COMPUTER SYSTEMS LABORATORY
WASHINGTON UNIVERSITY
ST. LOUIS, MO. 63110

LINC Document No. 37

April 1965

Introduction to Binary Numbers and Binary Arithmetic

Irving H. Thomae

Abstract

The introduction includes number base conversion procedures, ones' complement arithmetic, binary addition, multiplication, and division.

The Computer Systems Laboratory is supported in part by the Advanced Research Projects Agency of the Department of Defense under contract SD-302, and in part by the Division of Research Facilities and Resources of the National Institutes of Health under grant FR-00396.

AN INTRODUCTION TO BINARY NUMBERS AND BINARY ARITHMETIC

From a pragmatic viewpoint, any numerical notation or number system is merely a code for representing quantities - statements about "how many." In other words, a number system is a language in which topics like counting and arithmetic can be discussed conveniently. We may not expect that such a language will be unique. There may be, and in fact there is, a whole family of number systems, and the particular number system used by a particular digital computer is, in this sense, that computer's "language." While we do arithmetic in the decimal system, LINC and many other computers use the binary number system. Before explaining binary, let us recall what is meant by a "decimal" system.

Everyone learns in grade school that a decimal number such as 7,432 represents "two ones, three tens, four hundreds, and seven thousands." Reading from right to left, in other words, the successive columns are ascending powers of ten: 10^0(=1), 10^1, 10^2, 10^3, etc. The system is based on ten, as the name implies, and there are ten different symbols used, 0 through 9.

But there is nothing to prevent us from using some other number as a base, or radix. The addition tables, etc., would have to be rewritten, since the same quantities would be differently encoded, but two plus two, by any name, must still be four, even though we may write "$\beta + \beta = \delta$," or "10 + 10 = 100." In this paper, we will use spelled-out names of numbers to refer to the quantities they represent, independent of particular number systems. Thus, "two" is an invariant. It always means the number of dots in this circle: (:) . The mark "2," however, is undefined in some number systems, including binary; and the mark "10" has a different meaning in each different number system.

The binary system is based on the radix two. This means that there need be only two symbols, conventionally taken as 0 and 1. This is why computers use it, since an on-off, or two-state, device is much simpler than a ten-state device.

Reading from the right end of a binary number, successive columns are ones, twos, fours, eights, etc., - $2^0(=1)$, 2^1, 2^2, 2^3, etc. - ascending powers of two. Thus, the number 11001 represents, reading from right, "one one plus no twos plus no fours plus one eight plus one sixteen," or twenty-five. It must be admitted that binary numbers are less compact than decimal, but for computer use, we will see that this disadvantage is far outweighed by the advantages.

Compare the following numbers:

DECIMAL			BINARY						
10^2	10^1	10^0	2^6	2^5	2^4	2^3	2^2	2^1	2^0
hundreds	tens	ones	sixty-fours	thirty-twos	sixteens	eights	fours	twos	ones
0	0	1	0	0	0	0	0	0	1
0	0	2	0	0	0	0	0	1	0
0	0	3	0	0	0	0	0	1	1
0	0	4	0	0	0	0	1	0	0
0	0	5	0	0	0	0	1	0	1
0	0	6	0	0	0	0	1	1	0
0	0	7	0	0	0	0	1	1	1
0	0	8	0	0	0	1	0	0	0
0	0	9	0	0	0	1	0	0	1
0	1	0	0	0	0	1	0	1	0
0	1	1	0	0	0	1	0	1	1
0	1	2	0	0	0	1	1	0	0
0	1	3	0	0	0	1	1	0	1
0	1	4	0	0	0	1	1	1	0
0	1	5	0	0	0	1	1	1	1
0	2	0	0	0	1	0	1	0	0
0	2	6	0	0	1	1	0	1	0
0	6	3	0	1	1	1	1	1	1
0	6	4	1	0	0	0	0	0	0
1	1	7	1	1	1	0	1	0	1

Fractions are represented in the same way. Columns to the right of a decimal point represent increasingly negative powers of ten (tenths, hundredths, thousandths, or 10^{-1}, 10^{-2}, 10^{-3}, etc.). Similarly, to the right of the binary point we have halves, quarters, eighths — 2^{-1}, 2^{-2}, 2^{-3}, etc. Any fraction can be represented in this form. For instance, .1011 is "1 half plus no fourths plus 1 eighth plus 1 sixteenth," or eleven sixteenths, .6875. Similarly, 101.011 is 5 3/8, or 5.375.

