Computer Arithmetic

In this article we look at the way in which numbers are represented in binary form and manipulated in a computer.

Numbers have a long history. In Europe up to about 1400 numbers were represented by what we call Roman Numerals; for example 2014 is MMXIV. Roman numerals used the symbols I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, and M = 1,000 to represent a number. The larger numbers were placed on the left and values not represented were created by repetition; for example 3,003 would be MMMIII. However, in order to avoid a symbol being represented four times, the next higher value was used with a subtractor on the left; for example 4 is not IIII but IV (5 – 1). Note that there were variations in the way in which numbers were represented and that significant changes occurred after the fall of Rome; for example, 4 can be represented by IIII or IV.

The problem with Roman numerals is that is they are not suited to the operations of arithmetic (addition, subtraction, multiplication and division). The numbers we use today derive from the Arabic-Hindu system introduced into Northern Europe by Arab traders from Andalucía about 1100. It took until about 1500 to displace Roman numerals in Northern Europe and there was a period that if you wanted an accountant you had to choose either traditional Roman numerals or the more modern Arabic-Hindu system.

The Arabic-Hindu system uses ten symbols, 0,1,2,3,4,5,6,7,8,9 to indicate the integers in the base 10 system. This is a *positional system*, so that if you move a digit one place left, it is multiplied by the base; for example,

5 = 5
54 = 5 x 10 + 4
521 = 5 x 100 + 2 x 10 + 1

Suppose the digits are d_0, d_1, d_2, d_3…. d. The value of the number $d_4d_3d_2d_1d_0$ has the value

$d_4 \times 10^4 + d_3 \times 10^3 + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$. Remember that $10^0 = 1$.

Several other historic numbering systems also used the base 10 but were not positional; that is different symbols were used for powers of 10 and the order of the digits did not matter; for example, Egyptian hieroglyphics. The figure below represents 3,244 although you could totally scramble the order of the symbols without changing the value.

The positional notation system uses base 10 because we have 10 fingers. However, computers use the base 2 (the binary system) simply because digital components are in one of two states 0 or 1 (i.e., off or on). If we could make devices with ten states, we would use base 10. But we can't (yet) easily and cheaply manufacture such components. We are stuck with base two because of its simplicity and the ease with which we can manufacture binary logic elements. This is not a problem because we can convert between binary and decimal values quickly and efficiently. For most of the time, we can entirely forget that computers sue binary numbers. However, there are occasions where a programmer need to appreciate the implications of using base 2.

The fundamental rules of decimal and binary arithmetic are the same. Exactly the same. The only reason that we (people) have difficulty in manipulating binary numbers is that we are so used to working with decimal numbers that we don't consciously think about what we are doing – we do it instinctively; for example, if you add 7 and 8 you get 7 + 8 = 15. What we have actually done is to add 7 and 8, get a value greater than the maximum digit (i.e., 9), subtract 10, record the remainder as the result (i.e., 5), and then place a carry in the next position left (the 1).

In binary arithmetic we have only two values 0 and 1. A binary value $d_4d_3d_2d_1d_0$ is defined as:

$d_4 \times 2^4 + d_3 \times 2^3 + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$. Note that $2^0 = 1$.

We can rewrite this as: $d_4 \times 16 + d_3 \times 8 + d_2 \times 4 + d_1 \times 2 + d_0 \times 1$.

For example, the binary value 11011 is 1 x 16 + 1 x 8 + 0 x 4 + 1 x 2 + 1 x 1 = 16 + 8 + 0 + 2 + 1 = 27.

Adding binary numbers is easy because there are only 4 combinations of two digits; that is:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1

Consider the addition of the binary values 010101 and 100110.

Step 1: Start with the least-significant bit on the right (these are highlighted in red).

```
0 1 0 1 0 1
1 0 0 1 1 0
```

Step 2: Add these two bits. In this case 1 + 0 = 1. Record the result.

```
0 1 0 1 0 1
1 0 0 1 1 0
          1
```

Step 3: Highlight the next column to the left (the 2s column)

```
0 1 0 1 0 1
1 0 0 1 1 0
          1
```

Step 4: Add these two bits. In this case 0 + 1 = 1. Record the result.

```
0 1 0 1 0 1
1 0 0 1 1 0
        1 1
```

Step 5: Highlight the next column to the left (the 4s column)

```
0 1 0 1 0 1
1 0 0 1 1 0
        1 1
```

Step 6: Add these two bits. In this case 1 + 1 = 0 with a 1 carry. Record the result. The carry bit is in blue and is in the next column to the left because it represents 8. Now, we have to create a new row for any carry bits that we generate.

