ALGORITHMS FOR CORRECTING CARRIES IN A CARRYING ADDER, WITH AN EXAMPLE FROM ARRAY SOFTWARE

By

Nicholas Zvegintzov

Carnegie-Mellon University
Pittsburgh, Pennsylvania
June, 1969

This work, apart from the computing, was supported by the Advanced Research Projects Agency of the Office of the Secretary or Defense (F 44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

The computing was performed with the facilities of On-Line Systems, Inc., who generously granted the author an educational discount.

SUMMARY

A carrying adder is defined as a device that takes two integer n-tuples, adds corresponding positions sequentially from the right, mod some predefined radix, and transmits a 1 to be added in the next left position each time the mod operation is non-trivial. One of the two principal storage methods for arrays has the characteristic that the integer sum of two represented array points corresponds to their carrying sum. A fixed time algorithm is described for distinguishing true carries from propagated carries, and compared with other algorithms. A table lookup algorithm is described for detecting carries in the array example and its efficiency is discussed.

Key Words and Phrases:

adder, array, carry, carry correction, carrying adder, integer n-tuple, jump call, storage map, timing, table lookup

CR Categories

4.40, 6.32

1. This paper concerns n-tuples, objects of the form

$$A = a[1]\ a[2]\ \ldots\ a[n]$$

or equivalently

$$A = \langle a[1],\ a[2],\ \ldots,\ a[n] \rangle.$$

Context, spaces, or ',' are used to separate the elements, and spaces or '<' and '>' are used to delimit the n-tuple.

More specifically, it deals with integer n-tuples, each of whose elements a[i] is in the range [0,z[i]) for some pre-chosen positive z[i], i.e., each of whose elements is one of the integers mod z[i]. The z[i]'s will also be written as the n-tuple

$$Z = z[1]\ z[2]\ \ldots\ z[n].$$

The set of n-tuples satisfying the above restriction for a given Z will be known as the n-tuples mod Z and the set of n-tuples mod Z will be denoted IZ. The number of distinct members of IZ is $z = \prod_{i=1}^{n} z[i]$.

Given an n-tuple $A \notin IZ$, we define the operation A mod Z that takes it into a member of IZ by:

$$A \bmod Z =$$

$$\langle a[1] \bmod z[1],\ a[2] \bmod z[2],\ \ldots,\ a[n] \bmod z[n] \rangle.$$

This paper focusses on the problem of adding and subtracting members of IZ, where adding and subtracting are defined and written as follows:

$$A \pm \pm B =$$

$$\langle a[1] \pm b[1],\ a[2] \pm b[2],\ \ldots.,$$

$$a[n] \pm b[n] \rangle \mod Z.$$

The operations A + + B and A -- B are trivially performed by n adders operating in parallel, the first adder adding mod z[1], the second mod z [2], and so on. The paper, however, assumes a carrying adder which performs not A + + B and A -- B but A + * B and A - * B, defined as follows:

$$A + * B = D \mod Z$$

where D is formed sequentially from right to left (i.e., from n to 1) as follows:

$$d[n] = a[n] + b[n]$$

$$d[i] = \begin{cases} a[i] + b[i] \text{ if } d[i+1] \in [o, z[i+1]) \\ a[i] + b[i] + 1, \text{ otherwise.*} \end{cases}$$

Similarly

$$A - * B = D \mod Z$$

where D is formed sequentially from right to left (i.e., from n to 1) as follows:

$$d[n] = a[n] - b[n]$$

$$d[i] = \begin{cases} a[i] - b[i], \text{ if } d[i+1] \in [o, z[i+1]) \\ a[i] - b[i] - 1, \text{ otherwise .*} \end{cases}$$

These (disguised by formalism) are exactly the rules for adding and subtracting two integers mod z where A and B are interpreted using positional notation, with position i having radix z[i]. In the Arabic version of positional notation,

$$Z = 10\ 10\ \ldots\ldots\ 10$$

and these are the rules for adding and subtracting two integers mod 10^n.

In a given performance of the operations + * and - * an instance in position i of the case marked * in the definitions will be referred to as a carry into position i. Note that a carry into position i during + * (- *) is a 1 (-1) -- as is consistent with adding (subtracting) positional numbers whose elements a[i] and b[i] are restricted to the integers mod z[i].

One question answered in this paper is the following: given only a carrying adder which performs + * and - *, and a carry indicator C set by the carrying adder so that

$$c[i] = \begin{cases} 1, & \text{if a carry occurred into position i} \\ 0, & \text{otherwise} \end{cases}$$

how can the sum A ++ B or A -- B be reconstructed?

This problem is almost exactly the opposite of that encountered by the designer of hardware, who in general is faced with the problem of synthesizing a carrying adder using a set of parallel non-carrying adders (cf. Winograd, 1965), a demand which arose from the preeminence of arithmetic among the historical uses of computers. The problem of this paper arises at the software level for a user (or program) which has to employ the more sophisticated carrying adders to simulate the simpler parallel addition. Section 2 of this paper gives a practical example of the problem, found in the programming of jumps in arrays. Section 3 gives an unintuitive algorithm that solves the problem in constant time and compares it with an intuitive algorithm whose time rises unboundedly with the length of the n-tuples. Section 4 applies the algorithm to the array programming example and assesses the efficiency of the application.

2

2.1 An array is a computational structure that associates a distinct store with each of the z distinct members of IZ.

This is a direct characterization of an array--say, 'example'--defined in ALGOL

array example [0:z[1]-1,0:z[2]-1, ..., 0:z[n]-1]

or in FORTRAN

DIMENSION EXAMPLE (0/z[1]-1,0/z[2]-1, ..., 0/z[n]-1).

It also characterizes with only slightly greater indirection an array whose i'th element lies within some general limits [p[i], q[i]], since the integers in this interval have a direct and obvious equivalence to the integers mod (q[i] - p[i] + 1).

