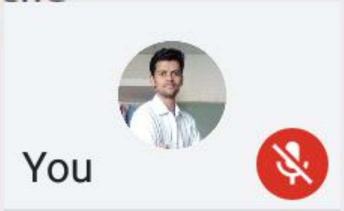
# Example problem for operators

If $A$ = 110, $B$ = 111, $C$ = 011000, and $D$ = 111011, the computation proceeds as follows:

```
C >> 2                          = 000110               (shift right 2
                                                        places)
C >> 2 & D                      = 000010               (bit-wise and)
~ B = 000                                               (bit-wise negate)
{A, ~ B}                        = 110000               (concatenation)
({A, ~ B}) | (C >> 2&D) = 110010                        (bit-wise or)
[({A, ~ B} | C >> 2) & D) == 110010] = TRUE             (the parentheses
                                                        force the equality)
                                                        test to be done
                                                        last and the
                                                        result is
```



The result of applying a relational operator is always a Boolean (FALSE or TRUE). Equals (==) and not equals (!=) can be applied to almost any type. The other relational operators can be applied to many numeric as well as to some array types. For example, if $A = 5$, $B = 4$, and $C = 3$, the expression `(A >= B) && (B <= C)` evaluates to FALSE. It is legal to use concatenate operator on the left side of the assignment, For example,

```
{Carry, Sum} = A + B;
```

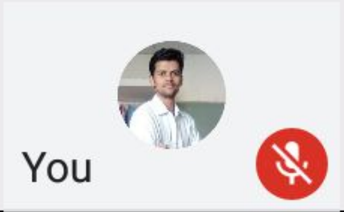
is legal. It adds A and B and the result goes into Sum and Carry. The most significant bit of the result is assigned to Carry. For the *logical equality* and *logical inequality* operators (== and !=), if, due to unknown or high-impedance bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (x). For the *case equality* and *case inequality* operators (=== and !==), bits that are x or z shall be included in the comparison and shall match for the result to be considered equal. The result of these operators shall always be a known value, either 1 or 0.

The shift operators can be applied to signed and unsigned registers. One can declare a register to be signed in the following manner:

```
reg signed [7:0] A = 8'hA5; //signed register A
```

In contrast, the register would have been unsigned if declared as follows:

```
reg [7:0] B = 8'hA5; //unsigned register B
```



If the register is unsigned, arithmetic and logic shifts do the same operation. The following example illustrates the difference between signed and unsigned shifts on signed and unsigned data.

```
reg signed [7:0] A = 8'hA5; // A is signed 1010 0101

A >> 4 is 00001010   (shift right unsigned by 4, filled with 0).
A >>> 4 is 11111010  (shift right signed by 4, filled with sign
                     bit).
A  << 4 is 01010000  (shift left unsigned, filled with 0).
A <<< 4 is 01010000  (shift   left   signed,   filled   with   0
                     irrespective of rightmost bit).

reg [7:0] B = 8'hA5; // B is unsigned 1010 0101

B >> 4 is 00001010     (shift right unsigned by 4, filled with 0)
B >>> 4 is 00001010    (shift right signed by 4, but B is unsigned,
                       filled with 0)
B  << 4 is 01010000    (shift left unsigned, filled with (
B <<< 4 is 01010000    (shift   left   signed,   filled   wi
                       irrespective of rightmost bit)
```

You

But if A is initialized to 8'shA5 as in

```
integer A = 8'shA5;
```

A >>> 4 yields 11111010 (shift right signed by 4, A's sign bit is 1). The 'sh indicates that the value is signed hex.

However, in **reg** declarations, if a signed register is desired, it should be explicitly mentioned. For instance,

```
reg [7:0] A = 8'shA5
```

does not make the register signed. It should be declared as **`reg signed [7:0] A = 8'hA5`**

The + and − operators can be applied to any types, including integer or real numeric operands. When types are mixed, the expression self-evaluates to a type according to the types of the operands. If a and b are 16 bits each, (a + b) will evaluate to 16 bits. However, (a + b + 0) will evaluate to integer. If any operand is real, the result is real. **If any operand is unsigned, the result is unsigned, regardless of the operator**.

When expressions are evaluated, if the operands are of unequal bit lengths and if one or both operands are unsigned, the smaller operand shall be zero-extended to the size of the larger operand. If both operands are signed, the smaller operand shall be sign-extended to the size of the larger operand. If constants need to be extended, signed constants are sign-extended and unsigned constants are zero-extended

The * and / operators perform multiplication and division on integer or flo point operands. The ** operator raises an integer or floating-point number to ar ger power. The % (modulus) operator calculates the remainder for integer oper

## Simple Synthesis Examples

- Synthesis tools try to infer the hardware components needed by “looking” at the Verilog code. In order for code to synthesize correctly, certain conventions must be followed.
- When writing Verilog code, you should always keep in mind that you are
-

  Designing hardware, not simply writing a computer program. Each Verilog state- ment implies certain hardware requirements. Consequently, poorly written Verilog code may result in poorly designed hardware. Even if Verilog code gives the correct result when simulated, it may not result in hardware that works correctly when syn- thesized. Timing problems may prevent the hardware from working properly even though the simulation results are correct.
- Consider the Verilog code in Figure. (Note that *B* is missing from the sen- sitivity list in the **always** statement.) This code will simulate as follows. Whenever *A* changes, it will cause the process to execute once. The value of *C* will reflect the values of *A* a
  the process began. If *B* changes now, that will not cause the process to execut

```
module Q1 (A, B, C);
input   A;
input   B;
output  C;

reg      C;

always @(A)
  C = #5 A | B;
endmodule
```

