LEFT SHIFTING ACCUMULATOR SYSTEMS WITH APPLICATIONS IN DIGITAL SIGNAL PROCESSING

by

Jesse Charles Booher

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Electrical Engineering

Program

Prof. Samuel J. Skarote 6/5/90

Advisor Date

Sally M. Hotchkiss June 6, 1990

Dean of Graduate School Date

YOUNGSTOWN STATE UNIVERSITY

JUNE, 1990

ABSTRACT

LEFT SHIFTING ACCUMULATOR SYSTEMS WITH APPLICATIONS IN DIGITAL SIGNAL PROCESSING

Jesse Charles Booher
Master of Science, Electrical Engineering
Youngstown State University, 1990

An algorithm is developed which performs the serial accumulation of parallel input data, while shifting each partial sum to the left. The algorithm is demonstrated to be applicable to the operation of multiplication, and evidence to support its applicability to division is presented. Various configurations of modular elements, including two types of configurations which support fully systolic communication patterns, are presented as suggested modes of algorithm implementation. These systems operate as on-line digital signal processors, in which data is processed in a most significant digit first serial fashion, so as to support the chaining of operations such as floating-point multiplication and division. Examples of the operation of the algorithm and its application to multiplication are presented, based on computer simulation data. Comparative data is presented which supports the conclusion that this system operates with a throughput rate a significant factor higher than existing on-line systems, while utilizing more simplistic hardware modules.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my family and friends who lent me encouragement and support when I really needed it. I would especially like to thank my thesis advisor, Professor Samuel Skarote, without whose infinite patience, as I worked and reworked my ideas, this thesis would not have been possible in its present form.

TABLE OF CONTENTS

LIST OF SYMBOLS

SYMBOL	DEFINITION
Σ	Summation of terms...
ϵ	Is a subset of...
$\forall$	For all...
$\mid$	Such that...
$\lvert x \rvert$	Absolute value of x
$\lVert x \rVert$	Greatest integer $\leq$ x
MSD	Most Significant Digit
LSD	Least Significant Digit
CSA	Carry Save Adder

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 DIGITAL SIGNAL PROCESSING

Modern digital signal processing applications require the use of very high-speed arithmetic processing circuitry. A particularly important and challenging problem is the design of multiplication networks that can keep up with the demands of real time processing environments.

Reference [7] contains an example of a traditional serial data flow approach to this multiplier design problem. In this system, data is accepted least significant digit first, the data stream is multiplied by a pre-loaded value, and after a brief latency period the output data stream is initiated in a least significant digit first protocol.

For many applications this approach is considered to be too slow. In order to accelerate the multiplication process, a variety of parallel multiplication procedures have been developed. Two common types of parallel multiplication networks are the tree multiplier and the array multiplier constructs. The most common type of tree multiplier is the Wallace tree design which uses a tree of carry save adders

(CSAs) feeding into a single carry propagating adder (CPA) to perform high speed multiplication. A more recent innovation in tree multiplier design is the binary adder tree developed by Harata et al [6]. This design replaces the CSAs used in the Wallace trees with redundant adders of a type suggested by Avizienis [1], achieving more regular hardware interconnection patterns at the cost of more complex hardware modules. Array multiplier designs do not generally support as high a processing rate as the tree designs, especially for high precision operations; their simplicity of layout and efficient use of chip area, however, make them the generally preferred method of performing parallel multiplication [6]. Examples of modern array multiplier designs are given in references [3] and [18].

Two basic problems are encountered in performing parallel multiplication: the amount of hardware required is relatively large (as compared to serial approaches) and grows larger approximately in proportion to the square of the precision of the operands being manipulated [6]; and the parallel communication methods required to operate these parallel multipliers efficiently are much more problematic than simple serial communication patterns, especially in a VLSI environment [12].

The development of on-line systems is an attempt to combine the best features of parallel and serial

multiplication methods, so as to support very high-speed signal processing while maintaining serial communication patterns. Since the original design constructs presented in this thesis represent an on-line approach to computing, the remainder of this chapter is devoted to an introductory study of digit on-line processing systems, with particular emphasis placed on the study of on-line multipliers.

1.2 THE ON-LINE COMPUTING APPROACH

Given the problem of computing the value of some function h at a point x defined such that

$$h(x) = g(f(x)),$$

the traditional approach is to first compute the value of f(x), and then apply the function g to the operand thus obtained. The on-line approach, however, is to generate the digits that define the value of f(x) in a serial fashion such that the value of h(x) can begin to be computed before the value of f(x) can be computed completely.

In order for this chaining operation to be made applicable to a broad range of functions of scientific importance, including floating-point addition and multiplication, the digits must be processed in a most significant digit (MSD) first fashion [8] - [11], rather than in a least significant digit (LSD) first fashion as in a traditional serial device [7]. The algorithmic

mechanisms by which this MSD first serial communication pattern is maintained is very different for each of the three systems considered here; however, for each of these systems the very nature of the operations performed requires the I/O data stream to be encoded in a redundant form [9]. The complications that arise from dealing with operands in a redundant form have been considered by a number of authors [1], [2], [11]. Of particular interest here are the methods established by Ercegovac and Lang [2] for efficiently converting these redundant operands to a conventional nonredundant form. The nature of this conversion algorithm and its applications are discussed briefly in Articles 1.6.2 and 2.4.2, and an example is provided in Table 3.3.

On-line computing techniques open up several interesting avenues of investigation not afforded by more traditional approaches. For one, on-line methods might be used to chain together a large number of simple functions in order to provide a rapid means of computing the value of some complicated mathematical expression. Another possibility is that if a system is constructed such that the output data stream is made available while the corresponding input data stream is still being injected into the system, the output might be used in some way to tailor the nature of the remaining inputs. The most obvious application of this is in variable precision arithmetic. For example, in computing the

difference of two products, (ab - cd), even though the individual products may be very large, the difference could be quite small. Thus if conventional floating-point methods were to be used (i.e., approximate the value of ab, approximate the value of cd, and subtract), then all or most of the significant digits in the product might be truncated. An on-line system, however, might be constructed such that if the MSDs of the output data are insignificant then the input is keyed, so as to continue computing the lower order digits of the products that would otherwise be truncated.

One of the most important motivations for designing on-line networks, however, is that the inherently serial nature of data transfer afforded by these systems relieves the many communication bottleneck problems, such as are often encountered when large amounts of parallel data need to be transferred [8], [9]. These communication concerns are particularly critical when implementing a system in VLSI circuit form, where packing densities, and speed are often limited by communication considerations [12].

1.3 NETWORK COMPARISON CRITERIA

In order to provide the reader with a concept of the relative merits of the system design presented herein, two other on-line networks will be briefly presented for comparative purposes. Each of the three

networks considered represents a very different approach to on-line arithmetic. So as to allow the performance of these networks to be compared in a more concise and objective manner, this section lays down some basic comparison criteria.

1.3.1 GENERAL SYSTEM PROPERTIES

In Section 1.2, it was suggested that on-line methods could be used to facilitate the solution of a problem of the form

$$h(x) = g(f(x)). \qquad (1.1)$$

In a conventional arithmetic processing system, the performance exhibited when evaluating h could simply be described by the sum of the computation times for performing f and g. In an on-line system, however, the performance analysis is complicated by the fact that the computation times of f and g overlap. In order to account for this computational overlap, the concept of latency must be introduced. The latency of the function f will be defined as the delay between the introduction of the data stream x and the output of the data stream f(x). This parameter will be referred to as the chaining latency, and will be given in terms of the cycle time (i.e., the time between consecutive digit generations). Thus in the analysis of operation (1.1) we may describe the chaining latency of h as the sum of the chaining latencies of f and g.

Operation (1.1) was defined such that only a single operand was chained between functions; however, since multiplication is generally defined as a two operand function, for this operation we might define a dual chaining operation such as

$$P(P(w,x),P(y,z)) = wxyz. \qquad (1.2)$$

This dual chaining operation is capable of performing the product of four operands in only two serial steps; three serial steps would be required for this operation using single operand chaining via

$$P(w,P(x,P(y,z))) = wxyz. \qquad (1.3)$$

Operations which chain both multiplicative operands, such as (1.2), will be referred to as geometric chaining operations since the order of the partial result may rise geometrically with the number of serial operations performed; single multiplicative operand chaining, such as (1.3), will be referred to as linear chaining for identical reasons.

Geometric chaining operations have the speed advantage when computing the value of high order polynomials; each is equally applicable to linear recurrence formulations. The fully digit on-line network investigated in Section 1.3 is capable of being configured in a multiprocessor environment to perform rapid geometric chaining operations. The other two networks do not possess this capability; this loss of flexibility will be justified in terms of hardware

considerations.

1.3.2 IMPLEMENTATION CONSIDERATIONS

Once the theoretical performance parameters have been established, the practical problems involved in implementation will be considered. The medium of choice for implementation will be assumed to be MOS VLSI circuitry, since this approach affords the highest packing densities of any medium available, and is amenable to efficient pipelining techniques [13].

Engineering considerations necessitate that VLSI systems be designed using simple modular components with regular intercomponent connection patterns. In addition communication problems are of overriding importance in MOS VLSI systems. C. L. Seitz succinctly sums up the nature of these problems in the following quotation:

> Communication is expensive in chip area; indeed, most of the area of a chip is covered with wires on several levels, with transistor switches rarely taking up more than about 5 percent of the area on the lowest levels...
>
> When it comes to performance, communication is expensive in delay, both internally and between chips. In MOS technologies which exhibit the highest circuit density but a poor relationship between transistor driving capabilities and the wiring parasitics, circuit speeds are dominated by parasitic wiring capacitance. The switching speed of an MOS transistor in modern processes, with one minimum size transistor driving the gate of an adjacent transistor is in the 0.1 ns range, but if one adds a few hundred microns of wiring, the delay is increased to several nanoseconds. Also the nonzero resistance of the wires, together with the parasitic capacitance of a wire, imposes a delay in the wire itself that is becoming increasingly significant at smaller geometries...
>
> Thus, both the cost and performance metrics of

VLSI favor architectures in which communication is localized. [12]

The communication problems that will be dealt with in describing these on-line networks fall into two basic types:

1. Input distribution problems which deal primarily with the problem of amplifying and distributing the serial output of one system component into the parallel input of another; and
2. The interleaving of parallel data sets internal to the system component itself.

