ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 1.1

The Halting Problem

For some of the topics in this course, we plan to cover the material more deeply than does the textbook. We will provide notes in this format to supplement the textbook for this purpose. In order to make these notes more useful as a reference, definitions are highlighted with boldface, and italicization emphasizes pitfalls or other important points.

This set of notes gives describes the first problem known to be undecidable, the halting problem. For our class, you need only recognize the name and realize that one can, in fact, give examples of problems that cannot be solved by computation. In the future, you should be able to recognize this type of problem so as to avoid spending your time trying to solve it.

Universal Computing Machines

As discussed in the textbook and in class, a **universal computational device** (or **computing machine**) is a device that is capable of computing the solution to any problem that can be computed, provided that the device is given enough storage and time for the computation to finish. The idea came out of a 1936 paper by Alan Turing, and today we generally refer to these devices as **Turing machines**.

The things that we call computers today, whether we are talking about a programmable microcontroller in a microwave oven or the Blue Waters supercomputer sitting on the south end of our campus (the United States' main resource to support computational science research), are all equivalent in the sense of what problems they can solve. All of them are provably equivalent to Turing machines. These machines do, of course, have access to different amounts of memory, and compute at different speeds.

Turing also conjectured that his definition of computable was identical to the "natural" definition. In other words, a problem that cannot be solved by a Turing machine cannot be solved in any systematic manner, with any machine, or by any person. This thesis remains unproven! However, neither has anyone been able to disprove the thesis, and it is widely believed to be true. Disproving the thesis requires that one demonstrate a systematic technique (or a machine) capable of solving a problem that cannot be solved by a Turing machine. No one has been able to do so to date.

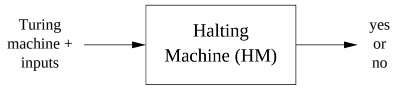
The Halting Problem

You might reasonably ask whether any problems can be shown to be incomputable. More common terms for such problems—those known to be insolvable by any computer—are **intractable** or **undecidable**. In the same 1936 paper, Alan Turing also provided an answer to this question by introducing (and proving) that there are in fact problems that cannot be computed by a universal computing machine. The problem that he proved undecidable, using proof techniques almost identical to those developed for similar problems in the 1880s, is now known as **the halting problem**.

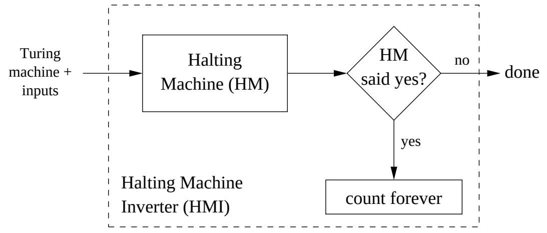
The halting problem is easy to state and easy to prove undecidable. The problem is this: given a Turing machine and an input to the Turing machine, does the Turing machine finish computing in a finite number of steps (a finite amount of time)? In order to solve the problem, an answer, either yes or no, must be given in a finite amount of time regardless of the machine or input in question. Clearly some machines never finish. For example, we can write a Turing machine that counts upwards starting from one.

You may find the proof structure for undecidability of the halting problem easier to understand if you first think about a related problem with which you may already be familiar, the Liar's paradox (which is at least 2,300 years old). In its stengthened form, it is the following sentence: "This sentence is not true."

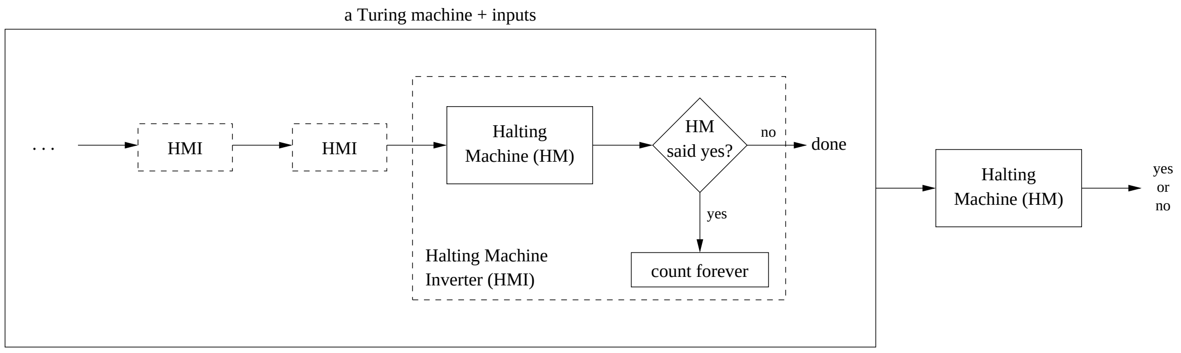
To see that no Turing machine can solve the halting problem, we begin by assuming that such a machine exists, and then show that its existence is self-contradictory. We call the machine the "Halting Machine," or HM for short. HM is a machine that operates on another machine and its inputs to produce a yes or no answer in finite time: either the machine in question finishes in finite time (HM returns "yes"), or it does not (HM returns "no"). The figure illustrates HM's operation.



From HM, we construct a second machine that we call the HM Inverter, or HMI. This machine inverts the sense of the answer given by HM. In particular, the inputs are fed directly into a copy of HM, and if HM answers "yes," HMI enters an infinite loop. If HM answers "no," HMI halts. A diagram appears to the right.



The inconsistency can now be seen by asking HM whether HMI halts when given itself as an input (repeatedly), as shown below. Two copies of HM are thus being asked the same question. One copy is the rightmost in the figure below and the second is embedded in the HMI machine that we are using as the input to the rightmost HM. As the two copies of HM operate on the same input (HMI operating on HMI), they should return the same answer: a Turing machine either halts on an input, or it does not; they are deterministic.



Let's assume that the rightmost HM tells us that HMI operating on itself halts. Then the copy of HM in HMI (when HMI executes on itself, with itself as an input) must also say "yes." But this answer implies that HMI doesn't halt (see the figure above), so the answer should have been no!

