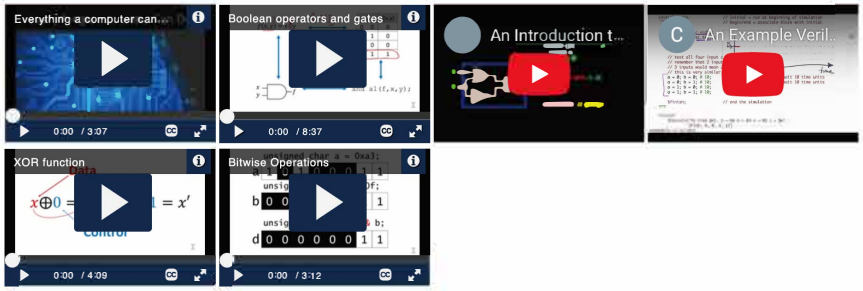
PRE01.1. 01 Text and Videos



Video credits: Geoffrey Herman and Craig Zilles, Text credits: Geoffrey Herman

The videos and text cover the same material from slightly different perspectives, so choose whichever helps you more or use one to help you clarify questions you had from the other.

Except for the "High Level Overview", all text and videos are also provided in the relevant, following questions. If you want, you may read and watch videos in smaller chunks in the later questions rather than reading/watching everything first and then start answering questions. This question primarily just aggregates everything into one spot for easier searching and referencing.

High Level Overview

The central organizing concept of computer hardware is that we store state (e.g., in RAM) and then manipulate that state (i.e., perform calculations) at regular intervals. We store state information as 1s and 0s, which then requires us to use Boolean functions to perform our state manipulations. In this class, you will need to be able to fluently translate between four representations of Boolean functions: Boolean algebraic expressions, truth tables, circuit diagrams, and Verilog hardware description language. These four representations are mathematically equivalent but help us perform different tasks more easily.

In programming languages like C, you will see Boolean operators used in two different ways: bitwise logical operations and logical operations. Bitwise logical operators (`&` (bitwise AND), `|` (bitwise OR), `~` (bitwise NOT), `^` bitwise XOR) perform logical operations on pairs of bits within a variable (e.g., `char x, y, z; z = x & y;`) . In contrast, logical operators (`&&` (AND), `||` (OR), `!` (NOT)) perform logical operations on entire bitstrings, treating 0 as `False` and everything that is not 0 as `True` (e.g., `if (x == 0 && y == 1)`). Bitwise logical operations are foundational computations that we can use to construct more complicated computations like addition and subtraction. Logical operations are generally used in program control structures like `if` statements or `for` loops. Note: When we say "logical operators," we do not include bitwise logical operators in that set in the same way that arithmetic operators like addition and subtraction would not be part of the set of logical operators. See below for some examples of how/when we might use these operators and how they work.

Details and Examples

Basic Boolean operators

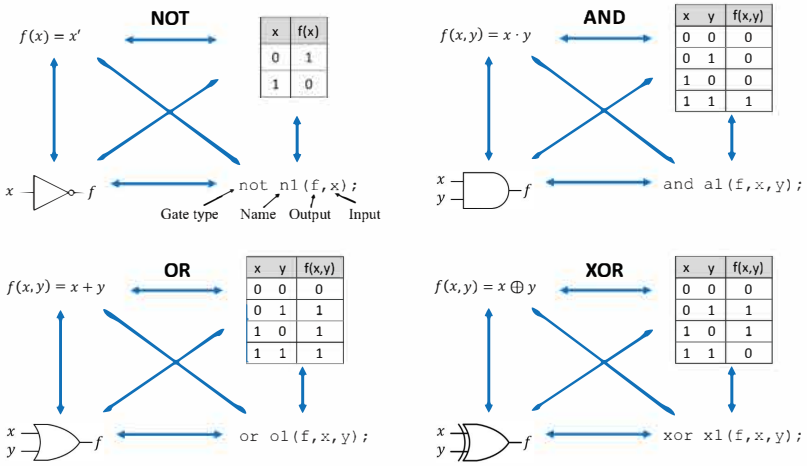


Figure 1: Boolean algebra, truth table, circuit diagram, and Verilog representations of the NOT, AND, OR, and XOR Boolean operators