The first of these problems may increase the system latency and increase the amount of hardware reserved for communication purposes. The second problem, however, since it deals with communication problems internal to the processing module itself, also has the potential for limiting the digit processing rate. Based on these considerations, a discussion of the efficiency with which each of the systems considered here might be implemented in MOS VLSI form will be presented.

1.3.3 MISCELLANEOUS CONSIDERATIONS

If it is assumed that the on-line network constructs presented will be used in the development of a mathematics coprocessor design, then it is important to consider the problem of communication with the host

system. In particular, the amount of time required for, and the amount of specialized hardware dedicated to the conversion of redundant on-line system operands to nonredundant operands compatible with host system conventions, will be considered along with the converse problem of converting host system operands into the on-line number system format.

Another consideration, that will be discussed in more detail in Chapter III, is the degree to which the system can effectively support normalized arithmetic in some form. This is an important concern in a variety of scientific applications in which the normalization of operands is necessary in order to guarantee accurate solution convergence [11].

1.4 A FULLY DIGIT ON-LINE NETWORK

The first on-line multiplication network to be investigated here, is one developed by Irwin and Owens [9], based on a modification of an algorithm suggested by Ercegovac and Trivedi [10]. This approach will be briefly summarized in the following discussion.

Given radix r signed-digit fractions [1] X and Y defined by streams of digits x_i and y_i, respectively, such that the ith digit of each fraction is made available to the system at time t = i, where i is measured in clock cycless, at any given time t = j, the best approximation available for X and Y are given by

$$X_j = \sum_{i=1}^{j} x_i r^{-i} = X_{j-1} + x_j r^{-j}$$

and

$$Y_j = \sum_{i=1}^{j} y_i r^{-i} = Y_{j-1} + y_j r^{-j}.$$

The best approximation of the product that can begin to be computed using these operands at time t = j is

$$X_j Y_j = X_{j-1} Y_{j-1} + (X_j y_j + Y_{j-1} x_j) r^{-j}. \qquad (2.4)$$

This recursive function generates successively closer approximations of $X_j Y_j$ for each iteration, but in order to generate one digit of the product for each cycle time a successive approximation algorithm must be used.

The block diagram of the mantissa multiplier module is given in Fig. 1.1.

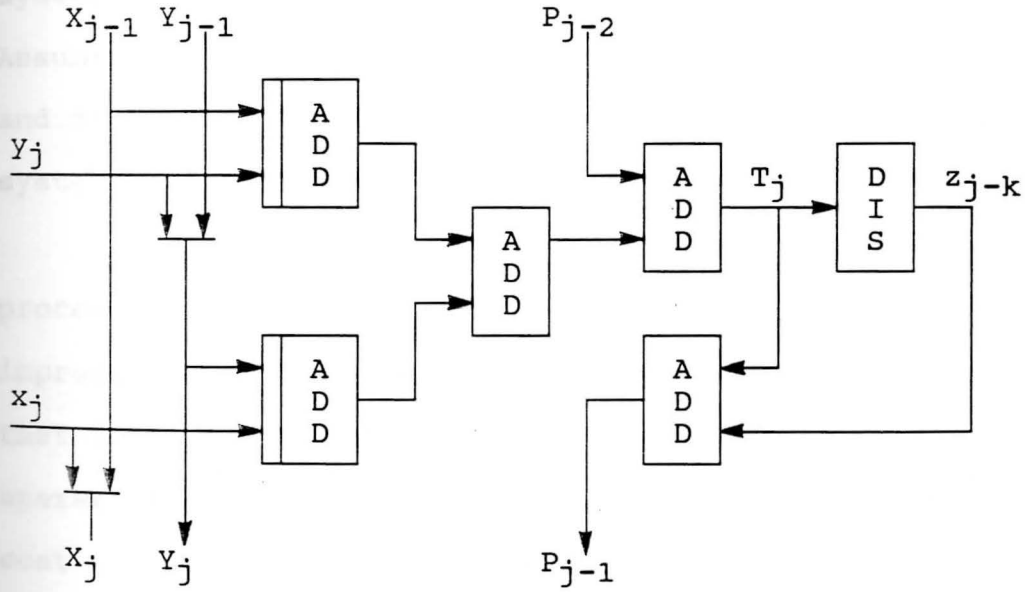


Fig. 1.1 Fully digit on-line multiplication module [9]

The components that make up the module include three Add components which perform addition or

subtraction in four gate delays; two modified Add components which generate the product of a multi-digit operand, and a single digit in six gate delays; and one DIS (discretization) table look-up component which generates a rounding, based on an inspection of the two MSDs of its input, in two gate delays. [9]

The entire time required to perform one complete iteration of the mantissa multiplier operation specified in (1.4), using the constructs of Fig. 1.1, is twenty gate delays. However, since there are only ten gate delays in the feedback loop, there exists the possibility of segmenting the system into two ten gate delay subsystems. Using such a pipelining scheme, the system could achieve a cycle time of ten gate delays. Assuming uniform clocking, and an input amplification and distribution delay of four gate delays or less, this system would exhibit a latency of three clock cycles.

Considering that the digits of this system are processed in a radix eight format, this system exhibits impressive performance, especially considering the fact that it provides contingencies for geometric chaining operations. This performance, however, comes at the cost of much more complex hardware constructs than the other designs to be considered here. It is not possible to make an exact comparison of gate or transistor counts; this data is not published for this system. However, given the operational complexities required of

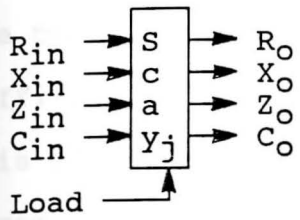
the devices used some comparisons are possible. On a per digit of word length basis, this system utilizes five radix eight redundant adders, and two sets of scaling hardware. In comparison the Paste-Up system investigated in the next section uses only two radix four redundant adders, and a single scaler. More importantly this system exhibits much more complex communication patterns that requires the parallel broadcasting of two radix eight input digits, as compared to a single radix four digit which is broadcast in the other two systems considered; and internally the system requires the interleaving of parallel data sets in two separate cases. Not only will this system require relatively large transistor outlays, but it will also exhibit lower packing densities, so as to accommodate the more complex communication patterns.

It should also be noted that, for this fully digit on-line multiplier, the low latency chaining operations considered here are only possible using unnormalized arithmetic operation which in some cases casts doubt on the accuracy of the solution set generated [11].

1.5 A PASTE-UP MULTIPLIER

The fully digit on-line system just considered [9], and the Paste-Up system [8] are both, as referenced here, detailed by the same pair of authors; the Paste-Up

system, however, is a later development and represents a very different approach to on-line computing. Paste-Up is a design system built around a small set of design primitives which can be configured in many different ways to form a host of useful arithmetic processing networks [8]. The operation of a Paste-Up scaler component is illustrated in Fig. 1.2.



$$R_o = R_{in}$$
$$X_o = X_{in}$$
$$s_j = y_j X_{in} + Z_{in} - 4C_o$$
$$Z_o = s_j + c_{in}$$
$$X_{in}, X_o \in \{-3,-2,-1,0,1,2,3\}$$
$$Z_{in}, Z_o \in \{-4,-3,-2,-1,0,1,2,3\}$$
$$y_j, s_j, C_{in}, C_o \in \{-2,-1,0,1,2\}$$

Fig. 1.2 Paste-Up scaler component.

Fig. 1.3 shows a group of these scaler components interconnected to form a multiplier.

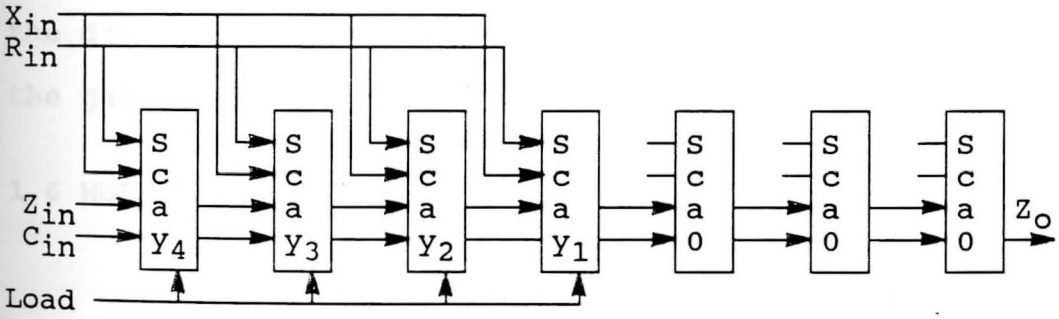
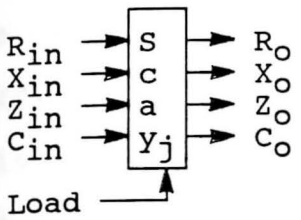


Fig. 1.3 Paste-Up multiplier block diagram [8]

This multiplier operates by successively broadcasting the multiplicand digits (X_{in}), MSD first across an array of single digit multipliers (scalers), that have been pre-loaded with the multiplier digits

system, however, is a later development and represents a very different approach to on-line computing. Paste-Up is a design system built around a small set of design primitives which can be configured in many different ways to form a host of useful arithmetic processing networks [8]. The operation of a Paste-Up scaler component is illustrated in Fig. 1.2.



$$R_o = R_{in}$$
$$X_o = X_{in}$$
$$s_j = y_j X_{in} + Z_{in} - 4C_o$$
$$Z_o = s_j + c_{in}$$
$$X_{in}, X_o \in \{-3,-2,-1,0,1,2,3\}$$
$$Z_{in}, Z_o \in \{-4,-3,-2,-1,0,1,2,3\}$$
$$y_j, s_j, C_{in}, C_o \in \{-2,-1,0,1,2\}$$

Fig. 1.2 Paste-Up scaler component.