Alternatively, we can assume that the rightmost HM says that HMI operating on itself does not halt. Again, the copy of HM in HMI must give the same answer. But in this case HMI halts, again contradicting our assumption.

Since neither answer is consistent, no consistent answer can be given, and the original assumption that HM exists is incorrect. Thus, no Turing machine can solve the halting problem.

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 1.2

The 2's Complement Representation

This set of notes explains the rationale for using the 2's complement representation for signed integers and derives the representation based on equivalence of the addition function to that of addition using the unsigned representation with the same number of bits.

Review of Bits and the Unsigned Representation

In modern digital systems, we represent all types of information using binary digits, or **bits**. Logically, a bit is either 0 or 1. Physically, a bit may be a voltage, a magnetic field, or even the electrical resistance of a tiny sliver of glass. Any type of information can be represented with an ordered set of bits, provided that *any given pattern of bits corresponds to only one value* and that *we agree in advance on which pattern of bits represents which value.*

For unsigned integers—that is, whole numbers greater or equal to zero—we chose to use the base 2 representation already familiar to us from mathematics. We call this representation the **unsigned representation**. For example, in a 4-bit unsigned representation, we write the number 0 as 0000, the number 5 as 0101, and the number 12 as 1100. Note that we always write the same number of bits for any pattern in the representation: *in a digital system, there is no "blank" bit value.*

Picking a Good Representation

In class, we discussed the question of what makes one representation better than another. The value of the unsigned representation, for example, is in part our existing familiarity with the base 2 analogues of arithmetic. For base 2 arithmetic, we can use nearly identical techniques to those that we learned in elementary school for adding, subtracting, multiplying, and dividing base 10 numbers.

Reasoning about the relative merits of representations from a practical engineering perspective is (probably) currently beyond your ability. Saving energy, making the implementation simple, and allowing the implementation to execute quickly probably all sound attractive, but a quantitative comparison between two representations on any of these bases requires knowledge that you will acquire in the next few years.

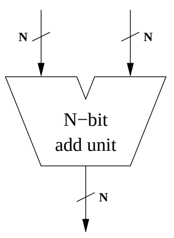
We can sidestep such questions, however, by realizing that if a digital system has hardware to perform operations such as addition on unsigned values, using the same piece of hardware to operate on other representations incurs little or no additional cost. In this set of notes, we discuss the 2's complement representation, which allows reuse of the unsigned add unit (as well as a basis for performing subtraction of either representation using an add unit!). In discussion section and in your homework, you will use the same idea to perform operations on other representations, such as changing an upper case letter in ASCII to a lower case one, or converting from an ASCII digit to an unsigned representation of the same number.

The Unsigned Add Unit

In order to define a representation for signed integers that allows us to reuse a piece of hardware designed for unsigned integers, we must first understand what such a piece of hardware actually does (we do not need to know how it works yet—we'll explore that question later in our class).

The unsigned representation using N bits is not closed under addition. In other words, for any value of N, we can easily find two N-bit unsigned numbers that, when added together, cannot be represented as an N-bit unsigned number. With $N = 4$, for example, we can add 12 (1100) and 6 (0110) to obtain 18. Since 18 is outside of the range $[0, 2^4 - 1]$ representable using the 4-bit unsigned representation, our representation breaks if we try to represent the sum using this representation. We call this failure an **overflow** condition: the representation cannot represent the result of the operation, in this case addition.

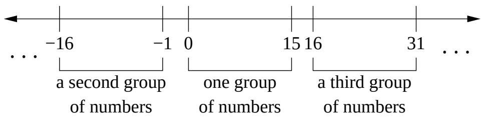
Using more bits to represent the answer is not an attractive solution, since we might then want to use more bits for the inputs, which in turn requires more bits for the outputs, and so on. We cannot build something supporting an infinite number of bits. Instead, we choose a value for N and build an add unit that adds two N-bit numbers and produces an N-bit sum (and some overflow indicators, which we discuss in the next set of notes). The diagram to the right shows how we might draw such a device, with two N-bit numbers entering at from the top, and the N-bit sum coming out from the bottom.



The function used for N-bit unsigned addition is addition modulo 2^N. In a practical sense, you can think of this function as simply keeping the last N bits of the answer; other bits are simply discarded. In the example to the right, we add 12 and 6 to obtain 18, but then discard the extra bit on the left, so the add unit produces 2 (an overflow).

$$\begin{array}{rl} \texttt{1100} & (12) \\ \underline{+\ \texttt{0110}} & (6) \\ \cancel{\texttt{1}}\texttt{0010} & (2) \end{array}$$

Modular arithmetic defines a way of performing arithmetic for a finite number of possible values, usually integers. As a concrete example, let's use modulo 16, which corresponds to the addition unit for our 4-bit examples.



Starting with the full range of integers, we can define equivalence classes for groups of 16 integers by simply breaking up the number line into contiguous groups, starting with the numbers 0 to 15, as shown to the right. The numbers -16 to -1 form a group, as do the numbers from 16 to 31. An infinite number of groups are defined in this manner.

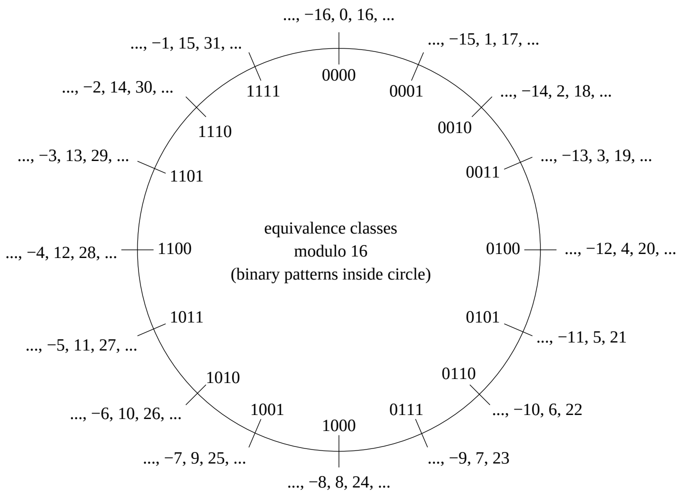
You can think of these groups as defining **equivalence classes** modulo 16. All of the first numbers in the groups are equivalent modulo 16. All of the second numbers in the groups are equivalent modulo 16. And so forth. Mathematically, we say that two numbers A and B are equivalent modulo 16, which we write as

$$(A = B) \bmod 16$$

if and only if $A = B + 16k$ for some integer k.