The tables of powers of two attached, especially the positive powers, should be at one's mental finger tips.

We will often refer to the columns of a binary number as "bits." Strictly speaking, a "bit" is any item of yes-or-no information, but in practice this distinction will usually be unimportant. We also frequently name a bit in a number by the power of two represented. Thus, the "0-bit" is the right-most bit, representing 2^0 or 1; "bit 4" is the fifth from the right, representing 2^4 or sixteen.

The numbers listed on the preceding page illustrate two important points about number systems. Consider first the counting process with respect to one column of a decimal number. As 1's are added, the column "fills up" until 9 is reached. This is the maximum capacity, so when the next 1 is added we must return our column to 0 and carry 1 to the next higher column.

In the binary system, however, a given column's value may only be 0 or 1, so every second time a bit receives a 1 it must clear and carry to the next. For a given bit, then, counting is a process of alternating, or "flipping," between "0" and "1," originating (sending out) a carry every time it reverts from "1" to "0."

In either system, when all columns are filled to capacity, the next "1" added will require a new column. In decimal, we see this happen in going from 9 to 10, or 99 to 100; in binary this happens, for example, between seven and eight, fifteen and sixteen, or sixty-three and sixty-four.

Notice also that it is always extremely easy to multiply by a power of the radix. In decimal, we may multiply by ten by shifting the entire number left one place, or by 10^n by shifting left n places. Correspondingly, in binary we can multiply by two by shifting left one place, or by 2^n by shifting left n places. Compare three, six, and twelve in binary with three, thirty, and three hundred in decimal; or thirteen and twenty-six in binary with thirteen and one hundred-thirty in decimal:

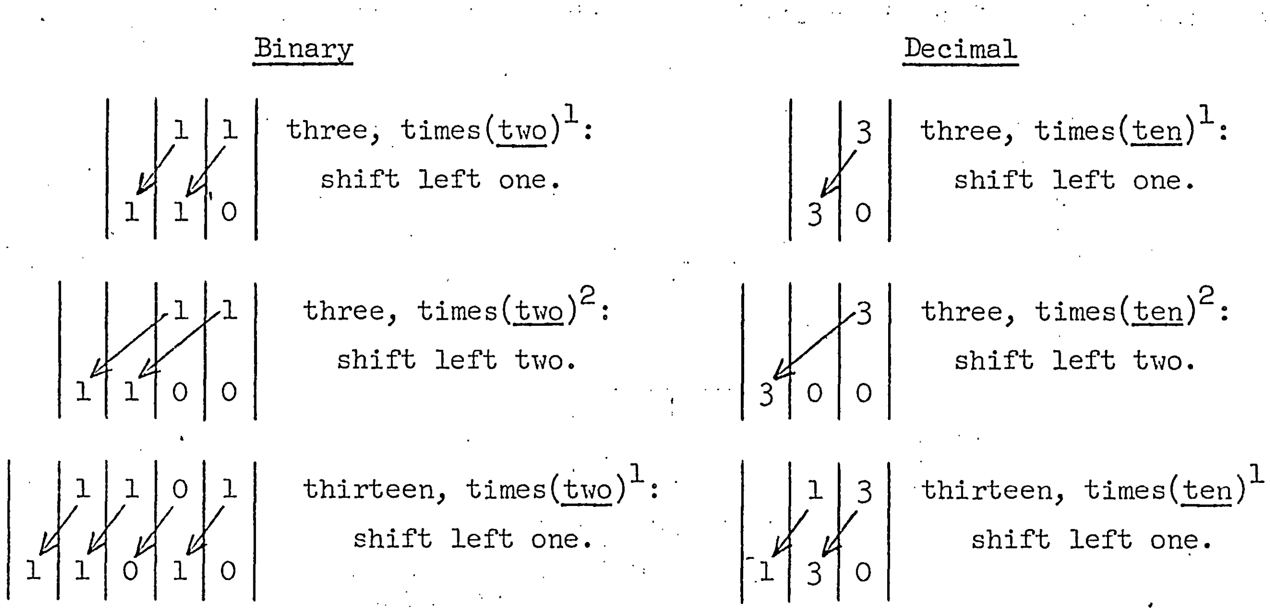
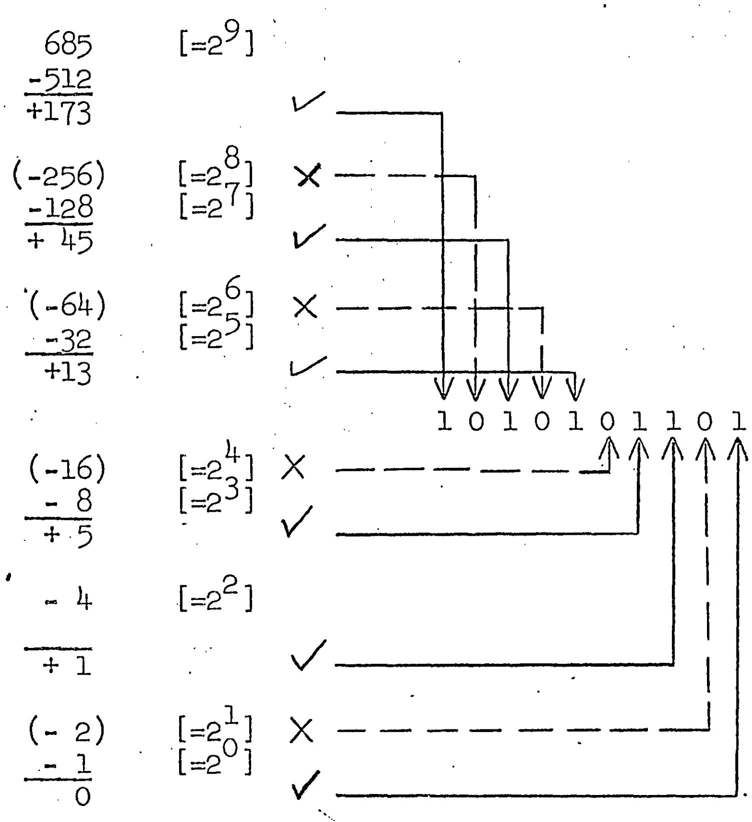


