Introduction to Computer Science – Representing Information in the Computer

Dr Iain Styles

September 23, 2016

1 *Physical Constraints*

The modern computer is built from networks of electronic switches called *transistors*, and all of its operations are performed by opening and closing these switches in different combinations, connecting the physical wires representing signals in the computer either to a "ground" or "earth", or to a positive (relative to the ground) electrical potential ("voltage"). The impact of this is that data is represented by physical wires that are connected either either to ground or to the positive voltage. Thus, the wires can take one of two states which we might crudely describe as "on" and "off", or equivalently, "1" and "0". We are therefore led by necessity to adopt a **binary** representation. Attempts to construct machines that use other representations have not been widely adopted (although analogue computers that represented data as a continuum of electrical voltages were used for a short time before the invention of the transistor). We must therefore be able to represent any item of data that we may wish to manipulate with the computer in terms of binary digits – **bits**.

We will be focussing on the different ways in which one can represent **numerical data**, and we will see why it is important for the programmer to understand the different numerical datatypes. A poor choice of datatype can have serious consequences on the accuracy of a calculation so knowing how to choose a correct type is extremely important. Our journey through the types begins with the very simplest – the positive, or **unsigned integers**.

2 *Positive Numbers*

2.1 *Integers*

When we refer to positive/unsigned integers, we mean the so-called **natural numbers** – those that are used for counting: 0, 1, 2, 3 ..., 75, ..., 23523, ... Perhaps rather obviously, we do not include negative integers such as -1, -2, -1827 or non-integer real numbers such as 1.2, 3.14159, -2.7182818. In other words, numbers that we (or a rather more digitally blessed species) can count on our digits (fingers & toes).

Our usual representation (largely as a result of the number of fingers that we have) for these numbers is **decimal** or **radix-10** representation in which there are ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) available to represent integers, and we concatenate sequences

of symbols to represent larger values with each one representing a different power of ten. Here are some examples:

$$\begin{aligned} 3 &= 3 \times 10^0 \\ 24 &= 2 \times 10^1 + 4 \times 10^0 \\ 826 &= 8 \times 10^2 + 2 \times 10^1 + 6 \times 10^0 \end{aligned}$$

The binary representation of data works in a very similar way, but we only have two symbols (0, 1) available and have to represent larger values in terms of powers of two. Here are the same examples in binary:

$$\begin{aligned} 3_{10} &\mapsto 11_2 = 1 \times 2^1 + 1 \times 2^0 \\ 24_{10} &\mapsto 11000_2 = 1 \times 2^4 + 1 \times 2^3 \\ 826_{10} &\mapsto 1100111010_2 = 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 \end{aligned}$$

where we have used subscripts to indicate which radix we are using and avoid confusion (we will always use base 10 for this).

Any positive integer can be represented in binary in principle, but there are practical limitations. Modern computers break data up into 32- or 64-bit chunks (referred to as **words**), and this limits the maximum size of numbers that we can store. With n bits, we can store integers from 0 to $2^n - 1$. In the case of 32-bit machines, the upper limit is $2^{32} - 1 \approx$ 4billion whilst for 64-bit machines, the limit is $2^{64} - 1 \approx 10^{20}$. [1] This is adequate for most purposes, but if larger numbers are needed then there are some tricks that can be used in software to emulate larger word sizes. We will not study these tricks in this course.

[1] A useful trick to estimate this is to note that $2^{10} = 1024 \approx 10^3$. Therefore $2^{32} \approx 2^2 \times 2^{30} = 2^2 \times 2^{10} \times 2^{10} \times 2^{10} = 2^2 \times 10^3 \times 10^3 \times 10^3 = 4 \times 10^9$.

2.2 *Hexadecimal Notation*

If you have ever used an image manipulation software package and have tried to change the colour of something, you may have come across hexadecimal notation. It is quite commonly used in computing because it is much simpler for humans to manipulate than binary, where the strings are too long for us to deal with or recognise easily. Hexadecimal, or radix-16 representations are quite commonly used instead, and you may be familiar with these if you have ever used

Long binary strings are tedious for humans to manipulate and mistakes are easily made. For this reason it is common to represent then in another base. We could, of course, use a radix-10 representation but the conversion is tedious. It is much more common to use radix-16, as the conversion to/from binary is very simple.

