1. NUMBER SYSTEMS USED IN COMPUTING: THE BINARY NUMBER SYSTEM

1.1 Introduction

Given that digital logic and memory devices are based on two electrical states (on and off), it is natural to use a number system, called the binary number system, which contains only two symbols, namely 0 and 1.

This Chapter begins by introducing conversion between decimal and binary numbers, then treats binary arithmetic. In common practice of using binary numbers, many conventions have been devised, so this chapter also introduces several conventions for manipulating and storing binary numbers, including the IEEE floating-point standard.

1.2 **Binary Numbers**

Positive Integers

In order to get some idea of the correspondence between binary numbers and the more familiar decimal numbers, one need only glance at the table below.

Decimal	Binary			Decimal	Binary		
0	0			9	1001	2^3+2^0	
1	1	2^0	2^1-1	10	1010	2^3+2^1	
2	10	2^1		11	1011	$2^3+2^1+2^0$	
3	11	2^1+2^0	2^2-1	12	1100	2^3+2^2	
4	100	2^2		13	1101	$2^3+2^2+2^0$	
5	101	2^2+2^0		14	1110	$2^3+2^2+2^1$	
6	110	2^2+2^1		15	1111	$2^3+2^2+2^1+2^0$	2^4-1
7	111	$2^2+2^1+2^0$	2^3-1	16	10000	2^4	
8	1000	2^3					

Some salient features of the correspondence include:

- a 1 followed by n 0's in binary represents decimal 2^n.
- a group of n 1's in binary represents decimal (2^n-1).

This is the basic structure of binary arithmetic.

Ex. 1: The decimal number 14 is represented in binary as:

$14_{\text{base } 10} = 8 + 4 + 2 + 0 = 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$
$= 1110_{\text{base } 2} = \%1110.$

The % indicates that we are dealing with a binary number.

It is useful (and easy) to generate a list of powers of 2:

2^0 = %1	=1	2^6 = %1000000	=64
2^1 = %10	=2	2^7 = %10000000	=128
2^2 = %100	=4	2^8 = %100000000	=256
2^3 = %1000	=8	2^9 = %1000000000	=512
2^4 = %10000	=16	2^{10} = %10000000000	=1024

2^5 = %100000 =32 2^{11} = %100000000000 =2048

Ex. 2: The decimal number 353 is represented in binary as:

353 = 256 + 64 + 32 + 1 = $1*2^8 + 0*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$ =%101100001

Ex. 3: The decimal number 251 is represented in binary as:
251 = 128 + 64 + 32 + 16 + 8 + 2 + 1
= $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$
= %11111011

An alternative view:
If you remember that 256 = 2^8 = %100000000,
then you can notice that 255 = 2^8-1 = %11111111,
and that 251 is 255-4 = 255-2^2 = %11111011
(simply remove the "1" corresponding to the 2^2 position.)

Ex. 4: One can use the same idea as in Ex 3 in many cases to convert from binary to decimal:
%11011101 = 255 - 2^5 - 2^1 = 255 – 32 – 2 = 221

Bits and bytes and words

Each binary digit is known as a bit. A grouping of 8 binary digits or bits is known as a byte. Given a fixed number of *n* bits, known as a word, which the arithmetic unit of a computer is designed to handle, then there are 2^n separate binary numbers that can be accommodated. For example, in 8 bits, one can accommodate the binary numbers corresponding to decimal 0 to 255 (256 different numbers).

Current computers have word lengths of 32 or 64 bits.

Fractions

Numbers smaller than 1 are represented using negative powers of 2. For reference, the first few negative powers of 2 are:

$2^{-1} = ½$	= 0.5	= %0.1
$2^{-2} = ¼$	= 0.25	= %0.01
$2^{-3} = 1/8$	= 0.125	= %0.001
$2^{-4} = 1/16$	= 0.0625	= %0.0001
$2^{-5} = 1/32$	= 0.03125	= %0.00001
$2^{-6} = 1/64$	= 0.015625	= %0.000001
$2^{-7} = 1/128$	= 0.0078125	= %0.0000001
$2^{-8} = 1/256$	= 0.00390625	= %0.00000001

Note that the number of places to the right of the decimal point is equal to the absolute value of the negative exponent (and is equal to the number of places to the right of the binary point.)

Ex. 5: The decimal number 3.375 is represented in binary as:

3.375 = 2 + 1 + 0.25 + 0.125
= $1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3}$
= %11.011

The fractions 1/3 or 3/7 cannot be represented as terminating decimal numbers, A little thought will lead one to realize that only those fractions whose denominator can be expressed as a power of 2 can be written as a terminating binary number. The binary number representations for ½, ¼, 3/8, 9/16, etc. all terminate. However, the binary number representations for 1/3, 1/5, 3/10 etc. need an infinite number of binary digits. The least-significant (right-most) bits of these representations must be truncated. As is the case of decimal numbers, we must decide how many digits beyond the binary point we wish to retain.

Division by twos method for decimal to binary conversion

A simple way to convert decimal integers to binary numbers is through repeated division by 2 and the saving of remaining 1's and 0's. For example:

Ex. 6: 11 = 2*5+**1**, 5 = 2*2+**1**, 2 = 2*1+**0**, 1 = 2*0+**1**. Stringing the remainders in reverse order, we get 11 = %1011.
This works since:
11 = 2*5+**1** and 5 = 2***2+1** so that 11 = 2*(2*2+**1)+1.**
Now 2 = 2*1+**0** so that 11 = 2*(2*[2*1+**0**]+**1)+1.**
Finally, 1 = 2*0+**1**, so that 11 = 2*(2*[2*{2*0+**1**}+**0**]+**1**)+**1**
= 2^3*1+2^2*0+2^1*1+2^0*1.

Ex. 7: Convert 353 to binary format using division by twos.

Number	Remainder		Number	Remainder
353			11	0
176	1		5	1
88	0		2	1
44	0		1	0
22	0		0	1

Taking remainders in reverse order, we get 353 = %101100001.

A similar method can be used to convert the fractional part of a non-integer – repeated multiplication by 2:

Ex. 8: Convert decimal 0.3 into binary. Since 0.3 is not a power of 2, the binary representation will not terminate. Therefore, we must decide how many binary digits will be used. Say we want to use 3 bits, we will do three multiplications by 2, to obtain a 3-bit approximation of 0.3:

multiply 0.3*2 = 0.6 = 0 + 0.6 → first bit = 0 → %0.0
multiply 0.6*2 = 1.2 = 1 + 0.2 → second bit = 1 → %0.01
multiply 0.2*2 = 0.4 = 0 + 0.4 → third bit = 0 → %0.010

In this case, we would be approximating 0.3 by 0.25.
Since this is not a very precise approximation, we might want to use more bits, say a total of 6 bits, for a 6-bit approximation. We could simply continue the procedure:

multiply 0.4*2 = 0.8 = 0 + 0.8 → fourth bit = 0 → %0.0100
multiply 0.8*2 = 1.6 = 1 + 0.6 → fifth bit = 1 → %0.01001
multiply 0.6*2 = 1.2 = 1 + 0.2 → sixth bit = 1 → %0.010011
Thus %0.010011 = 0.296875 is our 6-bit approximation of 0.3.

