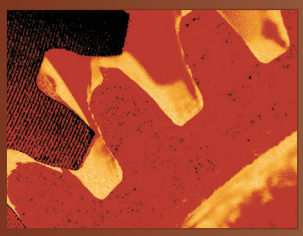
Binary Arithmetic

Neville Holmes, University of Tasmania



The basic aspects of computer arithmetic are how numbers are represented and the operations performed on those representations.

Digital technology works so well because at the heart of digital representation there are only a few basic components. Nowadays, these basic components usually are binary digits, *bits* for short, called binary because they are designed to stand for only two different values, conveniently called zero and one. A stored or transmitted bit's state can deteriorate quite badly before a processing device will be mistaken in deciding which of the two possible values is the original.

Most modern computers store the data uniformly in blocks in their main store. The blocks are numbered serially so that each block has its own number—its *address*. Thus, each block has two aspects: its address and its content. Addresses are a simple sequence of serial numbers. The content located at a specific address is the value of its bits, usually eight in number and called a *byte*, divided into two groups of four contiguous bits called *nibbles*.

As each bit can independently be either 0 or 1, a byte can hold 2^8 or 256 different values overall. There are too many of these to conveniently give a name to each, so bytes are usually considered as divided into nibbles of *hexadecimal digits—hex digits* for short. Table 1 lists binary digits and their hexadecimal equivalent.

For clarity, the hex digits beyond 9 are sometimes spoken as *able, baker, charlie, dog, easy,* and *fox*.

The value that any byte stores can then be shown as a pair of these hex digits. Strings of bytes can represent anything you need represented digitally. Being able to represent anything depends on having adopted a convention for representing things of that kind.

The earliest and still quite popular class of thing to be represented is numbers, though properly speaking these are not things but properties of things. The numerical operations that a computer carries out on numbers as numbers are together called its arithmetic.

ADDRESS ARITHMETIC

Working with a computer's data addresses requires only the simplest arithmetic, called address arithmetic or logical arithmetic. The arithmetic is simple because there is only a limited number of addresses and these are unsigned—that is, never negative—and start at zero.

If a computer used an address of one byte, it would have a main store of 256 bytes with addresses from 0 to 255 (decimal) or 00 to FF (hex). A 2-byte address would allow addresses from 0 to 65,535 or 0000 to FFFF, a 3-byte address 0 to 16,777,215 or 000000 to FFFFFF, and so on. Nowadays, computers usually have addresses larger than this. However, a one-byte address will serve here to illustrate logical arithmetic.

The main operation of logical arithmetic is addition. Addition is a series of steps starting with an augend and an addend. For simplicity, we use an 8-bit byte to illustrate the process, although 32-bit and 64-bit words are more common today.

Each step computes three new quantities: an augend, a carry, and an addend. The steps use all n bits of the operands in parallel (here $n = 8$). The new augend bit is 1 if and only if (denoted iff) one and only one of the two incoming bits is a 1 (that is, iff corresponding bits of the augend and addend are 01 or 10). The new carry bit is 1 iff both incoming bits are 1. The adder forms the new addend by shifting the carry bits one position to the left and putting a 0 in the right-most bit position. The adder repeats the three computations of a step (new augend, carry, and addend) until all the carry bits are 0. Figure 1 illustrates the example of adding 85 and 103 (decimal).

Quirks

The time taken for a straightforward addition depends on the number of steps needed, which can vary widely. However, shortcuts can remove this dependency.

When adding two numbers, the high order carry bit could be a 1. This is simply lost in shifting. The effect is like the hour-hand of a clock passing through 12, except that in the hexadecimal example the clock would have 256 hours labeled 00 to FF.

Circuitry could be provided for subtraction, but it's simpler to add the complement. For a hexadecimal clock, this hinges on 1 back, say, being the same as 255 forward, and vice versa.

Table 1. Hexadecimal digits.

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

For the binary representation of a byte, the complement is its bitwise complement with a one added in to the lowest position.

