division and multiplier recoding procedures," Dept. of Computer Sci., University of Illinois, Urbana, Rept. 252, December 1967.
[5] J. O. Penhollow, "A study of arithmetic recoding with applications to multiplication and division," Dept. of Computer Sci., University of Illinois, Urbana, Rept. 128, September 1962.
[6] C. S. Wallace, "Suggested design for a very fast multiplier," Dept. of Computer Sci., University of Illinois, Urbana, Rept. 133, February 11, 1963.
[7] J. E. Robertson, internal memo, February 11, 1968.
[8] ——, "Methods of selection of quotient digits during digital division," Dept. of Computer Sci., University of Illinois, Urbana, File 663, 1965.
[9] D. E. Atkins, "Higher radix division using estimates of the divisor and partial remainders," *IEEE Trans. Computers*, vol. C-17, pp. 925–934, October 1968.
[10] ——, "Illiac III computer system manual: Arithmetic units," vol. 1, Dept. of Computer Sci., University of Illinois, Urbana, Rept. 366, December 1969.

A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors

ALGIRDAS AVIŽIENIS, MEMBER, IEEE, AND CHIN TUNG, MEMBER, IEEE

Abstract—**The advent of large-scale integration of logic circuits requires the definition of digital computer structure in terms of large functional arrays of logic of very few types. This paper describes a single-package arithmetic processor called the arithmetic building element (ABE). The ABE accepts operands in either conventional or signed-digit radix-r representation and produces signed-digit results, which the ABE can reconvert to conventional form. Radix 16 is chosen for illustrations. Arrays of ABE's may be arranged to implement unit-time parallel addition, all-combinational multiplication, and more complex functions which are presently computed by subroutines. To facilitate such arithmetic design, a graph model is developed which permits a translation of the given arithmetical algorithm into an interconnection diagram of ABE's. The design procedure is illustrated by an array for polynomial evaluation. Speed, cost, and roundoff error of the array are considered. A computer program has been written for the automatic translation of the algorithm graph to an interconnection graph, and for the evaluation of the cost and speed for a given polynomial degree and a given precision requirement.**

Index Terms—**Arithmetic array modeling, arithmetic building element, arithmetic processors, computer-aided processor design, graph models of arithmetic, microelectronic building block, polynomial evaluation array, signed-digit number systems.**

I. Introduction

THE ADVENT of large-scale integration in the manufacturing of logic circuits has created a new challenge to the designers of digital systems. In order to take advantage of the potentially low cost per circuit it is necessary to define large functional packages of logic elements which have multiple usage in various systems [13]. Storage arrays are most directly suitable for such implementation because of their repetitive internal structure. A typical large arithmetic processor from current generation computers is not readily built up from sizable identical subsystems. To a large extent this is due to the need for a logic structure which provides fast carry propagation during addition. Either the addition time or the structure varies with varying operand lengths.

An algorithm for a carry-free addition and subtraction in which the addition/subtraction time remains independent of the length of the operands has been devised for redundant signed-digit (SD) number representations [1]. SD representations are redundant positional representations with a constant integer radix $r \geq 3$, in which the allowed digit values are a sequence of $2a+1$ integers:

$$\{\bar{a}, \cdots, \bar{1}, 0, 1, \cdots, a\}, \text{ with } r/2 < a < r.$$

In this paper the overbar ($\bar{1}$, $\bar{3}$, etc.) is used to designate negative digit values.

Further studies of SD number systems have shown that the redundancy of representation allows other unusual algorithms, such as wired-in significant-digit arithmetic [2], [14], most-significant-digit-first serial addition and multiplication [2], algorithms which can accept both SD and conventional operands to produce SD results [3], and two-digit product as well as multidigit sum algorithms whose results are suitable inputs to the carry-free two-digit sum algorithm [4]. The self-contained digit-by-digit nature of these algorithms suggested the possibility of a general arithmetic building block suitable for microelectronic implementation [4], [5], [7]. This paper presents the results of

Manuscript received December 15, 1969; revised March 13, 1970. This research was sponsored by Atomic Energy Commission Contract AT(11-1) Gen 10 Project 14.

A. Avižienis is with the Department of Computer Science, University of California, Los Angeles, Calif. 90024.

C. Tung was with the Department of Computer Science, University of California, Los Angeles, Calif. He is now with the IBM Research Laboratory, San Jose, Calif. 95114.

recent investigations which have led to the definition of a single universal arithmetic building element (ABE). The ABE incorporates all the desirable properties of SD arithmetic enumerated above as well as the means to accept conventional number inputs and to convert SD numbers back to conventional form. A storage register at the ABE output provides for "pipelining" [12] of ABE arrays, with the resulting increase in the effective rate of computing. A single ABE can serve as the entire arithmetic unit of a digit-serial computer.

A second important feature of the ABE approach is the separation of the functions of logic design and arithmetic design. The ABE is specified entirely by the numeric algorithms which express the values of output digits in terms of the input digits. These algorithms are called the *transfer algorithms* of the ABE. The logic design aspect is restricted to the internal realization of the transfer algorithm within the ABE. It will depend on the choice of the radix and of the encoding of digit values. The arithmetic design is the specification of ABE arrays for a given algorithm, which may range from addition of two operands to matrix inversion and to the evaluation of trigonometric, logarithmic, and similar functions [6], [7]. The arithmetic design is accomplished by a systematic procedure which is described in this paper. The arithmetic algorithm is expressed by a directed graph model which is translatable to an interconnection diagram for ABE's.

The procedure of arithmetic design is illustrated by the example of an ABE array for polynomial evaluation. A minimally redundant radix-16 SD number system ($\bar{9}$ to 9) is chosen for the specific illustrations [4]. The practical choice of radices for an ABE is limited to radix 10 and radices 2^k with $k \geq 2$. Radix 10 requires only 13 digit values ($\bar{6}$ to 6); thus storage requirements are not increased compared to the conventional form (0 to 9). Radices 2^k (i.e., 4, 8, 16, 32, etc.) require one additional bit of storage per digit when immediate reconversion is not performed. Their advantages are their compatibility with conventional binary input operands. Further, full utilization of available internal logic complexity is possible by choosing the largest acceptable value of k in $r=2^k$.

The choice of a specific radix is not necessary in the following description of the ABE transfer algorithms as long as the details of internal logic design are avoided.

II. The Arithmetic Building Element (ABE)

A set of five arithmetic building blocks called arithmetic microsystems was described in 1966 [4]. These building blocks employed SD number systems, and the examples were implemented in the minimally redundant radix-16 SD system. The number of building blocks was subsequently reduced from five to two, the universal building unit (UBU) and the reconversion and sign unit (RSU) [5]. The present paper describes a single building block, called the arithmetic building element (ABE). The reduction of the set to one element is attained at the cost of greater internal complexity and reduced speed of reconversion to conventional representation. New and more effective forms have been devised for most ABE transfer algorithms.

The ABE is implemented with combinational logic and storage elements for all output signals. The following description of the ABE presents a detailed statement of the algorithms which define its internal logic design. The description uses the following notation:

$\lfloor y \rfloor$ floor (integer $\lfloor y \rfloor \leq y < \lfloor y \rfloor + 1$)

$\lceil y \rceil$ ceiling (integer $\lceil y \rceil - 1 < y \leq \lceil y \rceil$)

r radix

a maximum SD digit magnitude ($\lfloor 1 + r/2 \rfloor \leq a \leq r-1$)

Arithmetic Operations:
- $-$ (overbar) additive inverse
- $+$ addition
- $-$ subtraction
- $\times$ (or adjacency) multiplication
- $/$ division

Logic Operations:
- $\vee$ OR
- $\barwedge$ NOR
- $\wedge$ AND
- $\barwedge$ NAND
- $\sim$ NOT
- $\neq$ EXCLUSIVE OR.

A. External Structure of the ABE

The external connections of the radix-r ABE are shown in Fig. 1. The *inputs* supply either control or data signals. There are four types of *control inputs* (individual logic variables):

1) Four Algorithm Lines: SS (simple sum), MS (multiple sum), PD (Product), and RS (Reconversion). One of these four lines is permanently set to the value 1 and determines the arithmetic transfer algorithm of the ABE.

2) Two Algorithm-Modifier Lines: CI (conventional input), and SF (significance arithmetic). When either one or both of these are set to the value 1, they modify the algorithm according to their specifications.

3) Two Output-Gating Lines: G1 and G0. The outputs of the ABE are held to zero by the logic condition $\sim G0 \vee G1 = 0$; otherwise the specified outputs are available.

