Chapter 1

Data Storage

The Binary Number System

Decimal Number System

Virtually all of us are familiar with the decimal system of numbers. In it, digits are represented by 0-9, and the position of the number dictates its value. How does the position determine the digits real value, though?

Think about a simple number, such as 8. All this number represents is 8 ones, and could even be shown as 1+1+1+1+1+1+1+1, if you really wanted. As we work left, the digits represent a higher value. Take 20, this number represents 2 tens and 0 ones. In fact, we can ignore the ones place if we're just looking at tens. Use 25 instead and the 2 still represents 2 tens. On its own, the 2 could potentially represent anything, but we use a ***positional*** number system, so we know that where the digit is matters. Each position is a different order of magnitude. In decimal's case, each digit is worth 10 times that of the same digit to its right (i.e. 300 is 10 times greater than 30).

As such, the value of each digit can be found by using the equation:

$$10^{\text{Position}} * \text{digit}$$

We actually use the ones place as position 0. For now, just know that this is true. Here's an example using 231:

Step 1: Find the value of each position.

$$10^2 * 2 = 200$$

$$10^1 * 3 = 30$$

$$10^0 * 1 = 1$$

Step 2: Add all positions together to get the final number.

$$200 + 30 + 1 = 231$$

But why do we start at position 0? If you recall, any number raised to the power of 0 is equal to 1. Considering the ones place can only be numbers with a single digit, this must be the case.

What about when we move right of the decimal point? Think of the position as a reversed number line. All numbers left of position 0 are positive, getting larger as we move left, while all numbers right of position 0 are negative, getting smaller as we move right. This fits perfectly with our equation, as negative exponents are used for fractions. So, we would use 0.5 to represent:

$$10^{-1} * 5 = 0.5$$

Binary Number System

Binary works in a very similar way to decimal, just with a different ***radix***. Radix is essentially just another term for base. Decimal, for instance, has the radix 10, while binary has a radix of 2. As we covered, a decimal digit has a value of ten times that of the digit to its right, and binary works the same way. Since the radix is 2, however, each digit has a value of twice that of its neighbor to the right. Due to this, we need to tweak our equation for representation a bit.

In binary, we can convert a number to its decimal representation by the equation:

$$2^{\text{Position}} * \text{digit}$$

Once again with the position starting as 0. Instead of simply applying this equation to binary, however, we can apply it to *all* number systems, by changing it to:

$$base^{position} * digit$$

What the base truly changes, though, is the number of digits possible before moving on to the next place holder. As said before, our decimal system contains the digits 0-9. Altogether, this is 10 unique digits. Binary, on the other hand, only has 0 and 1 to work with- two unique digits. Thus, all binary digits will be only 0 or 1. Having only these two values, we often refer to them in several

different ways. Some of these include: "on" as 1 and "off" as 0; "high" as 1 and "low" as 0; and "true" as 1 and "false" as 0.

Calling them binary digits all the time would get tiring, and as a result the shortened name ***bit*** was created. As bits are already in base 2 as well, they are generally grouped in powers of two (i.e. 1, 2, 4, 8). Commonly, eight bits are grouped together to create what is known as a ***byte.***

Binary Conversion

Since we're most familiar with decimal, it is always useful to be able to convert to-and-from binary and decimal. Converting binary to decimal is straightforward using our equation.

Let's use the 8-bit binary number 10110111. To convert this to decimal, we would do the following:

Step 1: Find the value of each position.

$2^0 * 1 = 1$

$2^1 * 1 = 2$

$2^2 * 2 = 4$

$2^3 * 0 = 0$

$2^4 * 1 = 16$

$2^5 * 1 = 32$

$2^6 * 0 = 0$

$2^7 * 1 = 128$

Step 2: Add all position values together.

$$1 + 2 + 4 + 0 + 16 + 32 + 0 + 128 = 183$$

Converting decimal to binary is far less straightforward. We will use the decimal number 53 as an example:

Step 1: Divide the number by 2 and keep the remainder, R.

$$\frac{53}{2} = 26 \text{ R } \mathbf{1}$$

Step 2: Store the remainder in its own number. We'll call this number our "marks." Use the division result as your new number.

New number: 26

Marks: **1**

Step 3: Repeat steps 1 and 2 until you are left with 0 as your number, placing the new remainder left of your current markings each time.

$$\frac{26}{2} = 13 \text{ R } \mathbf{0}$$

New number: 13

Marks: **0**1

$$\frac{13}{2} = 6 \text{ R } \mathbf{1}$$

New number: 6

Marks: **1**01

$$\frac{6}{2} = 3 \text{ R } \mathbf{0}$$

New number: 3

Marks: **0**101

$$\frac{3}{2} = 1 \text{ R } \mathbf{1}$$

New number: 1

Marks: **1**0101

$$\frac{1}{2} = 0 \text{ R } \mathbf{1}$$

New number: 0

Marks: **1**10101

Our new number is 0, so our marks are the final number converted to binary.

$$110101_2 = 53_{10}$$

Conversion brings up an interesting question: how many combinations are possible with a given number of bits? Once again, we can use the radix to find out. Since there are two possible states for each bit, and we have a number, n, bits, we must multiply 2 times itself n times, or in other words:

$$\text{Total combinations} = 2^n$$

For 4 bits this means we have 2^4, or 16, possible combinations. Since this is exponential, increasing the number of bits slightly can have a huge impact on the total combinations. Four bits could only produce 16 unique numbers, while eight bits can produce 2^8, or 256, unique numbers. As 0 takes up one number, values for these ***unsigned*** integers will range from 0 to 255. These numbers are called "unsigned" as they can only be positive, there is no sign to make them negative.