Logical arithmetic is error free. An address can be invalid, but only because the main store is not big enough to hold a value at every possible address, a fault that virtual addressing eliminates.

INTEGER ARITHMETIC

When possible values are quantitative and not indicative, the arithmetic must be able to handle negative numbers and to multiply and divide with them.

An early method of representing negativity was to use the leading bit as a simple arithmetic sign, but this raised the ambiguity of having two different zeroes, one negative and one positive.

The usual method nowadays is to offset the values in such a way that the arithmetic is not directly concerned with negativity, but the leading bit nevertheless acts as a negative sign. Thus the values for a single byte binary integer range from -128 (80) to -1 (FF) and from 0 (00) to 127 (7F). The four examples of addition in Figure 2 illustrate this, showing that the operation is practically the same as for logical arithmetic, even though the meanings have been changed.

Subtraction is carried out by adding the negation of the subtrahend. Multiplication is carried out by repeated addition, and division by repeated subtraction. Shortcuts are used to speed up these operations. Simple operations such as negation and magnitude are typically also provided in a computer's instruction set.

Step	Operand	Binary	Hex	Decimal
Add decimal 85 + 103				
	Augend	01010101	55	85
	Addend	01100111	67	103
	New augend bit is 1 iff incoming bits are 01 or 10			
1a	Augend	00110010	32	50
	Carry bit is 1 iff incoming bits are 11			
1b	Carry	01000101	45	69
	New addend is now the left-shifted carry with 0 fill			
1c	Addend	10001010	8A	138
	Repeat the three substeps until the carry is all zeroes			
2a	Augend	10111000	B8	184
2b	Carry	00000010	02	2
2c	Addend	00000100	04	4
	Repeat the three substeps until the carry is all zeroes			
3a	Augend	10111100	BC	188
3b	Carry	00000000	00	0
Finished because carry is all zeroes; answer is in the last augend				

Figure 1. Address arithmetic example. The arithmetic is simple because there is only a limited number of addresses and these are never negative.

Example	Binary	Hex	Decimal
(a) 85 + 39 = 124			
Augend	01010101	55	85
Addend	00100111	27	39
Result	01111100	7C	124
(b) 85+ (–39) = 46			
Augend	01010101	55	85
Addend	11011001	D9	–39
Result	00101110	2E	46
(c) (–85) + 39 = –46			
Augend	10101011	AB	–85
Addend	00100111	27	39
Result	11010010	D2	–46
(d) (–85) + (–39) = –124			
Augend	10101011	AB	–85
Addend	11011001	D9	–39
Result	10000100	84	–124

Figure 2. Integer addition examples. The values for a single byte binary integer range from -128 (80) to -1 (FF) and from 0 (00) to 127 (7F).

Quirks

Just as for address arithmetic, the results are exact if they will fit into the space provided for the result in the result register. However, because the values are no longer cyclic, the result of an addition can be longer than that space. If adding two positive numbers results in an apparently negative number, then the correct sum is too long for the result register to hold. This is called an *overflow* and must be signaled so that a program can deal with this exceptional result. An apparent nonnegative result from adding two negative numbers is also an overflow. Negation and magnitude of the lowest representable number will also cause an overflow, which will need to be

Sign	Exponent	Significand	Binary	Decimal
0	1000 0000	01000000. . . 0	$+1.01 \times 2^{128-127}$	$(1 + .25) \times 2^1 = 2.5$
1	1000 0001	11000000. . . 0	$-1.11 \times 2^{129-127}$	$-(1 + .5 + .25) \times 2^2 = -7$
0	0111 1111	00000000. . . 0	$+1 \times 2^{127-127}$	1

Figure 3. Examples of scaled values. The most significant bit is the sign of the number (0 indicates positive, and 1 indicates negative). The next eight bits are the exponent in base 2, expressed as an integer. The 8-bit exponent value is biased by +127, which makes the representable range –127 to +128. The significand is termed "normalized" because its arithmetic value is always less than 2 but not less than 1.

signaled. In the example using eight bits, –128 cannot be negated because +128 is larger than the largest positive number representable in eight bits (+127).

