AFIT/GCE/ENG/89J-1

SIGNED-DIGIT HIGH SPEED TRANSCENDENTAL FUNCTION PROCESSOR ARCHITECTURE

THESIS

Robert Alan Peterson
Captain, USAF

AFIT/GCE/ENG/89J-1

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/89J-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) SIGNED_DIGIT HIGH SPEED TRANSCENDENTAL FUNCTION PROCESSOR ARCHITECTURE	
12. PERSONAL AUTHOR(S) Robert A. Peterson, B.S. Captain, USAF	
13a. TYPE OF REPORT MS Thesis	13b. TIME COVERED FROM _____ TO _____
14. DATE OF REPORT (Year, Month, Day) 1989, JUNE	15. PAGE COUNT 160
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Chebyshev Polynomials, Approximation Algorithms, Signed-digit Representation, Pipeline Processor
12	01		
12	03		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman : Joseph DeGroat, Major, USAF

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph DeGroat, Major, USAF	22b. TELEPHONE (Include Area Code) 513-255-5633 22c. OFFICE SYMBOL AFIT/ENG

19. In support of the computation requirements of complex equations, a processor which can compute elementary transcendental functions with high throughput is becoming a hard requirement for many systems. In particular, the computation of components of the Vector Wave Equation are becoming bottlenecked by the reduced speed of the processor when computing the required elementary functions.

To speed up the computation of these type of functions, a pipelined processor with high throughput is developed. This processor will compute Sine, Cosine, Tangent, Cotangent, Arctangent, Exponential, Natural Logarithm and Division as a minimum. The accuracy of the computations will be greater than IEEE double precision. The majority of the approximation algorithms are derived from Chebyshev Polynomials, due to their error characteristics and compatability with a pipelined processor. The only approximation algorithm not derived from Chebyshev Polynomials is the division algorithm. Division is derived from an iterative form of a power series which has a similar computational form as that required by the algorithms developed from Chebyshev Polynomials. To prepare the algorithms for implementation in a pipelined processor, the algorithms are regrouped and rearranged into the from obtained by Horners' method. Then, the development of a unified Transcendental Function Processor is reviewed.

In an attempt to speed up the computations within the processor, alternate forms of data representation are investigated. Signed-Digit representation offers the greatest potential for increased speed over standard binary. This increased speed is due to the reduction of carry-barrow propagation delays throughout the hardware units. Signed-Digit modules are developed and performance estimates given. The modules are then described in VHDL and simulation results presented. From the VHDL module descriptions, a 16 digit by 16 digit multiplier is built and simulated.

AFIT/GCE/ENG/89J-1

SIGNED-DIGIT HIGH SPEED TRANSCENDENTAL FUNCTION PROCESSOR ARCHITECTURE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Robert Alan Peterson, B.S.

Captain, USAF

June, 1989

Preface

This research is a continued effort into the development of a Transcendental Function Processor. The processor has been baselined by Mickey Bailey and the approximation functions expanded and further elaborated to encompass a larger set of functions.

Intra-processor data representation is discussed and alternate forms of representing the data considered. Signed-Digit representation is discussed in great detail as a possible alternate to standard binary representation inside the processor. Signed-Digit hardware is presented along with its estimated performance parameters. The discussion of Signed-Digit representation proves to be the greatest thrust of this thesis.

I would like to thank AFIT and ENG in particular for the help and understanding during this thesis effort. Dr. D'Azzo and Major De Groat allowed me to have the time any motivation for me to complete the thesis. I would also like to thank my wife and family for their support and encouragement throughout the Master's Degree Program.

Robert Alan Peterson

Table of Contents

List of Figures

Abstract

In support of the computation requirements of complex equations, a processor which can compute elementary transcendental functions with high throughput is becoming a hard requirement for many systems. In particular, the computation of components of the Vector Wave Equation are becoming bottlenecked by the reduced speed of the processor when computing the required elementary functions.

To speed up the computation of these type of functions, a pipelined processor with high throughput is developed. This processor will compute Sine, Cosine, Tangent, Cotangent, Arctangent, Exponential, Natural Logarithm and Division as a minimum. The accuracy of the computations will be greater than IEEE double precision. The majority of the approximation algorithms are derived from Chebyshev Polynomials, due to their error characteristics and compatability with a pipelined processor. The only approximation algorithm not derived from Chebyshev Polynomials is the division algorithm. Division is derived from an iterative form of a power series which has a similar computational form as that required by the algorithms developed from Chebyshev Polynomials. To prepare the algorithms for implementation in a pipelined processor, the algorithms are regrouped and rearranged into the from obtained by Horners' method. Then, the development of a unified Transcendental Function Processor is reviewed.

In an attempt to speed up the computations within the processor, alternate forms of data representation are investigated. Signed-Digit representation offers the greatest potential for increased speed over standard binary. This increased speed is due to the reduction of carry-barrow propagation delays throughout the hardware units. Signed-Digit modules are developed and performance estimates given. The modules are then described in VHDL and simulation results presented. From the VHDL module descriptions, a 16 digit by 16 digit multiplier is built and simulated.

SIGNED-DIGIT
HIGH SPEED TRANSCENDENTAL
FUNCTION PROCESSOR ARCHITECTURE

I. Introduction

This effort studies approximation algorithms for various functions with the premise that the algorithms will be implemented in a pipeline processor. In an attempt to increase processing speed of the functions, alternate forms of data representation are investigated.

Approximation algorithms for trigonometric, exponential, natural logarithm, and the division function are developed. The structure of the approximation functions must be developed such that the processors pipeline will not require extensive re-configuration and control between the computation of different functions. Once the algorithms are developed, a unified processor can be designed to encompass pre-processing, pipeline processing, and post-processing.

A pipeline processor can increase the through-put of a system; however, the through-put is limited by the processing speed of the slowest stage. To increase the speed of the stages, either unique processing hardware must be designed or the data must be represented in a form which permits faster computation. This thesis looks at alternate data representation forms which reduce the carry-barrow propagation delays during computations.

Transcendental Function Processor Background

Approximation algorithms for Sine, Cosine, Tangent, Cotangent, Arctangent, Exponential, and Natural Logarithm have been long known and are quite numerous, [1, 2, 3, 4]. The algorithms were derived from Chebyshev Polynomials which are expanded, summed, and regrouped into a polynomial function of x. The pre-processing, pipeline processing, and post-processing requirements are similar for each function. A baseline processor was

defined to provide IEEE single precision accuracy for the computations. The performance estimates of the processor are based on the speed of an IEEE single precision floating point multiplier.

Other algorithms which have been investigated include the CORDIC algorithm and other ultra-spherical polynomials, [1, 4]. However, the primary algorithm of their investigations, other than those developed from Chebyshev Polynomials, is the CORDIC algorithm. The CORDIC algorithm is an iterative algorithm which can not be realistically implemented in a pipelined processor. Other problems involve the computation of non-trigonometric functions to which the CORDIC algorithm is not suited.

Alternate forms of data representation which have been studied include the Negative Base Number System, Residue Number System, and Signed-Digit Number System, [9, 10, 12, 13]. Each has advantages and dis-advantages associated with them and are discussed further in Chapter 4.

Objective

The objective is to complete the development of the approximation algorithms which are to provide IEEE double precision accuracy while investigating alternate forms of data representation to speed-up their processing. Once the algorithms are developed they will be mapped onto a pipelined processor architecture.

Scope

The scope of this thesis effort is to extend the previous work done on the development of approximation algorithms by extending the precision of the developed algorithms. The algorithm for division will be developed such that its general form is compatible to the processor defined by the algorithms developed from Chebyshev Polynomials. A unified processor will be defined to encompass the processing requirements of all of the approximation functions. Alternate forms of data representation will be studied and their benefits elaborated with emphasis on the reduction of carry-barrow propagation delays.

Assumptions

The assumptions made in this effort are that the physical size of the processor is not limited. There are no attempts to determine the resulting chip area that would be required to implement the processor. It is assumed that the processor will operate in an environment where the pipelines latency will not cause major problems.

Organization

The remained of this thesis is organized as follows. Chapter 2 is the rational behind using Chebyshev Polynomials for approximations in the Transcendental Function Processor; as well as the development of the division algorithm. The processors hardware is discussed in Chapter 3 with a breakdown of its pre-processing, pipeline processing, and post-processing requirements. Chapter 4 presents alternate forms of data representation and elaborates on Signed-Digit representation and its major functional units. Chapter 5 presents the basic Signed-Digit modules used to construct major functional units, components such as multipliers and adder/subtractors, and presents SPICE results as estimates of their preformance. Chapter 6 builds the VHDL descriptions of the basic modules and instantiates them to build a Signed-Digit multiplier with an accuracy greater than IEEE double precision. This multiplier is then simulated and performance estimates presented. The thesis is concluded in Chapter 7 with final conclusions and recommendations for follow-on research.

II. Approximation Methods and Algorithms

Approximation of Transcendental Functions

By definition, transcendental functions are functions which are not algebraic, [7]. Therefore, they cannot be expressed in terms of sums, differences, products, quotients, or roots. The only way to evaluate them is by approximation, which leads to the study of approximation methods, or algorithms. Each method has advantages and disadvantages associated with them. This study looks at the proven methods of approximation with the idea of implementing the algorithms in hardware.

There are hardware limitations which constrain the total class of approximating methods to looking at approximation algorithms which employ multiplication and addition. A large number of algorithms use quotient and root functions. In hardware, these functions are too time consuming for implementation as a one step function and are therefore discarded as not viable approximation algorithms for implementation. This dramaticly narrows the class of approximation methods. The remaining approximation algorithms may then be compared by looking at the error characteristics of each.

To decide which algorithm is the *best*, the term *best* must be clearly defined. In this paper, the best approximation algorithm is the one which requires the fewest mathematical operations and gives an error less than some maximum tolerable error. There are different types of error which are of interest when approximating; each may specify a different algorithm as being the best. If the the error associated with the *best* algorithm is defined as the average difference between the approximating function and the true function, across an interval, then the Least Square error is the error type of interest. However, if the maximum deviation between the approximating function and the true function, across an interval, is of interest, then, the type of error specifying the *best* approximation algorithm is termed the Maximum Norm error. When approximating a function to obtain the domain-range pair on a point- for-point basis, the Maximum Norm error is used to identify the best approximation algorithm. In this study, this is the type of error used to determine the *best* algorithm. Figure 2.1 shows how the Least Square error and the Maximum Norm error

differ, given the magnitude of the respective errors are equal. Note that the error function characterizing the Least Square error is near zero over a portion of the interval; however, the maximum deviation is greater than the error function characterizing Maximum Norm error. Since the domain is continuous over the interval of interest, the maximum magnitude of the error is used to compare approximation algorithms, Maximum Norm error.

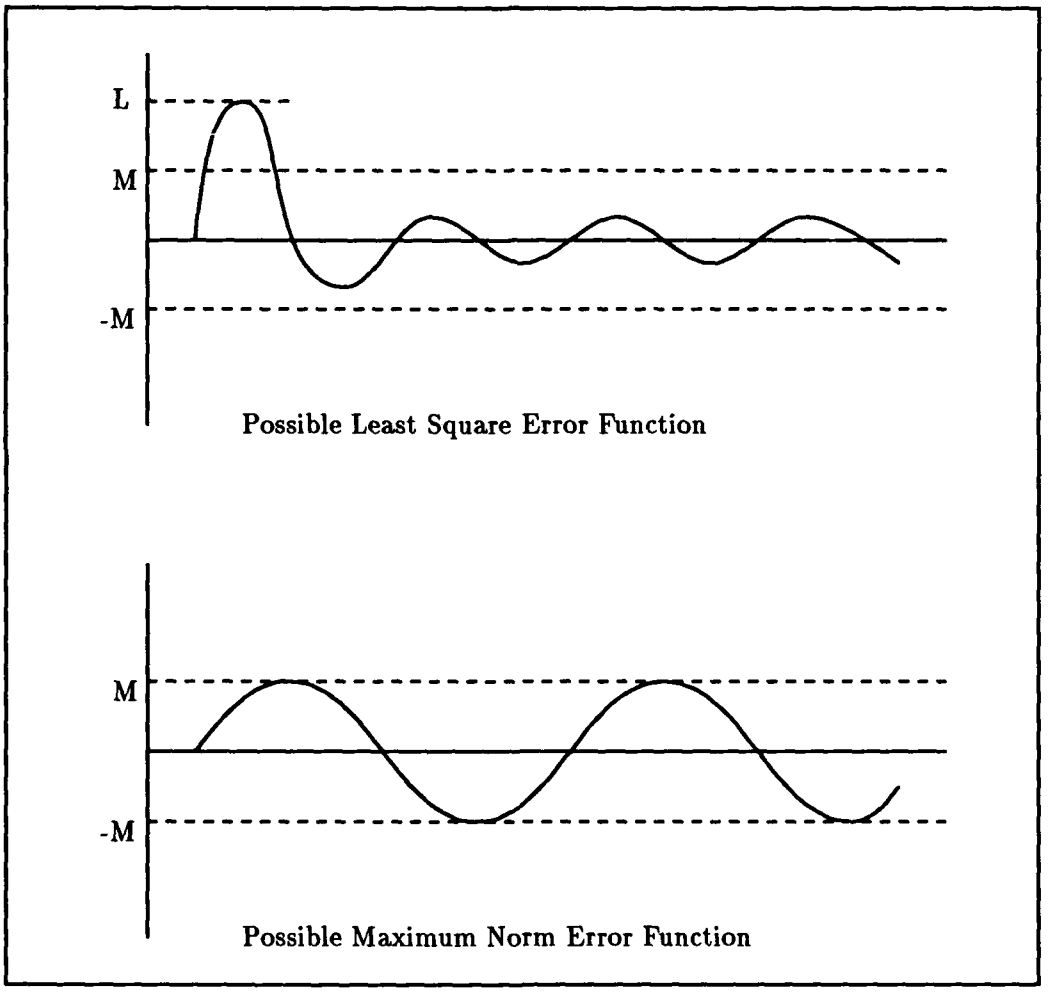

Figure 2.1. Least Square Error Compared to Maximum Norm Error.

Error functions associated with a specific approximation algorithm have characteristic shapes. These shapes not only indicated how well an algorithm, with a given number

of terms, approximates the true function, they give an indication of how the Maximum Norm error changes as the number of terms used for the approximation change. These shapes can lead to the selection of the *best* approximation method by understanding the relationship between the Maximum Norm error and the number of approximation terms. The error function associated with the Taylor's series, as shown in Figure 2.2, is shaped like a parabola with zero error in the center, or the point of differentiation. As the number of terms in the approximation function increase, the smaller the error is at the end-points of the parabola, corresponding to the end-points of the interval. Eventually, if an infinite number of terms are used in the approximating function, the error at the end-points becomes zero. Therefore, to get the Maximum Norm error below a specific value, the number of terms required is determined by the magnitude of the error at the end points while the error between the end points may be acceptable with considerably fewer terms. The error function associated with approximation using Legendre Polynomials oscillates around zero with the magnitude of the oscillations increasing as the end points of the interval are approached. Though the maximum error may not occur at the end points, the maximum error is near the end points. To get this Maximum Norm error below a specific value, the number of terms required is determined by the magnitude of the oscillation near the end points. This is better than the Taylor series since the maximum error does not correspond exactly with the end points of the interval. A better approach is to have the error oscillate with equal magnitude around zero. Then, as the number of terms increase, the Maximum Norm error decreases uniformly across the interval. This equal magnitude oscillation of the error is termed the *equal ripple* property [3]. The equal ripple property ensures a uniform maximum error across the interval, unlike the Taylor series or Legendre polynomials which achieve excellent approximations near zero but poor approximations at, or near, the end points. The approximation algorithms which exhibit the equal ripple property are the algorithms which approximate functions using Chebyshev Polynomials.

Approximation algorithms using the Taylor series, Chebyshev Polynomials, and the Legendre Polynomials are sub-classes of more general approximation algorithms using Ultra-spherical Polynomials [3]. The general form of the Ultra-spherical Polynomial is

$$P_n^{(\alpha)}(x) = C_n(1-x^2)^{-\alpha}\frac{d^n}{dx^n}(1-x^2)^{n+\alpha}$$

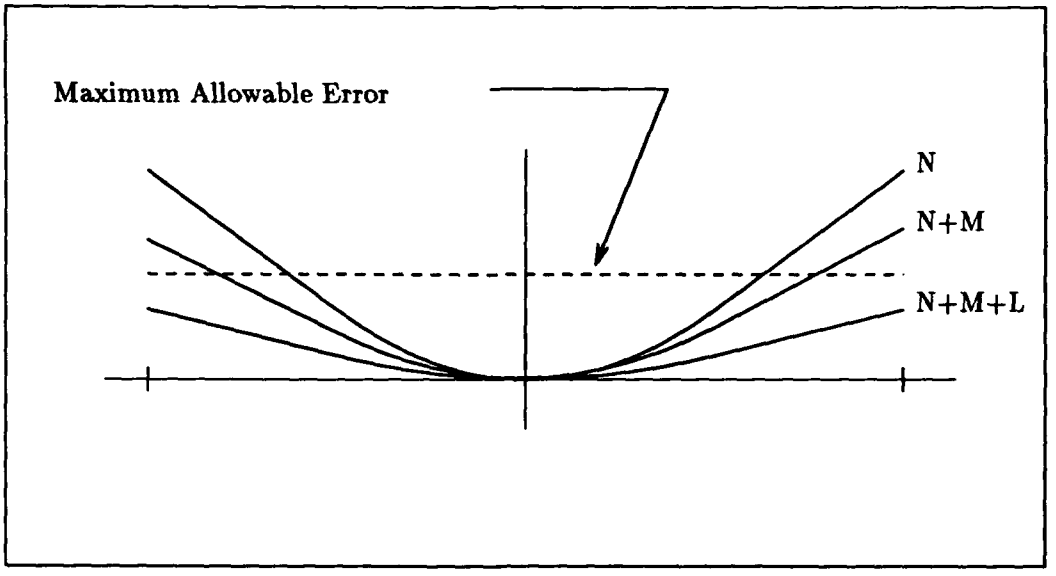

Figure 2.2. Error Function Using Taylor's Series Approximations.

where C_n is a constant and α is in the interval $(-1 \leq \alpha \leq \infty)$.

A general analysis of approximations using Ultra-spherical polynomials shows that, when α is greater than $-1/2$, the amplitude of the oscillations of the error function increases as x moves away from the origin. Ultimately, as α approaches ∞, the series of Ultra-spherical polynomials describes the Taylor series. When $\alpha = 0$, the ultra-spherical polynomial corresponds to the Legendre Polynomial. However, when α is less than $-1/2$, the magnitude of the oscillations of the error function decrease as x moves away from the origin. The value of α which gives the equal ripple property is $\alpha = -1/2$; this describes the Chebyshev Polynomial.

Chebyshev Approximation Methods

Chebyshev Polynomials are orthogonal polynomials, similar to the trigonometric functions of Sine and Cosine, and are derived from the more general class of Ultra-spherical polynomials. The Chebyshev polynomials, T, are related to trigonometric functions by the

identity

$$T_n(\cos x) = \cos nx.$$

From this identity, and the functional relations of the Cosine,

$$\begin{aligned} \cos 0 &= 1, \\ \cos x &= \cos x, \\ \cos 2x &= 2(\cos^2 x) - 1, \end{aligned}$$

the Chebyshev Polynomials may be derived.

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \end{aligned}$$

Additional Chebyshev polynomials are found by the recursions formula

$$T_{n+1}(x) = 2 * T_n^2(x) - T_{n-1}(x).$$

(The expanded Chebyshev polynomials, up to $n = 22$, are given in [2].)

When approximating a function with Chebyshev polynomials, each polynomial is weighted by a constant and then summed.

$$f(x) = \sum_{n=0}^{N} a_n T_n(x) \quad where - 1 \leq x \leq 1$$

Since the Chebyshev polynomials exhibit the orthogonality property, odd functions require summing of only the odd polynomials; likewise, even functions only require the summing of even polynomials.

The weighting constants for each polynomial are computed from the function

$$a_n = \frac{2}{\pi} \int_0^{\pi} f(\cos x) \cos nx \, dx$$

This functions is not simple to integrate; however, there are means to accomplish the integration; these are described in Appendix A. The last piece of information required to

completely define an approximation algorithm using Chebyshev Polynomials is to determine the number of terms, or polynomials, required for the approximation. To do this, a relationship between the maximum tolerable error and the number of polynomials required, such that the Maximum Norm error from the approximation is less than the maximum tolerable error, is needed. This relationship is

$$|\epsilon_N| \approx \left| \frac{f^N(x)}{2^{N-1}N!} \right| \tag{2.1}$$

From this relationship, the maximum magnitude of the error can be approximated for any function, given the number of polynomials used to approximate that function.

By using Equation 2.1 to estimate the number of terms required to have an error less than 2^{-60}, the general form of the Chebyshev polynomial approximations for the transcendental functions of interest are

$$\begin{aligned}
\sin(\frac{\pi}{2}x) &= \sum_{n=0}^{9} a_{2n+1} T_{2n+1}(x) \\
\cos(\frac{\pi}{2}x) &= \sum_{n=0}^{9} a_{2n} T_{2n}(x) \\
\tan(\frac{\pi}{4}x) &= \sum_{n=0}^{15} a_{2n+1} T_{2n+1}(x) \\
\cot(\frac{\pi}{4}x) &= \sum_{n=0}^{15} a_{2n+1} T_{2n+1}(x) \\
\arctan(x) &= \sum_{n=0}^{11} a_{2n+1} T_{2n+1}(x) \\
e^x &= \sum_{n=0}^{11} a_n T_n(x) \\
\ln(x+1) &= \sum_{n=0}^{11} a_n T_n(x)
\end{aligned}$$

Approximating with Chebyshev Polynomials has one problem. The form of the approximation algorithms does not fit well into a pipelined architecture. This is due to the computation, weighting, and summing of the terms as the approximation progresses.

$$f(x) = a_0 T_0(x) + a_2 T_2(x) + a_4 T_4(x) + \cdots + a_n T_n(x)$$

However, since all of the terms are polynomials, each term may be expanded and regrouped, using the distributive and associative properties, to form a single polynomial of degree N. This eliminates the computation and weighting of each polynomial term. However, the parallel summing of the powers of the resultant polynomial must still occur.

$$f(x) = B_0 + B_2x^2 + B_4x^4 + \cdots + B_nx^n$$

To eliminate this problem, the approximation polynomial may be rearranged by using Horner's method [8]. This results in an expression which is computed as a series of sum-product stages with the result from each stage used as the input for the next.

$$f(x) = C_0(C_2 + x^2(C_4 + x^2(\cdots(C_n + x^2)\cdots))) \tag{2.2}$$

This form of approximation is well suited for a pipelined architecture. However, when manipulating the coefficients of the Chebyshev Polynomials to obtain this arrangement, precision is lost. To achieve the same precision as that specified when implementing the approximation using the Chebyshev Polynomials directly, one additional term, or polynomial, is required.

Division Algorithm

Division is performed by finding the reciprocal of the divisor and multiplying the result to the dividend. Chebyshev polynomials cannot be used efficiently for the approximation of the reciprocal function. Therefore, alternate methods were investigated.

An algorithm is sought which requires only the sum and product operations. Also, the algorithm should be in a form similar to the general form defined by Horner's method, Equation 2.2. The algorithm which best meets these requirements is an iterative form of a power series for reciprocal [2]. This algorithm has the form

$$Y_{i+1} = Y_i(2 - xY_i) \tag{2.3}$$

where Y_i is the i^{th} approximation of $1/x$ and Y_{i+1} is the next approximation. This iterative equation differs from the form that Horner's method yields. However, Equation 2.3 can be rewritten as

$$Y_{i+1} = Y_i(2 + Y_i(0 - x)) \tag{2.4}$$

This is in the form required by the pipelined architecture presented in the preceding section. However, there are two sum-product functions required for each iteration. Therefore, if the k^{th} iteration gives a result which has a Maximum Norm error less than some specified error value then, $2k$ sum-product operations are required. As long as the number of iterations required is less than one-half the order of the highest polynomial used for the approximations by the rearranged Chebyshev Polynomials, no additional stages in the pipeline are required. This algorithm also requires x to be positive. However, Equation 2.4 inverts the sign of x, now requiring it to be negative. Sign corrections can be performed in the pre and post-processing stages of the architecture.

The number of iterations required to achieve a Maximum Norm error less than some specific value, ϵ, depends on the magnitude of ϵ and the magnitude of the error in Y_0, where Y_0 is the initial guess of the reciprocal and must be computed in a pre-processing stage. If the initial guess is defined as

$$Y_0 = \left(\frac{1}{x}\right) + \Delta$$

where Δ is some error term, then,

$$\begin{aligned} Y_1 &= \left(\frac{1}{x}\right) - x\Delta^2, \\ Y_2 &= \left(\frac{1}{x}\right) - x^3\Delta^4, \\ Y_3 &= \left(\frac{1}{x}\right) - x^7\Delta^8, and \\ Y_4 &= \left(\frac{1}{x}\right) - x^{15}\Delta^{16}. \end{aligned}$$

The i_{th} iteration yield an error term of

$$\epsilon_i(x) = x^{2^i-1}\Delta^{2^i}$$

As long as $\epsilon_i(x) \leq \epsilon$ for all x in an interval, then, $Y_i = 1/x$. Once the maximum tolerable error, ϵ, and the interval of x defined, then, the maximum allowable error for Y_0 is determined by the number of iterations, i.

$$\Delta_i(x) = \left(\frac{\sqrt[2^i]{\epsilon x}}{x}\right)$$

As the number of iterations increase, the required accuracy of Y_0 decreases.

The difficulty of the reciprocal algorithm is determining how to compute Y_0. To make full use of the pipeline hardware required to compute the transcendental functions from the preceding section, eight iterations of the reciprocal algorithm are used. Therefore, the maximum allowable error when $x = 1$ is $\Delta_8(1) \approx 0.85005$ and the maximum allowable error when $x = 1/16$ is $\Delta_8(1/16) \approx 13.45434$. A linear function can compute Y_0 for all x in the interval $(1/16 \leq x \leq 1)$ and give an error less than $\Delta_8(x)$. The linear function has the form $Y_0(x) = ax + b$. The error function between is $Y_0(x)$ and $1/x$ is

$$e(x) = ax + b - (1/x) = \frac{ax^2 + bx - 1}{x}.$$

The absolute value of the error generated from $e(x)$ must be less than, or equal to, $\Delta_8(x)$, the initial maximum allowable error, for all x in an interval. The *best* linear function will not give the line that bisects the function $1/x$ because the error of the linear function at the upper end point of the interval must be less than the error at the lower end point. What is required is to have the ratios of the errors at the end points, relative to their maximum allowable error, equal. By analyzing the error in this manner, the error across the interval is essentially normalized. The normalized error function is

$$n(x) = \frac{e(x)}{\Delta_8(x)} = \frac{ax^2 + bx - 1}{2^{0.0625}x^{0.0625}}. \tag{2.5}$$

Because of the shape of $1/x$ and the fact that it is being estimated by a linear function, the errors at each end point are negative. There is also some point between in which the error will be positive and a maximum. This can be seen from Equation 2.5 by realizing the slope of the line approximating $1/x$ must be negative, giving a negative a in the error function $n(x)$. Then, the numerator of $n(x)$ is a quadratic which opens downward; in the intervals of interest, the denominator is always positive. Also, in order to get the *best* fit for the approximation line, the line will cross $1/x$. Therefore, the location of maximum positive error of the normalized error function, $n(x)$, is found by setting the first derivative of $n(x)$ equals 0. This results in

$$0 = 2^{-0.0625}x^{-1.0625}(a * 1.9375 * x^2 + b * 0.9375 * x + 0.0625).$$

As long as x does not equal 0, the location of the maximum positive error, X_c, is obtainable from the quadratic term above.

$$X_c = -B \pm \frac{\sqrt{B^2 - 4AC}}{2A}$$

where $A = 1.9375a, B = 0.9375b$,and$C = 0.0625$. Since a is negative and the square root term is positive and larger than B, the negative of the square root term gives a positive X_c. In order to minimize the normalized error over an entire interval, the magnitude of the normalized error at the end points must equal the magnitude of the normalized error at X_c and be of opposite sign. The magnitude of the normalized error at these points are the maximums for the interval. As long as this maximum is less than 1, the reciprocal algorithm, with eight iterations, will converge to $1/x$ with an accuracy better than ϵ. To find a and b, the normalized error function must be used with x equal to the end points of the interval. Then, a normalized error, less than 1, is chosen. This results in two equations with two unknowns whereby a and b are determined. Then, with a and b, X_c is computed and the normalized error, $n(x)$, when $x = X_c$ is compared to the chosen normalized error used to determine a and b. If the normalized errors are not equal, the chosen normalized error is changed until the normalized error at X_c equals the chosen normalized error, within desired bounds. By using the linear equation and a and b in the pre-processing stages, the initial estimate, Y_0, will always cause the final iteration to converge to within the required accuracy, ϵ.

Summary of Algorithms

All of the algorithms used, with the exception of the algorithm for the approximation of the division function, are based on Chebyshev Polynomials. This is due the the error characteristics of Chebyshev Polynomials over other approximation algorithms. By using an algorithm which has the *equal ripple* property, fewer number of terms are required to achieve a specified precision. Then, by regrouping and rearranging the polynomials, a form suitable for pipeline processing emerges. The approximation algorithm for the division function is based on an iterative power series. The form of the power series is compatible to the form obtained from the modified Chebyshev Polynomials.

III. Processor Architecture

Pre-processing Stages

The pre-processing stages of the processor converts the arguments of the functions into the form required by the algorithms implemented in the pipeline. The conversion of the arguments takes the form of scaling and sign correction to prepared them for the pipeline. These operations of the pre-processor are fast and add little overhead to the entire processor function.

Sine and Cosine Pre-processing The Sine and Cosine functions are computed by using only the regrouped, rearranged, Chebyshev Polynomials to approximate $\sin(\pi x/2)$. This eliminates the lookup table entries for the coefficients required for the $\cos(\pi x/2)$ function in the pipeline and reduces the overall complexity of the control logic for the processor. The Cosine function is related to the Sine function by the identity

$$\cos x = \sin\left(\frac{\pi}{2} - x\right).$$

The first step in the pre-processing stage is to determine if the Sine or the Cosine function is being called. If the Cosine functions is being called then, the argument is transformed to an argument for the Sine function by subtracting it from $\pi/2$. If the Sine function is being called, then, the argument passes unaltered to the next stages of the pre-processor. From this point on, the pre-processing stages are the same for both the Sine and Cosine functions.

The required range of the argument passed to the pipeline is $(-1 \leq x \leq 1)$. To prepare the processors input to be within this range, the input is multiplied by a constant, $2/\pi$, and the result is factored into a sign component, integer component, and a fractional component. The sign component gives the direction of rotation for the functions while the integer component, with the sign component, gives the quadrant of the argument. If the integer component is odd then, the fractional component is subtracted from 1. Otherwise, the fractional component is unaltered. The sign of the fractional component is determined by the sign component xor'ed with the next least significant bit of the integer component.

Since the sign component is required to be stripped out of the argument, leaving the integer and fractional components both positive, the multiplication constant, $2/\pi$, to the argument may instead be the constant $-2/\pi$. Simple logic in the front end of the multiplier selects which constant to use. This choice also determines the sign component. The maximum value of the integer component required is only two least significant bits. Since the integer component is positive and it determines which quadrant the argument is in, zero to three, two bits are all that is required and all higher bits are discarded.

The overall pre-processing requirements for the Sine and Cosine functions are shown in Figure 3.1. The pre-processing stages are controlled by the command word directing the processor to compute the Sine or the Cosine of an argument. This *global* control is used only to select whether to multiplex x or $\pi/2 - x$ to the next stages. All other controls for the pre-processing stages are *local* control signals and do not need to extend beyond the pre-processor.