Fig. 1.3 shows a group of these scaler components interconnected to form a multiplier.

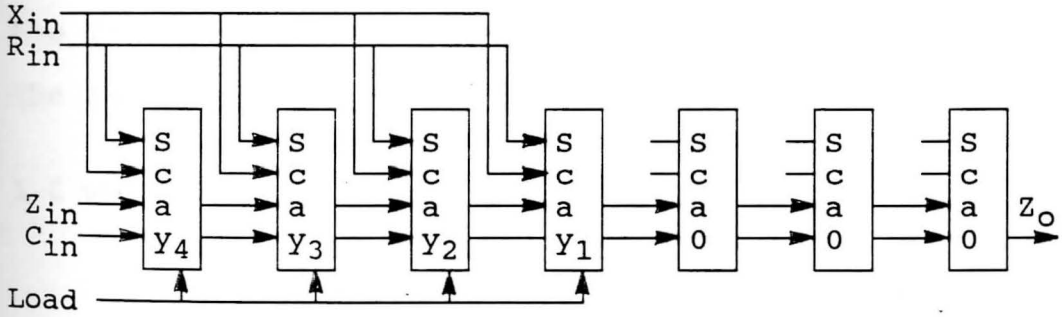


Fig. 1.3 Paste-Up multiplier block diagram [8]

This multiplier operates by successively broadcasting the multiplicand digits (X_{in}), MSD first across an array of single digit multipliers (scalers), that have been pre-loaded with the multiplier digits

(y_j). After the pre-loading of the multiplier digits, there is a delay of three clock cycles before the highest order digit of the output is made available. Since the Paste-Up system is designed to operate on a nominal ten gate delay clock cycle, this delay represents a device latency of thirty gate delays. This latency is independent of the digit length of the operands used. However, for longer multiplier operands, the problem of amplifying an internally generated signal sufficiently to allow for the parallel broadcasting of this signal across the array of scaler components, becomes increasingly costly, both in terms of hardware and delay. An advantage of the linear semi-systolic [15] layout of the array, however, is that the line delay associated with the broadcasting of this signal does not pose any intractable timing problems, because the data line delay is approximately offset by a skew in the gating signal (R_{in}) [15].

1.6 MULTIPLIER DIGIT ON-LINE SYSTEMS

As discussed in Section 1.3, an on-line multiplier can be designed such that both multiplier and multiplicand digit flow through the system in a consistent serial fashion. In Section 1.4, a multiplier was discussed that exhibited this digit flow property only with respect to the multiplicand. The original multiplier design presented here investigates the third

possibility (i.e., the on-line property is defined with respect to the multiplier digits).

Conventional serial multiplication is also executed via a serial application of multiplier digits; however, this multiplication is performed using the least significant digits of the multiplier first, and is therefore not applicable to floating point on-line multiplication [10].

One method of performing multiplier digit on-line multiplication was made possible by the introduction of redundant signed-digit arithmetic processing techniques by Avizienis [1]. Although the multiplication algorithm presented in [1] is defined in a conventional LSD first fashion, in a later work by Avizienis (quoted in [14]) the obvious observation is made that multiplication can also be performed in an MSD first manner using the same basic method.

The methods employed by Avizienis, however, will not be used here. The reasons for this divergence from the Avizienisian signed-digit approach are two fold. First, signed-digit redundant adders are slower and more complex than their conventional counterparts [1], [14]; and secondly simple methods of accelerating conventional multiplication such as using shift, and complement operations to perform radix four scaling, as used by [3] - [7] and [16], are not directly applicable to signed-digit operands.

2. The carry save adder is modified to shift left instead of right, thus causing the digits to be generated MSD first.
3. An "on-the-fly" conversion is used to convert the digits generated by the CSA into values from the set { -2, -1, 0, 1, 2 }.

The system serially generates product digits MSD first in the same format as the input digits, thus establishing a multiplier digit on-line system.

1.6.2 OPERAND CONVERSION METHODS

The Paste-Up system incorporates hardware for converting redundant on-line operands to non-redundant form, and other hardware to perform the inverse operation [8]. The conversion of redundant to nonredundant operands is achieved with minimal latency using a simple and easy to implement algorithm developed by Ercegovac and Lang [2]. In Paste-Up, the purpose for which this conversion operation is used is to facilitate communication with the host system [9].

In the design presented here, the conversion methods of Ercegovac and Lang are more central to the design, being used not only for external communication purposes, but also for the internal conversion of multiplier operands into multiplicand format. This recoding of the multiplicand into a nonredundant form permits the system to perform multiplication using

simple high-speed bit adders, rather than the more complicated slower redundant adders.

CHAPTER II

LEFT SHIFTING ACCUMULATOR THEORY

This chapter introduces the theoretical underpinnings of the left shifting accumulator (LSA) theory upon which the applications presented in Chapter III are based.

2.1 GENERAL MODULE OPERATION ALGORITHM

The algorithm presented in this section is used as the model for the implementation of all the algorithms and applications presented in this thesis. The system is based on two standard arithmetic processing operations. The first of these is the carry save adder (CSA) functions, defined in Table 2.1.

TABLE 2.1

CARRY SAVE ADDER FUNCTIONS

x	$F_C[x]$	$F_S[x]$
0	0	0
1	0	1
2	1	0
3	1	1

The second operation is the carry propagating adder operation defined as the mapping operation given in

formula (2.1).

$$X + Y + C_{in} = Z, \qquad (2.1)$$

where

$$X = \sum_{i=0}^{\sigma-1} x_i 2^i,$$

$$Y = \sum_{i=0}^{\sigma-1} y_i 2^i,$$

$$C_{in} \in \{ 0, 1 \},$$

and

$$Z = \sum_{i=0}^{\sigma} z_i 2^i.$$

The algorithm which follows is developed for an arbitrary shift factor $\sigma > 1$.

2.1.1 PROBLEM STATEMENT

Given a state value

$$A_{j-1} = \sum_{i=1}^{m} (a_i)_{j-1} 2^{-i} \mid (a_i)_j \in \{ 0, 1, 2 \} \; \forall \; i, j;$$

and inputs

$$B_j = \sum_{i=1}^{m} (b_i)_j 2^{-i} \mid (b_i)_j \in \{ 0, 1 \} \; \forall \; i, j$$

and

$$U_j = \sum_{i=0}^{\sigma} (u_i)_j 2^i \mid (u_i)_j \in \{ 0, 1 \} \; \forall \; i, j;$$

compute an output of the form

$$V_j = \sum_{i=0}^{\sigma} (v_i)_j 2^i \mid (v_i)_j \in \{ 0, 1 \} \; \forall \; i, j;$$

such that the next state value is

$$A_j = A_{j-1} 2^{\sigma} + B_j + U_j 2^{-m} - V_j. \qquad (2.2)$$

Given an initial value A_0, n recursions of expression (2.2) yields a string of output values such that

$$\sum_{j=1}^{n} V_j 2^{-\sigma j} = A_0 + \sum_{j=1}^{n} (B_j + U_j 2^{-m}) 2^{-j\sigma} - A_n 2^{-n\sigma} \quad (2.3)$$

2.1.2 SOLUTION ALGORITHM

CSA application :

$$\begin{aligned} (x_i)_j &= (a_{i+\sigma})_{j-1} + (b_i)_j \\ (s_i)_j &= F_s[(x_i)_j] \qquad \forall\, i \mid 1 \le i \le m-\sigma \\ (c_i)_j &= F_c[(x_i)_j] \end{aligned} \quad (2.4)$$

CPA application :

$$V_j = \sum_{i=1}^{\sigma-1} (a_i)_{j-1} 2^{\sigma-1-i} + (c_1)_j \quad (2.5)$$

Implicit mappings :

$$\begin{aligned} (a_i)_j &= (s_i)_j + (c_{i+1})_j \;\forall\, i \mid 1 \le i \le m-\sigma-1 \\ (a_{m-\sigma})_j &= (s_{m-\sigma})_j + (u_\sigma)_j \\ (a_i)_j &= (b_i)_j + (u_{m-i})_j \;\forall\, i \mid m-\sigma < i \le m \end{aligned} \quad (2.6)$$

2.1.3 RECURSION EXAMPLES

In Table 2.2, examples of single recursion operations of this algorithm are presented for various values of m and σ. In Table 2.3, a series of four recursions of the algorithm are performed for a $\sigma = 2$, $m = 8$ system. Each of these examples was generated using a simulation program that is listed in the appendix.