You might notice that the pattern will now repeat:
%0.010011001100110011 … for a better and better approximation of 0.3.

Suggestion ➔ for practice, try using more bits for the fractional part, such as 8 bits or 9 bits. The answer will be a better and better approximation to 0.3.

If one has a calculator at one's disposal, one can do the above procedure more rapidly:

One must first decide how many digits[1] we want beyond the binary point, 6 in this case:
We want to represent 0.3 in 6 places in binary: 0 . _ _ _ _ _ _

➔ So we convert this fraction-conversion problem to an integer-conversion problem by shifting[2] the contents of the register by the needed number of places (6), to the left. This is the same as shifting the binary point itself 6 places to the right, and corresponds to multiplying by 2^6.

0.3 | 0. _ _ _ _ _ _ | _ _ _ _ …

multiply by 2^6 = 19.2 | 0 _ _ _ _ _ _ . _ _ _ _ …
corresponds to shifting the binary point 6 places to the right

round: 19.2 ➔ 19 | 0 _ _ _ _ _ _ .

convert 19 into binary | 0 0 1 0 0 1 1 .

divide by 2^6 = 0.296875 | 0. 0 1 0 0 1 1
corresponds to shifting the binary point 6 places to the left

0.296875 ≈ 0.3 ➔ 6-bit approximation[3] of 0.3 = %0.010011

In general, suppose we have decided to use *r* digits to represent or approximate the fraction portion of the number. We multiply the fractional portion of the decimal number by 2^r, round the result to the nearest integer and convert the remaining integer.

---

[1] This decision is based on other factors, such as processor design, software conventions, beyond the scope of this text.
[2] See Appendix 2 for a more-detailed explanation of the correspondence of shifting to multiplication and division.
[3] Note that the 0 immediately to the right of the binary point in the fractional part, 010011, is very important, since we decided to use 6 bits.

Binary Number Registers:

In a computer, a number is represented physically in a **register**, which has a fixed length. For example, a register having 16 binary digits ("bits") limits the amount of information that can be represented.

- Note also that each bit must have a value of either 0 or 1 (there is no "blank" in a physical register).
- Note also that there is no provision for a binary point in a physical register. The position of the binary point must be inferred based on convention or other information.

1.3 **Hexadecimal numbers**

In order to avoid the writing of long strings of 1's and 0's, other number systems based on powers of 2 are used, the most common being the base 16 or hexadecimal system. Powers of two are used to enable easy conversion back and forth from binary.

In hexadecimal, we need six more symbols to take us beyond 0 to 9. These are A := 10, B := 11, C := 12, D := 13, E := 14, F := 15. In tabular form, the listing is:

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

The conversion between binary and hexadecimal is simple. To go from binary to hexadecimal, we start at the binary point, making sure that the number of digits to the left and right of the point are multiples of four. If not, simply add leading 0's in the first case and trailing 0's in the second. We then divide up the binary number into groups of four and replace each group by its hexadecimal equivalent. To go from hexadecimal to binary, we simply reverse the process.

Ex 1: To convert %1001011100.011001 to hexadecimal, we first pad with 0's as appropriate on both the left and the right, and rewrite it as:
0010|0101|1100.0110|0100. Then convert using the table to $25C.64. The $ indicates that the number is in hexadecimal format. Each hexadecimal symbol represents a power of 16
In decimal, this would be:
$= 2*16^2 + 5*16^1 + 12*16^0 + 6*16^{-1} + 4*16^{-2}$
$= 2*256 + 5*16 + 12 + 6/16 + 4/256$
$= 512 + 80 + 12 + 0.375 + 0.015625 \quad = 604.390625$

1.4 **Binary Arithmetic**

Addition: The addition of two binary numbers is carried out in much the same fashion as in the case of decimal numbers. To add 1003 and 501, we carry out the operation shown in the table below.

Addition													
11	10	9	8	7	6	5	4	3	2	1	0	Column number (i)	
0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**Carry digit - C**	
0	0	1	1	1	1	1	0	1	0	1	1	First number - A	1003
0	0	0	1	1	1	1	1	0	1	0	1	Second number - B	501
0	1	0	1	1	1	1	0	0	0	0	0	**Sum - S**	1504

It is important to analyze the process, as this will be crucial when it comes to the design of logic circuits (COEN 312) to carry out addition. To do this, we first label the columns proceeding from left to right from 0 to 11 as shown above. These column numbers will correspond to the index "i" in the terms below. Next, we note how the digit S_i (where i goes from 0 to 11) is obtained. S_i = 1 whenever the triple (C_i, A_i, B_i) contains a single 1 or three 1's. Otherwise S_i = 0. Finally we note that the carry digit C_{i+1} = 1 whenever at least two of the triple (C_i, A_i, B_i) are 1's. When we do Boolean Algebra (in the next chapter) we will see how to write these as logical expressions which can then be designed as a logic circuit known as a "full-adder".

Subtraction: Here again we can duplicate in binary, the technique learned for decimal numbers in elementary school.
For example:

```
 20   ➔ borrow ➔    1 10
-11                 -1  1
                 =  0  9 = 9

 10100 ➔ borrow ➔ 10 0  10 0  ➔ borrow ➔ 10 0 1 10
- 1011            - 1 0  1 1              - 1 0 1  1
                                         = 1 0  0  1
```

However, the implementation of such a borrowing scheme would require the design of a separate logic circuit to do subtraction. What we will introduce instead is a technique for representing negative numbers, which will allow us to use addition to perform subtraction.

Negative numbers without negative signs: There are several mechanisms available to represent negative numbers so as to avoid the use of the '-' sign. The simplest is to add a leading digit where 0 stands for a positive number and 1 for a negative number. While this method is used for the storage of numbers, it is not very useful for arithmetic operations, as it would be necessary to treat the leading digit differently from the others. The two main techniques are first, the "two's complement", and second, "biasing". Two's complement permits logic circuits designed for addition to be used for subtraction or the addition of negative numbers. Biasing is used for storing negative exponents as part of the IEEE 754 format (later in this chapter).

The key concept, which enables the use of two's complement, is the existence of a fixed register length. To illustrate, we will assume in base 10 arithmetic, a register that has a length of two. We now perform 44 + 56 = 100.

	4	4
+	5	6
1	0	0

Since our register only has a length of 2, we will have <u>*lost*</u> the leading digit, 1, through a phenomenon known as *carry-out*. If we read what remains, we have an answer of 0. This might suggest to us that, although the range of numbers is limited, we might use it to our advantage: let the number 56, (which is 100 minus 44) represent the number –44.

	4	4
-	4	4
	0	0

Extending this idea, and keeping in mind that we have a total of 10*10 = 100 numbers we can represent using the two place base ten register, we set up the following scheme:

- The numbers 0 to 49 are represented naturally. In so doing, we are using 50 of the available 100 numbers
- The negative numbers –49 through –1 are represented by the positive numbers 51 through 99, by the simple operation of subtracting from 100 (e.g 100 - 49 = 51, and 100 – 1 = 99). We have used a further 49 of the available numbers.

Note that we have not represented the number 50, since –50 would have the same representation and this would be an opportunity for error. Hence the total range of numbers extends from –49 to 49. We have used 99 of the possible 100 numbers.

