Computer Arithmetic

Vojin G. Oklobdzija
Electrical and Computer Engineering Department
University of California Davis

As the ability to perform computation increased from the early days of computers and up to the present so was the knowledge how to utilize the hardware and software to perform computation. Digital computer arithmetic emerged from that period in two ways: one as an aspect of logic design and other as development of efficient algorithms to utilize the available hardware.

Given that numbers in a digital computer are represented as a string of zeroes and ones and that hardware can perform only relatively simple and primitive set of Boolean operations, all of the arithmetic operations performed are based on a hierarchy of operations that are built upon the very simple ones.

What distinguishes computer arithmetic is its intrinsic relation to technology and the ways things are designed and implemented in a digital computer. This comes from the fact that the value of a particular way to compute, or a particular algorithm, is directly evaluated from the actual speed with which this computation is performed. Therefore there is a very direct and strong relationship between the technology in which digital logic is implemented to compute and the way the computation is structured. This relationship is one of the guiding principles in development of the computer arithmetic.

The subject of Computer Arithmetic can be, for simpler treatment, divided into: Number Representation, Basic Arithmetic operations (such as: addition, multiplication and division) and Evaluation of Functions.

Number Representation

The only way to represent information in a digital computer is via a string of bits i.e. zeroes and ones. The number of bits being used depends of the length of the *computer word* which is a quantity of bits on which hardware is capable on operating (sometimes also a quantity that is brought to the CPU from memory in a single access). The first question is what relationship to use in establishing correspondence between those bits and a number. Second, we need to make sure that certain properties that exist in the corresponding number system are satisfied and that they directly correspond to the operations being performed in hardware over the taken string of bits.

This relationship is defined by the rule which associates one numerical value designated as X (in the text we will use capital *X* for the numerical value) with the corresponding bit string designated as *x*.

$$x = \{x_{n-1}, x_{n-2}, \ldots\ldots, x_0\}$$

where :

$$xi \in 0, 1$$

In this case the associated word (the string of bits) is n bits long.

When for every value X exist one and only one corresponding bit string x, we define the number system as *non-redundant*. If however, we could have more than one bit string x that represents the same value X, the number system is *redundant*.

Most commonly we are using numbers represented in a *weighted* number system where a numerical value is associated to the bit string x according to the equation:

$$x = \sum_{i=0}^{n-1} x_i \times w_i$$

where:

$$w_0 = 1$$

and

$$wi = wi - 1 \times ri - 1$$

The value r_i is an integer designated as *radix* and in a *non-redundant* number system it is an integer equal to the number of allowed values for x_i. In general x_i could consist of more than one bit. Numerical value associated with x is designated as *explicit value* of *x*.
In conventional number systems the radix r_i is the same positive integer for all the digit positions x_i and with the canonical set of digit values:

$$\Sigma i = \{0, 1, 2, 3.... ri - 1\} for (0 \le i \le n - 1)$$

An example of weighted number system with a mixed-radix would be representation of time in weeks, days, hours, minutes and seconds with a range for representing 100 weeks:

$$r = 10, 10, 7, 24, 60, 60$$

In digital computers the radixes encountered are: 2, 4, 10 and 16 with 2 being most commonly used one.

The digit set x_i can be *redundant* and *non-redundant*.
If the number of different values x_i can assume is $n_x \le r$ than we have *non-redundant* digit set. Otherwise if $n_x > r$ we have *redundant* digit set. Use of the *redundant* digit set has its advantages in efficient implementation of algorithms (multiplication and division in particular).

Other number representations of interest are: *nonweighted* number systems where the relative position of the digit does affect the weight, so that the appropriate interchange of any two digits will not change the value x. The best example of such number system is Residue Number System (RNS).

We also define *explicit value* x_e and *implicit value* X_i of a number represented by a bit-string x. The *implicit value* is the only value of the interest to the user while *explicit value* provide the most direct interpretation of the bit string x. Mapping of the of the *explicit value* to the *implicit value* is obtained by an arithmetic function which defines the number representation used. It is a task of the arithmetic designer to devise algorithms which result in the correct implicit value of the result for the operations on the operand digits representing the explicit values. In the other words the arithmetic algorithm needs to satisfy the *closure* property.

The relationship between the *implicit value* and the *explicit value* is best illustrated by the table taken from [1].