TABLE 2.2

SINGLE RECURSION EXAMPLES

$(\sigma,\ m)$		Bit Map	Analysis
(2, 8)	$4A_0$	10 211021	884
	B_1	01100011	99
	S_1	000001	
	C_1	1 11010	
	U_1	011	+ 3
	V_1	011	768
	A_1	11010122	218
(3, 9)	$8A_0$	001 102101	936
	B_1	110110001	433
	S_1	010011	
	C_1	1 01100	
	U_1	1010	+ 10
	V_1	0010	1024
	A_1	021012011	355
(4, 12)	$16A_0$	1101 12001101	57552
	B_1	011001111010	1658
	S_1	11101010	
	C_1	0 1000101	
	U_1	11111	+ 31
	V_1	01101	53248
	A_1	211020212121	5993
(5, 8)	$32A_0$	12001 222	8896
	B_1	01110011	115
	S_1	011	
	C_1	1 11	
	U_1	100110	+ 38
	V_1	100010	8704
	A_1	12210121	345

TABLE 2.3

MULTIPLE RECURSION EXAMPLE

	BIT MAP	WEIGHTED INPUT	WEIGHTED OUTPUT	
$4A_0$	01 200122	35328		
B_1	11001100	13056		
S_1	110111			
C_1	1 00011			
U_1	111	448		
V_1	010		32768	
A_1	11022211			
$4A_1$	11 022211			
B_2	01100011	1584		
S_2	011011			
C_2	0 11100			
U_2	010	32		
V_2	011		12288	
A_2	12201121			
$4A_2$	12 201121			
B_3	10001111	572		
S_3	101110			
C_3	1 00011			
U_3	001	4		
V_3	101		5120	
A_3	10122012			
$4A_3$	10 122012			
B_4	00110110	54		
S_4	101111			
C_4	0 11001			
U_4	110	6		
V_4	010		512	
A_4	21112220		396	
		51084	51084	TOTAL

In each of these examples, the output is weighted by the appropriate power of two so as to guarantee that all outputs will be integral. The gap in the bit mappings represents the placement of the radix point in the algorithmic definition of the operands.

2.2 MODULAR OPERATION

Fig. 2.1 contains a functional block diagram of a module which executes the algorithm described in Article 2.1.1. In this functional description, it is assumed that B_j was the previous parallel input value.

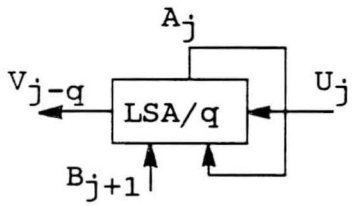


Fig. 2.1 LSA functional block diagram.

The parameter q is defined by

$$q = \tau_p - \tau_s, \tag{2.7}$$

where τ_s is the cycle period of the CSA operation described in (2.3), and τ_p is the cycle time of the CPA operation described in (2.4). If high-speed conditional sum techniques are used to generate V_j then the disparity between the operational speeds of the CSA and CPA could be eliminated ($q = 0$). However, if chains of CSA operations are used to synthesize the CPA operation, then the system would be constrained to operate with $q \geq \sigma-1$.

Since the inputs and outputs of these systems are of compatible form, these modules can be cascaded as illustrated in Fig. 2.2.

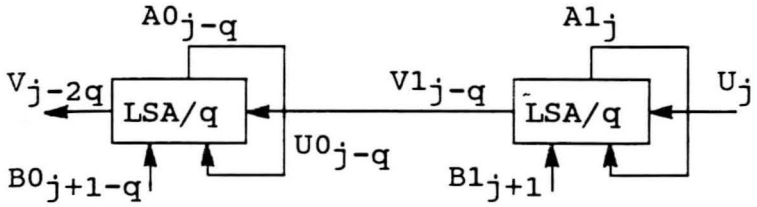


Fig. 2.2 Two module cascade.

This cascading of two identical module permits the basic recursion defined in formula (2.2) to be executed such that

$$B_j = B0_j + B1_j 2^{-m},$$

and

$$A_j = A0_j + A1_j 2^{-m}.$$

In an analogous manner, any number of modules may be cascaded to form an arbitrarily high precision processor. For example Figures 2.3, 2.4, and 2.7 are each four module configurations which process operands of the form

$$B_j = B0_j + B1_j 2^{-m} + B2_j 2^{-2m} + B3_j 2^{-3m},$$

and

$$A_j = A0_j + A1_j 2^{-m} + A2_j 2^{-2m} + A3_j 2^{-3m}.$$

Assuming that each value B_j is selected from some set of predetermined values, and further assuming that the value assigned to B_j is dependent on a selection variable y_j, then if the parameter q is reduced to zero the signal y_j must be distributed simultaneously across all of the selection modules as

illustrated in Fig. 2.3. The latency imposed by the selection module is assumed here to be equal to the CSA latency (i.e., one clock cycle).

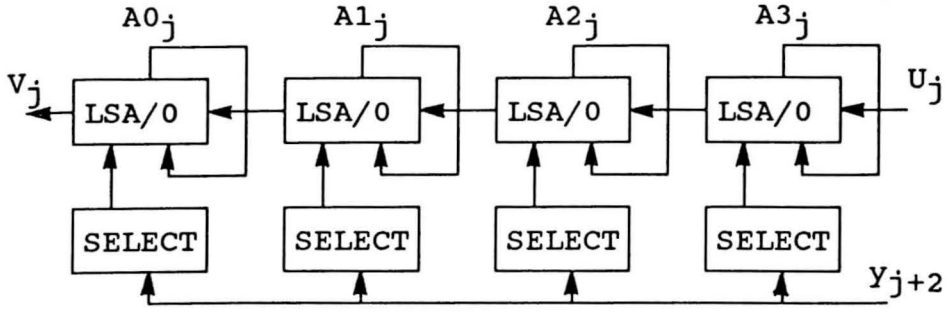


Fig. 2.3 Semi-systolic LSA configuration.

This configuration is termed semi-systolic [12] because, although nearest neighbor interconnect is maintained internally, the input signal y_j must be distributed globally. If, however, q is allowed to equal one, then the configuration illustrated in Fig. 2.4 is made possible.

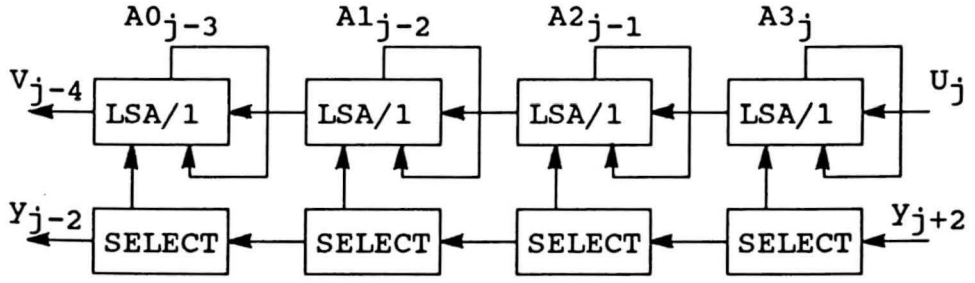


Fig. 2.4 Fully systolic LSA configuration.

This approach introduces one extra cycle of latency per module in cascade; however, the system is now fully systolic, with the select signal being sequentially propagated from module to module. As

discussed in detail in reference [12], this type of communication pattern is highly desirable in VLSI applications.

2.2.1 DELAYED FEEDBACK SYSTEMS

The possibilities afforded by inserting delay in the feedback loop of the LSA will now be investigated.

Let t be the system time measured in clock cycles, with some arbitrary reference, and j and k be defined such that

$$j = ||t/2||,$$

and

$$k = ||t/2|| + \tfrac{1}{2}.$$

Fig. 2.7 illustrates two successive operational cycles of an LSA with unary delay inserted in the feedback loop.

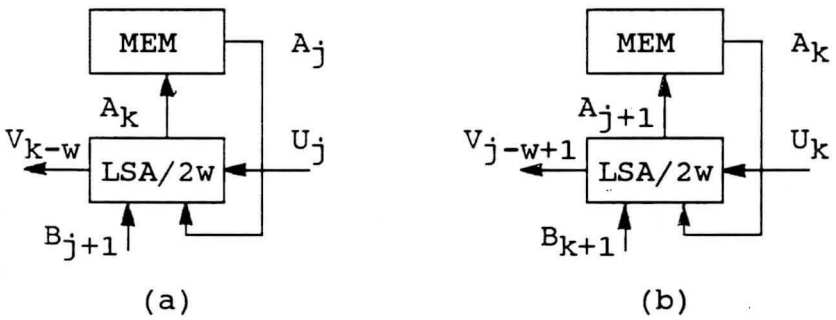


Fig. 2.5 Delayed feedback LSA operation.

This device exhibits the interesting property of toggling between two independent data sets, such that each data set is operated upon every alternate clock cycle. Fig. 2.6 illustrates the operation of a two element cascade of LSAs with delayed feedback.

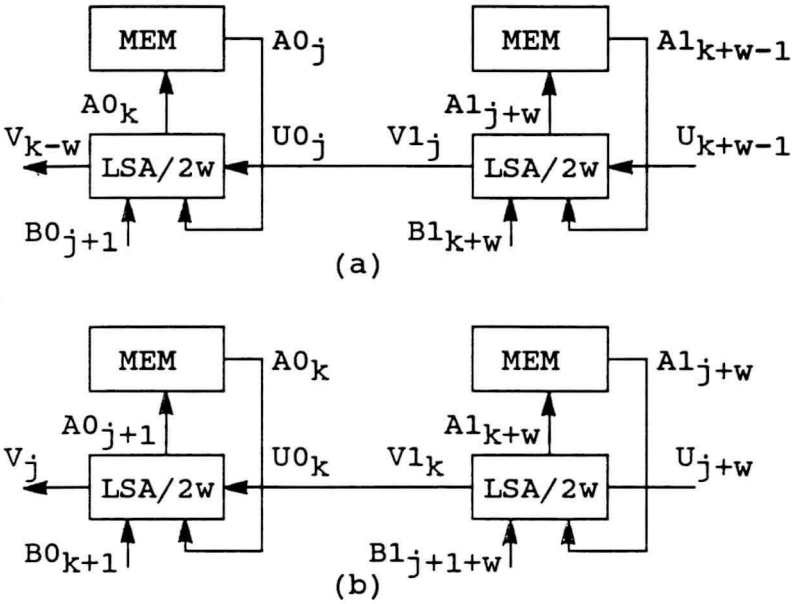


Fig. 2.6 Delayed feedback LSA two element cascade.

An important property of this type of LSA configuration is that it permits a system to be defined in such a way as to maintain fully systolic communication patterns, while still generating an MSD first on-line output data stream whose latency is independent of the number of modules in the cascade. Fig. 2.7 provides an illustration of the operation of such a system.

