TOWARDS QUANTITATIVE COMPARISON OF COMPUTER NUMBER SYSTEMS*

S. Ong and D. E. Atkins

Systems Engineering Laboratory
Department of Electrical and Computer Engineering
and
Program in Computer, Information and Control Engineering
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

This paper describes an evolving Arithmetic Design System (ADS) to support the quantitative evaluation of alternate number systems with respect to a given application and realization technology. In computer arithmetic we are concerned with establishing a correspondence between abstract quantities (numbers) and some physical representation (symbols), and with simulating the operations on these symbols. The ADS is intended to help study the cost and performance of alternate simulations. A finite number system is a triple consisting of a symbol set (elements are called "digit-vectors"), an interpretation set, a mapping between these two sets, and a set of operators (digit-vector algorithms) defined on its symbol set. A set of these digit vector algorithms are proposed for conducting arithmetic design. A number system matrix defines the digit vector algorithm for numerous number systems and a method for computing time and space complexity of compositions of these algorithms is proposed. An example of how the system could be used to compare addition, with and without overflow detection, for three number systems is given.

Background and Motivation

The design of high-performance arithmetic processors involves a complex interaction between choice of algorithm, application of parallelism, and hardware realization. This paper describes an evolving Arithmetic Design System (ADS) to support the quantitative evaluation of alternate number systems with respect to a given application and realization technology. Although computer arithmetic is a relatively old subject and a large number of alternate methods are described in the literature, little has been done to offer a unified treatment suitable for comparative studies. Furthermore, methods which have been described and evaluated in the past should be re-examined in view of the potentials and requirements of VLSI circuit technology. The need for computation far exceeding present capability will require optimization at all levels of design including the execution of primitive arithmetic operations: the inner-most, inner loops.

Much of the computer arithmetic literature has been devoted to characterizing various number systems and realizing the basic arithmetic operations of these number systems. Not only conventional decimal and binary number systems but also non-conventional ones, such as residue number systems [GAR59], [SZA62], [SZA67], [BAN69], [BAN74], negative radix number systems [WAD57], [SON63], [REG67], [SAN73], [AGR75], p-adic number systems [RAO75], [KRI75], [GRE78], signed digit number systems [MET59], [ROB59], [AVI60], [AVI61], imaginary number systems [KNU60], [SLE76], [SLE78], and a modified reflected binary number system [LUC59], have been considerably investigated. It is well-known that all these number systems are homomorphic to each other. They all "simulate" the same input-output functional specifications. It is generally unclear, however, which among the astronomical number of possible number systems, including variations in radix as a special case, is best suited for implementing a given arithmetic task. It is extremely useful to know which simulation best matches the requirements, especially the time and space complexity, with respect to a given implementation environment.

There are several levels at which we can analyze the time and space complexity of these alternate simulations. At the application level, arithmetic algorithms are usually specified using the basic operators and elementary functions of a high-level programming language such as FORTRAN, ALGOL, or PASCAL. A complexity analysis can be conducted in terms of the number of executions required and the number of processors employed. This level lacks details concerning the representation of numbers and thus does not reflect the relative advantages among number systems. At the realization level, where arithmetic tasks are realized using logical devices, we can count the number of devices used as well as the operation time. However, the numerous details of logic design are tedious and sometimes severely obscured

* This work is supported by the National Science Foundation, Division of Mathematics and Computer Science, under Grant No. MCS-8007298.

arithmetic properties. We therefore need an intermediate level of abstraction to compare and evaluate arithmetic algorithms and thus propose an arithmetic design level as introduced by Avizienis [AVI71].

To enable comparison and evaluation at the arithmetic design level, a set of primitives must be provided with which arithmetic algorithms can be composed. They serve as the basic building blocks. Once the functional description and complexity measures of these primitives are well-defined, one can compose them to perform more complicated arithmetic tasks and analyze their complexities.

The goal of the ADS project is to develop a design and study tool for digital computer arithmetic. This tool will not only allow the user to describe and simulate arithmetic tasks, but also present the designer with a wide range of choices and a method for assigning a figure of merit to various number systems and structures with respect to a given implementation environment. The major subtasks of the ADS project which will be discussed in this paper are:

(1) Characterization and classification of finite number systems.

(2) Development of a suitable language for the ADS.

(3) Description and simulation of arithmetic algorithms.

(4) Case studies of the application of the ADS to cost/performance analysis in the context of VLSI.

Finite Number Systems

Computer arithmetic deals with the simulation of algebraic systems which are at least abelian groups under addition. Each of these algebraic systems consists of a set of numbers and n-ary operations on the set. The number may be defined as a symbolic abstraction, while the operations may be described by the postulates which specify certain assumed properties. In computer arithmetic we are concerned with establishing a correspondence between these abstract quantities (numbers) and some physical representation (symbols), and with simulating the operations on these symbols. Due to the finiteness of digital computing systems, only finite sets of numbers can be physically represented.

Representation of Numbers

Definition 1 : A Finite Number Representation System (FNRS) is a triple FNRS = (S,I,F),
where S is the symbol set (set of n-tuple digit vectors),
I is the interpretation set (set of abstract numbers), and
F is the evaluation function which maps S onto I.

Figure 1 shows the sets associated with a finite number representation system. Note that

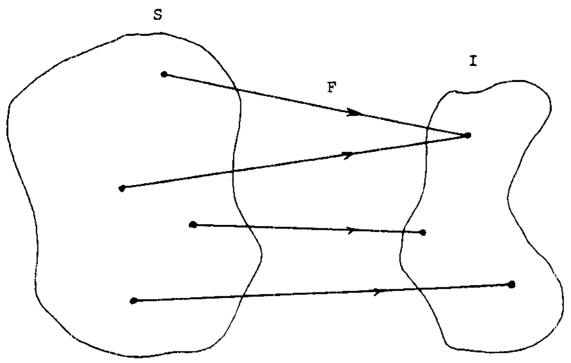


Figure 1 The sets associated with a FNRS.

all elements of the interpretation set have at least one corresponding representation in the symbol set. Figure 2 is an example of a formal, APL definition of the conventional radix complement FNRS [ATK78].

Definition 2 : A FNRS = (S,I,F) is said to be redundant, if F is a many-to-one function. In this case, there is at least one element of I which corresponds to more than one element of S.

Definition 3 : A FNRS = (S,I,F) is said to be positional, if the evaluation function F is dependent on the order of the elements of the digit vector.

Definition 4 : A FNRS = (S,I,F) is weighed, if F is defined such that

$$F(X) = \sum_{i=1}^{n} x_i w_i + c = q$$

where $(x_n, x_{n-1}, \ldots, x_1) = X \in S$,
$(w_n, w_{n-1}, \ldots, w_1) = W$ is a weight vector, and
c is a constant.

