INSTRUCTION SYSTOLIC ARRAYS FOR EXACT PARALLEL LINEAR ALGEBRAIC COMPUTATION

A thesis submitted for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

at the

UNIVERSITY OF WAIKATO

HAMILTON

NEW ZEALAND

by

V. KRISHNAMURTHY

1988

INSTRUCTION SYSTOLIC ARRAYS FOR EXACT PARALLEL LINEAR ALGEBRAIC COMPUTATION

Abstract

This thesis develops new computational algorithms for parallel/distributed error-free (exact) rational (real and complex) computing and its applications to exact linear algebraic computing.

The practical realization of these parallel algorithms in VLSI systems using a new concept known as Instruction Systolic Arrays (ISA) is considered.

Thus, the thesis represents the author's contribution to the theory, design, and implementation of parallel/distributed algorithms for exact rational/linear algebraic computing using ISA processors.

This thesis consists of six chapters with the following contents.

CHAPTER 1

INSTRUCTION SYSTOLIC ARRAYS - A PROGRAMMABLE PARALLEL ARCHITECTURE

Chapter 1 introduces the concept of Instruction Systolic arrays (ISA). The ISA is a new versatile programmable parallel architecture that retains all the advantages of systolic arrays. In the ISA, instructions and boolean selectors are pumped through the processing array. The ISA has tremendous potential for VLSI computing because of

its ability to execute different types of programs on the same processing array. We also introduce a variant of the ISA known as the Single Instruction Systolic Array (SISA). In the SISA, single instructions and selectors are pumped through the processing array.

CHAPTER 2

INSTRUCTION SYSTOLIC ARRAYS FOR DISTRIBUTED CHINESE REMAINDERING/ INTERPOLATION ALGORITHMS

We present a parallel/ distributed algorithm for a fundamental problem in Numerical Computing/algebraic computing, namely interpolation and the closely related Chinese remaindering. The ISA implementation of this algorithm is described. A generalization of the interpolation algorithm for multivariable interpolation is then presented. The multivariable interpolation algorithm can be realized in a ISA as well as in a pyramid architecture. The Occam simulation of a pyramid for multivariable interpolation is also described. We then present a parallel/ distributed algorithm for rational function interpolation and consider its ISA implementation.

CHAPTER 3

PARALLEL MATRIX COMPUTATION USING ISA

This chapter introduces a parallel algorithm for the generalized inversion (g-inversion) of matrices, which involves parallel matrix addition and multiplication operations. The ISA implementation of this algorithm is described. The solution of a homogeneous system of linear algebraic equations is based on g-inverse computation and we indicate its ISA implementation. The solution of such equations has important

practical applications in finding Petri net invariants, chemical equation balancing and dimensional analysis. Two other important g-inversion algorithms - one iterative and the other direct - are also described.

CHAPTER 4

PARALLEL ERROR-FREE RATIONAL ARITHMETIC - ISA IMPLEMENTATION

This chapter introduces a new parallel error-free (exact) rational arithmetic system called Parallel Rational Hensel code arithmetic (Para-Hensel code or PHC). We introduce algorithms for parallel element-wise arithmetic operations, and for encoding and decoding PHC codes. We briefly indicate the ISA implementation of these algorithms. We also consider the application of PHC for exact parallel matrix g-inversion with systolic processors.

CHAPTER 5

PARALLEL COMPLEX RATIONAL AND MATRIX ARITHMETIC USING GAUSSIAN PRIME CODES

In this chapter, we introduce a new parallel exact complex rational arithmetic system based on Gaussian prime codes. Two closely related methods for constructing these codes are described. The first method uses several distinct (multiple) Gaussian primes to construct Gauss-Hensel codes (GHC). The second method uses the powers of one or more Gaussian primes to construct Gauss p-adic codes (GPC). Their practical applications and extension to parallel inversion of complex matrices are described. These systems are amenable for massively parallel realization.

CHAPTER 6

ISA - APPLICATIONS TO ARRAY PROCESSING AND REAL-TIME COMPUTING

This chapter considers the suitability of the ISA for vector/ array processing computers. We also consider the relationship of the ISA to wavefront processing and dataflow computing. We then describe a linear variant of the ISA known as the Linear ISA (LISA), which is an area-efficient architecture. We then briefly indicate some possible application areas of the ISA: real-time robot control applications and high-speed vision systems.

PREFACE

The work reported in this thesis was carried out by the author at the University of Waikato. In each chapter of this thesis, the numbering scheme for pages, sections, tables and figures is independent. Also, the references pertaining to each chapter are collected and presented at the end of this thesis.

The author expresses his gratitude to Dr. O.Y. de Vel, University of Waikato and to Dr. H. Schroder, Australian National University for their guidance and valuable suggestions during the period this work was carried out.

The author wishes to thank the University of Waikato for providing the research facilities. Also, he thanks Prof. Richard Brent and the Australian National University for providing financial support and research facilities during his visit in the summer vacation 1987-1988.

CONTENTS

CHAPTER 1

INSTRUCTION SYSTOLIC ARRAYS - A PROGRAMMABLE PARALLEL ARCHITECTURE

CHAPTER 2

INSTRUCTION SYSTOLIC ARRAYS FOR DISTRIBUTED CHINESE REMAINDERING/ INTERPOLATION ALGORITHMS

CHAPTER 3

PARALLEL MATRIX COMPUTATION USING ISA

CHAPTER 4

PARALLEL ERROR-FREE RATIONAL ARITHMETIC - ISA IMPLEMENTATION

CHAPTER 5

PARALLEL COMPLEX RATIONAL AND MATRIX ARITHMETIC USING GAUSSIAN PRIME CODES

CHAPTER 6

ISA - APPLICATIONS TO ARRAY PROCESSING AND REAL-TIME COMPUTING

CHAPTER 1

INSTRUCTION SYSTOLIC ARRAYS - A PROGRAMMABLE PARALLEL ARCHITECTURE

ABSTRACT

Chapter 1 introduces the concept of Instruction Systolic arrays (ISA). The ISA is a new versatile programmable parallel architecture that retains all the advantages of systolic arrays. In the ISA, instructions and boolean selectors are pumped through the processing array. The ISA has tremendous potential for VLSI computing because of its ability to execute different types of programs on the same processing array. We also introduce a variant of the ISA known as the Single Instruction Systolic Array (SISA). In the SISA, single instructions and selectors are pumped through the processing array.

CHAPTER 1

Instruction Systolic Arrays - A Programmable Parallel Architecture

1. Introduction

Systolic array computer architectures provide a means for fast parallel realization of several important algorithms in linear algebra, graph theory, signal processing and sorting [Kung et al 1978 and 1979, Kung 1979, Kung 1988, Fortes et al 1987, Mead 1980, Murthy 1988]. A systolic array architecture consists of a mesh-connected array of processors, in which one or more streams of data are passed synchronously from one processor to neighbouring processors in a regular rythmic pattern under the control of a global clock. Since the processors are placed in the nodes of a grid, the algorithms can be realized with minimal propagation delay. This allows the clock cycle to be relatively short.

The main features that make systolic architectures highly suitable for VLSI implementation are the following :

a. regularity of the network
b. network of simple processing elements (PE)
c. modularity
d. local communication between processing elements

However, the main disadvantage of the systolic array architecture (SA) is its lack of flexibility. Its special purpose architecture can only execute a specific algorithm. For example, a SA designed for inverting matrices cannot perform sorting.

There have been several approaches to make the SA more flexible, so that different problems can be handled by a single systolic array [Chang et al 1982, Fisher 1981, Lang 1986]. In this dissertation, we will only consider the instruction systolic array (ISA) approach due to Lang et al [Kunde et al 1984, Lang 1986, Schroder 1986].

2. Principle of Instruction Systolic Arrays

The Instruction Systolic array (ISA) is a new versatile programmable parallel architecture that retains all the advantages of the SA. Its ability to efficiently execute many different types of algorithms using the same processing array make it highly suitable for VLSI/ WSI (Wafer-scale integration).

In the ISA, a sequence of instructions called the Top Program (TP), and an orthogonal sequence of boolean selectors called the Left Program (LP) are pumped through the mesh-connected array of processors (processing elements) (Fig. 1) (unlike a SA, where only data is pumped through the processors). If an instruction meets selector bit '1' at a processor, then the instruction is executed at that processor. Otherwise (if the instruction meets selector bit '0'), the instruction is not executed at that processor leaving the registers contents unchanged. Thus, a ISA program consists of the 2-tuple (TP,LP). Data is supplied to the ISA from external data queues, which can be read during program execution.

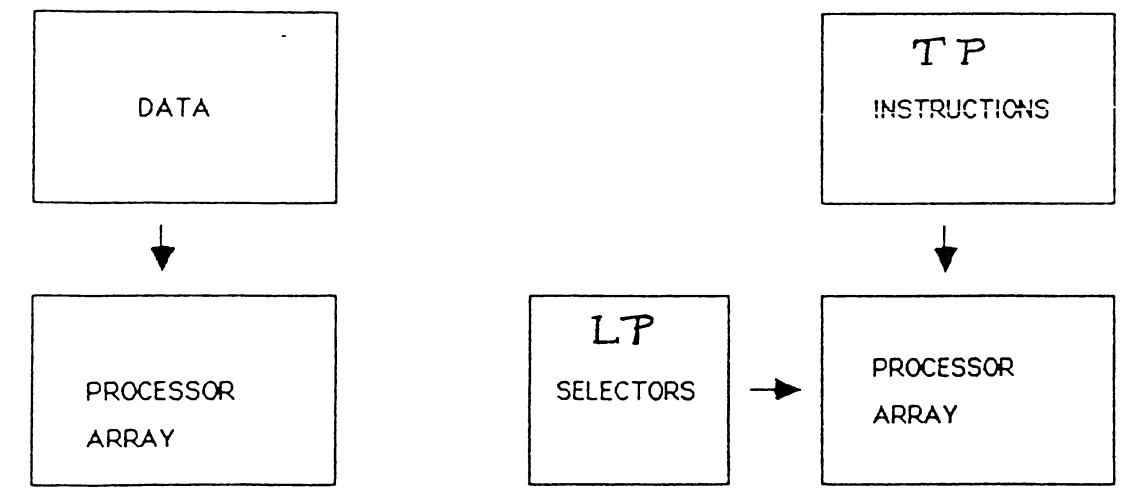


Fig. 1 (a) A SYSTOLIC ARRAY
(b) A INSTRUCTION SYSTOLIC ARRAY

It is clear that we can execute many different programs on the same architecture, simply by pumping in different sets of instructions; this makes the ISA very versatile. The flexibility of a ISA is further enhanced by the ability to inhibit instructions at certain processors using selectors. These two features make the ISA a programmable VLSI systolic architecture.

Note that the ISA described above uses the convention that instructions are fed from the top (north) and selectors are fed from the left (west); certainly, other configurations are admissible.

In contrast to the ordinary mesh-connected processing array which consists of independent processors (with their own control units and program storage) capable of communicating with their four direct neighbours, the ISA processors have no local memory for program storage. The program is stored in external memory instead, and pumped in when required. Thus, a ISA requires a smaller chip area, resulting in a greater degree of miniaturization; this in turn leads to high-speed computation.

Remark In the ISA, we may consider that instructions and selectors propagate in wavefronts through the processing array, starting from the processing element at the top-left corner and ending at the bottom right corner. These wavefronts never intersect with each other. That is, processors on different diagonals execute different instruction wavefronts at any given instant. The ISA is thus closely related to wavefront processing; based on this it seems possible to design a new architecture called the Instruction Wavefront Array (IWA) which is discussed in Chapter 6.

3. Basic ISA Model

The ISA consists of a mesh-connected m x n array of processors - processing elements (PE) - synchronized by a global clock. Each PE in the ISA has simple control units. Each processor has an Instruction Register (IR) which fetches instructions from a neighbour.

The processors can execute instructions (such as arithmetic, comparison and logical operations) from a fixed instruction set. The execution of each instruction is assumed to take the same time, the clock cycle time; this is used as the basic unit of time measurement. The processors have some data registers, including a communication register 'K', which is used for storing and propagating data. The processors do not store programs in local memory but are supplied with programs (instructions) stored in external memory.

The instruction cycle in the ISA consists of two distinct stages. In the first stage, a processor reads (fetches) instructions and data from its own registers or from those of a neighbour. In the second stage, the desired computations are performed, and the processors write the instructions and data to its own registers. This ensures that we read a register before its contents have been updated, that is, we avoid read-write conflicts.

The Instruction Register of each processor synchronously fetches an instruction from the IR of a neighbouring processor. The processors at the boundary fetch instructions from an external memory. In this way, rows of instructions are shifted through the ISA. The columns of selectors are simultaneously shifted in an analogous way. Note that the pumping of instructions and selectors in the ISA

forms a set of wavefronts propagating from PE at the top left corner, i.e., PE(1,1).

The communication register 'K' of each processor synchronously fetches data from the communication register 'K' of a neighbouring processor. Each processor can read its own communication register as well as the communication registers of its four direct neighbours. This means that atmost five processors can read from a communication register simultaneously. However, each processor can write only to its own K. Data can then be sent from a processor P to its neighbour Q as follows: P writes data to its K, and then Q reads P's K in the next instruction cycle. Note that processors can also have some internal memory which is not visible to its neighbours.

The processors at the boundary fetch data from external data queues. The data can be read during program execution.

Input and output to the ISA is done via the open-ended processor links on the boundary of the ISA. The ISA is assumed to be in an environment that is capable of

- supplying ISA with instructions and selectors (from external memory).
- supplying ISA with input data and storing its output data.

4. ISA programs - Left and Top programs

We now formally define a ISA program and introduce the notion of time complexity in ISA programs.

4.1 Formal Definitions

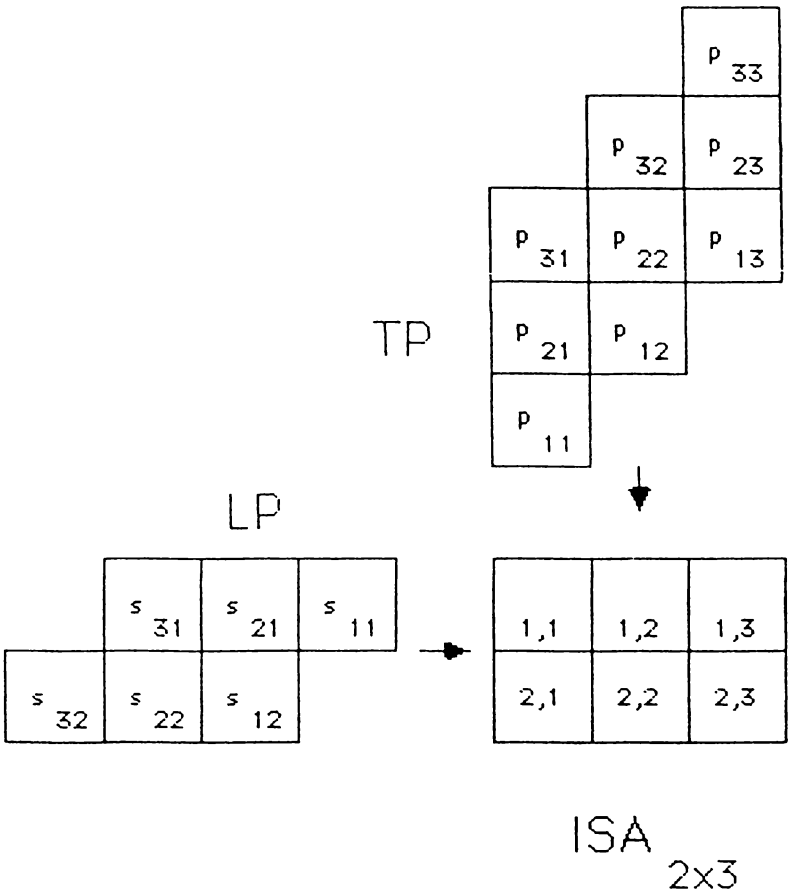
Let the ISA consist of a m x n processing array, where we assume that an instruction (or selector) moves from a processor to its neighbour in one time unit, the clock cycle time. We then define a ISA program as a sequence of instruction diagonals $p_1,p_2,...,p_r$ of n-tuples over Instruction set I (Top program - TP) and a sequence of boolean selector diagonals $s_1,s_2...,s_r$ of m-tuples over {0,1} (Left program - LP) (see Fig. 2); here, p_{ij} denotes the jth element in instruction diagonal p_i and s_{ij} denotes the jth element in the selector diagonal s_i. In Fig. 2, we have chosen m = 2, n = 3, r =3.

For all $i \leq m$, $j \leq n$ and $t \leq r$,
the p_t Instruction diagonal enters the ith row of ISA at time t+i-1, and
the s_t Selector diagonal enters the jth column of ISA at time t+j-1.

Then p_{tj} (the jth element in instruction diagonal p_t) enters the ith row at
time = t+i-1+ (j-1) = t+i+j-2

and s_{ti} (the ith element in selector diagonal s_t) enters the jth column at
time t+j-1+ (i-1) = t+i+j-2.

That is, instruction p_{tj} and selector s_{ti} simultaneously enter processor (i,j) (on row i and column j) at time t+i+j-2.
Thus, instruction $p_{t-i-j+2,j}$ is executed by processor (i,j) at time t if $s_{t-i-j+2,i} = 1$; otherwise, the instruction is not executed.



THE INSTRUCTION SYSTOLIC ARRAY

Fig. 2

Remark

The ISA program consists of instruction and selector diagonals and can be viewed as diamond-shaped (see Fig. 2). The action of a ISA program can be viewed as the combined effect of the interaction of each instruction diagonal with its corresponding selector diagonal. In this interaction, the ith element in a selector diagonal decides whether the corresponding instruction diagonal is executed in the ith row of a ISA.

The action of a ISA program can also be considered to be equivalent to finding the product of two matrices, the m x r boolean selector matrix (LP) and the r x n instruction matrix (TP). The interaction of each selector diagonal with its corresponding instruction diagonal corresponds to the outer product between a m x 1 column and 1 x n row vector under the following elementwise multiplication rule: 0 * x = NO-OP (no-operation) and 1*x = x, where '0' and '1' are selectors and 'x' is an instruction.

4.2 Period and Execution time of programs

We now define the period and execution time of a ISA program, where we have a m x n array of processors. In the definitions below, we can measure time in terms of the number of instructions pumped through the processor array since the execution of each instruction is assumed to take the same duration, the clock cycle, which is taken as the unit of time. In other words, an instruction moves from a processor to its neighbour in one time unit, and in x time units, an instruction moves to a processor x processing cells below.

The **period** of a ISA program is measured in terms of the number of active instruction diagonals (excluding no-operation (NO-OP) diagonals at the beginning or the end) which equals the number of selector diagonals = r.

The **execution time** (time) of a ISA program is the time interval between the first instruction entering the ISA and the last instruction leaving the ISA (which corresponds to the concept of latency in systolic arrays [Kung 1988]) =
the time for the last element of the last instruction to enter the ISA + the time to pump instructions through the ISA = (r+n-1) + (m-1) = r+n+m-2.

4.3 Concatenation of programs

When a program P is the concatenation of two subprograms P1 and P2,
Time(P) = Time(P1) + Period(P2) and
Period(P) = Period(P1) + Period(P2)
The relation for time is obvious, since the first diagonal of P2 can immediately follow the last diagonal of P1. The above formulae can be similarly extended to n concatenated subprograms.

Remark It is sometimes necessary to introduce a diagonal of NO-OP's between successive subprograms to prevent read-write conflicts.

We now illustrate the basic concepts of ISA using the familiar example of matrix multiplication.

5. ISA Matrix Multiplier

Let A be a m x n matrix and B a n x k matrix, where we have a m x k ISA processing array. Then, the m x k product C can be constructed as the sum of the outer products of each column of A with the corresponding column of B [Krishnamurthy et al 1981]

$$C = A \times B = \sum_{i=1}^{n} COLA_i \; ROWB_i$$

where $COLA_i$ is the ith column of A and $ROWB_i$ the ith row of B.

5.1 ISA program

The ISA program for multiplying two matrices A and B is shown in Fig. 3a; here, A is a 2 x 3 and B a 3 x 2 matrix . The input data, matrices A and B, are available in input queues to the left and top of the ISA respectively (see Fig. 3b). Let a_{ij} and b_{ij} denote elements in the ith row and jth column in A and B respectively. Let the matrix product be denoted by C, where c_{ij} is the element in the ith row and jth column in C.

Here, K is the communication register, and D1 and D2 are data registers. We have the following instruction set:

SR : K := K_L

ß : D1 := K

SD : K := K_T

* : D1 := D1*K

\+ : D2 := D2+D1

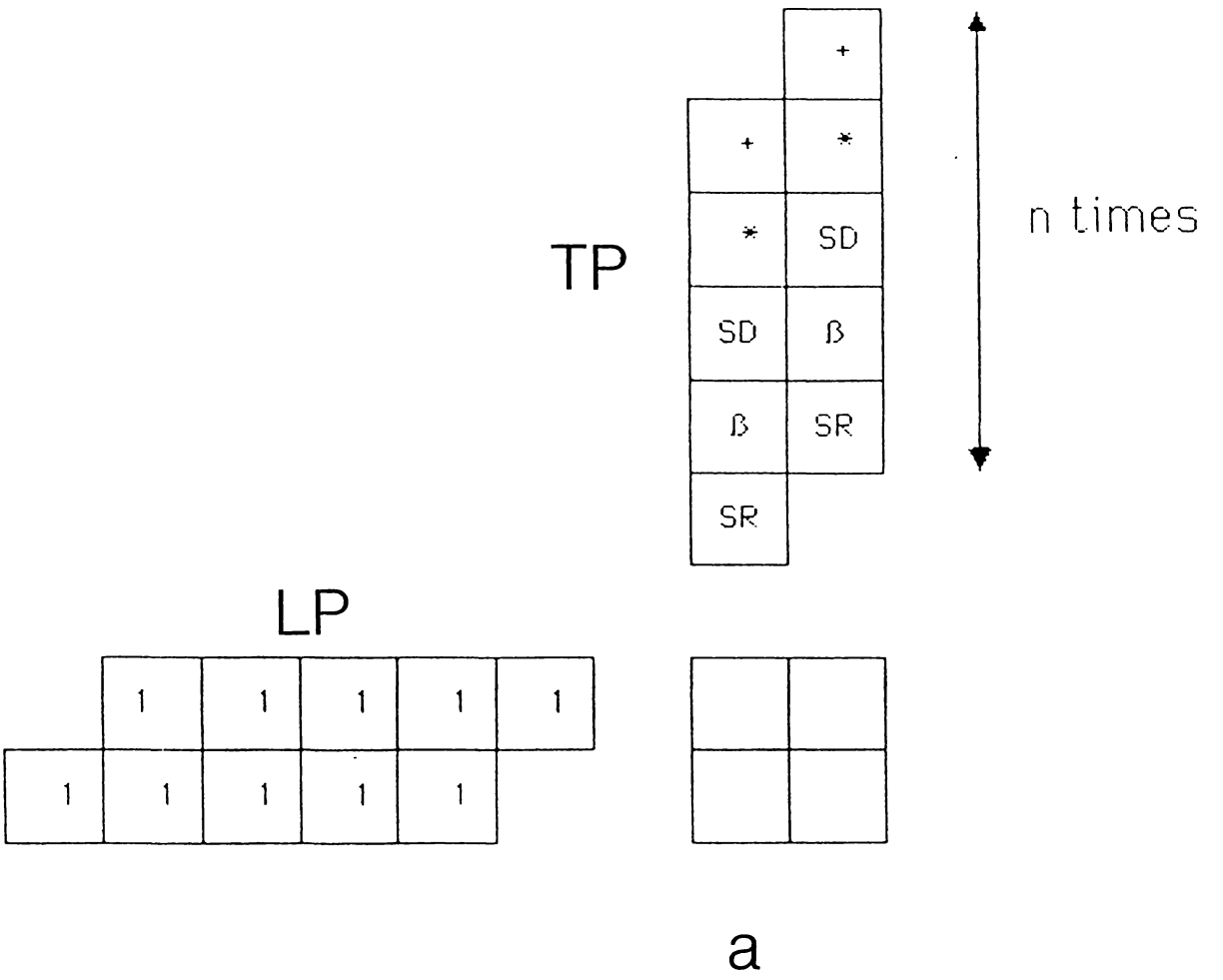


Fig. 3 ISA MATRIX MULTIPLIER

Instruction SR reads the contents of K of the left neighbour and writes it into its own K. Hence, the first instruction diagonal shifts the last column of A to all processors in a ISA.

Instruction ß stores the contents of communication register K in D1. Hence, the second instruction diagonal stores the elements of the last column of A in intermediate storage.

Instruction SD reads the contents of K of the top neighbour and writes it into its own K. Hence, the third instruction diagonal shifts the last row of B to all processors in the ISA.

Instruction '*' stores the product of the contents of D1 and K in D1. Hence, the fourth instruction diagonal computes $a_{i,k} * b_{k,j}$ in each processor (i,j) (k=3 here).

Instruction '+' stores the sum of D2 and D1 in D2. Hence, the fifth instruction diagonal computes $c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}$ in each processor(i,j).

The above instructions are repeated n times (n is the number of columns in A = number of rows in B), to generate the m x k product matrix C. The elements of C are stored in the processing elements of the m x k processing array.

We note that a single instruction diagonal can read and broadcast data to the last column in each row. Thus, although there is no common bus connecting the processors of a row, data can be input and broadcast with period 1. Similarly, we can broadcast data from the first row to the rows below in period 1. This illustrates the suitability of the ISA for certain problems where broadcasting is involved. However, broadcasting in the opposite direction to the flow of instructions and selectors (i.e. from right to left or from bottom to top) takes period $\Omega(n)$, where n is the number of rows (columns) in the processing array.

5.2 Time Complexity

We have a mxk ISA processing array. Here, m=2,k=2. The number of iterations of TP is n = 3. Also, r=5 = number of active instruction diagonals.
Then, Period of each iteration is the same and equals r =5.

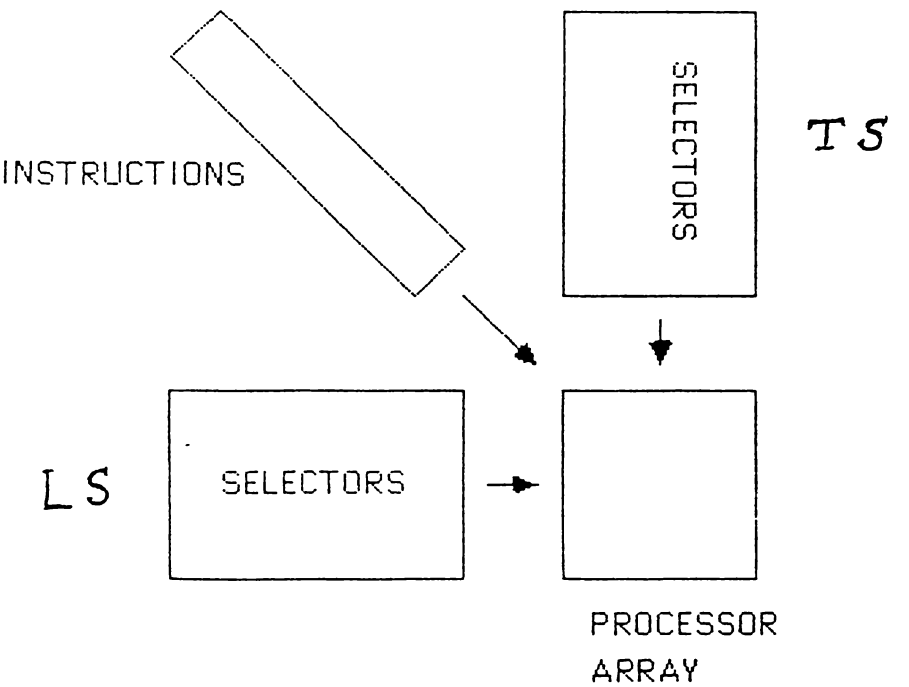
Execution time for one iteration = T1= r+k+m-2 = 7.

Total execution time for n iterations=T1 + Period (iteration2) + ...+Period(iteration n)
= T1 + (n-1)r = k + m + nr - 2 = 17.
The total period=nr = 15.

6. The Single Instruction Systolic Array

The Single Instruction Systolic Array (SISA) [Lang 1987] is a simpler variant of the ISA. Here, single instructions are pumped through a mesh-connected processing array in diagonal wavefronts, starting from the top left (north-west) corner and ending at the bottom right (south-east) corner. In addition, two orthogonal sequences of boolean selectors called top selectors (TS) (column selectors) and left selectors (LS) (row selectors) are pumped from the top (north) and left (west) respectively (see Fig. 4). If an instruction meets a TS selector bit '1' and a LS selector bit '1' at a processor, then the instruction is executed. Otherwise (if the selector bit of either TS or LS is '0'), the instruction is not executed in that processor.

to 110



THE SISA

FIG. 4

In contrast to a ISA, the SISA has a symmetric control flow since both the TS and LS consist of boolean selectors. These boolean selectors enhance the flexibility of the architecture since we can inhibit instructions at certain processors.

6.1. Basic SISA Model

The SISA consists of a mesh-connected m x n array of processors (processing elements). The processors have the same structure as ISA processors. Each processing element in the SISA has simple control units. Each processor has an Instruction Register (IR) which fetches instructions from a neighbour. A processor can execute instructions (such as arithmetic, comparison and logical operations) from a fixed instruction set. The execution of each instruction is assumed to take the same number of time units; this is used as the basic unit of measurement. The processors have some data registers, including a communication register 'K', which is used for storing and propagating data. The processors do not store programs in local memory but are supplied with programs (instructions) stored in external memory.

The instruction cycle in the SISA is identical to the ISA and consists of two stages, reading (fetching) instructions and data, and writing values to registers after the computation. However, unlike the ISA, each processor in the top row fetches instructions from the left neighbour. All other processors fetch instructions from the top neighbour. The top left processor (on north-west corner) fetches instructions from an external program memory. This is done synchronously so that instructions move in wavefronts from from the top left corner through the diagonals of the array to the bottom right corner. The TS and LS are simultaneously shifted towards the bottom and right respectively.

7. The SISA Program

We now formally define a SISA program and derive its time complexity.

7.1 Formal Definition

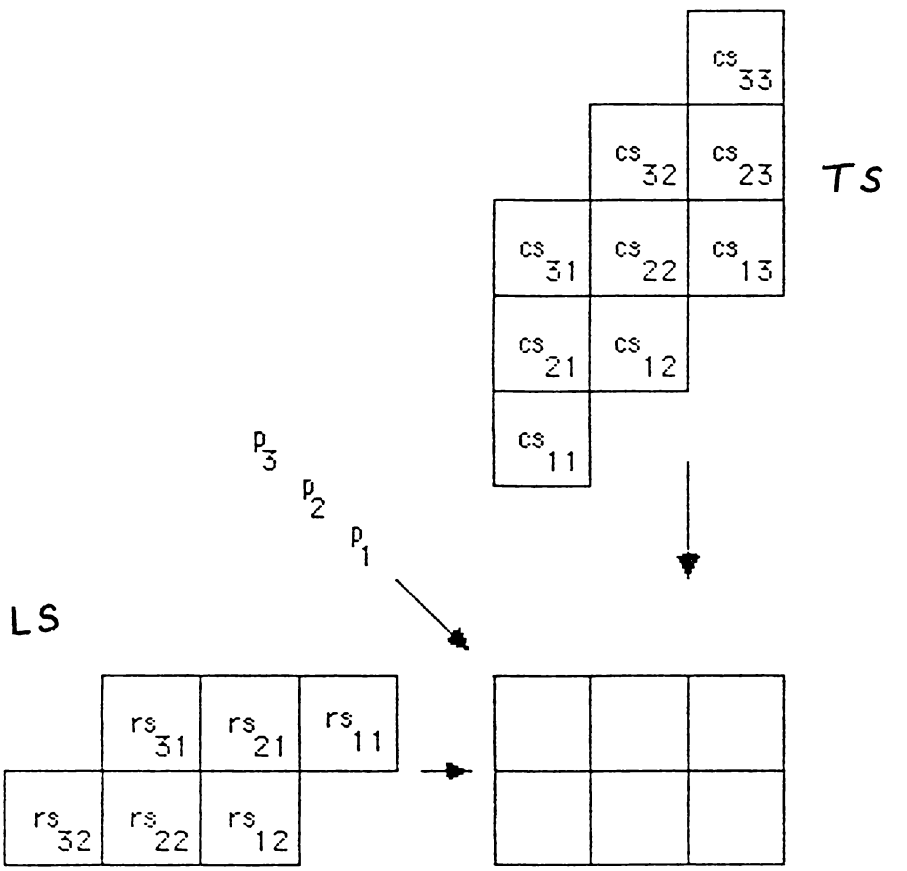
Let the SISA consist of a m x n processing array. We then define a SISA program as a sequence of single instruction $p_1,p_2,...,p_r$ over Instruction set I, a sequence of boolean selector diagonals cs_1, cs_2..., cs_r of n-tuples over {0,1} (TS - column selectors), and a sequence of boolean selector diagonals rs_1, rs_2..., rs_r of m-tuples over {0,1} (LS - row selectors) (see Fig. 5); here, rs_{ij} denotes the jth element in selector diagonal rs_i and cs_{ij} denotes the jth element in the selector diagonal cs_i. In Fig. 5, we have chosen m = 2, n = 3, r =3.

