

Lecture 02

Representing Data

1 Introduction

Two fundamental types in almost any programming language are Booleans and integers. Booleans are comparatively straightforward: they have two possible values (true and false) and conditionals to test boolean values. Integers $\dots, -2, -1, 0, 1, 2, \dots$ are considerably more complex, because there are infinitely many of them. Because memory is finite, only a finite subrange of them can be represented in computers. In this lecture we discuss how integers are represented, how we can deal with the limited precision in the representation, and how various operations are defined on these representations.

2 Binary Representation of Natural Numbers

For the moment, we only consider the natural numbers $0, 1, 2, \dots$ and we do not yet consider the problems of limited precision. Number notations have a *base* b . To write down numbers in base b we need b distinct *digits*. Each digit is multiplied by an increasing power of b , starting with 2^0 at the right end. For example, in base 10 we have the ten digits 0–9 and the string 9380 represents the number $9 * 10^3 + 3 * 10^2 + 8 * 10^1 + 0 * 10^0$. We call numbers in base 10 *decimal numbers*. Unless it is clear from context that we are talking about a certain base, we use a subscript_[b] to indicate a number in base b .

In computer systems, two bases are of particular importance. *Binary numbers* use base 2, with digits 0 and 1, and *hexadecimal numbers* (explained more below) use base 16, with digits 0–9 and A–F. Binary numbers are so important because the basic digits, 0 and 1, can be modeled inside the computer by two different voltages, usually “off” for 0 and “on” for 1. To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of a bit sequence

$$b_{n-1} \dots b_1 b_0_{[2]} = b_{n-1}2^{n-1} + \dots + b_12^1 + b_02^0 = \sum_{i=0}^{n-1} b_i2^i$$

For example, $10011_{[2]}$ represents $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19$.

We can also calculate the value of a binary number in a nested way, exploiting Horner’s rule for evaluating polynomials.

$$10011_{[2]} = (((1 * 2 + 0) * 2 + 0) * 2 + 1) * 2 + 1 = 19$$

In general, if we have an n -bit number with bits $b_{n-1} \dots b_0$, we can calculate

$$(\dots((b_{n-1} * 2 + b_{n-2}) * 2 + b_{n-3}) * 2 + \dots + b_1) * 2 + b_0$$

For example, taking the binary number $10010110_{[2]}$ write the digits from most significant to least significant, calculating the cumulative value from left to right by writing it top to bottom.

$$\begin{array}{r} 1 = 1 \\ 1 * 2 + 0 = 2 \\ 2 * 2 + 0 = 4 \\ 4 * 2 + 1 = 9 \\ 9 * 2 + 0 = 18 \\ 18 * 2 + 1 = 37 \\ 37 * 2 + 1 = 75 \\ 75 * 2 + 0 = 150 \end{array}$$

Reversing this process allows us to convert a number into binary form. Here we start with the number and successively divide by two, calculating the remainder. At the end, the least significant bit is at the top.

For example, converting 198 to binary form would proceed as follows:

$$\begin{aligned}198 &= 99 * 2 + 0 \\99 &= 49 * 2 + 1 \\49 &= 24 * 2 + 1 \\24 &= 12 * 2 + 0 \\12 &= 6 * 2 + 0 \\6 &= 3 * 2 + 0 \\3 &= 1 * 2 + 1 \\1 &= 0 * 2 + 1\end{aligned}$$

We read off the answer, from bottom to top, arriving at $11000110_{[2]}$.

3 Modular Arithmetic

Within a computer, there is a natural size of words that can be processed by single instructions. In early computers, the word size was typically 8 bits; now it is 32 or 64. In programming languages that are relatively close to machine instructions like C or C0, this means that the native type `int` of integers is limited to the size of machine words. In C0, we decided that the values of type `int` occupy 32 bits.

This is very easy to deal with for small numbers, because the more significant digits can simply be 0. According to the formula that yields their number value, these bits do not contribute to the overall value. But we have to decide how to deal with large numbers, when operations such as addition or multiplication would yield numbers that are too big to fit into a fixed number of bits. One possibility would be to raise overflow exceptions. This is somewhat expensive (since the overflow condition must be explicitly detected), and has other negative consequences. For example, $(n+n) - n$ is no longer equal to $n + (n - n)$ because the former can overflow while the latter always yields n and does not overflow. Another possibility is to carry out arithmetic operations *modulo* the number of representable integers, which would be 2^{32} in the case of C0. We say that the machine implements *modular arithmetic*.