A special subclass of weighed FNRS are those in which the weight vector is obtained from a radix vector $B = (b_n, b_{n-1}, \ldots, b_1)$. If

```
SYMBOLS USED IN DEFINING RADIX COMPLEMENT FNRS:
   SB = RADIX. SB≥2.
   N = THE LENGTH OF DIGIT VECTORS.
   ZB = (0,1, ... ,SB-1) = THE SET OF ALLOWED DIGITS.

SYMBOL SET: S = N-ARY CARTESIAN PRODUCT OF SET ZB.

INTERPRETATION SET: I = (-K1, ... ,¯1,0,1, ... ,K2),
                    WHERE K1=(⌊SB÷2)×SB*(N-1),
                          K2=((⌈SB÷2)×SB*(N-1))-1.

SI FUNCTION: F: S → I.
             F(X) = (SB⊥X)-((1↑X)≥SB÷2)×SB*N.
```

Figure 2 Formal definition of radix complement FNRS.

$w_1 = 1$ and

$w_i = w_{i-1} \times b_{i-1}$ for $i = 2,3,\ldots,n$,

the weighed FNRS is called a (weighed) radix system. Furthermore, if all elements of $\overline{B}$ are the same then it is a fixed radix system, otherwise it is a mixed radix system.

The Singularities of Digit Vector Algorithms

The algorithms for mechanizing arithmetic operations on a symbol set, S, are called Digit Vector Algorithms (DVAs). It is obvious that the arithmetic properties on the interpretation set must be preserved. This implies that F be a homomorphism. For example, the digit vector algorithm, SUM, corresponding to the addition, +, of two numbers needs to be defined such that

$$F(SUM(X,Y)) = F(X)+F(Y) \equiv +(F(X),F(Y))$$

where $X, Y \in S$.

We commonly take finite subsets of numbers to be the interpretation sets, and provide singularity symbols to indicate that the result of an arithmetic operation in the given algebraic system does not have a corresponding digit vector in the symbol set. Formally,

Definition 5 : Given FNRS = (S,I,F), let $\triangle$ be a DVA corresponding to the arithmetic operation, *, of a given algebraic system. For $X_1,X_2,\ldots,X_n \in S$, $\triangle(X_1,X_2,\ldots,X_n)$ is singular, if

$$F(\triangle(X_1,X_2,\ldots,X_n)) \neq *(F(X_1),F(X_2),\ldots,F(X_n)).$$

The interpretation set, I, of a given FNRS can generate a minimum convex region, CR(I), which we shall call the range of I. One singularity, so-called out-of-range, is defined as follows:

Definition 6 : Given FNRS = (S,I,F), let $\triangle$ be a DVA corresponding to the arithmetic operation, *, of a given algebraic system. For $X_1,X_2,\ldots,X_n \in S$, $\triangle(X_1,X_2,\ldots,X_n)$ is out-of-range, if

$$*(F(X_1),F(X_2),\ldots,F(X_n)) \notin CR(I).$$

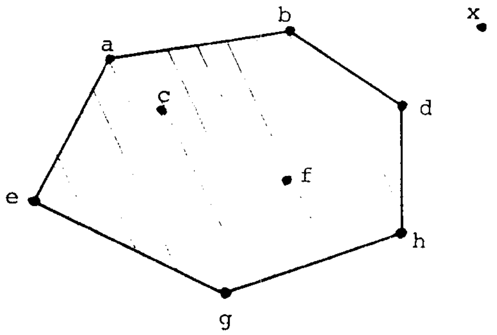
For example, as shown in Figure 3, if $I = \{a,b,c,d,e,f,g,h\}$ the shading region is the minimum convex region of I. Let A and B be the representations of a and b, respectively. If $*(a,b)=x$, then $\triangle(A,B)$ is out-of-range.

In case the set I is ordered, i.e. there exist MAX(I) and MIN(I) $\in$ I such that

$$MAX(I) \geq x \geq MIN(I)$$

for every $x \in I$, an out-of-range singularity can be further specified as follows:

Definition 7 : Given FNRS = (S,I,F), where I is an ordered set. Let $\triangle$ be a DVA corresponding to the arithmetic operation, *, of a given algebraic system. For $X_1,X_2,\ldots,X_n \in S$, $\triangle(X_1,X_2,\ldots,X_n)$ is overflow if



$I = \{a,b,c,d,e,f,g,h\}$

Figure 3 An example of out-of-range singularity.

$$*(F(X_1),F(X_2),\ldots,F(X_n)) > MAX(I).$$

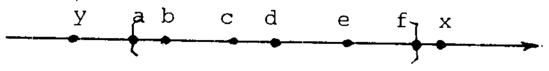
Definition 8 : Given FNRS = (S,I,F), where I is an ordered set. Let $\triangle$ be a DVA corresponding to the arithmetic operation, *, of a given algebraic system. For $X_1,X_2,\ldots,X_n \in S$, $\triangle(X_1,X_2,\ldots,X_n)$ is underflow if

$$*(F(X_1),F(X_2),\ldots,F(X_n)) < MIN(I).$$

As shown in Figure 4, given $I = \{a,b,c,d,e,f,\}$ we have $f = MAX(I)$ and $a = MIN(I)$. Let A and B be the representations of a and b, respectively. $\triangle(A,B)$ is overflow if $*(a,b)=x$, and $\triangle(A,B)$ is underflow if $*(a,b)=y$.

Another direct consequence of the finiteness of digital computing systems is the loss of precision in some arithmetic operations. A truncation signal will be issued as specified by the following definition:

Definition 9 : Given FNRS = (S,I,F), let $\triangle$ be a DVA corresponding to the arithmetic operation, *, of a given algebraic system. For $X_1,X_2,\ldots,X_n \in S$, $\triangle(X_1,X_2,\ldots,X_n)$ is truncated if



$I = \{a,b,c,d,e,f\}$
MAX (I) = f
MIN (I) = a

Figure 4 Singularity examples of overflow and underflow.

$$*(F(X_1),F(X_2),\ldots,F(X_n)) \not\in CR(I), \text{ and}$$

$$F(\triangle(X_1,X_2,\ldots,X_n)) \neq *(F(X_1),F(X_2),\ldots,F(X_n)).$$

For example, if $I = \{a,b,c,d,e,f,g,h\}$ as shown in Figure 5, then the minimum convex region of I is the shaded region. Let a and b be represented by A and B respectively. $\triangle(A,B)$ is truncated if $*(a,b) = x$.

When a singularity signal is issued, the DVA produces a pseudo-result instead of the expected result. These singularity detections impact the complexity analysis which we shall discuss later.

Classification of Finite Number Systems

Note in the previous section that we have made a distinction between a "finite number system" and a "finite number representation system." A finite number representation system is a triple consisting of a symbol set, an interpretation set, and a mapping between these two sets. A finite number system consists of a finite number representation system together with DVAs (operations) defined on its symbol set.