When referring to the leftmost or rightmost bits, we choose to call them the "most significant" and "least significant" bits respectively. This is because the least significant bit has the lowest positional value, only contributing 1 or 0 to the total value, while the most significant bit has the highest positional value. In an 8-bit number, the most significant bit contributes 128 in decimal.

Binary Arithmetic

Binary works in the same way but with far fewer digits. Let's add the binary equivalent of 11 + 2 by using 1011 + 0010; we will get 13 for an answer, which is 1101:

Step 1: Split all bits.

$$\mathbf{1}_3 + \mathbf{0}_2 + \mathbf{1}_1 + \mathbf{1}_0 = 1011$$

$$\mathbf{0}_3 + \mathbf{0}_2 + \mathbf{1}_1 + \mathbf{0}_0 = 0010$$

Step 2: Add least significant bits, moving to most significant (**right to left**).

$$\mathbf{1}_0 + \mathbf{0}_0 = \mathbf{1}_0$$

$$\mathbf{1}_1 + \mathbf{1}_1 = \mathbf{0}_1$$

Adding 1 + 1 will give us 2, but we aren't working with 2's, only 0's and 1's. So, the extra gets carried over to the $\mathbf{X}_2$ spot. Due to the carry-over, the first $\mathbf{0}_2$ will become a $\mathbf{1}_2$.

$$\mathbf{1}_2 + \mathbf{0}_2 = \mathbf{1}_2$$

$$\mathbf{1}_3 + \mathbf{0}_3 = \mathbf{1}_3$$

Step 3: Now we have finished with the addition and carry-over. Next is to realign, from right to left, to acquire our answer to 1011 + 0010.

$$\mathbf{1_3 1_2 0_1 1_0} \text{ or } \mathbf{1101}$$

$$\mathbf{1101 = 8 + 4 + 0 + 1 = 13}$$

Logic Operations, Bitwise Operators, and Truth Tables

Logic operations, while working with digital computers, are methods to test relationships between data. To do so, a truth table may be created to visually present outcomes of comparison. While there are several logic operations to compare data (AND, OR, NOT, NAND, NOR, XOR, and XNOR), there are four that are most commonly used in teaching/learning elementary computer science: AND, OR, NOT, and XOR; however, all seven will be covered. For the following examples of truth tables, we will compare two 4-bit binary numbers, 0110 and 1100.

Bitwise operators are symbols used to shorten code and replace having to type out the full names/values operators. Operators may be implemented slightly differently among the many different programming languages, but their general root form is as follows:

Bitwise Operators	
&	AND
|	OR
~	NOT
^	XOR
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift
&=	AND assignment
|=	OR assignment
^=	OR assignment
<<=	Left shift and assignment
>>=	Right shift and assignment
>>>=	Unsigned right shift and assignment

The AND operator returns True if all compared inputs are 1 or True. In computer programming AND is often represented by an & symbol to compare two or more inputs.

AND		
A	**B**	**AB**
0	1	False
1	1	True
1	0	False
0	0	False

The OR operator returns True if any of its inputs are 1 or True. In computer programming OR is often represented by a | symbol to compare two or more inputs.

OR		
A	**B**	**AB**
0	1	True
1	1	True
1	0	True
0	0	False

The NOT operator returns True if an input is 0 or False and will return False if an input is 1 or True. In computer programming NOT is often represented by using the ! symbol or ~ symbol. NOT is the *only* operator to take only one input.

NOT	
A	**AB**
0	True
1	False
1	False
0	True

The NAND operator returns True if none of the compared inputs are all 1's or True. NAND is a combination of NOT and AND and is often represented in computer programming by using a combination of ! or ~ with an &.

NAND		
A	**B**	**AB**
0	1	True
1	1	False
1	0	True
0	0	True

The NOR operator returns True if any of the compared inputs are 0 or False. NOR is a combination of NOT and OR and is often represented in computer programming by using a combination of ! or ~ with a |.

NOR		
A	**B**	**AB**
0	1	False
1	1	False
1	0	False
0	0	True

The XOR, *exclusive-OR*, logic operator is used to determine if **exactly** one of the inputs being compared is 1 or True. XOR, exclusive-OR, is often represented using a ^ symbol in computer programming.

XOR		
A	**B**	**AB**
0	1	True
1	1	False
1	0	True
0	0	False

The XNOR, *exclusive-NOT-OR*, bitwise operator is used to determine if two or more inputs are all the same, which would return True. The difference with this logic operator, is that all compared inputs must be either all True or all False. In computer programming, XNOR is often represented by using a combination of ! or ~ with a ^.

XNOR		
A	**B**	**AB**
0	1	False
1	1	True
1	0	False
0	0	True

Bit Shifting and Rotating

We have two more operations to discuss that may not be as familiar. These are bit shifts, and bit rotations. For both of these, the positions of the binary number are moved left or right, but the difference is what we will fill the number in with. In a bit shift, we consider the bits to "fall off" the end of the number, so we fill in the missing bits with zeroes. With rotations, we instead move the bits from one end to the missing end.

First, let's look at how shifts would work in decimal.

Let's shift the number 540 right. To do this, we will just take all our numbers and move them right once, while ignoring any numbers that become fractions. We will fill in the left side with zeroes.