A compact notation for a SISA program is to represent it as a sequence of tuples (p,cs,rs); here, p is an instruction, cs a column selector diagonal, and rs a row selector diagonal.

For all $i \leq m$, $j \leq n$ and $t \leq r$,
instruction p_t enters processor (1,1) at time t,
thus entering processor (i,j) (on row i and column j) at time t+i-1+j-1 = t+i+j-2.

Selectors rs_{ti} and cs_{tj} simultaneously enter processor (i,j) at time t+i+j-2.

to 13a



THE SINGLE INSTRUCTION SYSTOLIC ARRAY

FIG. 5

Thus, instruction p_t is executed by processor (i,j) at time t+i+j-2 if $rs_{ti} = cs_{tj} = 1$; otherwise, the instruction is not executed.

7.2 Period and Execution Time of SISA programs

We now define the period and execution time of a SISA program, where we have a m x n array of processors. In the definitions below, we can measure time in terms of the number of instructions pumped through the processor array since the execution of each instruction is assumed to take the same duration, the clock cycle, which is taken as the unit of time.

The **period** of a SISA program is measured in terms of the number of active instructions (excluding no-operation instructions at the beginning or the end) which equals the number of selector diagonals = r.

The **execution time** (time) of a SISA program is the length of the time interval between the first instruction entering the SISA and the last instruction leaving the SISA = r + (m-1) + (n-1) = r+m+n-2 (the same formula as the time complexity of a ISA program).

The time and period of concatenated programs is the same formula as that derived for the ISA (section 4.3). If program P is the concatenation of two subprograms P1 and P2,

Time(P) = Time(P1) + Period(P2)

Period(P) = Period(P1) + Period(P2)

The above formula can be similarly extended to n concatenated programs.

Remark As in the case of the ISA (section 4.3), it is sometimes necessary to introduce a diagonal of NO-OP's between successive subprograms to prevent read-write conflicts.

We now illustrate the concept of SISA using the following ringshift program.

8. SISA Ringshift Program

The ringshift of an array is defined as the combination of a right to left horizontal rotation of the rows and a bottom to top vertical rotation of the columns (with wraparound effect).

This program performs a ringshift of registers (say K) in a m x n SISA processing array, where m = n = 4 in this example.

Let us assume that the contents of the registers K are initially

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Then, the contents of the registers K after the ringshift are

6 7 8 5
10 11 12 9
14 15 16 13
2 3 4 1

8.1 SISA Implementation

The program (see Fig. 6) can be represented as the concatenation of two subprograms, ROTH K and ROTV K.

ROTH K:

$$(K := K_W, 01^{n-1}, 1^m)$$

$$(K := K_E, 1^{n-1}0, 1^m)$$

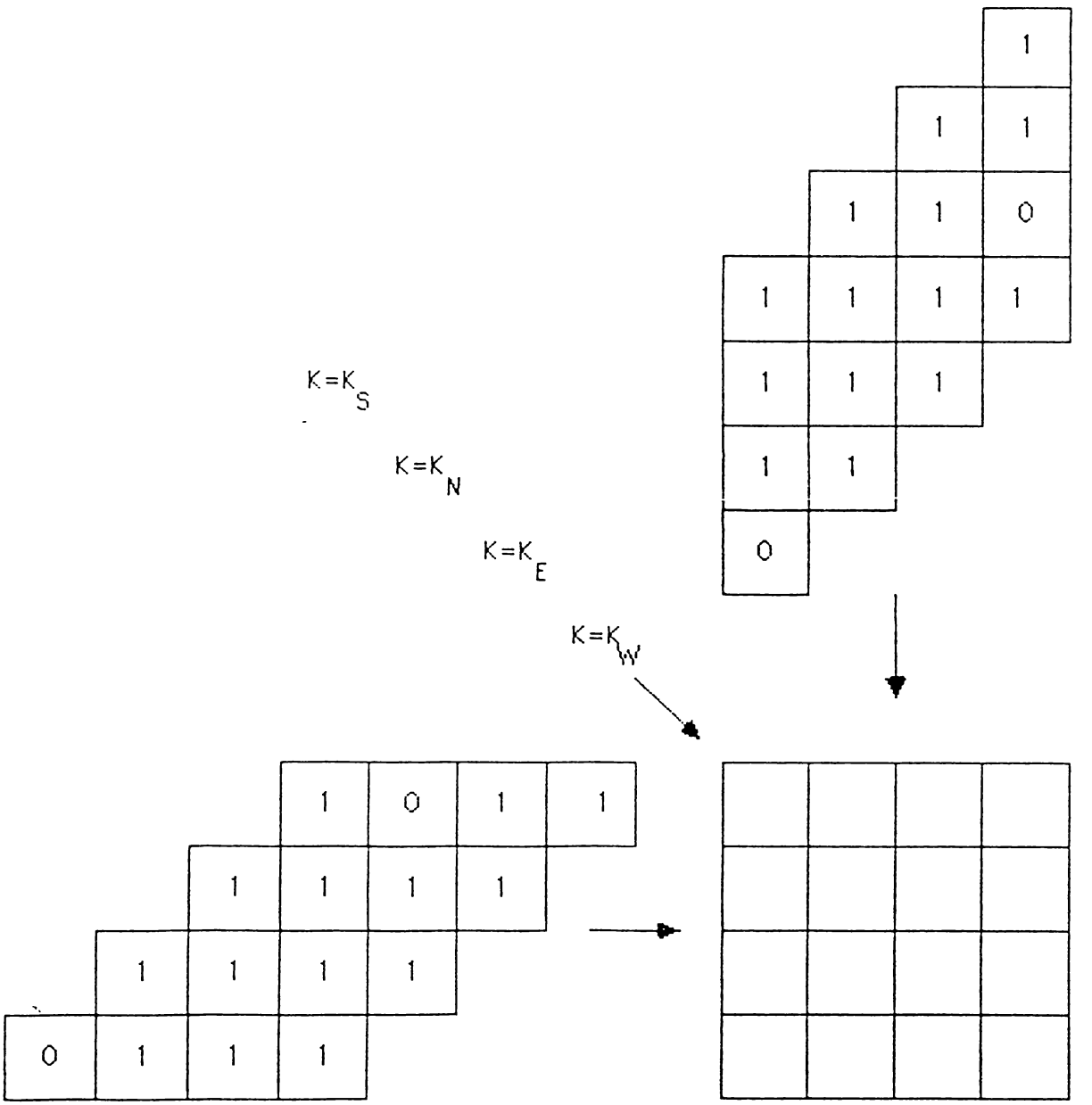
The ROTH K subprogram rotates the contents of register K horizontally from right to left, where the leftmost register is wrapped around. For example, let the contents of the registers in a row be 1 2 3 and 4 respectively. Then, after the rotation, the contents will be 2 3 4 1.

ROTH K consist of two instructions which have the following interpretations:
The first instruction $K := K_W$, reads the contents of K of the west (left) neighbour and writes it into its own K.
The second instruction $K := K_E$ reads the contents of K of the east (right) neighbour and writes it into its own K.
Here, the first instruction is not executed in the first column of processors since these processors don't have western neighbours. In an analogous way, the second instruction is not executed in the last column of processors since these processors don't have eastern neighbours.

ROTV K:

$$(K := K_N, 1^n, 01^{m-1})$$

$$(K := K_S, 1^n, 1^{m-1}0)$$



SISA RINGSHIFT PROGRAM

FIG. 6

The ROTV K subprogram rotates the contents of register K vertically upwards, where the topmost register is wrapped around.

ROTV K consist of two instructions which perform the following operations:

The first instruction K := K_N reads the contents of K of the north (above) neighbour and writes it into its own K.
The second instruction K := K_S reads the contents of K of the south (below) neighbour and writes it into its own K.

Here, the first instruction is not executed in the first row of processors since these processors don't have northern neighbours. In an analogous way, the second instruction is not executed in the last row of processors since these processors don't have southern neighbours.

Note that K is treated as a formal parameter in the above subprograms. Thus ROTH A rotates the contents of register A horizontally. Also, we will use the representation ROTH A,B to mean that we horizontally rotate two distinct registers A and B.

Remarks

1. Simultaneous execution of K := K_W and K := K_E in adjacent processors results in an exchange of the contents of the two communication registers due to the two phase execution of instructions.

2. The ringshift can be carried out in constant period, although there are no wrap-around connections between the processors. This illustrates the suitability of the SISA for solving certain problems where broadcasting is involved. However, ringshifting in the opposite directions can only be done in period $\Omega(n)$, where n is the number of rows (columns) in the processing array.

8.2 Time Complexity

We have a mxn SISA processing array, where m=n=4, and number of instructions = r=4.

Then, Period = r=4.

Execution time = m+n+r-2 = 10.

9. SISA and ISA Models - Their Relationship

The SISA is a restricted version of the ISA, where single instructions are pumped through the processors. A SISA program can be directly transformed to ISA by replacing each '1' in column selectors by the corresponding instruction, and each '0' by NO-OP (no operation instruction).

However, only certain ISA programs can be directly transformed to SISA programs. In these ISA programs, each instruction diagonal must have identical instructions other than NO-OP. This is quite frequently the case in most practical applications. Here, each instruction diagonal of identical instructions in a ISA can be replaced by a single instruction in a SISA.

Remark If k (where k > 1) different instructions other than NO-OP occur in one ISA instruction diagonal, Lang [Lang 1987] states (without proof) that in certain cases the original ISA instruction diagonal may be replaced by a sequence of these k instructions in the corresponding SISA program. The general transformation of ISA programs to SISA programs is currently being studied by the author. This is beyond the scope of the thesis and will be published elsewhere.

Having introduced the ISA, we now proceed to describe some of its important applications. In the next chapter, we consider the design of a ISA for distributed Chinese remaindering as well as polynomial and rational interpolation algorithms. The Chinese remaindering algorithm is a very basic algorithm that is applied to several problems to be described in the following chapters.

CHAPTER 2

INSTRUCTION SYSTOLIC ARRAYS FOR DISTRIBUTED CHINESE REMAINDERING/ INTERPOLATION ALGORITHMS

ABSTRACT

This chapter introduces a parallel / distributed algorithm for Chinese remaindering and the closely related polynomial interpolation. The implementation of this algorithm on a ISA using a top-down design technique is described. A generalization of the interpolation algorithm for multivariable interpolation is then presented. The multivariable interpolation algorithm can be realized in a ISA as well as in a pyramid architecture. Its implementation in the message passing language Occam is also described.

We then present a parallel / distributed algorithm for rational function interpolation. This algorithm constructs the continued fraction / Pade approximant from which the required rational interpolant can be obtained. The implementation of this algorithm on a ISA is then described.

CHAPTER 2

INSTRUCTION SYSTOLIC ARRAYS FOR DISTRIBUTED CHINESE REMAINDERING / INTERPOLATION ALGORITHMS

1. Introduction

This chapter describes parallel and distributed algorithms for Chinese remaindering, polynomial interpolation and rational function interpolation and their implementation in Instruction Systolic arrays (ISA).

First, we present a parallel / distributed algorithm for both Chinese remaindering (CRA) and the closely related polynomial interpolation over a field (real or finite); then we consider the design and implementation of a ISA for this algorithm. A generalization of this algorithm for multivariable interpolation is then described. This algorithm can be realized in a ISA as well as in a pyramid architecture.

We then describe the implementation of the multivariable interpolation algorithm in the message passing language Occam, which is a direct descendant of the Communicating Sequential Processes (CSP) language due to Hoare [Hoare 1985]. Such a description will be useful for the design of wavefront arrays [Kung 1988].

A closely related problem to polynomial interpolation is rational function interpolation. A parallel / distributed algorithm based on Thiele's reciprocal differences is described for this problem. This algorithm constructs the continued fraction / Pade approximant from which the required rational interpolant can be obtained to any desired degree of accuracy. The implementation of this algorithm on the ISA is also described.

For each of the above algorithms, we determine the period and execution time of ISA programs. This will be useful for deriving the required complexity measures for the VLSI implementation of the ISA.

2. Chinese Remaindering and Polynomial Interpolation

One of the oldest known algorithms for computing solutions to a simultaneous system of linear congruences over the integers makes use of a classical theorem from the theory of numbers, called the Chinese remainder theorem [Krishnamurthy 1985, Krishnamurthy et al 1987]. This algorithm has applications in coding theory, security systems [Shamir 1979, McEliece et al 1981], and exact rational computations [Krishnamurthy et al 1987]. A closely related problem is the polynomial interpolation over a field (real or finite), in which the values of a function at distinct points are given and one fits a polynomial whose evaluated values at these points coincide with the functional values.

From an algebraic point of view the Chinese remaindering and polynomial interpolation problems are equivalent. Given a set of remainders (residues) $\{r_0, r_1, \ldots r_n\}$ with respect to a set of moduli $\{p_0, p_1, \ldots p_n\}$ which are pairwise relatively prime, both problems reconstruct an element r such that $r_i = r \bmod p_i$ for i = 0, 1, ... n in respective domains; r is uniquely determined if

$$\text{size } r < \text{size } \prod_{i=0}^{n} p_i = M$$

where size is defined suitably thus: for integers 'the size' is the magnitude and for polynomials 'the size' is the degree. Element r is defined as

$$r = \sum_{i=0}^{n} (M/p_i)\, r_i\, T_i \bmod M$$

where T_i is the solution of $(M/p_i)\, T_i = 1 \bmod p_i$.

The equivalence between the Chinese remaindering algorithm and the Lagrange polynomial interpolation is readily seen from the following correspondence [Krishnamurthy 1985]:

$r(x)$ = n^{th} degree polynomial in a field F

$r(x_i) = r_i \quad (i = 0, 1, \ldots n)$

$p_i(x) = (x - x_i)$

$r(x) \bmod p_i(x) = r_i$

$$M = \prod_{i=0}^{n} p_i(x)$$

The Lagrange interpolant is obtained from

$$L_n(x) = \sum_{i=0}^{n} r_i\, T_i \prod_{\substack{k=0 \\ k \neq i}}^{n} (x - x_k)$$

where

$$T_i = \frac{1}{\prod_{\substack{k=0 \\ k \neq i}}^{n} (x_i - x_k)} = [M/p_i(x)]^{-1} \bmod (x - x_i) \quad (M \text{ and } p_i(x) \text{ are defined above})$$

Since in doing polynomial interpolation, we restrict ourselves to linear polynomials $(x - x_i)$ $(i=0,\ldots n)$, the remainder or residue computation for a polynomial r(x) is equivalent to

$r(x) \bmod (x-x_i) = r_i$ by the remainder theorem.

Thus, the reconstruction of an integer r in the range $0 \leq r \leq M-1$ from a set of residues $\{r_0, r_1, \ldots r_n\}$ with respect to a set of primes $\{p_0, p_1, \ldots p_n\}$ and the reconstruction of a polynomial r(x) of degree atmost n from a set of residues $\{r_0, r_1, \ldots r_n\}$ with respect to a set of distinct linear polynomials $\{p_0(x), p_1(x), \ldots p_n(x)\}$ can be achieved by identical algorithms for suitably defined data domains.

We now formally state the sequential algorithm.

2.1 Sequential Algorithm

The sequential algorithm given below takes as inputs residues $\{r_0, r_1, \ldots r_n\}$ and moduli $\{p_0, p_1, \ldots p_n\}$. For integers, r is reconstructed such that $r \bmod p_i = r_i$ where r is over modulo M, $M = \prod_{i=0}^{n} p_i$. For polynomials, r(x) is reconstructed such that $r(x) \bmod p_i = r_i$ where $p_i = x - x_i$; here, r(x) is a nth degree polynomial over field F.

Input: residues r_i and moduli p_i $(i = 0, 1, \ldots n)$.

Output: Reconstructed element r

M := 1;

R := r_0;

```
for k =1 to n do
begin
   M := M p_{k-1};
   u := M^{-1} mod p_k;
   d := (r_k - R)u mod p_k;
   R := R + dM
end;
r := R;
```

This algorithm expresses r in the form $r = d_0 + d_1p_0 + ... + d_np_0...p_{n-1}$, where $d_0 = r_0$.

For integers, r is in mixed-radix form where d_i's are the mixed-radix digits and $0 \leq d_i \leq p_i - 1$ $(i = 0, 1, ... n)$.

For polynomials, $r(x) = d_0 + d_1(x-x_0) + ... + d_n(x-x_0)...(x-x_{n-1})$ is in the Newton's Interpolation form, where $p_i = (x - x_i)$.

We now consider how the above algorithm can be converted to a parallel / distributed form.

3. Parallel / Distributed Algorithm for Chinese Remaindering/ Interpolation

We now present a parallel / distributed algorithm for Chinese remaindering and interpolation based on Newton's divided differences [Scheid 1970].

Let the list $\{X_k\}$ denote the list $\{x_0, x_1,...x_k\}$ and

the list $\{X_k, x\}$ denote $\{x_0, x_1,...x_k, x\}$ which is obtained by appending to $\{X_k\}$ the element x. The algorithm starts with the basis $D\{X_0\} = D(x_0) = r_0 = d_0$.

Then, the algorithm computes $D\{X_i\} = d_i$ using the following parallel recursion (on j):

$$D\{X_i, x_{i+j}\} = (D\{X_{i-1}, x_{i+j}\} - D\{X_i\}) / (x_{i+j} - x_i)$$

for a specified i (i = 0, 1,...n-1) and for all j = 1, 2,...(n-i).

Note that $\{X_{i-1}\}$ for i = 0 is a null list.

This algorithm can be programmed in a parallel machine (such as the Inmos Transputer) using a parallel and concurrent programming language such as Occam or Ada. We can also realize this algorithm in systolic architectures such as Instruction Systolic arrays. The design and implementation of such programs will be discussed later in this chapter.

This algorithm carries out computation in each specified field in parallel by distributing the computation among several processors; however, the d_i's depend on the values from the previous iterations, and have to be necessarily generated sequentially.

In the description of this parallel algorithm, we do not constrain ourselves to a particular parallel computational model. Suitable modifications, however, can be easily incorporated to suit a particular computational model.

3.1 Algorithm

This algorithm takes as inputs residues $\{r_0, r_1, \ldots r_n\}$ and moduli $\{p_0, p_1, \ldots p_n\}$.

For integers, r is reconstructed such that $r \bmod p_i = r_i$ where r is over modulo M, $M = \prod_{i=0}^{n} p_i$.

For polynomials, r(x) is reconstructed such that $r(x) \bmod p_i = r_i$ where $p_i(x) = x - x_i$; here, r(x) is a nth degree polynomial over field F.

We assume that there are n+1 processors PRO(i) (i = 0, 1,... n), each with five registers R_i, D_i, S_i, P_i and M_i. The operations - subtraction, multiplication, and inversion are carried out in the appropriate field for each processor.

Register R_i is initialized with r_i and register P_i with p_i (for polynomials, register X_i is initialized with x_i). At the end of the algorithm, register D_i in each processor contains the coefficient d_i,where $d_0 = r_0$. We can then reconstruct r using D_i.

Input: residues r_i and moduli p_i (i = 0, 1, ... n).

Output: Reconstructed element r

For i = 0 to n do

begin

$R_i := r_i$;

$P_i := p_i$;

end;

For j = 0 to n do

begin

$D_j := R_j$;

For s = j+1 to n in parallel do

begin

$S_S := R_j$;

$R_S := R_S - S_S$;

$M_S := P_j^{-1} \bmod P_S$;

$R_S := R_S M_S \bmod P_S$;

end;

end;

Then $r := D_0 + \sum_{j=1}^{n} D_j \prod_{s=0}^{j-1} P_s$

In the case of polynomials, we have $P_j = (x - X_j)$, $P_S = (x - X_S)$. The last three assignment statements in the above algorithm then are:

$M_S := (X_S - X_j)^{-1}$;

$R_S := R_S M_S$;

$$r := D_0 + \sum_{j=1}^{n} D_j \prod_{s=0}^{j-1} x - X_S$$

Remarks

1. In the last iteration of j, when j is n, 'for s = j+1 to n in parallel do begin' is not executed since j+1 equals n+1, and exceeds the limit n.

2. Computing r using the above formula can be deferred until one wants to evaluate the polynomial at some point.

3.2 Proof of Algorithm

We now provide a proof of correctness of the algorithm.

3.2.1. Chinese Remaindering

In order to reconstruct the integer r we assume that it is expressed uniquely in the mixed-radix form

$r = d_0 + d_1p_0 + ... + d_np_0...p_{n-1}$, where $0 \leq d_i \leq p_i - 1$ $(i = 0, 1, ... n)$ where d_i 's are the mixed-radix digits. This algorithm essentially determines the d_i 's using r_i and p_i.

For convenience, let $R_k = d_0 + d_1p_0 + ... + d_kp_0...p_{k-1}$.

We then have

$r \bmod p_0 = r_0 = d_0$

$r \bmod p_1 = r_1 = d_0 + d_1p_0$

...

$r \bmod p_k = r_k = R_k$

$r \bmod p_n = r_n = r$

Thus, $d_1 = (r_1 - d_0) \ p_0^{-1} \mod p_1$.

Similarly, $d_2 = (r_2 - (d_0 + d_1 p_0)) (p_0 p_1)^{-1} \mod p_2$

or $d_2 = (r_2 - R_1) (p_0 p_1)^{-1} \mod p_2$

and in general $d_k = (r_k - R_{k-1}) (p_0 \ldots p_{k-1})^{-1} \mod p_k$

Note that $r < M = \prod_{i=0}^{n} p_i$; thus, $d_i = 0$ for $i \geq n+1$.

3.2.2. Polynomial Interpolation

For polynomials, we replace r by r(x). We assume that r(x) is a nth degree polynomial expressed uniquely by the Newton's Interpolation formula,

$r(x) = d_0 + d_1(x-x_0) + \ldots + d_n(x-x_0)\ldots(x-x_{n-1})$.

The above proof can be directly extended to polynomial interpolation.

Thus, $d_i = 0$ for $i \geq n+1$.

3.3 Polynomial Evaluation

The Newton's interpolation formula is very convenient for polynomial evaluation. Since

$r(x) = d_0 + d_1(x-x_0) + \ldots + d_n(x-x_0)\ldots(x-x_{n-1})$ we can evaluate r(x) from the Newton's coefficients d_i stored in the registers by using Horner's nested multiplication rule.

4. Examples

We now illustrate the above algorithm using examples.

4.1 Chinese Remaindering

Let $p_i = \{5,7,11,13\}$, $r_i = \{1,5,9,11\}$. Table I illustrates the process of reconstructing an integer.

4.2 Single Variable Interpolation - Finite Field

Let $x_0 = 1$, $r_0 = 8$; $x_1 = 2$, $r_1 = 3$; $x_2 = 3$, $r_2 = 0$; $x_3 = 4$, $r_3 = 10$. Table 2 illustrates the reconstruction of the polynomial over the prime field modulo 11.

4.3 Single Variable Interpolation - Real Field

Let $x_0 = 1$, $r_0 = 0.5$; $x_1 = 2$, $r_1 = 2.5$; $x_2 = 3$, $r_2 = 6.5$; $x_3 = 0.5$, $r_3 = 0.25$. Table 3 illustrates the reconstruction of the polynomial over the real field.

5. ISA Implementation of Polynomial Interpolation/ Evaluation

We now consider the implementation of the above interpolation algorithm on the ISA, using a systematic top-down design technique. The time complexities for the ISA polynomial interpolation and evaluation programs are indicated.

The systematic top-down design process for a ISA is a generalized version of the design approach for systolic arrays [Kung 1987]. We start with a locally recursive version of the above algorithm and generate the dependence graph. A geometric

to 11a

Table 1 – Chinese Remaindering

	P_0 mod 5	P_1 mod 7	P_2 mod 11	P_3 mod 13	d_i	P
r_i	1	5	9	11	$d_0 = 1$	$1 + 5 \cdot 5 + 8 \cdot 5 \cdot 7 +$ $7 \cdot 5 \cdot 7 \cdot 11$ $= 3001$
$S_s := R_j$		1	1	1		
Subtract		4	8	10		
$M_s = 5^{-1}$		3	9	8	$d_1 = 5$	
$M_s \cdot R_s$		5	6	2		
$S_s := R_j$			5	5		
Subtract			1	10		
$M_s = 7^{-1}$			8	2	$d_2 = 8$	
$M_s \cdot R_s$			8	7		
$S_s := R_j$				8		
Subtract				12		
$M_s = 11^{-1}$				6	$d_3 = 7$	
$M_s \cdot R_s$				7		

Table 2 – Single Variable Interpolation

	P_0 $x_0 = 1$	P_1 $x_1 = 2$	P_2 $x_2 = 3$	P_3 $x_3 = 4$	d_i	P
r_i	8	3	0	10	$d_0 = 8$	$8 + 6(x-1) + 1(x-1)(x-2)$ $= x^2 + 3x + 4$
$S_s := R_j$		8	8	8		
Subtract		6	3	2	$d_1 = 6$	
$(x_s - x_0)^{-1}$		1	6	4		
Multiply		6	7	8		
$S_s := R_j$			6	6		
Subtract			1	2	$d_2 = 1$	
$(x_s - x_1)^{-1}$			1	6		
Multiply			1	1		
$S_s := R_j$				1		
Subtract				0	$d_3 = 0$	

Table 3 – Single Variable Interpolation

	P_0 $x_0 = 1$	P_1 $x_1 = 2$	P_2 $x_2 = 3$	P_3 $x_3 = 0.5$	d_i	P
r_i	0.5	2.5	6.5	0.25	$d_0 = 0.5$	$0.5 + 2(x-1) + 1(x-1)(x-2)$ $= x^2 - x + 0.5$
$S_s := R_j$		0.5	0.5	0.5		
Subtract		2	6	-0.25		
$(x_s - x_0)^{-1}$		1	0.5	-2	$d_1 = 2$	
Multiply		2	3	0.5		
$S_s := R_j$			2	2		
Subtract			1	-1.5		
$(x_s - x_1)^{-1}$			1	-0.666	$d_2 = 1$	
Multiply			1	1		
$S_s := R_j$				1		
Subtract				0	$d_3 = 0$	

layout of the dependence graph is then projected onto an array of processors using the signal flow graph concept [Kung 1988]. In a ISA, projections of the dependence graph in different directions are permitted. This allows optimization of the implementation under several parameters, including execution time.

We now convert the parallel interpolation algorithm to a locally recursive single-assignment algorithm.

5.1 Locally Recursive Single-assignment Interpolation Algorithm

We first recall some important definitions [Kung 1988].

a. A *recursive* algorithm consists of a set of recursive equations or recurrences. Here, a variable will appear on both the left-hand side and right-hand side of the equation with different indices, e.g., the occurrence of variable X in $X_{i+1} = X_i + 5$.

b. A *locally recursive* algorithm is a recursive algorithm in which the index seperations in the left and right hand sides are within a specified limit, e.g., $X_{i+k} = X_i + 5$ for $1 \leq k \leq N$ where N is a small fixed integer permitting local communication in a specified architecture.

A *single-assignment* algorithm consists of a set of single-assignment statements. In a single assignment statement a variable may appear on the left-hand side of an assignment only once within its scope and this same variable cannot appear on both sides of an assignment statement, e.g., x := x + 1 is not a single-assignment

statement. Thus, assignments take the form of equalities in the conventional mathematical sense.

Remark The single-assignment rule shows the data dependencies clearly, and is a key feature in data-flow programming languages [Ackermann 1982].

We now describe the recursive single-assignment form of the interpolation algorithm in section 3.1. In this algorithm, we have introduced additional indices (subscripts) in variables to satisfy the single-assignment rule.

```
For i = 0 to n do
begin
      R_i0 := r_i;
      X_i := x_i;
end;
For j = 0 to n do
begin
   D_j := R_j,2j
   For k = j+1 to n in parallel do
   begin
         S_k,2j := R_j,2j;
         R_k,2j+1 := R_k,2j - S_k,2j ;
         M_kj := 1/(X_k - X_j);
         R_k,2j+2  := R_k,2j+1 M_kj ;
    end;
end;
```

The dependence graph for this algorithm is presented in the next section.

$$\text{Then } r := D_0 + \sum_{j=1}^{n} D_j \prod_{k=0}^{j-1} x - X_k$$

The ISA implementation of the actual evaluation or reconstruction of a polynomial using the above formula is considered in section 5.5.

5.2 Dependence Graph

The concept of dependence graphs was first used by Karp et al [Karp et al 1967] in their system of uniform recurrence equations. They also proposed a graph theoretic approach to analyse locally recursive algorithms.

A dependence graph (DG) is a directed graph that shows the data dependencies of computations that occur in an algorithm. A DG can be considered to be the graphical representation of a single-assignment algorithm [Kung 1988].

The nodes in a DG represent computations and the arcs specify the data dependencies between computations. A dependence graph is constructed based on the spatial index of the various processors (nodes) in the array and the time steps (time index) of execution of the different instructions. In this sense, the DG represents the topological adjacency - both spatial and temporal - between the processors and the events occurring in them at different time steps. In other words, a DG is embedded in a space-time index space hereinafter referred to as the index space.

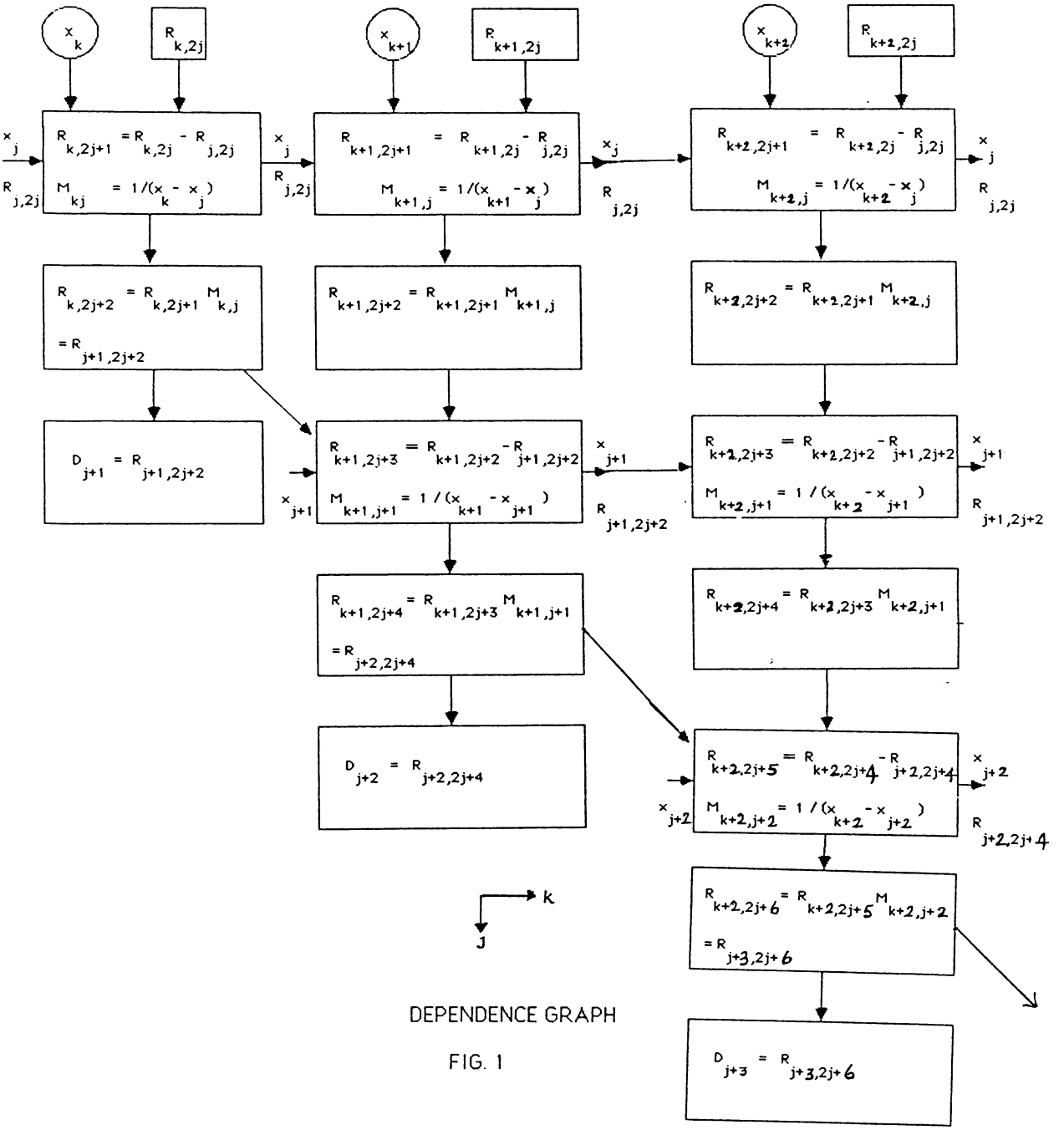
Remark The DG is analogous to a choreograph in a ballet where the spatial and temporal movements of the actors are represented.

There have been different approaches to construct valid (correct) and optimal systolic implementations from dependence graphs [Karp et al 1967, Moldavan 1983, Mongenet et al 1987]. Mongenet et al introduce transformations in the space-time index space to ensure that coefficients/variables are not used before they are available; this is the constraint for scheduling.

We now introduce the concept of shift invariance. A dependence graph is said to be *shift invariant* or uniform with respect to an axis in the index space, if the dependence arcs corresponding to all nodes in the index space are independent with respect to the node positions.

The structure of the dependence graph for the above interpolation algorithm is shown in Fig. 1, where index k is the horizontal axis and index j is the vertical axis (downwards). We show a typical set of adjacent computational streams which generate successive coefficients D_{j+1}, D_{j+2}, D_{j+3}, where $k = j + 1$. Hence, $R_{k,2j+2} = R_{j+1,2j+2}$ and $R_{k+1,2j+4} = R_{j+2,2j+4}$. In the DG, the circles represent input values, e.g., x_k is one of the input points, and the boxes represent the nodes.

In the systolic implementation described below, the equitemporal lines (lines of equal time) are along the south-west to north-east diagonal, i.e., the time axis is along the north-west to south-east diagonal. Hence, with respect to the time axis, the DG will appear to be skewed.



DEPENDENCE GRAPH

FIG. 1

It should also be noted that the axis of shift invariance for the DG is along the diagonal from the north-west to the south-east.

The next step in the design process is to map the nodes of a dependence graph to a processing array. There are several ways to achieve such a mapping. One method is to map each node in a DG to one processing element (PE). However, this does not result in an efficient utilization of processing elements since each PE is active only for a small time. We can improve the PE utilization by mapping nodes of the DG to a smaller number of processing elements. In order to achieve this, we first map the DG to an intermediate form known as the Signal Flow Graph.

Remark

In designing ISA programs, we have several choices for the direction of projection of a dependence graph since each processing element is capable of executing more than one instruction; thus, the projection direction need not necessarily be along the axis of shift-invariance. This permits us to optimize different parameters in the implementation, e.g., execution time or the number of processors.

5.3 Signal Flow Graphs

The Signal Flow Graph (SFG) is a directed graph whose nodes correspond to the processing elements and arcs correspond to the inter-processor communicating edges. The SFG is more specific than the DG since the SFG is at a lower level of representation and corresponds to the hardware level design. The SFG represents

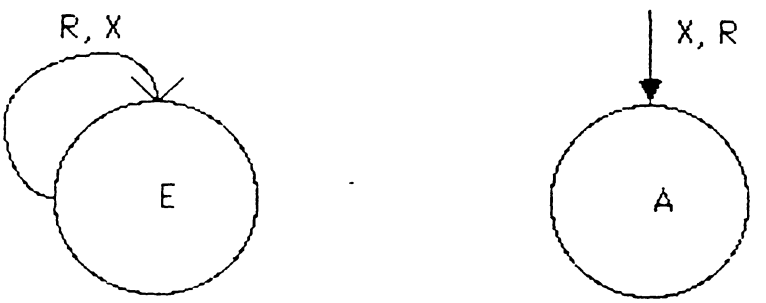
the activities taking place in ISA processing elements at some instant of time. The node labels indicate the instruction at that processing element and the input arc labels (including self-loop arc labels) specify where an instruction gets its inputs from. For instance, an arc from the left node indicates that inputs are obtained from registers of the left neighbour processing element while a self-loop indicates that the inputs are obtained from registers of the same processing element.

The ISA implementation described here uses a vertical projection of the DG, which is not along the direction of shift invariance (i.e., the diagonal from the north-west to the south-east). The main advantage of the design obtained from the vertical projection as compared to the horizontal projection is that less communication is required among the processors. However, each processor must be capable of executing all instructions mentioned below including the division instruction. If a horizontal projection is used, it is not necessary for each processor to have the capability to execute all the instructions. That is, we can have specialized processors, and assign the time-consuming division operation to a particular processor.

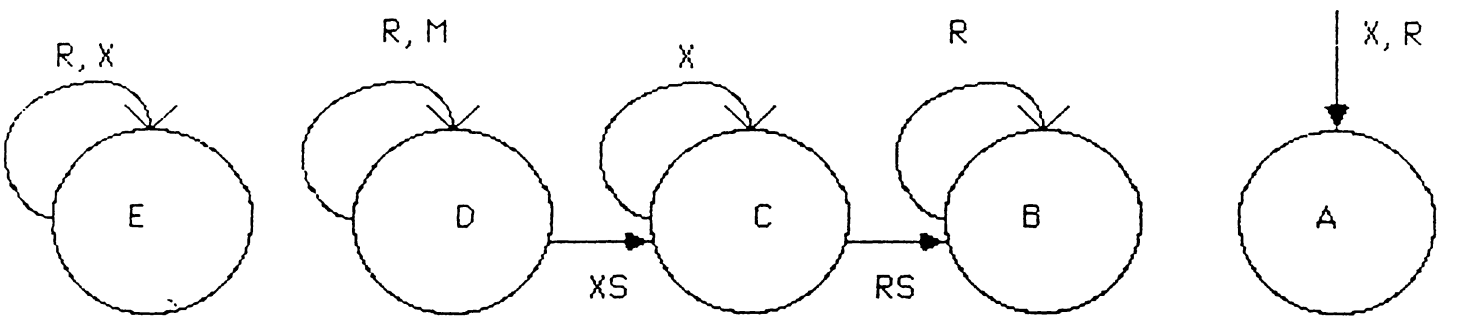
For the interpolation DG, the vertical projection results in two different Signal Flow Graphs which have to alternately applied, E-A and E-D-C-B-A (see Fig. 2); here, A, B, C, D and E are the instructions defined in the next section.

We now consider the ISA program for the locally recursive interpolation algorithm.

5.4 ISA Program for Interpolation



E-A SIGNAL FLOW GRAPH



E-D-C-B-A SIGNAL FLOW GRAPH

FIG. 2

We have a 1 x (n+1) array of n+1 processing elements (PE) in the ISA, P_0,..., P_n.

Each processing element in the ISA has 6 data registers:

X stores a point x_i

XS shifts x_i

R stores a residue r_i

RS shifts r_i

M stores M_{kj}

D stores D_i coefficients

We have five instructions in the instruction set, and each PE can execute all the instructions (subscripts L and T refer to the left and top neigbours respectively). The five instructions in the ISA program are:

A. $X := X_T,\ R := R_T$

B. $R := R - RS_L,\ RS := RS_L$

C. $M := 1/ (X - XS_L),\ XS := XS_L$

D. $R := R\ M$

E. $D := R,\ RS := R,\ XS := X$

The ISA interpolation program for four PE P_0, P_1, P_2 and P_3 (n = 3) is shown in Fig. 3. Note that the selectors (not shown) in a 1-dimensional ISA are always '1'. The input data is the set of residues r_i at points x_i (i = 0,..., n). At the end of the program, register D in each PE P_i contains value D_i.

We observe that the pattern of instructions executed in each processing element P_i is $A(BCD)^i E$, i.e., the sequence of instructions BCD is repeated i times in P_i. We can generate all possible combinations of instructions that are executed at the same time (on different PE), viz. instructions on the same horizontal line, using the following expression:

L = [(e, E, CB, B) $(DCB)^*$ (D, DC, A, e)] U C where 'e' is the empty string, 'U' is the union operation of the set of strings and '*' represents repetition of the string zero or more times.

Remark It is not necessary to have register D for the interpolation program since the D_i coefficients are also stored in R. However, register D is necessary for the evaluation program below.

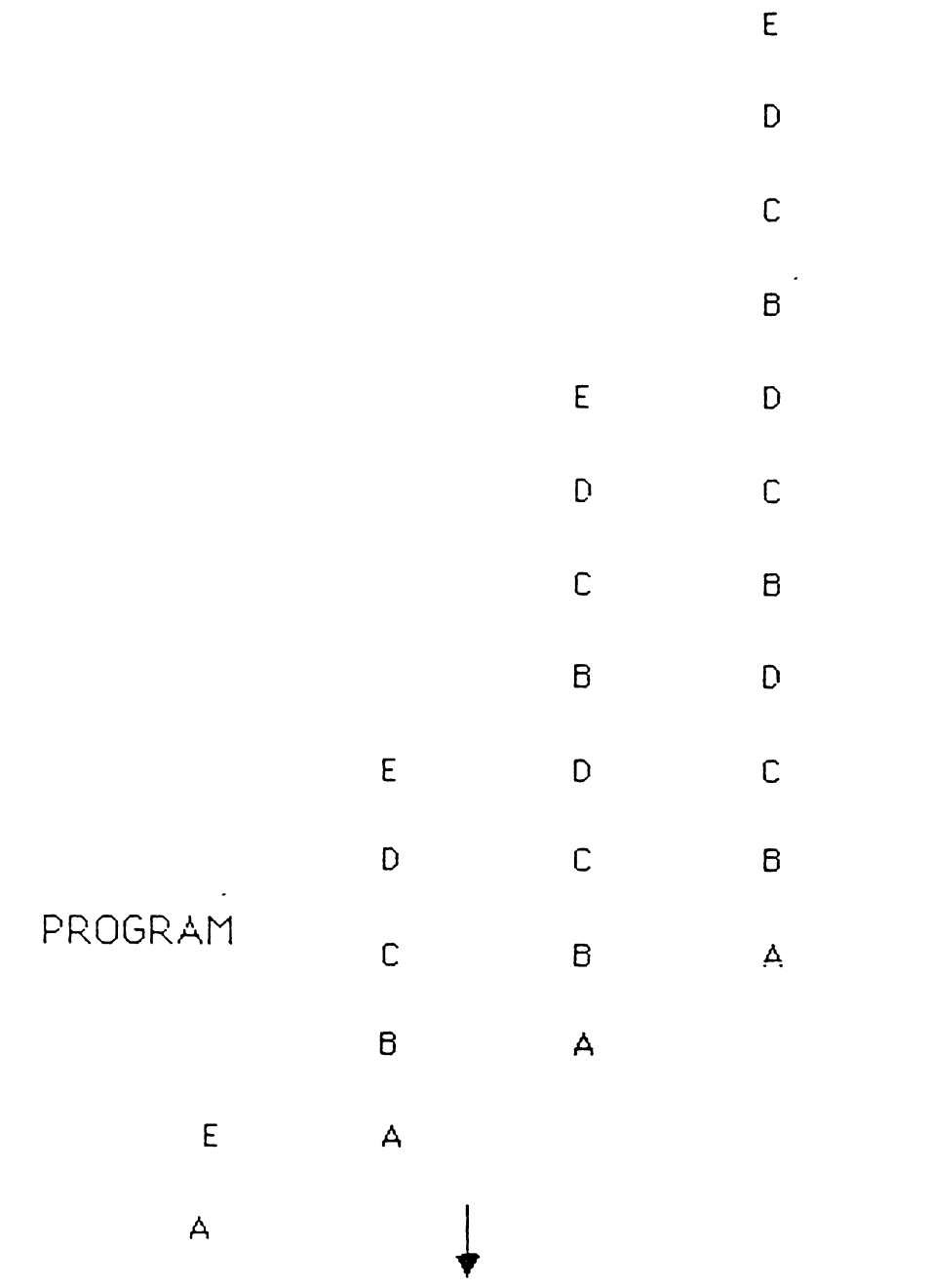
5.4.1 Time Complexity

Here, we have a (m x k) ISA, where m = 1 and k = 4.

The period of the ISA program = r = number of instruction diagonals = 3(k-1)+2 = 11.

The execution time of the ISA program = r + k + m - 2 = 14.

5.5 ISA program for Polynomial Evaluation



ISA POLYNOMIAL INTERPOLATION PROGRAM

FIG. 3

We now describe a ISA polynomial evaluation program that runs on the same 1 x (n+1) processing array used for polynomial interpolation. The polynomial is evaluated at some point y as follows (X_k denotes register X in PE P_k):

$$r := D_0 + \sum_{j=1}^{n} D_j \prod_{k=0}^{j-1} y - X_k$$

Remark This program can also be used to construct the nth degree polynomial interpolant r (in x) using the D_j coefficients:

$$r := D_0 + \sum_{j=1}^{n} D_j \prod_{k=0}^{j-1} x - X_k$$

The evaluation program can be concatenated immediately after the interpolation program, as it uses the D_j coefficients (stored in the D registers) produced by the interpolation program. Also, register X in each PE P_k contains the point x_k.

The program consists of only one instruction EV. The input data is value y, the point where the polynomial is to be evaluated. This value is shifted through the processors using register XS. The program is initialized with the following values: $R_L = 0$, $M_L = 1$, $XS_L = y$ (point of evaluation); these values are used when EV is executed in PE P_0.

EV. $XS := XS_L$, $M := M_L(XS_L - X)$, $R := R_L + D\, M_L$

The ISA program for n = 3 (4 PE) is shown in Fig. 4; the selectors are not shown.

The execution of instruction EV does not affect registers D and X. Thus, the period of this algorithm is 1, and a new point can be input at each time unit. As mentioned above, register D is necessary for the evaluation program since D_i coefficients stored in registers R are overwritten.

It is possible to reduce the number of multiplications by about half if Horner's nested multiplication rule is used during the evaluation. However, this would change the direction of the instruction diagonal to a perpendicular direction, viz. from north-west to south-east (in contrast, the instruction diagonal for interpolation is from south-west to north-east). This will cause delay if the evaluation program is concatenated immediately after the interpolation program.

5.5.1 Time Complexity

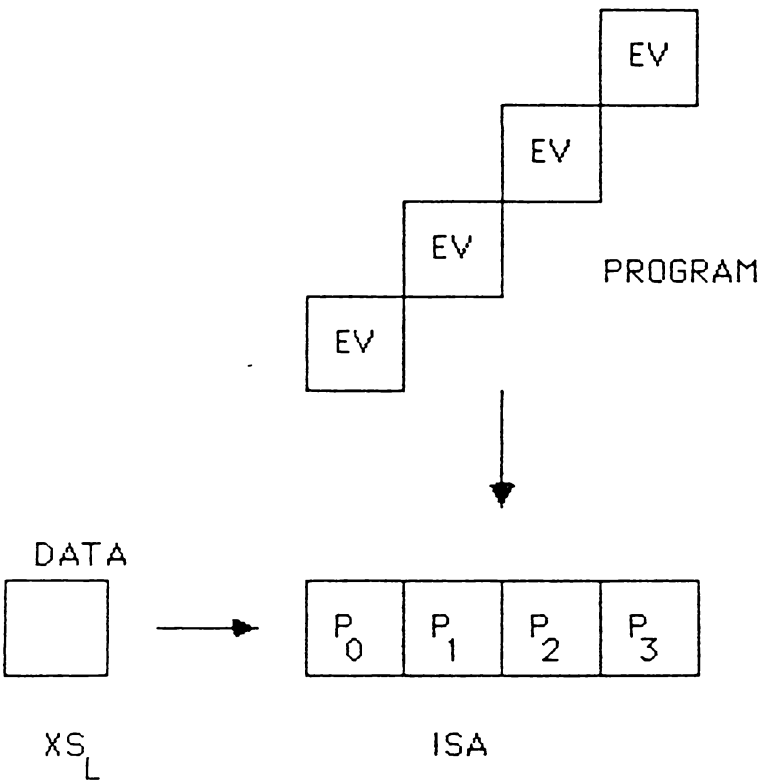
The period of the ISA program = r = 1 (number of instruction diagonals).

The execution time for a (m x k) processing array = r + k + m - 2.
Here, m = 1 and k = 4. Hence, execution time = 4.

6. Parallel / Distributed Multivariable Interpolation

This section indicates how we can extend the above parallel / distributed interpolation algorithm to multivariable interpolation.

6.1 Algorithm



ISA POLYNOMIAL EVALUATION PROGRAM

FIG. 4

We first consider the two-variable case and then the N-variable case.

In the two-variable case, let x,y be the variables. Let r(x,y) be specified at (m + 1)(n + 1) distinct points x_i (i = 0,1,...m), y_j (j = 0,1,...n) with the values $r(x_i, y_j) = r_{ij}$.Then we can fit a polynomial with highest degree $x^m y^n$. The parallel algorithm described earlier can be used twice for carrying out this interpolation: once for interpolating $r(x_i,y) = r_i(y)$ for each x_i (i = 0,1,...m) using the values of $r(x_i, y_j)$ (j = 0,1,...n); in the second step we reconstruct r(x,y) using $r_i(y)$'s as inputs; these $r_i(y)$'s parametrically define the residues for each x_i.

In the N-variable case, let $x_1,x_2,...x_N$ be N variables where the maximum degree of each variable is d. Let $r(x_1,x_2,...x_N)$ be specified at $(d + 1)^N$ distinct points x_{1i} (i = 0,1,...d), x_{2j} (j = 0,1,...d),...x_{Nk} (k = 0,1,...d).

Remark This algorithm assumes that the values of the interpolatory function are available at $(d+1)^N$ distinct points in a N-dimensional grid to fit a polynomial of degree atmost d in each variable. Otherwise, the interpolating procedure is not applicable.

At the first stage of interpolation, we use $(d + 1)^{N-1}$ sets of processors, each set consisting of (d + 1) processors taking (d + 1) inputs. All the $(d + 1)^{N-1}$ sets of processors act in parallel, yielding $(d + 1)^{N-2}$ parametric residues in x_N. At the second stage, we need $(d + 1)^{N-2}$ sets of processors yielding $(d + 1)^{N-3}$ parametric residues in x_N and X_{N-1}; again, each set consists of (d + 1) processors taking (d + 1) inputs. Finally, the N th stage consists of one set of (d + 1) processors taking (d + 1) inputs and yielding $r(x_1,x_2,...x_N)$.

The total number of processors in all N stages

$= (d + 1)^N + (d + 1)^{N-1} + ... + (d + 1) = ((d + 1)/d)\,[(d + 1)^N - 1] = O((d + 1)^N)$.

Thus, we can perform multivariable interpolation using $O((d + 1)^N)$ processors.

A suitable configuration of processors is a pyramid or tree-like architecture. Here, we have a flow of data through the processors from the base to the apex. Note that interpolation at the first stage requires more processors than at higher stages; however, only number processing capability is required for processors in the first stage. The number of processors decreases exponentially in the higher stages; however, each processor in a successive stage requires a greater capability to handle one extra variable for polynomial processing.

The setup for massively parallel multivariable interpolation is realizable in both VLSI and Optical technology [Uhr 1984]. We can also extend the ISA program for interpolation described above to realize multivariable interpolation.

6.1.1 Example

We now consider the interpolation of a three variable function $r(x_1,x_2,x_3)$ over a finite field modulo prime 11. The values of the function are given in Table 4. The fitted polynomial will have a maximum degree of one in each variable.

In this example, N =3 and d =1, we have a tree with three levels, where each box is a set of two processors which takes two inputs (see Fig. 5). The base (level 1) consists of 4 sets of processors, level 2 consists of 2 sets of processors and level

to 23a

Table 4

x_1	x_2	x_3	r
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	3
1	0	1	4
1	1	0	5
1	1	1	6

Table 5

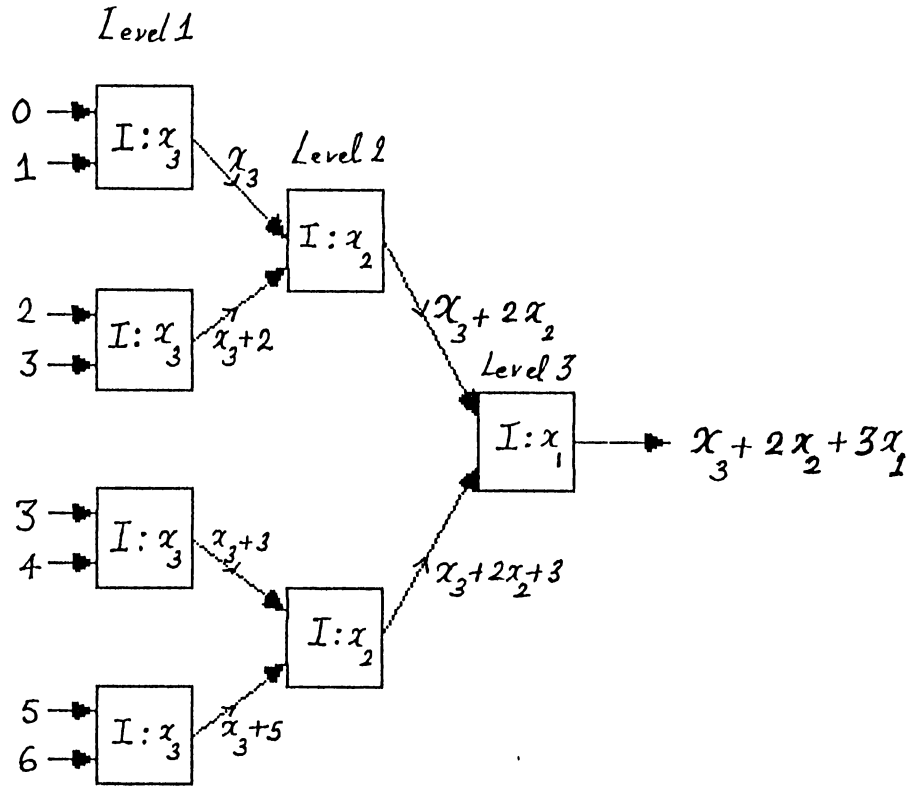
	P_{01}	P_{11}	P_1	P_{02}	P_{12}	P_2	P_{03}	P_{13}	P_3	P_{04}	P_{14}	P_4
	r $(0,0,0)$	r $(0,0,1)$	r $(0,0,x_3)$	r $(0,1,0)$	r $(0,1,1)$	r $(0,1,x_3)$	r $(1,0,0)$	r $(1,0,1)$	r $(1,0,x_3)$	r $(1,1,0)$	r $(1,1,1)$	r $(1,1,x_3)$
$S_s := R_j$	0	1 0		2	3 2		3	4 3		5	6 5	
Subtract		1	$0 + x_3$		1	$2 + x_3$		1	$3 + x_3$		1	$5 + x_3$
M_s		1			1			1			1	
Multiply		1			1			1			1	

Table 6

	P_{01}	P_{11}	P_1	P_{02}	P_{12}	P_2
	$r(0,0,x_3)$	$r(0,1,x_3)$	$r(0,x_2,x_3)$	$r(1,0,x_3)$	$r(1,1,x_3)$	$r(1,x_2,x_3)$
	x_3	$x_3 + 2$		$x_3 + 3$	$x_3 + 5$	
$S_s := R_j$		x_3			$x_3 + 3$	
Subtract		2			2	
M_s		1	$x_3 + 2x_2$		1	$x_3 + 3 + 2x_2$
Multiply		2			2	

Table 7

	P_{01}	P_{11}	P_1
	$r(0,x_2,x_3)$	$r(1,x_2,x_3)$	$r(x_1,x_2,x_3)$
$S_s := R_j$	$x_3 + 2x_2$	$x_3 + 3 + 2x_2$ $x_3 + 2x_2$	
Subtract		3	
M_s		1	$x_3 + 2x_2 + 3x_1$
Multiply		3	



PROCESS NETWORK TOPOLOGY
FOR MULTIVARIABLE INTERPOLATION

FIG. 5

3 consists of 1 set of processors. At level 1, we interpolate for variable x_3 denoted by I:x_3 in Fig. 5 using the values specified in table 4. At level 2, we interpolate for variable x_2 denoted by I:x_2 in Fig. 5. Finally, at level 3, we interpolate for variable x_1 denoted by I:x_1 in Fig. 5.

In level 1, we interpolate for variable x_3 for specified values of x_1 and x_2 (see Table 5). That is, we find $r(0,0,x_3)$, $r(0,1,x_3)$, $r(1,0,x_3)$ and $r(1,1,x_3)$ in parallel from the specified values. In level 2, we use $r(0,0,x_3)$, $r(0,1,x_3)$, $r(1,0,x_3)$ and $r(1,1,x_3)$ as inputs and interpolate for x_2 (Table 6), to obtain $r(0,x_2,x_3)$ and $r(1,x_2,x_3)$. In level 3, we use $r(0,x_2,x_3)$ and $r(1,x_2,x_3)$ as inputs and interpolate for x_1 (Table 7).

6.1.2 Verifying Correctness

We can verify correctness by introducing one more extra input to indicate the fixed point. For example, a quadratic function needs only three values at three distinct points; if we have one more distinct data point, the final function remains invariant corresponding to the fixed point, i.e., the d_i coefficients become zero when i is greater than the degree of the polynomial.

The next section describes the Occam simulation of a pyramid for multivariable interpolation.

6.2 Occam Simulation of a Pyramid for Multivariable Interpolation

We now describe the Occam simulation of a Pyramid for distributed multivariable interpolation [de Vel et al 1988]. A simulated pyramid with no overlap is used as

the processor network. Here, each level consists of sets of communicating processors in a linear pipeline linked to one parent processor, where each processor in the pipeline is linked to a set of children processors. The major advantage of such a pyramidal topology is that it minimizes the communication overheads and maximizes the computational throughput. The processors at each level of the pyramid evaluate the parametric residues for a given variable and forward the cumulative polynomial up to the parent level.

A skeletal program listing for multivariable interpolation over a finite field is given below. We assume that there are N levels in the pyramid, where N is the number of variables. At each level k of the pyramid, there are $(d + 1)^{N-k}$ sets of processors, each set consisting of (d + 1) interpolator processors 'crt' organized as a linear pipeline and taking (d + 1) inputs. At the end of each pipeline, the calculator processor 'calc' executes the multiplication step and forms the cumulative polynomial, which is then forwarded to the next level. All the interpolating and calculator processors operate in parallel. In the above example, N = 3 and d = 1, (section 6.1.1), each box in Fig. 5 performs a pipeline of two interpolating processes and a calculator process.

Due to restrictions in the dynamic creation of processes in Occam, we cannot specify pyramid configuration parameters at run-time. Such parameters can be modified only at compile-time.

Remark Occam is a suitable language for programming wavefront arrays and it can provide a specification for the design of wavefront arrays [Kung 1988]. The Occam simulation will thus be useful for designing wavefront arrays for multivariable interpolation.

Occam Implementation of the Interpolation Algorithm

```
-- PROGRAM Distributed multivariable interpolation.
-- Implementation of the distributed algorithm for multivariable interpolation
-- over a finite field. A modified pyramid with no overlap is used as the
-- processor topology.

-- Number of variables.
DEF num.variables = 3:
-- Degree of each variable.
DEF var.degree = 1:
-- Value of prime field.
DEF field = 11:
-- Length of polynomial = (var.degree + 1)^num.variables.
DEF polylength = 8:
-- Total number of processes = sum of (var.degree + 1) × (each entry in 'lev').
DEF totprocs = 14:
-- Number of interpolator processes in each set at each level.
DEF lev = TABLE[4, 2, 1]:
-- Cumulative number of interpolator processes in each set at each level.
DEF cumlev = TABLE[4, 6, 7]:

VAR i, j, k:
-- Define I/O and pyramid channels.
CHAN varin AT 3, varout AT 11, polyin AT 4, polyout AT 12:
CHAN din [totprocs + 1], res[(totprocs + cumlev[(num.variables - 1)])]:

PROC write(CHAN to, VALUE number)=
  -- Process to write a number to a channel.

PROC open.files=
  -- Process to open files.

PROC read.data(CHAN dataout [])=
  -- Process to read variable and function values from a file.

PROC write.data(CHAN pchan)=
  -- Process to write polynomial to a file.

PROC shift.data.in(CHAN link, VAR poly [],vals [], VALUE level,power)=
  -- Process to input variables and polynomial from a process.
  VAR i:
  SEQ
    SEQ i = [1 FOR (num.variables - level)]
      link?vals[i - 1]
    SEQ i = [0 FOR power]
      link?poly[i]:
```

```
PROC shift.data.out(CHAN link,VAR poly [],vals [],VALUE level,power)=
  -- Process to output variables and polynomial to a process.
  VAR i:
  SEQ
    SEQ i = [1 FOR (num.variables - level)]
      link!vals[i - 1]
    SEQ i = [0 FOR power]
      link!poly[i]:

PROC crt(CHAN datain,left,right,VALUE set.position,level)=
  -- Interpolator process.
  PROC multinv(VALUE num,VAR inv)=
    -- Process to find the multiplicative inverse of a number with respect to
    -- a given field.
    VAR a, a0, q, ra, rb, b, b0, p:
    SEQ
      PAR
        a0 := num
        a := field
        b0 := 1
        b := 0
      WHILE (a0 <> 0)
        SEQ
          q := a/a0
        PAR
          ra := a - (q * a0)
          rb := b - (q * b0)
        PAR
          SEQ
            a := a0
            a0 := ra
          SEQ
            b := b0
            b0 := rb
      b := b\field
      IF
        b < 0
          inv := b + field
        TRUE
          inv := b:

  VAR polynomial[polylength], prev.polynomial[polylength]:
  VAR var.values1[num.variables], var.values2[num.variables]:
  VAR temporary, i, j, power, m:

  SEQ
    -- Initialise values.
    -- Variable 'power' is the maximum offset within a polynomial (vector)
    -- for which calculations are needed. The value of 'power' is dependent
```

```
      -- on the level of the pyramid and will always be a power of the degree
      -- of the variable.
      power := 1
      j := 0
      WHILE j < level
        SEQ
          power := power * (var.degree + 1)
          j := j + 1
      temporary := set.position - 1

      -- Accept the variables and corresponding polynomial from the previous
      -- level in the pyramid. If the level is the first level, the polynomial will
      -- be a single function value.
      shift.data.in(datain, polynomial, var.values1, level, power)
      WHILE (temporary > 0)
        SEQ
          -- Shift the variables and polynomial from the left process to the
          -- process on the right.
          shift.data.in(left, prev.polynomial, var.values2, level, power)
          shift.data.out(right, prev.polynomial, var.values2, level, power)
          PAR
            -- Subtraction step (Step 2).
            SEQ i = [0 FOR power]
              SEQ
                polynomial[i] :=
                    (polynomial[i] - prev.polynomial[i])\field
                IF
                  polynomial[i] < 0
                    polynomial[i] := polynomial[i] + field
                  TRUE
                    SKIP
            -- Inverse step (Step 3).
            multinv((var.values1[0] - var.values2[0]), m)
          -- Multiplication step (Step 3).
          SEQ i = [0 FOR power]
            polynomial[i] := (polynomial[i] * m)\field
          temporary := temporary - 1
      -- shift the variables and final polynomial to the process on the right.
      shift.data.out(right, polynomial, var.values1, level, power):

PROC calc(CHAN left,dataout, VALUE level)=
  -- Polynomial calculator process.
  VAR polynomial[polylength], cum.polynomial[polylength]:
  VAR var.values[num.variables]:
  VAR multiplier1[var.degree + 1], multiplier2[var.degree + 1]:
  VAR temporary, power1, power2, i, j, pointer:
```

```
SEQ
  -- Initialise values.
  -- Variable 'power2' is the maximum size of polynomials input from the
  -- rightmost interpolator (CRT) process.
  -- Variable 'power1' is the maximum size of polynomials output to the
  -- next level in the pyramid.
  temporary := 0
  power1 := 1
  SEQ i = [1 FOR (level + 1)]
    power1 := power1 * (var.degree + 1)
  power2 := power1/(var.degree + 1)
  SEQ i = [0 FOR power1]
    SEQ
      polynomial[i] := 0
      polynomial[i] := 0
  multiplier1[0] := 1
  SEQ i = [1 FOR var.degree]
    multiplier1[i] := 0
  WHILE temporary <= var.degree
    SEQ
      -- Input variables and polynomial from the right most interpolator
      -- process.
      shift.data.in(left, polynomial, var.values, level, power2)
      -- Multiply the polynomial received by the multiplier and add to
      -- the cumulative polynomial.
      SEQ j = [0 FOR (temporary + 1)]
        SEQ i = [0 FOR power2]
          cum.polynomial[(power2 * j) + i] :=
                (cum.polynomial[(power2 * j) + i]+
                (polynomial[i] * multiplier1[j]))\field
      IF
        temporary < var.degree
          -- Multiply single variable polynomial by (X - a). Increase
          -- multiplier degree by multiplying it by (X - a), where
          -- 'a' is the respective variable in the variable set of the
          -- previous input. Variable 'multiplier2' is the previous
          -- multiplier ('multiplier1') multiplied by '-a'. Variable
          -- 'multiplier2' is added to the left shift of 'multiplier1'.
          SEQ
            SEQ i = [0 FOR (temporary + 1)]
              SEQ
                multiplier2[i] :=
                      (multiplier1[i] * (-var.values[0]))\field
                IF
                  multiplier2[i] < 0
                    multiplier2[i] := multiplier2[i] + field
                  TRUE
                    SKIP
```

```
                    pointer := temporary + 1
                    WHILE (pointer > 0)
                      SEQ
                        multiplier1[pointer] := multiplier1[pointer - 1]+
                              multiplier2[pointer]
                        pointer := pointer - 1
                    multiplier1[0] := multiplier2[0]
                TRUE
                  SKIP
            temporary := temporary + 1
        -- Output the variables and cumulative polynomial to the next level in
        -- the pyramid.
        j := 1
        WHILE j < (num.variables - level)
          SEQ
            dataout!var.values[j]
            j := j + 1
        SEQ i = [0 FOR power1]
          dataout!cum.polynomial[i] :

PROC control(VALUE j,i)=
  -- Process used to configure the set of processes at a given level of the
  -- pyramid. This consists of a pipeline of interpolator processes and
  -- polynomial calculator process.
  PAR
    PAR k = [1 FOR (var.degree + 1)]
      crt(din[(j * (var.degree + 1)) + (k - 1)+
                ((cumlev[i] - lev[i]) * (var.degree + 1))],
          res[(j * (var.degree + 2)) + (k - 1)+
                ((cumlev[i] - lev[i]) * (var.degree + 2))],
          res[(j * (var.degree + 2)) + k+
                ((cumlev[i] - lev[i]) * (var.degree + 2))], k, i)
    calc(res[(j * (var.degree + 1)) + (var.degree + 1) + j+
                ((cumlev[i] - lev[i]) * (var.degree + 2))],
         din[(cumlev[i] * (var.degree + 1)) + j], i) :

-- Main program.
SEQ
  open.files
  PAR
    read.data(din)
    -- Configure the pyramid levels by replicating the 'control' processes.
    PAR j = [0 FOR lev[0]]
      control(j, 0)
    PAR j = [0 FOR lev[1]]
      control(j, 1)
    PAR j = [0 FOR lev[2]]
      control(j, 2)
    write.data(din[totprocs])
```

7. Parallel / Distributed Algorithm for Rational Function Interpolation

This section describes a parallel / distributed algorithm for rational function interpolation. This algorithm uses the concept of reciprocal differences introduced by Thiele [Baker et al 1981] and has a structure similar to that for polynomial interpolation.