In order to apply a single complexity analysis to a set of finite number systems, it is useful to classify them. Figure 6 summarizes the traditional classification of a large set of finite number systems, based on the nature of the evaluation functions, F. An alternate taxonomy which we are attempting to establish is based not only upon evaluation functions, but also upon complexity measures.

Arithmetic Design Language

Introduction

The designer using the ADS works with building blocks such as shown in Figure 7.

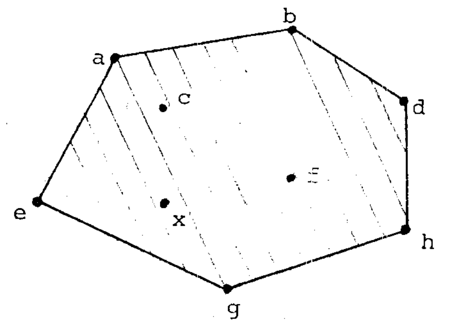


Figure 5 An example of truncation singularity.

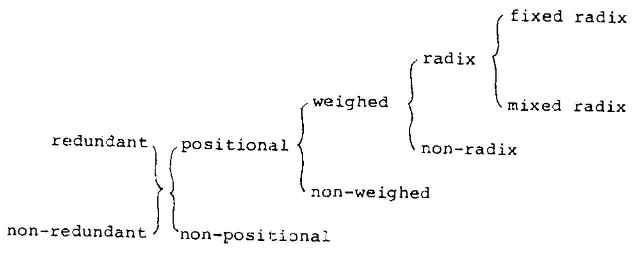


Figure 6 Traditional classification of finite number systems.

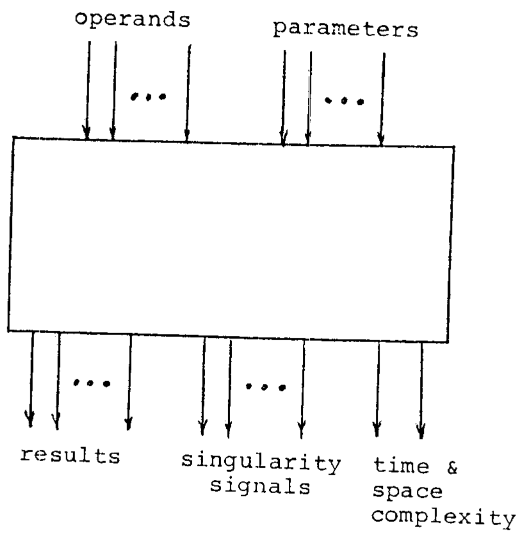


Figure 7 Inputs and outputs of a building block.

The inputs to each block are:

(a) One or more operands (digit vectors) represented in predefined formats.
(b) Parameters to fully specify the number system (e.g. base) and to specify the implementation model to be used for complexity analysis.

The outputs from each block are:

(a) One or more results (digit vectors) represented in prefined formats.
(b) Singularity signals.
(c) Time and space complexity measures.

Since there is no unique composition of these building blocks to realize a given arithmetic task, it is useful to have an executable language to describe, simulate, and compare alternate structures.

The objectives of the arithmetic design language (ADL) [ONG80] are as follows:

--- Provide both functional and structural descriptions of the building blocks.

--- Allow composition of building blocks to perform arithmetic tasks.

--- Be directly executable as a simulation program.

--- Facilitate the complexity analyses of alternate approaches to implementing arithmetic tasks with respect to a well-defined criteria.

--- Allow manipulation to suggest new approaches.

--- Be compact and easy to use.

The arithmetic design language includes two types of operators: processing operators and constructive operators. Processing operators establish the operational behavior (the function), while constructive operators describe structure. Using the basic processing and constructive operators, the designer can define a set of building blocks which he/she feels is adequate for the given implementation environment and then extended them as necessary. The building blocks are in turn composed to build more complex arithmetic processors.

Processing Operators

As suggested by Avizienis [AVI71], the primitive operators of the language APL [GIL70] [PAK72] appear to be a good choice for the processing operators of an arithmetic design language. The constructive operators of the language include the control-flow constructs of APL augmented by other functions which will be described.

Constructive Operators

For a given DVA functional specification we want to describe alternate structural options and analyze their complexity. We are particularly interested in specifying parallelism, iteration in time, and hardware replication in the context of computer arithmetic. Constructive operators are used to describe both the internal structure of the building blocks and the structure of the compositions of blocks. In order to compare and evaluate alternate compositions, by means of simulation, constructive operators must also update the complexity measures properly.

The synchronization primitives fork and join can be used to specify parallel computations explicitly [CON63]. The ADL constructive operator FORK activates concurrent operations at the same time, while JOIN waits until all precedent operations finish. These operators also keep track of the processing time required for each concurrent operation, and thus the overall processing time for the algorithms.

Figure 8 lists an APL-implementation of FORK and together with the two functions used to implement the JOIN operator. LEB is a list of statement labels of concurrent operations except the one immediately following the FORK statement. Note that JOIN operator as defined must be constructed using three contiguous statements as shown in lines [5], [6] and [7] of the function DEMOFJ in Figure 9. TMAX computes JTC, the latest time when concurrent operations finish, and COUNT counts the number of concurrent operations finished. M is the number of precedent operations to be finished before the successive ones are activated. A stack

```
      ∇ FORK LEB;X
[1]     X←ρ,LEB
[2]     STACK←((2,X)ρLEB,XρTCPX),STACK
      ∇


      ∇ Z←TMAX JTC
[1]     Z←TCPX←JTC⌈TCPX
      ∇

      ∇ Z←COUNT M
[1]     →(0=Z←M-1)/7
[2]     DONE←0
[3]     PTR←,STACK[ORG;ORG]
[4]     TCPX←,STACK[ORG+1;ORG]
[5]     STACK← 0 1 ↓STACK
[6]     →0
[7]     DONE←1
      ∇
```

Figure 8 An implementation of FORK and functios used to implement JOIN.

is used to keep track of timing such that the time complexity of the parallel construct can be derived correctly. Figure 9 shows an example of a flowchart and the corresponding program. Note that, all concurrent operations, even if they are data independent, must terminate with a JOIN.

Conventional APL loop instructions make no distinction between iteration using one unit of hardware (iteration in time) and iteration using unlimited hardware (iteration in space). Moreover, it is not clear whether the physical replication of identical units has the form of parallel structure (type 0*) such as logical AND of two vectors, or the form of linear recurrence structure (type 1*) such as carry ripple adders. In the ADL, the replication of identical hardware units in parallel is directly expressed by APL operators and functions. We designate the APL looping constructs for representing the recurrence structure. For the other case, iteration in time, we introduce TIB-TIE operators.