Tangent and Cotangent Pre-processing The Tangent and Cotangent pre-processing is similar to the pre-processing requirements of Sine and Cosine. The identity

$$\tan x = \cot\left(\frac{\pi}{2} - x\right)$$

is used to reduce the number of coefficients in the look-up tables and the amount of control in the pipeline by computing only the Cotangent function in hardware and converting the Tangent arguments to Cotangent arguments. This conversion hardware is the same as that required for the Cosine to Sine argument conversions. Therefore, if the Tangent functions is to be computed, the argument is subtracted from $\pi/2$ and the resultant argument is operated on as if the Cotangent function was called. The next step is to scale the argument into the range $(-1 \leq x \leq 1)$ and extract the sign, integer, and fractional components for the computation of rotation and quadrant of the argument. This is the same as the requirements for the Sine-Cosine argument. The argument is multiplied by $2/\pi$, or $-2/\pi$, and the result extracted into its three components. The least significant bit of the integer component is used to select whether to use the fractional component directly or to subtract it from 1. If the integer component is odd, the least significant bit is a 1, then the fractional component is subtracted from 1 to give the correct magnitude. Otherwise, the

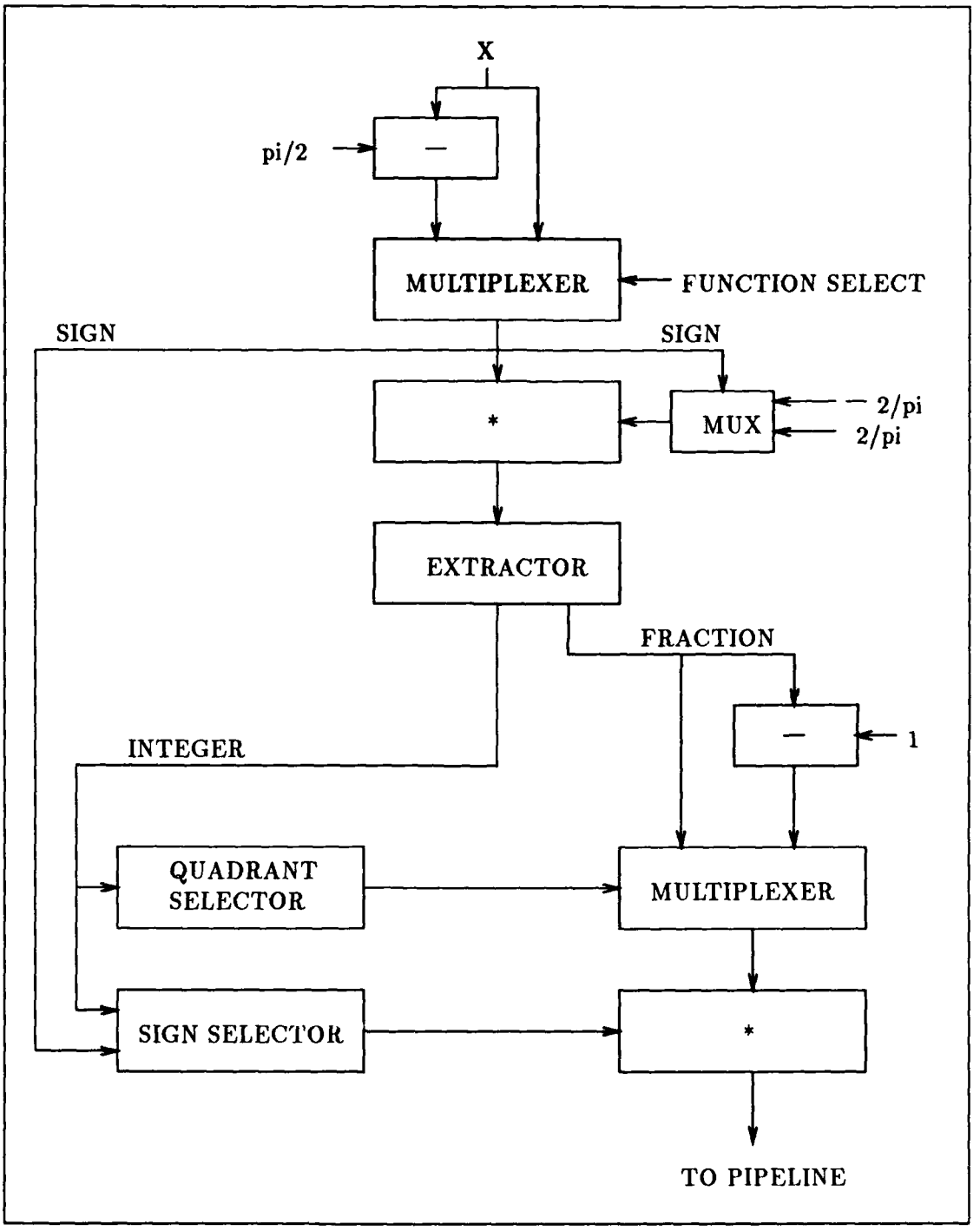

Figure 3.1. Sine/Cosine Pre-processing Requirements.

least significant bit is 0 and the fractional component is unaltered. The sign of the fractional component is determined by the XOR operation of the sign component and the next least significant bit of the integer component. Up to this point, the hardware requirements for pre-processing the Tangent and Cotangent arguments is the same as that required for pre-processing of the Sine and Cosine arguments. However, the range of the argument for the Tangent approximation is $(-n\pi/4 \leq x \leq \pi/2 - n\pi/4)$ and the range for the Cotangent argument is $(-\pi/4 \leq n\pi x/2 \leq \pi/4)$. The final pre-processing step is to multiply the resultant argument by 2. If the result is greater than 1, an internal error is generated which indicates that the argument is out-of-range for the called function and the co-function, in conjunction with the division function, must be used to compute the required result. This constrains the computation of the Tangent and Cotangent functions somewhat. However, it is necessary to limit the length of the pipeline to a reasonable number of stages. One method to overcome this problem is to increase the processors control section such that when it detects an out-of-range error, the co-function and division function are internally scheduled and performed to get the desired results. The addition of control logic hardware must be weighed against the alternative of having software check the arguments before requesting the function and against the frequency of the arguments being out-of-range.

The pre-processing requirements for the Tangent and Cotangent functions are shown in Figure 3.2. Like the Sine and Cosine functions, the *global* control is used only to select which function is to be performed. All other control operations do not extend beyond the pre-processing stages. The out-of-range error is used as discussed above.

Arctangent Pre-processing The pre-processing requirements for the Arctangent function is described in [1] and reviewed here. The range of the argument required for the pipeline is $(-1 \leq x \leq 1)$. If the argument of the Arctangent function is within this range it may be given directly to to pipeline for computation. However, if the argument is outside this range, the trigonometric identity

$$\arctan(x) = \pi/2 - \arctan(1/x)$$

must be used. The error signal must be generated and either handled internally, by the control section scheduling the proper operations, or by software to compute the desired

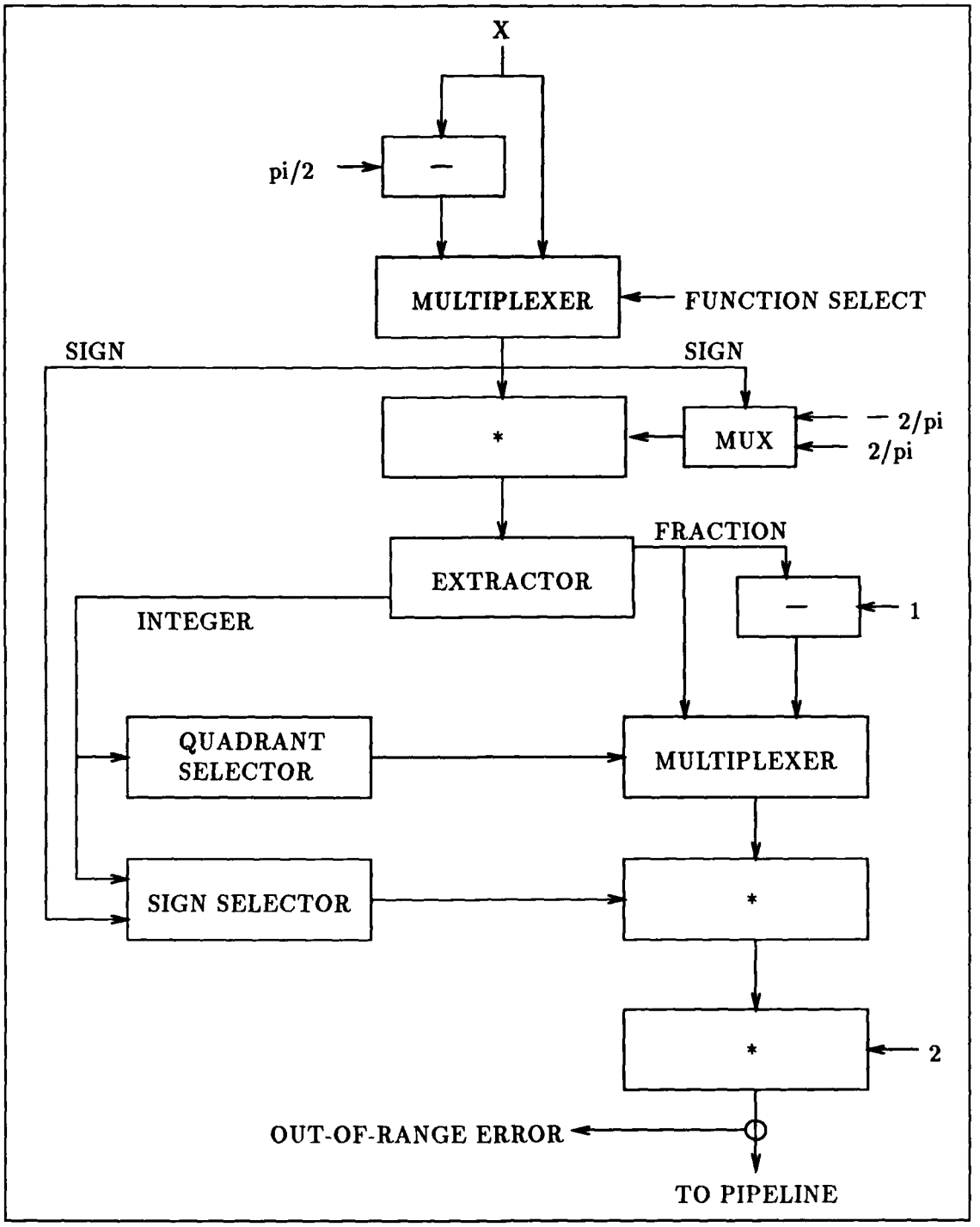

Figure 3.2. Tangent/Cotangent Pre-processing Requirements.

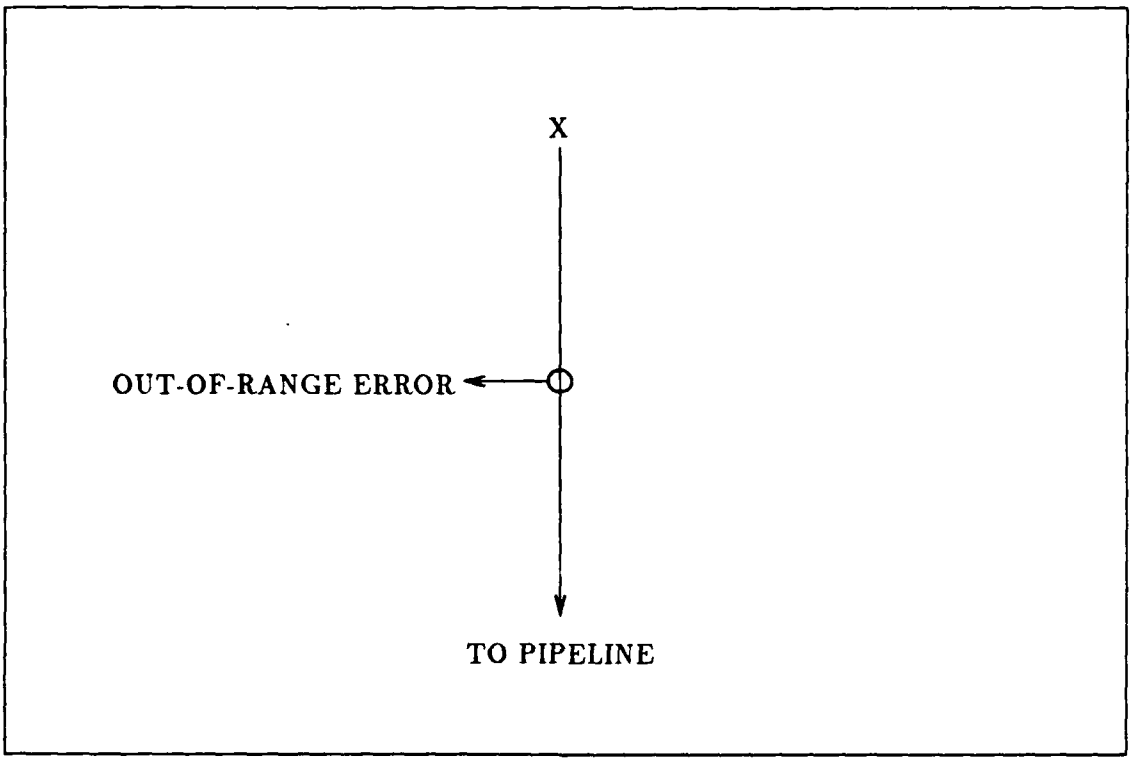

Figure 3.3. Arctangent Pre-processing Requirements.

value. Figure 3.3 shows the pre-processing requirements for the Arctangent function. This differs from Figure 3.4 in [1] due to the realization of the control section having to schedule the reciprocal operation as a separate function and not just a pre-processing operation.

Exponential Pre-processing The pre-processing requirements for the Exponential function, as described by [1], requires x be decomposed into an integer and fractional value.

$$e^x = e^N * e^F$$

The integer portion, e^N, is evaluated by using a ROM table to look-up the result. For IEEE single precision, the required ROM table is 89 words deep; for double precision, the ROM table is 712 words deep. The fractional component, e^F, is computed by submitting F to the pipeline for computation. The integer and fractional results are then multiplied in the

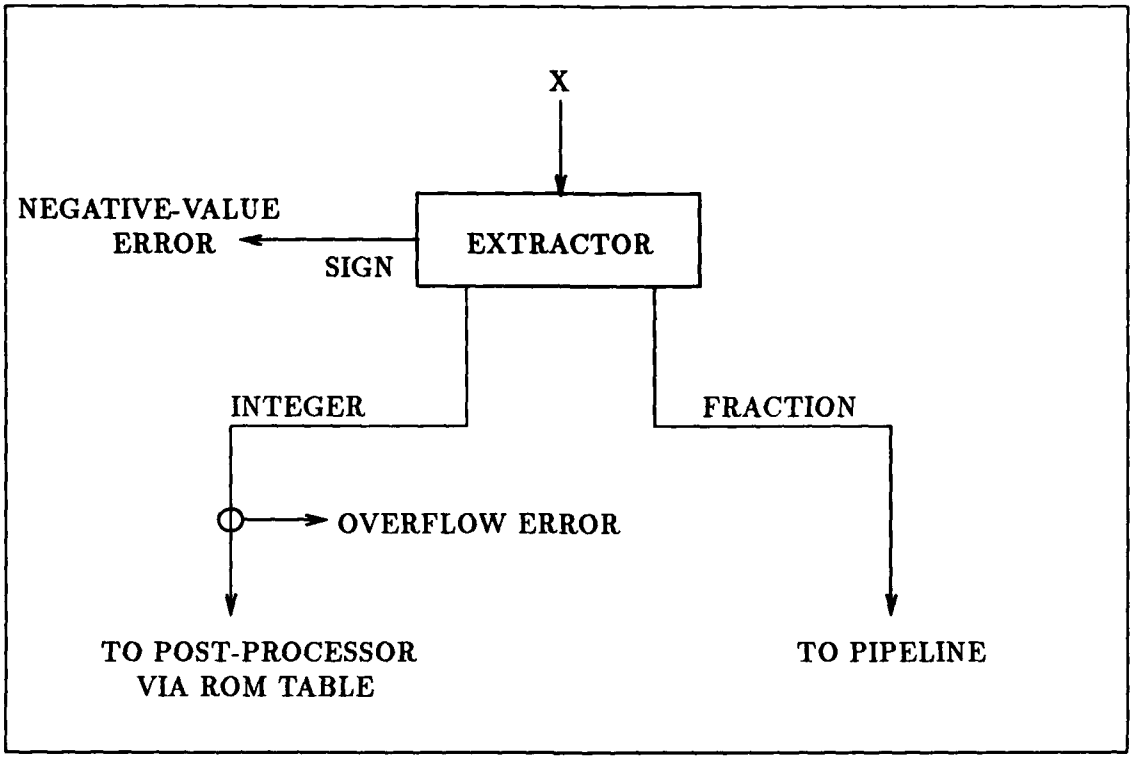

Figure 3.4. Exponential Pre-processing Requirements.

post-processor. If x is negative, an internal error is generated and the control section may either schedule the exponential and division functions for x or generate an external error and let the software handle the error. Figure 3.4 shows the pre-processing requirements for the exponential function. The extractor separates the argument into an integer part and a fractional part. The integer part is used to find the value of e^N in the ROM table while the fractional part is operated on in the pipeline. If the integer value is larger than the depth of the ROM table, an overflow error is generated. This error signifies that the value of e^N is larger than the largest value which can be represented in the data representation form, such as IEEE single or double precision.

Natural Logarithm Pre-processing The pre-processing requirements for the Natural Logarithm function is complex and well described by [1]. Presented here is an overview of

the requirements in order to get an understanding of the full pre-processing requirements of the processor.

To compute the $\ln(x)$ by using Chebyshev approximation, $\ln(x+1)$ must be computed where $x+1$ must be in the interval $(0.7071 \leq x+1 \leq 1)$. To scale $x+1$ to this range, the identity

$$\ln x^y = y \ln x$$

is used to separate the exponent of the argument from the mantissa. The exponent is then used in the post-processor stages. The mantissa is then scaled by a value which is selected by the magnitude of the mantissa in order to get a result in the required range. The identity

$$\ln mn = \ln m + \ln n$$

is used to justify the scaling and later subtraction of the Natural Logarithm of the scaling factor from the pipelines result in the post-processing stages. Figure 3.5 shows the pre-processing requirements for the Natural Logarithm function.

Division Pre-processing The pre-processing requirements for the division function consist of sign correction of the divisor, extraction of the mantissa and exponent of the divisor, and the computation of the initial *guess*, Y_0. The algorithm implemented in the pipeline requires the divisor to be positive. Therefore, if the divisor is negative, the numerator and denominator are both multiplied by -1. This performs the required sign correction for the divisor without any additional requirements imposed on the post-processor. The exponent and mantissa are separated and the mantissa shifted, with a corresponding adjustment of the exponent, such that it is in the range $(1/16 \leq M \leq 1)$. The exponent is then operated on separately from the mantissa. The mantissa is then used as the argument of a linear function to compute an initial guess of the reciprocal, Y_0. The linear function,

$$Y_0 = aM + b$$

is used where a and b are both constants which are not dependent on the value of M. Y_0 and M are then given to the pipeline for computation of the reciprocal of the denominator while the numerator and the exponent of the denominator are sent to the post-processor for eventual processing.

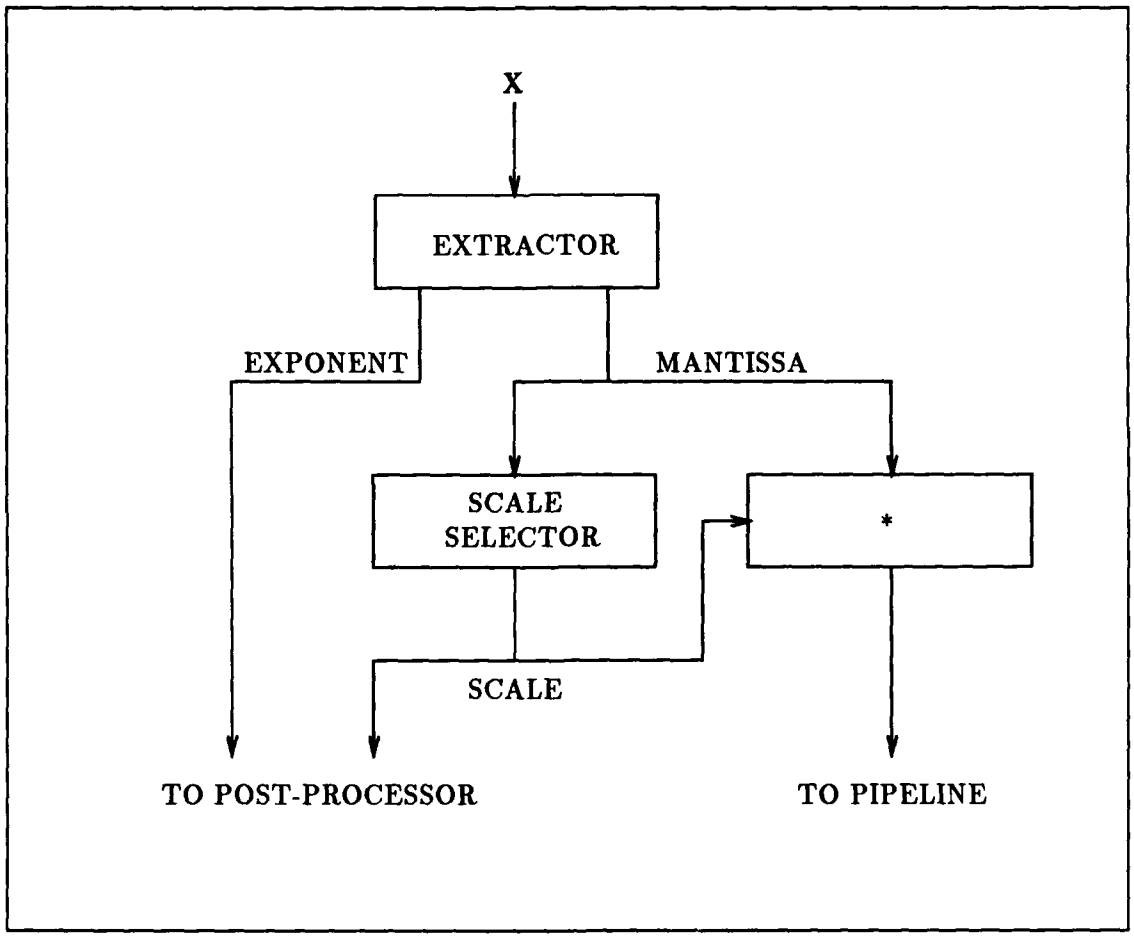

Figure 3.5. Natural Logarithm Pre-processing Requirements.

Figure 3.6 shows the pre-processing requirements for division. At a minimum, the control hardware must detect a zero denominator; and, the control hardware could be increased to detect a zero numerator.

Unified Pre-processor A Unified Pre-processor combines all of the requirements of the preceding sections and establishes one pre-processor to handle them all. This Unified Pre-processor can take on many different forms, the best form is not necessarily the best for all environments. The architecture of the pre-processor is dependent on the frequency of each operation requested. If a certain function is not requested often and it has a unique pre-processing requirement, then the architecture of the pre-processor will take on a different configuration than it would if the function was requested more often. In general, the configuration of the Unified Pre-processor will have to consist of a bus arrangement where data can be inserted, and pulled from, different points. By simple analysis, the two extreme pre-processing requirement are those of the Tangent/Cotangent functions and the Division function. Much of the hardware required for pre-processing of the Tangent/Cotangent functions, as well as its layout, is suitable for most of the other functions. The extractor stage can be constructed such that it is more general in nature, giving the fractional, integer, sign, exponent, and mantissa components.

The exact layout of a Unified Pre-processor requires a great deal of analysis of instruction frequencies before it can be properly designed.

Pipeline Architecture

The pipeline architecture is designed for the computation of algorithms which have been regrouped and rearranged such that they are expressed in the form generated from applying Horners' Method [8]. This yields a series of sum-product stages with each stage feeding the next. The algorithms developed by regrouping and rearranging Chebyshev polynomials all have a similar form, with one exception. The even functions only use even powers of the argument presented to the pipeline while odd functions only use odd powers. Functions such as the Exponential use both even and odd powers. However, when all of the functions are expressed using Horners' Method, only x and x^2 are required. Even

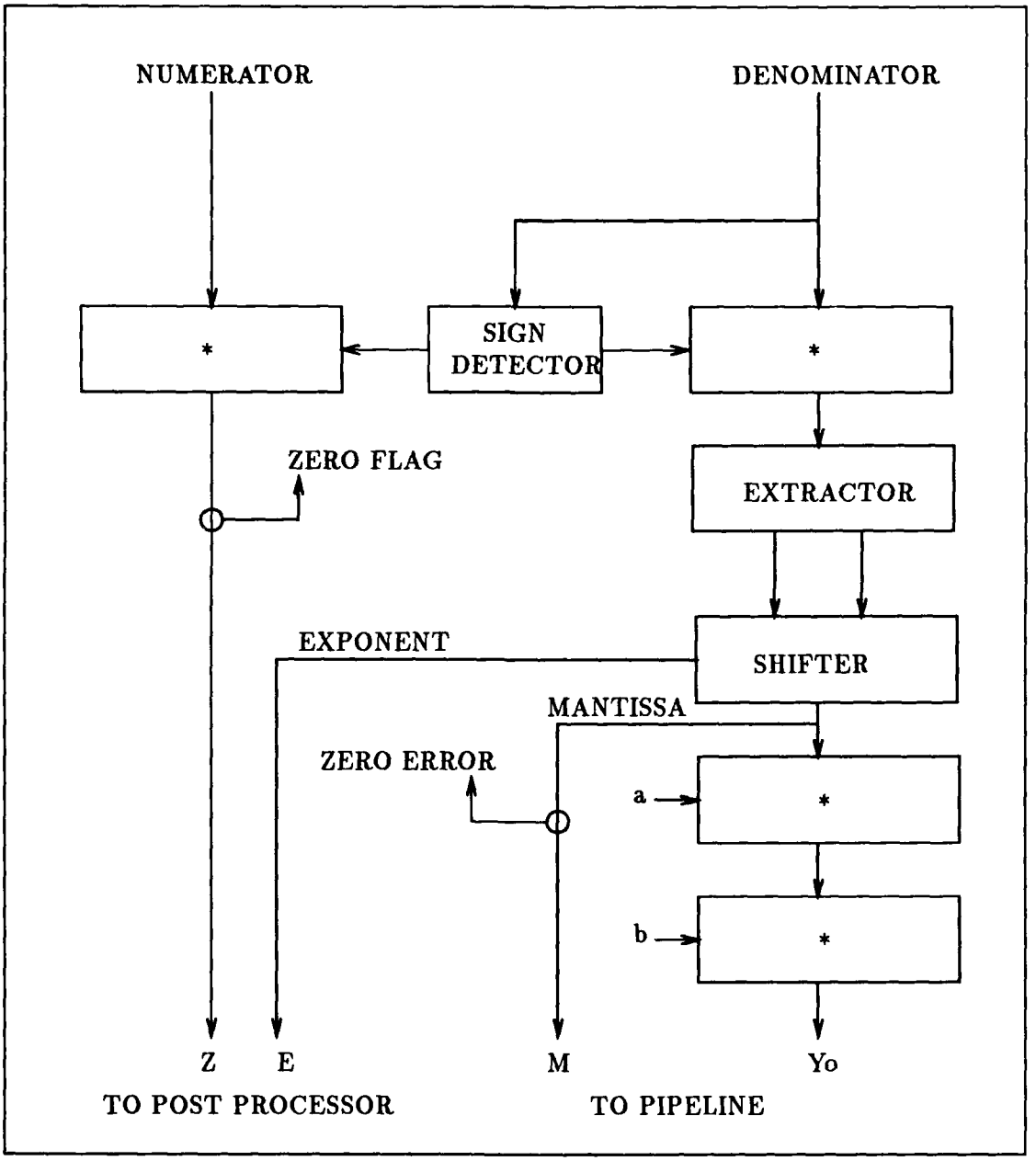

Figure 3.6. Division Pre-processing Requirements.

functions only use the x^2 term,

$$f_{even}(x) = c_0 + x^2(c_2 + x^2(\cdots(c_n + x^2)\cdots)),$$

while odd functions use both terms,

$$f_{odd}(x) = x(c_1 + x^2(c_3 + x^2(\cdots(c_m + x^2)\cdots))).$$

Functions which are neither even nor odd only use the x term.

$$f_{neither}(x) = c_0 + x(c_1 + x(\cdots(c_k + x)\cdots))$$

Therefore, the first stage of the pipeline, as shown in Figure 3.7, takes its argument and squares it. Then, the argument and its square are propagated down the entire length of the pipeline, with the pipeline control section selecting the argument to use, depending on the function being computed. The control section also selects the coefficients to sum with the product result from the previous stage.

This leads to the development of a control pipeline where, as the data advances down the data pipeline, a control word advances down a control pipeline, selecting coefficients and arguments for the data pipeline at each stage.

The division algorithm is the only algorithm not derived from Chebyshev Polynomials. Its general form is

$$Y_{i+1} = Y_i(2 - Y_i(0 + x)).$$

The general form shows the requirement of being able to block the propagation of x^2 down the the data pipeline and replacing it with Y_i at select points. This can easily be accomplished by the control word selecting, through the use of a multiplexer, whether to propagate x^2 or the output of the previous stage down the pipeline.

The total number of sum-product stages in the pipeline is developed around the requirement of obtaining, at a minimum, IEEE double precision accuracy. The algorithm requiring the greatest number of sum-product stages is the algorithm which computes Tangent/Cotangent. This algorithm requires 16 sum-product stages to achieve double precision accuracy, even with the limited range of its argument.

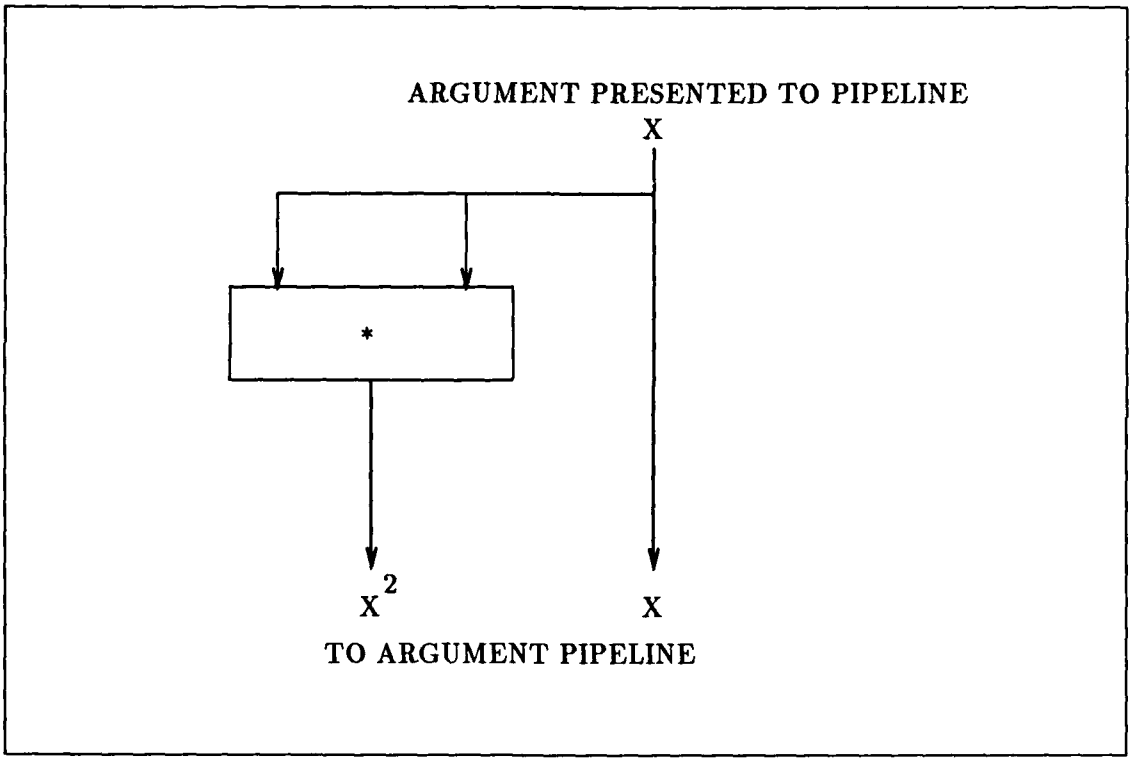

Figure 3.7. Stage One of Pipeline.

Figure 3.8 shows how the architecture of the pipeline is constructed. A total of 16 sum-product stages follow the initial squaring stage. The control word, passing down the control pipeline, selects coefficients and arguments for use in each sum-product stage as well as the argument to be propagated down the x^2 argument pipeline. The result from the last stage is given to the post-processor for computation of the final result.

Post-processor

There is no requirement of post-processing for the Sine/Cosine, Tangent/Cotangent, and Arctangent functions. The result from the last stage of the pipeline is the value which requires scheduling for return to memory or for additional processing. The Exponential function requires a multiplier in the post-processor to multiply the result of the last pipeline stage to the value obtained from a ROM table. This result can then be scheduled for return to memory. The Natural Logarithm function requires a subtractor to subtract the bias out of the exponent, a subtractor to subtract the Natural Logarithm of the scaling factor, obtained from a look-up table, from the result of the last stage of the pipeline, and a multiplier to multiply the two intermediate result. The result from the multiply operation can be scheduled for return to memory. The post-processing requirements of the Division function consist of a subtractor, complementor, and an adder for the exponent to compute the negative exponent of the denominator. A multiplier is also required to multiply the reciprocal of the denominator and the sign adjusted numerator to obtain a final result which can then be scheduled for return to memory.