All Boolean functions can be expressed with three operators: NOT, AND, and OR (we use all caps when using thes words as

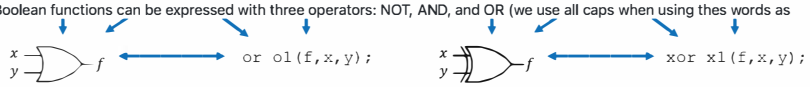


Figure 1: Boolean algebra, truth table, circuit diagram, and Verilog representations of the NOT, AND, OR, and XOR Boolean operators

All Boolean functions can be expressed with three operators: NOT, AND, and OR (we use all caps when using thes words as Boolean operators). Boolean operators are summarized in Figure 1 above and the list below.

- NOT
  - Truth table: When the input is `True` or 1, the output is `False` or 0. When the input is `False` or 0, the output is `True` or 1.
  - Algebra: We use the `'` operator after a variable (or sub-expression) for NOT.
  - Circuit: A triangle with input on the left and a circle on the right, the output side.
  - Verilog: `not name(out, in);` Note: May have only 1 input variable
- AND
  - Truth table: When both inputs are `True` or 1, the output is `True` or 1, otherwise the output is `False` or 0.
  - Algebra: We will use multiplication notation like `*` or adjacent variables with no operator.
  - Circuit: It is represented as a "D" shaped gate in circuit diagrams with inputs on the left and outputs on the right.
  - Verilog: `and name(out, in0, in1, ...);` Note: must have at least two input variables, may have more
- OR
  - Truth table: When any input is `True` or 1, the output is `True` or 1, otherwise the output is `False` or 0.
  - Algebra: We will use addition notation (i.e., `+`).
  - Circuit: It is represented as an "arrowhead" shaped gate in circuit diagrams with inputs on the left and outputs on the right.
  - Verilog: `or name(out, in0, in1, ...);` Note: must have at least two input variables, may have more

XOR

PRE 01

Assessment overview

Total points: 0/100

Score: 0%

Question PRE01.1

Value: 1

Total points: — /1

Auto-graded question

Previous question Next question

Personal Notes

No attached notes

Attach a file

Add text note

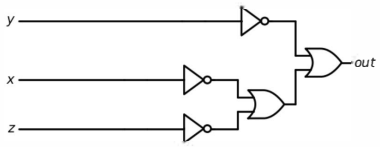
XOR

The XOR is a useful additional Boolean function, acting as an odd-even detection circuit (outputs 1 when the number of input bits is odd and 0 when even) or a controllable inverter circuit (for z = x XOR y, z = x when y = 0 and z = x' when y = 1). We can construct an XOR function from the basic Boolean operators ($x \oplus y = x' \cdot y + x \cdot y'$).

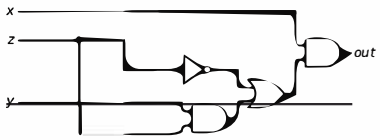
- XOR
  - Truth table: When an odd number of inputs are `True` or 1, the output is `True` or 1. When an even number of inputs are `True` or 1, the output is `False` or 0.
  - Algebra: We use the $\oplus$ operator for XOR.
  - Circuit: A triangle with input on the left and a circle on the right, the output side.
  - Verilog: `xor name(out, in1, in2, ...);` Note: must have at least two input variables, may have more

Converting a circuit diagram to a Boolean expression

To derive a Boolean expression, start at the inputs on the left and trace their values as they propogate to the right. Once all inputs to a gate have been evaluated, add that gate to the Boolean expression.

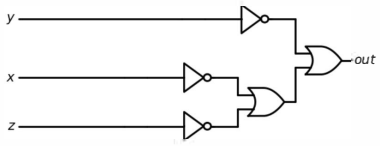


1. We have all inputs for all NOT gates, so we get `x'`, `y'`, and `z'`
2. We have all inputs to the bottom OR gate, so we get `x'+z'`
3. We have all inputs to the top OR gate, so we get `out = x'+y'+z'`

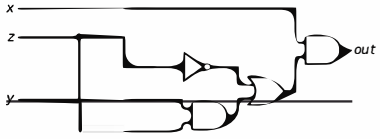


1. We have all inputs for the bottom AND gate and the NOT gate, so we separately have `z'` and `yz`
2. We have all inputs to the OR gate, so we get `z'+yz`
3. We have all inputs to the top AND gate, so we get `out = (z'+yz)x`.