7.1 Rational Interpolant

It is sometimes desirable to produce a more accurate interpolating formula than a polynomial. The rational functions which are quotients of polynomials constitute a richer class of functions than the polynomials. The rational Pade approximant [m / n] whose numerator is of degree m and denominator is of degree n, has the same accuracy as a polynomial of degree (m + n) [Cheney 1966]. Besides, the rational interpolation makes due allowances for rounding and representational errors and hence is very reliable.

The algorithm to be described yields a continued fraction representation of the n+1 point Pade approximation that fits the function values at n+1 distinct points.

Let a function r(x) be specified at (n + 1) points x_i such that $r(x_i) = r_i$ for i = 0,1,...n. We then need to determine r(x).

Let the list $\{X_k\}$ denote the list $\{x_0, x_1,...x_k\}$ and

the list $\{X_k, x\}$ denote $\{x_0, x_1,...x_k, x\}$ which is obtained by appending to $\{X_k\}$ the element x. We denote $P\{X_0\} = P(x_0) = r_0 = P_0$ and $P\{X_i\} = P_i$.

We then define

$$P\{X_1\} = P_1 = P\{x_0, x_1\} = (x_1 - x_0) / (r_1 - r_0)$$

$$P\{X_2\} = P_2 = P\{X_1, x_2\} = [(x_2 - x_1) / (P\{X_0,x_2\}-P\{X_1\})]+ P\{X_0\}$$

and in general, for a specified i,

$$P\{X_i,x_{i+j}\} = [(x_{i+j} - x_i) / (P\{X_{i-1}, x_{i+j}\} - P\{X_i\})] + P\{X_{i-1}\}$$

for all $j = 1, 2,...(n-i)$.

Note that the above recursion for rational interpolation is different from that for polynomial interpolation described in section 3.

The interpolant r(x) on $x_0, x_1,...x_n$ is then obtained as the following continued fraction [Baker et al 1981]:

$$r(x) = P_0 + \cfrac{x - x_0}{P_1 + \cfrac{x - x_1}{P_2 - P_0 + \cfrac{x - x_2}{P_3 - P_1 + \ldots + \cfrac{x - x_{n-1}}{P_n - P_{n-2}}}}}$$

This construction of the Pade approximant can be expressed as the following recurrences:

$A_0 = P_0$; $B_0 = 1$;

$A_1 = P_1P_0 + (x - x_0)$; $B_1 = P_1$;

$A_{i+1} = A_i (P_{i+1} - P_{i-1}) + A_{i-1} (x - x_i)$ for i =1,2...n-1

$B_{i+1} = B_i (P_{i+1} - P_{i-1}) + B_{i-1} (x - x_i)$ for i =1,2...n-1

$r = A_n / B_n$;

The above recurrences to compute A_i and B_i (i = 1, 2...n-1) can be expressed as the product of a (2 x 2) matrix and a (2 x 1) vector :

$$\begin{bmatrix} A_{i+1} \\ B_{i+1} \end{bmatrix} = \begin{bmatrix} A_i & A_{i-1} \\ B_i & B_{i-1} \end{bmatrix} \begin{bmatrix} P_{i+1} - P_{i-1} \\ x - x_i \end{bmatrix}$$

Note that the resulting vector is fed back in the next iteration as the left column of the right-side matrix; also, the previous left column moves to the right column. This permits a synchronous computation in a ISA.

Thus, we obtain the (n + 1) point Pade approximant [X / Y] where X denotes the degree of the numerator and Y the degree of the denominator.

If n + 1 = 2M + 1 (odd) where M is an integer then we get an [M / M] approximant. This means that we need to determine M + 1 coefficients in the numerator and M coefficients in the denominator (the constant term in the denominator is taken as unity).

Otherwise, if n + 1 = 2M + 2 (even) we get an [M + 1 / M] approximant. This means that we need to determine M + 2 coefficients in the numerator and M coefficients in the denominator .

For example, if n + 1 = 6, let the values available at points x_i be r_i (i = 0,1...5). We can then successively construct the [0 / 0], [1 / 0], [1 / 1], [2 / 1], [2 / 2], [3 / 2] Pade approximants, if they exist.

If some intermediate step involves a division by a zero, then the Pade approximant does not exist ; this happens when the data is inconsistent or the function has singularities.

We now describe the parallel / distributed algorithm for rational interpolation.

7.2 Algorithm

This algorithm takes as inputs (x_i, r_i) (i = 0, 1,...n) such that r_i are the values of the function at points x_i. The algorithm reconstructs a Pade approximant $r = A_n / B_n$ of order [((n + 1)/2) / ((n-1)/2)] when n is odd and [(n/2) / (n/2)] when n is even.

We assume that there are n+1 processors PRO(i) (i = 0, 1,... n). The registers in each processor are indexed by i.

Input: (x_i, r_i) (i = 0, 1,...n)

Register R_i is initialized with r_i and register X_i is initialized with x_i.

Output: $r = A_n / B_n$

$P_{-1} := 0;$

For i = 0 to n do

begin

$R_i := r_i ;$

$X_i := x_i;$

end;

For j = 0 to n do

begin

$P_j := R_j;$

For s = j+1 to n in parallel do

begin

$S_S := R_j;$

$R_S := R_S - S_S ;$

If $R_S = 0$ then 'ABORT';

else

$M_S := X_S - X_j ;$

$R_S := M_S / R_S ;$

$R_S := R_S + P_{j-1};$

end;

end;

The Pade Approximant can then be constructed as follows:

$A_0 := P_0; B_0 := 1;$

$A_1 := P_1P_0 + (x - X_0)$; $B_1 := P_1$;

For i = 1 to n-1 do

begin

$A_{i+1} := A_i (P_{i+1} - P_{i-1}) + A_{i-1} (x - X_i)$;

$B_{i+1} := B_i (P_{i+1} - P_{i-1}) + B_{i-1} (x - X_i)$;

end;

$r := A_n / B_n$;

Remarks

1. In the last iteration of j, when j is n, 'for s = j+1 to n in parallel do begin' is not executed since j+1 equals n+1, and exceeds the limit n.

2. The algorithm aborts when a division by zero arises.

Let $r(-1) = r_0 = 1$, $r(0) = r_1 = 1$, $r(1) = r_2 = 3$. Then the algorithm aborts when $(r_1 - P_0) = 0$ ($P_0 = r_0$). Hence there is no rational function of type [1 / 1] that fits this data.

to 31a

7.3 Example

Consider the function specified by Table 8

Table 8. Function Specifications

i	x_i	y_i
0	0	1/2
1	1	2/3
2	2	3/2
3	3	28/11
4	4	65/18
5	5	14/3

Table 9 shows the computational steps based on the algorithm. The P_i - values are :

$$P_0 = 1/2, \quad P_1 = 6, \quad P_2 = 1/4, \quad P_3 = 10/13, \quad P_4 = 56/3, \quad P_5 = 1.$$

We then compute the convergents as follows.

The various approximations are :

$$[0/0] = \begin{pmatrix} A_0 \\ B_0 \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1 \end{pmatrix}$$

$$[1/0] = \begin{pmatrix} A_1 \\ B_1 \end{pmatrix} = \begin{pmatrix} 1/2 \cdot 6 + x \\ 6 \end{pmatrix} = \begin{pmatrix} 3+x \\ 6 \end{pmatrix}$$

$$[1/1] = \begin{pmatrix} A_2 \\ B_2 \end{pmatrix} = \begin{pmatrix} 3+x & 1/2 \\ 6 & 1 \end{pmatrix} \begin{pmatrix} 1/4 - 1/2 \\ x-1 \end{pmatrix} = \begin{pmatrix} (x-5)/4 \\ (2x-5)/2 \end{pmatrix}$$

$$[2/1] = \begin{pmatrix} A_3 \\ B_3 \end{pmatrix} = \begin{pmatrix} (x-5)/4 & 3+x \\ (2x-5)/2 & 6 \end{pmatrix} \begin{pmatrix} \frac{10}{13} - 6 \\ x-2 \end{pmatrix} = \begin{pmatrix} \frac{13x^2-4x+7}{13} \\ \frac{10x+14}{13} \end{pmatrix}$$

$$[2/2] = \begin{pmatrix} A_4 \\ B_4 \end{pmatrix} = \begin{pmatrix} \frac{13x^2-4x+7}{13} & (x-5)/4 \\ \frac{10x+14}{13} & (2x-5)/2 \end{pmatrix} \begin{pmatrix} \frac{56}{3} - \frac{1}{4} \\ x-3 \end{pmatrix} = \begin{pmatrix} \frac{56x^2-23x+41}{3} \\ \frac{3x^2+26x+82}{3} \end{pmatrix}$$

$$[3/2] = \begin{pmatrix} A_5 \\ B_5 \end{pmatrix} = \begin{pmatrix} \frac{56x^2-23x+41}{3} & \frac{13x^2-4x+7}{13} \\ \frac{3x^2+26x+82}{3} & \frac{10x+14}{13} \end{pmatrix} \begin{pmatrix} 1 - \frac{10}{13} \\ x-4 \end{pmatrix} = \begin{pmatrix} x^3+1 \\ x^2+2 \end{pmatrix}$$

The final ~~fitted~~ function is $(x^3+1)/(x^2+2)$

Table 9 – Computational Steps

Operation	$PRO(0)$	$PRO(1)$	$PRO(2)$	$PRO(3)$	$PRO(4)$	$PRO(5)$	
X_i	0	1	2	3	4	5	
R_i	1/2	2/3	3/2	28/11	65/18	14/3	$P_0 = 1/2$
S_S		1/2	1/2	1/2	1/2	1/2	
$R_S - S_S$		1/6	1	45/22	28/9	25/6	
$M_s = X_s - X_j; j = 0$		1	2	3	4	5	
$R_s = M_s/R_s$		6	2	22/15	9/7	6/5	
P_{j-1}		0	0	0	0	0	
$R_s = R_s + P_{j-1}$		6	2	22/15	9/7	6/5	$P_1 = 6$
S_S			6	6	6	6	
$R_S - S_S$			−4	−68/15	−33/7	−24/5	
$M_s = X_s - X_j; j = 1$			1	2	3	4	
$R_s = M_s/R_s$			−1/4	−15/34	−7/11	−5/6	
P_{j-1}			1/2	1/2	1/2	1/2	
$R_s = R_s + P_{j-1}$			1/4	2/34	−3/22	−1/3	$P_2 = 1/4$
S_S				1/4	1/4	1/4	
$R_S - S_S$				−13/68	−17/44	−7/12	
$M_s = X_s - X_j; j = 2$				1	2	3	
$R_s = M_s/R_s$				−68/13	−88/17	−36/7	
P_{j-1}				6	6	6	
$R_s = R_s + P_{j-1}$				10/13	14/17	6/7	$P_3 = 10/13$
S_S					10/13	10/13	
$R_S - S_S$					12/221	8/91	
$M_s = X_s - X_j; j = 3$					1	2	
$R_s = M_s/R_s$					221/12	91/4	
P_{j-1}					1/4	1/4	
$R_s = R_s + P_{j-1}$					56/3	23	$P_4 = 56/3$
S_S						56/3	
$R_S - S_S$						13/3	
$M_s = X_s - X_j; j = 4$						1	
$R_s = M_s/R_s$						3/13	
P_{j-1}						10/13	
$R_s = R_s + P_{j-1}$						1	$P_5 = 1$

8. ISA Implementation of Rational Interpolation

We now consider the implementation of the rational interpolation algorithm on the ISA, using a systematic top-down design technique. The design approach followed is the same as that for polynomial interpolation described in section 5. The time complexities for the ISA programs are indicated.

We now convert the parallel rational interpolation algorithm described in section 7.2 to a locally recursive single-assignment algorithm.

8.1 Locally Recursive Rational Interpolation Algorithm

The locally recursive rational interpolation algorithm is given below.

$P_{-1} := 0$;
For i = 0 to n do
begin
    $R_{i0} := r_i$;
    $X_i := x_i$;
end;
For j = 0 to n do
begin
    $P_j := R_{j,3j}$;
    For k = j+1 to n in parallel do
    begin
        $S_{k,3j} := R_{j,3j}$;

$R_{k,3j+1} := R_{k,3j} - S_{k,3j}$;

If $R_{k,3j+1} = 0$ then 'ABORT';

else

$M_{kj} := X_k - X_j$;

$R_{k,3j+2} := M_{kj} / R_{k,3j+1}$;

$R_{k,3j+3} := R_{k,3j+2} + P_{j-1}$;

end;

end;

The dependence graph for this algorithm is presented in the next section.

The Pade Approximant can then be constructed as follows:

$A_0 := P_0$; $B_0 := 1$;

$A_1 := P_1P_0 + (x - X_0)$; $B_1 := P_1$;

For i = 1 to n-1 do

begin

$A_{i+1} := A_i (P_{i+1} - P_{i-1}) + A_{i-1} (x - X_i)$;

$B_{i+1} := B_i (P_{i+1} - P_{i-1}) + B_{i-1} (x - X_i)$;

end;

$r := A_n / B_n$;

The ISA implementation of the Pade approximant reconstruction is considered in section 8.4.2.

8.2 Dependence Graph

The structure of the dependence graph for the interpolation algorithm is shown in Fig. 6; index k is the horizontal axis and index j is the vertical axis (downwards). We show a typical set of adjacent computational streams which generate successive coefficients P_{j+1} and P_{j+2} respectively, where $k = j + 1$. Hence, $R_{k,3j+3} = R_{j+1,3j+3}$ and $R_{k+1,3j+6} = R_{j+2,3j+6}$. In the DG, the circles represent input values, e.g., x_k is one of the input points, and the boxes represent the nodes.

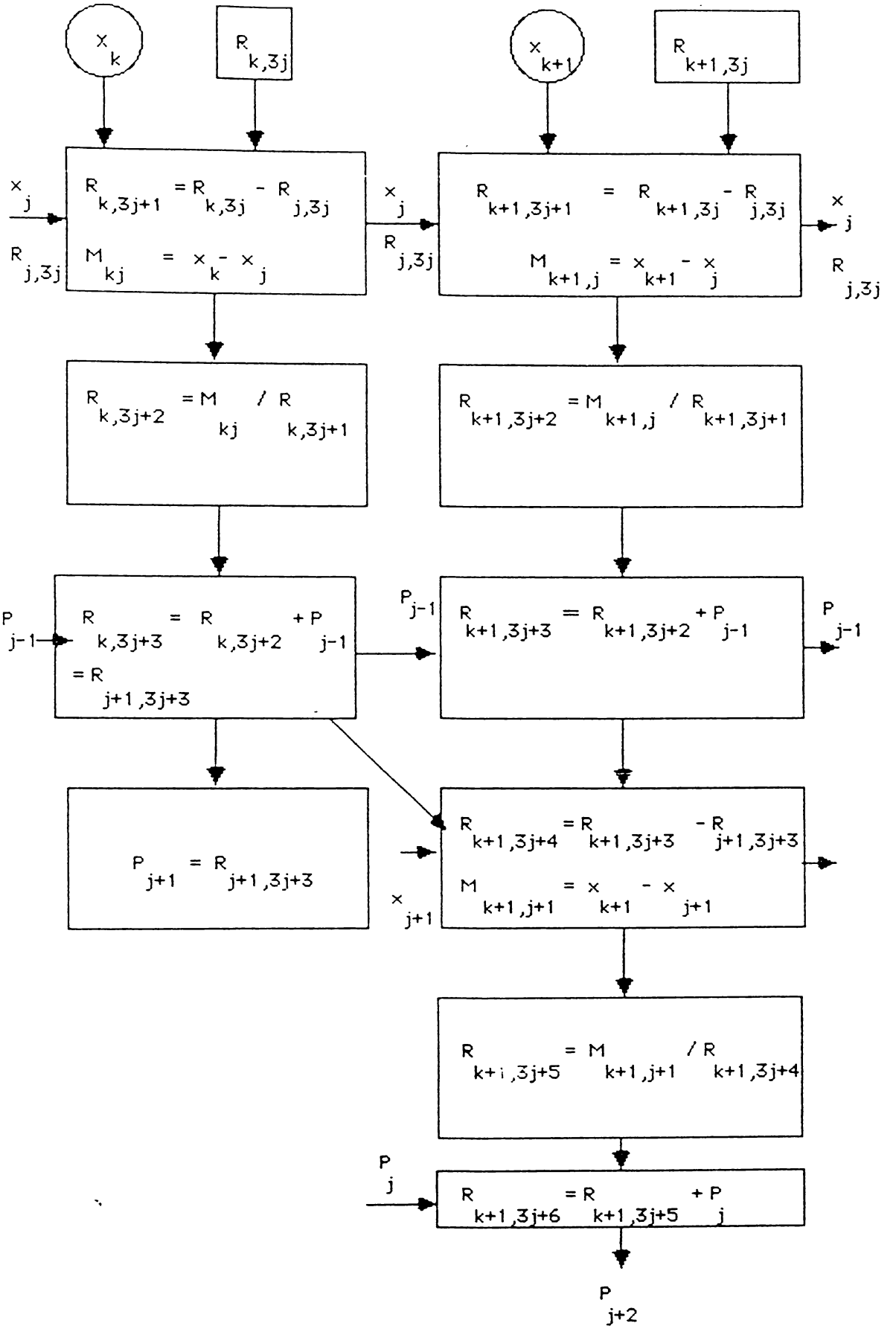
In the systolic implementation described below, the equitemporal lines (lines of equal time) are along the south-west to north-east diagonal, i.e., the time axis is along the north-west to south-east diagonal. Hence, with respect to the time axis, the DG will appear to be skewed.

It should also be noted that the axis of shift invariance for the DG is along the diagonal from the north-west to the south-east.

In the next step of the design process, we map the dependence graph to signal flow graphs.

8.3 Signal Flow Graphs

The Signal Flow Graph (SFG) represents the activities taking place in ISA processing elements at some instant of time as described in section 5.3. The node labels indicate the instruction at that processing element and the input arc labels (including self-loop arc labels) specify where an instruction gets its inputs from. For instance, an arc from the left node indicates that inputs are obtained from



DEPENDENCE GRAPH

FIG. 6

registers of the left neighbour processing element while a self-loop indicates that the inputs are obtained from registers of the same processing element.

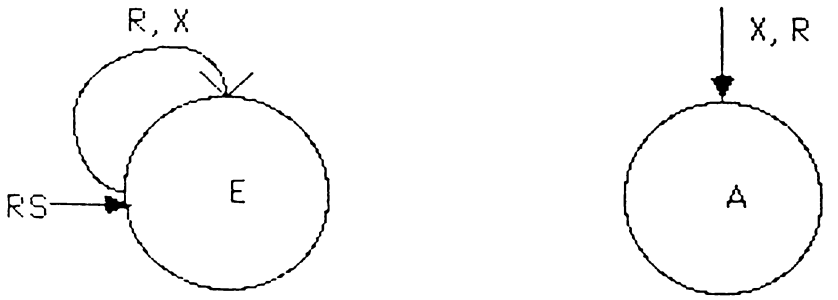
The ISA implementation described here uses a vertical projection of the DG, which is not along the direction of shift invariance (i.e., the diagonal from the north-west to the south-east). The main advantage of the design obtained from the vertical projection as compared to the horizontal projection is that less communication is required among the processors. However, each processor must be capable of executing all instructions mentioned below including the division instruction. If a horizontal projection is used, it is not necessary for each processor to have the capability to execute all the instructions. That is, we can have specialized processors, and assign the time-consuming divison operation to a particular processor.

The vertical projection of the DG results in two different Signal Flow Graphs which have to alternately applied, E-A and E-D-C-B-A (see Fig. 7); here, A, B, C, D and E are the instructions defined below.

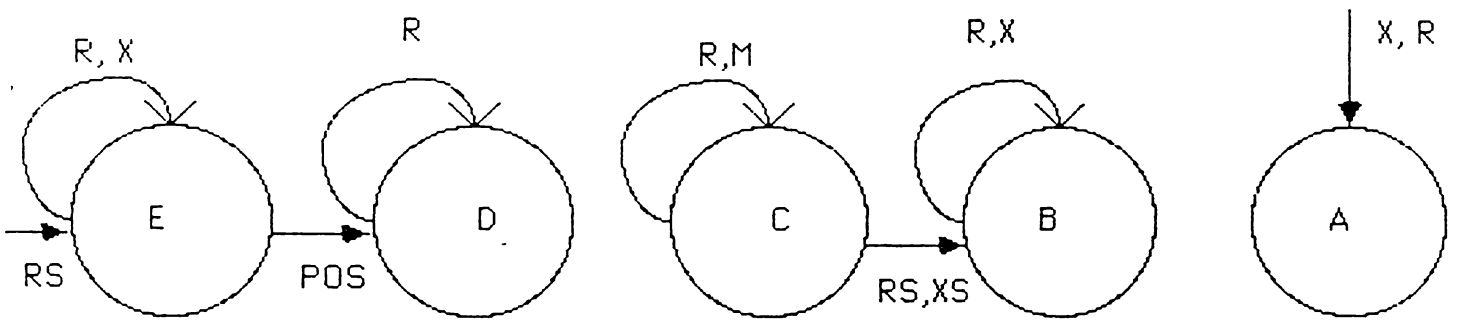
We now consider the ISA program for the locally recursive rational interpolation algorithm.

8.4 ISA Program for Rational Interpolation

Let us assume that we have a 1 x (n+1) array of n+1 processing elements (PE) in the ISA, P_0,..., P_n. The input data are the set of values r_i at points x_i (i = 0,..., n).



E-A SIGNAL FLOW GRAPH



E-D-C-B-A SIGNAL FLOW GRAPH

FIG. 7

Each processing element in the ISA has 6 data registers:

X stores a point x_k

XS shifts x_k

R stores value R_{kj}

RS shifts value $R_{j,3j}$

M stores M_{kj}

POS shifts P_{j-1}

We have five instructions in the instruction set, and each PE can execute all the instructions (subscripts L and T refer to the left and top neigbours respectively). The program is initialized with $\acute{R}S_L = 0$.

If $R=RS_L$ in instruction B and hence R = 0, then boolean ABORT is set to 1 and the ISA program is halted.

The five instructions are:

A. $X := X_T,\ R := R_T$

B. $R := R - RS_L,\ RS := RS_L,\ M := (X - XS_L),\ XS := XS_L,$
$ABORT := (R = RS_L)$

C. $R := M / R$

D. $R := R + POS_L,\ POS := POS_L$

E. $POS := RS_L,\ RS := R,\ XS := X$

The ISA interpolation program for n = 3 (i.e. there are 4 PE, P0, P1, P2 and P3) is shown in Fig. 8; the selectors are not shown. At the end of the program, register R in each PE P_i contains value P_i.

We observe that the pattern of instructions executed in each processing element P_i is $A(BCD)^i E$, i.e., the sequence of instructions BCD is repeated i times in P_i. We can generate all possible combinations of instructions that are executed at the same time (on different PE), viz. instructions on the same horizontal line, using the following expression:

L = [(e, E, CB, B) $(DCB)^*$ (D, DC, A, e)] U C where 'e' is the empty string, 'U' is the union operation of the set of strings and '*' represents repetition of the string zero or more times.

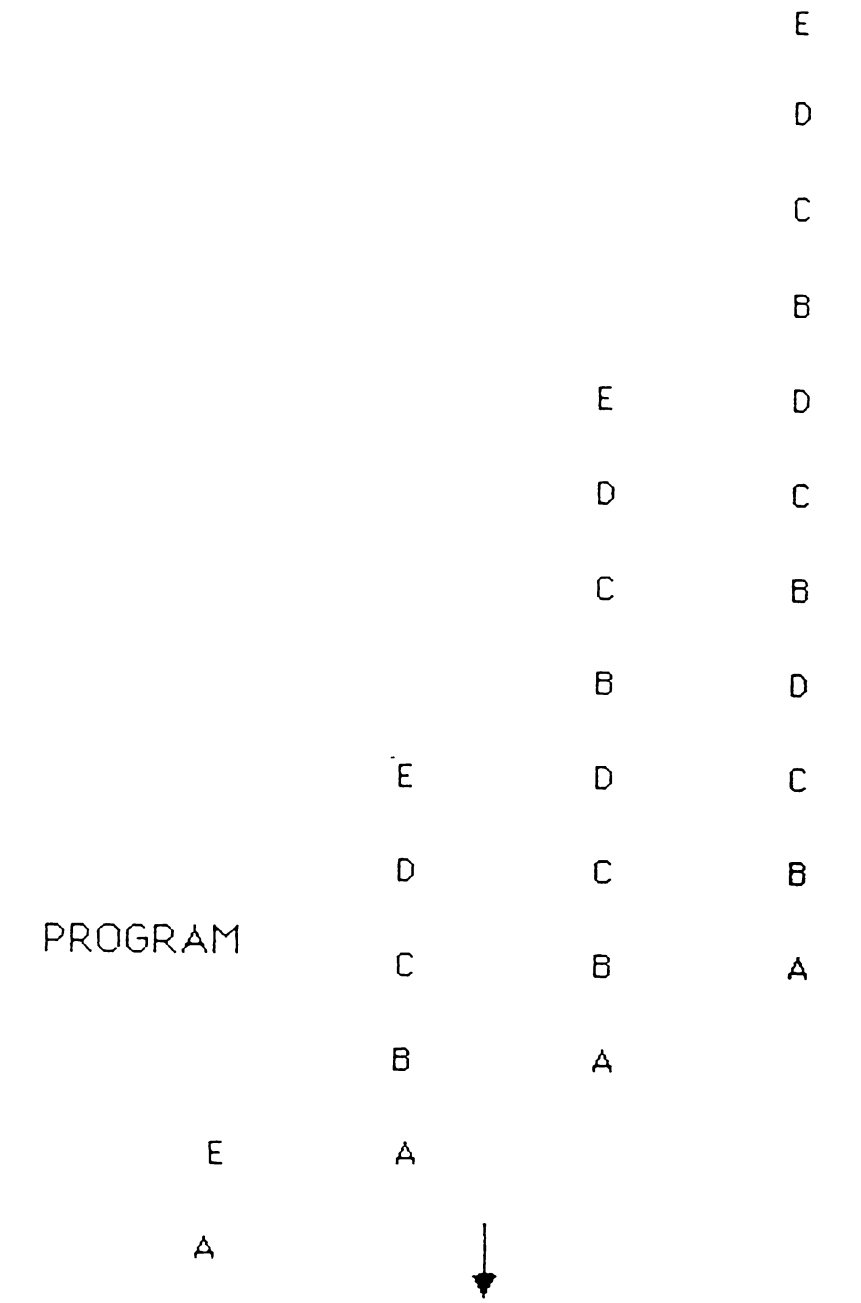
8.4.1 Period and Time of ISA program

We have a m x k array of processors, where m =1 and k =4.
Period = number of instruction diagonals = r = 3(k-1)+2 = 11.

Execution time = r + k + m - 2 = 14.

8.4.2 ISA program for Pade Approximant Construction

As described in section 7.1, A_i and B_i can be computed as the product of a matrix and a vector. The construction of such a product is a special case of the ISA matrix multiplication program described in Chapter 1, section 5.



ISA RATIONAL INTERPOLATION PROGRAM

FIG. 8

In the next chapter, we introduce parallel algorithms for matrix and linear algebraic computation and consider their implementation on a ISA. Some important applications of the algorithms are described.

CHAPTER 3

PARALLEL MATRIX COMPUTATION USING ISA

ABSTRACT

This chapter introduces a parallel algorithm for the generalized inversion (g-inversion) of matrices. This algorithm computes the inverse using matrix addition and multiplication operations. The ISA implementation of this algorithm is described. The solution of a homogeneous system of linear algebraic equations is based on g-inverse computation and we indicate its ISA implementation. The solution of such equations has numerous practical applications - in finding Petri net invariants, chemical equation balancing and dimensional analysis. We also describe two other important g-inversion algorithms - one iterative and the other direct.

CHAPTER 3

PARALLEL MATRIX COMPUTATION USING ISA

1. INTRODUCTION

This chapter introduces a parallel algorithm for the generalized inversion (g-inversion) of matrices. This algorithm computes the g-inverse using rational arithmetic operations and matrix addition and multiplication. A key feature of this algorithm is the divide and conquer strategy used. Accordingly, intermediate expression swell does not occur, and the range of numbers involved is not large. The systolic implementation of this algorithm using a variant of the ISA, the Single Instruction Systolic Array (SISA), is presented.

The solution of a homogeneous system of linear algebraic equations is based on matrix g-inversion computation and we discuss its ISA implementation. The solution of such equations has important practical applications; in particular, finding the S- or T- invariants in Petri nets, balancing chemical equations and dimensional analysis are considered.

We then describe two other important g-inversion algorithms. The first algorithm is direct and is based on a recurrence relation, while the second is iterative.

2. Parallel Matrix G-Inversion

It is well known that every nonsingular real or complex square matrix A has a unique inverse A^{-1} which has the property that $A\ A^{-1} = A^{-1}\ A = I$ (identity

matrix). This guarantees that the system of linear algebraic equations A x = b has the unique solution

$$x = A^{-1} b.$$

If a matrix is rectangular or if a square matrix is singular, it doesn't have a conventional inverse, but it does have a generalized inverse (g-inverse) A^-. A g-inverse A^- has the following properties [Rao et al 1971]:

1. $A A^- A = A$.
2. If A is square and nonsingular, a g-inverse reduces to the conventional inverse of a matrix, i.e., $A^- = A^{-1}$.

The g-inversion algorithm described here is suitable for rational error-free arithmetic, which is discussed in the following chapters. This algorithm uses a divide and conquer strategy to find the g-inverse of A column by column. This controls the growth of numbers and thus intermediate expression swell does not arise. Since the matrix A is processed incrementally (column by column) in a stream-oriented fashion [Murthy 1987], this algorithm is suitable for real-time stream-oriented programming.

2.1 Algorithm

The algorithm is based on the following definition and theorem.

Definition

Let A be a m x n matrix with integer or real elements denoted by

$A = [a_1, a_2, ..., a_n]$

where a_i (i = 1,2,...n) denotes the ith column in A of size m x 1. Also, let

$A_i = [a_1, a_2, ..., a_i]$

denote the matrix which contains the first i columns of A, starting with $A_1 = a_1$.

Then,

$A_{i-1} = [A_{i-2} : a_{i-1}]$, i = 3,4,...,n

where A_{i-1} is a m x (i - 1) matrix and a_{i-1} is a m x 1 vector.

Theorem

Let A_i be a m x i matrix and a_i be a m x 1 vector. Then A^- is generated using the following recursive equations:

$$e_{i+1} = (I - A_i A_i^-)^T (I - A_i A_i^-) a_{i+1} \quad (1)$$

$$d_{i+1} = A_i^- a_{i+1} \quad (2)$$

$$\text{if } e_{i+1}^T a_{i+1} \neq 0 \text{ then } b_{i+1} = e_{i+1} / e_{i+1}^T a_{i+1} \quad (3)$$

else if $e_{i+1}^T a_{i+1} = 0$ then $b_{i+1} = 0$

$$A_{i+1}^- = \begin{pmatrix} A_i^- - d_{i+1} b_{i+1}^t \\ b_{i+1}^t \end{pmatrix} \quad (4)$$

The recursion is started with $A_1^- = a_1^T / (a_1^T a_1)$ for $a_1 \neq 0$; otherwise, $A_1^- = 0$.

Note that the value of e_{i+1} is critical to the computation of A^{-}_{i+1}. If ordinary floating point arithmetic is used, the round off errors interfere with the computation; accordingly it is advisable to use either rational or error-free arithmetic, which is described in Chapter 4. The recursion above takes a different form for computing the Moore-Penrose inverse which is described in the context of error-free computing in Chapter 4.

Remark The algorithm described above considers only matrices whose elements are integers (or reals). However, this algorithm can be easily extended to polynomial matrices arising in Predicate-transition nets that model distributed database systems [Voss 1987].

We now describe how matrix g-inversion can be implemented on a variant of the ISA, the Single Instruction Systolic Array (SISA) (see Chapter 1).

3. SISA Implementation of Matrix G-Inversion

We assume that we have a n x m processing array in the SISA, where each processing element (PE) has some registers. For clarity, we use a consistent notation for naming registers; for example, elements of matrix A_i denoted by A_{ii} are stored in registers A of the SISA.

We first evaluate equation (1). Its evaluation is the most important and time consuming part of the algorithm, since it involves two matrix multiplications. We split this task into several stages: each stage will be implemented by a SISA subprogram described below.

$R_i := - A_i A_i^-$ MULT A, A⁻ TO R

$R_i := I + R_i$ ADD IDENTITY TO R

$T_i := R_i^T$ COPY R TO T

read a_{i+1} READ a

$e_{i+1} := T_i R_i a_{i+1}$ MULT T, R, a TO e

The computation for equation (2) is done by one subprogram:

$d_{i+1} := A_i^- a_{i+1}$ MULT A⁻,a TO d

The following subprograms evaluate equation (3):

$q := e_{i+1}^T a_{i+1}$ MULT e,a TO q

$t := (q \neq 0)$ TEST q to t

if t then $b_{i+1} := e_{i+1} / q$ DIVIDE e,q TO b

else $b_{i+1} := 0$

The evaluation of equation (4) is done as follows:

$A_i := A_i^- - d_{i+1} b_{i+1}^T$ SUBTRACT d,b FROM A⁻

$$\bar{A}_{i+1} := \begin{pmatrix} \bar{A}_i \\ b^T_{i+1} \end{pmatrix} \qquad \text{ATTACH a,b TO A,A}^-$$

The complete SISA implementation of the inversion algorithm is just the concatenation of the above subprograms.

repeat n times
begin
MULT A, A$^-$ TO R
ADD IDENTITY TO R
COPY R TO T
READ a
MULT T, R, a TO e
MULT A$^-$,a TO d
MULT e,a TO q
TEST q to t
DIVIDE e,q TO b
SUBTRACT d,b FROM A$^-$
ATTACH a,b TO A,A$^-$
end

3.1 SISA subprograms for Matrix Inversion

At the start of an iteration, the matrices A_i and $\bar{A}_i$ are stored in registers A and A$^-$ respectively of the last i rows of the SISA (see Fig. 1); here we have chosen a 6 x 6 SISA with the iteration index i = 4. Note that A is in column major order (transpose form) while A$^-$ is in row major order.

A_{11}^{-} A_{11}	A_{12}^{-} A_{21}	A_{13}^{-} A_{31}	A_{14}^{-} A_{41}	A_{15}^{-} A_{51}	A_{16}^{-} A_{61}
A_{21}^{-} A_{12}	A_{22}^{-} A_{22}	A_{23}^{-} A_{32}	A_{24}^{-} A_{42}	A_{25}^{-} A_{52}	A_{26}^{-} A_{62}
A_{31}^{-} A_{13}	A_{32}^{-} A_{23}	A_{33}^{-} A_{33}	A_{34}^{-} A_{43}	A_{35}^{-} A_{53}	A_{36}^{-} A_{63}
A_{41}^{-} A_{14}	A_{42}^{-} A_{24}	A_{43}^{-} A_{34}	A_{44}^{-} A_{44}	A_{45}^{-} A_{54}	A_{46}^{-} A_{64}

Fig. 1 Initial loading of registers A and A^{-}

MULT A, A^- TO R

This subprogram determines the product of A and A^-. We use the ROTV and ROTH subprograms (see Chapter 1, section 8) to perform a vertical and horizontal rotation of the registers respectively. The register S is used as intermediate storage. After the execution, the product is stored in registers R.

repeat n times
begin
($S := S_N - A \bullet A^-, 1^m, 1^n$)
($R := S, 1^m, 0^{n-1}1$)
ROTV R;
ROTH A;
end

ADD IDENTITY TO R

This subprogram adds the identity matrix to the matrix stored in R.

($R := R + 1, 1^m, 0^{n-1}1$)

COPY R TO T

This subprogram copies the contents of R to T. After its execution, the contents of registers R and T are as shown in Fig. 2.

T_{12} R_{21}	T_{23} R_{32}	T_{34} R_{43}	T_{45} R_{54}	T_{56} R_{65}	T_{61} R_{16}
T_{13} R_{31}	T_{24} R_{42}	T_{35} R_{53}	T_{46} R_{64}	T_{51} R_{15}	T_{62} R_{26}
T_{14} R_{41}	T_{25} R_{52}	T_{36} R_{63}	T_{41} R_{14}	T_{52} R_{25}	T_{63} R_{36}
T_{15} R_{51}	T_{26} R_{62}	T_{31} R_{13}	T_{42} R_{24}	T_{53} R_{35}	T_{64} R_{46}
T_{16} R_{61}	T_{21} R_{12}	T_{32} R_{23}	T_{43} R_{34}	T_{54} R_{45}	T_{65} R_{56}
T_{11} R_{11}	T_{22} R_{22}	T_{33} R_{33}	T_{44} R_{44}	T_{55} R_{55}	T_{66} R_{66}

Fig. 2 Registers R and T after execution of *COPY R TO T*

$(T := R, 1^m, 1^n)$

READ a

This subprogram reads the vector a_{i+1} (the new column of matrix A_i) into the last row of the SISA from external memory.

$(a := a_S, 1^m, 0^{n-1}\ 1)$

MULT T, R, a TO e

This subprogram performs the second matrix multiplication. Here, the matrices T_i and R_i and vector a_{i+1} are rotated n times; thus, the position of their elements remains the same. After the execution of the subprogram, vectors a_{i+1} and e_{i+1} are stored in the last row of the SISA.

repeat n times
begin
$(S := S_N + T \bullet R, 1^m, 1^n)$
$(e := S \bullet a, 1^m, 0^{n-1}\ 1)$
ROTV R;
ROTH T, a;
end

MULT A^-, a TO d

This subprogram produces $A_i^- a_{i+1}$, by first copying a_{i+1} to all rows of the SISA. In order to reduce execution time, we could have included the copying process in the previous subprogram. Note that it is necessary to introduce a diagonal of NOOP's* in between successive invocations of 'a := a_S' in order to prevent read-write conflicts. At the end of execution, vector d_{i+1} is stored in the last column of the SISA

repeat n-1 times

begin

$$(a := a_S,\ 1^m,\ 1^{n-1}\ 0)$$

$$(\text{NOOP},\ 1^m,\ 1^n)$$

end

$$(d := d_W + A^- \bullet a,\ 1^m,\ 1^n)$$

MULT e,a TO q

This subprogram yields q by computing the product of e and a. The element q is stored in the processing element (PE) at the south-east (lower right hand) corner.

$$(q := q_W + e \bullet a,\ 1^m,\ 0^{n-1}1)$$

TEST q TO t

This subprogram is initialized with t_W = false. Using the logical OR operation, we set boolean t to 'true' if at least one element of vector q_{i+1} is not zero; otherwise boolean t remains 'false'. At the end of execution, t is stored in the PE at the south-east corner.

* Same as NO-OP described in CHAPTER 1.

$(t := t_W \text{ OR } (q \neq 0), 1^m, 0^{n-1}\ 1)$

DIVIDE e,q TO b

Here, we first copy t, q and d_{i+1} in the horizontal direction. The vector b_{i+1} is then produced and copied vertically. At the end of execution, the contents of registers b and d are as shown in Fig. 3. Again, the NOOP diagonal is necessary to avoid read-write conflicts.

repeat m-1 times

begin

$(q := q_E, 1^{m-1}\ 0, 0^{n-1}\ 1)$

$(t := t_E, 1^{m-1}\ 0, 0^{n-1}\ 1)$

$(d := d_E, 1^{m-1}\ 0, 1^n)$

end

$(b := e / q \text{ if } t, 1^m, 0^{n-1}\ 1)$

$(b := 0 \text{ if } \neg t, 1^m, 0^{n-1}\ 1)$

repeat n-1 times

begin

$(b := b_S, 1^m, 1^{n-1}\ 0)$

$(\text{NOOP}, 1^m, 1^n)$

end

SUBTRACT d,b FROM A^-

This subprogram produces the first i rows of A^-_{i+1} in one sweep.

$d = 0$ b_1	$d = 0$ b_2	$d = 0$ b_3	$d = 0$ b_4	$d = 0$ b_5	$d = 0$ b_6
$d = 0$ b_1	$d = 0$ b_2	$d = 0$ b_3	$d = 0$ b_4	$d = 0$ b_5	$d = 0$ b_6
d_1 b_1	d_1 b_2	d_1 b_3	d_1 b_4	d_1 b_5	d_1 b_6
d_2 b_1	d_2 b_2	d_2 b_3	d_2 b_4	d_2 b_5	d_2 b_6
d_3 b_1	d_3 b_2	d_3 b_3	d_3 b_4	d_3 b_5	d_3 b_6
d_4 b_1	d_4 b_2	d_4 b_3	d_4 b_4	d_4 b_5	d_4 b_6

Fig. 3 Registers b and d after execution of $DIVIDE\ e, q\ TO\ b$

$$(A^- := A^- - d \bullet b, 1^m, 1^n)$$

ATTACH b, a TO A^-, A

Here, we copy vector b_{i+1} into the first row of the SISA as a row of matrix A^-. We then vertically rotate matrices A_{i+1} and $\bar{A}_{i+1}$ by one position and finally copy vector a_{i+1} (stored in the last row of SISA) as a new row of A^T (new column of A). At the end of execution, the last i+1 rows of the SISA contain the matrices A_{i+1} and $\bar{A}_{i+1}$.

$(A^- := b, 1^m, 1\ 0^{n-1})$

ROTV A, A^-;

$(A := a, 1^m, 0^{n-1}\ 1)$

The next iteration can then be started by executing the identical of subroutines.

3.2 Fault-diagnosis

The above algorithm uses "column pumping", viz. the columns of the input matrix are processed incrementally one by one. This makes it easy to devise diagnostic checks. For this purpose, we choose an arbitrary permutation matrix P (whose columns are permutations of the identity matrix) and use the algorithm to compute P^{-1}. It is well known that $P^{-1} = P^T$ and so a checking routine can be devised to verify the SISA program. The diagnostic steps are:

i. Pump an arbitrary P through the SISA

ii. Compute $P^{-1} - P^T$

iii. Check if $P^{-1} - P^T$ is zero

4. G-Inverse Applications to Linear Algebraic Systems

The g-inverse is essential in the solution of a homogeneous system of linear algebraic equations

$$Ax = 0 \qquad (5)$$

where A is a m x n matrix with rational entries and $x^T = (x_1, x_2,...,x_n)$ is the unknown n x 1 vector with integral or rational components (T denotes transpose). The solution is determined as follows:

$$x = (I - A^- A)y \quad \text{(y is an arbitrary vector)}$$

where the g-inverse A^- is a n x m matrix.

The rank of the n x n matrix $(I - A^- A)$ is $k = n - r$, where r is the rank of A. This means that $(I - A^-A)$ has $(n - r)$ linearly independent columns.

The solution of homogeneous linear equations has several applications, notably in Petri net analysis, balancing chemical equations, dimensional analysis and for synthesizing systolic arrays based on dependency graphs [Rajopadhye et al 1987]. These applications often require exact integral solutions. Such a solution x will be integral for arbitrary integral y iff the elements in A and $(I - A^- A)$ are integral. We can ensure that $(I - A^- A)$ has integral elements by suitably scaling it up in the final stage.

The determination of the above solution has three major issues:

1. If A is square and nonsingular, $x = 0$ is the trivial solution. A non-trivial solution exists only when A is singular. In this case, the solution is non-unique and most conventional matrix inversion procedures cannot be used directly without extensive modification.

2. When exact integral or rational solutions are required, the matrix inversion procedure used should be error-free.

3. The usual method to solve equation (1) is by using the Smith-canonical form. However, this method is quite involved, due to intermediate expression swell arising from the use of integer arithmetic and Gaussian elimination without division [Kruckeberg et al 1987].

The matrix g-inversion algorithm described here handles these issues. It directly determines the g-inverse and is suitable for exact computation methods. It also uses a divide and conquer strategy; this controls the growth of numbers and intermediate expression swell does not arise.

The above solution also has applications in a inhomogeneous linear algebraic system. It is well known that a inhomogeneous system $A^T y = b$ is consistent iff every solution x_i of the homogeneous equation $Ax = 0$ is orthogonal to b. That is, for consistency, we require $b^T (I - A^- A) = 0$. The above algorithm can easily check for this condition.

4.1 Solving Linear Algebraic systems using ISA

The ISA program to compute g-inverse A^- was described in section 3. We can then compute the homogeneous equation solution matrix ($I-A^-A$) using the subprograms
MULT A,A- TO R and ADD IDENTITY TO R (see section 3.1). Thus, we have a ISA program to compute the general solution of homogeneous linear algebraic systems.

We now briefly indicate some important applications of the inversion algorithm: determining Petri net invariants, balancing chemical equations and dimensional analysis.

5. Determining Petri net Invariants

Petri nets are very useful in the modelling and analysis of concurrent systems [Kruckeberg et al 1987, Murata 1986, Reisig 1986]. A Petri net consists of two disjoint sets of vertices called places and transitions, and a set of edges from a place to a transition or vice versa. The places essentially represent conditions and the transitions represent events. In order to simulate the dynamic behaviour of a system, tokens are used in Petri nets. Each place can hold tokens, and the token distribution over places is called the marking of a Petri net. The marking changes as transitions are enabled or fired. A marking M1 is then said to be reachable from another marking M0 if there exists a sequence of transition firings that transforms M0 to M1. This notion of reachability is used for studying the dynamic behaviour of systems and is important in Petri net analysis.

We can determine the reachability of markings by using Petri net S-invariants. These invariants can be determined by solving the equation $Ax = 0$ where A is the

transpose of the flow matrix of a Petri net and x is the unknown vector [Kruckeberg et al 1987]. The solution of this equation, $x = (I - A^{-}A)y$ (y is an arbitrary vector), gives us the Petri net S or T-invariants as required. That is, the n-r linearly independent columns of $(I - A^{-}A)$ give the desired invariants. The above algorithm determines the S or T invariants in a Petri analysis problem in a systematic manner without introducing row-column permutations. The S-invariants x then determines reachability as follows: A marking M is reachable from any fixed initial marking M0 if and only if $M^{T}x = M0^{T}x$ [Kruckeberg et al 1987].

As mentioned above, the rank of the solution matrix $I - A^{-}A$ is $k = n - r$, where r is the rank of A. Then depending on the rank of A, we have three cases to consider:

1. When $r = n$ and $k = 0$, we only have the trivial solution $x = 0$.

2. When $r = n-1$ and $k = 1$, there is only one linearly independent solution. This means that the S-invariant is unique within relative proportions of its components.

3. When $r \leq n - 2$ and $k \geq 2$, two or more linearly independent solutions are possible. In this case, two or more S-invariants exist.

In some cases of Petri net analysis, we further require that $Ax \geq 0$ and $x \geq 0$ with Ax and x containing components from only the set of positive integers I^{+}. For this purpose, algorithms given in [Gregory et al 1984, Kruckeberg 1987] may be used.

Example

Let the m x n matrix A be the transpose of the flow matrix of a Petri net, where m = 6 and n = 10 (see table I). The rank of A is r = 5.

Then, we obtain the n x n matrix I - $A^{-}A$ (see table I) which has rank k = n - r = 5, i.e., 5 linearly independent columns.

Thus, the S-invariants are the five vectors

$x_1^T = (1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$ $\quad x_2^T = (-1\ 0\ 1\ -1\ 1\ 0\ 0\ 0\ 0\ 0)$

$x_3^T = (1\ 0\ -1\ 1\ 0\ 1\ 0\ 0\ 0\ 0)$ $\quad x_4^T = (-1\ 0\ 0\ 0\ 0\ 0\ -1\ 1\ 0\ 0)$

$x_5^T = (1\ 0\ 1\ 1\ 0\ 0\ 2\ 0\ 1\ 1)$

It is easily verified that the above vectors satisfy Ax = 0.

The next section describes the application of the inversion algorithm in balancing chemical equations.

6. Balancing Chemical Equations

We can devise a deterministic method, free from trial and error, to balance chemical equations by determining the solution of linear algebraic equations [Gregory and Krishnamurthy 1984].

Consider the chemical equation

$$A = \begin{bmatrix} -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

$$(I - A^{-}A) = \begin{bmatrix} 0 & 1 & 0 & 0 & -1 & 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Table 1

$x_1\ Al + x_2\ HNO_3 = x_3 Al(NO_3)_3 + x_4 NO + x_5\ H_2O$

Here, aluminum reacts with nitric acid to produce aluminum nitrate, nitric oxide and water. Our aim is to find $x_1, \ldots, x_5$ to satisfy the law of conservation of atoms. Suppose we have m distinct chemical elements and n distinct chemical compounds in a chemical equation. We can form a m x n reaction matrix $A = (a_{ij})$ as follows. The integer a_{ij} is defined as the number of atoms of type i in compound j, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Integer a_{ij} is positive if compound j is a reactant and negative if compound j is a product. The chemical compounds and elements are ordered as they appear in the equation, reading from left to right.

For our example, the elements for $1 \leq i \leq 4$ are Al, H, N and O and the compounds for $1 \leq j \leq 5$ are Al, HNO_3, $Al(NO_3)_3$, NO and H_2O. The reaction matrix A is then

$$\begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -2 \\ 0 & 1 & -3 & -1 & 0 \\ 0 & 3 & -9 & -1 & -1 \end{pmatrix}$$

Then if we define vector $x^T = (x_1\ x_2\ x_3\ x_4\ x_5)$, we obtain the coefficients by solving the homogeneous linear algebraic equation $Ax = 0$. The solution matrix $(I - A^-A)$ provides the relative proportions of reactants and products which would balance the chemical reaction. (for $x^T = (1, 4, 1, 1, 2)$)

As mentioned above, the rank of the solution matrix $I - A^-A$ is $k = n - r$, where r is the rank of A. Then depending on the rank of A, we have three cases to consider:

1. When $r = n$ and $k = 0$, the only solution is $x = 0$. This implies that the chemical equation cannot be balanced and no chemical reaction is possible. Hence, we can mechanically rule out impossible reactions.

2. When $r = n-1$ and $k = 1$, there is only one linearly independent solution. Thus, the chemical equation is unique within relative proportions of reactants and products.

3. When $r \leq n - 2$ and $k \geq 2$, two or more linearly independent solutions are possible and the chemical equation is not unique within relative proportions.

The next section describes the application of the inversion algorithm to dimensional analysis.

7. Dimensional Analysis and Modelling

Dimensional analysis [David et al 1982] studies the restrictions placed on the form of an algebraic function by the requirements of dimensional homogeneity. This analysis has applications in experimental modelling and prototyping. The g-inverse algorithm can be used to obtain the solution of dimensional analysis problems [Krishnamurthy and Murthy 1984]. The solution provides the functional relationship between a set of variables.

The relationship between sets of quantities can be expressed in terms of a homogeneous equation. For example, consider the quantities P, Q and R expressed in terms of fundamental physical quantities- Mass (M), length (L) and time (T):

$P = M^{a11} L^{a21} T^{a31}$

$Q = M^{a12} L^{a22} T^{a32}$

$R = M^{a13} L^{a23} T^{a33}$

If P, Q and R are related physically by a relation then we must have

$P^{x1} = Q^{x2} R^{x3}$

or $Q^{x2} R^{x3} / P^{x1} = 1.$

In other words, $Q^{x2} R^{x3} / P^{x1}$ is dimensionless, i.e., the exponents of M, L and T are zero. Thus, we must determine x1, x2 and x3 such that

$a_{11} x_1 - a_{12} x_2 - a_{13} x_3 = 0$

$a_{21} x_1 - a_{22} x_2 - a_{23} x_3 = 0$

$a_{31} x_1 - a_{32} x_2 - a_{33} x_3 = 0$

or simply solve $Ax = 0$ where A is a m x n matrix with coefficient entries a_{ij} (i = 1,...m and j = 1,...n) and $x^T = (x_1, x_2, \ldots x_n)$ is the unknown vector.

The rank of the solution matrix $I - A^{-}A$ is $k = n - r$, where r is the rank of A. Then depending on the rank of A, we have three cases to consider:

1. When r = n and k = 0, the only solution is x = 0. This means that no one quantity can be expressed in terms of the others.

2. When r = n-1 and k = 1, there is only one linearly independent solution. In this case, there is a unique functional relationship between the quantities.

3. When $r \leq n - 2$ and $k \geq 2$, two or more linearly independent solutions are possible. In this case, we can have several functional relationships between the quantities.

The next section describes two other important algorithms for matrix g-inversion.

8. Iterative/ Recursive algorithms for Matrix G-Inversion

In this section we describe two g-inversion algorithms that are suitable for VLSI and optical computing. The first algorithm due to Ben Israel and Greville (B-G method) [Gregory and Krishnamurthy 1984, Soderstrom et al 1974] is iterative and is derived from the Newton-Schultz method. The second algorithm is due to Decell and Leverrier. It is direct and based on a recurrence relation.

8.1 Ben-Israel and Greville's method

The Ben Israel and Greville (B-G method) has three distinct advantages over the methods described in [Ghosh et al 1987, Rajbenbach et al 1987]:

1. If the matrix turns out to be singular, the Moore-Penrose g-inverse is obtained that can be used for obtaining the least square solution.

2. Here, convergence is assured and the speed of convergence can be varied by changing a parameter unlike the algorithms described above which require extensive preprocessing to achieve convergence.

3. The implementation can choose any suitable arithmetic system - error-free (as described in the next chapter) or floating-point arithmetic.

The B-G method determines the Moore-Penrose inverse A+ [Gregory and Krishnamurthy 1984] given a m x n matrix A. The inverse A^+ is unique and reduces to the conventional inverse A^{-1} when A is square and non-singular. Inverse A^+ has the following properties:

$A\,A^+\,A = A;\ A^+\,A\,A^+ = A^+;\ (A^+A)^T = A^+A;\ (AA^+)^T = AA^+$

The iteration is as follows:

While $\|X_{k+1} - X_k\| \neq 0$

do $X_{k+1} = 2\,X_k - X_k\,A\,X_k$

The iteration is started with $X_0 = \alpha\,A^T$ (A^T is the transpose of A)

where $0 < \alpha < 2 / \text{trace}(A\,A^T)$; the trace of a matrix is the sum of its diagonal elements.

The iteration converges to A^+, i.e., $X_k = A^+$. We can choose α suitably for fast convergence. In this method, we need to compute the difference matrix, $X_{k+1} - X_k$, at each step. For floating point numbers, we use the criterion

$||\Delta X_{k+1}|| = ||X_{k+1} - X_k||$

where $||\ ||$ denotes the least-square norm (sum of squares of differences between X_{k+1} and X_k). Thus, we compute $\Delta X_{k+1} = (X_{k+1} - X_k) = X_k (I - A X_k)$ and $X_{k+1} = X_k + \Delta X_{k+1}$.

For linear algebraic equations of the form $Ax = g$, this method can be used to determine the least-square-solution, $A^+ g$, when A is singular.

8.2 Decell-Leverrier method

This method computes the g-inverse based on recursive computation. It has the following special features:

1. It can be used for finding the characteristic equation.

2. The rank of the matrix is generated as a byproduct of the computation.

3. The implementation can be in any suitable arithmetic system - exact or floating-point. To determine the exact rank, no round off errors should be introduced and so it is desirable to use error-free arithmetic.

The Decell method computes the g-inverse A^+ of a m x n matrix A as follows:

We start with

$$A_1 = A\,A^T,\ q_1 = tr\,(A_1),\ B_1 = A_1 - q_1 I\ ;$$

then compute

$$A_2 = A_1\,B_1,\ q_2 = (1/2)\ tr(A_2),\ B_2 = A_2 - q_2\,I$$

...

$$A_k = A_1\,B_{k-1},\ q_k = (1/k)\ tr\,(A_k),\ B_k = A_k - q_k\,I$$

where $tr(A_k)$ = trace of matrix A_k.

The computation of A_k is continued until $A_1 B_k = 0$ for $k > 0$. Then

$A^+ = A^T B_{k-1} / q_k$.

The rank of A is then k. The characteristic equation of AA^T is then given by

$$(-1)^m\,(a_0\,\lambda^m + \ldots + a_m)$$

with $a_0 = 1$ and $q_i = tr(A_i) / i = -a_i$ where $q_0 = -1$.

Both methods described above are suitable for implementation on a ISA. This can be realized by splitting the algorithm into subprograms and then implementing them as in the case of the A^- inversion algorithm (see section 3), and we shall not describe it here. Both methods described will have applications in signal processing, automatic control and parameter estimation.

In the next chapter, we introduce a new parallel error-free rational arithmetic system which has applications for the parallel exact inversion of matrices.

CHAPTER 4

Parallel Error-free Rational Arithmetic - ISA Implementation

ABSTRACT

A carry-free (parallel) error-free rational arithmetic system based on residue and p-adic representation is introduced. In this system, Farey rational numbers, whose numerators and denominators are bounded, are encoded into Para-Hensel codes (Parallel rational Hensel code) and the parallel element-wise arithmetic is performed using these codes. The algorithms for encoding into and decoding from Para-Hensel code and the arithmetic algorithms are described. This system will have extensive applications in massively parallel processors. We also describe the application of Para-Hensel codes for exact inversion or generalized inversion of a matrix.

CHAPTER 4

Parallel Error-free Rational Arithmetic - ISA Implementation

1. Introduction

Many papers have appeared on residue arithmetic processors for Optical and VLSI Computing [Swartzlander 1986, Psaltis et al 1979, Jackson et al 1983, Beaudet et al 1986, Tai et al 1979, Huang et al 1979]. One of the important advantages of the residue arithmetic is that it requires no carry mechanism [Szabo et al 1967, Gregory et al 1984]. This allows fast parallel computations to be performed in a single clock cycle and so massively parallel systolic processors can be built [Kogge 1981, Hwang et al 1984].

The residue arithmetic processors essentially perform integer addition/subtraction/multiplication modulo a prime; however, division is a very involved and complicated operation. No efficient algorithms as such exist for the division operation [Szabo et al 1967]. Thus, as such, rational arithmetic cannot be realized using the residue arithmetic system.

CHAPTER 4

This chapter considers the extension of residue arithmetic system for performing carry-free (parallel), error-free rational arithmetic. The term carry-free is very well-known; it means there is no carry propagation in the arithmetic algorithms. By error-free, we mean that each rational is exactly represented by a finite length linear string [Gregory et al 1984], without any representational error that usually occurs in a conventional p-ary system, if the denominator is relatively prime to the base of representation. For example,

$(1/7)_{10}$ = .142857..., has no finite length exact representation.

Thus the conventional p-ary (or decimal) arithmetic system is neither error-free nor carry-free. One way to perform error-free rational arithmetic is by the use of ordered pair or fractional notation. This system is not carry-free. Also each rational addition/subtraction operation in such a system requires three integer multiplications and one integer addition/subtraction; each rational multiplication and division operations require two integer multiplications. Besides, we require another operation to reduce the result to the lowest form. Thus the fractional arithmetic system is quite expensive and cumbersome.

An alternative approach to error-free rational arithmetic is by using the p-adic system recently studied in detail [Gregory et al 1984].

The p-adic system is more economical than the fractional arithmetic, since it requires only one addition/subtraction operation for each rational addition/subtraction and one multiplication/division for each rational multiplication/division; also it does not require the reduction to the lowest form. However, this system is not carry-free.

For a massively parallel realization of rational arithmetic one needs to have a carry-free and error-free arithmetic system. This chapter describes the basic principles of a new rational arithmetic system with these properties. This system combines the residue and p-adic (Hensel Codes) representation of rational numbers to derive a new code called 'Para-Hensel Code' (for Parallel Rational Hensel Code) [Krishnamurthy and Murthy 1987]. Using the Para-Hensel Codes, parallel rational arithmetic can be realized.

We describe practical algorithms for

(i) Forward mapping from Farey rational numbers (rationals whose numerators and denominators are bounded) to Para-Hensel Codes (PHC).

(ii) Inverse mapping from PHC to rationals.

(iii) Arithmetic using PHC.

The basic theoretical framework for the construction of Hensel Codes and Para-Hensel Codes and the key theorems ensuring the validity of the encoding and decoding algorithms are available in [Gregory and Krishnamurthy 1984].

We also describe how PHC can be used for parallel error-free inversion of matrices. Indeed, the PHC system can be used for any linear algebraic algorithm when exact rational results are required.

2. Para-Hensel Codes

2.1 Farey Rationals F_N

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n distinct primes and

$$M = \prod_{i=1}^{n} p_i$$

Consider any rational number a/b that belongs to the finite subset of the rational numbers

$$F_N = \{a/b : \gcd(a,b) = 1 \text{ and } 0 \leq |a| \leq N \text{ and } 0 < |b| \leq N\}$$

(gcd = greatest common divisor)

known as Farey-rationals of order N, whose numerator and denominators are less than or equal to N an arbitrary integer.

Let M be such that $M \geq 2N^2+1$

or $N \leq \text{INT}(\text{SQRT}((M-1)/2))$

where INT = lower integral part and SQRT = square root function.

Then we can construct Para-Hensel Codes for order N Farey rationals, suitable for exact parallel rational arithmetic.

2.2 Para-Hensel Codes for F_N

Let a/b be any non-zero rational number. We can express a/b as

$$a/b = a_i/b_i \cdot (p_i)^{r_i}$$

with gcd (a_i,b_i) = gcd (a_i,p_i) = gcd(b_i,p_i)=1.

Then we define the Para-Hensel Code of a/b (denoted by PHC (a/b)) as the set of n ordered pairs :

PHC (a/b) = $\{(\alpha_1,r_1), (\alpha_2,r_2), \ldots(\alpha_n,r_n)\}$

where for i = 1,2,...,n

$\alpha_i = (a_i\ b_i^{-1}) \bmod p_i$

r_i = exponent of p_i (positive, negative or zero)

Note b_i^{-1} exists since gcd (b_i,p_i) = 1; it can be computed using Extended Euclidean Algorithm (see Section 3.2). In analogy with the floating-point number system, we call α_i the mantissa, and r_i the exponent.

Our aim is to establish a bijective mapping between F_N and a set of n ordered pairs denoting

PHC (a/b), a/b $\in F_N$,

so that we can encode $a/b \in F_N$ into PHC and decode PHC to its equivalent $a/b \in F_N$.

To represent zero, we may use integer z (usually z=0) and write

$0 \bmod p_i = (0,z_i)$, $i=1,2,\ldots,n$;

note that exponent for zero is not unique.

2.3 Example

Let $p = \{2,3,5,7\}$ and $a/b = 3/7$; then

PHC (3/7) = {(1,0), (1,1), (4,0), (3,-1)}

2.4 Additive/ Multiplicative inverses of PHC

In order to perform inverse operations, such as subtraction/ division, we define respectively the additive/multiplicative inverses of PHC(a/b) thus :

$$PHC(-a/b) = \{(-\alpha_1,r_1), \ldots,(-\alpha_n,r_n)\}$$

where $-\alpha_i$ $(i = 1,2, \ldots,n)$ is the additive inverse of $\alpha_i \bmod p_i$ which equals $(p_i-\alpha_i)$.

$$PHC(b/a) = \{(\alpha_1^{-1},-r_1), \ldots,(\alpha_n^{-1},-r_n)\}$$

where α_i^{-1} $(i = 1,2, \ldots,n)$ is the multiplicative inverse of $\alpha_i \bmod p_i$ and $-r_i$ denotes the opposite sign for the exponent.

2.5 Example

Let P = {2,3,5,7}, $N \leq 10$

PHC (3/7) = {(1,0), (1,1), (4,0), (3,-1)}

PHC (-3/7) = {(1,0), (2,1), (1,0), (4,-1)}

PHC (7/3) = {(1,0), (1,-1), (4,0), (5,1)}

3. Decoding a Para-Hensel Code - Algorithms and Proofs

Consider the set of images Q_p of the set of rational numbers Q :

Q_p = {PHC(a/b) : a/b $\in$ Q}

and the finite subset of Q_p consisting of the images of the order N Farey fractions,

Q_p^* = {PHC(a/b) : a/b $\in$ F_N}

The mapping PHC (a/b) : $F_N \rightarrow Q_p^*$ is onto (surjection) and it is one-to-one, in the sense the inverse mapping $Q_p^* \rightarrow F_N$ exists and is based on the two well-known classical algorithms, Chinese remaindering

and Extended Euclidean algorithm [Gregory et al 1984, Krishnamurthy 1985].

Before proceeding to describe the algorithms and their proofs, we define the following terms :

(a) Let $M_+ = \prod_i p_i$ where p_i is a modulus for which the exponent r_i in PHC(a/b) is greater than zero.
If no such modulus exists then $M_+ = 1$.

(b) Let $M_0 = \prod_i p_i$ where p_i is a modulus for which the exponent r_i in PHC(a/b) is zero. If no such modulus exists, the computation cannot proceed further, and the PHC(a/b) remains undefined. In such a case a new set of primes is to be chosen to compute PHC.

(c) Let $M_- = \prod_i p_i$ where p_i is a modulus for which the exponent r_i in PHC(a/b) is negative.
If no such modulus exists then $M_- = 1$.

Note that $M = M_+M_0M_-$.

3.1 Chinese Remaindering Algorithm

Let PHC(a/b) be given.

Let α in $I_{M_0} = \{0,1,2,\ldots,(M_0-1)\}$ is the integer which has the property that for each modulus p_i in M_0 ($i = 1,2,\ldots m$), $m \leq n$

$$\alpha \bmod p_i = a \cdot b^{-1} \bmod p_i$$

or in other words,

α is the integer representation mod M_0 of residues α_i mod p_i ($i=1,2,\ldots m$, $m \leq n$) for which $r_i = 0$. Then α can be computed using the Chinese Remaindering Algorithm (CRA) with inputs α_i (see Chapter 2).

Now we compute α^* defined by

$\alpha^* = [\alpha \cdot M_- \cdot M_+^{-1} \pmod{M_0}] \bmod M_0$

Then the following Extended Euclidean Algorithm (EEA) maps PHC (a/b) onto F_N.

3.2 Extended Euclidean Algorithm

This algorithm computes F_N corresponding to PHC(a/b).

Given M, find N = INT(SQRT((M-1)/2)). The extended Euclidean algorithm is described below.

