

Boolean Logic and Digital Design

The arithmetic operations performed by the CPU must be carried out using special electrical circuits called **logic circuits** that are used to implement boolean (or digital) logic in hardware. Here we investigate the design of such circuits, which is one small part of the broader area of computer architecture.

EARLY WORK

The foundations of circuit design were laid in the early 1900s by the English mathematician George Boole, after whom the C++ `bool` type is named. Boole formalized several axioms of logic, resulting in an algebra for writing logical expressions, which have since come to be known as boolean expressions.

In C++ syntax, some of the basic axioms of boolean logic are given in the following table. In the statements of these laws, the symbol \equiv denotes "is equivalent to." A statement of the form $p \equiv q$ means that p and q always have the same truth values (true or false).

The Relational Laws

- | | |
|---------------------------------|---------------------------------|
| 1a. $!(X == Y) \ \& \ (X != Y)$ | 1b. $!(X != Y) \ \& \ (X == Y)$ |
| 2a. $!(X < Y) \ \& \ (X >= Y)$ | 2b. $!(X >= Y) \ \& \ (X < Y)$ |
| 3a. $!(X > Y) \ \& \ (X <= Y)$ | 3b. $!(X <= Y) \ \& \ (X > Y)$ |

The Boolean Laws

- | | |
|---|---|
| 4a. $X \ \ \text{false} \ \& \ X$ | 4b. $X \ \&\& \ \text{false} \ \& \ \text{false}$ |
| 5a. $X \ \ \text{true} \ \& \ \text{true}$ | 5b. $X \ \&\& \ \text{true} \ \& \ X$ |

Idempotent Laws

- | | |
|---------------------------|-----------------------------|
| 6a. $X \ \ X \ \& \ X$ | 6b. $X \ \&\& \ X \ \& \ X$ |
|---------------------------|-----------------------------|
-

Involution Law

7a. $\!(\!X) \ \& \ X$

Laws of Complementarity

8a. $X \ || \ (\!X) \ \& \ \text{true}$

8b. $X \ \&\& \ (\!X) \ \& \ \text{false}$

Commutative Laws

9a. $X \ || \ Y \ \& \ Y \ || \ X$

9b. $X \ \&\& \ Y \ \& \ Y \ \&\& \ X$

Associative Laws

10a. $(X \ || \ Y) \ || \ Z \ \& \ X \ || \ (Y \ || \ Z)$

10b. $(X \ \&\& \ Y) \ \&\& \ Z \ \& \ X \ \&\& \ (Y \ \&\& \ Z)$

Distributive Laws

11a. $X \ \&\& \ (Y \ || \ Z) \ \& \ (X \ \&\& \ Y) \ || \ (X \ \&\& \ Z)$

11b. $X \ || \ (Y \ \&\& \ Z) \ \& \ (X \ || \ Y) \ \&\& \ (X \ || \ Z)$

Simplification Theorems

12a. $(X \ \&\& \ Y) \ || \ (X \ \&\& \ !Y) \ \& \ X$

12b. $(X \ || \ Y) \ \&\& \ (X \ || \ !Y) \ \& \ X$

13a. $X \ || \ (X \ \&\& \ Y) \ \& \ X$

13b. $X \ \&\& \ (X \ || \ Y) \ \& \ X$

14a. $(X \ || \ !Y) \ \&\& \ Y \ \& \ X \ \&\& \ Y$

14b. $(X \ \&\& \ !Y) \ || \ Y \ \& \ X \ || \ Y$

DeMorgan's Laws

15a. $\!(X \ \&\& \ Y) \ \& \ !X \ || \ !Y$

15b. $\!(X \ || \ Y) \ \& \ !X \ \&\& \ !Y$

It is especially useful for programmers to know DeMorgan's Laws because they can simplify complicated boolean expressions. As a simple illustration, suppose that `done` and `error` are `bool` objects, and consider the following `if` statement:

```
if (!done && !error)
    // ... do something...
```

DeMorgan's law tells us that the boolean expression involving two negated values,

```
!done && !error
```

can be simplified to

$$\!(\text{done} \ || \ \text{error})$$

The original expression contained 2 NOT operations and 1 AND operation, but the simplified expression contains only 1 NOT operation and 1 OR operation—1 less operation. Applying DeMorgan's law repeatedly to a boolean expression of the form

$$\!b_1 \ \&\& \ !b_2 \ \&\& \ \dots \ \&\& \ !b_n$$

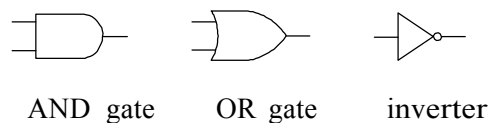
containing n NOTs and $n - 1$ ANDs, gives the simpler expression

$$\!(b_1 \ || \ b_2 \ || \ \dots \ || \ b_n)$$

containing only 1 NOT and $n - 1$ ORs. The complexity of the expression is thus reduced by $n - 1$ NOT operations, which can result in a significant increase in performance.