4) m Sign Lines: C1, $\cdots$, Cj, $\cdots$, Cm, with $m \leq r+1$. The logic value 1 on the sign line Cj commands a change of sign (additive inverse algorithm) for the input digit appearing on the associated input digit line Dj. The sign change occurs before the algorithm is applied to the input digit.

The remaining input lines are *data inputs*.

5) m Digit Lines: D1, $\cdots$, Dj, $\cdots$, Dm, with $m \leq r+1$. Each digit line is a bundle of d bit lines. The d logic variables applied on one digit line represent the value of one radix-r input digit. The value of d depends on r and on the choice of encoding.

6) One Borrow-In Line: BI. This input is used to represent an incoming borrow for the reconversion (RS) from SD to conventional form.

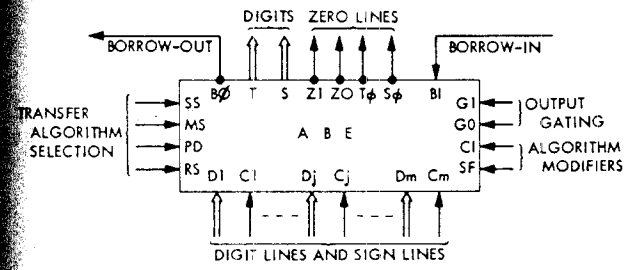


Fig. 1. Input and output lines of the arithmetic building element.

The *outputs* of the ABE represent the results computed by the specified algorithm. Two outputs (T and S represent digits; all others are individual logic variables. The following outputs are provided.

1) Two Output-Digit Lines: T and S. They represent the digits of the result and are also bundles of d bit lines each.

2) Two Zero Lines: Z1 and Z0. The line Z1 assumes the logic value 1 only if the result computed by the ABE has the numeric value zero. The line Z0 assumes the logic value of Z1.

3) Two Zero-Significance Lines: Tϕ and Sϕ. They assume the logic values 1 only if the input SD digits for the reconversion (RS) algorithm are nonsignificant ("space") zeros.

4) One Borrow-Out Line: B$\emptyset$. It is related to the BI input line and represents an outgoing borrow in the reconversion algorithm.

Additional and more specific discussion of the inputs and outputs will be found in the subsequent discussion of the ABE algorithms and their modifiers.

Our example employs $r=16$ with minimal redundancy ($a=9$). One digit line Dj receives five logic variables which represent the 19 digit values $\bar{9}$ to 9 inclusive. The maximum allowable number of digit lines is $m=17$.

B. Number Formats

Input operands and output results are positional, constant radix-r number forms:

$$x^1 = x^1_{-n} \cdots x^1_{-1} x^1_0 x^1_1 x^1_2 \cdots x^1_i \cdots x^1_n.$$

Either conventional (CONV) or signed-digit (SD) forms are employed. The allowed values of a digit x^1_i are $\{0, 1, \cdots, r-1\}$ for CONV forms, and $\{\bar{a}, \cdots, \bar{1}, 0, 1, \cdots, a\}$ for SD forms, with the constraint $r/2 < a < r$. Unless explicitly otherwise specified, SD forms with minimal redundancy ($a = \lfloor 1 + r/2 \rfloor$) are assumed.

Rightward indexing of digits is used unless otherwise stated:

$$i = \cdots, -n, -(n-1), \cdots, -2, -1, 0, 1, 2, \cdots, n, \cdots.$$

With this indexing convention, greater indices refer to less significant digit positions. Positive indices are used to indicate positions to the right of the radix point, while negative and zero indices identify digit positions to the left of the radix point.

When more than one operand is considered, the superscript is employed to identify a specific operand, i.e., x^1 and x^2 for two operands, or $x^1, x^2, \cdots, x^j, \cdots, x^m$ for m operands. The ith digit of x^j is uniquely identified as x^j_i.

C. The Additive-Inverse Algorithm ($Cj = 1$)

A sign line Cj is associated with every digit line Dj in the ABE (Fig. 1). When the digit x^j_i is the input on Dj, an associated *sign-change variable* c^j_i is applied on Cj. The logic value $c^j_i = 1$ causes the sign of the digit x^j_i to be changed (except when x^j_i is zero) to the opposite value before the selected ABE algorithm is applied to the input digits.

The sign-change variables c^j_i permit the specification of either the operand x^j or its additive inverse $-x^j$ as the input for any algorithm which is implemented by ABE's. For example, the simple-sum algorithm can form any one of the four sums:

$$x^1 + x^2;\ (-x^1) + x^2;\ x^1 + (-x^2);\ (-x^1) + (-x^2)$$

with appropriate application of the sign-change variables.

In the following discussion of the ABE algorithms the effect of c^j_i will be expressed in terms of the *arithmetic sign* variable σ^j_i which is defined as

$$\sigma^j_i \equiv (\sim c^j_i - c^j_i)$$

and assumes the values $+1$ and -1. The value of the modified input digit which is processed by the ABE algorithm is the product $\sigma^j_i x^j_i$. Unless explicitly otherwise stated, the same value of c^j is specified for the entire operand x^j. The value of an input operand is given as $\sigma^j x^j$ in the subsequent discussion, where σ^j is the *common sign* variable for the operand.

D. The Simple-Sum Algorithm ($SS = 1$)

The SS algorithm computes one SD sum digit s_i for the $n+1$-digit-long sum $s = \sigma^1 x^1 + \sigma^2 x^2$ of two n-digit SD operands.

1) Inputs: x^1_i, x^1_{i+1}, x^2_i, x^2_{i+1} on D1, D2, D3, D4, respectively; c^1 on C1 and C2; c^2 on C3 and C4.

2) Outputs: Sum digit s_i on S; zero-condition z_i, $\sim z_i$ on Z1, Z0, respectively ($z_i = 1$ if $s_i = 0$).

3) Algorithm: Define the digit sum K_h (for $h = i$ and $h = i+1$) as

$$\sigma^1_h x^1_h + \sigma^2_h x^2_h \equiv K_h.$$

Compute (for radix $r \geq 3$) the output digit s_i as follows:

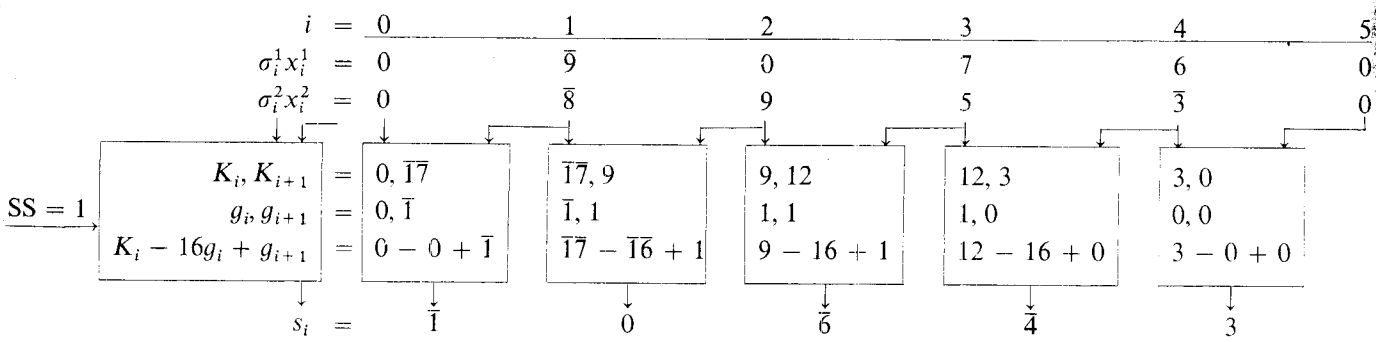
$$s_i = K_i - r g_i + g_{i+1}$$

where

$$g_h = \begin{cases} 0, & |K_h| < a \\ 1, & K_h \geq a \\ \bar{1}, & K_h \leq \bar{a}. \end{cases}$$

The operands x^1 and x^2 are extended by prefixing one zero digit at the left end of each. This extension is required to assure the correct $n+1$ digit sum s. A total of $n+1$ ABE's is used.