Alternatively, you can start at the right and trace backwards on each gate's inputs.



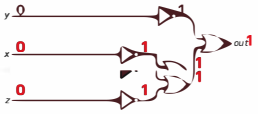
1. Start at the right-most OR gate `()+()`
2. Add the inputs of the OR gate `(()')+(()+())`
3. Go back one more level `((y)')+((()')+(()'))`
4. Go back one more level `((y)')+(((x)')+((z)'))`
5. Remove excess parentheses `out = y'+x'+z'`

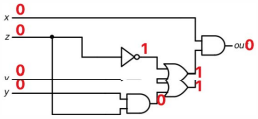


1. Start at the right-most AND gate `()*())`
2. Add the inputs of the AND gate `(x)*(()+())`
3. Go back one more level `(x)*((()')+(()*()))`
4. Go back one more level `(x)*(((z)')+((y)*(z))`
5. Remove excess parentheses `out = x(z'+yz)`

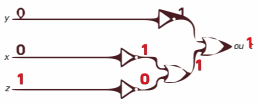
Converting a circuit diagram to a truth table

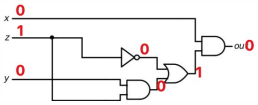
To derive a truth table, start with the values in the top row of the table (e.g., `x=0`, `y=0`, `z=0`) and substitute those into the circuit diagram. Then trace the values of 1s and 0s as they propogate through the circuit.





Repeat the process for all rows. We'll do just one more (`x=0`, `y=0`, `z=1`).





Converting a Boolean expression to a truth table

To derive a truth table, start with the values in the top row of the table (e.g., `x=0`, `y=0`, `z=0`) and substitute those into the Boolean expression. Then evaluate each of the operators in order of their precendence: parentheses `()`, NOT `'`, AND `*`, then OR `+`.

`f(x,y,z) = y'+x'+z'`

- `f(0,0,0) = 0'+0'+0'`
- `= 1+1+1`
- `= 1`

`f(x,y,z) = x(z'+y*z)`

- `f(0,0,0) = 0(0'+0*0)`
- `= 0(1+0)`
- `= 0(1)`
- `= 0`

Repeat the process for all rows. We'll do just one more.

- `f(0,0,1) = 0'+0'+1'`
- `= 1+1+0`
- `= 1`

- `f(0,0,1) = 0(1'+0*1)`
- `= 0(0+0)`
- `= 0(0)`
- `= 0`

Converting a Boolean expression to Verilog

Verilog is a **description language**, it only describes what components a circuit (called a `module`) has and how they are connected. **We recommend converting your Boolean expression to a circuit diagram before converting to Verilog.**. By convention, the output(s) of your Boolean function(s) should be listed first in your Verilog module declaration (i.e., `module circuit(out,...)`). Every output of your Boolean expression needs to be declared to be of type `output`. Similarly, the input(s) of your Boolean function(s) should be listed after the outputs in your Verilog module declaration (i.e., `module circuit(out1, out2, ..., in1, in2, ...)`. Every input needs to be declared to be of type `input`. Every wire in your circuit diagram will need to be declared to be of type `wire`. Every operator in your Boolean function or gate in your circuit diagram needs to have a corresponding gate in your Verilog module. You can use the following primitives for your logic gates.

```
wire   <id>, <id>, ...;
input  <id>, <id>, ...;
output <id>, <id>, ...;

and  <id>(out, in1, in2, ..., inN);   // N-input AND gate
or   <id>(out, in1, in2, ..., inN);   // N-input OR gate
xor  <id>(out, in1, in2, ..., inN);   // N-input XOR gate
nand <id>(out, in1, in2, ..., inN);   // N-input NAND gate (NOT AND)
nor  <id>(out, in1, in2, ..., inN);   // N-input NOR gate (NOT OR)
xnor <id>(out, in1, in2, ..., inN);   // N-input XNOR gate (NOT XOR)
not  <id>(out, in);                   // 1-input NOT gate
buf  <id>(out, in);                   // 1-input buffer: a "circuit" used to connect an input port to an out
```

