Design with Verilog

Chap 6 - Introduction to HDL (b)

Credit to: MD Rizal Othman
Faculty of Electrical & Electronics Engineering
Universiti Malaysia Pahang

Ext: 6036

Language Elements

1. **Operators**

   There are three types of operators: unary, binary, and ternary, which have one, two, and three operands respectively.

   Unary : Single operand, which precede the operand.
   Ex: x = ~y
   ~ is a unary operator
   y is the operand

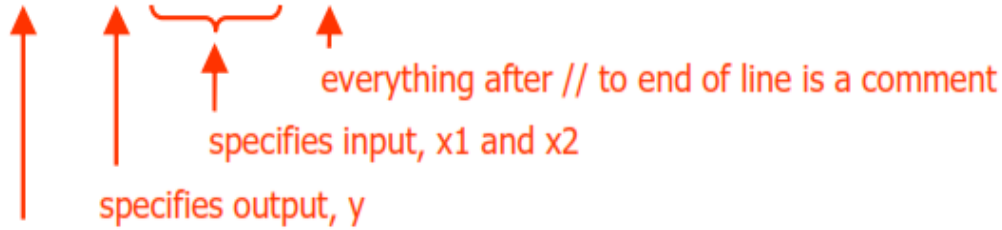
   binary : Comes between two operands.
   Ex: x = y || z
   || is a binary operator
   y and z are the operands

   ternary : Ternary operators have two separate operators that separate three operands.
   Ex: p = x ? y : z
   ? : is a ternary operator
   x, y, and z are the operands

3.1 Gate level Primitives

- Verilog includes a set of *gate level primitives* corresponding to commonly used logic gates

```
and (y, x1, x2); // 2-input AND gate
```



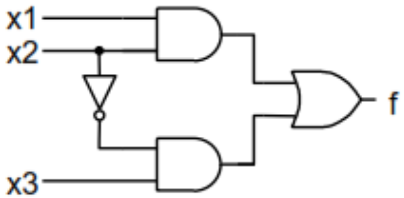

```
and (f, a, b, c); // 3-input AND gate
and (out, in1, in2, in3, in4); // 4-input AND gate
```

More primitives

```
or (out, in1, in2);
not (out, in);
nand (out, in1, in2);
nor (out, in1, in2);
xor (out, in1, in2);
xnor (out, in1, in2);
```

Example 1

- Using primitives let's implement a circuit in Verilog

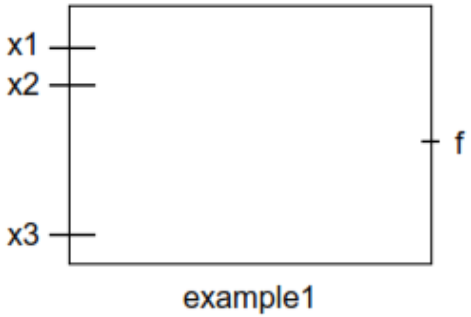



```
// example1.v
module example1 (x1, x2, x3, f);

endmodule
```

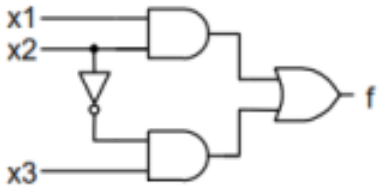
module indicates the start of our specification, endmodule indicates the end

we have a "black box" named example1

"black box" has 4 ports – x1, x2, x3, f



Example 1 Continued

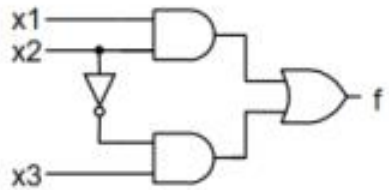



```
// example1.v
module example1 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;

endmodule
```

input statement indicates x1, x2, and x3 are inputs to the systems

output statement indicates f is an output of the system

Example 1 Continued




```
// example1.v
module example1 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    wire g;

    and (g, x1, x2);

endmodule
```