To illustrate that the scheme works, we carry out the operation 34 – 46 = -12. We replace –46 by 100 – 46 = 54.

	3	4
+	5	4
	8	8

Thus 34 - 46 is rewritten as 34 + 54 = 88. In this scheme (**convention**), the number 88 is the representation for –12, since 88 = 100 – 12.

We now introduce the same scheme for **binary**. For the purposes of illustration, we will assume a register length of n = eight bits, which will provide us with $2^n = 2^8 = 256$ possible numbers.

- We use the natural binary representation for the numbers 0 through 127.
- (number) + (it's 2's complement) = $2^8 = 2^n$, therefore
  (2's complement of a number) = 2^n – (the number)
- Thus, we represent the numbers –127 through –1, by the range (256 –127 = 129) through (256 – 1 = 255).

We do not use the number 128 since its representation would be the same as that for the number –128. In all we have used 255 of the possible 256 numbers. Note that the leading digit of this representation of a negative number will always be a 1. Again to show that the scheme works, we will carry out in binary, the operation 34 – 46 = -12. Recall that the representations of –46 and –12 are 210 = 256 – 46 and 244 = 256 – 12.

7	6	5	4	3	2	1	0	
0	0	1	0	0	0	1	0	(34)
1	1	0	1	0	0	1	0	(-46 written as 210)
1	1	1	1	0	1	0	0	(Sum is 244 which represents -12)

While what we have done above for getting the representation of negative numbers and vice-versa will always work, we want to simplify it and come up with something that could be implemented using logic circuitry. What we use is "one's complement" followed by two's complement. One's complement can be carried out using logic gates known as **inverters** (which simply switch each bit, from 0 to 1, or from 1 to 0), after which addition circuitry is used to convert one's complement to two's complement. We then simplify yet further and present a method, which, allows humans to carry out the conversion by inspection. This second method could also be implemented by even simpler logic circuitry.

<u>One's complement</u>: One obtains 1's complement[4] of a binary number by subtracting the number from the binary number consisting of the same number of bits, with all 1's. This is the same as switching all of the bits (substituting every 0 by a 1 and vice-versa).

For example, the binary representation of 43, in an 8-bit register, is %00101011. Its 1's complement is: %11111111 - %00101011 = %11010100.

The problem with 1's complement is that the addition of a number with its complement gives all 1's (e.g. %11111111), instead of 0 (e.g. %00000000), which would be much more desirable.

[4] Similar to 9's complement in the decimal number system: one obtains 9's complement of a decimal number by subtracting each digit from 9.

Two's complement: Similar to the 10's complement in decimal arithmetic, one obtains 2's complement of a binary number by first taking 1's complement, then adding 1. For example, the 2's complement of %0010 1011 is %1101 0100 + %0000 0001, which is %1101 0101.
This is the representation for –43 using 2's complement in an 8-bit word (the 2's complement representation for –43).

In this case, the addition of a number and its 2's complement gives 0 as follows:

<table>
<tr><th>Binary</th><th></th></tr>
<tr><td align="right">% 0010 1011</td><td align="right">Representation for decimal +43</td></tr>
<tr><td align="right">% 1101 0101</td><td align="right">Representation for decimal –43</td></tr>
<tr><td align="right">% 1 | 0000 0000</td><td align="right">Sum = 0</td></tr>
<tr><td colspan="2">Note that the carry into the 9th bit is discarded, since we started with only 8 bits, (and since the addition proceeds from right to left), leaving our desired answer of 0.</td></tr>
</table>

In using the 2's complement technique, it is very important to understand the impact of the fixed number of bits, which needs to be decided at the beginning. For example, in 8 bits, one can represent a total of 256 different binary arrangements. If there were no provision (convention) for negative numbers, these 256 arrangements would logically represent the decimal numbers 0 to 255 (%00000000 to %11111111). However, if the 2's complement convention is specified, the 256 arrangements have the following meanings:

Binary (range)	**Decimal (range)**
%0000 0000	0
%0000 0001 to %0111 1111	+1 to +127
%1000 0001 to %1111 1111	-127 to –1
%1000 0000	-128 (special case)

Special case: The last row of this table is a special case commonly used in computing. In an *n*-bit register, the pattern of a 1 followed by *n-1* 0's represents the largest negative number, $-(2^{n-1})$. Notice that the 2's complement of this bit-pattern is the same bit-pattern (the 2's complement of %1000 0000 is again %1000 0000). Also, this bit-pattern satisfies the desired constraint of 2's complement, in that addition of the number and its two's complement gives 0 (in this case %0000 0000, as shown below).

Binary	
% 1000 0000	Representation for decimal -128
% 1000 0000	Two's complement of -128
% 1 | 0000 0000	Sum = 0
Note that the carry into the 9th bit is discarded, since we started with only 8 bits, (and since the addition proceeds from right to left), leaving our desired answer of 0.	

In general, in an *n*-bit register, one can represent numbers between $+(2^{n-1}-1)$ and $-(2^{n-1})$, using the above system based on 2's complement.

Ex. Assuming 8-bit words, use two's complement to compute 112 – 61 = 51, and 61-112 = -51

➔ First, we have that 112 = %0111 0000 and 61 = %0011 1101. The one's complement of 61 = %00111101 is %11000010 and its two's complement is %1100 0011. We now add the two numbers to get, after discarding the 9th left-most digit, %0011 0011 = 51.

➔ The two's complement representation for –112 is %10010000. The representation for 61 is %00111101. Adding the two numbers gives:

<table>
<tr><th>Binary</th><th></th></tr>
<tr><td align="right">% 0011 1101</td><td align="right">Representation for decimal +61</td></tr>
<tr><td align="right">% 1001 0000</td><td align="right">Representation for decimal –112</td></tr>
<tr><td align="right">% 1100 1101</td><td align="right">Sum = -51</td></tr>
<tr><td colspan="2">Note that, since the left-most (8th) bit of the result is a 1, there is no carry-out, and the result represents a negative number (in this 2's complement system). The result, % 11001101, is the 2's complement of %00110011=51, which means that it represents –51.</td></tr>
<tr><td colspan="2">Note also that this binary arrangement, %11001101, would correspond to 255-32-16-2 = 255-50 = 205, without 2's complement.</td></tr>
</table>

A Quick Way to Convert to 2's Complement:

In doing 2's Complement conversions from (+) to (-) or from (-) to (+):

- start from the right (least-significant) bit,
- keep the same binary digits as in the original number, up to and including the first "1" that you reach,
- for all subsequent binary digits, replace "0" with "1" and "1" with "0".

For example: 01101010 is considered as 011010|10: the rightmost 2 digits are kept, while the remaining 6 are switched, giving 10010110.

Some Notes:

Two's complement of a (-) binary number gives a (+) binary number, so the above technique works for conversions in both directions, (-) to (+) and (+) to (-). One way to see why this works:

> The operation (flip all bits then add 1) is the same as the operation (subtract 1 then flip all bits), and both are equivalent to the quick technique described above.

Note also that simply adding the 2's complement representations for two negative binary numbers gives the correct answer, *provided that there are enough bits* to handle the 2's complement representation of the resulting negative number.

