Binary Numbers and Computer Arithmetic

Bob Brown
School of Computing and Software Engineering
Southern Polytechnic State University

Introduction

All modern computer systems use binary, or base two, to represent numbers internally. For this reason, computer scientists must be very familiar with the binary system. The principles of the binary number system are the same as the familiar decimal system. Computer arithmetic is similar to everyday paper-and-pencil arithmetic, but there are some fundamental differences. In this paper we will explore the binary number system, representations of numeric data, and methods of performing arithmetic that are applicable to automation.

Positional Number Systems

The idea of "number" is a mathematical abstraction. To use numbers, we must represent them in some way, whether by piles of pebbles or in some other way. It is common to create a **code** for representing numbers. Such codes are called number systems.

Every number system uses symbols to convey information about the value of a number. A **positional** (or **radix**) number system is one in which the value that a symbol contributes to a number is determined by the both symbol itself and the position of the symbol within the number. That's just a fancy way of saying that 300 is far different from 3. Compare the idea of a positional number system with Roman numbers, where X means 10 no matter where in a number it appears.

The decimal number system we use every day is a positional number system. Decimal numbers are also called base 10 or radix 10 numbers. The symbols are the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, the plus and minus signs, and the period or decimal point. The position of each digit within the number tells us the multiplier used with it.

Consider the number 1037. We learned in elementary school to call the rightmost digit the ones place, the next digit the tens place, then the hundreds place, then the thousands place, and so on. Mathematically, 1037 means $1\times10^3 + 0\times10^2 + 3\times10^1 + 7\times10^0$. Each digit in the number is multiplied by some power of ten, starting with 10^0 at the right and increasing by one for each position to the left. The digits of a decimal number are the coefficients of a power series in powers of ten.

Any number to the zeroth power is one, so 10^0 is one and 7×10^0 is just $7\times1 = 7$. A number raised to the first power is itself, so 10^1 is ten, and 3×10^1 is 3×10, or thirty.

In the hundreds place, we have 0×10^2 or "no hundreds." Even though the symbol zero

does not contribute to the value of the number, it is important as a placeholder. If we didn't have the zero, we could not distinguish between 1037 and 137.

Using just the digits zero through nine, the decimal number system can express any non-negative integer, however large. The value of an *n*-digit decimal integer is

$$a_{n-1}\times 10^{n-1} + a_{n-2}\times 10^{n-2} + \ \ldots + a_1\times 10^1 \ + a_0\times 10^0$$

This can be written more compactly as: $\sum_{i=0}^{n-1} a_i 10^i$

Adding a minus sign to the available symbols lets us express any integer, positive or negative. To express fractions, we use the period symbol. The digit immediately to the right of the period represents 10^{-1} or $^1/_{10}$, the next digit 10^{-2} or $^1/_{100}$, and so on. Although we can represent any integer with decimal numbers, the same is not true of fractions. For example, the fraction $^1/_3$ cannot be exactly represented as a decimal fraction. However, we can make an arbitrarily precise approximation; if 0.33 isn't close enough, we can write 0.333, or even 0.3333333333.

Using Other Bases

The discussion so far has been limited to base 10 or decimal numbers because they are familiar to us. It is possible to represent numbers in positional notation using bases other than 10. An *n*-digit non-negative integer in base *B* would be represented as

$$\sum_{i=0}^{n-1} a_i B^i$$

The only difference between this expression and the one above is that we have substituted some base *B* for 10. The choice of a base isn't entirely arbitrary; we'd like it to be an integer greater than one, and relatively small. Both of these constraints are because a base *B* number system requires *B* symbols. We need at least two symbols so that we can have a zero to serve as a placeholder. We don't want so many symbols that reading and writing numbers becomes unwieldy. In computer science, it is common to deal with numbers expressed as base two, base eight, and base 16.

Binary Numbers

Numbers in base two are called **binary numbers**. A binary number system requires two symbols; we choose 0 and 1. The positions within a binary number have values based on the powers of two, starting with 2^0 in the rightmost position. The digits of a binary number are called **bits**, which is a contraction of *bi*nary digi*ts*.

Consider the binary number 10101. This represents the value $1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$, or $1\times 16 + 0\times 8 + 1\times 4 + 0\times 2 + 1\times 1$, or 16 + 4 + 1, or 21.

Let's look at the same thing a different way:

1	0	1	0	1	
2^4	2^3	2^2	2^1	2^0	
16	8	4	2	1	
16		+ 4		+ 1	= 21

The first row is the binary number we want to examine. On the second row starting at the right, we write the power of two that corresponds to each position. The rightmost position is 2^0 and the power increases by one with each position to the left.

The third row is the decimal value of each of the powers of two. Notice that each of the numbers is twice the value of its predecessor. You simply start with one in the rightmost position and double each time you move left.

The decimal value of each digit is the digit itself, zero or one, multiplied by the power of two indicated by the digit's position. If the digit is a one, we copy the power of two to the fourth row; if the digit is a zero, we do not. This is equivalent to multiplying each positional value by the associated binary digit. Finally, we add across the bottom row to get the decimal value of the binary number.

As students of computer science, you will find it convenient to memorize the values of the first several powers of two, plus certain other values like 2^{10}, 2^{16}, and 2^{20}. You can easily find the value of any small power of two by starting with one you know and doubling until you reach the desired value. For example, if you know 2^{16}, you can find 2^{18} by doubling twice. If necessary, you can start with $2^0 = 1$ and double until you reach the value you need.

Usually the base of a number will be clear from the context; if necessary, the base is indicated by a subscript following the number, so we could write, for example, $1010_2 = 10_{10}$.

Why Binary?

We've gone to some length to describe a number system based only on zero and one. It is appropriate to digress briefly to explain why we choose to use binary numbers instead of the familiar decimal system when building computers. The answer is *reliability*. It turns out to be easy to design electronic circuits that can distinguish between on and off, or between positive and negative. It is much harder to build circuits that can reliably discriminate among several states. Seemingly identical electronic components will be slightly different even when new because of manufacturing tolerances. These differences are magnified with age and with differences in operating environment.