There are two kinds of call that can be made on an array--an absolute call and a jump call. In an absolute call the using program presents the array with an n-tuple in IZ and receives in return the corresponding store name. This is the kind of call implicit in the use of arrays in ALGOL and FORTRAN. The jump call is not implemented in these source languages. An array allowing jump calls must remember the n-tuple P associated with the last store returned. In a jump call the using program again presents the array with an n-tuple J from IZ, but in this case receives in return the store associated with P + + J -- (which becomes the n-tuple remembered).

It is argued more extensively elsewhere (Zvegintzov, 1969) that jump calls ought to be implemented in any source language that incorporates arrays. Briefly the reasons are:

(1) Absolute and jump calls form a pair that are jointly implicit in the structural definition of an array--corresponding exactly to the twin models of a vector in linear algebra as a point and as an arrow.

(2) Jump calls are the natural way to conceptualize algorithms which involve making or taking a path across an array.

(3) In an unbounded array, jump calls allow the specification of an n-tuple having arbitrarily large elements without fear of overflow.

(4) Jump calls can be used to remove common computation from inside some kinds of loop (see Samelson and Bauer, 1960).

The example promised in the last section is the programming of jumps in an array using the weighted sum storage algorithm (Sattley, 1961). This is one of the two algorithms used almost universally to store arrays. (The other is the sorting tree algorithm--see, for instance, Fredkin, 1960 and Jodeit, 1968. For a different and unusual storage algorithm see Kuck, 1968.)

The weighted sum algorithm involves a function f_n that gives an equivalence between the n-tuples mod Z and the integers mod z. A block of storage of length z is set aside in core starting at (say) b, and the location corresponding to an n-tuple P is $b + f_n(P)$.

The next subsection defines the function f_n. The relevance of this function to the problem of the paper is that if

$$p = f_n(P) \quad \text{and} \quad q = f_n(Q)$$

then $p + q \bmod z = f_n(P + * Q)$, i.e., the sum of the integer representations is equivalent to the carrying sum of the n-tuples.

2.2 The function f_n will be constructed recursively using the fact that an n-tuple

$$<p[1], p[2], ..., p[n]>$$

may be regarded as a pair (2-tuple) consisting of its rightmost element p[n] and the remaining (n-1)-tuple, thus:

$$<<p[1], p[2], ..., p[n-1]>, p[n]>.$$

First to be investigated, therefore, is a family of functions which provide an equivalence between the pairs mod <x y> and the integers mod x×y. The functions are defined by the functional f2(w) that takes as its parameter the radix of the second element of the pair. For a given y and a given pair

$$P = p[1]\ p[2]$$

define

$$f2(y)(P) = y \times p[1] + p[2].$$

Example Z = 2 3, z = 2×3 = 6

P	f2(3)(P)
00	0×3 + 0 = 0
01	0×3 + 1 = 1
02	0×3 + 2 = 2
10	1×3 + 0 = 3
11	1×3 + 1 = 4
12	1×3 + 2 = 5

f2(w) has the following four properties L1-L4. (The proofs may be skipped by all but the most suspicious.)

(L1) f2(w) is one-to-one

Proof: Let A = a[1] a[2], B = b[1] b[2], Z = x y.

Assume f2(y)(A) = f2(y)(B)

$\Rightarrow y\chi a[1] + a[2] = y\chi b[1] + b[2]$

$\Rightarrow y\chi(a[1] - b[1]) = b[2] - a[2]$

$\Rightarrow a[1] - b[1] = 0$ since $|b[2] - a[2]| < y$

$\Rightarrow a[1] = b[1]$ and $a[2] = b[2]$

$\Rightarrow A = B$ QED.

(L2) f2(w) is onto

Proof:

From number theory we have that for any integers s and y, $y \neq 0$, there is a unique integer t

$$s = y\chi t + s \bmod y.$$

Let G2(y) be the function that maps the integers mod $x\chi y$ by

$$g2(y)(s) = \langle t, s \bmod y\rangle$$

using this relation. Since $s \in [0, x\chi y)$, $t \in [0,x)$ and $G2(y)(s) \in I\langle x,y\rangle$, the domain of f2(y). Furthermore

$$\begin{aligned} & f2(y)(G2(y)(s)) \\ &= f2(y)(\langle t, s \bmod y\rangle) \\ &= y\chi t + s \bmod y \\ &= s \end{aligned}$$

i.e., G2(y) is the inverse of f2(y), and f2(y) is onto.

Having established the equivalence between $I\langle x\ y\rangle$ and the integers mod $x\chi y$ via f2(y), for $A \in I\langle x\ y\rangle$ and $a \in [0, x\chi y)$ write

$$A \approx a$$

if $f2(y)(A) = a$.

As defined in (L2), G2(y) applies only to integers mod $x\chi y$, but

it can obviously be extended to the domain of all integers. For an arbitrary integer s, G2(y)(s) is a pair

$$\langle t,\ s \bmod y \rangle$$

which fails to be a member of I<x y> only in that its first member is (perhaps) not in [0,x). The relation between integers in [0,x×y) and those not in [0,x×y) is that the operation 'mod x×y' on the integer corresponds to 'mod x' on the first element of the corresponding pair.

Formally:

(L3) $\underline{G2(y)(s) \bmod \langle x\ y\rangle = G2(y)(s \bmod x\times y)}$

Proof:

Let $s = x\times y\times t' + s \bmod x\times y$

and $s \bmod x\times y = y\times t + (s \bmod x\times y) \bmod y$

$= y\times t + s \bmod y.$

Then $G2(y)(s) \bmod \langle x\ y\rangle$

$= \langle x\times t' + t,\ s \bmod y\rangle \bmod \langle x\ y\rangle$

$= \langle (x\ t' + t) \bmod x,\ s \bmod y \bmod y\rangle$

$= \langle t,\ s \bmod y\rangle$

$= G2(y)(s \bmod x\times y).$ QED

The next result is the crucial one for the purpose of this paper: adding (subtracting) the integer representations of pairs A and B mod x×y is equivalent to a carrying add (subtract) of the pairs.