4) Example ($r = 16$, $a = 9$): $c^1 = c^2 = 0$; $x^1 = 0.\bar{9}076$; $x^2 = 0.\bar{8}95\bar{3}$ (internal values shown in decimal).



sum $s = \bar{1}.0\bar{6}\bar{4}3.$

E. The Multiple-Sum Algorithm ($MS = 1$)

The MS algorithm computes two SD digits s_i and t_{i-1}: one each for two n-digit words (s, t) which represent the sum of m words (n-digit SD operands) x^1 to x^m such that

$$s + t = \sum_{j=1}^{m} \sigma^j x^j, \quad \text{with} \quad m \leq r + 1.$$

1) *Inputs:* x_i^j on Dj and c^j on Cj for $j = 1, \cdots, m$.
2) *Outputs:* s_i on S and t_{i-1} on T; zero-condition variables z_i, $\sim z_i$ for the digit sum Σ_i (see below) on Z1, Z0.
3) *Algorithm:* Define the digit sum Σ_i and its functions (for the radix $r \geq 3$) as follows:

$$\Sigma_i \equiv \sum_{j=1}^{m} \sigma_i^j x_i^j$$

$$I(\Sigma_i) = \text{integer quotient } |\Sigma_i|/r$$

$$R(\Sigma_i) = \text{integer remainder } |\Sigma_i| - rI(\Sigma_i).$$

The output digits are computed as follows:

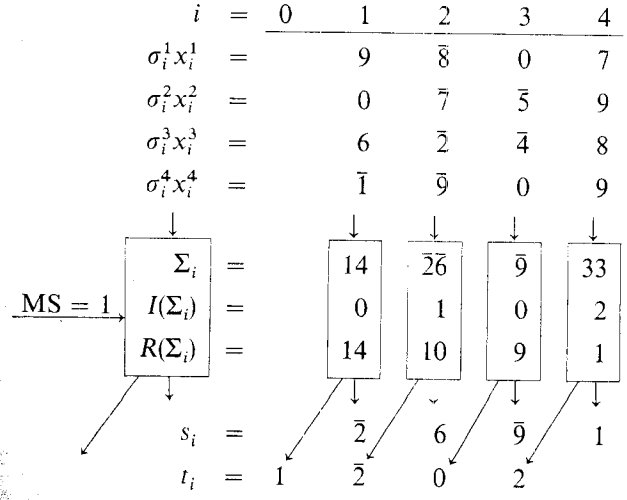
$$\text{sign}\,(t_{i-1}) = \text{sign}\,(\Sigma_i)$$

$$|t_{i-1}| = \begin{cases} I(\Sigma_i) + 1, & R(\Sigma_i) > a \\ I(\Sigma_i), & R(\Sigma_i) \leq a \end{cases}$$

$$s_i = \Sigma_i - rt_{i-1}.$$

The constraint $m \leq r+1$ assures that the output digits s_i and t_{i-1} do not exceed the allowed range of values of the input digits x_i^j. A total of n ABE's is used.

4) *Example* ($r = 16$, $a = 9$): $c^1 = c^2 = c^3 = c^4 = 0$; $x^1 = 0.9\bar{8}07$; $x^2 = 0.0\bar{7}\bar{5}9$; $x^3 = 0.6\bar{2}\bar{4}8$; $x^4 = 0.\bar{1}909$ (internal values shown in decimal).



sum $(s; t) = (0.\bar{2}6\bar{9}1;\ 1.\bar{2}02)$.

F. The Product Algorithm ($PD = 1$)

The PD algorithm computes one SD product digit s_{k+i}^k for the $n+1$-digit-long product $s^k = \sigma_k^1 x_k^1 \sigma^2 x^2$ of one SD digit x_k^1 and the n-digit SD operand x^2.

1) *Inputs:* x_k^1 on D1; x_i^2, x_{i+1}^2, x_{i+2}^2 on D2, D3, D4; c^1 on C1; c^2 on C2, C3, C4.
2) *Outputs:* Product digit s_{k+i}^k on S; zero-condition variables z_{k+i}^k, $\sim z_{k+i}^k$ (for s_{k+i}^k) on Z1, Z0.
3) *Algorithm:* Define three digit products π_{k+h} (for $h = i$, $i+1$, $i+2$) as follows:

$$\pi_{k+h} \equiv (\sigma_k^1 x_k^1) \times (\sigma_h^2 x_h^2).$$

Compute the output digit s_{k+i}^k in two steps. First, apply the MS algorithm to each π_{k+h} (substituting π_{k+h} for Σ_i, u_{k+h-1} for t_{i-1}, and v_{k+h} for s_i) to get the digits u_{k+h-1} and v_{k+h} such that

$$\begin{aligned} \pi_{k+i} &= ru_{k+i-1} + v_{k+i} \\ \pi_{k+i+1} &= ru_{k+i} + v_{k+i+1} \\ \pi_{k+i+2} &= ru_{k+i+1} + v_{k+i+2}. \end{aligned}$$

Second, perform the SS algorithm (substituting v_{k+i}, v_{k+i+1} for x_i^1, x_{i+1}^1, and u_{k+i}, u_{k+i+1} for x_i^2, x_{i+1}^2) to get the product digit:

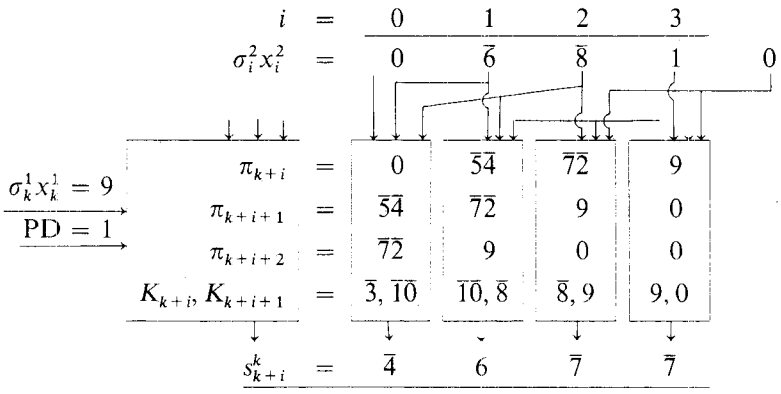
$$s_{k+i}^k = v_{k+i} + u_{k+i} - rg_{k+i} + g_{k+i+1}.$$

The n-digit operand x^2 is extended by prefixing one zero digit at its left end in order to yield the correct $n+1$ digit product. ABE count is $n+1$.

When the multiplier x^1 consists of more than one digit (say, g digits) we can simultaneously obtain all products $s^k = x_k^1 x^2$ ($k = 1, 2, \cdots, g$). The product $x^1 x^2$ is computed by summing the columns of product digits with the same index, using the MS algorithm followed by the SS algorithm. The total number of ABE's required to form the product $x_k^1 x^2$ is $gn+g$; therefore $g < n$ is the preferred choice when $g \neq n$. The maximum number of digits with the same index to be summed by the MS algorithm is $f = \min(g, n+1)$.

It is noted that the PD algorithm is internally more complex than the previously described product (PU) algorithm [5] which computed two output digits for every digit product $x_k^1 x_i^2$, but had $f = 2$ min (g, n).

4) *Example* ($r = 16$, $a = 9$): $c^1 = c^2 = 0$, $x_k^1 = 9$; $x^2 = 0.\overline{681}$ (internal values shown in decimal).



PD($i = 0$)

$\pi_k = 0 = (16)\ 0 + \overline{0}$

$\pi_{k+1} = \overline{54} = (16)\ \overline{3} + \overline{6}$

$\pi_{k+2} = \overline{72} = (16)\ \overline{4} + \overline{8}$

$K_k, K_{k+1} = \overline{3},\ \overline{10}$

$s^k_k = \overline{4}$

PD($i = 1$)

$\pi_{k+1} = \overline{54} = (16)\ \overline{3} + \boxed{\overline{6}}$

$\pi_{k+2} = \overline{72} = (16)\ \overline{4} + \boxed{\overline{8}}$

$\pi_{k+3} = 9 = (16)\ 0 + \boxed{9}$

$K_{k+1}, K_{k+2} = \overline{10},\ \overline{8}$

$s^k_{k+1} = 6$

PD($i = 2$)

$\pi_{k+2} = \overline{72} = (16)\ \overline{4} + \boxed{\overline{8}}$

$\pi_{k+3} = 9 = (16)\ 0 + \boxed{9}$

$\pi_{k+4} = 0 = (16)\ 0 + 0$

$K_{k+2}, K_{k+3} = \overline{8},\ 9$

$s^k_{k+2} = \overline{7}$

PD($i = 3$)

$\pi_{k+3} = 9 = (16)\ 0 + \boxed{9}$

$\pi_{k+4} = 0 = (16)\ 0 + \boxed{0}$

$\pi_{k+5} = 0 = (16)\ 0 + 0$

$K_{k+3}, K_{k+4} = 9,\ 0$

$s^k_{k+3} = \overline{7}$

product $s^k = (\overline{4}.6\ \overline{7}\ \overline{7}) \times 16^{-k}$.

