Representation of Numbers

This is an substantially updated version of an older file that can be found here.

The sections below labeled "***interactive***" contain modules that can let you explore scenarios of your own choosing.

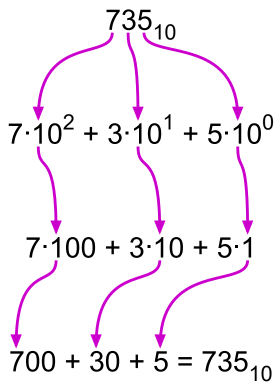
Contents

When working with any kind of digital system (electronics or computers), it is important to understand the different ways in which these numbers are represented. Almost without exception, numbers are represented by two voltage levels which can represent a one or a zero (an interesting exception to this rule are newer memory devices that use one four (or more) possible voltage levels, thereby increasing the amount of information that can be stored by a single memory cell). The number system based on ones and zeroes is called the *bi*nary system (because there are only two possible digits). Before discussing the binary system, a review of the *dec*imal (ten possible digits) system is in order because many of the concepts of the binary system will be easier to understand when introduced alongside their decimal counterpart. The "base" of the system is also called the "radix".

Finding the Decimal Equivalent of the Number with a Different Radix (*e.g.*, binary→decimal)

Positive Decimal Integers

You are familiar with the decimal system. For instance, to represent the positive integer one hundred and twenty-five as a decimal number, we can write (with the postivie sign implied). The subscript 10 denotes the number as a base 10 (decimal) number. After the first line, all numbers are implicitly base 10.



735_{10}
$7\cdot10^2 + 3\cdot10^1 + 5\cdot10^0$
$7\cdot100 + 3\cdot10 + 5\cdot1$
$700 + 30 + 5 = 735_{10}$

Some things to note (that we will be able to apply to the representation of unsigned binary numbers):
Some observations:

- The rightmost digit (5 in this case) is multiplied by 10^0, the next digit (3 in this case) to the left is multiplied by 10^1, and so on. Each successive digit to the left has a multiplier that is 10 times the previous digit.
- With n digits, 10^n unique numbers (from 0 to 10^n-1) can be represented. If n=3, 1000 (=10^3) numbers can be represented 0-999.

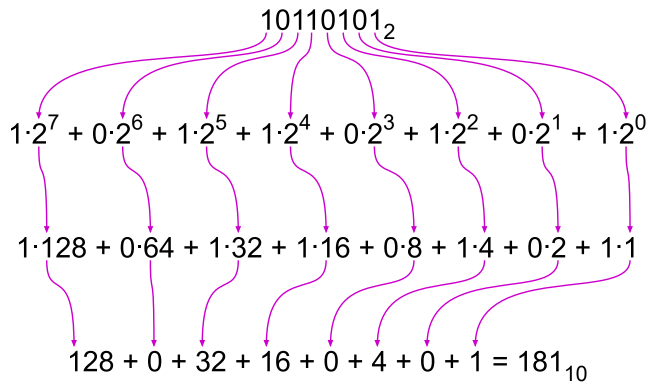
- To see how many digits a number needs, you can simply take the logarithm (base 10) of the absolute value of the number, and add 1 to it. The integer part of the result is the number of digits. For instance, $\log_{10}(33) + 1 = 2.5$; the integer part of that is 2, so 2 digits are needed.
  For this example $\log_{10}(725) + 1 = 3.86$; The integer part of that is 3, so 3 digits are needed.
- To multiply a number by 10 you can simply shift it to the left by one digit, and fill in the rightmost digit with a 0 (moving the decimal place one to the right). To divide a number by 10, simply shift the number to the right by one digit (moving the decimal place one to the left).
- Negative numbers are handled easily by simply putting a minus sign (−) in front of the number. This does lead, however, to the somewhat awkward situation where 0=−0. We will avoid this situation with binary representations, but with a little bit of effort.