Consider a decimal computing machine in which we choose to represent the digits zero through nine with signals of zero through nine volts. We design the machine so that an actual signal of 6.8 volts is interpreted as the digit seven, allowing some tolerance for error. We also decide that 6.4 volts represents six. What do we do with a voltage of 6.5? Does this represent seven or six? With this scheme, a difference of 0.5 volts, or five percent, causes an error that cannot be resolved.

With binary numbers, we need only the symbols zero and one. If we say that zero volts represents a zero and ten volts represents a one, we can interpret anything less than 5.0 volts as zero, anything greater as one. This design can tolerate an error of nearly 50% and still produce correct results.

How High Can We Count?

With pencil and paper, we can write down any number we can imagine, using either the decimal or binary number systems. If we need more digits, we just write them. With computing machinery, the number of bits available for a number is likely to be fixed by the architecture. There may be ways of representing larger numbers, but these are likely to be painful. So, the question, "How high can we count given a fixed number of bits?" becomes an important one. Fortunately, it's easy to answer.

An *n*-bit binary number has 2^n possible values. This is easy to see. A single bit has two possible values, zero and one. With two bits, you get four values: 00, 01, 10, and 11. Three bits can generate eight distinct combinations, and so on. Of the 2^n possible values of an *n*-bit number, one will be all zeroes and represent the value zero. So the largest value that can be represented using an *n*-bit number is $2^n - 1$. An eight bit binary number has 2^8 (256) possible values, but since one of those values represents zero, the largest possible number is $2^8 - 1$ or 255.

There's another implication to the fact that in computers, binary numbers are stored in fixed-size "words." It is that each binary number must be the same number of bits. For unsigned integers, this is accomplished by padding on the left with zeroes.

Converting Decimal to Binary

Converting binary numbers to decimal is easy. We just write down the powers of two, which correspond to each digit in the binary number, then sum those for which the binary digit is a one. To convert a decimal number to binary, we express the decimal number as a sum of powers of two. These indicate which binary digits are ones; the rest will be zeroes. We will consider two ways to approach this product. The first is to find the sum of powers of two directly:

- Find the largest power of two smaller than the number to be converted. This is 2^n for a decimal number D where $2^{n+1} > D > 2^n$.
- Write a one in the 2^n place of the binary number and subtract 2^n from D. Call the remainder R.
- For each power of two from 2^{n-1} down to 2^0, if the power of two is less than or equal to R, write a one in the corresponding binary digit's position and subtract the power of two from R. Otherwise, write a zero in the binary digit's position.

Let's do an example: we will convert 277_{10} to binary. First, we count off powers of two: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512. We stopped at 512 because it is larger than 277, the number we're trying to convert. We "back down" one to 256, which is the largest power of two smaller

than 277. Now we're ready to begin converting. As you will see in the table, we write down 277, subtract 256 from it, and write a one in the first (leftmost) binary digit.

The remainder after subtracting 256 from 277 is 21, and the next power of two is 128. Because 128 is larger than 21, we write a zero in the next bit but do not subtract. We write zeroes for the 64 and 32 places, too. Sixteen is not larger than 21, so we write a one and subtract 16 from 21. Work your way through the last rows of the table yourself.

Power Of two	*Number or Remainder*	*Binary Digits*
	277	
256	-256	1
128	21	0
64		0
32		0
16	-16	1
8	5	0
4	-4	1
2	1	0
1	-1	1
	0	

We've converted 277_{10} to 100010101_2. Convert the binary number back to decimal to check the result.

Finding powers of two has an intuitive appeal but there is a feeling of "cut and try" to it. Another approach to converting decimal to binary is repeated division by two. The original decimal number is divided by two. The remainder, which will be zero or one, is the rightmost bit of the binary number. The quotient is again divided by two and the remainder becomes the next bit. This process continues until a quotient of zero is produced.

As an example of the division by two method, we will convert the number 141_{10} to binary. The center column of numbers is the original number, 141, and the quotients that are generated by successive division by two. The right column is the remainder from each division.

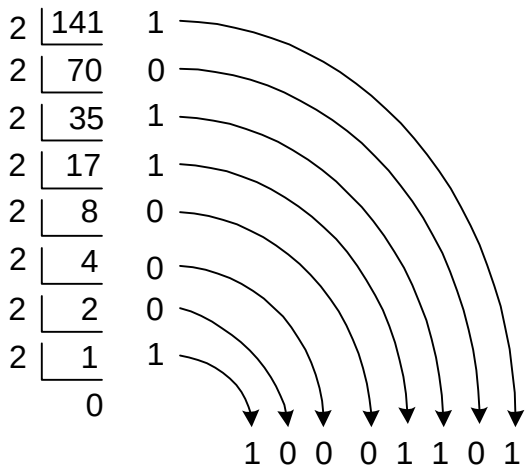

Figure 0. Converting decimal to binary by repeated division by two.

It is important to notice that the *first* remainder becomes the *rightmost* bit in the resulting binary number. We have converted 141_{10} to 10001101_2 by dividing by two and copying the remainders to the binary number. The first by two determines whether the number is even or odd. The second determines whether there is a factor of two; the third determines whether there is a factor of four, and so on.

Hexadecimal Numbers

Binary numbers are essential for the computer scientist, but binary numbers more than a few digits long are difficult to transcribe accurately. The hexadecimal (base 16) and octal (base 8) number systems can be used as number systems in their own right, but in computer science they are most often used as a shorthand for binary numbers. Since the bases of both systems are powers of two, translating between either base and binary is easy.