G. *The Reconversion Algorithm (RS = 1)*

The RS algorithm receives SD digits x^1_i, x^1_{i+1} and reconverts them to conventional (CONV) digits y_i, y_{i+1} such that the CONV radix-r form y represents the same value as the SD radix-r $\sigma^1 x^1$. A borrow-propagation must take place because the CONV digits cannot assume negative values.

1) *Inputs:* SD digits x^1_i on D1, x^1_{i+1} on D2; c^1 on C1 and C2; input borrow b_{i+1} on BI (from the positions $i+2$, $i+3$ of x^1).

2) *Outputs:* CONV digits y_i on T, y_{i+1} on S; output borrow b_{i-1} on B$\emptyset$; zero variables z_i, $\sim z_i$(for $y_i, y_{i+1} = 0$) on Z1, Z0.

3) *Algorithm:* For $h = i+1$ and $h = i$ compute sequentially

$$b_{h-1} = \begin{cases} 0, & \sigma^1_h x^1_h - b_h \geq 0 \\ 1, & \sigma^1_h x^1_h - b_h < 0 \end{cases}$$

$$y_h = r b_{h-1} + \sigma^1_h x^1_h - b_h.$$

An output borrow $b_{-n-1} = 1$ from the ABE in the most significant position $i = -n$ will occur when the SD number $\sigma^1 x^1$ is negative. The relationship between the values of x^1 and y is therefore

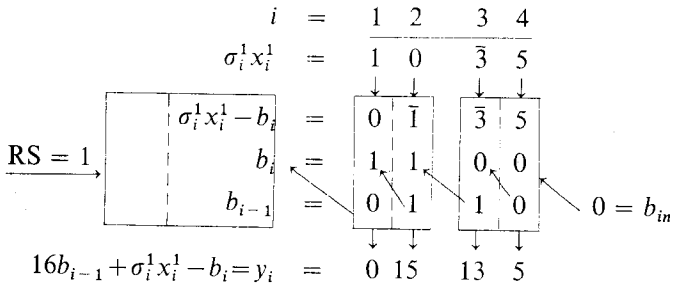
$$\sigma^1 x^1 = y - r^{n+1} b_{-n-1}.$$

The presence of the output borrow indicates that the result y represents the negative value $\sigma^1 x^1$ as the complement with respect to r^{n+1} (or "two's complement" when $r = 2^k$ is employed). Connecting the output borrow b_{-n-1} as the input ("end-around") borrow b_q into the ABE in the least significant position $i = q$ of x^1 will yield the complement with respect to $r^{n+1} - 1$ (or "one's complement" when $r = 2^k$) which is more readily converted to sign-and-magnitude representation of $\sigma^1 x^1$. Taking individual digitwise complements $y'_i = (r-1) - y_i$ will yield the magnitude y' of the negative number $\sigma^1 x^1$.

It is to be noted that the RS algorithm yields only two digits of y per ABE, while the previously described reconversion-and-sign unit (RSU) reconverted at least $m/2$ digits [5]. The number of output wires is the limiting factor in the ABE. An internal arrangement which would utilize some of the ABE input digit lines for CONV digit outputs during the RS algorithm would approximate the RSU capability.

The RS algorithm may be applied as a modifier for the SS and PD algorithms. The BI and B$\emptyset$ lines must be connected according to the RS specification. The output is the CONV form of the SS sum s or the PD product s^k.

4) *Example* ($r = 16$, $a = 9$): $c^1 = 0$; $x^1 = 0.10\overline{35}$.



H. Conventional-Input Modification ($CI=1$)

The similarity between CONV and SD forms is exploited by the CI modification which allows conventional digits ($0 \le x_i^j \le r-1$) as inputs to the ABE and thus provides a convenient conversion from CONV to SD forms. The mixing of SD and CONV-input operands applies to SS, MS, and PD algorithms.

1) *CONV-Input Simple Sum* ($SS=1$ and $CI=1$): The input operand x^1 is a CONV form (sign-and-magnitude). The input $c^1=1$ is applied when the sign of x^1 is minus. The digit values x_i^1, x_{i+1}^1 are 0 to $r-1$. The SS algorithm remains unchanged, since the maximum digit sum has the value

$$|K_h|_{\max} = (r-1) + a = r + (a-1)$$

which assures that $|g_h| \le 1$ and $|s_{ij}| \le a$ will hold.

2) *CONV-Input Product* ($PD=1$ *and* $CI=1$): The input digit x_k^1 is a CONV form digit ($0 \le x_k^1 \le r-1$). Input C1 receives $c^1=1$ when the sign of x^1 is minus. The PD algorithm remains unchanged, since the maximum digit product has the value

$$|\pi_{k+h}|_{\max} = a(r-1) = r(a-1) + (r-a)$$

which assures that $|u_{k+h-1}| \le a-1$ and $|v_{k+h}| \le a$ will hold.

3) *CONV-Input Multiple Sum* ($MS=1$ *and* $CI=1$): The digit sum Σ_i of the MS algorithm cannot exceed the limit

$$|\Sigma_i| \le ra + a \quad \text{or} \quad m \le r+1$$

in order to assume that $|t_{i-1}| \le a$ and $|s_i| \le a$ will be satisfied and the result will be SD forms. The maximum number of CONV input digits which satisfies the same requirement is m' such that $m'(r-1) \le (r+1)a$, or

$$m' \le \lfloor a(r+1)/(r-1) \rfloor.$$

In order to allow the mixing of CONV and SD input operands, we specify the following rule for MS=1 and CI=1:

for every pair of digit lines Dj, D($j+1$) with j odd ($j=1, 3, 5, \cdots$), either x_i^j is a CONV digit and $x_i^{j+1}=0$, or x_i^j and x_i^{j+1} are both SD digits.

The existence of the CI modification eliminates the need for special conversion algorithms. SD forms with radices 2^k ($k=2, 3, 4, \cdots$) are directly compatible with conventional binary number representations, since binary input operands are accepted by radix 2^k ABE's in the CI mode, and the RS algorithm returns results in binary form.

I. Significance-Arithmetic Modification ($SF=1$)

This optional feature is applicable to the SS, MS, PD, and RS algorithms and is compatible with the CI modification. The algorithms are altered to the significant-digit mode [14] by SF=1. A special "space-zero" digit value ϕ is used to identify nonsignificant positions in SD operands.

In the reconversion (RS) algorithm, input digit values $x_{i+1}^1=\phi$ and $x_i^1=\phi$ cause the outputs $y_{i+1}=0$, $S\phi=1$ and $y_i=0$, $T\phi=1$, respectively. The logic 1 outputs on $S\phi$ and $T\phi$ explicitly identify the zero value outputs on S and T to be nonsignificant positions in the conventional representation.

Details of the SF-mode algorithms have been described previously [2]–[4], [7]. The general rule requires a "space-zero" output value ϕ whenever at least one input digit in the same position has the value ϕ.

III. A Graph Model for Arithmetic Processor Design

A single ABE can serve as a digit-serial arithmetic processor. Arrays of ABE's can be arranged for the pipelined hardware implementation of an entire class of functions [6]. The arrangement of ABE's into such arrays can be called *arithmetic design*, in contrast to the term "logic design," which is needed only within the ABE. Arithmetic design is the process of converting an arithmetic expression to an interconnection diagram of ABE's. The present section presents graph models for algorithms and ABE arrays which make a systematic arithmetic design procedure possible. The procedure has been programmed for computer-aided design of large arithmetic processors.

A *combinational*[1] *arithmetic* (CA) *net* is an acyclically interconnected array of ABE's. Such a net can be described by a directed graph. The graphical representation of CA nets exists in two levels, the algorithm level and the hardware level. At the algorithm level, the CA net is described in terms of a loop-free directed graph with vertices representing fundamental arithmetic, logical, and test operations. At the hardware level the CA net is also described by a loop-free directed graph, but the vertices are specified in terms of the functions performed by the ABE's.

In the description of CA nets, radix-r fixed-point numbers are defined to be of the form

$$x = (x_{-p+1}x_{-p+2}\cdots x_{-1}x_0 . x_1x_2\cdots x_q)$$

and their format is described by the pair of nonnegative integers $(p . q)$, in which p is the length (in digits) of the integer part of x, and q is the length of the fraction part of x.

[1] The word "combinational" is used here to denote the absence of feedback loops within the CA net. Arithmetic nets with closed loops are called "iterative."

