SYSTEMATIC DESIGN OF

COMPUTATIONAL ARRAYS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Mokhtar A. Aboelaze

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

April 1988

this thesis is dedicated

to my mother and my father

ACKNOWLEDGMENTS

I want to express my gratitude to my major advisor Prof. Benjamin W. Wah for his guidance and encouragement during the course of this thesis. I also appreciate the willingness of prof's. Jose A. B. Fortes, Elias N. Houstis and Manoel F. Tenorio for serving on as committee members. I also want to thank Vickie DeMoss for carefully reviewing the manuscript of this thesis.

This work is partially supported by National Science Foundation grant #DMR 85-19649, a Joint Service Electronics Program grant #N00014-84-C-0149, a Semiconductor Research Corporation grant #87-DP-109

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

Table Page

Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

ABSTRACT

Mokhtar Aboelaze. Ph.D., Purdue University. August 1982. Systematic Design of Systolic Arrays Major Professor: B. W. Wah.

In this thesis we present a comprehensive overview for the design of a system of systolic arrays, from the VLSI layout level to the system level.

First, we discuss 3-D VLSI layout, where tighter lower and upper bounds for the volume and maximum wire length for the layout of the different families of graphs in a 3-D environment were developed. Except in two cases, all the bounds for the volume are optimal. The first case is the one-active-layer layout of the planar graphs, the other is the unrestricted layout for graphs with separators N^q ,$q = 2/3$. A cost model for reflecting the real cost of the layout, instead of taking the volume as a measure of cost, was also developed.

In Chapter 3, we develop a methodology for designing a systolic array starting from recurrence equations. The idea of *Control Flow Systolic Arrays* to handle uniform, as well as nonuniform recurrence equations, will be developed. This methodology is basically a search for a heuristic solution in the space of all the possible solutions. Because of the unlimited search space, the search process must be guided for the search to be completed in a reasonable amount of time

Chapter 4 introduces the idea of converting the data between two systolic arrays that were directly interfaced, instead of using a common memory which would be a bottleneck for the whole system. The minimum number of buffers required to convert the data between two given distributions was also calculated, a general purpose converter was also proposed.

CHAPTER 1
INTRODUCTION

1.1. Introduction

Recent years have seen parallel processing become a reality as numerous commercial and experimental machines with a variety of architectures have been introduced. In terms of functionality these machines span the whole spectrum of parallel processing from supercomputers, such as the Cray X-MP, to special-purpose application-dependent VLSI arrays. The major driving force behind this achievement lies in the great advances in VLSI technology. In the 1960's the average integrated circuits chip contained tens of transistors. Using today's technology, hundreds of thousands of transistors can be implanted on the same chip, thereby leading not only to more complicated but also less expensive chips.

In order to design a powerful and cost-effective computer system capable of solving complex problems, one must consider the underlying hardware and software system structure and the computing algorithms to be implemented on these computers. Computers were originally developed, and are still largely used, for data processing. Recently, the tendency has been to use computers for knowledge processing instead of for data processing. For example, a variety of expert systems has been developed for areas such as medicine, agriculture, oil exploration and manufacturing. This kind of application is computationally intensive, making the use of sequential computers inefficient.

Since the International Conference in Tokyo in October 1981, when Japan launched its National Fifth Generation Project [TrG82], there has been a growing belief in the computer architecture community that the traditional sequential control

flow computers will be superseded in the 1990's by a new generation of computers. One of the leading candidates for superseding traditional sequential machines is the VLSI processor arrays. This is true since for the past 15-20 years the number of transistors on a single chip is being doubled each one and half to two years. years. There is no indication that this trend will stop in the near future. In the rest of this Chapter, we will briefly review the concept of VLSI processor Arrays.

1.2. VLSI Processor Arrays.

Until recently, computation-intensive tasks in signal and image processing and in knowledge processing systems have been handled using high-performance general-purpose computers. These general-purpose computers, use extensive pipelining and parallel processing to enhance the performance of the system. Although somewhat effective, these general-purpose computers are very expensive and difficult to design. The operating system overhead also impedes the application of these computers in real time systems, where the speed and the throughput of the system is a crucial factor. To achieve the speed and throughput required for real time application, new architecture are needed.

An important architectural approach made possible by the availability of inexpensive special-purpose VLSI circuits is the systolic arrays. Systolic arrays make use of multiple regularly-connected processing elements to exploit the potential of pipelining and multiprocessing [Kun82] [Kun80a]. In a systolic array, several data items flowing along different pipes with the same or different rates may meet and interact. The major advantage of systolic processing is that each data item once accessed is used a number of times. Thus a high computational throughput can be achieved with a modest I/O bandwidth.

There are some constraints on the design of a VLSI chip [MeC80]. The quality of the chip depends upon the following criteria.

(1) Modularity: Simplicity and regularity have always been major concerns in designing special-purpose VLSI systems. Simple and regular interconnections lead to cheap implementations and high densities [MeC80], and high densities imply both

high performance and low overhead for support components [SuM77]. Another important concern is the cost-effectiveness. The cost of these special-purpose chips should be kept low enough to justify their limited applications. The cost of designing special-purpose Integrated Circuit chips can be reduced if the architecture of thee chip can be decomposed into regular and simple modules that will be repeated hundreds or thousands of times. This is especially efficient with chips that contain hundreds of thousands of transistors. To cope with complexity simple and regular designs are essential. VLSI systems based on simple regular layouts are likely to be modular and adjustable to various performance levels. For this reason we are interested in making the array as modular as possible. Figure 1.1 shows two different kinds of systolic arrays.

(2) Pipelineability: The three levels of pipelineability are the macro level, the intermediate level, and the micro level {RaK85}.

An array is said to be pipelineable at the macro level, if it can begin processing a new instance of a problem before it completely finishes processing the previous instance (when more than one problem instances are being solved at the same time in a pipelined fashion).

An array is said to be pipelineable at the intermediate level if for a single problem instance the maximum throughput achievable is independent of the number of processing elements, n, in the array for $n \geq K$, where K is some finite chosen integer.

The array is said to be pipelineable at the micro level if the processing elements (PE's) have pipelined adders, multipliers, or any functional devices.

(3) Communication: Communication between the processing elements should be restricted to immediate neighbors. A modular array of processing elements that restricts communication in this manner will be termed a VLSI array.

(4) Flexibility: There are two types of VLSI arrays. One type is characterized by inflexible hard-wired dedicated processors, the other allows some flexibility, such as programmability and reconfigurability. The first type is usually dedicated to solving one kind of problem, while the second can be reprogrammed to solve many problems. A PE in the programmable or reconfigurable array is usually more complicated than a

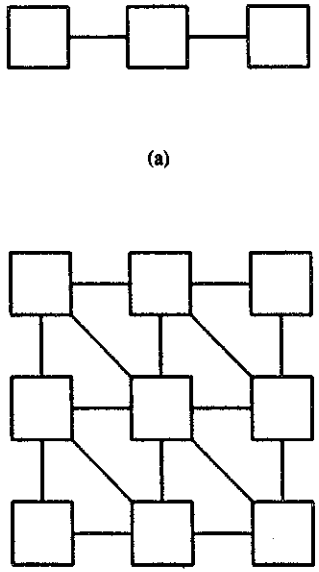


Figure 1.1
Two examples of systolic arrays. (a) a linear systolic array (b) a hexagonal systolic array I hope this is too long for one line

PE in the dedicated array. Therefore, the array architect must compromise between the complexity of the PE and the degree of flexibility he wants to introduce into the array.

Not every algorithm is suitable for implementation on a systolic array. A a compute-bound algorithm, in which the number of computing operations is larger than the total number of input and output elements is suitable for VLSI implementation For example, the matrix-matrix multiplication algorithm represents a compute-bound task, which has $O(n^3)$ multiply-add steps, but only $O(n^2)$ I/O elements. On the other hand, I/O bound problems, in which the number of I/O operations is larger than the number of computations, is not suitable for VLSI implementation. For example, adding two matrices is an I/O bound problem since there are n^2 computations and $3n^2$ I/O operations.

The first attempt to formally describe the systolic system was Charles Leiserson in [Lei83a]. He represented the systolic system with a graph, G=(V,E), where n Moore machines, represented by the vertices V, are interconnected together by an interconnection specified by E and operating under a common clock. The Moore machine model is used to eliminate the possibility of a combinatorial rippling of the output of one machine through a sequence of successor machines. Each edge is labeled by a triplet (u,v,w). The meaning of this triplet is as follows: the given edge originates at vertex u, ends at vertex v, and is labeled with $w \in Z^+$. w identifies the number of registers (delay) a datum originating at u has to pass through before reaching v. Leiserson also proved that this system is systolic if " and only if for each edge (u,v,w) the weight w is greater than 0 [Lei83a]. In essence this condition ensures that in a systolic system, no signal "data" from a node can reach another node without passing through an intermediate buffer. This eliminates the possibility of combinatorial ripples through the nodes of the graph. Two major results which follow from this systolic criterion are listed below.

Sufficient condition: A systolic system represented as a graph ,G, has a systolic equivalent if the constraint graph G - 1 has no negative cycles (G - 1 is obtained from G by decreasing all the labels w of G by 1)

Necessary Condition: A systolic system represented as a graph ,G, does not have any systolic equivalent if the graph G - 1 has a negative cycle and a path from some vertex on this cycle to an input port.

Leiserson and Saxe proved the sufficient and necessary condition, and introduced a procedure to convert a non-systolic graph to a systolic one [LeS81]. Although the conditions of uniformity and locality of communications were not formally introduced in Leiserson work, they are very important from the manufacturing point of view as we explained earlier.

Theree are a large number of university and industrial projects on systolic arrays. One of the early attempts to build a systolic system is the Warp project at Carnegie Mellon University [KuM84]. The Warp consists of an array of ten linearly connected cells. Each cell in the array is capable of performing 10 million 32-bit floating point operations per second (10 MFLOPS)

At Motorola, an effort was made to design an advanced digital signal processing systolic array (DSP) [Lee87]. The goals of this advanced architecture were

- to increase the processing performance by a factor of 16 over that of an existing Motorola DSP;
- to perform 32-bit floating-point arithmetic for application that require this kind of precision; and
- to reduce the amount of effort of software development of this kind of systems.

Motorola based this architecture on processing elements with 32-bit floating point operations. The processing element structure is shown in Figure 1.2. Every processing element is composed of a serial floating-point processor, two input registers ,A and B, and four communication registers. The four communication registers are used to exchange data with the four neighboring processors directly to the east, west, north, or south of this processing element. Each cell has a unique identifying address.

The serial floating-point processor takes 50 clock cycles to perform each operation and can run with a speed of 20 MHz. Since the operation takes 50 clock cycles, non-neighboring communications are possible. During the normally unusable times

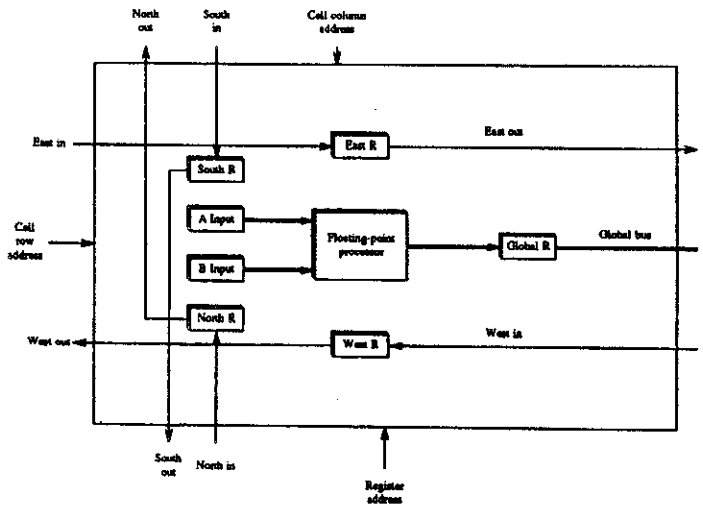


Figure 1.2

The architecture of a Motorola DSP [Lee87]

(the clock cycles), data is allowed to travel through cells transparently using the network of communication registers in each PE, thus allowing a communication between two PE's that are up to 50 cells away. Algorithms are implemented directly in this architecture. Every cell is initialized at least once after each power-up. From this moment on, each PE performs the same operation in every processing cycle. It also gets its input data from a pre-assigned input cell and deposits its results in a pre-assigned output cell. This pre-set operation makes it possible to assign the function of each cell directly from the signal flow graph.

Another example of recent VLSI architecture is the Systolic/Cellular system being designed at Hughes research laboratories [NaP87]. This system consists of a host and a programmable coprocessor. The coprocessor includes an array of 16×16 mesh-connected processors, dual port memory, and a controller with a separate program memory as shown in Figure 1.3. The program and the data are loaded from the host to the program memory and the dual port memory, respectively. There are two modes of operation for this processor, the systolic mode and the cellular mode. In the cellular mode of operation the data are loaded to the processors from the dual-port memory. This mode is useful in convolution, thresholding, FFT, and other transformations. In the systolic mode of operation, the data will flow from the array memory in a row-by-row fashion through the processors, and then back to the array memory using the other port. It was found that this kind of operation is useful in matrix operations like inversion, QR factorization, solution of linear systems of equations, and many other matrix computations.

The processors in Hughes systolic/cellular system are controlled by a single controller, which operates in an SIMD mode. One instruction is broadcast by the controller to be executed by all processors. However, the leftmost processors can be programmed to perform different operations, and some processors may be disabled by using a mask. The instructions set of this array is composed of 30 powerful instructions, each 112 bits wide. Each instruction has two separate fields of operations, one for the leftmost processors, and the other for the internal processors. The performance of this machine is in the neighborhood of 450 MOPS.

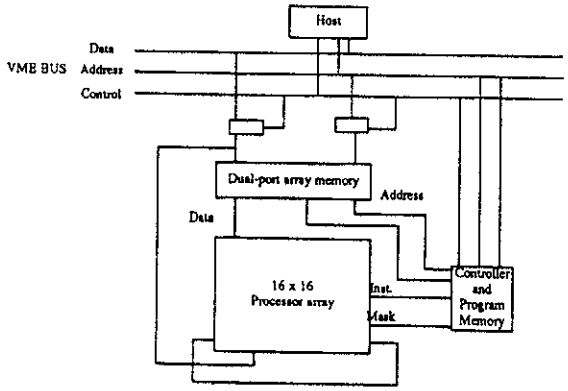


Figure 1.3

Hughes systolic/cellular system for signal processing [NaP87]

Another system is currently being built at ESL Inc. is for beamforming for acoustic signal [Kan87]. The system consists of a special-purpose systolic processor which is attached to a general purpose computer (DEC 11/750)

In the United Kingdom, an effort was made to define "application specific" systolic arrays for high performance signal processing computations. As a result, a systolic array was built that is capable of computing the vector of complex weights required to form the receiving beam for an adaptive antenna array [McM87]. Another major project in the United Kingdom is the building a state-of-the art digital signal processing system, which exploits the systolic array concept at the bit level and is capable of doing convolution, correlation, discrete Fourier transform and rank order filtering in real time.

1.3. Problem Definition and Thesis Objectives

With the new advances in VLSI technology, systolic computations are becoming more and more attractive. If the trend in VLSI technology is to continue, and there are no reasons to believe it will not, systolic arrays will become a major component in any real time application that requires both high speed and high throughput.

In this thesis we are interested in synthesizing systolic arrays. Any synthesis process begins with a high level specification of the problem to be solved. A final implementation is then systematically derived from this specifications. u Earlier work on synthesizing systolic arrays was based on the analysis of the data dependency of such initial specifications. These initial specifications are assumed to be a system of recurrence equations. We will continue to use the system of recurrence equations as an initial specification throughout this thesis. Three problems which have a great impact on the design of systolic arrays are presented in this thesis: 3-D VLSI layout, mapping a recurrence onto a systolic array, and pipelining of systolic arrays. The next three subsections will briefly review the importance of these three topics.

1.3.1. 3-D VLSI Layout.

One of the major objectives in VLSI circuit design is to increase the speed of the circuit. This objective can be reached in two different ways. The first is to increase the switching speed of the transistor. However, there are physical limits for this switching speed Which can not be exceeded. The second is to put more and more transistors on the chip, since the speed of transmitting a signal between two transistors on the same chip is at least one order of magnitude less than transmitting a signal between two chips. When putting more transistors on a single chip, the size of the transistor must be decreased to maintain a reasonable level of yield. Decreasing the transistor size, however, will decrease its driving capabilities, which will limit the maximum wire length on the chip and consequently the number of transistors on the chip. One solution to this problem is to use 3-D VLSI. 3-D VLSI technology is still in its enfancy, and there are some technological difficulties to be overcome before it can be accepted as a standard technology. But, with the work going on now at IBM and Hughes Research Lab, one can expect 3-D VLSI chips to be an available in the near future.

The advantages of using 3-D VLSI layout are two fold. First, as will be proved in Chapter 2, using 3-D VLSI layout will result in less total volume for any circuit than its 2-D counterpart. This will lead to more dense chips. Second, 3-D VLSI layout will result in less maximum wire length, which can lead to a faster circuits.

This thesis will study the possibility of 3-D VLSI, and introduce improved lower and upper bounds for both the volume and the maximum wire length required to layout the different families of graphs.

1.3.2. The Design of a Systolic Arrays.

The second part of this thesis is concerned with the design of a systolic array. Although the use of systolic arrays is increasing, there is as yet no methodology that can map any recurrence into a systolic array. Methodologies presented in current literature are either dedicated to solving one specific problem, or will take an exponential time to solve the problem. This thesis assumes that the problem is presented as a set of recurrence equations and that we want to map it into a square

array of processors. The second part of the thesis, introduces the concept of control flow systolic array. A heuristic methodology for mapping a set of recurrence equations into a two dimensional control flow systolic array is also presented.

We chosed to map the recurrence into a 2-D mesh of processors for two main reasons. The first is that 2-D layout is the predominant technology at the present time. Second, the work presented here for 2-D arrays can be easily extended to support three dimensional meshes when they become available.

1.3.3. Macropipelines of Systolic arrays.

Many applications in image and signal processing requires that more than one systolic computations be performed on the data in a pipeline fashion. Accordingly, output of one systolic array is fed directly to the input of another array. However, the output of one array may not be in the required format for input into the next array in the pipe. The use of a common memory will slow the system and can cause a bottleneck. The last part of this thesis is concerned with the design of converters to convert the data from the output format of one array to the required input format for the next array, in order to directly interface two systolic arrays.

1.4. Organization of the Thesis

The remaining of this thesis is organized into three chapters. Each chapter deals with the interface between Integrated Circuit design and system design. Chapter 2 is the most theoretical, being concerned with the inherent limitations on our ability to compute using VLSI. Chapter 3 deals with the design of the systolic array to solve a specific problem. Chapter 4 deals with the system level, where we are concerning with arranging the systolic arrays in a pipeline fashion to solve a certain problems.

Chapter 2 discusses the 3-D layout of VLSI circuits. In this chapter, we examine the complexities of volume and maximum wire length for mapping circuits represented as undirected graphs to 3-D systems. Tighter bounds than those previously known are shown for various families of graphs in both the one-active-layer and the unrestricted layouts. Finally, a cost model is developed to reflect the cost of

implementation in the third dimension in order to optimize the number of layers to minimize the overall cost.

Chapter 3, discusses the design of a systolic array. One of the most important factors in the design of an array is how to map the algorithm into the array. Several methods have been proposed for mapping the uniform recurrence into a systolic array. In this chapter, the integration of control flow and data flow for mapping nonuniform recurrences into a systolic array is presented and a methodology for integrating the control flow with the data flow is introduced

Chapter 4 discusses the macropipelining of systolic arrays. In a macropipeline of systolic arrays, outputs of one systolic array in a given format have to be fed as inputs to another systolic array in a possibly different format. As mentioned before, a common memory becomes a bottleneck and limits the number of systolic arrays that can be connected together. In this chapter, the designs of buffers to convert data from one format to another is studied. The minimum number of buffers is determined by a dynamic-programming algorithm with $\Theta(n^2)$ computational complexity, where n is the problem size. A general-purpose converter to convert data from any distribution to any other in a subset of the possible data distributions is also proposed. Finally, buffer designs for a macropipeline to perform feature extraction and pattern classification are used to exemplify the design process.

CHAPTER 2
3-D VLSI LAYOUT

2.1. Introduction

The increasing demands for faster processors in scientific as well as commercial computations indicate the need for tremendous computing capacity, in terms of speed and volume. One way to achieve this is to build chips with more active devices. To increase the number of devices in a single chip while maintaining a reasonable yield, the transistor size should be decreased. There exist problems with decreasing transistor size, such as the short-channel effect and the nonstatistical behavior of transistors that span only a few hundred or a few thousand silicon atoms [NuE85]. However, long before these problems become important, the problem of reduced driving capability of smaller transistors will have an equally profound impact on the layout of VLSI chips, since the average wire length grows linearly with the number of transistors. As a result, a limit will be reached at which the size of a transistor cannot be decreased any more without affecting its ability to correctly transmit a signal to another transistor.

Recently, 3-D VLSI circuits have been shown to be feasible. 3-D VLSI circuits are more flexible than their corresponding 2-D counterparts because wire routing is easier and more systematic, the runs of wires are shorter, and the volume of a 3-D realization may be less [Ros83a]. Wise has demonstrated this phenomenon in a two-layer layout of the Banyan/FFT networks; however, his work was directed towards the printed-circuit level rather than the VLSI-chip level [Wis81]. With increased flexibility of device placement in a 3-D circuit, the complexity of the resulting circuit can be reduced. Hence, the driving capability of a transistor and the overall power

requirement can be reduced. Dr. Gibbons, the president of Texas Instruments, predicted the feasibility of such chips in the earlier 1990s [Gib82]. Examples of current implementations include IBM's "modestly" three-dimensional Thermal Conduction Module (TCM) circuit package [BlB82] and Hughes' 32-by-32 3-D cellular computer to be finished in 1987 [NuE85]. Nudd, Etchells, and Grinberg have proposed a cellular machine employing 3-D technology to perform image understanding operations [GrN84] [NuE85].

3-D VLSI technology is still plagued by four major problems. One problem is the alignment of in successive layers of a chip. Another problem is the creation of truly cylindrical holes. Due to effects like diffraction, scattering, and nonuniform exposure to solvents, the holes tend to be accentuated at the top or the bottom [MeC80]. Recent work on X-ray beam and refined optical lithography [Ros83a] has suggested that this issue will be less of a problem in the future. The third problem is that the placement of active devices, using MOS technology, deep inside a 3-D volume requires multiple layers of monocrystalline silicon to be deposited, and subsequent processing of the chip would destroy the crystal structure of the monocrystalline silicon. Recent work at Texas Instruments [LaT80] and IBM [Wei81] has suggested that full layers of the monocrystalline silicon are not needed, and that transistors can be fabricated on islands of monocrystalline silicon that reside on a sea of oxide. The fourth problem is the cooling of such chips. However, the shorter wire lengths in 3-D circuits generates less heat than the corresponding 2-D circuits. Moreover, the problem is less severe in a one-active-layer chip in which active devices exist in only one layer, with the rest of the volume being used for wire routing [Ros83a] [Ros81].

In this chapter, improved bounds on volume and maximum wire length of 3-D layouts will be shown, in both the one-active-layer and unrestricted models. Section 2.2, will present a model for 3-D layouts. In Sections 2.3 and 2.4, tighter lower and upper bounds on volume and maximum wire length will be proposed and ropose layouts for the various families of undirected graphs will be presented. Finally, an optimization model will be presented to minimize the overall cost of the design.

2.2. A Model of 3-D VLSI Circuits

This section, describes the model used to obtain the lower and upper bounds of volume and maximum wire length for mapping various families of undirected graphs in 3-D circuits. Other attempts in this field will be described briefly.

The model used here is an extension of Thompson's 2-D model into three dimensions [Tho80]. The model consists of a 3-D grid of width W, length L, and height H (Figure 2.1). A vertex in this grid, (x,y,z), where $0 \le x \le W$, $0 \le y \le L$, and $0 \le z \le H$, denotes the location where a device may reside. An edge in the grid represents a wire in the circuit. It is assumed that three mutually perpendicular lines in the grid can pass through one point without physically touching each other. As a special case, the traditional 2-D circuit with two levels of metalization can be considered as a one-layer 3-D circuit because the two levels of metallic conductor can cross without touching each other. It is further assumed that any active device will require a unit volume, that the cross section of any wire is a unit area, and that the separation between the wires in any direction is of unit length. These assumptions are not ooverly restrictive as it is the order-of-magnitude asymptotic complexities that are under consideration.

To find the upper bounds of volume and maximum wire length for mapping an undirected graph G=(V, E), where V is a set of vertices and E is a set of edges connecting the vertices, it is necessary to find a one-to-one mapping between the set of vertices of the graph and the set of nodes of the grid and, at the same time, a one-to-one mapping between the set of edges in the graph and the set of disjoint paths of the grid. The volume of the layout is the minimum volume of a parallelepiped containing the layout, while the maximum wire length is the maximum lwngth of a wire between any two active devices.

Rosenberg has proposed two models of 3-D layouts [Ros81] [Ros83a]. The first model is the one-active-layer model, in which active devices are allowed to reside in either the top or the bottom layer, with the other layers used for the routing of wires. The second model is the unrestricted model in which active devices can be placed anywhere in the volume. In general, the one-active-layer model requires more volume and longer wires than the unrestricted model.

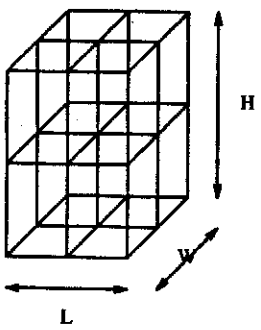


Figure 2.1

A 3-D grid

Rosenberg proved that there is an unrestricted 3-D realization of the n-input rearrangeable permutation network that consumes $\Theta(n^{3/2})$ volume, and that there is a one-active-layer 3-D realization of the same network with $\Theta(n^{3/2}\log n)$ volume* [Ros81] [Ros83a]. Preparata has proposed a layout for the cube-connected-cycles using the unrestricted model. Preparata also developed upper and lower bounds using the $VT^{3/2}$ and VT measures, where V is the volume, and T is the computation time [Pre83a]. He pointed out that the $VT^{3/2}$ measure is suitable for the unrestricted layout, while the VT measure is suitable for the one-active-layer layout. Leighton and Rosenberg have found lower and upper bounds for the layout of various families of undirected graphs [LeR83a] [LeR86].

In the next two sections, improved lower and upper bounds to map an undirected graph to a 3-D grid for the one-active-layer and the unrestricted layouts will be presented and compared to previous results. The undirected graphs considered are classified into families characterized by their separators, which define the relationship between the area or volume of layout and the connectivity of the graph. An N-node graph G is said to have f(N) separator if (a) G can be partitioned into two graphs, each with N/2 nodes, by cutting no more than f(N) edges; and (b) both of the two N/2-node subgraphs have f(N/2) separators. Lipton and Tarjan proved that any N-node planar graph has $O(\sqrt{N})$ separator [LiT77a] [LiT777b]. Since the result discussed above is an upper bound, it is possible for planar graphs to have separators less than $\Theta(\sqrt{N})$ and possible for non-planar graphs to have $\Theta(\sqrt{N})$ separators. The relationship between the separator and the corresponding area of layout was first observed by Thompson [Tho80], who showed that the lower-bound area lay out a graph with separator ω is $\Omega(\omega^2)$. Leighton obtained lower and upper bounds on the area and maximum wire length for layouts of various families of graphs with $\Theta(N^q)$ separator, where $q<1/2$, $q=1/2$, and $q>1/2$, and the family of planar graphs using 2-D technologies [Lei84].

* Θ indicates the set of functions with the same order-of-magnitude complexity; O indicates the set of functions with the upper-bound order-of-magnitude complexity; Ω indicates the set of functions with the lower-bound order-of-magnitude complexity.

