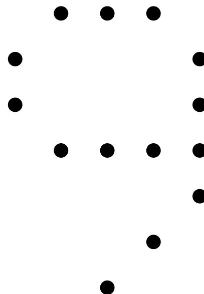
3. BINARY NUMBERS AND ARITHMETIC

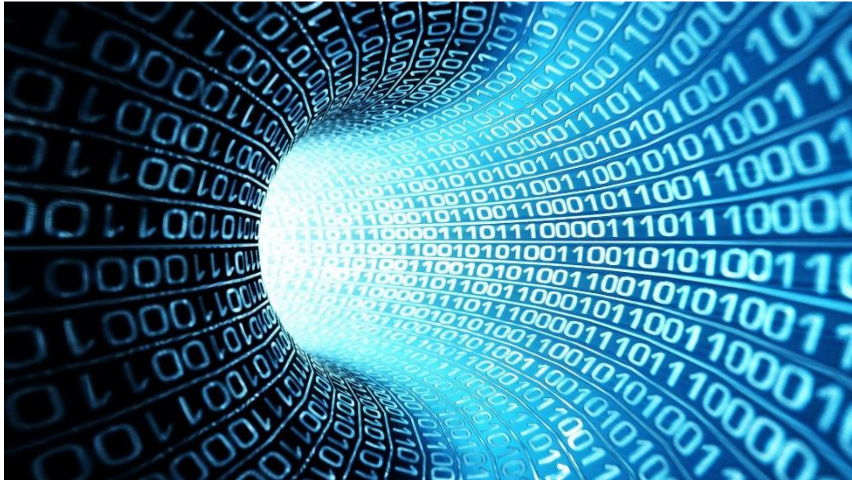
§3.1. Binary Numbers

The only place in a computer where you'll find the number 9 is on the keyboard. Inside the computer you'll only find 0's and 1's. All computer memory consists of electronic 'switches' that can be either ON or OFF. We usually represent 0 by a switch being OFF and a 1 by the switch being ON.

All digits, as they are entered, are immediately converted to binary strings. All of the arithmetic performed by a computer is carried out in binary. Only when a final answer is displayed would you again see a '9'. And even this is binary, in the sense that the shape of the '9' symbol is displayed by an array of 'pixels', tiny dots that can be either ON or OFF.



With the base 10 arithmetic we're used to we use the 10 digits 0 to 9. With base 2 arithmetic, or binary, we only use 0 and 1. We can't call them digits, because that word suggests our fingers. Instead they are called **bits**.



In base 10 arithmetic, when we reach 9 the next number is written '10'. This indicates 1 times 10 plus 0. We then use 2-digits numbers, such as 37. This represents $3 \times 10 + 7$. Finally we reach 99 and if we want to go higher we must use a third digit.

In binary we need to use 2 digits very quickly. The next number after 1 is 10. In binary notation this represents 1 times 2 plus 0. The following number is 11, representing $1 \times 2 + 1$.

The following table displays the first 20 integers in both decimal (base 10) notation and binary (base 2). Examine it carefully.

decimal	binary
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100

Example 1: Convert 1097 to binary.
Solution: We carry out our calculations in a table, with 2 columns. We begin with:

N	B
1097	

• In the **B** column we write the parity of N, that is 0 if N is even and 1 if *N* is odd.
• Then we divide N by 2, throwing away any remainder, and write this underneath the previous value of N.
• We continue in this way until N = 1.

N	**B**
1097	1
548	0
274	0
137	1
68	0
34	0
17	1
8	0
4	0
2	0
1	1

The binary representation of the original N can be read in the **B** column, reading from the bottom up. In this case it's 10001001001.

To convert a binary number to decimal we use a table with 3 columns. We begin as follows:

B	2^n	2^nB
	1	

• In the **B** column we write the bits in the binary number, starting with the right-hand end.
• In the **2^n** column we double from one row to the next.
• In the **2^nB** column we multiply 2^n by B. For simplicity we just leave this blank if B = 0.
• At the bottom we total the **2^nB** column. This is the decimal equivalent.

Example 2: Convert the binary number 10111001011 to decimal.
Solution:

B	2^n	2^nB
1	1	1
1	2	2
0	4	
1	8	8
0	16	
0	32	
1	64	64
1	128	128
1	256	256
0	512	
1	1024	1024
TOTAL		**1483**

§3.2. Binary Arithmetic

Addition of two numbers in binary is very easy. It follows the same process as addition of decimal numbers. The only difference is that you 'carry' whenever the total of a column is greater than or equal to 2, instead of 10. So you might say, "one plus one is 10 – put down the zero and carry the 1".