TABLE I
H-LEVEL VERTICES

Name and Notation	Symbol	Inputs (Not Modified)	Numeric Output Result	Multiplicity μ of Output Arc
Simple Sum: SS[p . q; h]	⊕	2 SD summands $\pm x^1$, $\pm x^2$; n digits each	single SD word s; $p+q=n+1$ digits	$\mu=1$
Multiple Sum: MS[m; p . q; h]	⊛	m SD summands $\pm x^1, \cdots, \pm x^m$; n digits each	double SD word s, t; $p+q=n+1$ digits	$\mu=2$
Product: PD[p . q; h]	⊛	SD multiplier $\pm x^1$, g digits; SD multiplicand $\pm x^2$, n digits	f-tuple SD word $s^1, s^2, \cdots, s^g$; $p+q=n+g$ digits	$\mu=f$; $f=\min(g, n+1)$
Reconversion: RS[p . q; h]	(RS)	SD word $\pm x^1$; n digits	CONV word y; $p+q=n$; output borrow b_{out}	$\mu=1$
Storage: S[p . q; h]	Ⓢ	SD word x^1; n digits	the input word	$\mu=1$

A vertex in either the algorithm or the hardware level graph is identified by the notation

$$\mathrm{OP}\,[m; p\,.\,q; h]$$

where

OP operation symbol of the vertex
m number of vertices providing inputs
$p\,.\,q$ format of the result of the vertex
h number of vertices receiving the output.

The parameter m may be omitted when its value is defined by the operation symbol. All input operands are signed-digit numbers except when conventional numbers are explicitly specified. Conventional form results are obtained only from reconversion (RS) vertex.

An *arc* represents either a numeric result or a logic (two-valued) variable which is generated by its source vertex. The receiving vertex is identified by the arrowhead of the arc. The transfer of information is unidirectional.

More specific properties of vertices and arcs are presented in the following discussion on the two levels of description.

A. The Hardware Level

The hardware-level (H-level) graph describes the algorithm which is carried out by the CA net in terms of ABE arrays and their interconnections. Five different types of H-level vertices are used in an H-level graph. They are described in detail in Table I. Each H-level vertex (H vertex) represents an array of ABE's, except for the storage vertex S which represents a register.

The operation symbol of an H vertex contains the following information: the ABE algorithm, the additive inverse input specifications, and the algorithm modifiers. The additive inverse is specified for any input operand x^j by listing its superscript j with an overbar in parentheses following the ABE algorithm symbol. For example SS($\bar{1}$) indicates the sum $(-x^1)+x^2$; PF($\bar{2}$) indicates the product $x^1(-x^2)$; and MS($\bar{1}$, $\bar{4}$) [5; $p\,.\,q$; h] indicates the sum $(-x^1)+x^2+x^3+(-x^4)-x^5$. On the graph, a solid dot at the arrowhead of an arc indicates the additive inverse of the input operand on this arc.

The algorithm modifiers are listed following the ABE algorithm symbol, separated by "slant" symbols. For example,

$$\mathrm{MS}(\bar{1}, \bar{4})/\mathrm{CI}(5)[5; p\,.\,q; h]$$

modifies the previous MS example by specifying that x^5 is a conventional-form operand. The storage vertex S/CI [$p\,.\,q$; h] indicates a CONV form operand, and SS($\bar{2}$)/SF [$p\,.\,q$; h] indicates a significance-mode addition $x^1+(-x^2)$.

An H-level arc (H arc) represents an output result (numeric or logic). An identifier *tag* is attached to an arc. For *numeric* results the tag identifies arithmetic shifts of k positions: $k\uparrow$ for left, and $k\downarrow$ for right shift. It also specifies the extraction of certain digits of the result, and identifies the receiving input line. For example, the tag

$$2\uparrow (s_1 s_2 s_3)|\mathrm{D2}$$

identifies that the output digits $s_1 s_2 s_3$ are extracted from the sum S and are supplied to the D2 input lines of the receiving H-vertex ABE's with a left shift of two positions. Omission of the digit extraction information means that the entire result is transmitted; zero shifts are also not shown.

An arc which represents a numeric result has an associated *multiplicity* μ. The value of μ is a function of the source vertex as shown in Table I, in which the outputs are single ($\mu=1$), double ($\mu=2$) or f-tuple ($\mu=f$) words.

Logic results are obtained from the ABE output lines Z1, Z0, Tϕ, Sϕ, B$\emptyset$. When an H arc represents a single logic variable, the tag contains the variable name, the index of the source ABE, and the receiving input line. For example, Z1(2)|G1 indicates that the Z1 output of the ABE in posi-

TABLE II
A-LEVEL VERTICES

Operation	Notation	Symbol	Inputs	Output	*H* Vertices Needed
Addition	$\sum[m; p \cdot q; h]$	Σ	m operands	sum	SS, MS
Multiplication	$\Pi[p \cdot q; h]$	Π	2 operands	product	SS, MS, PD
Reconversion	$RS[p \cdot q; h]$	RS	SD number	CONV number	RS
Storage	$S[p \cdot q; h]$	S	1 operand	input word	S
Logic Op. #	$L\#[k; h]$	#	k operands	1 logic word	SS, MS
Test Op. X	$TX[k; h]$	TX	k operands	1 logic variable	SS, MS, RS
Gating	$G[p \cdot q; h]$	G	1 operand gate variable G	input word if G = TRUE	SS, or S
Merging	$M[m; p \cdot q; h]$	M	m operands (only one nonzero)	1 word (nonzero input)	SS, MS

tion $i=2$ of the source vertex is conveyed to every G1 input of the receiving vertex. A logic variable (TRUE or FALSE) input to a digit line is received as the digit value 1 or 0, respectively.

A fully specified *H*-level graph provides a complete specification for the wiring diagram of the CA net.

B. The Algorithm Level

The purpose of the algorithm-level (*A*-level) graph is to describe the algorithm which is to be computed by the CA net in terms of elementary arithmetic, logic, transfer, test, selection, and storage operations. The *H*-level description is also acyclic and corresponds to a sequence of typical program (or microprogram) steps, which are identified as vertices on the *A*-level graph.

The principal constraint which must be satisfied by the *A*-level graph is that every *A* vertex should have an equivalent *H*-level vertex or graph. The conversion from *A* level to *H* level is carried out by direct replacement. The resulting *H*-level graph frequently may be simplified by *H*-vertex merging, which is discussed in a subsequent section.

The extent of the *A*-level vertex set is optional with the designer and depends on the complexity of the algorithms which are to be implemented. A typical set of *A* vertices is listed in Table II. The translations of these vertices to the *H* level are described in following sections.

The notation for *A* vertices has the same format as for *H* vertices, but different operation symbols are used. The additive inverse and algorithm modifier specifications, where applicable, follow the *H*-level rules.

The *A*-level vertices of Table II fall into five categories. The *arithmetic vertices* Σ, Π, RS (with additive inverse included in each) are used for the implementation of arithmetic. The *storage* vertex designates storage registers for initial conditions. The *logic operation* vertex L# is a generic vertex in which # represents any one of the logic operations (NOT, AND, OR, NAND, NOR, EXCLUSIVE OR, EQUIVALENCE, etc.). The *test operation* vertex TX is also generic. The symbol X represents the test; for example TP, TZ, and TN stand for test-positive, test-zero, and test-negative, respectively. The *selection operations* are implemented by the use of the gating vertex G and the merging vertex M. These vertices (as well as the logic vertices L#) represent auxiliary uses of the ABE's which may be performed by other types of elements.

A-level arcs have the same meaning as the *H*-level arcs. The *A*-level tag specifies a scale factor of $r^{\pm k}$ instead of an arithmetic shift. Digit extraction remains explicitly specified. The logic (control) signals on the *A*-level arcs have the values TRUE and FALSE. The multiplicity of the *A*-level arcs is one, since the representations of the results are not considered at the *A* level.

C. Elementary Graphs for Arithmetic Operations

The elementary *A*-level graphs for addition (Σ) and multiplication (Π) are shown in Fig. 2. The *H*-level implementation of the *A*-level addition graph is shown in Fig. 3. Three cases are distinguished:

1) for $n=2$ operands, an SS vertex replaces the Σ vertex [Fig. 3(a)];

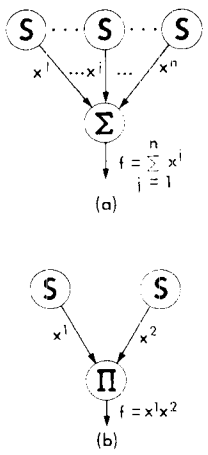


Fig. 2. *A*-level graphs for (a) addition and (b) multiplication with n operands.