2.3. Lower Bounds on Volume and Maximum Wire Length

In this section, improved lower bounds on the volume and maximum wire length are developed for the various families of undirected graphs in the one-active-layer model. To prove the lower bounds on embedding the various families of undirected graphs in a 3-D grid, it is necessary to find a representative graph in each family such that this graph will have the greatest lower bound. No improvement in lower bounds for the unrestricted model was found because the existing lower bounds on volume are equal to the improved upper bounds (Section 2.2), except for graphs with $\Theta(N^{2/3})$ separator, and hence are already tight. Table 2.1 summarizes the existing lower bounds on volume and maximum wire length for the unrestricted model [Ros83a] [LeR86] [LeR84].

The following theorem gives the plane area required when a 3-D circuit to be converted into a 2-D circuit. This theorem is an improvement over the one proved by Leighton and Rosenberg [LeR86], showing that the area required is $4BH^2$ instead of $9BH^2$ and that the degree of the graph can be six instead of four.

Theorem 1: Any 3-D layout of volume V, base area B, and height H can be transformed into a 2-D layout of area $A=4BH^2$. If the maximum wire length in the 3-D layout is W_{3d}, then the maximum wire length in the 2-D layout is $W_{2d} \le 2\cdot\max(H, 3)\cdot W_{3d}$.

Proof: Without loss of generality, a 3-D grid of base area $B=WL$ and height H will be transformed into a 2-D grid of area $A=4WLH^2$. Consider the 3-D grid in Figure 2.2a. Assume that the nodes of this grid are located in the Cartesian coordinates (x,y,z), where $0\le x<W$, $0\le y<L$, $0\le z<H$. Point (x,y,z) in the 3-D grid is mapped to point (x′,y′) in the 2-D grid such that

$$x' = Hx + z \; ; \qquad y' = Hy + z \tag{2.1}$$

Note that the width and length of the 2-D grid are $W'=HW$ and $L'=HL$, respectively.

Figure 2.2b shows the mapping of the 3-by-3-by-3 grid into a 9-by-9 grid, where solid lines represent connections in the first plane in Figure 2.2a, dashed lines represent connections in the second plane, dotted lines represent connections in the

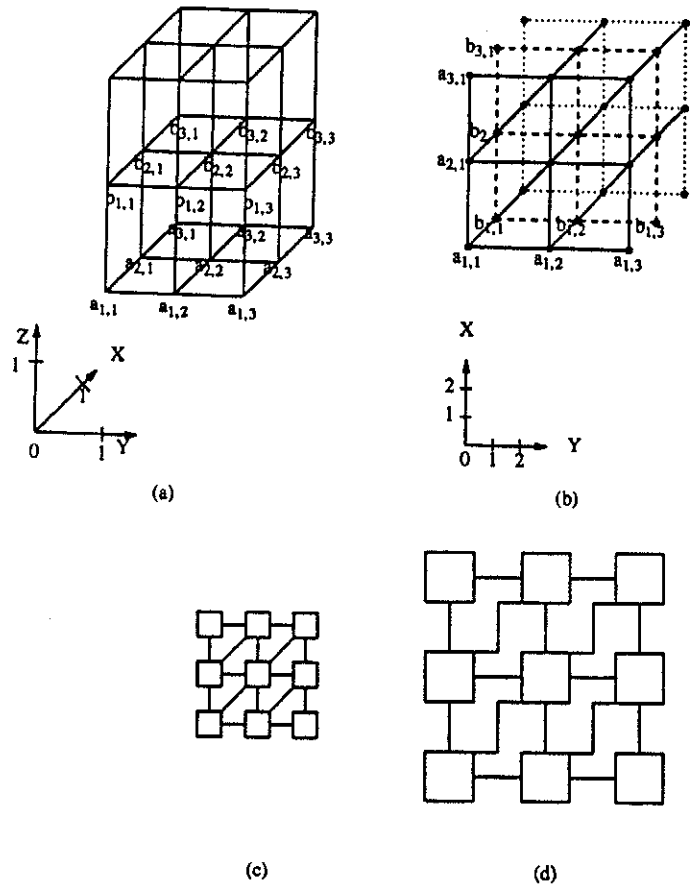


Figure 2.2

The mapping of a 3-by-3-by-3 grid into a 9-by-9 grid. (a) a 3-D grid; (b) a mapping of the 3-D grid in (a) to a 2-D grid; (c) a section of the 2-D grid with diagonal connections; (d) an expanded section of the 2-D grid without diagonal connections.

Table 2.1. Lower bounds on volume and maximum wire length for the unrestricted model. (Note that a lower bound is intended to mean the largest known lower bound for a graph in the given family.)

Graph Separator f(N)	Volume	Maximum Wire Length
$\Theta(N^q)$, $0 \le q \le 2/3$	$\Omega(N)$ [Ros83a], [LeR86],[LeR84]	$\Omega(N^{1/3}/\log N)$ [Ros83a]
Planar	$\Omega(N)$ [Ros83a], [LeR86],[LeR84]	$\Omega(N^{1/3}/\log N)$ [Ros83a]
$\Theta(N^q)$, $2/3 < q \le 1$	$\Omega(N^{3q/2})$ [Ros83a]	$\Omega(N^{q/2}/\log N)$ [Ros83a]

third plane, and diagonal lines represent connections across different planes. The effect of this mapping is that two nodes in a straight line in the x or y direction in the 3-D grid are mapped into two nodes in a straight line in the same direction in the 2-D grid, but the distance between them is multiplied by H. For example, $a_{1,1}$ and $a_{1,3}$ are separated by a distance of two units and are in the x direction in Figure 2.2a. These two points are separated by a distance of six units (H=3) and are also in the x direction in Figure 2.2b. Note that nodes in a straight line in the z direction are mapped to nodes in a straight line in the diagonal direction in the 2-D plane. Since most models in 2-D VLSI layouts do not allow connections in the diagonal direction, the problem can be circumvented by multiplying the area by four and mapping diagonal

connections to a sequence of horizontal and vertical connections. Figure 2.2c shows a generic node connected to its six neighbors, where the distance between adjacent nodes in the x or y direction is unity. Figure 2.2d shows the same set of nodes after doubling the distance between two nodes and quadrupling the area of each node, hence, multiplying the total area by four. Here, a diagonal connection is altered to be a horizontal segment followed by a vertical segment and finally a horizontal segment. As a result, the area is

$$A = 4(HW)\cdot(HL) = 4(WLH^2) \tag{2.2}$$

To prove the result on the maximum wire length, note that a wire connecting any two nodes in the 3-D layout is composed of wires running in the x, y, and z directions. The length of wires in the x or y direction is multiplied by 2H during the transformation, while the length of wires in the z direction is multiplied by a constant less than six. It is straightforward to show that

$$W_{2d} \le 2\cdot\max(H, 3)\cdot W_{3d}\,, \tag{2.3}$$

which proves the theorem. □

In the following theorem, the mesh of trees is considered as an example in the family of graphs with $\Theta(\sqrt{N})$ separator, while the tree of meshes is considered as an example in the family of planar graphs. These two example graphs were used by Leighton in proving the lower bound of bothe area and maximum wire length in 2-D layouts [Lei81].

The *mesh of trees* is defined as follows [Lei84] [Lei81]. Starting with an n-by-n matrix of nodes (n is assumed to be a power of 2) and adding nodes wherever necessary, a complete binary tree is constructed using nodes in each row and column of the matrix as leaves. Hence, each node in the mesh is a leaf of two orthogonal binary trees, one for the binary tree encompassing nodes in the row containing this node and another for the tree encompassing nodes in the column. (Orthogonal trees is another name for the mesh of trees.) An example of the mesh of trees is shown in Figure 2.3.

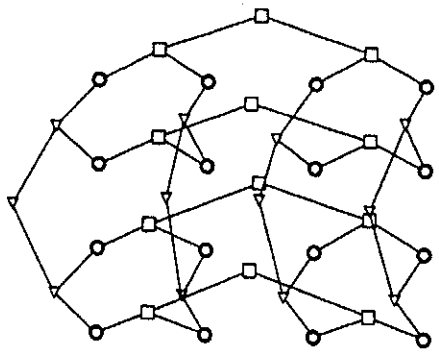


Figure 2.3
A 4-by-4 mesh of trees

The *tree of meshes* is defined as follows [Lei81] [Lei84]. In a complete binary tree, each node is replaced by a mesh and each edge by several edges that connect the meshes together. The root is replaced by an n-by-n mesh (n is assumed to be a power of 2). Its children are replaced by n/2-by-n meshes. These meshes' children are replaced by n/2-by-n/2 meshes. This continues until the leaves of the original binary tree are replaced by 1-by-1 meshes. Figure 2.4 shows a 4-by-4 tree of meshes.

Theorem 2: (a) Any 3-D one-active-layer layout of the mesh of trees will require $\Omega(N \log N)$ volume and $\Omega(\sqrt{N}/\log\log N)$ maximum wire length. (b) Any 3-D one-active-layer layout of the tree of meshes will require $\Omega(N\sqrt{\log N})$ volume and $\Omega(\sqrt{N}/\log N)$ maximum wire length.

Proof: These lower bounds will be proved by contradiction. Leighton proved that any 2-D layout of the N-node mesh of trees will require $\Omega(N\log^2 N)$ area, and that this layout must have $\Omega(\sqrt{N}\log N/\log\log N)$ maximum wire length [Lei84]. He also proved that any 2-D layout of the N-node tree of meshes will require $\Omega(N\log N)$ area, and that this layout must have $\Omega(\sqrt{N}/\sqrt{\log N})$ maximum wire length.

For the mesh of trees, assume the existence of a 3-D one-active-layer layout with a volume $V<\Theta(N\log N)$ and maximum wire length $W<\Theta(\sqrt{N}/\log\log N)$. The base area of this one-active-layer layout should be $\Omega(N)$, as it should be large enough to accommodate the N nodes of the graph. Therefore, the height of this layout is $H<\Theta(\log N)$. According to Theorem 1, this 3-D layout can be transformed into a 2-D layout with area $A<\Theta(N\log^2 N)$ and maximum wire length $W_{2d}<\Theta(\sqrt{N}\log N/\log\log N)$, which contradict Leighton's results [Lei84]. Thus, any 3-D one-active-layer layout of the mesh of trees will require $\Omega(N\log N)$ volume and $\Omega(\sqrt{N}/\log\log N)$ maximum wire length.

For the tree of meshes, assume the existence of a 3-D one-active-layer layout with volume $V<\Theta(N\sqrt{\log N})$ and maximum wire length $W<\Theta(\sqrt{N}/\log N)$. Since the base of this layout should have $\Omega(N)$ area, the height of this layout is $H<\Theta(\sqrt{\log N})$. Using Theorem 1, this layout can be transformed into a 2-D layout with area $A<\Theta(N\log N)$ and maximum wire length $W_{2d}<\Theta(\sqrt{N}/\sqrt{\log N})$, which contradict

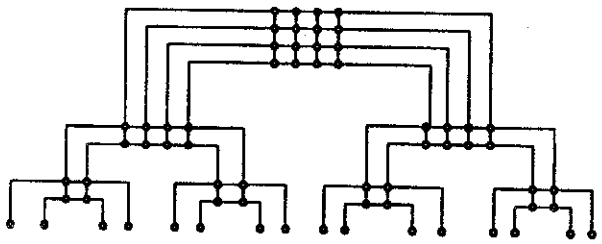


Figure 2.4.

A 4-by-4 Tree of meshs

Leighton's results. Hence, any 3-D one-active-layer layout of the trees of meshes should have $\Omega(N\sqrt{\log N})$ volume and $\Omega(\sqrt{N}/\log N)$ maximum wire length. □

The upper-bound volume of the family of undirected graphs with $\Theta(N^q)$ separator, $1/2<q\leq 1$, will be shown in Section 2.4 to be $O(N^{q+1/2})$. As the base area has $\Omega(N)$ complexity, the height of this layout has $O(N^{q-1/2})$ complexity. Leighton has proved the lower-bound maximum wire length for the family of undirected graphs with $\Theta(N^q)$ separator, $1/2<q\leq 1$, in a 2-D layout to be $\Omega(N^q)$ [Lei84]. From Theorem 1, the maximum wire length in the 3-D one-active-layer layout should be $W_{2d}/H = \Omega(N^q/N^{q-1/2}) = \Omega(\sqrt{N})$.

For the family of graphs with $\Theta(N^q)$, separator $0\leq q<1/2$, Paterson, Ruzzo, and Snyder have proved the lower-bound maximum wire length in a 2-D layout of a binary tree to be $\Omega(\sqrt{N}/\log N)$ [PaR81]. In a one-active-layer 3-D layout, a similar argument can be made such that nodes of a binary tree are in one layer, and that the maximum distance between two nodes separated by $2\log N$ edges is $\Omega(\sqrt{N})$. Hence, the lower bound in the 3-D case is the same as that of the 2-D case.

Table 2.2 summarizes the lower bounds obtained for the one-active-layer layout and compares them with previous results.

2.4. Upper Bounds

Before introducing the results on upper bounds, some of the mathematical background behind the theory of layouts should be reviewed. Thompson introduced the idea of the minimum bisection width of an undirected graph and proved a relation between the minimum bisection width and the minimum area required to lay out the given graph [Tho80]. Lipton and Tarjan introduced the idea of separator for a family of undirected graphs and proved that the family of planar graphs has an $O(\sqrt{N})$ separator [LiT77]. They also proposed a linear time algorithm to compute this separator. Bhatt and Leighton introduced the ideas of bifurcators and decomposition trees [BhL84]. An N-node undirected graph has (F,α) bifurcator if it can be decomposed into two subgraphs, G_1 and G_2, by removing no more than F edges. Both G_1 and G_2

Table 2.2.
Lower bounds on volume and maximum wire length for the one-active-layer model. (Note that a lower bound is intended to mean the largest known lower bound for a graph in the given family.)

Graph Separator f(N)	Volume		Maximum Wire Length	
	Previous	New	Previous	New
$\Theta(N^q)$, $0 \le q < 1/2$	$\Omega(N)$ [LeR86]	$\Omega(N)$ [LeR86]	constant [LeR86]	$\Omega(\sqrt{N}/\log N)$ [PaR81]
Planar	$\Omega(N)$ [LeR86]	$\Omega(N\sqrt{\log N})$	constant [LeR86]	$\Omega(\sqrt{N}/\log N)$
$\Theta(\sqrt{N})$	$\Omega(N)$ [LeR86]	$\Omega(N \log N)$	constant [LeR86]]	$\Omega(\sqrt{N}/\log\log N)$
$\Theta(N^q)$, $1/2 < q \le 1$	$\Omega(N^{q+1/2})$ [LeR86] [Ros83]	$\Omega(N^{q+1/2})$	$\Omega(N^{q-1/2})$ [LeR86]	$\Omega(\sqrt{N})$

can further be decomposed into two subgraphs by removing no more than F/α edges. In general, any subgraph in level i can be decomposed into two subgraphs by removing no more than F/α^i edges. This decomposition can be represented by the decomposition tree in Figure 2.5.

A decomposition tree is said to be a fully balanced decomposition tree if

(1) when decomposing any subgraph into two smaller subgraphs, the number of nodes in the two smaller subgraphs are equal; and

(2) when decomposing any subgraph into two smaller subgraphs, the number of edges connecting the subgraph to the rest of the original graph is divided into two equal sets that are distributed in the two decomposed subgraphs.

Bhatt and Leighton also proved that any graph with a (F,α) bifurcator has a fully balanced decomposition tree with a (F',α) bifurcator, where F' is related to F by a constant. Leighton showed that if $F=N^q$, then the total number of edges connecting any subgraph with $N/2^i$ nodes in level i of the decomposition tree to the rest of the original graph is $k(N/2^i)^q$, where k is a constant [Lei83b]. As a result, a graph with N^q separator has a $(N^q, 2^q)$ bifurcator.

In the rest of this section, upper bounds on volume and maximum wire length for the lauoyt of the various families of undirected graphs in both the one-active-layer and the unrestricted models will be developed.

2.4.1. One-Active-Layer Layouts

The following theorems prove the upper bounds on volume and maximum wire length for the layout of the various families of undirected graphs in the one-active-layer 3-D layouts. The family of planar graphs is treated in the same way as the family of graphs with $\Theta(\sqrt{N})$ separator.

In the following theorem, the upper bounds for graphs with $\Theta(\sqrt{N})$ separator are proved. Although Leighton and Rosenberg have proved the same bounds before, they have assumed in their proof the existence of a layout of an n-node subgraph in which the ports of this subgraph "are sufficiently sparse that the routing is guaranteed to be possible" [LeR86]. We will assume in the following proof that the ports of a

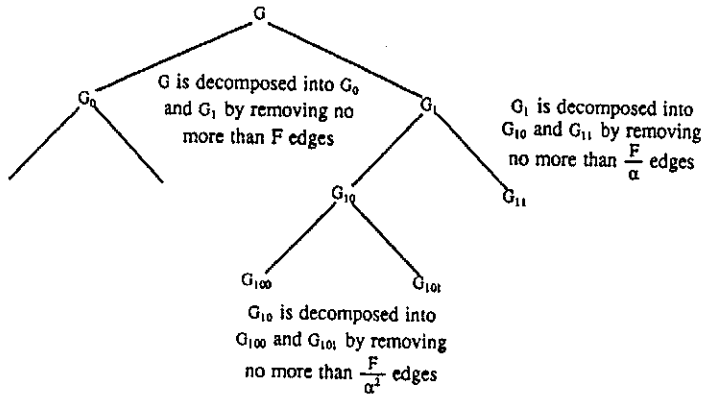


Figure 2.5.

Decomposition tree for a graph with (F,α) bifurcator.

subgraph are equally spaced along one side of its layout. In connecting two n-node subgraphs into a 2n-node subgraph, a complete crossbar switch will be used to perform the routing, and the ports in the resulting subgraph will also be equally spaced along one side of the resulting layout. The above model allows a better upper bound on volume for the family of graphs with $\Theta(N^q)$ separator to be proved. This will be shown in Theorem 4.

Theorem 3: Any undirected graph with $\Theta(\sqrt{N})$ separator has a 3-D one-active-layer layout with $O(N\log N)$ volume and $O(\sqrt{N})$ maximum wire length.

Proof: Assume that the balanced decomposition tree of the graph is known. The proof is by induction on a graph with n nodes. The case for n=1 is trivial. For the induction hypothesis, assume that an n-node graph can be mapped into a parallelepiped with volume V(n), height H(n), and a square base of side $L(n)=kc\sqrt{n}$, where k is a constant. It is further assumed that the $c\sqrt{n}$ ports to connect any node in this subgraph to another node outside this subgraph are aligned and equally spaced along one side of the top layer of this layout (see Figure 2.6, where the ports are represented by circles). In the induction step, consider the volume needed to lay out four n-node subgraphs. We will combine these four layouts to produce one 4n-node layout with volume V(4n), height H(4n), a square base of side $L(4n)=kc\sqrt{4n}$, and that the $c\sqrt{4n}$ ports of the 4n-node subgraph are aligned and equally spaced along one side of the top layer. This will be done by first showing that one additional layer is needed to accommodate the necessary interconnections when two n-node subgraph layouts are combined to form one 2n-node subgraph layout.

Consider two n-node layouts placed side by side as shown in Figure 2.6. Figure 2.7 shows the additional top layer that is created when the two n-node subgraph layouts are combined. It is necessary to (a) create $c\sqrt{2n}$ ports in the 2n-node subgraph layout; and (b) connect a maximum of $c\sqrt{n}$ ports in one of the n-node subgraphs to a maximum of $c\sqrt{n}$ ports in the other n-node subgraph. Since it has been assumed that the subgraph has a balanced decomposition tree, half of the $c\sqrt{2n}$ ports in the combined layout will be connected to ports in the first layout, while the other half will be

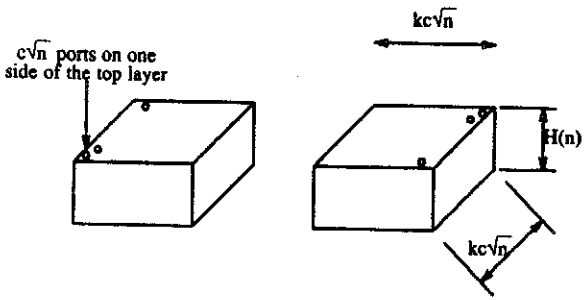


Figure 2.6.

Two n-node subgraph layouts with $\theta(\sqrt{n})$ separator. (The $c\sqrt{n}$ ports of each layout are represented as circles and are aligned on one side of the top layer.)

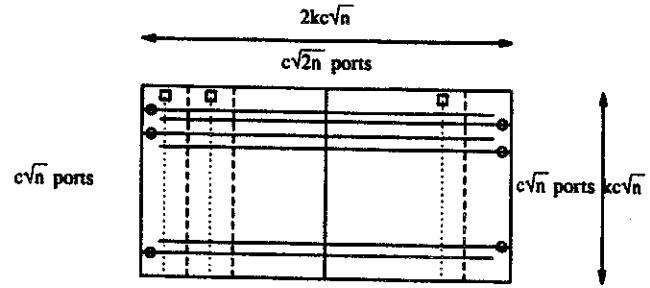


Figure 2.7

New layer on top is created when two n-node subgraph layouts are combined to form one 2n-node subgraph layout in the one-active-layer model. (Circles represent the $c\sqrt{n}$ ports in each n-node layout. Squares represent the $c\sqrt{2n}$ ports in the combined layout.)

connected to ports in the second layout.

In Figure 2.7, the $c\sqrt{n}$ ports of the two n-node subgraphs are represented by $c\sqrt{n}$ circles along the sides. The newly created $c\sqrt{2n}$ ports of the 2n-node subgraph are represented by squares equally spaced along the top side. The objective is to route the $c\sqrt{2n}$ ports in the combined layout to ports in the two original layouts and to connect the $2c\sqrt{n}$ ports in the original layouts together. This can be done by creating a track out of each port in the original layout and extending it across the top layer in the new layout. These tracks are represented by the solid horizontal lines in Figure 2.7. Tracks are also created for each of the $c\sqrt{2n}$ ports in the combined layout and extended across the top layer. These tracks are shown by the dotted vertical lines in Figure 2.7. These horizontal and vertical tracks allow us to form a complete crossbar switch that connects any port in the two n-node layouts to any port in the combined layout. To connect the $c\sqrt{n}$ ports in one of the n-node layouts to the $c\sqrt{n}$ ports in the other n-node layout, a maximum of $c\sqrt{n}$ vertical tracks (represented as dashed lines in Figure 2.7 are created) to form a complete crossbar switch. The above construction process is feasible for $k \geq 2$ in the induction hypothesis because the number of horizontal tracks is $2c\sqrt{n}$, which is less than $kc\sqrt{n}$, and the maximum number of vertical tracks is $(c\sqrt{2n} + c\sqrt{n})$, which is less $2kc\sqrt{n}$.

In combining two n-node subgraph layouts to form one 2n-node subgraph layout, an additional layer is needed. The number of layers in the resulting layout can be computed from the following recurrence.

$$H(2n) = \begin{cases} H(n) + 1 & n > 1 \\ 1 & n \leq 1 \end{cases} \tag{2.4}$$

Similarly, we can combine two 2n-node subgraph layouts to form one 4n-node subgraph layout. In general, for an N-node subgraph layout, where N is a power of two,

$$H(N) = \log N \tag{2.5}$$

Since the base area of an N-node layout is $(kc\sqrt{N})^2$, the total volume will be

$$V_{1AL}(N) = k^2c^2N\log N = O(N\log N) \tag{2.6}$$

In computing the volume, no constraint is put on the routing of wires, hence, a wire can run along the $\log N$ layers in a zig-zag fashion in the worst case. The maximum wire length is

$$W_{1AL}(N) = O\left(2\sum_{i=0}^{\log_2 N} kc\sqrt{\frac{N}{2^i}}\right) = O(\sqrt{N}) \tag{2.7}$$

According to the theory of induction, the theorem is proved. □

Theorem 4: Any undirected graph with $\Theta(N^q)$ separator, $1/2<q\leq 1$, has a 3-D one-active-layer layout with $O(N^{q+1/2})$ volume and $O(\sqrt{N})$ maximum wire length.

Proof: The proof is similar to that of Theorem 3 except that cn^q edges connect any n-node subgraph to the rest of the graph. Assume that there are two n-node subgraph layouts, each in the form of a parallelepiped with height H(n), square base with side $kc\sqrt{n}$, and that the cn^q ports are arranged in the form of a rectangle of width $n^{q-1/2}$ and length $c\sqrt{n}$ in the top layer (see Figure 2.8). In forming a 2n-node layout, $c(2n)^q$ new ports must be created and routed to a maximum of $2cn^q$ ports of the two n-node layouts, and the ports of the two n-node layouts must be connected together in a similar way as in the proof of Theorem 3. To form a complete 3-D crossbar switch to perform the routing between the $c(2n)^q$ ports in the combined 2n-node layout and the corresponding ports in the two n-node layouts. $n^{q-1/2}$ layers are necessary, another $n^{q-1/2}$ layers are needed to form a complete 3-D crossbar switch to connect the ports in the two n-node layouts.

The height of the layout can be computed from the following recurrence.

$$H(2n) = \begin{cases} H(n) + 2n^{q-1/2} & n > 1 \\ 1 & n \leq 1 \end{cases} \tag{2.8}$$

In general, for an N-node layout, where N is a power of 2,

$$H(N) = \sum_{i=1}^{\log_2 N} 2\left(\frac{N}{2^i}\right)^{q-1/2} = O(N^{q-1/2}) \tag{2.9}$$

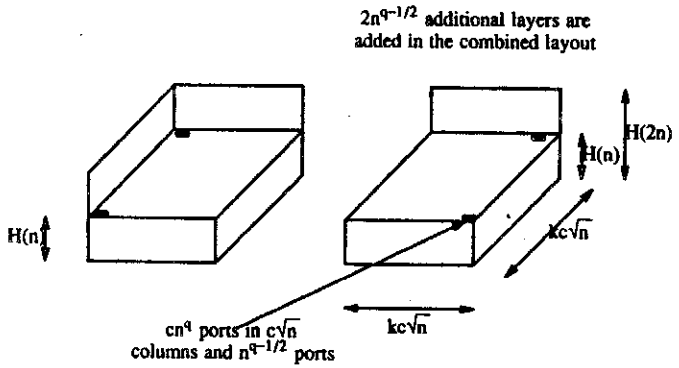


Figure 2.8.

Two n-node-subgraph layouts with $\Theta(N^q)$, $1/2<q\leq 1$, separator.

Since the base area of the N-node layout is $(kc\sqrt{N})^2$, the total volume is

$$V_{1AL}(N) = O(N \cdot N^{q-1/2}) = O(N^{q+1/2}) \tag{2.10}$$

Using the same argument as in the family of graphs with $\Theta(\sqrt{N})$ separator, the maximum wire length is

$$W_{1AL} = O(\sqrt{N}) \tag{2.11}$$

Note that the upper bounds on volume and maximum wire length are optimal because they are the same as the corresponding lower bounds (see Table 2.2). □

Table 2.3 shows the upper bounds in the one-active-layer model and compares them with previous results [LeR84] [LeR86]. Note that the upper bounds on volume are tight in all cases except for the family of planar graphs.

2.4.2. Unrestricted Layouts

The next two theorems give the complexities of layouts in the 3-D unrestricted model in which devices can be placed anywhere in the 3-D volume.

Theorem 5: Any complete binary tree with N nodes has a 3-D unrestricted layout with $V(N) = O(N)$ and a maximum wire length $W(N) = O\left(\frac{N^{1/3}}{\log N}\right)$.

Proof: Figure 2.9 shows a 3-D H-layout tree, which is a direct extension of the 2-D H-layout tree, with a volume $V(N) = O(N)$, and a maximum wire length $W(N) = N^{1/3}$, where N is the number of nodes in this tree. Notice that the layout has the shape of a cube, and the maximum wire length is half the sidelength of the cube. The technique used in this proof is similar to the 2-D minimax wirelength for the tree of meshs [CzR88] [Ram82].