In higher-level languages, one would be more inclined to think of the type of `int` to be inhabited by integers of essentially unbounded size. This means that a value of this type would consist of a whole vector of machine words whose size may vary as computation proceeds. Basic operations such as addition no longer map directly onto machine instruction, but are

implemented by small programs. Whether this overhead is acceptable depends on the application.

Returning to modular arithmetic, the idea is that any operation is carried out modulo 2^p for the precision p . Even when the modulus is not a power of two, many of the usual laws of arithmetic continue to hold, which makes it possible to write programs confidently without having to worry, for example, about whether to write $x + (y + z)$ or $(x + y) + z$. We have the following properties of the abstract algebraic class of *rings* which are shared between ordinary integers and integers modulo a fixed number n .

Commutativity of addition	$x + y = y + x$
Associativity of addition	$(x + y) + z = x + (y + z)$
Additive unit	$x + 0 = x$
Additive inverse	$x + (-x) = 0$
Cancellation	$-(-x) = x$
Commutativity of multiplication	$x * y = y * x$
Associativity of multiplication	$(x * y) * z = x * (y * z)$
Multiplicative unit	$x * 1 = x$
Distributivity	$x * (y + z) = x * y + x * z$
Annihilation	$x * 0 = 0$

4 An Algorithm for Binary Addition

In the examples, we use arithmetic modulo 2^4 , with 4-bit numbers. Addition proceeds from right to left, adding binary digits modulo 2, and using a carry if the result is 2 or greater. For example,

$$\begin{array}{r}
 1\ 0\ 1\ 1 \\
 +\ 1\ 0_1\ 0_1\ 1 \\
 \hline
 (1)\ 0\ 1\ 0\ 0
 \end{array}
 \begin{array}{l}
 = 11 \\
 = 9 \\
 = 20 = 4 \pmod{16}
 \end{array}$$

where we used a subscript to indicate a carry from the right. The final carry, shown in parentheses, is ignored, yielding the answer of 4 which is correct modulo 16.

This grade-school algorithm is quite easy to implement (see Section 8) in software, but it is not suitable for a hardware implementation because

it is too sequential. On 32 bit numbers the algorithm would go through 32 stages, for an operation which, ideally, we should be able to perform in one machine cycle. Modern hardware accomplishes this by using an algorithm where more of the work can be done in parallel.

5 Two's Complement Representation

So far, we have concentrated on the representation of natural numbers $0, 1, 2, \dots$. In practice, of course, we would like to program with negative numbers. How do we define negative numbers? We define negative numbers as additive inverses: $-x$ is the number y such that $x + y = 0$. A crucial observation is that in modular arithmetic, additive inverses already exist! For example, $-1 = 15 \pmod{16}$ because $-1 + 16 = 15$. And $1 + 15 = 16 = 0 \pmod{16}$, so, indeed, 15 is the additive inverse of 1 modulo 16.

Similarly, $-2 = 14 \pmod{16}$, $-3 = 13 \pmod{16}$, etc. Writing out the equivalence classes of numbers modulo 16 together with their binary representation, we have

...	-16	0	16	...	0000
...	-15	1	17	...	0001
...	-14	2	18	...	0010
...	-13	3	19	...	0011
...	-12	4	20	...	0100
...	-11	5	21	...	0101
...	-10	6	22	...	0110
...	-9	7	23	...	0111
...	-8	8	24	...	1000
...	-7	9	25	...	1001
...	-6	10	26	...	1010
...	-5	11	27	...	1011
...	-4	12	28	...	1100
...	-3	13	29	...	1101
...	-2	14	30	...	1110
...	-1	15	31	...	1111

At this point we just have to decide which numbers we interpret as positive and which as negative. We would like to have an equal number of positive and negative numbers, where we include 0 among the positive ones. From this considerations we can see that $0, \dots, 7$ should be positive and

$-8, \dots, -1$ should be negative and that the highest bit of the 4-bit binary representation tells us if the number is positive or negative.