The architecture of a unified post-processor depends directly on the level of complexity of its control section. At one extreme, the control section is relatively simple, having *dummy* stages in the post-processor such that all functions require the same number of clock cycles through the post-processor. At the other extreme; a complex control section has each post-processor operation selected by the control logic and the results, which require minimum computation, are scheduled for return to memory before results which require more computation, even though they may have arrived at the post-processor first.

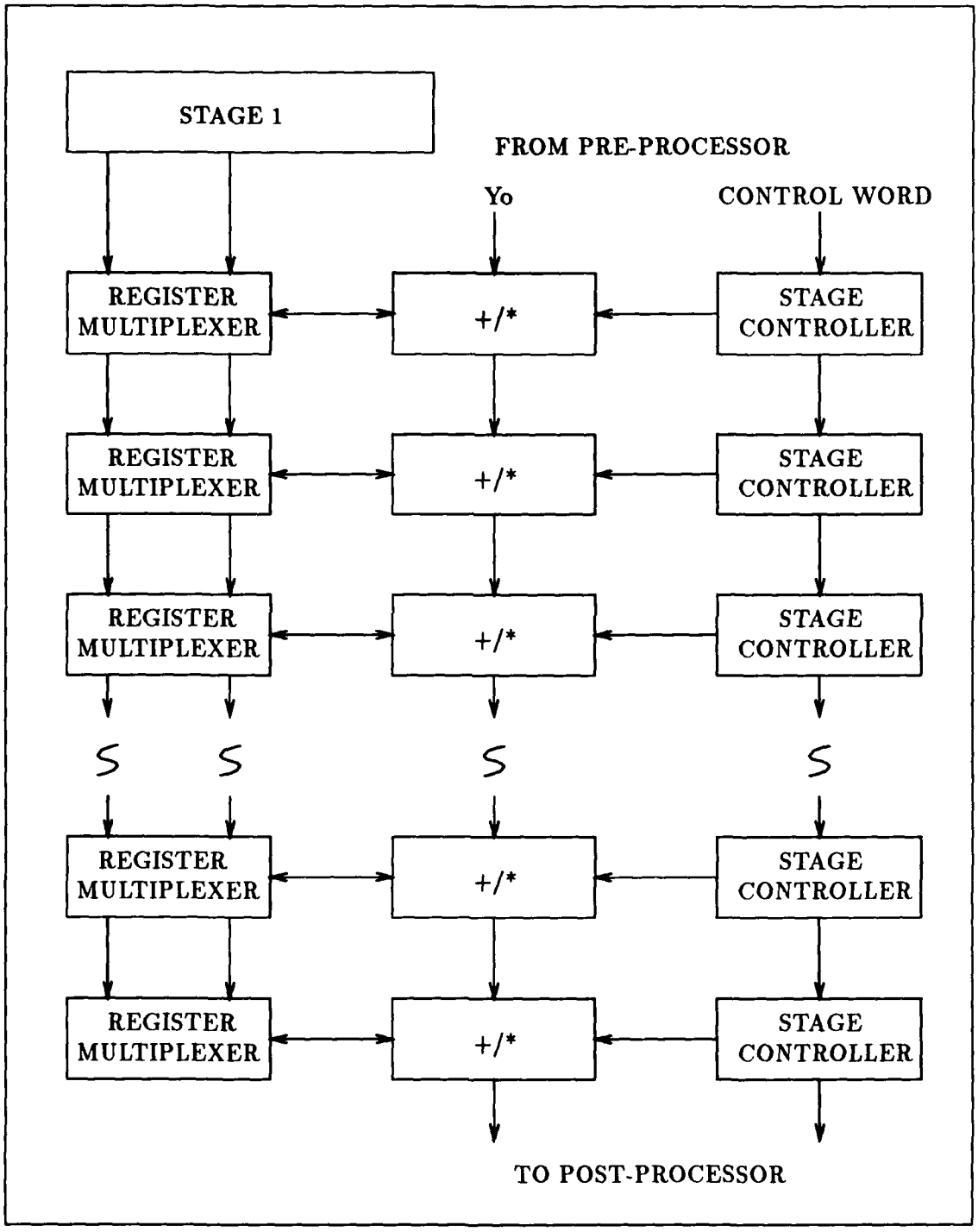

Figure 3.8. Pipeline Architecture.

IV. Intra-Processor Data Representation

Alternate Data Representations

The Transcendental Function Processor requires a look into alternate data representation schemes. The motivation behind this is to achieve the greatest speed from the algorithms and hardware designs before looking at the speed-up possible from different technologlies used to construct the hardware. By looking at alternate data representation schemes, the hardware design advantages may be analyzed.

The large number of sum-product stages in the processor warrant the analysis of data representation schemes which can make the computations faster. The primary method of speeding up the multiplication and addition operations is by reducing the carry-barrow propagation delay throughout each hardware component. The problem of propagation delays of the carry is not a significant problem with exponents but it is significant with mantissa values. This difference is due to the relative sizes, or number of bits, of each. The number of bits in the exponent of an IEEE double precision numbers is 11 whereas the number of bits of the mantissa is 52. The propagation delay across 52 bits is significant. There have been many methods proposed to eliminate the problem of carry-barrow propagation delays. Data representation schemes which have been studied in great depth include the Residue Number System, and the Signed-Digit Number System, [9]. The Residue Number System is a digit oriented system where no weighting factor is assigned to any digit. Instead, a residue number is represented by an n-tuple, n, which relates to another n-tuple, m, where m is a set of relatively prime numbers and n is a set of numbers which represent a *modulo* factor of each element in m such that the sum, for all pair wise elements in the sets, is the value of n. The major problems with this system are the digit set pairing and normalization of a residue number is not practical. Therefore, precision can not be maintained for all representable values. A number system similar to the Residue Number System is the Negative Base System; however, it has the additional complexity of determining the sign of the number.

The Signed-Digit Number System is a system which allows for a great amount of flexibility. A number is represented by a set of digits where each digit can only take on a value in the set D_ρ. The digit set, D_ρ, is a balanced set where both η and $-\eta$ are elements and $(-\rho \leq \eta \leq \rho)$. A Signed-Digit (SD) number is composed of digits which are positional weighted using some radix. This gives a degree of redundancy to the representation a number depending on the value of ρ in D_ρ.

Regardless of what alternate data representation form is used, there is a cost associated with using it. The costs occur from the requirement to convert numbers represented in the conventional form to, and from, the alternate representation. As long as these costs are out weighed by the benefits of the alternate representation, the alternate representation should be considered.

Signed-Digit Data Representation

As stated previously, a Signed-Digit number is composed of a set of digits where each digit is positionally weighted and is an element of the digit set D_ρ. SD number representation has the primary advantage of being free of carry propagation delays. The SD Number System has four basic properties associated with it [13, 12].

1. The radix r, associated with the positional weighting, is a positive integer.
2. Zero is represented by a unique set of digits.
3. Totally parallel addition and subtraction are possible.
4. There exist transforms between conventional data representation schemes, such as IEEE form, to SD representations.

The SD number, Z, is expressed as

$$Z = \{Z_0, Z_1, Z_2, Z_3, \cdots, Z_n\}$$

corresponding to

$$Z = Z_0 r^0 + Z_1 r^{-1} + Z_2 r^{-2} + \cdots + Z_n r^{-n}.$$

Each digit in Z is an element of the digit set D_ρ where

$$D_\rho = \{-\rho, 1-\rho, 2-\rho, \cdots 0 \cdots \rho-1, \rho\}$$

In general, the maximum value of ρ is

$$\rho_{max} \leq r-1$$

and its minimum value

$$\rho_{min} \geq \left\lceil \frac{r-1}{2} \right\rceil .$$

The above are general constraints defined by [12]. More specific constraints on ρ defined by [13] are

$$\rho_{max} \leq r-2$$

and

$$\rho_{min} \geq \left\lceil \frac{r-1}{2} \right\rceil + 2.$$

The more restrictive constraints on ρ simplifies the normalization procedure of a SD number. Another feature of SD numbers is that each digit carries it own sign and the sign of the SD number is given by the sign of its most significant non-zero digit.

Because the digit set D_ρ is balanced and each digit carries it own sign, numbers represented as SD may have a degree of redundancy associated with them. A minimally redundant SD Number System is defined as one where

$$\rho = \left\lceil \frac{r-1}{2} \right\rceil$$

; if $r = 16$, this defines a digit set where $\rho = 8$. Using this digit set, and two digits to represent a number, only one number can be represented in a redundant manner. For example, if the number 0.5 decimal is the number to be represented using a minimally redundant digit set where $r = 16$, it may be expressed as

$$Z = (1)r^0 + (-8)r^{-1}$$

or as

$$Z = (0)r^0 + (8)r^{-1}.$$

No other number may be expressed in this redundant fashion. In a maximumly redundant digit set, one where

$$\rho = r - 1,$$

all numbers except 0 are representable in a redundant manner. Zero is not representable in a redundant manner because $\rho_{max} = r - 1$ and a redundant representation of zero violates one of the four basic properties of the Sign-Digit Number System. The level of redundancy in a chosen system effect other aspects than simply the way which numbers can be represented. When a maximumally redundant digit set is chosen, the conversion transform between conventional representations and SD representations is simple. However, the normalization procedure is made complex. The opposite is true for a minimally redundant digit set, conversion is difficult but normalization is simple. The digit set for any SD Number System will range between these to extremes. When selecting the digit set, done by the selection of ρ, the tradeoffs between the chosen degree of redundancy and the complexity of the hardware must be examined. In a system where a number is converted, used extensively, and then assimilated back to a conventional representation, the frequency of the conversion process is much less than the frequency of normalization. Therefore, in this system, a digit set which is minimally redundant should be chosen. The opposite is true when the frequencies of conversion and assimilation approaches the frequency of normalization. The majority of the work presented in literature [13, 12, 15] has shown that when in an environment where the frequency of normalization is greater than the frequency of conversion, such as in a pipelined processor, $\rho = 10$ yields the best tradeoff between conversion and normalization complexities.

The normalization of a SD number is preformed by the shifting of digits and adjusting of the exponent. A SD number is normalized if

1. The most significant digit, $|Z_0|$ is 1 and $|Z_0 + Z_1 r^{-1}| \leq 1$ or
2. If $Z_0 = 0$ and $|Z_1 r^{-1} + Z_2 r^{-2}| \geq r^{-1}$ or
3. If all of the digits are 0.

Since normalization shifts digits and not bits, the exponent is adjusted by the binary equivalent of the log base 2 of the radix for each shift. The exponent of a SD number may

be represented in either SD or conventional form; however, by keeping it in a conventional form, the conversion, assimilation, and alignment processes are kept relatively simple. However, during the alignment process for addition, if the exponents are not the same or some multiple of the log base 2 of the radix apart, alignment cannot occur. Therefore, during the conversion process, the exponents must be adjusted such that all numbers represented in SD form have exponents which are a multiple of the log base 2 of the radix apart. This is done by shifting the conventionally represented input such that the n least significant bits, where $2^n = \log_2 r$, are the same for all SD number exponents. When the radix equals 16, the two least significant bits of the exponent are required to be the same.

Signed-Digit Numeric Units

The SD numeric units for the processor consist of the conversion, adder/subtractor, multiplier, and assimilation units. The conversion and assimilator units have only a single input while the adder/subtractor and multiplier each have two primary inputs. The processor represents SD numbers with radix-16 weighting and a minimally redundant digit set, $\rho_{max} = 10$. Each numeric unit is constructed from a common set of macrocells to be described in detail later.

Conversion Unit The conversion unit takes, as its input, a single number represented in some conventional form, such as IEEE double precision. Before the input can be operated on, it must be check to insure that it is a legal number and not an infinity or NaNs [6]. If the input is not a legal number, an error signal is generated and the conversion process aborted. However, if the input is a legal number, the conversion process begins. To explain the conversion process, an IEEE single precision number is used.

A single precision floating point, number is represented by a 23 bit mantissa, an 8 bit exponent, and a single sign bit. The mantissa has an implied 1 in front and is expressed as $1.XXXXX\cdots XX$ which can represent a value in the range $(1.0 \leq M < 2)$. To convert the number to SD representation, the range of the mantissa should be $[1/r, 1)$ which simplifies the normalization of the SD number after conversion to, at most, one left shift. Therefore, if the mantissa is shifted right one to four bit places, it is within the range required for SD

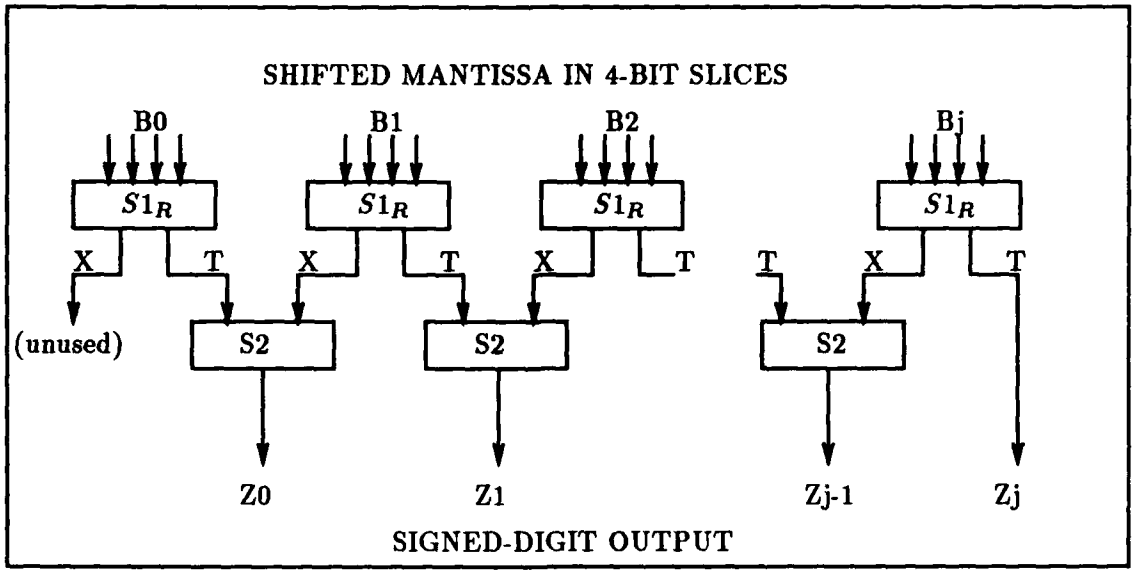

Figure 4.1. Conversion Recoding Hardware and Data Flow.

conversion. The number of places to shift the mantissa is determined by the exponent. The exponent is expressed by 8 bits and has a bias value of +127; the range of the un-biased exponent is −126 to +127. The two ends of the possible range of the biased exponent, 0 and 255, are used to represent 0 and ±inf which are handled separately. To convert to a radix-16 SD number, the exponent must have the form $XXXXXX00$. Therefore, the number of right shifts to the mantissa is equal to 4 minus the value represented by the two least significant bits of the exponent. This always shifts the mantissa at least one place. The only time the result will not be within the required range for SD conversion is when the mantissa is $1.000\cdots00$ and the exponent is XXXXXX00. This is the only condition where the mantissa requires zero shifts. Once the mantissa is shifted and the exponent is adjusted to reflect the right shifts, the SD conversion may occur.

SD conversion is a recoding process in which its input, the shifted mantissa, is split into four-bit slices and recoded to adhere to the SD digit set, D_ρ. Figure 4.1 shows the conversion recoding hardware and data flow. The shifted mantissa is input into a recoder, $S1_R$, and recoded such that the output of $S1_R$ is X and T, where X and T are elements

in the digit sets D_x and D_t respectively and whose value is related to the input by the function

$$B_i = Xr + T.$$

The digit set D_x is required to consist of the elements $\{0, 1\}$. The digit set D_t is determined by the requirement that

$$\left\lceil \frac{r-1}{2} \right\rceil \leq T_{max} \leq \rho.$$

When $\rho = \lceil ((r-1)/2) + 2 \rceil$, T_{max} should equal $\lceil (r-1)/2 \rceil$, [13]. This makes the digit set D_t minimally redundant when used with D_x.

$$D_t = \{-8, -7, \cdots, -1, 0, 1, \cdots, 7, 8\}$$

The outputs of $S1_R$ are the inputs into the summer $S2$. This summer adds the inputs, X and T, and outputs the digit Z which is an element of D_ρ. All digits are expressed in binary twos complement format. The sign bit of the floating point number is used below the $S2$ level to determine the correct representation of Z, either Z or its 2's complement. A simple example of the conversion process is shown in Figure 4.2. A mantissa of 12 bits and an exponent of 4 bits are shown for simplicity, one sign bit.

The value of the input, expressed in radix-16, to the conversion unit is

$$-(0 \cdot 16^0 + 3 \cdot 16^{-1} + 11 \cdot 16^{-2} + 14 \cdot 16^{-3}) = -3 \cdot 16^{-1} - 11 \cdot 16^{-2} - 14 \cdot 16^{-3} \quad (4.1)$$

The second term on the right hand side of expression 4.1 may be re-expressed as

$$-11 \cdot 16^{-2} = (5 - 16) \cdot 16^{-2} = 5 \cdot 16^{-2} - 16 \cdot 16^{-2} = 5 \cdot 16^{-2} - 1 \cdot 16^{-1}.$$

Similarly, the third term may be re-expressed as

$$-14 \cdot 16^{-3} = (2 - 16) \cdot 16^{-3} = 2 \cdot 16^{-3} - 16 \cdot 16^{-3} = 2 \cdot 16^{-3} - 1 \cdot 16^{-2}.$$

The right hand side of the expression 4.1 may be re-expressed as

$$-3 \cdot 16^{-1} + (5 \cdot 16^{-2} - 1 \cdot 16^{-1}) + (2 \cdot 16^{-3} - 1 \cdot 16^{-2}) = -4 \cdot 16^{-1} + 4 \cdot 16^{-2} + 2 \cdot 16^{-3}.$$

This expression is the same as the final conversion results shown in Figure 4.2. After conversion, the exponent is carried along with the SD number and used the same as in

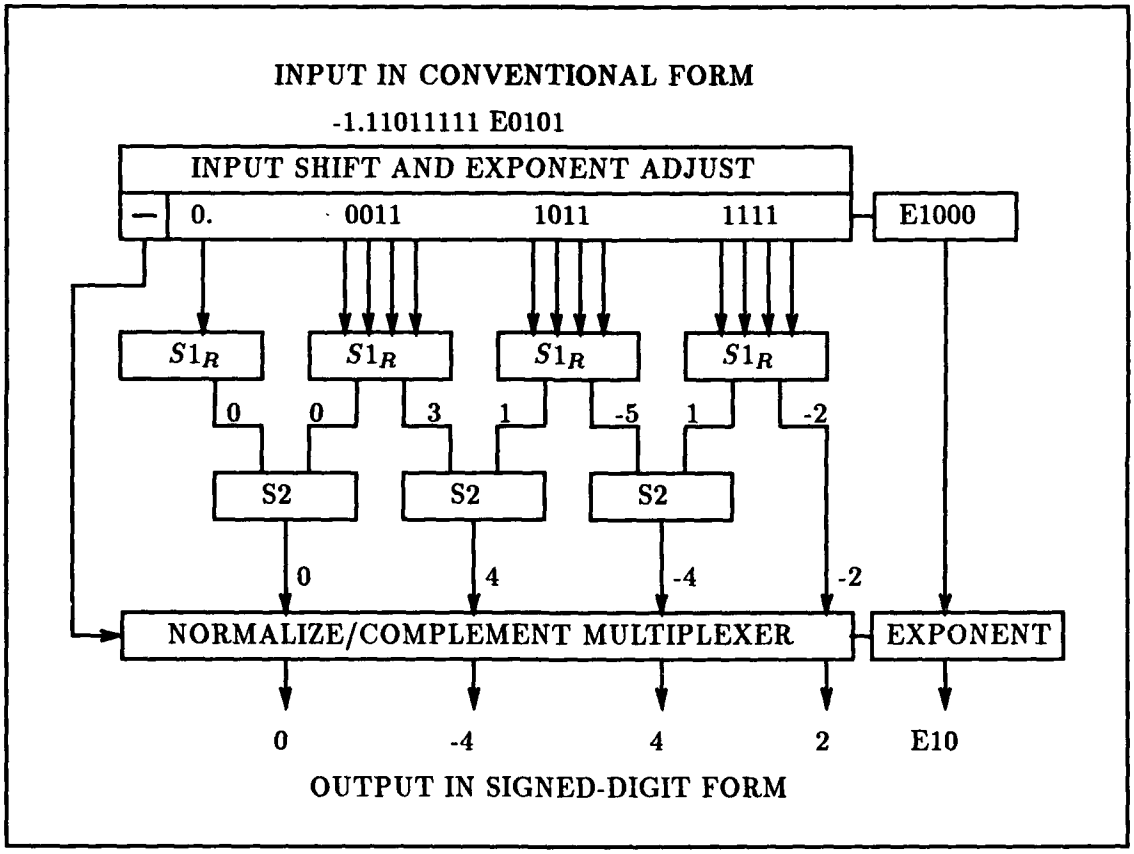

Figure 4.2. Conversion Recoder Example.

standard floating point arithmetic. However, the exponent has two less bits since the two least significant bits are dropped because they are assumed 0. A block diagram of the conversion stage is shown in Figure 4.3. As stated previously, the SD number out of the conversion process may require, at most, one left shift to normalize.

The level of complexity of conversion is minimal; however, an additional stage in the pipeline is required. This disadvantage must be offset by some advantage in addition/subtraction, and multiplication.

Adder/Subtractor Unit Addition is very similar to the conversion process with only minor exceptions. The first change is the alignment of the exponents. This is simpler than in standard representation since the exponents are two bits shorter and the number of digits to shift are less than the number of bits to shift in standard floating point. Then, instead of the recoder $S1_R$ having a single input, $S1_A$ is a summer and has, as its input, two numbers in SD format. The outputs of $S1_A$ are X and T, but the digit set D_x must now include a -1. The digit set for T, D_t, is unchanged. The summer $S1_A$ performs the function

$$Xr^{1-i} + Tr^{-i} = IN1r^{-i} + IN2r^{-i}.$$

The maximum sum of the inputs is defined by 2ρ and gives a maximum sum of 20. This range is covered by the range of $Xr + T$. The summer $S2$ is unchanged with the exception of the required -1 in the input digit set of X. The normalization of a SD number after addition requires, at most, one right shift or multiple left shifts. Rounding is required if a right shift occurs and is discussed at the end of the multiplication section. The complexity of SD addition is of the same order as the conversion process. In comparison to standard binary addition, the alignment of the exponents must still occur, though the exponents are two bits shorter for a SD number. Also, the maximum carry propagation for a number expressed in SD form is 1 digit; whereas, a number expressed in binary may require a carry propagation across its entire field. This is the benefit of SD addition over standard binary addition. The SD Adders data flow is shown in Figure 4.4 for four digit addition, less exponent adjust, normalization and rounding.

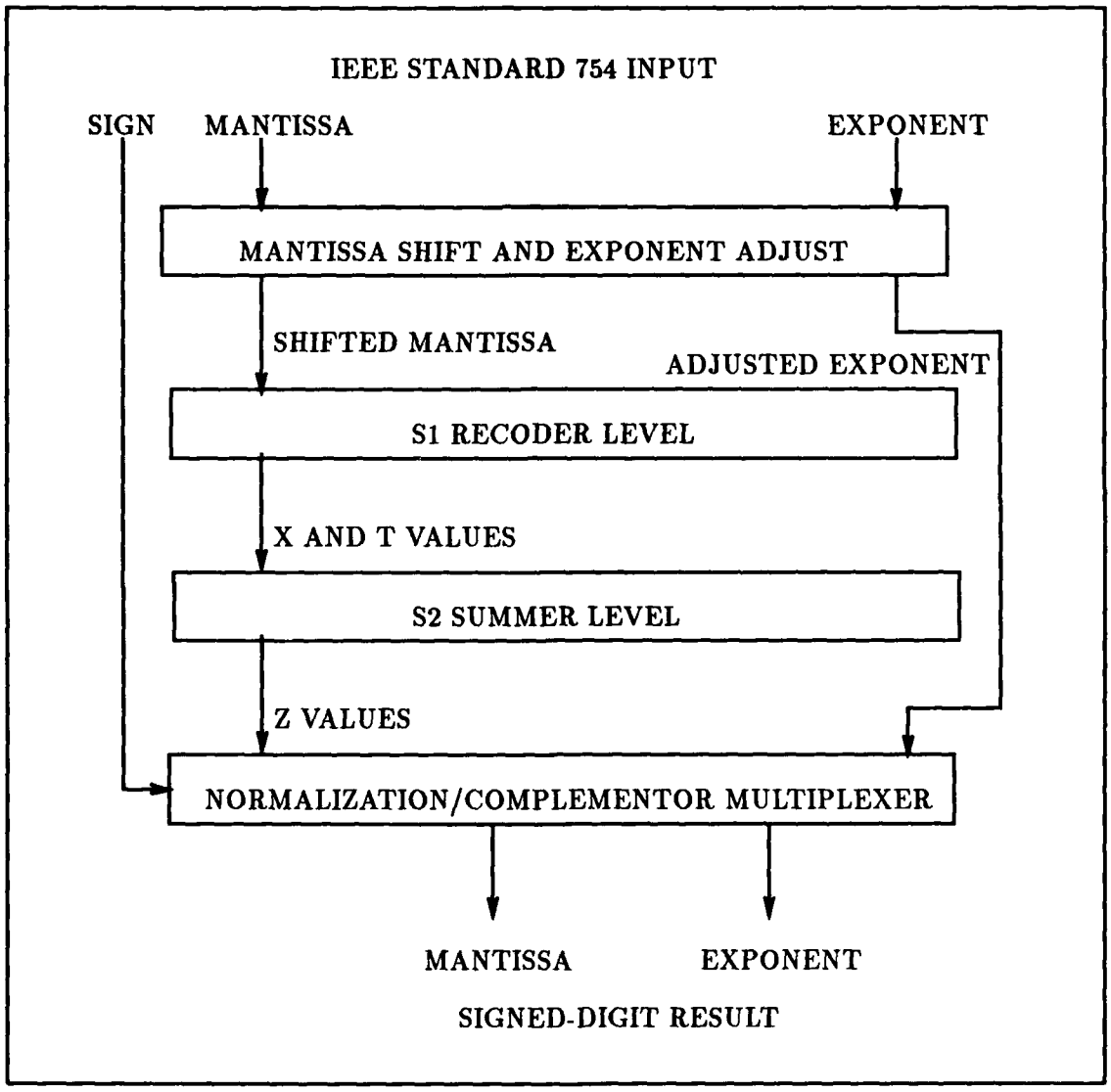

Figure 4.3. Block Diagram of Conversion Stage.

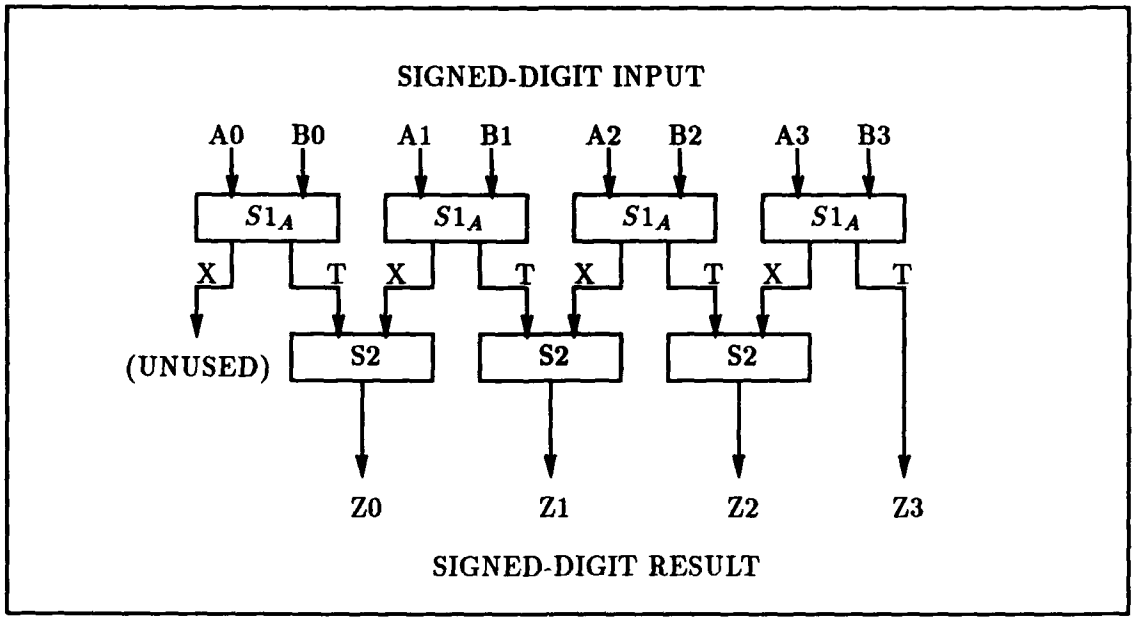

Figure 4.4. Data Flow in SD Adder.

SD subtraction is essentially the same as SD addition with the following exception. Prior to the $S1_A$ level, the digit to be subtracted is 2's complemented. The remainder of the the circuit is unchanged. A block diagram of the addition/subtraction unit is shown in Figure 4.5.

Multiplier Unit The SD Multiplier computes all of the partial products in parallel, in its first level. The next levels sum the partial products, two at a time, until a single result is obtained. Then, the result is normalized, rounded and a final result obtained. The multiplier stage used to compute the partial products is discussed first due to the additional digit sets used in the multiplication scheme which have not been presented yet.

A single digit multiplier, $M0$, is shown in Figure 4.6. The two additional digit sets for the multiplier are D_u and D_w from $M0$. The maximum values of these digit sets are determined by the requirement to cover the maximum range of the input product, ρ^2, and the requirement of redundancy for the output. $M0$ multiplies two digits in the digit set

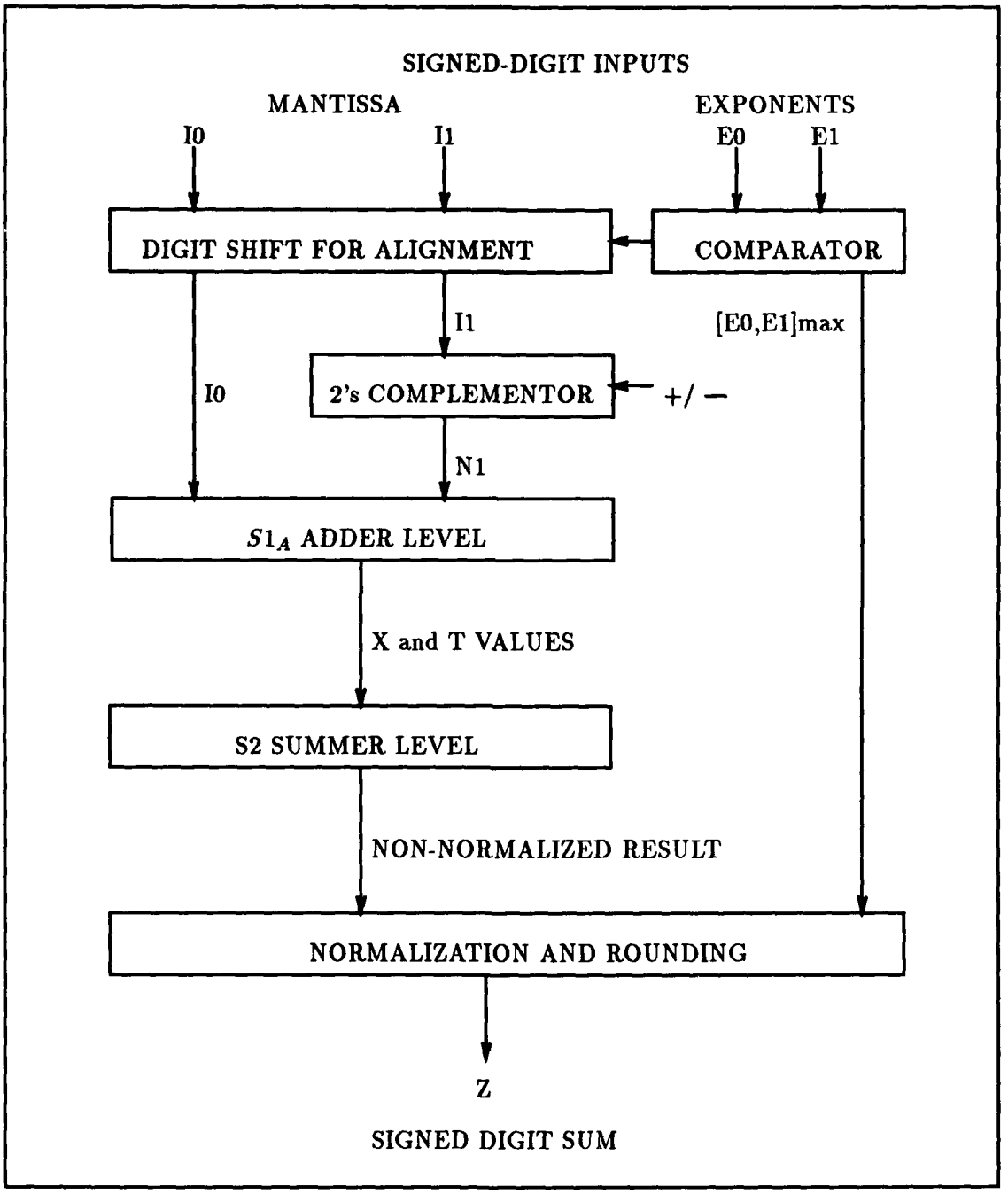

Figure 4.5. SD Addition/Subtraction Unit.