While a program can satisfactorily deal with an occasional overflow, simple multiplication would overflow far too often to be tolerated. The usual way to deal with this is to couple two result registers to provide a double-length product. It's then up to the program to use the two halves appropriately.

Integer division is more complicated still. First, while the quotient, as an integer, usually can be accommodated, the division will leave a remainder, so two result registers are needed. Second, the quotient can in effect overflow when the divisor is zero or when the lowest representable integer is divided by negative one.

SCALED ARITHMETIC

In scientific and engineering computing, multiplications and divisions are as frequent as additions and subtractions. Repeated multiplication, as in polynomial evaluation, is common. Such computation requires scaling.

Before electronic computers became available, scientists and engineers commonly used slide rules and logarithm tables for multiplication and division, and they did scaling mentally or with the help of pencil and paper. Although Konrad Zuse's early computers used automatic scaling, later machines, such as John von Neumann's IAS computer, did not. This forced the programmer to anticipate what scaling would be needed, although the IAS machine used 40-bit numbers to protect against the unexpected.

When the unexpected proved all too frequent, users built scaling into their programs but, because this was very slow, scientific computers soon did their scaling in hardware. Rather unfortunately, such arithmetic and the representation of scaled values came to be called *floating-point* arithmetic and numbers. The adjective *semilogarithmic* is sometimes used for clarity.

A floating-point number has three parts: the base b, the scale or exponent e, and the significand s. The value represented is $s \times b^e$. The exponent and the significand are variable in floating-point representations, but the base is fixed. Scaled values are printed out using a base of 10, so 45E6 represents 45,000,000 and 4.5E–6 represents 0.0000045. Internally, most computers use binary floating-point arithmetic and representation in which the base is 2.

IEEE Standard 754 for Binary Floating-Point Arithmetic specifies the format of floating-point numbers for both single-precision (32-bit) and double-precision (64-bit) representations. For simplicity, we consider only the single-precision format. The standard says that the most significant bit is the sign of the number (0 indicates positive, and 1 indicates negative). The next eight bits are the exponent in base 2, expressed as an integer. But because the number's true exponent must be allowed to be positive or negative, the 8-bit exponent value is biased by +127, which makes the representable range –127 to +128. The remaining 23 bits are the mantissa, also called the significand. Those 23 bits are used as fraction bits appended to an implied integer of 1, sometimes called the "hidden" bit. The significand is termed "normalized" because its arithmetic value is always less than 2 but not less than 1.

In floating-point multiplication the exponents are added and the significands multiplied. The exponent and significand of the result might need slight adjustment to bring the integer part for the hidden bit back to 1 before the leading 23 fraction bits of the product's significand, perhaps with rounding, are stored in the result. Division is handled in much the same way, but using subtraction on the exponents and full division (without remainder) on the significands.

Addition and subtraction are quite complex because the values of the exponents must be used to adjust the alignment of the two significands before the addition or subtraction can take place. For further explanation of floating-point arithmetic, see en.wikipedia.org/wiki/floating_point.

Quirks

The most significant quirk of scaled arithmetic is the loss of exactness. Basic laws of arithmetic no longer hold. For example,

$$(a + b) - a = a + (b - a) = b$$

in exact arithmetic but, because the significand of any result needs to be truncated or rounded in floating-point arithmetic before it is stored, the result of $(a + b) - a$ might not be the same as that of $a + (b - a)$. Such errors can accumulate significantly when a program carries out trillions of floating-point operations. Sophisticated use of interval arithmetic can avoid this problem, but this requires the type of rounding to be selectable so that it can preserve the interval properties.