Formally:

(L4) If $a \approx A$ and $b \approx B$, then

$$a \pm b \bmod x\chi y \approx A \pm^{*} B.$$

Proof for + (- is similar):

$$a + b = f2(y)(A) + f2(y)(B)$$

$$= y\chi(a[1] + b[1]) + (a[2] + b[2]).$$

Then if $0 \le a[2] + b[2] < y$,

$$G2(y)(a+b) = \langle a[1] + b[1],\ a[2] + b[2] \rangle,$$

else if $y \le a[2] + b[2] < 2y$,

$$a + b = y\chi(a[1] + b[1]) + y + (a[2] + b[2]) \bmod y,$$

and $G2(y)(a+b)$

$$= \langle a[1] + b[1] + 1,\ (a[2] + b[2]) \bmod y \rangle.$$

In either case $G2(y)(a+b \bmod x\chi y)$

$$= G2(y)(a+b) \bmod \langle x\ y \rangle \qquad \text{(by L3)}$$

$$= A +^{*} B. \qquad \text{QED}$$

Given these properties of the functional f2, define for a fixed n and $Z = \langle z[1]\ z[2]\ \ldots\ z[n] \rangle$, and $A \in IZ$:

$$f_1(a) = a[1]$$

$$f_i(A) = f2(z[i])(\langle f_{i-1}(A),\ a[i] \rangle).$$

$f_n(P)$ is a function from IZ to the integers mod z. By recursive extension of L1 and L2 f_n is one-to-one and onto and we may write

$$a \approx A \text{ if } a = f_n(A).$$

By recursive extension of L3 the operation 'mod z' on an integer corresponds to 'mod z[1]' on the first element of the corresponding n-tuple. By recursive extention of L4:

$$(a \pm b) \bmod z \approx A \pm^{*} B, \text{ if } a \approx A \text{ and } b \approx B.$$

(It is not surprising that integer addition of the f_n values corresponds to carrying add of the n-tuples; f_n, calculated for Z = <10 10 .... 10> is, of course, the function that gives the integer value of an n-digit positional (Arabic) number.)

For $f_3(A)$ for Z = 4 3 2 and Z = 3 3 3 see tables I and II.

Example n = 3, Z = 3 3 3

	A 021	A 021	$f_3(A)$ 7
+ +	B 002	+ * B 002	+ $f_3(B)$ 2
	= 020	= 100	= $f_3(A+*B)$ 9
		carry 110	

2.3 The remaining part of the paper will assume a carrying adder which for inputs A and B returns:

(1) A $\pm$ * B

(2) sets a carry indicator CARRY, an n-tuple mod <2 2 .... 2>, so that carry [i] = 1 if and only if a carry occurred into position i.

In the array case the information given by the carries is not wholly useless. We will say an overflow occurred in position i if

$$a[i] \pm b[i] \notin [0,z[i]).$$

Information of an overflow in position i will be of use to most types of array software: in a bounded array (ALGOL or FORTRAN) it will generate a run error 'subscript i out of bounds'; in an unbounded array (Zvegintzov, 1969) it will generate a move to a neighboring 'page'.

An overflow in position i > 1 will always generate a carry into position i-1. An overflow in position 1 will result, as L3 shows, in

$$f_n(A) \pm f_n(B) \notin [0,z).$$

Unfortunately it is not the case that all carries into position i-1 are caused by an overflow in position i. Define a carry into position i-1 as <u>true</u> if there is an overflow in position i. The other source of a carry is:

$$a[i] + b[i] = z[i] - 1$$

or $a[i] - b[i] = 0$

and, in either case, a carry into position i.

This is termed a <u>false</u> carry.

In the example given above carry[1] is false and carry[2] is true. Carry[3] = 0 is not a carry at all.

The next section gives an algorithm that uses the carrying adder and the carry indicator to reconstruct, for a given use of the adder, an n-tuple mod <2 2 .... 2> TRUCAR with the property that trucar[i] = 1 if and only if there was a true carry into position i during that use.

With TRUCAR the array can take the corrective actions mentioned above. In addition TRUCAR has the property that it is the n-tuple to be used to reverse the previous carries.

Formally:

(L5) <u>$(A \pm * B) \mp * TRUCAR = A \pm \pm B.$</u>

Proof for + :

Assume C is the carry from A + * B = D, CC is the carry from D - * TRUCAR. It is sufficient to show cc[i] = 1 if and only if c[i] is a false carry.

Proof by induction on i:

Case Clearly true for i = n

(c[i] = cc[i] = 0).

Case Assume true for j $\in$ (i,n].

Case c[i+1] = 0

Then trucar [i+1] = cc[i+1] = 0

$\Rightarrow$ cc[i] = 0.

Case c[i+1] = 1

Then either trucar[i+1] = 1 or cc[i+1] = 1, depending if

c[i+1] is true or false.

Case d[i+1] = 0.

Then cc[i] = 1, correct because c[i] is false.

Case d[i+1] > 0.

Then cc[i] = 0, correct because c[i] = 0 or is a true carry.

QED

3. This section describes an algorithm Q which finds TRUCAR and can hence by L3 be used to correct carries, and an algorithm R which corrects carries, and hence by L3 can be used to find TRUCAR. R is an intuitive algorithm which takes an amount of processing which rises unboundedly with the length n of the n-tuples processed. The interest of Q is that it takes a constant time, given constant time boolean and shifting operations in the source language, and a constant time carrying adder.

3.1 It will be remembered from the definition of false carries in 2.3 that if there is a carry into position i (c[i] = 1), it is false if

$$d[i+1] = 0 \qquad \text{(if D = A + * B)}$$

$$\text{or} \quad d[i+1] = z[i+1]-1 \qquad \text{(if D = A - * B)}$$

and in either case c[i+1] = 1.

TRUCAR can therefore be derived from C using an n-tuple and <2 2 .... 2> WARNING which has the property:

$$\text{warning}[i] = \begin{cases} 1, \text{ if } d[i] = 0 \quad \text{(case of '+ *')} \\ \quad \text{or } d[i] = z[i]-1 \quad \text{(in case '- *')} \\ \text{else } 0. \end{cases}$$

Given WARNING, TRUCAR can be constructed by noting that in each position i

(1) if there was no carry originally, certainly there was no true carry;

(2) if there was a carry, then it was a true carry only if either there was no carry to the right or there was no warning to the right.