Like decimal and binary numbers, the **hexadecimal**, or base 16 number system is a positional number system. We know that there must be 16 symbols, and we choose 0, 1, …, 9, A, B, C, D, E, and F. Symbols 0 through 9 have the same unit values they have in the decimal system, but of course the positional multiplier is different. Hexadecimal (or *hex*) **A** has the value 10_{10}, **B** is 11_{10}, **C** is 12_{10}, **D** is 13_{10}, **E** is 14_{10}, and **F** is 15_{10}.

The positions in a hexadecimal number have as their values powers of 16, starting with 16^0 at the right, then 16^1, 16^2 or 256, 16^3 or 4096, and so on. Four hexadecimal digits let us represent numbers up to $15\times16^3 + 15\times16^2 + 15\times16^1 + 15$, or $15\times4096 + 15\times256 + 15\times16 + 15$, or 61,440 + 3840 + 240 + 15, or 65,535. This number would be represented as FFFF. A value of 0100_{16} is equal to 256_{10}.

Hexadecimal numbers can be used as a kind of shorthand for binary numbers, to avoid writing out long strings of ones and zeroes. Study the following table:

Binary	*Hex*	*Decimal*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	*Hex*	*Decimal*
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

As you can see, each hex digit is exactly equivalent to one of the possible combinations of four binary digits, so we could write 7_{16} instead of 0111_2. This works for numbers larger than four bits or one hex digit. $7A_{16}$ is equivalent to 01111010_2. Four hex digits let us express a 16-bit binary number in four symbols instead of 16.

It is common to use indications other than a subscript 16 to identify numbers as hexadecimal when it is not clear from the context. The following are all examples of indicators of hexadecimal numbers: x'7A', 0x7A, and 7Ax. In the Motorola 68000 assembler we will be using in CS2224, hexadecimal numbers are indicated by a dollar sign, so \$08 is 8_{16}.

Converting from hexadecimal to binary is easy: for each hex digit, write the four binary digits which have the same value. For example, to convert $4C_{16}$ to binary, we first write 0100 which is the binary equivalent of 4_{16}, then we write 1100 which is the binary equivalent of C_{16}, so $4C_{16} = 01001100_2$.

To convert a binary number to hexadecimal, start at the right and divide the number into groups of four bits. If the last group is fewer than four bits, supply zeroes on the left. (If the binary number contains a radix point, move from right to left on the integer part and from left to right on the fraction part.) Then, for each group of four bits, write the hex digit which has the same value.

For example, we will convert 1100011101 to hex.

```
0011 0001 1101
  3    1    D
```

We first divide the binary number into groups of four bits, working from the right. The

leftmost group had only two digits, so two zeroes were supplied on the left. The leftmost group of bits has the numeric value three, and we write a three as the hex digit for that group. The next group has the numeric value one. The rightmost group has the numeric value 13, or hex D. We have converted 1100011101_2 to $31D_{16}$.

Octal Numbers

The **octal** number system is a positional number system with base eight. Like hexadecimal numbers, octal numbers are most frequently used as a shorthand for binary numbers. Octal numbers are seen less often than hexadecimal numbers, and are commonly associated with Unix-like operating systems. The octal digits are 0, 1, 2, 3, 4, 5, 6, and 7. These digits have the same unit values as the corresponding decimal digits. Each octal digit encodes three binary digits as shown in the table below.

Binary	*Octal*	*Decimal*
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

Octal numbers are sometimes indicated by a leading zero.

Numbers are converted from octal to binary one digit at a time. For each octal digit, write down the three binary digits which represent the same value. To convert 124_8 to binary we write 001 010 100, and $124_8 = 001010100_2$. Sometimes octal numbers are used to represent eight-bit quantities. Representing eight bits requires three octal digits, which translate to nine bits. In this case, the leftmost bit, which should be a zero, is discarded.

Converting a binary number to octal follows the same process as converting a hexadecimal number except that the binary number is divided into groups of three bits. To convert 01001100_2 to octal, we divide the number into groups of three bits, starting from the right, then write down the corresponding octal digit for each group.

```
001 001 100
 1   1   4
```

In this example, it was necessary to supply an extra zero on the left to make three bits. We have converted 01001100_2 to 114_8.

To convert octal numbers to hexadecimal, or vice-versa, first convert to binary, then convert to the desired base by grouping the bits.

Binary Addition

Since we've talked about binary numbers as the basis for the electronic circuits for computers, it won't surprise you that we can do arithmetic on binary numbers. All the operations of ordinary arithmetic are defined for binary numbers, and they work much the same as you are used to. Let's look at the rules for binary addition:

				1
0	0	1	1	+1
+0	+1	+0	+1	+1
0	1	1	10	11

The first three of those don't require any explanation, but the fourth and fifth might. Look at the fourth rule and recall that the result is a binary number, so 10_2 represents one two and no ones, and in binary one plus one is two, exactly as you would expect. The rule says that 1+1=0, with one to *carry* to the next place. This is the same principle as carrying numbers in decimal addition, except that we carry when the partial sum is greater than one. The fifth rule adds three ones to get a one as the partial sum and another one to carry. We're written 1+1+1=3, which is what we expect.

Now we will add two binary numbers with more than one bit each so you can see how the carries "ripple" left, just as they do in decimal addition.

```
  1 1 1
  0 0 1 1 0
+ 0 1 1 1 1
  1 0 1 0 1
```

The three carries are shown on the top row. Normally, you would write these down as you complete the partial sum for each column. Adding the rightmost column produces a one with no carry; adding the next column produces a zero with one to carry. Work your way through the entire example from right to left. Then convert the addend, augend, and sum to decimal to verify that we got the right answer.

One can also express the rules of binary addition with a *truth table*. This is important because there are techniques for designing electronic circuits, which compute functions expressed by truth tables. The fact that we can express the rules of binary addition as a truth table implies that we can design a circuit which will perform addition on binary numbers, and that turns out to be true.

