Unit BF

Boolean Functions and Computer Arithmetic

Section 1: Boolean Functions

We recall the concept of a function and some of the terminology.

Definition 1 (Function) *If A and B are sets, a function from A to B is a rule that tells us how to find a unique $b \in B$ for each $a \in A$. We write $f(a) = b$ and say that f maps a to b. We also say the value of f at a is b.*

We write $f : A \to B$ to indicate that f is a function from A to B. We call the set A the domain of f and the set B the range or, equivalently, codomain of f.

To specify a function completely you must give its domain, range and rule.

In this section, we'll study a special class of functions called "boolean functions." General properties of functions are studied in Unit SF.

A *Boolean function* is a function f from the Cartesian product $\times^n\{0,1\}$ to $\{0,1\}$. Alternatively, we write $f : \times^n\{0,1\} \to \{0,1\}$. The set $\times^n\{0,1\}$, by definition, the set of all n-tuples $(x_1, \cdots, x_n)$ where each x_i is either 0 or 1, is called the *domain* of f. The set $\{0,1\}$ is called the *codomain* (or, sometimes, *range*) of f. The Cartesian product $\times^n\{0,1\}$ is also written $\{0,1\}^n$. This corresponds to writing the product of n copies of y as y^n.

Example 1 (Tabular representation of Boolean functions) One way to represent a function whose domain is finite is with a table. Each element x of the domain has a row of the table listing the domain element x and the corresponding function value $f(x)$. For example, the two tables

p	q	f
0	0	0
0	1	1
1	0	0
1	1	1

p	q	r	g
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

define Boolean functions $f : \times^2\{0,1\} \to \{0,1\}$ and $g : \times^3\{0,1\} \to \{0,1\}$. In the case of f, the first two values of each row represent the argument of f and the third entry in the same row represents the value of f at that argument. We have $f(0,0) = 0$, $f(0,1) = 1$, $f(1,0) = 0$, and $f(1,1) = 1$. For g, the first three values of each row represent an element of $\times^3\{0,1\}$ (a triple (p,q,r) of 0's or 1's) and the fourth entry of the row represents the value of $g(p,q,r)$. Thus, $g(0,0,0) = 1$, $g(0,0,1) = 1$, $g(0,1,0) = 0$, etc.

Notice that we have used p, q and r for variables instead of x, y and z, which are usually the choice for variable names in algebra. We can choose any names we please for variables. Names such as p, q and r are commonly used in the study of Boolean functions. $\square$

The tabular form of a Boolean function is often called a *truth table* because of the connection between Boolean functions and logic, which we will study later.

Example 2 (The number of Boolean functions) How many Boolean functions are there with domain $\times^3\{0,1\}$? More generally, how many Boolean functions are there with domain $\times^n\{0,1\}$? We do this in two steps.

First, how many elements are there in the domain $\times^n\{0,1\}$? We use the notation $|S|$ to denote the number of elements of S (also called the cardinality of S). Thus we are asking for the value of $|\times^n\{0,1\}|$. We can select an element of the domain by making n choices $x_1, x_2, \ldots, x_n$ where $x_i \in \{0,1\}$ for $i = 1, 2, \ldots, n$. Thus there are two choices for x_1 and two choices for x_2 and so on. The total number of elements in the domain is thus $2 \times 2 \times \cdots \times 2 = 2^n$. In other notation, $|\times^n\{0,1\}| = |\{0,1\}|^n$. If you have trouble seeing why this is true, the footnote may help.[1]

Second, we must construct the Boolean function. In constructing a Boolean function $h : \times^3\{0,1\} \to \{0,1\}$, there are two possible choices that we could assign to $h(p,q,r)$ for each $(p,q,r) \in \times^3\{0,1\}$. We saw in the previous paragraph that there are 2^3 elements (p,q,r) in $\times^3\{0,1\}$. Thus the total number of Boolean functions $h : \times^3\{0,1\} \to \{0,1\}$ is $2^{(2^3)}$. In general, the number of Boolean functions $h : \times^n\{0,1\} \to \{0,1\}$ is $2^{(2^n)}$. $\square$

In this section we are concerned about a particular way of representing Boolean functions in terms of certain more "primitive" representations. An example that is familiar from high school mathematics is the representation of polynomial and rational functions, such as $1+x$, $1+x^2$, $(2+x^2-x^3)/(1+x^2)$, starting with the constant functions c and the identity function x. All of these functions are created by adding, multiplying, subtracting, and/or dividing the simple starting functions.

Example 3 (The simplest Boolean functions: constants, identity, not) The simplest functions are the constant functions. Since we have only two constants, 0 and 1, there are two constant Boolean functions.

The next simplest Boolean functions are those that depend only only a single variable, but are not constant. Since there are $2^{2^1} = 2^2 = 4$ one-variable Boolean functions and since there are two constants, there are $4 - 2 = 2$ nonconstant, one-variable Boolean functions. One is the identity function $f(p) = p$ for all $p \in \{0,1\}$. The other one-variable function is "**not** p." For the moment, let's call this function $n(p)$. The definition is simple: $n(p) = 0$ if $p = 1$, $n(p) = 1$ if $p = 0$. This function is usually denoted by $\sim p$ rather than $n(p)$. Thus,

[1] How many elements are there in the Cartesian product $S \times T$ of a set with s elements and a set with t? Imagine an s by t array. Can you see why each entry in the array can be thought of as an element of $S \times T$? If so, we are almost done. We have shown that $|S \times T| = |S| \cdot |T|$. Apply this over and over again to $\{0,1\} \times \{0,1\} \times \cdots \times \{0,1\}$.

we would write $\sim 0 = 1$ and $\sim 1 = 0$. The symbol "$\sim$" is called the "unary operator **not**." *Unary* means it is a function that has one variable. The minus sign in ordinary arithmetic is a familiar example of a unary operator. $\square$

Example 4 (Two-variable Boolean functions: and, or, exculsive or) We know there are $2^{2^2} = 2^4 = 16$ two-variable Boolean functions $f(p, q)$. From the previous example, 2 of these are constant, 2 of them are not constant and depend only on p, and 2 of them are not constant and depend only on q. This leaves $16 - 6 = 10$ functions that depend on both variables. We certainly don't want to give names to all of them! In this example, we define three commonly-used, two-variable Boolean functions.

The first function we define is "p **and** q." Again, for the moment, let's call this function (a function of two Boolean variables) $a(p, q)$. By definition, $a(p, q) = 0$ unless both $p = 1$ and $q = 1$, in which case $a(p, q) = 1$. The function $a(p, q)$ is denoted by $p \wedge q$. We write $0 \wedge 0 = 0$, $0 \wedge 1 = 0$, $1 \wedge 0 = 0$, and $1 \wedge 1 = 1$.

The next function is "p **or** q." Again, for the moment, let call this function (a function of two Boolean variables) $o(p, q)$. By definition, $o(p, q) = 1$ unless both $p = 0$ and $q = 0$, in which case $o(p, q) = 0$. The function $o(p, q)$ is denoted by $p \vee q$. We write $0 \vee 0 = 0$, $0 \vee 1 = 1$, $1 \vee 0 = 1$, and $1 \vee 1 = 1$.

The last function is the *exclusive or* function. It is written $\textbf{xor}(p, q)$ or $p \oplus q$. By definition $p \oplus q = 0$ if $p = q$ and $p \oplus q = 1$ if $p \neq q$.

Here are our functions in tabular form:

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

The symbols $\vee$, $\wedge$, and $\oplus$ are called "binary operators" because they involve two variables (like addition and multiplication in ordinary arithmetic). $\square$

Example 5 (Boolean functions and logic) People often call Boolean variables such as p and q *statement variables.* Why? Because Boolean functions are related to logic. In logic we think of p and q as statements, 0 as "false," and 1 as "true."

Suppose p stands for "I have classes tomorrow" and q stands for "I will stay home tomorrow." Let's look at our basic Boolean functions.

- $\sim p$ stands for "**not** (I have classes tomorrow)," which can be written in more normal English as "I do not have classes tomorrow." As mentioned earlier, 0 is thought of as "false" and 1 is thought of as "true." According to our definition of the function $\sim$, if p is true, then $\sim p$ is false. This is also true about our statements: If the statement "I have classes tomorrow" is true, then the statement "I do not have classes tomorrow" is false.
- $p \wedge q$ stands for the statement, "I have classes tomorrow and I will stay home tomorrow." You should verify that the definition of $p \wedge q$ agrees with our usual interpretation of

"and": $p \wedge q$ is true if and only if both p and q are true. The statement $p \wedge q$ is also read "p but q", especially if q is surprising as in "I have classes tomorrow, but I will stay home tomorrow."