Table 1: The relationship between the *implicit value* and the *explicit value*

Implied Attributes: Radix Point, Negative Number Representation,Others	Expression for Implicit value X_i as a function of explicit value x_e	Numerical implicit value X_i (in decimal)
Integer, Magnitude	$X_i = x_e$	27
Integer, “Two’s Complement”	$X_i = - 2^5 + x_e$	-5
Integer, “One’s Complement”	$X_i = - (2^5 - 1) + x_e$	-4
Fraction, Magnitude	$X_i = - 2^{-5} x_e$	27/32
Fraction, “Two’s Complement”	$X_i = - 2^{-4} (2^{-5} + x_e)$	-5/16
Fraction, “One’s Complement”	$X_i = - 2^{-4} (2^{-5} + 1 + x_e)$	-4/16

Representation of Signed Integers

The two most common representations of signed integers are Sign and Magnitude (SM) representation and True and Complement (TC) representation. While SM representation might be easier to understand and convert to and from, it has it’s own problems. Therefore we will find TC representation to be more commonly used.

Sign and Magnitude Representation (SM)

In SM representation signed integer X_i is represented by sign bit x_s and magnitude x_m (x_s, x_m). Usually 0 represents positive sign (+) and 1 represents negative sign (-). The magnitude of the

number x_m can be represented in any way choosen for the representation of positive integers. Disadvantage of SM representation is that two representations of zero exist, positive and negative zero: $x_s=0$, $x_m=0$ and $x_s=1$, $x_m=0$.

True and Complement Representation (TC)

In TC representation there is no separate bit used to represent the sign. Mapping between the *explicit* and *implicit* value is defined as:

$$X_i = \begin{cases} x_e & x_e < \frac{C}{2} \\ x_e - C & x_e > \frac{C}{2} \end{cases}$$

The illustration of the TC mapping is given in Table 2.[2]

Table 2: True and Complement Mapping

x_e	X_i
0	0
1	1
2	2
-	-
-	-
C/2 - 1	C/2 - 1
C/2 + 1	-(C/2 - 1)
-	-
-	-
C-2	-2
C-1	-1
C	0

In this representation positive integers are represented in the *True Form* while negative are represented in the *Complement Form*.

With respect to how is the complementation constant C chosen we can further distinguish two representations within the TC system.

If the complementation constant is chosen to be equal to the range of possible values taken by x_e, $C = r^n$ in a conventional number system where $0 \le x_e \le r^n - 1$ than we have defined *Range Complement* (RC) system. If on the other hand, complementation constant is chosen to be: $C = r^n - 1$ we have defined *Diminished Radix Complement* (DRC),(also known as *Digit Complement* (DC)) number system. Representations of the RC and DRC number representation systems are shown in Table 3.

Table 3: Mapping of the explicit value x_e into RC and DRC number representations

x_e	X_i (RC)	X_i (DRC)
0	0	0
1	1	1
2	2	2
-	-	-
-	-	-
$\frac{1}{2}r^n - 1$	$\frac{1}{2}r^n - 1$	$\frac{1}{2}r^n - 1$
$\frac{1}{2}r^n$	$-\frac{1}{2}r^n$	$-\left(\frac{1}{2}r^n - 1\right)$
-	-	-
-	-	-
-	-	-
$r^n - 2$	-2	-1
$r^n - 1$	-1	0

As can be seen from the Table 3. Radix Complement system provides for one unique representation of zero because the complementation constant $C = r^n$ falls outside the range. There are two representations of zero in Diminished Radix Complement system, $x_e = 0$ and $r^n - 1$. The RC representation is not symmetrical and it is not closed system under the change of sign operation. The range for RC is: $[-\frac{1}{2}r^n , \frac{1}{2}r^n - 1]$.
The DRC is symmetrical and has the range of: $[-\left(\frac{1}{2}r^n - 1\right) , \frac{1}{2}r^n - 1]$.

For the radix r=2 RC and DRC number representations are commonly known as *Two's Complement* and *One's Complement* number representation systems. Those two representations are illus-

trated by an example in the Table 4 for the range of values $-(4 \leq X_i \leq 3)$.

Table 4: Two‘s Complement and One‘s Complement representation

	Two‘s Complement	C=8	One‘s Complement	C=7
X_i	x_e	X_i 2’s complement	x_e	X_i 1’s complement
3	3	011	3	011
2	2	010	2	010
1	1	001	1	001
0	0	000	0	000
-0	0	000	7	111
-1	7	111	6	110
-2	6	110	5	101
-3	5	101	4	100
-4	4	100	3	-

Algorithms for Elementary Arithmetic Operations

The algorithms for the arithmetic operation are dependent on the number representation system used. Therefore their implementation should be examined for each number representation system separately given that the complexity of the algorithm, as well as it's hardware implementation is dependent on it.