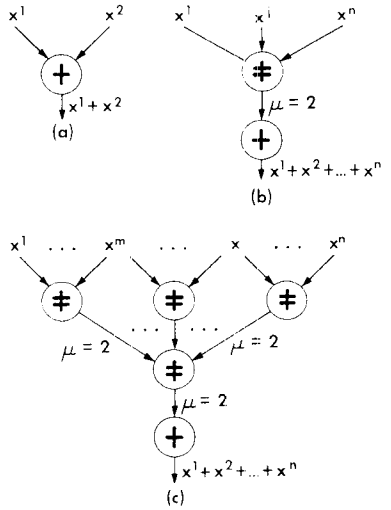


Fig. 3. *H*-level implementation of addition. (a) $n=2$. (b) $2<n\leq m$. (c) $m<n\leq m\lfloor m/2\rfloor$.

2) for $2<n\leq m$ operands, an MS vertex followed by an SS vertex replaces the Σ vertex [Fig. 3(b)];
3) for $m<n\leq m\lfloor m/2\rfloor$ operands, a two-level cascade of MS vertices followed by an SS vertex replaces the Σ vertex [Fig. 3(c)].

The *H*-level implementation of the *A*-level multiplication graph is shown in Fig. 4. The structure of the *H*-level graph depends on the length (q digits) of the shorter operand x^1. Two cases are shown in Fig. 4:

1) for $q\leq m$, the Π vertex is replaced by a PD vertex followed by the graph of Fig. 3(b) [Fig. 4(a)];
2) for $m\leq q\leq m\lfloor m/2\rfloor$, the PD vertex must be followed by the graph of Fig. 3(c) [Fig. 4(b)].

Case 1) above should suffice for most multiplications. For example, a radix-16 multiplier x^1 can have up to $m_{max}=17$ digits (or 68 bits) without requiring the graph of Fig. 4(b).

The conversion vertex RS and the storage vertex S remain unchanged in the translation from the *A* level to the *H* level. The S vertex is omitted in the graphs of Figs. 3–7, 10, and 12.

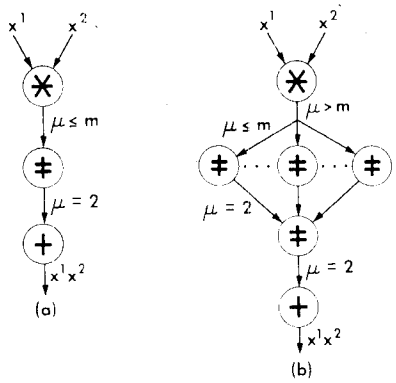


Fig. 4. *H*-level implementation of multiplication. (a) $q\leq m$. (b) $m<q\leq m\lfloor m/2\rfloor$.

D. Graphs for Logic, Test, and Selection Operations

In addition to the arithmetic functions, the *A*-level graphs may contain vertices which represent logic, test, and selection operations. In this section it is shown how these operations can be implemented at the *H* level using only the defined algorithms of the ABE.

1) *In the Implementation of Logic Operations:* The logic values (TRUE, FALSE) are defined in terms of numeric digit values (1, 0):

$$\text{TRUE} \equiv 1, \text{ and FALSE} \equiv 0.$$

The digit values representing k logic variables are designated $w_1, \cdots, w_k$ and are applied to the input digit lines D1, ···, Dk. The output is either z (on line Z1) or $\sim z$ (on line Z0).

The implementations of the logic operations NOT, AND, OR, NAND, NOR, EQUIVALENCE, and EXCLUSIVE OR by the ABE are as follows (using the SS or MS algorithm):

NOT: $\sim w_1 = z(w_1 + 0)$, with $k = 1$.

AND: $w_1 \wedge w_2 \wedge \cdots \wedge w_k = z\left(\left(\sum_{i=1}^{k} w_i\right) - k\right)$, with $k \leq m-2$ (two inputs accept the constant input value k).

OR: $w_1 \vee w_2 \vee \cdots \vee w_k = \sim z\left(\sum_{i=1}^{k} w_i\right)$, with $k \leq m$.

NAND: $\sim (w_1 \wedge w_2 \wedge \cdots \wedge w_k) = \sim z\left(\left(\sum_{i=1}^{k} w_i\right) - k\right)$, with $k \leq m-2$.

NOR: $\sim (w_1 \vee w_2 \vee \cdots \vee w_k) = z\left(\sum_{i=1}^{k} w_i\right)$, with $k \leq m$.

EQUIVALENCE: $\sim (w_1 \neq w_2) = z(w_1 - w_2)$, with $k = 2$.

EXCLUSIVE OR: $(w_1 \neq w_2) = z(w_1 + w_2 - 1)$, with $k = 2$.

The notation $z(f(w_i))$ indicates the z output (on Z1) of an ABE which computes $f(w_i)$.

Fig. 5 illustrates both *A*-level and *H*-level graphs for the logic function $f = \sim [(a \wedge \sim b) \vee c]$. The *H*-level vertices require one ABE per digit of the input word or words; the output is a logic word of the same length.

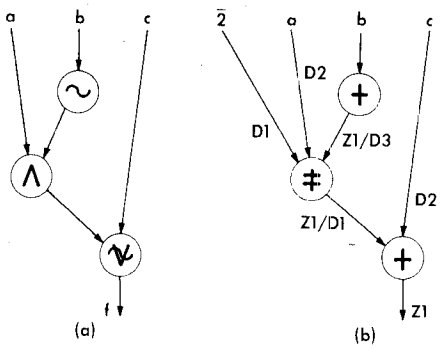


Fig. 5. Graphs for the logic function $f = \sim[(a \wedge \sim b) \vee c]$. (a) A level. (b) H level.

While it is interesting to note that logic operations can be performed using ABE's, such use is generally not considered practical because of the relatively high complexity and cost of an ABE.

2) Test Operations: Test operations at the H level refer to the test of an arithmetic result: the sum Σx^j of two or more ($k \geq 2$) numbers. The result is a pair of logic variables TRUE and FALSE (FALSE $\equiv \sim$TRUE), where TRUE $=1$ when the test is satisfied, and TRUE $=0$ if it is not. Test-positive (TP) is satisfied for $\Sigma x^j \geq 0$, test-zero (TZ) for $\Sigma x^j = 0$, and test-negative (TN) for $\Sigma x^j < 0$.

At the H level the TX vertices are implemented by SS, MS, and RS algorithms. Fig. 6 illustrates the test for $x = a$ which is implemented as TZ for $x - a = 0$ at both levels. At the H level the $\sim z_i$ outputs for all digits of $x - a$ are summed in an MS vertex, which has $z = 0$ if all $\sim z_i = 0$ at its inputs. Fig. 7 shows the test for $x < a$, implemented as TN for $x - a < 0$ at both levels. The negative sign of $x - a$ is obtained by the RS vertex which has the borrow $b_{out} = 1$ from its most significant position when the result is negative. The sign of an SD number is the same as the sign of its most significant nonzero digit; therefore leading zeros may conceal the sign of $x - a$. This possibility makes a reconversion to CONV form necessary.

More complex tests are implemented as combinations of the elementary tests. For example, the test for $a \leq x \leq b$ is implemented as the logic AND of two TN tests:

$$f = [(a - x) < 0] \wedge [(x - b) < 0].$$

A more detailed discussion of testing is presented in [7].

3) Selection Operations: The gating (G) and merging (M) H-level vertices provide the equivalents of selective gating (AND–OR) operations in conventional logic circuitry. They are included to demonstrate the extent of the functions provided by CA nets. Their use is not practical when simpler logic circuits are available for this purpose.

An illustration of the use of G and M vertices is shown in Fig. 8. The function of the CA net is as follows:

$$f = \begin{cases} x, & x \geq y \\ y, & x < y. \end{cases}$$

The gating function (replacing AND) is implemented by SS vertices (S can also be used). The merging function (replacing OR) uses an SS vertex; an MS vertex is needed for more than two inputs.

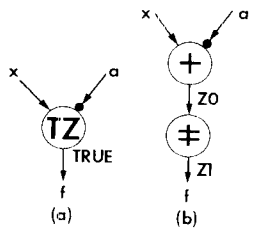


Fig. 6. Graphs for the zero test $x - a = 0$. (a) A level. (b) H level.

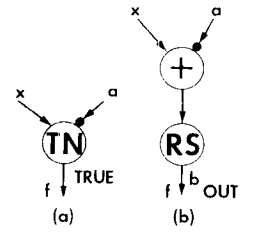


Fig. 7. Graphs for the negative test $x - a < 0$. (a) A level. (b) H level.