Finding the decimal equivalent of an unsigned (positive) binary integer (*interactive*)

To represent a number in binary, every digit has to be either 0 or 1 (as opposed to decimal numbers where a digit can take any value from 0 to 9).The subscript 2 denotes a binary (i.e., base 2) number. Each digit in a binary number is called a bit.. Likewise we can make a similar set of observations:

To see how the decimal equivalent of an 8 bit unsigned binary number can be calculated, enter an 8 bit unsigned binary number: 10110101. *(Input restrictions)*.

Any number can be broken down this way, by finding all of the powers of 2 that add up to the number in question; you can see this is exactly analagous to the decimal deconstruction done earlier


10110101_2
$1\cdot2^7 + 0\cdot2^6 + 1\cdot2^5 + 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0$
$1\cdot128 + 0\cdot64 + 1\cdot32 + 1\cdot16 + 0\cdot8 + 1\cdot4 + 0\cdot2 + 1\cdot1$
$128 + 0 + 32 + 16 + 0 + 4 + 0 + 1 = 181_{10}$

- The rightmost digit is multiplied by 2^0, the next digit to the left is multiplied by 2^1, and so on. Each successive digit to the left has a multiplier that is 2 times the previous digit.
- With n digits, 2^n unique numbers (from 0 to 2^n-1) can be represented. If n=8, 256 ($=20^8$) numbers can be represented 0−255.
- To see how many digits a number needs, you can simply take the logarithm (base 2) of the absolute value of the number, and add 1 to it. The integer part of the result is the number of digits. For instance, $\log_{10}(53) + 1 = 6.7$; the integer part of that is 6, so 6 digits are needed. *Recall that $\log_2(x)=\log(x)/\log(2)$ (where log() can be either the natural log, or log base 10).*
- To multiply a number by 2 you can simply shift it to the left by one digit, and fill in the rightmost digit with a 0 (moving the decimal place one to the right). To divide a number by 2, simply shift the number to the right by one digit (moving the decimal place one to the left).
- At this point we can't represent negative numbers, but will do so soon.

Try this:

- Convert 0001 0011 *(spaces are ignored but can help make the number easier to read, much like commas in large decimal numbers)* from binary to decimal (you can check yourself by entering it above). Make sure you understand the result.
- Enter 0000 0011. Then 0000 0110, then 0000 1100. *Note that as the binary number is shifted to the left by one, the value of the number doubles.*
- Enter 1111 1111. As expected the result is $255=2^8-1$.
- Think of a number, and try to find the binary representation. Later in this document we will find a systematic way to do this.

Hexadecimal, Octal, Bits, Bytes and Words (*interactive*).

It is often convenient to handle groups of bits, rather than dealing with theindividually. The most common grouping is 8 bits, which forms a byte. A single byte can represent 256 (2^8) numbers. Memory capacity is usually referred to in bytes. Two bytes is usually called a word, or short word (though word-length depends on the application). A two-byte word is also the size that is usually used to represent integers in programming languages. A long word is usually twice as long as a word. A less common unit is the nibble which is 4 bits, or half of a byte.

It is cumbersome for humans to deal with writing, reading and remembering the large number bits, and it takes many of them to represent even fairly small numbers. A number of different ways have been developed to make the handling of binary data easier for us. The most common is to use hexadecimal notation. In hexadecimal notation, 4 bits (a nibble) are represented by a single digit. There is obviously a problem with this since 4 bits gives 16 possible combinations, and there are only 10 unique decimal digits, 0 to 9. This is solved by using the first 6 letters (A..F or a..f) of the alphabet as numbers. The table shows the relationship between decimal, hexadecimal and binary.

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

There are some significant advantages to using hexadecimal when dealing with electronic representations of numbers (if people had 16 fingers, we wouldn't be saddled with the awkward decimal system). Using hexadecimal makes it very easy to convert back and forth from binary because each hexadecimal digit corresponds to exactly 4 bits ($\log_2(16) = 4$) and each byte is two hexadecimal digit. In contrast, a decimal digit corresponds to $\log_2(10) = 3.322$ bits and a byte is 2.408 decimal digits. Clearly hexadecimal is better suited to the task of representing binary numbers than is decimal.

