12-31-1991

Fast arithmetic operations on the hypercube using conditional sum addition and modified booth's algorithm

Umar Bin Iftikhar
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

FAST ARITHMETIC OPERATIONS ON THE HYPERCUBE USING CONDITIONAL SUM ADDITION AND MODIFIED BOOTH'S ALGORITHM

By

Umar Bin Iftikhar

A Thesis

Submitted to the Faculty of the Graduate School of the New Jersey Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering

1991

Acknowledgements

I would like to express my sincere appreciation and gratitude to Dr. Sotirios G. Ziavras, chairman of the committee, for his helpful suggestions and constractive criticism. I would also like to thank Dr. J. Carpinelli and Dr. Y. Q. Shi for serving in my thesis defense commitee and for their valuable suggestions.

Finally, I would Like to thank my mother for her continous help encouragement and secrifices she has made for me without which this work would not have been possible.

ABSTRACT

Algorithms for fast arithmetic operations (i.e., addition and multiplication) on the hypercube computer are presented. The hypercube network of dimension d interconnects $N = 2^d$ processors in such a way that each processor is directly connected to d neighboring processors; in order to communicate between processors, the maximum length of the path is d. The addition algorithm is based on the conditional sum technique. The computational time using this algorithm is $O(\log_2 N+q)$ where q is the number of the bits per processor in the hypercube of N processors. Operands of size N*q are distributed among the hypercube processors using high-order interleaving. Only two bits are exchanged in every cycle of communication for a total of $\log_2 N$ communication cycles. A modified version of Booth's algorithm is adopted for fast multiplication and generates $\lceil n/2 \rceil$ partial products, where the size of the multiplier is n bits. These $\lceil n/2 \rceil$ partial products are half of those required by conventional multiplication. Each processor of the hypercube generates $\lceil n/2N \rceil$ partial products, simultaneously. These partial products are distributed among the N processors using high-order interleaving. Each processor contains $\lceil (n+m)/N \rceil$ bits, where n and m are the numbers of bits in the multiplier and the multiplicand respectively. A carry-save addition (CSA) technique is used to add these multiple partial products. These multiple partial products are reduced to two operands (i.e., carry and sum) in $(\lceil n/2 \rceil - 2)$ iterations, which are finally added using the algorithm described earlier. The time taken by the multiplication process is $O(log_2 N(nq+1) + n(q+1+1/N) - q)$. These algorithms are simulated and comparative analysis is performed for various sizes of operands and hypercubes of various dimensions.

APPROVAL SHEET

Title of Thesis: Fast Arithmetic Operations on the Hypercube using Conditional Sum Addition Logic and Modified Booth's Algorithm

Candidate: Umar Bin Iftikhar
Master of Science in Electrical Engineer, 1991

Thesis and Abstract Approved by the Examining Commitee:

Dr. Sotirios G. Ziavras, Advisor — Date
Assistant Professor
Department of Electrical and Computer Engineering

Dr J. Carpinelli — Date
Assistant Professor
Department of Electrical and Computer Engineering

Dr. Y. Q. Shi — Date
Assistant Professor
Department of Electrical and Computer Engineering

New Jersey Institute of Technology, Newark, New Jersey.

Vita

Name Umar Bin Iftikhar

Permanent Address

Present Address

Degree and date to be conferred Master of Science in Electrical Engineering, December 1991

Date of Birth

Education

New Jersey Institute of Technology, Newark, NJ 07102

M.S.E.E, Dec, 1991

N.E.D University of Engineering & Technology, Karachi, PAKISTAN

B.Eng (Electrical) Dec, 1988

Adam Jee Science College, Karachi, PAKISTAN

H.Sc (Pre-Engineering) August 1982

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 INTRODUCTION TO COMPUTER ARITHMETIC

Computational speeds have unexpectedly increased during the past three decades due to faster and denser technologies and new concepts in computer architecture. The speed of large-scale processors has doubled approximately every three years. The arithmetic logic unit (ALU) is an essential part of a high speed processor and is responsible for all arithmetic and logical operations. Research in large scale integration (LSI) and very large scale integration (VLSI) technologies have made it possible to design sophisticated, high performance arithmetic processors for modern digital computers.

In order to achieve a high degree of speed-up in computation, the design of high speed processing elements is not enough; a suitable algorithm is also

required. Along with a high speed processing element and a good algorithm, the selection of a suitable number system might help to further improve the speed. Logical, shift and arithmetic operations are considered as the basis of all computations. The speed of these operations can be optimized by proper selection of the following:

(1) Number system.

(2) Suitable algorithm.

(3) High speed processing element.

1.1.1 NUMBER SYSTEM

Operands are represented in digital computers by using an appropriate number system Historically, we need a number system for counting, and the infinite natural numbers (i.e., 1,2,3,4,...) have been used for thousands of years for this purpose But with the development of digital technology, various number systems have been developed according to the requirement of digital circuit and speed of computation. The binary number system is the most conventional and easily implemented system for the internal use of digital computers. It is a positional number system; numbers are represented by a vector of bits in which each bit is weighted according to its position in the vector. Along with the binary number system, we also have other positional number systems of different radices, e.g., octal, hexadecimal, decimal and binary coded decimal [1]. In addition, redundant numbers [2], residue numbers [1] and positional residue numbers [4] play important roles in fast arithmetic operations.

1.1.2 SUITABLE ALGORITHM

Engineers and mathematicians have been working on the development of cost-effective and efficient algorithms for arithmetic and logical operations. Several such algorithms have been developed by these efforts; some of these algorithms are discussed in Chapter 3 and 4.

1.1.3 HIGH SPEED PROCESSORS

The developments in the computer industry over the past four decades can be divided into four generations. The computer industry went through the age of relays and vacuum tubes (1940-1950s) to the age of diodes and transistors (1950-1960s). In the first generation, the first electronic digital computer, namely ENIAC (Electronic Numerical Integrator and Computer), was developed, in 1946. which used relays and vacuum tubes for switching [3]. In the second generation, Bell labs developed TRADIC [3]. Then small and medium-scale integrated (SSI/MSI) circuits were introduced (1960-1970s); in this period CDC-6000 and CDC-7600 were developed. IBM 360/91, Illiac IV, TI-ASC, Cyber-175 and STAR-100 were the major break throughs of the early seventies. From the 1970's, large-scale and very-large-scale integrated (LSI/VLSI) technology have been playing important roles in the computer industry. During this time, high performance super computers were developed. To conclude, the increase in processor speed, reliability, and reduction in the hardware cost have greatly enhanced the computers' performance.

1.2 ADVANCE COMPUTER ARITHMETIC

Concurrent computers are the major breakthrough in the field of computation. Parallel processing has made operations enormously fast during the last ten years. Although the cost of computation has increased by the use of parallel processing, speed, accuracy and performance of the computation have increased dramatically as well .

In distributed processor systems, a task is distributed among the processors and is simultaneously processed by all the processors. These processors may be connected in various topologies, e.g., linear array, ring, tree, mesh, systolic array, pyramid, hypercube, etc. This processing paradigm has been adopted in almost every type of problem. Since the middle of 1980's , hypercube computers have been widely used due to their small diameter, high degree of fault tolerance and rich interconnection structure.

A large variety of algorithms have been developed for hypercube parallel computers [5]. The development of fast arithmetic operations for masively-parallel hypercube systems is also a major research topic. Addition is considered as the back bone of computer arithmetic operations, because all other arithmetic operations are based on addition; e.g., subtraction is the addition of one operand (minuend) to the 2's complement of the subtrahend. The first electronic computers used ripple carry addition, and parallelism could not be achieved in that method because resulting bits at any position depend upon carry-out of the preceding bit position. This thesis deals with addition and multiplication for hypercube-based systems.

1.2.1 ADDITION AND SUBTRACTION

Several techniques have been developed for addition and subtraction, on uniprocessor and multiprocessor systems [1],[2],[4],[6], [7], [8],[9] and [10]. A good addition algorithm is essential not only for the addition operation but also for other operations (i.e., multiplication, subtraction and division).

The conventional method for addition is Ripple Carry Addition, in which a carry may have to travel from the least significant bit to the most significant bit position. So, the addition of each bit pair requires the carry from the previous bit pair. Addition between the *i*th bit of operands A and B is performed as follows:

$$S_i = A_i \oplus B_i \oplus C_{i-1} \qquad (1.2.1)$$

where C_{i-1} is the carry-out of the (i-1)th bit position. The carry generated at any bit position *i* is

$$C_i = A_iB_i + C_{i-1}(A_i + B_i) \qquad (1.2.2)$$

Thus, by this method two *q*-bit operands may take up to *q-1* carry delays and one sum delay. So, in case of high speed arithmetic this method is not normally adopted; however, if minimum amount of hardware is required and the high speed requirement is not critical, then Ripple Carry addition can be proven advantageous.

High speed parallel arithmetic operations can be achieved either by using carry free addition algorithms or by algorithms in which carries at each bit position are generated simultaneously, prior to addition. Carries should be generated in such a way that this process does not take long time, as in the carry look ahead method in which the carry at any bit position does not explicitly depend on the preceding one but can be expressed as a function of the relevant augment and addend bits [2] and all carries are calculated prior to addition. This technique increases the speed

considerably but the hardware required to perform this operation increases as well. These algorithms and carry free addition algorithms can be implemented in a parallel processing environment, because the overall task can be divided in a number of somewhat independent subtasks, which can be processed independently to some extent. Ziavras and Davis have used carry look-ahead addition algorithm for hypercube systems, (they have also presented results for the Connection Machine) [10]. The computation time of their algorithm is $O(\log_2 p(1+q))$ where p is the number of processors and q is the number of bits in each processor; this algorithm is discussed in more detail later. Other algorithms which can be implemented in a parallel processing environment with some limitations are Independent-Dependent carry addition (IDA) [7], Distant-Carry addition (DCA) [7] and Conditional sum addition [6],[7],[1]. The conditional sum addition logic is adopted in this research to perform fast addition on hypercube computers. Some carry free addition algorithms can also be used to get extremely fast addition on distributed processor system. Addition in the redundant number system [1], [2], residue number system [1] and positional residue number system [4] can provide carry free operations which are discussed later in Chapter 3. The only limitation with these algorithms is that the conversion process from these number systems to conventional binary numbers can not be performed concurrently. This is a sequential operation which takes a large amount of time; the positional residue number system can overcome this problem to some extent [4].

1.2.2. MULTIPLICATION

Multiplication is an important arithmetic operation, performed between two operands, namely the multiplicand and the multiplier. Arithmetic processors for high speed multiplication use various add and shift methods.

Multiplication can be implemented as repeated additions and shifts; the operand which is shifted and added is the multiplicand and the number of additions is equal to the number of bits in the multiplier. The multiplication of two operands A and B having p and q bits respectively will generate a product with $(p+q)$ bits. The speed of multiplication can be improved by designing very high speed VLSI circuit multipliers, but the actual speed up heavily depends upon the technique adopted for multiplication. The first thing to consider is the proper selection of the multiplier from the two operands of the multiplication; the operand with the smaller number of bits is chosen as the multiplier, which causes the generation of a smaller number of partial products. Then, to achieve additional speed up, an algorithm must be chosen to further reduce the number of partial products generated; e.g., in multiple bit scanning and multiplier bit recoding techniques, as in the Booth's algorithm, where strings of 0's and 1's are skipped and the number of partial products generated is equal to the number of variations from 0's to 1's or vice versa in the multiplier. Various cases for Booth's algorithm are shown in Table 1.1. Table 1.2 shows the technique used for recoding the multiplier. Table 1.1 shows that the number of partial products in the worst case is the same as in the conventional sequential add and shift method. However in a modified version of Booth's algorithm, which is based on bit pair recoding, the number of partial products does not depend upon the bit patterns in the multiplier. This guarantees that an n-bit multiplier will generate $\lceil n/2 \rceil$ partial products. This technique is equally valid for positive and negative numbers [1] [2].

Another way to improve the speed of multiplication, is to adopt an addition algorithm that adds multiple operands (i.e., partial products). Multiple operands are required to be reduced to two operands by using any fast technique which does not waste time in carry propagation. One of the first reduction implementations was the Wallace tree reduction technique [1]. This technique is also adopted in this

Table 1.1 Booth recoded multipliers

Worst Case	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Normal	1	1	0	0	1	0	0	1	1	0	0	1	1	1	1	0
Good	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
Worst Case	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1
Normal	0	-1	0	+1	-1	0	+1	0	-1	0	+1	0	0	0	-1	0
Good	0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Table 1.2 Booth multiplier Recoding table

Multiplier		Version of multiplicand
Bit i	Bit i-1	determined by bit i
0	0	0 x multiplicand
0	1	+1 x multiplicand
1	0	-1 x multiplicand
1	1	0 x multiplicand

research for the reduction. Carry save addition (CSA) is used to reduce three partial products to two (i.e., carry and sum) without wasting time in carry propagation. In designing the hardware for a Wallace tree, the number of stages S required to reduce P partial products to two is given by:

$$S = \lceil \log_{3/2} (P/2) \rceil$$

$$= \lceil \log_{1.5} (P/2) \rceil.$$

This reduction gives two final operands, which can be added by various techniques. While the Wallace tree technique uses carry look ahead addition to add these operands, our research uses the conditional sum technique which is even faster than the carry look ahead technique. The time required by the conventional Wallace tree technique using CLA addition is

$$T \leq S.2 + CLA(2m)$$

where 2m is the total number of bits in the operands to be added. Although the Wallace tree technique for reduction is very fast, it is also expensive, because it requires a lot of hardware. Anderson proposed an iteration technique to reduce the hardware at the expense of time. He used smaller trees, where each tree performed I iterations. The time for this algorithm is given by:

$$T = I(2 \lceil \log 3/2 \lceil n/I+1 \rceil /2 \rceil + CLA(m+\lceil n/I+1 \rceil))$$

where

n = number of operands to be reduced,

n/I = operands reduced in one cycle.

In [1], parallel multiplication algorithms are classified in three classes:

1. Simultaneous generation of partial products and simultaneous reduction.
2. Simultaneous generation of partial products and iterative reduction.
3. Iterative arrays of cells.

Some representative algorithms are discussed in more detail in Chapter 4.

1.3 Motivations and Objectives

As computers have become an essential part of almost every part of our life, they are commonly used in scientific, business, medical, engineering and defence applications. Modern industry is fully controlled by computers. A major component a computer is its arithmetic and logic unit (ALU). The speed of a computer may be increased by speeding up the operations carried out by the ALU. For example, the addition and multiplication algorithms could be improved with respect to the required amount time for their implementation. In addition, there are various ways to improve the speed of computation in a parallel processor system. Arithmetic operations have been implemented on various parallel structures, but most of them involve a large amount of communication between the processors, due to the underlying parallel structure. Although these algorithms are relatively fast, further improvements are still possible.

Presently, the hypercube structure is the most commonly used parallel structure in parallel processing due to its powerful interconnection features, and the majority of the problems can be efficiently implemented on the hypercube.

An addition algorithm which uses a hypercube topology and applies a carry look-ahead technique was proposed by Ziavras and Davis [10]. Although it is one of the fastest addition algorithms (its computation complexity is the lowest possible), but improvements in constants of the computation time may be possible.

This research presents a hypercube addition algorithm which involves much less computation with in the processors and only two bits are exchanged in every communication cycle (the algorithm in [10] exchanges more information in every communication cycle due to the information required for the computation of carry generate and carry propagate terms). Our algorithm is based on the conditional sum logic. The hardware complexity used to implement this logic is proportional to the

length of the operands, but this algorithm gives good results if implemented on the hypercube.

A hypercube multiplication is also proposed. It is based on the modified Booth's algorithm which speeds up the multiplication process by reducing the number of partial products to half. Extra speed is obtained by using parallel processing and the new addition algorithm.

1.4 Outline

This report is organized as follows:

Chapter 2 gives a description of the hypercube structure and discusses its important topological properties. Some other interconnection networks are also discussed and the mapping of the linear array and the mesh onto the hypercube using the reflected binary Gray code are presented. Chapter 3 presents the new addition algorithm along with some other existing algorithms for addition. The last section of this chapter compares the proposed algorithm with some other existing algorithms. Chapter 4 presents the new multiplication algorithm along with other existing algorithms for multiplication. The last section provides a comparison between proposed and conventional algorithms. Chapter 5 presents conclusions.

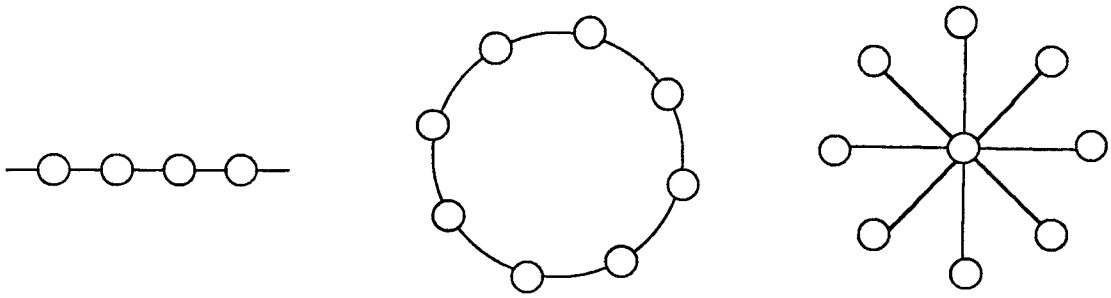
CHAPTER 2

THE HYPERCUBE TOPOLOGY

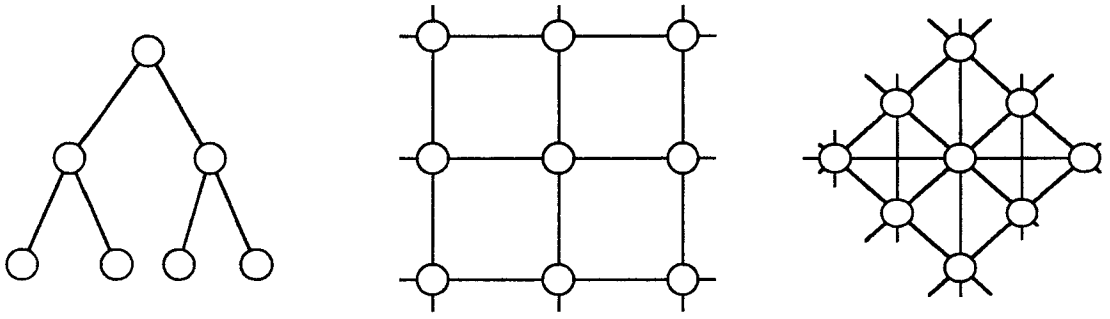
Fast and efficient computers are in high demand in many scientific, engineering, energy resource, medical, military, artificial intelligence, and basic research areas. Parallel processing computers are needed to meet these demands. This chapter describes some parallel structures and introduces the hypercube structure whose knowledge is required to understand the research work. Section 2.1 gives a brief introduction to different parallel processor networks. In Section 2.2, the hypercube topology is discussed; then, the topological properties of the hypercube are discussed in Section 2.3.