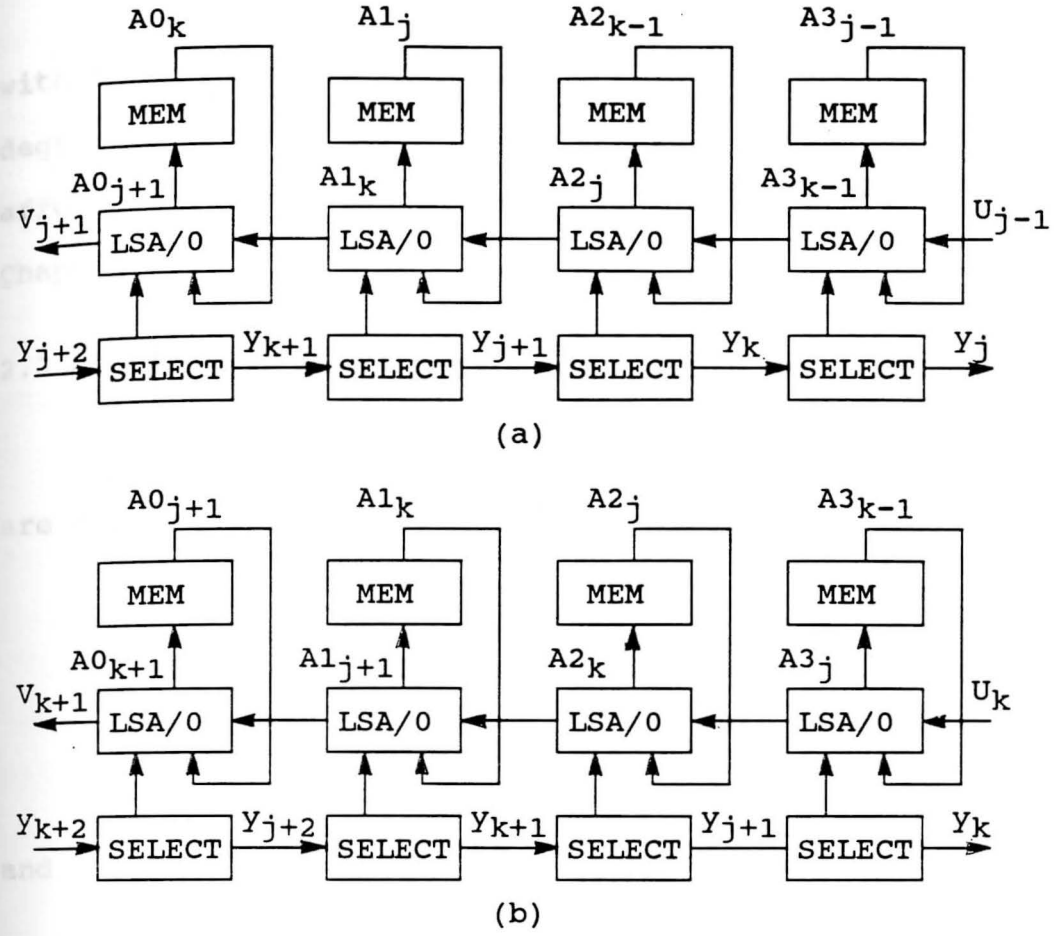


Fig. 2.7 Alternative systolic LSA configuration.

Due to the added latency in the feedback path, this LSA configuration has an operational cycle about twice as long as the systolic configuration illustrated in Fig. 2.4; however, the total effective throughput remains unchanged due to concurrency. This system has a great advantage when operating in a high-precision on-line environment since the output latency is independent of the number of modules in the cascade.

By inserting a series of delay units in the feedback loop of a q = 0 LSA, a system can be designed to operate on any number of data sets concurrently,

with the cycle time being directly proportion to the degree of concurrency supported. Some applications afforded by this design flexibility will be discussed in Chapter III.

2.3 SIGNED DATA HANDLING

In Article 2.1.1 the parameters A_j, B_j, and V_j are defined by

$$A_j = \sum_{i=1}^{m} (a_i)_j 2^{-i},$$

$$B_j = \sum_{i=1}^{m} (b_i)_j 2^{-i},$$

and

$$V_j = \sum_{i=0}^{\sigma} (v_i)_j 2^{i}. \qquad (2.7)$$

It is easily established that the algorithm defined in Article 2.1.2 is equally applicable to operands of the form

$$A_j = -\tfrac{1}{2} + \sum_{i=1}^{m} (a_i)_j 2^{-i},$$

$$B_j = -\tfrac{1}{2} + \sum_{i=1}^{m} (b_i)_j 2^{-i},$$

and

$$V_j = -2^{\sigma-1} + \sum_{i=0}^{\sigma} (v_i)_j 2^{i}. \qquad (2.8)$$

This simple offsetting of the operands in no way implies any change in the Boolean logic levels of the digits in the operands, but simply provides a reinterpretation of existing logical constructs. The

operand redefinition given in (2.8) will be referred to as the signed operand set, in order to distinguish them from the unsigned set given in (2.7). This signed set permits the input B to be in the range

$$-\tfrac{1}{2} \leq B_j < +\tfrac{1}{2},$$

while generating output digits such that

$$V_j \in \{ -2^{\sigma-1}, \ldots, -1, 0, 1, \ldots, 2^{\sigma} - 2^{\sigma-1} - 1 \}.$$

2.3.1 EXAMPLES RECONSIDERED

Tables 2.2 and 2.3 provide examples of unsigned applications of the LSA algorithm; in Tables 2.4 and 2.5 this data is reinterpreted as signed operations.

TABLE 2.4

SIGNED SINGLE RECURSION EXAMPLES

$(\sigma,\ m)$		Bit Map	Analysis
(2, 8)	$4A_0$	10 211021	372
	B_1	01100011	-29
	S_1	000001	
	C_1	1 11010	
	U_1	011	+ 3
	V_1	011	256
	A_1	11010122	90
(3, 9)	$8A_0$	001 102101	-1112
	B_1	110110001	177
	S_1	010011	
	C_1	1 01100	
	U_1	1010	+ 10
	V_1	0010	-1024
	A_1	021012011	99
(4, 12)	$16A_0$	1101 12001101	24784
	B_1	011001111010	-390
	S_1	11101010	
	C_1	0 1000101	
	U_1	1111	+ 31
	V_1	01101	20480
	A_1	211020212121	3945
(5, 8)	$32A_0$	12001 222	4800
	B_1	01110011	-13
	S_1	011	
	C_1	1 11	
	U_1	100110	+ 38
	V_1	100010	4608
	A_1	12210121	217

TABLE 2.5

SIGNED MULTIPLE RECURSION EXAMPLE

	BIT MAP	WEIGHTED INPUT	WEIGHTED OUTPUT	
$4A_0$	01 200122	2560		
B_1	11001100	4864		
S_1	110111			
C_1	1 00011			
U_1	111	448		
V_1	010		0	
A_1	11022211			
$4A_1$	11 022211			
B_2	01100011	-464		
S_2	011011			
C_2	0 11100			
U_2	010	32		
V_2	011		4096	
A_2	12201121			
$4A_2$	12 201121			
B_3	10001111	60		
S_3	101110			
C_3	1 00011			
U_3	001	4		
V_3	101		3072	
A_3	10122012			
$4A_3$	10 122012			
B_4	00110110	-74		
S_4	101111			
C_4	0 11001			
U_4	110	6		
V_4	010		0	
A_4	21112220		268	
		7436	7436	TOTAL

2.4 OUTPUT RECODING PROCEDURES

Output digits of the form generated by (2.8) are cumbersome to deal with. The following recoding procedures generate an output data stream which is in a minimally redundant symmetrical signed-digit form.

Let

$$W_j = \sum_{i=0}^{\sigma-1} (v_{-i})_j 2^i \tag{2.9}$$

and

$$p_j = W_j + (v_{-\sigma})_{j+1}, \tag{2.10}$$

so that $p_j \in \{ -2^{\sigma-1}, \ldots, -1, 0, 1, \ldots, 2^{\sigma-1} \}$, with the boundary conditions established by initializing W_0 in the range

$$W_j \in \{ -2^{\sigma-1}, \ldots, -1, 0, 1, \ldots, 2^{\sigma-1}-1 \},$$

implied by (2.9); and by defining

$$p_n = W_n$$

where p_n is the last digit generated by the recursion. Using this procedure the value P may be defined such that

$$P = \sum_{j=0}^{n} p_j 2^{-\sigma j} = W_0 + A_0 + \sum_{j=1}^{n} (B_j + U_j 2^{-m}) 2_{-\sigma j} - \delta, \tag{2.11}$$

where

$$\delta = A_n 2^{-n\sigma}, \tag{2.12}$$

which guarantees

$$-2^{-n\sigma-1} \leq \delta < 3(2^{-n\sigma-1}).$$

2.4.1 ZERO INITIALIZATION CASE

An important special case of (2.11) is the case for which A_0 and W_0 are zero so that P can be defined such that

$$P = \sum_{j=1}^{n} (B_j + U_j 2^{-m}) 2^{-\sigma j} - \delta. \qquad (2.13)$$

If the input data set is such that it can be guaranteed that for some value δ in the range implied by (2.12) there exists a value P such that

$$|P| \leq \sum_{j=1}^{n} (2^{\sigma-1}) 2^{-\sigma j}. \qquad (2.14)$$

then P can be guaranteed to be recodable into an n digit representation such that

$$P = \sum_{j=1}^{n} p_j 2^{-\sigma j}. \qquad (2.15)$$

In order to guarantee that the insignificant digit p_0 is not generated, the following recoding procedure makes use of a selective carry suppression algorithm.