Deriving 2's Complement

Given these equivalence classes, we might instead choose to draw a circle to identify the equivalence classes and to associate each class with one of the sixteen possible 4-bit patterns, as shown to the right. Using this circle representation, we can add by counting clockwise around the circle, and we can subtract by counting in a counterclockwise direction around the circle. With an unsigned representation, we choose to use the group from $[0, 15]$ (the middle group in the diagram markings to the right) as the number represented by each of the patterns. Overflow occurs with unsigned addition (or subtraction) because we can only choose one value for each binary pattern.



In fact, we can choose any single value for each pattern to create a representation, and our add unit will always produce results that are correct modulo 16. Look back at our overflow example, where we added 12 and 6 to obtain 2, and notice that $(2 = 18) \bmod 16$. Normally, only a contiguous sequence of integers makes a useful representation, but we do not have to restrict ourselves to non-negative numbers.

The 2's complement representation can then be defined by choosing a set of integers balanced around zero from the groups. In the circle diagram, for example, we might choose to represent numbers in the range $[-7, 7]$ when using 4 bits. What about the last pattern, 1000? We could choose to represent either -8 or 8. The number of arithmetic operations that overflow is the same with both choices (the choices are symmetric around 0, as are the combinations of input operands that overflow), so we gain nothing in that sense from either choice. If we choose to represent -8, however, notice that all patterns starting with a 1 bit then represent negative numbers. No such simple check arises with the opposite choice, and thus an N-bit 2's complement representation is defined to represent the range $[-2^{N-1}, 2^{N-1}-1]$, with patterns chosen as shown in the circle.

An Algebraic Approach

Some people prefer an algebraic approach to understanding the definition of 2's complement, so we present such an approach next. Let's start by writing $f(A, B)$ for the result of our add unit:

$$f(A, B) = (A + B) \bmod 2^N$$

We assume that we want to represent a set of integers balanced around 0 using our signed representation, and that we will use the same binary patterns as we do with an unsigned representation to represent non-negative numbers. Thus, with an N-bit representation, the patterns in the range $[0, 2^{N-1} - 1]$ are the same as those used with an unsigned representation. In this case, we are left with all patterns beginning with a 1 bit.

The question then is this: given an integer k, $2^{N-1} > k > 0$, for which we want to find a pattern to represent $-k$, and any integer $m \geq 0$ that we might want to add to $-k$, can we find another integer $p > 0$ such that

$$(-k + m = p + m) \bmod 2^N \quad ? \tag{1}$$

If we can, we can use p's representation to represent $-k$ and our unsigned addition unit $f(A, B)$ will work correctly.

To find the value p, start by subtracting m from both sides of Equation (1) to obtain:

$$(-k = p) \bmod 2^N \tag{2}$$

Note that $(2^N = 0) \bmod 2^N$, and add this equation to Equation (2) to obtain

$$(2^N - k = p) \bmod 2^N$$

Let $p = 2^N - k$. Then, since $2^{N-1} > k > 0$, we have $2^{N-1} < p < 2^N$. But these patterns are all unused (they all start with a 1 bit)! (They also match our circle diagram from the last section exactly, of course.)

Negating 2's Complement Numbers

The algebraic approach makes understanding 2's complement negation fairly straightforward, and gives us an easy procedure for doing so. Recall that given an integer k in an N-bit 2's complement representation, the N-bit pattern for $-k$ is given by $2^N - k$ (also true for $k = 0$ if we keep only the low N bits of the result). But $2^N = (2^N - 1) + 1$. Note that $2^N - 1$ is the pattern of all 1 bits. Subtracting any value k from this value is equivalent to simply flipping the bits, changing 0s to 1s and 1s to 0s. (This operation is called a **1's complement**, by the way.) We then add 1 to the result to find the pattern for $-k$.

Negation can overflow, of course. Try finding the negative pattern for -8 in 4-bit 2's complement.

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 1.3

Overflow Conditions

This set of notes discusses the overflow conditions for unsigned and 2's complement addition. For both types, we formally prove that the conditions that we state are correct. Many of our faculty want our students to learn to construct formal proofs, so we plan to begin exposing you to this process in our classes. Prof. Lumetta is a fan of Prof. George Polya's educational theories with regard to proof techniques, and in particular the idea that one builds up a repertoire of approaches by seeing the approaches used in practice.

Implication and Mathematical Notation

Some of you may not have been exposed to basics of mathematical logic, so let's start with a brief introduction to implication. We'll use variables p and q to represent statements that can be either true or false. For example, p might represent the statement, "Jan is an ECE student," while q might represent the statement, "Jan works hard." The **logical complement** or **negation** of a statement p, written for example as "not p," has the opposite truth value: if p is true, not p is false, and if p is false, not p is true.

An **implication** is a logical relationship between two statements. The implication itself is also a logical statement, and may be true or false. In English, for example, we might say, "If p, q." In mathematics, the same implication is usually written as either "q if p" or "$p \rightarrow q$," and the latter is read as, "p implies q." Using our example values for p and q, we can see that $p \rightarrow q$ is true: "Jan is an ECE student" does in fact imply that "Jan works hard!"

The implication $p \rightarrow q$ is only considered false if p is true and q is false. In all other cases, the implication is true. This definition can be a little confusing at first, so let's use another example to see why. Let p represent the statement "Entity X is a flying pig," and let q represent the statement, "Entity X obeys air traffic control regulations." Here the implication $p \rightarrow q$ is again true: flying pigs do not exist, so p is false, and thus "$p \rightarrow q$" is true—for any value of statement q!