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
      0 1 1
```

To perform binary addition you need a 3-bit adder (often called a full adder) to add two bits and any carry in generate by the previous state. The truth table for a full adder is as follows.

0 + 0 + 0 = 0
0 + 0 + 1 = 1
0 + 1 + 0 = 1
0 + 1 + 1 = 0 carry 1
1 + 0 + 0 = 1
1 + 0 + 1 = 0 carry 1
1 + 1 + 0 = 0 carry 1
1 + 1 + 1 = 1 carry 1

Step 7: Highlight the next column to the left (the 8s column).

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
      0 1 1
```

Step 8: Add these three bits. Note that we have thee bits to add because we have the carry out from the previous stage. We have to add 0 + 0 + 1 = 1

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
    1 0 1 1
```

Step 9: Highlight the next column to the left (the 16s column).

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
    1 0 1 1
```

Step 10: Add these two bits. In this case 1 + 0 = 1. Record the result.

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
  1 1 0 1 1
```

Step 11: Highlight the next column to the left (the 32s column).

```
0 1 0 1 0 1
1 0 0 1 1 0
    1
  1 1 0 1 1
```

Step 12: Add the two most-significan bits. In this case 0 + 1 = 1. Record the result.

```
0 1 0 1 0 1
1 0 0 1 1 0
      1
1 1 1 0 1 1
```

The final answer is 111011.

Is this correct? The two numbers that we added were 010101 and 100110. These are 1 + 4 + 16 = 21 and 2 + 4 + 32 = 38. Their sum is 21 + 38 = 59.

The binary sum was 111011 which is 1 + 2 + 8 + 16 + 32 = 59.

Let's do a second example with more carry bits. Here we add 011011 and 010111. We will shorten the example by combining the highlight stage and addition.

Steps 1 and 2: Add the least significant bits (the bits in the 1s column)

```
0 1 1 0 1 1
0 1 0 1 1 1
          1   carries
------------
            0
```

Steps 3 and 4: Add the bits in the 2s column. Note that we have 1 + 1 + 1. This is 1 carry 1 (i.e., 1 + 2 = 3).

```
0 1 1 0 1 1
0 1 0 1 1 1
        1 1   carries
------------
          1 0
```

Steps 5 and 6: Add the bits in the 4s column. Note that we have 0 + 1 + 1. This is 0 carry 1.

```
0 1 1 0 1 1
0 1 0 1 1 1
      1 1 1   carries
------------
        0 1 0
```

Steps 7 and 8: Add the bits in the 8s column. Note that we have 1 + 0 + 1. This is 0 carry 1.

```
0 1 1 0 1 1
0 1 0 1 1 1
  1 1 1 1   carries
    0 0 1 0
```

Steps 9 and 10: Add the bits in the 16s column. Note that we have 1 + 1 + 1. This is 1 carry 1.

```
0 1 1 0 1 1
0 1 0 1 1 1
1 1 1 1 1   carries
  1 0 0 1 0
```

Steps 11 and 12: Add the bits in the 32s column. Note that we have 0 + 0 + 1. This is 1 carry 0.

```
0 1 1 0 1 1
0 1 0 1 1 1
1 1 1 1 1   carries
1 1 0 0 1 0
```

Binary Subtraction

Binary subtraction (like addition) is the same as decimal subtraction. Instead of generating carries, borrow bits are generated. The rules for binary subtraction are

0 – 0 = 0
0 – 1 = 1 borrow 1
1 – 0 = 1
1 – 1 = 0

Note that two-bit subtraction and addition yield the same result apart from the carry or borrow bit.

Consider the following simple example of 1101 – 0111

```
1 1 0 1
0 1 1 1
1 1       borrows
0 1 1 0
```

In this case we have 1101 (13) – 0111 (7) = 0110 (6).

In general, this form of subtraction is not used with binary arithmetic because subtraction is formed by the addition of complements are we shall soon see.

Converting Binary to Decimal

There are many ways of converting between binary and decimal. One of the simplest algorithms is to take the leftmost bit and add it to the bit on its right. In order to do this we

must first double that bit before adding it, because a bit in a left-hand column has twice the value of a bit in the column on its right.

We then take this total, double it and add it to the bit on the right and continue doing this until we have added in the least-significant bit. As an example consider the previous result 110010.

Step 1: Take the left-most bit, double it and add it to the bit on the right.