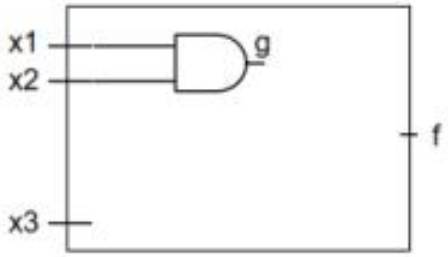
actual structure of the circuit specified by the primitives

first and statement specifies an AND gate with input x1 and x2, output g

g is internal value, use wire

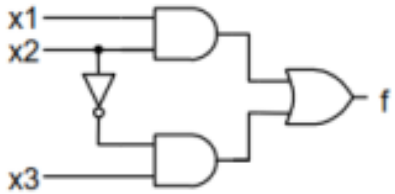
wire is just a connection, does not store value

CAUTION – Verilog will implicitly declare wire if you don't but it will be a 1-bit wire



example1

Example 1 Continued

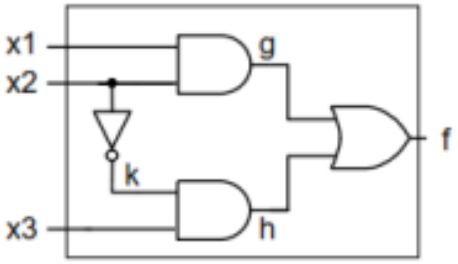



```
// example1.v
module example1 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    wire g, k, h;

    and (g, x1, x2);
    not (k, x2);
    and (h, k, x3);
    or (f, g, h);

endmodule
```

not statement specifies a NOT gate with input x2 and output k

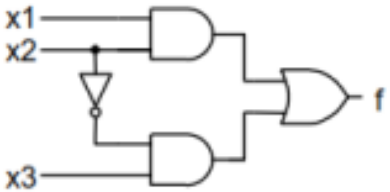
next and statement specifies an AND gate with input k and x3, output h

or statement specifies an OR gate with input g and h, output f



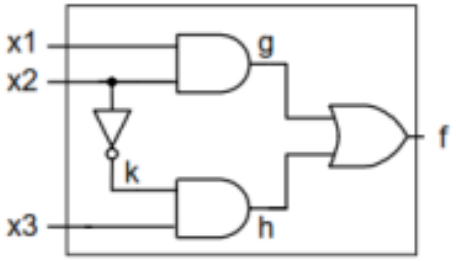
example1

Example 1 Continued




```
// example1.v
module example1 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    wire g, k, h;

    and (g, x1, x2);
    not (k, x2);
    and (h, k, x3);
    or (f, g, h);

endmodule
```

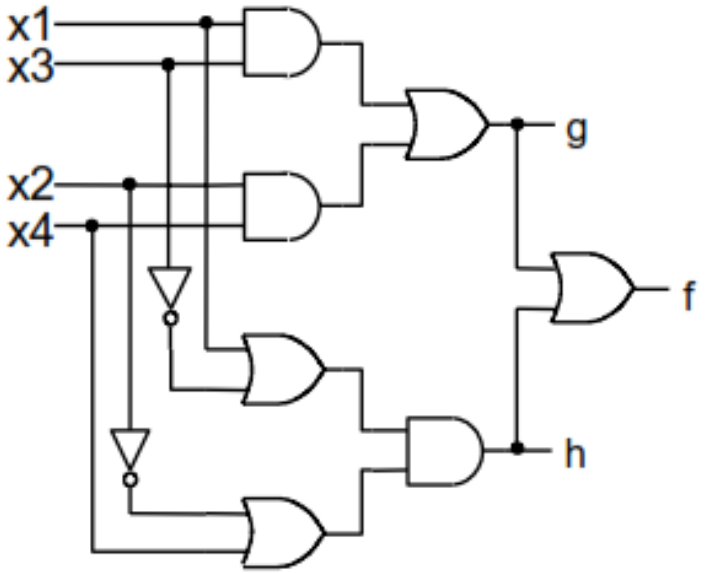
Done!
We have implemented our circuit in Verilog