We only need to write the rules for one column of bits; we start at the right and apply the

rules to each column in succession until the final sum is formed. Call the bits of the addend and augend **A** and **B**, and the carry in from the previous column $\mathbf{C}_i$. Call the sum **S** and the carry out $\mathbf{C}_o$. The truth table for one-bit binary addition looks like this:

A	B	C_i	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This says if all three input bits are zero, both **S** and $\mathbf{C_o}$ will be zero. If any one of the bits is one and the other two are zero, **S** will be one and $\mathbf{C_o}$ will be zero. If two bits are ones, **S** will be zero and $\mathbf{C_i}$ will be one. Only if all three bits are ones will both **S** and $\mathbf{C_o}$ be ones.

That's all there is to binary addition. It's remarkably similar to decimal addition. As you would expect, the other arithmetic operations are also defined for binary numbers.

Negative Numbers

So far we have dealt only with non-negative integers — whole numbers zero or greater. For a computer to be useful, we must be able to handle binary negative numbers and fractions. For pencil-and-paper arithmetic we could represent signed binary numbers with plus and minus signs, just as we do with decimal numbers. With computer circuits, our only symbols are zero and one. We must devise a way of representing negative numbers using only zeroes and ones. There are four possible approaches: signed magnitude, ones complement, twos complement, and excess 2^{n-1}. The first three of these take advantage of the fact that in computers numbers are represented in fixed-size fields. The leftmost bit is considered the sign bit. The ones complement is formed by complementing each bit of the binary number. Again a zero in the sign bit indicates a positive number and a one indicates a negative number. Signed-magnitude and excess 2^{n-1} numbers are used in floating point, and will be discussed there. Ones complement arithmetic is obsolete.

In signed-magnitude representation, a zero in the sign bit indicates a positive number, and a one indicates a negative number. There is a problem with signed magnitude: is has two representations for zero. Consider an eight-bit word. 00000000 is "plus zero" and 10000000 is "minus zero." Since testing for zero is something that's done very frequently in computer programming, we would like to develop a better idea.

The better idea is something called **twos complement.** Twos complement numbers are used almost universally for integer representation of numbers in computers. The sign still resides in the leftmost bit, and positive numbers are treated just like the unsigned integers we've already used except that results are never allowed to flow over into the sign bit.

Let's go back to the basic idea of a binary number. In the binary number system, we can express any non-negative integer as the sum of coefficients of powers of two:

$$a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots + a_1\times 2^1 \ + a_0\times 2^0 \ = \sum_{i=0}^{n-1} a_i 2^i$$

One way of looking at two's complement numbers is to consider that the leftmost bit, or sign bit, represents a negative coefficient of a power of two and the remaining bits represent positive coefficients which are added back. So, an *n*-bit two's complement number has the form

$$-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Consider 10000000, an eight-bit two's complement number. Since the sign bit is a one, it represents -2^7 or –128. The remaining digits are zeroes, so 10000000 = –128. The number 10000001 is –128+1 or –127. The number 10000010 is –126, and so on. 11111111 is –128 + 127 or –1.

Now consider 01111111, also an eight-digit two's complement number. The sign bit still represents -2^7 or –128, but the coefficient is zero, and this is a positive number, +127.

The two's complement representation has its own drawback. Notice that in eight bits we can represent –128 by writing 10000000. The largest positive number we can represent is 01111111 or +127. Two's complement is *asymmetric about zero.* For any size binary number, there is one more negative number than there are positive numbers. This is because, for any binary number, the number of possible bit combinations is even. We use one of those combinations for zero, leaving an odd number to be split between positive and negative. Since we want zero to be represented by all binary zeros and we want the sign of positive numbers to be zero, there's no way to escape from having one more negative number than positive.

If you think of a two's complement number as a large negative number with positive numbers added back, you could conclude that it would be difficult to form the two's complement. It turns out that there's a method of forming the two's complement that is very easy to do with either a pencil or a computer:

- Take the complement of each bit in the number to be negated. That is, if a bit is a zero, make it a one, and vice-versa.
- To the result of the first step, add one as though doing unsigned arithmetic.

Let's do an example: we will find the two's complement representation of –87. We start with the binary value for 87, or 01010111. Here are the steps:

01010111	original number
10101000	each bit complemented, or "flipped"
+ 1	add 1 to 10101000
10101001	this is the two's complement, or –87.

We can check this out. The leftmost bit represents –128, and the remaining bits have positive values which are added back. We have –128 + 32 + 8 + 1, or –128 + 41 = –87. There's

another way to check this. If you add equivalent negative and positive numbers, the result is zero, so –87 + 87 = 0. Does 01010111 + 10101001 = 0? Perform the addition and see.

In working with two's complement numbers, you will often find it necessary to adjust the length of the number, the number of bits, to some fixed size. Clearly, you can expand the size of a positive (or unsigned) number by adding zeroes on the left, and you can reduce its size by removing zeroes from the left. If the number is to be considered a two's complement positive number, you must leave at least one zero on the left in the sign bit's position.

It's also possible to expand the size of a two's complement negative number by supplying one-bits on the left. That is, if 1010 is a two's complement number, 1010 and 11111010 are equal. 1010 is –8+2 or –6. 11111010 is –128+64+32+16+8+2 or –6. Similarly you can shorten a negative number by removing ones from the left so long as at least one one-bit remains.

We can generalize this notion. A two's complement number can be expanded by replicating the sign bit on the left. This process is called *sign extension.* We can also shorten a two's complement number by deleting digits from the left so long as at least one digit identical to the original sign bit remains.

Fractions

In ordinary decimal numbers, we represent fractions as negative powers of ten, and we mark the division between the integer and fraction parts with a "decimal point." The same principle applies to binary numbers. 0.1_2 is 2^{-1} or $^1/_2$. 0.11_2 is $2^{-1} + 2^{-2}$ or $^1/_2 + {}^1/_4 = {}^3/_4$. As with decimal fractions, not all fractional values can be represented exactly, but we can get arbitrarily close by using more fraction bits.