Addition and Subtraction in Sign and Magnitude System

In SM number system addition/subtraction is performed on pairs (u_s,u_m) and (w_s,w_m) resulting in a sum (s_s,s_m), where u_s and w_s are sign bits and u_m and w_m are magnitudes. The algorithm is relatively complex because it requires comparisons of the signs and magnitudes as well. Extending the addition algorithm in order to perform subtraction is a relatively easy because it only involves change of the sign of the operand being subtracted, There fore we will consider only the addition algorithm.

The algorithm can be described as:

if $u_s = w_s$ (signs are equal) **then:**

$s_s = u_s$ and $s_m = u_m + w_m$ (the operation includes checking for the overflow)

if $u_s \neq w_s$ **then:**

if $u_m > w_m$: $s_m = u_m - w_m$, $s_s = u_s$

else: $s_m = w_m - u_m$, $s_s = w_s$

Addition and Subtraction in True and Complement System

Addition in TC system is relatively simple. It is sufficient to perform modulo addition of the explicit values therefore:

$$s_e = (u_e + w_e) \bmod C$$

Proof will be omitted.

In the **RC number system** this is equivalent to passing the operands through and adder and discarding the carry-out of the most significant position of the adder, which is equivalent to performing the modulo addition (given that $C=r^n$).

In the **DRC (DC) number system** complementation constant is $C = r^n - 1$. Modulo addition in this case is performed by subtracting r^n and adding 1. It turns that this operation can be performed by simply passing the operands through an adder and feeding carry out from the most significant digit position into the carry-in at the least significant digit position. This is also called addition

with **end-around-carry**.

Subtracting two numbers is performed as simply changing the sign of the operand to be subtracted presiding the addition operation.

Change of Sign Operation

The change of sign operation involves the following operation:

$$W_i = - Z_i$$

$$w_e = (- z_e) = (- z_e) \bmod C = C - Z_i \bmod C = C - z_e$$

which means that change of sign operation consists of subtracting the operand z_e from the complementation constant *C*.
In the **DRC (DC) system** complementation is performed by simply complementing each digit of the operand Z_i with respect to *r - 1*. In case of r=2 this result in simple inversion of bits.

In case of **RC system** the complementation is performed by complementing each digit with respect to *r - 1* and adding one to the result.

Multiplication Algorithm

Multiplication operation is performed in a variety of forms, in hardware and software. In the beginning of the computer development any complex operation was usually programmed in software or coded in the microcode of the machine. Some limited hardware assistance was provided. Today it is more likely to find full hardware implementation of the multiplication for the reasons of speed and reduced cost of hardware. However, in all of them multiplication shares the basic algorithm with some adaptations and modifications to particular implementation and number system used. For simplicity we will describe a basic multiplication algorithm which operates on positive n-bit long integers X and Y resulting in the product P which is 2n bit long:

$$P = XY = X \times \sum_{i=0}^{n-1} y_i r^i = \sum_{i=0}^{n-1} X \times y_i r^i$$

This expression indicates that the multiplication process is performed by summing *n* terms of a *partial product*: $X \times y_i r^i$. This product indicates that the *i-th* term is obtained by simple arithmetic left shift of X for the *i* positions and multiplication by the single digit y_i. For the binary radix *r=2*, y_i is 0 or 1 and multiplication by the digit y_i is very simple to perform. The addition of *n* terms can be performed at once, by passing the *partial products* through a network of adders (which is the case of full hardware multiplier) or sequentially, by passing the *partial product* through an adder *n* times. The algorithm to perform multiplication of X and Y can be described as:

$$p^{(0)} = 0$$

$$p^{j+1} = \frac{1}{r}(p^j + r^n X y_j) \text{ for } j=0,....,n-1$$

It can be easily proved that this recurrence results in $p^{(n)}=XY$.

Various modifications of the multiplication algorithm exist, one of the most famous is *"Modified Booth Recoding Algorithm"* described by Booth in 1951. This algorithm allows for the reduction of the number of partial products, thus speeding up the multiplication process. Generally speaking, Booth algorithm is a case of using the redundant number system with the radix higher than 2.

Division Algorithm

Division is more complex process to implement because unlike multiplication it involves *guessing* of the digits of the quotient. Here, we will consider an algorithm for division of two positive integers designated as *dividend Y, divisor X* and resulting in a *quotient Q* and an integer *remainder Z* according to the relation given:

$$Y = XQ + Z$$

In this case dividend contains 2n integers and divisor has n digits in order to produce a quotient with n digits.