2.1 Interconnection Networks

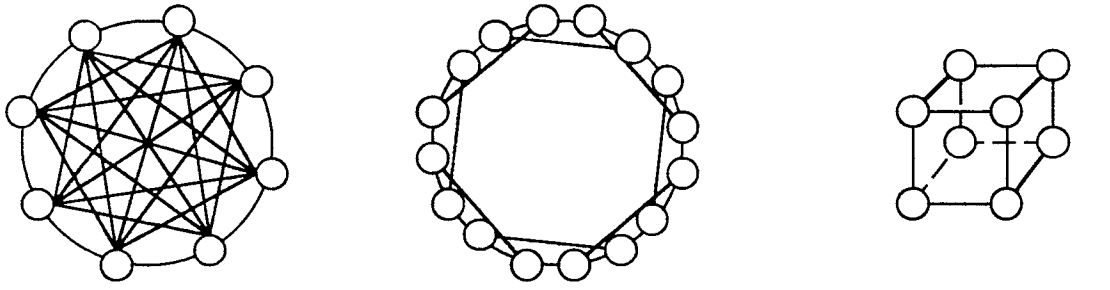
In this section, various network topologies are described. These network topologies may be classified according to the number of dimensions they implement. They are classified as one-dimensional, two-dimensional, three dimensional and multiple-dimensional structures Figure 2.1 shows some of the static interconnection



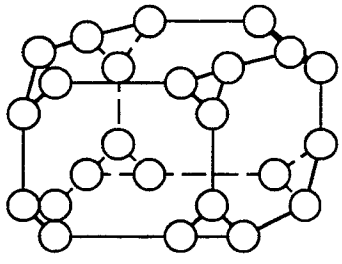
(*a*) Linear array (*b*) Ring (*c*) Star



(*d*) Tree (*e*) Near-neighbor mesh (*f*) Systolic array

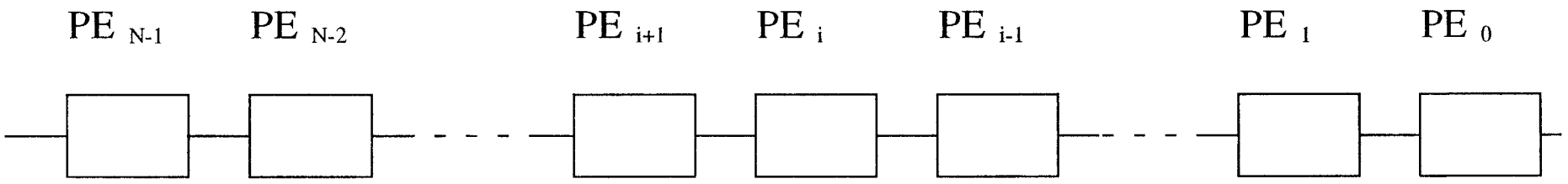


(*g*) Completely connected (*h*) Chordal ring (*h*) 3-cube



(*i*) 3-cube-connected cycle

FIG - 2.1 Static interconnection network topologies.



Linear array of N processors

Figure - 2.2

network topologies. The linear array is categorized as a one-dimensional topology; two dimensional topologies are the ring, star, tree, mesh and systolic array. Three-dimensional network topologies include the completely connected chordal ring, 3-cube and 3-cube connected cycle networks. The N-dimensional hypercube is a good example of a multidimensional interconnection network in which 2^N nodes are available. Each node is directly connected to N different nodes in N different dimensions. Some of the important topologies are further discussed in the following sections.

2.1.1 The Linear Array Network

The one-dimensional network topology shown in Fig 2.1 (a) is called linear array. In this type of interconnection, each processor, except the processors at the ends of the string, can communicate with two adjacent processors; for example any processor PE_i can transfer its data directly to PE_{i+1} and PE_{i-1}. But, in the string of N processors, as shown in Fig 2.2, when any distant processors wish to communicate with each other, then intermediate processors work as switches.
The only difference between the linear and the ring structures is that, in the ring structure processors at the end of the string are also connected with each other, so that every processor is connected with two adjacent processors in the ring structure.

2.1.2 The Mesh-Connected Network

In the 2D mesh-connected network, as shown the Fig 2.3(a) (b), each processor is connected directly with four adjacent processors. The mesh connected network has been implemented in the Illiac-IV parallel processors systems with 64 PEs [3].

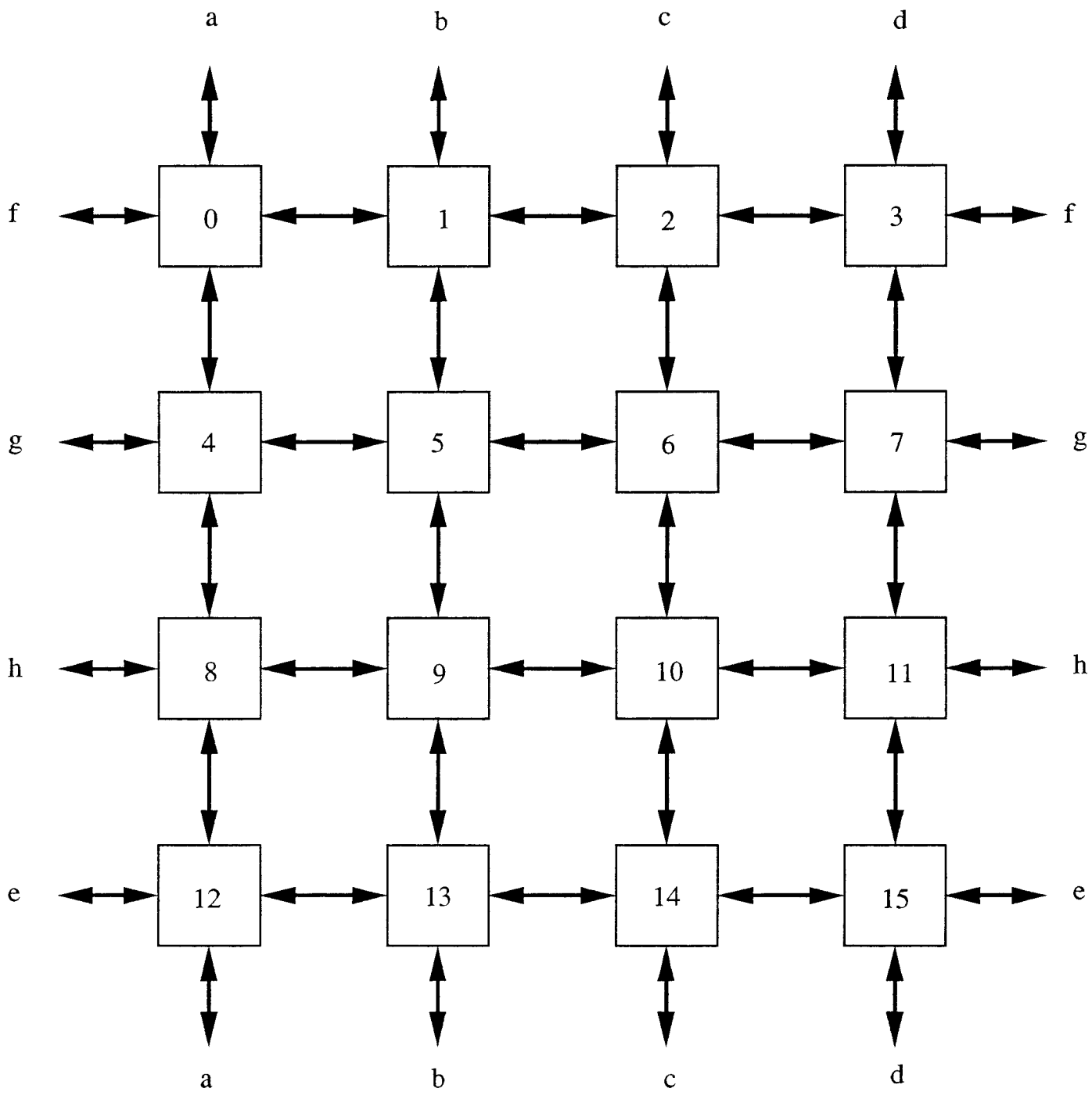


Fig - 2.3 (a). A 2-D mesh network with N = 16 PEs.

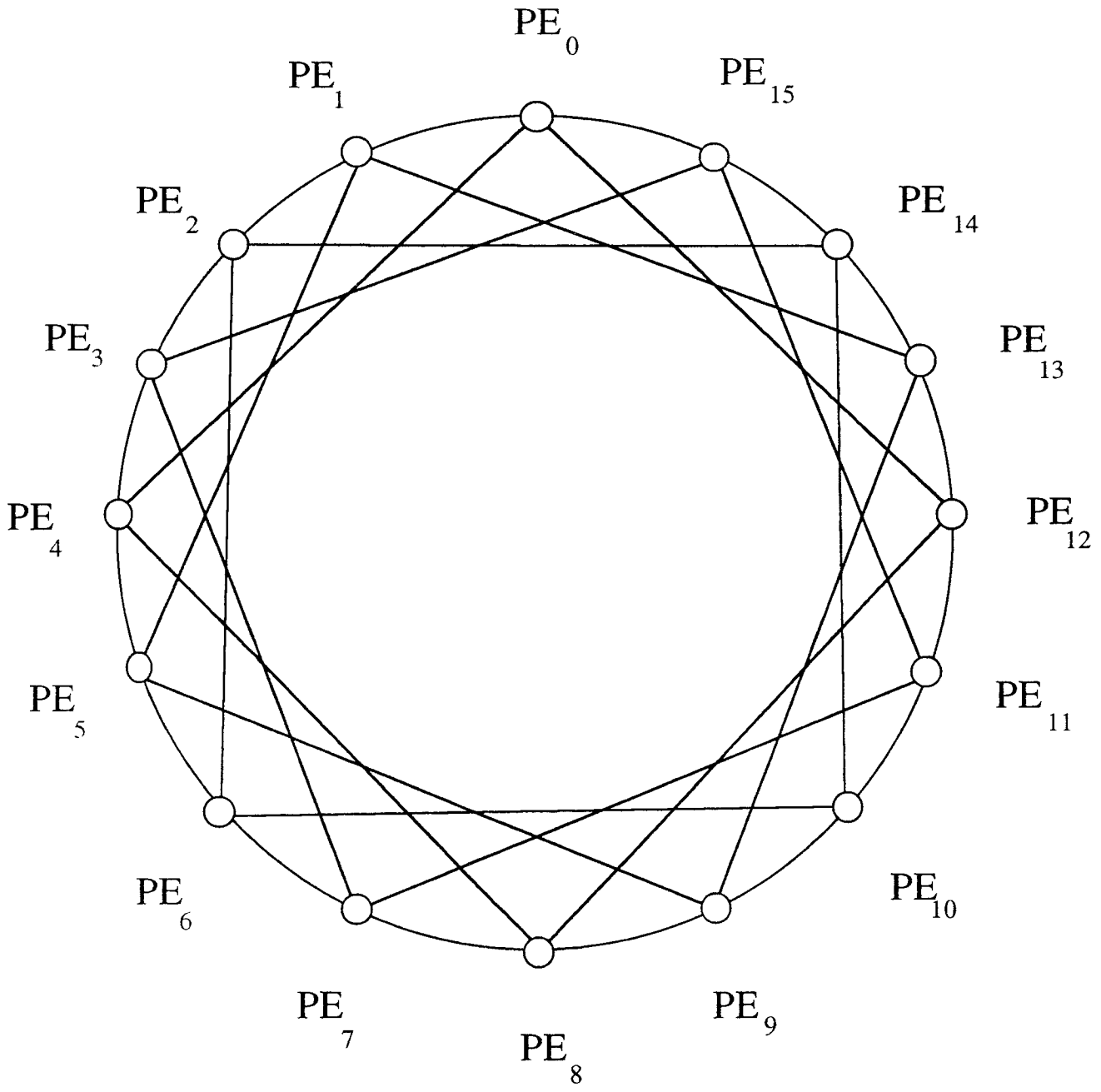


Fig - 2.3 (b). A 2-D Torus network with N = 16 PEs.

Each PE_i can communicate directly with any one of PE_{i+1}, PE_{i-1}, PE_{i+d} and PE_{i-d}, where $d = \sqrt{N}$ is the number of processors in one dimension of the mesh; the following four routing functions are described for the mesh structure.

$$R_{+1} = (i+1) \bmod N$$

$$R_{-1} = (i-1) \bmod N$$

$$R_{+d} = (i+d) \bmod N$$

$$R_{-d} = (i-d) \bmod N \qquad \text{where } 0 \leq i \leq N-1$$

In the torus topology, PE's at the edges of the square are connected with the processors at the opposite edges of the square. The mesh-connection in Fig. 2.3(a) can be "redrawn" as the torus topology Fig. 2.3(b). In order to communicate between two processors which are not directly connected, intermediate processors work as switches, and the maximum length of a path is equal to d in the torus and 2d in the mesh

2.1.3 The Completely-Connected Network

A completely connected network is categorized as a three dimensional network In a completely-connected network, as shown in Fig. 2.1(g), all PE's are directly connected with each other. This type of network is extremely complex, and complexity increases with the increase in the number of processors. However, the network provides good data communication features.

2.2 The Hypercube Network

Various parallel processor structures have been used in parallel systems. In recent years, hypercube computers have become popular parallel computers for a variety of applications because of their powerful network which is characterized by a small diameter, regularity and high degree of fault tolerance. Most of the topologies like the linear array, mesh, ring and pyramid can easily be mapped into the hypercube [12]. Therefore, most of the applications for these structures can be implemented on the hypercube very efficiently. Formally an n-dimensional hypercube containing 2^n nodes. Nodes are connected directly with each other if and only if their binary addresses differ by a single bit. Hypercubes of zero, one, two, three and four dimensions are shown in Fig 2.4. Hypercube computers are loosely coupled parallel processor systems based on the binary n-cube network, also known as cosmic cube, n-cube, binary n-cube, Boolean n-cube, etc.

Various parallel computers have been developed using this structure. The Connection machine is one of the best known and is manufactured by Thinking Machines Corp. It may contain up to 65536 processors, operating in the SIMD mode. The topological properties of the hypercube structure are presented in more detail in the next section.

2.3 Topological Properties of the Hypercube

In the d dimensional hypercube H_d, each processor is directly connected with d neighboring processors. Each processor has a d-bit binary address in the interval 0 to 2^d-1. In a hypercube computer, processing elements (PEs) are placed at each vertex of the hypercube and the edges of the hypercube represent communication

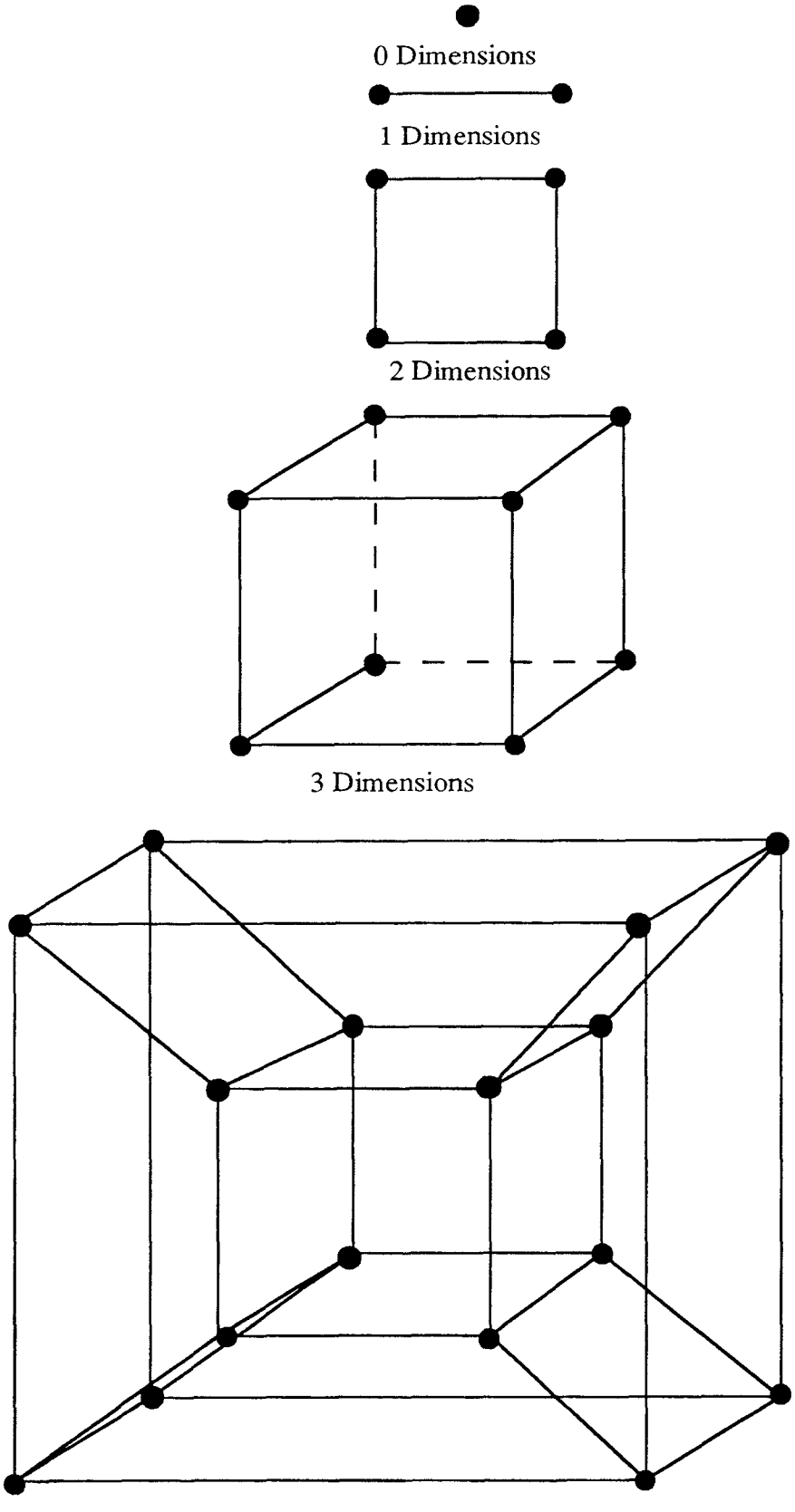


Fig . 2.4 Hypercubes of different dimensions

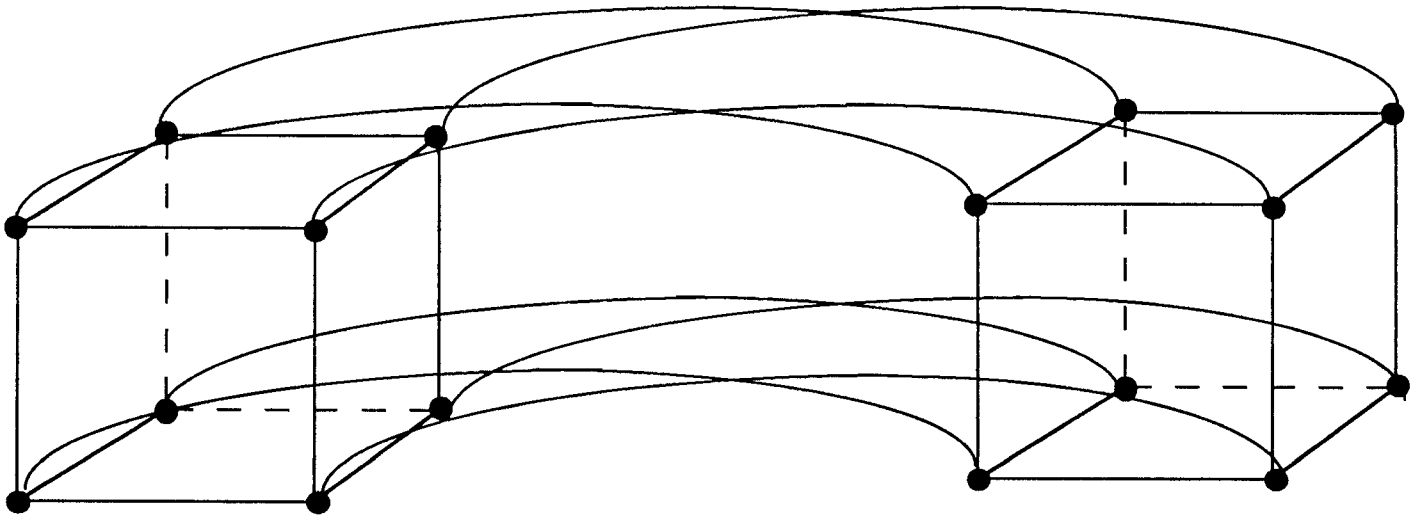


Fig - 2.5 A 4-cube formed from 3-cubes

links between the PEs. Each processor has its local memory, which makes every processor an independent unit. In the SIMD mode, this memory only contains data whereas in MIMD this memory contains instructions as well. Hypercube processors are homogeneous because all the nodes can be equally treated; any hypercube can be mapped onto itself by mapping on a node to any other . When a node *i* is mapped onto another node *j*, the addresses of all nodes are changed and the new address of a node is found by taking the XOR between its previous address and the address of node *i*.

