Evaluating Elementary Functions With Chebyshev Polynomials On Pipeline Nets*

Kai Hwang, H.C. Wang, and Z. Xu

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089, U.S.A.

Abstract: Fast evaluation of vector-valued elementary functions plays a vital role in many real-time applications. In this paper, we present a pipeline networking approach to designing a Chebyshev polynomial evaluator for the fast evaluation of elementary functions over a string of arguments. In particular, pipeline nets are employed to perform the preprocessing and postprocessing of various elementary functions to boost the overall system performance. Design tradeoffs are analyzed among representational accuracy, processing speed and hardware complexity.

1. Introduction

Numerical approximation of elementary functions is often performed via table lookup operations on approximated values stored in ROMs[4]. Another approach is to use CORDIC[14] or convergence transformations[2]. The third approach is to use polynomial evaluators such as Taylor's series or Chebyshev polynomials[3,14]. Agarwal et al. described a method which is based on a table lookup technique combined with polynomial approximation[1]. Tung and Avizienis[13] have presented a combinational arithmetic design for the approximation of functions. Their design is linearly pipelined. Our design, being quite different from theirs, is based on a dynamic systolization approach, called *pipeline networking*[9]. Pipeline nets are programmable and they support both linear and nonlinear interconnections among multiple arithmetic units.

The pipelined polynomial evaluator being presented can be used for fast evaluation of various elementary functions. Such functional pipelines are needed for vector processing of elementary functions, especially in real-time signal processing and control applications[17]. The quality of numerical approximation of elementary functions is assessed by *accuracy, speed,* and *cost.* Speed is determined by the rate at which the calculated value converges to the true value. It is important to have faster convergence rate in an approximation process, which leads to a saving in both computation time and hardware complexity. Accuracy is measured by the error incurred in the approximation scheme. Speed and accuracy are usually two conflicting goals and in many situations it is necessary to sacrifice one for the other. Approximation schemes based on Chebyshev polynomials are found to embrace both accuracy and efficiency.

Elementary functions are categorized into *exponential, logarithmic, trigonometric, inverse trigonometric* and other *transcendental* functions. Exponential function, $Exp(x)=e^x$, assumes valid values on the entire real axis and contains no singularity. Hyperbolic functions are defined in terms of exponential functions, such as $Cosh(x)=(e^x+e^{-x})/2$. The logarithmic function, $Ln(x)$, is the inverse of the exponential function. Trigonometric functions are either even or odd functions and contain only either even or odd powers of x in their respective power series expansions. Since trigonometric functions are periodic, inverse trigonometric functions are all multi-valued. While $Sin^{-1}(x)$ and $Cos^{-1}(x)$ are defined over the interval [-1,1], $Tan^{-1}(x)$, $Cot^{-1}(x)$, $Sec^{-1}(x)$ and $Csc^{-1}(x)$ are defined over the entire real axis. In digital arithmetic, we are mainly interested in "bounded" elementary functions that have finite values within machine representable ranges.

The rest of this paper is organized as follows. In Section 2, Chebyshev polynomials are reviewed for our purpose. Section 3 presents a hardware design for the evaluation of scalar elementary functions. A pipeline network conversion technique is developed using cut sets in Section 4. The design of an integrated system for the evaluation of vector-valued elementary functions is presented in Section 5. Section 6 elaborates on application and other related issues.

2. Chebyshev Polynomials Revisited

A *Chebyshev polynomial* $T_n(x)$ of the n-th order is defined as:

$$T_n(x)=Cos(nCos^{-1}x), \quad -1\leq x\leq 1. \tag{1}$$

* This research was supported in part by an NSF grant DMC-84-21022 and in part by an AFOSR grant 86-0008.

Obviously, $|T_n(x)| \leq 1$. It is easy to obtain the first few low-order Chebyshev polynomials, such as $T_0(x)=Cos(0)=1$ and $T_1(x)=Cos(Cos^{-1}x)=x$. For higher-order Chebyshev polynomials, the following recurrence formula is used:

$$T_{n+1}(x)=2xT_n(x)-T_{n-1}(x), \quad n \geq 1. \tag{2}$$

The above recurrence relation enables us to derive higher order polynomials recursively and provides a convenient way to sum up such polynomials, although other methods exist that allow for more direct expansion of $T_n(x)$ [3]. By applying the formula, it can also be shown that the leading coefficient of $T_n(x)$ expressed in terms of x is equal to 2^{n-1}.

The *orthogonality property* bears a strong relation with the zeros of a Chebyshev polynomial[3]. Let x_i be the zeros of $T_n(x)$. It is easy to show that $x_i=Cos((2i+1)\pi/(2n))$ for $i=0, \cdots, n-1$. For any nonnegative integers k and l with $k+l<2n$, we have:

$$\sum_{i=0}^{n} T_k(x_i)T_l(x_i) = \begin{cases} 0 & k \neq l \\ (n+1)/2 & k=l \neq 0 \\ n+1 & k=l=0 \end{cases} \tag{3}$$

Chebyshev polynomials also satisfy the *minimax property* [3]. Suppose $P_n(x)$ is defined over the interval [-1,1] as a polynomial of order n with leading coefficient being 1. Then $\max|P_n(x)| \geq 2^{1-n}$. Moreover, the equality holds, if and only if $P_n(x) = 2^{1-n}T_n(x)$.

Chebyshev polynomials have been used in both polynomial and rational numerical approximations. Chebyshev interpolation technique will be used in the ensuing discussion. Consider a function $f(x)$ defined in the interval [-1,1] that is to be approximated by a polynomial $P_n(x)$ of degree $\leq n$. One way to determine $P_n(x)$ is to choose a set of $n+1$ distinct points $x_0, x_1, x_2, \cdots x_n$, at which $f(x)$ and $P_n(x)$ have the same values. These nodes constitute a net. By Lagrange interpolation theorem, $P_n(x)$ is uniquely determined by the x_i's. Furthermore, if $f(x)$ has a bounded $(n+1)$-st derivative $f^{(n+1)}(x)$ in the range [-1,1], then the absolute error between $f(x)$ and $P_n(x)$ can be written as:

$$|P_n(x)-f(x)| = |\pi(x)f^{(n+1)}(\xi)/(n+1)!|, \tag{4}$$

where $\pi(x)=\prod_{i=0}^{n}(x-x_i)$ is an $(n+1)$-order polynomial with leading coefficient equal to 1, and $\xi \in [-1,1]$ is a number dependent on x. The error (remainder) is zero at the selected points where $f(x)$ and $P_n(x)$ coincide. We wish to minimize the absolute error. By the minimax property, the magnitude of $\pi(x)$ is minimized, if and only if it equals $2^{-n}T_{n+1}(x)$. $\pi(x)$ is made equal to $2^{-n}T_{n+1}(x)$ by taking x_i's to be the zeros of $T_{n+1}(x)$; that is, $x_i=Cos((2i+1)\pi/(2(n+1)))$, for $i=0, \cdots, n$. Then we have:

$$|P_n(x)-f(x)| = |T_{n+1}(x)f^{(n+1)}(\xi)/(2^n(n+1)!)| \tag{5}$$

Assume the absolute value of $f^{(n+1)}(x)$ in [-1,1] is upper bounded by M. Then the absolute error incurred in approximating $f(x)$ by $P_n(x)$ in [-1,1] is bounded by the following relation:

$$|f(x)-P_n(x)| \leq M/(2^n(n+1)!) \tag{6}$$

From this, the bound on relative error can be derived. In a truncated Chebyshev series, $P_n(x)=1/2\ c_0+c_1T_1(x)+\cdots+c_nT_n(x)$, the coefficients, c_k's, are obtained using the orthogonality property as follows:

$$c_k=2\sum_{i=0}^{n} f(x_i)T_k(x_i)/(n+1), \quad 0 \leq k \leq n, \tag{7}$$

where x_i's are the zeros of $T_{n+1}(x)$.

Chebyshev interpolation possesses several attractive features. In a truncated Taylor series of order n, the error is roughly proportional to $1/(n+1)!$, whereas the error incurred by Chebyshev interpolation is asymptotically $1/2^n(n+1)!$. Another distinction is that Chebyshev interpolation tends to distribute the errors evenly across the entire interval; whereas the error grows monotonically in a truncated Taylor series.

Chebyshev approximation method can be applied to functions defined in any arbitrary interval [a,b]. A simple linear transformation will convert a variable, y, in the interval [a,b] to a corresponding variable, x, in the interval [-1,1]. The transformation is specified by the relation $y=(b-a)x/2+(b+a)/2$ or $x=(2y-(b+a))/(b-a)$. The error incurred in using Chebyshev interpolation is found to be $2M((b-a)/4)^{n+1}/(n+1)!$ for interval [a,b]. Chebyshev polynomials defined over [0,1] are called *shifted* Chebyshev polynomials, denoted as $T^*(x)=T(2x-1)$.

3. Evaluation of Scalar Elementary Functions

A scalar function evaluator using Chebyshev approximation consists of three components as illustrated in Fig. 1. The *preprocessing unit* performs range reduction on the incoming data arguments; the *Chebyshev approximator* evaluates the truncated Chebyshev series and the *postprocessing unit* restores the function values with respect to the original arguments. The preprocessing and postprocessing tasks are complementary and closely interrelated. They are also function dependent.

The primary objective of range reduction is to reduce the number of terms in the approximating Chebyshev series. This is very important for the method to be practical. By performing range reduction, we usually get a sharper contraction of the Chebyshev coefficients

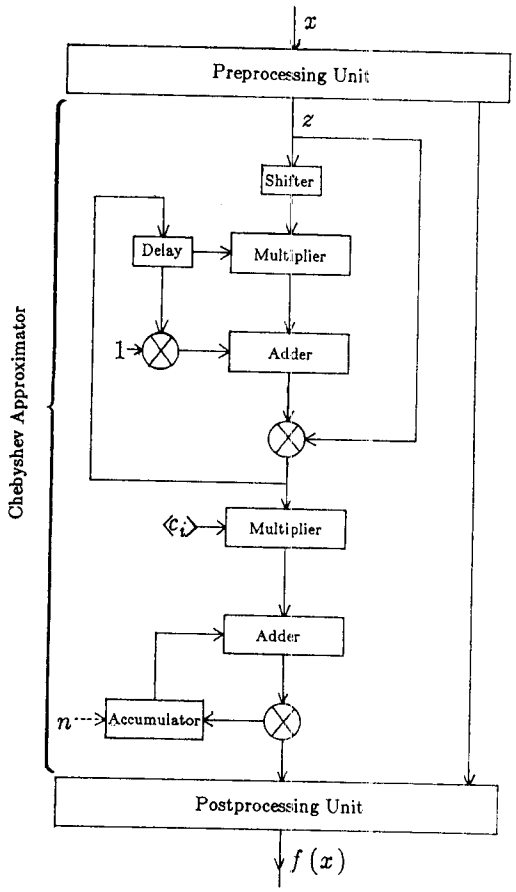


Figure 1. A scalar evaluator of elementary functions

and thus faster convergence rate. This means equivalently that the number of terms needed to achieve a prescribed precision is diminished. Another motivation for range reduction is to avoid singularity in a given interval. For example, one of the singular points of $Tan(x)$ occurs at $\pi/2$ and it would be very difficult to evaluate the function in a range which contains the singular point.

Range reduction is aided by taking advantage of such properties as *periodicity*, *symmetry*, and *recurrence relations* of elementary functions. This is clarified below with an example. Typically, a floating point number is represented by the triple $<s,m,e>$ = < sign, mantissa, exponent> with an implicit base of 2. We adopt the convention that for negative numbers $s=1$ and for positive numbers $s=0$. Also, the sign component of a number x is referred as x_s, and similarly for magnitude and exponent components. The program graph in Fig. 2 illustrates a systematic way to reduce the range of x in $y=Sin(x)$. By the intrinsic property of sine function, x can be any machine representable number. The reduction procedure consists of the following four major steps:

Step 1. Compute $u=(2/\pi)x$.

Step 2. Compute $v=u-4\lfloor (u+1)/4 \rfloor$

Step 3. Set $z=v$ if $v\leq 1$; set $z=2-v$ otherwise.

Step 4. Using $Sin(-z)=-Sin(z)$ to reduce the range of z to [0,1]. Set the sign bit $y_s=1$ if $z<0$; set $y_s=0$ otherwise.

After the above range reduction, $Sin(x)$ is evaluated as $Sin(\pi/2)z$, which is approximated by shifted Chebyshev polynomials. The sign bit y_s is sent to the postprocessing unit which ensures that the final result is correct in sign with respect to the quadrant the original argument x lies in. Basically, range reduction in this case amounts to folding the entire real axis into a relatively small interval by exploiting the periodicity property of the function in question.

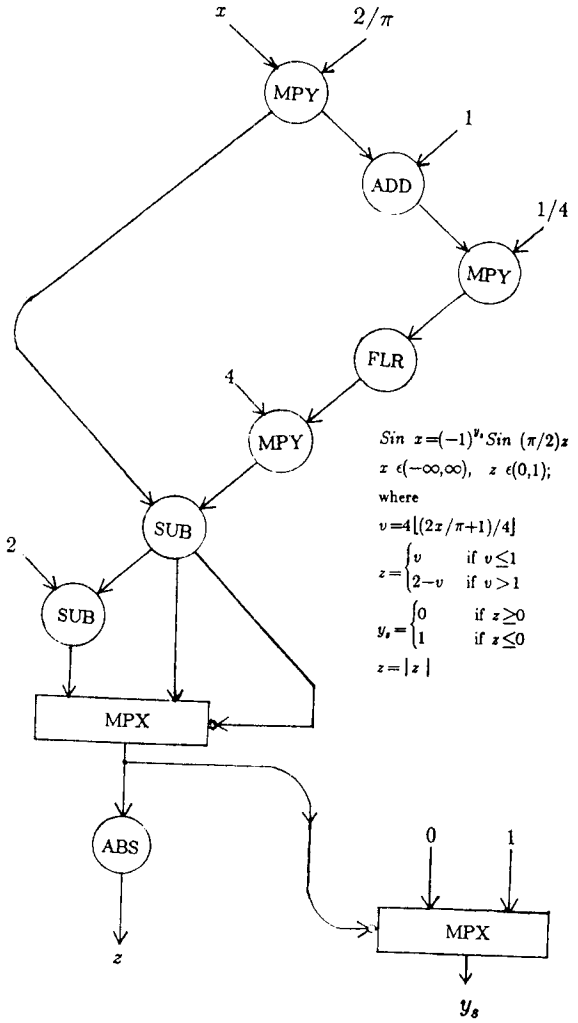


Figure 2. Program graph showing the preprocessing of the Sin(x) function

The number of terms used in the truncated Chebyshev series is dictated by the accuracy demand in the elementary functions being approximated. The IEEE floating-point standard [10] has four formats with mantissa lengths of 24, 32, 53 and 64 bits. These are recommended for *single, single extended, double and double extended* precisions, respectively. The numbers of terms needed for several representative functions are summarized in Table 1. Relative error is used as the metric as it is more relevant. Of course, the value n also depends on the interval of approximation. After n is determined, the coefficients can be precalculated for various functions and stored in ROMs for later use. We recommend that n =4, 5, 8, and 10 approximation terms be used for the mantissa lengths of 24, 32, 53, and 64 bits, respectively.

Table 1. Number of Terms Needed in Approximating Various Elementary Functions.

Function	Range	Mantissa length (in bits)			
		24	32	53	64
e^x	[0,1/16]	3	4	6	8
Ln x	[1/2,1]	3	4	6	8
Sin x	[0,π/2]	4	5	8	9
Cos x	[0,π/2]	4	5	8	9
Tan x	[0,π/8]	4	5	8	10
$Tan^{-1}x$	[0,*Tan* π/12]	3	5	8	10

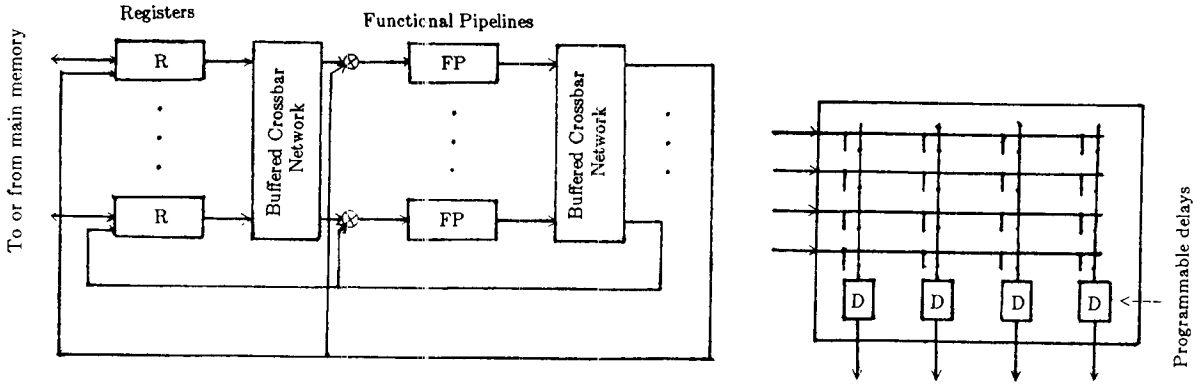
The Chebyshev approximator shown in the middle portion realizes the recurrence relation to generate Chebyshev polynomials of increasing orders, which are then summed up iteratively in the adder-accumulator pair. The value n is loaded into a counter and subsequently decremented until it becomes zero. For evaluation of scalar functions, this unit often appears as a coprocessor attached to a processor chip.

4. Pipeline Network Conversion Technique

When vector elementary functions are to be evaluated, the scalar function evaluator does not provide satisfactory performance. In this section, the concept of pipeline networking is introduced together with the discussion of a network conversion technique. The method will be used in the design of a pipelined elementary function evaluator.

Figure 3 shows the basic structure of a *pipeline net.* Multiple functional pipelines are interconnected through two crossbar networks to form a pipeline net (Fig. 3.a). The pipelines are multifunctional; that is, different arithmetic or logical operations can be performed at different times. The registers are used to hold operands and intermediate or final results. The buffered crossbar networks are used to provide dynamic connecting paths among the functional pipelines and the data registers. Programmable delays are inserted at all crosspoints in the crossbar network under instruction control (Fig. 3.b). Pipeline nets can be viewed as a programmable systolic array, which can be dynamically reconfigured to evaluate different algorithms. The reconfigurability of a pipeline net provides the necessary flexibility in implementing most Livermore loops, matrix algebra, complex arithmetic, polynomial evaluation, and signal/image processing algorithms as detailed in [8].

To start with, a computation process is represented as a *program graph* G = <V,E,f_1,f_2>, where V is the



(a) Schematic of a pipeline net

(b)The structure of 4x4 buffered crossbar network

Figure 3. The architecture of a pipeline net

set of nodes (operators), E is the set of directed arcs showing the data dependency relationships among the operators, and f_1 and f_2 are two mapping functions that associate nonnegative delays with arcs and nodes, respectively. A program graph in which each node has a delay no greater than k is called a $k-graph$. Thus, a 0-graph is a graph where each node has a delay of 0. Data are input to the program graph via the node v_{in} and results are output via v_{out}. The interval between successive data inputs is called data spacing. A node delay of t means that the function pipeline which carries out the operation is composed of t pipeline stages, whereas delays on arcs are inserted to ensure that data arrive at the individual function pipelines in a systolic fashion.

A *cut set* of a program graph is the minimum set of arcs, the removal of which separates the graph into two subgraphs, with the left one containing v_{in} and the right subgraph containing v_{out}. Rightbound arcs are those running from the left subgraph to the right subgraph and those running in the reverse direction are leftbound. A series of operations can be applied to a program graph to translate it into an equivalent graph- a graph that delivers exactly the same output when given the same input. Two equivalent graphs may have different arc delays and node delays. Listed below are four graph transformations which facilitate the derivation of equivalent graphs:

(1) Shifting several units of delays from a node to all incoming arcs or all outgoing arcs, or vice versa;

(2) Multiplying all arc and node delays and data spacing by the same positive factor;

(3) For any cut set, shifting the same amount of delays from all rightbound arcs to all leftbound arcs, or vice versa;

(4) Splitting any node in a 0-graph into a cascade of 0-delay nodes connected by 0-delay arcs.

Intuitively, each of the above graph transformations involves moving or scaling delays on the arcs or nodes while preserving the data dependency relationships in the original program graph.

Network conversion technique refers to a method which makes use of the above graph transformations to convert a given program graph into a pipeline net configuration corresponding to a k-graph. This is illustrated below by an example. Consider the program graph in Fig. 4.a, with delays associated with each node and arc. We want to convert it into a 4-graph with delays 2, 3, 3, 4 for nodes $v1, v2, v3$ and $v4$, respectively.

The first step is to delete some of the arcs from the original graph to obtain an acyclic graph, which is then topologically sorted to get a linear ordering of the nodes (Fig. 4.b). After that, those arcs previously removed are reattached. Node delays are moved to incoming arcs using transformation (1) to obtain a 0-graph (Fig. 4.c). Cut sets are drawn between neighboring nodes in the 0-graph. Transformation (2) is used as appropriate to scale up the delays on each arc by a factor of 2 (Fig. 4.d). Application of transformation (3) to the successive cut sets makes the delays of all incoming arcs at least as large as the required node delays. Finally, delays on all incoming arcs of a node are shifted to the node by the designated amount. The result is a 4-graph (Fig. 4.e), from which the pipeline net is constructed (Fig. 4.f). The four pipelines, corresponding to $v1$ through $v4$, have 2, 3, 3, and 4 stages, respectively. Note that the arc delays among the pipelines are different from those shown in Fig. 4.a. However, the resulting pipeline net performs exactly what the program graph dictates.

5. Evaluation of Vector Elementary Functions

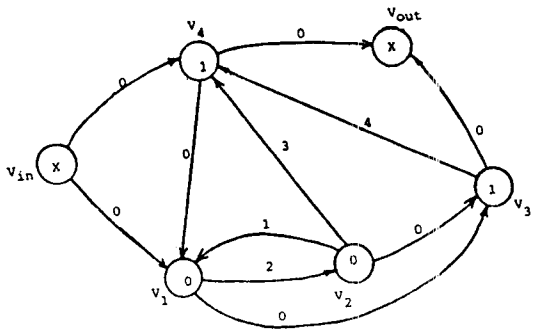
In this section, an integrated system for fast evaluation of vector elementary functions is presented. This type of processing is characterized by the repeated evaluations of the same function on a large number of arguments. We first address the issue of designing the preprocessing and postprocessing units. This will be followed by a description of the design of a pipelined Chebyshev approximator.

A. *Design of the Preprocessing and Postprocessing Units*

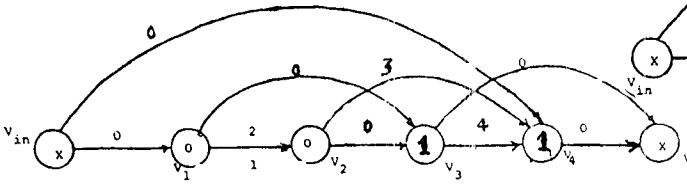
Because preprocessing and postprocessing are both function dependent, it is essential that these units be dynamically reconfigurable to best match with the different processing requirements. As discussed in the previous section, pipeline net provides very flexible interconnection patterns among functional pipelines and thus can satisfy the fundamental requirement easily.

Figure 2 shows the preprocessing program graph of $Sin(x)$. From the graph, the network conversion technique can be applied to derive the final pipeline net configuration. First, cut sets are used to divide the graph into several sections as illustrated in Fig. 5 by dashed lines. In order to achieve the highest efficiency, all the path delays across sections must be the same, counting both node delays and arc delays. This means that the operands needed for an operation will reach the node at the same time.

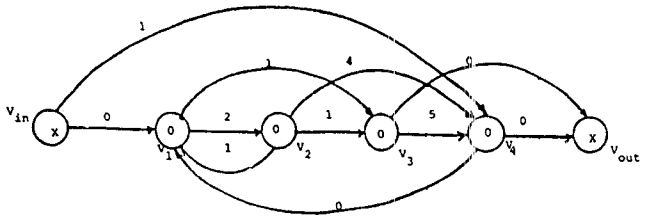
Assume that addition/subtraction, multiplication, floor and MPX operations each take 2, 4, 2, and 1 pipeline clock periods, respectively, as shown in the figure. (In fact, the multiplications in this example can be done by left or right shifting the arguments and the time



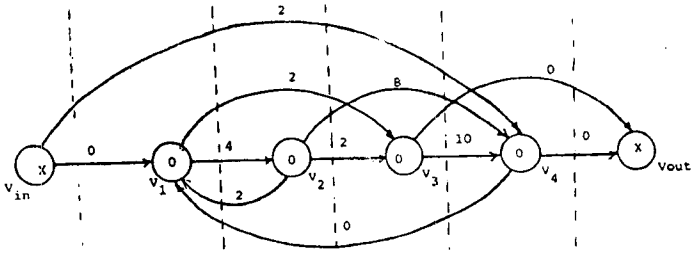
(a) A program graph is shown with delays associated with each node and arc. We want to convert it into a 4-graph with delays 2, 3, 3, 4 for nodes $v1$,$v2$,$v3$ and $v4$, respectively.



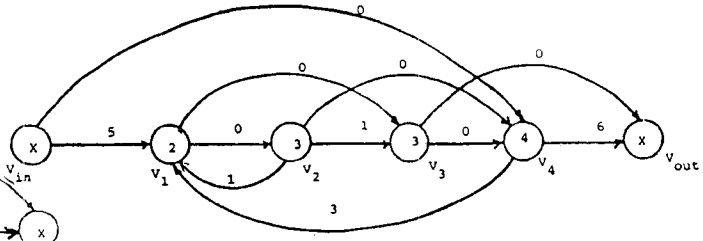
(b) Some arcs are deleted from the original graph to obtain an acyclic graph, which is then topologically sorted to get a linear ordering of the nodes.



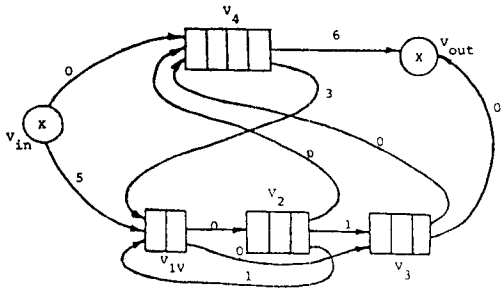
(c) Those arcs previously removed are reattached. Node delays are moved to incoming arcs using transformation (1) to obtain a 0-graph.



(d) Cut sets are drawn between neighboring nodes in the 0-graph. Transformation (2) is used to scale up the delays on each arc by a factor of 2.



(e) Application of transformation (3) to the successive cut sets makes the delays of all incoming arcs of a node at least as large as the desired node delay. Then delays on all incoming arcs of a node are shifted to the node by the designated amount to obtain a 4-graph.



(f) The final pipeline net is constructed from Fig.4.e.

Figure 4. An example illustrating the network conversion technique

required may be reduced.) Arc delays are deduced based on an "equal-path-delay" principle and inserted on the proper data paths. Fig. 6 depicts the final pipeline net configuration for the preprocessing of the $Sin(x)$ function.

The pipeline net configuration obtained in Fig. 6 is specially tailored to the preprocessing operations for the $Sin(x)$ function. As mentioned before, since the pipeline net is reconfigurable, it can also implement the preprocessing unit for other elementary functions. The reconfigurability is mainly supported by the buffered crossbar network, which dynamically establishes the connecting paths among the functional pipelines. The postprocessing unit can be similarly constructed using another pipeline net.

B. *Design of the Chebyshev Approximator*

A pipelined evaluator of the truncated Chebyshev series results from the unrolling of the DO loops implied in the summation process. In Fig. 7.a, we present a linear pipeline implementation. The design is composed of several identical segments. The functional design of each segment and the connection between segments are shown in Fig. 7.b. This linear pipeline is fixed for all elementary functions, as long as the number of pipeline segments is adequate for the prescribed accuracy. The number of pipeline segments used is equal to the number of approximating terms used, which is different for different functions. We select the largest value in each column of Table 1. For example, based on Table 1, at most 10 segments are sufficient for the listed functions

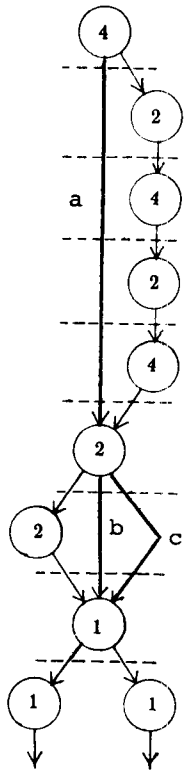


Figure 5. Cut sets for transforming the program graph in Figure 2 into a pipeline net configuration (Numbers in nodes correspond to delays and all edges have zero delays, except edge a, b, and c have 12, 2, and 2 delays respectively).

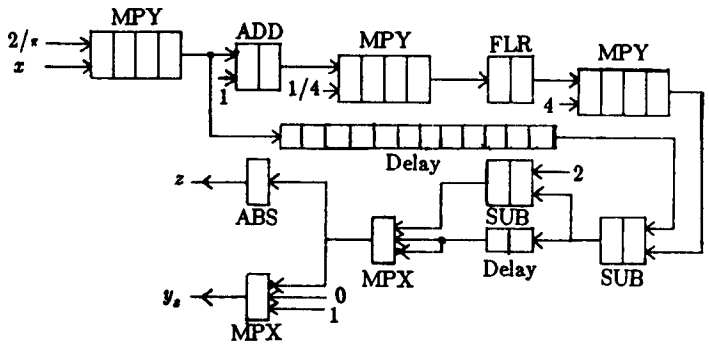


Figure 6. A pipeline net realization for the preprocessing of the Sin(x) function

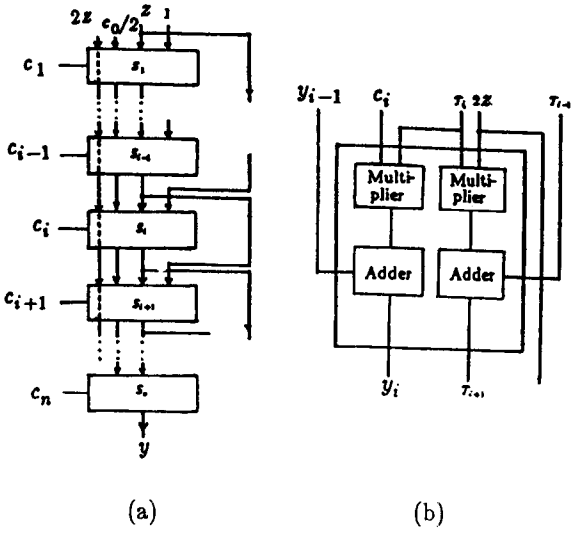


Figure 7. A pipelined Chebyshev approximator
(a) The pipeline structure
(b) Functional design of each pipeline segment

in IEEE double extended floating point data format over the specified intervals.

There are two separate data paths along the pipeline. One path is used to compute the Chebyshev terms and the other path computes their summation. Because these two paths are independent and consist of essentially the same amount of hardware (multipliers, adders/subtractors, and latches), they can proceed in parallel synchronously. While the first path is computing T_{i+1}, the second path is accumulating partial sum $y_i(z)=c_0/2+\sum_{j=1}^{i} c_j T_j(z)$ up to term T_i in segment i. In the last segment, a simple multiplier-adder is used to add $c_n T_n$ to produce the final summation.

The arithmetic operators in each segment can be implemented by several pipeline stages, just as the operators in the preprocessing and postprocessing units. Each stage works on a different data argument at the same time. Such overlapped evaluations is essential in maintaining a streamlined processing as needed in many real-time applications. By using pipeline nets for the preprocessing and postprocessing units, a balance in processing speeds is achieved among these three units. Moreover, pipeline stages in all three units can operate at the same clock rate.

As is obvious from the above discussion, the pipelined elementary function evaluator as a whole is a cascade of three pipeline conglomerates. The time required to set up the two pipeline nets for the preprocessor and postprocessor imposes some overhead in execution time. Denote the total number of pipeline stages in the overall system by M and the clock period by τ. Also let γ be the setup time for the pipeline nets and β the network delay through the buffered crossbar networks. Both γ and β are multiples of τ. For a vector of length N for

the same function to be evaluated, the throughput θ is found to be $\theta=MN/((\gamma+\beta+M+(N-1))\tau)$.

The system described above is most efficient in processing long vectors. In fact, the maximum throughput, M/τ, is obtained when $N>>\gamma+\beta+M$. This peak performance is intuitively correct, because $1/\tau$ represents the maximum throughput (frequency) of each stage and there are M stages connected in the cascade.

6. Conclusions

We have streamlined the the numerical approximation using Chebyshev polynomials and presented an innovative arithmetic pipeline design for fast evaluation of vector-valued elementary functions. The rapid convergence rate of Chebyshev polynomials appeals very much to the IEEE floating point standard. In practice, this design of the elementary function evaluator can be expanded into a multifunctional arithmetic processor, which can perform polynomial division[16], vector reduction[12], matrix algebra[7], array multiplication[5], vector compound functions[9], image processing and pattern analysis[6] and real-time applications[11].

Ever since the work of Tung and Avizienis[13], there has been very little progress made in designing pipelined Chebyshev approximators. This article presents the pipeline net approach to handling both preprocessing and postprocessing phases in evaluating various elementary functions. The pipeline nets are under instruction control to reconfigure their internal connections for various elementary functions. To save in hardware, the middle phase of Chebyshev approximation is realized with a fixed pipeline structure.

In this paper, we demonstrate the design of a vector elementary function evaluator at the logic level. Details of implementation in VLSI or other hardware techniques are not within the scope of this paper. However, it would be interesting to see the proposed logic structure be implemented with state-of-the-art VLSI electronics or even with the bistable optical gate arrays as proposed in [15].

References

[1] Agarwal, R.C., Cooley, J.W., Gustavson, F.G., Shearer, J.B., Slishman, G., and Tuckerman, B. "New Scalar and Vector Elementary Functions for the IBM System/370", *IBM J. Res. and Develop.*, vol. 30, no. 2, March, 1986.

[2] Chen, T.C. "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots", *IBM J. of Res. and Develop.*, July 1972.

[3] Fox, L. and Parker, I.B. *Chebyshev Polynomials in Numerical Analysis* , Oxford University Press, London, 1968.

[4] Hwang, K. *Computer Arithmetic: Principles, Architecture and Design,* John Wiley & Sons, New York, 1985.

[5] Hwang, K. "Global and Modular Two's Complement Array Multipliers", *IEEE Trans. Comp.*, vol. C-28, no. 4, April, 1979.

[6] Hwang, K. "VLSI Computer Arithmetic for Real-Time Image Processing", in *VLSI Electronics: Microstructure Science,* vol. 7 (Einspruch, Ed.), Academic Press, N.Y. 1984.

[7] Hwang, K. and Cheng, Y.H. "Partitioned Matrix Algorithms for VLSI Arithmetic Systems", *IEEE Trans. Comp.*, vol. C-31, no. 12, December, 1982.

[8] Hwang, K. and Xu, Z. "Pipeline Nets for Compound Vector Supercomputing", *IEEE Trans. Comp.*, Accepted to appear 1987.

[9] Hwang, K. and Xu, Z. "Multiprocessors for Evaluating Compound Arithmetic Functions", *Proc. of the 7th Symp. on Computer Arithmetic,* Urbana, Illinois, June 4-6, 1985.

[10] *IEEE Standard 754 for Binary Floating-Point Arithmetic,* IEEE Press, New York, 1985.

[11] Milutinovic, V., Lopez-Benitez, N., and Hwang, K. "A GaAs-Based Microprocessor Architecture for Real-Time Applications", *IEEE Trans. Comp.*, Accepted to appear 1987.

[12] Ni, L.M. and Hwang, K. "Vector Reduction Techniques for Arithmetic Pipelines", *IEEE Trans. Comp.*, vol. C-34, no. 5, May, 1985.

[13] Tung, C. and Avizienis, A. "Combinational Arithmetic Systems for the Approximation of Functions", *Proc. Spring Joint Computer Conference,* 1970.

[14] Valder, J.E. "The CORDIC Trigonometric Computing Techniques", *IEEE Trans. Comp.*, vol. C-9, no. 9, September, 1960.

[15] Xu, Z., Hwang, K., and Jenkins, B.K. "Opcom: An Architecture for Optical Computing Based on Pipeline Networking", *Proc. of the 20th Int'l Hawaii Conference on Systems Sciences,* Kona, Hawaii, Jan. 6-9, 1987.

[16] Zak, S.H. and Hwang, K. "Polynomial Division on Systolic Arrays", *IEEE Trans. Comp.*, vol. C-34, no. 6, June, 1985.