- $p \vee q$ stands for the statement, "Either I have classes tomorrow or I will stay home tomorrow." Unfortunately "or" is ambiguous in English. If I have classes *and also* stay home, some people may think the statement is not true. Others may think that it is true. Our definition of **or** is *not* ambiguous: $p \vee q$ is true if either p or q or both are true.
- $p \oplus q$ stands for the statement "Either I have classes tomorrow or I will stay home tomorrow, but not both." Why is this? The value of $p \oplus q$ is 1 (true) if and only if one of p and q is 1 (true) and the other is 0 (false).

When converting ordinary language into symbolic form, it is important to be aware of ambiguities so that the statements can be converted correctly. What we have been discussing is called *propositional logic*. We will explore the connection between Boolean functions and logic in the unit on logic. $\square$

Because of the close connection with logic, the tabular form of a Boolean function is often called a *truth table*.

Why are the Boolean functions represented by $\sim$, $\wedge$, $\vee$ and $\oplus$ important? In the previous example, we have seen a hint of their importance in logic. They are also important because they can be used to define more complex Boolean functions, in the same way that the basic operations of arithmetic can be used to define more complex algebraic functions.

Example 6 (Defining more complex Boolean functions) We could try to define a Boolean function using $\sim$, $\wedge$, and $\vee$ by stating that

$$f(p,q) = p \wedge {\sim}q \vee {\sim}p \wedge q. \qquad \text{(ambiguous)}$$

The problem with this "definition" is that it is not clear what the order of application of these operators should be. It is conventional to give top priority to the $\sim$ operator. Thus the expression used to define f can be clarified a bit: $f(p,q) = p \wedge ({\sim}q) \vee ({\sim}p) \wedge q$. In addition, the order in which the $\wedge$ and $\vee$ are performed must be specified by grouping them in the definition of the function f. Thus, as an example, we could group them

$$f(p,q) = (p \wedge ({\sim}q)) \vee (({\sim}p) \wedge q). \qquad \text{(not ambiguous)}$$

Now the function f is clearly defined. You should make sure that you use enough parentheses. Using too few may result in an ambiguous function definition. Using more than needed does not change a function. For example, the usage of $\sim$ is defined by the precedence rules, so we could just write

$$f(p,q) = (p \wedge {\sim}q) \vee ({\sim}p \wedge q). \qquad \text{(still not ambiguous)}$$

If you're unsure about precedence rules, be safe and use extra parentheses!

Using the formula for f we can compute values and give f in tabular form:

p	q	f
0	0	0
0	1	1
1	0	1
1	1	0

This is the exclusive or function. We have proved $p \oplus q = (p \wedge \sim q) \vee (\sim p \wedge q)$. $\square$

In the previous example, we found that $p \oplus q$ could be written as an equivalent Boolean function using $\sim$, $\wedge$ and $\vee$. This is a particular example of a much more general result:

Theorem 1 (Representing functions) *Suppose $n > 0$ and $f : \times^n\{0,1\} \to \{0,1\}$. There is a function g using only $\sim$, $\vee$ and $\wedge$ that is equal to f. In fact, we can use just $\sim$ and $\wedge$ or, if we prefer, we can use just $\sim$ and $\vee$.*

Proof: We'll illustrate how to do this with the function

p	q	r	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Look at the rows in the table where the $f = 1$. The first such is $(p, q, r) = (0, 0, 1)$. Note that $(\sim p) \wedge ((\sim q) \wedge r)$ equals 1 when $(p, q, r) = (0, 0, 1)$ and equals 0 otherwise.[2] Similarly, for the rows $(p, q, r) = (0, 1, 0)$ and $(p, q, r) = (1, 1, 1)$, we have $(\sim p) \wedge (q \wedge (\sim r))$ and $p \wedge (q \wedge r)$. You should be able to see why the function

$$g(p, q, r) = \Big((\sim p) \wedge ((\sim q) \wedge r)\Big) \vee \Big((\sim p) \wedge (q \wedge (\sim r))\Big) \vee \Big(p \wedge (q \wedge r)\Big)$$

is equal to f.

It should be clear how to do this in general: Construct an appropriate "anding" for each row of the table where the function equals 1. Then "or" all these andings together. This proves the first claim.

In this paragraph we take a break to discuss some terminology. An expression of this sort, namely an "or" of "ands" of variables and their negations, is called *disjunctive normal form*. Where did "disjunctive" come from? An "or" is sometimes called a *disjunction*. If we

[2] We don't need all these parentheses. The value of $\sim p \wedge \sim q \wedge r$ is unambiguous. We're just playing it safe!

replace the roles of "and" and "or," we obtain *conjunctive normal form*, so called because "and" is called a *conjunction*. End of terminology discussion and back to the proof.

One might object that we haven't taken care of the case when the function is a contradiction since then there are no rows where the function is 1. This is taken care of by 0 or, if you prefer, by $p \wedge \sim p$.

How can we get rid of $\vee$? We claim that

$$P \vee Q \vee R \vee \cdots \vee T \quad \text{and} \quad \sim\Big((\sim P) \wedge (\sim Q) \wedge (\sim R) \wedge \cdots \wedge (\sim T)\Big)$$

are equal. Why? The only way the first expression can be 0 is for *all* of $P, Q, \ldots, T$ to be 0. The only way the second expression can be 0 is for the expression inside the large parentheses to be 1. The only way this can happen is for *all* of $\sim P, \sim Q, \ldots, \sim T$ to be 1 — which is the same as *all* of $P, Q, \ldots, T$ being 0. Applying this to g with

$$P = (\sim p) \wedge ((\sim q) \wedge r), \quad Q = (\sim p) \wedge (q \wedge (\sim r)), \quad R = p \wedge (q \wedge r),$$

and none of the terms $\ldots, T$, we have the equivalent function

$$\sim\bigg(\sim\Big((\sim p) \wedge ((\sim q) \wedge r)\Big) \wedge \sim\Big((\sim p) \wedge (q \wedge (\sim r))\Big) \wedge \sim\Big(p \wedge (q \wedge r)\Big)\bigg).$$

You should see that this works in general.

What about getting rid of $\wedge$ instead of $\vee$? A similar trick can be used, working from the "inside" of the formula instead of from the "outside." We leave it to your inventiveness to find the trick. $\square$

In the last half of the previous proof, we used a trick that let us replace $\vee$ with $\wedge$. In fact, this trick is used often enough that we should call it a rule.[3] It is very useful to have a catalog of simple rules to help in deciding whether or not functions are equal, without having to always construct tables of functions. Here is such a catalog. In each case the standard name of the rule is given first, followed by the rules as applied first to $\wedge$ and then to $\vee$.

Theorem 2 (Algebraic rules for Boolean functions) *Each rule states that two different-looking Boolean functions are equal. That is, they look different but have the same table.*

Associative Rules:	$(p \wedge q) \wedge r = p \wedge (q \wedge r)$	$(p \vee q) \vee r = p \vee (q \vee r)$
Distributive Rules:	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
Idempotent Rules:	$p \wedge p = p$	$p \vee p = p$
Double Negation:	$\sim\sim p = p$	
DeMorgan's Rules:	$\sim(p \wedge q) = \sim p \vee \sim q$	$\sim(p \vee q) = \sim p \wedge \sim q$
Commutative Rules:	$p \wedge q = q \wedge p$	$p \vee q = q \vee p$
Absorption Rules:	$p \vee (p \wedge q) = p$	$p \wedge (p \vee q) = p$
Bound Rules:	$p \wedge 0 = 0 \qquad p \wedge 1 = p$	$p \vee 1 = 1 \qquad p \vee 0 = p$
Negation Rules:	$p \wedge (\sim p) = 0$	$p \vee (\sim p) = 1$

[3] It is sometimes said that a rule or method is a trick that is used more than once.

These rules are "algebraic" rules for working with $\wedge$, $\vee$, and $\sim$. You should memorize them as you use them. They are used just like rules in ordinary algebra: whenever you see an expression on one side of the equal sign, you can replace it by the expression on the other side. Each of the rules can be proved by constructing tables for the functions on each side of the equal sign and verifying that those tables give the same function values.