For example, consider this subtraction using 6 bits:
(-7) + (-6) = -13: %111001 + %111010 = 1|110011, which equals %110011 after truncating the carry-bit, which is the 2's complement of 13. This would not have worked if the result had been limited to 4 bits, since in 4 bits we only have 16 different arrangements, which can only represent the range of +7 to –7 in the 2's complement system, (not the full range of +15 to –15, which would be needed to represent –13 in 2's complement). In this case of adding (-7) + (-6) in 4 bits, we have the phenomenon called *overflow*.

Overflow: When the carry into the last (left-most) carry-bit is *different* from the carry-out, then we have overflow, and the result is erroneous.

Multiplication

This proceeds in a fashion analogous to base 10. For example if we wish to multiply 23 by 46, we use the following procedure or algorithm.

Multiplication in base 10

		2	3	
	*	4	6	
	1	3	8	6*23
	9	2	0	40*23
1	0	5	8	Sum

Note that 40*23 = 920 can be represented by shifting 4*23 =92 one position left and inserting a 0 in the vacated position

In decimal, the procedure relies on the fact that shifting the digits one place to the left (with respect to the decimal point), corresponds to multiplying by 10. In binary, we will carry out a similar series of multiply, shift and add operations. In binary, shifting the bits one place to the left (with respect to the binary point), corresponds to multiplying by 2. We will illustrate this by multiplying 19 by 13.

Multiplication in binary

	7	6	5	4	3	2	1	0	Column number
A	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	First number = 19
B	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	Second number = 13
	0	0	0	1	0	0	1	1	Number A times B0
	0	0	0	0	0	0	0	0	Number A times B1 shifted left by 1
	0	0	0	1	0	0	1	1	Sum
	0	1	0	0	1	1	0	0	Number A times B2 shifted left by 2
	0	1	0	1	1	1	1	1	Sum
	1	0	0	1	1	0	0	0	Number A times B3 shifted left by 3
	1	1	1	1	0	1	1	1	Sum
	-	-	-	-	-	-	-	-	Process continues. Not shown as all other Bi = 0
	1	**1**	**1**	**1**	**0**	**1**	**1**	**1**	Final result = 247

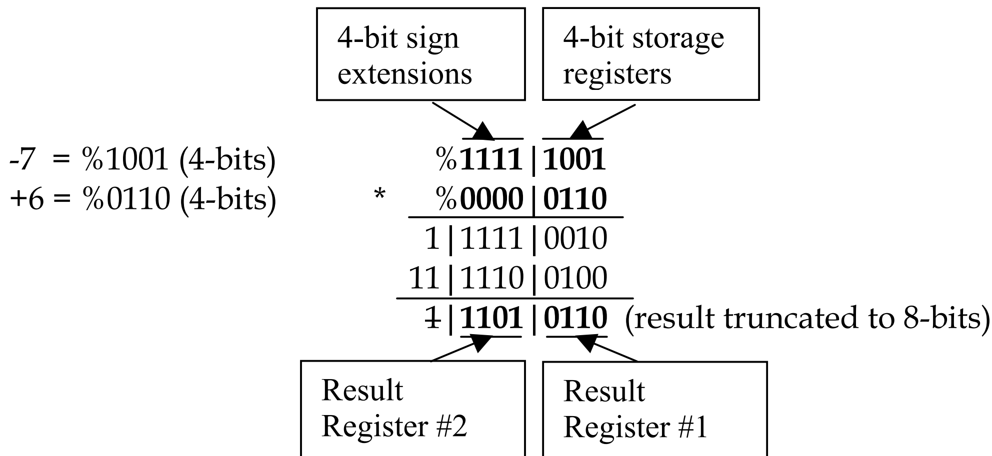
The multiplication operation requires only one new operation beyond the full adder, namely to recognize that 1*1 = 1 and that 0*1 = 1*0 = 0*0 = 0. We will see in the next two sections that this is nothing more than the logical or Boolean AND operation. As such, multiplication can be performed with two logic circuits that implement these two operations and software to control the sequence of operations. Faster multipliers are achieved by

having a series of interconnected full adders, multipliers and shifters thus reducing the amount of software. Note that our 8-bit word does not allow the multiplication of numbers whose result is greater than 255. In general, to house the product of two binary numbers having *i* bits, and *j* bits, respectively, one needs a result register of at least (*i*+*j*) bits.

- %111 x %111 = %110001 (3 bits x 3 bits → 6 bits)
- %11111 x %11111 = %1111000001 (5 bits x 5 bits → 10 bits)
- (8 bits x 4 bits → 12 bits)

Multiplication of Negative Integers[5]: The above examples treat only unsigned (positive) binary numbers. In computers, multiplication of integers where one or both of the numbers is/are negative, is handled by using 2's-complement. (In other words, multiplication also works for negative numbers represented in 2's-complement). However, some special handling is required for it to work properly.

We will show this by an example: In a 4-bit register using 2's-complement, one can represent integers between +7 and –8, inclusive. The result of the multiplication will fit into 8 bits (two 4-bit registers), where numbers between +127 and –128 can be represented. This range is enough to fit the largest positive and negative results of the multiplication: (-8)*(-8) = +64 and (+7)*(-8) = -56. However, the two four-bit registers used as input must each be "sign-extended" to the pre-determined size of the result (8 bits):



Note that %11010110 is the 2's complement representation for –42,

[5] This development only applies to integers. Multiplication of numbers represented in floating-point (scientific) notation does not use 2's complement. Instead (positive) magnitudes are multiplied and the sign of the result is determined by a simple logic procedure.

and %00101010 is the representation for +42.

Division

The procedure for division in binary is analogous to the common procedure of long-division in decimal. In base 10, we note that 253.5/13 = 19.5 can be written as 253.5 = $13*1*10^1 + 13*9*10^0 + 13*5*10^{-1}$. When we carry out long-division, we first obtain the number 1 by ascertaining that it is the largest multiple of 130 which when subtracted from 253.5, yields a non-negative number. Having done this we subtract 130 from 253.5 to obtain 123.5. We repeat the operation and obtain 9 as the largest multiple of 13 which when subtracted from 123.5, yields a non-negative number, 6.5 in this case. Finally, repeat the operation again, to obtain 5 as the largest multiple of 1.3 which when subtracted from 6.5 yields a non-negative number, 0 in this case.

In binary we follow an analogous procedure to obtain:

$253.5 = 13*1*2^4 + 13*0*2^3 + 13*0*2^2 + 13*1*2^1 + 13*1*2^0 + 13*1*2^{-1}$.

In this case, we only have to determine whether a subtraction of $13*1*2^k$ for a given value of k yields a non-negative number or not.

Divisor = 13											
					1	0	0	1	1	.1	Result of division = 19.5
1 1 0 1	)	1	1	1	1	1	1	0	1	.1	Number to be divided = 253.5
		1	1	0	1						Divisor * 1
				1	0						Remainder after subtraction
				1	0	1	1	0			After dropping three more bits
					1	1	0	1			Divisor * 1
					1	0	0	1	1		Remainder after subtraction
					1	0	0	1	1		After dropping another bit
						1	1	0	1		Divisor * 1
							1	1	0		Remainder after subtraction
							1	1	0	1	After dropping another bit
							1	1	0	1	Divisor * 1
							0	0	0	0	Remainder after subtraction

Division can be viewed as the inverse of multiplication. Shifts to the left are replaced by shifts to the right and additions are replaced by subtractions (carried out as additions using two's

complement. The only new operation is a comparison of the result of a subtraction with 0 to see if it is negative or not.