The algorithm for division is given with the following recurence relationship [2]:

$$z^{(0)}=Y$$

$$z^{(j+1)}=rz^{(j)} - Xr^n Q_{n-1-j} \text{ for } j=0,.....,n-1$$

this recurrence relation yields:

$$z^{(n)} = r^n(Y - XQ)$$

$$Y = XQ + z^{(n)}r^{-n}$$

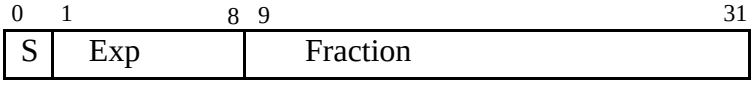
which defines division process with remainder $Z = z^{(n)}r^{-n}$.
The selection of the quotient digit is done by satisfying that $0 \le Z < X$ at each step in the division process. This selection is a crucial part of the algorithm and the best known are r*estoring* and *non-restoring* division algorithms. In the former one the value of the *tentative partial remainder* $z^{(j)}$ is restored after the wrong guess is made of the quotient digit q_j. In the later this correction is not done in a separate step, but rather in the step following. The best known division algorithm is so called SRT algorithm independently developed by: Sweeney, Robertson and Tocher. Algorithms for higher radix were further developed by Robertson and his students, most notably Ercegovac.

Floating Point Representation

Number represented as signed integers can only cover the range limited by the number of digits *n* and choice of the radix *r*. For the choice of radix *r* and *n* digits used, the maximum positive integer that can be represented is r^n*-1*.
Often scientific computation requires the use of very small numbers as well as very large ones represented with some required precision. To satisfy those needs of the scientific, or engineering computation *Floating Point* FP format is used to represent the numbers which are represented as:

$$X = S \times B^{\mp Exp} \times Fract$$

Where *S* is a sign bit (0,1), *B* is a selected base, *Exp is exponent* (which contains it's own sign, or is biased), *Fract* is a fraction of the number. Typical FP number represented in a 32 bit word is shown:



Using this particular representation we can represent the range of the numbers from:

$$-(1-2^{-24}) \times 2^{127} \le X \le -0.5 \times 2^{-128}$$

for the negative numbers and

$$0.5 \times 2^{-128} \le X \le (1-2^{-24}) \times 2^{127}$$

for the range of positive numbers.
Different computer manufacturers have adopted their own standards for the floating point number representation such as: IBM, DEC, Intel. This has led to an effort to introduce a standard for floating point representation and computation resulting in **IEEE Standard 754**. More about floating point computation and representation can be found in [3].

References:

[1] A. Avizienis, "*Digital Computer Arithmetic: A Unified Algorithmic Specification*", Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971.

[2] M. Ercegovac, "*Digital Systems and Hardware/Firmware Algorithms*", Chapter 12: Arithmetic Algorithms and Processors, John Wiley & Sons, 1985.

[3] S.Waser, M.Flynn, *"Introduction to Arithmetic for Digital Systems Designers"*, Holt, Rinehart and Winston 1982.

To Probe Further:

For more information about specific arithmetic algorithms and their implementation one can see: *Kai Hwang, "Computer Arithmetic: Principles, Architecture and Design", John Wiley & Sons 1979.* Also: *E. Swartzlander, "Computer Arithmetic, Volume I & II, IEEE Computer Society Press 1980, Los Alamitos, California* . Publications in *IEEE Transactions on Electronic Computers* and *Proceedings of the Computer Arithmetic Symposiums* by various authors, most notably by *Milos Ercegovac* are very good source for detailed information on particular algorithm or implementation.
A good coverage of the Floating Point Arithmetic could be found in book by Hwang and further details in *IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Standard 754, 1985* and its discussion in *IEEE Computer Magazine vol.14, No.3, p.51-6*2.

Defining Terms:

An Algorithm is decomposition of the computation into subcomputations with an associated precedence relation that determine the order in which these sub-computations are performed [2].

Number Representation System is a defined rule which associates one numerical value x_e with every *valid* bit string x.

Non-Redundant number system is the system where for each bit string there is one and only one corresponding numerical value x_e.

Redundant number system is the system in which the numeric value x_e could be

represented by more than one bit string.

Explicit value $\boldsymbol{x_e}$ is a value associated with the bit string according to the rule defined by the number representation system being used.

Implicit value $\boldsymbol{X_i}$ is the value obtained by applying the arithmetic function defined for the interpretation of the explicit value x_e.