Let a Boolean flag ϕ_j be defined such that

$$\phi_0 = 1,$$

$$\phi_{j+1} = (v_{-\sigma})_{j+1} \phi_j \quad \text{E } j \geq 0. \qquad (2.16)$$

Using this flag the digits that define P can be generated in the form given in (2.15) such that

$$p_j = W_j + \phi_j (v_{-\sigma})_j 2^{\sigma-1} + \bar{\phi}_j (v_{-\sigma})_{j+1}. \qquad (2.17)$$

2.4.2 CONVERSION TO NONREDUNDANT FORM

The problem of converting redundant operands, such as P, into a nonredundant form has been traditionally handled using carry propagating adders. A much simpler method of performing this conversion, however, has been suggested by Ercegovac and Lang [2]. In the Ercegovac/Lang algorithm, the conversion of redundant to nonredundant operands is achieved "on-the-fly," i.e., instead of waiting for all the redundant digits to become available before beginning the conversion, the conversion is executed as an ongoing process which commences as soon as the first digit of the result becomes available.

This system was designed specifically to meet the operand conversion needs of on-line arithmetic processing systems, such as those systems developed in this thesis. It is capable of very high speed operation, approximately equivalent to that of a CSA, and can be implemented using simple shift register constructs. For more detailed information on these conversion methods refer to reference [2].

CHAPTER III

DIGITAL SIGNAL PROCESSING APPLICATIONS

This chapter applies the theory developed in Chapter II to practical arithmetic processing problems.

3.1 MULTIPLICATION FUNDAMENTALS

This section defines the nature of the operands that will be utilized in the application of LSA theory to the problem of multiplication, and briefly reviews multiplication operand processing techniques.

3.1.1 PROBLEM DEFINITION

Given a multiplicand X and a multiplier Y, the product XY will be defined here as a summation of n partial products such that

$$XY = \sum_{j=1}^{n} Xy_j 2^{-\sigma j}, \tag{3.1}$$

where

$$X = -x_0 + \sum_{i=1}^{k-1} x_i 2^{-i}, \tag{3.2}$$

and

$$Y = \sum_{j=1}^{n} y_j 2^{-\sigma j}. \tag{3.3}$$

3.1.2 MULTIPLIER RECODING

If the multiplier is initially defined such that

$$2Y = -\ddot{y}_0 + \sum_{i=1}^{n\sigma-1} \ddot{y}_i 2^{-i},$$

these digits may be recoded into the form required in (2.3) via the operation:

$$y_{j+1} = -\ddot{y}_{j\sigma} 2^{\sigma-1} + \sum_{i=1}^{\sigma-1} \ddot{y}_{j\sigma+i} 2^{\sigma-i} + \ddot{y}_{j\sigma} \qquad (3.4)$$

$$\forall\ j \mid 1 \leq j < n-1,$$

with the boundary condition set by

$$y_1 = -\ddot{y}_0 2^{\sigma-1} + \sum_{i=1}^{\sigma-1} \ddot{y}_i 2^{\sigma-i} + \ddot{y}_{\sigma+1}$$

and $$y_n = -\ddot{y}_{n\sigma-\sigma} 2^{\sigma-1} + \sum_{i=1}^{\sigma-1} \ddot{y}_{n\sigma-\sigma+i} 2^{\sigma-i}.$$

This digit definition stipulates that

$$y_j \in \{ -2^{\sigma-1}, \ldots, -1, 0, 1, \ldots, 2^{\sigma-1} \}\ \forall\ j.$$

3.1.3 PARTIAL PRODUCT GENERATION

If the shift factor $\sigma = 2$ is chosen, the multiplier digits are defined such that

$$y_j \in \{ -2, -1, 0, 1, 2 \}.$$

With this multiplier digit set the partial products Xy_j can be easily generated using simple shifting and complementing operations. If a larger value of σ is chosen then cumbersome carry propagating adder hardware must be used to generate the partial product set; this

fact effectively limits practical partial product generation schemes to the radix four case [5].

In Table 3.1 the digits of the operand B_j are defined as a function of X and y_j for a $\sigma = 2$ system such that

$$B_j = Xy_j = \sum_{i=1}^{k+1} (b_i)_j 2^{-i} + 2^{-k-1}\beta. \qquad (3.5)$$

TABLE 3.1

PARTIAL PRODUCT DIGIT MAPPINGS FOR RADIX FOUR MULTIPLICATION

y_j	$(b_1)_j$	$(b_i)_j$	β_j
+2	x_0	x_i[a]	0
+1	x_0	x_{i-1}	0
+0	1	0	0
-0	0	1	1
-1	x_0	x_{i-1}	1
-2	x_0	x_i[a]	1

[a]Let $x_{k+1} = 0$.

3.2 LSA MULTIPLICATION

Obviously if we assign

$$(u_0)_j = \beta_j \text{ or alternatively } (u_2)_{j+1} = \beta_j, \qquad (3.6)$$

and $k = m-1$,

the zero initialization case of the LSA operation given in (2.13) is directly amenable to the multiplication problem:

$$XY = \sum_{j=1}^{n} B_j 4^{-j}. \qquad (3.7)$$

However, an m bit LSA processing module, as defined in Article 2.1.2, is only capable of processing an m-1 bit multiplicand using either of the operand mappings described in (3.6). An alternative operand mapping which supports an m+1 bit multiplicand is given in (3.8).

$$\begin{aligned} (u_1)_{j+1} &= (b_k)_j \\ (u_2)_{j+1} &= (b_{k+1})_j \\ (u_0)_{j+2} &= \beta_j \end{aligned} \qquad (3.8)$$

3.2.1 MULTIPLICATION EXAMPLE

Table 3.2 provides an example of multiplication of a nine bit two's complement multiplicand

$$X = 0.11010111_2 = 215 \cdot 2^{-8}$$

with a minimally redundant signed-digit multiplier

$$Y = 0.\ 2\text{-}1\text{-}2\ 1\ 0\ 0\ 0\ 0_4 = 105 \cdot 2_{-8},$$

using the zero initialization case of the LSA algorithm defined in expression (2.13) for m = 8, with the operand digit mapping being provided by Table 3.1 and the mappings given in (3.8).

Table 3.2 illustrates the operations performed by the LSA algorithm, in executing this multiplication, using the following format.

$4A_{j-1}$		a_1 a_2	a_3 a_4		a_5 a_6		a_7 a_8			
B_j			b_1 b_2		b_3 b_4		b_5 b_6		b_7 b_8	
U_j							u_2		u_1 u_0	
$2^m V_j + A_j$	V_{-2}	$V_{-1}V_0$	$\hat{a}_1$ $\hat{a}_2$		$\hat{a}_3$ $\hat{a}_4$		$\hat{a}_5$ $\hat{a}_6$		$\hat{a}_7$ $\hat{a}_8$	

In Table 3.3 the conversion methods of Ercegovac and Lang [2] are used to recode the resulting product,

$$P = 0.\ 1\ 2\text{-}2\ 0\ 1\ \text{-}1\ 0\ \text{-}1_4 = 22{,}575 \cdot 2^{-16},$$

into a two's complement normalized form such that

$$2P = 0.101100000101111_2.$$

Successive approximations are made of the value of P such that

$$P_j = \sum_{i=1}^{j} p_i 4^{-i}.$$

By maintaining storage of both P_j and $P_j - 4^{-j}$, the system permits these successive approximations to be made without using any carry propagation hardware [2].

TABLE 3.2

MULTIPLICATION EXAMPLE

```
        10 00 00 00
           11 10 10 11                                          y1 =  2
                 0 00
        ------------------
     0 10 11 10 10 11
           01 00 10 10                                          y2 = -1
                 0 10
        ------------------
        0 11 11 11 01 20
              00 01 01 00                                       y3 = -2
                    0 00
        ------------------
p1 =  1    0 11 11 11 01 00
                 10 11 01 01                                    y4 =  1
                       1 01
        ------------------
p2 =  2       1 00 01 20 02 02
                    10 00 00 00                                 y5 =  0
                          1 11
        ------------------
p3 = -2          0 10 10 10 11 11
                       10 00 00 00                              y6 =  0
                             0 00
        ------------------
p4 =  0             0 11 00 11 11 00
                          10 00 00 00                           y7 =  0
                                0 00
        ------------------
p5 =  1                0 01 01 11 00 00
                             10 00 00 00                        y8 =  0
                                   0 00
        ------------------
p6 = -1                   0 10 01 00 00 00
                                10 00 00 00                     y9 =  0
                                      0 00
        ------------------
p7 =  0                      0 01 10 00 00 00

p8 = -1
```

TABLE 3.3

REDUNDANT TO NONREDUNDANT OUTPUT CONVERSION
USING ERCEGOVAC/LANG ALGORITHM

j	p_j	$2P_j$	$2(P_j - 4^{-j})$
1	1	0.1	0.0
2	2	0.110	0.101
3	-2	0.10110	0.10101
4	0	0.1011000	0.1010111
5	1	0.101100001	0.101100000
6	-1	0.10110000011	0.10110000010
7	0	0.1011000001100	0.1011000001011
8	-1	0.101100000101111	0.101100000101110

3.3 LSA DIVISION

The idea of performing division using a left shifting accumulator is not a new one. In fact, division is inherently an on-line process, inasmuch as the quotient digits are most conveniently generated MSD first.

In reference [6], a division procedure is outlined such that the partial remainders are stored in a redundant accumulator similar to those considered here. In this system, each successive radix four quotient digit is generated based on an inspection of the most significant digits of the partial remainder such that $y_j \in \{ -2, -1, 0, 1, 2 \}$. This quotient digit

is then multiplied by the divisor X, and the resulting product (Xy_j) is subtracted from the previous partial remainder (A_{j-1}) to form a new partial remainder (A_j).