example1

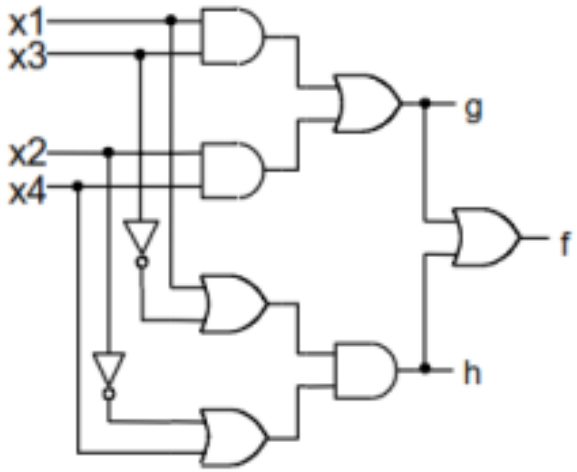
Example 2

- Let's try another example



Example 2

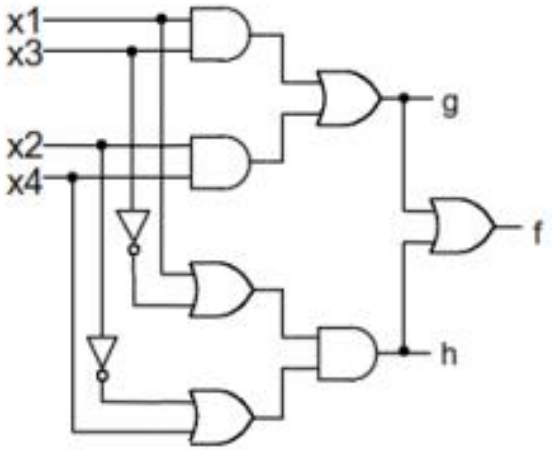
- Let's try another example




```
// Example 2
// g = x1x3 + x2x4
// h = (x1+x3')(x2'+x4)
// f = g +h

module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

enclose description between module and endmodule

Example 2 Continued

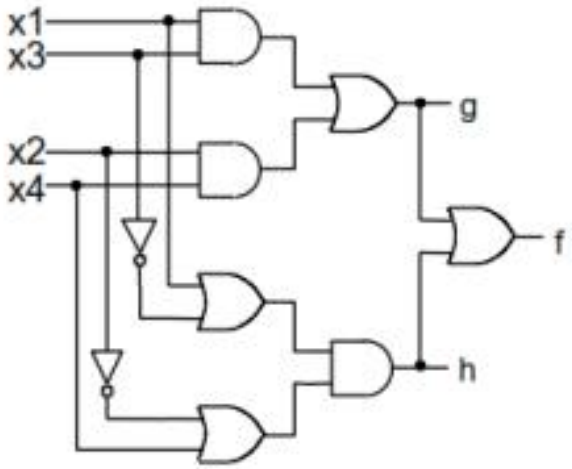



```
// Example 2
// g = x1x3 + x2x4
// h = (x1+x3')(x2'+x4)
// f = g +h

module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

module named example2

7 port signals
- x1, x2, x3, x4, f, g, h

Example 2 Continued

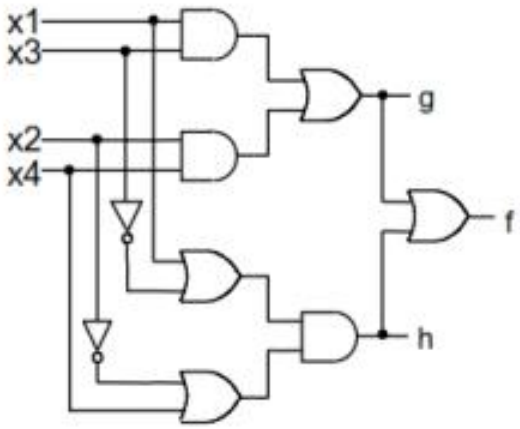



```
// Example 2
// g = x1x3 + x2x4
// h = (x1+x3')(x2'+x4)
// f = g +h

module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

x1, x2, x3, and x4 are inputs to the system

f, g, h are outputs of the system

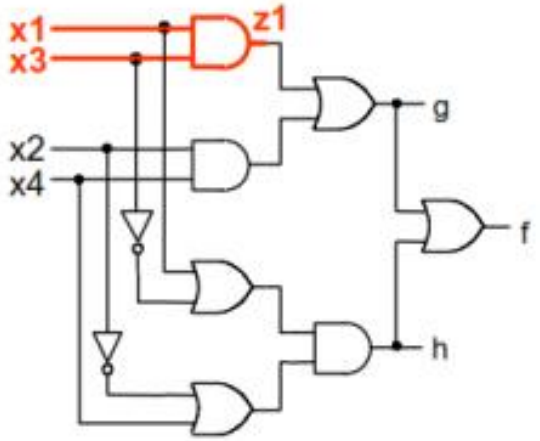
Example 2 Continued