Truth tables are similar to the tabular method for proving set identities (see Section 1 of Unit SF). The algebraic rules for Boolean functions are almost identical to the rules for sets in Section 1 of Unit SF. When two apparently very different situations (sets and Boolean functions in this case) are similar, one should look for an explanation. We provide an explanation in Unit Lo.

One useful consequence of this connection is that we can use "Venn diagrams" to prove identities for Boolean functions. (If you are not familiar with Venn diagrams, read the first four pages of Unit SF.) How does this work? Think of the variables p, q and so on as sets. Then make the translations:

$$\wedge \text{ to } \cap, \qquad \vee \text{ to } \cup, \qquad \sim \text{ to complement.}$$

The universal set is 1 and the empty set is 0. For example, you can prove the first of DeMorgan's Rules for Boolean functions, by showing that $(P \cap Q)^c$ and $P^c \cup Q^c$ give the same regions in the Venn diagram for two sets P and Q. Try it.

Example 7 (Manipulating functions) We want to simplify the function

$$(\sim(p \wedge \sim q)) \wedge (p \vee q).$$

Here are our calculations:

$$\begin{aligned}
(\sim(p \wedge \sim q)) \wedge (p \vee q) &= (\sim p \vee \sim\sim q) \wedge (p \vee q) && \text{(DeMorgan's rule)}\\
&= (\sim p \vee q) \wedge (p \vee q) && \text{(double negation)}\\
&= ((\sim p \vee q) \wedge p) \vee ((\sim p \vee q) \wedge q) && \text{(distributive rule)}\\
&= (p \wedge (\sim p \vee q)) \vee (q \wedge (q \vee \sim p)) && \text{(commutative rule 3 times)}\\
&= (p \wedge (\sim p \vee q)) \vee q && \text{(absorbtion rule)}\\
&= ((p \wedge \sim p) \vee (p \wedge q)) \vee q && \text{(distributive rule)}\\
&= (0 \vee (p \wedge q)) \vee q && \text{(negation rule)}\\
&= (p \wedge q) \vee q && \text{(bound rule)}\\
&= q && \text{(commutative and absorbtion rules)}
\end{aligned}$$

Except for the last step, we gave each step in detail. In actual calculations, you can combine steps as we did in the last step. How you decide to manipulate things can make a big difference. For example,

$$\begin{aligned}
(\sim(p \wedge \sim q)) \wedge (p \vee q) &= (\sim p \vee \sim\sim q) \wedge (p \vee q) && \text{(DeMorgan's rule)}\\
&= (\sim p \vee q) \wedge (p \vee q) && \text{(double negation)}\\
&= (q \vee \sim p) \wedge (q \vee p) && \text{(commutative rule)}\\
&= q \vee (\sim p \wedge p) && \text{(distributive rule)}\\
&= q \vee 0 && \text{(negation rule)}\\
&= q && \text{(bound rule)}
\end{aligned}$$

The same thing happens in algebra; however, you are more likely to do things the shorter way in algebra because you are more familiar with those manipulations. There is a trade off between taking time to try finding a shorter way and simply going ahead. This is a problem faced by designers of "symbolic manipulation" packages such as Maple® and Mathematica®. ☐

Exercises for Section 1

1.1. Let f = "she is out of work" and s = "she is spending more." Write the following statements in symbolic form:

(a) She is out of work but she is spending more.

(b) Neither is she out of work nor is she spending more.

1.2. Let r = "she registered to vote" and v = "she voted." Write the following statements in symbolic form: She registered to vote but she did not vote.

1.3. Make a truth table for $\sim\big((p \wedge q) \vee \sim(p \vee q)\big)$.

1.4. Make a truth table for $\sim p \wedge (q \vee \sim r)$.

1.5. Make a truth table for $\big(p \vee (\sim p \vee q)\big) \wedge \sim(q \wedge \sim r)$.

1.6. Using DeMorgan's rule, state the negation of the statement: "Mary is a musician and she plays chess."

1.7. Using DeMorgan's rule, state the negation of the statement: "The car is out of gas or the fuel line is plugged."

1.8. Show that $p \vee (p \wedge q) = p$ follows from the idempotent rule, distributive rule, and the absorption rule $p \wedge (p \vee q) = p$.

1.9. Is the function $(p \wedge q) \vee r$ equal to the function $p \wedge (q \vee r)$?

1.10. Is the function $(p \vee q) \vee (p \wedge r)$ equal to the function $(p \vee q) \wedge r$?

1.11. Is the function $\big((\sim p \vee q) \wedge (p \vee \sim r)\big) \wedge (\sim p \vee \sim q)$ equal to the function $\sim(p \vee r)$?

1.12. Is the function $(r \vee p) \wedge \big(\sim r \vee (p \wedge q)\big) \wedge (r \vee q)$ equal to the function $p \wedge q$?

1.13. Is the function $\sim(p \vee \sim q) \vee (\sim p \wedge \sim q)$ equal to the function $\sim p$?

1.14. Is the function $\sim\big((\sim p \wedge q) \vee (\sim p \wedge \sim q)\big) \vee (p \wedge q)$ equal to the function $\sim p$?

1.15. Is the function $\Big(p \wedge \big(\sim(\sim p \vee q)\big)\Big) \vee (p \wedge q)$ equal to the function $p \vee q$?

Section 2: Number Systems and Computer Arithmetic

The number system we are most familiar with is the "base 10"system. In that system, an "n-digit" number is represented by a sequence of "digits," $d_{n-1} \cdots d_1\, d_0$. For example, 243598102 is a 9-digit number base 10. The numerical value of the number $d_{n-1} \cdots d_1\, d_0$ is $d_{n-1}10^{n-1} + d_{n-2}10^{n-2} + \cdots + d_1 10^1 + d_0 10^0$. We are familiar, from elementary school, with various tedious algorithms (treated with mystical reverence by the "back-to-basics" educational advocates) for adding, subtracting, multiplying, and dividing numbers in the base 10 system using pencil and paper calculations. Although these algorithms are sometimes useful, mostly we use our hand calculators or computers to do these calculations. Here, for example, is the almost instantaneous result of asking a computer program to compute 500!, the product of the first 500 positive integers, and represent the answer in base 10.

1220136825991110068701238785423046926253574342803192842192413588385845
3731538819976054964475022032818630136164771482035841633787220781772004
8078520515932928547790757193933060377296085908627042917454788242491272
6344305670173270769461062802310452644218878789465754777149863494367781
0376442740338273653974713864778784954384895955375379904232410612713269
8432774571554630997720278101456108118837370953101635632443298702956389
6628911658974769572087926928871281780070265174507768410719624390394322
5364226052349458501299185715012487069615681416253590566934238130088562
2492468915641267756544818865065938479517753608940057452389403357984763
6394490531306232374906644504882466507594673586207463792518420045936969
2981022263971952597190945217823331756934581508552332820762820023402626
9078983424517120062077146409794561161276291459512372299133401695523638
5094288559201872743379517301458635757082835578015873543276888868012039
9882384702151467605445407663535984174430480128938313896881639487469658
8175045069263653381750554781286400000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000

Example 8 (Base 256) There is no reason why we have to use base 10. For certain applications, base 256 is used. In base 10 we have ten familiar symbols to use for the digits, namely

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9\,.$$

What do we use in base 256 for the symbols used for the digits of numbers? One obvious choice is to use

$$[0], [1], \ldots, [254], [255]\,.$$

We have written these digit symbols with square braces to avoid confusion when digit symbols are concatenated to form numbers. In the base-256 system, an "n-digit" number, as in base 10, is represented by a sequence $d_{n-1} \cdots d_1\, d_0$ where each d_i is a digit symbol (from $[0], [1], \ldots, [254], [255]$). For example, $[12][43][251][198][210][122][2][0][0][0]$ is a 10-digit number base 256. The numerical value of the number $d_{n-1} \cdots d_1\, d_0$ is $d_{n-1}256^{n-1} + d_{n-2}256^{n-2} + \cdots + d_1 256^1 + d_0 256^0$. The symbols d_i are used in two different ways here, but in practice there should be no confusion. For example, the 10-digit base-256 number [12][43][251][198][210][122][2][0][0][0] has numerical value

$$12 \cdot 256^9 + 43 \cdot 256^8 + 251 \cdot 256^7 + 198 \cdot 256^6 + 210 \cdot 256^5 + 122 \cdot 256^4 + 2 \cdot 256^3.$$