The communication time between two PEs of hypercube depends on the number of links between them. The maximum communication time between any two PEs in the d-dimensional hypercube is O(d) because the maximum number of intermediate links is d. The total number of 1s in the XOR between the binary addresses of two PEs gives the minimum number of communication links between these PEs. If PE Y is connected with X in its *i*th dimension, then the addresses of X and Y will differ only in the *i*th bit position. Let the binary address of a node X be $x_{d-1}...x_1x_0$, then the routing functions for the *d* adjacent PEs are given by

$$C_i(x_{d-1}...x_1x_0) = x_{d-1}...x_{i+1}\bar{x}_i\, x_{i-1}...x_0.$$

where $0 \leq i \leq d-1$

The hypercube can be partitioned into smaller dimensional cubes and a d-dimensional hypercube can be constructed recursively from lower dimensional cubes; for example if two (d-1) dimensional hypercubes are combined, they produce a d-dimensional hypercube. Consider two identical (d-1)-dimensional hypercubes with labels from 0 to 2^{d-1}-1; by joining vertices with the same address, a d-dimensional hypercube is obtained. Fig 2.5 shows how two 3-cubes are combined to produce a 4-cube.

To summarize:

1. Any d-cube can be teared in d possible ways into two (d-1)-subcubes.

2. There are d! x 2^d ways of numbering the 2^d nodes of the d-cube.

3. The maximum distance between any two nodes in the d-cube is equal to d, which is also called the diameter of the hypercube.

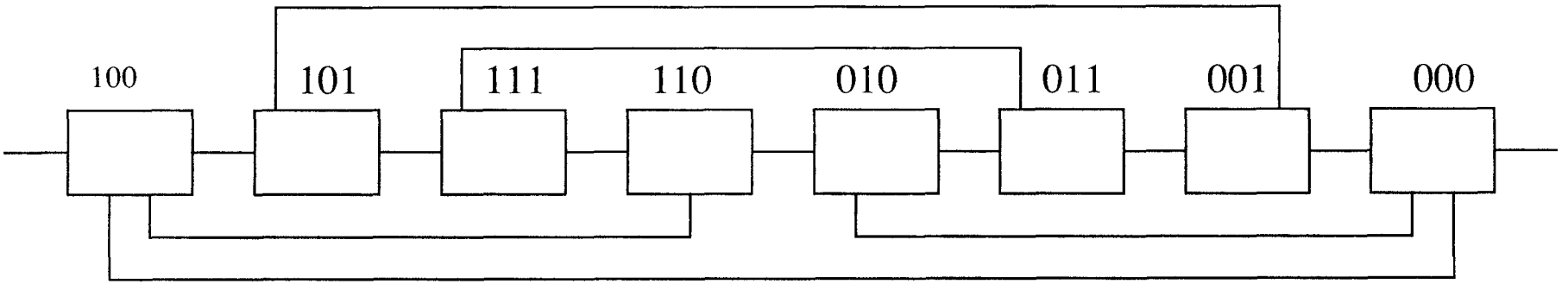
4. Any two processors in the d-cube can communicate with each other. In order to communicate, data has to travel atleast a distance which is equal to the number of 1s in the XOR between the addresses of these PEs (this is known as the Hamming distance H(X,Y) between PEs X and Y).

Various topologies can efficiently be mapped into the hypercube. There are basically two reasons for the importance of such a mapping.

i) Some algorithms may be developed for some other topology for which they fit perfectly Then one might wish to implement the same algorithm on the hypercube with little additional programming effort. If the original architecture can efficiently be mapped into the hypercube then this will be achieved easily.

ii) A given problem may have a well defined structure, which requires a particular pattern of communication. Mapping the pattern into the hypercube may result in short communication time. Our addition algorithm is a good example of this, because in our algorithm each of 2^d processors communicate with d other processors which is efficiently done only in the hypercube.

Some important mappings which are relevant to this work are discussed in the following sections.



Linear array mapped onto 3 - Cube, using 3 - bit binary gray codes.

Figure - 2.6

2.3.1 Mapping Rings and Linear Arrays into the Hypercube

Consider a ring structure containing 2^d processors. Also consider a target d-dimensional hypercube. The ring can be mapped into the hypercube in such a way that the proximity property is preserved. (i.e., any two adjacent vertices of the ring map on two neighboring nodes of the hypercube). Another way of visualizing this problem is that we are seeking the string of length $N=2^d$ that crosses each node of the hypercube once and only once In graph theory, this is called a Hamiltonian circuit in the hypercube.

According to the definition of the hypercube network, any two adjacent nodes have binary addresses that differ only by one bit. This means that a Hamiltonian circuit should be represented by a sequence of d-bit binary numbers such that any two successive numbers have only one different bit A binary sequence with such a property is the reflected Gray code [9].

There exit various ways to generate these Gray codes; the best way to generate these codes is described as follows One starts with the sequence of the two 1-bit numbers 0 and 1, this is called one bit Gray code. To get the two-bit Gray codes, take the same sequence and concatenate a zero in the highest bit position of each number, then take the mirror image (reverse order) of that sequence and insert a one in the highest bit position of each number. The sequence of 2-bit reflected Gray codes is

$$G_2 = \{00, 01, 11, 10\}$$

The reflected Gray codes with three or more bits can be generated in the same manner, for example the sequence of 3-bit codes is

$$G_3 = \{000, 001, 011, 010, 110, 111, 101, 100\}.$$

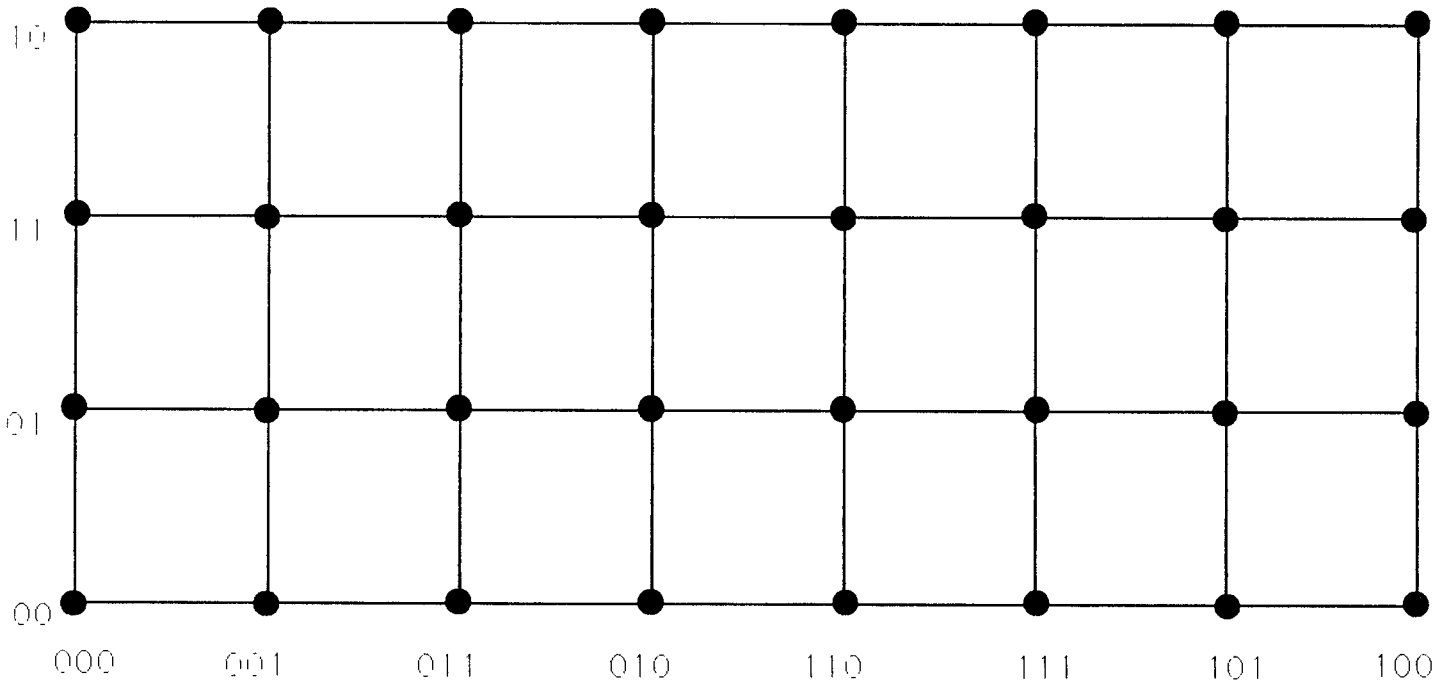


Fig - 2.7. Two-dimentional Gray code for an 8 x 4 grid

In general, if *image*[G_{d-1}] is the reflection or reverse order of sequence G_{d-1}, then the d-bit Gray code is given by

$$G_d = \{0G_{d-1}, 1image[G_{d-1}]\}.$$

The mapping of an 8-node ring into the 3-dimensional hypercube is shown in Fig 2.6. This figure shows the linear array with the extra connections which are present in hypercube.

2.3.2 Mapping Grids (Mesh) into the Hypercube

One of the most important reasons that the hypercube is popular is that meshes can easily be mapped into the hypercube. Consider an n-dimensional mesh that has size m_i in each dimension which is a power of 2 (i.e., $m_i = 2^{p_i}$).

Now consider the *d*-dimensional hypercube on which this mesh is to be mapped. Let $d = p_1+p_2+...+p_n$, where 2^d is the total number of processors in the n-dimensional grid, which is also the total number of nodes in the hypercube.

In order to map, the mesh into the hypercube, neighboring points in the mesh must be assigned to neighboring nodes in the hypercube. In the previous section, the mapping of the one dimensional mesh (i.e., the linear array or ring) was discussed. The mapping of higher dimensional meshes is done as follows. The nodes in each dimension are numbered sequentially using the respective reflected Gray codes. A node of the mesh is mapped onto that node in the hypercube whose address is obtained by concatenating the numbers of the particular node for all the dimensions. For example, Fig 2.7 shows a two-dimensional 8x4 mesh, as $p_1 = 2$ and $p_2 = 3$, and the appropriate Gray codes.

CHAPTER 3

ADDITION ALGORITHMS

This chapter is devoted to addition algorithms. In section 3.1, some of the existing addition algorithms are discussed, they are valid for both positive and negative numbers and some of them can be implemented on parallel processor systems. In section 3.2, a new algorithm is presented which is based on the conditional sum logic and can be implemented on an SIMD hypercube system. At the end of this chapter, a comparison of the new algorithm with the existing algorithms is presented.

3.1 EXISTING ADDITION ALGORITHMS

The operands of addition are called addend and augend, and similarly the operands of subtraction are subtrahend and minuend. To subtract one operand (subtrahend) from other (minuend), the 2's complement of one operand is added to the other.

Let A and B represent the addend and augend respectively in binary number form (vector), as follows:

$A = a_{n-1}\ a_{n-2}\ a_{n-3} \ldots a_1\ a_0$,

$B = b_{n-1}\ b_{n-2}\ b_{n-3} \ldots b_1\ b_0$.

Since A and B are signed binary numbers, a_{n-1} and b_{n-1} represent the sign of A and B respectively.

The addition with carry can be speeded up by either high speed carry propagation circuitry or using the algorithm in which the carry is generated prior to the addition operation, as in the carry look-ahead addition. The conditional sum addition logic can be used to reduce the carry propagation delay.

In the standard ripple carry addition technique, the sum of any bit position depends on the carry out of previous bit position. More specifically,

$S_i = A_i \oplus B_i \oplus C_{i-1}$,

where

$C_{i-1} = A_{i-1} B_{i-1} + C_{i-2}(A_{i-1} + B_{i-1})$

as discussed in the introduction.

The above equations show that the addition using the ripple carry technique can not be implemented in parallel. However, the hardware used in this sequential addition is minimum. In order to speed up the addition operation different algorithms are discussed, which can be implemented on parallel processor computers.

3.1.1 Conditional Sum Addition Logic

This high-speed algorithm was first presented by J. Sklansky in the mid 1960's [6]. In the conditional sum logic, a carry and a sum bit are generated at every bit position with the assumption that the carry into this position is 0; in addition, another carry and another sum bit are generated under the assumption that the carry-

i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ASSUMED INITIAL CARRY	TIME INTERVAL
A_i	1	0	1	1	1	0	1	1	0	1	1	0	1	1	0	1		
B_i	0	0	0	1	1	0	0	1	1	0	1	1	0	1	1	0		
S	1	0	1	0	0	0	1	0	1	1	0	1	1	0	1	1		
C	0	0	0	1	1	0	0	1	0	0	1	0	0	1	0	0	0	T_0
S	0	1	0	1	1	1	0	1	0	0	1	0	0	1	0			
C	1	0	1	1	0	0	1	1	1	1	1	1	1	1	1		1	
S	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	1		
C	0		1		1		1		0		1		1		0		0	T_1
S	1	1	0	1	0	1	0	1	0	0	1	0	0	1				
C	0		1		1		1		1		1		1				1	
S	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1		
C	0			1			1			1							0	T_2
S	1	1	0	1	0	1	0	0	1	0								
C	0			1			1										1	
S	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	1		
C	0						1										0	T_3
S	1	1	0	1	0	1	0	1										
C	0																1	
S	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1		
C'_{i+1}	0																0	T_4

Fig. 3.1. Example of conditional-sum addition [6]

in is 1. After this operation, consecutive pairs of conditional sum and carries are combined according to whether there is a carry into that pair of bits or not. This process continues until the final sum is obtained. The example given in Fig 3.1 shows the application of this logic for two operands A and B.

$S_i(0)$ and $C_i(0)$ are the sum and carry-out respectively for the *ith* bit position, considering a carry-in of 0. Similarly, $S_i(1)$ and $C_i(1)$ are the sum and carry-out, assuming 1 as the carry-in. T_0, T_1, T_2, T_3 and T_4 are successive time intervals during which conditional sum and carries are generated. During the first interval, the sum and carry-out for each column are calculated assuming a carry-in of zero into that column (i.e., $S_i(0)$ and $C_i(0)$). In the same time interval, $S_i(1)$ and $C_i(1)$ are also calculated. So, in the example of Fig. 3.1, first two rows contain the sum and carry-out bits generated assuming a carry-in of 0, while rows 3 and 4 contain the sum and carry-out under the assumption that carry-in is 1. The Boolean relations for the sum and carry-out bits are given as:

For $C_{i-1} = 0$

$$S_i(0) = A_i \oplus B_i$$

$$C_i(0) = A_i B_i,$$

for $C_{i-1} = 1$

$$S_i(1) = \overline{S_i(0)}$$

$$C_i(1) = A_i + B_i$$

During the time interval T_1, the conditional sums and carries are found for pairs of consecutive bits, (i.e., i=0 and 1, i=2 and 3, i=4 and 5 and onward) considering a carry in of 1 and 0. In the time interval T_2 , the conditional sums of operand bits i=0,1 combine with the conditional sums of i=2,3 under the assumption of carry in of 0 and 1, depending upon the carry out from the lower bit pair. Similarly, the conditional sums of i = 4,5 combine with those of i = 6,7, and onward . In the last

step, during the time interval T4, the conditional sums of the bits i=0,1,2,3,4,5,6,7 combine with the conditional sums of the bits i=8,9,10,11,12,13,14,15 This step gives the final sum A+B with the final carry-out.

3.1.2 Independent-Dependent Carry Addition

Extra speed can be achieved in this type of addition [7] in two ways:

1. Simultaneous generation of carries at all columns, where the carries are independent of previous carries (independent carries).

2. Use of extra fast carry transmission gates, in order to copy the carries which do not change (dependent carries).

The following example shows the basic procedure of this methodology.

Let

A_i = 10011010001

B_i = 11101001100

step 1. Independent carries 1 10 0 0

step 2. Dependent carries 111 000 0

let C_{ind} and C_d represent independent and dependent carries respectively, and C_i represents the final carry-in in bit position i. Then, in the above example:

C_{ind} = 100010000000

C_d = 11100000000

$C_i = C_{ind} + C_d$

C_i =111110000000

and sum at bit position i is

$$S_i = A_i \oplus B_i \oplus C_i.$$

step 3. S_i = 110000011101

In step 1, the existence of independent carries for any particular column is determined by detecting the equality of the pair of operand bits of the preceding column; each of these independent carries C_{ind} is then equal to the corresponding operand bits. In step 2, dependent carries C_d are determined, by copying the carries between any two independent carries and these C_d are equal to the right hand side's independent carry C_{ind} . Final carries C_i at each bit position can be determined by taking the OR between the bits of C_{ind} and C_d. Before taking the OR, the space between C_{ind} and C_d is filled by 0's.

The final sum bit is obtained in step 3 by taking the exclusive OR between the carry and operand bits of each column independently

Gilchrist, Pomerene and Wong [7] described a circuit realization of the basic IDA. This algorithm can be very fast if the copying of dependent carries is done quickly.

3.1.4 Ziavras-Davis Algorithm

In this algorithm, very high speed addition is achieved by using more than one processor in a hypercube system working simultaneously. A carry-look-ahead technique is applied. The operands of addition are distributed throughout the processing resources using low-order interleaving of bits. Since the addition of two n-bit operands gives $(n+1)$-bit result , and $n+1$ may not be evenly divisible by

the total number of processors, sign extension of the operands is performed. Therefore after sign extension each operand has $l=N\lceil(n+1)/N\rceil$ bits, where N is the total number of processors.

The $(2^n, 2^n)$ grid is first mapped onto 2n-dimensional hypercube as discussed earlier. Any point in that grid can be represented by Cartesian coordinates (x,y). The bits in the addend and augend are distributed among the processors using low-order interleaving. Consider q to be the number of operand bits per processor. Low-order interleaving means that the least significant bit (LSB) of the operand will go to the processor whose coordinates are (0,0), the next higher bit goes to processor (1,0), if operands are distributed horizontally, otherwise it will go to (0,1): here we have chosen horizontal distribution. This process continues until the last processor of that row (i.e., $(2^n - 1,0)$) receives the $(2^n - 1)$th bit. Then, the same process continues for the other rows consecutively. So, in the first cycle of distribution, the first 4^n bits are distributed among the 4^n processors. This process repeats itself q times, since q bits of each operand are assigned to each processor in the system.

To summarize, the processor (x,y) finally contains the q bits whose subscript s satisfies the condition, $s \bmod 4^n = 2^n y + x$

After this distribution of operands, the computation step starts. Since this algorithm is based on the carry-look-ahead technique [2], the carries at each bit position are calculated prior to addition. This computation step contains bidirectional communication and some calculation within the processors. Due to the powerful interconnection structure of the hypercube network, the communication operations are implemented efficiently.

More specifically, the following set of steps are carried out.

1) The initial carry-in at all bit positions is set to zero, except in the case where subtraction is performed in 2's complement form where the carry-in at the LSB position is set to 1

2) Each processor will calculate q bits of the results, so the individual propagate term for each bit position is also set to 1 by all the processors (i.e., $P_{i,j}(x,y)=1$).

3) Now every processor generates the group carry $Gc_{ij}(x,y) = GG_{i,i} = a_i b_i$ and group propagate terms $GP_{ij}(x,y) = GP_{i,i} = a_i + b_i$, at every bit position

4) After that, communication between the processors starts, assuming bidirectional communication channels. In this algorithm, each processor communicates with all its neighboring processors using all dimensions, one at a time. Basically, carry propagate and carry generate terms are found for groups of bits, where the group size increases linearly as $1,2,4,8,16,....2^{2n}$ [10]

5) After the completion of these 2n cycles of communications, the processors perform internal operations in order to calculate the carry-ins of individual bit positions. Finally, each processor finds the individual bits of the sum as,

$$s_i = a_i \oplus b_i \oplus c_i.$$

where s_i = generated sum at bit position i,

c_i = calculated carry-in at bit position i.