Given an implication "$p \rightarrow q$," we say that the **converse** of the implication is the statement "$q \rightarrow p$," which is also an implication. In mathematics, the converse of $p \rightarrow q$ is sometimes written as "q only if p." The converse of an implication may or may not have the same truth value as the implication itself. Finally, we frequently use the shorthand notation, "p if and only if q," (or, even shorter, "p iff q") to mean "$p \rightarrow q$ *and* $q \rightarrow p$." This last statement is true only when both implications are true.

Overflow for Unsigned Addition

Let's say that we add two N-bit unsigned numbers, A and B. The N-bit unsigned representation can represent integers in the range $[0, 2^N - 1]$. Recall that we say that the addition operation has overflowed if the number represented by the N-bit pattern produced for the sum does not actually represent the number $A + B$.

For clarity, let's name the bits of A by writing the number as $a_{N-1}a_{N-2}...a_1a_0$. Similarly, let's write B as $b_{N-1}b_{N-2}...b_1b_0$. Name the sum $C = A + B$. The sum that comes out of the add unit has only N bits, but recall that we claimed in class that the overflow condition for unsigned addition is given by the **carry** out of the most significant bit. So let's write the sum as $c_Nc_{N-1}c_{N-2}...c_1c_0$, realizing that c_N is the carry out and not actually part of the sum produced by the add unit.

Theorem: Addition of two N-bit unsigned numbers $A = a_{N-1}a_{N-2}...a_1a_0$ and $B = b_{N-1}b_{N-2}...b_1b_0$ to produce sum $C = A + B = c_Nc_{N-1}c_{N-2}...c_1c_0$, overflows if and only if the carry out c_N of the addition is a 1 bit.

Proof: Let's start with the "if" direction. In other words, $c_N = 1$ implies overflow. Recall that unsigned addition is the same as base 2 addition, except that we discard bits beyond c_{N-1} from the sum C. The bit c_N has place value 2^N, so, when $c_N = 1$ we can write that the correct sum $C \geq 2^N$. But no value that large can be represented using the N-bit unsigned representation, so we have an overflow.

The other direction ("only if") is slightly more complex: we need to show that overflow implies that $c_N = 1$. We use a range-based argument for this purpose. Overflow means that the sum C is outside the representable range $[0, 2^N - 1]$. Adding two non-negative numbers cannot produce a negative number, so the sum can't be smaller than 0. Overflow thus implies that $C \geq 2^N$.

Does that argument complete the proof? No, because some numbers, such as 2^{N+1}, are larger than 2^N, but do not have a 1 bit in the Nth position when written in binary. We need to make use of the constraints on A and B implied by the possible range of the representation.

In particular, given that A and B are represented as N-bit unsigned values, we can write

$$\begin{aligned} 0 \leq \quad & A \quad \leq 2^N - 1 \\ 0 \leq \quad & B \quad \leq 2^N - 1 \end{aligned}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$0 \leq \quad C \quad \leq 2^{N+1} - 2$$

Combining the new inequality with the one implied by the overflow condition, we obtain

$$2^N \leq \quad C \quad \leq 2^{N+1} - 2$$

All of the numbers in the range allowed by this inequality have $c_N = 1$, completing our proof.

Overflow for 2's Complement Addition

Understanding overflow for 2's complement addition is somewhat trickier, which is why the problem is a good one for you to think about on your own first. Our operands, A and B, are now two N-bit 2's complement numbers. The N-bit 2's complement representation can represent integers in the range $[-2^{N-1}, 2^{N-1} - 1]$. Let's start by ruling out a case that we can show never leads to overflow.

Lemma: Addition of two N-bit 2's complement numbers A and B does not overflow if one of the numbers is negative and the other is not.

Proof: We again make use of the constraints implied by the fact that A and B are represented as N-bit 2's complement values. We can assume **without loss of generality**[1], or **w.l.o.g.**, that $A < 0$ and $B \geq 0$.

Combining these constraints with the range representable by N-bit 2's complement, we obtain

$$\begin{aligned} -2^{N-1} \leq \quad & A \quad < 0 \\ 0 \leq \quad & B \quad < 2^{N-1} \end{aligned}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$-2^{N-1} \leq \quad C \quad < 2^{N-1}$$

But anything in the range specified by this inequality can be represented with N-bit 2's complement, and thus the addition does not overflow.

[1]This common mathematical phrasing means that we are using a problem symmetry to cut down the length of the proof discussion. In this case, the names A and B aren't particularly important, since addition is commutative ($A + B = B + A$). Thus the proof for the case in which A is negative (and B is not) is identical to the case in which B is negative (and A is not), except that all of the names are swapped. The term "without loss of generality" means that we consider the proof complete even with additional assumptions, in our case that $A < 0$ and $B \geq 0$.

We are now ready to state our main theorem. For convenience, let's use different names for the actual sum $C = A + B$ and the sum S returned from the add unit. We define S as the number represented by the bit pattern produced by the add unit. When overflow occurs, $S \neq C$, but we always have $(S = C) \bmod 2^N$.

Theorem: Addition of two N-bit 2's complement numbers A and B overflows if and only if one of the following conditions holds:

1. $A < 0$ and $B < 0$ and $S \geq 0$
2. $A \geq 0$ and $B \geq 0$ and $S < 0$

Proof: We once again start with the "if" direction. That is, if condition 1 or condition 2 holds, we have an overflow. The proofs are straightforward. Given condition 1, we can add the two inequalities $A < 0$ and $B < 0$ to obtain $C = A + B < 0$. But $S \geq 0$, so clearly $S \neq C$, thus overflow has occurred.

Similarly, if condition 2 holds, we can add the inequalities $A \geq 0$ and $B \geq 0$ to obtain $C = A + B \geq 0$. Here we have $S < 0$, so again $S \neq C$, and we have an overflow.