```
1   1   0   0   1   0
    2
    3
```

Step 2:

```
1   1   0   0   1   0
    2
    3   6
        6
```

Step 3:

```
1   1   0   0   1   0
    2
    3   6
        6
          12
          12
```

Step 4:

```
1   1   0   0   1   0
    2
    3   6
        6
          12
              24
              25
```

Step 4:

```
1   1   0   0   1   0
    2
    3   6
        6
          12
              25
                  50
                  50
```

The final result is that 110010 in binary is 50 in decimal. Is this correct? 110010 is $2^5 + 2^4 + 2^2 = 32 + 16 + 2 = 50$.

Binary Multiplication

This is a complicated topic because conventional multiplication is slow and sophisticated hardware and software techniques are often used to mechanise multiplication. However, in principle, binary multiplication is exactly the same as decimal multiplication – it involves multiplication and shifting. We will briefly look at the so called pencil and paper binary multiplication algorithm. Binary multiplication tables are rather easy than decimal multiplication tables because there are only four possibilities (compared to the 100 products from 0 x 0 to 9 x 9 that children have to learn).

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Consider the product 0101 x 1001 (i.e., 5 x 9)

			0	1	0	1	
			1	0	0	1	
			0	1	0	1	first partial product
		0	0	0	0		second partial product
	0	0	0	0			third partial product
0	1	0	1				fourth partial product
0	1	0	1	1	0	1	Sum of partial products

The result is 0101101 which is 45 (i.e., 5 x 9). This multip0lication is exactly the same as decimal multip0lication except that we down have to multiply a row by 0 to 9; we multiply only by 0 or 1 which means that we either write a row of zeroes or we repeat the multiplicand.

Hexadecimal Numbers

In most computer textbooks, you will find the hexadecimal (i.e., 16) number base. Since our base ten digits extend only from 0 to 9, base 16 requires six new digits. Instead of inventing six new digit symbols, computer scientists use the first six letters to represent decimal 10, 11, 12, 13, 14, and 15. Below, we represent the first 16 values in base 2, 10, 16, and 3 (I threw in base 3 for fun – I leave it to you to figure it out).

Computers don't use base 16. People (programmers and computer designers) use base 16 simply because it is more compact than base 2; for example, in base 2 the decimal number 254 is 11111110, whereas in base 16 it is FE. I can remember FE more easily than 11111110. Note that some computer languages indicate base 16 by using the prefix 0x; for example 0x12FE.

The advantage of hexadecimal as a number base is threefold:

It is very easy to convert binary numbers into hexadecimal and vice versa. It can be done mentally without having to use any complicated calculation.

Base			
2	10	16	3
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	10
0100	4	4	11
0101	5	5	12
0110	6	6	20
0111	7	7	21
1000	8	8	22
1001	9	9	100
1010	10	A	101
1011	11	B	102
1100	12	C	110
1101	13	D	111
1110	14	E	112
1111	15	F	120
11000	16	10	121

People (programmers and computer designers) use base 16 simply because it is more compact than base 2; for example, in base 2 the decimal number 254 is 11111110, whereas on base 16 it is FE. I can remember FE more easily than 11111110.

A hexadecimal digit corresponds to four bits. Most computers use data values and addresses that are integer multiples of four bits. Consequently, hexadecimal numbers are well-suited to computer arithmetic.

To convert a binary value into hexadecimal form, all we do is divide the bit string into groups of four starting at the binary point (the rightmost least-significant bit) and then we replace each of the four bits by the corresponding hexadecimal character. Consider the following example in 20 bits:

Binary 00110001101001111110
Binary regrouped 0011 0001 1010 0111 1110
Replace binary groups 3 1 A 7 F
Re-compact 31A7F

Converting Hexadecimal to Binary

This involves the reverse process. Each hexadecimal digit is replaced by the corresponding four binary bits; for example, consider the conversion of E12C into binary. Note that even if the hexadecimal digit is, say, 3, we cannot replace 3 by 11. We have to replace it by 0011.

E = 1110 1 = 0001 2 = 0010 C = 1100
Therefore, E12C = 1110000100101100

Hexadecimal Arithmetic

You can tackle hexadecimal arithmetic in two ways. One is to perform it using hexadecimal arithmetic. The other is to convert to binary. Perform the operation in binary and then convert the result back to hexadecimal. Consider the addition 1E + 2A

In hexadecimal, we add E to A (14 to 10). This gives us 24 decimal which is 18 hexadecimal (remember that F is the largest digit and adding 1 to that gets 10 hexadecimal which is 16 decimal).

Next we add 2 + 1 + 1 (the carry in) to get 4. Therefore, 1E + 2A = 48.

If we were to use binary arithmetic, we would proceed by

1E = 00011110
2A = 00101010

Adding these gets 01001000 which is 48 hexadecimal.

Some texts mention *octal arithmetic* where the base is 8 and the digits are 0,1,2,3,4,5,6,7. Each octal digit represents three binary bits. It's very much the same as hexadecimal arithmetic; for example the octal number 423 is 100010011 in binary. Similarly, 10111010 in binary becomes 10 111 010 when regrouped (remember to group from the right hand end) and is 272 octal.

Octal arithmetic is practically dead today and hexadecimal arithmetic used instead. There are two reasons: hexadecimal arithmetic is more condensed reduction binary strings by a factor of 4 rather than 3 in the case of octal arithmetic. More importantly, hexadecimal arithmetic maps well onto conventional computer systems where wordlengths are 8, 16, 32, or 64 bits, whereas octal arithmetic does not. For example, in 8 bits, the range of possible integer values is 00000000 to 11111111 or 00 to FF in hexadecimal or 000 to 377 in octal arithmetic. If we had nine bit computers the octal range would be 000 to 777 whereas the hexadecimal range would be 000 to 1FF (giving octal arithmetic the advantage). As long as the 8-bit byte reigns supreme, hexadecimal arithmetic will be a natural choice. We will not mention octal arithmetic further.

Example of Hexadecimal Addition

Each student will use their own way of performing hexadecimal addition. My technique is to mentally convert each character to a decimal vale. Add the two decimal numbers. If the sum is less than 16, convert the result to a hexadecimal character. If the sum is 16 or greater, subtract 16, convert the result to a hexadecimal character and record a 1 in the carry column. For example is we add A and 3 that's 10 = 3 = 13 decimal which is a D. If we add A and 9 we get 10 + 9 = 19. We subtract 16 to get 3 9the sum) and a carry of 1. Note that the maximum sum is F + F which is 15 + 15 = 30 which is 14 (i.e., E) carry 1.

