outputs of xor gates

// specify the function of a two's complement adder
   assign t = y ^ {4{c_in}};
   assign {c_out, sum} = x + t + c_in;
endmodule
```

Reduction Operator:

These operators are unary operator, performs bit wise operation on single vector operand and gives the single bit output. Reduction operator on one vector operand and work in bit by bit way from right to left. The reduction operator are shown in table.

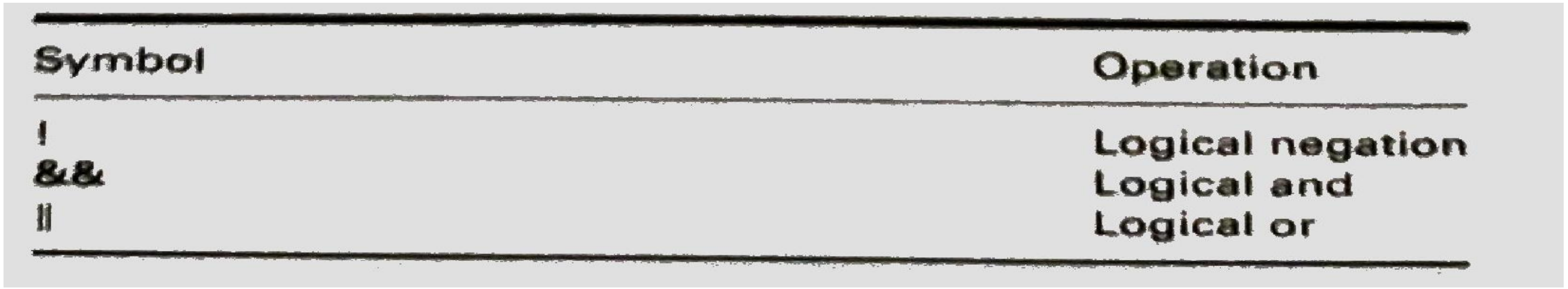


Symbol	Operation
&	Reduction and
~ &	Reduction nand
|	Reduction or
~ |	Reduction nor
^	Reduction exclusive or
~^, ^ ~	Reduction exclusive nor

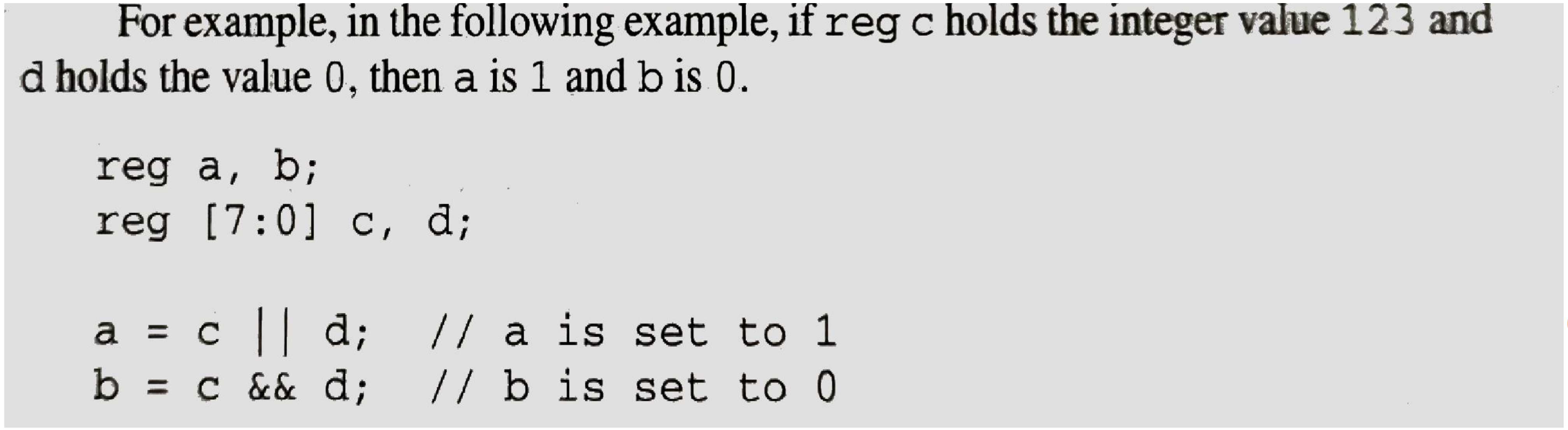
A 9-bit Parity Generator Using a Reduction Operator

```
module parity_gen_9b_reduction(x, ep,op);
// I/O port declarations
input  [8:0] x;
output ep, op;
// dataflow modeling using reduction operator
   assign ep = ^x;    // even parity generator
   assign op = ~ep;   // odd parity generator
endmodule
```

Logical operator: There are 3 logical operators. && -AND , II-OR and !- not as shown in table. These operator operate on logical values 0 or 1 and produces a0.1 or X. If any operand bit is x or z, treated as x and in simulator consider as a false condition.



Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or



For example, in the following example, if reg c holds the integer value 123 and d holds the value 0, then a is 1 and b is 0.

```
reg a, b;
reg [7:0] c, d;

a = c || d;  // a is set to 1
b = c && d;  // b is set to 0
```