Try naming your gates meaningful names like `nx` for a gate that **N**OTs **x**. Similarly, name your wires meaningful things like `a1_out` for the output of your first AND gate. The following verilog code provides a Verilog module for the Boolean expression `w = x'yz`

```
module example(w, x, y, z);          // new circuit definition
   output  w;                        // output of new circuit
   input   x, y, z;                  // input of new circuit
   wire    a1_out, a2_out, not_x;    // connecting wires

   not     nx(not_x, x);             // each circuit instance needs a unique name
   and     a1(a1_out, not_x, y, z);
   and     a2(a2_out, x, z);         // gate output first, inputs after
   or      o1(w, a1_out, a2_out);
endmodule // example done
```

A module can have multiple outputs if it comprises multiple Boolean expressions. For example, you could make a circuit comprised of `f = x'yz + xz` and `g = x'y + xy' + z`

```
module example(f, g, x, y, z);       // new circuit definition
   output  f, g;                     // outputs of new circuit
   input   x, y, z;                  // inputs of new circuit
   wire    a1_out, a2_out, a3_out, a4_out, not_x, not_y;

   // complemented variables
   not     nx(not_x, x);
   not     ny(not_y, y);

   // f output
   and     a1(a1_out, not_x, y, z);
   and     a2(a2_out, x, z);
   or      o1(f, a1_out, a2_out);

   // g output
   and     a3(a3_out, not_x, y);
   and     a4(a4_out, x, not_y);
   or      o2(g, a3_out, a4_out);

endmodule // example done
```

Converting Verilog to a Boolean expression

To convert Verilog code to a Boolean expression, start with the gate that produces the output value of the module and work back toward the inputs. The following examples use a substitution approach, using wire names as placeholders and then substituting the wire name with the gate operations that produced the value for the wire.

```
module example(w, x, y, z);   // new circuit definition
   output  w;                 // output of new circuit
   input   x, y, z;           // input of new circuit
   wire    a1_out, not_x;     // connecting wires

   not     n1(not_x, x);
   and     a1(a1_out, not_x, y); // circuit name(out, in1, in2);
   or      o1(w, a1_out, z);     // each circuit instance needs a unique name
endmodule // example done
```

Gate `o1` creates the output `w`: `w = z + (a1_out)`

Gate `a1` creates the signal for wire `a1_out`: `w = z + (not_x * y)`

Gate `n1` creates the signal for wire `not_x`: `w = z + x'y`

Because Verilog is a description language and not a programming language, you can re-arrange the order of gates without messing anything up. In hardware everything operates **in parallel**, so all gates are always producing their outputs based on their current inputs continuously. For example, the following Verilog would produce the same behavior. Note the declarations do need to happen first so the compiler knows what type each variable is.

```
module example(w, x, y, z);   // new circuit definition
   output  w;                 // output of new circuit
   input   x, y, z;           // input of new circuit
   wire    a1_out, not_x;     // connecting wires

   or      o1(w, a1_out, z);     // each circuit instance needs a unique name
   and     a1(a1_out, not_x, y); // circuit name(out, in1, in2);
   not     n1(not_x, x);
endmodule // example done
```

Here's one more example

```
module example2(w, x, y, z);              // new circuit definition
   output  w;                             // output of new circuit
   input   x, y, z;                       // input of new circuit
   wire    o1_out, o2_out, not_x, not_y;  // connecting wires


   not     nx(not_x, x);
   not     ny(not_y, y);
   or      o1(o1_out, not_x, y, z);  // each circuit instance needs a unique name
   or      o2(o2_out, not_y, z);     // primitive gates can have 2 or more inputs
   and     a1(w, o1_out, o2_out, x);
endmodule // example done
```