Just for verification, let's check that $7 + (-7) = 0 \pmod{16}$:

$$\begin{array}{r} 0 \ 1 \ 1 \ 1 \\ + \ 1_1 \ 0_1 \ 0_1 \ 1 \\ \hline (1) \ 0 \ 0 \ 0 \ 0 \end{array}$$

It is easy to see that we can obtain $-x$ from x on the bit representation by first complementing all the bits and then adding 1. In fact, the addition of x with its bitwise complement (written $\sim x$) always consists of all 1's, because in each position we have a 0 and a 1, and no carries at all. Adding one to the number $11 \dots 11$ will always result in $00 \dots 00$, with a final carry of 1 that is ignored.

These considerations also show that, regardless of the number of bits, -1 is always represented as a string of 1's.

In 4-bit numbers, the maximal positive number is 7 and the minimal negative number is -8 , thus spanning a range of $16 = 2^4$ numbers. In general, in a representation with p bits, the positive numbers go from 0 to $2^{p-1} - 1$ and the negative numbers from -2^{p-1} to -1 . It is remarkable that because of the origin of this representation in modular arithmetic, the "usual" bit-level algorithms for addition and multiplication can ignore that some numbers are interpreted as positive and others as negative and still yield the correct answer modulo 2^p .

However, for comparisons, division, and modulus operations the sign does matter. We discuss division below in Section 9. For comparisons, we just have to properly take into account the highest bit because, say, $-1 = 15 \pmod{16}$, but $-1 < 0$ and $0 < 15$.

6 Hexadecimal Notation

In C0, we use 32 bit integers. Writing these numbers out in decimal notation is certainly feasible, but sometimes awkward since the bit pattern of the representation is not easy to discern. Binary notation is rather expansive (using 32 bits for one number) and therefore difficult to work with. A good compromise is found in *hexadecimal notation*, which is a representation in base 16 with the sixteen digits 0–9 and A–F. "Hexadecimal" is often abbreviated as "hex". In the concrete syntax of C0 and C, hexadecimal numbers are preceded by 0x in order to distinguish them from decimal

numbers.

binary	hex	decimal
0000	0x0	0
0001	0x1	1
0010	0x2	2
0011	0x3	3
0100	0x4	4
0101	0x5	5
0110	0x6	6
0111	0x7	7
1000	0x8	8
1001	0x9	9
1010	0xA	10
1011	0xB	11
1100	0xC	12
1101	0xD	13
1110	0xE	14
1111	0xF	15

Hexadecimal notation is convenient because most common word sizes (8 bits, 16 bits, 32 bits, and 64 bits) are multiples of 4. For example, a 32 bit number can be represent by eight hexadecimal digits. We can even do a limited amount on arithmetic on them, once we get used to calculating modulo 16. Mostly, though, we use hexadecimal notation when we use bitwise operations rather than arithmetic operations.

Integer Division and Modulus

The division and modulus operators on integers are somewhat special. As a multiplicative inverse, division is not always defined, so we adopt a different definition. We write x/y for *integer division* of x by y and $x\%y$ for *integer modulus*. The two operations must satisfy the property

$$(x/y) * y + x\%y = x$$

so that $x\%y$ is like the remainder of division. The above is not yet sufficient to define the two operations. In addition we say $0 \leq |x\%y| < y$. Still, this leaves open the possibility that the modulus is positive or negative when y does not divide x . We fix this by stipulating that integer division truncates its result towards zero. This means that the modulus must be negative if x is negative and there is a remainder, and it must be positive if x is positive.

By contrast, the *quotient* operation always truncates down (towards $-\infty$), which means that the *remainder* is always positive. There are no primitive operators for quotient and remainder, but they can be implemented with the ones at hand.

An Algorithm for Binary Addition

In the examples, we use arithmetic modulo 2^4 , with 4-bit numbers. Addition proceeds from right to left, adding binary digits modulo 2, and using a carry if the result is 2 or greater. For example,

$$\begin{array}{rcccccc} & 1 & 0 & 1 & 1 & & = & 11 \\ + & 1 & 0_1 & 0_1 & 1 & & = & 9 \\ \hline (1) & 0 & 1 & 0 & 0 & & = & 20 = 4 \pmod{16} \end{array}$$

where we used a subscript to indicate a carry from the right. The final carry, shown in parentheses, is ignored, yielding the answer of 4 which is correct modulo 16.