We must now prove the "only if" direction, showing that any overflow implies either condition 1 or condition 2. By the **contrapositive**[2] of our Lemma, we know that if an overflow occurs, either both operands are negative, or they are both positive.

Let's start with the case in which both operands are negative, so $A < 0$ and $B < 0$, and thus the real sum $C < 0$ as well. Given that A and B are represented as N-bit 2's complement, they must fall in the representable range, so we can write

$$\begin{array}{rcl} -2^{N-1} \leq & A & < 0 \\ -2^{N-1} \leq & B & < 0 \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$-2^N \leq \quad C \quad < 0$$

Given that an overflow has occurred, C must fall outside of the representable range. Given that $C < 0$, it cannot be larger than the largest possible number representable using N-bit 2's complement, so we can write

$$-2^N \leq \quad C \quad < -2^{N-1}$$

We now add 2^N to each part to obtain

$$0 \leq \quad C + 2^N \quad < 2^{N-1}$$

This range of integers falls within the representable range for N-bit 2's complement, so we can replace the middle expression with S (equal to C modulo 2^N) to find that

$$0 \leq \quad S \quad < 2^{N-1}$$

Thus, if we have an overflow and both $A < 0$ and $B < 0$, the resulting sum $S \geq 0$, and condition 1 holds.

The proof for the case in which we observe an overflow when both operands are non-negative ($A \geq 0$ and $B \geq 0$) is similar, and leads to condition 2. We again begin with inequalities for A and B:

$$\begin{array}{rcl} 0 \leq & A & < 2^{N-1} \\ 0 \leq & B & < 2^{N-1} \end{array}$$

We add these two inequalities and replace $A + B$ with C to obtain

$$0 \leq \quad C < \quad 2^N$$

[2]If we have a statement of the form (p implies q), its contrapositive is the statement (not q implies not p). Both statements have the same truth value. In this case, we can turn our Lemma around as stated.

Given that an overflow has occurred, C must fall outside of the representable range. Given that $C \geq 0$, it cannot be smaller than the smallest possible number representable using N-bit 2's complement, so we can write

$$2^{N-1} \leq \quad C \quad < 2^N$$

We now subtract 2^N to each part to obtain

$$-2^{N-1} \leq \quad C - 2^N \quad < 0$$

This range of integers falls within the representable range for N-bit 2's complement, so we can replace the middle expression with S (equal to C modulo 2^N) to find that

$$-2^{N-1} \leq \quad S \quad < 0$$

Thus, if we have an overflow and both $A \geq 0$ and $B \geq 0$, the resulting sum $S < 0$, and condition 2 holds.

Thus overflow implies either condition 1 or condition 2, completing our proof.

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 1.4

Logic Operations

This set of notes briefly describes a generalization to truth tables, then introduces Boolean logic operations as well as notational conventions and tools that we use to express general functions on bits. We illustrate how logic operations enable us to express functions such as overflow conditions concisely, then show by construction that a small number of logic operations suffices to describe any operation on any number of bits. We close by discussing a few implications and examples.

Truth Tables

inputs		outputs	
A	B	C	S
00	00	0	00
00	01	0	01
00	10	0	10
00	11	0	11
01	00	0	01
01	01	0	10
01	10	0	11
01	11	1	00
10	00	0	10
10	01	0	11
10	10	1	00
10	11	1	01
11	00	0	11
11	01	1	00
11	10	1	01
11	11	1	10

You have seen the basic form of truth tables in the textbook and in class. Over the semester, we will introduce several extensions to the basic concept, mostly with the goal of reducing the amount of writing necessary when using truth tables. For example, the truth table to the right uses two generalizations to show the carry out C (also the unsigned overflow indicator) and the sum S produced by adding two 2-bit unsigned numbers. First, rather than writing each input bit separately, we have grouped pairs of input bits into the numbers A and B. Second, we have defined multiple output columns so as to include both bits of S as well as C in the same table. Finally, we have grouped the two bits of S into one column.

Keep in mind as you write truth tables that only rarely does an operation correspond to a simple and familiar process such as addition of base 2 numbers. We had to choose the unsigned and 2's complement representations carefully to allow ourselves to take advantage of a familiar process. In general, for each line of a truth table for an operation, you may need to make use of the input representation to identify the input values, calculate the operation's result as a value, and then translate the value back into the correct bit pattern using the output representation. Signed magnitude addition, for example, does not always correspond to base 2 addition: when the signs of the two input operands differ, one should instead use base 2 subtraction. For other operations or representations, base 2 arithmetic may have no relevance at all.

Boolean Logic Operations

In the middle of the 19th century, George Boole introduced a set of logic operations that are today known as **Boolean logic** (also as **Boolean algebra**). These operations today form one of the lowest abstraction levels in digital systems, and an understanding of their meaning and use is critical to the effective development of both hardware and software.

You have probably seen these functions many times already in your education—perhaps first in set-theoretic form as Venn diagrams. However, given the use of common English words *with different meanings* to name some of the functions, and the sometimes confusing associations made even by engineering educators, we want to provide you with a concise set of definitions that generalizes correctly to more than two operands. You may have learned these functions based on truth values (true and false), but we define them based on bits, with 1 representing true and 0 representing false.

Table 1 on the next page lists logic operations. The first column in the table lists the name of each function. The second provides a fairly complete set of the notations that you are likely to encounter for each function, including both the forms used in engineering and those used in mathematics. The third column defines the function's value for two or more input operands (except for NOT, which operates on a single value). The last column shows the form generally used in logic schematics/diagrams and mentions the important features used in distinguishing each function (in pictorial form usually called a **gate**, in reference to common physical implementations) from the others.