In base 10, this number is 57479750209175623303168. □

Example 9 (Number systems base b) In fact, a number system can be defined for any integer base $b > 1$. We need unique symbols for the *digit symbols*, say $[0], [1], \ldots, [b-1]$ or, more generally, $D_0, \ldots, D_{b-1}$. The digit symbol D_i is said to have *index* or *rank* i in the list of digit symbols for the base-b number system. If d is a digit symbol, we use $\iota(d)$ ("iota of d") to represent its rank in the list of digit symbols. Thus, $\iota(D_k) = k$ or $\iota([k]) = k$ in the lists of digit symbols just described. Then, an n-digit positive number can be represented uniquely as $d_{n-1} \cdots d_1\, d_0$ where each d_i is one of the digit symbols $D_0, \ldots, D_{b-1}$ and $\iota(d_{n-1}) > 0$. The numerical value of this number is $\iota(d_{n-1})b^{n-1} + \iota(d_{n-2})b^{n-2} + \cdots + \iota(d_1)b^1 + \iota(d_0)b^0$. In base b, using digit symbols $[0], [1], \ldots, [b-1]$, the nonnegative numbers, ordered according to numerical value, start off $[0], [1], [2], \ldots, [b-1]$. These are the 1-digit numbers, base b. Next come the 2-digit numbers

$$[1][0], \ldots, [1][b-1],\ [2][0], \ldots, [2][b-1], \ldots,\ [b-1][0], \ldots, [b-1][b-1].$$

If you wanted to do paper and pencil calculations for addition, subtraction, multiplication or long division in base b, you could use the "back-to-basics" algorithms with base-b digit symbols instead of base-10 symbols. We will study such algorithms for $b = 2$ later in this section.

The largest n-digit number in base b is

$$[b-1][b-1] \ldots [b-1] = (b-1)b^{n-1} + \cdots + (b-1) = (b-1)\frac{b^n - 1}{b-1} = b^n - 1.$$

If we add [1] to it, we get

$$[b-1][b-1] \ldots [b-1] + [1] = [1][0] \cdots [0],$$

which is the smallest $(n+1)$-digit number and has value b^n. □

Example 10 (Transforming numbers between bases) Given a number, how can we write it in base b?

It would be nice to begin with an example in base 10, but there's no familiar way to specify a number without using base 10. We can fake it a bit by writing the number in words. Let's write twenty-one hundred seven in base 10.

- If we divide by 10, we obtain two hundred ten and a remainder of seven. Thus our rightmost digit is [7]. Now start again with two hundred ten.
- If we divide by 10, we obtain twenty-one and a remainder of zero. Thus our rightmost digit is [0]. Now start again with twenty-one.

- If we divide by 10, we obtain two and a remainder of one. Thus our rightmost digit is [1]. Now start again with two.
- If we divide by 10, we obtain zero and a remainder of two. Thus our rightmost digit is [2]. Since we've reached zero, we're done.

Putting the rightmost digits together in order, we have [2][1][0][7], or 2107 in the usual notation for base-10 numbers.

Now suppose our number N is written in the familiar way (base 10) and we want to write it in base b. We proceed in the same manner as in the previous paragraph, dividing by b each time instead of by 10.

Here is an example for base 256, using digit symbols $[0], [1], \ldots, [255]$. Suppose we are given the decimal (base-10) number 3865988647 and want to write it in base 256.

- 38659886477/256 is 15101518 with a remainder of 39. Thus our rightmost digit is [39]. Now start again with 15101518.
- 15101518/256 is 58990 with a remainder of 78. Thus our rightmost digit is [78]. Now start again with 58990.
- 58990/256 is 230 with a remainder of 110. Thus our rightmost digit is [110]. Now start again with 230.
- 230/256 is 0 with a remainder of 230. Thus our rightmost digit is [230]. Since we've reached 0, we're done.

Thus the base-10 number 3865988647 equals [230] [110] [78] [39] in base 256. We can check that this is correct:

$$[230][110][78][39] = 230 \cdot 256^3 + 110 \cdot 256^2 + 78 \cdot 256 + 39 = 38659886477. \qquad \square$$

Computer Arithmetic

In the remainder of this section, we will study base 2, 8, and 16 numbers. Instead of bases 2, 8 and 16, people also say

binary for base 2, *octal* for base 8 and *hexadecimal* for base 16.

You should practice converting some base-10 numbers of your choosing into their binary, octal and hexadecimal equivalents.

Computers, for the most part, work directly with base $b = 2$. The symbols for the digits are usually taken to be $0, 1$. Thus, $10011 = 2^4 + 2^1 + 2^0$. In base ten, 10011 is denoted by 19. If the base needs to be mentioned explicitly in the mathematics, one usually writes it as a subscript: $10011_{\mathbf{2}} = 19_{\mathbf{10}}$. Base 2 is inconvenient for people because it leads to long strings of digits. (Try writing a small number like 2345 in base 2.) As we shall see, converting between base 2 and bases 8 and 16 is quite simple. As a result, when we need to deal with base 2 in studying a computer program, we often write it in base 8 or base 16 instead to avoid long strings of digits.

Boolean Functions and Computer Arithmetic

Example 11 (Octal and hexadecimal number systems) Given a binary number such as 1101100111010101110, one can start at the right and group the digits into groups of three as follows: 1|101|100|111|010|101|110. If each group of three is now replaced by its decimal equivalent, one obtains 1547256. This sequence of digits is the base-8 representation of the given binary number, using the base-8 digit symbols 0, 1, 2, 3, 4, 5, 6, 7. In summary, $1101100111010101110_{\mathbf{2}} = 1547256_{\mathbf{8}}$. The process is easily reversed. For example, $7237631_{\mathbf{8}} = 111010011111110011001_{\mathbf{2}}$. To understand why this works in general, consider the following computation for the latter example:

$$
\begin{aligned}
7237631_{\mathbf{8}} =& 7 \cdot 8^6 + 2 \cdot 8^5 + 3 \cdot 8^4 + 7 \cdot 8^3 + 6 \cdot 8^2 + 3 \cdot 8^1 + 1 \cdot 8^0 \\
=& (2^2 + 2^1 + 2^0)2^{18} + (2^1)2^{15} + (2^1 + 2^0)2^{12} + (2^2 + 2^1 + 2^0)2^9 \\
&+ (2^2 + 2^1)2^6 + (2^1 + 2^0)2^3 + (2^0)2^0 .
\end{aligned}
$$

In this calculation, we have replaced each power of 8, 8^k, by $(2^3)^k = 2^{3k}$. Each base-8 digit is replaced by its corresponding expansion in terms of powers of 2. By using the distributive law on each of the summands in the latter expression (e.g., $(2^2+2^1+2^0)2^{18} = 2^{20} + 2^{19} + 2^{18}$) we get the defining expansion as powers of two of the binary number $111010011111110011001_{\mathbf{2}} = 7237631_{\mathbf{8}}$.

Base-16 numbers relate to base-2 numbers in a manner directly analogous to the way base-8 numbers relate to base-2 numbers. One problem is to decide on the symbols used to represent the 16 digits needed for the base-16 number system. One could use $[0], [1], [2], \ldots, [10], [11], [12], [13], [14], [15]$. Thus, [11][8][10] would represent the number $11 \cdot 16^2 + 8 \cdot 16^1 + 10 \cdot 16^0$. Although this is sometimes done, it is more common to use `A`, `B`, `C`, `D`, `E`, and `F` in place of [10], [11], [12], [13], [14], [15]. In place of $[0], [1], [2], \ldots, [9]$, the ordinary base-10 digit symbols are used, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Thus [11][8][10] would be written `B8A`. To transform a base-2 number such as 1101100111010101110 to a base-16 number, first group the digits, right to left, in groups of four: 110|1100|1110|1010|1110. Then change each group to base 10, using the symbols `A`, `B`, `C`, `D`, `E`, and `F` for the base-10 numbers 10 through 15. For 110|1100|1110|1010|1110 the result is 6|`C`|`E`|`A`|`E` which we write `6CEAE`. Going in the reverse direction is just as easy, namely, replace each base-16 digit by its base two form.