Computers store all their information in binary. Each unit of memory is called a **bit,** which can represent a 0 or 1. A letter of the alphabet can be easily stored in a **byte**, which is a group of 8 bits. ASCII code does this, but it only uses 7 bits, with the 8th bit used as a **check-bit**. From time to time a piece of memory can be corrupted, by electrical interference or other electronic glitch. The check-bit is 0 if there is an even number of 1's in the other 7 bits and 1 if there is an odd number. So if all is correct the entire byte should have an even number of bits. If ever the computer detects a byte of odd parity (odd number of 1's) it knows that there is an error.

These days we want to be able to store a lot of extra information along with the symbol itself, such as whether it is to be in italics, or bold, or underlined, and which font is to be used. Therefore each letter or symbol is stored in 2 bytes (16 bits).

Numbers have quite a number of different formats, depending on the type of number and the amount of precision required.

The simplest format uses one byte, 8 bits. If a check-bit is used this leaves only 7 bits for the number, and so the only numbers that could be stored in this way are integers in the range 0 to 127 (127 being $2^7 - 1$). But we will illustrate computer arithmetic by using all 8 bytes. This will allow us to store numbers from 0 to 255. Suppose we want to add two 8 bit numbers.

Typically arithmetic processing goes on in the part of a computer called the CPU (central processing unit). The two numbers to be added are copied into the CPU. But, in addition, we need an additional bit, called the ‘carry bit’.

Suppose therefore that we have 17 bits of memory in the CPU. We shall denote these by:

							C
A0	A1	A2	A3	A4	A5	A6	A7
B0	B1	B2	B3	B4	B5	B6	B7

Each is capable of storing one bit, that is, either 0 or 1. The two numbers to be added, each in binary, are copied into the first 16 bits, in order. C is set to 0.

Example 3: Add 171 and 67.
Solution:
171 = 128 + 32 + 8 + 2 + 1, in binary it is 10101011.
67 = 64 + 2 + 1, in binary it is 01000011.

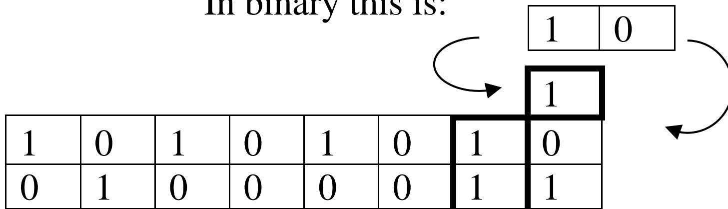
So our CPU contains the following:

							0
1	0	1	0	1	0	1	1
0	1	0	0	0	0	1	1

We start by calculating A7 + B7 + C. This will always be 0, 1, 2 or 3 in decimal. But we have to think binary, so this addition is either 00, 01, 10 or 11. We put the right-hand bit into the current A position (A7) and copy the left hand bit into C.

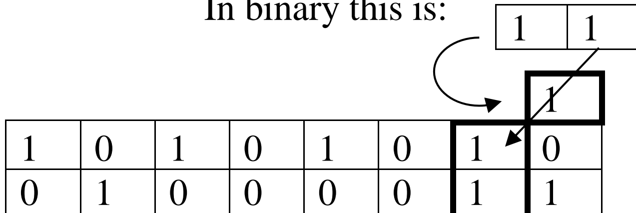
In our example A7 + B7 + C = 2 in decimal

In binary this is:



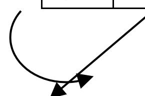
In our example A6 + B6 + C = 3 in decimal.

In binary this is:



In our example A5 + B5 + C = 1.

In binary this 0 1 is:



							0
1	0	1	0	1	1	1	0
0	1	0	0	0	0	1	1

Continuing in this way our CPU becomes:

							0
1	1	1	0	1	1	1	0
0	1	0	0	0	0	1	1

The second register remains unchanged at 67 while the first register now contains the sum, which in decimal is 238 (11101110 in binary). Notice that the check-bit C is 0. This is important, as we will see.

All this is very tedious for us to observe what is happening inside a CPU. If we had to perform the calculation on paper we'd proceed as we do with decimal addition, except for carrying only 1's. Here's the same calculation performed this way.