```
1 2 F B 1 9
0 3 A C 1 8
        1     carries
-------------
          1
```

```
1 2 F B 1 9
0 3 A C 1 8
        1     carries
-------------
        3 1
```

```
1 2 F B 1 9
0 3 A C 1 8
    1   1     carries
-------------
```

```
        7 3 1


1 2 F B 1 9
0 3 A C 1 8
  1 1   1     carries
-------------
    A 7 3 1


1 2 F B 1 9
0 3 A C 1 8
  1 1   1     carries
-------------
  6 A 7 3 1


1 2 F B 1 9
0 3 A C 1 8
  1 1   1     carries
-------------
1 6 A 7 3 1
```

Fractions

Fractions are numbers smaller than 1. In the decimal positional system we use a decimal point to separate the integer and fractional parts of a number (called a real number); for example 1.125. The fractional part is defined as

$a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + a_{-3} \times 10^{-3} \ldots$

The weightings (place values) for decimal fractions are .1, .01, .001 etc. For example, in 0.123, the weighting of the 3 is 0.01.

In binary arithmetic, a fraction is represented as

$a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + a_{-3} \times 2^{-3} \ldots$

and the weightings are 0.5, 0.25, 0.125, 0.0625 etc. For example, the binary fraction 0.101 is given by 1 x 0.5 + 0 x 0.25 + 1 x 0.125 = 0.5 + 0.125 =0.625.

Another way of looking at binary fractions is to say that 0.101 = ½ + 1/8 = 5/8 = 0.625.

You can convert a binary fraction to decimal form by taking the binary fractional string, starting at the rightmost bit (the least-significant bit) and then adding it to the bit on its left. Of course, if we do this we have to divide the bit by 2 when we move it left. Consider, 0.001. We take the 1 and move it left to get 0.0 ½ . Then we move the least-significant bit left to get 0. ¼ . Finally, we move that left to get 1/8 or 0.125.

Let's take a more interesting case 0.1011

0.101**1**

Begin with the rightmost bit; the highlighted 1. Halve it and add it to the bit on its left to get

0.1 0 3/2

Take the rightmost bit, halve it and add it to the bit on its left to get

0.1 3/4

Take the rightmost bit, halve it and add it to the bit on the left to get

0. 11/8

Take the rightmost bit, and halve it to get 11/16. This is the final result, 11/16 = 0.6875

We can check this by adding powers of 2: 0.1011 = ½ + 1/8 + 1/16 = 11/16 = 0.6875

Converting a Decimal Fraction to Binary

In this case, the fraction is multiplied by 2. Any integer part is recorded. The remaining non-integer part is multiplied by 2. The process continued until the fractional part is zero. Then, the recorded integer parts are used to form the fraction (the first bit retained being the most-significant bit). Consider, 0.625

0.625 x 2 = **1**.250 Integer = 1 Fraction = 0.25
0.250 x 2 = **0**,500 Integer = 0 Fraction = 0.50
0.500 x 2 = **1**.000 Integer = 1, Fraction = 0.00

As you can see, the fraction is 0.00, so we have finished. The final result is 0.101

Now let's try 0.609375

0.609375 x 2 = **1**.218750 Integer = 1 Fraction = 0.218750
0.218750 x 2 = **0**.437500 Integer = 0 Fraction = 0.437500
0.437500 x 2 = **0**.875000 Integer = 0 Fraction = 0.875000
0,875000 x 2 = **1**.750000 Integer = 1 Fraction = 0.750000
0.750000 x 2 = **1**.500000 Integer = 1 Fraction = 0,500000
0.500000 x 2 = **1**.000000 Integer = 1 Fraction = 0.000000 end process

The result is 0.100111

Now let's try the simple decimal value 0.1 or 1/10

0.10000 x 2 = 0.20000 Integer = 0 Fraction = 0.20000
0.20000 x 2 = 0.40000 Integer = 0 Fraction = 0.40000
0.40000 x 2 = 0.80000 Integer = 0 Fraction = 0.80000
0.80000 x 2 = 1.60000 Integer = 1 Fraction = 0.20000
0.60000 x 2 = 1.20000 Integer = 1 Fraction = 0.20000
0.20000 x 2 = 0.40000 Integer = 0 Fraction = 0.40000
0.40000 x 2 = 0.80000 Integer = 0 Fraction = 0.80000
0.80000 x 2 = 1.60000 Integer = 1 Fraction = 0.60000
0.60000 x 2 = 1.20000 Integer = 1 Fraction = 0.20000
0.20000 x 2 = 0.40000 Integer = 0 Fraction = 0.40000

Note that we are in a repetitive loop and that this will continue forever. The binary equivalent of 0.1 is

0.0001100110011001100110011001100…..

In other words you cannot exactly represent 0.1 decimal as a binary sequence because no sequence of fractions ½, 1/4, 1/8, 1/16, … ever adds up to exactly 0.1. You can represent 0.1 to any required level of precision if you use sufficient bits. But you cannot exactly represent the decimal 0.1 as a binary fraction in a finite number of bits.

If this seems strange, remember that we cannot represent 1/3, 1/7, π, or √2 as an exact decimal value.

This result tells us that fractional arithmetic is not exact and that many calculations will be an approximation to the correct answer. However, we can choose the level of accuracy we require by using an appropriate number of bits in a calculation.