As an example, the 16 bit number $0CA3_{16} = 0000\ 1100\ 1010\ 0011_2$ ($0000_2 = 0_{16}$, $1100_2 = C_{16}$, $1010_2 = A_{16}$, $0011_2 = 3_{16}$). It is convenient to write the binary number with spaces after every fourth bit to make it easier to read.

Try it yourself: (enter either a value into either the binary or hexadecimal (i.e., hex) text box)
Binary value 0000 1100 1010 0011 Hexadecimal value: 0CA3. *(Input Restrictions)*

Converting a hexadecimal number to its decimal equivalent is slightly more difficult, but can be done in the same way as before but multiplying each digit by the appropriate power of 16 (instead of powers of 2 (for binary) or powers of 10 (for decimal)).

$$\begin{tikzcd}
 & & & 0CA3_{16} \arrow[dlll] \arrow[dl] \arrow[dr] \arrow[drrr] & & & \\
0\cdot 16^3 \arrow[d] & + & C\cdot 16^2 \arrow[d] & + & A\cdot 16^1 \arrow[d] & + & 3\cdot 16^0 \arrow[d] \\
0\cdot 4096 \arrow[d] & + & 12\cdot 256 \arrow[d] & + & 10\cdot 16 \arrow[d] & + & 3\cdot 1 \arrow[d] \\
0 & + & 3072 & + & 160 & + & 3 = 3235_{10}
\end{tikzcd}$$

Octal notation is yet another compact method for writing binary numbers. There are 8 octal characters, 0...7. Obviously this can be represented by exactly 3 bits. Two octal digits can represent numbers up to 64, and three octal digits up to 512

Summary of binary types:

- **bit**: a single binary digit, either zero or one.
- **byte**: 8 bits, can represent positive numbers from 0 to 255.

- **hexadecimal**: A representation of 4 bits by a single digit 0..9,A..F (or 0..9,a..f). In this way a byte can be represented by two hexadecimal digits.
- **long word**: A long word is usually twice as long as a word
- **nibble** (or *nybble*): 4 bits, half of a byte.
- **octal**: A representation of 3 bits by a single digit 0..7. This is used much less commonly than it once was, but it is still seen in special circumstances
- **word**: Usually 16 bits, or two bytes. But a word can be almost any size, depending on the application being considered -- 32 and 64 bits are other common sizes.

Exercises (binary types):

Convert 3C from hexadecimal to decimal	Answer
Convert 1010 0111 1011 from binary to hexadecimal	Answer
Convert 7D0 from hexadecimal to binary	Answer
If you shift a hexadecimal number to the left by one digit, how many times larger is the resulting number?	Answer

Converting a Decimal Number to a Different Radix (*e.g.*, decimal→binary);

Positive Decimal Integers

To begin our discussion on converting decimal numbers to a different radix, let's begin with a discussion of how we could find the individual digits of a decimal number, say 735_{10}. Obviously we can just look at the number in this case (the 1's place is 5, the 10's place is thirty and the 100's place is 7), but let's find an *algorithmic* way of doing this, so a computer can do it. It will also lead us to a technique for converting to binary (or any other radix).

Find the decimal digits that comprise the number 735_{10}.

- We begin by performing an integer division by 10 (recall that decimal is radix 10).
  735 ÷ 10 = 73 r 5 (this is read as 735 divided by 10 is 73 with a remainer of 5; the quotient is 73 and the remainder is 5). This tells us that the rightmost digit (i.e., the remainder) is 5.
- Repeat process with quotient: 73 ÷ 10 = 7 r 3. So the next digit is 3.
- Repeat process with quotient: 7 ÷ 10 = 0 r 7. So the last digit is 7.
- Since the new quotient is 0, we are done.