We start with 540, then we move each digit one position ***right***.

540 becomes **054** **|0**

Our last digit "fell off" the right side of the number, so we discard it. We now have 54, but since we had three digits to start, we can fill in a zero at the beginning. Thus, our final number is 054, or just 54. This is the same as *dividing* our number by 10, since decimal is base 10.

Now let's shift the number 014 ***left***.

Once again, we start with 014, then move each digit one position left. Note that we have three digits total here. **|000** are spare, imaginary zeroes.

014 | 000 becomes **140 | 00**

As you can see. Our 0 digit "fell off" the left side and we discarded it. On the right side, we pulled a 0 from imagination land to fill in the one's place after the 1 and 4 shifted left.

Our final number is 140. This is similar to the right shift, except in this case, we *multiplied* the number by 10.

Moving over to binary, bit shifts work in the same way, just with a different base. Let's shift the binary number 011001 (25) ***left*** one.

011001 becomes **110010**

Our most significant bit falls off and is discarded, and now we need to fill in the missing bit. Our final number is 110010 in binary, which is equal to 50 in decimal. Since we are in base 2, it makes sense that the number should be multiplied by 2.

Next, if we move it ***right***, we should expect it to be divided by 2.

011001 becomes **001100**

Our number is now one bit short, so we fill in the missing slot on the far left with a zero to get 001100. The 1 on the far right ends up being dropped.

Finally, we get 001100, or 12 in decimal. This isn't exactly half of 25, as we had to get rid of our final bit. So, we can consider right bit shifts to be the same as integer division, as they leave no fractional portion. This bit has "fallen off" the right side of the number.

Let's look at what happens when a bit falls off the left side of a number. For this we will use 11010110- 214 in decimal. We will shift the number left once.

11010110 becomes **10101100**

We can already see that our new number is smaller than our original number by looking at the first two digits; 11 became 10. 10101100 is equal to 172 in decimal. This is because the most significant bit "fell off" when it was shifted left, since we only had 8 bits to work with. Our last step is to fill in the right side with a zero, and we have our number.

Rotations, on the other hand, fill in the missing bits with whatever falls off the other side. Again, let's look at this in decimal first.

Let's rotate the number 308 ***left*** once. We will consider this number to only have 3 digits at most.

308 becomes **083**

In this case, instead of simply removing the 3 on one end and adding a 0 on the opposite, we will fill in the missing digit with our fallen digit.

If we were to rotate 308 ***right*** once instead, it would look like this:

308 becomes **830**

Now we can extend this operation to binary easily. Let's use 1011 as an example, and we can rotate it left by one digit.

1101 becomes **1011**

And if we instead rotate it right once:

1101 becomes **1110**

These bit rotations are just a different method of how we fill in our missing bits after we shift them.

Negative Numbers: Two's Complement and Excess Notation

Computers can only read binary numbers at their core. This means -10 could not be read as -2. In fact, it couldn't be read at all. Instead, we must use a method to convert some numbers to negative numbers using only the bits we have.

Important to note is that, since we must use our limited bits to produce negative *and* positive numbers, the highest maximum value for these so-called ***signed*** numbers is half of the possible values minus 1 because '0' is part of the positive spread, and the lowest minimum value is half of the possible values and negative. For signed numbers, the most significant bit determines if it is positive or negative.

To demonstrate, let's once again use 1 Byte which holds 8 bits.

Step 1: Find the total possible values.

$$\mathbf{2^8} = 256$$

Step 2: Divide by 2 to get half of the total possible values. Each spread of positive and negative will this many values.

$$\frac{\mathbf{256}}{\mathbf{2}} = 128$$

Step 3: To get the lower bound (lowest negative value), make this number negative.

For the upper bound (highest maximum value), subtract 1.

$$\textbf{\textit{Upper bound}}: \mathbf{127}$$

$$\textbf{\textit{Lower bound}}: \mathbf{-128}$$

One method represent negative numbers in binary is ***Two's Complement***.

3-bit Two's Complement	
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

4-bit Two's Complement	
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Two's Complement works by checking if the sign bit (the most left bit) is a 1. If it is, then the remaining bits are flipped, 1's turn to 0 and 0's turn to 1, and one is added afterwards. The result is

read normally, but as a negative number. If the sign bit is 0, read the number normally as a positive. Let's use two examples, 101 and 010.

Step 1: Check if the most significant bit (on the left) is 1.

$$101 \rightarrow Yes$$

$$010 \rightarrow No$$

Step 2: If yes, flip the remaining bits and add 1 to get the final number. (i.e. all 0's become 1's and vice versa)

101 → **01** becomes **10** then becomes **11**

010 → **10** stays **10**

---

11 is 3 in binary

10 is 2 in binary

Step 3: If the sign bit is 1, the final number is negative.

$$101 = -3$$

$$010 = 2$$

Addition in Two's Complement works just like unsigned addition, but with one catch: any bits added onto the number by this addition are truncated back down to the original number of bits. We can show this by adding -2 and 3. We'll need only three bits for this, and the numbers will be represented 110 and 011 respectively.

Step 1: Add numbers together normally.

$$110 + 011 = 1001$$

Step 2: If the most significant bit extends past how many bits we started with, cut it off.

- 1001 is four bits, but we started with three.
- Get rid of the first digit, 1.
- The final number is 001
- Now read 001 in Two's Complement. 001 is positive, so no digits are changed. 001 is +1 in two's complement.