This is algorithm Q.

The algorithm can either be implemented by table lookup or calculated by boolean and shifting operations. The calculation takes constant time in the sense that:

(1) in principle boolean and shift operations can be performed in constant time regardless of length of word;

(2) in practice hardware offers constant time boolean and shift operations on words of a length (32, for instance) which is long in terms of the potential uses of the algorithm.

Algorithm Q now follows. (Given an n-tuple A, the notation A^+ is used for the result of shifting A left by one position, setting the rightmost bit arbitrarily.)

D := A $\pm$ * B, carry to C;

form WARNING on D;

TRUCAR := C and not (C and WARNING)$^+$.

In some applications it may be trivial to form WARNING because the positions of zeroes and nines (i.e., instances of z[i]-1) will be obvious, but this need not be so. It is not so for the array application given above. There is nothing about the integer 18 (see table II) that shows it corresponds to a 3-tuple mod <3 3 3> with zeroes in positions 2 and 3. The next subsection, therefore, gives a constant time algorithm that finds zeroes (nines) using only the carrying adder and its carry indicator.

3.2 Algorithm W^+

Given an n-tuple D find WARNING, an n-tuple mod <2 2 .... 2> such that:

$$\text{warning}[i] = \begin{cases} 1, & \text{if } d[i] = 0 \\ 0, & \text{else} \end{cases}, \quad 1 < i \leq n.$$

(warning[1] may be filled arbitrarily since it is not used by algorithm Q.)

The algorithm employs two predetermined n-tuples mod Z

$$\text{SUBADD}(0) = 0101 \ldots 01(0)$$

$$\text{SUBADD}(1) = 1010 \ldots 10(1).$$

These two n-tuples have alternating 0's and 1's, and they are complimentary. The last digit of each is the same as the first when n is odd.

This algorithm also may be implemented either by table lookup or by boolean and shift operations. Algorithm W^+ follows. (Given an n-tuple A, let A^- be the result of shifting A to the right by one position, setting the leftmost bit arbitrarily.)

```
for x := 0,1 do
   begin DUMMY := D - * SUBADD(x),
         carry to CARRY(x);
     WARNING(x) :=
         CARRY(x)⁻ and not (SUBADD(x) and CARRY(x))
   end;
WARNING := WARNING(0) or WARNING(1).
```

To see why this algorithm works, consider for each position i the pattern of subadd(x)[i], carry(x)[i] and their right neighbors.

Case S0

	i	i+1	
subadd(x)	0	1	warning[i+1] := 0
carry(x)	0		

d[i+1] was diminshed by at least 1 (subadd(x)[i+1] = 1) and caused no carry, hence d[i+1] ≥ 1.

Case S1

	i	i+1	
subadd(x)	1	0	warning[i+1] := 1
carry(x)	1		

d[i+1] was diminished by not more than 1 (since subadd(x)[i+1] = 0), and this caused a carry, hence d[i+1] = 0.

Case DD0

	i	i+1	
subadd(x)	1	0	warning[i+1] := 0
carry(x)	0	1	

d[i+1] was diminished by 1 (carry(x)[i+1]) and this caused no carry, hence d[i+1] ≥ 1.

Case DD1

	i	i+1	
subadd(x)	0	1	warning[i+1] := 1
carry(x)	1	0	

d[i+1] was diminished by 1 (subadd(x)[i+1]) and this caused a carry, hence d[i+1] = 0.

Case DS0

	i	i+1	
subadd(x)	1	0	warning[i+1] indeterminate
carry(x)	0	0	

d[i+1] was diminished by 0 = subadd(x)[i+1] + carry(x)[i+1], hence we have no information on d[i+1].

Case DS1

	i	i+1	
subadd(x)	0	1	warning[i+1] indeterminate
carry(x)	1	1	

d[i+1] was diminished by 2 = subadd(x)[i+1] + carry(x)[i+1], hence we have no information on d[i+1].

Cases DS0 and DS1 are left indeterminate by a given SUBADD(x). The algorithm relies on the fact that these two cases will be solved by the other SUBADD(1-x).

To see this, examine cases DS0 and DS1 more closely:

Case DS0 (continued)

	i	i+1	i+2
d			≥ 1
subadd(x)	1	0	1
carry(x)	0	0	
subadd(1-x)	0	1	0
carry(1-x)	1/0	0	

Note that case DS0 in position i involves case S0 in position i+1, showing that d[i+2] ≥ 1. Then since subadd(1-x)[i+2] = 0, carry(1-x)[i+1] = 0, and subadd(1-x)[i] will give either case S0 or case DD1, both determinate.

Case DS1 (continued)

	i	i+1	i+2
d			0
subadd(x)	0	1	0
carry(x)	1	1	
subadd(1-x)	1	0	1
carry(1-x)	1/0	1	

Note that case DS1 in position i involves case S1 in position i+1, showing that d[i+2] = 0. Then since subadd(1-x)[i+2] = 1, carry(1-x)[i+1] = 1, and subadd(1-x)[i] involves either case S1 or DD0, both determinate.

The algorithm sets indeterminate cases to 0 and performs an 'or' on WARNING(0) and WARNING(1) to derive the correct bit in each position.

Example n = 8, Z = 33333333

D	21001102
-*SUBADD(0)	01010101
=	12221001
CARRY(0)	11100000
WARNING(0)	00110000

D	21001102
-*SUBADD(1)	10101010
=	10200022
CARRY(1)	01000100
WARNING(1)	00100010
or WARNING(0)	00110000
= WARNING	00110010

The algorithm W^+ to find zeroes becomes the algorithm W^- to find nines by changing '-*' to '+ *'. In the explication following the algorithm change:

'-' to '+'

'diminished' to 'increased'

'd[j] ≥ 1' to 'd[j] ≤ z[j] ≠ 2'

'd[j] = 0' to 'd[j] = z[j] ≠ 1'.

This algorithm can also be implemented either with boolean and shift instructions or with table lookup. However, it also uses the carrying adder, and it is only constant time to the extent that a carrying add is. We know from Winograd,1965,that a constant time carrying add is theoretically impossible. The only claim made, then, is that the algorithm takes constant time in the context of a large fixed-time carrying adder whose capacity is certain not to be exceeded. (Such is the situation in the array example, where carrying adds are represented by integer adds on an integer adder of far greater capacity than would ever be needed.)