To transform numbers from base-10 representation to bases 2, 8, or 16, transform to any one of these bases, say base 2, and then transform between 2, 8, and 16 using the algorithms described above. Going in the reverse, from bases 2, 8, or 16 to base 10, just use the definition of the base expansion directly (e.g., $\texttt{6CEAE} = 6 \cdot 16^4 + 12 \cdot 16^3 + 14 \cdot 16^2 + 10 \cdot 16^1 + 14 \cdot 16^0$). ☐

Example 12 (Binary arithmetic) Here we show an addition, multiplication, and subtraction using the familiar base-10 algorithms, followed in each case by a similar base-2 computation.

$$
\begin{array}{r}
34783 \\
+\ \underline{61358} \\
96141
\end{array}
\qquad
\begin{array}{r}
1110 \\
+\ \underline{1011} \\
11001
\end{array}
\qquad
\begin{array}{r}
584 \\
\times\ \underline{307} \\
4088 \\
\underline{1752} \\
179288
\end{array}
\qquad
\begin{array}{r}
111 \\
\times\ \underline{101} \\
111 \\
\underline{111} \\
100011
\end{array}
\qquad
\begin{array}{r}
{\scriptstyle 3\ 9\ 9\ \ \ 5\ 9\ \ } \\
4\,\cancel{0}\,\cancel{0}\,{}^{1}0\,\cancel{6}\,\cancel{0}\,{}^{1}2 \\
-\ \underline{3876513} \\
124089
\end{array}
\qquad
\begin{array}{r}
{\scriptstyle 0\ 1\ 1\ 1\ \ } \\
1\,\cancel{1}\,\cancel{0}\,\cancel{0}\,\cancel{0}\,{}^{1}0 \\
-\ \underline{100001} \\
001111
\end{array}
$$

You should study each example and understand the similarities and the differences. The base-2 addition corresponds to $14_{\mathbf{10}} + 11_{\mathbf{10}}$, the base-2 multiplication corresponds to $7_{\mathbf{10}} \times 5_{\mathbf{10}}$, and the base-2 subtraction corresponds to $48_{\mathbf{10}} - 33_{\mathbf{10}}$.

The familiar long division algorithm from elementary school works fine for base-2 numbers. It is actually easier in base 2. Here is an example of long division carried out using the standard algorithm and a similar calculation using base-2 numbers (corresponding to $554_{\mathbf{10}}$ divided by $9_{\mathbf{10}}$ to get $61_{\mathbf{10}}$ with remainder $5_{\mathbf{10}}$).

```
              9 9 9 7 6
        _________________
4 2 5  | 4 2 4 9 0 0 0 1
         3 8 2 5
         -------
           4 2 4 0
           3 8 2 5
           -------
             4 1 5 0
             3 8 2 5
             -------
               3 2 5 0
               2 9 7 5
               -------
                 2 7 5 1
                 2 5 5 0
                 -------
                   2 0 1
```

```
                  1 1 1 1 0 1
         _____________________
1 0 0 1 | 1 0 0 0 1 0 1 0 1 0
            1 0 0 1
            -------
            1 0 0 0 0
              1 0 0 1
              -------
                1 1 1 1
                1 0 0 1
                -------
                  1 1 0 0
                  1 0 0 1
                  -------
                      1 1 1 0
                      1 0 0 1
                      -------
                        1 0 1
```

The layout of the long division algorithm is designed for pencil and paper calculation and is obviously not relevant to computer based calculations. The long division algorithm shown above that seeks to divide 42490001 (the "dividend") by 425 (the "divisor"), scans the dividend 42490001 from left to right to find the shortest sequence of digits, in this case 4249, that represents an integer greater than or equal to 425. This gives us our starting point. Since 425 has three digits, the shortest sequence will always have either three or four digits (that is the whole point of why this algorithm is used for elementary school kids). The first step is to divide 425 into this number 4249, thus representing the dividend as

$$42490001 = 9 \times 425 \times 10^4 + 4240001.$$

We then move over one digit in the remainder. Thus we divide 425 into 4240. As a result we get

$$42490001 = 9 \times 425 \times 10^4 + 9 \times 425 \times 10^3 + 415001.$$

The next step, applied to 415001, gives

$$42490001 = 9 \times 425 \times 10^4 + 9 \times 425 \times 10^3 + 9 \times 425 \times 10^2 + 32501.$$

The next step, applied to 32501, gives

$$42490001 = 9 \times 425 \times 10^4 + 9 \times 425 \times 10^3 + 9 \times 425 \times 10^2 + 7 \times 425 \times 10^1 + 2751.$$

The final step, applied to 2751 gives

$$42490001 = 9 \times 425 \times 10^4 + 9 \times 425 \times 10^3 + 9 \times 425 \times 10^2 + 7 \times 425 \times 10^1 + 6 \times 425 + 201.$$

This latter expression is, using standard base-10 notation,

$$42490001 = 99976 \times 425 + 201.$$

Each step could be carried out without being able to divide! For example, to divide 4249 by 425, simply subtract 425 from 4249 as many times as possible. It can be done nine times and the remainder is 415, the result after the nine subtractions. Thus $4249 = 9 \times 425 + 415$.

For base-2 division, avoiding division is easy since the number of subtractions we'll be able to do at each step is either one or zero. If you like to program, you may find it interesting to program this algorithm for base-2 numbers. The required divisions at each stage, for base-2 numbers, can be carried out by subtraction, as done in the above example. □

The important feature of binary arithmetic on a computer is that, at the most primitive hardware level, the size of the register is fixed. We have to understand binary addition with the constraint of fixed register size. Also, we need to discuss negative numbers.

Example 13 (Binary addition and register size) Suppose that the register has places for n binary bits. We number the bits right-to-left from 0 to $n-1$. Here is an example with $n = 16$.

bit position	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit value	0	1	0	1	1	1	0	0	0	1	1	1	0	1	0	1

The bit with position 0 (rightmost bit) is called the "least significant" bit and the bit with position 15 is called the "most significant" bit. In the example just given, the binary number has the value

$$2^{14} + 2^{12} + 2^{11} + 2^{10} + 2^{6} + 2^{5} + 2^{4} + 2^{2} + 2^{0} = 23669 \text{ (base-10)}.$$

We call arithmetic done using an n-bit register "n-bit binary arithmetic".

The largest number that can be represented with 16 binary bits is

$$1111111111111111 = \sum_{i=0}^{15} 2^i = 2^{16} - 1\,.$$

If you are using a 16-bit register to add binary numbers, everything is fine unless you add two numbers whose sum is greater than $2^{16} - 1$. If that happens, too bad! It is an "overflow" and, if you are lucky, you are informed of the problem without having the program mistakenly continue on, perhaps to start World War III. If we get an overflow we reconsider what we are doing and try to avoid it by being more clever. For modern computers, the register size is big enough (at least 32 bits) so that we have lots of room to work with basic integers. We can use these register operations to build software systems that can work with integers of arbitrary size by using as many registers as needed to store the binary form of a number. Such packages are found in software systems that do "multiple precision arithmetic," an example being GNU MP. □

So far, we have only mentioned positive numbers. So, subtracting a smaller number from a larger one on a 16-bit register should work just fine. But what if we want to allow negative numbers? We could use + and − just as we do with base-10 numbers. For example $-101_{\mathbf{2}}$ would be -5 and $+101_{\mathbf{2}}$ would be $+5$, which we often write simply as 5. Alternatively, we can use a bit somewhere to keep track of the sign, say 0 for $+$ and 1 for $-$. Following our usual convention about sign placement, we could use the leftmost bit, the bit in position 15. In our 16-bit register, -5 would then be 1000000000000101 and $+5$ would be 0000000000000101, which is the way we were storing 5 in our register. Notice that the size of our numbers is now restricted to 15 bits because one bit is used for the sign. Thus $|x| < 2^{15}$.

Next we need to work out rules for adding and subtracting when numbers may be positive or negative. This is the obvious way to do things, but there's an easier way. Surprisingly, this easier way exists because of the problem we ran into in the previous example!

Example 14 (Negative numbers and register size) Forget about negative numbers for a minute. Imagine adding 1 to the contents of a 16-bit register. It's clear what to do unless the register contains 1111111111111111. When we add 1, the result is 0000000000000000 with a 1 to carry and no place to put it — our overflow problem. Since we have no place to put it, let's just throw it away! Since 0000000000000000 is zero, we have the equation

$$1111111111111111 \text{ plus } 1 \text{ equals } 0.$$

Since we know that $(-1) + 1 = 0$, it looks like we should think of 1111111111111111 as being -1. Wait, the binary number 1111111111111111 equals $2^{16} - 1$. What happened?