1 0 1 0 1 0 1 1
$\underline{0\ 1\ 0\ 0\ 0\ 0^1\ 1^1\ 1}$ +
1 1 1 0 1 1 1 0

Example 4: Add 171 to 103.
Solution: We begin with:

							0
1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	1

And end with:

							1
0	0	0	0	1	1	1	1
0	1	1	0	0	0	1	1

Here the answer register contains 00001111 which is the binary equivalent of the decimal number 15.

The correct answer to 171 and 103 is 274, which is too big for our 8 bit registers. So we get 'overflow'. This shows up with a 1 in C. So our CPU would report ERROR.

Of course 8 bit registers are two small for normal use. They can only deal with integers up to $2^8 - 1 = 255$. Usually four bytes (this is called a **word**) are used for each integer. One bit is used to store the sign, leaving 31 bits for the number itself, and this allows for integers in the range – 2,147,483,647 to 2,147,483,647.

However, mostly numbers are stored in 'floating point' format. Here the number is expressed in the format $N \times 10^e$ where N is an integer.

In single precision floating point, 32 bits are used. One bit stores the sign: 0 for + and 1 for –.

Then 8 bits are used to store the exponent and 23 bits for the digits. One of the exponent bits stores the sign of the exponent. This means the exponent can range from –127 to +127. The 23 bits for the integer equivalent

allows for its value to go up to $2^{23} = 8{,}388{,}607 = 2^{23} - 1$, which comfortably allows for 6 significant figures.

Example 5: How would – 428.753 be stored in single precision floating point?

Solution: We write – 428.753 as -428753×10^{-3}.

The negative sign is stored as 1.

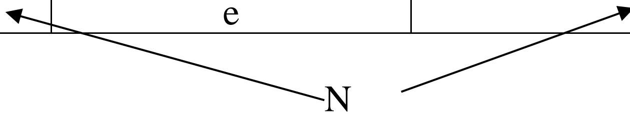
The negative sign of the exponent is stored as 1.

The magnitude of the exponent is 3, which is stored as 00000011.

The significant figures, 428753, are stored as:

00001101000101011010001

$N \times 10^{e}$ where N and e are integers			
sign of N	sign of e	magnitude of exponent	significant figures
1	1	00000011	00001101000101011010001
–	–	3	428753
	e		



Multiplication would be carried out in a somewhat similar way. Again, if we're doing it by hand we can set things out in much the same way as we do with decimal multiplication. But, unlike addition, we may have to carry more than just a 1.

Example 6: Calculate 1101 × 1111.
Solution:

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & & 1 & 1 & 1 & 1 \times \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & 0 \\
 & & 1 & 1 & 0 & 1 & 0 & 0 \\
0^{1} & 1^{10} & 1^{10} & 0^{10} & 1^{1} & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

§3.3. Applications of Binary Numbers

The main application of binary numbers is to computing science, though they do have some applications within mathematics. They also underlie some interesting magic tricks and games.

GUESS WHICH NUMBER I CHOSE

There are 7 cards on each of which there are 64 numbers. The magician shows the cards to a volunteer. “Think of a number between 1 and 63. Don’t tell me what it is. Now go through these cards and give me those cards where your number appears.”

1	3	5	7	9	11	13	15
17	19	21	23	25	27	29	31
33	35	37	39	41	43	45	47
49	51	53	55	57	59	61	63

2	3	6	7	10	11	14	15
18	19	23	24	26	27	30	31
34	35	38	39	42	43	46	47
50	51	54	55	58	59	62	63

4	5	6	7	12	13	14	15
20	21	22	23	28	29	30	31
36	37	38	39	44	45	46	47
52	53	54	55	60	61	62	63

8	9	10	11	12	13	14	15
24	25	26	27	28	29	30	31
40	41	42	43	44	45	46	47
56	57	58	59	60	61	62	63

16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Having collected the cards that contain the chosen number from the volunteer, the magician merely adds the numbers in the top-left corner to reveal the chosen number. For example, if the volunteer chose 37 she would

choose the cards with 1, 4 and 32 at the top left. The magician adds these: $1 + 4 + 32 = 37$.

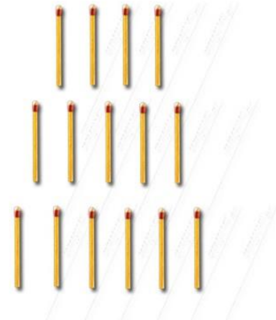
The trick is based on the fact that the first card lists all those numbers whose binary expressions have a 1 in the 1st place (counting from the right), the 2nd card lists all those with a 1 in the 2nd place, and so on. So from the choice of cards we get the binary representation of the chosen number. In our example it would be 100101. But we don't have to do too much mental arithmetic because the appropriate powers of 2 are listed in the top-left corner of the cards.