```
// Example 2
// g = x1x3 + x2x4
// h = (x1+x3')(x2'+x4)
// f = g +h

module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

we define how the circuit is connected

Example 2 Continued




```
// Example 2
// g = x1x3 + x2x4
// h = (x1+x3')(x2'+x4)
// f = g +h

module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

we define how the circuit is connected

3.2 Operator

Bit-Wise Operator

- Verilog includes a set of *bit-wise operators*
  - Takes 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 extended on the left side with zeroes to match the length of the longer operand

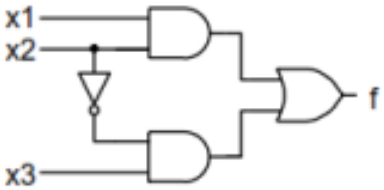
```
f = a & b; // bit-wise AND
    if a = 00, b = 01, then f = 00
    if a = 01, b = 01, then f = 01
    if a = 11, b = 10, then f = 10

f = ~a + b; // bit-wise NOT then OR
    if a = 00, b = 01, then f = 11
    if a = 01, b = 01, then f = 11
    if a = 11, b = 10, then f = 10
```

Bit-wise Operators

~	NOT
&	AND
|	OR
^	XOR
^~ or ~^	XNOR

Example 3 - Modeling a Circuit1 With Bit-wise Operators

- Model circuit from example 1 using bit-wise operators




```
// Example 3

module example3 (x1, x2, x3, f);
    input x1, x2, x3;
    output f ;

    assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

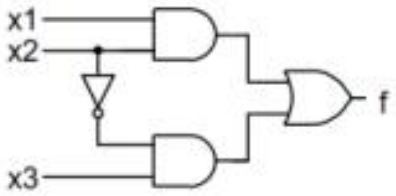
same module and endmodule

same module naming, port list

same input/output declarations

difference in how we describe the circuit

Example 3 Continued



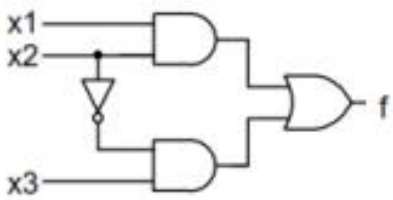
we know how to go from logic circuit to logic expression

$f = (x1 \cdot x2) + (x2' \cdot x3)$

replace AND, OR, NOT operations with Verilog operators

```
// Example 3

module example3 (x1, x2, x3, f);
    input x1, x2, x3;
    output f ;

    assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

Example 3 Continued

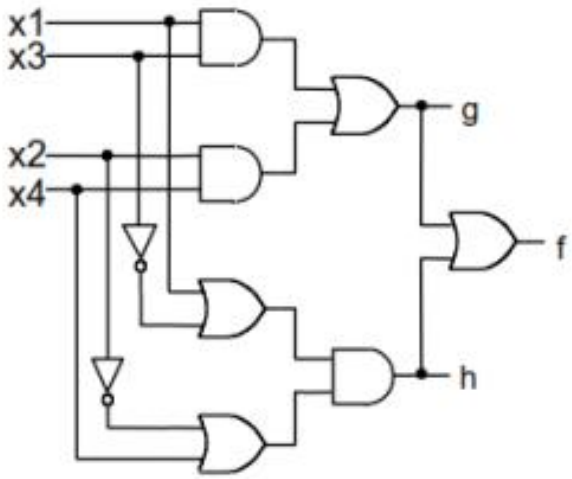



```
// Example 3

module example3 (x1, x2, x3, f);
    input x1, x2, x3;
    output f ;

    assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

assign keyword indicates anytime something changes on right hand side, re-evaluate left hand side

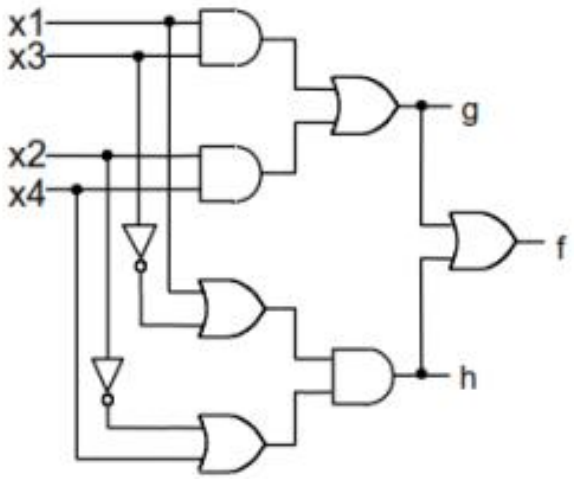
Example 4 - Modeling Circuit With Bit-wise Operators