- The three digits found are (left→right) 7, 3, and 5, which is also the list of remainders bottom→top.

Finding the unsigned binary equivalent of a positive decimal integer (*interactive*)

We can now perform conversions from decimal to binary using the same procedure but dividing the number by 2 each time, until the quotient of the division is zero.
Enter a decimal number between 0 and 255: 181 . *(Input Restrictions)*

The procedure is described verbally on the left, and is shown in a more compact tabular form on the right.

Find the binary digits that comprise the number 181_{10}.

- We begin by performing an integer division by 2 (recall that binary is radix 2).
  181 ÷ 2 = 90 r 1. This tells us the righmost bit (i.e., the remainder) is 1.
- Repeat the process with quotient: 90 ÷ 2 = 45 r 0. So the next digit is 0
- Repeat the process with quotient: 45 ÷ 2 = 22 r 1. So the next digit is 1
- Repeat the process with quotient: 22 ÷ 2 = 11 r 0. So the next digit is 0
- Repeat the process with quotient: 11 ÷ 2 = 5 r 1. So the next digit is 1
- Repeat the process with quotient: 5 ÷ 2 = 2 r 1. So the next digit is 1
- Repeat the process with quotient: 2 ÷ 2 = 1 r 0. So the next digit is 0
- Repeat the process with quotient: 1 ÷ 2 = 0 r 1. So the last (i.e., leftmost) digit is 1
- Since new the quotient is 0, we are done.

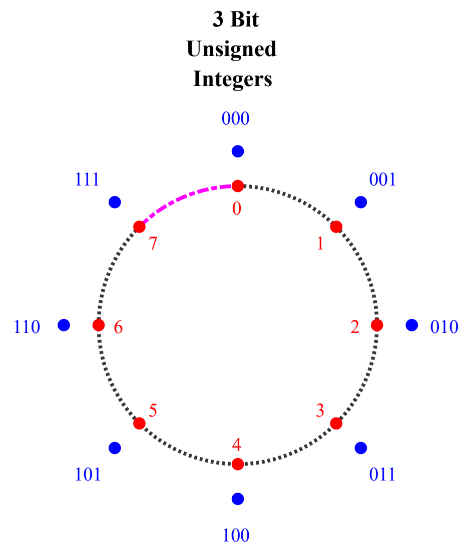
- We have completed the conversion from decimal to binary: $181_{10} = 10110101_2$.
  You can get the bits, left→right, by taking the remainders bottom→top.

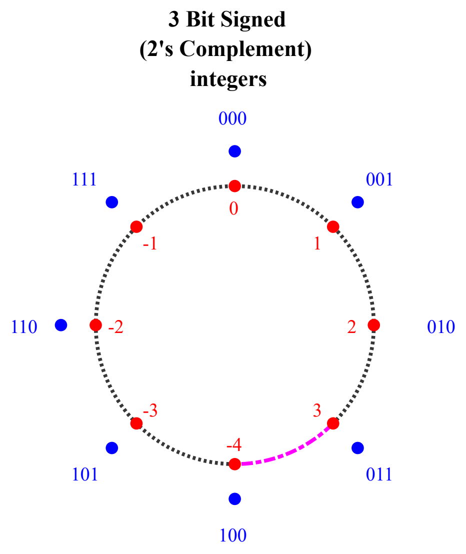
number	÷ 2	
	quotient	remainder
181	90	1
90	45	0
45	22	1
22	11	0
11	5	1
5	2	1
2	1	0
1	0	1

Signed Integers (i.e., the 2's complement representation)

Represented both positive and negative numbers

To represent signed integers we will use what is called the 2's complement representation. Let's start by representing 3 bit unsigned integers on a circle, as shown on the left half of the diagram shown below. This is the method we've been using up until now; only positive numbers (and 0) can be represented.