DIGITAL CIRCUITS

With the invention of the digital computer in the late 1930s, the work of Boole moved from obscurity to prominence. The axioms and theorems of his boolean algebra became extremely important as mathematicians, engineers, and physicists sought to build the arithmetic and logic circuitry of the early computers. These circuits utilize three basic electronic components: the **AND gate**, the **OR gate**, and the **NOT gate** or **inverter**, whose symbols are as follows:



The inputs to these gates are electrical voltages, where a voltage that exceeds a certain threshold value is interpreted as 1 (i.e., true), and a voltage below that threshold is interpreted as 0 (i.e., false). In the case of an AND gate, a 1 is produced only when there are 1s on both input lines. An OR gate produces a 1 only when there is a 1 on at least one of the input lines. The output of a NOT gate is the opposite of its input. Because these three components behave in the same fashion as the AND, OR, and NOT operators from boolean algebra, a circuit can be constructed to represent any boolean expression, and boolean expressions can be used to design circuits!

CIRCUIT DESIGN: A BINARY HALF-ADDER

To illustrate, consider the problem of adding two binary digits `digit1` and `digit2`. The truth table below summarizes the behavior of the addition operation, which produces two results—a `sum` bit and a `carry` bit:

digit1	digit2	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

There are two important things to note:

1. The `carry` output is 1 (true) only when `digit1` and `digit2` are both 1 (true)
2. The `sum` output is 1 (true) only when `digit1` is 0 (false) and `digit2` is 1 (true), or when `digit1` is 1 (true) and `digit2` is 0 (false)

It is easy to see that we can represent these outputs by the following pair of boolean expressions:

```
bool carry = digit1 && digit2,
    sum = (!digit1 && digit2) || (digit1 && !digit2);
```

The expression for `sum` has the form $(\neg A \ \&\& \ B) \ || \ (A \ \&\& \ \neg B)$ and can be simplified by applying the axioms from boolean logic as follows:

$$\begin{aligned}
 & \underline{(\neg A \ \&\& \ B)} \ || \ \underline{(A \ \&\& \ \neg B)} \\
 & \quad \downarrow \quad \text{(Apply 9a to switch two operands of } || \text{)} \\
 & \underline{(A \ \&\& \ \neg B)} \ || \ \underline{(\neg A \ \&\& \ B)} \\
 & \quad \downarrow \quad \text{(Apply 11b with } X = (A \ \&\& \ \neg B), Y = \neg A, Z = B \text{)} \\
 & ((A \ \&\& \ \neg B) \ || \ \neg A) \ \&\& \ ((A \ \&\& \ \neg B) \ || \ B) \\
 & \quad \downarrow \quad \text{(Apply 14b to second expression with } X = A \text{ and } Y = B \text{)} \\
 & ((A \ \&\& \ \neg B) \ || \ \neg A) \ \&\& \ (A \ || \ B) \\
 & \quad \downarrow \quad \text{(Apply 9a to switch two operands of first } \&\& \text{)} \\
 & ((\neg B \ \&\& \ A) \ || \ \neg A) \ \&\& \ (A \ || \ B) \\
 & \quad \downarrow \quad \text{(Apply 14b to first expression with } X = \neg B \text{ and } Y = \neg A \text{)} \\
 & (\neg B \ || \ \neg A) \ \&\& \ (A \ || \ B) \\
 & \quad \downarrow \quad \text{(Apply 15a to first } || \text{ expression with } X = B \text{ and } Y = A \text{)} \\
 & \neg(B \ \&\& \ A) \ \&\& \ (A \ || \ B) \\
 & \quad \downarrow \quad \text{(Apply 9a to switch two operands of first } \&\& \text{)} \\
 & \underline{\neg(A \ \&\& \ B)} \ \&\& \ \underline{(A \ || \ B)} \\
 & \quad \downarrow \quad \text{(Apply 9a to switch two operands of second } \&\& \text{)} \\
 & (A \ || \ B) \ \&\& \ \neg(A \ \&\& \ B)
 \end{aligned}$$

This means that the boolean expression for `sum` can be rewritten as

```
sum = (digit1 || digit2) && !(digit1 && digit2);
```

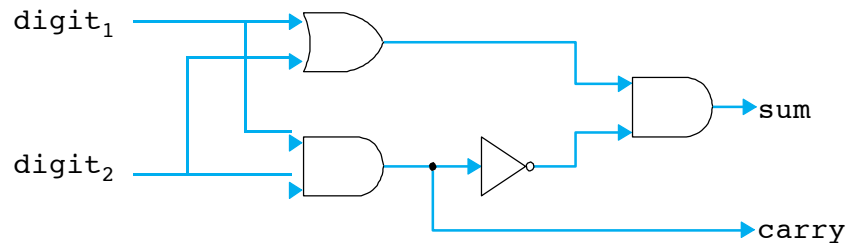
which has one less NOT operation than the original expression.

This may seem like a lot of work for not much improvement. On the contrary, this simplification means that a circuit for this expression will require one less inverter than a circuit for the original expression and will therefore be less expensive to manufacture. If half-adders are mass-produced, then this circuit may be manufactured millions of times with a savings that is millions of times the cost of an inverter!

Using the boolean expressions

```
bool carry = digit1 && digit2,
```

for `sum` and `carry`, we can design the following circuit, called a **binary half-adder**, that adds two binary digits:



It accepts two inputs, `digit1` and `digit2`, and produces two outputs, `sum` and `carry`.

As demonstrated in Example 5.3 in the textbook, once a boolean expression is found to represent a circuit, it is easy to write a simple program to check its behavior.