Once again, this is to be expected as -2 + 3 = +1.

Another method exists known as ***Excess Notation***.

3-bit Excess Notation	
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

4-bit Excess Notation	
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

In excess notation, the ***zero point*** is the number which corresponds to, well, 0. This is always when the most significant bit is 1 with only 0's following. For this reason, the form of excess

notation being used is known as "excess ____" with the blank being the zero point (i.e. excess 8 if four bits are used). Each number greater than the zero point is read normally as a positive number, while each number less than it is read as a negative number. The negative number has a magnitude of its distance away from the zero point. We'll use 001101 as an example in excess 32.

Step 1: Identify the zero point.

$$In\ excess\ 32, the\ zero\ point\ is\ 32$$

Step 2: If the number has a sign bit of 0, subtract it from the zero point.

$$100000 - 001101 = 010011$$

Step 3: Read this number normally, and if the first bit is a 0, it is negative. If the first bit is 1, read it normally, ignoring the most significant bit.

$$010011 = 19$$

$$Therefore\ 001101 = -19\ in\ excess\ 32$$

In addition, if all of the bits in a number in excess notation are 1, the number is the positive zero point minus 1. If all bits are 0, the number is the negative zero point.

In excess 32:

$$111111 = 32 - 1 = 31$$

$$000000 = -32$$

Binary Fractions

Up until now we have worked only with binary integers, but this ignores a rather large section of numbers- an infinitely large section, even. Binary fractions work just the same as they do in decimal.

Using our conversion method, we've been assuming the "ones" place to be position 0, this is because when fractions are involved, the first fractional place will use the position -1. While at first this may seem odd, think of it this way: the "tens" place in decimal uses position 1, so the tenths place should use position -1. The same is true for hundreds and hundredths and so on. As there is no "oneths" place, the "one's place must take a number with no negative value, such as 0.

Let's convert the binary number 10.011 to decimal using our positional equation:

Step 1: Convert each position to decimal.

$$2^1 * 2 = 2$$

$$2^0 * 0 = 0$$

$$2^{-1} * 0 = 0$$

$$2^{-2} * 1 = \frac{1}{4}$$

$$2^{-3} * 1 = \frac{1}{8}$$

Step 2: Add all positions together.

$$2 + 0 + \frac{1}{4} + \frac{1}{8} = 2\frac{3}{8} = 2.375$$

Problems are raised by this method, such as how to represent fractions that aren't powers of two. 0.3 is an easy number to represent in decimal, but how would you represent it with only a few bits? Let's use 4 bits, not including the whole numbers.

Step 1: First, we assign 0 as the whole number portion and add a radix point.

$$Current\ binary\ number: 0.$$

Step 2: We check if 0.1 (.1 = ½) will be too high. In this case, it is, so we place a 0 there.

$$Current\ binary\ number: 0.0$$

Step 3: After this, we repeat, checking if 0.01 will be too high and so on. 0.01 = .25, thus it is not too high and can be added on.

$$Current\ binary\ number: 0.01$$

Step 4: Adding 0.001, 0.125 in decimal, is too high, so another 0 is added.

$$Current\ binary\ number: 0.010$$

Step 5: We have only one bit remaining, and it corresponds to 0.0625, which is still too high.

$$Final\ binary\ number: 0.0100$$

Our final binary number then is 0.0100, corresponding to 0.25 in decimal. This means our fraction is 0.05 lower than our target of .30 when limiting ourselves to only four bits.

Floating Point

Once again, we've run into the problem of computers only reading numbers. Computers don't read radix points, so we need another new method of storing fractions. Solving this problem is ***floating-point*** **notation.** Floating point numbers work by breaking the number down into two or three segments, depending on whether it is a signed number or not. For our purposes we will assume our floating-point values are signed and will be represented with 8 bits.

The segments are as follows:

- The most significant bit is the sign bit.
- The three bits following this are the exponent bits.
- The four final bits are the mantissa, or significand.

First is the sign bit. Like in Two's Complement, if the sign bit is 1, the number is negative, and if it is 0, the number is positive. For example, the number 10011001 would be negative, while 00011001 would be positive.

Next is the mantissa. This number will always lie between 1 inclusive and 2 exclusive. This is achieved by assuming a 1 in front of the mantissa, along with a radix point. Thus, we look at the mantissa's four bits like this:

$$(1.)XXXX$$

Where the X's can be either 1 or 0 each. 1.5, for instance, would be represented as follows:

$$1.1000$$

While at first this looks like 8, we have to remember that a "1." is always assumed in front of the number. From this, we can determine that the smallest number possible for our mantissa is 1.0001, or 1.0625.

The exponent segment is comprised of three bits which will determine how to multiply the mantissa. Essentially, we take the number produced by these three bits and use it as a power of 2, which we then multiply the mantissa by. This number is found using excess notation. Let's say we wanted the exponent -2. For this we would just need to find -2 in excess notation.

Step 1: -2 is two down from the zero point, so we take 100 and subtract 010

$$100 - 010 = 010$$

Thus, in excess 4, our exponent of -2 is represented as 010.

The final equation is as follows:

$$(-1)^{sign\ bit} * 2^{exponent} * mantissa$$

It's definitely time for an example for this, so let's use 10010110 and convert it to decimal.

Step 1: Look at the sign bit.

As the sign bit is 1, the number will be negative

Step 2: Determine the exponent.

$$100 - 001 = 011$$

So the exponent is: -3

Step 3: Determine the mantissa.