TIB (time iteration begin) and TIE (time iteration end) describe the structures using a unit of hardware repetitively. An implementation of TIB and TIE is listed in Figure 10. STREND consists of two parameters, STR and END, which are the starting and the ending values of the iteration index respectively. DIR indicates the direction of increment: increasing by 1 if DIR=0; decreasing by 1 if DIR=1. A flag HOFF, which is initialized to zero, is used to indicate whether the iteration is over. HOFF is also used to derive the space complexity of algorithms. Note that TIE operator must be used with an auxiliary statement as shown in lines [13] and [14] of the function DEMOTI in Figure 11. Figure 11 illustrates how a 2-segment pipeline can be described using TIB-TIE and FORK-JOIN operators.

* Type 0 has a maximum speedup of $O(T_1)$, and type 1 has a maximum speedup of $O(T_1/\log T_1)$ [KUC78].

```
    ∇ DEMOFJ;M1;M2;M3;JTC1;JTC2;JTC3
[1]    M1←M2←M3←2
[2]    JTC1←JTC2←JTC3←0
[3]    FORK LBB
[4]    A
[5]   LEE:JTC1←TMAX JTC1
[6]    M1←COUNT M1
[7]    →(~DONE)/PTR
[8]    E
[9]   LHH:JTC3←TMAX JTC3
[10]   M3←COUNT M3
[11]   →(~DONE)/PTR
[12]   H
[13]   →0
[14]  LBB:B
[15]   FORK LDD
[16]   C
[17]   FORK LEE
[18]   F
[19]  LGG:JTC2←TMAX JTC2
[20]   M2←COUNT M2
[21]   →(~DONE)/PTR
[22]   G
[23]   →LHH
[24]  LDD:D
[25]   →LGG
    ∇
```

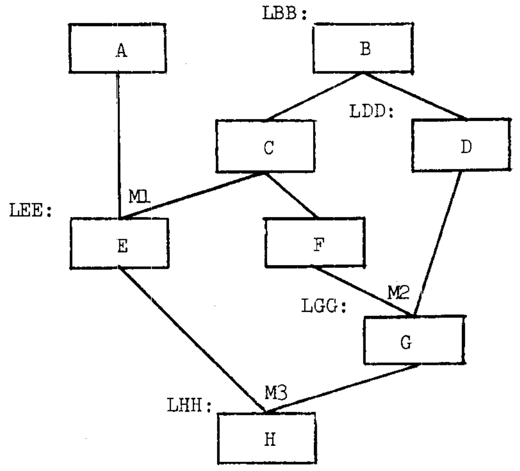


Figure 9 Example of use FORK-JOIN.

```
    ∇ TIB STREND
[1]    →(HOFF)/0
[2]    I←1↑STREND
[3]    ENDINDEX←¯1↑STREND
    ∇
```

```
    ∇ TIE DIR
[1]    →(DIR)/7
[2]    →(ENDINDEX<I←I+1)/5
[3]    HOFF←1
[4]    →0
[5]    HOFF←0
[6]    →0
[7]    →(ENDINDEX>I←I-1)/5
[8]    HOFF←1
    ∇
```

Figure 10 An implementation of TIB and TIE.

```
    ∇ DEMOTI;MTR;SVR;JTC;M
[1]    MTR←0
[2]    JTC←0
[3]    TIB 1 11
[4]    M←2
[5]    SVR←MTR
[6]    FORK LADD
[7]    MTR←STORE I DEMOMPY 2
[8]    →10
[9]   LADD:SVR DEMOADD 5
[10]   JTC←TMAX JTC
[11]   M←COUNT M
[12]   →(~DONE)/PTR
[13]   TIE 0
[14]   →(HOFF)/3
    ∇
```

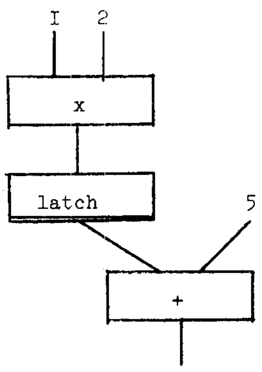


Figure 11 Example of use TIB-TIE together with FORK-JOIN.

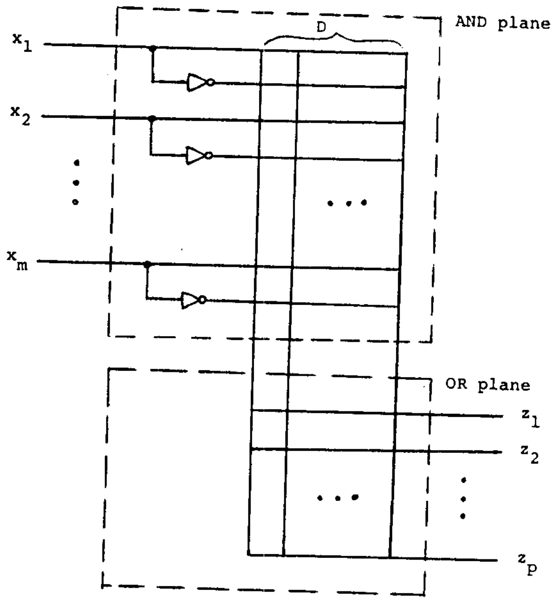
Number System Matrix

To compare and evaluate alternate approaches in arithmetic design level discussed earier, a set of primitive DVAs will be provided as building blocks. The proposed set which is motivated by Avizienis [AVI71] includes SUM (sum), DIF (difference), INV (additive inverse), SGN (sign test), EQZ (equal zero test), REX (range extension), RCN (range contraction), SDN (scale down), SUP (scale up), and PRD (product of a digit and a digit vector). One can compose these building blocks together with auxiliary logic to perform more complicated arithmetic algorithms. Figure 12 is an example of a radix complement multiplication using REX, SUM, PRD and INV.

The Number System Matrix (NSM) is a database in the Arithmetic Design System. Conceptually, the Number System Matrix, as shown in Figure 13, is a matrix with a row per primitive digit vector algorithm, a column per finite number system, and a set of descriptions at each intersection. A set of finite number systems which is extendible is provided. This set includes unsigned (US), sign-magnitude (SM), radix complement (RC), diminished radix complement (DRC), unsigned residue (USR), sign-magnitude residue (SMR), modular complement residue (MCR), symmetric signed digit (SSD), p-adic (PAD), modified k-imaginary (IMG), negative radix (NR), and modified reflected binary (MRB) finite number systems. Every astral mark in Figure 13 indicates that a functional description of the primitive digit vector algorithm for the finite number system has been implemented. The implementation, as well as the formal definitions of these finite number representation systems, is available and the interested reader is invited to contact the authors.

One can compose these primitive digit vector algorithms to perform more complicated arithmetic algorithms. To evaluate the time and space complexity of a composed arithmetic algorithm, it is necessary to know the time and space complexity measures of these primitives. It is very much dependent on the technology employed and the decomposition to match feasible physical elements to realize these building blocks. Hence the complexity measures vary case by case. In Nnumber System Matrix, some suggested implementations are supplied. If the existent alternatives can not meet the complexity specification, one can create private library for new implementations.