The image on the left shows the decimal number in red, and the equivalent binary number in blue. For example, the number 2_{dec} (on the right side of the circle) is the same as 010_2. Note:

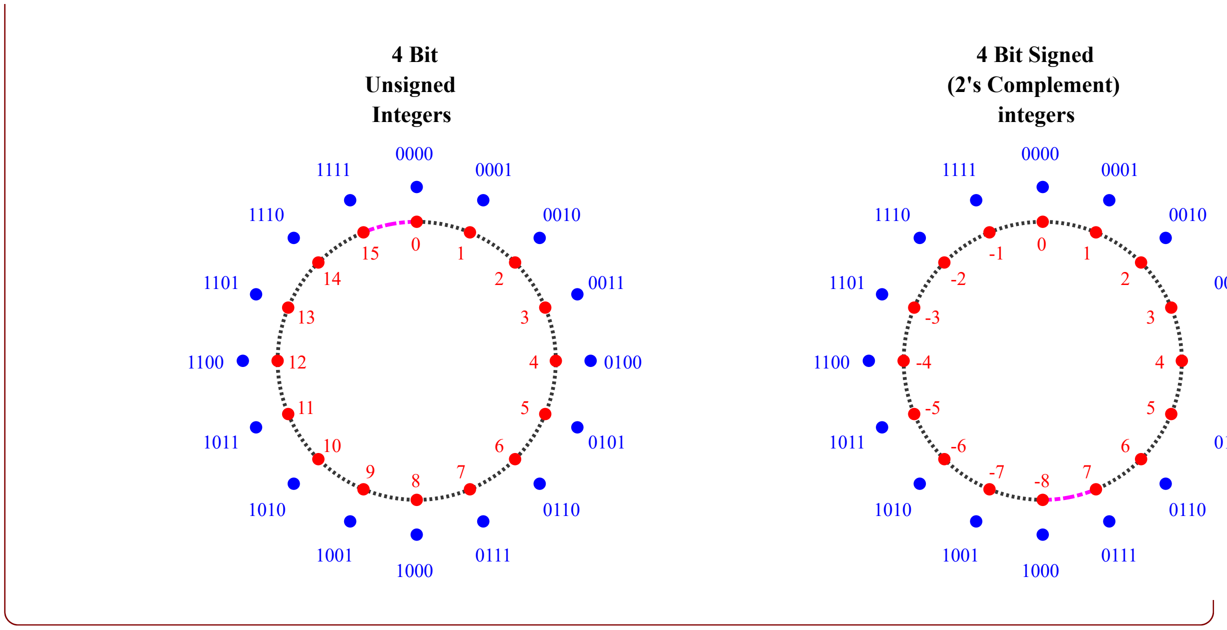
- Since there are n=3 bits, there are $2^n=2^3=8$ unique numbers that can be represented.
- The decimal numbers go from 0 to $2^n-1=2^3-1=7$ (i.e., 0→7).
- To add, we move clockwise around the circle (i.e., to go from $2_{dec}=010_2$ to $3_{dec}=011_2$, we move 1 place clockwise; to go from $2_{dec}=010_2$ to $5_{dec}=110_2$, we move 3 places). To subtract, we move counterclockwise.
- We get an error in addition or subtraction if we cross the part of the circle with the dashed magenta line (between 7_{dec} and 0_{dec}). For example if we are a $1_{dec}=001_2$, and subtract 3, then we move 3 spaces counterclockwise and end up at $6_{dec}=110_2$, which is clearly incorrect because we crossed the dashed magenta line.

To represent negative numbers in 2's complement notation, we refer to the right half of the diagram. We can make similar observations:

- Since there are n=3 bits, there are $2^n=2^3=8$ unique numbers that can be represented.
- The decimal numbers go from $-(2^{n-1})=-(2^2)=-4$ to $2^{n-1}-1=2^2-1=3$ (i.e., −4→3).
- To add, we move clockwise (i.e., to go from $2_{dec}=010_2$ to $3_{dec}=011_2$, we move 1 place clockwise; to go from $-2_{dec}=110_2$ to $1_{dec}=001_2$, we move 3 places). To subtract, we move counterclockwise.
- We get an error in addition or subtraction if we cross the part of the circle with the dashed magenta line (between 3_{dec} and -4_{dec}). For example if we are a $2_{dec}=010_2$, and add 3, then we move 3 spaces clockwise and end up at $-3_{dec}=101_2$, which is clearly incorrect.