Think of telling time, but forget about the hours and just talk about minutes. The time can be k minutes after the hour for $k = 0, 1, \ldots 59$. Sixty minutes after the hour is the start of the next hour, and since we aren't keeping track of what the hour is, that's simply zero minutes after the hour. Add one minute to 59 minutes after the hour and we're back to zero — just liking adding 1 to 1111111111111111. Where are the negative numbers? Fifty-nine minutes *after* the hour is also one minute *before* the (next) hour. So "after" is positive and "before" is negative. Well then, is it 59 or -1? That's up to you. If we want to be "fair" and have about as many negative minutes as positive ones, we could go up to 29 or 30 minutes after (positive) or before (negative) the hour.

You can picture all this on the dial of an old fashioned (analog) watch or clock. When the minute hand is before 6, we count minutes after the hour (clockwise) and get a positive number. When the minute hand is after 6, we count minutes before the hour (counter clockwise) and get a negative number. When the minute hand is on the 6, we have either $+30$ or -30, depending on what we decide. Our large numbers (more than 30 minutes after the hour) are now thought of as negative numbers (before the hour). Minutes before and after the hour are related by

$$(\text{minutes before the hour}) \quad + \quad (\text{minutes after the hour}) \quad = \quad 60.$$

"One hour" in our register has 2^{16} minutes and 30 minutes corresponds to 2^{15}. If you are familiar with modular arithmetic, you may recognize that the clock is doing arithmetic modulo 60 and our register is doing it modulo 2^{16}.

Just as with the clock, numbers that look large in the register are to be thought of as negative numbers. We can simply look at the leftmost bit to see if a number is large. If the leftmost bit is 1, the number is thought of as negative. This is different from the obvious idea; for example, consider -1:

$$\text{“obvious”}\ -1 = 1000000000000001 \qquad \text{new}\ -1 = 1111111111111111.$$

Recall that the new value is -1 (one before zero) because when we add 1 to it and throw away the carry we get 0000000000000000. $\square$

The idea we just introduced is called *two's complement* notation. It won't do us any good unless we understand how to do arithmetic with such numbers and how to convert a negative number like -5 into the bits in a register. That's the subject of the next example. As we'll see, the arithmetic is easy.

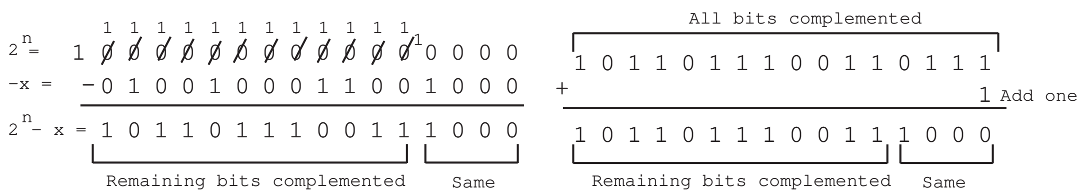
Example 15 (Two's complement arithmetic) Let's suppose we have an n-long register and let $x < 2^{n-1}$ be a positive number stored in the register. Since $x < 2^{n-1}$, the leftmost bit of x is zero. What should be stored in the register to represent $-x$? In the previous example we said it was the number we needed to add to a register containing x to get zero, after throwing away the carry bit that we had no place to put. If we were doing ordinary grade school arithmetic, the carry bit would have given us 2^n, one larger than $2^n - 1$, the register filled with ones. Let's restate what we just concluded:

The positive number that should be thought of as $-x$
is
the number y such that $x + y = 2^n$.

This is just the before and after minutes rule for a clock.

In other words, our representation of $-x$ is the number $y = 2^n - x$. The number $2^n - x$ is called the *two's complement* of x. More generally, if some number z, positive or negative, is stored in a register, we find $-z$ by computing the two's complement of the value stored for z.

Let's look at an example of computing the two's complement. We take $n = 16$ and $x = 0100100011001000$. We want to compute $2^n - x$. The lefthand calculation in the following figure is the obvious way to do the calculation.



The observation at the bottom of the lefthand calculation gives a simple short-cut rule for computing the n-bit two's complement of x directly in terms of the digits of x. Scan the digits of x from right to left until the first 1 is encountered. Complement all digits of x to the left of that first 1. Some texts describe the “short cut” process of taking the two's complement as complementing all bits of x and adding 1 to the result. As shown in the right-hand calculation in the above figure, this leads to the same thing. Use whichever short-cut method you prefer.

How do we add two numbers? For the time being, forget about overflow. Add the way we have already learned to do it, throwing away the carry bit. This gives the correct answer regardless of whether the numbers are positive or negative or one of each as long as the numbers are not too big in size. (We'll be specific about "too big" later.) Why is this?

First, let's try an example. Suppose have a 16-long register and want to add -5 and -8. You should compute the two's complements of 5 and 8, add the results and check that the answer is the two's complement of 13. Here's how we can see that without doing all the work. The two's complements are $2^{16} - 5$ and $2^{16} - 8$. Adding them as numbers we get $2^{16} + 2^{16} - 13$. Adding them in register arithmetic, one 2^{16} will be thrown away as a carry bit. (This has to happen because our register answer must be between 0 and $2^{16} - 1$.) Thus we get $2^{16} - 13$. Clearly this works in general. If we were to add -5 and 8, we would get $2^{16} - 5 + 8 = 2^{16} + 3$. The 2^{16} would be thrown away as a carry bit, giving us 3.

What about numbers that are "too big" (overflow)?

- If both numbers are positive and the answer looks negative, there was an "overflow." This will happen if we have $0 < x, y < 2^{15}$ but $x + y \geq 2^{15}$ because when we add there is a carry into the sign bit, making it 1 which is supposed to indicate a negative number.
- If both numbers are negative and the answer looks positive, there was overflow.

If you think about it and look at examples, you should be able to convince yourself that this is the only way overflow can happen. For example, if one number is positive and the other negative, there cannot be any overflow. This gives us a simple rule:

In two's complement arithmetic, there is overflow if and only if
x and y have the same sign bit and $x + y$ has a different sign bit.

What about subtracting two numbers, say x and y? You can either do the subtraction in the usual way, or take the n-bit two's complement of y and add (ignoring any value carried into the $(n + 1)^{\text{th}}$ bit). Both methods are illustrated in the following calculation ($n = 16$). You should write the binary numbers as ordinary base-10 numbers (including signs) and check the calculation.

$$\begin{array}{ccccccccccccccccc}
 & \overset{0}{\cancel{1}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & {}^{1}0 & \overset{0}{\cancel{1}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & {}^{1}0 & 1 & 1 & 1 \\
- & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\qquad
\begin{array}{ccccccccccccccccc}
 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
+ & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}$$

Thus, representing the negative of a number using two's complement, reduces subtraction to addition.

What about multiplying two numbers? Multiplication is really just shifting and adding: When we multiplied 111 by 101 we added 111 to the result of shifting 111 to the left by two bits. What if bits get shifted off the left end of the register? Shifting x by k bits is the same as adding 2^k copies of x together, so that's okay. Thus, multiplication works regardless of whether the numbers are positive or negative. Overflow is somewhat trickier and we won't discuss it.

What about dividing two numbers? When division is understood properly, it works okay. We won't go into that. You might like to think about it. $\square$

Boolean Functions and Computer Arithmetic

In the previous example, we saw how doing two's complement arithmetic reduces to addition. To complete our discussion, we look at how to build a circuit for binary addition. Computer circuits implement Boolean functions. We will discuss how to do addition using **and**, **or** and **xor**.

Example 16 (A circuit for binary addition) Suppose we want to add the two binary numbers $a_{n-1}a_{n-2}\cdots a_1a_0$ and $b_{n-1}b_{n-2}\cdots b_1b_0$. Let the answer be $s_ns_{n-1}\cdots s_1s_0$. (The leading digits of these numbers might be zero.) Here is how we did it in Example 12.

(0) Add the binary digits a_0 and b_0 to obtain the two digit sum c_0s_0. The digit c_0 is the "carry" digit.

(1) Add the binary digits a_1, b_1 and c_0 to obtain the two digit sum c_1s_1.

...

$(n-2)$ Add the binary digits a_{n-2}, b_{n-2} and c_{n-3} to obtain the two digit sum $c_{n-2}s_{n-2}$.

