7.2 Gates, truth tables, and logic equations

(Original section[1])

The electronics inside a modern computer are digital. Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values. (As we discuss later in this section, a possible pitfall in digital design is sampling a signal when it not clearly either high or low.) The fact that computers are digital is also a key reason they use binary numbers, since a binary system matches the underlying abstraction inherent in the electronics. In various logic families, the values and relationships between the two voltage values differ. Thus, rather than refer to the voltage levels, we talk about signals that are (logically) true, or 1, or are *asserted*; or signals that are (logically) false, or 0, or are *deasserted*. The values 0 and 1 are called complements or inverses of one another.

Asserted signal: A signal that is (logically) true, or 1.

Deasserted signal: A signal that is (logically) false, or 0.

Logic blocks are categorized as one of two types, depending on whether they contain memory. Blocks without memory are called combinational; the output of a combinational block depends only on the current input. In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the *state* of the logic block. In this section and the next, we will focus only on *combinational logic*. After introducing different memory elements in COD Sections A.8 (Memory elements: Flip-flops, latches, and registers), we will describe how *sequential logic*, which is logic including state, is designed.

Combinational logic: A logic system whose blocks do not contain memory and hence compute the same output given the same input.

Sequential logic: A group of logic elements that contain memory and hence whose value depends on the input as well as the current contents of the memory.

Truth tables

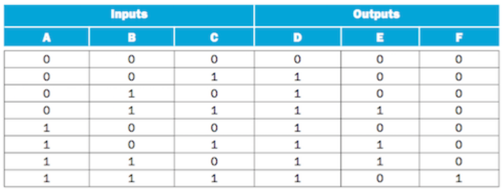
Because a combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally given as a *truth table*. For a logic block with *n* inputs, there are 2^n entries in the truth table, since there are that many possible combinations of input values. Each entry specifies the value of all the outputs for that particular input combination.

Example 7.2.1: Truth tables.

Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows: *D* is true if at least one input is true, *E* is true if exactly two inputs are true, and *F* is true only if all three inputs are true. Show the truth table for this function.

Answer

The truth table will contain 2^3 = 8 entries. Here it is:



Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Truth tables can completely describe any combinational logic function; however, they grow in size quickly and may not be easy to understand. Sometimes we want to construct a logic function that will be 0 for many input combinations, and we use a shorthand of specifying only the truth table entries for the nonzero outputs. This approach is used in COD Chapter 4 (The processor) and COD Appendix C (Mapping control to hardware).

Boolean algebra

Another approach is to express the logic function with logic equations. This is done with the use of *Boolean algebra* (named after Boole, a 19th-century mathematician). In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator is written as **+**, as in $A + B$. The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a *logical sum*, since its result is 1 if either operand is 1.
- The AND operator is written as ·, as in $A \cdot B$. The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called *logical product*, since its result is 1 only if both operands are 1.
- The unary operator NOT is written as $\bar{A}$. The result of a NOT operator is 1 only if the input is 0. Applying the operator NOT to a logical value results in an inversion or negation of the value (i.e., if the input is 0 the output is 1, and vice versa).

There are several laws of Boolean algebra that are helpful in manipulating logic equations.

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$
- Inverse laws: $A + \bar{A} = 1$ and $A \cdot \bar{A} = 0$
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$

In addition, there are two other useful theorems, called DeMorgan's laws, that are discussed in more depth in the exercises.

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

Example 7.2.2: Logic equations.

Show the logic equations for the logic functions, *D*, *E*, and *F*, described in the previous example.

Answer

Here's the equation for *D*:

$$D = A + B + C$$

F is equally simple:

$$F = A \cdot B \cdot C$$

E is a little tricky. Think of it in two parts: what must be true for *E* to be true (two of the three inputs must be true), and what cannot be true (all three cannot be true). Thus we can write *E* as

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

We can also derive *E* by realizing that *E* is true only if exactly two of the inputs are true. Then we can write *E* as an OR of the three possible terms that have two true inputs and one false input:

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

Proving that these two expressions are equivalent is explored in the exercises.

In Verilog, we describe combinational logic whenever possible using the assign statement, which is described in COD Sections A.4 (Using a hardware description language). We can write a definition for *E* using the Verilog exclusive-OR operator as `assign E = (A ^ B ^ C) * (A + B + C) * (A * B * C)`, which is yet another way to describe this function. D and F have even simpler representations, which are just like the corresponding C code: `D = A | B | C` and `F = A & B & C`.

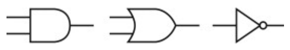
Gates

Logic blocks are built from *gates* that implement basic logic functions. For example, an AND gate implements the AND function, and an OR gate implements the OR function. Since both AND and OR are commutative and associative, an AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs. The logical function NOT is implemented with an inverter that always has a single input. The standard representation of these three logic building blocks is shown in the figure below.

Gate: A device that implements basic logic functions, such as AND or OR.

Figure 7.2.1: Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right (COD Figure A.2.1).

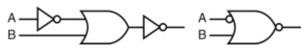
The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.



Rather than draw inverters explicitly, a common practice is to add "bubbles" to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted. For example, the figure below shows the logic diagram for the function $\overline{\overline{A} + B}$, using explicit inverters on the left and bubbled inputs and outputs on the right.

Figure 7.2.2: Logic gate implementation using explicit inverts on the left and bubbled inputs and outputs on the right. (COD Figure A.2.2).

Logic gate implementation of $\overline{\overline{A} + B}$. This logic function can be simplified to $A \cdot \overline{B}$ or in Verilog, `A & ~ B`.



Any logical function can be constructed using AND gates, OR gates, and inversion; several of the exercises give you the opportunity to try implementing some common logic functions with gates. In the next section, we'll see how an implementation of any logic function can be constructed using this knowledge.

In fact, all logic functions can be constructed with only a single gate type, if that gate is inverting. The two common inverting gates are called *NOR* and *NAND* and correspond to inverted OR and AND gates, respectively. NOR and NAND gates are called ***universal***, since any logic function can be built using this one gate type. The exercises explore this concept further.

NOR gate: An inverted OR gate.

NAND gate: An inverted AND gate.

PARTICIPATION ACTIVITY 7.2.1: Check yourself: Logical expression equivalence.

Given the following logical expressions:

$G = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
$H = B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$

1) Are the two logical expressions equivalent?

- ○ Yes
- ○ No

2) Which setting of the variables show that equation G and H are not equivalent.

- ○ A = 1, C = 1, B = 0
- ○ A = 0, B = 0, C = 0

(*1) This section is in original form.

Provide feedback on this section