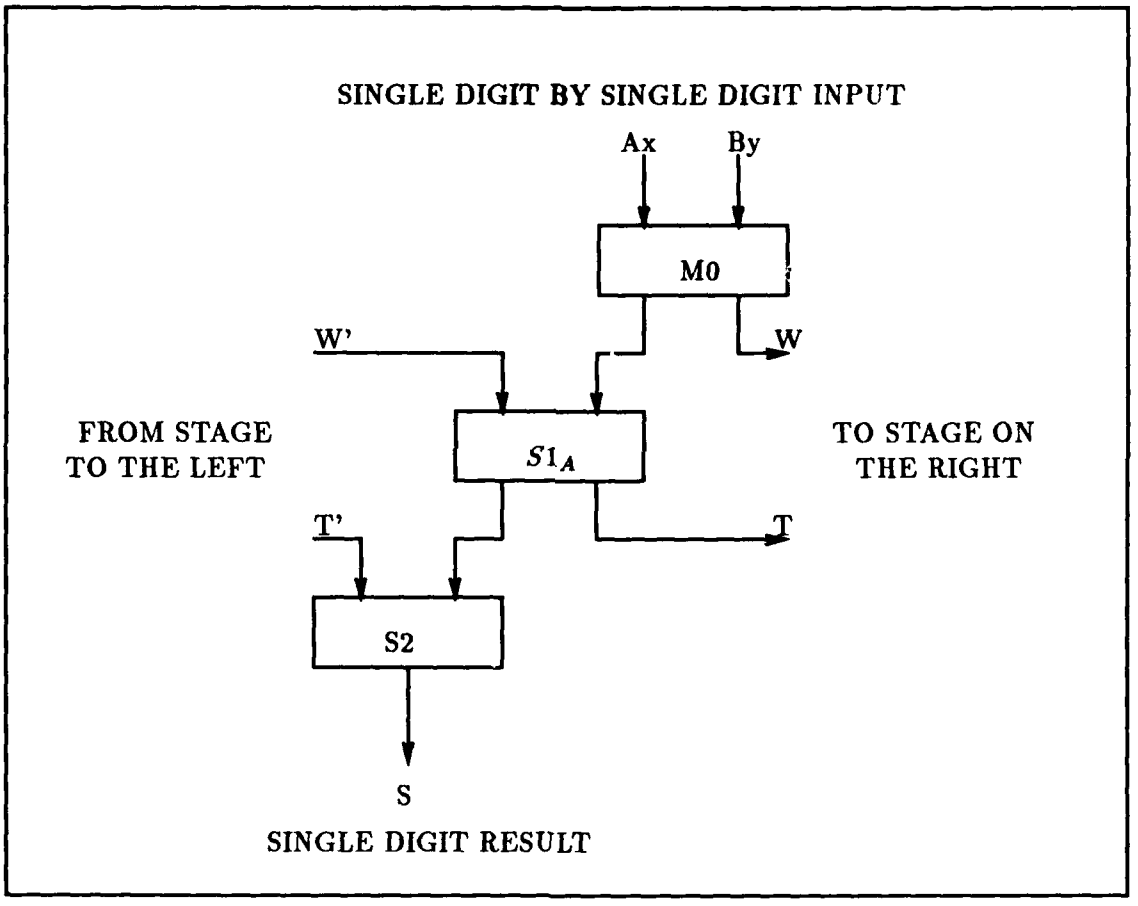

Figure 4.6. Single Digit by Single Digit Multiplier, *M0*

D_ρ and outputs the result as

$$Ur^{1-i} + Wr^{-i} = (Br^{-i}) \cdot (Ar^{-i}).$$

To express the product, in a redundant manner, the digit set of D_w must be, at least, minimally redundant. This requires $W_{max} \geq \lceil (r-1)/2 \rceil$ which is the same requirement for T_{max} discussed earlier. No benefit is achieved by having D_w more than minimally redundant but there is a cost in attempting to do so as the complexity of the entire multiply hardware increases as the redundancy increases. Therefore, D_w is established as a minimally redundant digit set.

$$D_w = \{-8, -7, \cdots, -1, 0, 1, \cdots, 7, 8\}$$

The required digit set for U can now be established. Since the maximum absolute value of the product of $|AB|$ is $100, \rho^2 = 100$, then

$$U_{max} = \left\lceil \frac{100 - W_{max}}{16} \right\rceil = 6$$

With these two digit sets, D_u and D_w, the entire range of the product of A and B is representable with minimal redundancy. The remaining digit sets in the multiplication scheme above, D_t and D_x, are unchanged from their definition given earlier, with the exception of $D_{t'}$ not equal to D_t. This will be explained later in this section.

The digit sets used for multiplication are D_ρ, D_w, D_u, D_t, and D_x; the values in each digit set are

$$\begin{aligned}
D_\rho &= \{-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \\
D_w &= \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8\} \\
D_u &= \{-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6\} \\
D_t &= \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8\} \\
D_x &= \{-1, 0, 1\}
\end{aligned}$$

In the Variable Precision Module presented by [13], the digit set of T' is allowed to be larger than the digit set of T, D_t. $D_{t'}$ may be as large as D_{t+x}. This increases the

size of $D_{t'}$ by 1 on each side of the symmetric set over D_t. Since the partial products are computed in full parallel and not in serial or in an array structure, the additional size of the digit set $D_{t'}$ is not required. However, to optimize later aspects of the multiplication scheme, specifically during the addition of the partial products to form the end result, the ability of inputing a T' in $D_{t'}$, as defined above, will prove useful at no cost.

In Figure 4.6, it is important to note the shifting of the resultant output as compared to the input. The most significant digit output is two digit places to the left of the most significant input digit. This is because of the output of $M0$, which outputs Ur^1, and the outputs of $S1_A$, which outputs Xr^1. Therefore, the resultant output, Z_{-2}, is r^2 times the digit place of the inputs.

For a full parallel multiplier to multiply a complete SD number, B, by a single digit, A_k, the single digit multiplier stage is duplicated for each digit in B. The result of replicating the stage is shown in Figure 4.7 and forms a full parallel multiplier block.

Since the computation of the partial products occurs in parallel, W'_0 and T'_0 are always 0. This simplifies the left most stage of the block. $S1_{A0}$ and $S2_0$ are not required because the maximum value of U out of $M0_0$ is $U_{max} = 6$ which, when added to 0 in $S1_{A0}$, results in $X = 0$ and $T_{max} = 6$. Therefore, $S1_{A0}$ may be removed completely and U, from $M0_0$, can go directly to the T input of $S2_1$. $S2_0$ is not required because both of its inputs are always 0. This eliminates $S1_{A0}, S2_0$, and the S_{-2} output.

To multiply two SD numbers, the multiplier block, shown in Figure 4.7, is replicated so that each digit in A is used to form a partial product with the number B.

The remaining levels of the multiplier unit sum the partial products after shifting the products to correct for the decimal point position of A_k. The following discussion simplifies the summation levels by reducing the number of digit adders required in each level. What must be kept in mind is that the inputs to the multiplier are normalized SD numbers. This is important because significant savings in the amount of hardware required to sum the partial products will result.

Because the inputs are normalized, the maximum absolute value of B_0 is 1. If B_0 is 1 then B_1 is either 0 or it has the opposite sign of B_0. This is a requirement of a

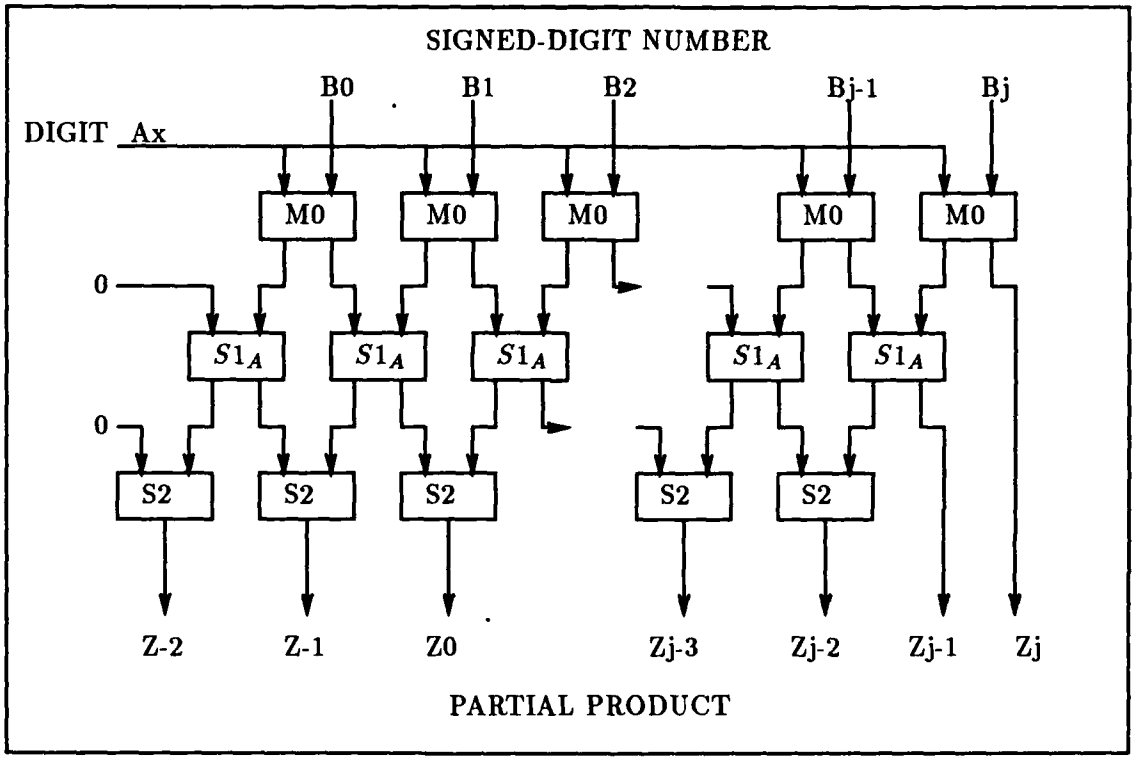

Figure 4.7. Single Digit by SD Number Multiplier Block.

normalized number; if $|B_0| = 1$ then $|B_0 + B_1|$ must be less than, or equal, to 1. Therefore, the maximum value of the resultant U out of $M0_0$ is 1; and, if $|U| = 1$ then W out of $M0_0$ must be $5 \leq |W| \leq 8$ and have the opposite polarity of U. The U out of $M0_1$ is in the range $0 \leq |U| \leq 6$ and has the same polarity as W out of $M0_0$. This is all with the condition that U out of $M0_0$ is not 0. Since W and U into $S1_{A1}$ have the same polarity, then $5 \leq |W + U| \leq 14$ and the sum has the opposite polarity of U out of $M0_0$. The resultant X out of $S1_{A1}$ has the same sign as $(W + U)$. Therefore, the inputs into $S2_1$ are U, from $M0_0$, and X, from $S1_{A1}$, with the constraints that $|U| = 1$ and X is $X = 0$ or $X = -U$. The value of Z_{-1} is the sum of these inputs and is either U, where $|U| = 1$, or 0 giving an $|Z_{-1}|_{max} = 1$. The next condition which needs to be looked at is when U out of $M0_0$ equals 0. When this is the case, $|W| \leq 7$. If W is any value except 0 then $B_0 = 1$ and the same condition holds for B_1 as above. The output U from $M0_1$ must be 0 or in the portion of the digit set D_u which has the opposite sign of W from $M0_0$. The summer $S1_{A1}$ sums W from $M0_0, |W| \leq 7$, and U from $M0_1, |U| \leq 6$ with the constraint of opposite polarity, and outputs an $X = 0$ and a $|T| \leq 7$. Therefore, the inputs into $S2_1$ are both 0 and the output $S_{-1} = 0$. The last condition to look at is when $B_0 = 0$ and B_1 is any element in D_ρ. With this condition, U and W out of $M0_0$ are 0 and U out of $M0_1$ may be any element in the set D_u. The inputs into $S1_{A1}$ are W, from $M0_0$, and U, from $M0_1$. With these inputs, X out of $S1_{A1}$ is 0 and $T = U$. Therefore, the inputs to $S2_1$ are both 0 so the output $Z_{-1} = 0$. These are the only possible combinations that can effect Z_{-1}, therefore, the possible values of Z_{-1} are $\{-1, 0, 1\}$. This proves to be an important fact which reduces the amount of hardware required in the partial product summers. It is also important to note that Z_{-1} will always equal 0 when $A_k = A_0$. The reason for this is as described above when U out of $M0_0$ equals 0, which is always the case when $A_k = A_0$.

As stated previously, the summer levels of the SD multiplier form a tree structure where the number of partial products half as they proceed down the tree. Figure 4.8 shows this tree structure summing eight partial products. The $SL2$ summer sums two partial products, P_n and P_{n+1}, which are shifted one digit position from each other due to the position of A_k with respect to each other. The most significant digit of P_n is, as described above, $-1, 0$, or 1. Therefore, when summing at this level, the most significant

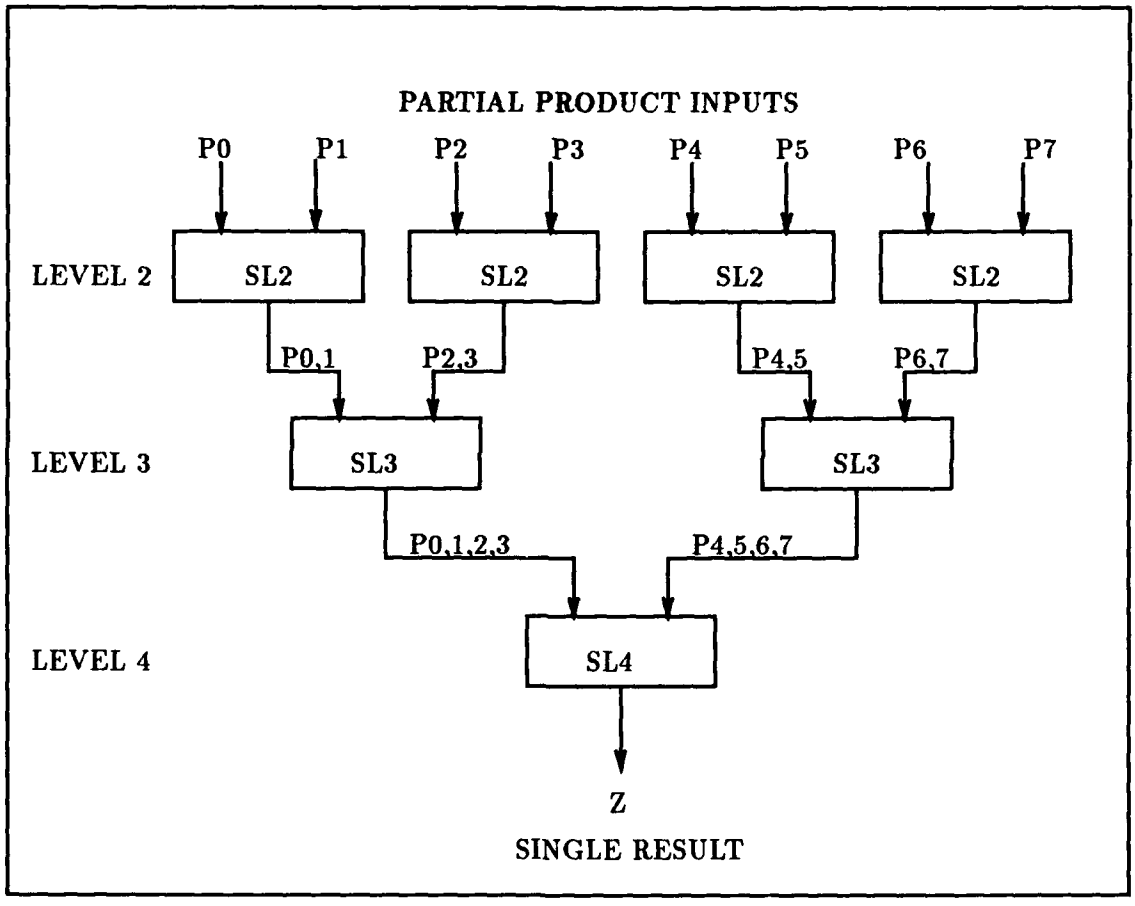

Figure 4.8. Partial Product Summer Structure.

$S1_A$ adder is not required and the digit may be input directly into the most significant $S2$ adder. The least significant digit of P_{n+1} bypasses the $SL2$ summer completely since Pn does not have an input to add with it. The $SL3$ summer sums the results of $SL2$ which are shifted two digit positions from each other. This is where the digit set $D_{t'}$ becomes a factor. If D_t is expanded to the size of $D_{t'}$, then, the most significant digit of $P_{n,n+1}$ bypass the $SL3$ summer and the next most significant digit may be input directly into an $S2$ adder. The maximum magnitude of this next most significant digit is 9 because it is an output from the previous level where $|T + X|_{max} = 9$. The least significant two digits of $P_{n+2,n+3}$ bypass the $SL3$ summer. The $SL4$ summer sums the results from the $SL3$ summers. These inputs are shifted four digit positions from each other. The three most significant digits of $P_{n,n+1,n+2,n+3}$ bypass the $SL4$ summer as well as the four least significant digits of $P_{n+4,n+5,n+6,n+7}$. The forth most significant digit of $P_{n,n+1,n+2,n+3}$ is input into the most significant $S2$ Adder. If more summation levels are required to sum the partial products, this process is continued until a single result is obtained. Once this single result is computed, the result is normalized. The result may require, at most, one digit shift to the right or two digit shifts to the left to normalize after multiplication.

The last step is to round the result to obtain the final output. The maximum round-off error is less than ρ/r^{-j-1} with simple truncation, where j is the number of digits used to represent a normalized SD number. Nearest rounding is easily accomplished in SD number representation. If a SD number is represented by J digits, 0 through $J-1$, then, nearest rounding will affect only the $J-1$ digit. The maximum value of the $J-1$ digit is $|J-1|_{max} = |T+X|_{max} = 9$; and, since rounding can affect the $J-1$ digit by, at most, 1, the maximum value of $J-1$ after rounding is 10, which is in D_ρ. The maximum error by nearest rounding is

$$Error_{max} = \left\lceil \frac{(r-1)/2}{r^{-j-1}} \right\rceil$$

The IEEE Standard 754 – 1985 requires the intermediate result to be computed to a greater precision and then rounded to the precision of its destination. Due to the way multiplication is performed in the full parallel, the least significant digits of the partial products which do not effect the rounding procedure could be dropped. However, very little hardware is saved by doing this and it will not conform to the IEEE standard.

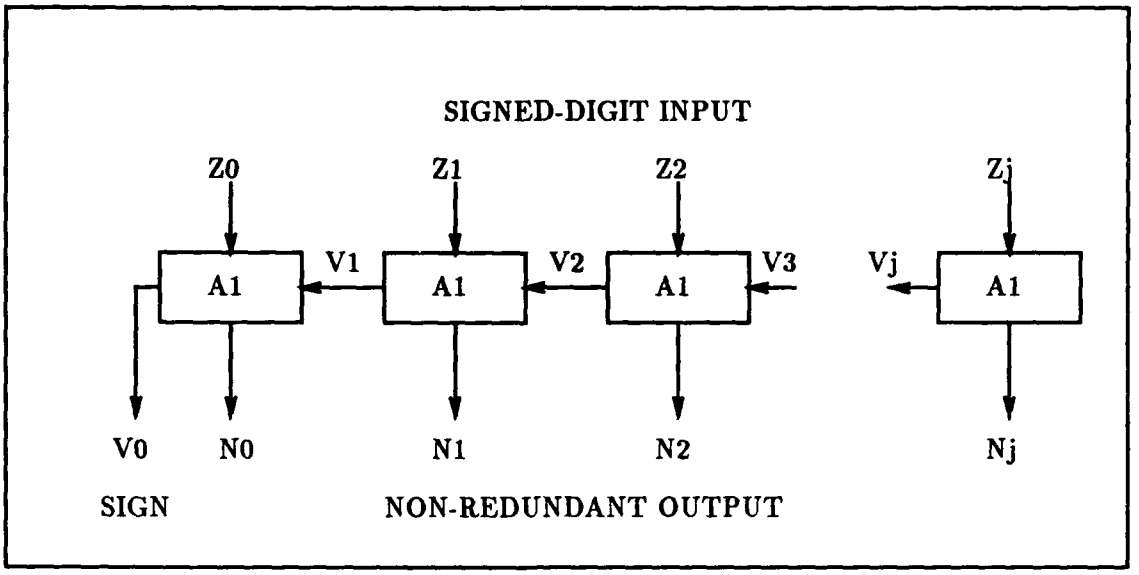

Figure 4.9. SD Assimilator Data Flow.

Assimilation Unit The final unit preforms the assimilation of a SD number to standard binary, such as IEEE floating point. The assimilator is an additional cost of using SD number representation and requires a separate stage in the pipeline. In fact, assimilation is the most costly part of SD representation because this is the only operation with significant carry-barrow propagation delays. The negative SD digits represent the problem. In order to convert the negative digits to positive, the assimilation stage performs the function

$$-r \cdot V_i + N_i = Z_i - V_{i+1}$$

where Z is a SD digit in D_ρ, N is a 4-bit number in non-redundant form, and V is an element in $\{0, 1\}$ which represents a barrow. The assimilator is shown in Figure 4.9. The barrow output from each stage has the possibility of propagating left across all of the stages in this level. The possible values of N_0 are $0, 1, 14$, or 15. If N_0 is 0 or 1, then, V_0 is 0 which indicates that the SD number assimilated is positive. However, if N_0 is 14 or 15 then, V_0 is 1, indicating the SD number is negative, and the output N_i is given in 2's complement form. A second level, in the assimilation process, 2's complements the output and a multiplexer, controlled by V_0, selects which output to pass as the result. The

final levels normalize the result, adjusts the exponent, and forms the final result to IEEE standard. The result may require, at most, four left shifts to normalize. Rounding is also required for the result and is as specified by the IEEE standard. A block diagram of the assimilation process is shown in Figure 4.10. To optimize the time required to perform the assimilation, the 2's complementor and the multiplexer should be placed before the assimilator. To perform a 2's complement on a SD number takes substantially less time than a standard binary number.

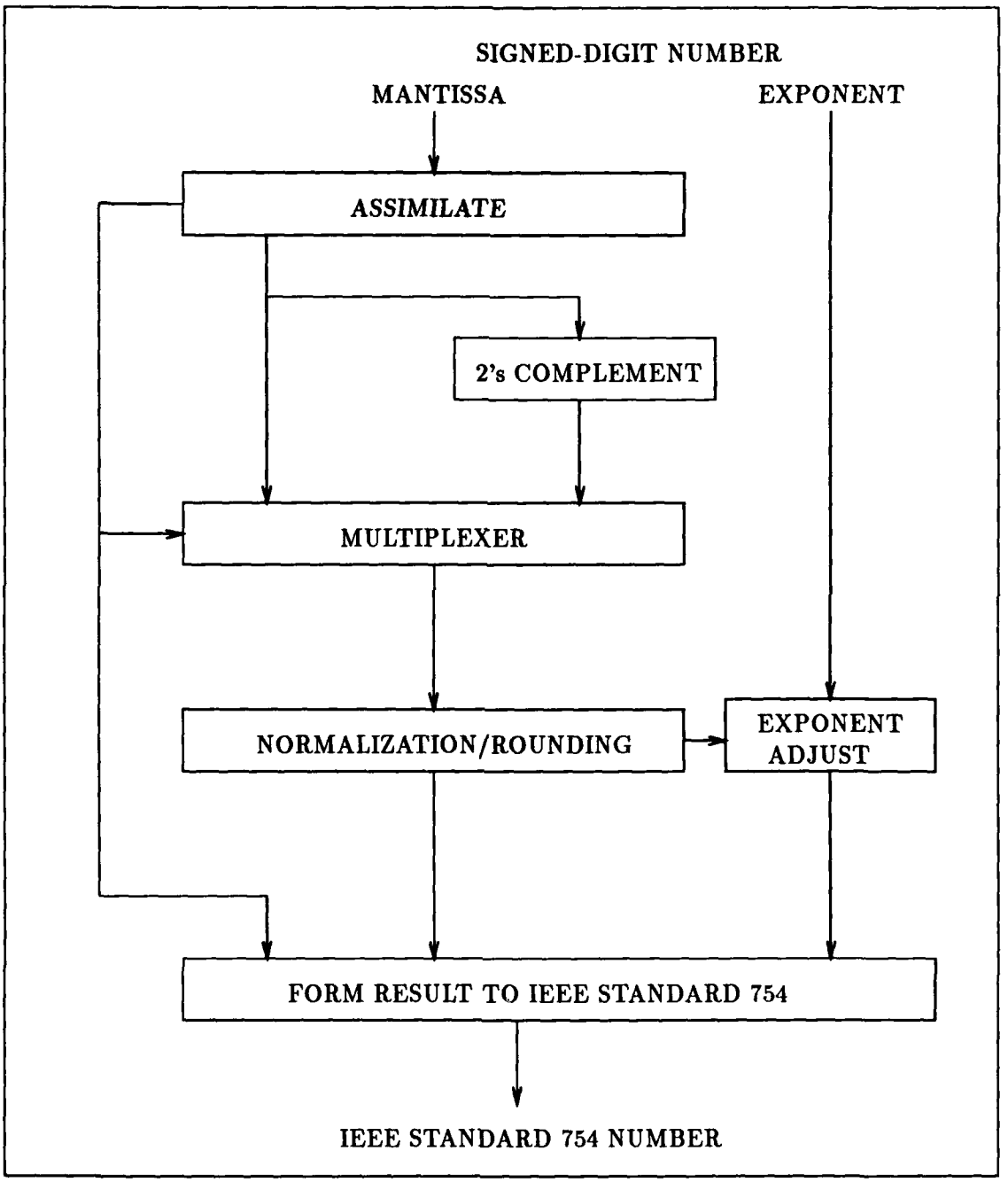

Figure 4.10. SD to IEEE Assimilator.

V. Signed-Digit Hardware Modules

When representing a number in SD form and performing operations on it, unique hardware must be designed. Since SD representation has great advantages over standard binary, these advantages should be exploited in the hardware. The primary modules used for the SD operations presented in Chapter 2 are the $S1_R$ Recoder, $S1_A$ Adder, $S2$ Adder, $M0$ Multiplier, and the $A1$ assimilator. Each of these are discussed as well as their estimated performance parameters. The performance parameters are obtained through the use of SPICE analysis. CIFPLOTs of the $S1_A$ Adder, $S2$ Adder, and $M0$ Multiplier are in Appendix B.

$S1_R$ Recoder

The $S1_R$ Recoder is the simplest of all SD hardware modules. It accepts a 4-bit slice input and outputs X and T in the digit sets D_x and Dt respectively. The input is expressed in binary non-redundant form which gives it a range of number representation from 0 to 15. The digit set of X is $\{-1, 0, 1\}$ and represents a radix-16 higher value than the least significant bit of the input. The digit set of T is $\{-8, -7, \cdots, 0, \cdots, 7, 8\}$ and represents a value which has the same positional weighting as the least significant bit of the input. Both X and T are represented in 2's complement form, as are all numbers in SD representation. The input is recoded by the $S1_R$ Recoder such that any value of input is recoded into X and T by the function

$$N = Xr + T.$$

For all input values in the range $(0, 8)$ the value may pass directly to T. However, if the input is in the range $(9, 15)$, the value of X is 1 while the value of T is $16 - N$. By analyzing the possible inputs and their required results, a simple solution is developed. When the input is in the range $(0, 7)$, its most significant bit is 0. When the input is greater than 7, the most significant bit is 1. Therefore, the $S1_R$ Recoder is designed without the use of any logic gates, it is simply a routing problem. The input is routed directly to T; however, the input is 4-bits wide while T is 5-bits wide. For sign extension of T, the most significant

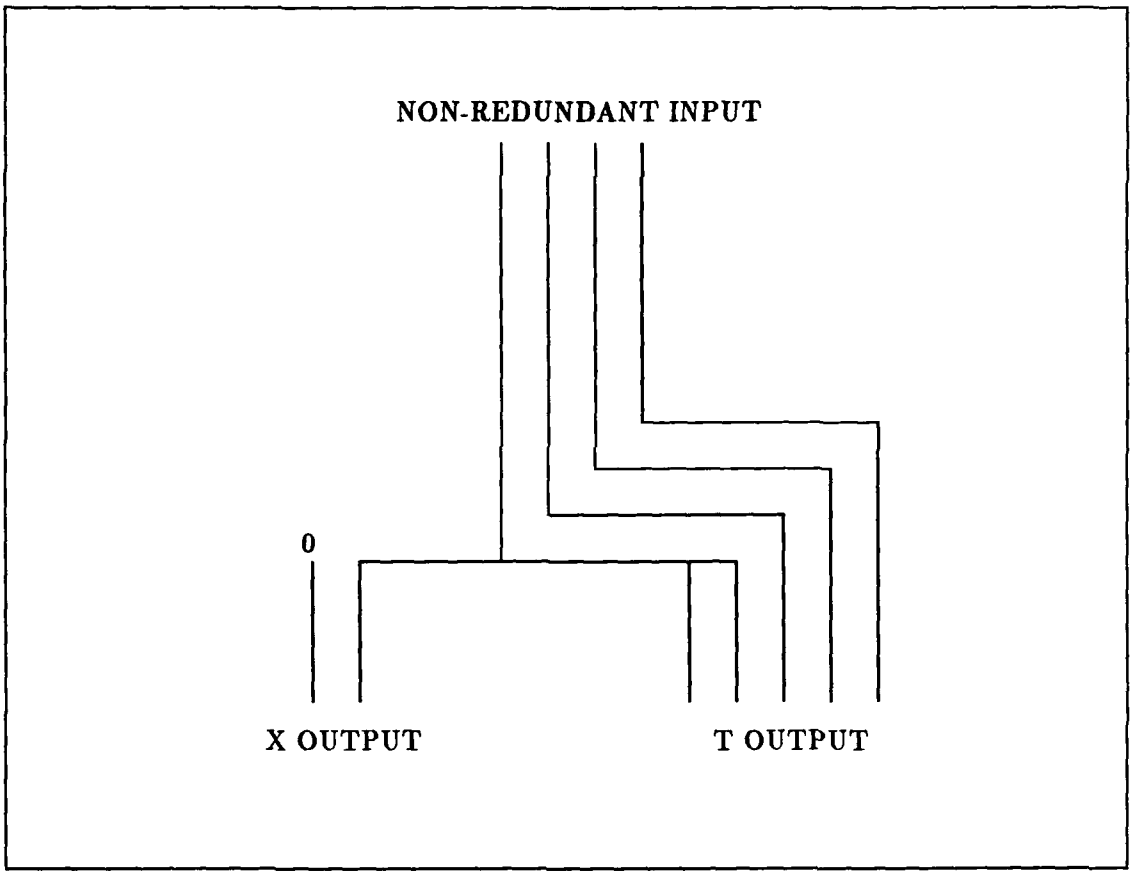

Figure 5.1. $S1_R$ Recoder Routing.

bit of the 4-bit input is extended to be the most significant bit of T. The most significant bit of the input is also used as the least significant bit of X. Since X is a 2-bit number and the input is expressed in a non-redundant form, X is only 0 or 1; therefore, the most significant bit is always 0. Figure 5.1 shows the routing of the $S1_R$ Recoder.