In next section, we shall compare and evaluate some PLA-based implementations of a given arithmetic task. As indicated in [ATK81], we can crunch the PLA, using folding and/or rearranging, to minimize the silicon area used. Programs to estimate the area for minimized PLAs are under development. In the meantime we choose unminimized binary coded PLA as a model for complexity measures. The model gives an upper bound on complexity measures using PLAs to realize arithmetic tasks. Figure 14 shows the simple model we are presently using and its complexity measures. TC corresponds to the units of gate delay, and SC indicates the number of cross points in the AND and OR planes.



$2^m \geq D > 2^{m-1}$.

D is the number of all possible inputs.

TC = 3.

SC = D x (p + 2 x m).

Figure 14 A model of complexity measures.

Preliminary Example of Application

In this section we will give a simple example of how the ADS could be used. We shall quantify the time and space complexity for the addition operations in radix complement, modulus complement residue, and symmetric signed digit number systems, with and without out-of-range detections.

Residue number systems [GAR59], [SZA62], [SZA67], [BAN69], [BAN74] and signed digit number systems [MET59]. [ROB59], [AVI60], [AVI61] have been considerably investigated. Probably the most notable feature of these number systems is the so called carry-free addition, i.e. the time required to perform addition is independent of the length of digit vectors. Despite this fact, even if addition is the only operation implemented, it may not be justified to conclude that residue and signed digit number systems are superior to radix complement number systems which require a carry propagation.

Figure 15 shows an implementation of addition with out-of-range detection for radix complement number systems, of which the twos complement is a very special case. Out-of-range detection for odd radices is more complicated than one might think based on experience with the binary case. Note that the out-of-range signal, OOR, is a function of x_1, y_1, and z_1, the most significant digits of

```
     ∇ Z←MD MPYRCSFA MR;JTC;M;I;LMD;LMR;LAST;X;Y;S;CIN
[1]    ⍝ MULTIPLICATION DVA FOR RADIX COMPLEMENT FNS.
[2]    LMD←⍴MD
[3]    LMR←⍴MR
[4]    LAST←LMD+LMR-NORG
[5]    I←LMR-NORG
[6]    JTC←0
[7]    Z←SFTREG(LMD⍴0),MR
[8]    TIB I,ORG
[9]    M←2
[10]   FORK 13
[11]   X←1 REXRC LMD↑Z
[12]   →14
[13]   Y←GENRC Z[LAST]
[14]   JTC←TMAX JTC
[15]   M←COUNT M
[16]   →(~DONE)/PTR
[17]   S←X SUMRCCRN Y
[18]   Z←¯1↓S,LMD↓Z
[19]   TIE 1
[20]   →(HOFF)/8
     ∇
```

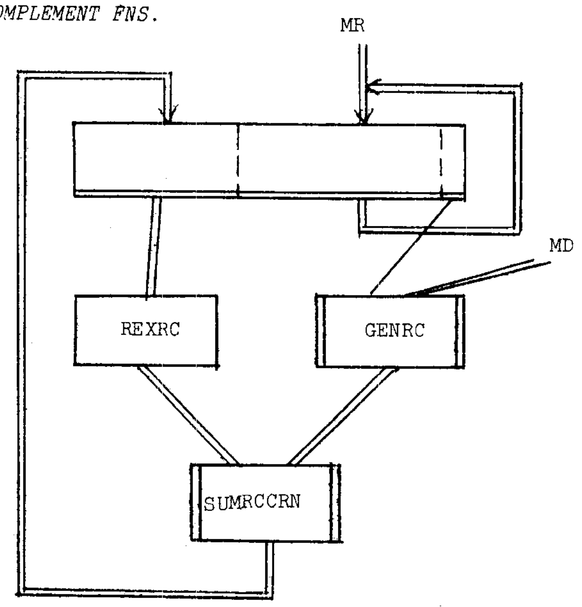


Figure 12 Multiplication for radix complement FNS.

	US	SM	RC	DRC	USR	SMR	MCR	SSD	PAD	IMG	NR	MRB
SUM	*	*	*	*	*	*	*	*	*	*	*	*
DIF	*	*	*	*	*	*	*	*	*	*	*	*
INV	NDEF	*	*	*	NDEF	*	*	*	*	*	*	NDEF
SGN	NDEF	*	*	*	NDEF	*	*	*	*	NDEF	*	NDEF
EQU	*	*	*	*	*	*	*	*	*	*	*	*
REX	*	*	*	*	*	*	*	*	NDEF	*	*	*
RCN	*	*	*	*	*	*	*	*	NDEF	*	*	*
SDN	*	*	*	*	*	*	*	*	*	*	*	*
SUP	*	*	*	*	NDEF	NDEF	NDEF	*	*	*	*	*
PRO	*	*	*	*	NDEF	NDEF	NDEF	*	*	*	*	NAVL

Figure 13 The Number System Matrix of the ADS.

```
     ∇ Z←X SUMRCCRO Y;LEN;C;K;MA;SGNX;SGNY;ZORG;COUT;NAD1;NAD2;M;JTC
[1]    ⍝ SUM DVA FOR RADIX COMPLEMENT FNS.
[2]    LEN←ρ,X
[3]    C←((LEN-1)ρ0),CIN
[4]    K←LEN-NORG
[5]    NAD1←NAD2←LENρSB
[6]    →(SB=2×⌈SB÷2)/28
[7]    JTC←0
[8]    M←4
[9]    FORK 24 26
[10]   MA←X ADD Y
[11]   →(ORG=K)/14
[12]   C[K-1]←MA[K]CRYUS1 C[K]
[13]   →(ORG<K←K-1)/12
[14]   FORK 22
[15]   Z←MA ADDUS1 C
[16]   ZORG←1↑,Z
[17]   JTC←TMAX JTC
[18]   M←COUNT M
[19]   →(~DONE)/PTR
[20]   OOR←(SGNX,SGNY)OSRC2A ZORG,COUT
[21]   →0
[22]   COUT←MA[ORG]CRYUS1 C[ORG]
[23]   →17
[24]   SGNX←GERC 1↑,X
[25]   →17
[26]   SGNY←GERC 1↑,Y
[27]   →17
[28]   JTC←0;M←3
[29]   FORK 41 43
[30]   MA←X ADD Y
[31]   →(ORG=K)/34
[32]   C[K-1]←MA[K]CRYUS1 C[K]
[33]   →(ORG<K←K-1)/32
[34]   Z←MA ADDUS1 C
[35]   ZORG←1↑,Z
[36]   JTC←TMAX JTC
[37]   M←COUNT M
[38]   →(~DONE)/PTR
[39]   OOR←(SGNX,SGNY)OSRC2B ZORG
[40]   →0
[41]   SGNX←GERC 1↑,X
[42]   →36
[43]   SGNY←GERC 1↑,Y
[44]   →36
     ∇
```