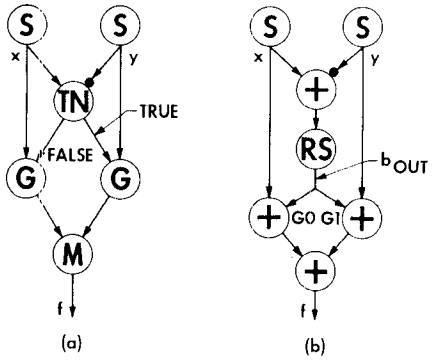


Fig. 8. Graphs for the selection: $f = x$ if $x \geq y$; $f = y$ if $x < y$. (a) A level, (b) H level.

E. Iterative Arithmetic Nets

The acyclic nature of the CA net may require too many ABE's to implement the net at an acceptable cost. In this case a smaller CA net can be designed and used iteratively to compute the same function. Such an iterative arithmetic (IA) net requires fewer ABE's and more time than the CA net which computes the same function in one pass through the net.

The CA net provides a reference point for the design of a processor. It requires the greatest number of ABE's and has the lowest computing time. An entire set of IA net designs usually can be derived from the CA net; these IA nets have gradually decreasing cost and increasing time requirements. The IA net with the highest acceptable computing time will yield the least costly design. The graph models of the original CA net explicitly display the regularities of the CA net structure and therefore facilitate the IA net design.

Illustrations of IA nets are provided by two nets for the implementation of division [4], [16]. A further example of an IA net is found in the design of an ABE-array arithmetic processor which computes a large class of functions [6]. The same processor also employs pipelining of CA nets in

der to achieve the highest possible rate of utilization. The orage registers at the output of the ABE are provided in der to make pipelining of CA nets feasible.

IV. CA Nets for the Evaluation of Polynomials

A. The Algorithm and Its Graphs

In this section we consider the application of CA nets to olynomial evaluation. A common method of approximating a given function is to employ polynomial approximation [8], [9]. The application of CA nets to polynomial valuation will serve as an illustration of their potential pplication. A method proposed by Estrin [10] permits the astest evaluation of explicit polynomials when sufficient arallel computing capacity is provided. The method to ompute the nth-degree polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

requires the computation of the terms $C_i^{(1)}$

$$C_i^{(1)} = a_i + x a_{i+1}, \quad i = 0, 2, \cdots, 2\lfloor n/2 \rfloor$$

followed by successive computations of terms $C_i^{(j)}$ for $j = 2, \cdots, m$:

$$C_i^{(2)} = C_i^{(1)} + x^2 C_{i+2}^{(1)}, \quad i = 0, 4, \cdots, 4\lfloor n/4 \rfloor$$
$$\vdots$$
$$C_i^{(m)} = C_i^{(m-1)} + x^{2^{m-1}} C_{i+2^{m-1}}^{(m-1)}, \quad i = 0, 2^m, \cdots, 2^m\lfloor n/2^m \rfloor.$$

This process will terminate when

$$m = \lfloor \log_2 n \rfloor + 1$$
$$p(x) = C_0^{(m)}.$$

The procedure corresponds to the factoring:

$$\begin{aligned} p(x) = {} & a_0 + a_1 x + x^2(a_2 + a_3 x) \\ & + x^4[a_4 + a_5 x + x^2(a_6 + a_7 x)] \\ & + x^8[a_8 + a_9 x + x^2(a_{10} + a_{11} x) \\ & + x^4(a_{12} + a_{13} x + x^2(a_{14} + a_{15} x))] + \cdots. \end{aligned}$$

We notice that for each j, all $C_i^{(j)}$ may be computed simultaneously in one multiplication time plus one addition time. Therefore, the minimum time to compute $p(x)$ using this algorithm is

$$\tau = \lceil \log_2 n \rceil + 1 \text{ multiplication–addition times.}$$

It is necessary to have a sufficient number of arithmetic units to compute $C_i^{(j)}$ simultaneously for any j in order to achieve this minimum time. This requirement can be met by a CA net in which each vertex performs addition and multiplication.

A simple example illustrating how CA nets are organized to evaluate third-degree polynomials using Estrin's method is shown in Fig. 9. Here the evaluation of a third-degree polynomial at the A level takes $\lfloor \log_2 3 \rfloor + 1 = 2$ units of time required by a pair of Π–Σ vertices. The speed at the H level requires further information. The delay through one vertex in the H-level CA net is the delay through one ABE. The total delay through the H-level CA net is equal to the number of vertices along the longest information flow path of the net. The configuration and complexity of the H-level CA net depend not only on the topology of the A-level CA net, but also on the precision of the data word.

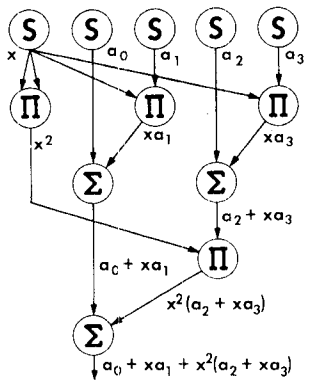


Fig. 9. A-level graph for the evaluation of polynomials (degree 3).

The cost of the CA net is defined as the count of ABE's in the net. It is a function of the degree of the polynomial and of the precision of the data word. Error in a CA net comes only from round-off operations if all input data are assumed to be free from inherent errors. Precision controls the size of the round-off errors. Since precision thus affects the speed, cost, and error of the CA net, it will be used, together with the degree n of the polynomial, as the independent variable in the study of speed, cost, and error of the CA net.

B. Complexity and Speed of the CA Net

In an A-level CA net for evaluating the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

the number of storage (S) vertices $N(\mathrm{S})$ is

$$N(\mathrm{S}) = n + 2$$

because there are $n+1$ coefficients a_i, $i = 0, 1, 2, \cdots, n$, and the independent variable x.

The number of multiplication vertices in the net is

$$N(\Pi) = n + \lfloor \log_2 n \rfloor$$

and the number of addition vertices is

$$N(\Sigma) = n.$$

The speed of this A-level CA net is

$$\tau = \lfloor \log_2 n \rfloor + 1 \ \Pi\Sigma \text{ time units}$$

where one $\Pi\Sigma$ time unit is the delay through one Π–Σ pair of vertices.

The H-level implementation of the A-level vertex Π is shown in Fig. 4. One PD vertex and one SS vertex are needed, i.e.,

$$N(\mathrm{PD}) = N(\mathrm{SS}) = N(\Pi).$$

The count of the MS vertices depends on the precision of the operands. Given the precision $q = \min(g, n+1)$ according to Table I, there are two cases:

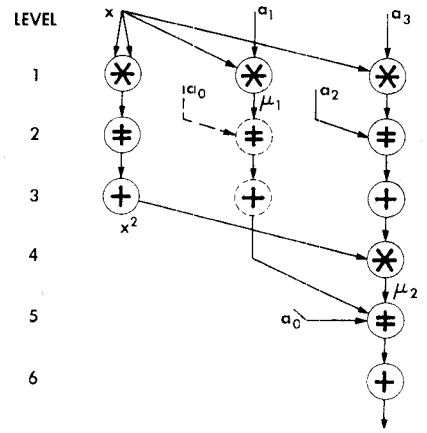


Fig. 10. H-level translation of the A-level graph of Fig. 9.

1) a single MS vertex is sufficient when $q \leq m$;
2) a two-level cascade of MS vertices is required for $m < q \leq m\lfloor m/2 \rfloor$.

The one-level cascade is sufficient for most cases. For instance, in the radix 16 and $m = 17$ example a precision of 17 radix-16 digits can be accommodated. The number of MS vertices in the first level of the cascade of case 2) varies from 1 to $\lfloor m/2 \rfloor$ as q increases, and is given by

$$K = \lceil (q - m)/(m - 2) \rceil.$$

The second level always contains one MS vertex.

It must be noted that in the formation of the term $a_i + a_{i+1}x$, the summation in the MS vertex includes the single word a_i along with the result word of the PD vertex. In this case the preceding results apply when q is taken to be the sum of the multiplicities of the product $a_{i+1}x$ and the word a_i. A direct translation of the A-level net to the H level is shown in Fig. 10 (including the dotted-line vertices) for $q \leq m$.

The detailed count of ABE's in the SS, MS, and PD vertices has been discussed in the algorithm description of Section II.