Since there is no logic required for the $S1_R$ Recoder, the is no appreciable propagation delay through it. However, there are important VLSI considerations which must be kept in mind. The loading on the most significant bit of the input is three times the loading of the other bits of the input. When designing the $S1_R$ Recoder for a specific application, the loading on the most significant bit must be compensated for by either using inverters

at the inputs and output ports or by ensuring the driver for the most significant bit is scaled large enough for the load. The use of inverters at the input and output ports give the advantage of isolating the input drivers from the load that the outputs of $S1_R$ sees. This allows for the independent design of the follow-on modules and scaling of the recoders output drivers for those follow-on modules. The cost is the addition of 11 inverters.

$S1_A$ Adder

The $S1_A$ adder accepts, as its inputs, two SD digits where each digit is an element of the digit set D_ρ. SD digits are represented in 2's complement by 5-bits. The outputs of the $S1_A$ Adder are X and T, where X and T are in the digit sets D_x and D_t respectively. The first requirement of the $S1_A$ Adder is to add the inputs, giving a result which is 6-bits wide. After the inputs are added, the result must be recoded into X and T.

The adder must be designed for inputs which are 5-bits and a carry-in bit. The carry-in bit is connected to the control logic and used in conjunction with an inverter to perform the subtraction operation. By designing the adder this way, it can perform addition and subtraction faster due to the short propagation delay through an inverter compared to a 2's complementor. The next step in the design of the adder is to select the type of adder to use in order to minimize its propagation delay. The adder which best suits the needs of minimum propagation delay is a carry-select adder. A carry-select adder is used to give rapid lateral carry propagation. Through the use of SPICE simulations, using 2μ technology, the estimated propagation delay through the worst case path of the adder is $4.9\,ns$.

Recoding of the adders 6-bit result is similar to the recoding in the $S1_R$ Recoder with the exception of the possibility of having a negative value for X. To perform the recoding, the four least significant bits of the adder results are routed to the four least significant bit of T. The most significant bit of T is a sign extension of its next most significant bit. X is determined by the two most significant bits of the adders result and the most significant bit of T. The two most significant bits of the adders result are added to the most significant bit of T to form X. This is done by using two half adders to compute X. SPICE simulations for this step estimates the worst case propagation time is $1.2\,ns$.

The complete $S1_A$ Adder is shown in Figure 5.2. An estimate of the overall propagation time of the $S1_A$ adder is 6.1 ns. This is the time required to obtain the most significant bit of X; however, the time required for T is only the adders time, 4.9 ns. A CIFPLOT of the $S1_A$ Adder is in Appendix B. A transistor count of the $S1_A$ Adder shows that 160 transistors are used.

S2 Adder

The $S2$ Adder is very similar to the $S1_A$ Adder with the exception of the recoding stage not required. The $S2$ Adder has two inputs, X and T, or T', which are in the digit sets D_x and D_t, or $D_{t'}$, respectively. Therefore, the maximum value of their sum is $T'_{max} + X_{max} = 9 + 1 = 10$. The addition is accomplished by using the same carry-select adder described in the preceding section for the $S1_A$ Adder. However, the adders hardware is reduced by recognizing that the CARRY-IN to the first adder is always 0. This reduces the hardware of the two least significant bit adders. Also, the hardware for the most significant bit adder is reduced since CARRY-OUT is not required. Figure 5.3 shows the requirements of the $S2$ Adder. SPICE simulation have shown that the worst case propagation delay is 4.9 ns. The CIFPLOT of the $S2$ Adder is in Appendix B. The S2 Adder requires 129 transistors.

M0 Multiplier

The $M0$ Multiplier is the most complex module for SD arithmetic. The multiplier has two inputs which are both elements of the digit set D_ρ. The results are two values, U and W which are in the digit sets D_u and D_w respectively. Multiplication is accomplished by converting one of the SD digits to a modified radix-4 representation.

$$A_i = 4K'_i + K_i$$

In this representation, K and K' are pseudo-numbers in that they represent numbers in the set $\{-2, -1, 0, 1, 2\}$ but they are not coded in a standard manner. The encoder forms K and K' from A by using the functions

$$K_0 \;=\; (A_1 \; xor \; A_4) \; and \; (\overline{A}_3 \; or \; A_4)$$

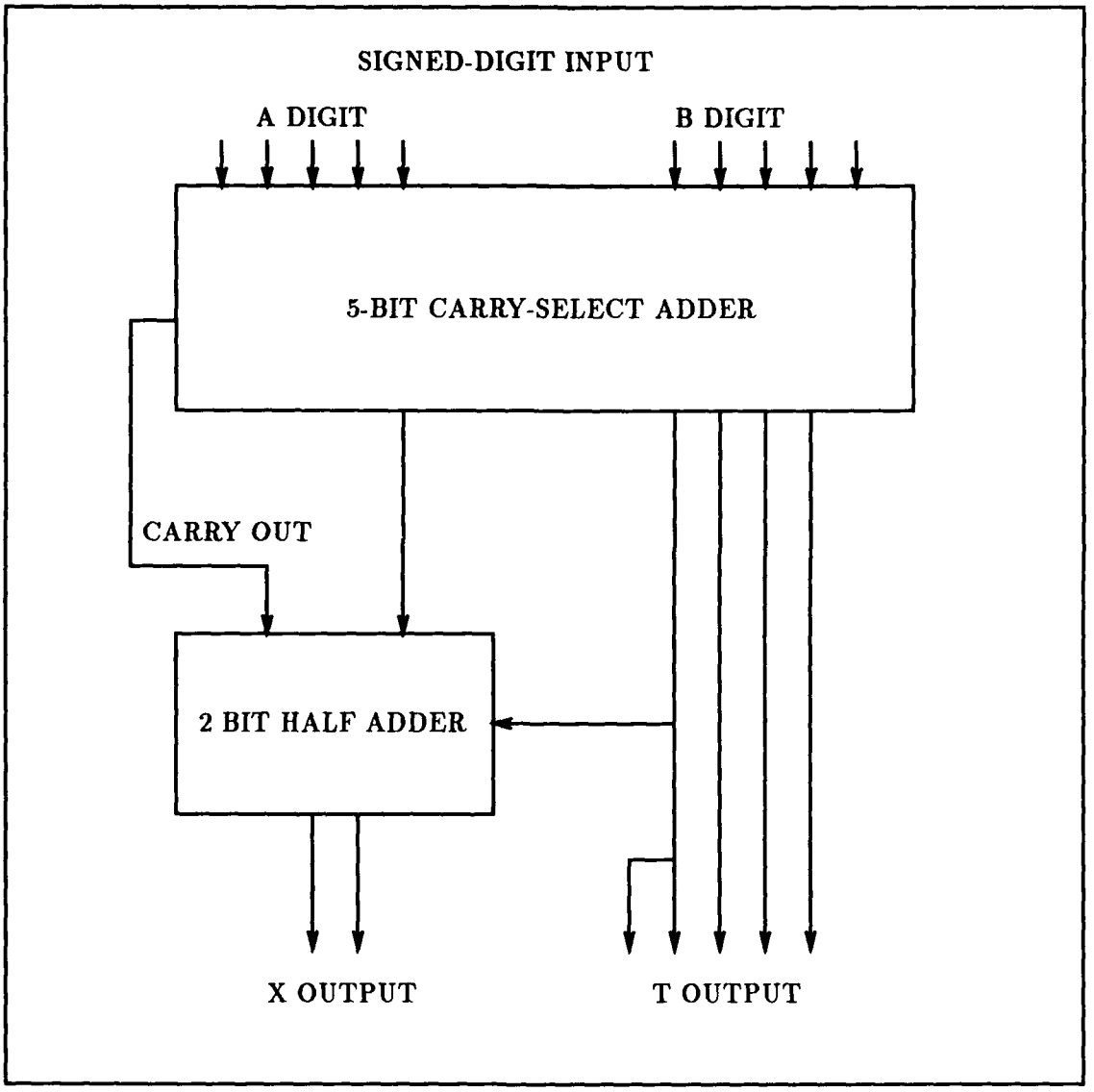

Figure 5.2. Complete $S1_A$ Adder.

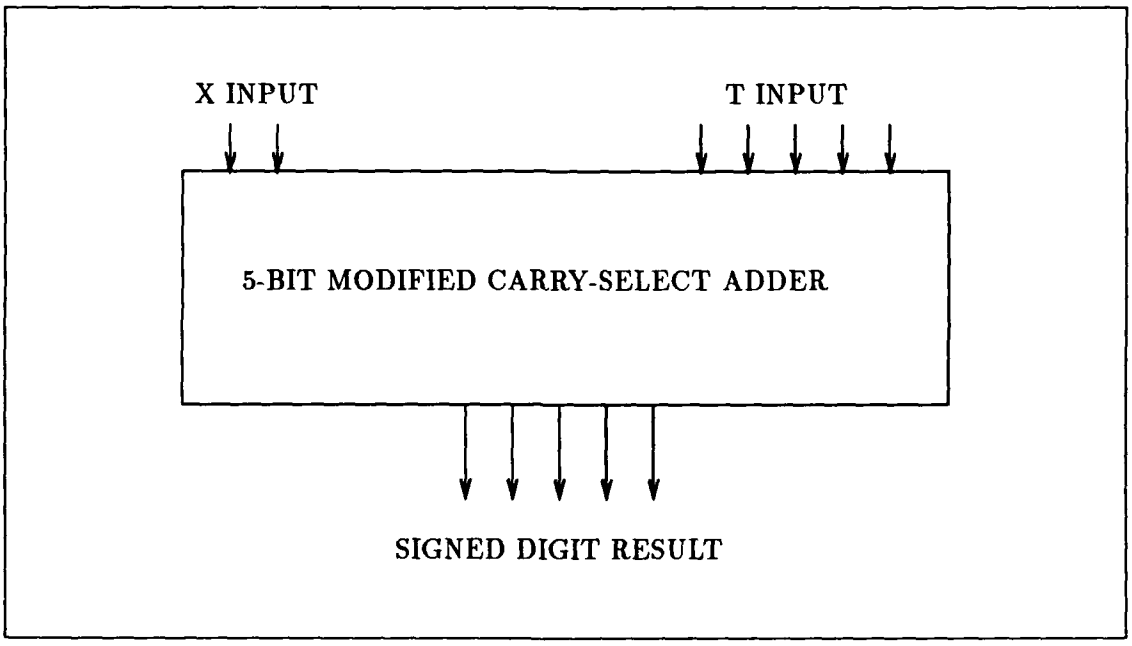

Figure 5.3. *S2* Adder Configuration.

$$
\begin{aligned}
K_1 &= \overline{A}_0 \\
K_2 &= (\overline{A_0 \, or \, A_1}) \\
K'_0 &= A_4 \\
K'_1 &= (A_1 \, xnor \, A2) \, or (A_3 \, and \, \overline{A}_4) \\
K'_2 &= \overline{(A_1 \, xor \, A_4) \, or \, (A_2 \, xor \, A_3)}
\end{aligned}
$$

K and K' are coded such that they can operate directly on a set of three multiplexers each where the first multiplexer selects the B digit or its 2's complement. The second multiplexer is used for selecting the output of the preceding multiplexer or shift that output left one bit. Finally, the third multiplexer select whether to pass the output of the second multiplexer or to pass all zeros. K and K' each operate on a set of these multiplexers. However, the K and K', as well as the outputs of the multiplexer sets, are a radix-4 apart. Figure 5.4 shows how the multiplexer sets are arranged and controlled. The least significant bit of K, and K', control the Complementor Multiplexer while the next least significant bit controls the Shifter Multiplexer. The most significant bit controls the Zero Multiplexer. The outputs of the two multiplexer sets form two partial products which are shifted two bit positions relative to each other.

The partial products are added by using a 6-bit carry-select adder, similar to the 5-bit version described previously. The two least significant bits of the multiplexer set controlled by K by-pass the adder since the multiplexer sets are radix-4 apart in their weighting. The final step is to recode the results of addition into the digit sets for U and W. W is coded the same way that T is coded in the $S1_A$ adder. The four least significant bits of the adder, where two of the four bits by-passed the adder, are routed to the four least significant bits of W. The most significant bit of W is the sign extension of its next most significant bit. U is coded the same way that X was coded except that U is 4-bits wide. Four half adders are used to recode the four most significant bits of the 6-bit adders result along with the most significant bit of W.

An overall diagram of the $M0$ multiplier is shown in Figure 5.5. The encoder for the generation of K and K' is shown as part of the multiplier. In reality, this encoder is used

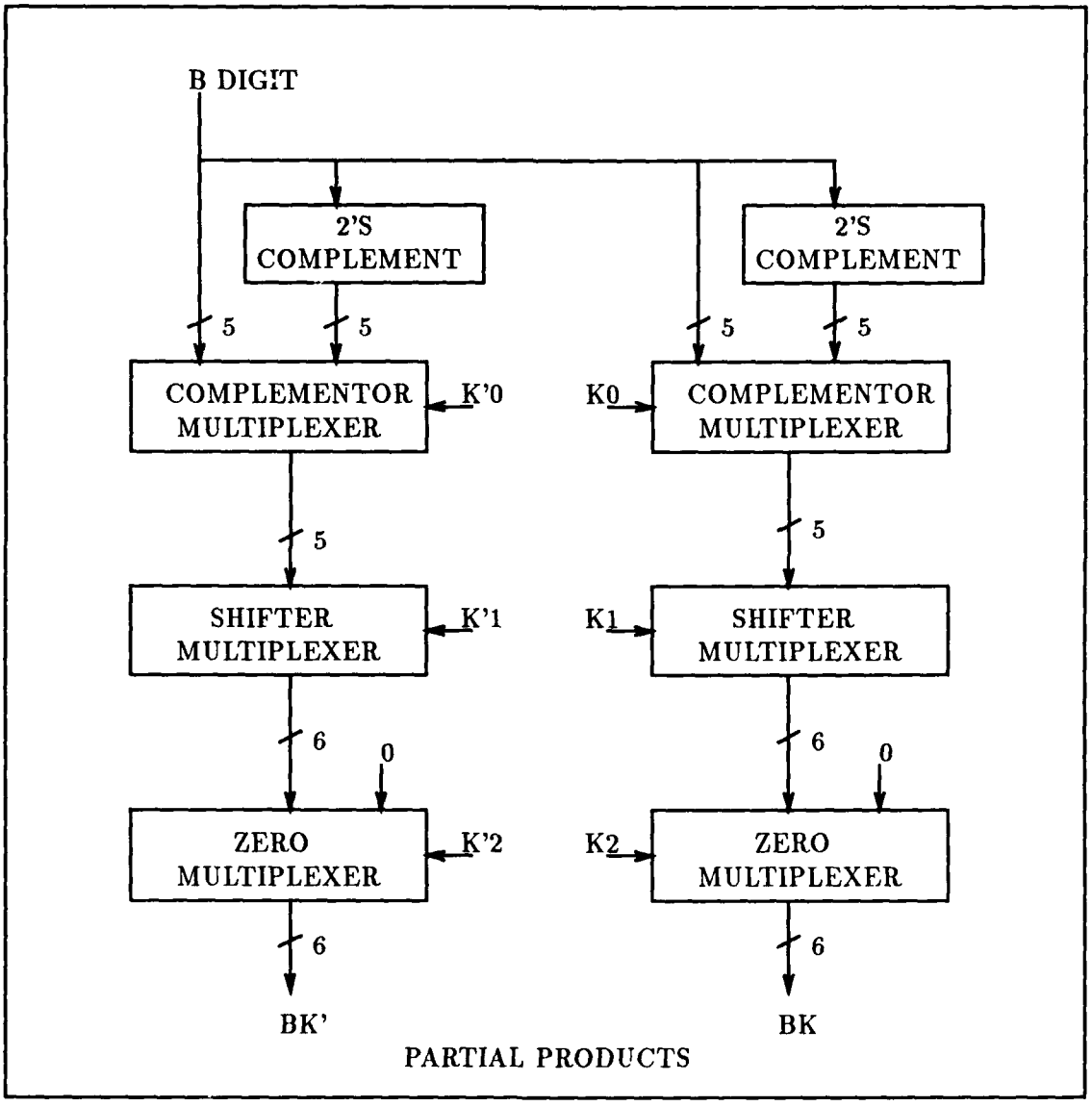

Figure 5.4. *M*0 Multipliers Multiplexer Arrangement.

as a separate block when a single digit is being multiplied to a complete SD number. In this case, the single digit is the input to the encoder and the resulting K and K' bits are used for each multiplexer set corresponding to each digit in the SD number. This reduces the required hardware.

The performance parameters obtained from SPICE analysis are worst case values. The time to encode a SD digit into K and K' is 2.5 *ns*. This time is done in parallel with the formation of the 2's complement of the multiplexer digit and, in part, with the multiplexer set. The total time to obtain partial product results from the multiplexer set, including the encoder time, is 4.3 *ns*. The addition of the partial products requires 5.7 *ns* and the recoding of its output requires 3.7 *ns*. However, a portion of the recoding stage overlaps the adder stage. Therefore, the partial product adder and the recoding of its output were estimated as requiring 9.0 *ns*. From the simulation results, the estimated time to multiply two SD digits is 13.3 *ns* for the formation of the U result and 10.0 *ns* for the W result. A CIFPLOT of the $M0$ Multiplier is in Appendix B. This plot shows $M0$ with the encoder as an internal structure. In this configuration, the M0 Multiplier requires 494 transistors, 113 of those are for the encoder.

A1 Assimilator

The $A1$ Assimilator is the most time consuming operation of all SD operations. This is due to the barrow propagation delays across the entire field. The assimilator accepts a SD digit, which is expressed in a redundant form, and outputs a result which is non-redundant. A barrow signal is used to propagate negative values from a digit which is weighted r^{-i} to the digit on the left which is weighted r^{1-i}. If the digit is positive, it value may be output directly. However, if the digit is negative, the value must be subtracted from 16 and its value output. The barrows are used to decrement the output of the stage on the left, a radix higher. The general configuration of the assimilator is shown in Figure 5.6. The assimilator recodes the SD digit into a non-redundant form, by stripping out the four least significant bits, and generates a barrow signal for the next stage. Once the digit is expressed in a non-redundant form, it is subtracted by the barrow from the stage on the right. The subtraction is accomplished by adding the barrow, with sign extension, to the

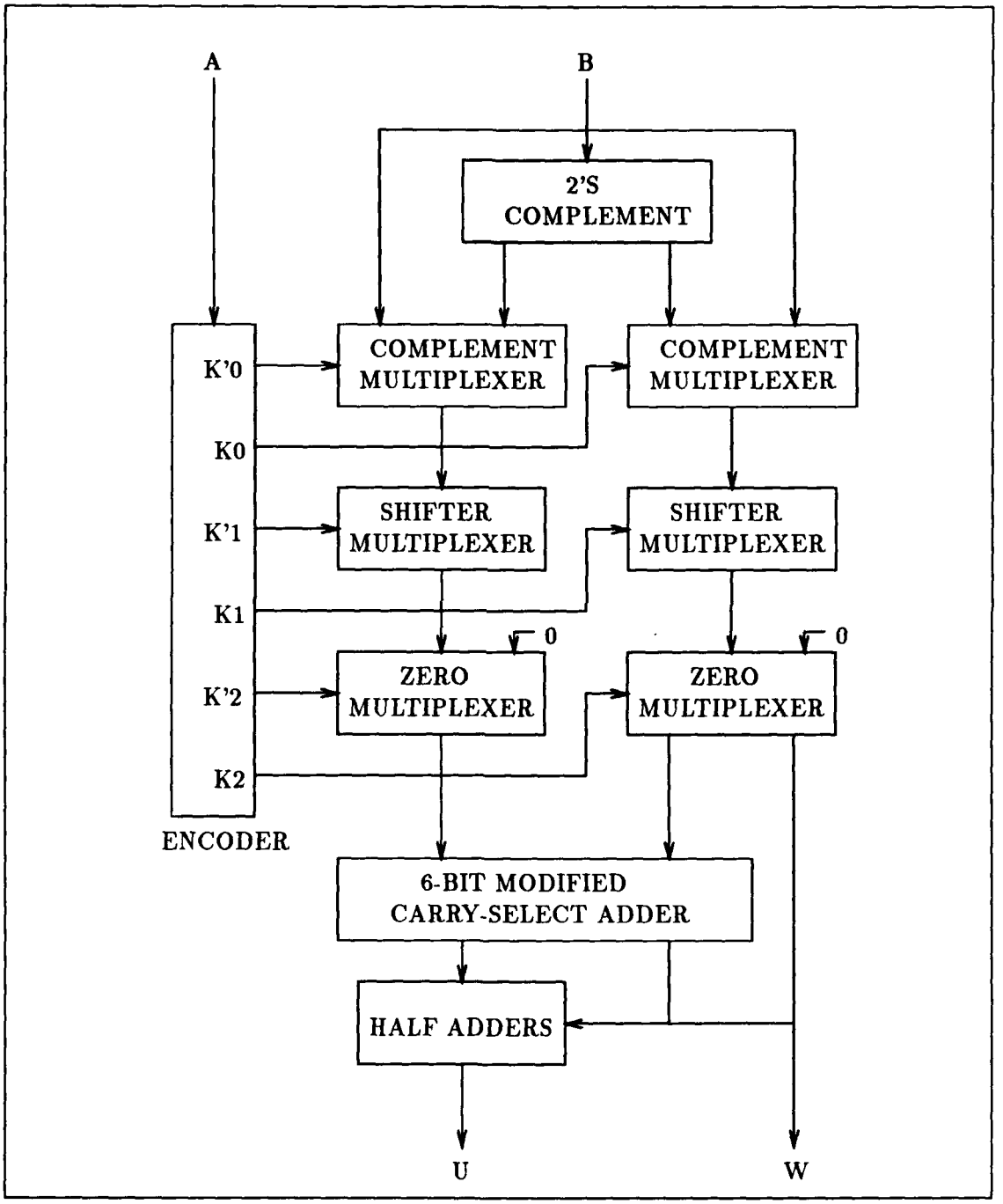

Figure 5.5. Complete $M0$ Multiplier Configuration.

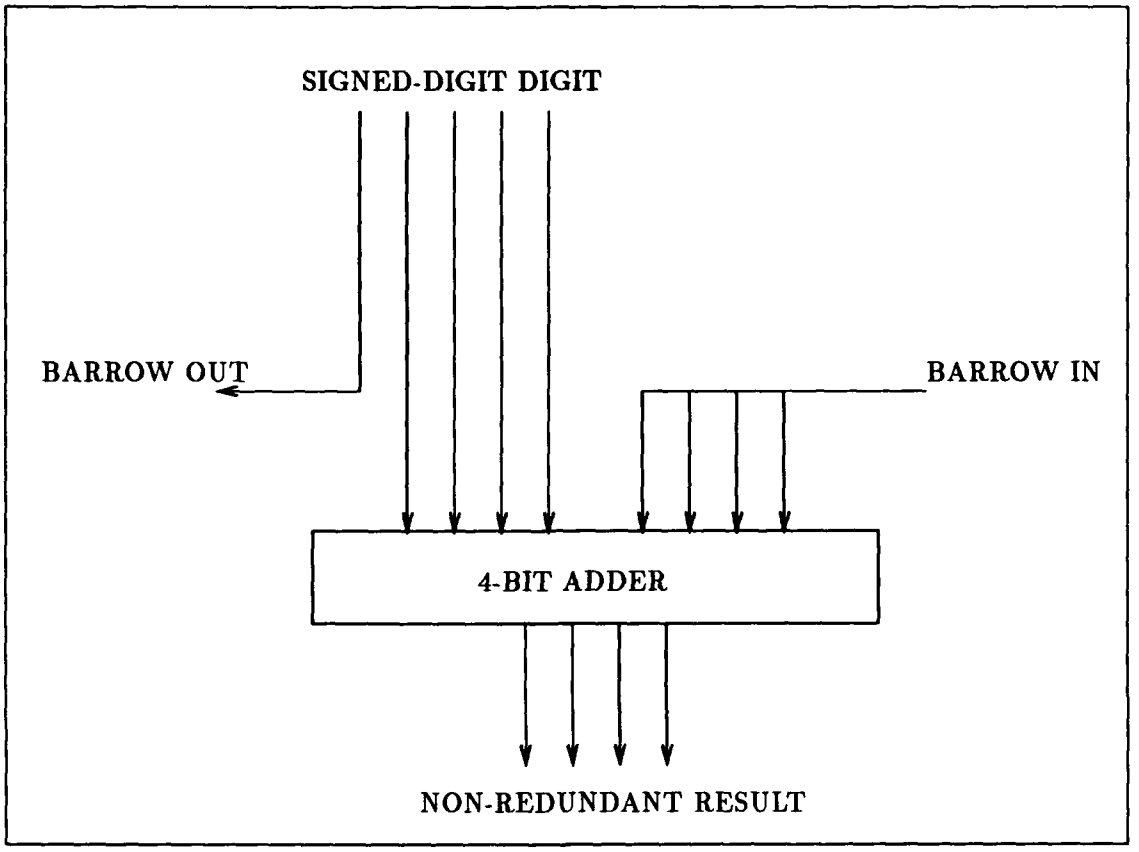

Figure 5.6. Assimilator for Signed-Digit Digit.

non-redundant result. The adder is configured as a 2-2 modified carry-select adder. The recoding of the digit is performed by simple routing and requires negligible time. The adder requires 4.5 *ns* to compute the final result.

VI. Signed-Digit Performance

In the preceding chapters, the SD operation units, and the modules with which the units are built, were described. Performance estimates for the modules were given in terms of propagation delays through each unit. By using these estimates of module performance, the SD modules can accurately be described in VHDL. Once the modules are described, SD units can be modeled and simulated.

Signed-Digit Module Descriptions

The $S1_R$ Recoder accepts a 4-bit input and provides the outputs T and X which are in the digit sets D_t and D_x respectively. The VHDL description of the entity interface is defined by these signals.

```
use work.SD_DEFINITIONS.all;
entity S1_RECODER is
     port ( DATA_IN : in bit_vector( 3 downto 0 );
            T_out   : out T_TYPE;
            X_out   : out X_TYPE );
end S1_RECODER;
```

The DATA_IN signal describes the 4-bit input which is a 4-bit slice of the total input mantissa. T_TYPE and X_TYPE are data types which describe subtypes of a bit_vector where T_TYPE is a bit_vector (4 downto 0) and X_TYPE is a bit_vector (1 downto 0). These subtypes are used to clarify the data types by giving them unique names corresponding to the digit sets which they represent. All of the data types are defined in the package SD_DEFINITIONS. The $S1_R$ Recoder is described behaviorally and only involves proper routing of the input signals to the correct output lines. No generic parameters are passed to the recoder since there is no requirement for altering the propagation delay, which is essentially 0 ns. The complete VHDL description of the $S1_R$ Recoder is given in Appendix C.

The $S1_A$ Adder is more involved than the recoder. It accepts two SD digits in the digit set D_ρ and outputs T and X in the digit sets D_t and D_x respectively. The $S1_A$ Adder also requires a control signal which indicates if it is performing and addition or a subtraction. The VHDL entity description defines these inputs.

```
use work.SD_DEFINITIONS.all;
entity S1_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( SD1_in           : in SD_DIGIT;
             SD2_in           : in SD_DIGIT;
             ADD_SUB          : in bit;
             X_out            : out X_TYPE;
             T_out            : out T_TYPE );
end S1_ADDER;
```

The data type SD_DIGIT is defined as a bit_vector (4 downto 0) in the package SD_DEFINITIONS. X_TYPE and T_TYPE are as defined previously while bit is a predefined type. The generic parameter TECHNOLOGY_SCALE is used to linearly alter the propagation delay through the adder. The default propagation delay, TECHNOLOGY_SCALE equal to 1.0, is determined through SPICE analysis using 2 micron technology. If a different technology is used, the propagation delay is changed by setting TECHNOLOGY_SCALE to linearly adjust for the new technology. The architectural description of the adder is a behavioral description. This description converts the SD digits to integer values, adds the integers, and converts the result into and X vector and a T value. The T value is then converted to a T vector. Two functions are used in this behavioral description, BIN_TO_INT and INT_TO_SD. These functions are defined in the package SD_DEFINITIONS and called when required. The complete VHDL description is given in Appendix C.

The S2 Adder accepts an X vector and a T vector, which are defined by the data types X_TYPE and T_TYPE respectively. The output is a SD digit defined by the data

type SD_DIGIT. The S2 Adder does not require any control signals. The entity description defines these inputs and the output.

```
use work.SD_DEFINITIONS.all
entity S2_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( X_in             : in X_TYPE;
             T_in             : in T_TYPE;
             SD_out           : out SD_DIGIT );
end S2_ADDER;
```

The architectural description of the S2 Adder is a behavioral description. T_in is converted to an integer and incremented, decremented, or un-altered depending on the bit fields of X_in. The result is then converted to a bit vector, SD_out, defined by the data type SD_DIGIT. TECHNOLOGY_SCALE is used as discussed previously. The complete VHDL description for the S2 Adder is given in Appendix C.

The M0 Multiplier multiplies two SD digits and outputs, as its result, U and W which are in the digit set D_u and D_w respectively. There are no control signals required for the multiplier. The inputs and outputs are defined in the entity description.

```
use work.SD_DEFINITIONS.all;
entity M0_MULT is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( SD_1             : in SD_DIGIT;
             SD_2             : in SD_DIGIT;
             U_out            : out U_TYPE;
             W_out            : out W_TYPE );
end M0_MULT;
```

The data types U_TYPE and W_TYPE are bit vectors which are defined in the package SD_DEFINITIONS. The architectural description of the multiplier is behavioral.

The two SD digits are converted to integers and multiplied. The result is then converted to a U vector and a W value, where the W value is then converted to a W vector through the function call INT_TO_SD. The complete VHDL description of the M0 Multiplier is given in Appendix C.

Once the VHDL descriptions of the SD modules were completed, each module was tested. The tests were designed to validated the correctness of each module before instantiating them in larger models. S1_RECODER_TB, S1_ADDER_TB, S2_ADDER_TB, and M0_MULT_TB test benches were written, analyzed, simulated, and reports generated to verify correctness. These test benches and their report generators are given in Appendix C. Simulation results are also presented in Appendix C.

Complete SD Multiplier

A SD number which corresponds to the precision of IEEE double precision requires the number to consist of 16 digits, 0 to 15. This provides a precision of $16^{-15} = 2^{-60}$. Therefore, to multiply two SD numbers, 16 multiplier blocks with 16 digit multipliers in each block are required. This will result in 16 partial products. The partial products are added in a tree structure with four levels until a single result is obtained. To build a VHDL model of the multiplier, several sub-components were built. A multiplier block which multiplies a single digit to a SD number was built. This block consists of 16 M0 Multipliers, 15 $S1_A$ Adders, and 15 S2 Adders. Since the $S1_A$ Adders are only used for addition in a multiplier, the ADD_SUB control signals are set to ADD. The result out of the block is a partial product which is 17 digits long. The entity description of the multiplier block defines the inputs.

```
use work.SD_DEFINITIONS.all;
entity MULT_BLOCK is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( DIGIT_C          : in SD_DIGIT;
             SD_NUMB          : in SD_NUMBER;
             RESULT           : out PARTIAL_P ( 0 to 16 ) );
```

```
end MULT_BLOCK;
```

The data type SD_NUMBER is defined in SD_DEFINITIONS as an array (0 to 15) of SD_DIGIT while PARTIAL_P is defined as an unbounded array of type SD_DIGIT. The distinction between the two is made to identify a SD number as a distinct type apart from any partial product types. The generic parameter is not directly used in the architecture but is passed down to the lower modules. A structural description of the multiplier block instantiates all of the modules required individually. The complete VHDL description is given in Appendix C.

The next sub-component written is ADDER_1. ADDER_1 is an adder composed of a single S1_ADDER and an S2_ADDER. This component was written to reduce the number of component instantiation statements in the partial adder sub-components. ADDER_1 requires two SD_DIGIT inputs, a T_in input, and outputs a SD_DIGIT and a T_out. The entity description defines its required signals.

```
use work.SD_DEFINITIONS.all;
entity ADDER_1 is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( SD1              : in SD_DIGIT;
             SD2              : in SD_DIGIT;
             T_in             : in T_TYPE;
             T_out            : out T_TYPE;
             SUMr             : out SD_DIGIT );
end ADDER_1;
```

The architectural description is structural and instantiates one S1_ADDER and one S2_ADDER. An X vector is declared within the architecture and provides the path between the adders for this signal. TECHNOLOGY_SCALE is passed down to the adder modules. A complete VHDL description is given in Appendix C.