3.1.5 Simultaneous Addition of Multiple Operands (Carry Save Adders)

During the multiplication process, multiple operands are required to be added. There are different ways to add multiple operands, one may use an iterative technique applied to pairs of operands. It is also possible to add a column of multiple bits, then propagate the carry generated from this column to the next column, and add this carry with the bits of that column. All these processes are

sequential and take a large amount of time. In order to save time, an algorithm may be developed where the carry and sum of each column are generated independently at every bit position and multiple operands are reduced to two (i.e., carry and sum); these two operands are then added by using any suitable addition algorithm. The Wallace tree reduction technique is based on this technique. Let X, Y and Z be three operands to be added, and X_i, Y_i and Z_i be the *i*th bit of these operands respectively. Then, the carry and sum bits for this bit position are given by

$$C_i = X_i \oplus Y_i \oplus Z_i$$
$$S_i = X_i Y_i + Z_i(A_i + B_i)$$

C_i and S_i are the ith bit of two operands C and S respectively, to be added. Before adding them C must be shifted left by one position. This algorithm for addition of multiple operands is called Carry Save Addition. CSA is considered as a three-to-two convertor, and it is used here to add multiple operands. An example of this algorithm is given below

X = 1 1 1 1 0 1
Y = 0 1 0 1 1 0
Z = 1 1 0 1 1 1
C = 1 1 0 1 1 1
S = 0 1 1 1 0 0
C = 1 1 0 1 1 1 (left shift)
C+S = 1 0 0 0 1 0 1 0

3.2 A New Addition Algorithm

This section describes an algorithm for fast addition of operands with a large number of bits, making use of the powerful interconnection features of the hypercube as described in the previous chapter. This algorithm is based on the conditional sum logic as described in the previous section and introduced by J. Slansky in 1960.

In this algorithm, each processor is required to communicate in all of its dimensions. If the d-dimensional hypercube is considered, each processor will communicate with d different processors. Bidirectional communication channels are considered which means that any processor can receive and send data on the same channel at the same time.

In order to visualize the hypercube properly for this algorithm, a linear array of 2^d processors is first mapped onto the d-dimensional hypercube as discussed in the previous chapter. The processors 0 to 2^d-1 in the linear array are labeled using the d-bit reflected Gray code. Processors whose binary addresses differ by only one bit are then directly connected with each other. Similarly, the number of different bits in the binary address, which is the Hamming distance between the two addresses, gives the minimum physical distance between the processors. For example, two processors whose addresses differ in two bits are two communication links away. The mapping of a linear array consisting of eight processors onto the 3-dimensional hypercube is shown in Fig 2.6. Each processor in the linear array of 2^d processors can be addressed in two ways, one is d-bit binary linear address i, other is d-bit reflected binary Gray code address G[i] (i.e., $i = y_{d-1}.. y_0$ and G[i] = x_{d-1} ..x_0).

A method to build the sequence of reflected binary Gray codes was discussed in the previous chapter, but here we will discuss the same topic in a slightly different way. Let I be the sequence of binary codes, such that $0 \leq i \leq 2^d$-1; each number of

this sequence corresponds to the address of a particular processor in the array or hypercube. Let G[i] be the reflected Gray code of the binary number *i*.

The relation between the binary number i and its Gray code is given by

$$G[i] = \lfloor i/2 \rfloor \oplus i.$$

The first term of the above equation represents the shifting of the binary number toward right by one bit.

The next step is the data distribution among the processors. Let n be the number of bits in the signed operands to be added. The addition of these operands will generate an $n+1$ bits sum, therefore the operands are sign extended to $n+1$ bits. Now it is required to distribute the $n+1$ bits among the processing resources evenly, but it is possible that $n+1$ is not evenly divided by 2^d (i.e., $(n+1) \bmod 2^d \neq 0$). In order to overcome this problem, further sign-extension of the operands is performed. This does not change the actual value of the operands but makes their total number of bits to be a perfect multiple of 2^d (i.e., if l is the number of bits in the extended operands, then $l \bmod 2^d = 0$).

Let the two sign-extended operands be

$$A = a_{l-1}a_{l-2}..\ a_{n+1}a_n a_{n-1}..\ a_1 a_0$$

$$B = b_{l-1}b_{l-2}..\ b_{n+1}b_n b_{n-1}...b_1 b_0$$

The total number of bits in each sign extended operands is equal to

$$l = 2^d \lceil (n+1)/2^d \rceil$$

and the total number of bits stored in each processor is equal to

$$q = \lceil (n+1)/2^d \rceil.$$

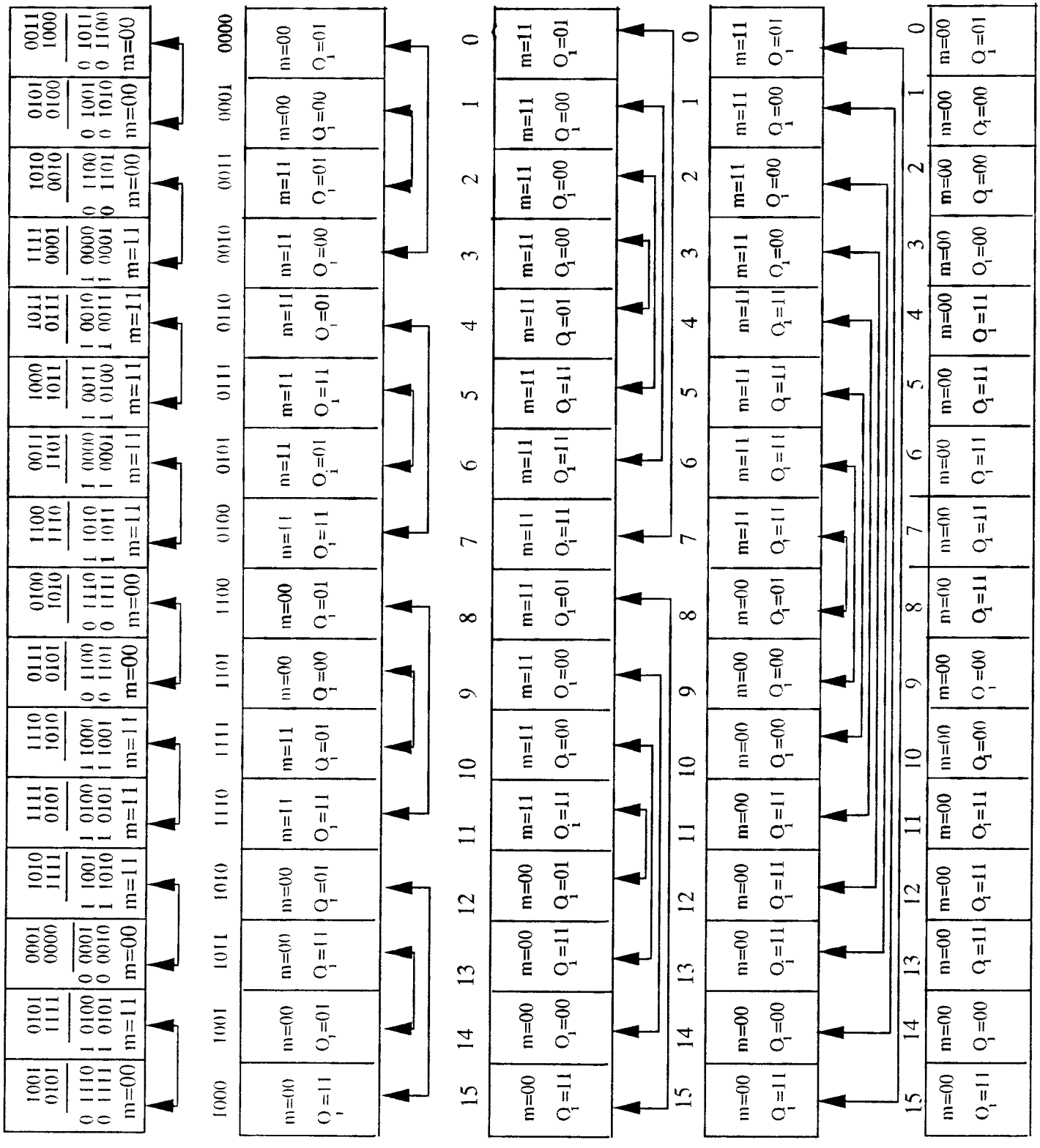
The sign-extended operands may be distributed among the processors using high-order or low-order interleaving with respect to the bits' subscripts. Low-order interleaving for this algorithm is not suitable because it requires a large number of communication cycles. So, the high-order interleaving technique is chosen for data

distribution. In high-order interleaving, the first q successive bits of each operand are stored in the processor with address G[0], the next set of q successive bits are placed in the processor with address G[1], and so on, until the last processor G[2^d-1] gets the q most significant bits (i.e., from the bit position l-(q+1) to l-1). The processor with address G[i] contains all the bits with subscripts s such that $q(i) \leq s \leq q(i + 1) -1$, or $s = (q \times i - 1) + j$, where $0 \leq i < 2^d$ and $0 < j \leq q$. Both the addend and the augend are distributed according to the method described above. After this distribution of operands, the second phase starts which implements the internal computations within the processors and communication between processors. Assuming that the hypercube processors operate in the SIMD mode, the following sequence of steps are applied.

1) Various local variables are initialized. The variable m_i of each processor G[i], is updated after every communication cycle and represents the concatenation of two conditional carries. $C_i(0)$ and $C_i(1)$ are the conditional carry-outs for the ith group of q bits in processor G[i], assuming a carry-in of 0 and 1 respectively. We will be referring to them as the ith group carries. In addition, the variable O_i is initially set to 01 and its value is updated after every communication cycle. The processor uses the value of O_i to make a decision in the last step.

2) Each processor adds its groups of q bits twice, considering a carry-in of one and zero respectively. Although processor 0, which contains the group of least significant bits, is not required to calculate both sums (i.e., $S_0(0)$ and $S_0(1)$), they are calculated for the sake of uniformity. However, this may become necessary when subtraction in 2's complement form is performed. Let A_i and B_i be the

Fig - 3.2. Computation and communication steps for addition of 64 bit operands using the new addition algorithm.



The action in the last Step depends upon the value of O_1

1111	0100	0010	1010	0101	1000	1100	1111	1011	0001	0100	0011	0000	1100	1001	1011
1111	0100	0010	1010	0101	1000	1100	1111	1011	0001	0100	0011	0000	1100	1001	1100

The addition of 64-bit operands

1001	0101	0001	1010	1111	1110	0111	0100	1100	0011	1000	1011	1111	1010	0101	0011
0101	1111	0000	1111	0101	1010	0101	1010	1110	1101	1011	0111	0001	0010	0100	1000
1111	0100	0010	1010	0101	1000	1100	1111	1011	0001	0100	0011	0000	1100	1001	1011

Fig 3.2 Computation and communication Steps for addtion of 64-bit operands on the 4-dimensional hypercube.

groups of the operand bits to be added, in processor G[i]. $S_i(0)$ and $S_i(1)$ are the conditional sums assuming carry-in of zero and one respectively. The following set of equations represent the above operations

$S_i(0) = A_i + B_i$.

$C_i(0) = \lfloor (A_i+B_i) \cdot 2^q \rfloor \bmod 2^q$,

or $C_i(0) = \lfloor (A_i+B_i)/2^q \rfloor$ (shifting right by q bits)

Similarly considering a carry-in equal to 1:

$S_i(1) = A_i+B_i+1$,

$C_i(1) = [(A_i+B_i+1).2^q] \bmod 2^q$ or $\lfloor (A_i+B_i+1)/2^q \rfloor$.

Now the variable m_i is found by concatenating $C_i(0)$ and $C_i(1)$ and contains two bits: $m_i = C_i(0)C_i(1)$.

3) In this step, processors start communication Each processor of the hypercube is required to communicate with all of its neighbors, starting from the 0th dimension Communication in the 0th dimension means that each processor communicates with the processor whose address differs in the LSB position from this processor's address. Such processors exchange their m_i variable (assuming bidirectional communication channels). Then, every processor updates the values of m_i and O_i The communication and computation steps for the 4-dimensional hypercube are shown in Fig 3.2.

Let $P_j = x_{d-1}x_{d-2}...x_j...x_1x_0$,

and $P_j' = x_{d-1}x_{d-2}...\bar{x}_j...x_1x_0$.

be the labels of processors which are directly connected in the jth dimension. All possible such pairs of processors exchange their two bit variable m_i in the first cycle of communication. Then all processors update the values of m_i and O_i, depending upon the values of m_i, m_i' (i e., incoming value) and O_i. As these are

two bit variables, one of four actions may be possible. The processor i with y_0=1 performs a little bit different operation than the processor having y_0=0.

All processors with y_0=1 perform the following set of operations.

i) Processor G[i] checks the variable m_i, which has three possible values (i.e., 00,01 and 11) If m_i=00 or 11, the processor does not change the value of m_i and goes to step (ii) For m_i=01, it checks the value of the incoming variable m_i' and updates the value of m_i depending on the value of m_i'.

If m_i'=00 then $m_i = m_i \oplus 01$.

If m_i'=01 then $m_i = m_i$. (no action),

If m_i'=11 then $m_i = m_i \oplus 10$.

ii) In this step, the value of the variable O_i is also updated by all the processors, depending upon the previous value of O_i and m_i'.

If O_i = 00 or 11 then O_i does not change.

If O_i = 01 then $O_i = m_i'$.

The value O_i = 10 is not possible.

The processors with y_0 = 0 will perform the following set of operations in order to update the value of m_i, because it is required that both P_0 and P_0' have the same value of m_i before starting a new cycle of communication. This can be done in two ways P_0' can transfer its updated m_i to P_0 , but in this way P_0 remains idle during the transfer. Another way, which is adopted here, is for P_0 to calculate the value of m_i by itself. As bidirectional communication channels are considered and the variable m_i is exchanged before all calculations, P_0 has the value of m_i' (i.e., the incoming value of m_i from P_0'), therefore it can update the value of its m_i by itself as follows.

If m_i' = 00 or 11 then $m_i = m_i'$.

otherwise, check the present value of m_i

If $m_i = 00$ then $m_i = m_i' \oplus 01$.

If $m_i = 01$ then $m_i = m_i'$.

If $m_i = 11$ then $m_i = m_i' \oplus 10$.

as it is required that after the first cycle of communication both P_0 and P_0' have the same value of m_i, because they will work as a group in the next cycle.

iii) Now all the processors communicate with their neighbors in their 1st dimension Therefore, all the processors whose addresses differ in the 1st bit position exchange their data and perform the operations described in (i) and (ii). This communication and computation sequence continues until all the processors communicate in their *(d-1)*th dimension and perform the operations described in (i) and (ii). This operation is completed in $d=\log_2 N$ cycles.

iv) The fourth and final step of addition, places the data in the right place in every processor, depending on the values of O_i in the processors In every cycle of communication, data from two processors are compared and the value of m_i from the processor that contains the least significant group selects the correct sum in the processor containing the group of higher significance, according to the actual carry-in in that group This action is recorded in the variable O_i and finally implemented by all processors simultaneously. After executing this step, $S_i(0)$ of all processors contains the final sum, which is generated due to a carry-in of zero in the LSB position of the operand while $S_i(1)$ of all processors contains the sum assuming a carry-in of 1 in the LSB position. The following set of operations operations are performed in all the processors simultaneously

If $O_i = 00$ then $S_i(1) = S_i(0)$.

If $O_i = 01$ then no action.

If $O_i = 11$ then $S_i(0) = S_i(1)$.

Communications between processors for this algorithm are performed in one hop. For the d-dimensional hypercube, the total number of communication cycles is *d*

with only two bits, exchanged in any cycle; this saves a lot of communication time when compared to other algorithms described for addition on hypercubes.

All steps of communication and computation are shown in Fig 3.2 for a 4-dimensional hypercube mapped into an array containing $2^4=16$ processors.

This method is suitable for a large numbers of operand bits per processor. It does not give good performance for small numbers of bits/processor.

3.3 Comparison With Existing Algorithms

The conventional way of addition is the ripple carry addition which takes at most *(n-1)* carry delays and one sum delay for the addition of n-bit operands. This method is completely sequential, because addition in every bit position depends upon the carry from the previous bit position; in this way, the task of addition can not be broken into independent sub-tasks. A speed up of the addition operation can be achieved either by using advance hardware technology or by any suitable algorithm that minimizes the carry delay. The fastest way of addition is implemented by carry free addition in which a carry is not generated and addition between two operands at any bit position is performed independently of the values in other bit positions. However, the carry free operation is not possible in the ordinary binary system; it is used only with numbers represented in a redundant binary system. The major drawback of these numbers is that their conversion to the binary system or vise versa is a sequential operation and takes a large amount of time; therefore, the entire process of addition is very time consuming. Dependent-Independent carry addition, as described in a preceding section is also a good algorithm for fast addition, but the generation of dependent carries takes a lot of

THE FORMULAS FOR G AND τ FOR THE FIVE ADDERS IN TERMS OF THE SUMMAND LENGTH, n

Type of Adder	Number of Gates G	Number of Gate Delays τ
Simple Series Adder (SSA)	7	$4n$
Simple Iterated Adder (SIA)	$7n$	$2(n+1)$
Independent-Dependent Carry Adder (IDA)	$17n - 1$	$4 + n$ (lower bound)
Distant-Carry Adder (DCA_p)	$6n + 1 + \frac{p+1}{p-1}q + \frac{p^2+2p-1}{p}\left[k\left(n + \frac{1}{p-1}\right) - \frac{pq}{(p-1)^2}\right]$, where $k \triangleq \log_p[1 + n(p-1)] - 1$, $q \triangleq 1 + (n-1)p - n$	$4 + k(p+1)$, where $k \triangleq \log_p[1 + n(p-1)] - 1$
$p \triangleq$ carry span		
Conditional-Sum Adder (CSA)	$3n[2 + \log_2(n+1)]$	$2 + 2\log_2(n+1)$

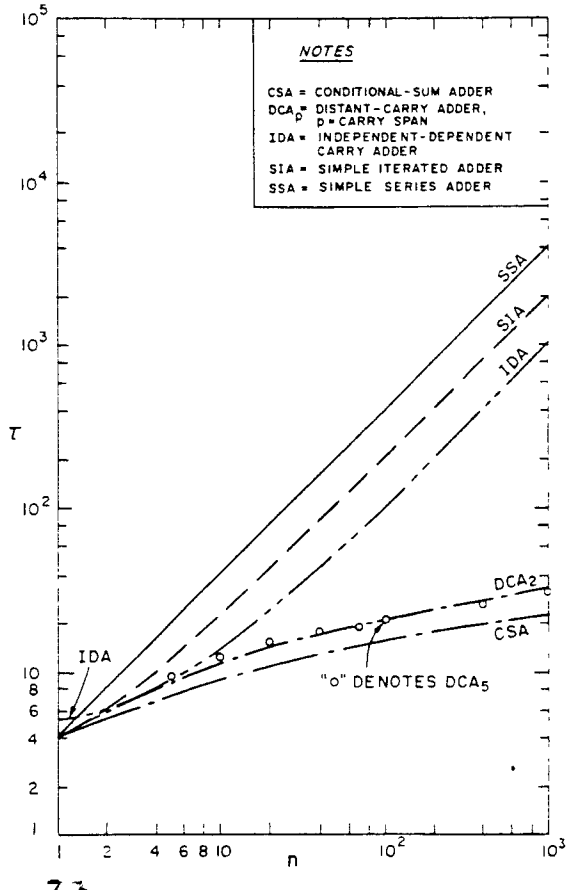


Fig 3.3—Gate-normalized addition time, τ, plotted vs n

time. Different types of fast adders were discussed in [1], [2], and [7] A comparative analysis of various addition algorithms for different number of bits in operands is shown by the graphs in Fig. 3.3 [7]. This analysis shows that the conditional sum logic is the fastest way of addition among these algorithms. Winograd has proved it by showing that the lowest bound on addition is achieved using this algorithm. Carry-look-ahead addition was first described by Weinberger and Smith and has been implemented in standard ICs. These adders are frequently used in most of the computers. These adders give compromising speeds for the operands when dealing with a small number of bits. However, if the operands become very large, then these adders do not achieve high performance and their hardware cost is also increased. Speed up can be achieved for operands containing a large number of bits by using parallel computers. Several algorithms for addition have been developed for parallel structures. Ziavras and Davis have developed an algorithm for fast addition on the hypercube which is based on the carry-look-ahead technique. This algorithm has the best possible computation complexity. However, it may involve relatively large amounts of communication in each cycle (i.e., q group carries and group propagates are transfered by each processor in every cycle, where q is the number of bits from each operand assigned to a single processor).