Since hexadecimal is radix-16, we need 16 symbols to represent values: we use the familar ten decimal symbols, plus alphabetic characters A–F, as shown in the table.[2]

Thus the value $0x1C3F$ is the hexadecimal representation of (in decimal):

$$1 \times 16^3 + 12 \times 16^2 + 3 \times 16^1 + 15 \times 16^0 = 4096 + 3072 + 48 + 15 = 7231$$

[2] Hexadecimal symbols.

Dec	Hex	Bin
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

Since 16 is an integer power of 2, converting between binary and hexadecimal is then simply a matter of string substitution: each group of four bits can be converted directly to their hex equivalent and vice-versa:

$$0x1C3F \leftrightarrow \underbrace{0001}_{1}\ \underbrace{1100}_{C}\ \underbrace{0011}_{3}\ \underbrace{1111}_{F}$$

2.3 *Converting between binary and decimal*

The conversion from radix 10 to radix 2 is quite simple if you notice that odd numbers, when represented in radix 2, will always have a right-most digit of 1, as this is the only odd-valued power of 2. If we therefore divide our number by 2, the *remainder* (which must always be 0 or 1 for division by 2) immediately gives us the right-most, or **least significant bit** of the radix-2 representation. We can then apply this iteratively to compute the radix-2 representation. Here's an example:

$$\begin{aligned} 25/2 &= 12\text{r}1 \\ 12/2 &= 6\text{r}0 \\ 6/2 &= 3\text{r}0 \\ 3/2 &= 1\text{r}1 \\ 1/2 &= 0\text{r}1 \end{aligned}$$

Thus, we have that $25_{10} = 11001_2$.

The conversion in the opposite direction is most easily done by multiplying each digit by its corresponding power of two and adding, so for the current example:

$$11001_2 = 2^4 + 2^3 + 2^0 = 16_{10} + 8_{10} + 1_{10} = 25_{10}$$

2.4 *Positive Real Numbers*

It is not too difficult to extend the unsigned integers to the unsigned reals: 2.1, 3.14 and so on. The key to this is to understand what the radix-10 representation means. When we write 3.14_{10}, we really mean

$$3.14 = 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2}$$

where a negative index, e.g. 10^{-2} means $\frac{1}{10^2}$ and so the above equation translates to

$$\begin{aligned} 3.14 &= 3 \times 10^0 + 1 \times \frac{1}{10^1} + 4 \times \frac{1}{10^2} \\ &= 3 \times 1 + 1 \times 0.1 + 4 \times 0.01 \end{aligned}$$

The translation of this to radix-2 is direct and so we can easily

see that if we write an expression such as 101.001_2, we mean:

$$\begin{aligned} 101.001_2 &= 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-3} \\ &= 4 + 1 + \frac{1}{8} \\ &= 5\frac{1}{8} \end{aligned}$$

2.5 *Converting Real Numbers*

The process for converting reals into binary is similar to that for integers. Indeed, that part of the calculation for the integer part is identical! The non-integer part uses a similar method, but in reverse: instead of dividing by 2, we multiply the fractional part by 2, and take the integer part of the result. For example,

$$\begin{aligned} 0.142 \times 2 &= 0.284 \\ 0.284 \times 2 &= 0.568 \\ 0.568 \times 2 &= 1.136 \\ 0.136 \times 2 &= 0.272 \\ 0.272 \times 2 &= 0.544 \\ 0.544 \times 2 &= 1.088 \\ 0.088 \times 2 &= 0.176 \\ 0.176 \times 2 &= 0.352 \end{aligned}$$

At this point we could carry on, but computers have only a finite number of bits in which to store things (in this case, we have assumed eight bits). Our result, that $0.142_{10} \approx 0.00100100_2$. is therefore **not exact**[3]: for an exact representation, we should continue until the non-integer part of the result is exactly zero, but there is no guarantee that this limit will ever be reached.