Note that, in binary, shifting the bits one place to the right (with respect to the binary point), corresponds to dividing by 2.

If the result was not a terminating binary number, we would simply keep going, like we do in decimal long-division, dropping 0's until we had reached the desired precision in the result.

Another view of a similar long-division procedure: 247/13 =19.

Division in binary

Divisor = 13	7	6	5	4	3	2	1	0		Column number
				1	**0**	**0**	**1**	**1**		Result of division
1 1 0 1	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**		Number to be divided = 247
	1	1	0	1	0	0	0	0	**1**	*13*16 = 208
	0	0	1	0	0	1	1	1		Result of subtraction = 39 (non-negative)
	0	0	0	0	0	0	0	0	**0**	*13*8 = 0 since 1*13*8 = 104 when subtracted from 39 would give a negative
	0	0	1	0	0	1	1	1		Result of subtraction = 39 (non-negative)
	0	0	0	0	0	0	0	0	**0**	*13*4 = 0 since 1*13*4 = 52 when subtracted from 39 would give a negative
	0	0	1	0	0	1	1	1		Result of subtraction = 39 (non-negative)
	0	0	0	1	1	0	1	0	**1**	*13*2 =26
	0	0	0	0	1	1	0	1		Result of subtraction = 13 (non-negative)
	0	0	0	0	1	1	0	1	**1**	*13*1 =13
	0	0	0	0	0	0	0	0		Result of subtraction = 0 (non-negative)

1.5 Scientific Notation and the IEEE 754 Floating-Point Standard

Conversion of integers and fractions, and binary arithmetic are fundamental skills. Beyond that, there are many available systems for storing and manipulating numbers represented in binary. In these systems, the meaning of the specific bits changes, according to specific conventions. In this section we explore one general type of such convention, binary scientific notation, with focus on one specific system, the IEEE 754 Floating-Point Standard.

<u>*Scientific (Floating Point) Notation*</u>

Scientific notation follows the conventions of decimal notation and is used in computing to carry out "floating point" arithmetic. In decimal notation, a number such as 226.25 is expressed in scientific notation as $2.2625*10^2$. The portion 2.2625 is known as the <u>significand</u> and the power of 10 as the <u>exponent</u>. To express this number in binary scientific notation we first convert it to normal binary.

226.25 = 128 + 64 + 32 + 2 + ¼ =

$= 1*2^7+1*2^6+1*2^5+0*2^4+0*2^3+0*2^2+1*2^1+0*2^0+0*2^{-1}+1*2^{-2}$

= %11100010.01

To put this into scientific notation, we move the "binary (no longer decimal) point" seven spaces to the left to obtain 226.25 = $\%1.110001001*2^7$, or $\%1.110001001*2^{\%111}$. The significand is 1.110001001 and the exponent is 7 = %111. The significand, the exponent and an additional bit to designate the sign together constitute the floating-point representation of a number.

Ex: In 16 bits, one could reserve: 1 bit - sign bit
4 bits - exponent
11 bits - significand

The 16 bits could then be of the form: s | xxxxxxxxxxx | eeee

If one were trying to represent 173.4 in binary, this scheme would give:

$= 1*2^7 +0*2^6 +1*2^5 +0*2^4 +1*2^3 +1*2^2 +0*2^1 +1*2^0$
$+0*2^{-1} +1*2^{-2} +1*2^{-3} +0*2^{-4} +0*2^{-5} +1*2^{-6} + \ldots$

= %10101101.011001…

This is a non-terminating binary number.

Transforming to the above-described scientific notation, for representation in the 16-bit register, formatted as we show above:

s = 0 (positive number)

xxxxxxxxxxx = 10101101011 (eleven binary digits, with the point assumed to be after the left-most “1”)

eeee = 0111 (representing binary 7, since the point was shifted by 7)

The 16-bit register would then contain: 0101011010110111

which means: 0|10101101011|0111
sign| significand| exponent

Handling of Negative Exponents:

While the above example has a positive exponent, note that one might need to represent a negative exponent. There are several ways to represent negative numbers, introduced later in this section. These schemes usually involve the use of the left-most bit as an indicator of the sign (for example, 2’s complement or a biasing scheme, described later in this section). If we assumed the use of 2’s complement, we would be able to represent exponents between +7 and –8 in the 4 bits that we have reserved in our example above.

The IEEE 754 Floating-Point Standard

The Institute of Electrical and Electronic Engineers (IEEE) has introduced a standard method for storing (only) floating point numbers, which has been used for virtually all computers manufactured since 1980.

Single Precision The standard word length used (for "single precision") is 32 bits. One bit is used for the sign, s, (0 for positive numbers and 1 for negative numbers), 8 bits for the exponent, E, and 23 bits for the significand (or mantissa).

The significand or mantissa: Since in scientific binary notation the significand always begins with a 1, this number is made implicit and not stored at all. We thus represent the significand as 1.M, where M is the mantissa, and where the 1 is not stored. This allows a precision in the representation of the number to be 1 part in $2^{\wedge 23}$ = 8,388,608 or roughly an error of ±1.2 in the seventh decimal digit.

The exponent: We have 8 digits available or 256 places. The procedure used for storing negative and positive exponents is called **biasing**. This is achieved by adding 127 (the maximum number which can be represented in 8-1=7 bits) to the exponent of 2. (Note: this is not the same as 1's complement.) The biased exponents 255 = %11111111 and 0 = %00000000 are reserved for special purposes as given below. As such:

The exponent –126 maps into the exponent 1, represented as 00000001,
----
The exponent 0 maps into the exponent 127, represented as 01111111,
The exponent 1 maps into the exponent 128, represented as 10000000,
----
The exponent 127 maps into the exponent 254, represented as 11111110.

The following table outlines the impact of the biasing scheme:

System for using an 8-bit exponent with Bias = 127				
Desired Exponent	+ 127 =	Converted to binary	Comment	
+∞	n/a	11111111	Reserved special case (Case 2 of IEEE754)	
+127	+254	11111110	Highest natural exponent available	
+126	+253	11111101		
…	…	…		
+60	+187	10111011		
…	…	…		
+2	+129	10000001		
+1	+128	10000000		
0	+127	01111111	The middle of the range of exponents	
-1	+126	01111110		
-2	+125	01111101		
…	…	…		
-60	+67	01000011		
…	…	…		
-124	+3	00000011		
-125	+2	00000010		
-126	+1	00000001	Lowest natural exponent available	
lower	n/a	00000000	Reserved special case (Case 4 of IEEE754)	
The “natural” range of exponents is called “Case 3” in the full IEEE754 floating-point system. (see later in this chapter)				

Ex. Find the IEEE 754 single precision floating-point representation of -0.21875.

➔ Following a procedure set out earlier in this chapter, we find that -0.21875 = - 0.125 – 0.0625 – 0.03125 = -%0.00111 = -%1.11*2^{-3}.
➔ We now represent this in the IEEE 754 format. The sign bit is %1. The significand is %1100...0. (Remember that we have discarded the leading 1)
The exponent is –3 + 127 = 124 = %01111100. The representation is thus:
1|01111100|11000000000000000000000.