Figure 1. Multiplication by the radix as a shifting process.

The process of "translating," or reconverting from binary to decimal is obvious; it might be helpful to describe decimal-to-binary conversion explicitly. Starting with the largest possible power of two, we attempt to subtract successively smaller powers of two from the current remainder and get a positive result. For each successful subtraction a 1 is recorded, otherwise, a 0. Thus the decimal number 685 converts as follows:



ADDITION

Binary addition is very simple. The basic table has only four entries, compared to one hundred in decimal:

Decimal:

+	0	1	2	3 ...
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

etc.

Binary:

+	0	1
0	0	1
1	1	10

Notice that for a particular bit, 1+1 gives 0 with a carry. This we have just seen in considering the counting process. Indeed, from the viewpoint of a particular bit, addition is always basically a counting process.

We may illustrate with an example.

```
                          1 1 0 1 0
                          1 0 1 1 0
         column sums      0 1 1 0 0
    primary carries     1     1
    first carry sum     1 0 1 0 0 0
  secondary carries         1
secondary carry sum     1 0 0 0 0 0
   tertiary carries       1
             result     1 1 0 0 0 0
```

Of course, in doing the sum one would normally add in the carries as they appeared, but this form shows what is going on more clearly.

MULTIPLICATION

Binary multiplication is, if anything, even simpler than addition. The basic table is:

	0	1
0	0	0
1	0	1

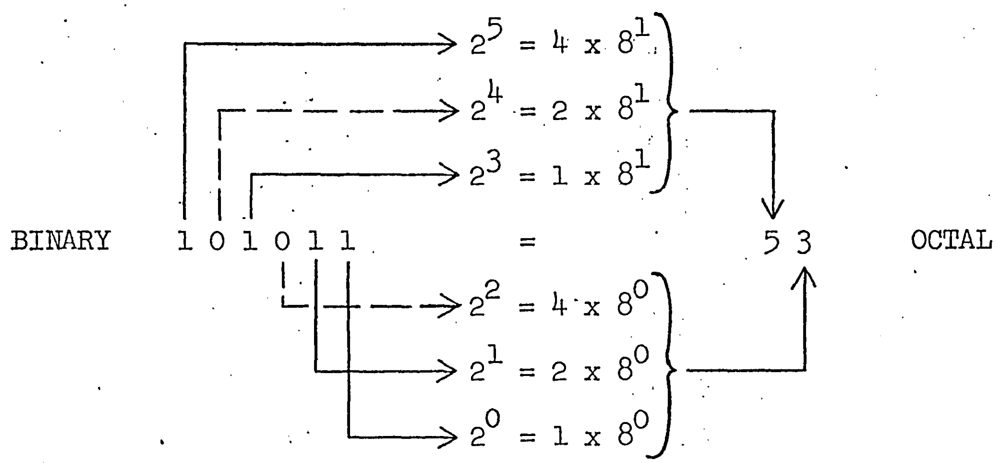
1 times 1 is 1, and anything else is 0. It is easy enough to combine this with the standard methods. Compare decimal and binary multiplication:

a.
```
    356
    708
   2848
  0000
 2492
 252048
```

b.
```
   A          Decimal:    29          Binary:      11101
 x B                      21                       10101
   C                      29                       11101
                         58                       00000
                         609                     11101
                                                00000
                                               11101
                                               1001100001
```

Of course we normally omit the rows of zeros. But notice that in binary, multiplication by each multiplier digit is reduced to the decision whether or not to copy the shifted multiplicand. So, in this example, we take $(1)\cdot[(2^0)(A)] + (0)\cdot[(2^1)(A)] + (1)\cdot[(2^2)(A)] + (0)\cdot[(2^3)(A)] + (1)\cdot[(2^4)(A)]$, or in all, twenty-one times A, or B times A, which is exactly what we want.

By now it should be quite clear that binary arithmetic is both simple and cumbersome. There is available a very convenient way to avoid the awkward chains of ones and zeros. We introduce another number system, octal, which is based on eight. This system has 8 characters, for which we use the Arabic numerals 0 through 7. Interconversion between binary and octal can be done by sight, since a group of three binary digits is completely equivalent to one octal digit. This, of course, is so because $8 = 2^3$. So we have this equivalence:



BINARY	OCTAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

See also the tables of powers of two.

A binary number grouped in sets of 3 bits each can thus be read off in octal, a far more convenient notation.

Compare these numbers:

DECIMAL	BINARY	OCTAL
5	101	5
8	1 000	10
9	1 001	11
12	1 100	14
15	1 111	17
16	10 000	20
29	11 101	35
32	100 000	40
40	101 000	50
54	110 110	66
100	1 100 100	144
3739	111 010 011 011	7233

One can clearly also do arithmetic in octal, and although LINC actually operates in binary, it is customary and proper to use octal almost exclusively when programming and operating the computer.

The addition and multiplication tables are perfectly straightforward, except that the digits 8 and 9 are missing. For convenience they have been appended. A brief glance at them will indicate that the same counting processes hold as in any other system: when the capacity of a particular column is exceeded, clear it and carry.

One pitfall to be avoided carefully is that octal numbers look, superficially at least, much like ordinary numbers. The complete absence of 8's and 9's may not be immediately evident, and much confusion can result. Therefore, wherever ambiguity seems possible, numbers will be written with the subscript "8" or "10" to indicate "octal" or "decimal."

COMPLEMENT ARITHMETIC

Up to this point we have assumed that we could represent any number, no matter how large. In a digital computer, each digit is represented by some kind of physical hardware, such as a wire, a "flip-flop," or a light-bulb. The computer thus necessarily has a finite range of numbers, determined by the number of bits available. For instance, the LINC has twelve bits, so the largest possible number which could normally be represented is 4095_{10} (7777_8). If we add 1 to this number, we ought to get $1\ 000\ 000\ 000\ 000_2$ (10000_8), but only the rightmost twelve bits are represented, so the next number in sequence is 0. More simply, but in exactly the same way, if we had three bits available the biggest number possible would be 111 (7_8), followed again by 0. In that closed system 1, 9, and 17 are completely equivalent. They are said to be equal "modulo 8"; in other words, they all give the same remainder when divided by 8.