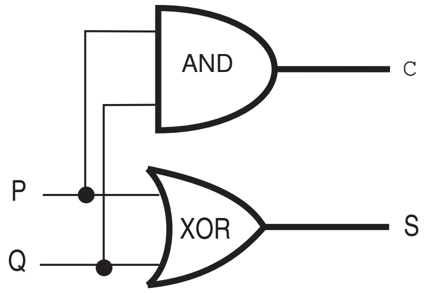
$(n-1)$ Add the binary digits a_{n-1}, b_{n-1} and c_{n-2} to obtain the two digit sum s_ns_{n-1}.

We discuss the circuit for this in two steps. First, adding two binary digits — called a "half adder." Then, adding three binary digits — called a "full adder."

Let p, q, s and c be the binary digits in $p+q=cs$. Thus c is the carry digit. Considering the four possibilities for (p,q), we get the following table for the Boolean functions s and c:

p	q	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Note that $s = p \oplus q$, the "exclusive or" function, and $c = p \wedge q$, the "and" function. We can visualize these two Boolean functions in one circuit diagram as follows:



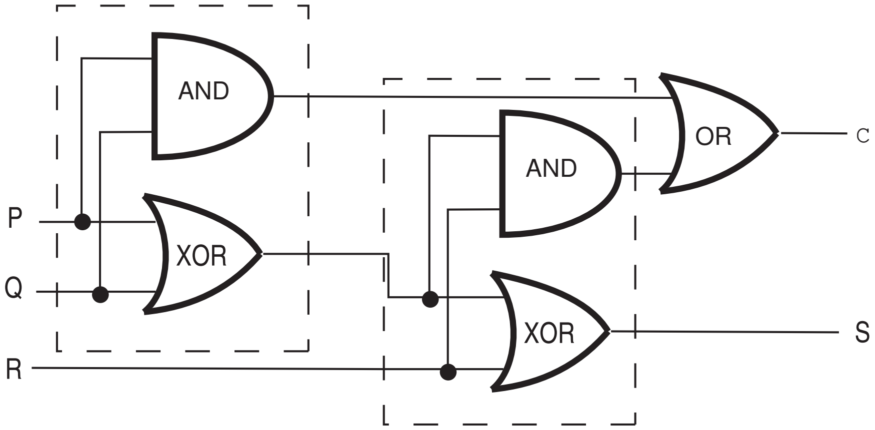
This circuit with two Boolean variables as inputs and two as outputs is sometimes referred to as a "half adder." The functions **and** and **xor** in the circuit are referred to as *logic gates* or just *gates*. You should imagine the values of p and q entering at the points labeled P and Q and moving along the wires. A • indicates a branch — the value moves along both wires leading out from the •. Thus the value of p enters both gates and the value of q enters both gates. The values of c and s emerge at the points labeled C and S. These points can be used as inputs to another circuit, if desired.

The half adder circuit can be used to compute $a_0 + b_0 = c_0 s_0$ in Step (0). Now we need to add three binary digits for Steps (1) through $(n-1)$. In other words, we want to find c and s so that $p + q + r = cs$. This can be done in steps:

$$p + q = c's' \quad \text{then} \quad s' + r = c''s \quad \text{then} \quad c' + c'' = c.$$

Why doesn't the last step give a two digit answer? It's impossible. Suppose there were two digits c^*c. We are adding three digits, p, q and r. By the above calculations, the answer (in binary) is c^*cs. The largest possible answer, obtained when $p = q = r = 1$ is 11, which has only two digits. Thus $c^* = 0$. Actually, this is nothing more than the fact that when we do addition the carry is never more than a single digit.

We can do the calculations described above with three half adders, one for each addition. Since the third half adder cannot produce a carry, we'll throw away that portion of the circuit. Here's the result, with the two complete half adders in dashed boxes:



Wait! The figure has a misprint — an **or** where there should be an **xor**. In fact, either one works. Why is that? Here's the table for computing $c' + c''$:

c'	c''	c
0	0	0
0	1	1
1	0	1
1	1	$*$

The table has no entry for $c' = c'' = 1$ because that never occurs. (Remember the discussion in the previous paragraph about the carry being a single digit.) Since it never happens, we can define this entry any way we choose. Choosing 0 gives $c = c' \oplus c''$. Choosing 1 gives $c = c' \vee c''$.

Our device for adding two binary numbers can now be built. Use a half adder for Step (0). Next, take the carry from this half adder, a_1 and b_1 as inputs to a full adder. This is Step (1). Next, take the carry from this adder, a_2 and b_2 as inputs to a full adder. This is Step (2). Continue in this manner.

How long does our circuit take to add two n-bit registers? Suppose a single gate takes time T. Then a half adder takes time T because the two gates do their calculations at the same time. You should be able to see that our full adder takes time $3T$. We use $n-1$ full adders and one half adder for an n-bit register. Thus the total time is $(n-1)3T + T = (3n-2)T$. $\square$

Exercises for Section 2

2.1. For each of the following, construct a Boolean function equal to the function S defined by the given truth table. Make your function as simple as you can. Then design a circuit for the function.

(a)

P	Q	R	S
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(b)

P	Q	R	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

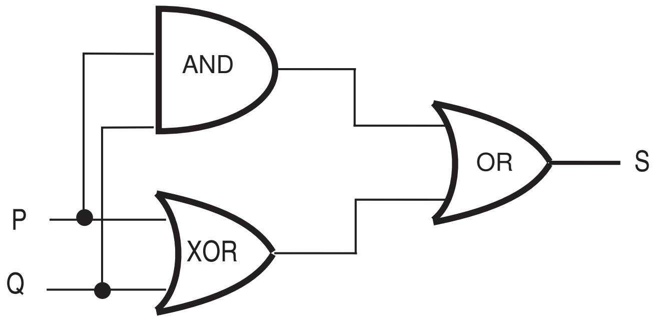
2.2. Design a circuit that represents the Boolean function S where $S(P, Q, R) = 0$ if and only if $(P, Q, R) = (0, 0, 0)$ or $(P, Q, R) = (1, 1, 1)$.

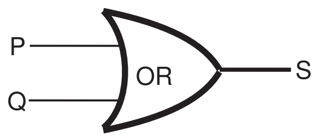
2.3. Design a circuit to represent the response of the lights in a room to the light switches under each of the following conditions.

(a) There are two switches, moving either switch to the opposite position turns the room lights on if off and off if on.

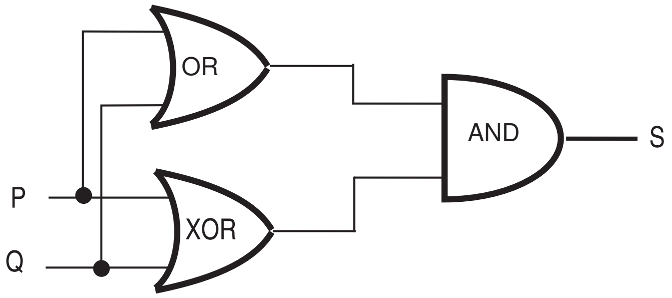
(b) There are three switches, moving any switch to the opposite position turns the room lights on if off and off if on.

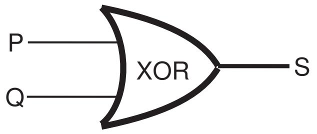
2.4. Show that the following two circuits represent the same Boolean function.



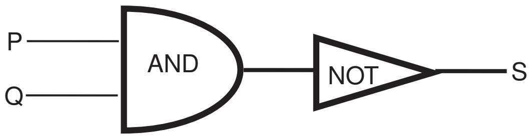


2.5. Show that the following two circuits represent the same Boolean function.





2.6. Show that the Boolean function $(\sim P \wedge \sim Q) \vee (P \oplus Q)$ equals the Boolean function computed by the following circuit with just two logic gates (NOT and AND):



2.7. Find a circuit with at most three logic gates, each of which is allowed to have at most two inputs, that is equal to the Boolean function defined by the following truth table.

P	Q	R	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

2.8. Compute the difference of the two base two numbers: $1110100_{\mathbf{2}} - 10111_{\mathbf{2}}$.

2.9. Convert $1011011111000101_{\mathbf{2}}$ from binary to hexadecimal (i.e., base 16) and octal (i.e., base 8).

2.10. Convert the following as indicated.

(a) Convert $61502_{\mathbf{8}}$ to decimal.