Note that the binary numbers are identical, it is solely the way that we interpret them that changes. The substantive difference between the two representations is that 2's complement numbers can be either positive or negative as indicated by the leftmost bit. Note that for all of the positive numbers (and 0) the leftmost bit is a zero, and for all of the negative numbers the leftmost bit is a one.

Aside: the diagram below shows the circular representation for 4 bit numbers. To test you knowledge, make sure that you can make equivalent assertions to those listed above for 3 bits (which can represent the numbers −4→3) for the case when there are 4 bits (−8→7).



Finding the decimal equivalent of a 2's complement binary integer (*interactive*)

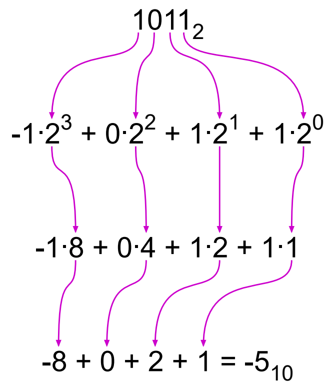
The process for finding the decimal equivalent of a 2's complement binary integer is very similar to the way we do it for unsigned numbers (see above), but we must interpret the leftmost bit differently. For an n bit number, if the leftmost bit is 1 we interpret it as a value of $-(2^{n-1})$, if it is 0 we ignore it. All the other bits are interpreted as before.

4 bit 2's complement:

For a four bit number $-(2^{n-1})=-(2^{4-1})=-(2^{3-1})=-8$.

To check your knowledge of signed numbers, explore 4 bit 2's complement numbers. Enter a 4 bit 2's complement number:
1011 . *(Input restrictions)*.

You can also try entering a 4 bit number here, and then covert it to 8 bits via sign extension, and enter it above.
If the 4 bit binary number $1011_2=-5_{dec}$, how do you represent -5_{dec} with 8 bits?


1011_2
$-1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 1\cdot2^0$
$-1\cdot8 + 0\cdot4 + 1\cdot2 + 1\cdot1$
$-8 + 0 + 2 + 1 = -5_{10}$

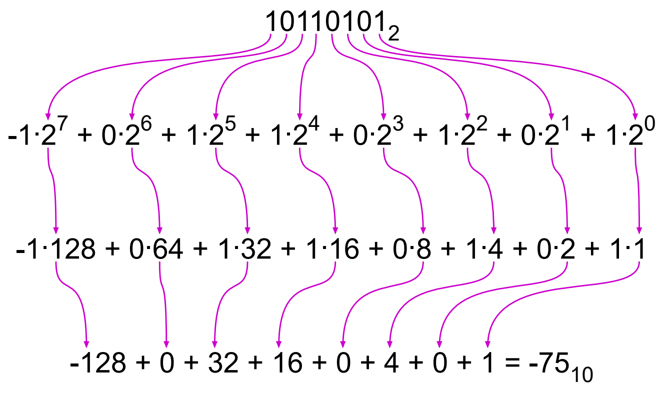
Try these things: *You may want to check the 4 bit number wheel shown previously.*

- Enter the number "0011", and the result is 3. Change the leftmost bit to "1" (so the number is "1011"), and the result is −5 (or −8+3; the −8 is from the (leftmost) sign bit, and the number 3 is from the rightmost 3 bits, or "011").
- Convert the 2's complement binary number "0111" to decimal. Predict, then check the result if the number is changed to "1111". Where would these two numbers be on the 4 bit 2's complement number wheel?
  Note if any 2's complement binary number is all 1's (i.e., "1111" for four bits, or "1111 1111" for 8 bits) it is always equivalent to the decimal number −1. If you are not sure why, check the number wheels shown previously.
- Convert (by hand) the number "1101", then check your result Hint (hover the mouse here for an explanation).
  Where would this number be on the number wheel? If you are not sure, check the 4 bit number wheel shown previously.