Gate `a1` creates the output `w`: `w = (o1_out)*(o2_out)*x`

Gate `o2` creates the signal for wire `o1_out`: `w = (not_y + z)*(o2_out)*x`

Gate `ny` creates the signal for wire `not_y`: `w = (y' + z)*(o2_out)*x`

Gate `o1` creates the signal for wire `o2_out`: `w = (y' + z)*(not_x + y + z)*x`

Gate `nx` creates the signal for wire `not_x`: `w = (y' + z)*(x' + y + z)*x`

Bitwise Logical Examples

Bitwise logical operators (a.k.a. Bitwise operators) treat variables as individual bits. Each individual bit of the output is a function of the two bits in the same bit position. For example, the two least-significant bits of 89 and 103 are both 1, so the least significant bit of the output is 1 & 1 = 1. In contrast, the next pair of least significant bits of 89 and 103 are 0 and 1 respectively, so 0 & 1 = 0.

0	1	0	1	1	0	0	1

&

0	1	1	0	0	1	1	1

=

0	1	0	0	0	0	0	1

Example 2, the two least-significant bits of 89 and 103 are both 1, so the least significant bit of the output is 1 | 1 = 1. The next pair of least significant bits of 89 and 103 are 0 and 1 respectively, so 0 | 1 = 1. The most-significant pair of bits are 0 and 0, so 0 | 0 = 0.

0	1	0	1	1	0	0	1

|

0	1	1	0	0	1	1	1

=

0	1	1	1	1	1	1	1

Logical Example

Logical operations treat variables holistically as Boolean values (i.e., a single true or false). Anything that is not 0 is `True`. For example, the numbers 89 and 103 are both treated as `True` because they are not 0. If a logical expression evaluates to `False`, it outputs 0 (i.e., 00000000). If a logical expression evaluates to `True`, it outputs 1 (i.e., 00000001). In the example below, the operator is actually performing `True` && `True`, which outputs `True` (00000001).

0	1	0	1	1	0	0	1

&&

0	1	1	0	0	1	0	1

=

0	0	0	0	0	0	0	1

Bitwise Logical in code

Bitwise logical operations can be used to make many calculations faster by taking advantage of how information is encoded in bits. For example, suppose we store the state of a number (i.e., declare a variable) as a `char` or `int` in C. As you will learn in this course, all odd `int` or `char` end with a 1 in the least-significant bits place and all even `int` or `char` end with a 0 in the least-significant bits place. Naively, we could use the modulus operation and a conditional statement (i.e., `in % 2 == 1`) to determine if a `int` or `char` is odd or even. You could then write the following C code to count the number of 1s in an `int` or `char`.

```
unsigned pop_count(unsigned in) {
  unsigned count = 0;
  for ( ; in != 0; in = in >> 1) {
    if (in % 2 == 1) {
      count ++;
    }
  }
  return count;
}
```

Unfortunately as you will learn about more later in the course, conditionals can be slow and the modulus operation is also slow. A faster version of this code would use bitwise logical operations like the following. The `& 1` is just a one-bit bitmask that adds 1 when the least-significant bit of `in` is 1 and 0 otherwise. If you don't fully understand why try bitwise 1110 and 1111 with 0001 and then 1100 1101 with 0001 and see what happens. Don't worry if you don't fully understand this idea yet, we will revisit bitwise logical operators again later as we learn more.

```
unsigned pop_count(unsigned in) {
  unsigned count = 0;
  for ( ; in != 0; in = in >> 1) {
    count += (in & 1);
  }
  return count;
}
```

Here is another example of how you can creatively encode information and use bitwise logical operations to achieve insane code performance improvements (start here for the juicy bits, but we recommend watching the whole thing)

Logical Operations in code

We almost exclusively use logical operators to create compound conditional statements like `if (x == 0 && y == 1)`. Yeah...that's about it!

Verilog Reference Guide

When we ask you to write Verilog code, we will provide you with access to the Verilog reference guide and Verilog style guide when appropriate. You can read it now if you want, though it's totally okay just to refer back to these when you need them.

▶ **Verilog Reference Guide**

▶ **Verilog Style Guide**

Supplemental Optional Reading: Mano & Kime 4th Ed. , 2.1

Mark as read

- [ ] (a) I've read this!