[3] $0.00100100_2 = 2^{-3} + 2^{-6} = 0.125 + 0.00156 = 0.141$

Thus, given the decimal expansion of $\pi = 3.142$, we can write this in binary as 11.00100100, which requires ten bits of storage if we truncate the non-integer part at 8 bits. Most computers have 32- or 64-bit words (here we will use 16-bits to save space) and there is a choice as to where to put the radix point?[4], trading off the range of values that can be expressed against the accuracy of the non-integer part. In *fixed point* representations, the radix point is put at a known, fixed location. For example, we may allocate eight bits for the integer part and eight bits for the non-integer part, in which case we would write $3.142_{10} \approx 0000001100100100$, where we have added six leading zeros to make up the full sixteen bits, and removed the radix point as its location is now implicit. This fixed-point format is suitable for representation in the computer and is commonly used in small, low-cost microprocessors used in domestic appliances and other types of embedded system that do not support **floating point** numbers (section 5).

[4] In everyday language the radix point is referred to as the decimal point, but this would be rather inappropriate here!

3 *Simple Arithmetic*

So far, our discussion has been limited to positive numbers. Before we consider negative numbers, it is useful to consider how radix-2 arithemetic works.

3.1 Addition

Let us first remind ourselves how to do addition in radix-10 (decimal). Consider the addition problem $26 + 175$. Most people will be able to calculate that this is equal to 201 quite quickly using the intuition and experience we have developed over a lifetime of working with decimal numbers[5]. Most of us have no such intuition for binary arithmetic and have to resort to long-addition, long-multiplication etc. The long-addition version of our simple sum is as follows:

[5] For example, one may observe that $175 = 200 - 25$, and therefore $26 + 175 = 1 + 200 = 201$.

			2	6
	+	1	7	5
Carry		1	1	
Sum		2	0	1

When the sum of a particular column totals more than nine, part of the sum is carried over to the next column. In binary, we do exactly the same, except we carry at greater-than-one, not greater-than-nine. For example, $(6 + 5)_{10} = (110 + 101)_2$:

		1	1	0
+		1	0	1
Carry	1			
Sum	1	0	1	1

Let's now repeat our earlier example: $(26 + 175_{10} = (11010 + 10101111)_2$.

				1	1	0	1	0
+	1	0	1	0	1	1	1	1
Carry		1	1	1	1	1		
Sum	1	1	0	0	1	0	0	1

We can easily verify that this is correct.

Let us also check that it works with non-integers $(5.5 + 6.75)_{10} = (101.1 + 110.11)_{10}$. The vertical line indicate the position of the radix point:

		1	0	1	1	
+		1	1	0	1	1
Carry	1	1	1	1		
Sum	1	1	0	0	0	1

Again, it is easy to verify that this is correct.

3.2 *Subtraction*

In long subtraction, instead of carrying to the left, we have to borrow from the left. Here is a radix-10 example

	1	2	3	4	5
−		5	6	7	8
Borrow	-1	-1	-1	-1	
Diff		6	6	6	7

In binary, a similar principle applies and the calculation proceeds in much the same way.

	1	0	1	1
-		1	1	0
Borrow	-1			
Diff	0	1	0	1

Similarly for $(201 - 26)_{10}$ we have

	1	1	0	0	1	0	0	1
-				1	1	0	1	0
Borrow		-1	-1	-1	-1	-1		
Diff	1	0	1	0	1	1	1	1

By hand, this calculation is laborious, and some find it very awkward and error-prone. Of course, it is much easier in the computer, but it is not the same as addition and this has an important implication: it means that dedicated hardware will be required and this can be costly. However, there is an easier way which we will study in section 4. In order to understand how this arises, it is useful to examine what happens when numbers get too big for the computer.