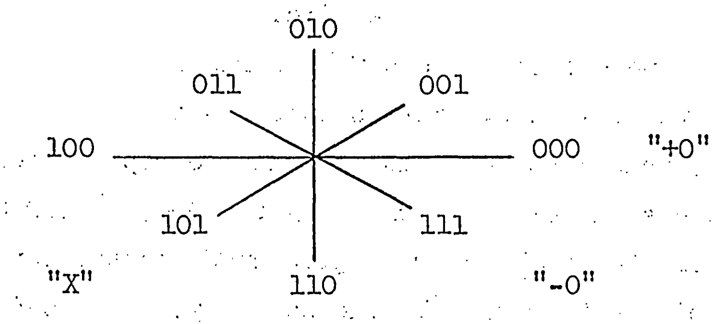
Suppose now we arrange things so that, in our simple closed system, the extra carry generated when we add 1 to 7 is brought around the end - as an "end-carry" - and added in at the 0-bit. When this occurs, the result is 001 - as if we had added 1 to 0 instead of to 7:

```
3-bit closed system:    1 1 1        0 0 0
                      +     1      +     1
                     (1)0 0 0        0 0 1
end-around carry        \---->1
                        0 0 1
```

```
12-bit
closed
system:   1 1 1  1 1 1  1 1 1  1 1 1     0 0 0  0 0 0  0 0 0  0 0 0
        +                          1   +                          1
       (1)0 0 0  0 0 0  0 0 0  0 0 0     0 0 0  0 0 0  0 0 0  0 0 1
end carry \----------------------->1
          0 0 0  0 0 0  0 0 0  0 0 1
```

In fact, in end-carry n-digit binary arithmetic, the number composed of n 1's behaves exactly like 0. It is customarily referred to as "minus zero" to distinguish it from the more familiar form which is then referred to as "plus zero."

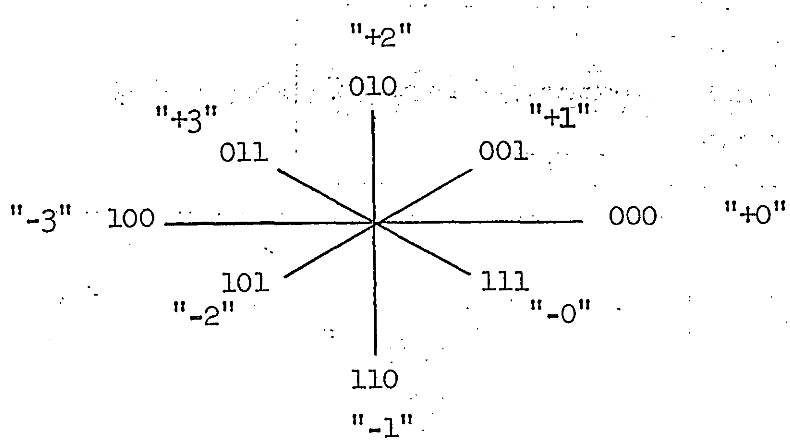
Continuing in our 3-digit end-carry system, we consider the following arrangement of numbers:



These are, of course, all the possible numbers we may have.

Define positive rotation counter clockwise (↺). Counting around the wheel, the position "X" is then plus five. However, if we start at "-0" and count clockwise, it becomes minus two. Try adding this "-2" to +3, using end-around carry. (Count around the wheel treating +0 and -0 as one point.) The result is +1.

It will be found that the following designations can be assigned:



These definitions permit subtraction, if we limit ourselves to a system of just the numbers 0,1,2,3. In effect we have made the leftmost bit represent the sign of the number. If it is a "1", the number is presumed to be negative, and is counted down from minus zero (111), instead of up from plus zero (000).

Notice in the diagram that -2, (101), has 1's where +2, (010), has 0's, and vice versa. The process of replacing 1's with 0's and 0's with 1's is called complementing, and in a 3-bit system, 101 is the complement of 010. So, to encode a negative number, we complement the corresponding positive binary number.

Returning now to the LINC's 12-bit numbers, we restrict ourselves to numerical use of 11 bits. To code a negative number, we complement it. Here, too, complementing turns out to be equivalent to counting down from -0. Since the binary magnitude of numbers will have a zero in the leftmost bit, complementing renders bit 11 a one. This bit is therefore a sign indicator, and the LINC can "find out" whether a number is positive or negative simply by testing it. If the 11-bit is 0, the number is positive; if 1, it is negative. Furthermore, if we never try to give numerical significance to the sign bit, we can subtract numbers by adding their complements, using end-around carry.