$\mathbf{0110}$ becomes $\mathbf{1.0110}$

$$\mathbf{1.0110 = 1.375}$$

Step 4: Use the floating-point equation to find the final number.

$$(-1)^1 * 2^{-3} * 1.375$$

Which simplifies to:

$$-\frac{1}{8} * \frac{11}{8} = -\frac{11}{64} = -0.171875$$

How could we convert *to* floating point, instead of *from* it? The process is as follows, using 13 as the starting value:

Step 1: Convert the number to binary.

$$13 \rightarrow 1101$$

Step 2: Place a radix point between the first two bits.

$$1101 \rightarrow 1.101$$

Step 3: Add zeroes *or* remove digits at the end until you're left with 4 bits following the radix. The bits following the radius are the mantissa.

$$(1).101 \rightarrow (1).\mathbf{1010}$$

Note: the most significant bit is assumed to be there, but we can ignore it.

Step 4: Shift the radix point to where it would be in the original number and count how far it moved. If it moved right, the number is positive, if it moved left, negative.

$$(1).1010 \rightarrow (1)101.0$$

Step 5: The radix point moved three places to the right. Convert this number to excess notation. This is the exponent.

$$3 \rightarrow 111$$

Step 6: Finally, put the exponent and mantissa together, along with the sign bit; 0 for positive, 1 for negative.

(**Sign bit** :: **exponent** :: **mantissa**)

$$0\ 111\ 1010 \rightarrow 01111010$$

This is how we would represent 13 in floating point notation. To check it, let's run through the conversion.

- The sign bit is **0**, so the number is positive.
- Since we have 3 exponent bits, the exponent is in excess 4. The first bit, 1, means the number is positive, and the remaining 11 represents **3**.
- The mantissa is 1010, which really represents 1.1010, which is **1.625** in decimal.
- Finally, use the equation:

$$(-1)^0 * 2^3 * 1.625 = 1 * 8 * 1.625 = 8 * 1.625 = 13$$

It's easy to see how useful floating-point notation can be. Of course, floating-point numbers don't use only 8 bits normally. Generally, in accordance with IEEE-754, 32- and 64-bit floating-point numbers are used, giving a very wide range of numbers of far higher precision than what we used.

Precision is important for many tasks, but floating-point does run into issues with it. As mentioned before, some numbers simply aren't achievable with a set number of bits, such as 0.3. With 32 or 64 bits, however, we can get very close to this value. A number such as 10000.3 would be significantly harder to reach, due to the nature of exponents.

The number of floating-point values between each interval, being powers of two in binary, is equal to all other intervals. In other words, if there are 100 values between 1 and 2, there are also only 100 values between 64 and 128. While that may not be much of an issue at lower intervals, imagine only having 100 values to represent numbers between 4096 and 8192.

Of course, with 32 or 64 bits, this isn't too much of a problem until the numbers become very large, but it is important to consider as numbers will become less accurate the larger they become.

EXAMPLES

1) Convert the binary number **1011** to decimal.

First, we need to know what each position value is. We start with 2^3, then 2^2, 2^1, and 2^0. This means the positions are equal to:

1 0 1 1

↑ ↑ ↑ ↑

8 4 2 1

Now we can add together the values of the positions where we have ones.

$$8 + 0 + 2 + 1 = 11$$

So, the value of the binary number 1011 in decimal is 11.

2) Convert the binary number **01011101** to decimal. We'll first get the value of each position.

$$2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, \text{and } 2^0$$

$$128, 64, 32, 16, 8, 4, 2, 1, 0$$

Now convert those powers of two into decimal numbers.

64 16 8 4 1
↓ ↓↓↓ ↓
0 1 0 1 1 1 0 1
↑ ↑ ↑
128 32 2

Then we can add together the values of the positions with ones.

$$64 + 16 + 8 + 4 + 1 = 93$$

Thus, the decimal value of the binary number 01011101 is 93.

3) Convert the decimal number **13** to a **4-bit** binary number.

First, we'll divide the number by two until we end up with 0, holding onto the remainder each time.

$$13 / 2 = 6 \text{ R } \mathbf{1}$$

$$6 / 2 = 3 \text{ R } \mathbf{0}$$

$$3 / 2 = 1 \text{ R } \mathbf{1}$$

$$1 / 2 = 0 \text{ R } \mathbf{1}$$

Now, we can place the remainders together in reverse order to get our binary number.

$$1011 = 13$$

So, we find that the decimal number 13 as a 4-bit binary number is represented as 1101.

4) Convert the decimal number **122** to an **8-bit** binary number.

Divide our first number by two until we get to 0, keeping track of the remainder each time.

122 / 2 = 61 R **0**

61 / 2 = 30 R **1**

30 / 2 = 15 R **0**

15 / 2 = 7 R **1**

7 / 2 = 3 R **1**

3 / 2 = 1 R **1**

1 / 2 = 0 R **1**

Once we have our remainders, we can place them together from the bottom up to get our binary number.

1111010

In this case, we need just one more bit to give us an 8-bit number, so we can add a zero to the beginning.

122 = 01111010

5) Find the binary and decimal results of adding **1100 + 0110**.