3.3 *Overflow*

Consider the following apparently simple addition which is equivalent to $(3 + 1)_{10}$;

		1	1
		0	1
Carry	1	1	
Sum	1	0	0

The end result is indeed correctly equal to six, but there is a problem. The original numbers (operands) had two-bit representations, but their sum cannot be represented similarly. An extra bit is required and if it is not available, the most significant bit (MSB) is lost. If only two bits were available to perform this calculation, then the left-most 1 would be lost and the calculation would appear to be $(11 + 01 = 00)_2$, which is clearly not correct. Considering the four-bit analogue, we have a similar problem: $(1111 + 01 = [1]0000)_2$. In a fixed-bit representationa $111\ldots111 + 000\ldots001 = 000\ldots000$.

Now consider adding $111\ldots11$ to any numbers:

$$\begin{aligned} 1111 + 0010 &= [1]0001 \\ 1111 + 0011 &= [1]0010 \\ 1111 + 0101 &= [1]0100 \end{aligned}$$

The effect of adding $111\ldots111$ to any other number is to subtract one from the other number (when the number of bits available is fixed). This is actually a rather useful result: it leads us to a slightly less intuitive, but rather more useful radix-2 number representation: **two's complement**.

4 *Negative Numbers and Two's Complement*

One possible and rather obvious way of implementing negative numbers in binary is to simple treat one of the bits as a "sign" bit. Since the sign of a calculation cannot be determined until the whole calculation has been done, it makes sense to make this the left-most bit. This requires us to use an extra bit to represent numbers (or to lose range) but provides an intuitive representation of positive and negative numbers that is (easily?) human-readable. Unfortunately it is not very optimal for the computer. As we have seen in the previous section, addition and subtraction (never mind multiplication and division) require different "algorithms", and since addition of a negative number is subtraction, this is problematic. It means that hardware for both addition and subtraction has to be incorporated into the computer.

The observation we made about overflow provides a useful clue to how we may come up with an alternative representation. We saw that adding $111\ldots111$ to any number was equivalent to adding -1. Thus, we can use $111\ldots111$ to represent -1.

Now consider the following addition problem:

		0	1	0	1	1	0	1	0
		1	0	1	0	0	1	1	0
Carry		1	1	1	1	1	1		
Sum	(1)	0	0	0	0	0	0	0	0

The portion of the result that can fit within the available 8 bits is 00000000. As far as the computer is concerned, 01011010 and 10100110 are the negative of each other as they add to give 0 (ignoring the overflow). They are called each other's **two's complement**. This is a very convenient way of representing both positive and negative numbers and is by far the most common way of implementing them in the computer.

To form a number's twos complement:

1. Invert all bits.
2. Add 1 to the LSB.

If we treat positive numbers (and zero) to always begin with zero and be otherwise identical to unsigned numbers, it then follows that all negative numbers must begin with at least one leading one. Thus, the first few positive and negative integers are:

Radix-10	Radix 2
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101

This method works equally well for fixed-point non-integers. In 8-bits, assuming 4-bits of integer and 4-bits of non-integers, $\pi \approx (00110010)_2$ and its two's-complement is $(\pi \approx 11001110)_2$.

Let us do a sum using two's complement. First, consider $(7 - 15)_{10}$, which is equivalent to $(00111 - 01111)_2$. Note that we are using five bits in order to make sure we have sufficient range First, we convert $(01111)_2$ into it's two's complement form to get $(10001)_2$ which is the two's complement representation of -15_{10}. Now we add to get $(00111 + 10001 = 11000)_2$. Since the lead digit is 1, this must be negative, and we take its two's complement by inverting all bits to get 00111_2 and then adding 1 to get $01000_2 = 8_{10}$.

It is worth noting what the range of numbers represented in two's complement is. The largest positive value is $0111\ldots11$, and the smallest positive value is $000\ldots001$. For the negative numbers, the smallest ("least negative") is $111\ldots11$ whilst the largest ("most negative") is $100\ldots00$. Thus the range of values that can be represented in two's complement is from -2^{n-1} to $2^{n-1} - 1$.