Given a 3-D H-layout of a tree of height $3\log\log N^{1/3}$, the number of nodes in this tree is $2^{1+3\log\log N^{1/3}} = 2\log^3 N^{1/3}$ nodes, and the volume of this layout is $V = O(\log^3 N^{1/3})$. In the previous layout (3-D H-layout of a tree of height $3\log\log N^{1/3}$ each linear dimension is expanded with a factor β, where $\beta = \frac{\alpha N^{1/3}}{\log N^{1/3}}$ α

Table 2.3.
Upper-bound volume and maximum wire length for the 3-D one-active-layer layouts. (The previous and new results may be the same but obtained by different methods.)

Graph Separator f(N)	Volume		Maximum Wire Length	
	Previous	New	Previous	New
$\Theta(N^q)$, $0 \le q < 1/2$	$O(N)$ [LeR86]	$O(N)$ [LeR86]	$O(\sqrt{N}/\log N)$ [LeR86]	$O(\sqrt{N}/\log N)$ [LeR86]
Planar	$O(N \log N)$ [LeR86]	$O(N \log N)$	$O(\sqrt{N})$ [LeR86]	$O(\sqrt{N})$
$\Theta(\sqrt{N})$	$O(N \log N)$ [LeR86]	$O(N \log N)$	$O(\sqrt{N})$ [LeR86]	$O(\sqrt{N})$
$\Theta(N^q)$, $1/2 < q \le 1$	$O(N^{q+1/2} \log N)$ [LeR86]	$O(N^{q+1/2})$	$O(\sqrt{N})$ [LeR86]	$O(\sqrt{N})$

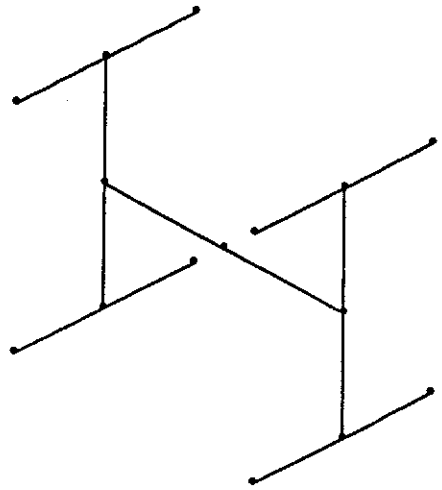


Figure 2.9

3-D H-layout of a binary tree

is a constant. Each point in this layout is expanded into a cube with side β and volume β^3, and every edge is expanded into a channel of side length β and height β. A tree of height logN will be embedded into this expanded H-layout. First, the root of this tree is embedded in the center of the layout, with the second level nodes on either side and the nodes in the top levels stacked together in a breadth-first manner. As shown in Figure 2.10, nodes in successive levels are equally spaced at a distance $\frac{N^{1/3}}{\log N}$ apart. Assume that k levels of the tree are embedded in this manner.

It is necessary to ensure that every channel is wide enough to accommodate the nodes stacked in it. To do this, assume that all the nodes in the first k levels are stacked together in the same channel. This is an over-estimation, but suffices for this proof. In the first k levels, there are 2×2^k nodes. it will be proved that this number is smaller than the channel side.

$$2^{1+k} \le \beta \tag{2.12}$$

substituting the value of β we get

$$k \le \log N^{1/3} + \log\alpha - 1 - \log\log N^{1/3} \tag{2.13}$$

Solving for k we get

$$k = c\log N^{1/3} \qquad c \le 1 \tag{2.14}$$

The rest of the levels ($\log N - c\log N^{1/3}$) still need to be embedded. This is done in the leaves of the original H-layout, which became cubes of side β. The condition to be satisfied is that the total volume of these cubes is not smaller than the number of the rest of the nodes (in the H-layout, O(N) volume is needed for layout of tree with O(N) nodes).

$$\beta \times \log^3 N^{1/3} \ge 2^{c\log N^{1/3}} \times 2^{\log N - c\log N^{1/3}} \tag{2.15}$$

$$\frac{\alpha^{1/3} N}{\log N^3 N^{1/3}} \times \log^3 N^{1/3} = N \tag{2.16}$$

which is satisfied for $\alpha \ge 1$.

The total volume of the layout is $\beta^3 \times \log^3 N^{1/3} = \alpha^3 N$. The maximum wire length in

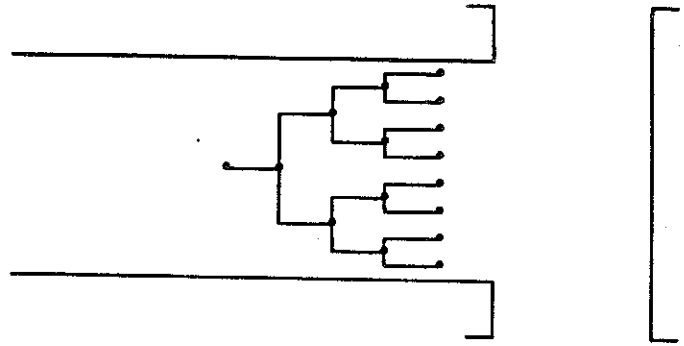


Figure 2.10.

A cross section in the 3-D H-layout of a binary tree

the first stage is bounded by β, and in the second stage by the sidelength of the cube, which is β. The maximum wire length $W(N) \le O\left(\frac{N^{1/3}}{\log N}\right)$ □

Theorem 6: Any undirected graph with $\Theta(N^q)$ separator, $0 \le q \le 1$, has a 3-D unrestricted layout with volume

$$V(N) = \begin{cases} O(N) & 0 \le q < 2/3 \\ O(N \log^3 N) & q = 2/3 \\ O(N^{3q/2}) & 2/3 < q \le 1 \end{cases} \quad (2.17)$$

with maximum wire length

$$W(N) = \begin{cases} O(N^{1/3}) & 0 \le q < 2/3 \\ O(N^{1/3} \log N) & q = 2/3 \\ O(N^{q/2}) & 2/3 < q \le 1 \end{cases} \quad (2.18)$$

Proof: Assume that the balanced decomposition tree of the graph is known. The proof is by induction on a graph with n nodes. The case for n=1 is trivial. For the induction hypothesis, assume that an n-node layout is in the form of a cube. Further, assume that the cn^q ports of this layout are arranged in the form of a square with side $k\sqrt{cn^q}$ in one of the faces of the cube, where k is a constant. In the induction step, it will be shown that eight n-node subgraphs can be combined into one 8n-node layout in the form of a cube, with the $c(8n)^q$ ports of this layout arranged in the form of a square of side $k\sqrt{c(8n)^q}$.

The induction step is proved by first arranging the eight n-node layouts in the corners of a larger cube, such that ports of the four upper cubes are directed downwards, while ports of the four lower cubes are directed upwards (see Figure 2.11). First two n-node subgraph are combined layouts to form one 2n-node layout. It is necessary to (a) create $c(2n)^q$ new ports for the 2n-node layout, and (b) connect the ports in the two n-node layouts. Figure 2.12 shows the $2(cn^q)$ ports of the upper and lower layouts, each in the form of a square with side $k\sqrt{cn^q}$, where k is a constant. By adding $2^q k\sqrt{cn^q}$ layers between the upper and lower cubes in Figure 2.12, a

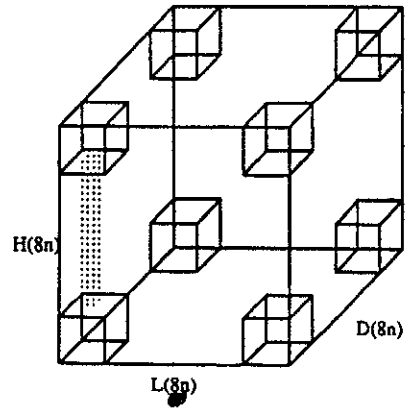


Figure 2.11

Combining eight n-node layouts to form one 8n-node layout.

complete 3-D crossbar switch can be created to perform the necessary routing between the newly created $c(2n)^q$ ports and any of the $2cn^q$ ports in the two original n-node layouts. The newly created $c(2n)^q$ ports are arranged in the form of a rectangle of length $k\sqrt{cn^q}$ and width $2^q k\sqrt{cn^q}$. To perform the necessary connections between the two n-node subgraph layouts, each with a maximum of cn^q ports, another complete 3-D crossbar switch with $k\sqrt{cn^q}$ layers is created between the upper and lower layouts, as shown in Figure 2.12.

In a similar way, four 2n-node layouts can be combined to form two 4n-node layouts, and two 4n-node layouts can be combined into one 8n-node layout. In each case, $\Theta(\sqrt{cn^q})$ layers have been addeed between the two layouts concerned. The height, length, and width of the 8n-node layout can be computed from the following recurrences.

$$H(8n) = \begin{cases} 2H(n) + k_1\sqrt{cn^q} & n>1 \\ 1 & n=1 \end{cases} \qquad (2.19)$$

$$L(8n) = \begin{cases} 2L(n) + k_2\sqrt{cn^q} & n>1 \\ 1 & n=1 \end{cases} \qquad (2.20)$$

$$D(8n) = \begin{cases} 2D(n) + k_3\sqrt{cn^q} & n>1 \\ 1 & n=1 \end{cases} \qquad (2.21)$$

where k_1, k_2, and k_3 are constants. Solving these equations,

$$D(N) = L(N) = H(N) = \begin{cases} O(N^{1/3}) & 0\le q<2/3 \\ O(N^{1/3}\log N) & q=2/3 \\ O(N^{q/2}) & 2/3<q\le 1. \end{cases} \qquad (2.22)$$

The volume of the layout will be

$$V(N) = \begin{cases} O(N) & 0\le q<2/3 \\ O(N\log^3 N) & q=2/3 \\ O(N^{3q/2}) & 2/3<q\le 1. \end{cases} \qquad (2.23)$$

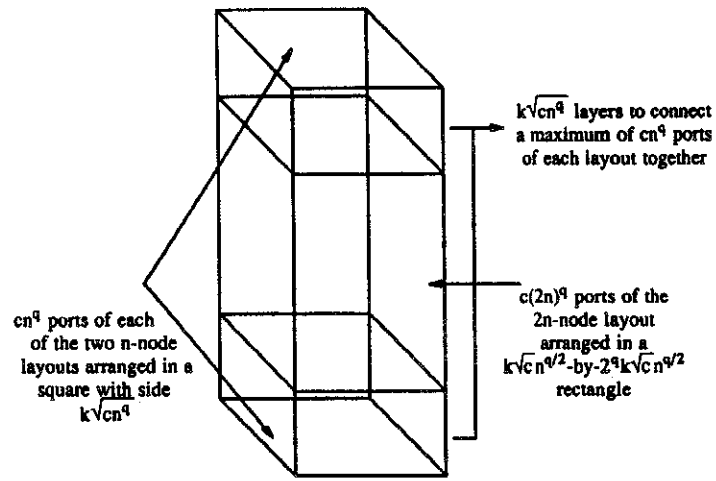


Figure 2.12

Combining two n-node layouts to form one 2n-node layout in the unrestricted model.

For the maximum wire length, note that the maximum wire length of the 8n-node layout is equal to the maximum wire length of the n-node layout plus $\alpha L(8n)$, where $L(8n)$ is the length of each side of the 8n-node layout, and α is a constant. (Due to the crossbar connection, the maximum distance between any two ports is $\alpha L(8n)$). The maximum wire length can be computed from the following recurrence.

$$W(8n) = W(n) + \alpha L(8N). \tag{2.24}$$

Substituting L from Eq. (17),

$$W(N) = \begin{cases} O(N^{1/3}) & 0 \le q < 2/3 \\ O(N^{1/3}\log N) & q = 2/3 \\ O(N^{q/2}) & 2/3 < q \le 1. \end{cases} \tag{2.25}$$

According to the theory of induction, the theorem is proved. □

Table 2.4 shows the upper bounds obtained here for the volume and maximum wire length, and compares them with previousl results [LeR83a] [LeR86] [LeR84]. Comparing the upper bounds in Table 2.4 and the lower bounds in Table 2.1, all upper bounds on volume obtained here are tight except for the family of graphs with $\Theta(N^{2/3})$ separator.

2.5. Optimization of Total Cost of Implementation

Up to this point, the volume occupied by the components of a 3-D VLSI layout has been considered to be the cost of the layout. In general, the volume is not directly related to the cost of implementation because the cost of running a wire or placing a device in a 3-D volume may depend on its location in the chip. In contrast, in a 2-D implementation, the cost of running a wire or placing a device is independent of its location, and hence, the area is related to the cost of implementation by a constant. To compare the trade-off between 2-D and 3-D implementations, the criterion used must be based on costs.

In this section, it is assumed that the cost is a function of the number of layers in the 3-D chip and the total cost, instead of volume, is minimized in a one-active-layer

Table 2.4.
Upper-bound volume and maximum wire length for the 3-D unrestricted layouts. (The previous and new results may be the same but obtained by a different method.)

Graph Separator f(N)	Volume		Maximum Wire Length	
	Previous	New	Previous	New
$\Theta(N^q)$, $0 \le q < 1/2$	$O(N)$ [LeR86]	$O(N)$	$O(N^{1/3})$ [LeR86]	$O(N^{1/3})$
Planar	$O(N\log^{3/2}N)$ [LeR86]	$O(N)$	$O(N^{1/3}\sqrt{\log N})$ [LeR86]	$O(N^{1/3})$
$\Theta(\sqrt{N})$	$O(N\log^{3/2}N)$ [LeR86]	$O(N)$	$O(N^{1/3}\sqrt{\log N})$ [LeR86]	$O(N^{1/3})$
$\Theta(N^q)$, $1/2 < q < 2/3$	$O(N^{q+1/2}\log^{3/2}N)$ [LeR86]	$O(N)$	$O(N^{q/3+1/6}\sqrt{\log N})$ [LeR86]	$O(N^{1/3})$
$\Theta(N^{2/3})$	$O(N^{7/6}\log^{3/2}N)$ [LeR86]	$O(N\log^3 N)$	$O(N^{7/18}\sqrt{\log N})$ [LeR86]	$O(N^{1/3}\log N)$
$\Theta(N^q)$, $2/3 < q < 1$	$O(N^{q+1/2}\log^{3/2}N)$ [LeR86]	$O(N^{3q/2})$	$O(N^{q/3+1/6}\sqrt{\log N})$ [LeR86]	$O(N^{q/2})$

layout. In the following discussion, planar graphs are treated in the same way as graphs with $\Theta(\sqrt{N})$ separator, although planar graphs may have separators less than $\Theta(\sqrt{N})$.

Consider the problem of laying out a graph with $\Theta(N^q)$ separator, $0 \le q < 1/2$. Since the complexities of a 3-D layout are $\Theta(N)$, which is the same as that of a 2-D layout [Lei84], the cost of a 3-D implementation will differ from that of a 2-D implementation by a constant factor.

To lay out a graph with $\Theta(N^q)$ separator, $1/2 \le q \le 1$, Leiserson has proved that the area required for a 2-D implementation is [Lei80]

$$A = \alpha N D^2(N), \tag{2.26}$$

where

$$D(N) = \sum_{i=0}^{\log_2 N} \left[\frac{N}{2^i} \right]^{q-1/2} \tag{2.27}$$

α is a graph-dependent constant, and N is a power of 2. The i'th term in the summation in Eq. (2.27), $(N/2^i)^{q-1/2}$, is the increase in area to connect two $(N/2^i)$-node subgraphs together in a 2-D implementation. It was shown in Section 2.1 that $2(N/2^i)^{q-1/2}$ layers are needed in a 3-D implementation to connect two $(N/2^i)$-node subgraphs together (one layer will be needed if $q = 0.5$). Hence, if the interconnections of a set of subgraphs are implemented in the third dimension, then the corresponding terms in Eq. (2.27) should be eliminated in computing the area of the base, and the height of the chip will be increased by the sum of the terms eliminated. Note that this is an upper bound on the number of layers, since it has been assumed that a crossbar connection was used to perform the routing. Suppose that h layers are used in the third dimension, then a number of terms will be chosen from Eq. (2.27) to sum up to h. The terms chosen will depend on the graph concerned and the cost of implementation in the third dimension. The area of the base of the layout using h layers is

$$B_h = \alpha N[D(N) - h + 1]^2 \qquad 1 \le h \le H, \tag{2.28}$$

where H is the maximum height. The maximum height can be computed from Eq's (2.5) and (2.9).

$$H = \begin{cases} \log N & \text{for graphs with } \Theta(\sqrt{N}) \text{ separator} \\ \beta N^{q-1/2} & \text{for graphs with } \Theta(N^q), \text{ separatori, } 1/2 < q \le 1 \end{cases} \tag{2.29}$$

where β is a graph-dependent constant. The volume of this chip is

$$V_h(N) = B_h h = \alpha N[D(N) - h + 1]^2 h \tag{2.30}$$

If c(j) is the cost of implanting layer j in a 3-D VLSI layout, then the total cost of using h layers is

$$cost_h = \alpha N[D(N) - h + 1]^2 \sum_{j=1}^{h} c(j) \qquad 1 \le h \le H \tag{2.31}$$

Hence, to minimize the total cost of implementation, it is necessary to solve the following optimization problem:

$$\min_{1 \le h \le H} cost_h = \min_{1 \le h \le H} \left\{ \alpha N[D(N) - h + 1]^2 \sum_{j=1}^{h} c(j) \right\} \tag{2.32}$$

As an illustration, if c(j) is a constant independent of j, then the cost of the circuit is the same as its volume. The number of layers h should be set as D(N) to minimize the volume, and the maximum number of layers will be used.

2.6. Summary

In this chapter, we have developed improved lower and upper bounds on volume and maximum wire length in both the one-active-layer and unrestricted layouts [AbW87]. Optimal complexities on the volume of layout have been found except for two cases: the layout of planar graphs in the one-active-layer model and the layout of graphs with $\Theta(N^{2/3})$ separator in the unrestricted model. To compare between 2-D and 3-D implementations, a simple model has been proposed to compute the total cost of layout. Table 2.5 summarizes the results obtained in this chapter.

Table 2.5.
Summary of 3-D VLSI layouts results.

Graph Separator f(N)	Layout	Comments
$\Theta(N^q)$, $0 \le q < 1/2$	one active layer	Bounds on volume and maximum wire length are already tight.
	unrestricted	Bounds on volume are already tight. Improved bounds on maximum wire length, bounds are now tight.
Planar	one active layer	Improved lower bounds on volume and maximum wire length have been found. Bounds on volume and maximum wire length are not tight.
	unrestricted	Improved upper bounds on volume and maximum wire length have been found. Bounds on volume are tight.
$\Theta(\sqrt{N})$	one active layer	Improved lower bounds on volume and maximum wire length have been found. Bounds on volume are tight.
	unrestricted	Improved upper bounds on volume and maximum wire length have been found. Bounds on volume are tight.
$\Theta(N^q)$, $1/2 < q \le 1$	one active layer	Improved lower bound on maximum wire length and upper bound on volume have been found. Bounds on volume and maximum wire length are tight.
	unrestricted	Improved upper bounds on volume and maximum wire length have been found. Bounds on volume are tight, except for graphs with $\Theta(N^{2/3})$ separator.

CHAPTER 3

THE DESIGN OF SYSTOLIC ARRAYS

3.1. Introduction

This chapter focuses on synthesizing systolic arrays. Any synthesis process begins with a high level specification of the problem to be solved. A final implementation is then systematically derived from this specifications. Earlier work on synthesizing systolic arrays was based on the analysis of the data dependency of initial specifications. The initial specifications were assumed to be a system of recurrence equations. This chapter continues to use the system of recurrence equations as an initial specification. As we will see in Section 3.3, most of the previous methods for synthesizing systolic arrays are most successful for uniform recurrence equations, but fall short of expectation in the case of linear or non-uniform recurrence equations.

In this chapter, the idea of a *Control Flow Systolic Array* is introduced. The control flow systolic array is a powerful tool that can solve uniform, linear, or non-uniform recurrence equations. A methodology for mapping a system of recurrence equations into the control flow systolic array will be proposed. Section 3.2 briefly reviews the recurrence equations. Section 3.3 focuses on the previous work in synthesizing systolic arrays. Section 3.4 introduces the idea of a control flow systolic array. Section 3.5 proposes the model to be used throughout the rest of the Chapter. Section 3.6 proposes a methodology to efficiently search for a good solution in the space of all possible solutions. In section 3.7 we use the methodology to implement a control flow systolic array for both the transitive closure and dynamic programming problems, and show an example for a system of recurrences with nonlinear data dependencies. Section 3.8 discusses some experimental results. Section 3.9 is a

conclusion section.

3.2. Classification of Recurrences

Recurrence equations have long been used by mathematicians to express a large class of computations [KaM67] [Raj86] [RaF86]. These computations involve the evaluation of a function f at all points in a domain D. The recurrence equation specifies how the value of f at a point p in D depends on the value of f at other points in the domain. Based on these dependencies, recurrence equations are classified as uniform, linear, or non-linear; one-dimensional or multidimensional.

A simple example is the well known factorial function which is specified by the following equation.

$$f(n) = \begin{cases} 1 & n \le 0 \\ n * f(n-1) & n > 1 \end{cases} \qquad (3.1)$$

The factorial recurrence is a uniform recurrence with non-constant coefficients. To calculate f(n) we have to know f(n−1), which is a constant distance away from f(n) in the n hyperspace. However, because n is not a constant, this problem has non-constant coefficients. Notice also that the subscripts in the above example (representing the domain over which we want to determine f) range over only one index. In other words, the domain D is the set of integers. In general, D may be n-dimensional (typically a subset of the lattice points in Euclidean n-space E^n), and the right hand side of the equation may be any non-linear function of the values of the predecessors.

In combinatorial mathematics, where the primary concern is solving recurrence equations, the problem can be stated as follows. Given a recurrence relation describing $f(\vec{n})$ in terms of $f(\vec{n_i})$, determine a closed-form expression for $f(\vec{n})$, i.e. an expression of $f(\vec{n})$ that does not involve the value of f at any other points. Our objective here is somewhat different. We are not interested in finding a closed-form expression for f. We are interested in mapping the computation of $f(\vec{n})$ onto a systolic architecture. This means that the taxonomy of constant coefficients versus non-constant coefficients is of no concern to us here. We are interested in the relation between the

point $\vec{n}$ (at which the function f is to be computed) and the other points on which it depends. The following three definitions, the first two of which are due to Karp and Miller [KaM67], will be helpful at this point.

Definition 1. A recurrence equation over a domain D is defined as an equation in the form:

$$f(p) = \Phi\left[f(q_1), f(q_2) \ldots f(q_k)\right] + \Psi(p) \qquad (3.2)$$

where $p \in D$; $q_i \in D$ for $i = 1 \ldots k$. Φ is a single valued function which is strictly dependent on each of its arguments, and Ψ represents the input.

A system of m recurrence equations over a domain D is defined to be a family of of m mutually recursive equations, where each of the f_i's is defined by an equation of the form

$$f_i(p) = \Phi\left[f_{i_1}(q_1), f_{i_2}(q_2) \ldots f_{i_k}(q_k)\right] + \Psi_i(p) \qquad (3.3)$$

Definition 2 A. recurrence equation of the form defined above is called a ***uniform recurrence equation*** iff $q_i = p + d_i$ for $i = 1 \ldots k$, where d_i's are constant n-dimensional vectors, and n is dimension of the hyperspace (the domain D) in which the recurrence is to be calculated. An example of a uniform recurrence is the matrix multiplication operation. Multiplying two matrices, A and B, and storing the result in C, can be represented by the following recurrence.

$$C(I,J,K) = C(I,J,k-1) + A(I,K) \times B(K,J) \qquad (3.4)$$

In this case d = (0,0,1). Many problems, such as convolution, FIR filters, numerical solution of partial differential equations, and singular value decomposition, can be represented as uniform recurrences.

A family of uniform recurrence equations may be defined analogously. Consider the matrix multiplication example mentioned above. Pipelining the two input variables, results in a system of uniform recurrence equations as follows.

$$C(I,J,K) = C(I,J,K-1) + A(I,K,J) \times B(K,J,I) \qquad (3.5a)$$

$$A(I,K,J) = A(I,K,J-1) \tag{3.5b}$$

$$B(K,J,I) = B(K,J,I-1) \tag{3.5c}$$

Intuitively, a uniform recurrence equation can define a computation in which the dependencies are completely described by a finite number of constant vectors d_i, regardless of the size of the domain. This classification was first proposed and studied by Karp and Miller [KaM67]

Definition 3. A recurrence equation as defined by Definition 1, is said to be a *linear dependency recurrence equation* (or linear recurrence equation) if for $i = 1 \ldots k$, $q_i = A_i p + b_i$, where A_i is a constant n×n matrix and b_i is a constant n-dimensional vector. The linear recurrence equation has the following form.

$$f(p) = \Phi\left(f(A_1 p + b_1), f(A_2 p + b_2) \ldots f(A_k p + b_k)\right) + \Psi(p) \tag{3.6}$$

As with uniform recurrence equations, the definition for a linear recurrence equation can be extended to a family of m mutually recursive equations. It is obvious from the previous definition that uniform recurrence equations are a special case of the linear recurrence equations, when A_i is the identity matrix. Many problems in image processing can be formulated as linear recurrences. Examples include the transitive closure problem, shortest path problem, matching problems, optimal search trees, and context free language recognition.

A good example of a linear recurrence is the transitive closure problem. To find the transitive closure of a matrix C(I,J). the problem is formulated as

$$C(I,J,K) = C(I,J,K-1) \vee C(I,K,K-1) \wedge C(K,J,K-1) \tag{3.7}$$

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad b_1 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{3.8}$$

and

$$A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad b_2 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{3.9}$$

and

$$A_3 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad b_3 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{3.10}$$

A system of m recurrence equations can be defined analogously. Example of a system of two mutually dependent recurrence equation is as follows.

$$\begin{aligned} A(I,J,K) &= A(I+J,K,K-J) + B(I+K+J,K+J,K-1) \\ B(I,J,K) &= A(I,J+I,K-1) + B(I,J-I,K-1) \end{aligned} \tag{3.11}$$

Definition 4: A recurrence is called a *non-linear recurrence* if $q_i = \Psi(p_i)$ and the function Ψ is any non-linear function. Notice that Ψ could be given by a table indicating for each point p in D the points q_i that are needed as arguments. Many problem in optimization can be formulated as non-linear dynamic programming problem with non-linear recurrences. An example of non-linear recurrence is the knapsack problem.

In the knapsack problem we are given n objects, where object j has weight a_j and value b_j . The desired goal is to pack a knapsack with a subset of these objects so as to maximize the value (or to minimize the negative of the value) without exceeding a given maximum weight W [AhH74].

One formulation for this problem is as a non-uniform recurrence equation [KaH67].

$$f(0,0) = 0$$

$$f(k,w) = \min_{j \in J_k} \left\{ f(j, w - a_k) \right\} \tag{3.12}$$

where $j_1 = \{ j: j < k$ and $(j, w - a_k)$ corresponds to an equivalence class $\}$

A state corresponding to the equivalence class consists of all sequences $\{i_1 \ldots i_r\} \subset S$ such that $i_r = k$ and $a_{i_1} + a_{i_2} + \ldots a_{i_r} = w$ are donated as (k, w) , and S is the set of all sequences $i_1, i_2, \ldots i_r$ such that $i_1 < i_2 < \ldots < i_r$ and $a_{i_1} + a_{i_2} + \ldots a_{i_r} \leq W$.

Similarly, a set of mutually dependent non-linear recurrences can be defined. As an example,

$$A(I,J,K) = A(I^2,J\times K,K-1) + B(I\times J,K,K-1)$$
$$B(I,J,K) = A(I,J^3,K) + B(I\times K,J,k-1) \qquad (3.13)$$

are a set of non-linear recurrences.