The above-described range of exponents allows for representation of numbers as small as ~$2^{-126} \approx 1.18*10^{-38}$ or as large as $(2 - 2^{-23})*2^{127} \approx 3.4*10^{38}$.
Note that %1.111 1111 1111 1111 1111 1111 = $(2 - 2^{-23})$.

The designers of this IEEE standard noticed that one doesn't often need numbers in vicinity of $2^{-126} \approx 10^{-38}$ with 23-bit precision. More common is the need to represent smaller numbers, albeit with less precision. So another special case was devised within the convention, to extend the range by many negative powers of 2. This is accomplished by removing the assumed "1" in the significand, assuming a "0". This allows the representation of much smaller numbers, down to 2^{-149}.

We now list the <u>complete IEEE 754 code</u> where case 3 below is the normal situation described above (where the “1” is assumed), and case 4 assumes a leading “0”. In what follows, the numbers N and E are in normal base-10 notations. S is either 0 (positive) or 1 (negative). The symbol %M stands for the mantissa (the 23-bit fractional part of the significand). The base-10 number M is the corresponding base-10 fraction.

s | <u>eeeeeeee</u> | <u>mmmmmmmmmmmmmmmmmmmmmmm</u>
%E %M
E is the decimal equivalent of %E
M is the decimal equivalent of %M

To inpterpret a 32-bit binary arrangement as a decimal number using the full IEEE 754 convention, examine the bits corresponding to exponent and mantissa according to the following 5 cases:

1. If E = 255 and %M $\neq$ 0, then this means that N is not a number, abbreviated as NaN.
2. If E = 255 and %M = 0, then N is $\pm\infty$, depending on the value of the sign bit.
3. If 0 < E < 255, then in binary notation, the mantissa is 1.%M and in base 10 notation, is 1.M. This is the situation described above and $N = (-1)^S * (2^{E-127}) *(1.M)$.
4. If E = 0 and %M $\neq$ 0, then we assume that the leading 1 of the significand has <u>not</u> been stored. As such, in binary notation, the mantissa is 0.%M, and in base 10, is 0.M = M. $N = (-1)^S * (2^{-126}) *M$. Although we lose precision as the number becomes smaller and smaller, we can store numbers as small as $\pm 2^{-149}$.
5. If E = 0 and %M = 0, then N = 0, regardless of the value of the sign bit.

Examples of the use of these cases are presented below.

Examples and special cases of the IEEE 754 Standard (32-bit) format:

Case #	Decimal and/or Binary Format	IEEE 754 Format 1| 8 | 23 (bits)
5	0 =%0.0	0|00000000|00000 …00000 1|00000000|00000 …00000
4	Smallest non-zero positive number in this format = 2^{-149} = 0.00000…00001*2^{-126}	0|00000000|00000 …00001
4	2^{-130} =%0.0001*2^{-126}	0|00000000|00010 …00000
4	2^{-128} =%0.01*2^{-126}	0|00000000|01000 …00000
4	2^{-127} =%0.1*2^{-126}	0|00000000|10000 …00000
4	$2^{-127} + 2^{-129}$ =%0.101*2^{-126}	0|00000000|10100 …00000
4	Largest number in Case 4: $2^{-126} - 2^{-149}$ =%0.111…111*2^{-126}	0|00000000|11111 …11111
3	Smallest number in Case 3: 2^{-126} =%1.0*2^{-126}	0|00000001|00000 …00000
3	0.5 =2^{-1} =%1.0*2^{-1}	0|01111110|00000 …00000
3	0.75 =$2^{-1}+2^{-2}$ =%1.1*2^{-1}	0|01111110|10000 …00000
3	1 =2^{0} =%1.0*2^{0}	0|01111111|00000 …00000
3	1.5 =$2^{0}+2^{-1}$ =%1.1*2^{0}	0|01111111|10000 …00000
3	2 =2^{+1} =%1.0*2^{+1}	0|10000000|00000 …00000
3	4096 =2^{+12} =%1.0*2^{+12}	0|10001011|00000 …00000
3	5120 =$2^{+12} + 2^{+10}$ =%1.01*2^{+12}	0|10001011|01000 …00000
3	Largest possible number in this format = %1.11111…11111*2^{+127}	0|11111110|11111 …11111
2	$+\infty$	0|11111111|00000 …00000
1	Not a number (NaN) - Example	0|11111111|00010 …00100

Without case #4, the smallest possible non-zero number would be,

	$2^{-127}+2^{-150}$ =%1.000…001*2^{-127}	0|00000000|00000 …00001

assuming 0|00000000|00000 …00000 would be reserved for 0.

Ex: Represent the following decimal numbers in the IEEE 754 format:
$8.5*2^{-128}$, $4.375* 2^{-137}$.

In representing very small numbers such as these in the IEEE754 system, one must pay close attention to whether the converted result will be in Case 3 (greater than or equal to $1.0*2^{-126}$) vs. Case 4 (less than $1.0*2^{-126}$).

➔ so the first step is to convert the decimal numbers into binary such that there is a simple leading "1" to the left of the binary point in the significand.

$8.5*2^{-128} = \%1000.1*2^{-128} = \%1.0001*2^{-125}$
Since the resulting number is greater than $1.0*2^{-126}$, we are in Case 3, and the conversion proceeds as shown before:
<u>0 | 00000010 | 00010000…000</u>

On the other hand,
$4.375* 2^{-137} = \%100.011* 2^{-137} = \%1.00011* 2^{-135}$ which is $< 1.0* 2^{-126}$.
This will be Case 4, where we need to further convert the number such that the exponent becomes –126:

$\%1.00011* 2^{-135} = \%0.00000000100011* 2^{-126} =$
<u>0 | 00000000 | 00000000100011000…000</u>

Double Precision

Here we use two 32-bit words or a single 64-bit word, distributed as follows:

Sign bit: 1-bit,

Exponent: 11 bits (bias is 1023, represented by 10 bits),

Significand: 52 bits.

This significantly increases the range of numbers which can be considered to the order of $10^{\pm 308}$ and the accuracy of the significand which can now be trusted up to the 15th decimal place.

Advantage and disadvantage of the IEEE 754 format

Advantage: The exponent-biasing scheme allows easy sorting of numbers by size.

Disadvantage: Numbers must be converted before arithmetic operations can be carried out.

Why did IEEE choose this format? Probably because most calls on the arithmetic unit involve sorting as opposed to arithmetic operations.

Arithmetic with the IEEE 754 standard:

Addition of any two numbers requires that the decimal or binary points of the two numbers be aligned. For example, in order to add $2.78*10^0 + 4.219*10^2 = 2.78 + 421.9$, one must shift the numbers as follows:

			2	.	7	8
+	4	2	1	.	9	0
=	4	2	4	.	6	8

Similarly, addition of two numbers stored in the floating-point involves alignment of the exponents. This means that one of the binary arrangements must be shifted so that the exponents of the two numbers are the same. For example:

Two numbers are represented below in the binary 32-bit IEEE 754 floating-point format. Add the two numbers (using any method), and convert your binary answer to the same 32-bit IEEE 754 floating-point format. Show your reasoning.