3.3 R is a much more intuitive algorithm whose principle is: 'keep subtracting the carries until they go away'. It happens to be an effective strategy.

In defining R assume for convenience that C is a signed carry indicator--that is, after a use of the carrying adder,

c[i] = if there was a carry into position i then
    if the operation was '+ *' then +1
    else -1
  else 0.

Then R is:

D := E := A $\pm$ * B, signed carry to C;
while C $\neq$ 00....0 do
  E := E - * C, signed carry to C.

The result is E = A $\pm\pm$ B.

To prove this, note:

(1) initially E -- C = A $\pm\pm$ B, by definition of signed carry;

(2) this is preserved by each cycle through the while statement since the carries are subtracted and any induced carries are recorded again in C;

(3) at termination C = 00...0, hence

E -- 00...0 = E = A $\pm\pm$ B;

(4) the algorithm terminates since at each cycle through the while statement at least the rightmost carry is eliminated.

EXAMPLE

n = 15, Z = 3 3 3

```
A
  2 2 2 1 0 0 0 1 2 1 1 2 2 2 1
+* B
  1 2 0 2 2 2 2 1 1 2 0 2 1 2 1
=  D
  1 2 0 1 0 0 0 0 1 0 2 2 1 1 2
CARRY
  1 1 1 1 1 1 1 1 1 0 1 1 1 0 0
D -* CARRY
=
  0 0 1 2 1 1 1 2 0 0 1 1 0 1 2
CARRY
  0-1-1-1-1-1-1 0 0 0 0 0 0 0 0
D  -* CARRY
=
  0 2 0 0 2 2 2 2 0 0 1 1 0 1 2
CARRY
  0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
D -* CARRY
=
  0 0 2 0 2 2 2 2 0 0 1 1 0 1 2
CARRY
  0-1 0 0 0 0 0 0 0 0 0 0 0 0 0
D -* CARRY
=
  0 1 2 0 2 2 2 2 0 0 1 1 0 1 2
CARRY
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

3.4 An exact measure of speed for these algorithms depends on their exact implementation. Instead a crude measure will be used--cycles, defined as the number of uses of the carrying adder.

Q obviously uses 2 cycles to find WARNING, hence 2 to find TRUCAR, and 3 to find A $\pm\pm$ B, regardless of the length n of the n-tuples.

The expected number of cycles for R is related to clusters, where a cluster is defined most simply to be a set of consecutive positions with carries after the first correction E - * C. (In the previous example the only cluster is underlined.) In general a cluster of length p requires p/2 cycles to be reduced by this algorithm, and in general the clusters can be reduced in parallel; therefore the key variable is the expected maximum cluster length m. This is not a very tractable variable, but simulation studies with z[1] = z[2] = ... z[n] and A and B taken uniformly from the n-tuples mod Z show

$$m \sim \ln(n).$$

m also varies inversely with z[i], since it depends on the absolute probability of a 0 or a 1 in a given position.

Simulation gives the following rough values for algorithm R, with n-tuples A and B uniform on IZ:

(1) Z = 33....3

n	m
10	2.3
11	2.5
12	2.6
15	2.9
20	3.3
30	3.8

(2) $Z = 10\ 10\ \ldots\ 10$

n	m
30	1.9

In any actual use a simulation study of algorithm R should be made since for low n or high z[i] R will probably be faster than Q. However, the existence of Q sets an absolute ceiling on the task of calculating TRUCAR.

In the next section a table lookup method will be given for detecting carries in the array example and the above considerations will be used to calculate the efficiency of this method over its obvious rival.

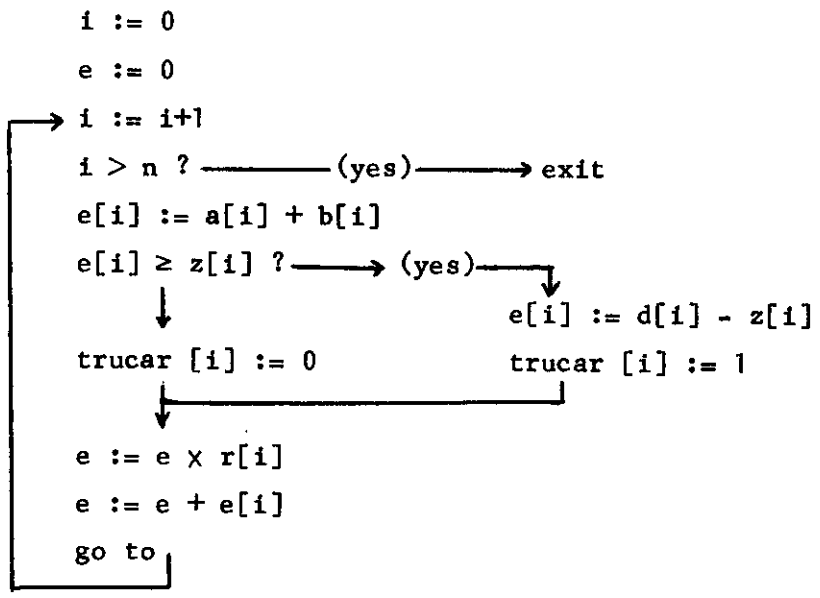
4. Recall from section 2 that the weighted sum algorithm for arrays converts an n-tuple P into an integer $f_n(P)$. In this section two algorithms S and T are given for processing jumps.

Assume A is the n-tuple remembered, B is the jump call n-tuple, and E := A + + B is the new n-tuple. S is the intuitive algorithm which stores A and accepts B as n-tuples and recreates $f_n(E)$. T stores A and receives B as $a = f_n(A)$ and $b = f_n(B)$ and uses either algorithm Q or R to find $e = f_n(E)$.

S and T are now described and their efficiency compared.

4.1 Algorithm S

E := A + + B, e := f_n(E), form TRUCAR



Counting one time unit for each instruction, and assuming equally probable the two branches of the inner loop, S takes 8.5 units to go through the outer loop, hence

$$\text{time}(S) \approx 8.5 \times n.$$

4.2 Algorithm TQ

$e := f_n(A + + B)$ (where $a = f_n(A)$, $b = f_n(B)$),

form TRUCAR, using algorithm Q

		1:2.5	d := a + b mod z
		2:4	form CARRY
Q	W+	3:2.5	dummy := d - f_n(SUBADD(0)) mod z
		4:4	form CARRY(0)
		5:4	form WARNING(0)