Many other problems in combinatorial mathematics, such as the traveling salesman problem, finding a shortest reset sequence, and 1-dimensional stock cutting, can be represented as non-linear recurrence equations.

As mentioned earlier, a recurrence equation can be viewed as defining a dependency graph in which a precedence relation is defined between the evaluation of f at various points $p \in D$. According to Karp and Miller [KaM67], point p directly depends on point q, and is denoted by $p \xrightarrow{1} q$ if and only if

(1) $p \in D$ and $q = p + d_i$ for the uniform recurrence:

(2)$p \in D$ and $q = A_i p + d_i$ for the linear recurrence;

(3)$p \in D$ and $q = \Psi_i(p)$ for the non-linear recurrence.

Thus $p \xrightarrow{1} q$ if and only if f(q) is one of the arguments in f(p). t step dependency is defined as follows: $q \xrightarrow{0} q$ and $p \xrightarrow{t} q$ if there exists r such that $p \xrightarrow{t-1} r$ and $r \xrightarrow{1} q$. It is also true that $p \rightarrow q$ if $p \xrightarrow{t} q$, for some positive integer t.

The objective of this chapter is to reorganize this dependency graph into an alternate configuration that preserves the functionality of the original recurrence and that it can be mapped directly to a systolic array. A graph can be mapped directly to a systolic array if it is at most two dimensional and if all communications (dependencies) are between neighboring points. In this case a function t(p) is assigned that indicates the time at which the computation of f(p) will take place. t(p) must obey certain conditions imposed on it by the dependency graph. For example, if $p \rightarrow q$, then $t(p) \geq t(q)$, which means that if the computation of f(p) requires f(q) as one of its arguments, then f(p) should be calculated after computing f(q). Notice that this condition is necessary but not sufficient. If f(q) was calculated in a non-neighboring processor to the one that computes f(p), the time needed to transmit f(q) from where it was originated to where it is needed must also be accounted for.

3.3. Previous work

This section will review previous work in designing systolic arrays. Table 3.1 shows 19 methods for synthesizing systolic arrays and briefly explains each method. A complete description for each method can be found elsewhere [FoF86].

Most of the methods shown in Table 3.1 are dedicated to solving one type of recurrences, namely, a system of uniform recurrence equations, while few methods are proposed for general recurrences. Moreover, all of these methods formulate the problem as a search in the space of all possible solutions. This space is usually exponential in size and in some cases is infinite [Mol83].

The next section presents the idea of a control flow systolic array, which proves to be effective in solving both linear and non-uniform recurrence equations.

3.4. Control Flow Systolic Arrays

Most of the methods shown in Table 3.1 are particularly successful in solving uniform recurrences. The main reason for this is that there is a direct relationship between uniform recurrences and systolic arrays. In using uniform recurrences to calculate the function at a point p, we need the value of the function at a fixed distance away from p (usually a distance of +1 or −1 away from p). In systolic arrays , we allow communications only between neighboring processors On the other hand, for linear and non-uniform recurrences, where the dependencies between different points in the domain may not allow all adjacent points to be mapped to neighboring processors and may vary with time, it is much more difficult to map this problem into a systolic array.

In this chapter, the idea of *Control Flow Systolic Arrays* is introduced. A control flow systolic array is an ordinary systolic array, with all of the limitations on systolic arrays such as modularity and nearest neighbor connections, but with one major difference. This difference lies in the propagation of control signals in addition to data

Table 3.1.
Summary for the previous work in the design of systolic arrays

Method	Description
Li and Wah [LiW83a] [LiW83b] [LiW85a]	Starting from a uniform recurrence equation, they defined 3 parameters for the systolic array, speed of data, period of computations, and data distribution. The 3 parameters are combined to form a system of constrained vector equations. A solution that minimize a certain criterion is found by systematically searching the space of all possible solutions
Moldovan and Fortes [Mol82] [Mol83] [FoM85]	Starting from a high level language algorithm (equivalent to a non-linear recurrence), they modeled the algorithm as five tuplets (index set, set of dependencies, set of computations, input set and output set). A linear transformation is used to transform the original algorithm into a computationally equivalent algorithm with a prespecified set of dependencies.
Johnson and Cohen [JoC81a] [JoC81b] [JoW81]	Starting from a mathematical expression (equivalent to a uniform recurrence), they mapped it directly into a VLSI circuit. They then improve the design by symbolically manipulating the original expression to eliminate unwanted operations (simultaneous addition or multiplication) and introduce pipelining.
Jagadish, Kailath, Newkirk and Mathews [JaK84]	Starting from a block diagram describing the algorithm (equivalent to a uniform recurrence equation). Each module is represented as a graph. The graph is sorted to ensure that there are no separator free loops (separator is a delay node). A schedule is sought for each separator in the graph, and these schedules are combined together to produce a schedule for the whole graph.

Table 3.1 (continued)

Method	Description
Miranker and Winkler [MiW84]	Starting from a do loop or a set of uniform recurrence equations, They interpreted the computation process as a graph in physical space and time, which directly corresponded to a systolic array. They embedded the graph representing the data flow of the program loop or the recurrence equation into the graph representing the computation process.
Melhem and Guerra [MeG85]	Starting from a high-level-problem specification, they transformed it into a (possibly non-linear) recurrence equation. Their procedure began with determining a coarse timing function for the different computations in the recurrence. The timing function, together with a subset of the data dependencies, were used to guide the search for an index transformation to map the computations into the systolic array.
Delosme and Ipsen [DeI85a] [DeI85b]	Starting from a a system of linear recurrence equations, they derived a mathematical description for the data dependency of the recurrence equations. By applying different affine transformations, different systolic arrays can be obtained. The problem of synthesis the array is formulated as an integer programming problem.
Leiserson, Rose, and Saxe [LeR83b] [LeS81]	Starting from a graph representation of the problem (equivalent to a non-linear recurrence), they mapped it directly into a VLSI array, which is not necessarily systolic. Transformations, such as retiming and k-slowdown are used to eliminate global communication and convert the array into a systolic array.

Table 3.1 (continued)

Method	description
Quinton [Qui83] [Qui84]	Starting from a system of (possibly non-linear) recurrence equations, a timing function that satisfies the dependency conditions is found. This timing function, together with the dependency conditions, is used to find a space mapping function such that, if two computations are mapped to the same time unit, they should be mapped to two different PE's. Using these two functions, a recurrence equation can be mapped directly into a systolic array.
Cappello and Steiglitz [CaS81] [CaS84]	Starting from an algebraic equation (equivalent of uniform recurrence equation), the concept of time is introduced to produce a recurrence equation. A space transformation and a time transformation are used to map the recurrence into a systolic array. A geometric transformation is then used to transform the systolic array to a different, but computationally equivalent, systolic array. This process is repeated until a satisfactory design is found.
Rao and Kailath [RaK85]	Their model is a linear systolic array with each PE represented as a transfer or scattering matrix. Starting from a mathematical expression (equivalent to a uniform recurrence), they showed how to systematically map the expression into the model array.
Schartz and Barnwell [ScB84]	Starting from an algorithm described by shift-invariant flow graph (equivalent to a uniform recurrence equation), their method consists of applying a set of rules to the flow graph that systematically manipulate the flow graph into a systolic form, which can be mapped directly into a systolic array.

Table 3.1 (continued)

Method	description
Jover and Kailth [JoK84]	Starting from a graph that represents certain computations (equivalent to a uniform recurrence), they introduced the notion of Line of Code (LOC). The LOC represents a stream of values in different stages of computation. Since the LOC in a graph is not unique, they searched for the LOC that satisfies some desirable properties. Knowing a LOC determines the speed of the data, the rate of pipelineability, the input rate, and the output rate of the systolic array.
Lam and Mostow [LaM83]	Starting from a high level specification (high level language), they applied some software transformation so as to obtain an algorithm suitable for systolic implementation. This transformed algorithm is the input to a program called SYS, which decomposes the algorithm into two parts: primitive code segments and control and data-access information. SYS designs a systolic implementation based on the control structure and data access pattern. The high level language is equivalent to a non-linear recurrence.
H. T. Kung and Lin [KuL83]	Starting from an algebraic representation (equivalent to a uniform recurrence), algebraic transformation similar to the transformations used in linear algebra are used to generate alternative but computationally equivalent designs satisfying some desirable properties.
S. Y. Kung [Kun84]	Starting from a signal flow graph (equivalent to a non-linear recurrence), a methodology is proposed to convert a Signal Flow Graph into a synchronous systolic array or a data-driven array.

Method	description
Rajopadhye and Fujimoto [RaF83] [RaF86]	Starting from a linear recurrence equation, they determined a timing function and an allocation function to map the recurrence into a systolic array. They presented a technique to determine such two functions. They also presented a technique called explicit pipelining which allows them to derive a systolic implementation for any afine recurrence equation.
Kung, Lo, and Lewis [KuL87b]	Starting from a linear recurrence equation, which defines a dependency graph, they mapped the recurrence directly into a two dimensional systolic array. A transformation is then used to reindex the nodes of the dependency graph to eliminate any global communication.
Chen [Che86] [Che85a] [Che85b] [Che85c]	The initial specification of the problem is in the language Crystal, (equivalent to a non-linear recurrence) The language is a general purpose language for parallel programming. The synthesis process is decomposed into two transformations. The first transformation limits the degree of fan in and fan out for the different computations. The second transformation incorporates pipelining into the design to fully utilize the hardware resources. Using these two transformations, a systolic algorithm can be systematically derived for any problem.

in the systolic array. There are two major ways to incorporate control signals in a systolic array. The first is to continuously execute a microprogram stored at each PE. No global information is available except for the address of the PE (the address of a PE is a pair (i,j) which indicates the Cartesian coordinates of the PE in a mesh of processors, or a number i to indicate the position of the PE in a linear array). The control in this case is implicit, as the information needed for the control is propagated as data. The second form of control is in the form of additional control bits traveling with the data, either at the same speed or at a different speed. The action taken at each processor depends on the control signals that reach the PE.

3.5. The Model

The model we use throughout the rest of this chapter is a square (or linear) array of processors. The architecture of the individual processing element will be discussed later. We will also study the effect of various control strategies (different control strategies wii be discussed later) on the complexity of the processor element.

The model is a square mesh of processors with nearest neighbor connection as shown in Figure 3.1. There are three reasons for choosing this architecture.

(1) Modularity: The square mesh of processors is modular. This makes it suitable for VLSI implementation.

(2) Balancing I/O and computations: The square array of processors is the best for balancing the I/O bandwidth and the computation time for a great variety of problems.

(3) Reconfiguration: The square array of processors can easily be reconfigured into a linear array [LeL82] [LeL85], thus achieving all the benefits of linear arrays, such as fault tolerance, minimal clock skews [FiK83a], and limited I/O requirements.

Figure 3.1 shows a schematic diagram for the systolic system. n^2 Processor Elements (PE) are connected together as a mesh. $\sqrt{n}$ global buses, one for each row, are connected together by one global bus to the main controller. The host controls both the main controller and the mesh of PE's. The internal architecture of the PE is shown in Figure 3.2.

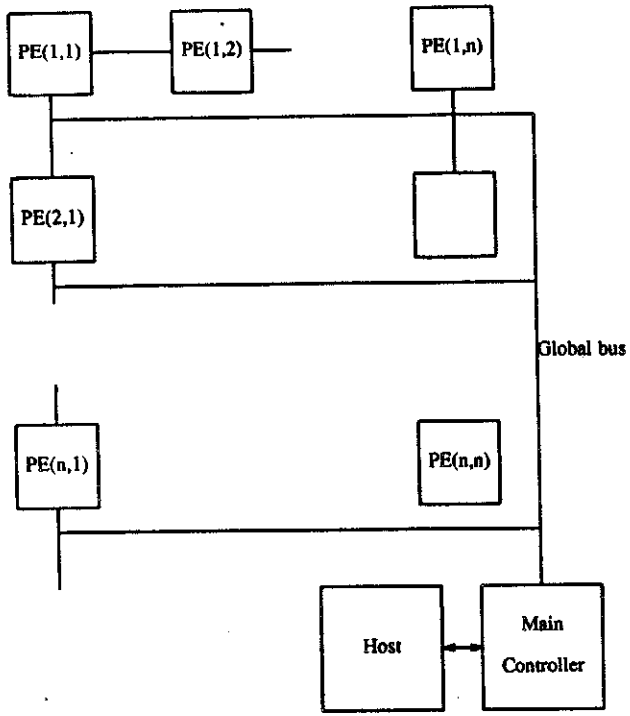


Figure 3.1

n by n Control flow systolic array

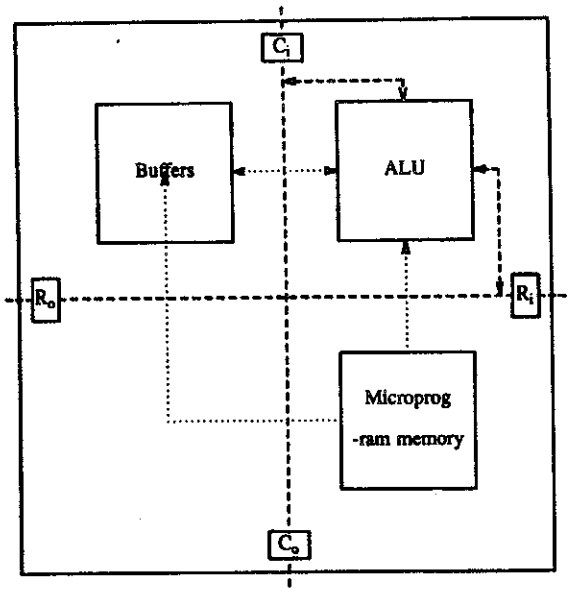


Figure 3.2

The internal architecture of a PE (dashed lines represents data paths, while dotted line represents control paths)

Each PE is composed of:

(1) Each PE is connected by four communication registers to each of its four neighbors. R_i and R_o are connected to the two processors in the same row. R_i of a certain PE will be connected to R_o of the next PE. R_i represents the input row register for this PE. This is the register that receives the data from a neighboring PE and makes it available for use in this PE. R_o is the output register of this PE. This is the register that holds the output data to be received by R_i of the next PE. These functions will be served by C_i and C_o for connected PE's in neighboring rows. Notice that R_i and R_o are not fixed. They are interchangeable according to the direction of the data flow. If the data are traveling from left to right, the left register will be R_i and the right register R_o. If the data are traveling from right to left, the right register will be R_i, and the left register will be R_o.

(2) The ALU is the part of the processor responsible for the actual computation. It is capable of performing simple arithmetic and logical operations. The ALU also contains an accumulator to store the output of the last operation performed in the PE. It can receive its input from R_i and C_i or from the storage unit. Similarly, it can direct its output to R_o or C_o, or it can store the output in the storage unit. The ALU is also capable of transferring a piece of data from the storage unit to either C_o or R_o. The ALU is controlled by the microprogram unit.

(3) The storage unit is composed of registers to hold intermediate results and to introduce delay into the data traveling across rows or columns. The number of the registers is very important in the design of the array. As will be seen later in this chapter, some algorithms require a constant number of buffers, while others require O(n) buffers, where n is the size of the problem.

(4) The microprogram unit contains the microinstructions to be executed by the ALU during the solution of a problem. The microprogram is loaded from the the main controller before the beginning of the solution process. It is capable of recognizing the PE address, where the PE address is taken to be the position of the PE in the mesh, (i.e. (i,j)). If the same microprogram is to be executed at all PE's, the main controller will broadcast it with a specific address mask which will be accepted by all PE's in the mesh. Otherwise, each microprogram will be sent to a specific PE according to its address (i,j). The microprogram unit should be capable of executing instructions, such as, get data from R_i or C_i, load data to R_o or C_o, Store contents of accumulator in buffers, and load data from buffers to the accumulator.

3.6. The Methodology

A considerable amount of research has been dedicated to the design of systematic methods for synthesizing specified systolic arrays [FoF86]. Most of the methods are most successful in solving either uniform recurrence equations or program segments that can be represented as uniform recurrence equations. Moldovan [Mol82,Mol83] has shown that finding the optimal transformation to map a uniform recurrence into a systolic arrays can be solved as n×m diophontine equations with n^2 variables. This requires exponential complexity, where n is the dimensions of the space in which the recurrence equation is to be solved, and m is the number of data dependencies. Very few methods that can be used with linear or non-uniform recurrence equations have been proposed. This chapter is concerned with the development of a heuristic methodology that can handle both uniform and non-uniform recurrences, and do the design in a reasonable amount of time.

The basic idea behind the following methodology is to develop a relationship between the data dependencies of the recurrence and the hardware model to be used. The complexity of the solution and the time required to solve a given problem will both depend on the complexity of the hardware in the Processor Element (PE) and the complexity of the instruction set of the microprogrammed unit. A very simple case is when there is only one buffer for any PE, and the PE uses FIFO control strategy. The more complicated PE architectures contain a variable number of buffers and random-access buffer capabilities.

This methodology will be implemented in the form of a procedure. The input of this procedure is the given recurrence. The output is a complete timing function that gives the location and time of the different computations. The output will also contain

complete information about the variables that is broadcast, and the location of these variables.

The rules developed in this chapter are not intended to be the complete set of rules for solving any recurrence. We will begin with approximately 10 rules. Combination of these rules can lead to approximately 30 different designs, although some of these designs may not be feasible. As we gain experience in solving more problems, more powerful rules will be discovered. These rules can be added to improve the capabilities of our procedure to solve additional problems.

The solution for a specific problem is implemented as a search operation in the space of all different solutions. Finding the solution for a specific problem can be represented by a search in the space of all combination for a solution that optimizes a certain criterion. A large amount of time is required for the procedure to go through all of the different combinations for solving a given problem. It is known, for instance, that it will take exponential time to explore all combinations for simple cases involving uniform recurrences. In addition, increasing the number of rules will lead to an even larger number of combinations. As the space of all possible solutions grows, it will not be practical to search through all the possibilities. In this case the search operation must search a subspace of the whole space to find the best solution within this subspace. Because time is limited, and because the procedure searches in a subspace of the solution space, the search must be ordered in such a way that the parts of the space that will most probably contain the best solution are searched first.

To order the search, the recurrences are divided according to distinguishing features into $\emptyset$ groups, G_j, where $1 \le j \le \emptyset$. The different rules are then given different weights where

$$W_{i,j} \quad 1 \le i \le L,\ 1 \le j \le \emptyset \qquad \text{where } L = \text{the number of rules} \tag{3.14}$$

$W_{i,j}$, then, is the weight of Rule i with respect to Group j, or the effectiveness of Rule i when applied to a problem which belongs to Group j. When the procedure considers a problem in Group j, it applies Rule k first, where $W_{k,j} = \min_i W_{i,j}$. It then tries the rule with the next smallest $W_{i,j}$ and so on, until the allowed time for computation is

over. After completing the search process, the procedure orders the solutions obtained according to a certain criterion from the best to the worst, and gives a weight $\alpha_{i,j}$ to the solution produced by Rule i. The rule that produced the best solution is given a weight of 1. The rule producing the second best solution is given a weight of 2, and so on. The ordering of rules may change after each subspace is searched. Modifications are made to reflect the experience learned from each solution.

$$W_{i,j} = W_{i,j} + \beta \times \alpha_{i,j} \qquad \text{where } \beta \text{ is a constant} < 1 \tag{3.15}$$

$W_{i,j}$ should then be normalized to prevent unlimited growth of $W_{i,j}$

$$W_{i,j} = \frac{W_{i,j}}{B} \qquad \text{where } B = \min_i W_{i,j} \tag{3.16}$$

The two main difficulties in mapping a non-uniform recurrence into a systolic array are non-neighboring communications, and communications that change with time. For both of these situations, pipelining of variables to communicate between non-neighboring processors will be used. The rules that will be used are grouped into three major groups: (1) projection, (2) pipelining, and (3) delay and timing calculations.

Projection: In calculating a recursive function f(p), the values of f at all points of a hyperspace D must be calculated. The easiest way to do this is to assign a PE to each point in the domain D. If the dimension of D is n. Then, the value of f at point $(i_1, i_2, \ldots, i_n)$ will be calculated at PE $(i_1, i_2, \ldots, i_n)$. This will not be an efficient way to calculate the recurrence for two reasons. First, if the dimension of D is more than 2, the resulting array will be complicated and irregular. Second, each PE will perform one computation for one time unit and will be idle for the rest of the time needed to complete the computation.

An alternative way to calculate the recursive functions on a two dimensional array is by projecting the hyperspace into a two dimensional space $(i_1, i_2, \ldots, i_m) \rightarrow (x,y)$, where m is the dimension of the hyperspace (domain) in which the recurrence is to be solved. The most straightforward way to map the m dimensional hyperspace onto a two-dimensional space is to chose $x = i_1$ and $y = i_2$.

Although more complicated mappings exists [Mol83], we will not investigate them in this thesis. In our methodology, the number of ways to project computations into two-dimensional array is m(m−1)/2. (equal to choosing 2 out of m dimensions).

Pipelining. Pipelining is one way to overcome communication difficulties between non-neighboring processors. In principle, it involves feeding data through a pipeline of neighboring processors.

Assume that we have a recurrence relation of the form:

$$f(i_1,i_2 \ldots, i_m) = \Phi \left[f(j_1,j_2 \ldots, j_m) \right] \tag{3.17}$$

where $(j_1,j_2 \ldots, j_m)$ is any function of $(i_1,i_2 \ldots, i_m)$. If we project on i_1, i_2, then the left hand side of the previous recurrence is to be calculated at PE (i_1,i_2), and the right hand side is calculated at PE (j_1,j_2). If (i_1,i_2) and (j_1,j_2) are not neighboring PE's, then the result generated at PE (j_1,j_2) must be sent to PE (i_1,i_2). This could be done using processors along the rows and columns of the array. In cases where diagonal connections are allowed, data can be sent along the diagonal connections directly.

Timing and delay calculations: In the rest of this section, the different control strategies are described.

Control Strategy 1. This is the fastest strategy for solving any recurrence. It is assumed that there is an unlimited number of buffers in each PE, and that each PE can randomly access any data item in its buffers. The only restriction is that in calculating the recurrence at any point p, it is necessary for all q_i's that have already been computed to be transmitted from where they were computed to where f(p) is being computed.

Assume that we have a recurrence of the form,

$$f(i_1,i_2, \ldots, i_m) = \Phi \left[f(j_1,j_2 \ldots, j_m) \right] \tag{3.18}$$

and that this recurrence is projected on the i_1,i_2 directions. (This is not a constraint on the solution, since the order of the index variables inside the recurrence can be changed to project in any chosen direction). Denote the time at which $f(i_1,i_2, \ldots, i_m)$ is calculated by $t(i_1,i_2, \ldots, i_m)$. Let

(1) $tri(i_1,i_2,j_1,j_2, \ldots, j_m)$: the time at which the broadcast message originating from $f(j_1,j_2, \ldots, j_m)$ has reached the row input register of PE (i_1,i_2);

(2) $tro(i_1,i_2,j_1,j_2, \ldots, j_m)$ be the time at which the broadcast message originating from $f(j_1,j_2, \ldots, j_m)$ reaches the row output register of PE (i_1,i_2);

(3) $tci(i_1,i_2,j_1 \ldots, j_m)$ be the time at which the broadcast message originating from $f(j_1,j_2 \ldots, j_m)$ has reached the column input register of PE (i_1,i_2);

(4) $tco(i_1,i_2,j_1 \ldots, j_m)$ be the time at which the broadcast message originating from $f(j_1,j_2 \ldots, j_m)$ has reached the column output register of PE (i_1,i_2).

The conditions to guarantee a correct execution are:

$$t(i_1,i_2 \ldots, i_m) = \max_j \left\{ tri(i_1,i_2,j_1, \ldots, j_m),\ p(i_1,i_2) + 1 \right\}, \tag{3.19}$$

where $p(i_1,i_2)$ is an array that holds the time of the last operation performed at PE (i_1,i_2), and the maximum is taken over all the arguments of the original recurrence in Eq. 3.18

$$tri(i_1,i_2,j_1, \ldots, j_m) = \begin{cases} tro(i_1,i_2-1,j_1 \ldots, j_m) + 1 & i_2 > j_2 \\ \max\{tco(i_1,i_2,j_1,j_2, \ldots, j_m), r(i_1,i_2) + 1\} & i_2 = j_2 \\ tro(i_1,i_2+1,j_1 \ldots, j_m) + 1 & i_2 < j_2 \end{cases} \tag{3.20}$$

where r(i,j) is an array that holds the time of the last data item that was sent through the row direction of PE (i_1,i_2). For $tci(i_1,i_2,j_1, \ldots, j_m)$,

$$tci(i_1,i_2,j_1, \ldots, j_m) = \begin{cases} tco(i_1-1,i_2,j_1 \ldots, j_m) + 1 & i_1 > j_1 \\ \max\{t(i_1,i_2 \ldots, i_m),\ c(i_1,i_2) + 1\} & i_1 = j_1 \\ tco(i_1+1,i_2,j_1, \ldots, j_m) + 1 & i_1 < j_1 \end{cases} \tag{3.21}$$

where $c(i_1,i_2)$ is an array holding the time of the last data item that was sent through the column direction of PE (i_1,i_2),

$$tro(i_1,i_2,j_1 \ldots, j_m) = tri(i_1,i_2,j_1 \ldots, j_m). \tag{3.22}$$

$$tco(i_1,i_2,j_1 \ldots, j_m) = tci(i_1,i_2,j_1 \ldots, j_m). \tag{3.23}$$

Notice that in this strategy the only constraint for calculating the function at a certain point $(i_1,i_2 \ldots, i_m)$ is that its arguments must have already been calculated and transmitted to the PE where the point $f(i_1,i_2 \ldots, i_m)$ is being calculated. By solving this system of recurrence equations, we get the values of $t(i_1,i_2, \ldots, i_m)$, $tri(i_1,i_2,j_1,j_2 \ldots, j_m)$, $tci(i_1,i_2,j_1,j_2 \ldots, j_m)$, $tro(i_1,i_2,j_1,j_2 \ldots, j_m)$, and $tco(i_1,i_2,j_1,j_2 \ldots, j_m)$ at all points in the domain D. From these values we can determine the maximum number of buffers in each PE and the delay encountered by each data element at any given PE. The maximum number of buffers needed in each PE can be calculated using $t(i_1,i_2, \ldots, i_m)$, which represents the time at which the computations of $f(i_1, \ldots, i_m)$ takes place at PE (i_1,i_2), and $tri(i_1,i_2,j_1,j_2 \ldots, j_m)$, which represents the time at which $f(j_1,j_2 \ldots, ,j_m)$ arrives PE (i_1,i_2).

The boundary conditions for these recurrences represent the input or the initial conditions for the original problem. For example, if we assume that $t(i_1, \ldots, i_{m-1},0) = 0$, then we are assuming that all of the initial conditions of $f(i_1 \ldots, i_{m-1},0)$ are stored before the beginning of the computations in PE (i_1,i_2). Alternatively, if we assume that $t(i_1, \ldots, i_{m-1},0) = i_1$, then, we are assuming that the initial conditions are pipelined through the first row to the rest of the array.