If this code is synthesized, most synthesizers will output an OR gate as in Figure 2-31. The synthesizer will warn you that B is missing from the sensitivity list in always statement, but will go ahead and synthesize the code properly. The synthesizer will also ignore the 5 ns delay on the preceding statement. If you want to model an exact 5 ns delay, you will have to use counters. The simulator output will not match the synthesizer's output since the always statement will not execute when B changes. This is an example of where the synthesizer guessed a little more than what you wrote; it assumed that you probably meant an OR gate and created that circuit (accompanied by a warning). But this circuit functions differently from what was simulated before synthesis. It is important that you always check for synthesizer warnings of missing signals in the sensitivity list. Perhaps the synthesizer helped you; perhaps it created hardware that you did not intend to.

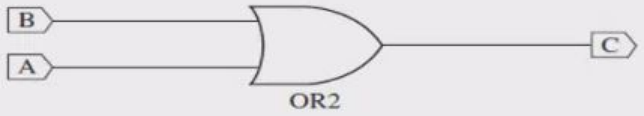
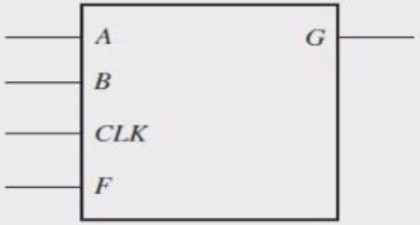


FIGURE 2-31: Synthesize Output for Code in Figure 2-30

```
module Q3 (A, B, F, CLK, G);
input   A;
input   B;
input   F;
input   CLK;
output  G;

reg     G;
reg     C;

always @(posedge CLK)
begin
  C <= A & B;  // statement 1
  G <= C | F;  // statement 2
end

endmodule
```

Figure 2-33. The ability to hide details and use abstractions is an important part of good system design.

FIGURE 2-33: Block Diagram for Verilog Code in Figure 2-32



Note that *C* is an internal signal, and therefore, it does not show up in the block diagram.

Now, let us think about the details of the circuit inside this block. This circuit is not two cascaded gates; the signal assignment statements are in a process (an always statement). An edge-triggered clock is implied by the use of **posedge** or **negedge** in the clock statement preceding the signal assignment. Since the values of *C* and *G* need to be retained after the clock edge, flip-flops are required for both *C* and *G*. Please note that a change in the value of *C* from statement 1 will not be considered during the execution of statement 2 in that pass of the process. It will be considered only in the next pass, and the flip-flop for *C* makes this happen in the hardware also. Hence the code implies hardware shown in Figure 2-34.

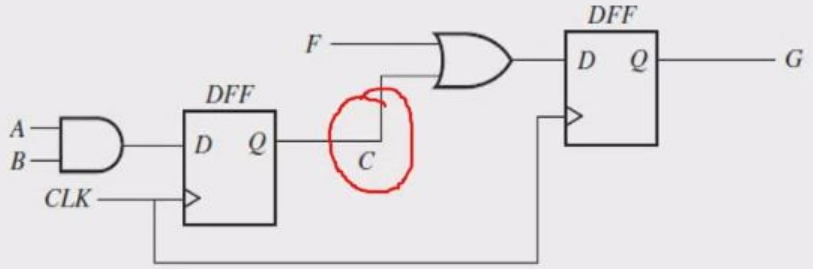


FIGURE 2-34: Hardware Corresponding to Verilog Code in Figure 2-32

We saw earlier that the following code represents a D-latch:

```
always @(G or D)
begin
 if (G)  Q <= D;
end
```

Let us understand why this code does not represent an AND gate with $G$ and $D$ as inputs. If $G = 1$, an AND gate will result in the correct output to match the **if** statement. However, what happens if currently $Q = 1$ and then $G$ changes to 0? When $G$ changes to 0', an AND gate would propagate that to the output; however, the device we have modeled here should not. It is expected to make no changes to the output if $G$ is not equal to 1. Hence, it is clear that this device has to be a D-latch and not an AND gate.

In order to infer flip-flops or registers that change state on the rising edge of a clock signal, most synthesizers require that the sensitivity list in an always statement should include an edge-triggered signal as in

```
always @(posedge CLK)
```

For every assignment statement in an always statement, a signal on the left side of the assignment will cause creation of a register or flip-flop. The moral to this story is that if you do not want to create unnecessary flip-flops, do not put the signal assignments in a clocked always statement. If `clock` is omitted in the sensitivity list of an always statement, the synthesizer may produce latches instead of flip-flops.

Now consider the Verilog code in Figure 2-35. If you attempt to synthesize this code, the synthesizer will generate an empty block diagram. This is because *D*, the output of the block shown in the Figure, is never assigned. The code assigns the new value to C, which is never brought out to the outside world. It will generate warnings that

```
Input <CLK> is never used.
Input <A> is never used.
Input <B> is never used.
Output <D> is never assigned.
```

FIGURE 2-35: Example Verilog Code That Will Not Synthesize

```
module no_syn (A, B, CLK, D);
input   A;
input   B;
input   CLK;
output  D;

reg     C;

always @(posedge CLK)
  C <= A & B;

endmodule
```