Our algorithm was described in the previous section. An analysis of this algorithm is carried out below. In this section, we will calculate the time required to add different sized operands using hypercubes of various sizes.

In this calculation, bidirectional communication channels are considered and all the data transfers are performed in one hop, as discussed in the previous section. This calculation can be divided into the following steps.

Table 3-1. Time using the conditional sum addition logic for operands and hypercube of various sizes.

Dimensions	Processors	Bits = 8		Bits = 20		Bits = 30		Bits =50		Bits = 100		Bits = 500		Bits = 1000		Bits = 5000		Bits =10000	
		q	T_8	q	T_{20}	q	T_{30}	q	T_{50}	q	T_{100}	q	T_{500}	q	T_{1000}	q	T_{5000}	q	T_{10000}
0	1	8	25	20	61	30	91	50	151	100	301	500	1501	1000	3001	5000	15001	10000	30001
1	2	4	19	20	37	15	52	25	82	50	157	250	757	500	1507	2500	7507	5000	15007
2	4	2	19	5	28	8	37	13	52	25	88	125	388	250	763	1250	3763	2500	7513
3	8	1	22	3	28	4	31	7	40	13	58	63	208	125	394	625	1894	1250	3769
4	16	1	28	2	31	2	31	4	37	7	46	32	121	63	214	313	964	625	1900
5	32	1	34	1	34	1	32	2	37	4	43	16	79	32	127	157	502	313	970
6	64	1	40	1	40	1	38	1	40	2	43	8	61	16	85	79	274	157	508
7	128	1	46	1	46	1	44	1	46	1	46	4	55	8	67	40	163	79	280
8	256	1	52	1	52	1	50	1	52	1	52	2	55	4	61	20	109	40	169
9	512	1	58	1	58	1	56	1	58	1	58	1	58	2	61	10	85	20	115
10	1024	1	64	1	64	1	62	1	64	1	64	1	64	1	64	5	76	10	91
11	2048	1	70	1	70	1	68	1	70	1	70	1	70	1	70	3	76	5	82
12	4096	1	76	1	76	1	74	1	76	1	76	1	76	1	76	2	79	3	82
13	8192	1	82	1	82	1	80	1	82	1	82	1	82	1	82	1	82	2	85
14	16384	1	88	1	88	1	86	1	88	1	88	1	88	1	88	1	88	1	88
15	32768	1	94	1	94	1	92	1	94	1	94	1	94	1	94	1	94	1	94
16	65536	1	100	1	100	1	98	1	100	1	100	1	100	1	100	1	100	1	100

1) The time required to calculate the conditional sums and carries $S_i(0)$, $S_i(1)$, $C_i(0)$, and $C_i(1)$, assuming one-bit processors, is given by

$T_1 = 2q$,

where

q = number of bits from each operand in each processor.

2) Two cycles are consumed in concatenating conditional carries to produce $mi = C_i(0)C_i(1)$, therefore

$T_2 = 2$

3) Two bits are exchanged in each cycle of communication. Assume that one cycle is consumed for transmitting a single bit, two cycles are required for set-up with one cycle on each side (i.e., source and destination). Also, let *d* be the dimension of hypercube, so the total number of cycles required to exchange the data between the processors is given by

$T_3 = 4d$

as four cycles are required in each stage of communication.

4) After each communication, some calculations are performed to update the values of m_i and O_i, except in the last stage in which updating of m_i is not required. Since O_i is updated *d* times and m_i is updated *d-1* times, therefore the total time required for this step is given by

$T_4 = d+d-1 = 2d-1$.

5) The last step of this algorithm is performed in parallel using all processors. In this step, the processor decides about the right place for $S_i(0)$ and $S_i(1)$, which depends on the value of O_i. The time taken by the processor in this step is

$T_5 = q$.

To conclude, the total time it takes to perform the complete addition is given by

$T = T_1+T_2+T_3+T_4+T_5$.

$= 2q + 2 + 4d + 2d-1 + q$

$= 3q + 6d + 1$,

where

$q = \lceil bits/proc \rceil$ = the number of bits from each operand in each processor, and

$d = \log_2 proc$.

Therefore,

$T = 3q + 6\log_2 proc + 1$.

The above equation is used to analyze the performance of this algorithm. Results are shown in Tables 3.1 and 3.2 and graphically in the Fig 3.4 for different sizes of operands.

In order to compare our new algorithm with some other algorithms, it is required to find out the time required by them under the same processing "environment".

In the conventional ripple carry addition, operands can not be broken into independent groups of bits, therefore this operation is completely sequential Assume that q bits of each operand are distributed among the processors using high-order interleaving and *proc* represents the total number of processors in the d-dimensional hypercube (i.e., proc = 2^d). The calculation of time required by this algorithm can be divided into the following steps.

1) Processors cannot work simultaneously. As q cycles are required to add q bits in any processor, the total time required for q x proc bits is

$T_1 = q \times proc$.

2) It is required to communicate a carry-out from a lower significant group to a higher significant group. A total of three cycles are required to communicate one bit of carry from one processor to another neighboring processor (i.e., two cycles

Table 3.2[5] Comparative analysis of various addition algorithms for the number of bits in operands = 8

Dimension of Hypercube	proc.	q (bits/proc)	Tseq cycles	Tcla cycles	Tcsa cycles
0	1	8	10	32	25
1	2	4	13	34	19
2	4	2	19	28	19
3	8	1	31	22	22
4	16	1	63	28	28
5	32	1	127	34	34
6	64	1	255	40	40
7	128	1	511	46	46
8	256	1	1023	52	52
9	512	1	2047	58	58
10	1024	1	4095	64	64
11	2048	1	8191	70	70
12	4096	1	16383	76	76
13	8192	1	32767	82	82
14	16384	1	65535	88	88
15	32768	1	131071	94	94
16	65536	1	262143	100	100

(b) the number of bits in operands = 500000

Dimension of Hypercube	proc.	q (bits/proc)	Tseq cycles	Tcla cycles	Tcsa cycles
0	1	500000	500002	2000000	1500001
1	2	250000	500005	2000002	750007
2	4	125000	500011	1500004	375013
3	8	62500	500023	1000006	187519
4	16	31250	500047	625008	93775
5	32	15625	500095	375010	46906
6	64	7813	500223	218776	23476
7	128	3907	500479	125038	11764
8	256	1954	500991	70360	5911
9	512	977	501759	39098	2986
10	1024	489	503807	21536	1528
11	2048	245	507903	11782	802
12	4096	123	516095	6420	442
13	8192	62	532479	3498	265
14	16384	31	557055	1888	178
15	32768	16	622591	1054	139
16	65536	8	720895	576	121

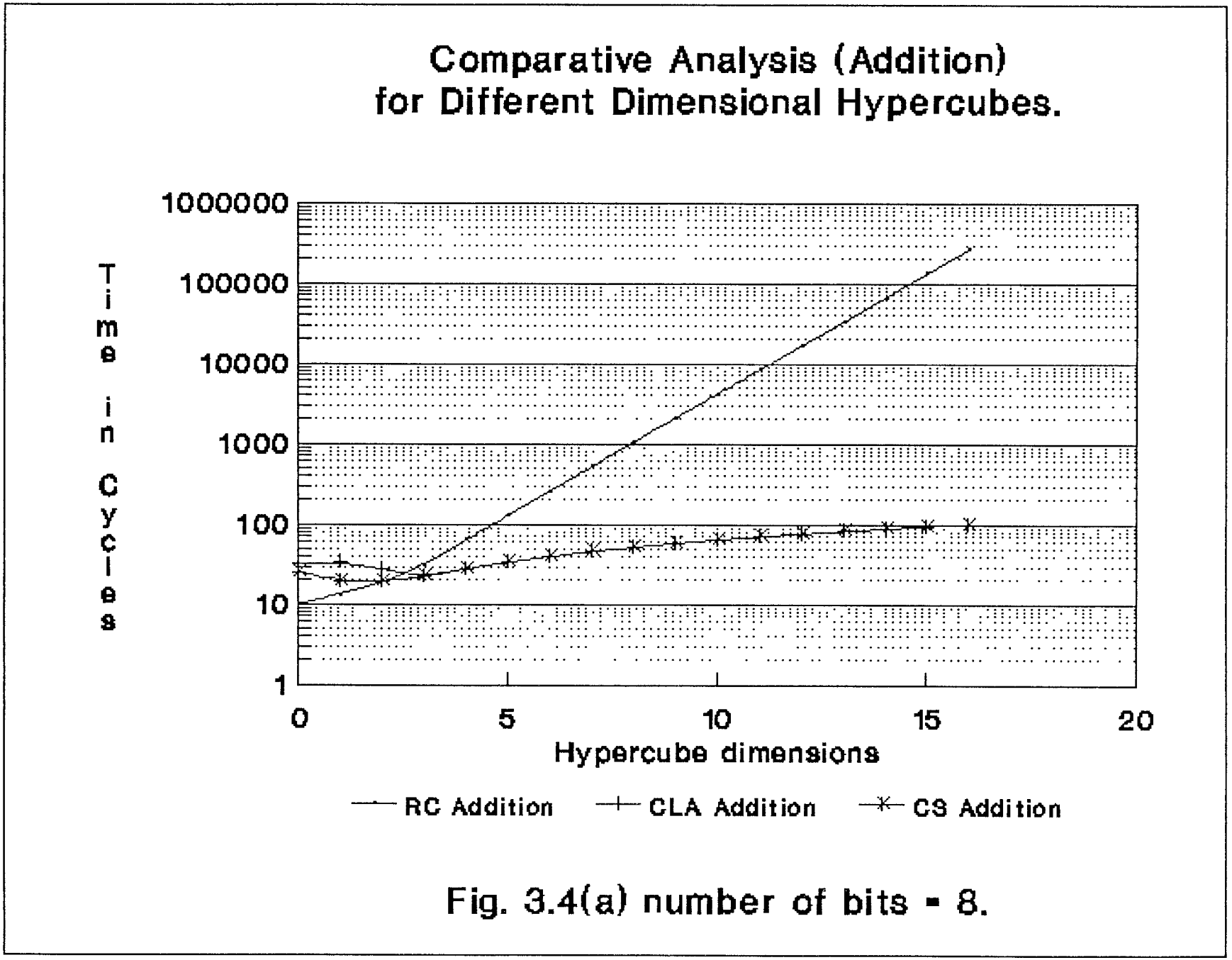


Fig. 3.4(a) number of bits = 8.

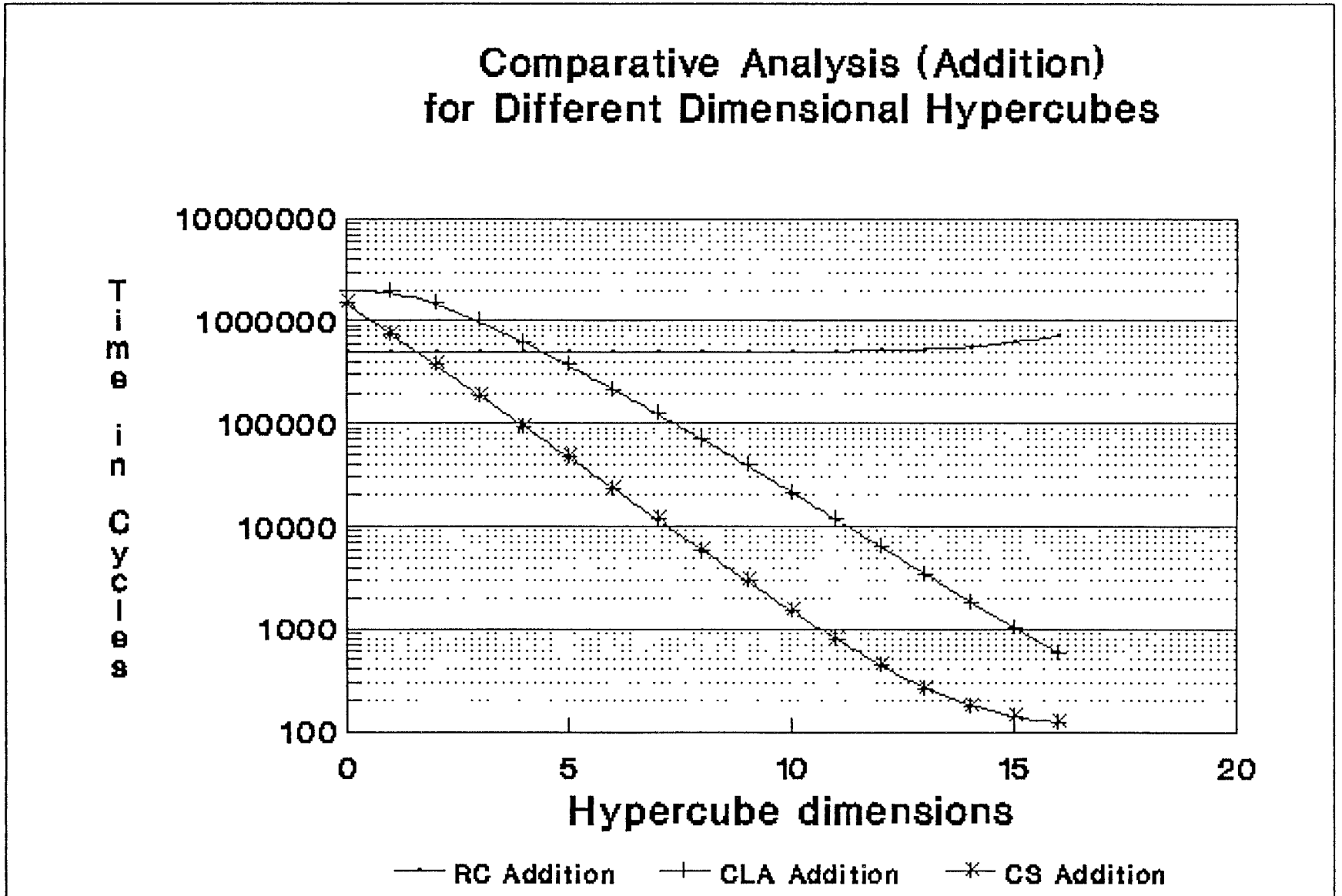


Fig. 3.4(b) number of bits = 500000.

for the setup time and one cycle for communicating that carry bit). The total time for the communication of carries is

$T_2 = 3(proc-1)$.

Therefore the total time to perform the addition is given by

$T_{seq} = proc \times (q + 3) - 3$

Another algorithm which is used in the comparative analysis is the one proposed by Ziavras and Davis [10], which is also implemented on hypercube structures. In this algorithm, all processors work simultaneously in the SIMD mode of computation. The total time consumed in this addition is found as follows.

1) A total of q cycles are consumed in order to generate the group carries at each bit position i (i.e., $a_i\ b_i$) and q more cycles in order to generate the group propagate terms (i.e , $a_i + b_i$). Therefore,

$T_1 = 2q$

2) In this algorithm, q bits of group carries and q more bits of group propagates, are exchanged between the processors in each communication step, so the total time taken by communication is

$T_2 = d(2q + 2)$.

3) The next step is to update the values of q group carry and q group propagate terms. Their values are updated after every cycle of communication, therefore the total time taken by this step is

$T_3 = 2dq$.

4) The last step contains computation operations within the processors, where the final carry c_i is generated at all bit positions and then the sum bit is obtained for all bits in the operands (i.e , $s_i = a_i \oplus b_i \oplus c_i$). The total time for this step is given by

$T_4 = 2q$.

Therefore the total time for addition using this algorithm is given by

$T_{CLA} = T1 + T_2 + T_3 + T_4$.

$= 4q + 4dq + 2d$

The comparative analysis of these addition algorithms involving simulation results using up to a 16-dimensional hypercube, for various sizes of operand, are shown in Table 3.2 and Fig3.4. These simulation results show that the performance of the new algorithm is not good for small numbers of bits. These results show that the ripple carry addition gives better result under these conditions.

Chapter 4

MULTIPLICATION ALGORITHMS

Multiplication is a very important arithmetic operation. Various existing add-shift methods are presented in Section 4.1 to speed-up multiplication In addition, a new algorithm is proposed in Section 4.2 which can be implemented on hypercube computers. A comparison of these algorithms is presented in the last section of this chapter.

4.1 Existing Multiplication Algorithms

Multiplication can be implemented as a sequence of repeated additions. The number to be added is called multiplicand, the number of times to be added is called the multiplier, and the result is called the product. The sum generated after each addition is known as the partial product. This technique of multiplication is extremely slow, therefore various other algorithms have been suggested to speed up this operation

If the two operands of the multiplication, namely the multiplicand and the multiplier, contain n and m bits respectively, including the sign bits, then the product will consist of $n+m$ bits, including the sign bit Let A, B and P be the

multiplicand, the multiplier and the product respectively If $m=n$ and the 2's complement representation is used for the signed numbers, then

$A = a_{n-1}a_{n-2}...a_1a_0$.

$B = b_{n-1}b_{n-2}...b_1b_0$

$P = p_{2n-1}p_{2n-2}...p_1p_0$

The speed of multiplication heavily depends on the way addition of partial products is performed, and the best way to add multiple operands is to use the Carry Save addition (CSA) discussed in the previous chapter.

4.1.1 Sequential Add-Shift Multiplication

The simple add-shift method that was briefly discussed earlier is the conventional way of multiplying two numbers A and B. It is described as follows. For A and B as above. the product P of A and B is obtained by the following set of additions. Partial products are shown with the sign extension.

$a_{n-1}.b_0, \ldots a_{n-1}.b_0, a_{n-1}.b_0, a_{n-2}.b_0, \ldots, a_1.b_0, a_0.b_0$

$a_{n-1}.b_1, \ldots a_{n-1}.b_1, a_{n-2}.b_1, \ldots, a_0.b_1$

$\ldots\ldots$

$\ldots\ldots a_0.b_{n-1}$

$p_{2n-1}, p_{2n-2} \ldots\ldots, p_1, p_0$.

There are n partial products generated in the above multiplication, where each one is shifted by one position to the left with respect to the previous one, and the MSB gives the sign of the partial product. These partial products are added using any suitable addition algorithm; a good choice is the CSA algorithm which can reduce multiple operands into two operands which are finally added using any fast algorithm

4.1.2 Booth's Algorithm

Multiplication can be speeded up by reducing the number of partial products. It is possible to achieve this goal by using some encoding techniques. A bit scanning technique is introduced by Booth's algorithm in order to reduce the number of partial products. It is based on the idea that a string of 0's in the multiplier can be skipped because they generate partial products of zero value, so the partial products are only generated due to the 1's in the multiplier. Therefore, the larger the number of zeros in the multiplier, the faster the multiplication. In addition, a string of continuous 1's can also be skipped by only generating partial products for the boundaries of that string. Consider a string of k consecutive 1's in the multiplier, where the first 1 is in bit position i and the last 1 is in bit position $i+k$. These k consecutive 1's can be replaced by a -1 in bit position i, a +1 in bit position $i+k+1$, and 0's in bit positions $i+1$ through $i+k$. This recoding of multiplier bits is repeated, for all strings of consecutive 1's in the multiplier

Two partial products are generated for each related pair of a -1 and a +1 The one corresponding to -1 is the 2's complement of the multiplicand shifted $i-1$ times while the other is the multiplicand shifted i+k times [2]. So, according to Booth's algorithm the number of partial products generated is equal to the number of variations in the multiplier from 0 to 1 or vise versa. The speed of multiplication in Booth's algorithm depends upon the bit configuration in the multiplier. Another advantage of this algorithm is that it treats signed and unsigned numbers in the same way

4.1.3 Modified Booth's Algorithm

The drawback of Booth's algorithm is that the speed of this multiplication is absolutely data dependent and in the worst case the speed of this algorithm becomes same as that of the simple add-shift multiplication. So, Booth's algorithm is modified to get better results The modified Booth's algorithm guarantees that an n-bit multiplier will generate $\lceil n/2 \rceil$ partial products. There is no need for precomplementing the multiplier or postcomplementing the product.

In the modified Booth's algorithm, the multiplier is first shifted to the left by one bit position to get a *0* in the LSB position, then the multiplier is divided into substrings of three consecutive bits in such a way that adjacent groups share a common bit. The partial products are then generated according to the bit configuration of these substrings. There are eight possible combinations of three bits, hence there are eight possible actions to be taken The shifted version of the multiplier is scaned from the least significant bit group to the most significant bit group, and partial products are generated according to the bit configuration of each group as given in Table 4.1 Each of these partial products is shifted to the left twice with respect to the previously generated partial product

This encoding technique reduces the number of partial products to half of those appearing in the worst case of the standard algorithm, therefore the speed of the multiplication may be doubled [1,2] In this research, the modified Booth's algorithm is selected for multiplication on parallel hypercube computers. An extension of the modified Booth's algorithm involves the encoding of 3 bits simultaneously while examining 4 multiplier bits. Using this scheme, $\lceil n/3 \rceil$ partial products are generated. These partial products are reduced to two using the Wallace tree reduction technique which reduces n partial products to two summands, which are later added using any suitable technique

Bit			Operation	
2^{1}	2^{0}	2^{-1}		
Y_{i+1}	Y_i	Y_{i-1}		
0	0	0	add zero (no string)	+0
0	0	1	add multiplicand (end of string)	+X
0	1	0	add multiplicand (a string)	+X
0	1	1	add twice the multiplicand (end of string)	+2X
1	0	0	subtract twice the multiplicand (begining of string)	-2X
1	0	1	subtrct the multiplicand (-2X and +X)	-X
1	1	0	subtract the multiplicand (begining of string)	-X
1	1	1	subtract zero (center of string)	-0

Table - 4.1. Encoding the 3 multiplier bits in the modified Version of booth's algorithm.

4.1.3 LRCF Multiplication algorithm

This algorithm was proposed by Ercegovac and Lang. The LRCF (left-to-right carry free) scheme can be used both for sequential and combinational implementation. Attention is given to the combinational case in order to achieve the speed advantage. Let X be the radix-2 representation of the normalized fractional magnitude x

$$x = \sum_{i=1}^{n} X_i 2^{-i} \qquad X_i = \{0,1\}$$

and Y be the radix-r representation of the normalized fractional magnitude y

$$y = \sum_{i=0}^{n/q} Y_i\, r^{-i} \qquad Y_i = \{-r/2,\ldots,r/2\}$$

where $r = 2^q$.

The LRCF multiplication algorithm is a recurrence that produces a sequence of two accumulated partial products, w and p, which are given by the following equations

$w[j] = r(\text{fraction}(w[j-1] + xY_j)$

where $j = 0,\ldots,n/q$

and

$p[j] = p[j-1] + Z_j r_{-j}$

where $Z_j = \text{integer}(w[j-1] + xY_j)$

The initial values of p and w are $w[-1] = p[-1] = 0$.

In this algorithm, the multiplier is scanned from most significant digit to the least significant digit, unlike in the conventional multiplication scheme where the multiplier is scanned from right to left.

According to the LRCF algorithm, the sum of the partial products after k steps is

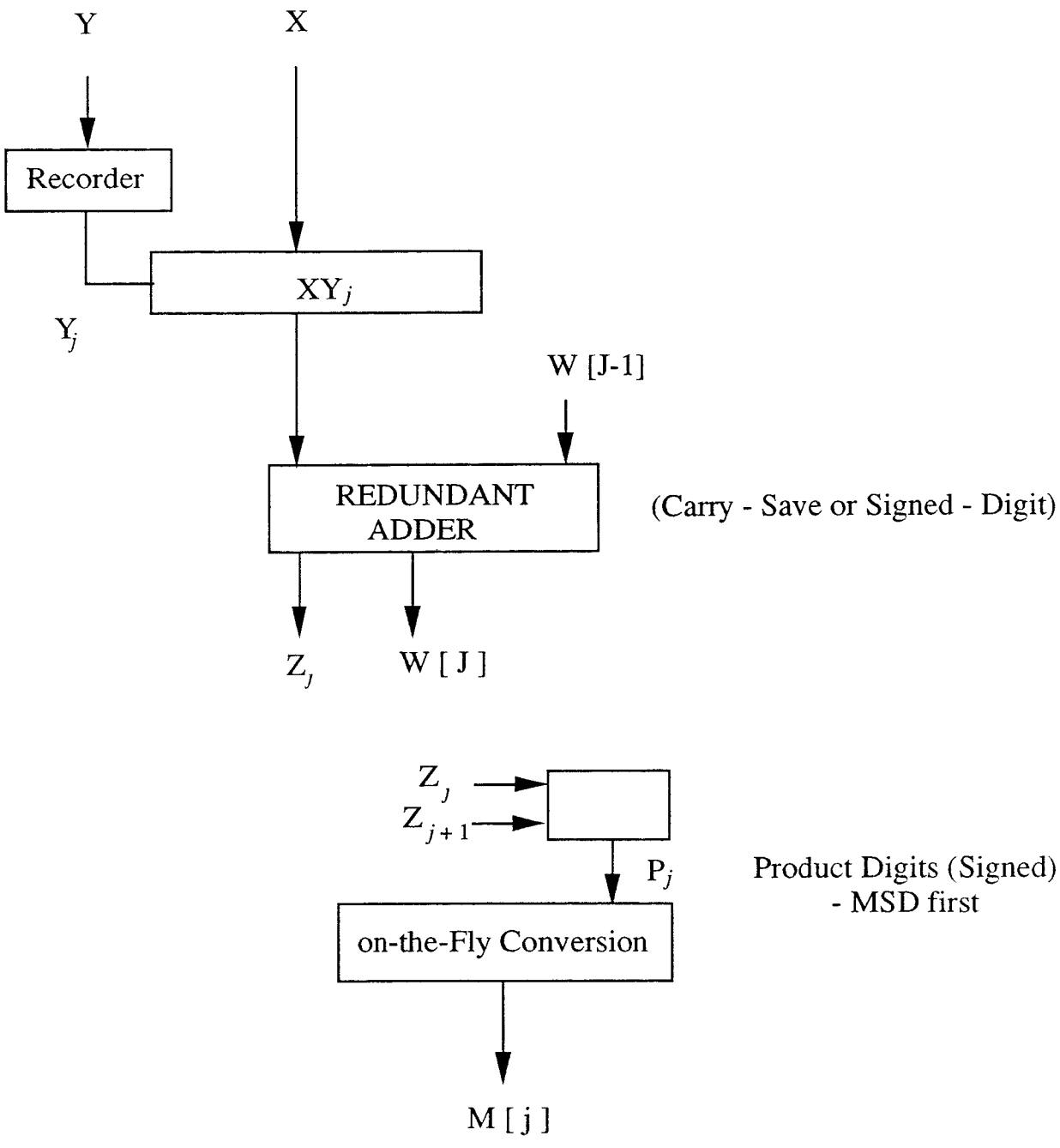


Fig - 4.1 One Step of the LRCF Multiplication Scheme

$$p[k]+w[k]xr^{-k-1} = \sum_{i=0}^{k} xY_i \times r^{-i}.$$

Similarly, after n/q steps we have

$$p[n/q] + w[n/q] \times r^{-n/q-1} = \sum_{i=0}^{n/q} xY_i \times r^{-i} = xy.$$

p[n/q] and w[n/q] are the upper and lower parts of the product.

A block diagram of the one step of recurrence is shown in Fig 4.1. A fast implementation of this scheme also requires fast adders (redundant adder [11]), addition by concatenation and on-the-fly conversion of the signed digit representation of p[j].

Several other multiplication algorithms exist, like table look-up multiplication, planar array multiplication and 2's complement array multiplication [2].

4.2 New Multiplication Algorithm

High speed multiplication is required in various applications In this section, a new parallel algorithm for fast multiplication is developed, based on the modified Booth's algorithm. Since the multiplication time heavily depends on the number of partial products, the modified version of Booth's algorithm reduces the number of partial products to half. Additional speedup is obtained by making use of parallel processing. The task of multiplication is divided into independent subtasks, which are processed simultaneously by different processors of the hypercube. The partial products are added by using the newly developed method discussed in the previous chapter. The multiplication operation is performed between the two operand (i.e , the multiplicand Md and the multiplier Mr). If the multiplier Mr contains *n* bits and the multiplicand Md contains *m* bits, then their product will contain *n+m* bits. The multiplication algorithm discussed in this section is composed of two phases. In the

first phase, the partial products are generated in parallel in different processors, and in the second phase they are reduced to two operands using carry save addition and finally these two operands are added using the conditional sum logic as discussed in the previous chapter.

In the first phase, the operands are first distributed among the processors of the hypercube. The hypercube is mapped into a linear array as discussed in the second chapter by labeling each processor using reflected Gray codes. Processors with their binary labels differing in a single bit position are directly connected to each other, the operand with the smaller number of bits is chosen as the multiplier (i e., $n \leq m$), because this produces the smaller number of partial products. The two's complement of the multiplicand Md is also found by using the addition algorithm described previously. The number of partial products generated when using the modified Booth's algorithm is equal to $a=\lceil n/2 \rceil$, and the number of partial products per processor in the d dimensional hypercube containing 2^d processors is equal to $\lceil \lceil n/2 \rceil / 2^d \rceil = b$. If the number of processors available is greater then the number of partial products, then only those processors work that contain multiplier's bits. Before distribution of the operands, the multiplier *Mr* is shifted toward left by one bit position and its sign extension is done in order to make its total number of bits an exact multiple of the number of processors.

The number of processors in use is equal to $\lceil a/b \rceil = P_{used}$ and the total number of multiplier bits to be distributed is equal to $l=(2xP_{used}\ xb)$, therefore Mr is extended to l bits. The technique for distribution is high-order interleaving and groups of Z ($Z = 2xb+1$) consecutive bits are divided among the processors in such a way that the processor with label G[i] contains the group Z_i and two consecutive groups Z_i and Z_{i+1} have one bit common between them. Therefore, the processor G[0] contains group Z_0, where Z_0 contains all the bits with the subscript j such that $0 \leq j \leq 2xb$. the processor G[1] contains the group Z_1 which contains $2xb \leq j \leq 4xb$.

So the processor with the label G[i] contains the group Z_i, ,which is composed of the bits of the multiplier with subscript j such that $i2xb \leq j \leq 2xb(i+1)$.

Then left shifted versions of the multiplicand and its two's complement are stored in each processor The number of left shifts in these operands depends on the address of the processor in which they are stored. The number of left shifts in the *i*th processor having label G[i] is equal to $2xixb$. Sign extension of these operands is first performed. As the product of Md and Mr contains $n+m$ bits, both the multiplicand and its two's complement must be extended to make their number of bits equal to $n+m$.

After this distribution, the generation of partial products starts, simultaneously in all the processors. Each processor G[i] contains b partial products obtained by scanning triples of consecutive bits in the multiplier Z_i stored in it and generating the partial product depending upon the bit configuration of that scanned group. There are eight possible combinations for the scanned three-bit group and the corresponding partial product is generated according to Table 4.1. If b>1, this means that more than one partial product is generated in any processor, and this requires proper shifting of the multiplier, the multiplicand and the two's complement of the multiplicand, after the generation of each partial product: this shifting process is repeated b times. This process is described mathematically as follows. Let Z_i, be the group of multiplier bits in processor G[i], and Y_i be the scanned three bits group then the value of Y_i gives the partial product as described in Table 4.1.

Then shifting of all those operands by two bits places is done as,

$Z_i = \lfloor Z_i/2^2 \rfloor$ (right by two bit positions)

$Md_i = Md_i * 2^2$ (left by two bit positions)

$CMd_i = CMd_i * 2^2$ (left by two bit positions)

Where CMd_i is the two's complement of Md_i.

Therefore, each processor generates b partial products The total number of partial products is a, but some time it is possible that $b*P_{used} > a$, so only the first a partial products are useful.

The second phase of the algorithm involves the reduction of the partial products. We remind that the number of processors available in the d-dimensional hypercube is 2^d and the hypercube is mapped into a linear array of 2^d processors. Each partial product contains $n+m$ bits and is distributed among the 2^d processors using high-order interleaving. This distribution is done in parallel by all the processors in $b*q*d*2^{d-1}$ cycles. But $n+m$ may not be evenly divided by 2^d, therefore sign extension of each operand is performed to make the number of bits equal to the nearest multiple of 2^d. Therefore, all the partial products are extended to l bits where $l=2^d\lceil(n+m)/2^d\rceil$ Let for each processor the group Q_k contains q consecutive bits of the partial product k, where $q=\lceil(n+m)/2^d\rceil$ and $0\leq k<a$ This interleaving is like in the addition algorithm discussed in the previous chapter, but in this case each processor contains more than two groups of q consecutive bits

The iterative reduction of partial products is performed in all processors simultaneously using the CSA technique discussed in the previous chapter. In any processor i three groups Q_0, Q_1 and Q_2 are reduced to two groups of the same number of bits (i.e , S and C) Let Q_{kr} represent the rth bit of the kth group in processor G[i], where $0\leq r<q$. Following parallel operations performed in the first cycle of the reduction process, we get

$S_r = Q_{0r} \oplus Q_{1r} \oplus Q_{2r}$,

$C_r = Q_{0r}.Q_{1r} + Q_{2r}(Q_{0r}+Q_{1r})$

where S_r and C_r represent the rth bit of the sum and the carry group respectively.

The next step involves the shifting of the carry toward left by one bit position, to get the proper alignment of the carry and sum bits. This alignment is achieved by

left shifting C by one bit position in all the processors simultaneously, such that the MSB of C in processor G[i] is copied in the LSB position of C in processor G[i+1], except for the MSB of C in G[2^d-1], which is discarded In this process, communications that require one hop are performed.

After that the following set of iterations in the reduction process are performed in parallel.

$S_r = S_r \oplus C_r \oplus Q_{kr}$,

$C_r = S_r C_r + Q_{kr}(S_r+C_r)$,

where $3 \leq k < a$.

In each reduction cycle, three operands are reduced to two, in parallel, by using the CSA technique and total of ($\lceil n/2 \rceil$ - 2) iterations are required to reduced $\lceil n/2 \rceil$ partial products to two and after each iteration proper alignment of C is performed. This is a totally parallel operation, because S and C can be calculated simultaneously for all bit positions independently of one another.

The last iteration results in two operands S and C properly aligned, which are added using the algorithm discussed in the previous chapter.

4.3 Comparison with Existing Algorithms

The algorithm discussed in the last section is based on the modified Booth's algorithm which approximately doubles the speed of the multiplication process by reducing the number of partial products to half. More specifically, in conventional multiplication, with n multiplier bits, the number of partial products generated is equal to n, whereas using Booth's recoding technique the number of partial products is $\lceil n/2 \rceil$. Another factor that causes further speed up is the use of parallel processing Using the hypercube structure, the task of multiplication is divided into subtasks which are processed simultaneously in all the processors. The third factor for its high speed is the way, the addition of multiple operands (partial products) is

performed. The carry save addition technique is used to reduce these operands into two operands, with each iteration reducing three operands into two operands (carry and sum).

If the hypercube of dimension d, which contains 2^d processors, is used to perform the multiplication, then the multiplication process is divided into the following steps, and the time for each step is given as follows.

1) Generation of partial products

Each processor scans consecutive bits of the multiplier and generates b partial products; after each generation of a partial product, it shifts the multiplier, the multiplicand and the 2's complement of the multiplicand by two bits, except for the generation of the last partial product. b cycles are consumed for the generation of partial products and 3(b-1) cycles in shifting three operands. So, the total time for the first step of multiplication is

$$T_1 = b + 3(b-1).$$

2) Partial products are redistributed among all the processors, and this operation is performed in parallel by all the processors in time

$$T_2 = bxqxdx2^{d-1}.$$

2) Reduction of partial products

The carry save addition technique is used in this step and $\lceil n/2 \rceil$ partial products are reduced to two. All of these partial products are distributed among the 2^d processors using high-level interleaving, as discussed in the previous section $q=\lceil (n+m)/2^d \rceil$ is the number of bits of each partial product present in one processor, and in each iteration three q-bit groups are reduced into two (i.e., carry and sum) and then the carry is shifted toward left by one bit position. Each shift operation requires the communication of one bit to an adjacent processor. The total time taken by this step is

$$T_3 = (a-2)(2\lceil (n+m)/2^d \rceil + 3)$$

Table 4.2 (a) Comparative Analysis for multiplication for different sizes of operands. Multiplier bits = 10 and multiplicand bits = 100.

Dimension of Hypercube	proc.	q (bits/proc)	Tb cycles	Tc cycles	Ts cycles
0	1	110	1017	2143	1018
1	2	55	685	1364	1305
2	4	28	503	912	1432
3	8	14	323	649	1537
4	16	7	322	407	1638
5	32	4	397	452	2310
6	64	2	449	484	3238
7	128	1	510	535	5030
8	256	1	1092	1117	10214
9	512	1	2378	2403	20710
10	1024	1	5200	5225	41958
11	2048	1	11350	11375	84966
12	4096	1	24668	24693	172006
13	8192	1	53346	53371	348134
14	16384	1	114792	114817	704486
15	32768	1	245870	245895	1425382
16	65536	1	524404	524429	2883558

(b) Multiplier bits = 100 and multiplicand bits = 1000.

Dimension of Hypercube	proc.	q (bits/proc)	Tb cycles	Tc cycles	Ts cycles
0	1	1100	109242	219493	109198
1	2	550	68448	137399	136845
2	4	275	41731	82605	137364
3	8	138	25442	49340	132940
4	16	69	15845	29525	129226
5	32	35	9245	18500	131297
6	64	18	5420	10829	139675
7	128	9	5111	6161	155800
8	256	5	5809	6459	207576
9	512	3	7409	7859	310744
10	1024	2	10644	10994	516824
11	2048	1	11575	11825	821976
12	4096	1	24893	25143	1646296
13	8192	1	53571	53821	3296984
14	16384	1	115017	115267	6602456
15	32768	1	246095	246345	13221592
16	65536	1	524629	524879	26476248

Table 4.2[(c)] Comparative Analysis for multiplication for different sizes of operands. Multiplier bits = 500 and multiplicand bits = 1000.