To make the trick a little more mystifying one should jumble the numbers. But the powers of 2 should still be in a uniform position, perhaps in the bottom-right corner. Also with the numbers going up to 63 it might give the game away that it has something to do with powers of 2. Instead it would be a good idea to only list numbers up to 60, even though this would mean that some cards would have fewer numbers than others.

THE GAME OF NIM

This is a game between two players. Place several piles of matches on the table. The piles should contain different numbers of matches to start with. Players alternate their moves. At each move a player must take 1 or more

matches from one of the piles (they can take a whole pile if they wish, or as little as 1 match). The player who takes the last match wins.



There's a simple strategy based on binary numbers which give you a very high probability of winning, irrespective of whether you go first or second. You would only lose by a fluke, unless the other player is using the same strategy.

The trick is to record the numbers of matches in the piles in binary, with these binary representations being written one under the other.

For example, suppose there are 3 piles, containing 4, 5 and 6 matches. You secretly write (or do it in your head) the binary representations of these numbers as follows. Each row is the binary representation of the number of matches in one of the piles.

1	0	0
1	0	1
1	1	0

The strategy is to take a match. or matches, from one of the piles so that in the binary table there will be an even number of 1's.

The precise algorithm for doing this is as follows:

- Start at the left column.
- Go right to find the first odd column.
- Choose a row with a 1 in this column. This corresponds to the pile you'll choose.
- Change that 1 to a 0.
- Move right across the remaining columns, and in that chosen row change 0's to 1's or vice versa in order that the remaining columns will all be even.

In the above example we can choose any pile. Say we chose the first. We change it to 011, that is, 3. So we take 1 match from the first pile, reducing it from 4 to 3. If we chose the second pile we'd change it to 010, that is 2. This means that we'd take 3 matches from the second pile, reducing it from 5 to 2. Or, if we chose the third pile we would change it to 001, that is 1 in decimal. In this case we would take 5 matches from the third pile, reducing it from 6 to 1.

If we can leave our opponent with even columns in the binary table our opponent is forced to leave you with an odd column. Continuing in this way you're guaranteed to win.

However if, when it's your go, all the columns are already even, you can't make such a move. All you can do is to make a random move, hoping that your opponent doesn't know the binary strategy. If he does, you may lose. But if

he ever leaves you with an odd column you can seize the opportunity and be guaranteed a win.

EXERCISES FOR CHAPTER 3

Exercise 1: Convert 2016 to binary.

Exercise 2: Convert the binary number 11010010 to decimal.

Exercise 3: Perform the following addition in binary: 11101101 + 1100110.

Exercise 4: Perform the following multiplication in binary, using the usual “long multiplication technique”. Then convert the numbers to decimal to check your answer.

$$1011 \times 110.$$

Exercise 5: I choose a number and declare to you, on examining the seven cards, that it is only on the first, third and last cards. What was my number?

Exercise 6: You are playing NIM with 4 piles, with 6, 8, 5 and 13 matches respectively. You have the first move. What pile should you choose, and how matches should you take?

SOLUTIONS FOR CHAPTER 3

Exercise 1:

N	B
2016	0
1008	0
504	0
252	0
126	0
63	1
31	1
15	1
7	1
3	1
1	1

So the number in binary is 11111100000.

Exercise 2:

B	2^n	2^nB
0	1	
1	2	2
0	4	
0	8	
1	16	16
0	32	
1	64	64
1	128	128
TOTAL		**210**

Exercise 3:

$$\begin{array}{r} 1\,1\,0\,0\,1\,1\,0 \\ \underline{1\,1\,1\,0\,1\,1\,0\,1} \\ 1\,0\,1\,1\,0\,0\,0\,1\,1 \end{array}$$

Exercise 4:

$$\begin{array}{rl} 1\,0\,1\,1 & \\ \underline{\quad\quad 1\,1\,0} & \times \\ 1\,0\,1\,1\,0 & \\ \underline{\;\,1\,0\,1\,1\,0\,0} & \\ 1\,0\,0\,0\,0\,1\,0 & \end{array}$$

Exercise 5: 1 + 4 + 64 = 69.

Exercise 6: Take all of the first pile or 2 from the third or 2 from the fourth.