Example:

Decimal	Binary	Octal
1978	011 110 111 010	3672
-1568	-011 000 100 010	-3042
410		
	011 110 111 010	3672
Add complement of subtrahend:	100 111 011 101	4735
	(1) 000 110 010 111	(1) 0627
End-around carry:	→ 1	→ 1
	000 110 011 000	0630

Notice that in octal, we may form a complement by subtracting each digit of the number to be complemented from 7. Then, by using end-around carry, we get the same result as we did in binary.

Counting now is rather odd if one exceeds the allowed eleven numerical bits. For, the next number after 011 111 111 111, (3777_8), the largest positive number, is 100 000 000 000, the biggest (in magnitude) negative number ($4000_8 = -3777_8$).

DIVISION

The last and perhaps most confusing operation is long division. Division is the process of finding out how many times the divisor is contained in the dividend. At bottom, therefore, it is an elaborate method of subtracting and counting, although the familiar procedures tend to obscure this.

For example, in the simple division $2\overline{)9}$, we all know the quotient is 4 and the remainder is 1. But if we didn't know that, we could find out by subtracting 2 repeatedly, counting the number of times we were successful. When, after such a series, the result turns up negative, we know we have subtracted once too often. The correct remainder is then recovered by adding back the divisor once, and the correct quotient is one less than the total number of subtractions we have executed.

Let us illustrate this with another very simple example, $3\overline{)14}$:

Operations	Count	
14		
- 3		
11	1	
- 3		
8	2	
- 3		
5	3	
- 3		
2	4	
- 3		
- 1	5	Negative result, so add back the divisor.
+ 3		
2	5-1 = 4	

Quotient 4, remainder 2

This obviously is impractical for large quotients, and so the familiar long division uses a very important shortcut.

Consider this example:

```
    142
28/3979
   28
   117
   112
     59
     56
      3
```

In the first step, we actually divide not by 28, but by 2800. To obtain the right answer for this problem, that result is automatically multiplied by 100 when it is put in the third column from right in the answer. That is, the "1" in the quotient represents $\left(\frac{3979}{28 \times 100} \right) \times 100$.

The remainder obtained is really 1179. In the second set of steps 1179 is divided by 28×10^1, and the result, 4, is multiplied by 10^1 when it is put in the second quotient column. Finally, 59 is divided by 28×10^0, and the result, 2, is multiplied by 1 and placed in the right-most column to give the answer 142.

The same "shortcut" of taking out the divisor "a hundred at a time" can be used in the subtraction method, too, as follows:

Operation	Comments	Count	Quotient
28/3979			
3979	Divide by 28 x 100.		
-2800			
1179		1	
-2800			
-1621	Negative, so add back	2	
+2800	the divisor.	2-1 = 1	1 x 100
1179			
- 280	Now use 28 x 10 as		
899	divisor.	1	
- 280			
619		2	
- 280	Positive remainder,		
339	so keep going.	3	
- 280			
59		4	
- 280			
- 221	Negative, so add back	5	
+ 280	the divisor again.		
59	This count is the 10's		
	digit of the quotient.	5-1 = 4	4 x 10
- 28	Now use 28 x 1 as divisor.		
31		1	
- 28			
3	Positive, so keep going.	2	
- 28			
- 25	Negative - back up.	3	
+ 28			
+ 3		3-1 = 2	2 x 1

The final quotient is 100 + 40 + 2 = 142, and the final remainder is 3.

Of course, the process of multiplying the divisor by various powers of ten is customarily accomplished purely by shifting it along beneath the dividend, and the zeros have been filled in here purely for clarity.

Notice that, if we wished, we could shift the dividend left instead of shifting the divisor right. Their positions relative to each other will be unchanged and if we don't get our quotient score-keeping mixed up, the result will be exactly the same. This is convenient in a finite number system like the

LINC's, where shifting a number may mean discarding digits. It is then clearly better to discard higher-order current remainder bits, which are 0 anyway when they fall due for shifting "off into space."

We will not attempt to do binary division with complemented numbers. If either the divisor or the dividend is negative, we will re-complement it before dividing, and remember the sign.

As an illustration, let us find the quotient of two 6-bit binary fractions. As usual, the left-most bit is the sign; and we assume also that the binary point lies directly to its right. With the binary points of both divisor and dividend in the same place, this is completely equivalent to dividing a pair of integers. However, use of the left-most bit as sign-bit requires that the divisor be greater than the dividend. For, if the quotient came out equal to or greater than 1, it would then be interpreted as a negative number, and this clearly would be wrong, since as we have already said, both the divisor and the dividend will always be positive.