C. Computer-Aided CA Net Design

The graph of a CA net displays its structure and is a convenient means to design a CA net for relatively simple problems. For more complex problems, however, the graphical approach becomes tedious, especially when many designs are needed for purposes of evaluation. A programming effort was undertaken to demonstrate that a graphical CA net may be represented by tables acceptable to conventional computers, and that nontrivial operations on these tables can be performed by the program. The program is so far limited to evaluating polynomials up to 15th degree with Estrin's method. It is written in FORTRAN IV for running on the SDS Sigma 7 computer of the Digital Technology Research group at UCLA.

The topological relations among the vertices and arcs of a graph can be described by a family of matrices (connection matrices, precedence matrices, etc.) or threaded lists. The list representation was adopted for the following reasons. The description of vertices and arcs in a CA net consists of many parameters, and it is expected that, as problems get more complicated, more attributes will need to be added. Consequently, greater numbers of matrices are required to fully describe a CA net; this may result in an overflow of memory capacity. On the other hand, a threaded list is much less storage consuming, even though searching through the threaded list may be slower.

The program consists of four parts:

1) generation of CA-15 net for evaluating

$$p(x) = \sum_{j=0}^{15} a_j x^j \text{ at the } A \text{ level,}$$

2) generation of CA-i net for evaluating

$$p(x) = \sum_{j=0}^{i} a_j x^j \text{ with } i \leq 15$$

at the A level from the CA-15 net;
3) translation of CA-i net from A level to H level;
4) simplification of the H-level CA-i net.

The first step is to generate tables fully describing the topological configuration of a CA net for evaluating a 15th degree polynomial at the A level. The graph shown in Fig. 11 is translated to a tabular form. With these tables as a base, the tables describing CA nets at the A level for polynomials up to degree 15 are obtained through a process of elimination. A direct-replacement translation from the A level to the H level then takes place to provide more information about cost, speed, and accuracy. (Figs. 3 and 4 show parts of the translation process.) Redundancy may exist in the H-level tables thus obtained, because the translation is not simplified. A final simplification step eliminates redundancies in the H-level CA net.

An example of simplification is given in Fig. 12. A typical portion of the CA net of Fig. 11 is shown in Fig. 12(a). A direct translation would result in an H-level CA net shown in Fig. 12(b). If the output multiplicity of the PD(*) vertex does not exceed $m-1$, the input capacity of the MS(+) vertex is not filled. Consequently, the second SS(+) vertex can be removed by connecting its input to the multiple-sum vertex. Fig. 12(c) is the simplified form of the H-level net.

Another example of simplification is shown in Fig. 10. Translation to the H level by direct replacement will yield the graph including the dashed-line vertices at levels 2 and 3. The SS vertex at level 3 is removed if one unused input exists in the level-5 MS vertex. Furthermore, the dashed-line MS vertex at level 2 is removed if $\mu_1 + \mu_2 + 1 \leq m$ holds at the level-5 MS vertex.

A detailed description of the implementation of the CA-net design program is presented in [7]. The existence of the CA-net design program allows the designer to try several designs with differing choices of precision and with differing values of the polynomial order. The program tabulates the cost (in ABE's) and the speed [in ABE delays] for every design. The study of tradeoffs between speed, cost, precision, and error of approximation becomes feasible even for large CA nets and for numerous changes in these parameters.

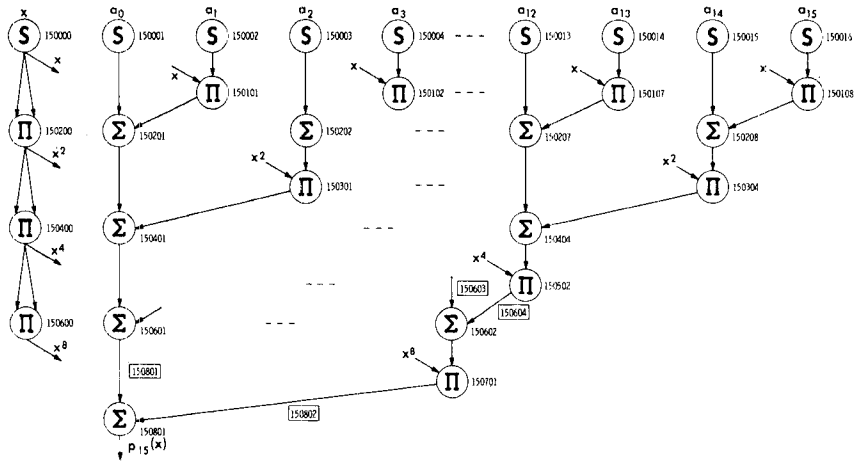


Fig. 11. The A-level CA-15 net (partial graph).

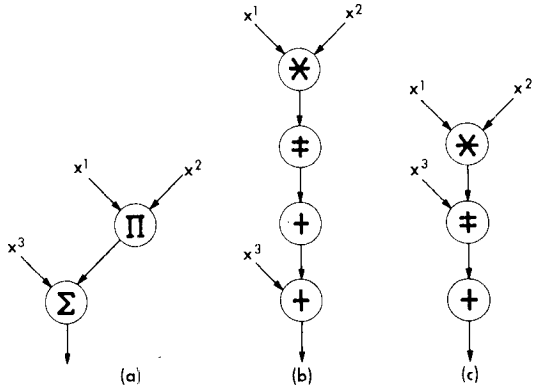


Fig. 12. Simplification by vertex merging at the H level. (a) A level. (b) H level, direct. (c) H level, simplified.

V. Current Work

The study of arithmetic building blocks has been stimulated by the Variable Structure Computer research at the Computer Science Department of UCLA and is being conducted as part of this research project [10], [11]. The ABE is intended to serve as a standard inventory item of the variable structure resources.

Work is currently progressing in three directions: the design of a pipelined function generator to replace arithmetical subroutines [6], the study of the computational complexity aspects of signed-digit arithmetic [15], and the logic design of a radix-16 ABE for LSI implementation.

The present paper omits the logic design aspects of the ABE in order to avoid excessive length. The design follows the approach of [4] and [7] and will be reported in a separate presentation.

References

[1] A. Avižienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electronic Computers*, vol. EC-10, pp. 389–400, September 1961.

[2] ——, "On a flexible implementation of digital computer arithmetic," *Information Processing 1962*, C. M. Popplewell, Ed. Amsterdam: North-Holland, 1963, pp. 664–670.

[3] ——, "Binary-compatible signed-digit arithmetic," *1964 Fall Joint Computer Conf., AFIPS Proc.*, vol. 26, pt. 1, Baltimore, Md.: Spartan, 1964, pp. 663–672.

[4] ——, "Arithmetic microsystems for the synthesis of function generators," *Proc. IEEE*, vol. 54, pp. 1910–1919, December 1966.

[5] A. Avižienis and C. Tung, "Design of combinational arithmetic nets," *Dig. 1st Ann. IEEE Computer Conf.* (Chicago, Ill.), pp. 25–28, September 6–8, 1967.

[6] C. Tung and A. Avižienis, "Combinational arithmetic systems for the approximation of functions." *1970 Spring Joint Computer Conf., AFIPS Proc.*, vol. 36. Washington, D. C.: Spartan, 1970, pp. 95–107.

[7] C. Tung,"A combinational arithmetic function generation system," Ph.D. dissertation, University of California, Los Angeles, Engrg. Rept. 68-29, June 1968.

[8] C. W. Clenshaw, *Chebyshev Series for Mathematical Functions, Mathematical Tables*, vol. 5. London: National Physical Lab., 1962.

[9] W. Dorn, "Generalization of Horner's rule for polynomial evaluation," *IBM J. Res. Develop.*, vol. 6, pp. 239–245, April 1962.

[10] G. Estrin, "Organization of computer systems—the fixed plus variable structure computer," *1960 Western Joint Computer Conf.*, vol. 17, pp. 33–40.

[11] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *IRE Trans. Electronic Computers*, vol. EC-12, pp. 747–755, December 1963.

[12] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, December 1966.

[13] R. A. Henle and L. O. Hill, "Integrated computer circuits—past, present, and future," *Proc. IEEE*, vol. 54, pp. 1849–1860, December 1966.

[14] N. Metropolis and R. L. Ashenhurst, "Significant digit computer arithmetic," *IRE Trans. Electronic Computers*, vol. EC-7, pp. 265–267, December 1958.

[15] A. Avižienis, "On the problem of computational time and complexity of arithmetic functions," *Proc. ACM Symp. on the Theory of Computing* (Los Angeles, Calif.), pp. 255–258, May 5–6, 1969.

[16] C. Tung, "Signed-digit division using combinational arithmetic nets," *IEEE Trans. Computers*, this issue, pp. 746–748.