This process can obviously be implemented using the same basic recursive operation that was developed for multiplication in the previous section. However, unlike multiplication, in which the next selection operand y_{j+1} is known independent of the output of the system A_j, division requires that the partial remainder be inspected after each cycle in order to determine the next appropriate selection value. In the ILLIAC design [4] this inspection procedure is accomplished by first recoding the six most significant digits of the partial remainder into a nonredundant form and then using Table look-up methods. This procedure introduces a large amount of latency into the feedback loop of the system, thus this division algorithm could only be expected to cycle at a fraction of the rate associated with the multiplication algorithm considered in Section 3.2.

In order to compensate for this slower cycle rate, the delayed feedback systems discussed in Article 2.3.1 might be used, such that the delay imposed by the accumulator inspection techniques is balanced by a series of delays in the accumulator feedback loop. This could allow the CSAs to still cycle at maximum rate by operating on a number of data sets concurrently. One

would expect such a division system to operate with the same total effective throughput as an LSA multiplier in spite of the slower individual operand processing rates. A detailed simulation of the operation of such a division system has not yet been developed. This is left as a topic for future research into the operation of LSA networks.

CHAPTER IV

CONCLUSION

4.1 SUMMARY

This thesis has investigated possible applications of a class of devices referred to here as left shifting accumulators. The general left shifting accumulation algorithm was precisely defined in terms of well known arithmetic operations. Various configurations of modules developed from this algorithm were presented and their operational characteristics discussed. Included in this development were configurations to support the highly desirable systolic communication pattern [15]. Through the use of computer simulation data, it was demonstrated that the operation of these devices could be applied equally well to both signed and unsigned operations.

The algorithm was then applied to the development of a multiplier which performs radix four multiplication in a multiplier digit on-line fashion with a possible maximum cycle rate equivalent to that of a carry save adder. Examples of system operation were provided using computer simulation data.

Facilities for the conversion of the output data stream into a nonredundant form are provided using the

methods of Ercegovac and Lang [2]. This system generates successive approximations of the value of output data stream in two's complement form, using simple shift functions.

Evidence to support the contention that LSA systems could be configured to support high efficiency division operations is presented, although simulation data for this operation is not yet available.

4.2 FUTURE RESEARCH DIRECTIONS

In Chapter I, it was suggested that the LSA configurations developed here might be used in high-speed digit signal processing applications. In order for this to become a reality, much more research and development will be necessary. At the module level, the design task is simplified by the algorithm's usage of already well developed arithmetic functions. At the system level, the various possible ways in which the modules can be configured provides a great deal of flexibility to the IC layout engineer. Ultimately, the choice of an optimum network configuration will be dependent upon the particular application to which the system is being applied.

Other research directions should include the development of redundant adders that could be used to combine data streams, so as to support the operation of addition using simple serial constructs. (Paste-Up [8]

already includes such a redundant adder, but it is likely that this device would have to be modified in order to support the higher communication bandwidth possible using LSA multiplication systems.)

The systems presented in this thesis are new and for the most part hypothetical in nature. This makes it difficult to predict with any certainty into what form future research will mold these concepts. Section 4.3, however, concludes this thesis with a discussion of two general types of digital signal processing applications that seem to hold promise.

4.3 TWO TYPES OF APPLICATIONS

In Section 2.2, two different systolic module configurations were developed. The first of these is illustrated in Fig. 2.4. The primary advantage exhibited by this network is a fast (single clock cycle) operational period. Its most notable disadvantage is that the latency of the network is dependent upon the number of modules in the cascade. The network illustrated in Fig. 2.7 also exhibits systolic communication patterns; however, its other operational properties are very much different from that of the system in Fig. 2.4. This delayed feedback system has a cycle rate double that of its direct feedback counterpart; however, overall the system maintains the same high throughput rate through the use of concurrent

processing. The main advantage of this configuration is that it has a very low latency, and this low latency is maintained independent of the number of modules in the cascade.

If a system is to be designed with the requirement that it multiply together two floating-point operands, returning the final result as quickly as possible, then Fig. 2.4 is obviously the most appropriate choice of configurations. If, on the other hand, an on-line arithmetic processing network that takes full advantage of multiprocessing capabilities to perform long chains of high precision operations before returning a final result is desired, then Fig. 2.7 is unquestionably the superior circuit for the job.

These two cases represent extreme examples; in real world applications the choice between high cycle speed and low latency might not be as easy to make. What is important, however, is that whichever option is chosen an LSA configuration with systolic communication patterns can be applied to the task. To put the problem in perspective with respect to existing technologies, the slow cycle rate associated with Fig. 2.7 can still be expected to be significantly faster than Paste-Up's ten gate delay clock cycle [8].