- Model circuit from example 2 using bit-wise operators



Example 4 - Modeling Circuit With Bit-wise Operators

- Model circuit from example 2 using bit-wise operators




```
// Example 4

module example4 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    assign g = (x1 & x3) | (x2 & x4);
    assign h = (x1 | ~x3) & (~x2 | x4);
    assign f = g | h;

endmodule
```

3.2. Operators

Arithmetic Operators

- These perform arithmetic operations. The **+ and - can be used as either unary (-z) or binary (x-y) operators.**
- ***Operators***

  + (addition)

  - (subtraction)

  * (multiplication)

  / (division)

  % (modulus)

- Examples:

```
parameter n = 4;
reg[3:0] a, c, f, g, count;
f = a + c;
g = c - n;
count = (count +1)%16;          //Can count 0 thru 15.
```

Relational Operators

- Relational operators compare two operands and return a single bit 1 or 0.
- example a < b
  - 0 if the relation is false (a is bigger then b)
  - 1 if the relation is true (a is smaller then b)
  - x if any of the operands has unknown x bits (if a or b contains X)
- These operators synthesize into comparators.
- ***Operators***

  < (less than)

  <= (less than or equal to)

  > (greater than)

  >= (greater than or equal to)

  == (equal to)

  != (not equal to)

Example

- //a=5 b=10,

  a <= b → 1

- // a=5 b=10

  a>= b➔ 0

- // a=x b=10

  – a<=b ➔ x

- // a=z b=10

  – a<=b ➔ x

- Equality Operators
- There are two types of Equality operators. Case Equality and Logical Equality.

Operator	Description
a === b	a equal to b, including x and z (Case equality)
a !== b	a not equal to b, including x and z (Case inequality)
a == b	a equal to b, result may be unknown (logical equality)
a != b	a not equal to b, result may be unknown (logical equality)

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- Result is 0 (false) or 1 (true)
- For the == and != operators, the result is x, if either operand contains an x or a z
- For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true. The result is always 0 or 1.

- 4'bx001 === 4'bx001 = 1
- 4'bx0x1 === 4'bx001 = 0
- 4'bz0x1 === 4'bz0x1 = 1
- 4'bz0x1 === 4'bz001 = 0
- 4'bx0x1 !== 4'bx001 = 1
- 4'bz0x1 !== 4'bz001 = 1
- 5 == 10 = 0
- 5 == 5 = 1
- 5 != 5 = 0
- 5 != 6 = 1

Logical Operators

- Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands.
- They can work on expressions, integers or groups of bits, and treat all values that are nonzero as “1”.
- Expressions connected by && and || are evaluated from left to right
  - The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has x (unknown) bits
- Logical operators are typically used in conditional (**if ... else)** statements since they work with expressions.
- ***Operators***
  - ! (logical NOT)
  - && (logical AND)
  - || (logical OR)

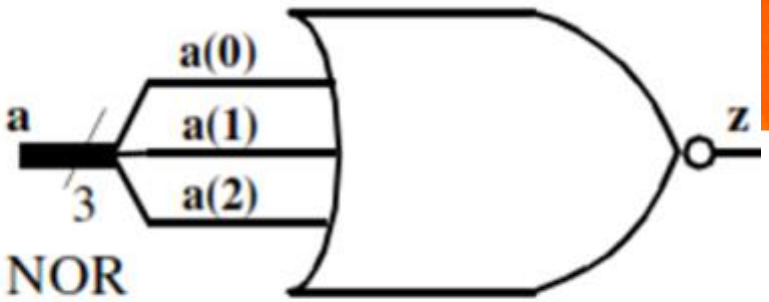
```
wire[7:0] x, y, z;          // x, y and z are multibit variables.
reg a;
. . .
if ((x == y) && (z)) a = 1; // a = 1 if x equals y, and z is nonzero.
   else  a = !x;            // a =0 if x is anything but zero.
```