		6:2.5	dummy := d - f_n(SUBADD(1)) mod z
		7:4	form CARRY(1)
		8:4	form WARNING(1)
		9:1	WARNING := WARNING(0) or WARNING(1)
		10:4	form TRUCAR
		11:1	t := f_n(TRUCAR)
		12:2.5	e := d - t mod z

The steps have been labeled 'i:j', where i is the step number and j is the time required for the step, calculated as before.

Notes

steps 1,3,6,12:

'a := b + c mod z' is assigned time = 2.5 as the expansion of:

a = b + c

a ≥ z? ——— (yes) ⟶ a := a - z

steps 2,4,7: Time = 4 assigned for forming carrying indicator is justified in section 4.4, which describes the algorithm to find carries.

steps 3,6: Assume f_n(SUBADD(x)) pre-stored

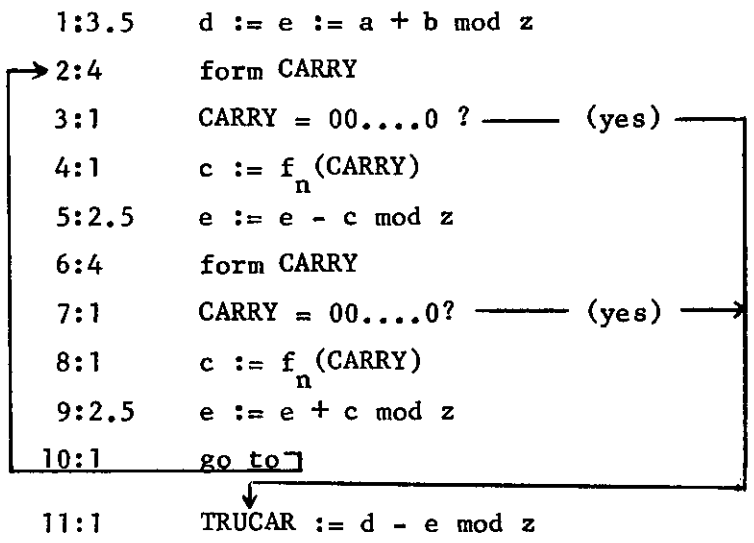
step 11: by table lookup.

Thus time(TQ) = 36.

4.3 Algorithm TR

$e := f_n(A + + B)$ (where $a = f_n(A)$ and $b = f_n(B)$),

form TRUCAR, using algorithm R



Note

steps 4,8: by table lookup

The signed carry of algorithm R has been accomplished by an alternation of sign in the carry correction. The while statement of R now takes 9 units. Hence time(TR) = 7.5 + 9m, where m is the expected number of cycles for R.

Since time(TQ) = 36 and time(TR) = 7.5 + 9m, in this case TQ should be used when

$$7.5 + 9m > 36$$

$$\text{or} \quad m > 3.2.$$

Since TQ sets an absolute upper limit of 36 regardless of n, and time(S) = (8.5)n, T may be used in preference to S when

$$(8.5)n > 36$$

$$\text{or} \quad n > 5.$$

This must be balanced against the space requirements of T. These will be examined in connection with the explanation of an algorithm to detect carries in the array application, which now follows.

4.4 The algorithm requires a table of length z (sometimes less) whose elements are n-1 bit binary words--one bit for each position i = 1,2,....,n-1 into which a carry could occur. The a'th word in the table, where $a = f_n(A)$, is HH(A), a function defined below. The importance of the function HH is that it enables non-carrying binary addition (written '$\oplus$'), provided by the source language or hardware, to be used as a check on the carrying addition mod Z.

The algorithm is:

$$D := A \pm * B$$
$$C := HH(A) \oplus HH(B) \oplus HH(D)$$

and C is the carry indicator.

With table lookup the algorithm looks like:

$$d := a + b \bmod z$$
$$C := T(a) \oplus T(b) \oplus T(d)$$

where $a = f_n(A)$, $b = f_n(B)$,
and $T(f_n(X)) = HH(X)$.

For $A \in IZ$, HH(A) is defined by:

$$HH(A)[i] = [f_i(A) + f_n(A)] \bmod 2$$

using the recursively defined functions f_i of section 2.

We will say that a function ϕ <u>has property</u> P_i if, in the context of a carrying add,

$$\phi = \begin{cases} 1, & \text{if there was a carry into position } i \\ 0, & \text{else.} \end{cases}$$

It remains to prove that the function

$$HH(A)[i] \oplus HH(B)[i] \oplus HH(A +^* B)[i]$$

has property P_i for $1 \le i < n$.

In proving this the following notation will be used. Given an n-tuple mod Z

$$A = a[1]\ a[2]\ \ldots.\ a[n]$$

let

$$A_j = a[1]\ a[2]\ \ldots.\ a[j],\ 1 \le j \le n,$$

be the <u>truncated j-tuple</u> A_j, a member of the j-tuples mod Z_j. Let

$$z_j = \prod_{i=1}^{j} z[i].$$

The algorithm uses the fact that in a carrying add there is no carry into the rightmost position. For a given A and B in IZ, $(A +^* B)_i$, the truncated carrying sum of A and B, differs from $A_i +^* B_i$, the carrying sum of the truncated parts, if and only if there is a carry into position i of $A +^* B$; if so

$$(A +^* B)_i = A_i +^* B_i +^* 1_i$$

where 1_i is the i-tuple with i-1 leading 0's and a trailing 1.

<u>Example</u> n = 4, Z = 3 3 3 3

A 0212	A_2 02
$+^*$B <u>0212</u>	$+^*B_2$ <u>02</u>
= 1201	= 11

Then $(A +^* B)_2 = 12 = A_2 +^* B_2 +^* 01$.

Now $(A +^* B)_i \approx f_i(A +^* B)$,

$A_i +^* B_i \approx f_i(A) + f_i(B) \bmod z_i$, and $1_i \approx 1$.

Hence:

(L6) S_i has property P_i,

where $S_i = [f_i(A) + f_i(B)] \bmod z_i + f_i(A +^* B)$.

Now assume either:

Case C0: z[1] is even--hence $z_1, z_2, \ldots, z_n = z$ are all even;

or

Case C1: z[1], z[2],...,z[n] are all odd, hence $z_1, z_2, \ldots, z_n = z$ are all odd.

One or other case must **exist**, since, from the software's point of view the order of the elements in the n-tuple is irrelevant, and any z[i] even (if there is one) may be taken as z[1].

In either case $(z + z_i) \bmod 2 = 0$ for $1 \le i \le n$.

Now $HH(A)[i] \oplus HH(B)[i] \oplus HH(A +^* B)[i]$