Four levels of partial product adders were modeled, SL2_ADDER, SL3_ADDER, SL4_ADDER, and SL5_ADDER. Each of these adders requires the same number of

ADDER_1 components, 16, but there interface signals are different. SL2_ADDER accepts partial products from the MULT_BLOCK and sums them. The result is a partial product which is 18 digits long, 0 to 17. The SL3_ADDER then adds two of these results and outputs a partial product 20 digits long, 0 to 19. SL4_ADDER adds two of these results and outputs a partial product 24 digits long, 0 to 23. Finally, SL5_ADDER adds the two partial products from SL4_ADDER and outputs the final partial product which is 32 digits long, 0 to 31. The entity descriptions for the partial product adders define there signals.

```
use work.SD_DEFINITIONS.all;
entity SL2_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( PARTIAL_H        : in PARTIAL_P ( 0 to 16 );
             PARTIAL_L        : in PARTIAL_P ( 0 to 16 );
             P_out            : out PARTIAL_P ( 0 to 17 ) );
end SL2_ADDER;

use work.SD_DEFINITIONS.all;
entity SL3_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( PARTIAL_H        : in PARTIAL_P ( 0 to 17 );
             PARTIAL_L        : in PARTIAL_P ( 0 to 17 );
             P_out            : out PARTIAL_P ( 0 to 19 ) );
end SL3_ADDER;

use work.SD_DEFINITIONS.all;
entity SL4_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( PARTIAL_H        : in PARTIAL_P ( 0 to 19 );
             PARTIAL_L        : in PARTIAL_P ( 0 to 19 );
             P_out            : out PARTIAL_P ( 0 to 23 ) );
```

```
end SL4_ADDER;

use work.SD_DEFINITIONS.all;
entity SL5_ADDER is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( PARTIAL_H        : in PARTIAL_P ( 0 to 23 );
             PARTIAL_L        : in PARTIAL_P ( 0 to 23 );
             P_out            : out PARTIAL_P ( 0 to 31 ) );
end SL5_ADDER;
```

Complete VHDL descriptions for the partial product adders are given in Appendix C.

From these components, a SD multiplier which multiplies the mantissas of two SD numbers, corresponding to a precision greater than IEEE double precision, can be built. The mantissa multiplier, SD_MULT, accepts two SD numbers of type SD_NUMBER, and outputs a result which is of type PARTIAL_P with a range 0 to 31. The entity description of SD_MULT defines the multipliers signals.

```
use work.SD_DEFINITIONS.all;
entity SD_MULT is
   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( SD_A             : in SD_NUMBER;
             SD_B             : in SD_NUMBER;
             SD_out           : out PARTIAL_P ( 0 to 31 ) );
end SD_MULT;
```

The result, SD_out, is shifted to the right one digit due to the multiply algorithm discussed in Chapter 4. The architectural description of the multiplier is structural and instantiates the components MULT_BLOCK, SL2_ADDER, SL3_ADDER, SL4_ADDER, and SL5_ADDER. MULT_BLOCK is instantiated 16 times while SL2_ADDER is instantiated 8 times. SL3_ADDER is instantiated 4 times; and, SL4_ADDER is instantiated 2 times. SL5_ADDER is instantiated only once. The generic parameter is passed down

through each instantiation. The complete VHDL description of SD_MULT is given in Appendix C.

Testing of the Signed-Digit Multiplier

Testing of the multiplier consists of writing a test bench which instantiates the multiplier and mapping test vectors to its inputs. Then, the result is analyzed after the report is generated. The instantiation of the multiplier is a single instantiation of SD_MULT. However, to generate a set of test vectors becomes complex. This is due to the requirements of the digit set of a SD digit. To work around this problem, a test bench package was developed, TB_PACKAGE. Within the package, two functions are used to easy the generation of test vectors and result analysis. The function SD_MAKE is passed a real number and returns a normalized SD number while the function SD_TO_REAL is passed a SD number and returns its real number equivalent. Care must be used when calling these functions. When SD_MAKE is called and passed a number which is not in the range of a normalized SD number, the result returned will not have the same value as that passed but will be some factor of 16 of the argument. The function SD_TO_REAL assumes that the most significant digit is weighted with a 1. When being passed the 16 most significant digits of the multipliers result, this is not true. Therefore, the value returned is a factor of 16 smaller than the actual result. However, by passing the function P_out(1 to 16), the value returned is correct. The test bench SD_MULT_TB is given in Appendix C.

Once the test bench was analyzed, model generated, and built, the model was simulated. Various reports were generated from the simulation. The correctness of the test bench package functions were analyzed first. Once the correctness of the functions verified, the propagation delay of the multiplier was analyzed. These propagation delays assume that the inputs have already been converted to SD form and that the mantissa section of the multiplier requires more time than the addition of the exponents, a reasonable assumption. When using the default TECHNOLOGY_SCALE, indicating 2 micron technology, the worst case propagation delay is 65 ns. If the technology is changed to 1.25 micron, the TECHNOLOGY_SCALE factor is change to roughly approximate the speed-up associated with the change in technology. Linear scaling gives the approximate speed-up of

2, implying that TECHNOLOGY_SCALE equals 0.5. Using this scaling factor, the worst case propagation delay is 32 ns. The report generators and the reports are given in Appendix C. One note regarding the report generated is that the VHDL report generator has a problem reporting negative real numbers. This is a problem with the VHDL report generator, Intermetrics Version 1.5 running on the Suns.

VII. Conclusions and Recommendations

Conclusions

The original motivation behind the study into developing a processor to compute transcendental functions was driven by the requirements of solving the Vector Wave Equation. Mickey Bailey, [1], expanded the set of transcendental functions to encompass a greater number of functions than required. These functions all were derived from Chebyshev Polynomials. With the development of the division algorithm, together with the expanded trigonometric, exponential, and natural logarithm functions to give IEEE double precision accuracy, an extensive Transcendental Function Processor can be developed.

Chapter 2 and Chapter 3 developed the approximation algorithms and the rational for their use. The fewest number of terms to achieve an error below a specified value was used as the determining factor in the selection of the best approximation method. This section of the thesis covered important information which did not appear in the original effort. The structure of the approximations algorithms are based on Horners' method of restructuring a polynomial such that its computational form is suitable for a pipelined processor. The pre-processing, pipeline processing, and post-processing requirements of a unified processor were discussed. However, the structure of a unified Transcendental Function Processor did not evolve. The reasons for this are that the pre-processor requires different operations performed on the arguments of different functions. Therefore, the pre-processor can be optimized by knowing the mix of the functions requested. The more complex the mix of the requests, the more complex the control section of the pre-processor must be. Post-processing has the same complexity problem; if an complex control section for the post-processor is designed, the through-put of the processor can remain high. However, if the control section is simple, the processor will have to have dummy stages inserted into the post-processing stages to synchronize data for return to memory or further processing. The pipeline processing section is the best developed section. The pipeline consists of a data pipeline, an argument pipeline, and a control pipeline. This permits rapid reconfiguration of the pipeline to compute the approximation functions in any order, without delays in the arguments into the pipeline.

Chapter 4 presented an overview on alternate forms of data representation for use in the processor. The most interesting and advanced form is Signed-Digit representation. SD representation offers a number of advantages when compared to standard binary representation. The greatest advantage is the reduction of carry-barrow propagation delays. This increases the computation speeds possible from adders and multipliers. However, the advantages of SD representation do have a cost associated with its use; this is the penalty of converting IEEE double precision numbers to, and from, SD form. The penalty of the conversion operation to SD form is minor due to its limited carry propagation. The assimilation penalty is more sever since there exist the possibility of having a barrow propagate across the entire mantissa. However, in a pipelined processor environment, these conversions need only occur once.

Chapter 5 expands of the hardware required for numbers represented in SD form. The basic module were presented as well as their performance estimates obtained from SPICE models with LAMBDA equal to 1.0 microns. The S1_RECODER does not have any propagation delay since it consists of only routing of the input bits to their proper output. The S1_ADDERs T output has a propagation delay of 4.9 ns while the X output requires 6.1 ns. The S2_ADDER requires 4.9 ns to propagate the input to the output. The M0_MULTs propagation delay is 10 ns for the W output and 13.3 ns for the U output. Each of the modules were built in VLSI and presented in Appendix B.

In Chapter 6, the basic modules were describe in VHDL and each simulated to ensure their function and propagation times agree with the times obtained from the SPICE simulation. Then, a 16 digit by 16 digit multiplier was constructed and simulated. Simulation estimates the worst case propagation delay of the SD mantissa multiplier as 65 ns when using 2.0 micron technology, excluding conversion and assimilation time. This propagation time drops to 32 ns when the technology is changed to 1.25 micron. The additional time required for only the conversion of the mantissa is the propagation time of the S2 Adder, 4.9 ns. Assimilation of the mantissa is dependent of the construction of the Assimilation Unit. The simulation results, as well as the VHDL descriptions of the hardware, were presented and shown in Appendix C. The speed of the SD hardware is comparable to a step in technology size when compared to standard methods of computation.

Recommendations

The Transcendental Function Processor requires further investigations into the trade-offs between control complexity and throughput for its pre and post processors. This will rely heavily of the type and frequency of functions to be computed. However, the dedication of hardware of any form to the processor is still premature. Further work is required into the realizable advantages of SD representation. A tiny chip was constructed a part of this thesis effort and is shown in Appendix B. This chip needs to be fabricated and tested with results compared to those expected from a VHDL model. If the results show that SD representation does provide an appreciable speed-up then, a full SD multiplier should be built and tested. Though this thesis did not consider the size requirement of SD hardware, this must be studied when considering its use in the Transcendental Function Processor.

Appendix A. *Determination of Chebyshev Constants*

The evaluation of the integral

$$a_n = \frac{2}{\pi}\int_0^{\pi} f(\cos x)\cos nx dx$$

is not simple for most functions, $f(x)$. However, the accuracy of the summed Chebyshev Polynomials is dependent on the accuracy of these constants. To obtain a resultant accuracy of double precision, the precision of these constants is required to be greater than double precision. Therefore, for those function in which the integral can be evaluated, the accuracy of the result can easily be reached. For functions where the integral can not be evaluated directly, the result must be approximated by using an integral approximation method such as Simpson's Rule. Using these types of approximation methods requires a great deal of care. The limiting factor in making these approximations is the precision of the computer used. If the computer only has the ability to compute up to double precision accuracy, then, the resultant error will be somewhat greater depending on the distribution of the truncation errors in the computation. For all of the coefficients used in the Transcendental Function Processor, the error term of the coefficients is required to be less than 2^{-60}. This is due to the internal precision ability of the processor when numbers are represented in Signed-Digit form.

Additional problems appear when trying to approximate to the required accuracy of the coefficients. The shape of the graph of the integrand must be considered. If the integrand has the shape of a negative parabola, then the approximation must begin with the outer edges where the magnitude is the smallest and sum towards the center. The opposite is true if the shape is similar to a positive parabola. Virtually all of the transcendental functions of interest exhibit one of these shapes. The important point to remember is the smallest magnitude of the curve must be summed first. Also, when trying to approximate using a method such as Simpsons Rule, to obtain the required precision, the number of intervals required to be summed is quite large. However, if the programs are written carefully and the library routines validated for accuracy, a method which computes the area under the curves by summing intervals can be used.

As stated previously, there are ways to solve for the integration. One such method involves Residue Analysis. As an example of how this analysis works, the coefficients for $f(x) = \sin(\pi x/2)$ will be solved. Therefore, the equation for the coefficients is

$$a_n = \frac{2}{\pi}\int_0^{\pi} \sin\left(\frac{\pi\cos x}{2}\right)\cos nx\, dx$$

The limits of integration are changed by recognizing that the integrand is an odd function. The result is a circular interval of integration.

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} \sin\left(\frac{\pi\cos x}{2}\right)\cos nx\, dx$$

The first step in Residue Analysis is to generate a series in the complex plane to represent the integrand. Euler's Equation is used to do this conversion.

$$\cos x = \frac{e^{ix} + e^{-ix}}{2}$$

and

$$\cos nx = \frac{e^{inx} + e^{-inx}}{2} \ .$$

If

$$e^{ix} = Z$$

then

$$ie^{ix}dx = dZ \ .$$

Rearranging

$$dx = \frac{dZ}{iZ} \ .$$

Therefore, the integral is

$$a_n = \frac{1}{\pi}\oint_C \sin\left(\frac{\pi}{4}(Z + Z^{-1})\right)\left(\frac{Z^n + Z^{-n}}{2}\right)\frac{dZ}{iZ}$$

$$a_n = \frac{1}{2i\pi}\oint_C \sin\left(\frac{\pi}{4}(Z + Z^{-1})\right)(Z^{n-1} + Z^{-n-1})dZ$$

where C is the unit circle centered at the origin transversed in the counter clockwise direction. To perform simple Residue Analysis, there should only be one unique pole in the unit radius around zero, which is the case here. Therefore,

$$a_n = Res_{Z=0}\left(\sin\left(\frac{\pi}{4}(Z + Z^{-1})\right)(Z^{n-1} + Z^{-n-1})\right)$$

To derive a series from the above equation, the trigonometric series for Sine is used.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Solving in steps,

$$\sin\left(\frac{\pi}{4}(Z+Z^{-1})\right) = \sum_{k=0}^{\infty} (-1)^k \frac{(\frac{\pi}{4}(Z+Z^{-1}))^{2k+1}}{(2k+1)!}$$

$$\sin\left(\frac{\pi}{4}(Z+Z^{-1})\right) = \sum_{k=0}^{\infty} \frac{(-1)^k (\frac{\pi}{4})^{2k+1} (Z+Z^{-1})^{2k+1}}{(2k+1)!} \quad .$$

And,

$$(Z+Z^{-1})^{2k+1} = Z^{2k+1} + (2k+1)Z^{2k}Z^{-1} + (2k)(2k+1)Z^{2k-1}Z^{-2} + \cdots$$

or

$$(Z+Z^{-1})^{2k+1} = \sum_{j=0}^{2k+1} \left(\frac{(2k+1)!}{(j)!(2k+1-j)!}\right) Z^{2j-2k-1}$$

Therefore,

$$\sin\left(\frac{\pi}{4}(Z+Z^{-1})\right) = \sum_{k=0}^{\infty} \frac{(-1)^k (\frac{\pi}{4})^{2k+1}}{(2k+1)!} \sum_{j=0}^{2k+1} \left(\frac{(2k+1)!}{(j)!(2k+1-j)!}\right) Z^{2j-2k-1} \quad .$$

The coefficients equal

$$a_n = \sum_{k=0}^{\infty} \frac{(-1)^k (\frac{\pi}{4})^{2k+1}}{(2k+1)!} \left(\sum_{j=0}^{2k+1} \left(\frac{(2k+1)!}{(j)!(2k+1-j)!}\right) (Z^{2j-2k+n-2} + Z^{2j-2k-n-2}) \right)$$

In Residue Analysis, when looking for the first integration of a series whose pole is at Z = 0, the integration value is obtained from the coefficient of Z^{-1}. Therefore, from the equation above, the value of j which will give a power of -1 to Z must be solved.

$$2j - 2k + n - 2 = -1$$

and

$$2j - 2k - n - 2 = -1 \quad .$$

Therefore, from the first equation,

$$j = k - \frac{n-1}{2}$$

and from the second equation,

$$j = k + \frac{n+1}{2} \ .$$

Using these values for j and solving,

$$a_n = 2 \sum_{k=(n-1)/2}^{\infty} (-1)^k \left(\frac{\pi}{4}\right)^{2k+1} \left(\frac{1}{(k-\frac{n-1}{2})!(k+\frac{n+1}{2})!}\right)$$

This infinite series is evaluated by summing to a finite number. Since the denominator of the series is a factorial, the number of terms required to be summed to obtain the needed precision is small, on the order of 30 terms. To maintain precision, the summation must occur in reverse order; that is, the sum should be computed from $k = 30$ down to 0 when computing a_1.

Appendix B. *Signed-Digit CIFPLOTS*

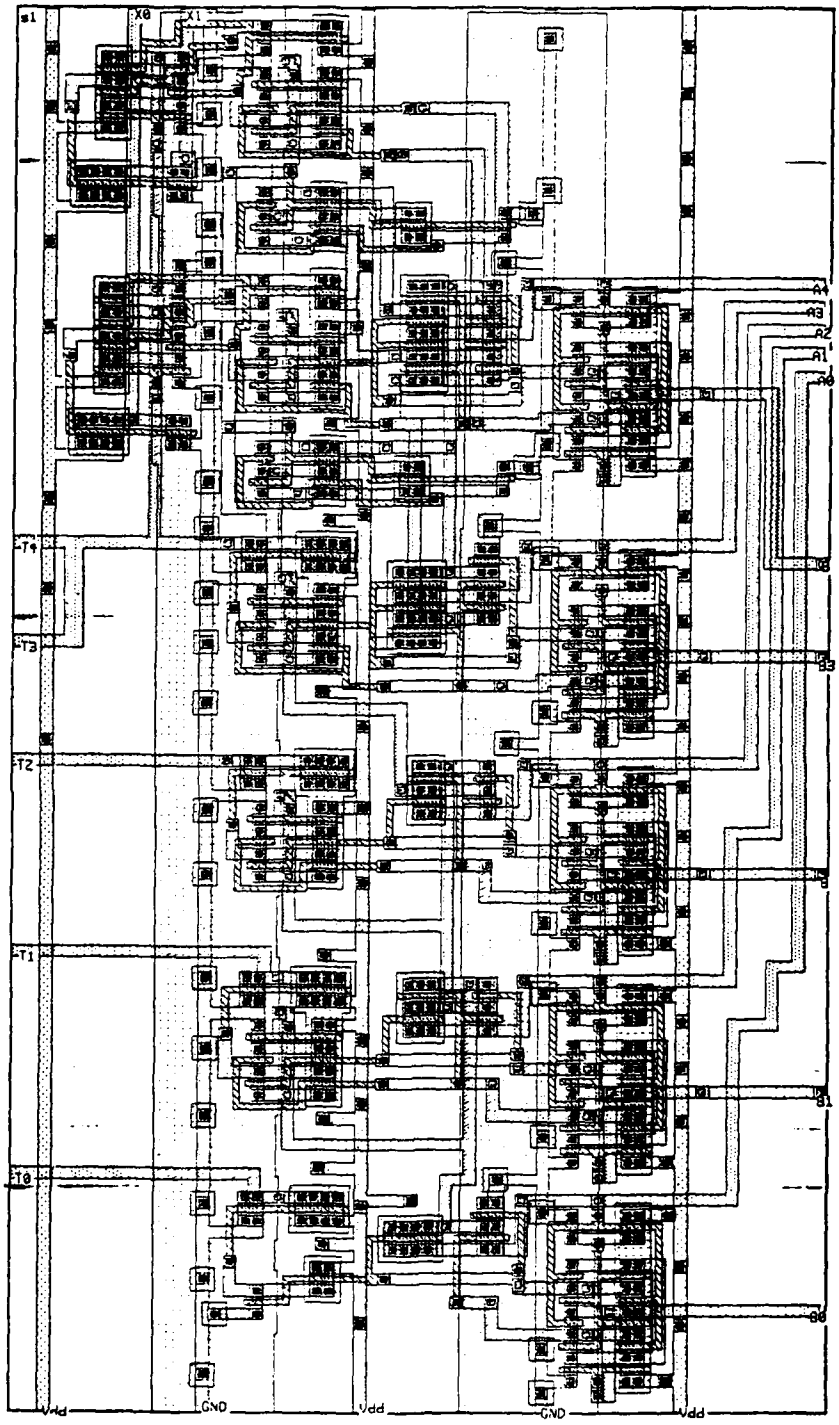

Figure B.1. CIFPLOT of $S1_A$ Adder.

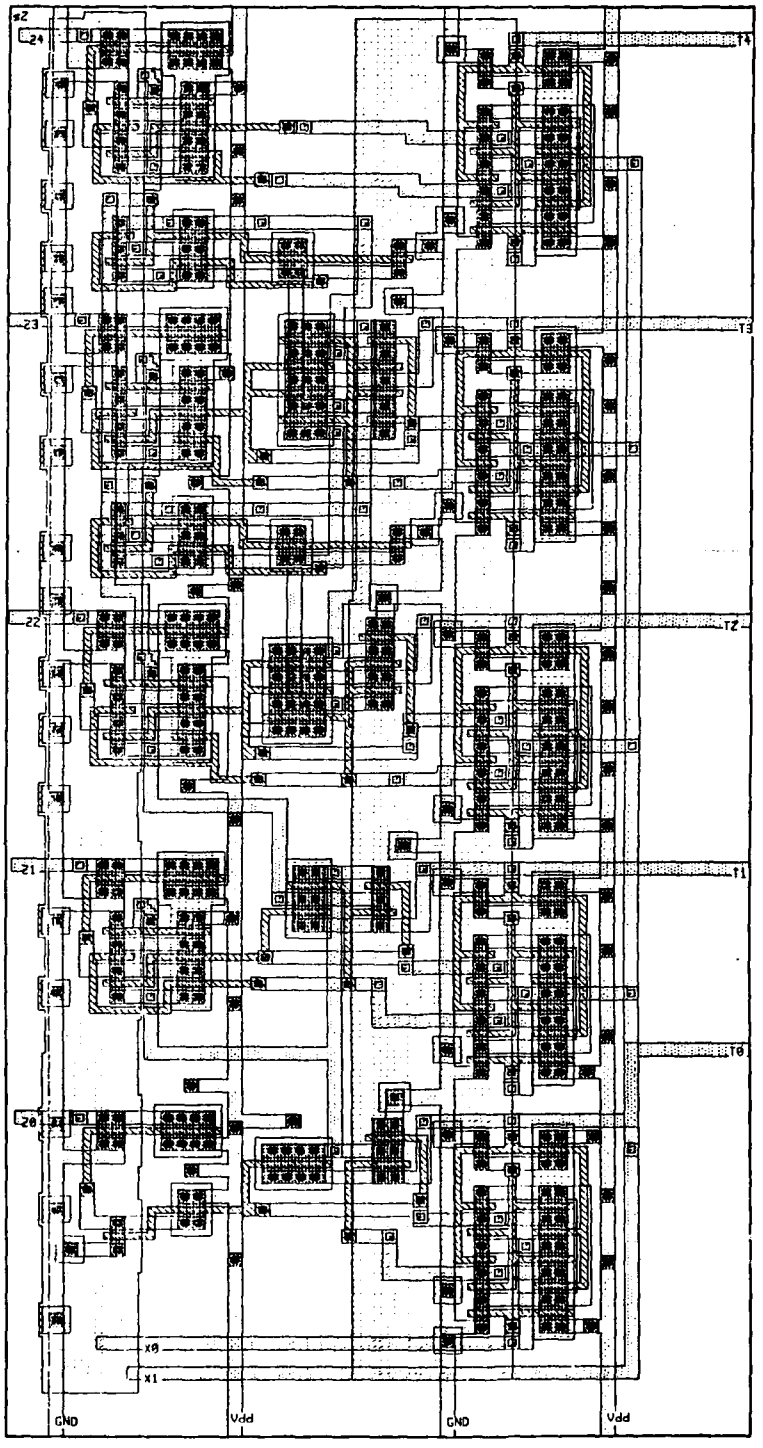

Figure B.2. CIFPLOT of S2 Adder.

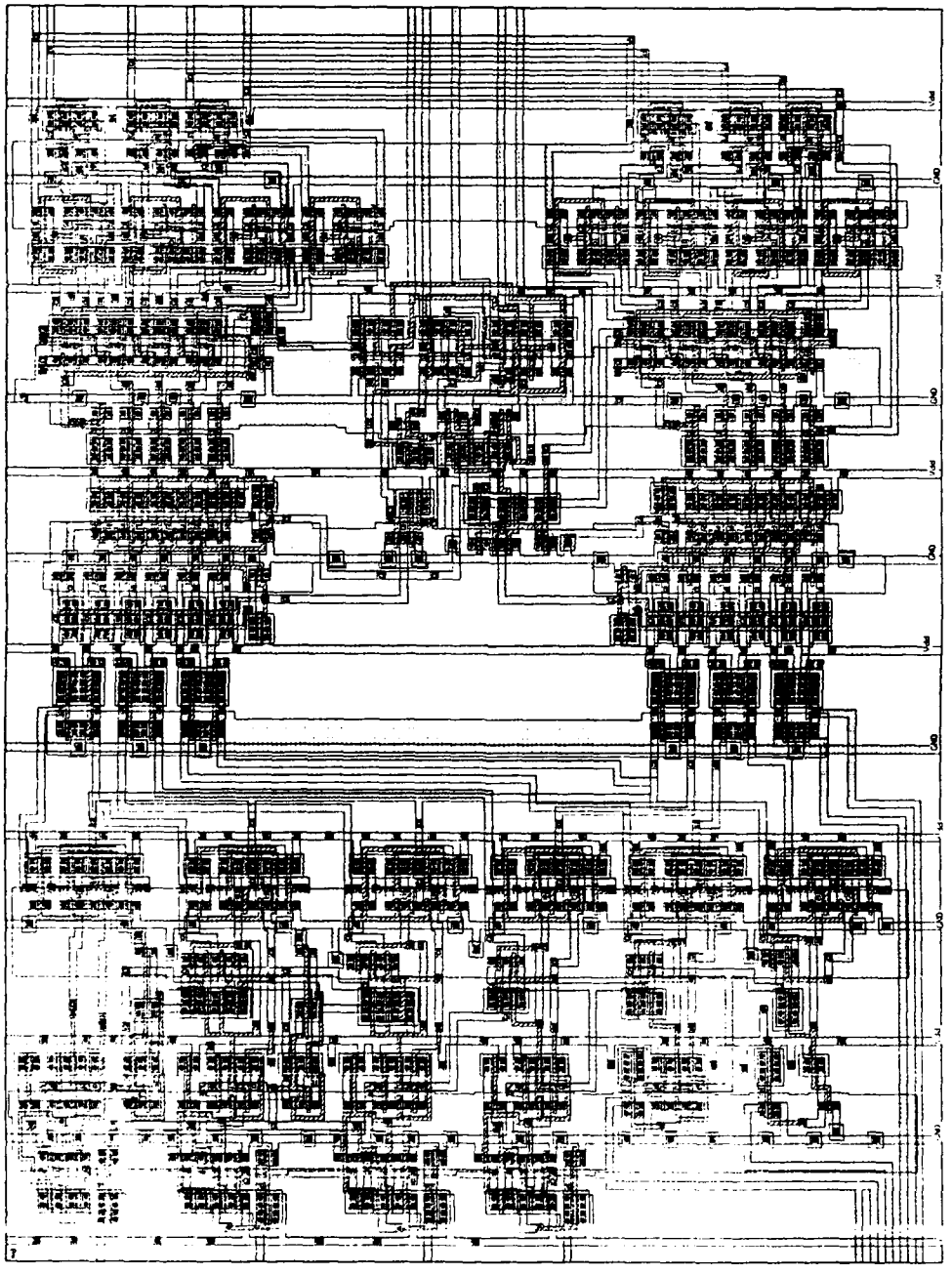

Figure B.3. CIFPLOT of M0 Multiplier.