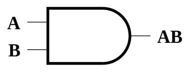
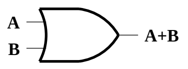
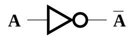
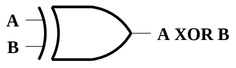
Function	Notation	Explanation	Schematic
AND	A AND B AB $A \cdot B$ $A \times B$ $A \wedge B$	the "all" function: result is 1 iff **all** input operands are equal to 1	 flat input, round output
OR	A OR B $A + B$ $A \vee B$	the "any" function: result is 1 iff **any** input operand is equal to 1	 round input, pointed output
NOT	NOT A A' $\overline{A}$ $\neg A$	logical complement/negation: NOT 0 is 1, and NOT 1 is 0	 triangle and circle
XOR exclusive OR	A XOR B $A \oplus B$	the "odd" function: result is 1 iff an **odd** number of input operands are equal to 1	 OR with two lines on input side
English "or"	A, B, or C	the "one of" function: result is 1 iff exactly **one of** the input operands is equal to 1	(not used)

Table 1: Boolean logic operations, notation, definitions, and symbols.

The first function of importance is **AND**. Think of **AND** as the "all" function: given a set of input values as operands, AND evaluates to 1 if and only if *all* of the input values are 1. The first notation line simply uses the name of the function. In Boolean algebra, AND is typically represented as multiplication, and the middle three forms reflect various ways in which we write multiplication. The last notational variant is from mathematics, where the AND function is formally called **conjunction**.

The next function of importance is **OR**. Think of **OR** as the "any" function: given a set of input values as operands, OR evaluates to 1 if and only if *any* of the input values is 1. The actual number of input values equal to 1 only matters in the sense of whether it is at least one. The notation for OR is organized in the same way as for AND, with the function name at the top, the algebraic variant that we will use in class—in this case addition—in the middle, and the mathematics variant, in this case called **disjunction**, at the bottom.

The definition of Boolean OR is not the same as our use of the word "or" in English. For example, if you are fortunate enough to enjoy a meal on a plane, you might be offered several choices: "Would you like the chicken, the beef, or the vegetarian lasagna today?" Unacceptable answers to this English question include: "Yes," "Chicken and lasagna," and any other combination that involves more than a single choice!

You may have noticed that we might have instead mentioned that AND evaluates to 0 if any input value is 0, and that OR evaluates to 0 if all input values are 0. These relationships reflect a mathematical duality underlying Boolean logic that has important practical value in terms of making it easier for humans to digest complex logic expressions. We will talk more about duality later in the course, but you should learn some of the practical value now: if you are trying to evaluate an AND function, look for an input with value 0; if you are trying to evaluate an OR function, look for an input with value 1. If you find such an input, you know the function's value without calculating any other input values.

We next consider the **logical complement** function, **NOT**. The NOT function is also called **negation**. Unlike our first two functions, NOT accepts only a single operand, and reverses its value, turning 0 into 1 and 1 into 0. The notation follows the same pattern: a version using the function name at the top, followed by two variants used in Boolean algebra, and finally the version frequently used in mathematics. For the NOT gate, or **inverter**, the circle is actually the important part: the triangle by itself merely copies the input. You will see the small circle added to other gates on both inputs and outputs; in both cases the circle implies a NOT function.

Last among the Boolean logic functions, we have the **XOR**, or **exclusive OR** function. Think of XOR as the "odd" function: given a set of input values as operands, XOR evaluates to 1 if and only if *an odd number* of the input values are 1. Only two variants of XOR notation are given: the first using the function name, and the second used with Boolean algebra. Mathematics rarely uses this function.

Finally, we have included the meaning of the word "or" in English as a separate function entry to enable you to compare that meaning with the Boolean logic functions easily. Note that many people refer to English' use of the word "or" as "exclusive" because one true value excludes all others from being true. Do not let this human language ambiguity confuse you about XOR! For all logic design purposes, *XOR is the odd function.*

The truth table to the right provides values illustrating these functions operating on three inputs. The AND, OR, and XOR functions are all associative—(A op B) op C = A op (B op C)—and commutative—A op B = B op A, as you may have already realized from their definitions.

inputs			outputs			
A	B	C	ABC	$A+B+C$	$\overline{A}$	$A \oplus B \oplus C$
0	0	0	0	0	1	0
0	0	1	0	1	1	1
0	1	0	0	1	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	0	0
1	1	0	0	1	0	0
1	1	1	1	1	0	1

Overflow as Logic Expressions

In the last set of notes, we discussed overflow conditions for unsigned and 2's complement representations. Let's use Boolean logic to express these conditions.

We begin with addition of two 1-bit unsigned numbers. Call the two input bits A_0 and B_0. If you write a truth table for this operation, you'll notice that overflow occurs only when all (two) bits are 1. If either bit is 0, the sum can't exceed 1, so overflow cannot occur. In other words, overflow in this case can be written using an AND operation:

$$A_0B_0$$

The truth table for adding two 2-bit unsigned numbers is four times as large, and seeing the structure may be difficult. One way of writing the expression for overflow of 2-bit unsigned addition is as follows:

$$A_1B_1 + (A_1 + B_1)A_0B_0$$

This expression is slightly trickier to understand. Think about the place value of the bits. If both of the most significant bits—those with place value 2—are 1, we have an overflow, just as in the case of 1-bit addition. The A_1B_1 term represents this case. We also have an overflow if one or both (the OR) of the most significant bits are 1 and the sum of the two next significant bits—in this case those with place value 1—generates a carry.

The truth table for adding two 3-bit unsigned numbers is probably not something that you want to write out. Fortunately, a pattern should start to become clear with the following expression:

$$A_2B_2 + (A_2 + B_2)A_1B_1 + (A_2 + B_2)(A_1 + B_1)A_0B_0$$

In the 2-bit case, we mentioned the "most significant bit" and the "next most significant bit" to help you see the pattern. The same reasoning describes the first two product terms in our overflow expression for 3-bit unsigned addition (but the place values are 4 for the most significant bit and 2 for the next most significant bit). The last term represents the overflow case in which the two least significant bits generate a carry which then propagates up through all of the other bits because at least one of the two bits in every position is a 1.

$$\begin{array}{rl} & A_{N-1}A_{N-2}\ldots A_2A_1A_0 \\ + & B_{N-1}B_{N-2}\ldots B_2B_1B_0 \\ \hline & S_{N-1}\,S_{N-2}\ldots S_2\,S_1\,S_0 \end{array}$$

The overflow condition for addition of two N-bit 2's complement numbers can be written fairly concisely in terms of the first bits of the two numbers and the first bit of the sum. Recall that overflow in this case depends only on whether the three numbers are negative or non-negative, which is given by the most significant bit. Given the bit names as shown to the right, we can write the overflow condition as follows:

$$A_{N-1}\ B_{N-1}\ \overline{S_{N-1}} + \overline{A_{N-1}}\ \overline{B_{N-1}}\ S_{N-1}$$

The overflow condition does of course depend on all of the bits in the two numbers being added. In the expression above, we have simplified the form by using S_{N-1}. But S_{N-1} depends on the bits A_{N-1} and B_{N-1} as well as the carry out of bit $(N-2)$.