$= [f_i(A) + f_n(A) + f_i(B) + f_n(B)$

$+ f_i(A +^* B) + f_n(A +^* B) \bmod 2$

$= (HI + HN) \bmod 2$

where $HI = f_i(A) + f_i(B) + f_i(A +^* B)$

and $HN = f_n(A) + f_n(B) + f_n(A +^* B)$.

It remains to prove $(HI + HN) \bmod 2$ has property P_i.

Case $f_n(A) + f_n(B) < z$

$\Rightarrow f_i(A) + f_i(B) < z_i$.

Then $HN \bmod 2 = [2 \times (f_n(A) + f_n(B))] \bmod 2 = 0$,

and $(HI + HN) \bmod 2 = HI \bmod 2$

$= \{[f_i(A) + f_i(B)] \bmod z_i + f_i(A + * B)\} \bmod 2$

$= S_i \bmod 2$

has property P_i by L6.

Case: $z_n \leq f_n(A) + f_n(B) < 2 \times z$.

In this case

$HN \bmod 2 = [2 \times (f_n(A) + f_n(B)) - z] \bmod 2$

$= z \bmod 2.$

There are two subcases: either it is also true that

$z_i \leq f_i(A) + f_i(B) < 2 \times z$

or $f_i(A) + f_i(B) = z_i - 1 \approx$

$<z[1]-1, z[2]-1, \ldots, z[i]-1>$

and there is a carry into position i.

Case: $z_i \leq f_i(A) + f_i(B) < 2 \times z_i$

Then $HI \bmod 2$

$= \{[f_i(A) + f_i(B)] \bmod z_i + z_i + f_i(A + * B)\} \bmod 2$

$= S_i \bmod 2 \oplus z_i \bmod 2,$

and $(HN + HI) \bmod 2 = S_i \bmod 2 \oplus (z + z_i) \bmod 2$

$= S_i \bmod 2,$

has property P_i by L6.

Case: $f_i(A) + f_i(B) = z_i - 1$, $f_i(A + * B) = 0$.

Then $HI \bmod 2 = (z_i - 1 + 0) \bmod 2$

and $(HI + HN) \bmod 2 = (z + z_i - 1) \bmod 2 = 1$, which is

correct because in this case there is a carry into position i.

This concludes the proof that the function HH works as described. Review of the proof will show that the algorithm works also for the case A - * B. Tables of HH are given for Z = 4 3 2 and Z = 3 3 3 in tables I and II.

Example (cf example of section 2.2, and table II)

n = 3, Z = 3 3 3

A	021	$f_3(A)$	7
+*B	002	$+f_3(B)$	2
=D	100	$=f_3(D)$	9
CARRY	110		

HH(A)11
⊕HH(B)00
=11
⊕HH(D)00
=CARRY11

To construct a table of HH by hand for an arbitrary Z, first write down the n-tuples P in order of $f_n(p)$. Since $1_n \approx 1$, this corresponds to:

(1) 00....0 is the 0'th n-tuple

(2) if P is the p'th, p' = p + * 1_n is the (p+1)'th.

Then construct a table of H(P), where $H(P)[i] = f_i(P) \bmod 2$:

(1) 00....0 is H(<0 0 .... 0>)

(2) given the p'th entry H(P), form the (p+1)'th, H(P') by:

$$H(P')[i] = \begin{cases} H(P)[i] & \text{if } p[i] = p'[i] \\ H(P)[i] \oplus 1, & \text{else .} \end{cases}$$

To construct HH(P), let HH(P) = H(P) when $f_n(P)$ is even, else HH(P)[i] := H(P)[i] ⊕ 1. This corresponds to modifying

$$H(P)[i] = f_i(p) \bmod 2$$

by the parity of $f_n(P)$.

It remains to calculate k, the time taken to form a carry indicator by the above algorithm. First note that the length of the table can be cut from

$$z = z[1] \times z[2] \times \ldots \times z[n]$$

to

$$\hat{z} = \begin{cases} \text{in case C0, } 2 \times z[2] \times z[3] \times \ldots \times z[n] \\ \text{in case C1, } z[2] \times z[3] \times \ldots \times z[n] \end{cases}$$

by the relation

$$T(a) = T(a \bmod \hat{z})$$

which will be proved below.

Then given D := A + * B in the form 'd := a + b mod z', the algorithm to find C the carry indicator, is:

1:1 C := HH(A) $\oplus$ HH(B)
2:2 j := d mod $\hat{z}$
3:1 C := C $\oplus$ T(j).

(The timing of step 1 assumes, as is consistent with the use of the algorithm, that HH(A) and HH(B) are known without table lookup.)

Thus k = 4.

The space requirements for the table are for a length of $z/z[1]$ or $2 \times z/z[1]$ and a width of n-1 bits. This is large in itself but not relatively large as software for handling an array or array page of length z and width (say) 32 bits. It is in fact comparable with the space used by the rival sort tree algorithm for storing arrays.

(L7) T(a mod $\hat{z}$) = T(a)

Proof for case C0 (C1 is similar).

$\hat{z} = 2 \times z[1] \times z[2] \times \ldots \times z[n] \approx \langle 2\ 0\ 0\ \ldots\ 0 \rangle$,

and if $a = s + k\hat{z}$, $0 < s \leq \hat{z}$,

then $A = S + * \langle 2k\ 0\ 0\ \ldots\ 0 \rangle$, where $S \approx s$.

Let $K = \langle 2k\ 0\ 0\ \ldots\ 0 \rangle$.

Note first that

$HH(K) = 00\ldots0$

since $HH(K)[i] = [f_i(K) + f_n(K)] \bmod 2$

$= [2 \times k \times z[2] \times z[3] \times \ldots \times z[i] + 2 \times k \times z[2] \times z[3] \times \ldots \times z[n]] \bmod 2$

$= 0$

Note second that $+ * K$ causes no carry (since $k[i] = 0$, $1 < i \leq n$).

Hence the carry indicator for $A - * K$

$C = HH(A) \oplus HH(K) \oplus HH(A - * K) = 00\ldots0.$