First, we split the bits to find their positions.

$$1100 \rightarrow 1_3 + 1_2 + 0_1 + 0_0$$

$$0110 \rightarrow 0_3 + 1_2 + 1_1 + 0_0$$

Next, we add each position from the first number with the corresponding position of the second number. Carry values when necessary (highlighted yellow).

$$0_0 + 0_0 = \mathbf{0_0}$$

$$0_1 + 1_1 = \mathbf{1_1}$$

$$1_2 + 1_2 = \mathbf{0_2} + 1_3$$

$$1_3 + 0_3 + 1_3 = 0_3 + \mathbf{1_4}$$

Now we can put together our resulting bits in their correct positions.

$$11010$$

So, the addition of 1100 and 0110 is equal to 11010. Now let's convert that to decimal.

1 1 0 1 0

↑ ↑ ↑ ↑ ↑

16 8 4 2 1

16 + 8 + 0 + 2 + 1 = 26

Finally, we find that 1100 + 0110 is equal to the decimal value 26.

6) Find the binary and decimal results from adding **10011110 + 00111011**.

First, split all bits into their positions.

$$10011110 \rightarrow 1_7 + 0_6 + 0_5 + 1_4 + 1_3 + 1_2 + 1_1 + 0_0$$

$$00111011 \rightarrow 0_7 + 0_6 + 1_5 + 1_4 + 1_3 + 0_2 + 1_1 + 1_0$$

Next, we add each position of the first number with the respective positions of the second. Carry values when necessary (highlighted yellow).

$$0_0 + 1_0 = \mathbf{1_0}$$

$$1_1 + 1_1 = \mathbf{0_1} + 1_2$$

$$1_2 + 0_2 + 1_2 = \mathbf{0_2} + 1_3$$

$$1_3 + 1_3 + 1_3 = \mathbf{1_3} + 1_4$$

$$1_4 + 1_4 + 1_4 = \mathbf{1_4} + 1_5$$

$$0_5 + 1_5 + 1_5 = \mathbf{0_5} + 1_6$$

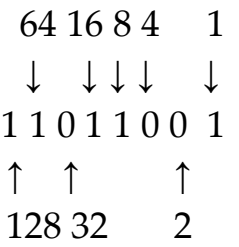
$$0_6 + 0_6 + 1_6 = \mathbf{1_6}$$

$$1_7 + 0_7 = \mathbf{1_7}$$

Now, we can put together the results that we didn't carry over to get our number.

$$11011001$$

Since we have the binary number, now let's get the decimal value.



128 + 64 + 16 + 8 + 1 = 217

We now have our binary number of 11011001 and our decimal value of 217 by adding 10011110 and 00111011.

7) Find the value of **1010** in **two's complement**.

Check if the sign bit is 1 or 0. The sign bit is the most significant bit.

$$\mathbf{1}010$$

Since it is, flip the remaining bits.

$$010 \rightarrow 101$$

Add 1 to the value.

$$101 \rightarrow 110$$

Now read the value as if it were unsigned.

$$1\ 1\ 0$$

$$\uparrow\ \uparrow\ \uparrow$$

$$4\ 2\ 1$$

$$4 + 2 + 0 = 6$$

Since the sign bit in the beginning was 1, the number is negative. Thus, our final number is **-6**.

8) Find the value of **0111** in **two's complement**.

Check if the sign bit is 1 or 0. The sign bit is the most significant bit.

0111

As the sign bit is 0, we can read the number as if it were unsigned.

1 1 1

↑ ↑ ↑

4 2 1

4 + 2 + 1 = 7

Our final number is +7.

9) Find the value of **10010100** in **two's complement**.

Check the sign bit. The sign bit is the most significant bit.

$$\mathbf{1}0010100$$

Since the sign bit is 1, flip the remaining bits, then add 1 to the result.

$$0010100 \rightarrow 1101011$$

Add 1 to the value.

$$1101011 + 1 \rightarrow 1101100$$

Now we can read this number normally, but as a negative.

$$\begin{array}{c} 64\ 16\ 8\ 4\quad 1 \\ \downarrow\quad \downarrow\downarrow\downarrow\quad \downarrow \\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ \ 0 \\ \uparrow\quad \uparrow\qquad \uparrow \\ 128\ 32\qquad 2 \end{array}$$

$$64 + 32 + 0 + 8 + 4 + 0 + 0 = 108$$

Remember, at the start our sign bit is negative, making our final number -108.

10) Find the value of **00011010** in **two's complement**.

Check the sign bit.

00011010

Since the sign bit is 0, the number is positive and can be read normally.

64 16 8 4 1
↓ ↓↓↓ ↓
0 1 1 0 1 1 0 0
↑ ↑ ↑
128 32 2

32 + 16 + 2 = 50

Therefore, our final number is 50.

11) Add the binary numbers **1011 + 0111** in **4-bit two's complement**.

Remember that two's complement addition works almost exactly like unsigned addition. We will start by separating the bits by their positions (carried numbers are highlighted).

$$1011 = 1_3 + 0_2 + 1_1 + 1_0$$

$$0111 = 0_3 + 1_2 + 1_1 + 1_0$$

Next, we add the positions, carrying over numbers when necessary.

$$1_0 + 1_0 = \mathbf{0_0} + 1_1$$

$$1_1 + 1_1 + 1_1 = \mathbf{1_1} + 1_2$$

$$0_2 + 1_2 + 1_2 = \mathbf{0_2} + 1_3$$

$$1_3 + 0_3 + 1_3 = \mathbf{0_3} + 1_4$$

$$1_4 = \mathbf{1_4}$$

Now we can put the numbers we didn't carry together to form our binary number.