8 bit 2's complement

For an 8 bit number this means that if the leftmost bit is set, we interpret it as −128 (=$-(2^{8-1})=-(2^7)$).

To see how the decimal equivalent of an 8 bit 2's complement binary number can be calculated, enter an 8 bit 2's complement number: 10110101. *(Input restrictions)*.



$$10110101_2$$

$$-1\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0$$

$$-1\cdot 128 + 0\cdot 64 + 1\cdot 32 + 1\cdot 16 + 0\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1$$

$$-128 + 0 + 32 + 16 + 0 + 4 + 0 + 1 = -75_{10}$$

This is almost exactly the way we broke down an unsigned number previously. ***The only difference*** is how we treat the leftmost bit. If it is a "1" we add in $-(2^{n-1})\cdot 1=-128\cdot 1=128$; if it is a 0 we add in $-(2^{n-1})\cdot 0=-128\cdot 0=0$. In other words, if the leftmost bit is "0" we proceed exactly as we did with an unsigned number.

Try these things: *(Note: I put a space in the middle of the binary numbers to make them easier to read - you can choose to add these (or not) to the input text box*)

- Enter the number "0011 0101", and the result is 53. Change the leftmost bit to "1" (so the number is "1011 0101"), and the result is −75 (or −128+53; the −128 is from the sign bit, and the number 53 is from the remainder of the bits).
- Convert the 2's complement binary number "0111 1111" to decimal. Predict, then check the result if the number is changed to "1111 1111".
  Where would these two numbers be on a number wheel?
  Note if any 2's complement binary number is all 1's (i.e., "1111" for four bits, or "1111 1111" for 8 bits) it is always equivalent to the decimal number −1. If you are not sure why, check the number wheels shown previously.
- Convert (by hand) the number "1000 0101", then check your result Hint (hover the mouse here for an explanation).
  Where would this number be on a number wheel?

Sign Extension (increasing the number of bits in a 2's complement number)

For an unsigned number, increasing the number of bits used to represent a number can be accomplished by simply adding 0's to the left side of a number. For example, the number 3_{dec} can be represented with 3 bits as 011_2, or as 4 bits as 0011_2.

011
0011
Add "0" to left

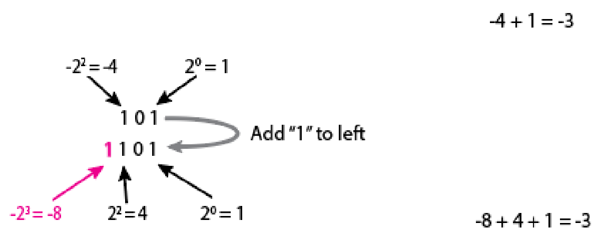
We can continue this process for 8 bits, or even more, and the number represented doesn't change.

0011
0000 0011
Add multiple "0's" to left

The process above works for positive 2's complement numbers, but not for negative numbers. Recall that for negative numbers that the leftmost bit is a "1." To increase the number of bits, we place an additional "1" to the left of the original number. For example, consider the 3 bit representation of the number $-3_{dec}=101_2$ *(the leftmost bit represents −4, and the rightmost bit represents 1, and when added together the result is 3)*. To increase to 4 bits we add a "1" to the left, which yields $1101_2=-3_{dec}$ *(see the discussion above for 4 bit 2's complement numbers if this is unclear)*. This is shown below, with the 3 bit version on the top line, and the 4 bit version below.

101
1101
Add "1" to left

This can be understood using the image below.



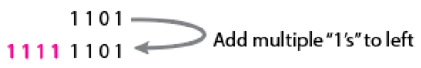
The top line (101) shows the original 3 bit number.