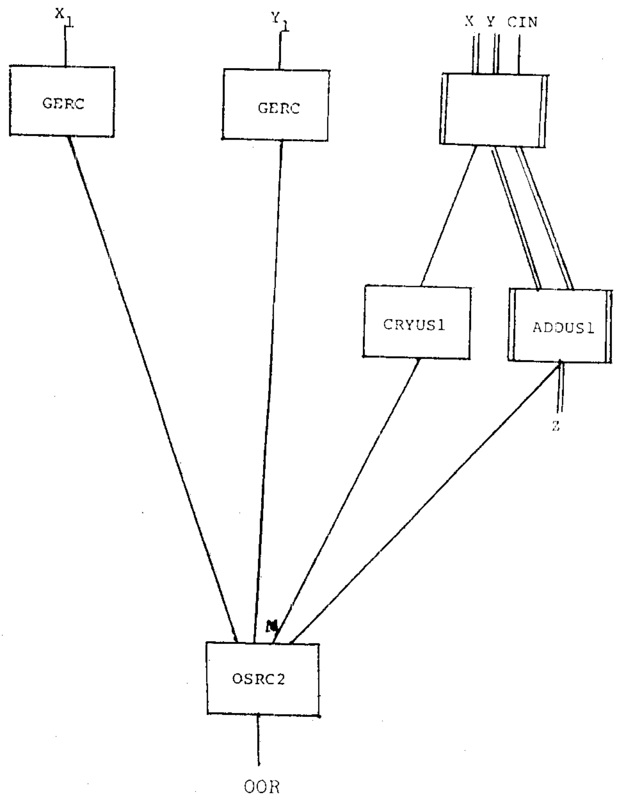


Figure 15 Addition with out-of-range detection for radix complement FNS.

X, Y, and Z respectively.

A block diagram and the corresponding description of addition with out-of-range detection for symmetric signed digit number systems are shown in Figure 16. The DVA, SUMSSDO, is a generalized description for all radix SB ≥ 3. Due to the fact that symmetric signed number systems are redundant, an (N+1)-digit intermediate result might be represented by an N-digit vector. Therefore, the out-of-range signal, OOR, can not be determined until the conversion procedure is done. In the worst case, using a carry ripple scheme, it will take N units of time to convert.

Figure 17 shows a block diagram and the corresponding description of addition with out-of-range detection for modulus complement residue number systems. The result obtained, Z, is congruent to the expected result modulo M. M is the product of all elements of the radix vector. To detect out-of-range, one should convert the obtained result to the corresponding mixed radix representation. Unless more efficient algorithms can be developed, it will take time, which is linear in the length of digit vectors, to perform the conversion.

A comparison in terms of time and space complexity as functions of digit vector length is shown in Figure 18 (a) and (b) respectively. The complexity analysis is conducted with B=(32 31 29 27 25 23) for modulus complement residue, SB=29, A=16 for symmetric signed digit, and SB=32 for radix complement number systems based on the assumption that 10 input-variable PLAs are available. To implement addition without out-of-range detection, modulus complement residue number systems are the fastest and require least space, and symmetric signed digit number systems are faster but require more hardware than radix complement number systems. However, if we also require out-of-range detection, it will take less time and hardware for radix complement number systems to perform addition than the other two. Figure 19 (a) and (b) shows the simulation results. This is not surprising since, as we pointed out earlier, modulus complement residue and symmetric signed digit number systems require linear time and extra hardware to detect out-of-range.

Let N be digit vector length, and X denote the percentage which requires out-of-range detection over all operations in a given addition-only application. According to the simulation results, we can derive the threshold as a function of X and N such that one of these three number systems is best suited. Table 1 summarizes the thresholds between any two number systems where the delay time is the only concern.

Table 2 displays some conclusions from trial computations. For N equal 2 through 10, it shows the conditions such that one of three number systems is the fastest. Note that, under the assumptions made, symmetric signed digit number systems offer no advantage over the other two for $N \leq 6$.

MC is faster than RC	$X < \frac{N}{2(N-1)}$
SD is faster than RC	$X < \frac{N-2}{N+1}$
MC is faster than SD	$X < \frac{2}{N-3}$

MC denotes Modulus Complement residue number systems.
SD denotes symmetric Signed Digit number systems.
RC denotes Radix Complement number systems.

Table 1 Thresholds between RC, SSD, and MCR.

N	modulus complement residue	symmetric signed digit	radix complement
2	X < 100%	-	-
3	X < 75%	-	X > 75%
4	X < 66.7%	-	X > 66.6%
5	X < 62.5%	-	X > 62.5%
6	X < 60%	-	X > 60%
7	X < 50%	50% < X < 62.5%	X > 62.5%
8	X < 40%	40% < X < 66.7%	X > 66.7%
9	X < 33.3%	33.3% < X < 70%	X > 70%
10	X < 29%	29% < X < 73.3%	X > 73.3%

- denotes not-suited.

Table 2 Conditions such that a number system is fastest.

```
    ∇ Z←X SUMSSDO Y
[1]   ⍝ SUM DVA FOR SYMMETRIC SIGNED DIGIT FNS.
[2]   Z←X SUMSSDN Y
[3]   Z←NORSSD OT,Z
    ∇
```