Algorithm EEA

```
begin
        A  := M+ •M0 ;
        B  :=  0   ;
        A0 := M+ •α* ;
        B0 := M-    ;
        RA := A0  ;
        RB := B0  ;
        If A0/B0 ∈ FN   Stop ;
        else
        while RA/RB ∉ FN do
        begin
              Q  := quotient (A, A0) ;
              RA := A - Q•A0         ;
              RB := B - Q•B0         ;
              A  := A0  ;  A0 := RA   ;
              B  := B0  ;  B0 := RB   ;
        end;
        Result := RA/RB
end.
```

Note: If no r_i equals zero, M_0 remains undefined and hence PHC(a/b) is not defined and so not decodable.

3.3 Theorems - decoding PHC using CRA and EEA

In this section, we prove the validity of the above algorithms using the following theorem and lemmas.

3.3.1 Theorem GK [Gregory and Krishnamurthy 1984, theorem 6.39, p. 44]

If r/s is an order N-Farey fraction satisfying

$$r/s \bmod m = k \bmod m = k$$
$$\text{where } 0 < |r| \leqslant N \quad \text{and } 0 < |s| \leqslant N$$

then there exists an integer i such that $(r,s) = (a_i, b_i)$

where $\{(a_j, b_j)\}$, $j = 1,2\ldots,n+1$ is the sequence of integer pairs generated by the EEA using the initial assignments

$A:= m$, $B:= 0$, $A_0 := k$, $B_0 := 1$

(For proof, see [Gregory and Krishnamurthy 1984].)

We call the initial assignment

$$\begin{bmatrix} A:=m & B:=0 \\ A_0:=k & B_0:=1 \end{bmatrix}$$

the seeding matrix for the EEA.

3.3.2 Lemma 1

Let the seeding matrix for a EEA be:

$$\begin{bmatrix} m & 0 \\ k & 1 \end{bmatrix}$$

Let the EEA generate the sequence of computations of the form

		m	0
		k	1
q1	|	a1	b1
	|		
q2	|	a2	b2
...	|	...	
qn	|	0	bn

Then the seeding matrix

$$\begin{bmatrix} rm & 0 \\ rk & s \end{bmatrix}$$

generates the sequence

```
        rm   0

        rk   s
      ----------
 q1   |  ra1  sb1
      |
 q2   |  ra2  sb2
 ..   |   ...
 qn   |  0    sbn
```

where each left and right member is multiplied by r and s respectively.

Proof

We have a1 = m - q1 k and b1 = 0 - q1 = - q1.

Then it follows that

r a1 = r m - r q1 k and

s b1 = 0 - s q1 = - s q1 and so on.

3.3.3 Definitions

Consider a rational $x \in F_N$ expressed in PHC with the distinct primes P_i (those primes with positive exponents), N_j (primes with negative exponents) and Z_k (primes with zero exponents).

Let $x = x_n / x_d = (a / b) (\Pi_i P_i^{r_i} / \Pi_j N_j^{s_j})$

where b^{-1} exists since $GCD(b,Z_k) = 1$ for all k.

Then, we may write

$$x = (a / b) \left(\prod_i P_i^{\,r_i^{-1}} / \prod_j N_j^{\,s_j^{-1}}\right) (M_+ / M_-)$$

where $\prod P_i = M_+$, $\prod N_j = M_-$ and $\prod Z_k = M_0$ as defined above.

Further, define $x^* = (a / b)\left(\prod_i P_i^{\,r_i^{-1}} / \prod_j N_j^{\,s_j^{-1}}\right) = x\, M_- / M_+$

Then $x^* = (x_n / x_d)\,(M_- / M_+) = (x_n / M_+) / (x_d / M_-) = x_n^* / x_d^*$; in other words, the range of numerator x_n^* is (x_n / M_+) and denominator x_d^* is (x_d / M_-).

Define $\alpha^* = x^* \bmod M_0 = (x_n / M_+) / (x_d / M_-) \bmod M_0$ and

$\alpha = x_n/x_d \bmod M_0$

Note $M_+^{-1} \bmod M_0$ exists since the primes are distinct.

3.3.4 Lemma 2

Given α^* and the seeding matrix

$$\begin{bmatrix} M_0 & 0 \\ \alpha^* & 1 \end{bmatrix}$$

it follows directly from Theorem GK (see section 3.3.1) that the EEA will decode a rational, whose numerator range is x_n/M_+ and denominator range is x_d/M_-.

3.3.5 Theorem - PHC decoding

Let x_n/x_d be the desired rational to be decoded, which belongs to F_N. Then the following seeding matrix will generate the desired rational as a member of the sequence of integers pairs generated by the EEA.

$$\begin{bmatrix} M_+M_0 & 0 \\ M_+\alpha^* & M_- \end{bmatrix}$$

Proof Combining Lemma 2 and Lemma 1 we see that the sequence generated using the above seeding matrix will consist of left and right members which are multiples of M_+ and M_- respectively of the members of the EEA with the seeding matrix given in Lemma 2. Thus, we will now have $(x_n/M_+)M_+ = x_n$ and $(x_d/M_-)M_- = x_d$ occurring as a left and right member of the sequence respectively.

Remark The EEA can be modified to compute the multiplicative inverse

α^{-1} of α mod p by setting A := p, B := 0, A_0 := α and B_0 := 1 in the seeding matrix. We also replace the "while" and "if" statements by "while RA $\neq$ 1 do" and the terminating condition "if RA = 1, α^{-1} mod p = RB" respectively.

3.4 Example

a. PHC (a/b) = {(1,-1), (1,0), (5,-1), (10,0)}

P = { 3, 5, 7, 11 }

M = 1155, N = 24

Here M_0 = 5•11 = 55 ; M_+ = 1 ; thus M_+^{-1} (mod M_0) = 1

M_- = 21

Using Algorithm CRA, we get α = 21,

α^* = [21•21] mod 55 = 1

Using Algorithm EEA, we get

a/b = 1/21 $\in F_{24}$.

b. PHC(a/b) = {(1,-3), (1,1), (4,0), (4,0)}

P = { 2, 3, 5, 7 }

Thus M = 210, N = 10, M_0 = 35, M_+ = 3, M_- = 2

M_+^{-1} mod M_0 = 3^{-1} mod 35 = 12

α = 4, α^* = [4 • 2 • 12] mod 35 = 26

Using Algorithm EEA, we get (a/b) = -3/8 $\in F_{10}$.

4. Parallel Rational Arithmetic Algorithms

The parallel rational arithmetic algorithms use elementwise addition, subtraction, multiplication and division on the PHC. In the following we use

$$+ \quad , \quad - \quad , \quad \cdot \quad , \quad / \quad \text{for PHC add, subtract,}$$

multiply, divide operations respectively. The symbol $+, -, \cdot, {}^{-1}$ (superfix) are used for addition, subtraction, multiplication and inversion modulo a prime p_i, respectively.

4.1 Algorithms

Let (α_i, r_i) and (β_i, s_i) denote the two rational operands in PHC form, for $i = 1,2,\ldots,n$.

a. Addition +

The rules for addition for $i = 1,2,\ldots n$ in parallel are:

(i) $(0, z) + (\alpha_i, r_i) = (\alpha_i, r_i)$

(ii) $(\beta_i, s_i) + (0, z) = (\beta_i, s_i)$

(iii) For $\alpha_i, \beta_i \neq 0$:

$$(\alpha_i, r_i) + (\beta_i, s_i) = \begin{cases} (\gamma_i, r_i) & \text{for } r_i = s_i \\ (\alpha_i, r_i) & \text{for } r_i < s_i \\ (\beta_i, s_i) & \text{for } s_i < r_i \end{cases}$$

where $\gamma_i = (\alpha_i + \beta_i) \bmod p_i$

For example, if P = {3,5,7,11}, N=24,

PHC (5/7) = {(2,0), (3,1), (5,-1), (7,0)}

PHC (2/7) = {(2,0), (1,0), (2,-1), (5,0)}

PHC (5/7) + PHC (2/7) = {(1,0), (1,0), (0,-1), (1,0)}

To convert, we find :

M = 1155, $M_+ = 1$, $M_0 = 165$, $M_- = 7$

$1^{-1} \bmod 165 = 1$; $\alpha = 1$, $\alpha^* = 7$;

on using EEA algorithm we obtain (7/7).

b. **Subtraction** - : For i=1,2,...,n in parallel :

$(\alpha_i, r_i) - (\beta_i, s_i) = (\alpha_i, r_i) + (p_i - \beta_i, s_i)$

(This is a complemented addition)

For example, if P = {3,5,7,11}, N = 24

PHC (5/7) - PHC (2/7) = {(0,0), (4,0), (3,-1), (2,0)}

To convert we find :

M = 1155, $M_+ = 1$, $M_0 = 165$, $M_- = 7$.

$1^{-1} \bmod 165 = 1$;

$\alpha = 24$, $\alpha^* = 3$; on using EEA algorithm we obtain 3/7.

c. **Multiplication** $\cdot$: For i = 1,2,...,n in parallel :

$(\alpha_i, r_i) \cdot (\beta_i, s_i) = ([\alpha_i \cdot \beta_i] \bmod p_i, r_i + s_i)$

For example, if P = {3,5,7,11}, N = 24

PHC (5/7) $\cdot$ PHC $(\frac{1}{2})$ =

{ (2,0), (3,1), (5,-1), (7,0)} $\cdot$
{ (2,0), (3,0), (4,0), (6,0) }

= {(1.0), (4,1), (6,-1), (9,0)}

To convert we find :

$M = 1155$, $M_= = 5$, $M_- = 7$, $M_0 = 33$

$5^{-1} \bmod 33 = 20$;

$\alpha = 31$, $\alpha^* = 17$; on using EEA algorithm we obtain 5/14.

d. Division / : For $i = 1,2,\ldots,n$ in parallel :

$(\alpha_i, r_i) \; / \; (\beta_i, s_i) = ([\alpha_i \cdot \beta_i^{-1}] \bmod p_i, \; r_i - s_i)$

For example, if $P = \{3,5,7,11\}$

PHC (5/7) / PHC (2/7) = {(1,0), (3,1), (6,0), (8,0)}

To convert we find :

$M = 1155$, $M_+ = 5$, $M_0 = 231$, $M_- = 1$

$5^{-1} \bmod 231 = 185$;

$\alpha = 118$, $\alpha^* = 116$; on using EEA algorithm we obtain 5/2.

Remarks

The PHC addition and multiplication operations are commutative, but not associative, and the distributive law fails. This is analogous to the conventional floating point addition and multiplication operations.

As an example, for $p_i = 2$

$$[(1,0) + (1,0)] + (1,2) = (1,2)$$

$$(1,0) + [(1,0) + (1,2)] = (0,0)$$

This phenomenon leads to a non-canonical form of representation for PHC discussed below. Although this poses no problem for the conversion of PHC, it could force us to abandon the computation for a particular p_i when the mantissa is zero and inverse of a PHC is needed. In the above case (1,2) is invertible, but not (0,0). Thus, during a computation, if we get the form (1,2) and its inverse is to be taken the computation proceeds; if, however, we get the (0,0) form, the computation with that particular prime p_i is abandoned, broadcasting failure. This situation can occur in the g-inversion of matrices using the algorithm described in section 7 [Murthy 1988].

4.2 Non-Canonical nature of PHC

The Para-Hensel Codes are not truly canonical like the p-adic representations or Hensel Codes; there could be more than one PHC representation, when either the numerator or the denominator of a given rational is not relatively prime to one or more p_i's.

For instance, in the example of last section

{(0,0), (4,0) (3,-1), (2,0)} on conversion yields 3/7.

However, we obtain PHC (3/7) = {(1,1), (4,0), (3,-1), (2,0)} if we use the algorithm of Section 2 for P = {3,5,7,11}. This phenomenon is similar to having, for instance $\frac{3}{7} = \frac{6}{14}$ in rational numbers. The non-canonical nature does not, however, interfere with the decoding of the Para-Hensel Codes.

4.3 On-line Rational arithmetic based on Continued Fractions

There has recently been a lot of effort on hybrid (integrated) approximate and exact rational computation based on continued fraction arithmetic [Kornerup and Matula 1988, Seidensticker 1983]. Kornerup and Matula describe algorithms for online rational arithmetic based on continued fractions using a boolean cube architecture. These computations are pipelineable and this will enable us to achieve a suitable ISA design. This study is not included in this thesis.

One advantage of the use of continued fraction arithmetic is its ability to do both error-free and approximate arithmetic. In this sense, it is better than the p-adic systems which can do only exact arithmetic with rationals. The p-ary system on the other hand suffers from the disadvantage of being unable to carry out exact arithmetic. The continued fraction method can therefore be useful for integrated approximate/ error-free rational arithmetic.

5. Implementation

The Para-Hensel code system will have extensive applications in constructing massively parallel VLSI/ optical arithmetic processors to do rational computation in a specified range. For instance, if we use sixty four 16-bit processors and sixty four primes of 16- bit size, the order of Farey fraction N that can be handled is

$N = 2^{511}$ or 153 digit rational numbers.

The PHC system requires only vector processing operations in which the parallelism is at the component level. Hence it can be easily implemented on the Instruction Systolic Array (ISA). The encoding of rational numbers into PHC form involves division modulo a prime. This conversion can be done in parallel, where the computations for each prime is executed in a different processor.

Remark Note that the primes chosen for conversion to PHC should be of roughly the same order of magnitude so that processors of similar complexity (number of bits) can be used in a ISA.

The arithmetic operations in PHC can be realized in a ISA easily since this involves parallel elementwise operations, i.e., we essentially have vector processing operations.

The decoding of PHC to Farey rationals is achieved using the CRA and EEA. We have described the design and implementation of a ISA for CRA in detail in chapter 2. The EEA can also be realized on a ISA; this implementation is quite simple and we shall not describe it here.

Thus, the entire process of conversion to PHC, PHC arithmetic, and decoding can be realized on a ISA.

6. Exact Parallel Matrix Inversion

In order to illustrate the practical utility of PHC in fast exact computation we choose an example of matrix g-inversion, which is particularly sensitive to rounding errors.

The algorithm we now describe is a variant of the algorithm described earlier in Chapter 3 for the computation of the generalized inverse A^-. This algorithm is based on Greville's algorithm and computes the Moore-Penrose inverse A^+ of a rectangular matrix A satisfying the relation

$$A\,A^+\,A = A$$
$$A^+\,A\,A^+ = A^+$$
$$(AA^+)^T = AA^+$$
$$(A^+A)^T = A^+A$$

This algorithm [Murthy 1988, Gregory and Krishnamurthy 1984] is suitable for systolic computation due to parallel elementwise operations. The algorithm is fail-safe, as it computes the exact Moore-Penrose generalized inverse of a rectangular matrix, which reduces to the conventional inverse for non-singular matrices. Thus, this algorithm is well-suited for VLSI computation.

7. Algorithm

Let $A = [\, a_1 \;\; a_2 \;\; \ldots \;\; a_n]$ denote a $(m \times n)$ matrix with rational elements, where a_k $(k = 1,2,\ldots,n)$ denotes the k-th column of A, and let

$A_k = [\, a_1 \;\; a_2 \;\; \ldots \;\; a_k]$ denote the matrix consisting of the first k columns, starting with $A_1 = a_1$. Thus, $A_k = [A_{k-1} \mid a_k]$ in its partitioned form.

Step 1. Convert all the elements of A into PHC form in parallel by choosing q primes $(p_1, p_2,\ldots,p_q)$ such that the product of these primes

$M = \prod_{i=1}^{q} p_i$ satisfies the following relation

$$M \geqslant 2*N^2 + 1,$$

where N is the maximum size of the numerator or denominator, i.e., the set of Farey rationals F_N that can occur in A^+ (g-inverse of A). N is determined using the following criteria known as Hadamard's inequality [Gregory et al 1984, Springer 1986].

$$N \geq 2 \left(\text{sqrt} \prod_{i=1}^{m} \sum_{k=1}^{n} a_{ik}^2\right)$$

where a_{ik} is the element in the ith row and kth column in A.

Step 2. For each prime p_i, compute in parallel the PHC of the following terms for k = 2,...n :

$$d_k = A_{k-1}^{+} a_k$$

$$c_k = a_k - A_{k-1} d_k$$

If $c_k \neq 0$ then

$b_k = c_k^{+} = (c_k^T c_k)^{-1} c_k^T$ (superscript T denotes transpose)

else, if $c_k = 0$ (null column vector) then

$b_k = (1 + d_k^T d_k)^{-1} d_k^T A_{k-1}^{+}$

Then A_k^{+} is recursively defined for k = 2,...,n thus:

$$A_k^{+} = \begin{bmatrix} A_{k-1}^{+} - d_k b_k \\ b_k \end{bmatrix}$$

The basic step of the recursion is initiated with $A_1^{+} = 0$ for $a_1 = 0$ (null column vector); otherwise, $A_1^{+} = (a_1^T a_1)^{-1} a_1^T$

Step 3. The $PHC(A_n^+)$ for primes p_i $(i = 1,2,...,q)$ are then combined using CRA to obtain a matrix of integer values.

Step 4. The integer elements of the matrix obtained in step 3 are then converted to the equivalent Farey rationals, using the EEA algorithm. The resultant matrix obtained is A^+ (g-inverse of A).

7.1 Implementation details

a. We can compute $PHC(a_k)$ of each $(m \times 1)$ column vector in A as they are input. This can be performed in parallel for each element as well as for each prime p_i $(i = 1,2,...,q)$ using $m * q$ systolic processors, where q is the number of primes chosen.

b. In step 2, we can compute the expressions in parallel for each prime using q processors. If c_k is not null but $PHC(c_k^T c_k) = (0,z)$ for some prime p_i, then b_k is undefined. In such a case, further processing with this prime is discontinued, and a failure is broadcast. Note that the results generated for this p_i so far should not be used when the integer equivalent of g-inverse A^+ is constructed from the various $PHC(A_n^+)$ (w.r.t. primes $p_1,...,p_q$) using the CRA (see step 3).

c. Since such failures can occur, it is advisable to choose more number of primes at the start, than what is required. The extra primes chosen act as guarding primes and prevent a total breakdown of the process. Such guarding primes are useful for fault tolerant computing.

d. In step 3, the CRA to reconstruct each α in A_n^+ from the corresponding α_i's in each $PHC(A_n^+)$ (for primes $p_1,p_2,\ldots,p_q$) can be performed in parallel, using m * n systolic processors, where each $PHC(A_n^+)$ is a m x n matrix.

e. In step 4, each element in A_n^+ (of size m x n) can be converted to the corresponding Farey rational, in parallel, using the EEA with m * n processors. Thus, we obtain the g-inverse A^+, each of whose elements are $\in F_N$.

The inverses $PHC(A_k^+)$ for k = 1,2...n are generated at every fourth clock cycle. Here, each inverse is the actual inverse of the matrix formed by the columns processed so far. For instance, one can obtain the normal inverse of A_3 from $PHC(A_3^+)$ using steps 3 and 4. Indeed, the inversion procedure can be used for real-time applications, using stream-oriented programming [Murthy 1987]. Here, matrix A (a second-order stream) is incrementally processed as the columns (first-order streams) are input in real-time. Thus, one obtains a sequence of inverses of the matrices $A_1,A_2,\ldots,A_n$ respectively.

8. Example

Obtain the Moore-Penrose inverse of

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & -1 & 0 \\ 2 & 1 & 0 & 1 \end{bmatrix}$$

Here,

$$A_1 = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \; a_1 = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \; a_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \; a_3 = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \; a_4 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Here, $N \geq 12$. Thus, the primes must be chosen such that their product M satisfies the criteria $M \geq 289$ (as described above). A suitable choice of primes is 7, 11 and 13. We now compute the PHC expressions for 7, 11 and 13 in parallel (Table I).

Note that in Table I, all PHC expressions computed have a zero exponent, e.g.,

$$PHC(a_1) \; (p_i = 11) = \begin{bmatrix} (1,0) \\ (0,0) \\ (2,0) \end{bmatrix};$$

for convenience, only the mantissa is shown.
Note however, that the exponent may be non-zero for a different choice of prime, e.g.,

$$PHC(a_1) \; (p_i = 2) = \begin{bmatrix} (1,0) \\ (0,0) \\ (1,1) \end{bmatrix}$$

Now, from Table I, we combine the $PHC(A_4^+)$ for $p_i = 7$, 11 and 13 using CRA to obtain

$$A_4^+ = \begin{bmatrix} 1000 & 1000 & 1 \\ 668 & 1 & 667 \\ 668 & 0 & 667 \\ 335 & 1 & 333 \end{bmatrix}$$

Then, the integer elements in A_4^+ are converted to their Farey rational equivalents using EEA, thus obtaining the g-inverse A^+.

$$A^+ = \begin{bmatrix} -1 & -1 & 1 \\ 2/3 & 1 & -1/3 \\ 2/3 & 0 & -1/3 \\ 4/3 & 1 & -2/3 \end{bmatrix}$$

PHC EXPRESSION	$p_1 = 7$	$p_2 = 11$	$p_3 = 13$
$A_1^{\dagger} = a_1^{\dagger}(a_1^{\dagger} a_1)^{-1}$	(3 0 6)	(9 0 7)	(8 0 3)
$d_2 = A_1^{\dagger} a_2$	(6)	(7)	(3)
$c_2 = a_2 - A_1 d_2$	$\begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}$	$\begin{bmatrix} 4 \\ 1 \\ 9 \end{bmatrix}$	$\begin{bmatrix} 10 \\ 1 \\ 8 \end{bmatrix}$
$b_2 = c_2^{\dagger}(c_2^{\dagger} c_2)^{-1}$	(2 2 6)	(7 10 2)	(4 3 11)
$A_2^{\dagger} = \begin{bmatrix} A_1^{\dagger} - d_2 b_2 \\ b_2 \end{bmatrix}$	$\begin{bmatrix} 5 & 2 & 5 \\ 2 & 2 & 6 \end{bmatrix}$	$\begin{bmatrix} 4 & 7 & 4 \\ 7 & 10 & 2 \end{bmatrix}$	$\begin{bmatrix} 9 & 4 & 9 \\ 4 & 3 & 11 \end{bmatrix}$
$d_3 = A_2^{\dagger} a_3$	$\begin{bmatrix} 3 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 8 \\ 8 \end{bmatrix}$	$\begin{bmatrix} 5 \\ 1 \end{bmatrix}$
$c_3 = a_3 - A_2 d_3$	$\begin{bmatrix} 5 \\ 6 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 4 \\ 2 \\ 9 \end{bmatrix}$	$\begin{bmatrix} 9 \\ 11 \\ 2 \end{bmatrix}$
$b_3 = c_3^{\dagger}(c_3^{\dagger} c_3)^{-1}$	(2 1 6)	(2 1 10)	(2 1 12)
$A_3^{\dagger} = \begin{bmatrix} A_2^{\dagger} - d_3 b_3 \\ b_3 \end{bmatrix}$	$\begin{bmatrix} 6 & 6 & 1 \\ 2 & 2 & 6 \\ 2 & 1 & 6 \end{bmatrix}$	$\begin{bmatrix} 10 & 10 & 1 \\ 2 & 2 & 10 \\ 2 & 1 & 10 \end{bmatrix}$	$\begin{bmatrix} 12 & 12 & 1 \\ 2 & 2 & 12 \\ 2 & 1 & 12 \end{bmatrix}$
$d_4 = A_3^{\dagger} a_4$	$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$
$c_4 = a_4 - A_3 d_4$	0=null vector	0	0
$b_4 = d_4^{\dagger} A_3^{\dagger}(1 + d_4^{\dagger} d_4)^{-1}$	(6 1 4)	(5 1 3)	(10 1 8)
$A_4^{\dagger} = \begin{bmatrix} A_3^{\dagger} - d_4 b_4 \\ b_4 \end{bmatrix}$	$\begin{bmatrix} 6 & 6 & 1 \\ 3 & 1 & 2 \\ 3 & 0 & 2 \\ 6 & 1 & 4 \end{bmatrix}$	$\begin{bmatrix} 10 & 10 & 1 \\ 8 & 1 & 7 \\ 8 & 0 & 7 \\ 5 & 1 & 3 \end{bmatrix}$	$\begin{bmatrix} 12 & 12 & 1 \\ 5 & 1 & 4 \\ 5 & 0 & 4 \\ 10 & 1 & 8 \end{bmatrix}$

Table 1 G-INVERSE COMPUTATION

Remark For primes $p_i = 2$ or 3 or 5, the process fails. Therefore, these primes have been eliminated. For example,

if $p_i = 2$, then
$PHC(d_2) = ((1,1))$ and

$$PHC(c_2) = \begin{bmatrix} (1,1) \\ (1,0) \\ (1,0) \end{bmatrix}$$

Thus, $PHC(c_2^T\, c_2) = [(1,2) + (1,0)] + (1,0) = (0,0)$. When such failures arise, however, other parallel processes continue processing PHC terms for their respective primes.

9. Conclusion

In this chapter, we described a carry-free error-free rational arithmetic system based on Para-Hensel codes which is suitable for parallel computation in systolic processors. We have also considered an exact and highly-parallel matrix inversion procedure that is suitable for systolic VLSI and optical computation. Further, this algorithm is particularly useful for real-time applications if stream-oriented programming is used to incrementally process a matrix.

In the next chapter, we introduce a new parallel error-free arithmetic system for complex rationals. Their extension to parallel inversion of complex matrices is also considered.

CHAPTER 5

PARALLEL COMPLEX RATIONAL AND MATRIX ARITHMETIC

USING GAUSSIAN PRIME CODES

Abstract

This chapter introduces two closely related parallel error-free complex rational arithmetic systems. The first system uses several Gaussian primes to encode complex Farey-rational numbers (with bounded numerators and denominators) into several pairs of mutually independent codes called Gauss-Hensel Codes (GHC). The algorithms for encoding into and decoding from GHC and parallel arithmetic using GHC are described.

The second system uses powers of a Gaussian prime to obtain the p-adic representation. In this system, complex Farey rationals are encoded into a pair of mutually independent codes called Gauss p-adic codes (GPC). The algorithm for encoding into and decoding from GPC are also presented.

Both the GHC and GPC (and its multiple prime extension) enable us to perform cross product free pairwise parallel complex rational as well as matrix computations - including the inversion and g-inversion of complex matrices. Hence these systems are suitable for implementation in massively parallel Optical/VLSI architectures for complex rational/ matrix processing.

CHAPTER 5

PARALLEL COMPLEX RATIONAL AND MATRIX ARITHMETIC USING GAUSSIAN PRIME CODES

1. INTRODUCTION

In many signal processing problems where the signals are acquired from a heterodyning process, we need to deal with the arithmetic of complex rational numbers and matrices. When complex matrices are involved, it is well-known that the multiplication/inversion operations take at least four times longer than corresponding operations needed for the real matrices. To speed up the process of complex multiplication, many attempts have been made. An ingenious technique due to [Goutzoulis and Davies 1986] is based on the circularly polarized sampling technique. This technique consists of two steps :

(i) A sampling process using *delta* functions.

(ii) The use of such samples in multiplication.

The sampling process function is a series of *delta* functions, the phase of which is shifted by $\frac{\pi}{2}$; in other words, the complex amplitude part forms a

series (1,j,-1,-j,....). In the first sampling period, it samples the real part of the input function; in the second sampling period, it samples the imaginary part; in the third period, it samples the real part with negative polarity and in the fourth period, it samples the imaginary part with negative polarity. This four steps cyclic pattern is repeated to produce a *Right Hand Circularly Polarized (RCP)* sampling. Similarly, a *Left Hand Circularly Polarized (LCP)* sampling uses the *delta* function series (1,-j,-1,j,....). It can be shown mathematically that LCP and RCP are complex conjugates. Then a convolution operation on these samples realizes the complex multiplication directly.

In this chapter, we propose a novel error-free digital technique for parallel complex rational arithmetic based on the extension of Hensel Codes to Gaussian primes [Despain et al 1985, MacLane et al 1967, Collins et al 1984, Taylor et al 1985]. The Gaussian primes are primes of the form

$$p = a^2 + b^2 \quad (a,b \text{ integers})$$

where p is factorable into Gaussian integers :

$$p = (a+ib)\,(a-ib) \quad (i = \sqrt{-1})$$

It has been shown long ago by the famous mathematician Euler that primes of the form 4k+1 (k an integer) have this property.

Earlier this concept has been used for performing Gaussian residue arithmetic [Despain et al 1985]. Here we extend this concept to devise two closely related arithmetic systems for complex rationals. The

first system is based on the use of many Gaussian primes and is called 'Gauss Hensel Code' (GHC). The second system uses powers of a Gaussian prime to obtain the p-adic representation for complex rationals and is called 'Gauss p-adic code' (GPC).

The Gauss Hensel code is a natural extension of the PHC code described in Chapter 4. The algorithms for encoding a complex rational into GHC and decoding from GHC are similar to those for Hensel Codes; these are based on Chinese remaindering algorithm (CRA) and extended Euclidean algorithm (EEA) (see Chapters 2 and 4).

The Gauss p-adic code turns out to be a natural extension of the Hensel's p-adic Code studied in detail by [Gregory and Krishnamurthy 1984, Krishnamurthy 1985]. The algorithms for encoding a complex rational into a GPC and decoding from GPC are similar to those for Hensel Codes; these are based on the extended Euclidean algorithm (the CRA is not required since we have a single prime).

Both the GHC and GPC systems essentially split a complex rational number (or a matrix) into a pair of mutually independent Hensel Codes on which elementwise (Convolution-like) arithmetic operations can be performed without developing the cross products. Thus, this pair of codes is essentially a digital analogue of RCP and LCP.

Both the GHC and GPC systems permit exact parallel arithmetic operations on complex matrices. Also, the GPC permits the use of

Hensel's lifting technique [Krishnamurthy 1985] to iteratively invert or g-invert matrices.

In view of the fact that very fast VLSI algorithms are available for Chinese remaindering [Alia et al 1984] and Euclidean gcd computation [Kannan et al 1987, Yun et al 1986], complex rational arithmetic using GHC or GPC is amenable for massively parallel realization.

2. GAUSS-HENSEL CODES

2.1 Complex Farey Rationals

Consider the set of complex rationals in the canonical form

$$z = a/b + i\ c/d$$

where gcd (a,b) = 1, gcd (c,d) = 1 [gcd = greatest common divisor] and

$0 \leq |a| \leq N$, $0 < |b| \leq N$, $0 \leq |c| \leq N$, and $0 < |d| \leq N$

where N is a positive integer denoting the size of computation at hand. We call this set 'Farey Complex Rationals of Order N' denoted by F_N in analogy with the real case [Krishnamurthy and Murthy 1987, Gregory and Krishnamurthy 1984].

If, however, we choose the form

$Z = (X + iY)/(U + iV)$ then we can reduce it to the canonical form

$$Z = \frac{XU + YV}{U^2 + V^2} + i\ \frac{(YU - XV)}{U^2 + V^2}$$

For the purpose of analysis the canonical form $z = a/b + i\ c/d$ is more convenient; if we use the other form we need to make suitable redefinitions on the order of the Farey rationals used.

For instance, if X,Y,U,V are all integers less than or equal to N, then the Farey rational corresponding to Z is of the order $2N^2$.

We will now explain how to construct a unique Gauss-Hensel Code for order N Complex Farey rationals. As in all numerical computations, the choice of N limits the size of the output results. Hence the choice of N is very crucial for the choice of primes to construct the code. It turns out that n, the number of distinct primes p_k (k=1,2,...n) chosen should be such that

$$\prod_{k=1}^{n} p_k \geq 2N^2+1$$

2.2 Encoding F_N into GHC

Let $Z = a/b + i\ c/d$

Let p_k denote a Gaussian prime such that

$$p_k = (\alpha_k + i\beta_k)\ (\alpha_k - i\beta_k) = \alpha_k^2 + \beta_k^2$$

Then the following three steps can be used to construct the GHC(Z) for each p_k (assuming b^{-1}, d^{-1} exists modulo p_k; these inverses can be computed using EEA algorithm in Section 4.

Step 1 : Construct Z mod $p_k = (ab^{-1} + i \cdot cd^{-1}) \bmod p_k = e_k + i\ f_k$

Step 2 : Then GHC(Z) with respect to p_k is defined as the ordered triplet (p_k, x_k, y_k) where

$x_k = (e_k + \alpha_k^{-1}\beta_k f_k)$; $y_k = (e_k - \alpha_k^{-1}\beta_k f_k)$ where

α_k^{-1} indicates multiplicative inverse of α_k modulo p_k.

Step 3 : Construct the set of k ordered triplets (p_k, x_k, y_k) for each k = 1,2,...,n. The GHC is then represented as

$[(p_1,x_1,y_1); \ldots ; (p_n, x_n, y_n)]$

For the uniqueness of coding/decoding we need

$$M = \prod_{k=1}^{n} p_k \geq 2N^2 + 1 \text{ (see remark below)}$$

Remark

From theorem 5.14, page 27 [Gregory et al 1984] it turns out that the choice of a prime modulus p should be greater than $2N^2+1$ for constructing a unique GHC.

However, since we want to parallelize the rational arithmetic, it is necessary to choose n distinct primes p_k whose product

$$\prod_{k=1}^{n} p_k \geq 2N^2 + 1$$

In such a case the primes chosen are smaller than the numbers occurring in the computation and so it is possible that the denominators of rationals are multiples of the primes used so that the inverses do not exist.

We will discuss this aspect later.