Control Strategy 2. The second control strategy assumes that the PE is capable of randomly accessing any buffer of its memory and that the number of buffers is bounded by $\emptyset$. In this case, a PE cannot store more than $\emptyset$ data items at the same time. Because of this limit on the number of buffers, no PE can accept a new variable unless it has cleared the same variable from $\emptyset$ iterations ago. Assume that $f(\emptyset_1,\emptyset_2, \ldots, \emptyset_m)$ is the variable used in calculating $f(i_1,i_2, \ldots, i_m)$ $\emptyset$ iterations ago. In this case, we have

$$tci(i_1,i_2,j_1,\ldots,j_m) = \begin{cases} tco(i_1-1,i_2,j_1\ldots,j_m) + 1 & i_1 > j_1 \\ \max\{\, t(i_1,i_2,j_1,\ldots,j_m)\,,\, c(i_1,i_2) + 1\,\} & i_1 = j_1 \\ tco(i_1+1,i_2,j_1,\ldots,j_m) + 1\,, & i_1 < j_1 \end{cases} \quad (3.24)$$

$$tri(i_1,i_2,j_1,\ldots,j_m) = \begin{cases} tro(i_1,i_2-1,j_1\ldots,j_m) + 1 & i_1 > j_1 \\ \max\{\, tco(i_1,i_2,j_1\ldots,j_m)\,,\, r(i_1,i_2) + 1\,\} & i_1 = j_1 \\ tro(i_1,i_2+1,j_1,\ldots,j_m) + 1 & i_1 < j_1 \end{cases} \quad (3.25)$$

$$tro(i_1,i_2,j_1,\ldots,j_m) = \max\left\{tri(i_1,i_2,j_1,\ldots,j_m)\,,\, tro(i_1,i_2,\emptyset_1 \cdots \emptyset_m) + 1\right\} \quad (3.26)$$

$$tco(i_1,i_2,j_1,\ldots,j_m) = \max\left\{tci(i_1,i_2,j_1,\ldots,j_m)\,,\, tco(i_1,i_2,\emptyset_1 \cdots \emptyset_m) + 1\right\} \quad (3.27)$$

Control Strategy 3. In the third control strategy, each PE has an unlimited number of buffers but can access these buffers in a FIFO order only. This accessing scheme greatly simplifies the PE design and results in a smaller number of microinstructions. In essence this strategy means that if PE (i_1,i_2) needs $f(k_1,k_2\ldots,k_m)$ before it needs $f(j_1,j_2\ldots,j_m)$, then PE (i_1,i_2) cannot accept the broadcast message $f(i_1,i_2,j_1,j_2\ldots,j_m)$, unless it has already accepted the broadcast package $f(i_1,i_2,k_1,k_2\ldots,k_m)$. If PE (i_1,i_2) needs $f(k_1,k_2, \ldots, k_m)$ before it needs $(j_1,j_2, \ldots, j_m)$, then

$$tri(i_1,i_2,j_1 \ldots, j_m) = \begin{cases} tro(i_1,i_2-1,j_1 \ldots, j_m) + 1 & i_2 > j_2 \\ \max\{tco(i_1,i_2,j_1,j_2, \ldots, j_m)\,,r(i_1,i_2\} + 1] & i_2 = j_2 \\ tro(i_1,i_2+1,j_1 \ldots, j_m) + 1 & i_2 < j_2 \end{cases} \quad (3.28)$$

$$tci(i_1,i_2,j_1, \ldots, j_m) = \begin{cases} tco(i_1-1,i_2,j_1 \ldots, j_m) + 1 & i_1 > j_1 \\ \max\{t(i_1,i_2 \ldots, i_m)\,,\, c(i_1,i_2) + 1\,\} & i_1 = j_1 \\ tco(i_1+1,i_2,j_1, \ldots, j_m) + 1 & i_1 < j_1 \end{cases} \quad (3.29)$$

$$tro(i_1,i_2,j_1,\ldots,j_m) = \max\left\{tri(i_1,i_2,j_1,\ldots,j_m)\,,\, tri(i_1,i_2,k_1, \ldots, k_m) + 1\right\} \quad (3.30)$$

$$tco(i_1,i_2,j_1,\ldots,j_m) = \max\left\{tci(i_1,i_2,j_1,\ldots,j_m)\,,\, tci(i_1,i_2,k_1, \ldots, k_m) + 1\right\} \quad (3.31)$$

Control Strategy 4. The fourth control strategy employs a limited number of buffers, $\emptyset$, and FIFO access capabilities. This is a combination of Strategy 2 and Strategy 4. The equations for this control strategy are:

$$tci(i_1,i_2,j_1,\ldots,j_m) = \begin{cases} tco(i_1-1,i_2,j_1\ldots,j_m) + 1 & i_1 > j_1 \\ \max\{t(i_1,i_2\ldots,i_m)\,,\, c(i_1,i_2) + 1\} & i_1 = j_1 \\ tco(i_1+1,i_2,j_1,\ldots,j_m) + 1 & i_1 < j_1 \end{cases} \quad (3.32)$$

$$tri(i_1,i_2,j_1,\ldots,j_m) = \begin{cases} tro(i_1,i_2-1,j_1\ldots,j_m) + 1 & i_1 > j_1 \\ \max\{tco(i_1,i_2,j_1\ldots,j_m),\ r(i_1,i_2) + 1\} & i_1 = j_1 \\ tro(i_1,i_2+1,j_1,\ldots,j_m) + 1 & i_1 < j_1 \end{cases} \quad (3.33)$$

$$tro(i_1,i_2,j_1,\ldots,j_m) = \max\{tri(i_1,i_2,j_1,\ldots,j_m),\ tri(i_1,i_2,\emptyset_1,\ldots,\emptyset_m) + 1,$$
$$tri(i_{1},i_2,k_1,\ldots,k_m) + 1\} \quad (3.34)$$

and

$$tco(i_1,i_2,j_1,\ldots,j_m) = \max\{tci(i_1,i_2,j_1,\ldots,j_m),\ tci(i_1,i_2,\emptyset_1,\ldots,\emptyset_m) + 1,$$
$$tci(i_{1},i_2,k_1,\ldots,k_m) + 1\} \quad (3.35)$$

where $f(\emptyset_1,\ldots,\emptyset_m)$ and $f(k_1,\ldots,k_m)$ have the same meaning as in Strategies 2 and 3.

3.7. Examples

To illustrate the methodology, it will be applied to implement systolic arrays for the transitive closure and the dynamic programming problems.

3.7.1. The Transitive Closure Problem

Consider a directed graph G(V,E), where V is a set of vertices, and E is a set of directed edges. The graph $G^+(V,E^+)$, which has the same vertex set V as graph G, and has an edge from v to u if and only if there is a path from v to u in G, is called the reflective and transitive closure of G [AhH74]. This graph G can be represented by its adjacency matrix A, whose elements $a_{i,j} = 1$ if there is an edge from vertex i to vertex j; otherwise $a_{i,j} = 0$. The transitive closure problem requires computations of the transitive closure matrix A^+, whose elements $a^+_{i,j} = 1$ if there is a path of length zero or more from vertex i to vertex j. The shortest path problem can be stated in the same way if $a_{i,j}$ represents the distance between vertex i and vertex j and is equal to ∞; if there is no edge between i and j, then $a^+_{i,j}$ is equal to the shortest distance between vertex i and vertex j, and $a^+_{i,j}$ equal to ∞ if there is no path from vertex i to vertex j

The Warshal-Floyd Algorithm uses A as an input matrix and produces A^+ as the output. This algorithm is given below, where $x(i,j,0) = a_{i,j}$, and $x(i,j,N) = a^+_{i,j}$:

for k <— 1 to N

for i <— 1 to N

for j <— 1 to N

$$x(i,j,k) \leftarrow x(i,j,k-1) + x(i,k,k-1) \times x(k,j,k-1)$$

Notice that + represents the Boolean OR, and × represents the Boolean AND. If + is changed to the minimum operation and × to the addition operation, we get the shortest path problem.

A number of other researchers have proposed systolic arrays for the transitive closure problem. All of these proposals use approximately N^2 PE's to perform the computations in O(N) time, and all require some control signals to temporarily phase the PE's (phasing the PE means each PE will perform one or more functions according to a specific control signal traveling with the data).

The first systolic array developed for the transitive closure problem was a three-pass mesh connected array described by Guibas, Kung, and Thompson [GuK79]. This structure is relatively slow and requires wrap around interconnection. Hexagonal arrays have been proposed by Kung and Lo [KuL85] [LiW85], by Rote [Rot85], by Robert and Trystram [RoT85], and by Kung, Lo and Lewis [KuL87b]

The following section uses the methodology described in this thesis to design a systolic array for the transitive closure problem. The resulting design is faster than its predecessors.

The first step in designing such an array is to choose the axes that we will project on. In this case, the k axis will serve as a first trial. Figure 3.3 shows the data dependency for the transitive closure problem, where n=4. The different i-j planes are drawn separately, and the arrows indicate the data movements between the different PE's. It is obvious that there is communication between non-neighboring processors. Moreover, the data dependency changes as k changes (i.e., the data dependency changes

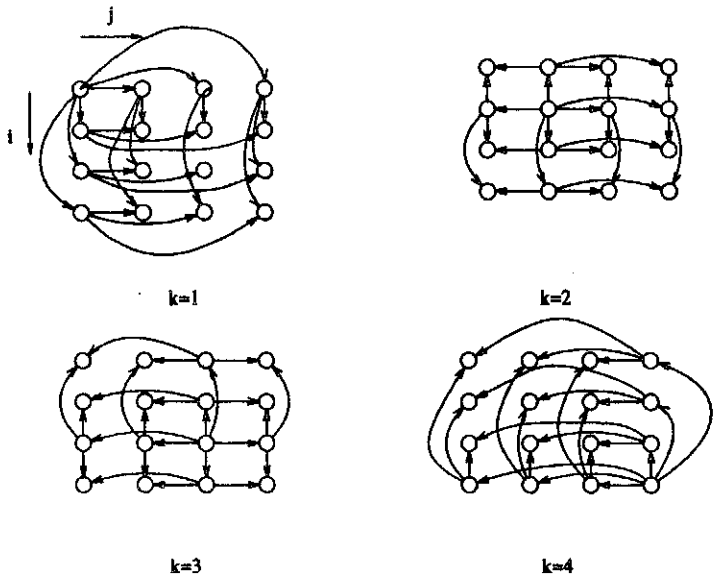


Figure 3.3

Data dependency graph of the transitive closure problem

with time). Using strategy 1, which puts no limits on the number of buffers or accessing of the buffers,

$$t(i,j,k) = \max\left\{ tri(i,j,i,k,k-1)\ ,\ tri(i,j,k,j,k-1)\ ,\ t(i,j,k-1)\right\} \tag{3.36}$$

$$tri(i,j,i_1,j_1,k_1) = \begin{cases} tro(i,j+1,i_1,j_1,k_1) + 1 & j < j_1 \\ \max\left\{tco(i,j,i_1,j_1,k_1)\ ,\ r(i,j)\right\} & j = j_1 \\ tro(i,j-1,i_1,j_1,k_1) + 1 & j < j_1 \end{cases} \tag{3.37}$$

$$tro(i,j,i_1,j_1,k_1) = tri(i,j,i_1,j_1,k_1) \tag{3.38}$$

$$tci(i,j,i_1,j_1,k_1) = \begin{cases} tco(i+1,j,i_1,j_1,k_1) + 1 & i < i_1 \\ \max\left\{t(i,j,i_1,j_1,k_1)\ ,\ c(i,j)\right\} & i = i_1 \\ tco(i-1,j,i_1,j_1,k_1) + 1 & i < i_1 \end{cases} \tag{3.39}$$

and

$$tco(i,j,i_1,j_1,k_1) = tci(i,j,i_1,j_1,k_1)\ . \tag{3.40}$$

Figure 3.4 shows the modified data dependencies after pipelining the broadcast variables through the different rows and columns. By solving these equations, assuming the initial conditions $t(i,j,0) = 0$, we get a solution for the transitive closure problem.

Figure 3.5 shows the array used to solve this problem, where the PE's in Figure 3.6 are replaced by the times at which the different computations will take place. Notice that the initial conditions is $t(i,j,0) = 0$ which means that the a_{ij} was stored at PE (i,j) before computations begin. Notice that in iteration k, the k-th row and the k-th column will act as source nodes to broadcast the content of its accumulator in both directions. This can be achieved with a single control bit that travels across the

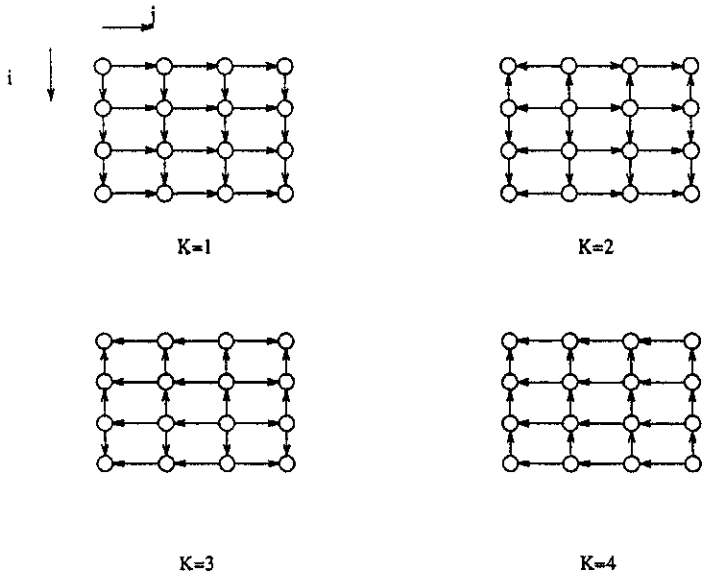


Figure 3.4

Data dependency for the transitive closure problem after pipelining

1	2	3	4
2	2	3	4
3	3	3	4
4	4	4	4

k=1

4	4	5	6
4	3	4	5
5	4	5	6
6	5	6	7

k=2

8	7	8	9
7	6	7	8
8	7	6	7
9	8	7	8

k=3

13	12	11	12
12	11	10	11
11	10	9	10
12	11	10	9

k=4

Figure 3.5

The time of the different computations for the transitive closure problem using strategy 1 (n=4)

rows and the columns, signaling to each PE when to broadcast the content of its accumulator. The rest of the PE's will use a handshaking mechanism, where each PE will wait until it receives the two data items through its input row and column registers to complete its computation. The time for this solution, $5N - 7$, is faster than any previous solution. Because for any i,j $t(i,j,k) < tri(i,j,i_1,j_1,k_1)$, the maximum number of buffers needed in each PE is one.

If Strategy 3, which allows one buffer in each PE, is considered, the equations are the same, except that tco has the following form

$$tco(i,j,i_1,j_1,k_1) = \begin{cases} t(i,j,k_1+1) & j=j_1 \text{ and } i_1=k_1+1 \\ tci(i,j,i_1,j_1,k_1) & \text{otherwise} \end{cases} \qquad (3.41)$$

Figure 3.6 shows the array used to solve this problem, where the PE's in Figure 3.6 are replaced by the times at which the different computations takes place. The time is $5N - 6$, which is faster than the array proposed by Kung and Lois [KuL87b].

3.7.2. Dynamic Programming Problem

Many problems in computer science can be solved by using dynamic programming techniques; including shortest path, optimal parenthesization, partition and matching problems. For a fuller discussion of this spectrum see the review article by K. Brown [Bro78] and the references mentioned therein. The following example on the optimal parenthesization problem, which can be formulated as

$$C(i,j) = \min_{i<k<j} \left\{ c(i,k) + c(k,j) \right\} \qquad (3.42)$$

This in turn, can be translated to

1	2	3	4
2	2	3	4
3	3	4	5
4	4	5	6

k=1

4	4	5	6
3	3	4	5
4	4	5	6
5	5	6	7

k=2

9	8	9	8
8	7	7	8
7	6	6	7
8	7	7	8

k=3

14	13	12	12
13	12	11	11
12	11	10	10
11	10	9	9

k=4

Figure 3.6
The time of the different computations for the transitive closure problem using strategy 3 (n=4)

for i <— 1 to n–1

for j <— i+1 to n

for k <— i to j–i

$$c(i,j,k) = c(i,j,k-1) + \min\left\{ c(i,k,k-i)\ ,\ c(k,j,k-j) \right\}$$

Any direct implementation of this recurrence will lead to a solution with time $O(n^2)$. Figure 3.7 shows the data dependency for the dynamic programming problem, projected on the k axis, where node (i,j) represents the computations of c(i,j,k). Close examination of Figure 3.7 shows that to calculate c(i,j) the first available terms are c(i,j/2) and c(i/2,j). A better algorithm, then, is

for i <— 1 to n–1

for j <— i+1 to n

for k <— 1 to $\left\lfloor \frac{j-i}{2} \right\rfloor$

$$c(i,j,k) = c(i,j,k-1) + \min\left\{ c(i,i+m-k+1,k_1)\ ,\ c(i+m-k+1,j,k_2) \right\} \qquad (3.43)$$

where $k_1 = \left\lfloor \frac{m-k+1}{2} \right\rfloor$ $\qquad k_2 = \left\lfloor \frac{j-i-m+k-i}{2} \right\rfloor$

By applying the rules in Strategy 1, we get

$$t(i,j,k) = \max\left\{ t(i,j,k-1)+1\ ,\ tri(i,j,i_1,j_1,k_1)\ ,\ tri(i,j,i_2,j_2,k_2) \right\} \qquad (3.44)$$

where

$i_1 = 1, \quad j_1 = i+m-k+1, \quad k_1 = \left\lfloor \frac{j_1-i_1}{2} \right\rfloor,$

$i_2 = i+m-k+1, \quad j_2 = j, \quad k_2 = \left\lfloor \frac{j_2-i_2}{2} \right\rfloor.$

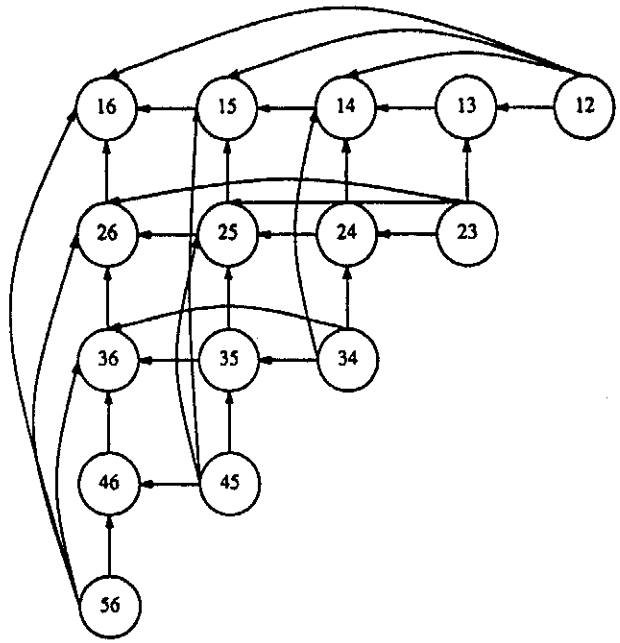


Figure 3.7

Data dependency graph for dynamic programming problem

$$tri(i,j,i_1,j_1,k_1) = \begin{cases} tro(i,j-1,i_1,j_1,k_1) + 1 & j > j_1 \\ \max(tco(i,j,i_1,j_1,k_1),\ r(i,j)) & j = j_1 \\ tro(i,j+1,i_1,j_1,k_1) + 1 & j < j_1 \end{cases} \quad (3.45)$$

$$tro(i,j,i_1,j_1,k_1) = tri(i,j,i_1,j_1,k_1) \quad (3.46)$$

$$tci(i,j,i_1,j_1,k_1) = \begin{cases} tco(i-1,j,i_1,j_1,k_1) + 1 & i > i_1 \\ \max(t(i,j,i_1,j_1,k_1),\ c(i,j)) & i = i_1 \\ tco(i+1,j,i_1,j_1,k_1) + 1 & i < i_1 \end{cases} \quad (3.47)$$

and

$$tco(i,j,i_1,j_1,k_1) = tci(i,j,i_1,j_1,k_1). \quad (3.48)$$

Solving these equations results in the array shown in Figure 3.8. This array is similar to the array proposed by Guibas, Kung, and Thompson [GiK79]. The time it takes to solve a problem of size n is 2(n−3). Table 3.2 shows the movement of c(1,2),c(1,3),c(1,4),c(1,5),c(1,6) assuming that n = 12. Notice that each data element will travel with a speed of 1 until it will be used by any computation. Then it will travel with a speed of 1/2 except in some prespecified processors (PE(1,3) andPE(1,9)) which will always have a speed of 1. This means that each PE has two buses, one with a delay of 1 for the unused data, and the other with a delay of 2 for the data that was used by this PE, except for processors PE(1,3) and PE (1,9) which has a delay of 1 for the two buses. Table 3.3 shows the movement of the same elements according to the design Guibas, Kung, and Thompson design [GuK79].

3.7.3. Examples with non-linear Data Dependencies

In this section, we discuss in detail two examples. The first, although has no physical meaning, was chosen because it has a very complicated data dependency. It shows the power of this method in dealing with complicated data dependencies. The second example is to show how this method can be applied in a case of non-linear data dependencies.

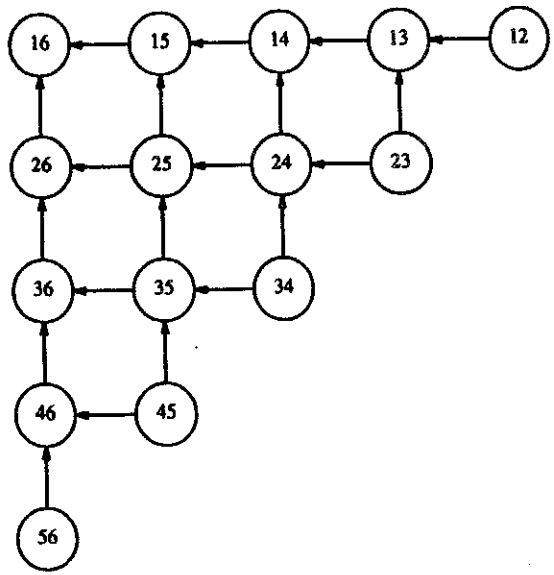


Figure 3.8

Systolic array for the dynamic programming problem

Table 3.2.
The time at which c(i,j) will reach PE (i,j)
using our proposed design

	PE										
c(i,j)	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12
c(1,2)	2	3	4	6	7	9	11	13	14	16	18
c(1,3)		3	4	5	6	8	10	12	13	15	17
c(1,4)			4	5	6	7	9	11	12	14	16
c(1,5)				6	7	8	9	10	11	13	15
c(1,6)					7	8	9	10	11	12	14

Table 3.3.
The time at which c(i,j) will reach PE(i,j)
using Guibas, Kung, and Thompson design [GuK79]

	PE										
c(i,j)	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12
c(1,2)	2	3	5	7	9	11	13	15	17	19	21
c(1,3)		4	5	6	8	10	12	14	16	18	20
c(1,4)			6	7	8	9	11	13	15	17	19
c(1,5)				8	9	10	11	12	14	16	18
c(1,6)					10	11	12	13	14	15	17

Example: Solve the following recurrence on a mesh connected systolic array.

$$f(i,j,k) = f(i+k, j+i-k, k-i) + f(i-1, j-i, k-1) \tag{3.49}$$

Assume that we projected on the k axis. Figure 3.9 shows the data dependency of recurrence equation 3.49 when n=3 in the I–J plane. Applying Strategy 1 to this equation, we get the solution in Figure 3.10 in which the systolic array is drawn for different values of k, and the time at which the calculations of f(i,j,k) is shown in the I–J plane.

Figure 3.11 shows the same situation applying Strategy 2, with the maximum number of buffers 0 set to 1.

Example: Solve the following system of recurrences using a mesh connected control flow systolic array.

$$f(i,j,0,k) = \Phi_1 \left\{ f(i-1, \max(j,0), 0, k-1),\ g(i, u(j-2), 0, k-1) \right\} \tag{3.50}$$

$$g(i,j,0,k) = \Phi_2 \left\{ f(i+1, jk-i-6, \frac{0}{j}, k-1),\ i, \frac{j0-3}{k}, \frac{0-10}{j}, k-1) \right\} \tag{3.51}$$

where u(x) is the unit step function.

First, we map this system into a 2-dimensional systolic array using the mapping $(i,j,0,k) \rightarrow (i,j)$, which means that each PE will perform n^2 computational steps to compute n^2 values of f and g. The data dependency represented as a graph is too complicated to be presented here. Using Strategy 1, and $1 \le i,j,l,k \le 4$ we get a the systolic array shown in Figure 3.12 where the tuple (x,y) shows the time to calculate f(i,j,l,k) and g(i,j,l,k) for $1 \le i,j,l,k \le 4$.

3.8. Experimental Results

In this section we show some experimental results for the running time of this method on randomly chosen samples of system of linear recurrences. We investigate equations on the form.

$$f_i(p) = \Phi_i \left\{ f_{i_j}(A_{i,j}p + b_{i,j}) \right\}$$

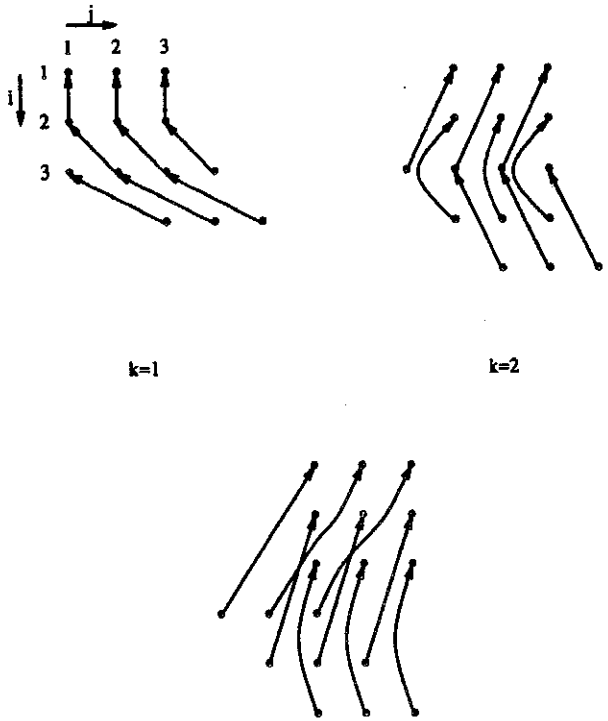


Figure 3.9

Data dependency for recurrence 3.49

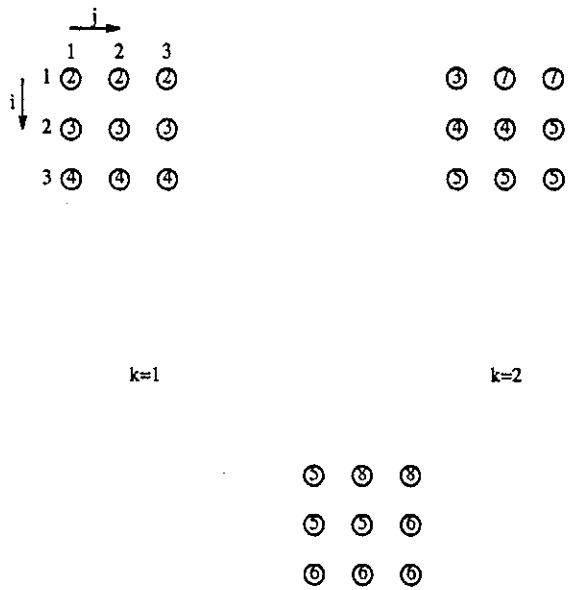


Figure 3.10

Systolic array for solving equation 3.49 using strategy 1

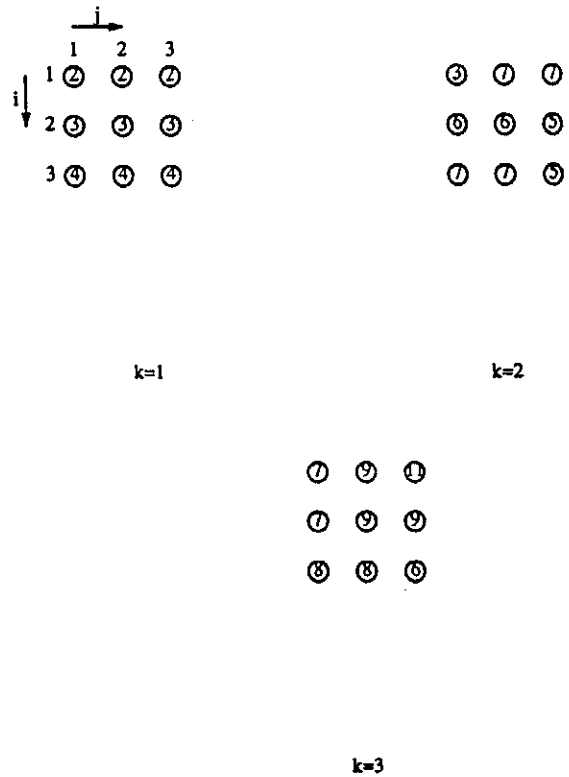


Figure 3.11

Systolic array for solving equation 3.49 using strategy 3

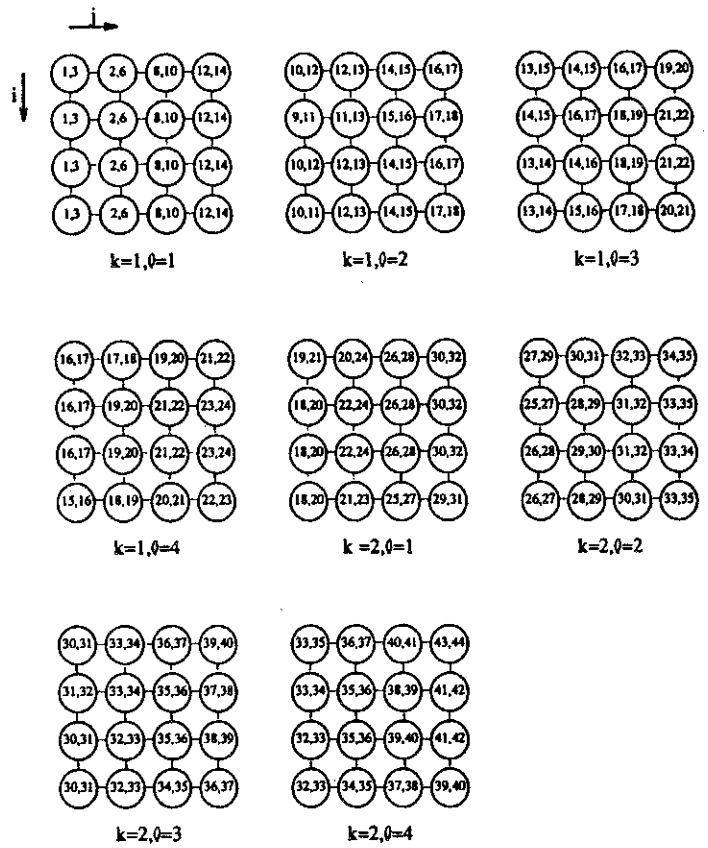


Figure 3.12

Systolic array for solving equation 3.50,3.51 using Stratrgy 1 (The numbers inside each PE represents the time at which f(i,j,l,k) and g(i,j,l,k) is computed

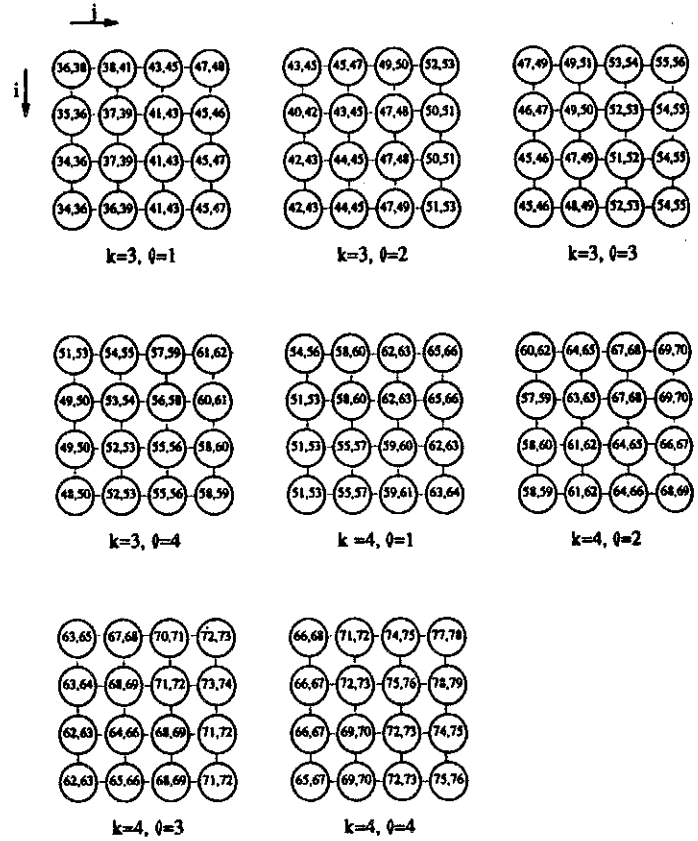


Figure 3.12 (Continued)

where i represents the number of equations in the system, j represents the number of arguments for a single equations, $p = (x,y,z)^T$, and $0 < x, y, z \leq 10$. To do this, we randomly generated the matrix $A_{i,j}$ and the vector $b_{i,j}$, and we ran the program for a large number of inputs until we got a 90% interval confidence (the number of runs is bettween 10 and 50). Table 3.3 shows the time to run this method on a VAX 11/780 for $i = 1, 2, 5$ and $j = 2, 3, 4, 5$. In this table μ represents the average time to run the program, and E represents the margin of errors to guarantee a 90% confidence (i.e. when running this program for a certain number of arguments and functions, we are 90% confident that it will take time between $\mu - E$ and $\mu + E$).

3.9. Conclusion

In this chapter, we have studied *Control Flow Systolic Array* which is capable of solving more complicated problems than the ordinary systolic arrays. We have also introduced a methodology to map the recurrence into the Control Flow Systolic Array. Finally we illustrate the power of this methodology by designing an array for the transitive closure problem. The resulting array is faster than any previously known arrays.

Table 3.4

The time needed to run our proposed method on a randomley choosen set of linear dependency recurrence equations

# of equations	2 arguments	3 arguments	4 arguments	5 arguments
1	μ=0.620 E=0.008	μ=0.900 E=0.022	μ=1.120 E=0.009	μ=1.700 E=0.017
2	μ=1.770 E=0.025	μ=3.690 E=0.077	μ=5.930 E=0.249	μ=7.430 E=0.301
5	μ=7.200 E=0.197	μ=11.540 E=1.583	μ=15.480 E=1.907	μ=19.350 E=1.195

CHAPTER 4
BUFFERING IN MACROPIPELINES OF SYSTOLIC ARRAYS

4.1. Introduction

This chapter will consider the architecture of a large system of systolic arrays, in which more than one systolic array cooporate together to solve a specific problem.

In a large system, especially in real-time applications, a pool of systolic arrays of different types can be configured into a macropipeline to solve a given problem. A *macropipeline* is a pipeline of systolic arrays with the outputs of one array acting as inputs to another array in the pipe. Each stage of the pipe is a systolic array that performs one operation, such as matrix addition or multiplication. This structure of macropipelines characterizes most image-processing algorithms [Nud80], [NuN83]. Examples include real-time vision systems [NiH79], analysis of motion [AgJ80], image reconstruction from projections [Far78], radar signal processing [Arm79], air traffic control [Han73], pattern analysis and image database management [FuH85], recursive filtering [ScK85], and pattern recognition [HwS83]. The Programmable Systolic Chip [FiK84] and the Warp array processor [KuM84] are examples of reconfigurable systolic arrays dedicated to handling compute-bound problems in image and digital signal processing. A number of these arrays can be used in a pipelined fashion to perform the various tasks in image and signal processing.

A *data distribution* of a systolic array is either the format of inputs fed into the systolic array or the format of outputs exiting the systolic array. The input data distribution of one systolic array may be different from the output data distribution of another. Hence, when two systolic arrays are connected together, it may be necessary

to convert the outputs of the systolic array that feeds data to the other array into thw required input data distribution form. A conventional approach for this problem is to use a common memory to buffer the outputs of the systolic arrays. This becomes a bottleneck, however, when many systolic arrays are sharing the common memory. Another approach is to design the systolic arrays such that the output format of one array is the same as the input format of the next array in the macropipeline and to connect the systolic arrays directly. This may not be always possible, especially when the macropipeline is reconfigurable. A third approach is to design a converter between two stages of the macropipeline, which consists of multiple buffers and a control unit to select the appropriate buffers for inputs and outputs [BrF82]. This approach is exemplified by MOSAIC [LiS86], a project carried out at ESL Inc. The system consists of a statically scheduled crossbar switch that connects multiple Warp processors, each with local memory modules, into a macropipeline. The local memory modules are used to store input data and to restructure them into the required input format. Since Warp processors are reconfigurable general-purpose systolic arrays, the memory requirement for each module is not defined at design time. Hence, all memory modules are of the same size.

The concept of using buffers to perform data conversion is illustrated in Figure 4.1a. C_1 and C_2 are converters used to convert the output data into the required input formats. Figure 4.1b shows this conversion. To convert data from distribution D_i to D_o, at least six buffers are needed. The first column of D_o cannot be output until the third column of D_i has arrived. Six buffers are needed to store the data in D_i such that elements in the first column of D_o are available in the buffers. Likewise, five buffers are needed for the second column, and three buffers are needed for the last. Note that the input and output rates may not be the same when the minimum number of buffers are used.

In a previous work by Wah and Shang [WaS85] the minimum number of buffers required to convert between any two distributions was calculated. this work will be extended in this chaoter to find the necessary transformations to convert the data between any two given distributions. The design of converters to interface systolic

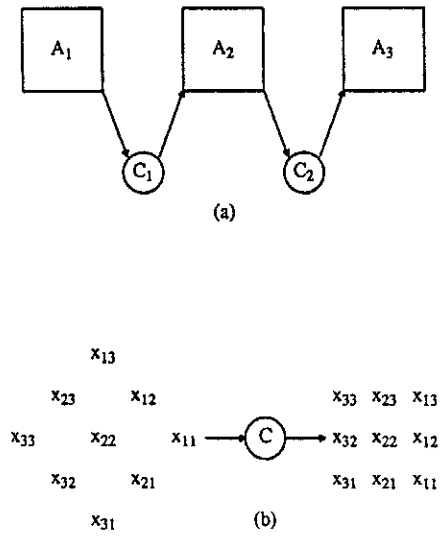


Figure 4.1

Macropipelining of systolic arrays. (a) A macropipeline of systolic arrays. (b) Conversion of data from one distribution to another.

arrays in a macropipeline will also be studied. This design depends on the type of macropipeline. A *static macropipeline* consists of a fixed pipeline of systolic arrays with a fixed function in each array. The conversion of data distributions between adjacent stages is fixed as well, and special-purpose converters are needed. In contrast, in a *dynamic macropipeline*, a subset of systolic arrays are selected from the pool and configured into a pipeline depending on the application. Since the configuration of a dynamic macropipeline may not be fixed and data of different formats may be fed into a given array, general-purpose converters are needed here.

The objective of this chapter is to provide a methodology to design an efficient converter for given input and output distributions. It is assumed that both the inputs and the outputs are two-dimensional arrays in which the elements are equally-spaced along the rows and columns in the data distributions, that there are no duplicated data in the distribution, and that data can be described by two vectors to be discussed in Section 4.3.1. The macropipeline is asynchronous, and the interarrival times of data may be different for different systolic arrays. In the remaining sections, we will study the minimum number of buffers for a given conversion [WaS85] [WaA88], propose design procedures for general-purpose and special-purpose converters, and exemplify the design process.

4.2. Minimum Number of Buffers

A converter is made up of buffers, the interconnections among the buffers, and the necessary control hardware that issues signals to buffers to accept or send data at the proper times. B_{min} is defined as the minimum number of buffers in a converter to buffer incoming data before they are output. In this section, an algorithm will be presented to find B_{min} for given input and output distributions.

4.2.1. Data Distributions

To describe different data distributions, two vectors are introduced in [LiW85a]. Suppose that the row and column indices of X are i and j, respectively. The row vector of X is defined as the directional distance between $x_{i,j}$ and $x_{i+1,j}$ and is denoted by

$\vec{I}$. Similarly, the column vector of X, denoted by $\vec{J}$, is defined as the directional distance between $x_{i,j}$ and $x_{i,j+1}$. A data distribution with vectors $\vec{I}$ and $\vec{J}$ is denoted by $D(\vec{I}, \vec{J})$. Two data distributions are illustrated in Figure 4.2.

The geometric layout of a data distribution can be described in the Cartesian plane. Without loss of generality, it is assumed that $x_{1,1}$ for both the input and output distributions is placed at the origin, that $x_{1,1}$ is among the data items to be input and output first, and that data is moving in the direction of the positive x-axis. Vectors $\vec{I}$ and $\vec{J}$ are used to define a data distribution and to determine the locations of its elements uniquely. $C_x(i,j)$ and $C_y(i,j)$ denote the x and y coordinates of element $x_{i,j}$. I_x and J_x are the projections of vectors $\vec{I}$ and $\vec{J}$ on the x-axis. Likewise, I_y and J_y are the corresponding projections on the y-axis. Therfore,

$$C_x(i, j) = (i-1)\,I_x + (j-1)\,J_x \tag{4.1}$$

$$C_y(i, j) = (i-1)\,I_y + (j-1)\,J_y \tag{4.2}$$

Note that if I_x, J_x, I_y, and J_y are integers, then the coordinates will be integers.

In the Cartesian-coordinate representation we have adopted, the x-coordinate indicates timing. That is, elements with the same x-coordinate arrive at (or depart from) the converter at the same time. Data with the smallest x-coordinates arrive at (or depart from) the converter first, while data with the largest x-coordinates arrive (or depart) last. The *i'th* (*input* or *output*) *step* is defined as the set of elements in the (input or output) distribution with the x-coordinate equal to i.

4.2.2. Finding the Minimum Number of Buffers

In this section a dynamic-programming formulation is developed to find B_{min}, the minimum number of buffers to convert the data distribution from D_i to D_o. Let b_i be the number of buffers needed after the (i-1)'th output step has been carried out and before the i'th step of D_o can be output, while the necessary data to output in the i'th output step have been received. In deriving b_i, it is assumed that all input data items are buffered before they can be output. Further, let B_i be the maximum number of buffers needed when the i'th step of D_o is output, and the boundary conditions are

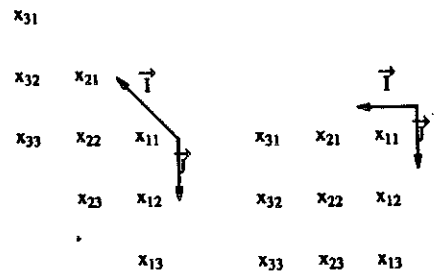


Figure 4.2

Two data distributions and their corresponding vectors. (The first data distribution has three streams of dataflow, and the second one has five streams, assuming that data are moving from left to right.)

B_0=0. For the example in Figure 4.1, before $x_{1,1}$, $x_{1,2}$, and $x_{1,3}$ can be output, b_1 (= 6) buffers are needed to buffer the first three columns of D_i. Before the second output step of D_o can be carried out, data in the first four input steps with eight data items must have been received. Hence, b_2 is five assuming that the first output step has been completed. Similarly, b_3 = 3. As a result,

$$B_0 = 0; \tag{4.3a}$$

$$B_i = \max\{b_i, B_{i-1}\}; \quad i = 1,2,3. \tag{4.3b}$$

$$B_{min} = B_3 = \max\{b_3, \max\{b_2, \max\{b_1, B_0\}\}\} = b_1 = 6. \tag{4.4}$$

Note that in deriving the minimum number of buffers, the input and output clocks may be running at different rates.

To allow a more precise formulation, two partitions on the data set $X = \{x_{i,j}: 1 \le i,j \le n\}$ and a partial ordering of these partitions are introduced.

An *input partition* partitions the input array X into N_i disjoint subsets, I_p, $1 \le p \le N_i$, where

$$I_p = \{ x_{i,j} \mid C_x(i,j) = (i-1)I_x^i + (j-1)J_x^i = a_p \}, \quad p = 1, ..., N_i, \tag{4.5}$$

and N_i is the number of input steps. $\vec{I}^i$ and $\vec{J}^i$ are the row and column vectors of the input distribution, respectively, and I_x^i and J_x^i are the corresponding projections on the x-axis. I_p represents the set of input elements with the same x-coordinate a_p. I_1 is the set of input data that arrive at the converter first, and I_p is the p'th arrival set.

An *output partition* partitions the output array X into N_o disjoint subsets O_k, $1 \le k \le N_o$, where

$$O_k = \{ x_{i,j} \mid C_x(i,j) = (i-1)I_x^o + (j-1)J_x^o = a_k \}, \quad k = 1, ..., N_o, \tag{4.6}$$

and N_o is the number of output steps for the output distribution. $\vec{I}^o$ and $\vec{J}^o$ are the vectors of the output distribution, respectively, and I_x^o and J_x^o are the corresponding projections on the x-axis. O_k represents output elements with the same x-coordinate a_k. O_1 represents the set of data that departs from the converter first, and O_k is the k'th departure set.

Figure 4.3a shows D_i and D_o of a 3-by-3 array X. Since X arrives in three steps, there are three input partitions (i.e., N_i = 3). Similarly, there are seven output partitions (i.e., N_o = 7) because the outputs depart in seven steps. Let S and O be the sets of input and output partitions and Π be their union.

$$S = \{ I_p \mid 1 \le p \le N_i \} \tag{4.7}$$

$$O = \{ O_k \mid 1 \le k \le N_o \} \tag{4.8}$$

$$\Pi = S \cup O \tag{4.9}$$

For $\pi_i \in \Pi$, $|\pi_i|$ represents the number of elements in π_i. The corresponding input and output partitions in Figure 4.3a are shown in Figure 4.3b.

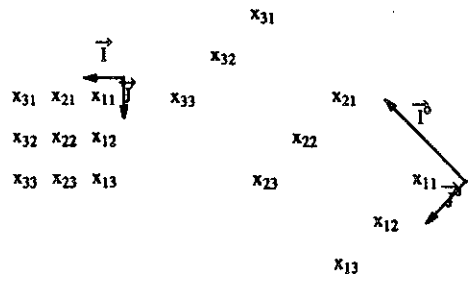
The example in Figure 4.3 shows that there exists a relationship between the O_ks and I_ps. A partial ordering "$\rightarrow$" can be defined on Π as follows. If $I_k \rightarrow I_p$, then data in I_p will arrive earlier than that of I_k. If $O_k \rightarrow O_p$, then data in O_p will leave earlier than that of O_k. Further, if $O_k \rightarrow I_p$, then data in I_p must arrive before data in O_k can depart. To output the elements in O_p, all elements in I_ps such that $O_k \cap I_p \neq \emptyset$ must have arrived at the converter. In short,

(1) $I_k \rightarrow I_p$ if $k > p$;

(2) $O_k \rightarrow O_p$ if $k > p$;

(3) $O_k \rightarrow I_p$ if either $O_k \cap I_p \neq \emptyset$ or there exists an integer q such that $I_q \cap O_k \neq \emptyset$ and that $I_q \rightarrow I_p$.

The above definitions imply that if $O_k \rightarrow I_p$, then $O_k \rightarrow I_{p-1} \rightarrow I_{p-2} \rightarrow \cdots \rightarrow I_1$. The integer q_k=p such that $O_k \rightarrow I_p$ and that $O_k \not\rightarrow I_{p+1}$ is defined as the *key number* for O_k. Note that all the relationships among the I_ps and O_k will be known once q_k is found. The partial ordering of the partitions can be represented in a lattice. Figure 4.3c shows the lattice of the partial ordering for the example in Figure 4.3a. For instance, q_3 = 2 is the key number for O_3 since $O_3 \cap I_2 = \{x_{2,1}\}$, and O_3 can be output once elements in I_2 have arrived.



(a)

$S = \{I_1, I_2, I_3\}$ $O = \{O_1, \ldots, O_7\}$

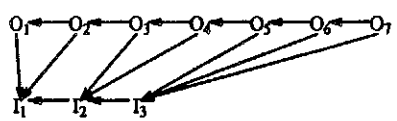
$I_1 = \{x_{1,1}, x_{1,2}, x_{1,3}\}$, $O_1 = \{x_{1,1}\}$, $O_2 = \{x_{1,2}\}$,

$I_2 = \{x_{2,1}, x_{2,2}, x_{2,3}\}$, $O_3 = \{x_{2,1}, x_{1,3}\}$, $O_4 = \{x_{2,2}\}$,

$I_3 = \{x_{3,1}, x_{3,2}, x_{3,3}\}$ $O_5 = \{x_{3,1}, x_{2,3}\}$,

$O_6 = \{x_{3,2}\}$, $O_7 = \{x_{3,3}\}$

(b)



(c)

Figure 4.3
Partitioning and partial ordering. (a) Input distribution D_i and output distribution D_o. (b) Input and output partitions. (c) Lattice for the partial ordering (transitive arcs are not shown).

To use dynamic programming to find B_{min}, $O_1, O_2, \ldots, O_k$ are examined sequentially. It is assumed that the inputs and outputs may be driven by different clocks. That is, the minimum amount of data are input to generate the necessary outputs. If q_k is the key number for O_k, then $I_1, I_2, \ldots, I_{q_k}$ must have arrived at the converter before elements in O_k can depart. The reason is that either I_p, $1 \le p \le q_k$, contains data in O_k, or I_p does not contain data in O_k but $I_{q_k} \rightarrow I_p$. Therefore, elements in the set $I_1 \cup I_2 \cup \cdots \cup I_{q_k}$ that remain after I_{q_k} has arrived and O_{k-1} has left must be buffered. In other words, the number of buffers needed, b_k, is

$$b_k = \sum_{i=1}^{q_k} |I_i| - \sum_{i=1}^{k-1} |O_i| \qquad k = 1, \ldots, N_o \tag{4.10}$$

By the Principle of Optimality, which states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision, we can formulate the problem in dynamic programming as follows.

$$b_0 = 0; \qquad |O_0| = 0; \qquad q_0 = 0; \tag{4.11a}$$

$$b_k = b_{k-1} - |O_{k-1}| + \sum_{j=q_{k-1}+1}^{q_k} |I_j| \qquad k = 1, \ldots, N_o \tag{4.11b}$$

$$B_0 = 0; \tag{4.12a}$$

$$B_k = \max(b_k, B_{k-1}) \qquad k = 1, \ldots, N_o \tag{4.12b}$$

Note that the summation in equation 4.11b is zero if the lower limit is greater than the upper limit.

To establish the partial ordering of partitions, a counter is used to count the number of elements in each partition, and the key number q_k is kept for each O_k. $C_k^i(i,j)$ and $C_k^o(i,j)$ are computed for every element $x_{i,j}$ in the input and output distributions. If $x_{i,j}$ is in O_k and I_p, then $O_k \rightarrow I_p$, and the counters for O_k and I_p are incremented. q_k is updated to p if p is larger than the previous value of q_k. The algorithm to compute the partial ordering is shown in Figure 4.4. The computational

```
procedure compute_partial_ordering;

/* Inputs:   I^1 and J^1: vectors for input distribution;
             C_x^i(1,1) = C_x^o(1,1) = 0, and all C_x^i(i,j) and C_x^o(i,j) are integers;
             I^0 and J^0: vectors for output distribution;
             N_i: number of input steps;
             N_o: number of output steps;
  Outputs:   Three arrays Φ(N_0), Q(N_0), S(N_i), where
             Φ(k) = |O_k|, the number of elements in O_k;
             Q(k) = q_k, the key number for O_k;
             S(p) = |I_p|, the number of elements in I_p.  */
(1) Initialize Φ, Q, and S to zeroes.
(2) for i=1 to n do [
        for j=1 to n do [
            k := C_x^o(i,j) + 1;   Φ(k) := Φ(k) + 1;
            p := C_x^i(i,j) + 1;   S(p) := S(p) + 1;
            Q(k) := max {Q(k), p}
        ]
    ]
```

Figure 4.4

Algorithm to compute the partial ordering of partitions.

complexity of the algorithm is $O(n^2)$, since all elements in D_i and D_o must be considered. A better algorithm with a computational complexity of $O(\max\{N_i, N_o\} \cdot \min\{I_x^o, J_x^o\})$ can be devised but will not be presented here.

The example in Figure 4.3a is used to illustrate the algorithm. Initially, all key numbers are initialized to zeroes, $N_i = 3$, and $N_o = 7$. Since $C_x^o(1,1)=0$, and $C_x^i(1,1)=0$. $x_{1,1} \in O_1$ and $x_{1,1} \in I_1$. q_1 and the counters for O_1 and I_1 are updated to ones. Similarly, it is found that $x_{1,3} \in O_3$ and $x_{1,3} \in I_1$. The counters for O_3 and I_1 are incremented, and q_3 is set to one. For $x_{2,1}$, it is found that $x_{2,1} \in I_2$ and that $x_{2,1} \in O_3$. The counters for O_3 and I_2 are incremented, and q_3 is set to $\max\{q_3, 2\} = 2$. Likewise, the remaining elements in X can be examined. The results of applying the algorithm in Figure 4.4 are

$$\Phi = [1, 1, 2, 1, 2, 1, 1]; \quad Q = [1, 1, 2, 2, 3, 3, 3]; \quad S = [3, , 3, 3].$$

Applying equation 4.11 results in $[b_k] = [3, 2, 4, 2, 4, 2, 1]$. From equation 4.12, we obtain $B_7 = 4$.

4.3. Combinations Of Data Distributions

In this section, we will discuss some properties of data distributions that are useful for designing the converters. As mentioned before, a data distribution is characterized by two non-parallel vectors. Two data distributions, $D(\vec{I}^1, \vec{J}^1)$ and $D(\vec{I}^2, \vec{J}^2)$, are said to be *equivalent* (or belong to the same *equivalence class*) if

$$I_x^1 = I_x^2 \text{ and } J_x^1 = J_x^2 \text{ and} \tag{4.13a}$$

$$(I_x^1 J_y^1 - I_y^1 J_x^1) \cdot (I_x^2 J_y^2 - I_y^2 J_x^2) > 0 \tag{4.13b}$$

where I_x^i (resp. J_x^i) is the projection of $\vec{I}^i$ (resp. $\vec{J}^i$) on the x-axis. The first condition (equation 4.13a) ensures that the data distributions have the same projections on the x-axis. Consequently, the orders in which data arrive at the systolic array for the two data distributions are identical. The second condition (equation 4.13b) ensures that the data arriving at the systolic array at the same time have the same permutations. Note that for two vectors, $\vec{I}$ and $\vec{J}$, if θ is the angle in a clockwise direction from $\vec{I}$ to $\vec{J}$,

then $\sin(\theta)=(I_xJ_y-I_yJ_x)/(|\vec{I}|\cdot|\vec{J}|)$. The number of streams of data flowing into the systolic array for two equivalent data distributions may not be equal and can range from n to (2n−1). As an example, the two data distributions in Figure 4.2 are equivalent, but have different number of streams of dataflow.

The following theorem shows the number of possible equivalence classes of data distributions.

Theorem: There are $\Omega(2^n)$ equivalence classes of data distributions for an n-by-n array of data.