(b) Convert $\texttt{EB7}C5_{\mathbf{16}}$ to octal.

2.11. Let $b, m, l, f, -, z, k, a, n, y, e, x, j, w, d, v, o, u, c, g, t, p, h, s, q, i, r$ be the digit symbol list for base 27. Let $n, m, k, j, f, s, q, h, z, p, c, x, y, e, d, w$ be the digit symbol list for base 16.

(a) Convert $hi - there$ from base 27 to base 16.

(b) Convert $cfemxysnnjnq$ from base 16 to base 27.

2.12. Find the 8-bit two's complement of $67_{\mathbf{10}}$.

2.13. Find the 8-bit two's complement of $108_{\mathbf{10}}$.

2.14. The number $10001001_{\mathbf{2}}$ is the 8-bit two's complement of a number k. What is the decimal representation of k?

2.15. The number $10111010_{\mathbf{2}}$ is the 8-bit two's complement of a number k. What is the decimal representation of k?

2.16. Using base-2 arithmetic, compute $79 - 43$. Then compute it using 8-bit two's-complement registers. Remember to check for overflow.

2.17. Using base-2 arithmetic, compute $-15 - 46$. Then compute it using 8-bit two's-complement registers. Also compute $46 + 46 + 46$ Remember to check for overflow.

2.18. We have defined and learned how to use the idea of two's complement for n-bit binary numbers. What about "n-digit ten's complement" for base ten arithmetic? Define the appropriate notion of ten's complement and show, by example, how to compute with it in a way that is analogous to computing with two's complement.

Multiple Choice Questions for Review

In each case there is one correct answer (given at the end of the problem set). Try to work the problem first without looking at the answer. Understand both why the correct answer is correct and why the other answers are wrong.

1. Let
m = "Juan is a math major,"
c = "Juan is a computer science major,"
g = "Juan's girlfriend is a literature major,"
h = "Juan's girlfriend has read Hamlet," and
t = "Juan's girlfriend has read The Tempest."
Which of the following expresses the statement "Juan is a computer science major and a math major, but his girlfriend is a literature major who hasn't read both The Tempest and Hamlet."

(a) $c \wedge m \wedge (g \vee (\sim h \vee \sim t))$

(b) $c \wedge m \wedge g \wedge (\sim h \wedge \sim t)$

(c) $c \wedge m \wedge g \wedge (\sim h \vee \sim t)$

(d) $c \wedge m \wedge (g \vee (\sim h \wedge \sim t))$

(e) $c \wedge m \wedge g \wedge (h \vee t)$

2. The function $((p \vee (r \vee q)) \wedge \sim(\sim q \wedge \sim r)$ is equal to the function

(a) $q \vee r$

(b) $((p \vee r) \vee q)) \wedge (p \vee r)$

(c) $(p \wedge q) \vee (p \wedge r)$

(d) $(p \vee q) \wedge \sim(p \vee r)$

(e) $(p \wedge r) \vee (p \wedge q)$

3. The truth table for $(p \vee q) \vee (p \wedge r)$ is the same as the truth table for

(a) $(p \vee q) \wedge (p \vee r)$

(b) $(p \vee q) \wedge r$

(c) $(p \vee q) \wedge (p \wedge r)$

(d) $p \vee q$

(e) $(p \wedge q) \vee p$

4. The Boolean function $[\sim(\sim p \wedge q) \wedge \sim(\sim p \wedge \sim q)] \vee (p \wedge r)$ is equal to the Boolean function

(a) q (b) $p \wedge r$ (c) $p \vee q$ (d) r (e) p

5. Which of the following functions is the constant 1 function?

(a) $\sim p \vee (p \wedge q)$

(b) $(p \wedge q) \vee (\sim p \vee (p \wedge \sim q))$

(c) $(p \wedge \sim q) \wedge (\sim p \vee q)$

(d) $((\sim p \wedge q) \wedge (q \wedge r)) \wedge \sim q$

(e) $(\sim p \vee q) \vee (p \wedge q)$

6. Consider the statement, "Either $-2 \leq x \leq -1$ or $1 \leq x \leq 2$." The negation of this statement is

(a) $x < -2$ or $2 < x$ or $-1 < x < 1$

(b) $x < -2$ or $2 < x$

(c) $-1 < x < 1$

(d) $-2 < x < 2$

(e) $x \leq -2$ or $2 \leq x$ or $-1 < x < 1$

7. The truth table for a Boolean expression is specified by the correspondence $(P, Q, R) \rightarrow S$ where $(0,0,0) \rightarrow 0$, $(0,0,1) \rightarrow 1$, $(0,1,0) \rightarrow 0$, $(0,1,1) \rightarrow 1$, $(1,0,0) \rightarrow 0$, $(1,0,1) \rightarrow 0$, $(1,1,0) \rightarrow 0$, $(1,1,1) \rightarrow 1$. A Boolean expression having this truth table is

(a) $[(\sim P \wedge \sim Q) \vee Q] \vee R$

(b) $[(\sim P \wedge \sim Q) \wedge Q] \wedge R$

(c) $[(\sim P \wedge \sim Q) \vee \sim Q] \wedge R$

(d) $[(\sim P \wedge \sim Q) \vee Q] \wedge R$

(e) $[(\sim P \vee \sim Q) \wedge Q] \wedge R$

8. Which of the following statements is **FALSE**:

(a) $(P \wedge Q) \vee (\sim P \wedge Q) \vee (P \wedge \sim Q)$ is equal to $\sim Q \wedge \sim P$

(b) $(P \wedge Q) \vee (\sim P \wedge Q) \vee (P \wedge \sim Q)$ is equal to $Q \vee P$

(c) $(P \wedge Q) \vee (\sim P \wedge Q) \vee (P \wedge \sim Q)$ is equal to $Q \vee (P \wedge \sim Q)$

(d) $(P \wedge Q) \vee (\sim P \wedge Q) \vee (P \wedge \sim Q)$ is equal to $[(P \vee \sim P) \wedge Q] \vee (P \wedge \sim Q)$

(e) $(P \wedge Q) \vee (\sim P \wedge Q) \vee (P \wedge \sim Q)$ is equal to $P \vee (Q \wedge \sim P)$.

9. To show that the circuit corresponding to the Boolean expression $(P \wedge Q) \vee (\sim P \wedge Q) \vee (\sim P \wedge \sim Q)$ can be represented using two logical gates, one shows that this Boolean expression is equal to $\sim P \vee Q$. The circuit corresponding to $(P \wedge Q \wedge R) \vee (\sim P \wedge Q \wedge R) \vee (\sim P \wedge (\sim Q \vee \sim R)$ computes the same function as the circuit corresponding to

(a) $(P \wedge Q) \vee \sim R$

(b) $P \vee (Q \wedge R)$

(c) $\sim P \vee (Q \wedge R)$

(d) $(P \wedge \sim Q) \vee R$

(e) $\sim P \vee Q \vee R$

10. Using binary arithmetic, a number y is computed by taking the n-bit two's complement of $x - c$. If n is eleven, $x = 10100001001_{\mathbf{2}}$ and $c = 10101_{\mathbf{2}}$ then $y =$

(a) $01100001111_{\mathbf{2}}$

(b) $01100001100_{\mathbf{2}}$

(c) $01100011100_{\mathbf{2}}$

(d) $01000111100_{\mathbf{2}}$

(e) $01100000000_{\mathbf{2}}$

11. In binary, the sixteen-bit two's complement of the hexadecimal number $\mathtt{DEAF}_{\mathbf{16}}$ is

(a) $0010000101010111_{\mathbf{2}}$

(b) $1101111010101111_{\mathbf{2}}$

(c) $0010000101010011_{\mathbf{2}}$

(d) $0010000101010001_{\mathbf{2}}$

(e) $0010000101000001_{\mathbf{2}}$

12. In octal, the twelve-bit two's complement of the hexadecimal number $2\mathtt{AF}_{\mathbf{16}}$ is

(a) $6522_{\mathbf{8}}$

(b) $6251_{\mathbf{8}}$

(c) $5261_{\mathbf{8}}$

(d) $6512_{\mathbf{8}}$

(e) $6521_{\mathbf{8}}$

Answers: **1** (c), **2** (a), **3** (d), **4** (e), **5** (b), **6** (a), **7** (d), **8** (a), **9** (c), **10** (b), **11** (d), **12** (e).