But $T(a \bmod \hat{z}) = HH(A - * K)$

$= C \oplus HH(K) \oplus HH(A)$

$= 00\ldots0 \oplus 00\ldots0 \oplus HH(A)$

$= HH(A) = T(a).$ QED

5. Summarizing this paper: it has explained the relation of the weighted sum array storage algorithm to carrying adders; it has investigated two algorithms for correcting carries, one intuitive, one less obvious but having the advantage of requiring fixed time; and it has described a table lookup algorithm for detecting carries in the array example.

The disappointment of the study is that though these algorithms together provide a complete system for processing jump calls they do not in practice appear to be preferable to the most straightforward algorithm S. The practical limit of the size of an array or array page in core (say 4K words) puts a practical limit on the dimensionality of an array to be stored by the weighted sum method (say 8). At n = 8 the carry correction algorithm R does not appear to have a sufficient speed advantage to justify its tables and non-intuitive software.

The strength of the study is that a rigorous examination of the algorithms puts the above judgment on a quantitative basis; and perhaps the separate algorithms themselves will be of use to the reader.

References

1. Fredkin, Edward,"TRIE memory", Comm. ACM, 3, Sept. 1960, 490-499.

2. Jodeit, Jane G., "Storage organization in programming systems," Comm. ACM, 11, Nov. 1968, 741-746.

3. Kuck, D. J.,"ILLIAC IV software and application programming," IEEE Transac on Computers, C-17, Aug. 1968, 758-770.

4. Samelson, K., and Bauer, F. L., "Sequential formula translation," Comm. ACM, 3, Feb. 1960, 76-83.

5. Sattley, Kirk, "Allocation of storage for arrays in ALGOL 60,"Comm. ACM, 4, Jan. 1961, 60-65.

6. Winograd, Shmuel, "On the time required to perform addition," J. ACM, 12, April 1965, 277-285.

7. Zvegintzov, Nicholas, Algorithms for Unbounded Arrays, (in preparation).

TABLE I

AUXILIARY FUNCTIONS FOR $n = 3$, $Z = 4\ 3\ 2$

P	F1(P)	F2(P)	F3(P)	H(P)	HH(P)
000	0	0	0	00	00
001	0	0	1	00	11
010	0	1	2	01	01
011	0	1	3	01	10
020	0	2	4	00	00
021	0	2	5	00	11
100	1	3	6	11	11
101	1	3	7	11	00
110	1	4	8	10	10
111	1	4	9	10	01
120	1	5	10	11	11
121	1	5	11	11	00 $\hat{z}$
200	2	6	12	00	00
201	2	6	13	00	11
210	2	7	14	01	01
211	2	7	15	01	10
220	2	8	16	00	00
221	2	8	17	00	11
300	3	9	18	11	11
301	3	9	19	11	00
310	3	10	20	10	10
311	3	10	21	10	01
320	3	11	22	11	11
321	3	11	23	11	00

TABLE II

AUXILIARY FUNCTIONS FOR n = 3, Z = 3 3 3

P	F1(P)	F2(P)	F3(P)	H(P)	HH(P)
000	0	0	0	00	00
001	0	0	1	00	11
002	0	0	2	00	00
010	0	1	3	01	10
011	0	1	4	01	01
012	0	1	5	01	10
020	0	2	6	00	00
021	0	2	7	00	11
022	0	2	8	00	00
100	1	3	9	11	00
101	1	3	10	11	11
102	1	3	11	11	00
110	1	4	12	10	10
111	1	4	13	10	01
112	1	4	14	10	10
120	1	5	15	11	00
121	1	5	16	11	11
122	1	5	17	11	00
200	2	6	18	00	00
201	2	6	19	00	11
202	2	6	20	00	00
210	2	7	21	01	10
211	2	7	22	01	01
212	2	7	23	01	10
220	2	8	24	00	00
221	2	8	25	00	11
222	2	8	26	00	00

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE
ALGORITHMS FOR CORRECTING CARRIES IN A CARRYING ADDER, WITH AN EXAMPLE FROM ARRAY SOFTWARE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)
Nicholas Zvegintzov

6. REPORT DATE
June 1969

7a. TOTAL NO. OF PAGES
39

7b. NO. OF REFS
7

8a. CONTRACT OR GRANT NO.
F44620-67-C-0058

b. PROJECT NO.
9718

c.
6154501R

d.
681304

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT
1. This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES
TECH, OTHER

12. SPONSORING MILITARY ACTIVITY
Air Force Office of Scientific Research (SRMA)
1400 Wilson Boulevard
Arlington, Virginia 22209

13. ABSTRACT

A carrying adder is defined as a device that takes two integer n-tuples, adds corresponding positions sequentially from the right, mod some predefined radix, and transmits a 1 to be added in the next left position each time the mod operation is non-trivial. One of the two principal storage methods for arrays has the characteristic that the integer sum of two represented array points corresponds to their carrying sum. A fixed time algorithm is described for distinguishing true carries from propagated carries, and compared with other algorithms. A table lookup algorithm is described for detecting carries in the array example and its efficiency is discussed.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT