# Multiprocessors For Evaluating Compound Arithmetic Functions*

Kai Hwang and Zhiwei Xu

Computer Research Institute
University of Southern California
Los Angeles, CA 90089-0781

**Abstract** A dynamic network approach is proposed for designing multifunctional arithmetic processors to support *complex*, *interval*, *vector*, *matrix*, *polynomial*, and other *compound arithmetic* operations. This arithmetic-network approach is extended from the multipipleline chaining concept implemented in Cray Research supercomputers. The proposed design methodology offers a viable way of developing very powerful and flexible arithmetic multiprocessors for scientific supercomputing.

## 1. Introduction

Reviewing the history of computer arithmetic, we observe a constant trend of shifting from software to hardware implementations of various arithmetic functions. Early processors had rather a rudimentary capability of performing only bit-serial addition and shifting, while today's fast processors like FPS 164/Max and Cray X-MP [7] are equipped with multiple arithmetic pipelines which can be linearly chained together for processing various arithmetic/logic operations. With the advent of VLSI technology, the time has arrived to implement *complex,interval,matrix* and other *compound arithmetic* functions directly in hardware.

The common approach at present is to use a special hardware device for each dedicated function. For instance, the Cray X-MP has 15 dedicated functional pipelines for logic, fixed-point, floating-point, scalar, and vector operations. This approach demands high memory bandwidth and a large number of dedicated functional units. This will definitely increase the hardware cost and add to the control and scheduling overhead. With dedicated functionality, the utilization of the arithmetic processor may be low, if concurrency cannot be fully explored in user programs. Furthermore, the application flexibility is rather limited and the fault tolerance can not be enhanced with fixed hardware functions.

An alternative approach is to interconnect a number of simple, programmable arithmetic units to implement various arithmetic functions. We have proposed a reconfigurable multiprocessor architecture, called Remps [9], based on this approach. This paper describes the arithmetic aspects of the Remps computer. A new concept of *arithmetic networks* is introduced. The key idea is to set up a network of arithmetic pipelines, which will best match the data flow pattern of any given arithmetic algorithm.

*This research was supported in part by AFOSR grant 84-0385.

In this paper, we first identify major compound arithmetic functions, such as *complex functions*, *interval arithmetic*, *vector/matrix operations*, and *polynomial evaluations*. We also consider *butterfly computations* used in FFT and *arithmetic loops containing IF statements*. Special arithmetic network architecture and necessary functional features are presented. Supporting compound arithmetic functions is demonstrated with benchmark studies.

## 2. Compound Arithmetic Functions

It has been recognized that in addition to conventional arithmetic, other arithmetic types such as interval, complex, vector/matrix are also important for a computer system that supports extensive numerical computations. The arithmetic design requirements of these compound arithmetic functions via the real arithmetic operations are specified below:

A. *Complex Arithmetic*: Let $a,b,c,d$ be real numbers. A complex number is represented by a pair $(a,b) = a + jb$. The four basic arithmetic operations on complex numbers are specified below:

Complex Add/Subtract
$$(a,b)\pm(c,d)=(a\pm c,\ b\pm d)$$

Complex Multiply
$$(a,b)\times(c,d)=(ac-bd,ad+bc)$$

Complex Divide
$$(a,b)/(c,d)=((ac+bd)/(c^2+d^2),(bc-ad)/(c^2+d^2))$$

B. *Interval Arithmetic*: An interval is represented as a pair of real numbers, $(a,b)$, where $a$ is the lower bound while $b$ is the upper bound. Interval arithmetic operations are formulated as follows:



CH2146-9/85/0000/0266$01.00 © 1985 IEEE

Interval Add
$(a,b)+(c,d)=(a+c,b+d)$

Interval Subtract
$(a,b)-(c,d)=(a-d,b-c)$

Interval Multiply
$(a,b)\times(c,d)=(\min(ac,ad,bc,bd), \max(ac,ad,bc,bd))$

Interval Invert
$(a,b)^{-1}=(1,1)\times(1/a,1/b)$

Interval Divide
$(a,b)/(c,d)=(a,b)\times(c,d)^{-1}$

C. *Vector Arithmetic*: We only consider vector, matrix, and polynomial arithmetics over real numbers. The corresponding complex and interval extensions can be treated similarly, once the scalar operations of complex and interval arithmetics are defined. Consider two vectors $\mathbf{v},\mathbf{u} \in R^n$, and a scalar $a \in R$. Let $\circ \in \{+,-,\times,/\}$ be any of the four primitive arithmetic operators.

Vector Arithmetic $\quad \mathbf{u}\circ\mathbf{v}=(u_1\circ v_1, \cdots, u_n\circ v_n)$

Inner Product $\quad \mathbf{u}^T\cdot\mathbf{v}=\sum_{i=1}^{n} u_i v_i$

Outer Product $\quad a\times\mathbf{u}=(au_1, \cdots, au_n)$

D. *Matrix Arithmetic*: Let $\mathbf{A}=(a_{ij})$, $\mathbf{B}=(b_{ij})$ be two $n\times n$ matrices. Matrix arithmetic often includes:

Matrix Add/subtract $\mathbf{A}\pm\mathbf{B}=(a_{ij}\pm b_{ij})$

Matrix Multiply $\mathbf{A}\times\mathbf{B}=(\sum_{k=1}^{n} a_{ik}b_{kj})$

Matrix Inversion
(1). L-U decomposition $\mathbf{A}=\mathbf{L}\cdot\mathbf{U}$;
(2). Triangular matrix inversions, $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$;
(3). Matrix multiplication $\mathbf{A}^{-1}=\mathbf{U}^{-1}\times\mathbf{L}^{-1}$.

E. *Butterfly Computation in FFT*: Each butterfly computation has two complex input and two complex output vectors, apart from the constant coeffecients ($\sin\phi$ and $\text{con}\phi$). Eight real vectors are involved, denote by $A,B,C,D,E,F,G,H$.

```
procedure Butterfly(A,B,C,D,E,F,G,H,i,j)
parallel begin
  A(j):= E(i)+G(i)cosφ-H(i)sinφ
  B(j):= E(i)-G(i)cosφ+H(i)sinφ
  C(j):= F(i)+G(i)sinφ+H(i)cosφ
  D(j):= F(i)-G(i)sinφ+H(i)cosφ
parallel end
```

F. *Polynomial Arithmetic*: The addition/subtraction of single-variable polynomials are carried out by the corresponding vector addition/subtraction. Polynomial multiplications can be realized via FFT and vector multiplications. The value of a polynomial is evaluated by Horner's form:

$$y=a_0+a_1x+\cdots+a_nx^n$$
$$=((\cdots(a_nx+a_{n-1})x+\cdots)x+a_0$$

G. *Arithmetic Loops with IF statements*: A typical example is given below:

```
DO 10  I = 1, N
   X(I) = A(I)*C(I)-B(I)*D(I)
   IF X(I) ≤ 0.0 THEN Y(I) = A(I)*B(I)+C(I)*D(I)
   ELSE Y(I) = A(I)*B(I)-C(I)*D(I)
10 CONTINUE
```

## 3. Arithmetic Networks

Compound arithmetic functions demand a supercomputer which has the following functional capabilities:

A. *Multiprocessing*: Many compound arithmetic functions demand the concurrent executions of multiple real arithmetic operations. For example, to realize a Complex Divide, we need to coordinate the *multiply*, *add*, *subtract*, and *divide* operations.

B. *Pipelinability*: In vector and matrix arithmetic, pipelining is an economic way to implement the component computations. If we have $m$ pipelines each of $k$ stages, the maximum speedup is $km$ over a non-pipelined scalar processor.

C. *Reconfigurability*: The implementations of compound arithmetic functions do not follow a fixed interconnection pattern. A dynamic arithmetic multiprocessor must be reconfigurable to achieve different networking patterns with low overhead.

D. *Delay matching*: When two operand streams arrive at a certain arithmetic unit in the network, they may have traversed through data paths of different delays. These delays must be equalized in order to have the correct operand pairs arrived at the right place at the right time.

The concept of multiple arithmetic networks to be used in the Remps is extended from the pipeline chaining in Cray X-MP. The *Remps* is a reconfigurable multiprocessor system, as redrawn in Fig.1. A dynamic arithmetic processor in Remps consists of five major components. The $m$ Processing Elements (PEs) and the routing network constitute an arithmetic network, which is used to evaluate various arithmetic/logic operations. The allocation network interconnects the memory modules with the $m$ PEs. The controller coordinates the operations of the entire system.

The processor can reconfigure its PEs into different arithmetic networks at different times to match with different arithmetic algorithms. Pipelining is exploited at the intra-PE level (each PE is itself pipelined), the inter-PE level (the network is also pipelined), and the processor level (macropipelining between memory modules and the arithmetic network).

Mathematically, an arithmetic network is modeled by a weighted, directed graph $G(V,E,W)$, where the node set $V$ represents the PEs, the arc set $E$ represents the inter-PE connections, and $W$ is a time delay function from $V \cup E$ to integer set $I$ which specifies the delay of each PE and each connection link. Special graphic properties and constraints in making the network pipelinable are to be discussed.

Feedforward links are allowed in the network. An arithmetic network with (or without) feedbacks is denoted by a cyclic (or acyclic)graph. Almost all the compound arithmetic functions defined in section 2 can be evaluated by an acyclic arithmetic network. Other functions such as linear recursive systems may have to use a cyclic network with feedbacks. Any user job is processed in the Remps with the following sequence of operational steps:

(1). *Partitioning*: A job is partitioned into a number of tasks, each of which is executable by a proper arithmetic network. Different tasks may be assigned to one or more processor for concurrent execution.

(2). *Mapping*: The dataflow graph of a task is mapped into an arithmetic network. This includes: (1) Matching the node functions of the dataflow graph with the node functions of the arithmetic network. And (2) Necessary delays are inserted into the data paths of the network.

(3). *PE networking*: The controller checks the PE demand with the PE resource availability. The routing network is programmed to achieve a special connection pattern among the PEs. The allocation network provides necessary data paths to or from the memory.

(4). *Execution*: The sequence of data movements from the memory to the allocation network, to the arithmetic network, and then back to the memory via the allocation network, forms a macropipeline [4]. The controller supervises the flow of operands through the macropipeline.

To clarify the above discussions, let us consider an example of DO-loop computations.

```
DO 10  I = 1, n
   E(I) = [A(I)+B(I)+C(I)]/D(I)
10 CONTINUE
```

A dataflow graph is shown in Fig.2a for the above DO-loop. Assume that each PE can perform two additions simultaneously. But the reciprocation needs two cascaded PEs. The two addition nodes are executed by a single PE, while each division node is executed in three PEs. The resulting dataflow graph is demonstrated in Fig.2b. Some noncompute delays are added to the left data path in Fig.2c. Finally, the desired arithmetic network is set up as shown in Fig.2d.

Two factors must be considered in implementing the dynamic arithmetic networking concept. Programmable noncompute delays are used in the routing network so that unequal delays can be matched dynamically. The routing network must have enough connectivity. Furthermore, an arithmetic network must be *pipelinable* so that it can process an operand block for each pipeline cycle. An inter-PE link is *blocked*, if more than one data paths use the same link. An arithmetic network without feedbacks is pipelinable, if all the PEs are linear pipelines and no inter-PE link is blocked.

A global bus or a multistage interconnection network(such as the *baseline*, *omega*, or *cube* networks) may have blocking links in some applications. These networks are not suitable to be used as the routing network. We choose the crossbar switching network because it is fully-connected, non-blocking, and easy to control. Some programmable delay units are used. Such an 8-by-8 crossbar chip, called LINC, has been designed and implemented in CMU [5].

Several possible choices can be made about the PE structure; such as the INMOS transputor, the iterative pipelines used in IBM 360/91, or a linear pipeline used in Cray X-MP. Commercially available microprocessors are too slow for floating-point operations due to microcoded operations. Iterative pipelines do not support the pipelinability of the arithmetic network. Linear pipelines are ideal for our purpose. What we need are pipelined PEs which are both linear and universal in functionality.

Branching instructions may become detrimental if it is not treated properly. Instruction prefetch will not help much, since switching from one side of the branch to another often requires the set up of a different arithmetic network. To overcome this difficulty, we simply merge both sides of a branching into a single arithmetic network. Although some PEs are producing results which may not be used later, the pipelinability of the network is preserved by merging.

## 4. Pipelined PE Designs

The PEs to be used in an arithmetic network should be designed to possess the following properties:

(1). *Uniformity*: All PEs are multifunctional and identical. This will increase the utilization flexibility and fault tolerance of the dynamic processor. It also reduces the development overhead.

(2). *Flexibility*: Since all the PEs are identical, each of them is dynamically programmable to executed different arithmetic/logic operations at different times. An arithmetic network of PEs can be used to perform many compound arithmetic functions.

(3). *Linearity* : Each PE is linearly pipelined for various arithmetic operations. No feedback are allowed among the pipeline stages. This property demands that each PE generates one result per each pipeline cycle for all arithmetic operations.

(4). *Merging of Branches*: Each PE is capable of merging both sides of a branch operation into one sequence. Multi-levell branches can be merged by cascading the PEs.

In the PE design, we partition the arithmetic functions into several pipeline stages and exploit hardware sharing as much as possible , so long as it dose not violate the pipeline linearity. The resulting PE has a hardware complexity comparable to that of the *Reciprocate* pipe in Cray-1 computer. The design of a 3-input/3-output PE is shown in Fig.3. This design merges the two sides of a branch.

The *Receive* and the *Transmit* stages are the I/O stages as detailed in Fig.4. The I/O ports A, B, and C are connected to the memory via the allocation network. A', B', and C' come from the routing network. D and E are feedbacks from the *Transmit* stage of the same PE. This is needed for implementing vector inner-product. Conditional branch instructions are controlled by the sign bit and the most significant bit of the mantissa of a normalized floating-point number (i.e., to determine if x<0, or x>o, or x=0).

The two *Exponent/Logic* stages are used to perform exponential addition/subtraction and mantissa alignment. They are also used to execute some logic and shift operations. The *Normalize* stage normalizes the floating-point results. An ROM is used to store the initial guess of the reciprocal values. Several iterations of multiplications are needed to produce the final reciprocal. Other constants for evaluating transcendental functions and for FFT computation are also stored in the ROM.

The *Multiply/Add* stages are designed to support four arithmetic computations in the form of $(x\pm y)$,$(x\times y)$,$2-(x\times y\pm y)$, and $x\times y\pm y$, where x and y are real fractions. The last two operations are needed to compute $(2-A\times B)\times A$ in a Newton-Raphson iterative step for finding the reciprocal of a given fraction.

Let $A=a_0.a_1a_2\cdots a_{n-1}$, $B=b_0.b_1\cdots b_{n-1}$ be two's complement fractions. The one's complement of $A$ is denoted as $A$. Denote the double summations $\sum_{i=1}^{n-1}\sum_{j=1}^{n-1} a_ib_j2^{-i-j}$ by $D$. Its one's complement is denoted as $\overline{D}$.The following formulae are used for two's complement addition, subtraction, and multiplication [6]:

$$A+B=-a_0-b_0+\sum_{i=1}^{n-1}(a_i+b_i)\times 2^{-i}$$

$$A-B=A+B+2^{-n+1}$$

$$\begin{aligned}A\times B &= a_0b_0+D-\sum_{i=1}^{n-1}(a_0b_i+a_ib_0)2^{-i}\\ &= a_0b_0+\overline{a}_0+\overline{b}_0+\sum_{i=1}^{n-1}(\overline{a}_ib_0+a_0\overline{b}_i)2^{-i}\\ &\quad+(a_0+b_0)2^{-n+1}+\overline{D}\end{aligned}$$

Suppose $B$ is a normalized fraction and $B\neq 1/2$, then we have $1/2<|B|<1<|1/B|<2$. Let $A_0'$ be the initial approximation to $1/B$. Then we use the following intermediate expressions to evaluate $1/B$ in two's complement notation:

$$A_0'=\begin{cases}1+0.s_1\cdots s_{n-1} & B>0\\ -1+1.s_1\cdots s_{n-1} & B<0\end{cases}$$

$$A_0=\begin{cases}0.s_1\cdots s_{n-1} & B>0\\ 1.s_1\cdots s_{n-1} & B<0\end{cases}$$

$$B_0'=2-A_0'\times B$$

$$B_0'=\begin{cases}2-(A_0\times B+B) & B>0\\ 2-(A_0\times B-B) & B<0\end{cases}$$

$$A_1'=\begin{cases}A_0\times B_0'+B_0' & A_0>0\\ A_0\times B_0'-B_0' & A_0<0\end{cases}$$

The value of $A_0'$ is precomputed so that $|A_0'|>|1/B|$, and thus $A_0'\cdot B>1$. A tedious error analysis has been performed which shows that $A_i'\cdot B>1$ for all $i$, and this sequence converges to $1/B$ quadratically. The two *Multiply/Add* stages perform $2-(A\cdot B\pm B)$ and $A\cdot B\pm B$ in one Newton-Raphson iterative step. From the above discussion, we obtain:

$$A\times B\pm B=\begin{cases}D+\sum_{i=1}^{n-1} b_i2^{-i} & A>0\\ D-\sum_{i=1}^{n-1}\overline{b}_i2^{-i+1}+2^{-n+2} & A<0\end{cases}$$

$$2-(A\times B\pm B)=\begin{cases}\overline{D}+2^{-2n+2}+2^{-n+1}+\sum_{i=1}^{n-1}\overline{b}_i2^{-i} & B>0\\ \overline{D}+2^{-2n+2}+\sum_{i=1}^{n-1}b_i2^{-i+1}+\sum_{i=1}^{n-1}a_i2^{-i} & B<0\end{cases}$$

The above arithmetic equations are executed in several stages. Figure 5 shows the design of the required stages. Note that all the operations share the carry-lookahead adder. And the iterative array is shared by three operations $A \times B$, $2-(A \times B \pm B)$, and $A \times B \pm B$. Compared with the Baugh-Wooley multiplier [1], the only significant hardware increase is the carry-lookahead adder being added.

Several examples are presented below to show the versatility of the pipelined PE design. Most arithmetic/logic operations can be implemented with a single PE as demonstrated in Fig.6 for *fixed-point multiply*, *floating-point inner-product*, and a *logic operation* $x \vee (y \wedge z)$. *Complex divide* and *interval multiply* operations are realized by the circuits in Fig.7.

In Cray X-MP, the reciprocal of a number is evaluated with a Newton iterative method. To find 1/B, iterative operations $A_{i+1}=(2-A_i \times B)A_i$ are performed, where $A_0$ is the initial approximation and $A_i$ is the i-th approximation to 1/B. Each PE generates the new value $A_i$ from the old $A_{i-1}$. Two PEs are cascaded to generate $A_2$, as shown in Fig.8. More PEs could be used to increase the number of iterations and thus the precision.

The network shown in Fig.9a is for butterfly operations. The two bottom PEs executes the needed multiplications. The four top PEs perform the required additions/subtractions. The network in Fig.9b is set up to support an arithmetic loop with a single IF statement.

Dynamic arithmetic processor must be able to support matrix arithmetic. We demonstrate the matrix operations involved in solving a linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$

**Step 1. L-U decomposition** $\mathbf{A} \Rightarrow \mathbf{L} \cdot \mathbf{U}$;
**Step 2. Triangular matrix inversions** $\mathbf{L}^{-1}$, $\mathbf{U}^{-1}$;
**Step 3. Back and forward substitutions** $\mathbf{y} = \mathbf{L}^{-1} \cdot \mathbf{b}$,
$\mathbf{x} = \mathbf{U}^{-1} \cdot \mathbf{y}$.

Three arithmetic networks are required to perform the above three steps. A hexagonal array is used for L-U decomposition(Fig.10a); a triangular array for matrix inversions(Fig.10b); and a pipeline ring for back substitution(Fig.10c). In setting up the hexagonal array, noncompute delays are used to equalize the delays on various data paths.

To sum up, the proposed PEs to be used in arithmetic networking have three important functional features:

(1). All PEs are identical and universal in functionality. Each PE is a 7-stage pipeline that can be programmed to execute simultaneously two arithmetic/logic operations. The only exception is division, which must be executed by a pipeline-chain of three or more PEs.

(2). Each PE has a hardware complexity comparable with the Reciprocate pipeline in Cray-1. A 10-PE arithmetic multiprocessor can support the various arithmetic operations defined in section 2.

(3). Each PE generates at least one result per pipeline cycle. By changing the arithmetic network size, the user can trade between the accuracy and the number of PEs to be used. This tradeoff does not affect the throughput.

## 5. Performance Analysis

Compared with static arithmetic pipelines with fixed interconnection links (systolic arrays), our dynamic arithmetic processor is much more flexible from application point of view. In fact, by using a crossbar as the routing network, a dynamic processor can establish all desired interconnection patterns. However, the flexibility is obtained at the expense of two additional overheads:

(1). *The network reconfiguration overhead*: Whenever a network is to be set up, the PEs must be programmed and the routing network must be reconfigured to establish the desired connection. This overhead, denoted by $\alpha$, is often dominated by the control time of the routing network.

(2). *The network delay*: Because the PEs in an arithmetic network are connected by the routing network, data transmissions from one PE to another must experience additional network delays. the delay is $\beta$ pipeline cycles, if the routing network is clocked with $\beta$ stages.

The following parameters are used to analyze the performance of the proposed arithmetic multiprocessor. The computation consists of $N$ operand blocks($N$=1 for scalar operations). Each block will be used in $m$ arithmetic operations, which will be performed by an arithmetic network of $m$ PEs. This computation is performed by the proposed multiprocessor in the following way. First, a network of $m$ PEs , which has a critical dataflow path of length $c$, is set up in $\alpha$ pipeline cycles.

Each operand block passes through the allocation network twice for operand fetching and result storing. Because the critical path is the longest dataflow path in the network, the operand block will pass $c$+1 PEs and pass the routing network $c$ times. Suppose that each PE has $k$ pipeline stages( $k = 7$ in the proposed design), then the fill-up time of the arithmetic network, is $2\beta + (c+1)k + c\beta$. After the network is filled up, an operand block is processed for every pipeline cycle. Denote by $T_d$ the total time needed to process the $N$ operand blocks, we have

$$T_d = \alpha + [(c+1)k + (c+2)\beta] + N\text{-}1.$$

The proposed multiprocessor is compared first with a processor having a single PE and then compared with an

equivalent systolic array with $m$ PEs. For a single-PE processor, it requires at least $T_1 = k + mN - 1$ cycles, where $mN$ is the total number of arithmetic operations to be performed. The speedup of the arithmetic multi-PE network over a single-PE processor is then:

$$S_m = \frac{T_1}{T_d} = \frac{k+mN-1}{\alpha+(c+1)k+(c+2)\beta+N-1}$$

If the above computation is executed by a static systolic array, there are no reconfiguration overhead ($\alpha$=0) and no network delay ($\beta$=0) involved. We obtain the following computation time for a static systolic array:

$$T_s = (c+1)k + N - 1.$$

The throughputs of the static systolic array and of the dynamic arithmetic network are $1/T_s$ and $1/T_d$ respectively. Thus $(1/T_d)/(1/T_s)=T_s/T_d$ represents the *performance ratio* of the dynamic multi-PE network versus static systolic arrays. This performance ratio is plotted in Fig. 11 under the assumption $\alpha = 50$, $k = 7$, and $c = 15$. The curves correspond to two dynamic arrays using a crossbar routing network and a Benes network respectively.

A dynamic arithmetic processor can perform equally well as a static systolic array, if the problem size is sufficiently large. This is especially true for a moderate size network, say with 64 or less PEs. A 10-PE network can support most compound arithmetic computations. Such a processor can maintain 80% of the peak performance of a static systolic array for all the compound arithmetic functions.

It is fair to conclude that *for general-purpose applications, the flexibility in implementing many different algorithms in the proposed dynamic multiprocessor is far more important than a minor speed degration due to the added network overheads.* There is no doubt that static systolic arrays are still superior for implementing fixed algorithms. However, they become useless when the application algorithms are changed.

We analyze below the performance of a dynamic arithmetic processor with, say $m$=10 PEs. We are interested in finding the throughput of the processor in evaluating typical compound arithmetic functions. The throughput is measured as the number of floating-point operations executed divided by the total computation time $\tau \cdot T_d$, where $\tau$ is the pipeline cycle time.

The benchmark computations include a *complex vector division*, an *interval vector multiplication*, an *N-point FFT* computation, and *an arithmetic loop with a single IF statement,* as specified in section 2. The arithmetic networks to be used for performing these functions were shown in Fig. 7 and 8. We have

$$T_d=\alpha+(c+1)k+(c+2)\beta+N-1=66+12c+N$$

for the assumed values of $\alpha$=50, $\beta$=5, and $k$=7.

The following throughput expressions are obtained through simple operational analysis. For example, a *Complex Divide* needs to execute 6 multiplications, 2 divisions, and 3 addition/subtractions. Altogether there are 11 floating-point operations involved in a single complex division. An $N$-component vector division needs to perform $11N$ operations. From Fig.7a, the arithmetic network for *Complex Divide* has a critical data path of length $c$=4. Thus the total computation time is $\tau \cdot T_d=(66+12c+N)\cdot\tau=(114+N)\cdot\tau$. Thus $11N/(\tau \cdot T_d)$ gives the throughput expression.

$$\text{Throughput}(\textit{Complex Divide})=\frac{11N}{(114+N)\tau}$$

$$\text{Throughput}(\textit{Interval Multiply})=\frac{8N}{(90+N)\tau}$$

$$\text{Throughput}(\textit{FFT})=\frac{12N\log_2 N}{(156+N\log_2 N)\tau}$$

$$\text{Throughput}(\textit{IF in Loop})=\frac{7N}{(90\ \ +N)\tau}$$

The throughput performances are plotted in Fig. 12. If the pipeline cycle time is $\tau$ = 10 ns, a 10-PE arithmetic multiprocessor can achieve multigigaflops performance for most compound arithmetic computations. Furthermore, the above computations achieve half of the maximum throughput, if $N \geq$ 114, 90, 19, and 90 respectively. This implies that the dynamic arithmetic processor can significantly speed up compound arithmetic computations even when the vector has only moderate length.

## 6. Conclusions

With today's VLSI technology, it is not premature to consider implementing advanced arithmetics directly in hardware. On the other hand, designing a very powerful arithmetic multiprocessor with the capability of supporting various types of compound arithmetics is still a yet-to-be-challenged task by computer industry . This paper suggests a networking solution to this problem. Performance analysis shows that if the problem size is reasonably large, the dynamic arithmetic multiprocessor network can perform almost equally fast as their static counterparts. Continuing works are being conducted on the instruction set design, the VLSI design of the PE, and the effectiveness of using the proposed multiprocessor for solving partial differential equations(PDE) problems.

## References


1. Baugh, C.R. and Wooley, B.A. "A Two's Complement Parallel Array Multipication Algorithm". *IEEE Trans. on Computers C-22*, 12 (Dec. 1973), 1045-1047.

2. Chin, C.Y. and Hwang, K. "Packet Switching Networks for Multiprocessors and Dataflow Computers". *IEEE Trans. on Computers C-33*, 11 (Nov. 1984), 1110-1111.

3. Cray Research, Inc. *The Cray X-MP Series of Computer Systems.* Minneapolis, Minnesota, 1984.

4. Handler, W. "The Impact of Classification Schemes on Computer Architecture". *Proc. Int'l. Conf. on Parallel Processing* (1977), 7-15.

5. Hsu, F.H., Kung, H.T., Nishizawa, T., and Sussman, A. LINC: The Link and Interconnection Chip. Dept. of Computer Science, Carnegie-mellon Univ, May, 1984.

6. Hwang, K.. *Computer Arithmetic.* Wiley & Sons, New York, 1979.

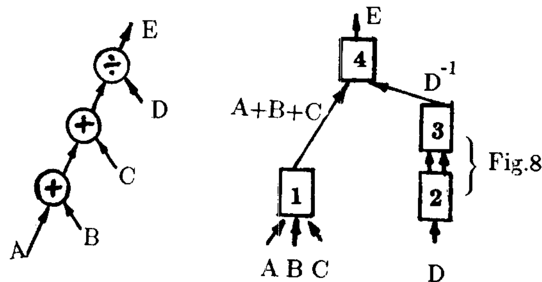
7. Hwang, K. "Multiprocessor Supercomputers and Scientific Applications". *IEEE Computer Magazine* (June 1985).

8. Hwang, K. and Cheng, Y.H. "Partitioned Matrix Algorithms for VLSI Arithmetic Systems". *IEEE Trans. on Computers C-31*, 12 (Dec. 1982), 1215-1224.

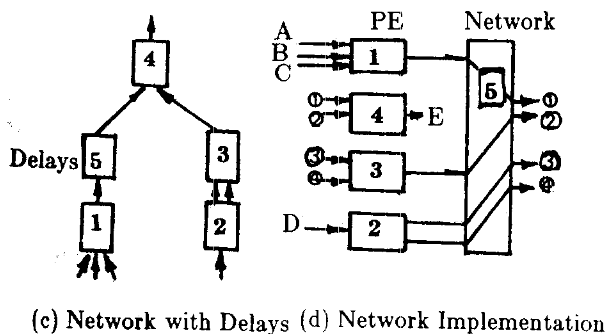
9. Hwang, K. and Xu, Z. Dynamic Systolization for Developing Multiprocessor Supercomputers. TR-EE 84-42, School of Electrical Engineering, Purdue University, Oct., 1984.

10. Ni, L.M. and Hwang, K. "Vector Reduction Techniques for Arithmetic Pipelines". *IEEE Trans. on Computers C-34*, 5 (May 1985).

11. Snyder, L. "Introduction to the Configurable, Highly Parallel Computer". *IEEE Computer* (Jan. 1982), 47-64.

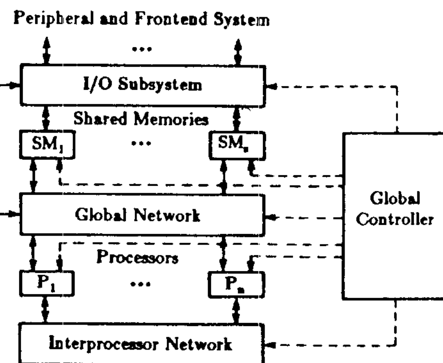



(a) Dataflow Graph (b) Multi-PE Network

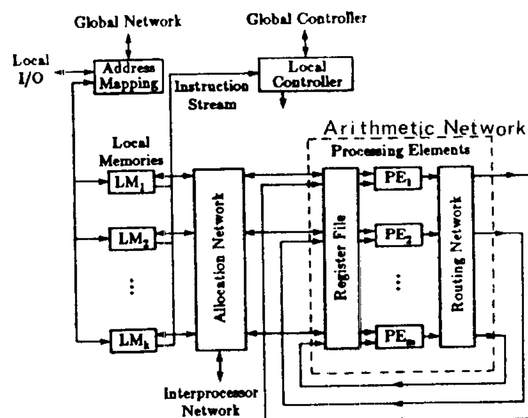



(c) Network with Delays (d) Network Implementation

Fig.2. Mapping a dataflow graph into an arithmetic network




(a) The multiprocessor structure




(b) The arithmetic network in each processor

Fig.1. The system architecture of the Remps

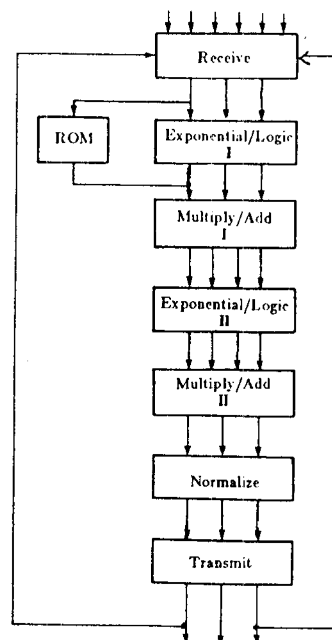



Fig.3. Arithmetic pipeline in each PE

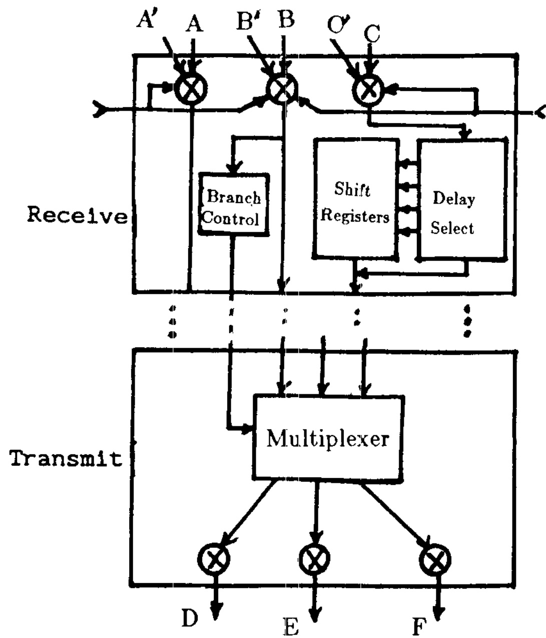



Fig.4. The Receive and the Transmit stages in PE

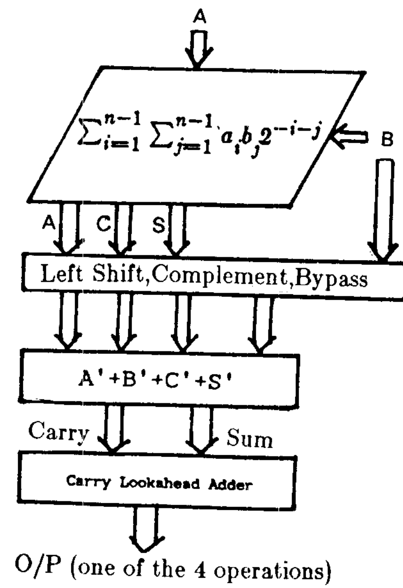



Fig.5. The Multiply/Add stage in each PE pipeline

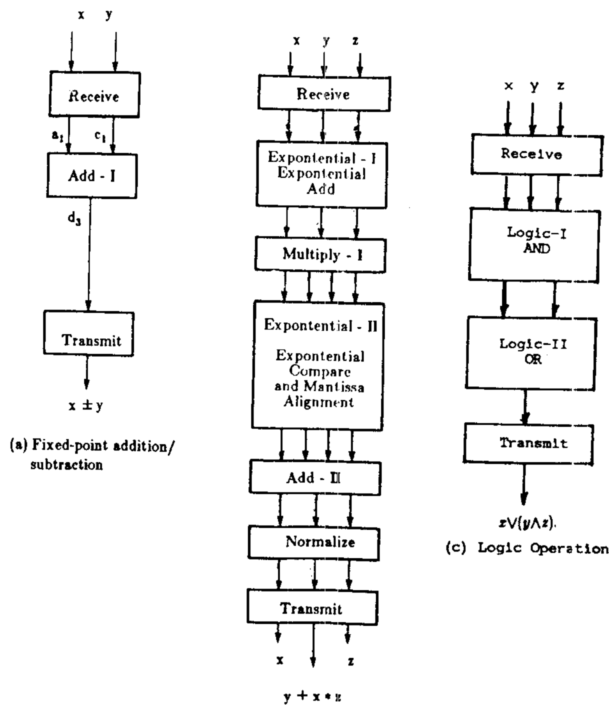



Fig.6. Typical arithmetic functions performed by each PE pipeline.

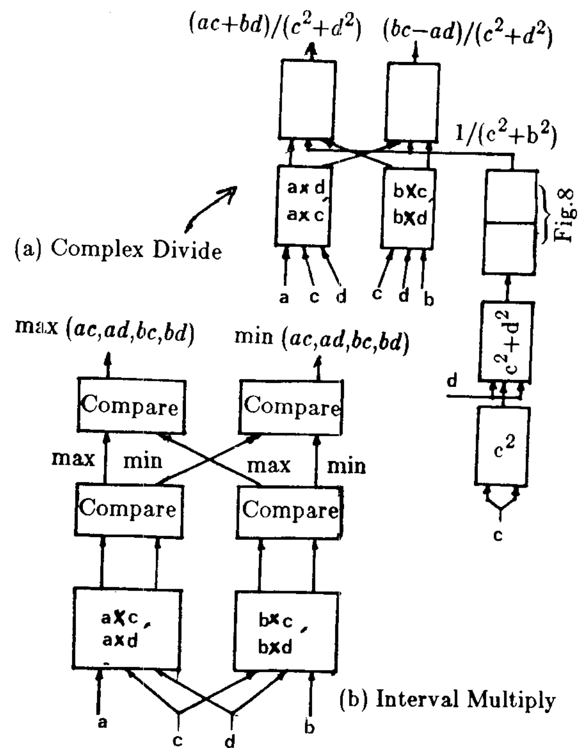



Fig.7. Arithmetic networks for complex divide and interval multiply

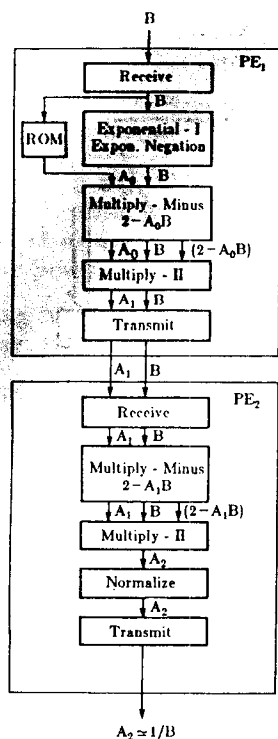
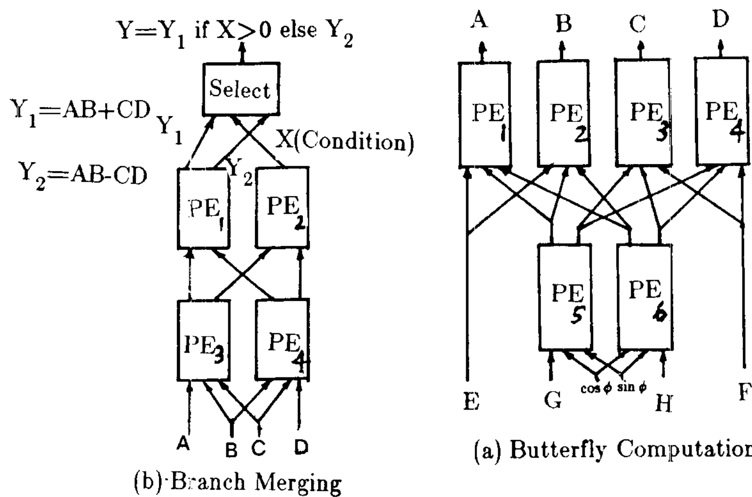


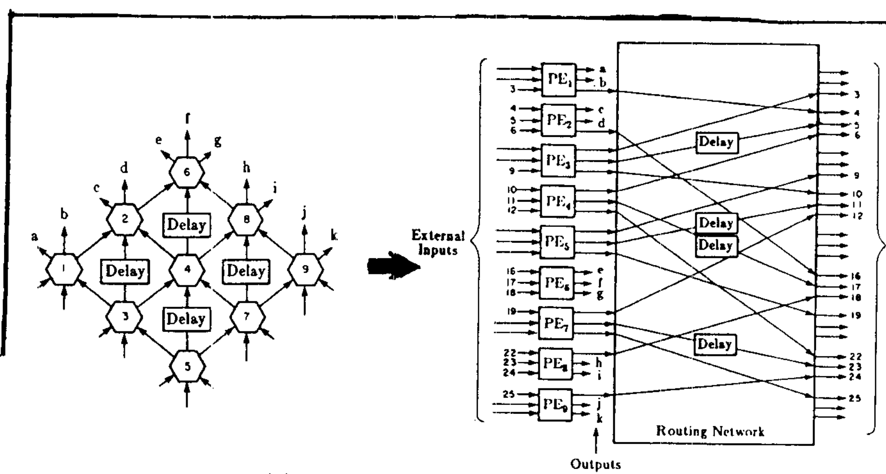

Fig.8. Floating-point reciprocation using two PEs in cascade



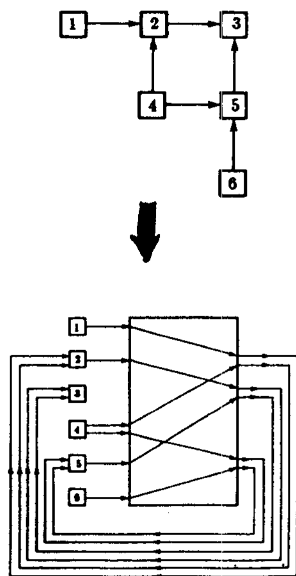

(b) Branch Merging

(a) Butterfly Computation

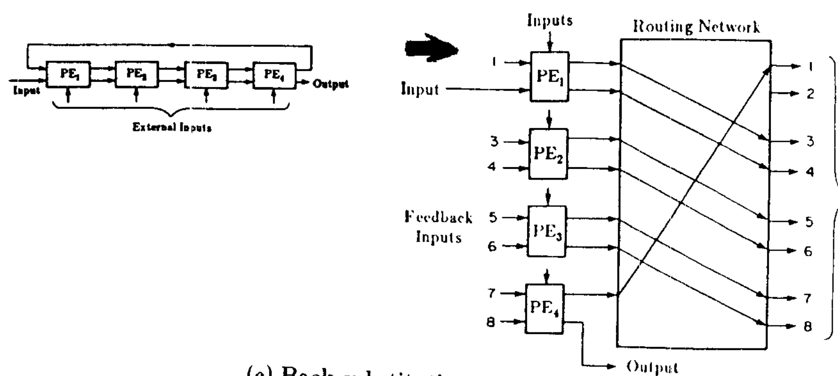
Fig.9. Arithmetic networks for butterfly and branch operations




(a) L-U decomposition

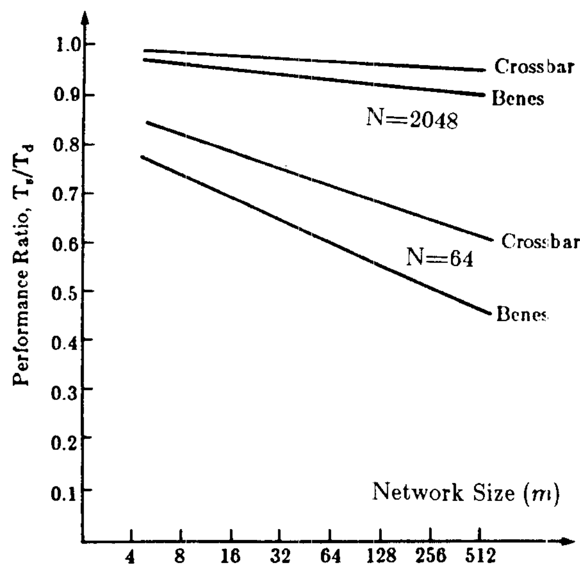


(b) Triangular matrix inversion

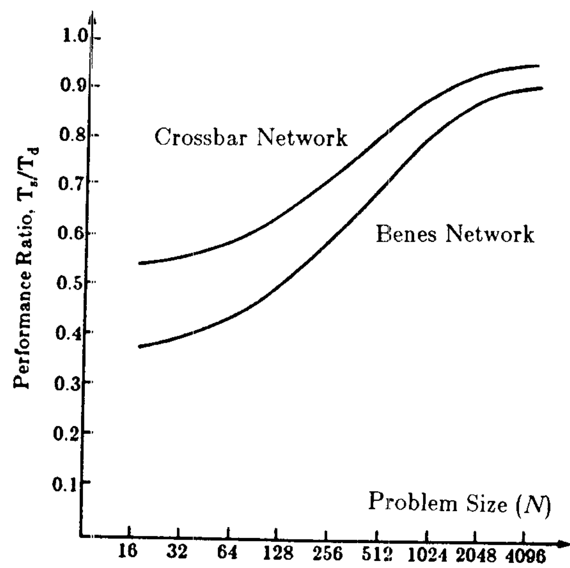



(c) Back substitution

Fig.10. Arithmetic networks and multi-PE implementations for solving linear system of equations




(a) Performance vs. network size




(b) Performance vs. problem size

Fig.11. Relative performance of dynamic arithmetic networks and static systolic arrays

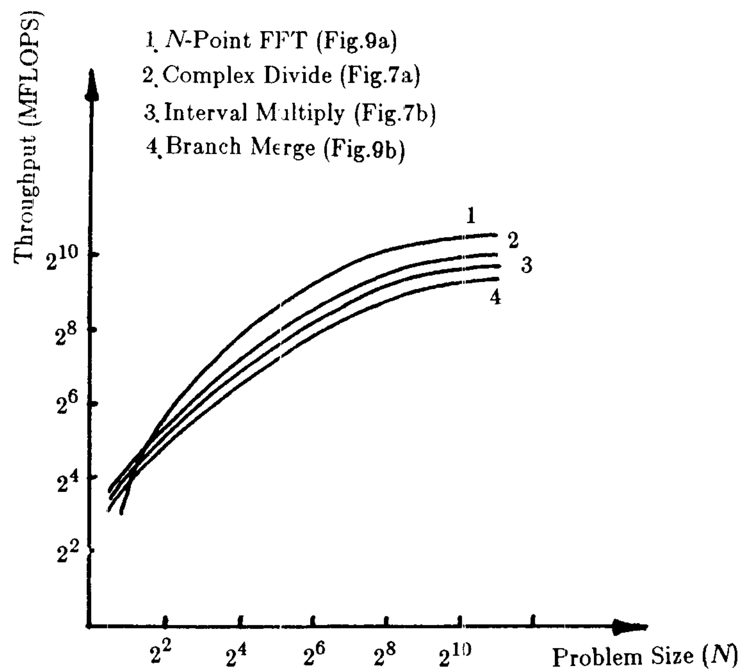



Fig.12. Throughput performances of several compound arithmetic functions implemented on the multi-PE arithmetic networks