0 01111111 01000000000000000000000
0 10000000 01000000000000000000000

➔ The first number is $\%1.01* 2^0$, while the second number is $\%1.01*2^1$.
One could use either exponent as the reference for this addition.
In this case, the reference 0 is most convenient: $\%1.01* 2^0 + \%10.1* 2^0$
$= \%11.11* 2^0 = \%1.111* 2^1$
➔ 0 10000000 11100000000000000000000

Multiplication of two numbers in scientific notation involves multiplying the two significands, adding the two exponents, and, if necessary, accounting for the resulting significand being ≥ %10.00. For example, multiplying the following numbers:

0 10000000 11100000000000000000000
0 10000001 11000000000000000000000

➔ The first number is $\%1.111* 2^1$, while the second number is $\%1.11*2^2$.
Multiplying the significands: 1.111 * 1.11 = 11.01001.
Adding the exponents: 1 + 2 = 3 = %11. Since 11.01001 ≥ 10.0, we need to shift the significand to the right, (1.101001), and add 1 to the exponent: 3+1 = 4
➔ in IEEE format: = 10000011

➔ 0 10000011 10100100000000000000000

Appendix 1: Summary of Conventions Introduced for Negative Binary Numbers

Below are 5 different binary representations for (–29).
Note that the binary representation for +29 in an 8-bit register is %00011101.

Convention	Example	Comments
Minus sign	-%11101	Use only on paper.
Sign bit in 8-bit register	1|0011101	This would require separate logic circuitry for subtraction. Magnitude must fit in n-1=8-1=7 bits.
1's complement in 8-bit register [Number + its negative] = 11111111	11100010	Magnitude must fit in 8-1=7 bits. Left-most bit is still like a sign bit. Not convenient for subtraction
2's complement in 8-bit register [Number + its negative] = 00000000 = 1's complement +1	11100011	Magnitude must fit in 8-1=7 bits. Left-most bit is still like a sign bit. (convenient for subtraction) Not convenient for sorting
Biasing in 8-bit register Add 2^7-1=127 to all numbers. 29+127=156, -29+127=98	-29 = %01100010 29 = %10011100	Magnitude must fit in 8-1=7 bits. Left-most bit is like reversed sign bit. [Number + its (-)] ≠ 0 Used to code exponents in IEEE 754 format.

Appendix 2: Shifting Binary Arrangements to Left/Right: Correspondence to Multiplication/Division by 2

In understanding multiplication and division, it is important to see that shifting a binary arrangement to the left/right with respect to the binary point corresponds to multiplying/dividing by 2.

From our basic knowledge of binary numbers, any binary arrangement represents a number, N, which is the sum of powers of 2. In general:

$$N = \sum_{i=-\infty}^{+\infty} a_i 2^i$$

where each a_i is a "1" or "0", and the sequential arrangement of all the a_i constitute the binary number. The binary point would be located between the a_0 and a_{-1} .

Consider the action of multiplying by 2: This corresponds to $2N = \sum_{i=-\infty}^{+\infty} a_i 2^{i+1}$. Since the range of the summation index is infinite, this is equivalent to: $2N = \sum_{i=-\infty}^{+\infty} a_{i-1} 2^i$, which corresponds to shifting all bits to the left by 1 bit with respect to the binary point.

The same type of analysis can be applied to division by two, corresponding to shifts to the right with respect to the binary point.

Number Systems: Summary of Important Skills

- Conversion from decimal to binary and binary to decimal. There are several acceptable ways to do these conversions.
- Conversion of fractional and decimal numbers.
- Physical binary registers
- Hexadecimal numbers and binary/hexadecimal conversion: start from the binary point.
- Binary addition and how a full adder works
- Systems for representation of negative binary numbers
- Subtraction of binary numbers
- The 2's complement system
- Binary multiplication and division
- Decimal and binary floating-point representation
- IEEE 754 floating-point standard
- Arithmetic with numbers represented in IEEE754 standard

Summary of Scientific Notation Formats in this Section:

Normal:	$1.xxxx * 2^y$	s | ee…e | 1xxx…xxx
IEEE754 Case 3:	$1.xxxx * 2^y$	s | eeeeeeee | xxx…xxx
IEEE754 Case 4:	$0.xxxx * 2^{-126}$	s | 00000000 | xxx…xxx

Problems – Number Systems

(1) Express each of the following binary numbers in decimal:
- (a) %10, %0010, %10000000
- (b) %0.10,%0.000001,%0.00000100
- (c) %1010.101, %00011011101.1011
- (d) %0.00110011001100110011….
- (e) %0.11001100110011001100….

(2) What is the base-10 equivalent of %1101010010.101?

(3) Express each of the following decimal numbers in binary:
- (a) 1, 2, 4, 8, 16
- (b) 3, 5, 6, 7, 9
- (c) 10, 15, 20
- (d) 512, 1024, 1048576
- (e) 100, 1000, 971, 555, 222, 444

(4) Express each of the following decimal numbers in binary using 10 bits:
- (a) 0.5, 0.25, 0.125
- (b) 6.875, 10.03125
- (c) 6.6, 2.8, 0.1

(5) Express each of the following fractions in binary:
(Note: (b) and (c) will be repeating patterns.)
- (a) 1/16, 1/64
- (b) 1/100
- (c) 1/3, 3/7

(6) Express the following decimal numbers in <u>decimal</u> scientific notation:
Use 5 digits for the significand.
- (a) 185.3
- (b) 1/3
- (c) $10*10^{6}$
- (d) $144*10^{-4}$

(7) For each of the numbers in (6) above, convert to binary,
and express it in binary scientific notation, using 14 bits for the significand.

(8) (a) For each of the numbers in (1) above, express it in binary scientific notation.
(b) For each of the numbers in (4) above, convert to binary, and express it in binary scientific notation, using 14 bits for the significand.
(use 2's complement to represent the exponent)

(9) For each of the binary numbers in (1) above, express it in hexadecimal.

(10) Express the following hexadecimal numbers in binary:
$ABC, $FFFF, $F0E1, $1A, $A1.0F

(11) (a) What are the decimal equivalents of $FF.F, $10.1, $9A.B ?
(suggestion: in any problem involving hexadecimal representations, convert to binary first)
(b) What are the hexadecimal representations of the decimal numbers:
512, 827, 238, 5000, 5000.375?

(12) Express in decimal the same hexadecimal numbers as in (10) above.

(13) Express the following decimal numbers in hexadecimal:
1000, 1024, 1.05, 5.01, 16, 0.375
(Use up to 3 hex characters for the fractional parts.)

(14) Represent the decimal number 0.238 as a binary number in a 16-bit register. Consider the binary point to be at the center of the register.

(15) Express 327 in binary, assuming that you have available a word length of 12 bits. First do the conversion by expressing 327 as a sum of powers of 2. Redo the conversion using the division by twos method. Answer: 327 = %000101000111.

(16) a) Express decimal 42.8 as a binary number in scientific format where you have available eleven digits for the significand and three digits for the exponent.
b) How about if there are five digits for the exponent?