Unhappily, we don't have a symbol we can use for the "binary point." A programmer who wants to use binary fractions must pick a location for the implied binary point and scale all numbers to maintain binary point alignment.

Binary Arithmetic

Arithmetic is at the heart of the digital computer, and the majority of arithmetic performed by computers is binary arithmetic, that is, arithmetic on base two numbers. Decimal and floating-point numbers, also used in computer arithmetic, depend on binary representations, and an understanding of binary arithmetic is necessary in order to understand either one.

Computers perform arithmetic on fixed-size numbers. The arithmetic of fixed-size numbers is called finite-precision arithmetic. The rules for finite-precision arithmetic are different from the rules of ordinary arithmetic.

The sizes of numbers which can be arithmetic operands are determined when the architecture of the computer is designed. Common sizes for integer arithmetic are eight, 16, 32, and recently 64 bits. It is possible for the programmer to perform arithmetic on larger numbers or on sizes which are not directly implemented in the architecture. However, this is usually so painful that the programmer picks the most appropriate size implemented by the architecture.

This puts a burden on the computer architect to select appropriate sizes for integers, and on the programmer to be aware of the limitations of the size he has chosen and on finite-precision arithmetic in general.

We are considering binary arithmetic in the context of building digital logic circuits to perform arithmetic. Not only do we have to deal with the fact of finite-precision arithmetic, we must consider the complexity of the digital logic. When there is more than one way of performing an operation we choose the method which results in the simplest circuit.

Finite-Precision Arithmetic

Consider what it would be like to perform arithmetic if one were limited to three-digit decimal numbers. Neither negative numbers nor fractions could be expressed directly, and the largest possible number that could be expressed is 999. This is the circumstance in which we find ourselves when we perform computer arithmetic because the number of bits is fixed by the computer's architecture. Although we can usually express numbers larger than 999, the limits are real and small enough to be of practical concern. Working with unsigned 16-bit binary integers, the largest number we can express is $2^{16}-1$, or 65,535. If we assume a signed number, the largest number is 32,767.

There are other limitations. Consider again the example of three-digit numbers. We can add 200 + 300, but not 600 + 700 because the latter sum is too large to fit in three digits. Such a condition is called *overflow* and it is of concern to architects of computer systems. Because not all operations which will cause overflow can be predicted when a computer program is written, the computer system itself must check whether overflow has occurred and, if so, provide some indication of that fact.

Tanenbaum points out that the algebra of finite-precision is different from ordinary algebra, too. [TANE90] Neither the associative law nor the distributive law applies. Two examples from Tanenbaum illustrate this. If we evaluate the expression

$$a + (b - c) = (a + b) - c$$

using a = 700, b= 400, and c=300, the left-hand side evaluates to 800, but overflow occurs when evaluating $a + b$ in the right-hand side. The associative law does not hold.

Similarly if we evaluate

$$a \times (b - c) = a \times b - a \times c$$

using a = 5, b = 210, and c = 195, the left-hand side produces 75, but in the right-hand side, $a \times b$ overflows and distributive law does not hold.

These two examples show the importance of understanding the limitations on computer arithmetic. This understanding is important to programmers as well as designers of computers.

Addition of Signed Numbers

We have already considered addition of unsigned binary numbers. Binary addition of two's complement signed numbers can be performed using the same rules given above for unsigned addition. If there is a carry out of the sign bit, it is ignored.

Since we are dealing with finite-precision arithmetic, it is possible for the result of an addition to be too large to fit in the available space. The answer will be truncated, and will be incorrect. This is the overflow condition discussed above. There are two rules for determining whether overflow has occurred:

- If two numbers of opposite signs are added, overflow cannot occur.
- If two numbers of the same sign are added, overflow has occurred if and only if the result is of the opposite sign.

Subtraction

Addition has the property of being commutative, that is, $a+b = b+a$. This is not true of subtraction. 5 – 3 is not the same as 3 – 5. For this reason, we must be careful of the order of the operands when subtracting. We call the first operand, the number which is being diminished, the minuend; the second operand, the amount to be subtracted from the minuend, is the subtrahend. The result is called the difference.

```
   51     minuend
–  22     subtrahend
   29     difference.
```

It is possible to perform binary subtraction using the same process we use for decimal subtraction, namely subtracting individual digits and borrowing from the left. This process quickly becomes cumbersome as you borrow across successive zeroes in the minuend. Further, it doesn't lend itself well to automation. Jacobowitz describes the "carry" method of subtraction which some of you may have learned in elementary school, where a one borrowed in the minuend is "paid back" by adding to the subtrahend digit to the left. This means that one need look no more than one column to the left when subtracting. Subtraction can thus be performed a column at a time with a carry to the left, analogous to addition. This is a process which can be automated, but we are left with difficulties when the subtrahend is larger than the minuend or when either operand is signed.

Since we can form the complement of a binary number easily and can add signed numbers easily, the obvious answer to the problem of subtraction is to take the two's complement of the subtrahend, then add it to the minuend. We aren't saying anything more than that 51–22 = 51+(–22). Not only does this approach remove many of the complications of subtraction by the usual method, it means we don't have to build special circuits to perform subtraction. All we need is a circuit which can form the bitwise complement of a number and an adder.