APPENDIX

LIST OF SIMULATION PROGRAM

The following simulation program was employed to generate the data presented in tables 2.2, 2.3, 2.4, 2.5, and 3.2. The program is written in the Basic programming language, and was executed on an Atari 800XL personal computer.

```
10 REM LSA SYSTEM SIMULATION
11 REM BY JESSE BOOHER
12 REM
20 REM GENERAL INITIALIZATIONS
30 DIM A(16), B(16), S(14), C(14), U(6), V(6), FS(3),
FC(3), X(10), Y(8)
35 DIM A$(16), B$(16), U$(7), S$(14), C$(14), V$(7),
N$(5), M$(5), P$(80), BL$(15), X$(10)
40 FS(0)=0:FC(0)=0
50 FS(1)=1:FC(1)=0
60 FS(2)=0:FC(2)=1
70 FS(3)=1:FC(3)=1
80 BL$="                "
90 GOTO 3000:REM ACCESS MENU
100 REM PARTIAL PRODUCT GENERATION
110 X(M+1)=0
120 SH=1-INT(ABS(Y/2)+0.1)
130 INV=SGN(Y)
132 IF INV=-1 THEN B(1)=X(0)
134 IF INV=0 THEN B(1)=1
136 IF INV=1 THEN B(1)= NOT X(0)
137 B$(1,1)=STR$(B(1))
140 FOR I=2 TO M+2
150 IF INV=-1 THEN B(I)= NOT X(I-SH-1)
160 IF INV=0 THEN B(I)=0
170 IF INV=1 THEN B(I)=X(I-SH-1)
175 B$(I,I)=STR$(B(I))
180 NEXT I
189 BETA=0
190 IF Y<0 THEN BETA=1
200 U(2)=U2A:U$(3,3)=STR$(U2A)
210 U(1)=U1A:U$(2,2)=STR$(U1A)
220 U(0)=U0A:U$(1,1)=STR$(U0A)
230 U2A=U2B
240 U1A=B(M+1)
250 U0A=B(M+2)
260 U2B=BETA
270 RETURN
300 REM MULTIPLICATION INITIALIZATION
305 A(1)=1
310 FOR I=2 TO M
320 A(I)=0:A$(I,I)="0"
330 NEXT I
340 U2A=0:U1A=0:U0A=0:U2B=0
350 RETURN
400 REM MULTIPLIER RECODING
410 F(M)=0
420 FOR J=0 TO N-1
430 E(J+1)=-F(2*J)+F(2*J+1)+F(2*J+2)
440 NEXT J
450 RETURN
500 REM ACCUMULATION
510 FOR I=1 TO M-SIG
520 X=A(I+SIG)+B(I)
```

```
530 S(I)=FS(X)
540 C(I)=FC(X)
550 NEXT I
560 RETURN
600 REM V DIGIT GENERATION
610 C=C(1)
620 FOR I=SIG TO 1 STEP -1
630 X=C+A(I)
640 V(SIG-I)=FS(X)
650 C=FC(X)
660 NEXT I
670 V(SIG)=C
680 RETURN
700 REM IMPLICIT MAPPINGS
710 FOR I=1 TO M-SIG-1
720 A(I)=S(I)+C(I+1)
730 NEXT I
740 X=M-SIG
750 A(X)=S(X)+U(SIG)
760 FOR I=1 TO SIG
770 A(X+I)=B(X+I)+U(SIG-I)
780 NEXT I
790 RETURN
800 REM PRINT A(I)
810 FOR I=1 TO M
820 A$(I,I)=STR$(A(I))
830 NEXT I
840 RETURN
850 REM PRINT S(I)
860 FOR I=1 TO M-SIG
870 S$(I,I)=STR$(S(I))
880 NEXT I
890 RETURN
900 REM PRINT C(I)
910 FOR I=1 TO M-SIG
920 C$(I,I)=STR$(C(I))
930 NEXT I
940 RETURN
950 REM PRINT V
960 FOR I=SIG TO 0 STEP -1
970 V$(SIG-I+1,SIG-I+1)=STR$(V(I))
980 NEXT I
990 RETURN
1000 REM INPUT A
1005 READ A$
1010 FOR I=1 TO M
1020 A(I)=VAL(A$(I,I))
1030 NEXT I
1040 RETURN
1050 REM INPUT B
1055 READ B$
1060 FOR I=1 TO M
1070 B(I)=VAL(B$(I,I))
1080 NEXT I
```

```
1090 RETURN
1100 REM COMPUTE (2^M)*B
1110 B=0
1120 FOR I=1 TO M
1130 B=B+B+B(I)
1140 NEXT I
1150 RETURN
1200 REM COMPUTE (2^M)*A
1210 A=0
1220 FOR I=1 TO M
1230 A=A+A+A(I)
1240 NEXT I
1250 RETURN
1300 REM COMPUTE V
1310 V=0
1320 FOR I=SIG TO 0 STEP -1
1330 V=V+V+V(I)
1340 NEXT I
1350 RETURN
1400 REM COMPUTE U
1410 U=0
1420 FOR I=SIG TO 0 STEP -1
1430 U=U+U+U(I)
1440 NEXT I
1450 RETURN
1500 REM INPUT U
1505 READ U$
1510 FOR I=0 TO SIG
1520 U(SIG-I)=VAL(U$(I+1,I+1))
1530 NEXT I
1540 RETURN
1600 REM RIGHT JUSTIFICATION
1605 M$="      "
1610 LN=LEN(N$)
1615 FOR I=1 TO LN
1620 M$(5-LN+I,5)=N$(I,I)
1625 NEXT I
1630 RETURN
1700 REM FIND V
1710 FOR I=0 TO SIG
1720 V(SIG-I)=VAL(V$(I+1,I+1))
1730 NEXT I
1740 RETURN
2100 REM LSA RECURSION
2150 GOSUB 4020:REM PRINT A*(2^SIG)
2155 GOSUB 4065:REM LOAD AND PRINT B
2160 GOSUB 500:REM ACCUMULATE
2190 GOSUB 4180:REM DRAW DIVIDER LINE
2200 GOSUB 4207:REM PRINT S AND C
2225 GOSUB 4105:REM LOAD AND PRINT U
2230 GOSUB 4180:REM DRAW DIVIDER LINE
2240 GOSUB 600:GOSUB 950:GOSUB 1300:REM GENERATE V
2250 GOSUB 4300:REM PRINT V
2260 GOSUB 700:REM REGENERATE ACCUM.
```

```
2270 GOSUB 4400:REM PRINT A
2280 GOSUB 5100:REM SCREEN DUMP
2300 RETURN
2500 REM SYSTEM INITIALIZATION
2505 PRINT "DO YOU WANT SIGNED OUTPUT";:INPUT N$
2506 SN=(N$(1,1)="Y")
2507 GOSUB 5200
2510 READ N
2520 IF N=0 THEN RETURN
2530 READ M,SIG
2540 GOSUB 1000
2550 FOR J=1 TO N
2560 PRINT ")";
2570 GOSUB 2100
2580 NEXT J
2590 GOTO 2510
3000 REM MENU
3005 SN=1
3010 PRINT ")"
3020 PRINT
3030 PRINT "CHOOSE OPERATION DESIRED:"
3040 PRINT "      1) MIXED EXAMPLES."
3045 PRINT "      2) MULTIPLE RECURSION EXAMPLE."
3050 PRINT "      3) MULTIPLICATION SIMULATION."
3060 PRINT "      4) EXIT."
3065 PRINT
3070 INPUT I9
3080 IF I9=1 THEN RESTORE 9000:GOSUB 2500
3090 IF I9=2 THEN RESTORE 9200:GOSUB 2500
3100 IF I9=3 THEN RESTORE 9500:GOSUB 7000
3110 IF I9=4 THEN END
3200 GOTO 3000
4000 REM I/O SUBROUTINES
4020 REM PRINT A*(2^SIG)
4025 N$=STR$(2^SIG):GOSUB 1600
4030 PRINT M$(4,5);"*A";J-1;"    ";A$(1,SIG);" ";
4050 GOSUB 1200:A=A-SN*2^(M-1)
4055 N$=STR$(A*2^SIG):GOSUB 1600
4060 PRINT A$(SIG+1,M);
4061 FOR I=1 TO SIG+1:PRINT " ";:NEXT I:PRINT M$
4062 RETURN
4065 REM READ AND PRINT B
4070 IF MULT=0 THEN GOSUB 1050
4080 PRINT "    B";J;:FOR I=1 TO SIG+4:PRINT " ";:NEXT I
4095 GOSUB 1100:B=B-2^(M-1)*SN
4097 N$=STR$(B):GOSUB 1600
4100 PRINT B$(1,M);" ";M$
4102 RETURN
4105 REM READ AND OUTPUT U
4110 IF MULT=0 THEN GOSUB 1500
4120 PRINT "    U";J;BL$(1,M+3);
4130 GOSUB 1400
4135 N$=STR$(U):GOSUB 1600
4140 PRINT U$;" ";M$
```

```
4150 RETURN
4180 REM DRAW DIVIDER LINE
4190 PRINT "        ";
4200 FOR I=1 TO M+SIG+2:PRINT "-";:NEXT I:PRINT "
";"-----"
4205 RETURN
4207 REM OUTPUT S AND C
4208 GOSUB 850
4209 GOSUB 900
4210 PRINT "    S";J;BL$(1,4+SIG);S$(1,M-SIG)
4220 PRINT "    C";J;BL$(1,2+SIG);C$(1,1);" ";C$(2,M-SIG)
4230 RETURN
4300 REM OUTPUT V
4310 V=V-2^(SIG-1)*SN
4320 PRINT "    V";J;"   ";V$(1,SIG+1);
4340 N$=STR$(V*2^M):GOSUB 1600
4350 PRINT BL$(1,M+2);
4360 PRINT M$
4370 RETURN
4400 REM OUTPUT A
4420 PRINT "    A";J;BL$(1,SIG+4);
4430 GOSUB 1200:A=A-SN*2^(M-1)
4440 N$=STR$(A):GOSUB 1600
4450 GOSUB 800
4460 PRINT A$;" ";M$
4470 RETURN
5000 REM SCREEN DUMP
5005 LPRINT :LPRINT
5010 FOR I=0 TO 8
5015 FOR K=1 TO 40
5020 T1=PEEK(39999+40*I+K)
5025 P$(K,K)=CHR$(32+T1)
5030 NEXT K
5040 LPRINT BL$,P$
5060 NEXT I
5070 RETURN
5100 REM SD OPTION
5110 PRINT :PRINT :PRINT
5115 IF AP THEN 5000
5120 PRINT "DUMP OUTPUT TO PRINTER";
5130 INPUT N$
5140 IF N$(1,1)="Y" THEN 5000
5150 RETURN
5200 REM AUTO PRINT OPTION
5300 PRINT "AUTOMATIC PRINTING";
5310 INPUT N$
5320 AP=(N$(1,1)="Y")
5330 RETURN
7000 REM MULTIPLICATION SIMULATION
7005 MULT=1:SN=1
7010 READ X$:M=LEN(X$)-1
7015 SIG=2
7020 X=0
7030 FOR I=1 TO M+1
```

```
7040 X(I-1)=VAL(X$(I,I))
7045 X=X+X+X(I-1)
7050 NEXT I
7060 X=X-2^M
7070 J=1:GOSUB 300
7075 GOSUB 5200
7080 REM BEGIN RECURSION
7085 PRINT ")";
7090 READ Y:IF Y=-99 THEN RETURN
7100 GOSUB 100:REM PPG
7110 GOSUB 2100:REM RECURSION
7120 J=J+1:GOTO 7080
9000 REM MIXED EXAMPLES
9005 REM DATA SET 1
9010 REM N,M,SIG,A$
9020 DATA 1,8,2,10211021
9030 REM INPUT B & U
9040 DATA 01100011,011
9050 REM DATA SET 2
9060 DATA 1,9,3,001102101
9070 DATA 110110001,1010
9080 REM DATA SET 3
9090 DATA 1,12,4,110112001101
9100 DATA 011001111010,11111
9110 REM DATA SET 4
9120 DATA 1,8,5,12001222
9130 DATA 01110011,100110
9190 DATA 0
9200 REM MULTIPLE RECURSION EXAMPLE
9210 DATA 4,8,2,01200122
9220 DATA 11001100,111
9230 DATA 01100011,010
9240 DATA 10001111,001
9250 DATA 00110110,110
9260 DATA 0
9500 REM MULTIPLICATION SIMULATION
9510 DATA 011010111
9520 DATA 2,-1,-2,1
9530 DATA 0,0,0,0
9999 DATA -99
```

LIST OF REFERENCES

[1] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," IRE Trans. on Electronic Computers, Sept. 1961, pp. 389-400.

[2] M. D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," IEEE Trans. on Computers, July 1987, pp. 895-7.

[3] M. Uya et al, "A CMOS Floating Point Multiplier," IEEE Journal of Solid-State Circuits, Oct. 1984, pp. 697-701.

[4] D. E. Atkins, "Design of the Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods," IEEE Trans. on Computers, Aug. 1970, pp. 720-33.

[5] C. S. Wallace, "A Suggestion for a Fast Multiplier," IEEE Trans. on Electronic Computers, Feb. 1964, pp. 14-17.

[6] Y. Harata et al, "A High-Speed Multiplier Using a Redundant Binary Adder Tree," IEEE Journal of Solid-State Circuits, Feb. 1987, pp. 28-33.

[7] R. F. Lyon, "Two's Complement Multipliers," IEEE Trans. on Communications, April 1976, pp. 418-25.

[8] M. J. Irwin and R. M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial," Computer, April 1987, pp. 61-73.

[9] M. J. Irwin and R. M. Owens, "Fully Digit On-Line Networks," IEEE Trans. on Computers, April 1983, pp. 402-6.

[10] K. S. Trivedi and M. D. Ercegovac, "On-Line Algorithms for Division and Multiplication," IEEE Trans. on Computers, July 1977, pp. 681-7.

[11] M. J. Irwin and R. M. Owens, "Techniques to Reduce the Inherent Limitations of Fully Digit On-Line Arithmetic," IEEE Trans. on Computers, April 1983, pp. 406-11.

[12] C. L. Seitz, "Concurrent VLSI Architectures," IEEE Trans. on Computers, Dec. 1984, pp. 1247-65.

[13] J. Mavor et al, Introduction to MOS LSI Design, Addison-Wesley Pub. Co., 1983.

[14] D. E. Atkins, "Introduction to the Role of Redundancy in Computer Arithmetic," Computer, June 1975, pp. 74-7.

[15] H. T. Kung, "Why Systolic Architectures?" Computer, Jan. 1982, pp. 37-46.

[16] K. Hwang, Computer Arithmetic, New York: Wiley, 1979.

[17] S. Waser, "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing," Computer, Oct. 1978, pp. 19-29.

[18] J. Kernhof et al, "High-Speed CMOS Adder and Multiplier Modules for Digital Signal Processing in a Semicustom Environment," IEEE Journal of Solid-State Circuits, June 1989, pp. 570-5.