2.3 Example

Let $Z = \frac{1}{3} + \frac{1}{2}\,i$

Choose $p_1 = 13 = (3+2i)(3-2i)$, $p_2 = 17 = (4+i)(4-i)$

$(\frac{1}{3} + \frac{1}{2}i) \bmod 13 = (9 + 7i) = e_1 + f_1 i$

$(\frac{1}{3} + \frac{1}{2}i) \bmod 17 = (6 + 9i) = e_2 + f_2 i$

Now $x_1 = (e_1 + \alpha_1^{-1}\beta_1 f_1) \bmod p_1$

where $p_1 = 13$, $\alpha_1 = 3$, $\beta_1 = 2$, $e_1 = 9$, $f_1 = 7$.

Substituting we get

$x_1 = (9 + 3^{-1} \cdot 2 \cdot 7) \bmod 13 = 5$

$y_1 = (9 - 3^{-1} \cdot 2 \cdot 7) \bmod 13 = 0.$

Similarly

$x_2 = (e_2 + \alpha_2^{-1}\beta_2 f_2) \bmod p_2$

where $p_2 = 17$, $\alpha_2 = 4$, $\beta_2 = 1$, $e_2 = 6$, $f_2 = 9$.

Substituting we get

$x_2 = (6 + 4^{-1} \cdot 1 \cdot 9) \bmod 17 = 4$

$y_2 = (6 - 4^{-1} \cdot 1 \cdot 9) \bmod 17 = 8$

Thus GHC $(\frac{1}{3} + \frac{1}{2}i) = [(13,5,0)\ ;\ (17,4,8)]$

3. Arithmetic in GHC

The arithmetic algorithms addition (+), subtraction (-), multiplication (*) and division (/) are carried out in parallel within each pair (x_k, y_k) and among all the pairs.

For this purpose we introduce the following basic definitions.

3.1 Basic Definitions

(a) Complex Conjugate :

From the construction of the GHC, it is clear that if

GHC(Z) = (p_k, x_k, y_k) then

$\bar{Z}$ = a/b - i c/d, the complex conjugate of Z has the GHC given by

GHC $(\bar{Z}) = (p_k, y_k, x_k)$

(b) Zero :

The additive null element (0 + i•0) has the GHC representation

$(p_k, 0, 0)$.

(c) Unity :

The multiplicative unit element (1 + i•0) has the GHC representation $(p_k, 1,1)$.

(d) Imaginary unity :

The imaginary unit (0 + i•1) has the representation

$(p_k, \alpha_k^{-1}\beta_k, -\alpha_k^{-1}\beta_k)$

3.2 Arithmetic Algorithms

Let GHC(Z)= $\{(p_1,x_1,y_1); \ldots(p_n,x_n,y_n)\}$

and

$$GHC(Z^1) = \{(p_1, x_1^1, y_1^1); \ldots (p_n, x_n^1, y_n^1)\}$$

(a) Addition (+) :

$$GHC(Z+Z^1) = \{(p_1,\ x_1+x_1^1 \bmod p_1,\ y_1+y_1^1 \bmod p_1), \ldots .$$
$$(p_n, x_n+x_n^1 \bmod p_n,\ y_n+y_n^1 \bmod p_n)\}$$

(b) Subtraction (-) :

$$GHC(Z-Z^1) = \{(p_1,\ x_1-x_1^1 \bmod p_1,\ y_1-y_1^1 \bmod p_1), \ldots$$
$$(p_n,\ x_n-x_n^1 \bmod p_n,\ y_n-y_n^1 \bmod p_n)\};$$

here we can replace $-x_k^1$ by $p-x_k^1$, the complement of x_k^1;

hence subtraction is complemented addition.

(c) Multiplication (*)

$$GHC\ (Z*Z^1) = \{(p_1,\ x_1*x_1^1 \bmod p_1,\ y_1*y_1^1 \bmod p_1) \ldots .$$
$$(p_n,\ x_n*x_n^1 \bmod p_n,\ y_n*y_n^1 \bmod p_n)\}$$

(d) Division (/)

$$GHC\ (Z/Z^1) = \{(p_1,\ x_1*x_1^{-1} \bmod p_1,\ y_1*y_1^{-1} \bmod p_1) \ldots .$$
$$(p_n,\ x_n*x_n^{-1} \bmod p_n,\ y_n*y_n^{-1} \bmod p_n)\}$$

If $x_k^{-1} \bmod p_k$ or $y_k^{-1} \bmod p_k$ is undefined, division operation fails for that p_k.

For example

if $(x_k, y_k) = (0,\alpha)$ or $(\alpha,0)$ or $(0,0)$ the inverse is not defined and we write

$(o,\alpha)^{-1} = (\perp, \alpha^{-1})$

$(\alpha,0)^{-1} = (\alpha^{-1}, \perp)$

$(0,0)^{-1} = (\perp, \perp)$

where $\perp$ = undefined.

3.3 Proof of Arithmetic Algorithms

For a given p

let $GHC(a_1/b_1 + ic_1/d1) = (p, x_1, y_1)$

$GHC(a_2/b_2 + ic_2/d_2) = (p, x_2, y_2)$

Let $a_1 b_1^{-1} = e_1$, $c_1 \cdot d_1^{-1} = f_1$

and $a_2 \cdot b_2^{-1} = e_2$, $c_2 \cdot d_2^{-1} = f_2$

(i) Addition/Subtraction

We have $x_1+x_2 = (e_1+e_2) + \alpha^{-1}\beta(f_1+f_2) = x$

$y_1+y_2 = (e_1+e_2) - \alpha^{-1}\beta(f_1+f_2) = y$

Hence decoding GHC (p,x,y) yields

$2^{-1}(x+y) = (e_1+e_2) \bmod p = e$

$2^{-1}\alpha\beta^{-1}(x-y) = (f_1+f_2) \bmod f = f$

This is true for all p analogously.
Then using the CRA and EEA, e and f will decode to a/b and c/d respectively provided $2N^2+1 \leq M$ where N is the range of result.

Similarly the proof for subtraction algorithm can be given.

(ii) Multiplication/Division

The proof of the algorithm rests on the fact that :

$\alpha^2 + \beta^2 = p$

or $1 + \beta^2/\alpha^2 = p/\alpha^2$

or $\beta^2/\alpha^2 \equiv -1 \bmod p \equiv (p-1) \bmod p.$

Consider

$$x = x_1 \cdot x_2 = (e_1e_2 + \alpha^{-1}\beta(f_1e_2 + e_1f_2) + (\alpha^{-1}\beta)^2 f_1f_2)$$

$$y = y_1 \cdot y_2 = (e_1e_2 - \alpha^{-1}\beta(f_1e_2 + e_1f_2) + (\alpha^{-1}\beta)^2 f_1f_2)$$

Thus

$2^{-1}(x+y) = (e_1e_2 - f_1f_2)$

$2^{-1}\alpha\beta^{-1}(x-y) = (f_1e_2 + e_1f_2)$

which correspond respectively to the real and imaginary parts

of $(e_1+i\ f_1) * (e_2+i\ f_2) \bmod p$.

The division algorithm uses multiplication with inverse elements (when defined). Hence the proof is similar.

4 Uniqueness of GHC and Decoding

4.1 Uniqueness

The decoding of GHC to its equivalent Farey rational in F_N is possible only, if it is defined and unique within the range of the Farey rationals.

Using the arguments similar to that in Theorem 5.14, page 27 of [Gregory et al 1984], it is easy to prove that the GHC is unique provided for each pair of order N-Farey rationals a/b, c/d we have the condition

$$2N^2+1 \leq M = \prod_{k=1}^{n} p_k$$

If, however, the rational Z is in the non-canonical form $Z = \frac{X+iY}{U+iV}$

where X,Y,U,V are all integers with a maximum value N, then the range of M should be,

$$2(2N^2)^2 +1 \leq M = \prod_{k=1}^{n} p_k$$

or $$M \geq 8N^4+1.$$

4.2 Decoding GHC

In decoding GHC, we first assume that all the triplets (p_k,x_k,y_k) are defined. We will later describe a thresholding technique that can be used, when some of the triplets become undefined.

The decoding of GHC proceeds in three steps :

Step 1 Finding e_k,f_k from (x_k,y_k)

For a given GHC, using each triplet (p_k,x_k,y_k) we compute e_k and f_k using :

$$e_k = 2^{-1} (x_k + y_k) \bmod p_k$$

$$f_k = 2^{-1} \alpha_k \beta_k^{-1} (x_k - y_k) \bmod p_k$$

These formula follow from the definitions of x_k,y_k.

Step 2 Finding e, f using Chinese remaindering

We then find the integers e,f that satisfy

$$e \equiv e \bmod p_k = e_k \quad (k = 1,2, \ldots n)$$

$$f \equiv f \bmod p_k = f_k \quad (k = 1,2, \ldots n)$$

using the Chinese remainder algorithm (CRA) given in Chapter 2. The CRA takes the inputs e_k, f_k and produces e and f which are integers

in the set $I_M = \{0,1,2,\ldots,(M-1)\}$

where

$$M = \prod_{k=1}^{n} p_k$$

Step 3 : Finding a/b and c/d in F_N using EEA

We now use the extended Euclidean algorithm (EEA) given in Chapter 4 to obtain a/b (and c/d). These results are obtained when e (and f)
and M are given as inputs and $2N^2 + 1 \leq M = \prod_{k=1}^{n} p_k$

4.3 Example

Let GHC(Z) = [(13,5,0) ; (17,4,8)]

Then by using Step 1 :

$e_1 = 2^{-1}\ (5+0) \bmod 13 = 9$

$f_1 = 2^{-1} \cdot 3 \cdot 2^{-1}\ (5-0) \bmod 13 = 7$

$e_2 = 2^{-1}\ (4+8) \bmod 17 = 6$

$f_2 = 2^{-1} \cdot 4 \cdot 1^{-1}\ (4-8) \bmod 17 = 9$

using Step 2 (CRA), we obtain

$e = 74$; $f = 111$

so that $e \bmod 13 = e_1 = 9$, $f \bmod 13 = f_1 = 7$

and $e \bmod 17 = e_2 = 6$, $f \bmod 17 = f_2 = 9$

using Step 3 (EEA), we obtain (Table 1)

$a/b = \frac{1}{3}$ and $c/d = \frac{1}{2}$

	221(A)	0(B)
Q	74(RA)	1(RB)
2	73	-2
1	1	3

	221(A)	0(B)
Q	111(RA)	1(RB)
1	110	-1
1	1	2

T a b l e 1

5. Failures of GHC Arithmetic

It is possible that one or more of the triplets (p_k,x_k,y_k) become undefined during the computation. In such a case e_k, f_k are not computable. Such undefined triplets (p_k,x_k,y_k) become useless any further from a computational point of view and are to be discarded soon after their creation. In such undefined cases, the reconstruction of the rationals may not always be carried out with only the defined triplets since we lost some information. If one wants to decode correctly one has to choose an additional number of primes than needed (called 'guarding primes'- see Chapter 4), say m primes where m>>n, in such a way that the products of those primes for which the triplets are defined is always greater than M. This would be similar to a threshold decoding scheme used for sharing secrets among m individuals among whom at least n have to cooperate (which is called n out of m secret lock) [Shamir 1979].

However, it is possible that several triplets become undefined so that the products of primes is less than M. In such a case failure is to be signalled and the computation is restarted with a new set of primes. Since we are assuming that a very large number of processors are available, it is perhaps possible to choose a reasonable number of guarding primes to avoid the failure. This will be an interesting area of study.

We now present the second complex rational arithmetic system based on powers of one or more primes.

6. Gaussian p-adic Codes

The conventional Hensel Code is based on using a general prime-power residue representation for a rational number. The Gaussian p-adic Code (GPC) on the other hand uses a Gaussian prime as the base.

Earlier, the Gaussian prime has been used by Despain et al and Taylor et al for performing division-free complex residue arithmetic [Despain et al 1985, Taylor et al 1985]. Our approach here is far more general in the sense that it is applicable to complex rational, as well as, matrix arithmetic, in a way similar to the Para-Hensel Code for rationals described in Chapter 4.

6.1 Complex Farey Rationals

Again, consider the set of complex rationals in the canonical form

$$z = \frac{a}{b} + i\,\frac{c}{d}$$

where a,b,c,d are integers with gcd(a,b) = 1; gcd(c,d) = 1;
(gcd = greatest common divisor)
and $0 \leq a \leq N$, $0 < b \leq N$, $0 \leq c \leq N$, $0 < d \leq N$
where N is a suitably chosen positive integer.

As before, we call this set 'Farey complex rationals of order N' denoted by F_N in analogy with the real case. We will now explain how to construct a unique GPC for z in F_N. As in all numerical computations, the choice of N limits the size of the output results. N plays a very crucial role in the choice of the prime p and its power r to construct the GPC. It turns out that for the uniqueness of GPC we should have :

$$p^r \geq 2N^2 + 1$$

(see Theorem 5.14, page 27, [Gregory and Krishnamurthy 1984]

or

$$N \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor$$

We denote this code by GPC(p,r,z).

The condition

$$N \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor$$

is analogous to the *bandwidth condition (Nyquist criterion)* given by [Goutzoulis et al 1986], namely the bandwidth of the original function B is smaller than that of the sampling frequency

$$B < \frac{1}{t_s}$$

where t_s is the sampling period.

6.2 Encoding F_N into GPC

Let $z = \frac{a}{b} + i\frac{c}{d} \in F_N$

Let p denote a Gaussian prime such that

$$p = (\alpha + i\beta)(\alpha - i\beta) = \alpha^2 + \beta^2$$

Then to establish a one-to-one and onto mapping (Isomorphism) between the elements of F_N and an ordered pair of integers modulo p^r such that $p^r \geq 2N^2+1$, we use the following three step algorithm :-

Step 1 :

Map $i = \lfloor -1$ to $\lfloor (p^r-1) \bmod p^r = w_r$

(Note : $w_1 = \sqrt{(p - 1)} \bmod p = \beta\alpha^{-1} \bmod p$,

since $\beta^2 + \alpha^2 = p$ implies $\frac{\beta^2}{\alpha^2} = -1 \bmod p$).

The choice of r for a given p is made to suit the requirement that

$$p^r \geq 2N^2 + 1$$

Step 2 :

Construct $z \bmod p^r = (ab^{-1} + icd^{-1}) \bmod p^r$

where b^{-1}, d^{-1} are multiplicative inverses modulo p^r.

(which are computed using Euclidean algorithm - see Chapter 4).

This results in Hensel Codes for the real and imaginary parts.

Let $ab^{-1} \bmod p^r = e$

$cd^{-1} \bmod p^r = f$

Step 3 :

We then define GPC(p,r,z) or simply GPC(z) as the ordered pair (x,y), where

$$x = (e + w_r f) \bmod p^r$$

$$y = (e - w_r f) \bmod p^r$$

6.3 Properties of w_r

Note that w_r is chosen as one of the square roots of $\sqrt{(p^r - 1)}$ modulo p^r. Also, it can be proved that w_r exists if w_1 exists; this is a consequence of Hensel's lemma [Krishnamurthy 1985]. Further w_1 exists for all the Gaussian primes, which are primes of the form 4k + 1 (k an integer); for these primes w_1 exist.

Also note that $w_r^2 = -1 \bmod p^r$.

Thus,

$$w_r^{-1} = -w_r \bmod p^r = (p^r - w_r) \bmod p^r$$

6.4 Finding w_r

In order to compute w_r, we use the square rooting algorithm based on Hensel-Newton method [Krishnamurthy 1985].

For this purpose, we define :

$$w_1 = \sqrt{(p - 1)} \bmod p = \beta\alpha^{-1} \bmod p = a_0$$

We then obtain the $(i + 1)$ digits a_i, of the square root of

$$(p^{i+1} - 1) \bmod p^{i+1}$$

defined by

$$a_i = \sum_{j=0}^{i} b_j p^j$$

using the following recurrence :

$$b_0 = a_0$$

and

$$b_i p^i = \left(\frac{p^{i+1} - a_{i-1}^2 - 1}{2a_{i-1}} \right) \bmod p^{i+1}$$

If we start with $a_0 = -\alpha^{-1}\beta$ we can compute the other square root whose value is $-w_r = p^r - w_r = w_r^{-1}$.

6.5 Example for computing w_r

Evaluate w_4 for $p = 5$.

We have

$$5 = 1^2 + 2^2 = \alpha^2 + \beta^2;$$

therefore

$$w_1 = 2 = a_0;$$

hence

$$b_1 \cdot 5 = \left[\frac{5^2 - 4 - 1}{2 \cdot 2}\right] \bmod 5^2 = 5;$$

thus

$$b_1 = 1$$

Hence,

$$a_1 = b_0 + b_1 \cdot 5 = 2 + 1 \cdot 5 = 7 = w_2$$

and

$$a_1^2 = 49$$

Similarly,

$$b_2 \cdot 5^2 = \left[\frac{5^3 - 49 - 1}{14}\right] \bmod 5^3$$

$$= 50$$

or $b_2 = 2$

Thus,

$$a_2 = 2 + 1 \cdot 5 + 2 \cdot 5^2 = 57 = w_3$$

and

$$a_2^2 = 3249$$

Similarly,

$$b_3 \cdot 5^3 = \left[\frac{5^4 - 3249 - 1}{6498}\right] \bmod 5^4$$

$$= 125$$

or $b_3 = 1$

and

$$a_3 = 2 + 1 \cdot 5 + 2 \cdot 5^2 + 1 \cdot 5^3 = 182$$

Thus,

$$w_4 = \sqrt{5^4 - 1} \bmod 625$$

$$= \sqrt{624} \bmod 625$$

$$= 182$$

and $-w_4 = 443 = w_4^{-1}$

We will use these results in the following sections.

Remark The algorithm given above for computing w_r is based on Hensel's lemma and is linearly convergent, i.e. it produces the digits of w_r one by one. However, it is possible to use Zassenhauss-Hensel lemma [Krishnamurthy 1985] to devise an algorithm that has quadratic or higher order convergence.

The following recurrence gives the nth order convergence

$$b_{f(i)}\, p^{f(i)} = ((p^{f(i+1)} - 1 - w_{f(i-1)}^2) / 2\, w_{f(i-1)}) \bmod p^{f(i+1)}$$

where $f(i) = n^i$

Accordingly, if w_r is needed with a large number of digits this would produce faster results. For n = 2, we get quadratic convergence.

6.6 Example for GPC encoding

Let $z = \frac{1}{2} + \frac{1}{3}\,i$; find GPC(5,4,z).

Here we have chosen p = 5, r = 4;

therefore

$$w_4 = \sqrt{624} \bmod 625 = 182$$

(see the previous subsection 6.5).

We have then

$$e = 1 \cdot 2^{-1} \bmod 625 = 313$$

$$f = 1 \cdot 3^{-1} \bmod 625 = 417$$

Then

$$x = (313 + 182 \cdot 417) \bmod 625 = 582$$

$$y = (313 - 182 \cdot 417) \bmod 625 = 44$$

Thus

$$GPC(5,4,z) = GPC(\tfrac{1}{2} + \tfrac{1}{3}i) = (582,44)$$

Remark In future, where the context is clear and p and r are specified, we simply denote GPC(p,r,z) by GPC(z).

7. Uniqueness of GPC and decoding

7.1 Uniqueness

The decoding of GPC to its equivalent complex Farey rational is possible only if it is defined and unique.

Using arguments similar to that in Theorem 5.14, page 27 [Gregory and Krishnamurthy 1984], it is easy to prove that the GPC is unique provided that for each pair of order N-Farey rationals a/b, c/d have the condition

$$2N^2 + 1 \leq M = p^r$$

7.2 Decoding GPC

The decoding of GPC proceeds in two steps :

Step 1 : Finding e and f from x and y.

For a given GPC, we compute

$$e = 2^{-1}(x + y) \bmod p^r$$

$$f = 2^{-1}w_r^{-1}(x-y) \bmod p^r$$

Note that $w_r^{-1} = -w_r$ and it exists for a Gaussian prime p.

Step 2 : Finding an element in F_N corresponding to e and f.

For this purpose, we use the extended Euclidean algorithm (see Chapter 4).

7.3 Example of decoding GPC

Let GPC(5,4,z) = (582,44) ; find z.

Step 1 : Finding e and f.

$$e = 2^{-1} \cdot (582 + 44) \bmod 625 = 313$$

$$f = 2^{-1} \cdot (-182)(582 - 44) \bmod 625 = 417$$

Step 2 : Finding F_N.

Q	625(A) 313(RA)	0(B) 1(RB)
1	312	-1
1	1	2

Q	625(A) 417(RA)	0(B) 1(RB)
1	208	-1
2	1	3

Thus, we obtain $\frac{a}{b} = \frac{1}{2}$, $\frac{c}{d} = \frac{1}{3}$, and $z = \frac{1}{2} + \frac{1}{3}\,i$

8. GPC Arithmetic

We assume that p and r are specified. Then, the arithmetic algorithms *addition* (+), *subtraction* (-), *multiplication* (*) and *division* (/) are carried out in parallel for the pair (x,y), using the same principles as for Hensel p-adic Codes.

8.1 Basic definitions

(i) Complex Conjugate

From the construction of GPC, it is clear that

if

$$GPC(z) = (x,y)$$

then

$$GPC(\bar{z}) = (y,x)$$

where $\bar{z}$ = complex conjugate.

(ii) Zero

The additive *null* elements (0 + i•0) is denoted by (0,0).

(iii) Unity

The multiplicative unit element $(1 + i \cdot 0)$ has the GPC representation $(1,1)$.

(iv) Imaginary unity

The imaginary unit $(0 + i \cdot 1)$ has the representation $(w_r, -w_r)$.

8.2 Arithmetic algorithms

Let $GPC(z_1) = (x_1, y_1)$ where $z_1 = e_1 + i \cdot f_1 \bmod p^r$ and

$GPC(z_2) = (x_2, y_2)$ where $z_2 = e_2 + i \cdot f_2 \bmod p^r$

(i) Addition (+), subtraction (-)

$$GPC(z_1 \pm z_2) = (x_1 \pm x_2 \bmod p^r,\ y_1 \pm y_2 \bmod p^r)$$

(ii) Multiplication (*)

$$GPC(z_1 * z_2) = (x_1 * x_2 \bmod p^r,\ y_1 * y_2 \bmod p^r)$$

(iii) Division (/)

$$GPC(z_1/z_2) = (x_1 * x_2^{-1} \bmod p^r,\ y_1 * y_2^{-1} \bmod p^r)$$

8.3 Proof of algorithms

(i) Addition

We have,

$$x_1 + x_2 = (e_1 + e_2) + w_r(f_1 + f_2) = x$$

$$y_1 + y_2 = (e_1 + e_2) - w_r(f_1 + f_2) = y$$

Therefore,

$$\tfrac{1}{2}(x + y) \bmod p^r = (e_1 + e_2) \bmod p^r$$

$$\tfrac{1}{2}w_r^{-1}(x - y) \bmod p^r = (f_1 + f_2) \bmod p^r$$

(ii) Subtraction

The proof is similar to (i).

(iii) Multiplication

The proof rests on the fact that

$$w_r^2 = -1 \bmod p^r$$

Consider,

$$x = x_1 * x_2 = (e_1e_2 + w_r(f_1e_2 + e_1f_2) + w_r^2f_1f_2) \bmod p^r$$

$$y = y_1 * y_2 = (e_1e_2 - w_r(f_1e_2 + e_1f_2) + w_r^2f_1f_2) \bmod p^r$$

Thus,

$$\tfrac{1}{2}(x + y) \bmod p^r = e_1e_2 - f_1f_2$$

and

$$\tfrac{1}{2}w_r^{-1}(x - y) \bmod p^r = f_1e_2 + e_1f_2$$

(iv) Division

The division algorithm uses multiplication with inverse elements. Hence, the proof is similar to multiplication.

8.4 Example

Let $z = (4 + i)$; find z^{-1}.

To compute the reciprocal of a rational complex number, we need to choose an appropriate size of r to get the result in the required range of the Farey rational (F_N).

In this case, N = 17 and so we need

$$p^r \geq 2N^2 + 1 = 578$$

If we choose p = 5, we need $r \geq 4$ to represent z^{-1} uniquely.

Thus, $w_r = 182$ and $w_r^{-1} = -182$.

We have,

$$x = 4 + 182 \cdot 1 = 186 \bmod 625$$

$$y = 4 - 182 \cdot 1 = -178 \bmod 625$$

$$= 447 \bmod 625$$

Thus,

$$GPC(z) = (186,447)$$

$$GPC(z^{-1}) = (186^{-1},447^{-1})$$

Using EEA we can compute inverses.

	625	0
Q	186	1
3	67	-3
2	52	7
1	15	-10
3	7	37
2	1	-84*
7	0	625

	625	0
Q	447	1
1	178	-1
2	91	3
1	87	-4
1	4	7
21	3	-151
1	1	158*
3	0	-625

Thus $GPC(z^{-1}) = (-84,158) \bmod 625$

$= (541,158) \bmod 625$

Decoding $GPC(z^{-1})$, we obtain

$$e = \frac{158 - 84}{2} = 37$$

and

$$f = \tfrac{1}{2}(-182)(-84 - 158) = 147$$

Decoding of e and f using EEA results in F_N

$$e = \frac{4}{17} \; ; \qquad f = -\frac{1}{17} \; .$$

9. Matrix Computations

Let M be an m x n matrix with complex rational entries

or

$$M = ((m_{ij})) = A + iB = ((a_{ij})) + i((b_{ij}))$$

where $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$.

We then write, for a specified p and r,

$$GPC(M) = (X,Y)$$

where $X = A + w_r B$ and $Y = A - w_r B$

or in elementwise form we have,

$$((x_{ij})) = ((a_{ij})) + w_r((b_{ij})) \text{ and}$$

$$((y_{ij})) = (a_{ij})) - w_r((b_{ij}))$$

Thus,

$$\frac{X + Y}{2} = A \qquad \text{and} \qquad \frac{X - Y}{2w_r} = B$$

9.1 Matrix addition/subtraction

If $GPC(M_1) = (X_1;Y_1)$, where $M_1 = A_1 + iB_1$ and

$GPC\ (M_2) = (X_2;Y_2)$ where $M_2 = A_2 + iB_2$

and M_1 and M_2 are conformable, and a_{ij} and b_{ij} respectively the real and imaginary parts of the elements of M satisfy

$$|a_{ij}|, |b_{ij}| \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor$$

Then , $GPC(M_1 \pm M_2) = (X_1 \pm X_2;\ Y_1 \pm Y_2)$

Proof :

We have by definition,

$$M_1 = \frac{X_1 + Y_1}{2} \pm i \frac{X_1 - Y_1}{2w_r}$$

$$M_2 = \frac{X_2 + Y_2}{2} \pm i \frac{X_2 - Y_2}{2w_r}$$

Thus,

$$M_1 \pm M_2 = \frac{X_1 + X_2 + Y_1 + Y_2}{2} \pm i \frac{(X_1 - Y_1) + (X_2 - Y_2)}{2w_r} \bmod p^r$$

or $GPC(M_1 \pm M_2) = (X_1 \pm X_2,\ Y_1 \pm Y_2)$

9.2 Matrix multiplication

Let M_1 and M_2 be two conformable matrices for multiplication.

Also, let the elements of product matrix $M = M_1 \cdot M_2$ be m_{ij}, such that the elements a_{ij} and b_{ij} respectively the real and imaginary parts of the elements of M satisfy

$$|a_{ij}|, |b_{ij}| \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor$$

then, $GPC(M_1 \cdot M_2) = (X_1 \cdot X_2, Y_1 \cdot Y_2)$

Proof :

We have,

$$M_1 \cdot M_2 = \left(\frac{X_1 + Y_1}{2} + i\,\frac{X_1 - Y_1}{2w_r}\right)\left(\frac{X_2 + Y_2}{2} + i\,\frac{X_2 - Y_2}{2w_r}\right)$$

$$= \frac{X_1X_2 + Y_1Y_2}{2} + i\left[\frac{X_1X_2 - Y_1Y_2}{2w_r}\right] \bmod p^r$$

Thus,

$$GPC(M_1 \cdot M_2) = (X_1 \cdot X_2, Y_1 \cdot Y_2)$$

9.3 Matrix transposition

If GPC(M) = (X,Y), then it is easily shown that M^T (transpose of M) has

$$GPC(M^T) = (X^T, Y^T)$$

The proof is straightforward.

9.4 Matrix conjugate transposition

If GPC(M) = (X,Y) then the conjugate transpose M^* satisfies

$$GPC(M^*) = (X^*, Y^*) = (Y^T, X^T)$$

The proof is straightforward.

9.5 Inversion

Let M be a non-singular square matrix modulo p^r.

Let GPC(M) = (X,Y).

Then,

$$GPC(M^{-1}) = (X^{-1}, Y^{-1})$$

provided the elements c_{ij} and d_{ij} respectively the real and imaginary parts of the elements of M^{-1} satisfy

$$|c_{ij}|, |d_{ij}| \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor ,$$

where $M^{-1} = C + iD$.

Proof :

Let M and M^{-1} be as above.

We must show $M \cdot M^{-1} = M^{-1} \cdot M = I + i \cdot 0$, where I is the square identity matrix of appropriate size.

We have,

$$M = \frac{X + Y}{2} + i \frac{X - Y}{2w_r}$$

$$M^{-1} = \frac{X^{-1} + Y^{-1}}{2} + i \frac{X^{-1} - Y^{-1}}{2w_r}$$

Thus,

$$MM^{-1} \bmod p^r = \left(\frac{X + Y}{2} + i \frac{X - Y}{2w_r} \right) \left(\frac{X^{-1} + Y^{-1}}{2} + i \frac{X^{-1} - Y^{-1}}{2w_r} \right)$$

$$= I + i \cdot 0, \quad \text{since } w_r^2 = -1 \bmod p^r$$

Similarly, $M^{-1}M = I + i \cdot 0$

Thus, the inversion of a complex rational matrix is achieved by inverting the individual codes in the pair and then decoding the result.

In fact, we have a stronger result for the generalized inversion of matrices.

9.6 Generalized Inversion

If M is an m x n matrix, then we say G is a generalized inverse if

$$MGM = M$$

If, however, G satisfies the conditions :

$$MGM = M$$

$$GMG = G$$

$$(GM)^* = GM$$

$$(MG)^* = MG$$

then, G is the *Moore Penrose generalized inverse* [Gregory and Krishnamurthy 1984, Krishnamurthy 1985, Murthy 1988] of M and denoted by M^+.

The Moore-Penrose inverse has many applications in filtering, signal processing and estimation.

We now show that,

if

$$GPC(M) = (X,Y)$$

then

$$GPC(M^+) = (X^+,Y^+)$$

provided that the elements c_{ij} and d_{ij} respectively the elements of the real and imaginary elements of M^+ satisfy

$$|c_{ij}|, |d_{ij}| \leq \sqrt{\frac{p^r - 1}{2}}$$

Proof : We will show first $MM^+M = M$.

We have,

$$MM^+ = \left(\frac{X + Y}{2} + i\,\frac{X - Y}{2w_r}\right)\left(\frac{X^+ + Y^+}{2} + i\,\frac{X^+ - Y^+}{2w_r}\right)$$

$$= \frac{XX^+ + YY^+}{2} + i\,\frac{XX^+ - YY^+}{2w_r}$$

and $$MM^+M = \left(\frac{XX^+ + YY^+}{2} + i\,\frac{XX^+ - YY^+}{2w_r}\right)\left(\frac{X + Y}{2} + i\,\frac{X - Y}{2w_r}\right)$$

$$= \frac{XX^+X + YY^+Y}{4} + \frac{XX^+X + YY^+Y}{4} + i\left[\frac{XX^+X + XX^+X - YY^+Y - YY^+Y}{4w_r}\right]$$

By definition, $XX^+X = X$, $YY^+Y = Y$, since X^+ and Y^+ are Moore-Penrose inverses of X and Y respectively.

Therefore,

$$MM^+M = \left(\frac{X + Y}{2} + i\,\frac{X - Y}{2w_r}\right) = M$$

We prove the other equalities similarly.

9.7 Example of Complex Matrix Inversion

We will illustrate the matrix inversion by a very simple example. The crucial fact to remember is that the elements of the inverse matrix are representable within the range of $\left\lfloor\frac{p^r - 1}{2}\right\rfloor$ (similar to Nyquist criterion). Therefore, at the very first step, we need to estimate the size of the elements of the inverse to make a choice of p and r so that the final decoding results in the correct answer without any overflow.

Let

$$M = \begin{bmatrix} 2 + i & 1 + i \\ 1 - i & 2 - i \end{bmatrix} = A + iB$$

with

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} ; \quad B = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$$

Since the elements of M^{-1} can be estimated to be of size N = 3, we choose p = 5, r = 2 so that $3 \leq \left\lfloor \frac{24}{2} \right\rfloor$. We then find $w_2 = \sqrt{5^2 - 1} \bmod 5^2$ (section 6.4). We have $w_2 = 7$.

Hence, GPC(5,2,M) = (X,Y) where

$$X = (A + 7B) \bmod 25 = \begin{bmatrix} 9 & 8 \\ 19 & 20 \end{bmatrix}$$

$$Y = (A - 7B) \bmod 25 = \begin{bmatrix} 20 & 19 \\ 8 & 9 \end{bmatrix}$$

We then find X^{-1}, Y^{-1} modulo 25 using any standard method [Krishnamurthy 1985].