5 *Floating Point*

We have seen how to represent both integers and real numbers (both positive and negative) in the computer using *fixed point* methods. However, these methods have some disadvantages. Notably, they impose considerable restrictions on the range of numbers that are available to use. Consider an eight-bit fixed-point representation in two's complement using four bits to store the integer, and four bits to store the non-integer. This structure is bound by the following limits:

	Two's complement	Radix 10
Largest positive value	0111.1111	7.9375
Smallest positive value	0000.0001	0.0625
Largest negative value	1000.0000	-8
Smallest negative value	1111.1111	-0.0625

We see that the range of values supported is rather small, and the precision available is also very limited: the small difference

between values is $0.0001_2 = 0.0625$. Fixed point data has limited range and limited precision which can be circumvented to some degree by allocating more bits, but there are practical limitations on this, notably lack of hardware support.

To support more accurate numerical calculation, **floating-point** number types have been devised which are much more flexible. We will consider a very simple floating point representation with two parts: a **mantissa** M (or argument, fraction, significand) and an **exponent** E which are combined to give a value V in the following way:

$$V = M \times 2^E$$

Each binary word is still subdivided as with fixed-point notation into two parts, but now they are the mantissa and the exponent. Note that these neither of these can be simply related to the integer and non-integer parts of fixed-point notation as both parts contribute to the integer and non-integer components. In 8-bits, allowing four bits of mantissa and four of exponent, and placing the radix point after the first bit of the mantissa we have (using two's complement for both exponent and mantissa):

		Floating Point	Radix 10
Largest positive value	01110111	0.111×2^{0111}	112
Smallest positive value	00011000	0.001×2^{1000}	4.9×10^{-4}
Largest negative value	10000111	1.000×2^{0111}	128
Smallest negative value	11111000	1.111×2^{1000}	-4.9×10^{-4}

There is now a greatly increased range of values available from the same number of bits. This comes at the cost of resolution: the use of an exponent means that the increments are non-linear, and hence larger numbers have a lower resolution than smaller numbers (as the differences get exponentially bigger). Note that this means that you should be extremely careful when you use floating point numbers: the Java documentation recommends that you should not use floating point for "for precise values, such as currency" for this reason.[6].

[6] http://docs.oracle.com/javase/tutorial/java/nutsandb

5.1 *Arithmetic*

Floating-point arithmetic is rather complex and the details are beyond the scope of these introductory lectures. Addition and subtraction of floating-point numbers are simple, provided that the exponents are the same. In this case, one can simple add/subtract the mantissas. If the exponents are not the same the bits of the mantissa need to be shifted in order to allow the exponent to be changed. A shift to the left doubles the mantissa and so the exponent will need to be decremented; and shift to the right halves the mantissa and the exponent will need to be incremented. This is done until the exponents of the two terms match. Normally, the mantissa of the

term with the smallest exponent is right-shifted and the exponent increased as right-shifting means that the least-significant bits are lost (instead of the most significant bits). However, chip designeers employ clever tricks to recognise what strategy is most appropriate (for examples, some mantissas can be left-shifted without loss of information). The design of high-precision floating-point arithmetic hardware is extremely complicated!

6 Numeric Datatypes in Java

The following table summarises the properties of the main numeric datatypes that you will encounter in Java:

Name	# bits	Representation	Min/Max
byte	8	Signed two's complement integer	$-128 \mapsto 127$
short	16	Signed two's complement integer	$-32,768 \mapsto 32,767$
int	32	Signed two's complement integer	$-2,147,483,648 \mapsto 2,147,483,647$
long	64	Signed two's complement integer	$-9,223,372,036,854,775,808 \mapsto 9,223,372,036,854,775,807$
float	32	Single precision floating point	See specification
double	64	Double precision floating point	See specification

7 Further Reading

[RT08, Ch3] gives basic coverage. My personal favourite source is a rather old book: [SACR91, Ch2] but this is very difficult to obtain, although second hand copies can be found on Amazon.

Bibliography

[RT08] Carl Reynolds and Paul Tymann. *Schaum's Outline of Principles of Computer Science*. McGraw-Hill, Inc., 2008.

[SACR91] Ian L. Sayers, Alan E. Adams, E. Graeme Chester, and Adrian P. Robson. *Principle's of Microprocessors*. CRC Press, 1991.