- The leftmost bit represents $-(2^{n-1})$ where n=3, or $-(2^{3-1})=-4$.
- The middle bit is zero and doesn't contribute
- The rightmost bit represents 1.
- The sum of the 3 bits is −3.

The second line (1101) show the 4 bit representation of the same number.

- The leftmost bit represents $-(2^{n-1})$ where n=4, or $-(2^{4-1})=-8$.
- The next bit represents 2^2 or 4.
- The third bit from the left is zero and doesn't contribute.
- The rightmost bit represents 1.
- The sum of the 4 bits is −3.

Note that the sum of the 2 leftmost bits in the 4 bit number is equal to −4, which is the same as the leftmost bit, by itself, in the three bit number — so adding the bit to the left has no effect on the final sum. This means we can keep repeating this process to find the 8 (or more) bit representation.



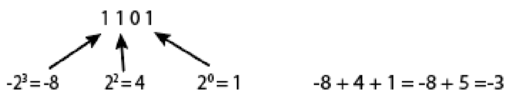
This process of increasing the number of bits used to represent is called "sign extension" because we simply extend the leftmost (sign) bit to fill in the added bits in the larger number. If the sign bit is "1", all of the new bits will be 1; if the sign bit is zero, all of the new bits will be 0. In the image below the sign bit of the 4 bit numbers is underlined, and you can see that it is simply repeated in the new bits of the 8 bit representation.

As a final exercise, you can look at the 3 bit and 4 bit number wheels shown above, and verify that you can convert 3 bit numbers to their 4 bit equivalent by using sign extension.

Convert from 2's complement binary to decimal (*interactive*).

Converting a 2's complement number from binary to decimal is very similar to converting an unsigned number, but we must account for the sign bit. If the number is positive (i.e., the sign bit is "0") the process is unchanged. If the number is negative, we know the sign bit is 1, and we just need to find the rest of the bits. An example may help to clarify things.

Consider the 4 bit 2's complement number, $1101_2=-3_{dec}$ (this is the same number used in the discussion of sign extension). The value of the bits that are set are shown below.



Since the number is negative we know that the sign bit (the leftmost bit) is set to "1". We just need to find the rest of the bits. We see that we can effectively remove the sign bit by adding +8 to the number ($+8=2^{n-1}=2^3$, where n is the number of bits in the number — in this case n=4 bits); this eliminates the effect of the sign bit. This process yields a result of $5_{dec}=101_2$. So the resulting 2's complement number is 1101_2, where the leftmost bit is the sign bit (i.e., −8), and the other bits represent the number 5 (so the total is −8+5=−3, as desired).

You can try this for yourself by typing in a number that can be represented as an 8 bit 2's complement number (i.e., since the number of bits is n=8, the number must be in the range from −128→+127 *(the lower limit is $-(2^{n-1})=-(2^7)=-128$, and the upper limit is $2^{(n-1)}-1=128-1=127$).*

Enter a number between −127 and 128, and you can see how it is converted to and 8 bit 2's complement number. -75 . *(Input Restrictions)*

The number is negative. To get the bits of the unsigned part we must first add 128 *(i.e., 2^{n-1}, where n=number of bits = 8)*. This gives us -75+128=53. We now perform the decimal to binary conversion on this number (53) using the same technique that was introduced earlier.

number	÷ 2	
	quotient	remainder
53	26	1
26	13	0
13	6	1
6	3	0
3	1	1
1	0	1

The unsigned part (53) is converted to binary as 0110101 *(Note: the rightmost (least significant) bit is the top remainder in the table, and the leftmost (most significant) bit is the bottom remainder in the table. If the result of the conversion is less then 7 bits (i.e., fewer than 7 rows in the table), it is padded to 7 bits by adding 0's to the left)*. Then the sign bit (in this case 1 because the number is negative). The final result is

$-75_{dec}=10110101_2$.

Erik Cheever
Professor Emeritus
Engineering Department
Swarthmore College