We obtain,

$$X^{-1} \bmod 25 = \begin{bmatrix} 15 & 14 \\ 2 & 3 \end{bmatrix}$$

and

$$Y^{-1} \bmod 25 = \begin{bmatrix} 3 & 2 \\ 14 & 15 \end{bmatrix}$$

Hence,

$$M^{-1} = \left[\frac{X^{-1} + Y^{-1}}{2} + i \frac{X^{-1} - Y^{-1}}{2} \right] \bmod 25$$

$$= \begin{bmatrix} 9 + 8i & 8 + 8i \\ 8 - 8i & 9 - 8i \end{bmatrix}$$

which on decoding using EEA yields

$$M^{-1} = \begin{bmatrix} \frac{2}{3} - \frac{1}{3}i & -\frac{1}{3} - \frac{1}{3}i \\ -\frac{1}{3} + \frac{1}{3}i & \frac{2}{3} + \frac{1}{3}i \end{bmatrix}$$

as the final result.

10. Conclusion

We now make the following remarks concerning the GHC and GPC and their applications.

a. Complexity

The transformations described to construct GPC(M) of a matrix M and the final conversions are $O(n^2)$ operations. However, the matrix inversion procedure takes $\sim 2(n^3)$ operations for inverting the GPC(X,Y) whereas conventional complex matrix inversion will need $\sim (2n)^3 = 8n^3$ operations. Further, parallelism within the ordered pair results in $\sim n^3$ operations.

b. Use of several primes

The methods described can be extended to several powers of primes $(p_i^{r_i})$; however, in such a case, the results have to be combined first using the Chinese remaindering algorithm; then the Euclidean algorithm is to be applied to convert the elements to the rationals (see Chapter 4).

It is also possible to set the power of each prime $r_i = 1$ for all i. This would then correspond to the Gauss Hensel code (GHC) for a complex

rational, which is similar to the Despain et al complex arithmetic [Despain et al 1985] when division is not used.

c. Bandwidth or Range

The choice of $N \leq \left\lfloor \frac{p^r - 1}{2} \right\rfloor$ is a very crucial one; since N is not known a priori, estimation of the elements of the inverse of generalized inverse of the given matrix is needed in matrix problems. There are several guiding factors available for this estimation; one of them is the well known *Hadamard's inequality* or other criteria [Gregory and Krishnamurthy 1984] (also see Chapter 4).

d. Use of Fermat-Gauss-primes

The primes $p=2^{2**n} + 1$ are known as Fermat primes for n=1,2,3,4 for which p=5,17,257,65537. Note that these are also Gaussian primes. Using these primes the matrix multiplication process can possibly be speeded up by Fourier transform methods. This is an area for future research.

e. Hensel's Lifting Techniques

Using the GPC, we can devise iterative techniques for finding the reciprocal of complex numbers and generalized inversion of complex matrices similar to those discussed for real p-adic systems in [Gregory et al 1984, Krishnamurthy 1985]. These techniques can now be extended to invert complex polynomial matrices which are very important for several applications in Physics and Systems theory.

f. Failures of residue based techniques

It is necessary at this point to indicate some of the difficulties which are associated with the use of residue arithmetic and indicate how to take precautionary measures to avoid them.

The failures of residue methods in matrix computations essentially arise due to the fact that there need not be a one-to-one correspondence between the rank of a matrix over a real or complex field and its mapped version over the finite or the *p-adic* field. As a result, the intermediate steps may breakdown, especially when small primes are used.

Gregory and Krishnamurthy [Gregory and Krishnamurthy 1984] suggest some precautionary measures based on the comparison of ranks for different choices of the prime. Also it may be necessary to reject some previous choice of primes and select a new set of primes in certain cases.

g. Eigenvalue Problems

The GHC/GPC concept cannot be easily extended to solve the *Eigenvalue problems* which are basically non-linear involving irrational elements. The residue/p-adic techniques are essentially suitable for exact rational computations. It is not clear at this time how to extend residue/p-adic fields to deal with more general non-linear problems, although some extensions seem possible for quadratic problems [Loos 1983].

The next chapter considers some possible applications of the ISA to array processing and real-time computing.

CHAPTER 6

ISA - APPLICATIONS TO ARRAY PROCESSING AND REAL-TIME COMPUTING

ABSTRACT

This chapter considers the suitability of the ISA for vector/ array processing computers. We then consider the relationship of the ISA to wavefront processing and dataflow computing. We also describe a linear variant of the ISA known as the Linear ISA (LISA), which is an area-efficient architecture. We then briefly indicate some possible application areas of the ISA: real-time robot control applications and high-speed vision systems.

CHAPTER 6

ISA - APPLICATIONS TO ARRAY PROCESSING AND REAL-TIME COMPUTING

1. Introduction

This chapter considers the role of the ISA as a high-speed versatile array processing unit in a computer. It also discusses the suitability of the ISA for vector/array processing supercomputers. We also consider the relationship of the ISA to wavefront processing and dataflow computing. We also describe a linear variant of the ISA known as the Linear ISA (LISA). The LISA can form the basis of a powerful parallel architecture, which is both area-efficient and expandable. We also briefly indicate some possible application areas of the ISA: real-time robot-control applications and high-speed vision systems.

2. ISA - A General-purpose Array Processor

The ISA is a massively parallel programmable processor array. The ISA can be used as an attached processor that interfaces with a host machine or as a stand-alone processor connected to a global control processor. A suitable computer system configuration can then be devised consisting of the following components: a ISA processor array and a suitably linked host computer .

The ISA processor array is very flexible since any combination of the processing elements (PE) can be enabled (or disabled); this means that a PE either executes an

instruction or performs a NOOP. The ISA can also execute several different types of problems although each PE has a fixed instruction set. Also, high-speed parallel computation is performed in a ISA. Thus, several major advantages are obtained by using the ISA instead of SA as the array processing unit.

The host computer serves as the ISA global control unit. It determines and schedules the parallel processing tasks and executes them on available ISA array processors. The host also provides an external memory to store ISA programs. It supplies the ISA with instructions and selectors, and handles the transfer of data to and from the ISA.

We now discuss the suitability of the ISA as a vector processing unit in supercomputers.

2.1 ISA Applications to Vector Processing Supercomputers

The ISA is very suitable as a vector processing/ matrix processing unit in supercomputers. A ISA can carry out vector/ matrix processing operations (see Chapter 3 and 4). It uses the technique of pipelining, where operations are simultaneously performed on the components of a vector or matrix.

Further, we can introduce parallelism with different granularities (at the operational level and at a higher task level) if the PE of a ISA performs operations using the PHC/GHC systems introduced in Chapters 4 and 5:

i. At the arithmetic operational level using PHC/GHC systems; here, the arithmetic operations involve parallel elementwise vector processing operations and the granularity of parallelism is at the elemental data level.

ii. At the matrix task level using the algorithms described in Chapter 3.

Such a two-level granularity will result in massively parallel computation. Such a system will be extremely suitable for supercomputer applications such as aerodynamic flow problems and numerical weather forecasting [Perrott 1987, Babb 1988]. Thus, the ISA has the potential to be a suitable VLSI vector/ matrix processor for supercomputers.

The next section describes the relationship of the ISA to wavefront processing and dataflow computing.

3. ISA and Wavefront Processing

The ISA is closely related to the concept of wavefront processing introduced by Kung [Kung et al 1987]. In the ISA, we may consider that a set of instruction and selector diagonals propagate in wavefronts through the processing array from PE(1,1) (a point source), which is the PE at the top-left corner. These wavefronts in a ISA propagate just like the ones in a wavefront array (see Fig. 1). Also, these wavefronts never intersect with each other. That is, processors on different diagonals execute different instruction wavefronts at any given instant. The propagation of these wavefronts in a ISA is similar to the propagation of electromagnetic wavefronts. Here, each processor can be considered to be a secondary source that is responsible for activating the next front.

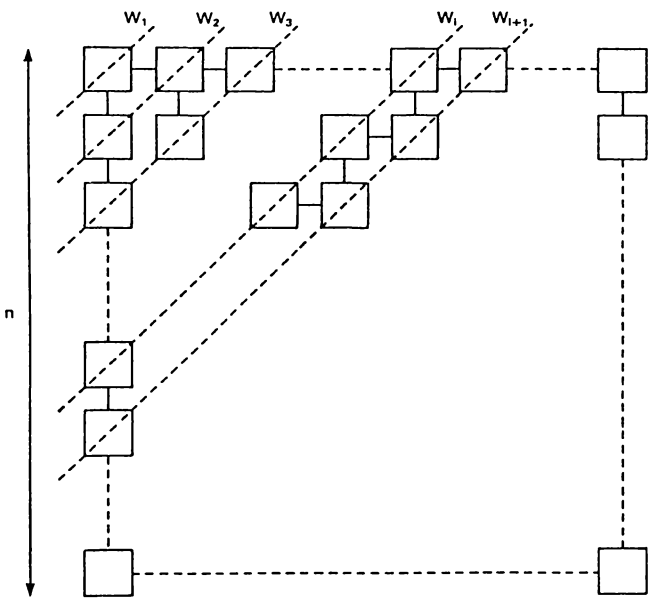


Fig. 1 Wavefront Array

The ISA can then simulate multiple wavefront algorithms where Huygen's principle applies and no two waves from the same source can interfere with each other, i.e., there is no backtracking of waves. The ISA can also simulate single wavefront algorithms, where there is at most 1 active wavefront. In this case, the ISA program (for a n x n ISA) has one instruction diagonal followed by 2n-1 NOOP diagonals to push the wavefront through the array.

Remark A programming language for wavefront arrays should express parallel data-driven computing. The Occam programming language [Inmos 1984] (which is designed to program the Inmos Transputer) is a good example of such a language. Also, since we can derive ISA from wavefront arrays, it may be possible to use Occam as a specification language for designing ISA. When simulating wavefront algorithms, it may be required to replicate the ISA program a fixed number of times. We can achieve this by using Occam to duplicate a process a fixed number of times. In Occam, construct 'seq I = [0 for N]' creates N sequential processes while 'par I = [0 for N]' creates N parallel processes [Inmos 1984]. Also, a Occam-like language called Replicating ISA language (RISAL) [Megson et al 1986] can be used for ISA program specification. We also may need to express conditional expressions 'if then else'. The Occam 'alt' construct can be used for this purpose. It is also possible to use a subset of Ada to replicate processes and express conditional expressions.

We now indicate the design of a ISA-based architecture for wavefront processing and dataflow computing.

3.1 The IWA Architecture

It seems possible to design a new wavefront processing architecture called the Instruction Wavefront Array (IWA) based on the ISA concept. In the IWA, instruction and boolean selector diagonals are pumped through a mesh-connected processor array in wavefronts, from the PE(1,1) at the top-left corner to the bottom-right corner. As in the case of the ISA, an instruction is executed in a processor only if it meets selector bit '1'.

The propagation of instruction/selector wavefronts in a IWA is similar to the propagation of electromagnetic wavefronts. The set of wavefronts propagate out from PE(1,1) (a point source). Each processor can be considered to be a secondary source that is responsible for activating the next front. This means that the order of operation sequencing can be correctly followed, without using a global clock. Conceptually, the requirement for correct timing is replaced by the requirement for correct sequencing. The computation in a IWA can then be considered to be asychronous and data-driven. Thus, the IWA concept has the potential to realize dataflow computing [Ackermann 1982, Sharp 1985].

The IWA concept outlined above has all the advantages of wavefront processing arrays (WFA). The IWA can also be considered to be more versatile than WFA since a IWA can execute different programs on the same architecture. Further, it has the flexibility of a masking mechanism introduced by the selectors. Further study, however, will be required to consider the theoretical aspects and practical

feasibility of such an architecture. In this sense, the concepts outlined are more speculative at present than realistic.

We now briefly indicate the advantages/ disadvantages of asynchronous wavefront computing (in a IWA-like architecture) as compared to synchronous computing.

1. Synchronized data transfer between a large number of processors can lead to large current surges as the components are simultaneously energized or change state [Fisher et al 1985, Kung et al 1987]. These problems are eliminated to a large extent in asynchronous processing.

2. Synchronous arrays may suffer a loss of speed when the execution times of processors differ considerably (which may happen when the granularity of parallelism is not fine), since the array has to accomodate the slowest processor by using a slower clock. In wavefront arrays, slower processors do not hold back the faster ones since they are all data-driven. However, an overhead of communication (such as handshaking protocols) is required in wavefront arrays for correct sequencing; this can be expensive.

3. A wavefront array can be more easily programmed than a synchronous array because the former only requires the assignment of computation to processors, whereas the latter requires both assignment and scheduling of computations. However, as mentioned above, this is achieved at the expense of additional communication costs in wavefront arrays.

While this thesis was under preparation, the author came across another interesting variant of the ISA, which can also realize powerful parallel architectures. This

variant is called the Linear ISA (LISA) [Megson and Evans 1986], which is described below for the sake of completeness.

4. The LISA Architecture

The concept of a linear ISA (LISA) [Megson and Evans 1986] results in a area-efficent ISA architecture. Basically, a n x n ISA array is simulated by a 1-D linear array of n LISA cells. This means that the LISA architecture has reduced chip area requirements since only n processors are used, rather than n*n required by a ISA. Each LISA cell has four bi-directional I/O lines for data, a select line passing horizontally from west to east, and an instruction line from north to south (see Fig. 2). We have the following correspondence between a LISA and a ISA:

1. LISA(i) simulates the ith row of a ISA

2. If LISA(i) acts as the (i,j)th ISA processor, then LISA(i-1) and LISA(i+1) act as the (i-1,j+1)th and (i+1,j-1)th ISA processors respectively

3. If LISA(i) acts as the (i,j)th processor of ISA at time t, then this processor acts as the (i,j+1)th ISA processor at time step t+1.

A LISA can be derived from a wavefront processor by extending Yang and Lee's technique [Yang et al 1986] which transforms a single wavefront algorithm into a linear array. There are two steps:

1. Convert the wavefront processor to an equivalent ISA as described above
2. Map the ISA to a LISA

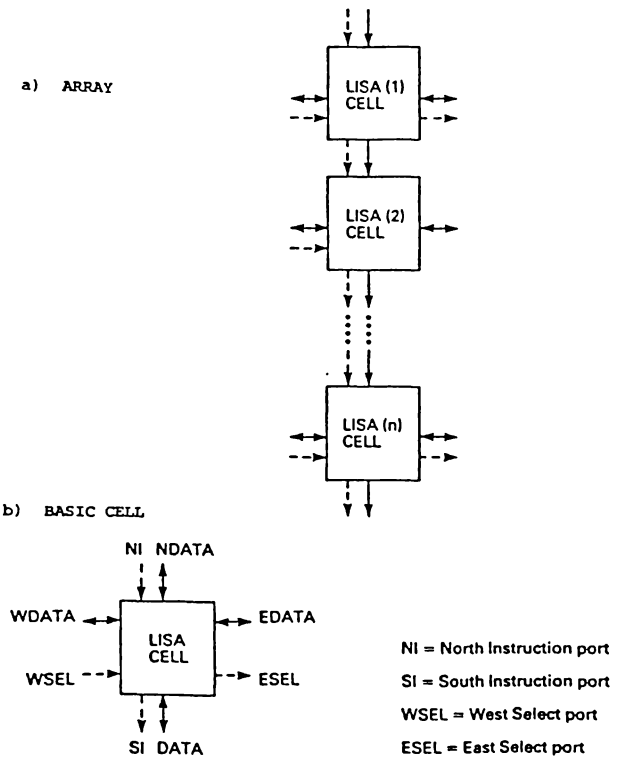

a) ARRAY
LISA (1) CELL
LISA (2) CELL
LISA (n) CELL
b) BASIC CELL
NI NDATA
WDATA
EDATA
LISA CELL
WSEL
ESEL
SI DATA
NI = North Instruction port
SI = South Instruction port
WSEL = West Select port
ESEL = East Select port

Fig. 2 LISA

The transformation of a ISA to a LISA however has undesirable features. The program size increases and it becomes large relative to the size of the mesh. Also, the LISA cell size is dependent on the mesh size. However, we can obviate these problems by using a divide and conquer approach to partition a ISA into blocks forming a block-ISA [Megson et al 1986]. We can partition a ISA into k*k blocks, where the blocksize k equals the number of LISA cells (length of LISA pipeline) required to simulate the block. Our aim is to maximize the size of the block (the number of LISA cells) and thus minimize the increase in the program size (program execution time). That is, we have a tradeoff between the number of LISA cells and program execution time.

We can then build a powerful parallel ISA architecture based on LISA chips, where each LISA chip is a pipeline (linear array) of LISA cells required to simulate a block in the ISA. This architecture is modular, permitting extension by using additional modules when desired.

The next section describes some possible applications of the ISA to real-time robot-control.

5. ISA Applications to Real-time Robot Control

The ISA can simulate wavefront processing algorithms as mentioned above. The ISA can then realize parallel asynchronous data-driven computation. The ISA can thus be used to drive and control real-time asynchronous tasks, which involve real-time recurrences.

In robot-control applications, we need to process sensory information in real-time and respond appropriately. This can be achieved by high-speed parallel computation in a ISA. Also, many robot control applications involve incremental recurrences in real-time. These computations can be realized as a systolic pipeline [de Vel and Krishnamurthy 1987, Jagadish et al 1986] of ISA processors. Here, the input stream is processed incrementally generating the output stream.

The versatility of the ISA means that it can handle real-time variation in algorithmic calls by modifying ISA instructions according to sensory feedback, i.e., actions can be performed in real-time depending on certain conditions. This ability to make quick real-time decisions results in robust and adaptable real-time systems. Such systems are very useful in obstacle avoidance during robot motion planning. Also, this reprogrammability enables robots to perform a variety of tasks.

In robotics, we use a homogeneous matrix representation called the frame [Lozano-Perez 1983] to represent the position and orientation of an object. The notion of a frame is very important in robot programming, trajectory planning and motion. It enables us to specify the position of an object with respect to one reference frame, and then determine that position with respect to another reference frame. For example, an object can be defined in terms of a local coordinate system. However, when a robot picks up the object, it needs to know the position of the object with respect to the world coordinate system. These transformations require matrix operations (usually small matrices of size n x n where n is the number of degrees of freedom). The ISA can perform matrix operations at high-speed and can be used as a real-time processor unit for robots.

We now outline some possible ISA applications in vision systems.

6. ISA Applications in High-speed Vision systems

A ISA can realize parallel polynomial and rational function interpolation as described in Chapter 2, which has applications in estimation and control. Also, a set of mesh connected ISA processors can implement multivariable interpolation (see Chapter 2) which corresponds to multidimensional object reconstruction from projections. This reconstruction of objects has many applications in image processing. In particular, it is required in the stereo problem [Guerra et al 1985], where the 3-D depth information of a scene is recovered given 2-D images taken from different view points.

The main issue in stereo computation is the correspondence problem where corresponding points in a pair of images (2-D projections) are determined. This requires an enormous amount of computation, which can be processed in real-time only if parallel systems such as the ISA are used. Indeed, the computation can be organized in a highly regular fashion and is suitable for ISA implementation.

It may be possible to use the ISA principles for multi-processor high-speed vision systems such as prism/ pyramid machines [Rosenfeld 1985]. These vision systems have applications in the computation of geometric transforms such as rotation and scaling, and in multiresolution image processing and analysis [Rosenfeld 1984].

REFERENCES

CHAPTER 1

Chang, T.L.and Fisher, P.D. [1982]: Programmable systolic array, IEEE Comcon, 48-53.

Fisher, A.L. [1981]: Systolic algorithms for running order statistics in signal and image processing, in Kung et al [1981], 265-272.

Fortes, Jose A.B. and Wah, B.W. (eds.) [1987]: Special issue on systolic arrays, Computer, **20**, July 1987.

Krishnamurthy, E.V. and Klette, R. [1981] : Fast parallel realization of matrix multiplication, Electronische Informationsverarbeitung und Kybernetik, **17**, 279-292.

Kunde, M., Lang, H.W., Schimmler, M., Schmeck, H. and Schroder, H. [1984]: The Instruction Systolic array and its relation to other models of parallel computers, Technical Report 8409, Institut fur Informatik und Praktische Mathematik, Universitat Kiel, F.R. Germany.

Kung, H.-T. and Leiserson, C.E. [1978 & 1979]: Systolic arrays (for VLSI), in Sparse Matrix Proc. 1978, 1979, Duff, I.S. and Stewart, G.W. (eds.), Academic Press, Orlando, Fla., 256-282.

Kung, H.-T. [1979]: Let's design algorithms for VLSI systems, Proc. of Caltech Conf. on VLSI: Architecture, Design, Fabrication, 65-90.

Kung, S.Y. [1988]: *VLSI Array Processors* , Prentice Hall, Englewood Cliffs, N.J.

Lang, H.W. [1986]: The Instruction systolic array, a parallel architecture for VLSI, Integration: The VLSI Journal, **4**, 65-74.

Lang, H.W. [1987]: ISA and SISA: Two variants of a general purpose systolic array architecture, Proc. Second Int. Conf. on Supercomputing, Vol. 1, 460-465.

Mead, C. and Conway, L. (eds.) [1980]: *Introduction to VLSI systems* , Addison-Wesley, Reading, Mass.

Murthy, V.K. [1988]: Exact parallel matrix inversion using Para-Hensel codes with systolic processors, Applied Optics, **27**, 2022-2024.

Schroder, H. [1986]: Systolic arrays versus Instruction systolic arrays, Technical report, Computer Science Laboratory, Australian National University, Canberra, Australia.

REFERENCES

CHAPTER 2

Ackermann, W.B. [1982]: Data flow languages, Computer, **15**, Feb. 1982, 15-25.

Baker, G.A. and Graves-Morris, P. [1981]: *Pade Approximants* , Parts I and II, Addison Wesley, New York, 1981.

Cheney, E.W. [1966]: *Introduction to Approximation Theory* , McGraw Hill, New York.

de Vel, O.Y., Murthy, V.K. and Thomas, P.G. [1988]: A distributed algorithm for interpolation and its implementation in Occam, Tech. Rep., Univ. of Waikato.

McEliece, R.J and Sarwate, D.V. [1981]: On sharing secrets and Reed-Solomon codes, Comm. ACM, 24, 583.

Hoare, C.A.R. [1985]: *Communicating Sequential Processes* , Prentice-Hall.

Karp, R.M., Miller, R.E. and Winograd, S. [1967]: The organization of computations for uniform recurrence equations, JACM, **14**, Jul. 1967, 563-590.

Krishnamurthy, E.V. [1985]: *Error-free Polynomial Matrix Computations* , Springer Verlag, New York.

Krishnamurthy, E.V. and Murthy, V.K [1987]: Error-free parallel rational arithmetic for Optical and VLSI computing, Applied Optics, **26**, 4819-4822.

Kung, S.Y [1987]: VLSI array processors, in *Systolic Arrays* , Moore, W. et al (eds.), Adam Hilger, Bristol and Boston.

Kung, S.Y [1988]: *VLSI array processors* , Prentice Hall, Englewood Cliffs, N.J.

Moldavan, D.I. [1983]: On the design of algorithms for VLSI systolic arrays, Proc. IEEE, **71**, Jan. 1983, 113-120.

Mongenet, C. and Perrin, G.R. [1987]: Synthesis of systolic arrays for inductive problems, Lecture Notes in Comp. Sc. **258**, Springer Verlag, 260-277.

Scheid, F. [1970]: *Numerical Analysis* , McGraw-Hill, New York.

Schmeck, H. [1986]: A Comparison-based instruction systolic array, in *Parallel Algorithms and Architectures* , Cosnard, M. et al (eds.), Elsevier Science Publishers B.V., North-Holland, 281-292.

Shamir, A. [1979]: How to share a secret, Comm. ACM, 22, 612.

Uhr, L. [1984]: *Algorithm - Structured Computing, Arrays and Networks* , Academic Press, New York, 1984.

R.

REFERENCES

CHAPTER 3

David, F.W and Nolle, H. [1982]: *Experimental Modelling in Engineering* , Butterworth London.

Ghosh, A. and Paparao, P. [1987]: Matrix preconditioning: a robust operation for optical linear algebra processors, Applied Optics, 26, 2734-2737.

Gregory, R.T. and Krishnamurthy, E.V. [1984]: *Methods and Applications of Error-free Computation* , Springer Verlag, New York.

Krishnamurthy, E.V. and Murthy, V.K. [1984]: Scaling relationship for model to prototype experiments: use of g-inverse of a matrix for dimensional analysis, Int. J. Math. Educ. Sci. Tech., Vol. 16, 57-62.

Kruckeberg, F. and Jaxy, M. [1987]: Mathematical models for calculating invariants in Petri nets, Advances in Petri Nets, Lect. Notes in Comp. Sc. 266, Springer Verlag, New York, 104-131.

Murata, T. [1986]: Modelling and analysis of concurrent systems, in Handbook of Software Engineering, (eds.) Ramamoorthy, C.V et al, Von Nostrand, 39-63.

Murthy, V.K. [1987]: Lazy functional stream-oriented programming, Proc. Tenth Austr. Comp. Sc. Conf. ACSC-10, Vol. 9, Deakin Univ.

Murthy, V.K. and Schroder, H. [1988]: Systolic arrays for parallel matrix g-inversion and finding Petri net invariants, Parallel Computing, (accepted for publication).

Rajbenbach, H., Fainman, Y. and Lee, S.H. [1987]: Optical implementation of an iterative algorithm for matrix inversion, Applied Optics, 26, 1024-1031.

Rao, C.R. and Mitra, S.K. [1971]: *Generalized Inverse of Matrices and its Applications* , John Wiley, New York.

Reisig, W. [1986]: *Petri Nets* , Springer Verlag, New York.

Rajopadhye, S.V. and Fujimoto, R.M. [1987]: Systolic array synthesis, LNCS 258, Springer Verlag, New York, 295-309.

Soderstrom, T. and Stewart, G.W. [1974]: On the numerical properties of an iterative method for computing the Moore-Penrose inverse, SIAM J. Numer. Anal. 11, 61-74.

Voss, K. [1987]: Nets in databases, LNCS 255, Springer Verlag, New York, 97-134.

REFERENCES

CHAPTER 4

Beaudet, P.R., Goutzoulis, A.P., Malarkey, E.C. and Bradley, J.C. [1987]: Residue Arithmetic Techniques for Optical Processing of Adaptive Phased Array Radars, Appl.Opt **25**, 3097.

Gregory, R.T. and Krishnamurthy, E.V. [1984]: *Methods and Applications of Error-free Computation*, Springer Verlag, New York.

Huang, A., Tsunoda, Y., Goodman, J.W. and Ishihara, S. [1979]: Optical Computation using Residue Arithmetic, Appl.Opt **18**, 149.

Hwang, K. and Briggs, F.A. [1984]: *Computer architecture and parallel processing*, McGraw-Hill, New York.

Jackson, J. and Casasent, D. [1983]: Optical Systolic Array Processor using Residue Arithmetic, Appl.Opt **22**, 2817.

Kogge, P.H. [1981]: *The architecture of pipelined computers*, McGraw-Hill, New York.

Kornerup, P. and Matula, D. [1988]: An on-line arithmetic unit for bit-pipelined rational arithmetic, J. Para. Dist. Comp., 5, 310-330.

Krishnamurthy, E.V. [1985]: *Error-free Polynomial Matrix Computations*, Springer Verlag, New York.

Krishnamurthy, E.V. and Murthy, V.K. [1987]: Error-free parallel rational arithmetic for Optical and VLSI computing, Applied Optics, 26, 4819-4822.

Murthy, V.K. [1987]: Lazy Functional stream-oriented programming, Proc. Tenth Australian Comp. Sc. Conf. ACSC-10, Vol. 9, Deakin Univ., Feb. 1987.

Murthy, V.K. [1988]: Exact Parallel Matrix Inversion using Para-Hensel Codes with Systolic Processors, Applied Optics 27, 2022-2024.

Psaltis, D. and Casasent, D. [1979]: Optical Residue Arithmetic: A Correlation Approach, Appl. Opt **18**, 163.

Schroder, H. and Krishnamurthy, E.V., Generalized matrix inversion using Instruction Systolic Arrays, Integration- the VLSI Journal (to appear).

Seidensticker, R.B. [1983]: Continued fraction for high speed and high accuracy computer arithmetic, IEEE Proc. 6 Symp. Comp. Arith.

Springer, J. [1986]: Exact solution of general integer systems of Linear equations, ACM Trans. Math. Software 12.

Swartzlander, E. [1986]: Digital Optical Arithmetic, Appl.Opt. 25, 3021.

Szabo, N.Z. and Tanaka, R.I. [1967]: *Residue Arithmetic and its Applications to Computer Technology*, McGraw Hill, New York.

Tai, A., Cindrich, I., Fienup, J.R. and Aleksoff, C.C. [1979]: Optical Residue Arithmetic Computer with Programmable Computation Modules, Appl. Opt. 18, 2812.

REFERENCES

CHAPTER 5

Alia, G., Barsi, F., Martinelli, E. [1984]: A fast VLSI conversion between binary and residue systems, Info. Proc. lett. 18, 141-145, 1984.

Collins, G.E., Mignotte, M. and Winkler, F. [1984]: Arithmetic in Basic Algebraic Domains, Computer Algebra, Ed. B. Buchberger, et al, Springer Verlag, 189-210, 1984.

Despain, A.M., Peterson, A.M. and Rothans, O.S. [1985]: Fast Fourier Transform Processors using Gaussian Residue Arithmetic, J. Parallel and Distributed Computing, 2, 219-237, 1985.

Goutzoulis, A.P. and Davies, D.K. [1986]: Optical Architectures for Matrix Processing, Technical Report, Westinghouse R&D Center, Pittsburg, Pennsylnvania, 1986.

Gregory, R.T. and Krishnamurthy, E.V [1984]: *Methods and Applications of Error-free Computation*, Springer Verlag, New York, 1984.

Kannan, R., Miller, G. and Rudolph, L. [1987]: Sublinear parallel Algorithm for Computing the Greatest Common Divisor of Two Integers, SIAM J. Computing, Vol.16, 7-16, 1987.

Krishnamurthy. E.V. [1985]: *Error-free Polynomial Matrix Computations*, Springer Verlag, New York, 1985.

Krishnamurthy, E.V. and Murthy, V.K. [1987]: Error-free Parallel Rational Arithmetic for Optical and VLSI Computing, Applied Optics, 26, 4819-4822, 1987.

Loos, R. [1983]: Computing in Algebraic Extensions, in Computer Algebra (eds.) Buchberger, B. et al, Springer-Verlag, 172-187.

MacLane, S. and Birkhoff, G. [1967]: Algebra, MacMillan, New York, 1967.

Murthy, V.K. [1988]: Exact Parallel Matrix Inversion Using Para Hensel Codes with Systolic Processors, Appl. Opt. 27, 2022-2024.

Shamir, A. [1979]: How to Share a Secret, Comm ACM 22, 612-614, 1979.

Szabo, N.Z. and Tanaka, R.I. [1967]: Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill, New York, 1967.

Taylor, F.J., Papdourakis, G., Skavantzos, A. and Stouraitis, A. [1985]: A radix-4 FFT using Complex RNS Arithmetic, IEEE Trans. Comp., C-34, 573-576, 1985.

Yun, D.Y.Y. and Zhang, C.N. [1986]: A fast carry free algorithm and hardware design for extended GCD computation, Proc 1986 Symp. on Symbolic and Algebraic Computing SYMSAC 86, Waterloo. ACM, 82-83, 1986.

REFERENCES

CHAPTER 6

Ackermann, W.B. [1982]: Data flow languages, IEEE Computer, 15, 15-25.

Babb, R.G. (ed.) [1988]: *Programming Parallel Processors* , Addison-Wesley, Reading, Mass.

de Vel, O.Y. and Krishnamurthy, E.V. [1987]: An iterative pipelined array architecture for the generalized matrix inversion, Info. Proc. Lett. 26, 263-267.

Fisher, A.L. and Kung, H.T. [1985]: Synchronizing large VLSI arrays, IEEE Trans. Computers, Aug. 1985, 734-740.

Guerra, C. and Kanade, T. [1985]: A systolic algorithm for stereo matching, in VLSI: Algorithms and Architectures, Bertolazzi, P. et al (eds.), Elsevier Sc. Publ., North-Holland, 103-112.

Inmos Ltd. [1984]: *The Occam Programming Manual* , Prentice-Hall International, Englewood Cliffs, New Jersey.

Jagadish, H.V. et al [1986]: A study of pipelining in computing arrays, IEEE Trans. Comp. C-35, 431-440.

Kung, S.Y. et al [1987]: Wavefront Array Processors- concept to implementation, (special issue on systolic arrays), Computer, 20, July 1987, 18-33.

Lozano-Perez, T. [1983]: Robot programming, Proc. IEEE, 71, 821-841.

Megson, G.M. and Evans, D.J. [1986]: LISA: A parallel processing architecture, Lecture Notes in Comp. Sc. 237, CONPAR 86, 361-375.

Megson, G.M. and Evans, D.J. [1986]: The soft-systolic program simulation system (SSPS), Loughborough Univ. of Tech., Comp. Stud. Rep. 272.

Perrott, R.H. [1987]: *Parallel Programming* , Addison-Wesley, Reading, Mass.

Rosenfeld, A. (ed.) [1984]: *Multiresolution Image Processing and Analysis* , Springer-Verlag, Berlin.

Rosenfeld, A. [1985]: The prism machine: an alternative to the pyramid, Jl. of Par. and Distr. Comp. 2, 404-411.

Sharp, J. [1985]: *Dataflow Computing* , Ellis-Horwood, Chichester.

Yang, C.B. and Lee, R.C.T. [1986]: The mapping of 2-D array processors to 1-D array processors, Parallel Computing, 3.