- 1'b1 && 1'b1 = 1
- 1'b1 && 1'b0 = 0
- 1'b1 && 1'bx = x
- 1'b1 || 1'b0 = 1
- 1'b0 || 1'b0 = 0
- 1'b0 || 1'bx = x
- ! 1'b1 = 0
- ! 1'b0 = 1

Reduction Operators

- Reduction operators operate on all the bits of an operand vector and return a single-bit value.
- These are the unary (one argument) form of the bit-wise operators above.
- ***Operators***
  - & (reduction AND)
  - | (reduction OR)
  - ~& (reduction NAND)
  - ~| (reduction NOR)
  - ^ (reduction XOR)
  - ~^ or ^~(reduction XNOR)

```
module chk_zero (a, z);
  input [2:0] a;
  output z;
  assign z = ~| a; // Reduction NOR
endmodule
```



& 4'b1001 = 0

& 4'bx111 = x

~& 4'b1001 = 1

~& 4'bx001 = 1

~& 4'bz001 = 1

|4'b1001 = 1

| 4'bx000 = x

~| 4'b1001 = 0

~| 4'bx001 = 0

~| 4'bz001 = 0

^ 4'b1001 = 0

^ 4'bx001 = x

~^ 4'b1001 = 1

~^ 4'bx001 = x

Shift Operators

- Shift operators shift the first operand by the number of bits specified by the second operand.
- Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).
- ***Operators***
  - << (shift left)
  - >> (shift right)

```
assign c = a << 2;      /* c = a shifted left 2 bits;
                           vacant positions are filled with 0's */
```

- 4'b1001 << 1 = 0010
- 4'b10x1 << 1 =
- 4'b10z1 << 1 =
- 4'b1001 >> 1 = 0100
- 4'b10x1 >> 1 =
- 4'b10z1 >> 1 =

- 4'b1001 << 1 = 0010
- 4'b10x1 << 1 = 0x10
- 4'b10z1 << 1 = 0z10
- 4'b1001 >> 1 = 0100
- 4'b10x1 >> 1 = 010x
- 4'b10z1 >> 1 = 010z

Concatenation Operator

- The concatenation operator combines two or more operands to form a larger vector.
- ***Operators***
  - {}(concatenation)

```
wire [1:0] a, b;     wire [2:0] x;       wire [3;0] y, Z;
assign x = {1'b0, a}; // x[2]=0,  x[1]=a[1],  x[0]=a[0]
assign y = {a, b};   /* y[3]=a[1],  y[2]=a[0],  y[1]=b[1],
y[0]=b[0] */

assign {cout, y} = x + Z;  // Concatenation of a result
```

Replication Operator

- The replication operator makes multiple copies of an item.
- ***Operators***
  - {n{item}} (n fold replication of an item)

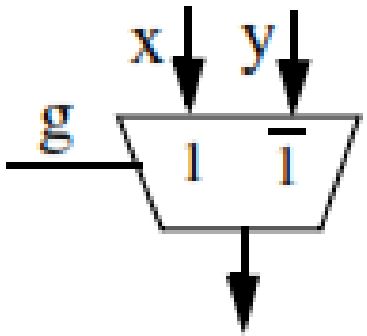
```
wire [1:0] a, b;     wire [4:0] x;
assign x = {2{1'b0}, a};  // Equivalent to  x = {0,0,a }
assign y = {2{a}, 3{b}};   //Equivalent to  y = {a,a,b,b}
```

Conditional Operator: "?"

- Conditional operator is like those in C/C++. They evaluate one of the two expressions based on a condition.
- It will synthesize to a multiplexer (MUX).
- ***Operators***
  - (cond) ? (result if cond true):

    (result if cond false)

```
assign a = (g) ? x : y;
assign a = (inc = = 2) ? a+1 : a-1;
        /* if (inc), a = a+1,  else a = a-1
```



5.10. Operator Precedence

- Table 5.1 shows the precedence of operators from highest to lowest.
- Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define order of precedence and improve the readability of your code.

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3'B101 and Y=3'B110, then X&Y=3'B100, X^Y=3'B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B101, 3'B110} = 6'B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
= =, !=	logical equality, logical inequality
= = =, != =	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&,	logical AND. Treat all variables as False (zero) or True (nonzero). logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
||	
? :	conditional. x=(cond)? T : F;

Table 5.1: Verilog Operators Precedence