Later in this set of notes, we present a technique with which you can derive an expression for an arbitrary Boolean logic function. As an exercise, after you have finished reading these notes, try using that technique to derive an overflow expression for addition of two N-bit 2's complement numbers based on A_{N-1}, B_{N-1}, and the carry out of bit $(N-2)$ (and into bit $(N-1)$), which we might call C_{N-1}. You might then calculate C_{N-1} in terms of the rest of the bits of A and B using the expressions for unsigned overflow just discussed. In the next month or so, you will learn how to derive more compact expressions yourself from truth tables or other representations of Boolean logic functions.

Logical Completeness

Why do we feel that such a short list of functions is enough? If you think about the number of possible functions on N bits, you might think that we need many more functions to be able to manipulate bits. With 10 bits, for example, there are 2^{1024} such functions. Obviously, some of them have never been used in any computer system, but maybe we should define at least a few more logic operations? In fact, we do not even need XOR. The functions AND, OR, and NOT are sufficient, even if we only allow two input operands for AND and OR!

The theorem below captures this idea, called **logical completeness**. In this case, we claim that the set of functions {AND, OR, NOT} is sufficient to express any operation on any finite number of variables, where each variable is a bit.

Theorem: Given enough 2-input AND, 2-input OR, and 1-input NOT functions, one can express any Boolean logic function on any finite number of variables.

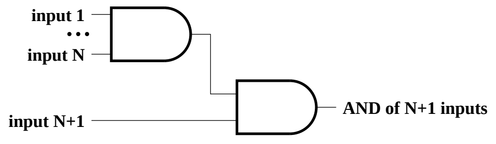
The proof of our theorem is **by construction**. In other words, we show a systematic approach for transforming an arbitrary Boolean logic function on an arbitrary number of variables into a form that uses only AND, OR, and NOT functions on one or two operands. As a first step, we remove the restriction on the number of inputs for the AND and OR functions. For this purpose, we state and prove two **lemmas**, which are simpler theorems used to support the proof of a main theorem.

Lemma 1: Given enough 2-input AND functions, one can express an AND function on any finite number of variables.

Proof: We prove the Lemma **by induction**.[1] Denote the number of inputs to a particular AND function by N.

The base case is $N = 2$. Such an AND function is given.

To complete the proof, we need only show that, given any number of AND functions with up to N inputs, we can express an AND function with $N+1$ inputs. To do so, we need merely use one 2-input AND function to join together the result of an N-input AND function with an additional input, as illustrated to the right.



[1]We assume that you have seen proof by induction previously.

Lemma 2: Given enough 2-input OR functions, one can express an OR function on any finite number of variables.

Proof: The proof of Lemma 2 is identical in structure to that of Lemma 1, but uses OR functions instead of AND functions.

Let's now consider a small subset of functions on N variables. For any such function, you can write out the truth table for the function. The output of a logic function is just a bit, either a 0 or a 1. Let's consider the set of functions on N variables that produce a 1 for exactly one combination of the N variables. In other words, if you were to write out the truth table for such a function, exactly one row in the truth table would have output value 1, while all other rows had output value 0.

Lemma 3: Given enough AND functions and 1-input NOT functions, one can express any Boolean logic function that produces a 1 for exactly one combination of any finite number of variables.

Proof: The proof of Lemma 3 is by construction. Let N be the number of variables on which the function operates. We construct a **minterm** on these N variables, which is an AND operation on each variable or its complement. The minterm is specified by looking at the unique combination of variable values that produces a 1 result for the function. Each variable that must be a 1 is included as itself, while each variable that must be a 0 is included as the variable's complement (using a NOT function). The resulting minterm produces the desired function exactly. When the variables all match the values for which the function should produce 1, the inputs to the AND function are all 1, and the function produces 1. When any variable does not match the value for which the function should produce 1, that variable (or its complement) acts as a 0 input to the AND function, and the function produces a 0, as desired.

The table below shows all eight minterms for three variables.

inputs			outputs							
A	B	C	$\overline{A}\,\overline{B}\,\overline{C}$	$\overline{A}\,\overline{B}\,C$	$\overline{A}\,B\,\overline{C}$	$\overline{A}\,B\,C$	$A\,\overline{B}\,\overline{C}$	$A\,\overline{B}\,C$	$A\,B\,\overline{C}$	$A\,B\,C$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

We are now ready to prove our theorem.

Proof (of Theorem): Any given function on N variables produces the value 1 for some set of combinations of inputs. Let's say that M such combinations produce 1. Note that $M \leq 2^N$. For each combination that produces 1, we can use Lemma 1 to construct an N-input AND function. Then, using Lemma 3, we can use as many as M NOT functions and the N-input AND function to construct a minterm for that input combination. Finally, using Lemma 2, we can construct an M-input OR function and OR together all of the minterms. The result of the OR is the desired function. If the function should produce a 1 for some combination of inputs, that combination's minterm provides a 1 input to the OR, which in turn produces a 1. If a combination should produce a 0, its minterm does not appear in the OR; all other minterms produce 0 for that combination, and thus all inputs to the OR are 0 in such cases, and the OR produces 0, as desired.