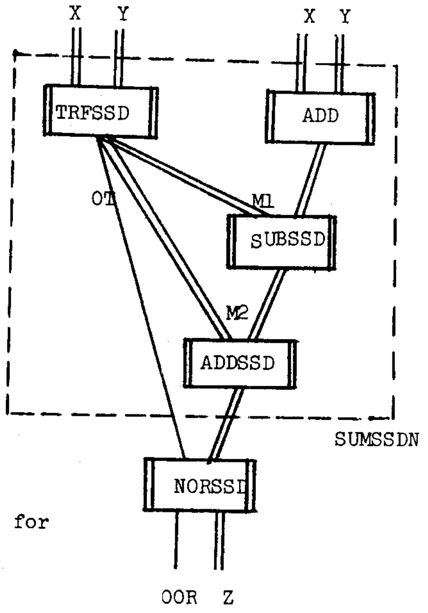


Figure 16 Addition with out-of-range detection for symmetric signed digit FNS.

```
    ∇ Z←X SUMMCRO Y;MOD;NAD1;NAD2;JTC1;JTC2;M1;M2;FX;FY;FF;SGNX;SGNY
[1]   ⍝ SUM DVA FOR MODULUS COMPLEMENT RESIDUE FNS.
[2]   MOD←NAD1←NAD2←B
[3]   JTC1←JTC2←0
[4]   →(B[ORG]=2×⌈B[ORG]÷2)/28
[5]   M1←2
[6]   M2←4
[7]   FORK 15 24
[8]   Z←X MODADD Y
[9]   FZ←1↑CONTOMR Z
[10]  JTC2←TMAX JTC2
[11]  M2←COUNT M2
[12]  →(~DONE)/PTR
[13]  OOR←(SGNX,SGNY)OSMC2A FZ,FF
[14]  →0
[15]  FY←1↑CONTOMR Y
[16]  FORK 19
[17]  SGNY←GEMC FY
[18]  →10
[19]  JTC1←TMAX JTC1
[20]  M1←COUNT M1
[21]  →(~DONE)/PTR
[22]  FF←FX SRTMC FY
[23]  →10
[24]  FX←1↑CONTOMR X
[25]  FORK 19
[26]  SGNX←GEMC FX
[27]  →10
[28]  M2←3
[29]  FORK 37 40
[30]  Z←X MODADD Y
[31]  FZ←1↑CONTOMR Z
[32]  JTC2←TMAX JTC2
[33]  M2←COUNT M2
[34]  →(~DONE)/PTR
[35]  OOR←FZ OSMC2B SGNX,SGNY
[36]  →0
[37]  FY←1↑CONTOMR Y
[38]  SGNY←GEMC FY
[39]  →32
[40]  FX←1↑CONTOMR X
[41]  SGNX←GEMC FX
[42]  →32
    ∇
```

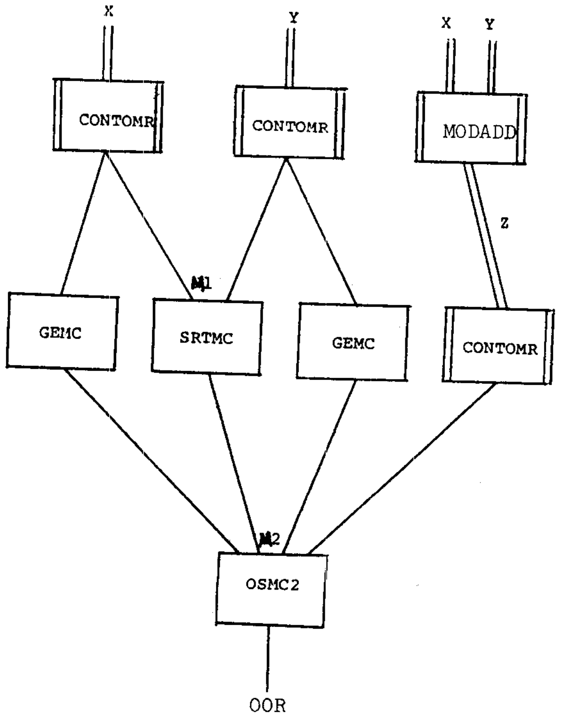


Figure 17 Addition with out-of-range detection for modulus complement residue FNS.

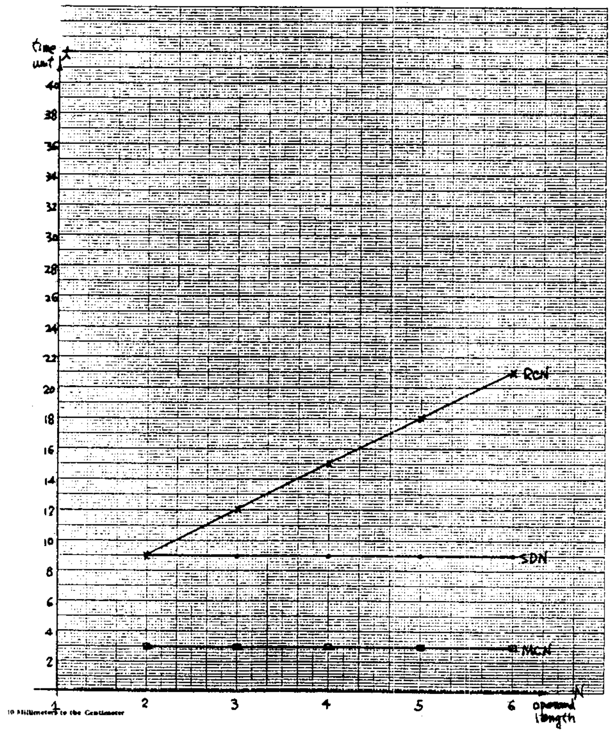


Figure 18 (a) Operational delay versus digit vector length for addition without out-of-range detection of RC, SSD, and MCR.

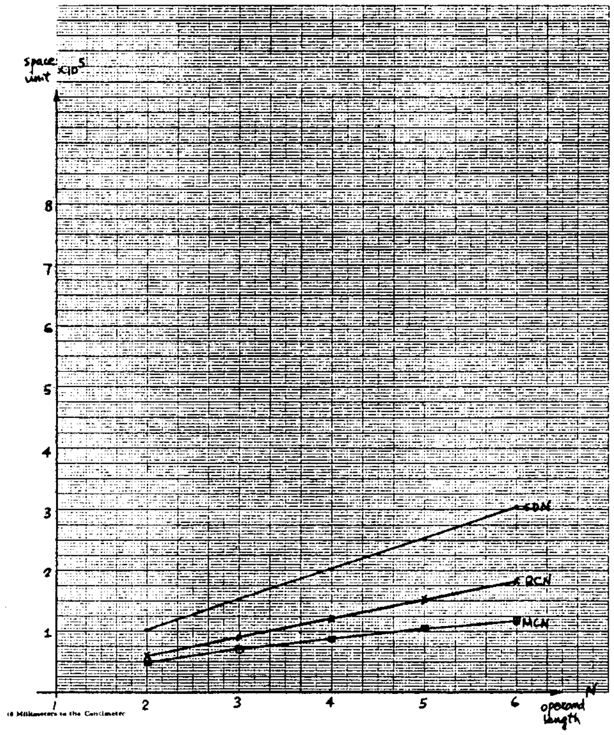


Figure 18 (b) Required space versus digit vector length for addition without out-of-range detection of RC, SSD, and MCR.

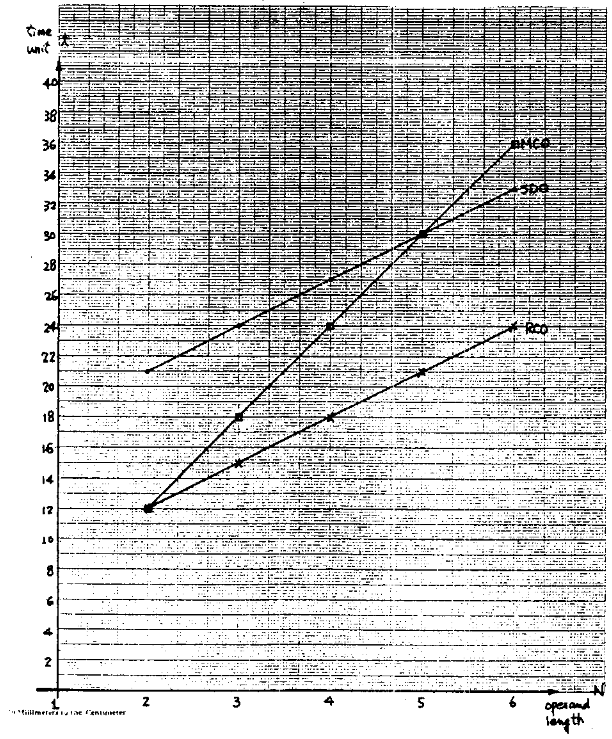


Figure 19 (a) Operational delay versus digit vector length for addition with out-of-range detection of RC, SSD, and MCR.