Because the significand is normalized, there is no straightforward way to represent zero. A tweak is needed. Once this tweak is provided, the various possible results of division by zero need tweaks in turn to represent them. For example, different representations are needed for $0 \div 0$, $1 \div 0$ and $-1 \div 0$. The arithmetic's various

operations must be able to handle all these special values in combination with each other and with ordinary values. To handle these special cases, exponent values of all zeroes and all ones are reserved to signal special values such as 0, denormalized numbers, infinity, and not-a-number. Thus, for ordinary arithmetic the exponent actually only has a range of –126 to +127.

Floating-point arithmetic is not only still subject to overflow when a result becomes too large to represent, but a result also can be too small to represent, an exception called *underflow*.

COMPLETE ARITHMETIC

Traditional floating-point arithmetic tolerates the introduction of error, but the errors tend to accumulate in unpredictable ways. In the past, providing longer and longer representations lessened the error, but this is a losing battle as computers become faster and faster and problems larger and larger. The result of a large scientific computation might now need many trillions of floating-point operations.

Such computations typically include focused subsections traversing very large arrays of values to arrive at only a few results such as the sum of products. The truncation error within such subsections can be unpredictably large, but it can be eliminated by using a result register large enough to keep the complete result, which is exact. With computers as capacious as today's, that exact result can then be kept as is for intermediate results, pressing it into floating-point format only for final results.

Complete arithmetic was available more than 20 years ago as a special feature for an IBM mainframe computer, but was later removed from sale. Perhaps it was before its time, as implementation in a microprogram was relatively slow.

With today's integrated circuitry, complete arithmetic has become quite practical and has been satisfactorily implemented on special chips. Its widespread adoption is overdue as its support for a branch of computation called *validated numerics* is crucial.

Validated numerics can provide solutions to most scientific computations with certainty. Prominent among its techniques is a sophisticated application of interval arithmetic, arithmetic that works with paired values that specify the bounds within which an entity's value must lie. Control of rounding type ensures that values stay within their bounds. Mathematicians can design algorithms so that convergence of intervals is proof that the solution is completely valid. Complete arithmetic can greatly speed up convergence, and it can induce convergence that would not be possible with traditional floating-point arithmetic.

Quirks

While complete arithmetic can greatly reduce the incidence of overflows and underflows, it cannot completely eliminate them. Valid results can be too large or small to be represented even in a complete result register—for example, in the unlikely event of repeated exponentiation of extreme values. Such invalidity is not of practical concern, however, because it can be completely avoided in its most particular use—the ubiquitous scalar or dot product. A more significant problem is the inability to fit extreme complete results into standard floating-point formats.

Large arrays of complete results need large amounts of storage space and time to store and fetch, although using a variable-length format for external storage of complete results could greatly reduce both the space and time needed. Nevertheless, programs need to keep down the number of times they convert complete results to floating-point format.

An arithmetic and representation such as symmetric-level indexing, which compares to semilogarithmic arithmetic somewhat the way semilogarithmic compares to integer, can eliminate overflow and underflow. The drawback is the severe complexity of arithmetic operations.

Perhaps a more serious problem with binary arithmetic is the isolation of scaled arithmetic from integer arithmetic. This isolation means that programmers must make decisions about which to use when coding programs, and making such decisions is not always easy. Also, they might need to write several versions of a computational function for different representations of its arguments. An early Burroughs mainframe computer in which an integer represesentation would switch to scaled rather than signaling an overflow implemented such an arithmetic.

Further discussion of these possibilities can be found in "Composite Arithmetic" (*Computer*, Mar. 1997, pp. 65-73). Ulrich Kulisch's *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units* (Springer-Verlag, 2002) provides a description of complete and interval arithmetics and their implementation.

Neville Holmes *is an honorary research associate at the University of Tasmania's School of Computing. Contact him at neville.holmes@utas.edu.au.*

Computer welcomes your submissions to this bimonthly column. For additional information, or to suggest topics that you would like to see explained, contact column editor Alf Weaver at weaver@cs.virginia.edu.