Dimension of Hypercube	proc.	q (bits/proc)	Tb cycles	Tc cycles	Ts cycles
0	1	1500	750242	1501493	749998
1	2	750	469248	938999	938245
2	4	375	282631	564005	940864
3	8	188	166892	331640	903290
4	16	94	95864	191775	869301
5	32	47	54337	108684	857109
6	64	24	31202	62393	897661
7	128	12	17532	35039	978097
8	256	6	9932	19829	1160491
9	512	3	9209	11459	1538344
10	1024	2	12044	13794	2563624
11	2048	1	12575	13825	4097576
12	4096	1	25893	27143	8198696
13	8192	1	54571	55821	16402984
14	16384	1	116017	117267	32815656
15	32768	1	247095	248345	65649192
16	65536	1	525629	526879	131332648

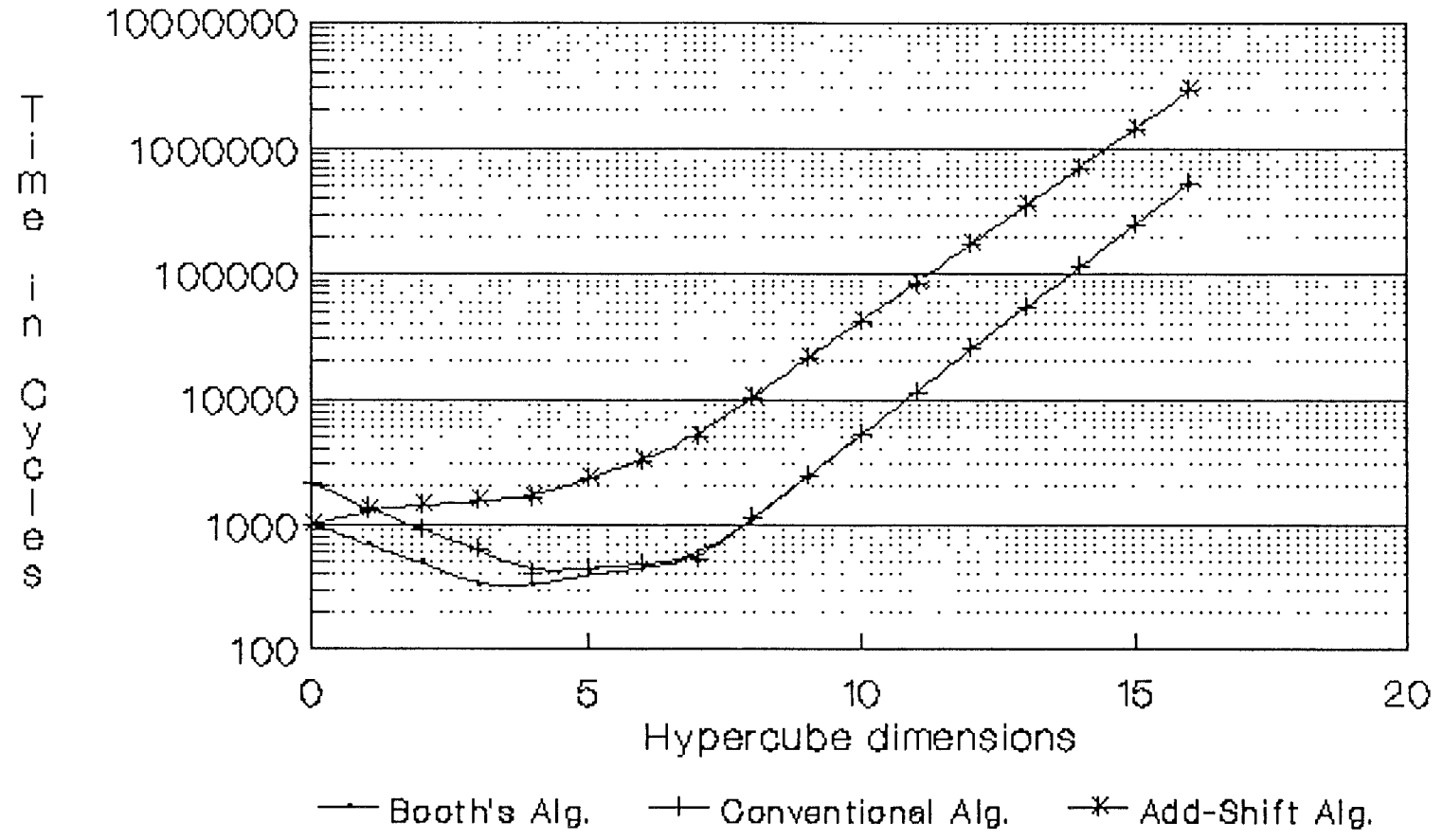


Fig 4.2 (a) multiplier bits - 10,
multiplicand bits - 100.

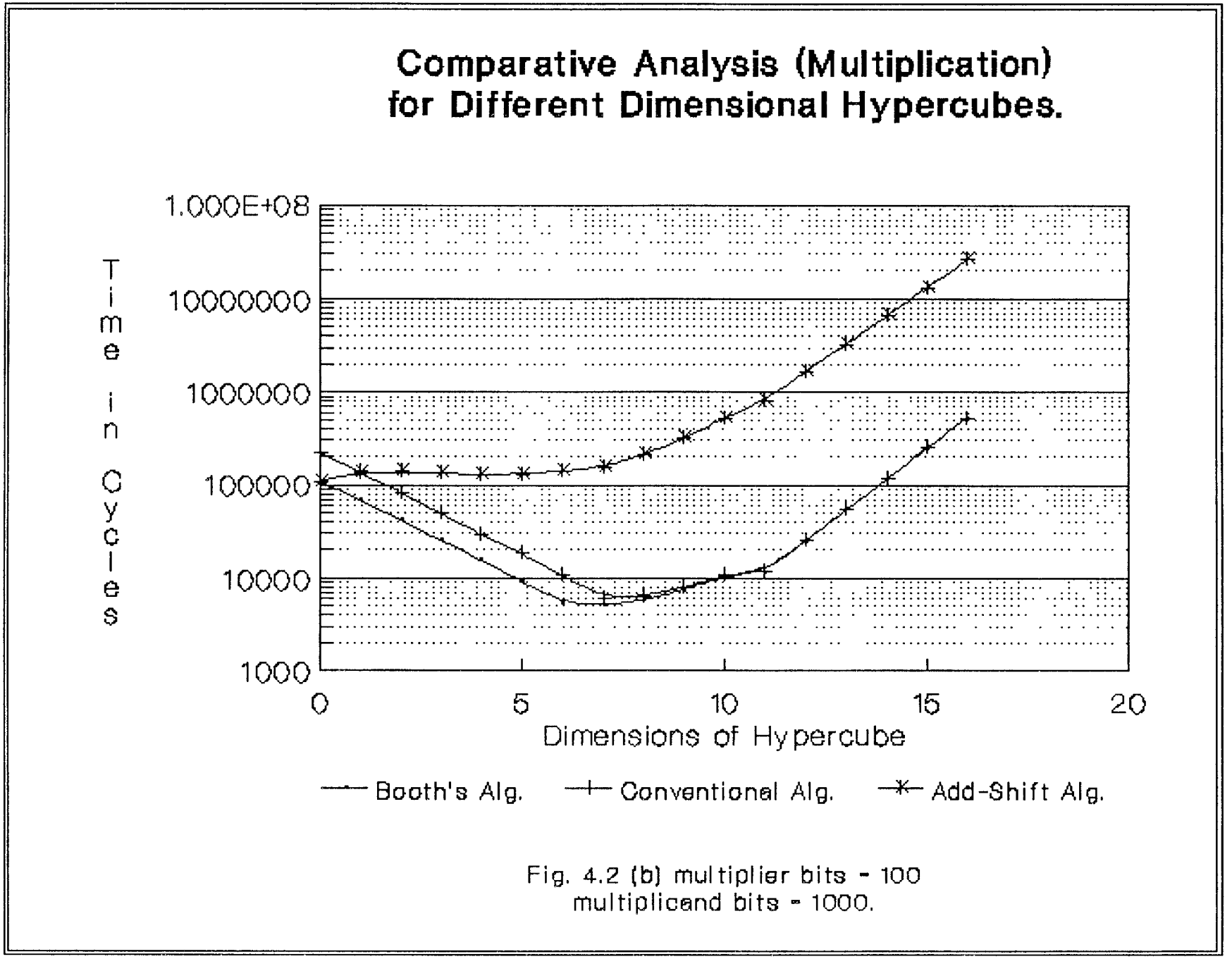


Fig. 4.2 (b) multiplier bits - 100
multiplicand bits - 1000.

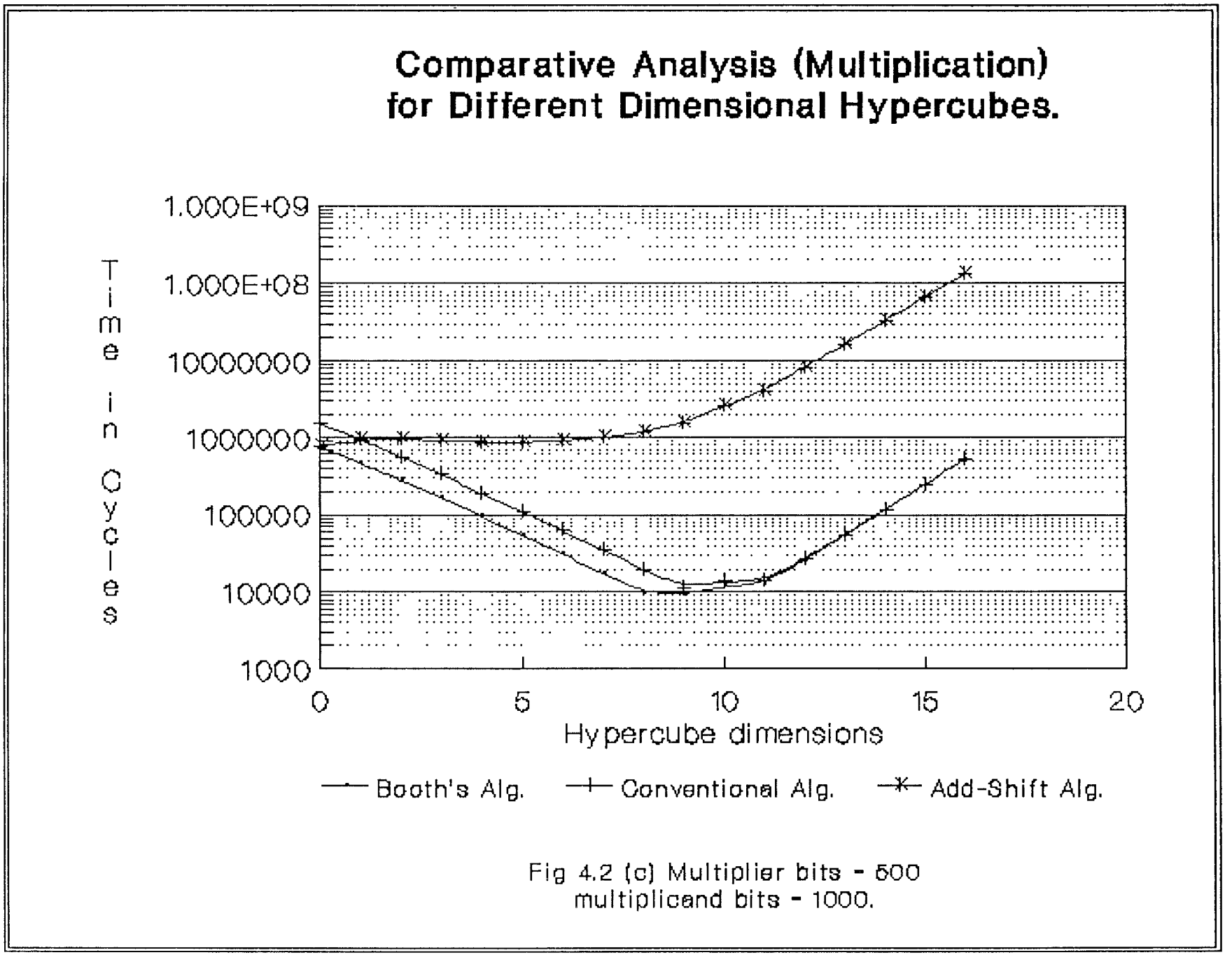


Fig 4.2 (c) Multiplier bits - 500
multiplicand bits - 1000.

3) The third and final step is the addition of the sum and the carry bits using the method discussed in the previous chapter. This process takes time

$$T_4 = T_{add} = 3\lceil (n+m)/2^d \rceil + 6\log_2(2^d) + 1$$

Therefore, the time for multiplication is given by

$$T_{mul} = T_1+T_2+T_3+ T_4$$

$$T_{mul} = b + 3(b-1) + (a-2)(2\lceil (n+m)/2^d \rceil + 3) + 3\lceil (n+m)/2^d \rceil + 6\log_2(2^d) + 1 + bxqx2^{d-1}\log_2(2^d), \quad (4.3.1).$$

where

$q = \lceil (n+m)/2^d \rceil$ = number of bits from each operand per processor to be added,

$d = \log_2(2^d)$ = the dimension of the hypercube,

$a = \lceil n/2 \rceil$ = total number of partial products generated, using the modified Booth's algorithm, and

$b = \lceil a/2^d \rceil$ = number of partial products generated per processor.

Using the above notations, Eq. 4.3.1 can be rewritten as

$$T_{mul} = b + 3(b-1) + (a-2)(2q+3) + 3q + 6d + 1 + bxqxdx2^{d-1} \quad (4.3.2)$$

In order to compare the performance this new algorithm with that of conventional multiplication algorithm, it is required to find out the time taken by the conventional multiplication algorithm, if it is implemented in the same parallel processing environment. In conventional multiplication, the number of partial products generated is equal to the number of bits in the multiplier, therefore the total number of partial products is n which can be reduced by half by using the modified Booth's algorithm. Even if we use the CSA technique for the reduction of the partial products and the new addition algorithm for addition, the time taken by the conventional method of multiplication will be higher due to different values of a and

b in Eq. 4.3.2, but the second term of above equation is reduced to 2(b-1) because only two operands are required to be shifted after the generation of each partial product (i.e., the multiplier and the multiplicand). The total time taken by multiplication using the conventional way of multiplication is

$T_{mul} = b + 2(b-1) + (a-2)(2q+3) + 3q + 6d + 1 + bxqxdx2^{d-1}$.

where

$a = n$ and

$b = \lceil n/2^d \rceil$.

The third algorithm for multiplication used here in the comparative analysis is completely sequential add-shift method. The time consumed in multiplication using this method is divided into following steps.

1) Generation of partial products

Each processor generates $b = \lceil n/2^d \rceil$ partial products and two operands are required to be shifted after each generation. Total time for this step is given by

$T_1 = b + 2(b-1)$.

2) Redistribution of partial products among all the processors in parallel takes the time

$T_2 = bxqxdx2^{d-1}$.

3) Total time to add $a=n$ partial products using conventional ripple carry addition is

$T_3 = (a-1)[2^d(q+3)-3]$.

Total time using this method of multiplication is

$T = b + 2(b-1) + (a-1)[2^d(q+3)-3] + bxqxdx2^{d-1}$.

Results obtained by using both these two algorithms are summarized in Table 4.2 and Fig 4.3 for comparison between these two algorithms. Simulation for comparative analysis for these methods of multiplication is given in appendix.

Chapter 5

CONCLUSIONS

The results obtained in previous chapters show that our initial goal to improve the speed of addition and multiplication have been achieved. The comparative analysis of the new and existing algorithms shows that our new algorithms give higher performance for operands operands containing a large number of bits.

The Conditional Sum Addition logic has been chosen for addition. This algorithm has been proved the fastest algorithm for addition by Winograd [1] and by J. Sklansky [7] as shown in Fig. 3.3. Using this method of addition, the conditional sum is found assuming conditional carries, and then groups of bits are combined with each other according to their correct position, as described in the previous chapter. Assuming conditional carries, this addition can be performed in parallel and a small amount of communication is involved. The modified Booth's algorithm is the basis of our multiplication process; this process generates $\lceil n/2 \rceil$ partial products for n bits in the multiplier, whereas in the conventional way the number of partial products is n. Therefore the smaller number of partial products may double the speed of multiplication. Additional speed is obtained by the generation of

partial products in parallel and by reducing the partial products to two, using the carry save addition (CSA) technique.

These algorithms were designed for hypercube parallel computer, operating in the SIMD mode of computation. The hypercube is well suited to these techniques due to its rich interconnection structure. Due to this elegant feature, the communication delay between the processors is minimum. Although the cost of arithmetic operations is increased by using the hypercube structure, but the speed of computation is also increased enormously for operands containing a large number of bits. The analysis performed in previous chapters shows that the performance of our algorithms is not good for small numbers of bits in the operands due to the large portion of communication overhead.

Although the speed obtained by our techniques is better then the speed of previously existing techniques, improvements are still possible. In the proposed algorithm for addition, groups of q bits work independently of each other to some extent. But the proper combination of these groups still depends on each other, and this part takes time $O(\log_2 N)$, where N is the number of processors in the hypercube.

Carry free operations are only possible in some particular number representations; although these operations are then extremely fast they consume large amounts of time during the conversion of the result to the standard binary system. However, the positional residue number system [4], can overcome this problem to some extent, because the conversion of numbers is performed digitwise, i.e., this conversion does not depend on adjacent digits. The incorporation of this technique in the conversion process can yield efficient arithmetic operations. This can be a future research goal.

Bibliography

[1] S. Waser and M.J. Flynn, *Introduction to Arithmetic for Digital System Designers*, Dryden, Holt, Rinehart & Winston, 1982.

[2] J.J.F. Cavanagh, *Digital Computer Arithmetic, Design and Implementation*, McGraw-Hill, New York, 1984.

[3] K. Hwang and F A Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[4] S. Mukhopadhyay, A. Basuray, and A.K. Datta, "New Technique of Arithmetic Operation using the Positional Residue System", *Applied Optics*. vol. 29, No 20, July, 1990, pp. 2981-2983.

[5] Michael T. Heath, *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, 1987.

[6] J. Sklansky, "Conditional Sum Addition Logic", *Trans. IRE*. vol. EC-9, No. 2, June 1960, pp. 226-230.

[7] J. Sklansky, "An Evaluation of Several Two-Summand Binary Adders", *Trans.*

IRE. vol. EC-9, No 1, June 1960, pp. 213-225.

[8] S. Ranka and S. Sahni, "Odd Even Shifts in SIMD Hypercubes", *IEEE Trans. on Parallel and Distributed Systems,* vol. 1, No. 1, Jan 1990, pp. 77-82.

[9] S.L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Artectures", *J. of Parallel and Distributed Computing,* vol. 4, 1987, pp. 133-172.

[10] S.G. Ziavras and L.S. Davis, "Fast addition on the FAT Pyramid and its Simulation on the Connection Machine", *Report No. CAR-TR-383 and CS-TR-2093, Center for Automation Research,* University of Maryland, August 1988.

[11] M.D. Ercegovac and T. Lang, "Fast Multiplication without Carry Propagation Addition", *IEEE Trans on Computers*, vol. 39, No 11, November 1990, pp. 1385-1393.

[12] Y. Saad and M.H. Schultz, "Topological Properties of Hypercubes", *IEEE Trans. on Computers*, vol. 37, No. 7, July 1988, pp. 867-872.