Multiplication

A simple way to perform multiplication is by repeated addition. In the example below, we could add 42 to the product register 27 times. In fact, some early computers performed multiplication this way. However, one of our goals is speed, and we can do much better using the familiar methods we have learned for multiplying decimal numbers. Recall that the multiplicand is multiplied by each digit of the multiplier to form a partial product, then the partial products are added to form the total product. Each partial product is shifted left to align on the right with its multiplier digit.

```
   42   multiplicand
× 27    multiplier
 294    first partial product (42 × 7)
 84     second partial product (42 × 2)
1134    total product.
```

Binary multiplication of unsigned (or positive two's complement) numbers works exactly the same way, but is even easier because the digits of the multiplier are all either zero or one. That means the partial products are either zero or a copy of the multiplicand, shifted left appropriately. Consider the following binary multiplication:

```
    0111    multiplicand
× 0101      multiplier
  0111      first partial product (0111 × 1)
 0000       second partial product (0111 × 0)
0111        third partial product (0111 × 1)
0000        fourth partial product (0111 × 0)
0100011     total product.
```

Notice that no true multiplication is necessary in forming the partial products. The fundamental operations required are shifting and addition. This means we can multiply unsigned or positive integers using only shifters and adders.

With pencil-and-paper multiplication, we form all the partial products, then add them. It isn't necessary to do that; we could simply keep a running sum. When the last partial product is added, the running sum will be the total product. We can now state an algorithm for binary multiplication suitable for a computer implementation:

1. Set the product variable to zero
2. Set a counter to the number of bits in the multiplier
3. Shift the multiplier right one bit. If the bit shifted out was a one, add the multiplicand to the product variable.
4. Shift the multiplicand left one bit and go to Step 3; the algorithm terminates when all the digits of the multiplier have been examined.

Since the underlying arithmetic operation is addition, the possibility of overflow exists. We handle the possibility a little differently in multiplication. If two *n*-bit numbers are

multiplied, the largest possible product is 2*n* bits. Multiplication is usually implemented such that the register which receives the product is twice as large as the operand registers. In that case, overflow cannot occur.

Notice also that if the multiplier is *n* bits long, the multiplicand will have been shifted left *n* bits by the time the algorithm terminates. For this reason, multiplication algorithms make a copy of the multiplicand in a register 2*n* bits wide. Examination of the bits of the multiplier is often performed by shifting a copy of the multiplier right one bit at a time. This is because shift operations often save the last bit "shifted out" in a way that is easy to examine.

Unfortunately, this algorithm does not work for signed numbers. If the multiplicand is negative, the partial products must be sign-extended so that they form 2*n*-bit negative numbers. If the multiplier is negative, the situation is even worse; the bits of the multiplier no longer specify an appropriately-shifted copy of the multiplicand. One way around this dilemma would be to take the two's complement of negative operands, perform the multiplication, then take the two's complement of the product if the multiplier and multiplicand are of different signs. This approach would require a considerable amount of time before and after the actual multiplication, and so is usually rejected in favor of a faster but less straightforward algorithm. One such algorithm is Booth's Algorithm, which is discussed in detail in Stallings.

Division

As with the other arithmetic operations, division is based on the paper-and-pencil approach we learned for decimal arithmetic. We will show an algorithm for unsigned long division that is essentially similar to the decimal algorithm we learned in grade school. Let us divide 0110101 (53_{10}) by 0101 (5_{10}). Beginning at the left of the dividend, we move to the right one digit at a time until we have identified a portion of the dividend which is greater than or equal to the divisor. At this point a one is placed in the quotient; all digits of the quotient to the left are assumed to be zero. The divisor is copied below the partial dividend and subtracted to produce a partial remainder as shown below.

```
                    1        quotient
divisor  0101 )0110101       dividend
              0101
                 1           partial remainder
```

Now digits from the dividend are "brought down" into the partial remainder until the partial remainder is again greater than or equal to the divisor. Zeroes are placed in the quotient until the partial remainder is greater than or equal to the divisor, then a one is placed in the quotient, as shown below.

```
                  101
0101 )0110101
      0101↓↓
         110
```

The divisor is copied below the partial remainder and subtracted from it to form a new partial remainder. The process is repeated until all bits of the dividend have been used. The quotient is complete and the result of the last subtraction is the remainder.

```
          1010
0101 ) 0110101
       0101
         110
        0101
          11
```

This completes the division. The quotient is 1010_2 (10_{10}) and the remainder is 11_2 (3_{10}), which is the expected result. This algorithm works only for unsigned numbers, but it is possible to extend it to two's complement numbers. As with the other algorithms, it can be implemented using only shifting, complementation, and addition.

Limitations of Binary Integers

The natural arithmetic operand in a computer is the binary integer. However, the range of numbers that can be represented is limited by the computer's word size. We cannot represent very large or very small numbers. For example, in a computer with a 32 bit word, the largest signed number is $2^{31} - 1$. The range is further diminished if some bits of the word are used for fractions. There are techniques for performing integer arithmetic on groups of two or more words, but these are both painful for the programmer and consuming of CPU time.

It is not uncommon for very large and very small numbers to occur in the kinds of problems for which computers are used. These numbers do not lend themselves to representation in integer form, or integer and fraction form. Another approach is needed for problems whose variables are not small integers.

Scientific Notation

Scientists and engineers have developed a compact notation for writing very large or very small numbers. If we wrote it out, the mass of the sun in grams would be a two followed by 33 zeroes. The speed of light in meters per second would be a three followed by eight zeroes. These same numbers, when expressed in scientific notation, are 2×10^{33} and 3×10^{8}. Any number *n* can be expressed as

$$n = f \times 10^{e}$$

where *f* is a fraction and *e* is an exponent. Both *f* and *e* may be negative. If *f* is negative the number *n* is negative. If *e* is negative, the number is less than one.

The essential idea of scientific notation is to separate the significant digits of a number from its magnitude. The number of significant digits is determined by the size of *f* and the range of magnitude is determined by the size of *e*.

We wrote the speed of light as 3×10^8 meters per second. If that is not precise enough, we can write 2.997×10^8 to express the same number with four digits of precision.

Floating Point Numbers

Floating-point number systems apply this same idea – separating the significant digits of a number from its magnitude – to representing numbers in computer systems. Relatively small numbers for the fraction and exponent part provide a way to represent a very wide range with acceptable precision.