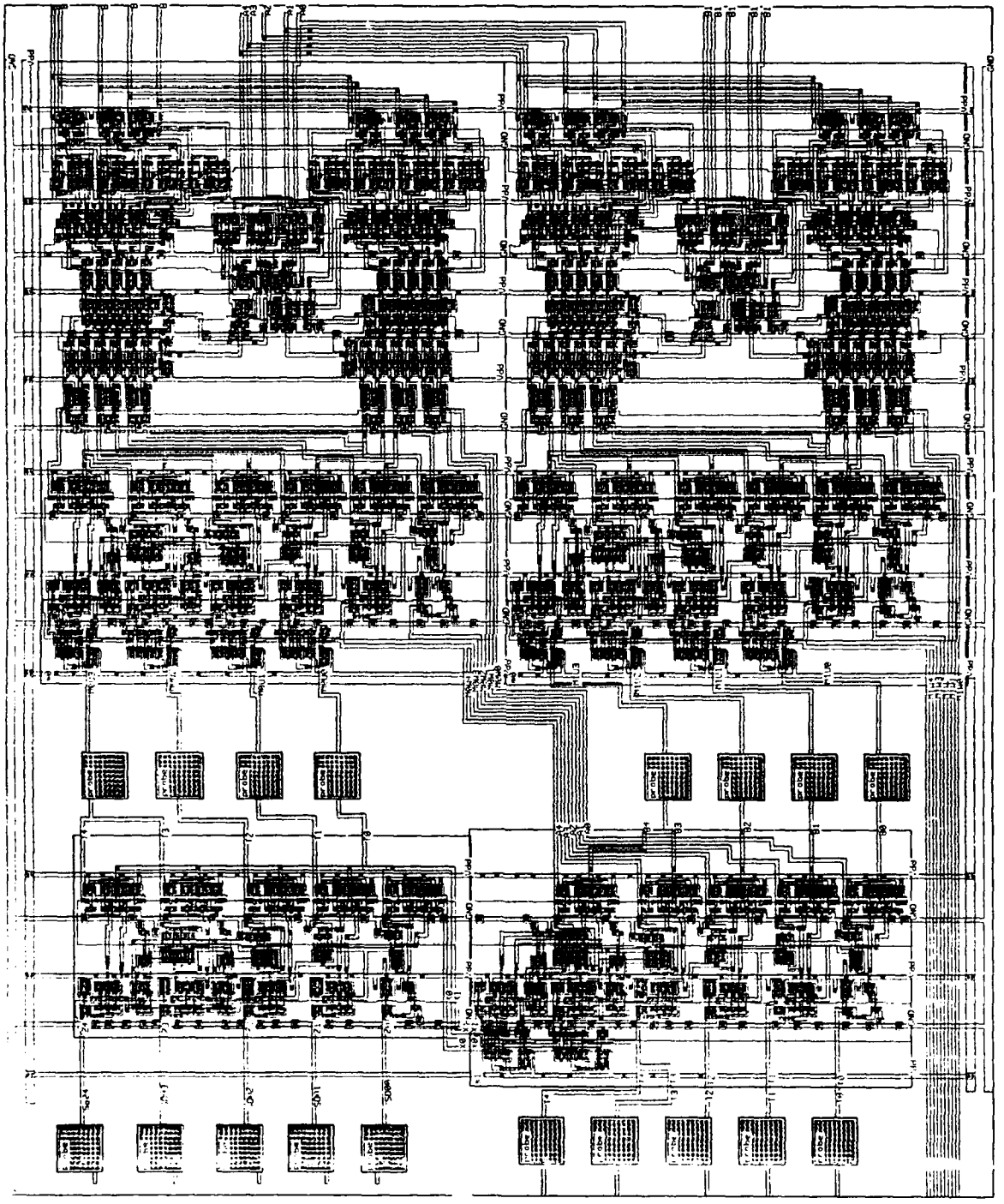

Figure B.4. CIFPLOT of Proposed SD Tiny Chip.

```
package SD_DEFINITIONS is

   subtype SD_DIGIT is bit_vector( 4 downto 0 );
   type SD_NUMBER is array ( 0 to 15 ) of SD_DIGIT;
   type PARTIAL_P is array ( integer range <> ) of SD_DIGIT;
   subtype X_TYPE is bit_vector( 1 downto 0 );
   subtype T_TYPE is bit_vector( 4 downto 0 );
   subtype U_TYPE is bit_vector( 3 downto 0 );
   subtype W_TYPE is bit_vector( 4 downto 0 );
   type T_ARRAY is array ( integer range <>) of T_TYPE;
   type X_ARRAY is array ( integer range <>) of X_TYPE;
   type U_ARRAY is array ( integer range <>) of U_TYPE;
   type W_ARRAY is array ( integer range <>) of W_TYPE;

   function U_TO_SD ( U_value : U_TYPE ) return SD_DIGIT;
   function U_TO_T  ( U_value : U_TYPE ) return T_TYPE;
   function BIN_TO_INT ( IN_VECT : bit_vector ) return INTEGER;
   function INT_TO_SD ( INT_VAL : integer ) return SD_DIGIT;

end SD_DEFINITIONS;

package body SD_DEFINITIONS is

   function BIN_TO_INT ( IN_VECT : bit_vector ) return INTEGER is

      variable vect_high, int_val, scale : integer;

   begin

      int_val := 0;
      scale   := 1;

      for i in 0 to ( IN_VECT'high - 1 ) loop
         if ( IN_VECT(i) = '1' ) then
            int_val := int_val + scale;
         end if;
         scale := scale*2;
      end loop;

      vect_high := IN_VECT'high;

      if ( IN_VECT(vect_high) = '1' ) then
         int_val := int_val - scale;
```

```
   end if;

   return ( int_val );

end BIN_TO_INT;


function INT_TO_SD ( INT_VAL : integer ) return SD_DIGIT is

   variable int_vect : SD_DIGIT;
   variable range_ck, temp : integer;

begin

   if ( INT_VAL < 0 ) then
      int_vect(4) := '1';
      temp := 16 + INT_VAL;
   else
      int_vect(4) := '0';
      temp := INT_VAL;
   end if;

   range_ck := 8;

   for i in 3 downto 0 loop
      if ( temp >= range_ck ) then
         int_vect(i) := '1';
         temp := temp - range_ck;
      else
         int_vect(i) := '0';
      end if;
      range_ck := range_ck/2;
   end loop;

   return ( int_vect );

end INT_TO_SD;



function U_TO_SD ( U_value : U_TYPE ) return SD_DIGIT is

   variable SD_value : SD_DIGIT;

begin
```

```
      SD_value(0) := U_value(0);
      SD_value(1) := U_value(1);
      SD_value(2) := U_value(2);
      SD_value(3) := U_value(3);
      SD_value(4) := U_value(3);

      return ( SD_value );

   end U_TO_SD;


   function U_TO_T ( U_value : U_TYPE ) return T_TYPE is

      variable T_value : T_TYPE;

   begin

      for I in 0 to 3 loop
         T_value(I) := U_value(I);
      end loop;

      T_value(4) := U_value(3);

      return ( T_value);

   end U_TO_T;


end SD_DEFINITIONS;
```

```
use work.SD_DEFINITIONS.all;
entity S1_RECODER is

   port    ( DATA_IN     : in bit_vector ( 3 downto 0 );
             X_out       : out X_TYPE;
             T_out       : out T_TYPE);

end S1_RECODER;


use work.SD_DEFINITIONS.all;
architecture Structural of S1_RECODER is

begin

   T_out(0) <= DATA_IN(0);
   T_out(1) <= DATA_IN(1);
   T_out(2) <= DATA_IN(2);
   T_out(3) <= DATA_IN(3);
   T_out(4) <= DATA_IN(3);
   X_out(0) <= DATA_IN(3);
   X_out(1) <= '0';

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity S1_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( SD1_in      : in SD_DIGIT;
             SD2_in      : in SD_DIGIT;
             ADD_SUB     : in bit;
             X_out       : out X_TYPE;
             T_out       : out T_TYPE);

end S1_ADDER;


use work.SD_DEFINITIONS.all;
architecture Behavioral of S1_ADDER is

begin

   process

      variable SD1_val, SD2_val, SUM : integer;
      variable X_temp : bit_vector ( 1 downto 0 );

   begin

      wait on SD1_in, SD2_in, ADD_SUB;

      SD1_val := BIN_TO_INT( SD1_in );
      SD2_val := BIN_TO_INT( SD2_in );

      if ( ADD_SUB = '0' ) then
         SUM := SD1_val + SD2_val;
      else
         SUM := SD1_val + SD2_val + 1;
      end if;

      if ( SUM >= 8 ) then
         SUM := SUM - 16;
         X_temp(0) := '1';
         X_temp(1) := '0';
      elsif ( SUM <= -8 ) then
         SUM := SUM + 16;
         X_temp(0) := '1';
         X_temp(1) := '1';
      else
```

```
            X_temp(0) := '0';
            X_temp(1) := '0';
        end if;

        X_out <= X_temp after ( TECHNOLOGY_SCALE * 6.1 ns);
        T_out <= INT_TO_SD( SUM ) after ( TECHNOLOGY_SCALE * 4.9 ns);

    end process;

end Behavioral;
```

```
use work.SD_DEFINITIONS.all;
entity S2_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( X_in         : in X_TYPE;
             T_in         : in T_TYPE;
             SD_out       : out SD_DIGIT);

end S2_ADDER;


use work.SD_DEFINITIONS.all;
architecture Behavioral of S2_ADDER is

begin

   process

      variable T_VAL, X_VAL, SUM : integer;

   begin

      wait on X_in, T_in;
      T_VAL := BIN_TO_INT( T_in );
      X_VAL := BIN_TO_INT( X_in );
      SUM := T_VAL + X_VAL;
      SD_out <= INT_TO_SD( SUM ) after ( TECHNOLOGY_SCALE * 4.9 ns);

   end process;

end Behavioral;
```

```
use work.SD_DEFINITIONS.all;
entity MO_MULT is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port    ( A_DIGIT     : in SD_DIGIT;
             B_DIGIT     : in SD_DIGIT;
             W_OUT       : out W_TYPE;
             U_OUT       : out U_TYPE);

end MO_MULT;

use work.SD_DEFINITIONS.all;
architecture Behavioral of MO_MULT is

begin

   process

      variable A_val, B_val, PROD, U_val : integer;
      variable long_U : bit_vector ( 4 downto 0 );

   begin

      wait on A_DIGIT, B_DIGIT;
      A_val := BIN_TO_INT( A_DIGIT );
      B_val := BIN_TO_INT( B_DIGIT );
      PROD  := A_val*B_val;
      U_val := 0;

      if ( PROD >= 0 ) then
         for i in 1 to 6 loop
            if ( PROD >= 8 ) then
               PROD := PROD - 16;
               U_val := U_val + 1;
            end if;
         end loop;
      else
         for i in 1 to 6 loop
            if ( PROD <= -8 ) then
               PROD := PROD + 16;
               U_val := U_val - 1;
            end if;
         end loop;
      end if;
```

```
        long_U := INT_TO_SD( U_val );

        U_OUT(0) <= long_U(0) after ( TECHNOLOGY_SCALE * 13.3 ns);
        U_OUT(1) <= long_U(1) after ( TECHNOLOGY_SCALE * 13.3 ns);
        U_OUT(2) <= long_U(2) after ( TECHNOLOGY_SCALE * 13.3 ns);
        U_OUT(3) <= long_U(3) after ( TECHNOLOGY_SCALE * 13.3 ns);
        W_OUT    <= INT_TO_SD( PROD ) after ( TECHNOLOGY_SCALE * 9.6 ns);

    end process;

end Behavioral;
```

```
use work.SD_DEFINITIONS.all;
entity CONVERSION_TB is
end CONVERSION_TB;


use work.SD_DEFINITIONS.all;
architecture TEST_CO of CONVERSION_TB is

   component S1_RECODER
      port    ( DATA_IN     : in bit_vector ( 3 downto 0 );
                X_out       : out X_TYPE;
                T_out       : out T_TYPE);
   end component;

   component S2_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( X_in        : in X_TYPE;
                T_in        : in T_TYPE;
                SD_out      : out SD_DIGIT);
   end component;

   for all : S1_RECODER use entity work.S1_RECODER(Structural);
   for all : S2_ADDER use entity work.S2_ADDER(Behavioral);

   signal SLICE0, SLICE1  : bit_vector ( 3 downto 0 );
   signal X_1, X0 : X_TYPE;
   signal T0 : T_TYPE;
   signal SD0, SD1 : SD_DIGIT;

begin

   S1R : S1_RECODER
         port map ( DATA_IN  => SLICE0,
                    X_out    => X_1,
                    T_out    => T0);

   S2R : S1_RECODER
         port map ( DATA_IN  => SLICE1,
                    X_out    => X0,
                    T_out    => SD1);

   S2A : S2_ADDER
         port map ( T_in     => T0,
                    X_in     => X0,
                    SD_out   => SD0);
```

```
    SLICE1  <= "0001" after 20 ns, "0010" after 40 ns,
               "0100" after 60 ns, "0110" after 80 ns,
               "1000" after 100 ns, "1010" after 120 ns,
               "1100" after 140 ns, "1110" after 160 ns,
               "1111" after 180 ns, "1110" after 220 ns,
               "1100" after 240 ns, "1010" after 260 ns,
               "1000" after 280 ns, "0110" after 300 ns,
               "0100" after 320 ns, "0010" after 340 ns,
               "0001" after 360 ns, "0000" after 380 ns;

    SLICE0  <= "0001" after 200 ns;

end TEST_C0;
```

```
                    SD Conversion module report"

                       Vhdl Simulation Report

                 Report Name: SD Conversion module report"
                  Kernel Library Name: <<RPETERSO>>TEST_CO
                     Kernel Creation Date: MAR-31-1989
                       Kernel Creation Time: 15:37:49
                             Run Identifer: 1
                          Run Date: MAR-31-1989
                            Run Time: 15:37:49

            Report Control Language File: conversion_report.rcl
                 Report Output File : conversion_report.rpt

                       Max Time: 9223372036854775807
                           Max Delta: 2147483646

Report Control Language :

  Simulation_report CONVERSION_report is
  begin
     Report_name is "SD Conversion module report";
     Page_width is 80;
     Page_length is 50;
     Signal_format is horizontal;
     Sample_signals by_event in ns;
     Select_signal    : SLICE0;
     Select_signal    : SLICE1;
     Select_signal    : SD0;
     Select_signal    : SD1;
  end CONVERSION_report;

Report Format Information :

  Time is in NS relative to the start of simulation
  Time period for report is from 0 NS to End of Simulation
 Signal values are reported by event  ( ' ' indicates no event )
```

SD Conversion module report"

TIME	SIGNAL NAMES			
	SLICE0	SLICE1	SD0	SD1
(NS)	(3 DOWNTO 0)	(3 DOWNTO 0)	(4 DOWNTO 0)	(4 DOWNTO 0)
0	"0000"	"0000"	"00000"	"00000"
20		"0001"		
+1				"00001"
40		"0010"		
+1				"00010"
60		"0100"		
+1				"00100"
80		"0110"		
+1				"00110"
100		"1000"		
+1				"11000"
104*			"00001"	
120		"1010"		
+1				"11010"
140		"1100"		
+1				"11100"
160		"1110"		
+1				"11110"
180		"1111"		
+1				"11111"
200	"0001"			
204*			"00010"	
220		"1110"		
+1				"11110"
240		"1100"		
+1				"11100"
260		"1010"		
+1				"11010"
280		"1000"		
+1				"11000"
300		"0110"		
+1				"00110"
304*			"00001"	
320		"0100"		
+1				"00100"
340		"0010"		
+1				"00010"

```
use work.SD_DEFINITIONS.all;
entity ADDER_TB is
end ADDER_TB;


use work.SD_DEFINITIONS.all;
architecture TEST_ADDER of ADDER_TB is

   component S1_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( SD1_in      : in SD_DIGIT;
                SD2_in      : in SD_DIGIT;
                ADD_SUB     : in bit;
                X_out       : out X_TYPE;
                T_out       : out T_TYPE);
   end component;

   component S2_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( X_in        : in X_TYPE;
                T_in        : in T_TYPE;
                SD_out      : out SD_DIGIT);
   end component;

   for all : S1_ADDER use entity work.S1_ADDER(Behavioral);
   for all : S2_ADDER use entity work.S2_ADDER(Behavioral);

   signal SD0, SD1, SD2, SDA, SDB, SD00, SD01, SD02 : SD_DIGIT;
   signal X0, X1 : X_TYPE;
   signal T1 : T_TYPE;
   signal ADD_CNTL : bit;

begin

   S1A : S1_ADDER
         port map ( SD1_in   => SD1,
                    SD2_in   => SDA,
                    ADD_SUB  => ADD_CNTL,
                    X_out    => X0,
                    T_out    => T1);

   S1B : S1_ADDER
         port map ( SD1_in   => SD2,
                    SD2_in   => SDB,
                    ADD_SUB  => ADD_CNTL,
```

```
                    X_out     => X1,
                    T_out     => SD02);

    S2A : S2_ADDER
          port map ( T_in     => SD0,
                     X_in     => X0,
                     SD_out   => SD00);

    S2B : S2_ADDER
          port map ( T_in     => T1,
                     X_in     => X1,
                     SD_out   => SD01);

    SD0  <= "00000";
    ADD_CNTL <= '0';

    SD1  <= "00100" after 25 ns, "01000" after 50 ns,
            "00000" after 75 ns, "11100" after 100 ns,
            "11000" after 125 ns, "10110" after 150 ns,
            "01010" after 175 ns, "00000" after 200 ns,
            "00100" after 225 ns, "01000" after 250 ns,
            "00000" after 275 ns, "11100" after 300 ns,
            "11000" after 325 ns, "10110" after 350 ns,
            "01010" after 375 ns;

    SDA  <= "00011", "11101" after 200 ns;

    SD2 <= SDA;
    SDB <= SD1;

end TEST_ADDER;
```

```
MAR-31-1989 15:39:25              VHDL Report Generator          PAGE 1
                          SD Adder module report"

                              Vhdl Simulation Report

                          Report Name: SD Adder module report"
                       Kernel Library Name: <<RPETERSO>>TEST_ADDER
                            Kernel Creation Date: MAR-31-1989
                              Kernel Creation Time: 15:38:42
                                    Run Identifer: 1
                                 Run Date: MAR-31-1989
                                   Run Time: 15:38:42

                        Report Control Language File: adder_report.rcl
                           Report Output File : adder_report.rpt

                              Max Time: 9223372036854775807
                                 Max Delta: 2147483646

Report Control Language :

  Simulation_report ADDER_report is
  begin
     Report_name is "SD Adder module report";
     Page_width is 80;
     Page_length is 50;
     Signal_format is vertical;
     Sample_signals by_event in ns;
     Select_signal   : SD1;
     Select_signal   : SD2;
     Select_signal   : SDA;
     Select_signal   : SDB;
     Select_signal   : SD00;
     Select_signal   : SD01;
     Select_signal   : SD02;
  end ADDER_report;

Report Format Information :

  Time is in NS relative to the start of simulation
  Time period for report is from 0 NS to End of Simulation
  Signal values are reported by event  ( ' ' indicates no event )
```

SD Adder module report"

TIME	SIGNAL NAMES						
(NS)	SD1(4)	SD2(4)	SDA(4)	SDB(4)	SD0(0(4)	SD0(1(4)	SD0(2(4)
	SD0WNT0(0)	SD0WNT0(0)	SD0WNT0(0)	SD0WNT0(0)	SD0WNT0(0)	SD0WNT0(0)	SD0WNT0(0)
0	"00000"	"00000"	"00000"	"00000"	"00000"	"00000"	"00000"
+1			"00011"				
+2		"00011"					
4*							"00011"
9*						"00011"	
25	"00100"						
+1				"00100"			
29*							"00111"
34*						"00111"	
50	"01000"						
+1				"01000"			
54*							"11011"
59*						"11111"	
61					"00001"	"11100"	
75	"00000"						
+1				"00000"			
79*							"00011"
84*						"00100"	
86					"00000"	"00011"	
100	"11100"						
+1				"11100"			
104*							"11111"
109*						"11111"	

SD Adder module report"

TIME	SIGNAL NAMES						
(NS)	SD1(4)	SD2(4)	SDA(4)	SDB(4)	SD0(4)	SD1(4)	SD2(4)
	SD0WN T0)	SD0WN T0)	SD0WN T0)	SD0WN T0)	SD0WN T0)	SD0WN T0)	SD0WN T0)
125	"11000"						
+1				"11000"			
129*							"11011"
134*						"11011"	
150	"10110"						
+1				"10110"			
154*							"11001"
159*						"11001"	
175	"01010"						
+1				"01010"			
179*							"11101"
184*						"11101"	
186					"00001"	"11110"	
200	"00000"		"11101"				
+1		"11101"		"00000"			
211					"00000"	"11101"	
225	"00100"						
+1				"00100"			
229*							"00001"
234*						"00001"	
250	"01000"						

SD Adder module report"

TIME	SIGNAL NAMES						
(NS)	SD1(4)DOWNTO0)	SD2(4)DOWNTO0)	SDA(4)DOWNTO0)	SDB(4)DOWNTO0)	SD00(4)DOWNTO0)	SD01(4)DOWNTO0)	SD02(4)DOWNTO0)
+1				"01000"			"00101"
254*						"00101"	
259*							
275	"00000"						
+1				"00000"			"11101"
279*						"11101"	
284*							
300	"11100"						
+1				"11100"			"11001"
304*						"11001"	
309*							
325	"11000"						
+1				"11000"			"00101"
329*						"00101"	
334*					"11111"	"00100"	
336							
350	"10110"						
+1				"10110"			"00011"
354*						"00010"	
359*							

```
use work.SD_DEFINITIONS.all;
entity MO_TB is
end MO_TB;


use work.SD_DEFINITIONS.all;
architecture TEST_MO of MO_TB is

   component MO_MULT
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( A_DIGIT     : in SD_DIGIT;
                B_DIGIT     : in SD_DIGIT;
                W_OUT       : out W_TYPE;
                U_OUT       : out U_TYPE);
   end component;

   for all : MO_MULT use entity work.MO_MULT(Behavioral);

   signal A_DIGIT, B_DIGIT : SD_DIGIT;
   signal W_out : W_TYPE;
   signal U_out : U_TYPE;

begin

   MOO : MO_MULT
         port    map ( A_DIGIT     => A_DIGIT,
                       B_DIGIT     => B_DIGIT,
                       W_OUT       => W_out,
                       U_OUT       => U_out);

   A_DIGIT <= "01010" after 50 ns, "10110" after 100 ns,
              "00000" after 150 ns, "01010" after 200 ns,
              "10110" after 250 ns, "00000" after 300 ns,
              "01010" after 350 ns, "10110" after 400 ns,
              "00000" after 450 ns, "01010" after 500 ns,
              "10110" after 550 ns;

   B_DIGIT <= "00001" after 150 ns, "01010" after 300 ns,
              "10110" after 450 ns;

end TEST_MO;
```

M0 Multiplier module report"

Vhdl Simulation Report

Report Name: M0 Multiplier module report"
Kernel Library Name: <<PETERSON>>TEST_M0
Kernel Creation Date: APR-12-1989
Kernel Creation Time: 10:48:07
Run Identifer: 1
Run Date: APR-12-1989
Run Time: 10:48:07

Report Control Language File: m0_report.rcl
Report Output File : m0_report.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report M0_report is
begin
   Report_name is "M0 Multiplier module report";
   Page_width is 80;
   Page_length is 50;
   Signal_format is vertical;
   Sample_signals by_event in ns;
   Select_signal   : A_DIGIT;
   Select_signal   : B_DIGIT;
   Select_signal   : U_out;
   Select_signal   : W_out;
end M0_report;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by event (' ' indicates no event)

MO Multiplier module report"

TIME	SIGNAL NAMES			
(NS)	A	B	U	W
	-	-	-	-
	D	D	0	0
	I	I	U	U
	G	G	T	T
	I	I	(	(
	T	T	3	4
	(	(		
	4	4	D	D
			0	0
	D	D	W	W
	0	0	N	N
	W	W	T	T
	N	N	0	0
	T	T		
	0	0	0	0
			)	)
	0	0		
	)	)		
0	"00000"	"00000"	"0000"	"00000"
50	"01010"			
100	"10110"			
150	"00000"	"00001"		
200	"01010"			
209*				"11010"
213*			"0001"	
250	"10110"			
259*				"00110"
263*			"1111"	
300	"00000"	"01010"		
309*				"00000"
313*			"0000"	
350	"01010"			
359*				"00100"
363*			"0110"	
400	"10110"			
409*				"11100"
413*			"1010"	

MO Multiplier module report"