Proof: In proving the number of equivalence classes, only the projections of the vectors on the x-axis have to be considered. Without loss of generality, assume that $\vec{J}$ is not orthogonal to the x-axis. From the x-projections of the first row of data, $C_x(1,1)$, $C_x(1,2)$, ..., $C_x(1,n)$, it is necessary to determine the number of possible x-projections for the remaining rows. Consider the x-projection of $x_{2,1}$. Assuming that $C_x(2,1)\geq C_x(1,1)$, there are 2n possible positions for $C_x(2,1)$, namely, $C_x(2,1) = C_x(1,i)$, i=1, ..., n, $C_x(1,i) < C_x(2,1) < C_x(1,i+1)$, i=1, ..., n−1, and $C_x(2,1) > C_x(1,n)$. Suppose that $C_x(1,1) < C_x(2,1) < C_x(1,2)$. In this case there are three possibilities for $C_x(3,1)$, namely, $C_x(3,1) = C_x(1,2)$, $C_x(2,1) < C_x(3,1) < C_x(1,2)$, and $C_x(1,2) < C_x(3,1) < C_x(1,3)$ (see Figure 4.5). When $C_x(3,1) = C_x(1,2)$, the positions of the remaining elements are determined. However, when either $C_x(2,1) < C_x(3,1) < C_x(1,2)$ or $C_x(1,2) < C_x(3,1) < C_x(1,3)$, then $C_x(4,1)$ can fall in three possible ranges, as shown on the second level of the tree in Figure 4.5. The same argument can be applied to the remaining levels of the tree for $x_{5,1}$, ..., $x_{n,1}$. In level ℓ, $1<\ell<n-3$, there are $2^{\ell-1}$ terminals, while in level n−2, there are $3\cdot2^{n-3}$ terminals. The total number of terminals is

$$\sum_{\ell=1}^{n-3} 2^{\ell-1} + 3\cdot2^{n-3} = 2^{n-3}-1+3\cdot2^{n-3} = 4\cdot2^{n-3}-1 \qquad (4.14)$$

A similar argument can be made when $C_x(1,1) > C_x(2,1)$ or $C_x(2,1) > C_x(1,2)$. Since each of the above data distributions belong to a distinct equivalence class, the total number of possible data distributions is $\Omega(2^n)$. □

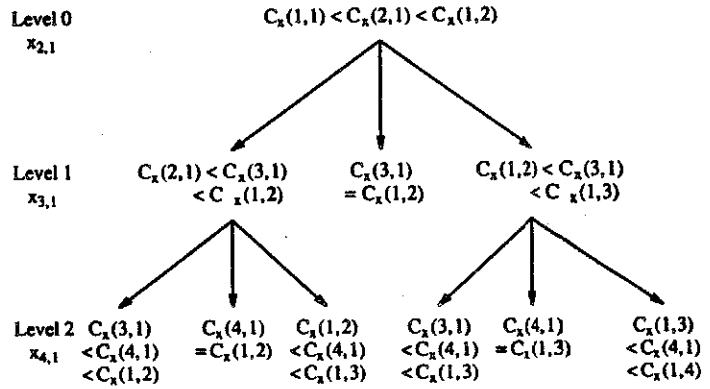


Figure 4.5

Possible positions for $x_{2,1}$, $x_{3,1}$, ..., $x_{n,1}$.

It is practically impossible to design a general-purpose converter to perform all the possible data transformations. Some restrictions are necessary to reduce the space of data distributions.

If vectors $\vec{I}$ and $\vec{J}$ are restricted to have only unitary or zero projections on the x- and y-axis, then there will be eight possible directions for $\vec{I}$ pointing at 0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315°. For each direction of $\vec{I}$, there are six possible directions for $\vec{J}$, excluding the cases in which $\vec{I}$ and $\vec{J}$ are pointing in the same or opposite directions. Thus, there are 8×6=48 possible combinations of data distributions (Figure 4.6a). Out of these 48 cases, there are only 16 equivalence classes (distributions in the same column of Figure 4.6a belong to the same equivalence class).* Data distributions in class i', $1 \le i' \le 8$, can further be combined with the corresponding data distributions in class i, $1 \le i \le 8$, into a new equivalence class if a reversal circuit is available to reverse the order of data arriving simultaneously at the converter. The resulting eight standard distributions are shown in Figure 4.6b.

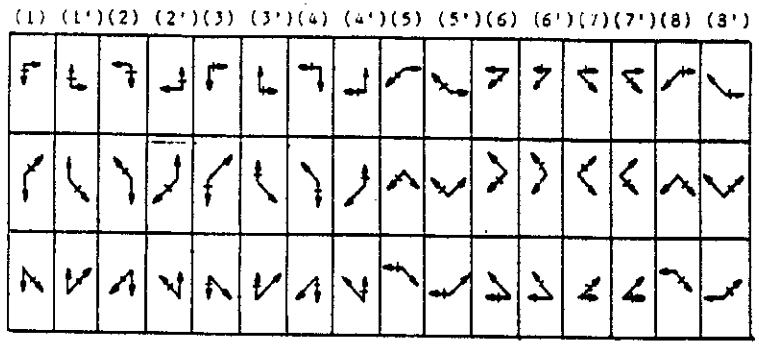
4.4. General-Purpose Converters

In this section, we discuss the design of a general-purpose converter that can convert data from any distribution to any other distribution provided that the vectors representing the data distributions have zero or unitary projections on the x- and y-axes. We will give the mathematical representation of this converter as a series of transformations applied to the input data distribution to produce the required output distribution. It is assumed that vector $\vec{I}$ is represented by the corresponding projections on the x and y axes. That is,

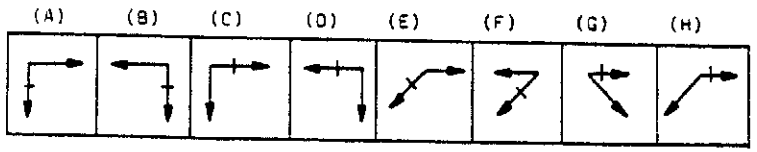
$$\vec{I} = I_x\vec{x} + I_y\vec{y} = [I_x, I_y]^T \tag{4.15}$$

$\vec{J}$ can be represented similarly. The data-distribution vectors are represented as a 2-by-2 matrix $D = [\vec{I}, \vec{J}]$. A transformation process T is a two-by-two matrix, and a

* Note that elements in each equivalence class are not unique according to Eq. (4.13b). Figure 4.6a shows one of the possible sets of equivalence classes.



(a) All possible combinations of data distributions.



(b) Eight standard output data distributions.

Figure 4.6

Possible data distribution when $\vec{I}$ and $\vec{J}$ have either zero or unitary projections on the x- and y- axis

transformation on a data distribution is the product of the transformation matrix and the corresponding matrix representation of the data-distribution vectors. It is further assumed that the input distribution $D_i(\vec{I}^i, \vec{J}^i)$ and the output distribution $D_o(\vec{I}^o, \vec{J}^o)$ are given, and that they belong to one of the 48 data distributions in Figure 4.6a. Figure 4.7 shows the transformation process.

The first transformation is on reducing the number of data streams from the outputs of the previous stage of the macropipeline. It is assumed that the input matrix enters the converter in n streams, that is, the distributions in the first row or columns (5) thru (8′) in the third row of Figure 4.6a are used. If the output data from the previous stage require more than n streams, then both vectors $\vec{I}^i$ and $\vec{J}^i$ of this data distribution must have nonzero x and y projections, that is, the distribution belongs to those in the second row or columns (1) thru (4′) in the third row of Figure 4.6a. In this case, the output data are multiplexed into n streams using n multiplexers before they are output from the previous stage (Figure 4.8a). This multiplexing is equivalent to a linear transformation T_1.

$$T_1 = \begin{bmatrix} 1 & 0 \\ \alpha_1 & \beta_1 \end{bmatrix}, \tag{4.16}$$

which changes the data distribution in the second row or columns (1) thru (4′) in the third row of Figure 4.6a into that of the first row or columns (5) thru (8′) in the third row. The data distribution after multiplexing can be represented by $D_1(\vec{I}^1, \vec{J}^1)$.

$$D_1 = T_1 \cdot D_i \quad \text{where} \quad \begin{cases} \alpha_1 = \dfrac{-I_y^i J_y^i}{I_x^i J_y^i - I_y^i J_x^i}, \ \beta_1 = -\alpha_1 \dfrac{I_x^i}{I_y^i} & \text{if } I_x^i, I_y^i, J_x^i, J_y^i \neq 0 \\ \alpha_1 = -\dfrac{J_y^i}{J_x^i}, \ \beta_1 = 1 & \text{if } I_x^i = 0,\ J_x^i \neq 0,\ J_y^i \neq 0 \\ \alpha_1 = -\dfrac{I_y^i}{I_x^i}, \ \beta_1 = 1 & \text{if } I_x^i \neq 0,\ I_y^i \neq 0,\ J_x^i = 0 \\ \alpha_1 = 0, \ \beta_1 = 1 & \text{otherwise} \end{cases} \tag{4.17}$$

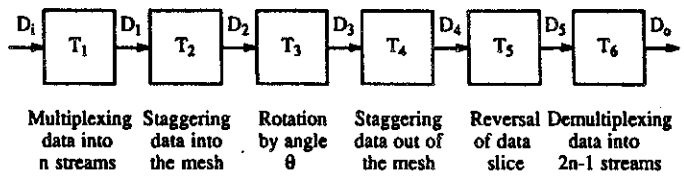


Figure 4.7

The sequence of transformations in the conversion process.

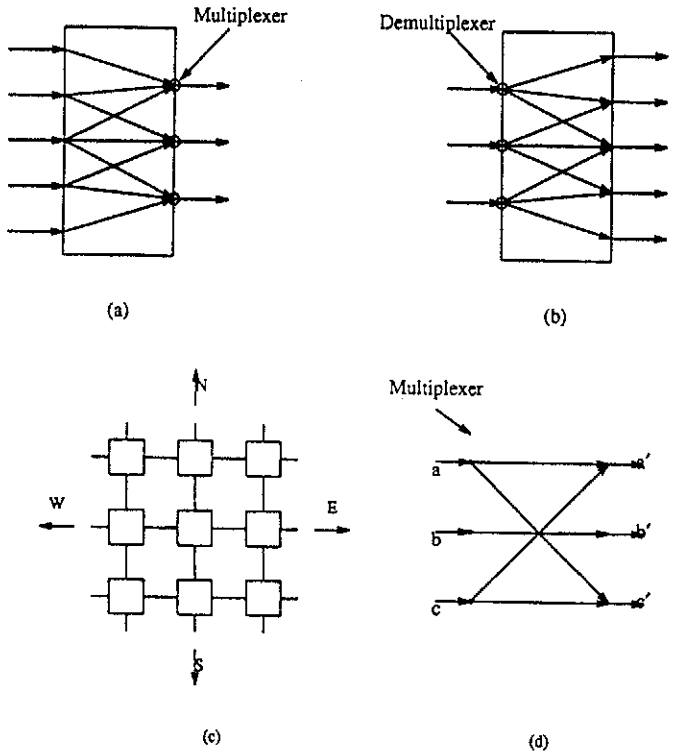


Figure 4.8

Architecture of the general-purpose converter (n=3). (a) Multiplexing data from (2n−1) streams into n streams using n multiplexres (N=3) (b) Demultiplexing data from n streams into (2n−1) streams using n demultiplexers (n=3) (c) 3-by-3 mesh of buffers (d) Reversal network

The n-by-n array of data is routed into an n-by-n mesh of buffers with four-neighbor connections until they are filled (Figure 4.8c). If the data distribution belongs to one of those in columns (5) thru (8′) in rows 1 and 3 of Figure 4.6a, then it needs to be transformed into one of the data distributions in column (1) thru (4′) in row 1, which is the distribution of data that can be stored in the mesh of buffers. This transformation is represented by T_2,

$$T_2 = \begin{bmatrix} \alpha_2 & \beta_2 \\ 0 & 1 \end{bmatrix} \tag{4.18}$$

which transforms $D_1(\vec{I}^1, \vec{J}^1)$ into $D_2(\vec{I}^2, \vec{J}^2)$.

$$D_2 = T_2 \cdot D_1 \quad \text{where} \quad \begin{cases} \alpha_2 = \dfrac{1}{|I_x^1|}, \ \beta_2 = -\alpha_2 \dfrac{J_x^1}{J_y^1} & \text{if } I_y^1 = 0 \\ \alpha_2 = \dfrac{1}{|J_x^1|}, \ \beta_2 = -\alpha_2 \dfrac{I_x^1}{I_y^1} & \text{if } J_y^1 = 0 \end{cases} \tag{4.19}$$

The two-dimensional interconnections in the buffers allow data to be shifted in one of the four directions. Data are input in one direction and may come out from any one of the four directions. Accessing data in one of the four directions can be represented as a rotation of vectors $\vec{I}$ and $\vec{J}$ by an angle θ_3, where θ_3 is 0°, 90°, 180°, or 270°. This rotation can be represented as transformation T_3,

$$T_3 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) \end{bmatrix} \quad \theta_3 = 0°, 90°, 180°, 270°, \tag{4.20}$$

which transforms $D_2(\vec{I}^2, \vec{J}^2)$ into $D_3(\vec{I}^3, \vec{J}^3)$.

$$D_3 = T_3 \cdot D_2, \tag{4.21}$$

where

$$\theta_3 = \begin{cases} 90^\circ & \text{if } (I_x^2, I_y^\circ=0 \text{ and } I_x^\circ=-I_y^2) \text{ or } (I_x^\circ, I_y^2=0 \text{ and } J_x^\circ=-J_y^2) \text{ or} \\ & (J_x^2, J_y^\circ=0 \text{ and } J_x^\circ=-J_y^2) \text{ or } (J_y^2, J_x^\circ=0 \text{ and } I_x^\circ=-I_y^2) \\ 180^\circ & \text{if } (I_x^\circ=-I_x^2 \text{ and } I_y^2=0 \text{ and } |J_y^\circ| = |J_y^2|) \text{ or} \\ & (J_x^\circ=-J_x^2 \text{ and } J_y^2=0 \text{ and } |I_y^2| = |I_y^\circ|) \\ 270^\circ & \text{if } (I_x^2, I_y^\circ=0 \text{ and } I_x^\circ=I_y^2) \text{ or } (I_y^2, I_x^\circ=0 \text{ and } J_x^\circ=I_y^2) \text{ or} \\ & (J_x^2, J_y^\circ=0 \text{ and } J_x^\circ=J_y^2) \text{ or } (J_y^2, J_x^\circ=0 \text{ and } I_x^\circ=I_y^2) \\ 0^\circ & \text{otherwise.} \end{cases} \quad (4.22)$$

T_3 is equivalent to transforming one of the distributions (1) thru (4′) in the first row of Figure 4.6a to another one in the same set.

In conjunction with the rotation, the shifts of data from the buffers may be controlled by different clocks to allow staggering of data in different rows. This is equivalent to transforming one of the distributions (1) thru (4′) in the first row of Figure 4.6a into one of the distributions in columns (5) thru (8′) in rows 1 and 3. This transformation can be represented by T_4,

$$T_4 = \begin{bmatrix} \alpha_4 & \beta_4 \\ 0 & 1 \end{bmatrix}, \quad (4.23)$$

where β_4 is the time difference between the output of the first element in row i and the first element in row (i+1). T_4 transforms $D_3(\vec{I}^3, \vec{J}^3)$ into $D_4(\vec{I}^4, \vec{J}^4)$.

$$D_4 = T_4 \cdot D_3 \quad \text{where} \quad \begin{cases} \alpha_4 = I_x^\circ / I_x^3, \ \beta_4 = J_x^\circ / J_y^3 & \text{if } J_y^3 \neq 0 \\ \alpha_4 = J_x^\circ / J_x^3, \ \beta_4 = I_x^\circ / I_y^3 & \text{if } I_y^3 \neq 0. \end{cases} \quad (4.24)$$

Note that if the distributions are limited to those in Figure 4.6a, then $\beta_4 = 0$ or ± 1. However, T_4 can also be applied to more general data distributions that will be discussed in Section 4.6.

Next, data may be routed through a reversal network to obtain the proper permutation (Figure 4.8d). The reversal network maps data with output distribution (i) into that of (i′) in Figure 4.6a. This can be represented by a transformation T_5,

$$T_5 = \begin{bmatrix} 1 & 0 \\ 0 & \alpha_5 \end{bmatrix} \quad (4.25)$$

which maps $D_4(\vec{I}^4, \vec{J}^4)$ into $D_5(\vec{I}^5, \vec{J}^5)$.

$$D_5 = T_5 \cdot D_4 \quad \text{where} \quad \alpha_5 = \begin{cases} \dfrac{J_y^4}{|J_y^4|} \dfrac{J_y^\circ}{|J_y^\circ|} & \text{if } J_y^4 \neq 0 \\ \dfrac{I_y^4}{|I_y^4|} \dfrac{I_y^\circ}{|I_y^\circ|} & \text{if } I_y^4 \neq 0 \end{cases} \quad (4.26)$$

Before data exit the converter and are sent into the systolic array in the macro-pipeline, they may be demultiplexed by n demultiplexers from n streams into (2n−1) streams as shown in Figure 4.8b. This can be represented as a transformation T_6.

$$T_6 = \begin{bmatrix} 1 & 0 \\ \alpha_6 & \beta_6 \end{bmatrix}, \quad (4.27)$$

which maps $D_5(\vec{I}^5, \vec{J}^5)$ into the output distribution $D_o(\vec{I}^o, \vec{J}^o)$.

$$D_o = T_6 \cdot D_5 \quad \text{where} \quad \begin{cases} \alpha_6 = \dfrac{I_y^\circ}{I_x^5}, \ \beta_6 = 1 - \alpha_6 \dfrac{J_x^5}{J_y^5} & \text{if } I_y^5 = 0 \\ \alpha_6 = \dfrac{J_y^\circ}{J_x^5}, \ \beta_6 = 1 - \alpha_6 \dfrac{I_x^5}{I_y^5} & \text{if } J_y^5 = 0 \end{cases} \quad (4.28)$$

The transformations described above are sufficient to transform any one of the 48 distributions in Figure 4.6a to any other distribution in the same figure. By using multiplexers (T_1) and by controlling the timing of different rows of data input into the mesh of buffers (T_2), any one of the 48 distributions in Figure 4.6a can be transformed into a distribution represented by vectors in the x and y directions only. Note that these distributions belong to one of those in columns (1) thru (4′) in row 1 of Figure 4.6a. To transform between any two distributions represented by vectors in the x and y directions, a rotation of angle θ, where θ is 0°, 90°, 180°, or 270°, is needed. This can be achieved by selecting the direction to output the data in the mesh

of buffers (T_3) and a reversal network (T_5). Likewise, by using demultiplexers (T_6) and by controlling the timing of the different rows of data output from the mesh of buffers (T_4), one of the data distributions in columns (1) thru (4') in row 1 of Figure 4.6a can be transformed into any one of the 48 distributions.

The above design requires the entire matrix to be stored in the buffers before they are output. This simplifies the control but increases the delay. An alternative design uses demultiplexers to input data into selected buffers other than those on the perimeter. n demultiplexers, d_1, d_2, ..., d_n, are added to the n rows of buffers in Figure 4.8c. In the resulting design shown in Figure 4.9, d_1 and d_n are two-way demultiplexers, while the rest are four-way demultiplexers. For buffers in row i, $1<i<n$, the four output lines of d_i are connected to cells 1, i, (n−i+1), and n. These connections are used to adjust the dataflow by outputting data as soon as possible and to obtain output distributions in columns (5) thru (8') in rows 1 and 3 of Figure 4.6a. Note that this is equivalent to applying T_4 to the data distribution with $\beta_4 = 1$, if we route d_i to cell (n−i+1) and with $\beta_4 = -1$ if we route d_i to cell i. For example, to convert from input distribution (1) to output distribution (5) in the first row of Figure 4.6a, demultiplexer d_i is connected to cell (n−i+1), $1 \le i \le n$. Elements in the first row will stay in the buffers for one time unit, while elements in the i'th row will go through i buffers and, hence, will stay in the buffers for i time units. Data will be output in the eastern direction.

4.5. Special-Purpose Converters

This section will discuss the heuristic design of special-purpose converters. An optimal design of these converters is difficult because they are problem dependent.

The conversion between any pair of the eight standard distributions in Figure 4.6b is straightforward and is illustrated in the following examples. To convert from distribution (A) to distributions (B), (C), or (D), n^2 buffers are needed. The input data are propagated from left to right and are output in the western, southern, or northern direction after the buffers are filled. To convert from distribution (A) to (E), n(n+1)/2 buffers are arranged as shown in Figure 4.10a. The conversion from distribution (A)

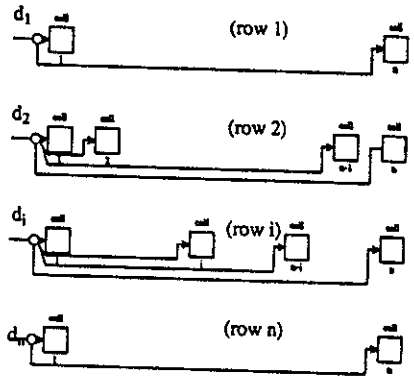
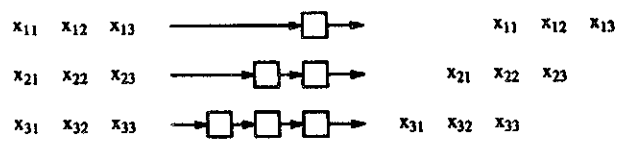
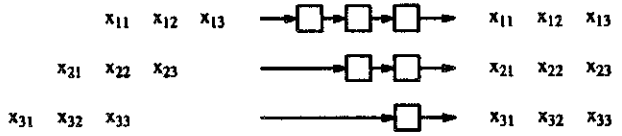


Figure 4.9

Organization of buffers for the modified general-purpose converter.



(a)



(b)

Figure 4.10

Special-purpose converter. (a) Conversion from data distribution (A) to (E) (n=3). (b) Conversion from data distribution (E) to (A) (n=3).

to (F) needs n^2 buffers. Data are output from the west after the buffers are filled. Data in row i are output one step ahead of that of row (i+1). The conversion from (A) to (G) is similar to that from (A) to (E), and that from (A) to (H) is similar to that from (A) to (F). The conversion from distribution (E) to (A) requires n(n+1)/2 buffers (Figure 4.10b). For the conversions from distribution (E) to (B) or (C), n^2 buffers are needed. The conversion from (E) to (D) is similar to that in Figure 4.1b. The conversion from (E) to (F) requires n^2 buffers, and data in row i are output one step ahead of data in row (i+1). The conversions from (E) to (G) or (H) are similar to that from (E) to (F).

The design of a special-purpose converter between data distributions not defined in Figure 4.6 may be complicated, and a heuristic procedure is proposed here. First, B_{min}, the minimum number of buffers, is found by the algorithm in Section 4.2. A feasible control circuit with B_{min} buffers is then searched. The control circuit contains demultiplexers that are individually controlled by stored microprogram. If a feasible solution cannot be found easily or if the control circuit is too complex, then more buffers are added, and the procedure is repeated.

4.6. Data Conversion In Feature Extraction and Pattern Classifications

This section, contains an example of interfacing the systolic arrays in a macro-pipeline using converters. The specific problem to be discussed are pattern classification [ScK85] [HwS83].

Given an m′-by-one input vector x, a feature extractor has to produce a set of m′ transformation vectors, $D = \{d_i \mid i=1, ..., m'\}$, using S, a set of training samples with known classes, where d_i is an n-by-one column vector, m_s is the sample mean of class s, and γ_j^s is the j'th training feature vector of class s. The output of the extractor is the feature vector $y = D \cdot x$. Figure 4.11a shows the schematic design of a VLSI feature extractor, which has a macropipeline of matrix multiplication, LU decomposition, and triangular-matrix inversion.

For the pattern classifier, it is necessary to compute the feature offset vector $m=m_{s_1}-m_{s_2}$, solve the linear system $(G_{s_1}+G_{s_2})\cdot V = m$ for the discriminant vector V,

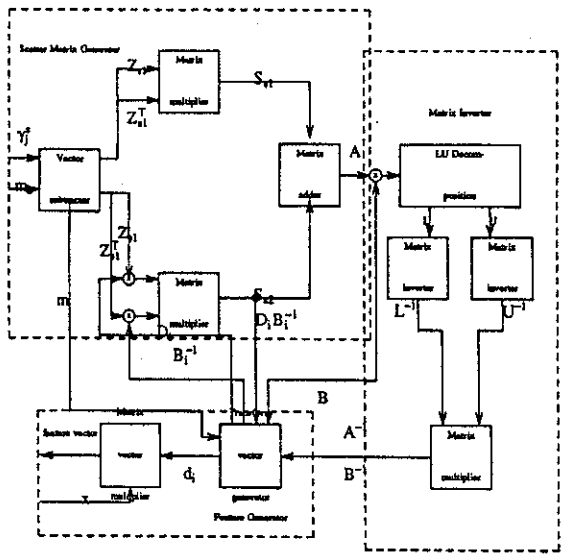


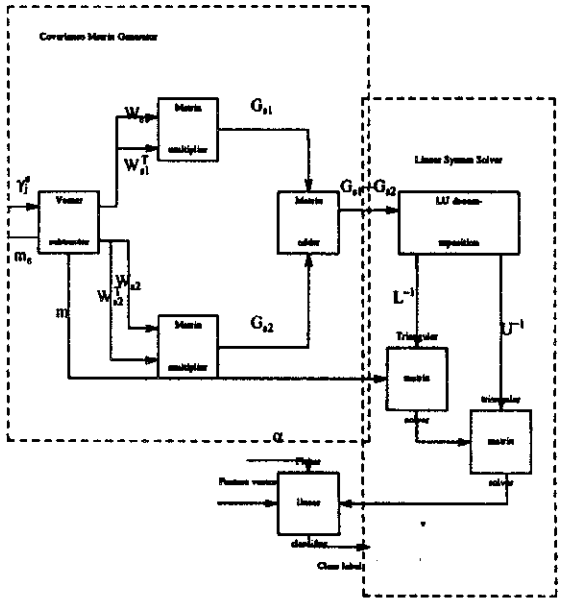
Figure 4.11

Applications of macropipelining in image processing and pattern recognition. (a) Feature extractor.

and use V to compute the discriminant function F(y) and classify the vector y. Figure 4.11b shows a schematic diagram for the VLSI feature classifier, which has a macro-pipeline of matrix multiplication, LU decomposition, and triangular-system solver.

Figure 4.12 shows a fast systolic array to multiply two n-by-n matrices in a pipelined fashion (n=4). The outputs of the systolic array exit in (2n−1) streams and are multiplexed by the converter into n streams. Figure 4.13 shows the systolic array for LU decomposition [Kun80a] and the associated input and output data formats. Although the outputs of the matrix-multiplication systolic array are in the same format as the inputs of the LU-decomposition systolic array, the outputs of the multiplication systolic array are multiplexed from (2n−1) streams into n streams to decrease the number of connections between the two chips. Hence, it is necessary to demultiplex the input data into (2n−1) streams in the LU-decomposition array. Note that this conversion is not needed if the two systolic arrays are on the same chip.

Figure 4.14 shows the triangular-matrix inverter and the associated input and output data distributions [LiW85a]. Figure 4.15 shows the conversion of the output matrix U from the LU-decomposition systolic array (Figure 4.3) into the inputs of the matrix-inverter array (Figure 4.14). Figures 4.15b and 4.15f show the data-distribution vectors of the inputs and outputs of the converter, respectively. Although the input distribution in Figure 4.15b is not one of the 48 standard data distributions in Figure 4.6a, it can be converted by n multiplexers into the data distribution in Figure 4.15c. These multiplexers are an implementation of the transformation process T_1 in equation 4.16 with $\alpha_1=1/3$ and $\beta_1=2/3$. The data distribution in Figure 4.15c is then converted into that of Figure 4.15d by entering the data into the n-by-n mesh of buffers until they are filled. This is an implementation of transformation T_2 in equation 4.18 with $\alpha_2=1/2$ and $\beta_2=-1/2$. The data are output from the north side of the array, which is equivalent to a rotation of 270°. The resulting distribution is shown in Figure 4.15e. Finally, n demultiplexers are used to convert the data distribution in Figure 4.15e to that of Figure 4.15f, which is transformation T_6 in equation 4.28 with $\alpha_6=\beta_6=1$. The conversion for the triangular array L can be done similarly.



(b)

Figure 4.11 (b)

Pattern recognizer.