The construction that we used to prove logical completeness does not necessarily help with efficient design of logic functions. Think about some of the expressions that we discussed earlier in these notes for overflow conditions. How many minterms do you need for N-bit unsigned overflow? A single Boolean logic function can be expressed in many different ways, and learning how to develop an efficient implementation of a function as well as how to determine whether two logic expressions are identical without actually writing out truth tables are important engineering skills that you will start to learn in the coming months.

Implications of Logical Completeness

If logical completeness doesn't really help us to engineer logic functions, why is the idea important? Think back to the layers of abstraction and the implementation of bits from the first couple of lectures. Voltages are real numbers, not bits. *The device layer implementations of Boolean logic functions must abstract away the analog properties of the physical system.* Without such abstraction, we must think carefully about analog issues such as noise every time we make use of a bit! Logical completeness assures us that no matter what we want to do with bits, implementating a handful of operations correctly is enough to guarantee that we never have to worry.

A second important value of logical completeness is as a tool in screening potential new technologies for computers. If a new technology does not allow implementation of a logically complete set of functions, the new technology is extremely unlikely to be successful in replacing the current one.

That said, {AND, OR, and NOT} is not the only logically complete set of functions. In fact, our current complementary metal-oxide semiconductor (CMOS) technology, on which most of the computer industry is now built, does not directly implement these functions, as you will see later in our class.

The functions that are implemented directly in CMOS are NAND and NOR, which are abbreviations for AND followed by NOT and OR followed by NOT, respectively. Truth tables for the two are shown to the right.

inputs		outputs	
		$\overline{AB}$	$\overline{A+B}$
A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Either of these functions by itself forms a logically complete set. That is, both the set {NAND} and the set {NOR} are logically complete. For now, we leave the proof of this claim to you. Remember that all you need to show is that you can implement any set known to be logically complete, so in order to prove that {NAND} is logically complete (for example), you need only show that you can implement AND, OR, and NOT using only NAND.

Examples and a Generalization

Let's use our construction to solve a few examples. We begin with the functions that we illustrated with the first truth table from this set of notes, the carry out C and sum S of two 2-bit unsigned numbers. Since each output bit requires a separate expression, we now write S_1S_0 for the two bits of the sum. We also need to be able to make use of the individual bits of the input values, so we write these as A_1A_0 and B_1B_0, as shown on the left below. Using our construction from the logical completeness theorem, we obtain the equations on the right. You should verify these expressions yourself.

inputs				outputs		
A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

$$\begin{aligned} C &= \overline{A_1}\;A_0\;B_1\;B_0 + A_1\;\overline{A_0}\;B_1\;\overline{B_0} + A_1\;\overline{A_0}\;B_1\;B_0 + \\ &\quad A_1\;A_0\;\overline{B_1}\;B_0 + A_1\;A_0\;B_1\;\overline{B_0} + A_1\;A_0\;B_1\;B_0 \end{aligned}$$

$$\begin{aligned} S_1 &= \overline{A_1}\;\overline{A_0}\;B_1\;\overline{B_0} + \overline{A_1}\;\overline{A_0}\;B_1\;B_0 + \overline{A_1}\;A_0\;\overline{B_1}\;B_0 + \\ &\quad \overline{A_1}\;A_0\;B_1\;\overline{B_0} + A_1\;\overline{A_0}\;\overline{B_1}\;\overline{B_0} + A_1\;\overline{A_0}\;\overline{B_1}\;B_0 + \\ &\quad A_1\;A_0\;\overline{B_1}\;\overline{B_0} + A_1\;A_0\;B_1\;B_0 \end{aligned}$$

$$\begin{aligned} S_0 &= \overline{A_1}\;\overline{A_0}\;\overline{B_1}\;B_0 + \overline{A_1}\;\overline{A_0}\;B_1\;B_0 + \overline{A_1}\;A_0\;\overline{B_1}\;\overline{B_0} + \\ &\quad \overline{A_1}\;A_0\;B_1\;\overline{B_0} + A_1\;\overline{A_0}\;\overline{B_1}\;B_0 + A_1\;\overline{A_0}\;B_1\;B_0 + \\ &\quad A_1\;A_0\;\overline{B_1}\;\overline{B_0} + A_1\;A_0\;B_1\;\overline{B_0} \end{aligned}$$

Now let's consider a new function. Given an 8-bit 2's complement number, $A = A_7A_6A_5A_4A_3A_2A_1A_0$, we want to compare it with the value -1. We know that we can construct this function using AND, OR, and NOT, but how? We start by writing the representation for -1, which is 11111111. If the number A matches that representation, we want to produce a 1. If the number A differs in any bit, we want to produce a 0. The desired function has exactly one combination of inputs that produces a 1, so in fact we need only one minterm! In this case, we can compare with -1 by calculating the expression:

$$A_7 \cdot A_6 \cdot A_5 \cdot A_4 \cdot A_3 \cdot A_2 \cdot A_1 \cdot A_0$$

Here we have explicitly included multiplication symbols to avoid confusion with our notation for groups of bits, as we used when naming the individual bits of A.

In closing, we briefly introduce a generalization of logic operations to groups of bits. Our representations for integers, real numbers, and characters from human languages all use more than one bit to represent a given value. When we use computers, we often make use of multiple bits in groups in this way. A **byte**, for example, today means an ordered group of eight bits. We can extend our logic functions to operate on such groups by pairing bits from each of two groups and performing the logic operation on each pair. For example, given $A = A_7A_6A_5A_4A_3A_2A_1A_0 = 01010101$ and $B = B_7B_6B_5B_4B_3B_2B_1B_0 = 11110000$, we calculate A AND B by computing the AND of each pair of bits, A_7 AND B_7, A_6 AND B_6, and so forth, to produce the result 01010000, as shown to the right. In the same way, we can extend other logic operations, such as OR, NOT, and XOR, to operate on bits of groups.

$$\begin{array}{r} A\ 01010101 \\ \text{AND}\quad B\ 11110000 \\ \hline 01010000 \end{array}$$