(17) Approximate 327.21 in binary using a 16-bit word. There will be 7 bits available to approximate 0.21. (Why?). What is the value of the 16-bit approximation? Answers: 101000111.0011011, 327.2109375

(18) a) Express 327.21 in floating point notation, assuming that in addition to the 16 bits used to express the significand, we have four additional bits to code the exponent. Answer: 1000| 1010001110011011
b) What are the smallest and largest positive numbers that can be expressed with a 4-bit exponent and a 16-bit significand?

(19) Convert the answer to 18a) floating point hexadecimal notation.
Answer: 8| A39B

(20) Using binary notation, and 12-bit words, add 721 and 638.

(21) Using 12-bit words
a) Express –638 and –721 in two's complement notation

b) Carry out as a binary addition, 721 – 638
c) Carry out as a binary addition, -721 + 638

(22) Using 12-bit words, multiply 21 by 53. What maximum product can be handled by the 12-bit word?

(23) Repeat question 7, where now the 12-bit word is subdivided to handle floating point multiplication of positive numbers. The first four bits contain the exponent and the second four the significand.

(24) In binary divide 1113 by 53.

(25) Express –123.456 in IEEE 754 single precision format.

(26) Express decimal 971 and 555 as binary numbers and then add the two binary numbers. Check your work by converting 1526 (=971+555) to binary.

(27) Convert the following decimal numbers to binary, and multiply (in binary): 1000 by 100. Verify that your binary answer is equivalent to 100000.

(28) Find the binary representation of decimal 238/512. Do this in three ways:
(a) Divide 238 by 512 to obtain a decimal number, then convert to binary.
(b) Convert 238 and 512 to binary, and divide in binary.
(c) Find the binary representation of 238, then multiply by 2^{-9}. In your answer, show why multiplying by 2^{-9} is the same as shifting the binary point by nine places to the left.
(Obviously, you should obtain the same answer in all three ways!)

(29) (a) Using two's complement, convert –333 to binary, using a 10-bit word.
(b) Subtract 333 from 444 using two's complement and binary 16-bit words.

(30) (a) Convert to binary and divide (in binary) 38 by 23, using 10 bits for your answer.
(b) Then convert your answer back to decimal.
(c) Divide 38 by 23 using your electronic calculator, and compare to your answer in (b).

(31) Translate the following binary numbers into the IEEE 754 Standard:
%11100101000110, %-11101.101, %-0.0001110101, %0.11111

(32) Represent the following decimal numbers in binary using the IEEE 754 Standard:
0, 1, $5*10^6$, $-3*2^{-37}$.

(33) For all possible cases in a 4-bit binary arrangement, demonstrate that the operation (flip all bits then add 1) is the same as the operation (subtract 1 then flip all bits). Note: for the special case of 0000, when subtracting 1 from 0000, you must borrow from a "virtual" 1, situated to the left of the 4-bit register.

(34) Do the subtraction +12.875 –13.000 using 2’s complement. Keep the binary point 3 bits from the right of the 8-bit register.

(35) The following six strings of binary digits are 32-bit representations of six different decimal numbers in IEEE 754 format.

(a) Select any two of the six, and convert each of the two to decimal format. Make sure you indicate which ones you are converting.

(b) Sort the six <u>decimal</u> numbers in increasing order. For example, one possible (wrong) order is (n1,n2,n3,n4,n5,n6). (Be careful: You are <u>not</u> sorting raw 32-bit binary numbers. You are sorting the <u>decimal</u> numbers which the binary arrangements represent.)

n1: 0 1 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
n2: 0 0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
n3: 0 1 0 0 0 0 0 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
n4: 1 0 1 1 1 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 0
n5: 1 1 0 0 0 0 0 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
n6: 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

(36) Consider this 32-bit binary string:

01000001101100000000000000000000

(a) Convert the represented binary number to hexadecimal, assuming that the binary point is at the right side of the sequence.

(b) Assume that the 32-bit string is a representation of a binary number in the IEEE754 format. Find the decimal equivalent of the represented binary number.

(37) <u>Using binary 2’s complement and 12-bit registers</u>, subtract 221 from 112. Leave your answer in binary 2’s complement, and verify that your answer is equivalent to the decimal number (-109).

(38) Express the number 23.4 in binary scientific notation where you will use 9 bits to display the number, namely, it should be of the form "1.xxxxxxxx" and 3 bits to display the exponent, namely it should be of the form "yyy". Note that in order to fit the number into 9 bits, it will be necessary to truncate trailing bits.

(39) Divide 178 by 16 (using any method), and leave your answer in binary. Your answer must be exact, without any approximation or truncation. Show your reasoning. (Suggestion: First convert 178 and 16 to binary.)

(40) Two numbers are <u>represented</u> below in the binary 32-bit IEEE 754 floating-point format. <u>Add the two numbers</u> (using any method), and convert your binary answer to the same 32-bit IEEE 754 floating-point format. Show your reasoning.

00111111101000000000000000000000
01000000010000000000000000000000

Do the same for the following pair of numbers:

01111011101010100000000000000000
01111011001010100000000000000000

(41) Consider the two decimal numbers 77.5 and 38.75.

(a) Convert both numbers to binary (no restriction on number of bits).
(b) Convert both numbers to hexadecimal.
(c) In binary, multiply 38.75 by 2. Show your work.
(d) Perform the subtraction 38 – 77 using 2's complement and 16-bit words, leaving your answer in binary.
(e) Show that your answer (d) is the 2's complement representation for –39.
(f) Convert -38.75 into the IEEE floating-point standard format.

(42) The following two strings of binary digits are 32-bit <u>representations</u> of two different decimal numbers in IEEE 754 format. <u>Multiply the two decimal numbers</u> (using any method), write down the decimal number, <u>and</u> convert your answer into the same 32-bit IEEE floating-point format. Show your reasoning.

n1: 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
n2: 0 1 0 0 0 0 0 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

(43) How many different <u>negative</u> decimal numbers can be represented in a <u>5-bit</u> register using the 2's complement system? (Be careful: you also need to be able to represent positive numbers in the same scheme.)

(44) Convert the following decimal numbers into binary, and then into hexadecimal.
For the fractional parts of the numbers, use up to 12 bits (3 hex symbols).
1000, 1024, 16, 0.875, 1023.01625, 1.05, 5.01

(45) Find the binary representation of decimal 163/512. Do this in three ways:

(a) Divide 163 by 512 to obtain a decimal number, then convert to binary.
(b) Convert 163 and 512 to binary, and divide in binary.
(c)Find the binary representation of 163, then multiply by 2^{-9}. In your answer, show why multiplying by 2^{-9} is the same as shifting the binary point by nine places to the left.

(Obviously, you should obtain the same answer in all three ways!)

(46) What are the smallest and largest positive, non-zero decimal numbers that can be represented with an unsigned 16-bit significand, and a 4-bit exponent which uses the 2's complement system?

(47) Consider the sequence of binary numbers, each consisting of alternating 1's and 0's, beginning and ending with 1's: 1, 101, 10101, 1010101 …
Specify the sequence of arithmetic operations needed to create the next one from the preceding one. (e.g. how to create 10101 from 101 by simple arithmetic operations?)

(48) Represent the following decimal numbers in the IEEE 754 format:
$4.375 * 2^{-137}$, $8.5*2^{-128}$, $36.25*2^{-129}$, $36.25*2^{-137}$.