In the early days of computers, each manufacturer developed their own floating-point representation. These were incompatible. In some cases, they even produced wrong answers. Floating-point arithmetic has some subtleties which are beyond the scope of this paper.

In 1985, the Institute of Electrical and Electronic Engineers published IEEE Standard 754 for floating-point arithmetic. Virtually all general purpose processors built today have floating-point units which conform to IEEE 754. The examples in this paper describe IEEE 754 floating-point number formats.

Instead of using base ten and powers of ten like scientific notation, IEEE 754 floating-point uses a binary fraction and an exponent that is considered to be a power of two. The format of a single-precision floating-point number is shown in Figure 1. The leftmost bit indicates the sign of the number, with a zero indicating positive and a one indicating negative. The exponent occupies eight bits and is also signed. A negative exponent indicates that the fraction is multiplied by a negative power of two. The exponent is stored as an excess 127 number, which means that the value stored is 127 more than the true value. A stored value of one indicates a true value of -126. A stored value of 254 indicates a true value of +127. Exponents values of zero or 255 (all ones) are used for special purposes described later. The fraction part is a 23-bit binary fraction with the binary point assumed to be to the left of the first bit of the fraction. The approximate range of such a number is $\pm 10^{-38}$ to $\pm 10^{38}$. This is substantially more than we can express using a 32-bit binary integer.

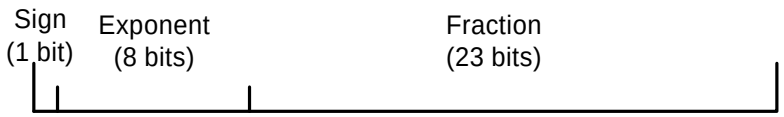

Figure 1. Format of an IEEE 754 single-precision floating-point number.

Normalized Numbers

We represented the speed of light as 2.997×10^8. We could also have written 0.2997×10^9 or 0.02997×10^{10}. We can move the decimal point to the left, adding zeroes as necessary, by increasing the exponent by one for each place the decimal point is moved. Similarly, we can compensate for moving the decimal point to the right by decreasing the exponent. However, if we are dealing with a fixed-size fraction part, as in a computer implementation, leading zeroes in the fraction part cost precision. If we were limited to four digits of fraction, the last example

would become 0.0299×10^{10}, a cost of one digit of precision. The same problem can occur in binary fractions.

In order to preserve as many significant digits as possible, floating-point numbers are stored such that the leftmost digit of the fraction part is non-zero. If, after a calculation, the leftmost digit is not significant (*i.e.* it is zero), the fraction is shifted left and the exponent decreased by one until a significant digit – for binary numbers, a one – is present in the leftmost place. A floating-point number in that form is called a **normalized** number. There are many possible unnormalized forms for a number, but only one normalized form.

Storing numbers in normalized form provides an opportunity to gain one more significant binary digit in the fraction. If the leftmost digit is known to be one, there is no need to store it; it can be assumed to be present. IEEE 754 takes advantage of this; there is an implied one bit and an implied binary point to the left of the fraction. To emphasize this difference, IEEE 754 refers to the fractional part of a floating-point number as a **significand**.

Range of Floating Point Numbers

Although the range of a single-precision floating-point number is $\pm 10^{-38}$ to $\pm 10^{38}$, it is important to remember that there are still only 2^{32} distinct values. The floating-point system can not represent every possible real number. Instead, it approximates the real numbers by a series of points. If the result of a calculation is not one of the numbers that can be represented exactly, what is stored is the nearest number that *can* be represented. This process is called **rounding**, and it introduces error in floating-point calculations. Since rounding down is as likely as rounding up, the cumulative effect of rounding error is generally negligible.

The spacing between floating-point numbers is not constant. Clearly, the difference between 0.10×2^1 and 0.11×2^1 is far less than the difference between 0.10×2^{127} and 0.11×2^{127}. However, if the difference between numbers is expressed as a percentage of the number, the distances are similar throughout the range, and the relative error due to rounding is about the same for small numbers as for large.

Not only cannot all real numbers be expressed exactly, there are whole ranges of numbers that cannot be represented. Consider the real number line as shown in Figure 2. The number zero can be represented exactly because it is defined by the standard. The positive numbers that can be represented fall approximately in the range 2^{-126} to 2^{+127}. Numbers greater than 2^{+127} cannot be represented; this is called **positive overflow**. A similar range of negative numbers can be represented. Numbers to the left of that range cannot be represented; this is **negative overflow**.

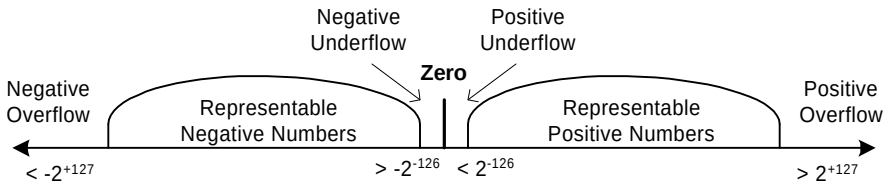

Figure 2. Zones of floating-point numbers along the real number line.

There is also a range of numbers near zero that cannot be represented. The smallest positive number that can be represented in normalized form is 1.0×2^{-126}. The condition of trying to represent smaller numbers is called **positive underflow**. The same condition on the negative side of zero is called **negative underflow**. Prior to IEEE 754, manufacturers just set such results to zero or signaled an error exception. The IEEE standard provides a more graceful way of handling such conditions: the requirement that numbers be normalized is relaxed near zero. The exponent is allowed to become zero, representing 2^{-127}, the implicit one at the left of the binary point becomes a zero, and the fraction part is allowed to have leading zeroes. Such a number approaches zero with increasing loss of significant digits.