10010

The difference between unsigned addition and two's complement addition is that our final number in two's complement must be truncated back down to 4-bits.

$$10010 \rightarrow 0010$$

Finally, we can read this number as a regular two's complement number. As the sign bit is 0, it will be positive and can be read normally.

$$0\ 0\ 1\ 0$$

$$\uparrow\ \uparrow\ \uparrow\ \uparrow$$

$$8\ 4\ 2\ 1$$

$$0 + 0 + 2 + 0 = 2$$

Thus, our final number is 2.

12) Add the binary numbers **01101101 + 01011101** in **8-bit two's complement**.

Start by breaking apart the numbers into their positions.

$$01101101 \rightarrow 0_7 + 1_6 + 1_5 + 0_4 + 1_3 + 1_2 + 0_1 + 1_0$$

$$01011101 \rightarrow 0_7 + 1_6 + 0_5 + 1_4 + 1_3 + 1_2 + 0_1 + 1_0$$

Then add the positions, carrying when necessary. Carried numbers will be highlighted.

$$1_0 + 1_0 = \mathbf{0_0} + 1_1$$

$$0_1 + 0_1 + 1_1 = \mathbf{1_1}$$

$$1_2 + 1_2 = \mathbf{0_2} + 1_3$$

$$1_3 + 1_3 + 1_3 = \mathbf{1_3} + 1_4$$

$$0_4 + 1_4 + 1_4 = \mathbf{0_4} + 1_5$$

$$1_5 + 0_5 + 1_5 = \mathbf{0_5} + 1_6$$

$$1_6 + 1_6 + 1_6 = \mathbf{1_6} + 1_7$$

$$0_7 + 0_7 + 1_7 = \mathbf{1_7}$$

Now we can put the numbers in our results together, ignoring the carried numbers.

$$11001010$$

The sign bit is 1, meaning the number is negative. Why is this the case despite adding two positive numbers? Recall that if a binary number becomes too large, it can overflow and become very small. This is due to only having 7-bits to represent our value, meaning the largest number we can represent is only 127. Any larger and we wrap back around. So, we flip the remaining bits and add 1 to this result to get our final number.

$$1001010 \rightarrow 0110101$$

$$0110101 + 1 = 0110110$$

Now let's convert this number to decimal to see what our result is.

$$\begin{array}{c} 64\ 16\ 8\ 4\quad 1 \\ \downarrow\quad \downarrow\downarrow\downarrow\quad \downarrow \\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ \ 0 \\ \uparrow\quad \uparrow\qquad \uparrow \\ 128\ 32\qquad 2 \end{array}$$

$$32 + 16 + 4 + 2 = 54$$

We know our final number must be negative because the sign bit is 1, so our result is -54.

13) Convert the decimal number **-5** into a **4-bit two's complement** binary number.

First, we know that the sign bit must be 1, so the remaining 3 bits are what we will use for the 5. We can just convert the 5 into binary normally.

$$5 / 2 = 2 \text{ R } \mathbf{1}$$

$$2 / 2 = 1 \text{ R } \mathbf{0}$$

$$1 / 2 = 0 \text{ R } \mathbf{1}$$

$$101$$

Now we can reverse the process of finding a negative value in two's complement by subtracting one from the value then flipping the bits.

$$101 - 001 = 100$$

$$100 \rightarrow 011$$

Finally, we can put the sign bit on the current number to get our two's complement value.

$$1011$$

So, 1011 is our binary representation of -5 using two's complement.

14) Convert the decimal number **98** into an **8-bit two's complement** binary number.

We know that the sign bit must be 0, as the number is positive. We will then use the remaining 7 bits to represent the number.

98 / 2 = 49 R **0**

49 / 2 = 24 R **1**

24 / 2 = 12 R **0**

12 / 2 = 6 R **0**

6 / 2 = 3 R **0**

3 / 2 = 1 R **1**

1 / 2 = 0 R **1**

1100010

Since the number is positive, all we must do now is place the sign bit at the beginning of the number.

01100010

So, our 8-bit two's complement binary representation of 98 is 01100010.

15) Convert the binary number **0110**, represented in **excess 8**, to its decimal value.

First, take a look at the sign bit. In excess, 0 represents negative numbers.

$$\mathbf{0}110$$

Since our sign is 0, we subtract the number from the "zero point," in this case, 1000.

$$1000 - 0110 = 0010$$

Our number is, of course, negative, and we can now read it normally.

$$0\ 1\ 0$$

$$\uparrow\ \uparrow\ \uparrow$$

$$4\ 2\ 1$$

$$0 + 2 + 0 = 2$$

We find that 0110 in excess 8 is equal to -2 in decimal.

16) Convert the binary number **11010100**, represented in **excess 128**, to its decimal value.

Let's look at the sign bit first.

1**1**010100

Since it is a positive number, we can read the remaining bits normally.

64 16 8 4 1
↓ ↓↓↓ ↓
0 1 0 1 0 1 0 0
↑ ↑ ↑
128 32 2

64 + 16 + 4 = 84

So, our decimal representation of 11010100 is 84.

17) Convert **7** to its **excess 8** representation.

We know the number will have a sign bit of 1, as it is positive. The remaining 3 bits will be used to get the binary representation as normal.

$$7 / 2 = 3 \text{ R } \mathbf{1}$$

$$3 / 2 = 1 \text{ R } \mathbf{1}$$

$$1 / 2 = 0 \text{ R } \mathbf{1}$$

$$111$$

Now we simply add the sign bit to the beginning of the number, and we have our excess 8 representation.

$$\mathbf{1}111$$

Thus, our excess 8 representation of the decimal number 7 is 1111.