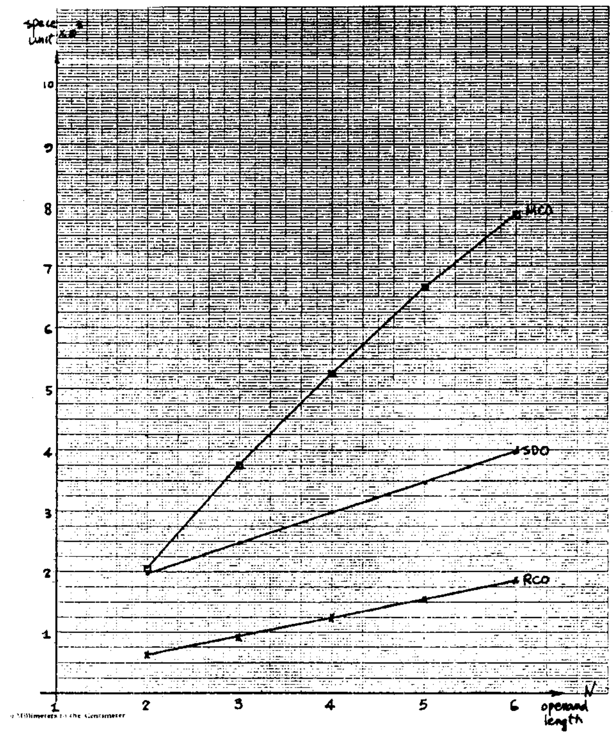


Figure 19 (b) Required space versus digit vector length for addition with out-of-range detection of RC, SSD, and MCR.

REFERENCES

[AGR75] D. P. Agrawal, "Arithmetic algorithms in a negative base," IEEE Trans. Comp. Vol. C-24, pp. 998-1000, Oct. 1975.

[ATK78] D. E. Atkins, "A suggested approach to computer arithmetic for designers of multi-valued logic processors," Proc. of 8th Int. Sym of Multi-Valued Logic, pp. 33-46, 1978.

[ATK81] D. E. Atkins, W. Liu and S. Ong, "Overview of an Arithmetic Design System," Proc. of 18th ACM Design Automation Conference, Nashville, June 1981.

[AVI60] A. Avizienis, "A study of redundant number representations for parallel digital computers," Ph.D Thesis, University of Illinois, May 1960.

[AVI61] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," IRE Trans. Electronic Computers, pp. 389-400, Sept. 1961.

[AVI71] A. Avizienis, "Digital computer arithmetic: a unified algorithmic specification," Proc. of Sym. on Computer and Automata, pp 509-525, 1971.

[BAN69] D. K. Banerji and J. A. Brzozowski, "Sign detection in residue number systems," IEEE Trans. Comp. Vol. C-18, pp. 313-320, April 1969.

[BAN74] D. K. Banerji, "On the use of residue arithmetic for computation," IEEE Trans. Comp. Vol. C-23, pp. 1315-1317, Dec. 1974.

[CON63] M. E. Conway, "A multi-processor system design," AFIPS Conference Proceedings FJCC, Vol. 24, pp. 139-146, 1963.

[GAR59] H. L. Garner, "The residue number system," IRE Trans. Electronic Computers, Vol. EC-8, pp. 140-147, June 1959.

[GIL70] L. Gilman and A. Rose, APL 360, An Interactive Approach. New York: Wiley, 1970.

[GRE78] R. T. Gregory, "The use of finite-segment p-adic arithmetic for exact computation," BIT 18, pp. 282-300, 1978.

[KNU60] D. E. Knuth, " An imaginary number system," ACM Communications, Vol. 3, pp. 245-247, April 1960

[KRI75] E. V. Krishnamurthy, T. M. Rao and K. Subramanian, "Finite segment p-adic number systems with application to exact computation," Proc. Indian Acad. Sci., 81A, pp 58-79, 1975.

[KUC78] D. J. Kuck, The Structure of Computers and Computations, Vol. 1, New York: Wiley, 1978.

[LUC59] H. M. Lucal, "Arithmetic operations for digital computers using a modified reflected binary code," IRE Trans. Electronic Computers, Vol. EC-8, pp. 449-458, Dec. 1959.

[MET59] G. Metze and J. E. Robertson, "Elimination of carry propagation in digital computers," Proc. Int. Conf. on Information Processing, June 1959.

[ONG80] S. Ong, "A proposal for quantitative comparison of computer number systems," SEL, Dept. of Elec. and Comp. Eng., University of Michigan, 1980.

[PAK72] S. Pakin, APL 360 Reference Manual, 2nd ed., Chicago, IL: SRA, 1972.

[RAO75] T. M. Rao, "Finite field computational techniques for exact solution of numerical problems," Ph.D. Thesis, Dept. of Math, Indian Institute of Science, 1975.

[REG67] M. P. DeRegt, "Negative radix arithmetic," Computer Design, Vol. 6, part 1-8, May 67-Jan1968.

[ROB59] J. E. Robertson, "Redundant number systems for digital computer arithmetic," Topics in the Design of Digital Computing Machines, University of Michigan, Engineering Summer Conference, 1959.

[SAN73] P. V. Sankar, S. Chakrabarti and E. V. Krishnamurthy, "Arithmetic algorithms in a negative base," IEEE Trans. Comp., Vol. C-22, pp. 120-125, Feb. 1973.

[SLE76] A. G. Slekys, "Design of complex number digital arithmetic unit based on a modified bi-imaginary number system," Ph.D. Thesis in Engineering, UCLA, 1976.

[SLE78] A. G. Slekys and A. Avizienis, "A modified bi-imaginary number system," Proc. 4th Sym. on Computer Arithmetic, pp. 48-55, Oct. 1978.

[SON63] G. F. Songster, "Negative-base number representation systems," IEEE Trans. Electronic Computers, Vol. EC-12, pp. 274-277, June 1963.

[SZA62] N. Szabo, "Sign detection in nonredundant residue systems," IRE Trans. Electronic Computers, Vol. EC-11, pp. 494-500, Aug. 1962.

[SZA67] N. S. Szabo and R. I. Tanaka, Residue Arithmetic and Its Application to Computer Technology, McGraw-Hill, 1967.

[WAD57] L. B. Wadel, "Negative base number systems," IRE Trans. Electronic Computers, Vol. EC-6, pp. 123, June 1957.