TIME	SIGNAL NAMES			
(NS)	A	B	U	W
	-	-	-	-
	D	D	0	0
	I	I	U	U
	G	G	T	T
	I	I	(	(
	T	T	3	4
	(	(		
	4	4	D	D
			0	0
	D	D	W	W
	0	0	N	N
	W	W	T	T
	N	N	0	0
	T	T		
	0	0	0	0
			)	)
	0	0		
	)	)		
450	"00000"	"10110"		
459*				"00000"
463*			"0000"	
500	"01010"			
509*				"11100"
513*			"1010"	
550	"10110"			
559*				"00100"
563*			"0110"	

```
use work.SD_DEFINITIONS.all;
entity MULT_BLOCK is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( DIGIT_C : in SD_DIGIT;
          SD_NUMB : in SD_NUMBER;
          RESULT  : out PARTIAL_P ( 0 to 16));

end MULT_BLOCK;


use work.SD_DEFINITIONS.all;
architecture Structural of MULT_BLOCK is

   component MO_MULT
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( A_DIGIT     : in SD_DIGIT;
                B_DIGIT     : in SD_DIGIT;
                W_OUT       : out W_TYPE;
                U_OUT       : out U_TYPE);
   end component;

   component S1_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( SD1_in      : in SD_DIGIT;
                SD2_in      : in SD_DIGIT;
                ADD_SUB     : in bit;
                X_out       : out X_TYPE;
                T_out       : out T_TYPE);
   end component;

   component S2_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( X_in        : in X_TYPE;
                T_in        : in T_TYPE;
                SD_out      : out SD_DIGIT);
   end component;


   for all : MO_MULT use entity work.MO_MULT(Behavioral);
   for all : S1_ADDER use entity work.S1_ADDER(Behavioral);
   for all : S2_ADDER use entity work.S2_ADDER(Behavioral);

   signal W_ARR : W_ARRAY ( 0 to 15 );
   signal U_ARR : U_ARRAY ( 0 to 15 );
```

```
    signal UDIG  : PARTIAL_P( 0 to 15 );
    signal X_ARR : X_ARRAY ( 0 to 14 );
    signal T_ARR : T_ARRAY ( 0 to 14 );
    signal ADD_CNTL : bit;

begin

    M00 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
                     B_DIGIT  => SD_NUMB(0),
                     W_OUT    => W_ARR(0),
                     U_OUT    => U_ARR(0));

    M01 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
                     B_DIGIT  => SD_NUMB(1),
                     W_OUT    => W_ARR(1),
                     U_OUT    => U_ARR(1));

    M02 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
                     B_DIGIT  => SD_NUMB(2),
                     W_OUT    => W_ARR(2),
                     U_OUT    => U_ARR(2));

    M03 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
                     B_DIGIT  => SD_NUMB(3),
                     W_OUT    => W_ARR(3),
                     U_OUT    => U_ARR(3));

    M04 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
                     B_DIGIT  => SD_NUMB(4),
                     W_OUT    => W_ARR(4),
                     U_OUT    => U_ARR(4));

    M05 : M0_MULT
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( A_DIGIT  => DIGIT_C,
```

```
                B_DIGIT  => SD_NUMB(5),
                W_OUT    => W_ARR(5),
                U_OUT    => U_ARR(5));

M06 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(6),
                 W_OUT    => W_ARR(6),
                 U_OUT    => U_ARR(6));

M07 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(7),
                 W_OUT    => W_ARR(7),
                 U_OUT    => U_ARR(7));

M08 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(8),
                 W_OUT    => W_ARR(8),
                 U_OUT    => U_ARR(8));

M09 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(9),
                 W_OUT    => W_ARR(9),
                 U_OUT    => U_ARR(9));

M10 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(10),
                 W_OUT    => W_ARR(10),
                 U_OUT    => U_ARR(10));

M11 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(11),
                 W_OUT    => W_ARR(11),
                 U_OUT    => U_ARR(11));
```

```
M12 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(12),
                 W_OUT    => W_ARR(12),
                 U_OUT    => U_ARR(12));

M13 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(13),
                 W_OUT    => W_ARR(13),
                 U_OUT    => U_ARR(13));

M14 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(14),
                 W_OUT    => W_ARR(14),
                 U_OUT    => U_ARR(14));

M15 : MO_MULT
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( A_DIGIT  => DIGIT_C,
                 B_DIGIT  => SD_NUMB(15),
                 W_OUT    => W_ARR(15),
                 U_OUT    => U_ARR(15));

 UDIG(0) <= U_TO_T( U_ARR(0));
 UDIG(1) <= U_TO_T( U_ARR(1));
 UDIG(2) <= U_TO_T( U_ARR(2));
 UDIG(3) <= U_TO_T( U_ARR(3));
 UDIG(4) <= U_TO_T( U_ARR(4));
 UDIG(5) <= U_TO_T( U_ARR(5));
 UDIG(6) <= U_TO_T( U_ARR(6));
 UDIG(7) <= U_TO_T( U_ARR(7));
 UDIG(8) <= U_TO_T( U_ARR(8));
 UDIG(9) <= U_TO_T( U_ARR(9));
 UDIG(10) <= U_TO_T( U_ARR(10));
 UDIG(11) <= U_TO_T( U_ARR(11));
 UDIG(12) <= U_TO_T( U_ARR(12));
 UDIG(13) <= U_TO_T( U_ARR(13));
 UDIG(14) <= U_TO_T( U_ARR(14));
 UDIG(15) <= U_TO_T( U_ARR(15));
```

```
S10 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(1),
                 SD2_in   => W_ARR(0),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(0),
                 T_out    => T_ARR(0));

S11 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(2),
                 SD2_in   => W_ARR(1),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(1),
                 T_out    => T_ARR(1));

S12 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(3),
                 SD2_in   => W_ARR(2),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(2),
                 T_out    => T_ARR(2));

S13 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(4),
                 SD2_in   => W_ARR(3),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(3),
                 T_out    => T_ARR(3));

S14 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(5),
                 SD2_in   => W_ARR(4),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(4),
                 T_out    => T_ARR(4));

S15 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(6),
                 SD2_in   => W_ARR(5),
```

```
                ADD_SUB  => ADD_CNTL,
                X_out    => X_ARR(5),
                T_out    => T_ARR(5));

S16 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(7),
                 SD2_in   => W_ARR(6),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(6),
                 T_out    => T_ARR(6));

S17 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(8),
                 SD2_in   => W_ARR(7),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(7),
                 T_out    => T_ARR(7));

S18 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(9),
                 SD2_in   => W_ARR(8),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(8),
                 T_out    => T_ARR(8));

S19 : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(10),
                 SD2_in   => W_ARR(9),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(9),
                 T_out    => T_ARR(9));

S1A : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(11),
                 SD2_in   => W_ARR(10),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(10),
                 T_out    => T_ARR(10));

S1B : S1_ADDER
```

```
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(12),
                 SD2_in   => W_ARR(11),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(11),
                 T_out    => T_ARR(11));

S1C : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(13),
                 SD2_in   => W_ARR(12),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(12),
                 T_out    => T_ARR(12));

S1D : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(14),
                 SD2_in   => W_ARR(13),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(13),
                 T_out    => T_ARR(13));

S1E : S1_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( SD1_in   => UDIG(15),
                 SD2_in   => W_ARR(14),
                 ADD_SUB  => ADD_CNTL,
                 X_out    => X_ARR(14),
                 T_out    => T_ARR(14));

 S20 : S2_ADDER
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( X_in  => X_ARR(0),
                  T_in  => UDIG(0),
                  SD_out => RESULT(0));

 S21 : S2_ADDER
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( X_in  => X_ARR(1),
                  T_in  => T_ARR(0),
                  SD_out => RESULT(1));

 S22 : S2_ADDER
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
```

```
      port map ( X_in  => X_ARR(2),
                 T_in  => T_ARR(1),
                 SD_out => RESULT(2));

S23 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(3),
                 T_in  => T_ARR(2),
                 SD_out => RESULT(3));

S24 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(4),
                 T_in  => T_ARR(3),
                 SD_out => RESULT(4));

S25 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(5),
                 T_in  => T_ARR(4),
                 SD_out => RESULT(5));

S26 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(6),
                 T_in  => T_ARR(5),
                 SD_out => RESULT(6));

S27 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(7),
                 T_in  => T_ARR(6),
                 SD_out => RESULT(7));

S28 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(8),
                 T_in  => T_ARR(7),
                 SD_out => RESULT(8));

S29 : S2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( X_in  => X_ARR(9),
                 T_in  => T_ARR(8),
                 SD_out => RESULT(9));
```

```
    S2A : S2_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( X_in  => X_ARR(10),
                     T_in  => T_ARR(9),
                     SD_out => RESULT(10));

    S2B : S2_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( X_in  => X_ARR(11),
                     T_in  => T_ARR(10),
                     SD_out => RESULT(11));

    S2C : S2_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( X_in  => X_ARR(12),
                     T_in  => T_ARR(11),
                     SD_out => RESULT(12));

    S2D : S2_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( X_in  => X_ARR(13),
                     T_in  => T_ARR(12),
                     SD_out => RESULT(13));

    S2E : S2_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( X_in  => X_ARR(14),
                     T_in  => T_ARR(13),
                     SD_out => RESULT(14));

    RESULT(15) <= T_ARR(14);
    RESULT(16) <= W_ARR(15);

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity ADDER_1 is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( SD1      : in SD_DIGIT;
          SD2      : in SD_DIGIT;
          T_in     : in T_TYPE;
          T_out    : out T_TYPE;
          SUMr     : out SD_DIGIT);

end ADDER_1;

use work.SD_DEFINITIONS.all;
architecture Structural of ADDER_1 is

   component S1_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( SD1_in        : in SD_DIGIT;
                SD2_in        : in SD_DIGIT;
                ADD_SUB       : in bit;
                X_out         : out X_TYPE;
                T_out         : out T_TYPE);
   end component;

   component S2_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port    ( X_in          : in X_TYPE;
                T_in          : in T_TYPE;
                SD_out        : out SD_DIGIT );
   end component;

   for all : S1_ADDER use entity work.S1_ADDER(Behavioral);
   for all : S2_ADDER use entity work.S2_ADDER(Behavioral);

   signal XDIG : X_TYPE;
   signal ADD_SIG : bit;

begin

   S1 : S1_ADDER
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1_in     => SD1,
                   SD2_in     => SD2,
                   ADD_SUB    => ADD_SIG,
                   X_out      => XDIG,
```

```
                    T_out      => T_out );

    S2 : S2_ADDER
         generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
         port map ( X_in       => XDIG,
                    T_in       => T_in,
                    SD_out     => SUMr );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SL2_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( PARTIAL_H : in PARTIAL_P ( 0 to 16 );
          PARTIAL_L : in PARTIAL_P ( 0 to 16 );
          P_out     : out PARTIAL_P ( 0 to 17 ));

end SL2_ADDER;

use work.SD_DEFINITIONS.all;
architecture Structural of SL2_ADDER is

   component ADDER_1
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( SD1   : in SD_DIGIT;
             SD2   : in SD_DIGIT;
             T_in  : in T_TYPE;
             T_out : out T_TYPE;
             SUMr  : out SD_DIGIT );
   end component;

   for all : ADDER_1 use entity work.ADDER_1(Structural);

   signal T_ARR : T_ARRAY ( 0 to 16 );

begin

   T_ARR(0) <= PARTIAL_H(0);

   ADD0 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 1 ),
                     SD2   => PARTIAL_L( 0 ),
                     T_in  => T_ARR( 0 ),
                     T_out => T_ARR( 1 ),
                     SUMr  => P_out( 0 ) );

   ADD1 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 2 ),
                     SD2   => PARTIAL_L( 1 ),
                     T_in  => T_ARR( 1 ),
                     T_out => T_ARR( 2 ),
                     SUMr  => P_out( 1 ) );
```

```
ADD2 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 3 ),
                  SD2   => PARTIAL_L( 2 ),
                  T_in  => T_ARR( 2 ),
                  T_out => T_ARR( 3 ),
                  SUMr  => P_out( 2 ) );

ADD3 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 4 ),
                  SD2   => PARTIAL_L( 3 ),
                  T_in  => T_ARR( 3 ),
                  T_out => T_ARR( 4 ),
                  SUMr  => P_out( 3 ) );

ADD4 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 5 ),
                  SD2   => PARTIAL_L( 4 ),
                  T_in  => T_ARR( 4 ),
                  T_out => T_ARR( 5 ),
                  SUMr  => P_out( 4 ) );

ADD5 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 6 ),
                  SD2   => PARTIAL_L( 5 ),
                  T_in  => T_ARR( 5 ),
                  T_out => T_ARR( 6 ),
                  SUMr  => P_out( 5 ) );

ADD6 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 7 ),
                  SD2   => PARTIAL_L( 6 ),
                  T_in  => T_ARR( 6 ),
                  T_out => T_ARR( 7 ),
                  SUMr  => P_out( 6 ) );

ADD7 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 8 ),
                  SD2   => PARTIAL_L( 7 ),
```

```
                T_in  => T_ARR( 7 ),
                T_out => T_ARR( 8 ),
                SUMr  => P_out( 7 ) );

ADD8 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 9 ),
                  SD2   => PARTIAL_L( 8 ),
                  T_in  => T_ARR( 8 ),
                  T_out => T_ARR( 9 ),
                  SUMr  => P_out( 8 ) );

ADD9 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 10 ),
                  SD2   => PARTIAL_L( 9 ),
                  T_in  => T_ARR( 9 ),
                  T_out => T_ARR( 10 ),
                  SUMr  => P_out( 9 ) );

ADDA : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 11 ),
                  SD2   => PARTIAL_L( 10 ),
                  T_in  => T_ARR( 10 ),
                  T_out => T_ARR( 11 ),
                  SUMr  => P_out( 10 ) );

ADDB : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 12 ),
                  SD2   => PARTIAL_L( 11 ),
                  T_in  => T_ARR( 11 ),
                  T_out => T_ARR( 12 ),
                  SUMr  => P_out( 11 ) );

ADDC : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 13 ),
                  SD2   => PARTIAL_L( 12 ),
                  T_in  => T_ARR( 12 ),
                  T_out => T_ARR( 13 ),
                  SUMr  => P_out( 12 ) );

ADDD : ADDER_1
```

```
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 14 ),
                   SD2   => PARTIAL_L( 13 ),
                   T_in  => T_ARR( 13 ),
                   T_out => T_ARR( 14 ),
                   SUMr  => P_out( 13 ) );

    ADDE : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 15 ),
                   SD2   => PARTIAL_L( 14 ),
                   T_in  => T_ARR( 14 ),
                   T_out => T_ARR( 15 ),
                   SUMr  => P_out( 14 ) );

    ADDF : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 16 ),
                   SD2   => PARTIAL_L( 15 ),
                   T_in  => T_ARR( 15 ),
                   T_out => T_ARR( 16 ),
                   SUMr  => P_out( 15 ) );

    P_out( 16 ) <= T_arr( 16 );
    P_out( 17 ) <= PARTIAL_L( 16 );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SL3_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( PARTIAL_H : in PARTIAL_P ( 0 to 17 );
          PARTIAL_L : in PARTIAL_P ( 0 to 17 );
          P_out     : out PARTIAL_P ( 0 to 19 ));

end SL3_ADDER;

use work.SD_DEFINITIONS.all;
architecture Structural of SL3_ADDER is

   component ADDER_1
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( SD1   : in SD_DIGIT;
             SD2   : in SD_DIGIT;
             T_in  : in T_TYPE;
             T_out : out T_TYPE;
             SUMr  : out SD_DIGIT );
   end component;

   for all : ADDER_1 use entity work.ADDER_1(Structural);

   signal T_ARR : T_ARRAY ( 0 to 16 );

begin

   P_out(0) <= PARTIAL_H(0);

   T_ARR(0) <= PARTIAL_H(1);

   ADD0 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 2 ),
                     SD2   => PARTIAL_L( 0 ),
                     T_in  => T_ARR( 0 ),
                     T_out => T_ARR( 1 ),
                     SUMr  => P_out( 1 ) );

   ADD1 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 3 ),
                     SD2   => PARTIAL_L( 1 ),
                     T_in  => T_ARR( 1 ),
```

```
                T_out => T_ARR( 2 ),
                SUMr  => P_out( 2 ) );

ADD2 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 4 ),
                  SD2   => PARTIAL_L( 2 ),
                  T_in  => T_ARR( 2 ),
                  T_out => T_ARR( 3 ),
                  SUMr  => P_out( 3 ) );

ADD3 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 5 ),
                  SD2   => PARTIAL_L( 3 ),
                  T_in  => T_ARR( 3 ),
                  T_out => T_ARR( 4 ),
                  SUMr  => P_out( 4 ) );

ADD4 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 6 ),
                  SD2   => PARTIAL_L( 4 ),
                  T_in  => T_ARR( 4 ),
                  T_out => T_ARR( 5 ),
                  SUMr  => P_out( 5 ) );

ADD5 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 7 ),
                  SD2   => PARTIAL_L( 5 ),
                  T_in  => T_ARR( 5 ),
                  T_out => T_ARR( 6 ),
                  SUMr  => P_out( 6 ) );

ADD6 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 8 ),
                  SD2   => PARTIAL_L( 6 ),
                  T_in  => T_ARR( 6 ),
                  T_out => T_ARR( 7 ),
                  SUMr  => P_out( 7 ) );

ADD7 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
```

```
        port map ( SD1   => PARTIAL_H( 9 ),
                   SD2   => PARTIAL_L( 7 ),
                   T_in  => T_ARR( 7 ),
                   T_out => T_ARR( 8 ),
                   SUMr  => P_out( 8 ) );

ADD8 : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 10 ),
                   SD2   => PARTIAL_L( 8 ),
                   T_in  => T_ARR( 8 ),
                   T_out => T_ARR( 9 ),
                   SUMr  => P_out( 9 ) );

ADD9 : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 11 ),
                   SD2   => PARTIAL_L( 9 ),
                   T_in  => T_ARR( 9 ),
                   T_out => T_ARR( 10 ),
                   SUMr  => P_out( 10 ) );

ADDA : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 12 ),
                   SD2   => PARTIAL_L( 10 ),
                   T_in  => T_ARR( 10 ),
                   T_out => T_ARR( 11 ),
                   SUMr  => P_out( 11 ) );

ADDB : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 13 ),
                   SD2   => PARTIAL_L( 11 ),
                   T_in  => T_ARR( 11 ),
                   T_out => T_ARR( 12 ),
                   SUMr  => P_out( 12 ) );

ADDC : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 14 ),
                   SD2   => PARTIAL_L( 12 ),
                   T_in  => T_ARR( 12 ),
                   T_out => T_ARR( 13 ),
                   SUMr  => P_out( 13 ) );
```

```
    ADDD : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 15 ),
                      SD2   => PARTIAL_L( 13 ),
                      T_in  => T_ARR( 13 ),
                      T_out => T_ARR( 14 ),
                      SUMr  => P_out( 14 ) );

    ADDE : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 16 ),
                      SD2   => PARTIAL_L( 14 ),
                      T_in  => T_ARR( 14 ),
                      T_out => T_ARR( 15 ),
                      SUMr  => P_out( 15 ) );

    ADDF : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 17 ),
                      SD2   => PARTIAL_L( 15 ),
                      T_in  => T_ARR( 15 ),
                      T_out => T_ARR( 16 ),
                      SUMr  => P_out( 16 ) );

    P_out( 17 ) <= T_ARR( 16 );
    P_out( 18 ) <= PARTIAL_L( 16 );
    P_out( 19 ) <= PARTIAL_L( 17 );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SL4_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( PARTIAL_H : in PARTIAL_P ( 0 to 19 );
          PARTIAL_L : in PARTIAL_P ( 0 to 19 );
          P_out     : out PARTIAL_P ( 0 to 23 ));

end SL4_ADDER;

use work.SD_DEFINITIONS.all;
architecture Structural of SL4_ADDER is

   component ADDER_1
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( SD1   : in SD_DIGIT;
             SD2   : in SD_DIGIT;
             T_in  : in T_TYPE;
             T_out : out T_TYPE;
             SUMr  : out SD_DIGIT );
   end component;

   for all : ADDER_1 use entity work.ADDER_1(Structural);

   signal T_ARR : T_ARRAY ( 0 to 16 );

begin

   P_out(0) <= PARTIAL_H(0);
   P_out(1) <= PARTIAL_H(1);
   P_out(2) <= PARTIAL_H(2);

   T_ARR(0) <= PARTIAL_H(3);

   ADD0 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 4 ),
                     SD2   => PARTIAL_L( 0 ),
                     T_in  => T_ARR( 0 ),
                     T_out => T_ARR( 1 ),
                     SUMr  => P_out( 3 ) );

   ADD1 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 5 ),
```

```
                SD2   => PARTIAL_L( 1 ),
                T_in  => T_ARR( 1 ),
                T_out => T_ARR( 2 ),
                SUMr  => P_out( 4 ) );

ADD2 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 6 ),
                  SD2   => PARTIAL_L( 2 ),
                  T_in  => T_ARR( 2 ),
                  T_out => T_ARR( 3 ),
                  SUMr  => P_out( 5 ) );

ADD3 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 7 ),
                  SD2   => PARTIAL_L( 3 ),
                  T_in  => T_ARR( 3 ),
                  T_out => T_ARR( 4 ),
                  SUMr  => P_out( 6 ) );

ADD4 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 8 ),
                  SD2   => PARTIAL_L( 4 ),
                  T_in  => T_ARR( 4 ),
                  T_out => T_ARR( 5 ),
                  SUMr  => P_out( 7 ) );

ADD5 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 9 ),
                  SD2   => PARTIAL_L( 5 ),
                  T_in  => T_ARR( 5 ),
                  T_out => T_ARR( 6 ),
                  SUMr  => P_out( 8 ) );

ADD6 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 10 ),
                  SD2   => PARTIAL_L( 6 ),
                  T_in  => T_ARR( 6 ),
                  T_out => T_ARR( 7 ),
                  SUMr  => P_out( 9 ) );
```

```
ADD7 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 11 ),
                  SD2   => PARTIAL_L( 7 ),
                  T_in  => T_ARR( 7 ),
                  T_out => T_ARR( 8 ),
                  SUMr  => P_out( 10 ) );

ADD8 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 12 ),
                  SD2   => PARTIAL_L( 8 ),
                  T_in  => T_ARR( 8 ),
                  T_out => T_ARR( 9 ),
                  SUMr  => P_out( 11 ) );

ADD9 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 13 ),
                  SD2   => PARTIAL_L( 9 ),
                  T_in  => T_ARR( 9 ),
                  T_out => T_ARR( 10 ),
                  SUMr  => P_out( 12 ) );

ADDA : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 14 ),
                  SD2   => PARTIAL_L( 10 ),
                  T_in  => T_ARR( 10 ),
                  T_out => T_ARR( 11 ),
                  SUMr  => P_out( 13 ) );

ADDB : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 15 ),
                  SD2   => PARTIAL_L( 11 ),
                  T_in  => T_ARR( 11 ),
                  T_out => T_ARR( 12 ),
                  SUMr  => P_out( 14 ) );

ADDC : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 16 ),
                  SD2   => PARTIAL_L( 12 ),
                  T_in  => T_ARR( 12 ),
```

```
                    T_out => T_ARR( 13 ),
                    SUMr  => P_out( 15 ) );

    ADDD : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 17 ),
                      SD2   => PARTIAL_L( 13 ),
                      T_in  => T_ARR( 13 ),
                      T_out => T_ARR( 14 ),
                      SUMr  => P_out( 16 ) );

    ADDE : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 18 ),
                      SD2   => PARTIAL_L( 14 ),
                      T_in  => T_ARR( 14 ),
                      T_out => T_ARR( 15 ),
                      SUMr  => P_out( 17 ) );

    ADDF : ADDER_1
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( SD1   => PARTIAL_H( 19 ),
                      SD2   => PARTIAL_L( 15 ),
                      T_in  => T_ARR( 15 ),
                      T_out => T_ARR( 16 ),
                      SUMr  => P_out( 18 ) );

    P_out( 19 ) <= T_ARR( 16 );
    P_out( 20 ) <= PARTIAL_L( 16 );
    P_out( 21 ) <= PARTIAL_L( 17 );
    P_out( 22 ) <= PARTIAL_L( 18 );
    P_out( 23 ) <= PARTIAL_L( 19 );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SL5_ADDER is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( PARTIAL_H : in PARTIAL_P ( 0 to 23 );
          PARTIAL_L : in PARTIAL_P ( 0 to 23 );
          P_out     : out PARTIAL_P ( 0 to 31 ));

end SL5_ADDER;

use work.SD_DEFINITIONS.all;
architecture Structural of SL5_ADDER is

   component ADDER_1
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( SD1   : in SD_DIGIT;
             SD2   : in SD_DIGIT;
             T_in  : in T_TYPE;
             T_out : out T_TYPE;
             SUMr  : out SD_DIGIT );
   end component;

   for all : ADDER_1 use entity work.ADDER_1(Structural);

   signal T_ARR : T_ARRAY ( 0 to 16 );

begin

   P_out(0) <= PARTIAL_H(0);
   P_out(1) <= PARTIAL_H(1);
   P_out(2) <= PARTIAL_H(2);
   P_out(3) <= PARTIAL_H(3);
   P_out(4) <= PARTIAL_H(4);
   P_out(5) <= PARTIAL_H(5);
   P_out(6) <= PARTIAL_H(6);

   T_ARR(0) <= PARTIAL_H(7);

   ADD0 : ADDER_1
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( SD1   => PARTIAL_H( 8 ),
                     SD2   => PARTIAL_L( 0 ),
                     T_in  => T_ARR( 0 ),
                     T_out => T_ARR( 1 ),
                     SUMr  => P_out( 7 ) );
```

```
ADD1 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 9 ),
                  SD2   => PARTIAL_L( 1 ),
                  T_in  => T_ARR( 1 ),
                  T_out => T_ARR( 2 ),
                  SUMr  => P_out( 8 ) );

ADD2 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 10 ),
                  SD2   => PARTIAL_L( 2 ),
                  T_in  => T_ARR( 2 ),
                  T_out => T_ARR( 3 ),
                  SUMr  => P_out( 9 ) );

ADD3 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 11 ),
                  SD2   => PARTIAL_L( 3 ),
                  T_in  => T_ARR( 3 ),
                  T_out => T_ARR( 4 ),
                  SUMr  => P_out( 10 ) );

ADD4 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 12 ),
                  SD2   => PARTIAL_L( 4 ),
                  T_in  => T_ARR( 4 ),
                  T_out => T_ARR( 5 ),
                  SUMr  => P_out( 11 ) );

ADD5 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 13 ),
                  SD2   => PARTIAL_L( 5 ),
                  T_in  => T_ARR( 5 ),
                  T_out => T_ARR( 6 ),
                  SUMr  => P_out( 12 ) );

ADD6 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 14 ),
                  SD2   => PARTIAL_L( 6 ),
```

```
                T_in  => T_ARR( 6 ),
                T_out => T_ARR( 7 ),
                SUMr  => P_out( 13 ) );

ADD7 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 15 ),
                  SD2   => PARTIAL_L( 7 ),
                  T_in  => T_ARR( 7 ),
                  T_out => T_ARR( 8 ),
                  SUMr  => P_out( 14 ) );

ADD8 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 16 ),
                  SD2   => PARTIAL_L( 8 ),
                  T_in  => T_ARR( 8 ),
                  T_out => T_ARR( 9 ),
                  SUMr  => P_out( 15 ) );

ADD9 : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 17 ),
                  SD2   => PARTIAL_L( 9 ),
                  T_in  => T_ARR( 9 ),
                  T_out => T_ARR( 10 ),
                  SUMr  => P_out( 16 ) );

ADDA : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 18 ),
                  SD2   => PARTIAL_L( 10 ),
                  T_in  => T_ARR( 10 ),
                  T_out => T_ARR( 11 ),
                  SUMr  => P_out( 17 ) );

ADDB : ADDER_1
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( SD1   => PARTIAL_H( 19 ),
                  SD2   => PARTIAL_L( 11 ),
                  T_in  => T_ARR( 11 ),
                  T_out => T_ARR( 12 ),
                  SUMr  => P_out( 18 ) );

ADDC : ADDER_1
```

```
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 20 ),
                   SD2   => PARTIAL_L( 12 ),
                   T_in  => T_ARR( 12 ),
                   T_out => T_ARR( 13 ),
                   SUMr  => P_out( 19 ) );

    ADDD : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 21 ),
                   SD2   => PARTIAL_L( 13 ),
                   T_in  => T_ARR( 13 ),
                   T_out => T_ARR( 14 ),
                   SUMr  => P_out( 20 ) );

    ADDE : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 22 ),
                   SD2   => PARTIAL_L( 14 ),
                   T_in  => T_ARR( 14 ),
                   T_out => T_ARR( 15 ),
                   SUMr  => P_out( 21 ) );

    ADDF : ADDER_1
        generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
        port map ( SD1   => PARTIAL_H( 23 ),
                   SD2   => PARTIAL_L( 15 ),
                   T_in  => T_ARR( 15 ),
                   T_out => T_ARR( 16 ),
                   SUMr  => P_out( 22 ) );

    P_out( 23 ) <= T_ARR( 16 );
    P_out( 24 ) <= PARTIAL_L( 16 );
    P_out( 25 ) <= PARTIAL_L( 17 );
    P_out( 26 ) <= PARTIAL_L( 18 );
    P_out( 27 ) <= PARTIAL_L( 19 );
    P_out( 28 ) <= PARTIAL_L( 20 );
    P_out( 29 ) <= PARTIAL_L( 21 );
    P_out( 30 ) <= PARTIAL_L( 22 );
    P_out( 31 ) <= PARTIAL_L( 23 );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SD_MULT is

   generic ( TECHNOLOGY_SCALE : real := 1.0 );
   port ( SD_A   : in SD_NUMBER;
          SD_B   : in SD_NUMBER;
          SD_out : out PARTIAL_P ( 0 to 31 ) );

end SD_MULT;

use work.SD_DEFINITIONS.all;
architecture Structural of SD_MULT is

   component MULT_BLOCK
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( DIGIT_C : in SD_DIGIT;
             SD_NUMB : in SD_NUMBER;
             RESULT  : out PARTIAL_P ( 0 to 16));
   end component;

   component SL2_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( PARTIAL_H : in PARTIAL_P ( 0 to 16 );
             PARTIAL_L : in PARTIAL_P ( 0 to 16 );
             P_out     : out PARTIAL_P ( 0 to 17 ));
   end component;

   component SL3_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( PARTIAL_H : in PARTIAL_P ( 0 to 17 );
             PARTIAL_L : in PARTIAL_P ( 0 to 17 );
             P_out     : out PARTIAL_P ( 0 to 19 ));
   end component;

   component SL4_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( PARTIAL_H : in PARTIAL_P ( 0 to 19 );
             PARTIAL_L : in PARTIAL_P ( 0 to 19 );
             P_out     : out PARTIAL_P ( 0 to 23 ));
   end component;

   component SL5_ADDER
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( PARTIAL_H : in PARTIAL_P ( 0 to 23 );
             PARTIAL_L : in PARTIAL_P ( 0 to 23 );
```

```
            P_out      : out PARTIAL_P ( 0 to 31 ));
    end component;

    for all : MULT_BLOCK use entity work.MULT_BLOCK(Structural);
    for all : SL2_ADDER use entity work.SL2_ADDER(Structural);
    for all : SL3_ADDER use entity work.SL3_ADDER(Structural);
    for all : SL4_ADDER use entity work.SL4_ADDER(Structural);
    for all : SL5_ADDER use entity work.SL5_ADDER(Structural);

    type PL12 is array ( 0 to 15 ) of PARTIAL_P( 0 to 16 );
    type PL23 is array ( 0 to 7 ) of PARTIAL_P( 0 to 17 );
    type PL34 is array ( 0 to 3 ) of PARTIAL_P( 0 to 19 );
    type PL45 is array ( 0 to 1 ) of PARTIAL_P( 0 to 23 );

    signal PARTIAL_1 : PL12;
    signal PARTIAL_2 : PL23;
    signal PARTIAL_3 : PL34;
    signal PARTIAL_4 : PL45;

begin

    MU00 : MULT_BLOCK
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( DIGIT_C  => SD_A( 0 ),
                      SD_NUMB  => SD_B,
                      RESULT   => PARTIAL_1( 0 ) );

    MU01 : MULT_BLOCK
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( DIGIT_C  => SD_A( 1 ),
                      SD_NUMB  => SD_B,
                      RESULT   => PARTIAL_1( 1 ) );

    MU02 : MULT_BLOCK
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( DIGIT_C  => SD_A( 2 ),
                      SD_NUMB  => SD_B,
                      RESULT   => PARTIAL_1( 2 ) );

    MU03 : MULT_BLOCK
           generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
           port map ( DIGIT_C  => SD_A( 3 ),
                      SD_NUMB  => SD_B,
                      RESULT   => PARTIAL_1( 3 ) );
```

```
MU04 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 4 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 4 ) );

MU05 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 5 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 5 ) );

MU06 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 6 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 6 ) );

MU07 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 7 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 7 ) );

MU08 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 8 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 8 ) );

MU09 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 9 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 9 ) );

MU10 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 10 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 10 ) );

MU11 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 11 ),
```

```
                SD_NUMB   => SD_B,
                RESULT    => PARTIAL_1( 11 ) );

MU12 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 12 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 12 ) );

MU13 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 13 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 13 ) );

MU14 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 14 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 14 ) );

MU15 : MULT_BLOCK
       generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
       port map ( DIGIT_C  => SD_A( 15 ),
                  SD_NUMB  => SD_B,
                  RESULT   => PARTIAL_1( 15 ) );

A10 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 0 ),
                 PARTIAL_L => PARTIAL_1( 1 ),
                 P_out     => PARTIAL_2( 0 ) );

A11 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 2 ),
                 PARTIAL_L => PARTIAL_1( 3 ),
                 P_out     => PARTIAL_2( 1 ) );

A12 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 4 ),
                 PARTIAL_L => PARTIAL_1( 5 ),
                 P_out     => PARTIAL_2( 2 ) );
```

```
A13 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 6 ),
                 PARTIAL_L => PARTIAL_1( 7 ),
                 P_out     => PARTIAL_2( 3 ) );

A14 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 8 ),
                 PARTIAL_L => PARTIAL_1( 9 ),
                 P_out     => PARTIAL_2( 4 ) );

A15 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 10 ),
                 PARTIAL_L => PARTIAL_1( 11 ),
                 P_out     => PARTIAL_2( 5 ) );

A16 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 12 ),
                 PARTIAL_L => PARTIAL_1( 13 ),
                 P_out     => PARTIAL_2( 6 ) );

A17 : SL2_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_1( 14 ),
                 PARTIAL_L => PARTIAL_1( 15 ),
                 P_out     => PARTIAL_2( 7 ) );

A20 : SL3_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_2( 0 ),
                 PARTIAL_L => PARTIAL_2( 1 ),
                 P_out     => PARTIAL_3( 0 ) );

A21 : SL3_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_2( 2 ),
                 PARTIAL_L => PARTIAL_2( 3 ),
                 P_out     => PARTIAL_3( 1 ) );

A22 : SL3_ADDER
      generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
      port map ( PARTIAL_H => PARTIAL_2( 4 ),
```

```
                    PARTIAL_L => PARTIAL_2( 5 ),
                    P_out     => PARTIAL_3( 2 ) );

    A23 : SL3_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( PARTIAL_H => PARTIAL_2( 6 ),
                     PARTIAL_L => PARTIAL_2( 7 ),
                     P_out     => PARTIAL_3( 3 ) );

    A30 : SL4_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( PARTIAL_H => PARTIAL_3( 0 ),
                     PARTIAL_L => PARTIAL_3( 1 ),
                     P_out     => PARTIAL_4( 0 ) );

    A31 : SL4_ADDER
          generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
          port map ( PARTIAL_H => PARTIAL_3( 2 ),
                     PARTIAL_L => PARTIAL_3( 3 ),
                     P_out     => PARTIAL_4( 1 ) );


    A4 : SL5_ADDER
         generic map ( TECHNOLOGY_SCALE => TECHNOLOGY_SCALE )
         port map ( PARTIAL_H => PARTIAL_4( 0 ),
                    PARTIAL_L => PARTIAL_4( 1 ),
                    P_out     => SD_out );

end Structural;
```

```
use work.SD_DEFINITIONS.all;
entity SD_MULT_TB is
end SD_MULT_TB;

use work.SD_DEFINITIONS.all;
use work.TB_PACKAGE.all;
architecture TEST_SD_MULT of SD_MULT_TB is

   component SD_MULT
      generic ( TECHNOLOGY_SCALE : real := 1.0 );
      port ( SD_A   : in SD_NUMBER;
             SD_B   : in SD_NUMBER;
             SD_out : out PARTIAL_P ( 0 to 31 ) );
   end component;

   for all : SD_MULT use entity work.SD_MULT(Structural);

   signal NUMBER_A, NUMBER_B : SD_NUMBER;
   signal RESULT : PARTIAL_P ( 0 to 31 );
   signal A_VALUE : real := 0.0;
   signal B_VALUE : real := 0.0;
   signal R_VALUE : real := 0.0;
   alias RESULT_H : PARTIAL_P ( 0 to 15 ) is RESULT( 0 to 15);
   alias RESULT_L : PARTIAL_P ( 0 to 15 ) is RESULT( 16 to 31 );
   alias SD_RESULT : PARTIAL_P ( 0 to 15 ) is RESULT( 1 to 16 );

begin

   M10 : SD_MULT
         generic map ( TECHNOLOGY_SCALE => 1.0 )
         port map ( SD_A    => NUMBER_A,
                    SD_B    => NUMBER_B,
                    SD_out  => RESULT );

   NUMBER_A <= SD_MAKE( A_VALUE );
   NUMBER_B <= SD_MAKE( B_VALUE );

   A_VALUE <= 1.0 after 200 ns, 0.5 after 300 ns, -0.50 after 500 ns,
              -1.0 after 700 ns, 0.9 after 900 ns, 0.99 after 1100 ns;
   B_VALUE <= 1.0 after 100 ns, 0.5 after 400 ns, -0.50 after 600 ns,
              0.1 after 800 ns, 0.9 after 1000 ns, 0.99 after 1200 ns,
              -1.0 after 1300 ns;
   R_VALUE <= SD_TO_REAL(SD_RESULT);

end TEST_SD_MULT;
```

Multiplier Unit report"

Vhdl Simulation Report

Report Name: Multiplier Unit report"
Kernel Library Name: <<PETERSON>>TEST_SD_MULT
Kernel Creation Date: APR-13-1989
Kernel Creation Time: 10:25:39
Run Identifer: 1
Run Date: APR-13-1989
Run Time: 10:25:39

Report Control Language File: mult_report.rcl
Report Output File : mult_report.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report MULT_report is
begin
   Report_name is "Multiplier Unit report";
   Page_width is 80;
   Page_length is 50;
   Signal_format is vertical;
   Sample_signals by_event in ns;
   Select_signal   : A_VALUE;
   Select_signal   : B_VALUE;
   Select_signal   : R_VALUE;
end MULT_report;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by event (' ' indicates no event)

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A - V A L U E	B - V A L U E	R - V A L U E
0	0.000000E+00	0.000000E+00	0.000000E+00
100		1.000000E+00	
200	1.000000E+00		
234*			
+3			1.000000E+00
300	5.000000E-01		
335*			
+3			0.000000E+00
339			
+3			**********
340*			
+3			5.000000E-01
400		5.000000E-01	
439			
+3			1.000000E+00
440*			
+3			0.000000E+00
442*			
+3			2.500000E-01
500	**********		
542*			
+3			**********
600		**********	
642*			
+3			2.500000E-01
700	**********		
739			
+3			**********
740*			
+3			7.500000E-01
742*			

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A - V A L U E	B - V A L U E	R - V A L U E
+3			5.000000E-01
800		1.000000E-01	
839			
+3			8.750000E-01
840*			
+3			**********
843*			
+2			**********
848*			
+2			**********
853*			
+1			**********
858*			
+1			**********
900	9.000000E-01		
939			
+3			1.500000E-01
940*			
+3			8.750000E-02
947*			
+1			8.750000E-02
+2			9.142151E-02
950			
+1			9.142151E-02
+2			8.977165E-02
951*			
+2			9.001579E-02
953*			
+1			9.001579E-02
+2			9.000053E-02
954*			

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A - V A L U E	B - V A L U E	R - V A L U E
+1			9.000006E-02
957*			
+1			9.000006E-02
958*			
+1			9.000000E-02
959*			
+1			9.000000E-02
961			
+1			9.000000E-02
962*			
+1			9.000000E-02
963*			
+1			9.000000E-02
1000		9.000000E-01	
1034*			
+3			1.090000E+00
1039			
+3			5.900000E-01
1040*			
+3			8.400000E-01
1043*			
+1			8.400000E-01
+2			8.400610E-01
1048*			
+1			8.400610E-01
+2			8.088110E-01
1050			
+1			8.088110E-01
+2			8.090275E-01
1052*			
+1			8.090275E-01

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A	B	R
	-	-	-
	V	V	V
	A	A	A
	L	L	L
	U	U	U
	E	E	E
+2			8.100041E-01
1053*			
+1			8.100041E-01
+2			8.100003E-01
1054*			
+1			8.100003E-01
1058*			
+1			8.100000E-01
1059*			
+1			8.100000E-01
1061			
+1			8.100000E-01
1063*			
+1			8.100000E-01
1100	9.900000E-01		
1139			
+3			9.350000E-01
1143*			
+3			8.725000E-01
1145*			
+2			8.726678E-01
1146*			
+2			8.724237E-01
1147*			
+2			9.036737E-01
1148*			
+2			8.919550E-01
1150			
+1			8.919550E-01
+2			8.919464E-01
1152*			

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A - V A L U E	B - V A L U E	R - V A L U E
+2			8.909698E-01
1153*			
+1			8.909698E-01
+2			8.910003E-01
1154*			
+1			8.910003E-01
+2			8.909994E-01
1156*			
+1			8.909994E-01
1158*			
+1			8.910000E-01
1159*			
+1			8.910000E-01
1161			
+1			8.910000E-01
1162*			
+1			8.910000E-01
1162*			
+1			8.910000E-01
1163*			
+1			8.910000E-01
1164*			
+1			8.910000E-01
1200		9.900000E-01	
1239			
+3			1.016000E+00
1243*			
+2			9.808438E-01
1248*			
+1			9.808438E-01
+2			9.808476E-01

Multiplier Unit report"

TIME	SIGNAL NAMES		
(NS)	A	B	R
	-	-	-
	V	V	V
	A	A	A
	L	L	L
	U	U	U
	E	E	E
1250			
+2			9.810764E-01
1252*			
+2			9.800999E-01
1253*			
+1			9.800999E-01
1258*			
+1			9.801000E-01
1259*			
+1			9.801000E-01
1261			
+1			9.801000E-01
1262*			
+1			9.801000E-01
1263*			
+1			9.801000E-01
1300		**********	
1334*			
+3			**********
1343*			
+1			**********
+2			**********
1348*			
+2			**********
1350			
+2			**********
1351*			
+2			**********
1353*			
+1			**********
1354*			

```
   +1  |                                                  ***********
1358*  |
   +1  |                                                  ***********
1359*  |
   +1  |                                                  ***********
 1361  |
   +1  |                                                  ***********
```

Bibliography

1. Bailey, Mickey J. *High Speed Transcendental Elementary Function Architecture in Support of the Vector Wave Equation (VWE).* MS Thesis, AFIT/GE/ENG/87D-3. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987.
2. Lyusternik, L. A. and others. *Handbook for Computing Elementary Functions.* Oxford: Pergamon Press, 1965.
3. Snyder, M. A. *Chebyshev Methods in Numerical Approximation.* Englewood Cliffs: Prentice-Hall, Inc., 1966.
4. Cosnard, M. and others. *The FELIN Arithmetic Coprocessor Chip,* Proceedings on the Eighth Symposium on Computer Arithmetic. 107-112. Washington: IEEE, 1987.
5. Hwang, Kai and others. *Evaluating Elementary Functions with Chebyshev Polynomials on Pipeline Nets,* Proceedings on the Eighth Symposium on Computer Arithmetic. 121-128. Washington: IEEE, 1987.
6. *IEEE Standard 754 for Binary Floating-Point Arithmetic.* New York: IEEE Press, 1985.
7. Swokowski, E. W. *Calculus with Analytic Geometry.* Boston: Prindle, Weber, and Schmidt, 1979.
8. Purcell, E. J. and Varberg, D. *Calculus with Analytic Geometry.* Englewood Cliffs: Prentice-Hall, Inc., 1984.
9. Hwang, Kai *Computer Arithmetic Principles, Architecture, and Design,* New York: John Wiley and Sons, Inc., 1979.
10. Avizienis, Algirdas *Redundancy in Number Representation as an Aspect of Computational Complexity of Arithmetic Functions,* IEEE Symposium on Computer Arithmetic, 87-89. Washington: IEEE, 1980.
11. Avizienis, Algirdas *Arithmetic Microsystems for the Synthesis of Function Generators,* Proceedings of the IEEE, 1910-1920 Washington: IEEE, 1966.
12. Avizienis, Algirdas *Signed-Digit Number Representation for Fast Parallel Arithmetic,* IRE Transactions on Electronic Computers, Washington: IRE 1966.
13. Chow, Chaterine *A Variable Precision Processor Module,* Ph. D. Disertation, Department of Computer Science, University of Illinois Urbana-Champaignm 1980.
14. Robertson, James E. *Design of the Cominational Logic for a Radix 16 Digit Slice for a Variable Precision Processor Module,* IEEE International Confrence on Computer Design : VLSI in Computers, 696-699. Washington: IEEE, 1983.
15. Robertson, James E. *A Systematic Approach to the Design of Structures for Arithmetic,* IEEE Symposium on Computer Arithmetic, 35-41. Washington: IEEE, 1981.

Vita

Captain Robert A. Peterson [redacted] American River Junior College for one year prior to enlisting in the Marine Corps. As an enlisted member, he performed maintenance and quality assurance duties on CH-46 helicopters. After release from the Marine Corps, he attended the University of California, Sacramento were he graduated in 1983 with a BSEE degree. After receiving a commission through Officers Training School, he was assigned to the 6520 Test Group, Edwards AFB, California were he servered as Chief Instrumentation Design Engineer for the ALCM/GLCM Chase fleet, MC-130 Modifications, and various F-15, F-16, A-10, and NKC-130 projects. He entered the School of Engineering at the Air Force Institute of Technology, Wright-Patterson AFB, Ohio in May 1987.

[redacted]