Example: $\frac{1.10001}{0.10100} = ?$

First, we note that the numerator is negative, so we must complement it, and remember to complement the quotient we get when we are all done.

So we have 0.10100/$\overline{0.01110}$

First subtraction: (by adding complement of divisor)	001110 101011 111001	Negative: Record 0 in quotient, add back divisor. (This was expected, since the first quotient digit is a sign bit.)
add back:	+010100 001110	
Shift remainder left one:	011100	
Subtract:	101011 001000	Positive - record 1 in quotient, continue.
Shift remainder left one:	010000	
Subtract:	101011 111011	Negative - add back divisor, record 0 in quotient.
Shift:	100000	
Subtract:	101011 001100	Positive - record 1 in quotient, continue.
Shift:	011000	
Subtract:	101011 000100	Positive - record 1, continue.
Shift:	001000	
Subtract:	101011 110011 010100 001000	Negative - record 0, add back.
	etc.	Quotient: 0.10110

0.10110

The first 6 bits of the quotient are therefore 0.10110. Complementing, the final result is $\frac{1.10001}{0.10100} = 1.01001$.

The reader may verify that in decimal this would read $\frac{-.4375}{.6250} = -.700$,

and that the binary equivalent of .700 is 0.10110011.......

Now, there is one possible "shortcut" peculiar to binary. We have seen that when subtraction of the divisor gives a negative result, the divisor must be added back before shifting and subtracting again. In binary, upon getting a negative result, we can shift first and then add the divisor.* However, when we shift before restoring, we are working with a complement, and cannot discard bits shifted "off the left end." In order to make the end-around carry come out right, it is necessary to bring the shifted bits around to the right and fill them in there. In the LINC, this is called rotation to distinguish it from ordinary scaling.

* We are shifting the remainder left, which multiplies it by two. So, if R is the number from which we just subtracted, and D is the divisor, the negative result is (R-D). It is obvious that $2[(R-D) + D] - D = 2(R-D) + D$.

Here is the same example, using the "shortcut":

0.10100/0.01110

Subtract:	001110	
	101011	
	111001	Negative: Record 0 in quotient, rotate left one place.
rotate:	110011	
add:	010100	
	001000	Positive: record 1 in quotient, continue.
rotate:	010000	(Notice that rotating a positive number
subtract:	101011	left one place is indistinguishable
	111011	from scaling it left one place)
		Negative: so record 0, rotate, add.
rotate:	110111	
add:	010100	
	001100	Positive: record 1, continue.
rotate:	011000	
subtract:	101011	
	000100	Positive: record 1, continue.
rotate:	001000	
subtract:	101011	
	110011	Negative: record 0, continue.
rotate:	100111	
add:	010100	
	111011	
	etc.	Quotient: 0.10110

0.10110

POWERS OF TWO AND EIGHT

Positive Powers			Decimal Equivalents
2^0	8^0		1
2^1		2×8^0	2
2^2		4×8^0	4
2^3	8^1		8
2^4		2×8^1	16
2^5		4×8^1	32
2^6	8^2		64
2^7		2×8^2	128
2^8		4×8^2	256
2^9	8^3		512
2^{10}		2×8^3	1,024
2^{11}		4×8^3	2,048
2^{12}	8^4		4,096
2^{13}		2×8^4	8,192
2^{14}		4×8^4	16,384
2^{15}	8^5		32,768

POWERS OF TWO AND EIGHT

Negative Powers			Decimal Equivalents
2^{0}	8^{0}		1.0
2^{-1}		4×8^{-1}	.5
2^{-2}		2×8^{-1}	.25
2^{-3}	8^{-1}		.125
2^{-4}		4×8^{-2}	.0625
2^{-5}		2×8^{-2}	.03125
2^{-6}	8^{-2}		.015625
2^{-7}		4×8^{-3}	.0078125
2^{-8}		2×8^{-3}	.00390625
2^{-9}	8^{-3}		.001953125
2^{-10}		4×8^{-4}	.0009765625
2^{-11}		2×8^{-4}	.00048828125
2^{-12}	8^{-4}		.000244140625
2^{-13}		4×8^{-5}	.0001220703125
2^{-14}		2×8^{-5}	.00006103515625
2^{-15}	8^{-5}		.000030517578125

OCTAL ADDITION

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

OCTAL MULTIPLICATION

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61