More on IEEE 754

In addition to the single-precision floating-point numbers we have discussed, IEEE 754 specifies a double-precision number of 64 bits. The double-precision format has one bit of sign, eleven bits of exponent, and 52 bits of fraction. This gives it a range of approximately $\pm 10^{-308}$ to $\pm 10^{308}$. There is also an 80-bit extended-precision representation used mainly internally to floating-point processors.

Each format provides a way to represent special numbers in addition to the regular normalized floating-point format. An exponent of zero with a non-zero fraction is the denormalized form discussed above to handle positive and negative overflow. A number with both exponent and fraction of zero represents the number zero. Both positive and negative representations of zero are possible.

Positive and negative overflow are handled by providing a representation for infinity. This is a positive or negative sign bit, an exponent of all ones, and a fraction of zero. This representation is also used for the result of division by zero. Arithmetic operations on infinity behave in a predictable way.

Finally, all ones in the exponent and a non-zero fraction represents **Not a Number**, also called **NaN**. Arithmetic on NaNs also behaves in a predictable fashion.

Summary

The binary number system is a positional number system based on powers of two. We choose a two-valued system because it can be implemented reliably with electronic circuits. Fractions depend on an implied radix point (binary point) and are expressed as the sum of negative powers of two: ½, ¼, and so on. Negative numbers can be expressed using signed magnitude, excess 2^{n-1}, or twos complement. For integer arithmetic, we choose twos complement. Hexadecimal (base 16) and octal (base 8) numbers are used in computer science as shorthand for binary numbers.

The rules of binary addition can be expressed as a truth table, therefore it is possible to build an electronic circuit that can perform binary addition. Subtraction is performed by taking the twos complement of the subtrahend and adding it to the minuend. Other arithmetic circuits are also possible.

Computer arithmetic is finite-precision arithmetic. The algebra of finite-precision arithmetic is different from ordinary algebra, especially near the boundaries of number size. The rules of binary arithmetic are similar to the rules for decimal arithmetic. Complications arise in multiplication and division of signed numbers.

It is difficult to express very large or very small numbers using only binary integers and binary fractions. The key idea behind floating point numbers is separating the precision of a number from its magnitude. The floating point number system trades precision for range. The choice between floating point and integer representations of numbers is a system design decision.

Exercises

BN-1. Convert the following unsigned binary numbers to decimal:

10101
10000
01111
11111

BN-2. Convert the following decimal numbers to binary:

32
127
255
275

BN-3. Which of the following binary numbers are even? How can you tell by inspection whether a binary number is even or odd?

101010
101011
111111
111110

BN-4. Convert 42_{10} to binary, negate it by forming the two's complement, and compute –42 + 42 using binary numbers. What answer do you expect and why?

BN-5. Write the binary equivalent of $3^1/_{16}$. To make your work clear, use a period as a "binary point."

BN-6. Using only the methods and material presented above, suggest a strategy for performing binary subtraction.

BN-7. Which of the following are valid hexadecimal numbers?

BAD DEAD CABBAGE ABBA 127

BN-8. Convert the following binary numbers to hexadecimal.

00000000
10001000
00011010
11111111

BN-9. Convert the binary numbers in Exercise 8 to octal.

BN-10. Convert the following octal numbers to both binary and hexadecimal.

377
127
4066
01

BN-11. Explain why there is one more negative number than there are positive numbers for a given number of bits in two's complement representation.

BN-12. Add the following unsigned binary numbers:

00101011	01100001	01000010	01111111	01111111
01001100	00001111	01010101	00000001	00000010

BN-13. Convert the following unsigned binary numbers to two's complement:

01101111	00011000	00110101	01011011	01011000

BN-14. Compute 01101010×01110111. Show your work. *(Beware. This is hard.)*

BN-15. Divide 01011 into 011011000. Show your work.

BN-16. Convert 137 and 42 to binary. Compute the binary equivalent of $137 - 42$. Show your work.

BN-17. Compute $42 - 137$ in binary. Show your work.

BN-18. Rewrite the following numbers in normalized form and express the answer using decimal numbers in scientific notation. State the rule for normalizing a floating-point number.

a) 6.02257×10^{23} b) 0.0005×10^{6} c) 427×10^{0} d) 3.14159

BN-19. Add the following pairs of numbers; express the sums in normalized form using scientific notation. State the rule for addition of floating point numbers.

a) $0.137 \times 10^{2} + 0.420 \times 10^{2}$ b) $0.288 \times 10^{3} + 0.650 \times 10^{4}$

BN-20. Multiply the following pairs of numbers; express the products in normalized form. State the rule for multiplication of floating point numbers.

a) $(0.137 \times 10^{2}) \times (0.420 \times 10^{2)}$ b) $(0.288 \times 10^{3}) \times (0.650 \times 10^{4})$

BN-21. Add this pair of numbers. Normalize the result and write the fraction as a finite-precision number with four decimal digits of precision, *i.e.* four digits to the right of the decimal point. Explain why you got the result you did. What observation can you make

about addition and subtraction of floating-point numbers?

$0.5000 \times 10^8 + 0.4321 \times 10^2$

BN-22. Convert 0.640×10^2 to an IEEE 754 single-precision floating point number; separate the various parts of the number with vertical lines and label them as shown in Figure 1. Show your work.

BN-23. Convert the number ¾ to an IEEE 754 single-precision floating point number; separate the various parts of the number with vertical lines and label them as shown in Figure 1. Show your work.

BN-24. Explain the concept of *underflow*.

BN-25. What is the essential idea behind scientific notation and floating-point numbers?

Bibliography

Jacobowitz, Henry, *Computer Arithmetic*, John F. Rider Publisher, 1962.

Stallings, William, *Computer Organization and Architecture, Fourth Edition,* Prentice-Hall, 1996.

Tanenbaum, Andrew S., *Structured Computer Organization, Third edition,* Prentice-Hall, 1990.

Youse, Bevan K., *The Number System,* Dickenson Publishing Company, 1965.