18) Convert **-83** to its **excess 128** representation.

This number is negative, so the sign bit will be a 0. We can add our number to 128 to get a number easily convertible to excess notation.

$$-83 + 128 = 45$$

Now we can convert this to binary how we have before.

$$45 / 2 = 22 \text{ R } \mathbf{1}$$

$$22 / 2 = 11 \text{ R } \mathbf{0}$$

$$11 / 2 = 5 \text{ R } \mathbf{1}$$

$$5 / 2 = 2 \text{ R } \mathbf{1}$$

$$2 / 2 = 1 \text{ R } \mathbf{0}$$

$$1 / 2 = 0 \text{ R } \mathbf{1}$$

$$101101$$

We need to use 7 bits, so a zero can be placed at the beginning of this, then the sign bit at the beginning of that.

$$101101 \rightarrow 0101101 \rightarrow 00101101$$

This means that our decimal value of -83 is represented as 00101101 in excess 128.

19) Convert the binary number **10100110**, represented in **floating-point**, to its decimal value.

First, we can look at the sign bit. A sign of 1 means the number will be negative. Next, we can determine the exponent as represented in excess 4 notation.

$$010 \rightarrow \text{negative number}$$

$$100 - 010 = 10$$

$$1\ 0$$

$$\uparrow\ \uparrow$$

$$2\ 1$$

$$2 + 0 = 2$$

Now, we can calculate the mantissa.

1 is automatically assumed to be in front of the four bits of the mantissa.

$$0110 \rightarrow (1.)0110$$

$$\begin{array}{c} .5\ \ .125 \\ \downarrow\ \ \downarrow \\ 1\ 0\ 1\ 1\ 0 \\ \uparrow\ \ \uparrow\ \ \uparrow \\ 1\ \ \ .25\ .0625 \end{array}$$

$$1 + .25 + .125 = 1.375$$

Finally, we can put all of this together with the floating-point equation.

$$-1^{1} * 2^{-2} * 1.375 = -0.34375$$

Thus, the decimal value of the floating-point value 10100110 is -0.34375.

20) Convert the binary number **01111001**, represented in **floating-point**, to its decimal value.

First, we can see that the sign is 0, meaning it will be a positive number. Next, we can determine the decimal value of the exponent.

$$111 \rightarrow \text{positive}$$

$$\begin{matrix} 1\ 1 \\ \uparrow\ \uparrow \\ 2\ 1 \end{matrix}$$

$$2 + 1 = 3$$

$$\text{Exponent} = 3$$

Now, we find the mantissa.

$$\begin{matrix} .5 & .125 \\ \downarrow & \downarrow \\ 1\ 1\ 0\ 0\ 1 \\ \uparrow & \uparrow & \uparrow \\ 1 & .25 & .0625 \end{matrix}$$

$$1 + .5 + 0 + 0 + .0625 = \mathbf{1.5625}$$

Lastly, we use the floating-point equation.

$$-1^{0} * 2^{3} * \mathbf{1.5625} = 12.5$$

So, we find that the floating-point-encoded binary number 01111001 represents the decimal value 12.5.

21) Convert the decimal value **1.75** to its **floating-point** binary representation.

First, we convert the number to its binary counterpart. We can use fractions to accomplish this.

$$1.75 \rightarrow 1.11$$

.5 .125
↓ ↓
1 1 1 0 0
↑ ↑ ↑
1 .25 .0625

Now, we can create our mantissa. Remember, we have four bits to represent this, and the whole-number and radix point we will just assume are there, we don't need to waste bits to hold this.

$$1.11 \rightarrow (1.)1100$$

Since this number does not need to be multiplied at all, our exponent will be 0. Remember that we find this exponent in excess 4 notation, so our zero-point is simple to find.

$$0 \rightarrow 100$$

Now that we have our exponent and mantissa, all we need is our sign bit. We already know that the number is positive, so a 0 can be used, and then our final number can be put together.

01001100

So, to represent the decimal number 1.75 in floating-point-encoded binary, we use the number 01001100.

22) Encode the number **-12.1875** in **floating-point** binary.

Convert the number to a binary fraction.

$$12.1875$$

$$12 \rightarrow 1100$$

$$0.1875 \rightarrow 0.0011$$

$$1100.0011$$

Now we'll move the radix to sit between the first two bits.

$$1.1000011$$

We can now remove the first bit as it is implied by our encoding. We then truncate the remaining bits down to the first 4 only.

$$1.1000011 \rightarrow 1000011$$

$$1000011 \rightarrow 1000$$

Now we move the radix to find the exponent.

$$.1000 \rightarrow 100.0$$

The radix moved 3 positions to the right. This means our exponent must be 3. Convert this to excess-4 notation.

Positive numbers → sign bit **1**

Make the remaining bits fit **3**.

3 = 11

111

Now finish by determining the sign bit. Negative numbers will have a sign bit 1.

11111000

Since we had to truncate our mantissa, this won't be exactly the same as our original value. Let's convert it back to see how closely it matches.

$$1.1000 \rightarrow 1.5$$

$$111 \rightarrow 3$$

$$(-1)^1 * 2^3 * 1.5 = -12$$

Our answer is -12. This is certainly close to our original value but has completely lost its fractional value. It is important to always keep in mind both the upsides and downsides of floating-point notation.