Appendix

```
/************************************************/
/* Simulation of Conditional Sum Addition Logic   */
/* on the Hypercube Structure.                    */
/************************************************/









#include<stdio.h>
#include<math.h>

     /* Global variables */

     static int g[64];
     static long int x[64],y[64],sum;




main()

{

     static int
i,j,k,intsz,mdbit,b_pp,lintsz,dim,nproc,p_used,used_dim;
     static long int na,nb,m_d,plast;


     intsz = 8*sizeof(int);
     lintsz = 2*intsz;
     /* Get the value of two operands to be added   */

     puts("\ngive the no. na and nb to be added\n");
     scanf("%ld\n%ld",&na,&nb);
     /* Get the dimension of hypercube available for
addition  */

     puts("\ngive the dimension of the hypercube\n");
     scanf("%d",&dim);
     printf("\nna = %ld\nnb = %ld\ndim = %d",na,nb,dim);


     /* Choose the  multiplier */
```

```
     if(labs(na)>labs(nb))
                    m_d = na;
                else    m_d = nb;
     /* Count the number of bits in operands */

     mdbit = count(labs(m_d)) + 1;

     printf("\n no: of bits to be added = %d\n",mdbit);

     /* find the number of processors  in the hypercube */

     nproc = pow((double)2,(double)dim);

     /* call the function to get the Gray codes */

     gray(nproc);

     /* Find the number of bits per processor   */

     b_pp = ceil((double)mdbit/(double)nproc);

     if(nproc>mdbit)
     {

     p_used = mdbit;
     plast = p_used;
     used_dim =
ceil((double)(log((double)p_used)/log((double)2)));
     p_used = pow((double)2,(double)used_dim);
     }
     else
     {
     p_used = nproc;
     plast = p_used;
     used_dim = dim;

     }

     /* distribute operands among the processors of
hypercube using
        high-order interleaving */

     dist(na,nb,p_used,b_pp);

     /* Call the function to add these operands in parallel
*/

     addition(p_used,used_dim,b_pp);

     /* recover the sum from the processors     */

     recov(b_pp,p_used);
```

```
      printf("sum = %ld", sum);
      }




      count(n)
      long int n;
      {
        int x;
        x =
ceil((double)log((double)(n+1))/(double)log((double)2)) + 1;
        return(x);
        }


      gray(n)
      int n;
      {
           int i;
           for(i=0; i<n; i++)
           {
                g[i] = i>>1;
                g[i] ^= i;
            }
        }

        dist(na,nb,p_used,b_pp)
        long int na,nb;
        int p_used,b_pp;
        {
      int i,mask;
      mask = pow(2,b_pp)-1;

      for(i=0; i<p_used; i++)
      {

           x[g[i]] = na>>(b_pp*i) & mask;
           y[g[i]] = nb>>(b_pp*i) & mask;


      }
      }


      addition(p_used,used_dim,b_pp)
      int p_used,used_dim,b_pp;
      {

static    int
i,j,gc[64],gn[64],shift,ini,fin,a,b,mask[64],k,t_ini,t_fin;
```

```
static    long int temp1,temp2,tempn[64],tempc[64];
static    int oper[64],t_mask[64],tempo;

     for(i=0; i<p_used; i++)
     {

               if(b_pp ==1)

          {
          temp1 = x[g[i]] ^ y[g[i]];
          temp2 = x[g[i]] ^ y[g[i]]^ 1;


          gn[g[i]] = x[g[i]] & y[g[i]];
          gc[g[i]] = x[g[i]] & y[g[i]] | (x[g[i]] |
y[g[i]]);


          x[g[i]] =  temp1;
          y[g[i]] =  temp2;


     }

               else if(b_pp>>1)

     {
          temp1 = x[g[i]] + y[g[i]];
          temp2 = x[g[i]] + y[g[i]] + 1;

          x[g[i]] = temp1 & ((int)pow(2,b_pp)-1);
          y[g[i]] = temp2 & ((int)pow(2,b_pp)-1);
            shift =pow(2,b_pp);
          if(temp1 & (int)pow(2,b_pp))

               gn[g[i]] = 1;
          else gn[g[i]] = 0;

          if(temp2 & (int)pow(2,b_pp))
               gc[g[i]] = 1;
          else gc[g[i]] = 0;



          }

          mask[g[i]] |= gn[g[i]];
          mask[g[i]] <<= 1;
          mask[g[i]] |= gc[g[i]];
          oper[g[i]]  = 1;

          }
```

```
for(i=1; i<=used_dim; i++)

{
t_ini = 0;
t_fin = pow(2,i)-1;
for(j=1; j<=pow(2,(used_dim-i)); j++)

{
ini = t_ini;
fin = t_fin;


for(k=1; k<=pow(2,(i-1)); k++)


{
t_mask[g[fin]] = mask[g[ini]];

if(!(mask[g[fin]] == 0 || mask[g[fin]] == 3))
{

switch (t_mask[g[fin]])

{
     case 0:
          mask[g[fin]] ^=1;
          break;

     case 1:
          break;
     case 2:
          mask[g[fin]] ^=3;
          break;
     case 3:
          mask[g[fin]] ^=2;
          break;

}       }

switch (oper[g[fin]])

 {
     case 0:
     case 3:
          break;

     case 1:

          oper[g[fin]] = t_mask[g[fin]];
          break;
     case 2:

          switch (t_mask[g[fin]])
           {
```

```
            case 0:
                oper[g[fin]] = 3;
                break;

            case 3:
                oper[g[fin]] = 0;
                break;

            case 1:

                break;
            case 2:
                oper[g[fin]] =1;
                break;

           }
           }

if(i != used_dim) mask[g[ini]] = mask[g[fin]];

++ini;
--fin;
}

t_ini += pow(2,i);
t_fin += pow(2,i);
}
}

for(i=0; i<=p_used; i++)

    {
     switch (oper[g[i]])

     {
     case 0:
          y[g[i]] = x[g[i]];
          break;

     case 1:

          break;

     case 2:
          tempo = x[g[i]];
          x[g[i]] = y[g[i]];
          y[g[i]] = tempo;
          break;

     case 3:

          x[g[i]] = y[g[i]];
          break;
```

```
                }
        }
        }

recov(b_pp,p_used)
int b_pp,p_used;
{

static int i,check,carry=0;

static long int a;

check = b_pp*p_used;
for(i=p_used-1; i>=0; i--)

{
if(carry==0)

        sum |= x[g[i]];

else
        sum |= y[g[i]];
if(i != 0) sum <<= b_pp;
}
a=1;

if(sum & (a<<check-1))

sum |= ~((a<<check)-1);

}
```

```
/*************************************************/
/* Simulation of Modified Booth's Algorithm for  */
/* the Multiplication on the Hypercube Structure, */
/* this algorithm uses carry save addition      */
/* technique for reduction of partial products   */
/* and  conditional sum addition technique for   */
/* final addition.                               */
/*************************************************/
```

```
# include <stdio.h>
# include <math.h>
# include <conio.h>
# include <graphics.h>
# include <dos.h>
/*# include "title.c"*/

/* global variables */
static int m_d,dim,cm_d,g[64],bin[64][8*sizeof(int)];
static int mdbit,mrbit,cou,sa,sb;
static long int
multi_d[64],cmulti_d[64],p[64][20],s[64],x[64],y[64],multi_r
[64];
static long int m_r,sum;
static long int z[64][128];



main()

{

     static int na,nb,ct,i,intsz,k,l,m;
     static int a,b,n_proc,pp_used,p_used,used_dim,b_pp;

/*   title();     */
     textbackground(11);
     textcolor(4);
     clrscr();


/*  get two operands of multiplication (multiplier and
multiplicand) */
```

```
/*  and the dimension of the hypercube used for
multiplication.      */


     puts("\n\tgive the value of na and nb\n");
     puts("\t");
     scanf("%d\n%d",&na,&nb);
     puts("\tgive the dimension of hypercube\n");
     puts("\t");
     scanf("%d",&dim);
     puts("\n");
     printf("\n\tna = %d\n\tnb = %d\n\tdim =
%d\n",na,nb,dim);

/*  chose the multiplier from two inputs
*/

     if(abs(na)>abs(nb))
               {m_d=na;
                m_r=nb;}
           else
               {m_d=nb;
                m_r=na;}

 /* call the function to find the no: of bits in each
operand with sign bit*/


    mdbit = count(abs(m_d));
    printf("\tno: of bits in multiplicand = %d\n",mdbit);
    mrbit = count(abs(m_r));
    printf("\tno: of bits in multiplier = %d\n",mrbit);

 /* call the function to find out the 2'complement of
multiplicand */

    cm_d = com(m_d);

    n_proc = pow((double)2,(double)dim);

 /* no of partial products in case of using modified Booth's
algorithm*/

    if(!(mrbit%2))
          a = mrbit/2;
       else  a = (mrbit + 1)/2;

 /* no of partial products per processor */

    b = ceil((double)a/(double)n_proc);
    printf("\n\tno of p.p = %d\n\tno of p.p / processor =
%d",a,b);

 /* call the function to generate the gray codes */
```

```
     gray(n_proc);

 /* call the function for data distribution */
 /* first shift m_r to left by one */

    p_used = ceil((double)a/(double)b);
    m_r <<= 1;
    dist(b,p_used);

 /* call the function to mask the multiplier and to generate
the
     partial products in each processor */

    p_pro(p_used,b);
    pp_used = p_used;

/* As the total number of bits in product is the sum of
multipier
   and multiplicand bits*/

    mdbit = mdbit+mrbit;
    b_pp = ceil((double)mdbit/(double)n_proc);
    if(n_proc>mdbit)
    {
    p_used = mdbit;
    used_dim =
ceil((double)(log((double)p_used)/log((double)2)));
    p_used=pow(2,used_dim);
    }
    else
    {p_used = n_proc;
    used_dim = dim;
    }

 /* distribute the partial products in the extended form
using
    high-order interleaving   */

     for(k=0; k<pp_used; k++)
          for(l=0; l<b; l++)
          {
          if(++ct <= a)
          {
          cou = ct-1;
          dist2(k,l,p_used,b_pp);
          }
          else goto next;
          }

 next:  if(a<3)

     for(i=0; i<p_used; i++)
     {
```

```
     if(a==1) x[g[i]] = z[g[i]][0];
     else if(a==2) {     x[g[i]] = z[g[i]][0];
               y[g[i]] = z[g[i]][1];
               }
               }
/* perform the carry save addition in case of more than two
   partial products */

     else if(a>=3)  csa(a,p_used,b_pp);

/* perform the addition operation using conditional sum
addition logic*/

     addition(p_used,used_dim,b_pp);

/* get the final sum from distributed hypercube processors
*/

     recov(b_pp,p_used);
     printf("\n\t ans = %ld\n",sum);


    }


    count(n)
    int n;
    {
     int x;
     x =
ceil((double)log((double)(n+1))/(double)log((double)2)) + 1;
     return(x);
     }


    com(a)
    int a;
    {
     int x;
     x = ~a + 1;
     return(x);
     }

   gray(n)
    int n;
    {
     int i,j;
     for(i=0; i<n; i++)
     {
          g[i] = i>>1;
          g[i] ^= i;
      }
    }
```

```
dist(b,p_used)
int b,p_used;
{
 int i,j,sz;

 for(i=0; i<p_used; i++)
      {
           multi_d[g[i]] = (long)m_d << 2*i*b;
           cmulti_d[g[i]]= (long)cm_d << 2*i*b;
           multi_r[g[i]] = (long)m_r >> 2*i*b;
                    puts("\n");
 /*                 sz = 8*sizeof(cmulti_d[g[i]]);
           for(j=0; j<sz; j++)
           printf("%d",(cmulti_d[g[i]]>> sz-(j+1)&1));
      */
       }
  }


  p_pro(p_used,b)
  int p_used,b ;
  {
 int i,j,k,mask = 0;
 for(i=0; i<p_used; i++)
 {

      for(j=0; j<b; j++)
          {
          mask = multi_r[g[i]] & 7;
          switch (mask)
          {
           case 0:
           case 7:
                p[g[i]][j] = 0;
                break;
           case 1:
           case 2:
                p[g[i]][j] = multi_d[g[i]];
                break;
           case 3:
                p[g[i]][j] = multi_d[g[i]] << 1;
                break;
           case 4:
                p[g[i]][j] = cmulti_d[g[i]] << 1;
                break;
           case 5:
           case 6:
                p[g[i]][j] = cmulti_d[g[i]];
                break;
             }

      multi_r[g[i]] >>= 2;
      multi_d[g[i]] <<= 2;
      cmulti_d[g[i]] <<= 2;
```

```
        }
    }
    }

dist2(k,l,p_used,b_pp)
int    k,l,p_used,b_pp;
{
static int i;
static long int m;

m = pow(2,b_pp)-1;


for(i=0; i<p_used; i++)
{
if(p_used == 1)   z[g[i]][cou] = p[g[k]][l];
else             z[g[i]][cou] = (p[g[k]][l] >> (b_pp*i))
& m;
}
}
csa(a,p_used,b_pp)
int a,p_used,b_pp;
{
static int i,j,k,mas[64];
static long int temp;
for(i=0; i<p_used; i++)
{
x[g[i]] = z[g[i]][0] ^ z[g[i]][1] ^ z[g[i]][2];
y[g[i]] = z[g[i]][0] & z[g[i]][1] | (z[g[i]][2] &
(z[g[i]][0] | z[g[i]][1]));
}
for(i=0; i<p_used; i++)
 {
mas[g[i]] = (y[g[i]] >> (b_pp-1))&1;
y[g[i]] = (y[g[i]] <<1) & ((int)pow(2,b_pp)-1);
}
for(i=0; i<p_used-1; i++)
{
y[g[i+1]] = y[g[i+1]] | mas[g[i]];
mas[g[i]] = 0;
}
for(j=3; j<a; j++)
{
  for(i=0; i<p_used; i++)
  {
  temp = x[g[i]];
  x[g[i]] = x[g[i]] ^ y[g[i]] ^ z[g[i]][j];
  y[g[i]] = temp & y[g[i]] | (z[g[i]][j] & (temp |
y[g[i]]));
  temp = 0;
  }
  for(i=0; i<p_used; i++)
```

```
     {
     mas[g[i]] = (y[g[i]] >> (b_pp - 1)) & 1;
     y[g[i]] = (y[g[i]] << 1) & ((int)pow(2,b_pp)-1);
     }
     for(i=0; i<p_used-1; i++)
     {
     y[g[i+1]] = y[g[i+1]] | mas[g[i]];
     mas[g[i]] = 0;
     }

     }


   }



     addition(p_used,used_dim,b_pp)
     int p_used,used_dim,b_pp;
     {

static    int
i,j,gc[64],gn[64],shift,ini,fin,a,b,mask[64],k,t_ini,t_fin;
static    long int temp1,temp2,tempn[64],tempc[64];
static    int oper[64],t_mask[64],tempo;

     for(i=0; i<p_used; i++)
     {

               if(b_pp ==1)

          {
          temp1 = x[g[i]] ^ y[g[i]];
          temp2 = x[g[i]] ^ y[g[i]]^ 1;


          gn[g[i]] = x[g[i]] & y[g[i]];
          gc[g[i]] = x[g[i]] & y[g[i]] | (x[g[i]] |
y[g[i]]);


          x[g[i]] =  temp1;
          y[g[i]] =  temp2;


     }

               else if(b_pp>1)

     {
          temp1 = x[g[i]] + y[g[i]];
          temp2 = x[g[i]] + y[g[i]] + 1;
```

```
x[g[i]] = temp1 & ((int)pow(2,b_pp)-1);
y[g[i]] = temp2 & ((int)pow(2,b_pp)-1);
  shift =pow(2,b_pp);
if(temp1 & (int)pow(2,b_pp))

     gn[g[i]] = 1;
else gn[g[i]] = 0;

if(temp2 & (int)pow(2,b_pp))
     gc[g[i]] = 1;
else gc[g[i]] = 0;



}

mask[g[i]] |= gn[g[i]];
mask[g[i]] <<= 1;
mask[g[i]] |= gc[g[i]];
oper[g[i]]  = 1;

}


for(i=1; i<=used_dim; i++)

{
t_ini = 0;
t_fin = pow(2,i)-1;
for(j=1; j<=pow(2,(used_dim-i)); j++)

{
ini = t_ini;
fin = t_fin;


for(k=1; k<=pow(2,(i-1)); k++)


{
t_mask[g[fin]] = mask[g[ini]];

if(!(mask[g[fin]] == 0 || mask[g[fin]] == 3))
{

switch (t_mask[g[fin]])

{
     case 0:
          mask[g[fin]] ^=1;
          break;

     case 1:
          break;
```

```
      case 2:
           mask[g[fin]] ^=3;
           break;
      case 3:
           mask[g[fin]] ^=2;
           break;

}       }

switch (oper[g[fin]])

 {
      case 0:
      case 3:
           break;

      case 1:

           oper[g[fin]] = t_mask[g[fin]];
           break;
      case 2:

           switch (t_mask[g[fin]])
            {

            case 0:
                oper[g[fin]] = 3;
                break;

            case 3:
                oper[g[fin]] = 0;
                break;

            case 1:

                break;
            case 2:
                oper[g[fin]] =1;
                break;

           }
           }

if(i != used_dim) mask[g[ini]] = mask[g[fin]];

++ini;
--fin;
}

t_ini += pow(2,i);
t_fin += pow(2,i);
}
}
```

```
      for(i=0; i<=p_used; i++)

           {
            switch (oper[g[i]])

            {
            case 0:
                 y[g[i]] = x[g[i]];
                 break;

            case 1:

                 break;

            case 2:
                 tempo = x[g[i]];
                 x[g[i]] = y[g[i]];
                 y[g[i]] = tempo;
                 break;

            case 3:

                 x[g[i]] = y[g[i]];
                 break;

               }
      }
      }

recov(b_pp,p_used)
int b_pp,p_used;
{

static int i,check,carry=0;

static long int a;

check = b_pp*p_used;
for(i=p_used-1; i>=0; i--)

{
if(carry==0)

      sum |= x[g[i]];

else
      sum |= y[g[i]];
if(i != 0) sum <<= b_pp;
}
a=1;

if(sum & (a<<check-1))

sum |= ~((a<<check)-1);
```

```
/*****************************************************/
/* Comparative Analysis of  Various Addition         */
/* Algorithms for different sizes of operands and    */
/* on the different dimensions of the hypercube      */
/*****************************************************/



#include <stdio.h>
#include <math.h>


main()
{
static long int q,tcar,ts,tcsa,proc,bits,i,j,k;
static int d;

scanf(" \n %ld",&bits);
puts("\n\n\n");

printf("\tTable 3.2 Comparative analysis of various addition\n\t algorit
puts("_____________________________________________________________");
puts("|Dimension of      |proc.|  q                |Tseq    |Tcla    |Tcsa    |
puts("|Hypercube         |     |(bits/proc)        | cycles|cycles |cycles |
puts("|__________________|_____|___________________|________|________|_______|")
for(i=0; i<=16; i++)
{
proc =  pow(2,i);
q = ceil((double)bits/(double)proc);
tcar = 4*q + 4*i*q + 2*i;
tcsa = 3*q + 6*i + 1;
ts = proc *(q+3) -1;
printf("   %d    \t",i);
printf(" %ld\t\t",proc);
printf(" %ld\t",q);
printf(" %ld\t",ts);
printf(" %ld\t",tcar);
printf(" %ld\n",tcsa);
        }
        }
```

```
/*********************************************************/
/* Comparative analysis of Modified Booth's Algorithm    */
/* and Sequential Add and Shift method                   */
/*********************************************************/


#include <stdio.h>
#include <math.h>


main()
{
static long int q,ts,proc,bits,i,j,k,a,b,tb,tc,mr,md,a1,b1;
static int d;

scanf(" \n %ld",&mr);
scanf("\n  %ld",&md);
puts("\n\n\n");
printf("Table 4.2 Comparative Analysis for multiplication for\n\t  diffe


puts("_____________________________________________________________________");
puts("|Dimension of      |proc.|  q                |Tb      |Tc      |   Ts   |");
puts("|Hypercube         |     |(bits/proc)        | cycles|cycles  | cycles |");
puts("|__________________|_____|___________________|________|________|________|");
for(i=0; i<=16; i++)
{
proc =  pow(2,i);
q = ceil((double)(mr+md)/(double)proc);
a = ceil((double)mr/(double)2);
b = ceil((double)a/(double)proc);
a1= mr;
b1= ceil((double)a1/(double)proc);
tb = b+3*(b-1)+(a-2)*(2*q+3)+ 3*q + 6*i + 1 + b*q*proc*i/2;
tc = b1+2*(b1-1)+(a1-2)*(2*q+3) +3*q + 6*i +1 + b1*q*proc*i/2;
ts = b1 +2*(b1-1)+ (a1-1)*(proc*(q+3)-3)+b1*q*proc*i/2;
printf("   %d    \t",i);
printf(" %ld\t\t",proc);
printf(" %ld\t",q);
printf(" %ld\t",tb);
printf(" %ld\t   ",tc);
printf(" %ld\t\n",ts);
        }
                }
```