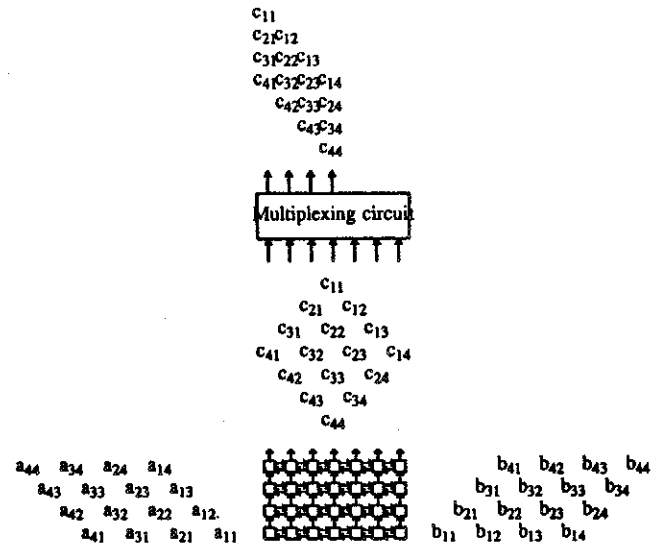


Figure 4.12

Matrix-multiplication systolic array and the corresponding input and output data distributions.

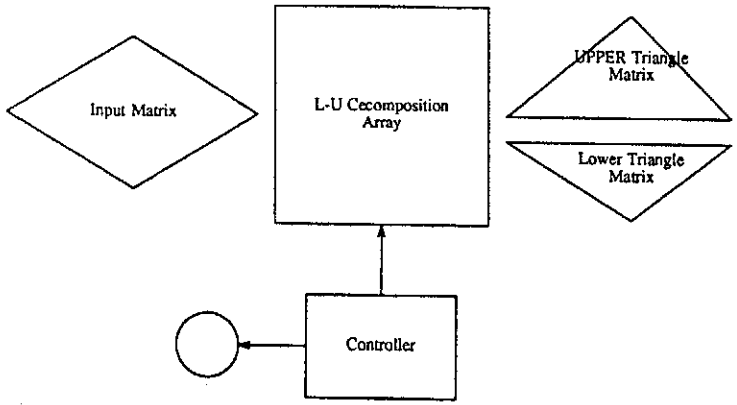


Figure 4.13

Systolic array for LU decomposition

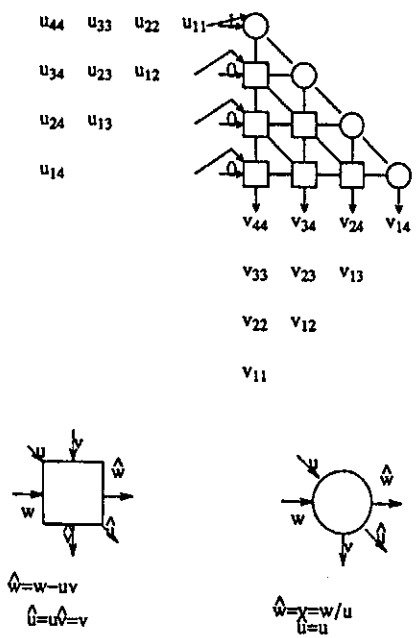


Figure 4.14

Triangular-matrix inverter.

Figure 4.16 shows the conversion from the outputs of the matrix inverter (Figure 4.14) to the inputs of the matrix-multiplication systolic array (Figure 4.12). Figure 4.16b shows the data distribution of the outputs of the matrix inverter, which can be transformed by n multiplexers into the data distribution in Figure 4.16c. Only n of the $(2n-1)$ inputs to the multiplexers are used since the input matrix is an upper triangular matrix. The data will fill the n-by-n mesh of buffers and will be output in a staggered fashion to obtain the resulting data distribution in Figure 4.16d. This corresponds to transformation T_4 with $\alpha_4 = 2$ and $\beta_4 = 1$. Finally, the data distribution in Figure 4.16d is converted into that of Figure 4.16e by a reversal network. This corresponds to T_5 with $\alpha_5 = -1$.

Similar operations are used in the pattern classifier except that it is necessary to solve the system $\mathbf{L \cdot U \cdot v = f}$ after the LU decomposition $\mathbf{A = L \cdot U}$. This is done by first solving $\mathbf{L \cdot E = f}$ and then $\mathbf{U \cdot v = E}$ to get the solution vector v. Figure 4.17 shows a special-purpose converter to transform the output matrix L of the LU-decomposition array (Figure 4.13) into the inputs required by the linear-system solver [Kun80a]. The superscript in an input element indicates the number of time units that this data item will stay in the buffers. n multiplexers will route a data item to the appropriate row in the buffers, which cause the necessary delay, and n demultiplexers will convert the data to the required format of the linear-system solver. The conversion of array U into the required format is done similarly.

4.7. Conclusion

Macropipelining of systolic arrays can be used in a wide range of applications, especially in signal and image processing. This chapter studied synchronization of the dataflow in a macropipeline of systolic arrays. To avoid the bottleneck of a common memory, converters are necessary to convert the output data from a systolic array into the required input data format of the next systolic array in the pipeline. An efficient algorithm was also developed to find the minimum number of buffers for any conversion. By studying a special subset of frequently used conversions, a methodology for designing a general purpose converter was presented. Methods to design special

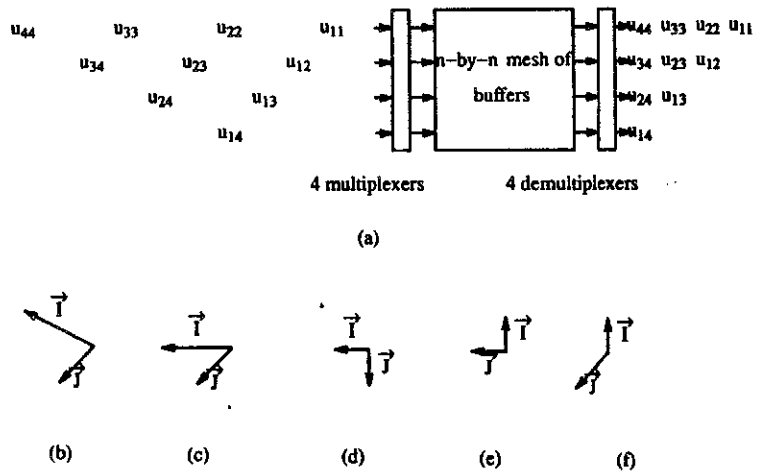


Figure 4.15

Data conversion from the outputs of the LU-decomposition systolic array to the inputs of the triangular-matrix-inverter array. (a) The converter. (b) Input data distribution before entering the converter. (c) Input data distribution to buffers after multiplexing. (d) Data distribution after the buffers are filled. (e) Output data distribution from the buffers. (f) Output data distribution after demultiplexing.

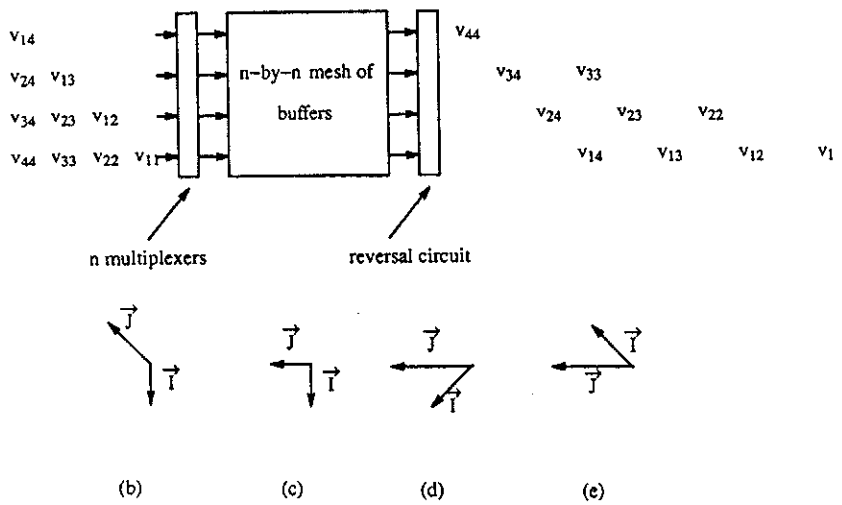


Figure 4.16

Conversion from the outputs of the matrix inverter to the inputs of the matrix-multiplication systolic array. (a) The converter. (b) Input data distribution before multiplexing. (b) Input data distribution after multiplexing. (d) Output data distribution from the buffers. (e) Output data distribution after reversal.

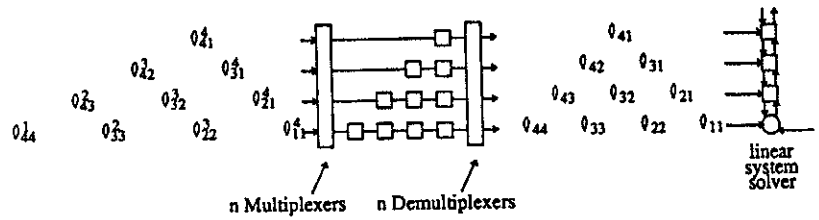


Figure 4.17

Conversion of data from the outputs of LU-decomposition array into the required input data format of the linear-system solver.

purpose converters were also examined. The proposed design methods are exemplified by the design of converters for macropipelines in feature extraction and pattern classification.

4.8. SUMMARY

This Chapter, focused on the macropipelines of systolic arrays. Macropipelines of systolic arrays can be used in a wide range of applications, especially in signal and image processing. To synchronize the dataflows in a macropipleline and to avoid the bottleneck of a common memory, converters are necessary to convert the output data from one systolic array into the required input data format of the next systolic array in the pipeline. In a previous work [WaS85], an efficient algorithm was developed to find the minimum number of buffers for any conversion. By studying a special subset of frequently used conversions, a methodology to design a general-purpose converter was presented. Methods to design special-purpose converters have also been examined. The proposed design methods have been exemplified by the design of converters for macropipelines in feature extraction and pattern classification.

CHAPTER 5
CONCLUSIONS AND FUTURE RESEARCH

5.1. Conclusions

This thesis, has introduced a comprehensive overview for the design of a system of systolic arrays, from the VLSI layout level to the system level.

Chapter 2 discussed 3-D VLSI layout, where tighter lower and upper bounds for the volume and maximum wire length for the layout of the different families of graphs in a 3-D environment were discussed. Except in two cases, all the bounds for the volume are optimal. The first case is the one-active-layer layout of the planar graphs, the other is the unrestricted layout for graphs with separators N^q ,$q = 2/3$. A cost model for reflecting the real cost of the layout, instead of taking the volume as a measure of cost, was also developed.

In Chapter 3, a methodology was developed for designing a systolic array starting from recurrence equations. The idea of *Control Flow Systolic Arrays* to handle uniform, as well as nonuniform recurrence equations, was also developed. This methodology is basically a search for a heuristic solution in the space of all the possible solutions. Because of the unlimited search space, the search process must be guided for the search to be completed in a reasonable amount of time

Chapter 4 introduced the idea of converting the data between two systolic arrays that were directly interfaced, instead of using a common memory which would be a bottleneck for the whole system. The minimum number of buffers required to convert the data between two given distributions was also calculated, a general purpose converter was also proposed.

5.2. Future Research

Two main areas for future research are proposed here.

The first is in 3-D layout. Although optimal layouts for volume have been developed in this thesis, there is still much work to be done in minimizing the maximum wire length, which directly affect the speed of the circuit. The tree of meshes and the mesh of trees are very good graphs to embody the different families of graphs for 2-D VLSI layout. However in 3-D layout, these graphs lead to inferior results. Graphs for 3-D that are analogous to the mesh of trees and the tree of meshes and will lead to an optimal wire length in 3-D VLSI have yet to be found.

The second major area for future research is in the design of systolic arrays. In this thesis, when solving a recurrence equation in a domain D, the points of D have been projected on two indices i_1 and i_2. The question of how the recurrences governing the different strategies will look when using more complicated projections is still open.

LIST OF REFERENCES

[AbW87] Mokhtar A. Aboelaze and Benjamin W. Wah, "Complexity of Layouts in Three-Dimensional VLSI Circuits," *Proc. IEEE International Symposium on Circuits and Systems*, Philadelphia, PA. May 4-7. 1987

[AgJ80] D. P. Agrawal and R. Jain, "Computer Analysis of Motion Using a Network of Processors," *Proc. 5th Int'l Conf. Pattern Recognition*,

[AhH74] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*, 1974. Adison-Wesley, 1974.

[Arm79] C. V. Armstrong, et al., "An Adaptive Multimicroprocessor Array Computing Structure for Radar Signal Processing Applications," *Proc. 6th Annual Symposium on Computer Architecture*, 1979.

[BhL84] S. N. Bhatt and F. T. Leighton, "A Framework for Solving VLSI Graph Layout Problems," *J. of Computer and System Sciences*, Vol. 28, 1984, pp. 300-343.

[BlB82] A. J. Blodgett and D. R. Barbour, "Thermal Conduction Module: A High Performance Multilayer Ceramic Package," *IBM J. of Research and Development*, Vol. 26, Jan. 1982, pp. 2-29.

[BrF82] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS Architecture for Pattern Analysis and Image Database Management," *Trans. on Computers*, Vol. C-31,N 10, Oct. 1982, pp. 969-983.

[Bro78] K. Q. Brown, *Dynamic Programming in Computer Science*, CMU Tech. Rep. Carnegie Mellon University, 1978.

[CaS81] P. R. Cappello and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," in *VLSI Systems and Applications*, R. R. Sproull, and G. Steele, Jr. H. T. Kung, ed., Computer Science Press, 1981.

[CaS84] P. R. Cappello and K. Steiglitz, "Unifying VLSI Array Design with Linear Transformations of Space-Time," in *Advances in Computing Research*, F. Preparata, ed., JAI Press, Inc., 1984, pp. 23-65.

[Che85a] M. C. Chen, *A Parallel Language and its Compilation to Multiprocessor Machines or VLSI*, Yale University, New Haven, CT, 1985.

[Che85b] M. C. Chen, *A Synthesis Method for Systolic Design*, Yale University, New Haven, CT, 1985.

[Che85c] M. C. Chen, *The Generation of a Class of Multipliers: A Synthesis Approach to the Design of Highly Parallel Algorithms in VLSI*, Yale University, New Haven, CT, 1985.

[Che86] Marina C. Chen, "Synthesizing VLSI Architecture: Dynamic Programming Solver," *Proceedings of the International Conference on Parallel Processing*, August 1986, pp. 776-784.

[CzR88] Paul Czerwinski and V. Ramachandran, "Optimal VLSI Graph Embedding in Variable Aspect Ratio Rectangles," *to appear in Algorithmica*, 1988, pp. 369-378

[DeI85a] J.-M. Delosme and I. C. F. Ipsen, "An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI," *Proc. Second Int'l Symp. on VLSI Technology, Systems, and Applications*, 1985, pp. 268-273.

[DeI85b] J.-M. Delosme and I. C. F. Ipsen, *Efficient Systolic Arrays for the Solution of Toeplitz Systems: An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI*, Yale University, New Haven, CT, 1985.

[Far78] E. J. Farrel, "Processing Limitations of Ultrasonic Image Reconstruction," *Proc. Conf. on Pattern Recognition and Image Processing*,

[FiK83a] A. L. Fisher and H. T. Kung, "Synchronizing Large VLSI Processor Arrays," *Proc. 10th Annual Int'l Symposium on Computer Architecture*, June 1983, pp. 54-58.

[FiK84] A. L. Fisher, H. T. Kung, and K. Sarocky, "Experience with the CMU Programmable Systolic Chip," *Proc. of the SPIE Symposium on Real-Time Signal Processing*, Vol. 495, 1984, pp. 120-129.

[FoF86] J. A. B. Fortes, K. S. Fu, and B. W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," in *Computer Architecture*, V. Milutinovic, ed., Elsevier Science Pub. Co., 1986.

[FoM85] J. A. B. Fortes and D. I. Moldovan, "Parallelism Detection and Transformation Techniques Useful For VLSI Algorithms," *J. of Parallel and Distributed Computing*, Vol. 2, 1985.

[FuH85] K. S. Fu, K. Hwang, and B. W. Wah, "VLSI Architecture for Pattern Analysis and Image Database Mangement," in *VLSI and Modern Signal Processing*, S. Y. Kung, et al., ed., Prentice-Hall, 1985, pp. 434-450.

[Gib82] J. F. Gibbons, "SOI--A Candidate for VLSI?," *VLSI Design*, Vol. 3, 1982, pp. 54-55.

[GrN84] J. Grinberg, G. R. Nudd, and R. D. Etchells, "A Cellular VLSI Architecture," *Computer*, Vol. 17,N 1, Jan. 1984, pp. 69-81.

[GuK79] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. Very Large Scale Integration: Architecture,Design and Fabrication*, Jan. 1979, pp. 509-525.

[Han73] W. Handler, "The Concept of Macropipelining with High Availability," *Elektronische Rechenanhagan,N 15*, 1973, pp. 269-274.

[HwS83] K. Hwang and S. P. Su, "VLSI Architectures for Feature Extraction and Pattern Classification," *Int'l J. on Computer Vision, Graphics, and Image Processing*, Vol. 24,

[JaK84] H. V. Jagadish, T. Kailath, J. A. Newkirk, and R. G. Mathews, "On Hardware Description From Block Diagrams," *International Conference on ASSP*, March 1984, pp. 8.4.1-8.4.4.

[JoC81a] L. Johnson and D. Cohen, "Computational Arrays for the Discrete Fourier Transform," *Proceedings of the IEEE COMPCON*, 1981, pp. 236-244.

[JoC81b] L. Johnson and D. Cohen, "A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks," in *VLSI Systems and Computation*, B. Sproul, G. Steele H. T. Kung, ed., Computer Science Press, Rockville MD., 1981.

[JoK84] J. M. Jover and T. Kailath, "Design Framework for Systolic-Type Arrays," *Proc. ICASSP*, March 1984, pp. 8.5.1-8.5.4.

[JoW81] L. Johnson, U. Weiser, D. Cohen, and A. Davis, "Towards a Formal Treatment of VLSI Arrays," *Proceedings of the 2nd Caltech Conference on VLSI*, 1981, pp. 375-398.

[KaH67] Richard M. Karp and Michael Held, "Finite-State Processes and Dynamic Programming," *SIAM J. of Applied Mathematics*, Vol. 15, May 1967, pp. 693-718.

[KaM67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM*, Vol. 14, July 1967, pp. 563-590.

[Kan87] Douglas A. Kandle, "A Systolic Signal Processor for Signal-Processing Application," *IEEE Computers*, Vol. 20, July 1987, pp. 94-95.

[KuL83] H. T. Kung and W. T. Lin, "An Algebra for VLSI Algorithm Design," *Proc. Conf. on Elliptic Problem Solvers*, 1983.

[KuL85] S. Y. Kung and S. C. Lo, "A spiral Systolic Architecture/Algorithm for Transitive Closure Problems," *Proc. IEEE Int. Conf. Computer Design*, 1985.

[KuL87b] Sun-Yuan Kung, Sheng-Chun Lo, and Paul S. Lewis, "Optimal ystolic Design form the transitive Closure and the Shortest Path Problems," *IEEE Transactions on Computers*, Vol. C-36, May 1987, pp. 603-614.

[KuM84] H. T. Kung and O. Menzilcioglu, "Warp: A Programmable Systolic Array Processor," *Proc. of the SPIE Symposium on Real-Time Signal Processing*, Vol. 495, 1984, pp. 130-136.

[Kun80a] H. T. Kung, "Highly Concurrent Systems Introduction to VLSI System," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, ed., Addison-Wesley, 1980.

[Kun82] H. T. Kung, "Why Systolic Architecture," *Computer*, Vol. 15,N 1, Jan. 1982, pp. 37-46.

[Kun84] S. Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. IEEE*, Vol. 72,N 7, 1984.

[LaM83] M. Lam and J. Mostow, "A Transformational Model of VLSI Systolic Design," *IFIP 6th Int'l Symp. on Computer Hardware Descriptive Languages and their Applications*,

[LaT80] H. W. Lam, A. F. Tasch Jr., and T. C. Holloway, "Characteristics of MOSFETs Fabricated in Laser-Recrystallized Polysilicon Islands with a Retaining Wall Structure on an Insulating Substrate," *IEEE Electron Dev. Letters*, Vol. EDL-1, 1980, pp. 206-208.

[LeL82] F. T. Leightoon and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *Proc. 23rd Annual Symp. on Foundation of Computer Science*, Nov. 1982, pp. 297-310.

[LeL85] F. T. Leightoon and C. E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEEE Transactions on Computers*, Vol. C-34, May 1985, pp. 448-461.

[LeR83a] F. T. Leighton and A. L. Rosenberg, "Automatic Generation of Three-Dimensional Circuit Layouts," *IEEE Int'l Conf. on Computer Design: VLSI in Computers*, 1983, pp. 633-636.

[LeR83b] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Proc. Third Caltech Conference on Very Large Scale Integration*, R. Bryant, ed., Computer Science Press, 1983.

[LeR84] F. T. Leighton and A. Rosenberg, *Three-Dimensional Circuit Layouts*, Laboratory of computer science , 1984.

[LeR86] F. T. Leighton and A. L. Rosenberg, "Three-Dimensional Circuit Layout," *SIAM J. on Computing*, Vol. 15, Aug. 1986, pp. 793-813.

[LeS81] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Twenty-Second Annual Symp. on Foundations of Computer Science*, Oct. 1981, pp. 23-36.

[Lee87] Steven B. Leeland, "An Advanced DSP Systolic-Array Architecture," *IEEE Computer*, Vol. 20, July 1987, pp. 95.

[Lei80] C. E. Leiserson, "Area-Efficient Graph Layouts (for VLSI)," *Proc. 21st Annual Symposium on Foundation of Computer Science*, Oct. 1980, pp. 270-281.

[Lei81] F. T. Leighton, "New Lower Bounds Techniques for VLSI," *Proc. 22nd Annual IEEE Symposium on Foundation of Computer Science*, Oct. 1981, pp. 1-12.

[Lei83a] F. T. Leighton, *Complexity Issues in VLSI*, 1983. Foundations in Computing Series, M.I.T. Press, 1983.

[Lei83b] C. E. Leiserson, *Area-Efficient VLSI Computations*, 1983. MIT Press, 1983.

[Lei84] F. T. Leighton, "New Lower Bounds for VLSI," *Mathematical Systems Theory*, 1984, pp. 47-70.

[LiS86] Frank C. H. Lin, John W. Shen, Phil J. Kuekes, Yul J. Inn, and Mike T. McNamara, "MOSAIC: A Heterogeneous Architecture for Signal Processors," *Proc. of the 12th DARPA Strategic Systems Symposium*,

[LiT77a] R. J. Lipton and R. E. Tarjan, "Application of a Planar Separator Theorem," *18th Annual Symp. on Foundation of Computer Science*, October 1977, pp. 162-170.

[LiT77b] R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs," *Conference on Theoretical Computer Science*, Aug. 1977, pp. 1-10.

[LiW83a] G. J. Li and B. W. Wah, "The Design of Optimal Systolic Algorithms," *Proc. Computer Software and Applications Conference*, 1983, pp. 310-319.

[LiW83b] G. J. Li and B. W. Wah, "Optimal Design of Systolic Arrays for Image Processing," *Proc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Oct. 1983, pp. 134-141.

[LiW85a] F. C. Lin and Wu, "Systolic Arrays for Transitive Closure algorithms," *Proc. of Int. Symposium on VLSI System Design*,

[McM87] John V. McCanny and John G. McWhirter, "Some Systolic Array Developments in the United Kingdom," *IEEE Computer*, Vol. 20, July 1987, pp. 51-63.

[MeC80] C. Mead and L. Conway, *Introduction to VLSI Systems*, 1980. Addison-Wesley, 1980.

[MeG85] Rami Melhem and Concettina Guerra *The Application of a Sequence Notation to the Design of Systolic Computations*, Tech. Rep. 5668, Department of Computer Science, Purdue University, West Lafayette, IN. 1986.

[MiW84] W. L. Miranker and A. Winkler, "Space-Time Representations of Computational Structures," *Computing*, Vol. 32, 1984, pp. 93-114.

[Mol82] D. I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *Trans. on Computers*, Vol. C-31,N 11, Nov. 1982, pp. 1121-1126.

[Mol83] D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. IEEE*, Vol. 71,N 1, Jan. 1983, pp. 113-120.

[NaP87] J. Greg Nash, K. Wojtek Przytula, and S. Hansen, "The Systolic/Cellular System for Signal Processing," *IEEE Computer*, Vol. 20, July 1987, pp. 96-97.

[NiH79] G. Nicolac and K. H. Hohne, "Multiprocessor System For the Real-Time Digital Processing of Video-Image Series," *Elektronische Rechenanhagan*, 1979.

[NuE85] Graham R. Nudd, R. David Etchells, and Jan Grinberg, "Three-Dimensional VLSI Architecture for Image Undestanding," *J. Parallel and Distributed Computing*, Vol. 2, Feb. 1985, pp. 1-29.

[NuN83] G. R. Nudd, J. G. Nash, S. S. Narayan, and A. K. Jain, "An Efficient VLSI Structure for Two-Dimensional Data Processing," *Proc. Int'l Conference On Computer Design*, 1983, pp. 553-556.

[Nud80] G. R. Nudd, "Image Understanding Architectures," *Proc. National Computer Conference*, 1980, pp. 377-390.

[PaR81] M. S. Paterson, W. L. Ruzzo, and L. Snyder, "Bounds on Minimax Edge Length for Complete Binary Trees," *Procedings of the 13th Annual Symposium on the Theory of Computing*, 1981, pp. 293-299.

[Pre83a] F. P. Preparata, "Optimal Three-Dimensional VLSI Layouts," *Mathematical System Theory*, Vol. 16, Jan. 1983, pp. 1-8.

[Qui83] P. Quinton, *The Systematic Design of Systolic Arrays*, IRISA, 1983.

[Qui84] P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *Proc. 11'th Annual Symp. on Computer Architecture*, 1984, pp. 208-214.

[RaF83] I. V. Ramakrishnan, D. S. Fussell, and A. Sillberschatz, "On Mapping Homogeneous Graphs on a Linear Array-Processor Model," *Proc. Int'l Conf. on Parallel Processing*, 1983, pp. 440-447.

[RaF86] S. V. Rajopadhye and R. M. Fujimoto, *Systolic Arrays with Control signals from Recurrence Equations*, Tech. Rep. CIS CIS-TR-86-12, University of Oregon, Computer and Information Science Department, 1986.

[RaK85] Sailesh K. Rao and Thomas Kailath, "VLSI Arrays for Digital Signal Processing: Part 1- A Modsel Identification Approach to Digital Filter Realization," *IEEE Transactions on Circuits and SSystemss*, Vol. CASS-32, November 1985, pp. 1105-1118.

[Raj86] S. V. Rajopadhye, *Synthesis, Otimization and Verification of Systolic Architecture*, Ph.D. thesis, University of Utah, Salt Lake City, Utah, 1986.

[Ram82] V. Ramachandran, "On driving many long wires in a VLSI layout," *Proc. the 23rd Annual Symposium on Foundation of Computer Science*, 1982, pp. 369-378

[RoT85] Y. Robert and D. Trystarm, *An Orthogonal Systolic Array for the Algebric Path Problem*, Res. Report 553,TIM3/IMAG Grenoble, France, 1985.

[Ros81] A. L. Rosenberg, "Three-dimensional Integrated Circuits," in *VLSI Systems and Computations*, B. Sproull, and G. Steele H. T. Kung, ed., Computer Science Press, Rockville, MD., 1981, pp. 69-80.

[Ros83a] A. L. Rosenberg, "Three-Dimensional VLSI: A Case Study," *Journal of the ACM*, Vol. 30,N 3, July 1983, pp. 397-416.

[Rot85] G. Rote, "A Systolic Array Algorithm for the Algebric Path Problem (Shortest Paths, Matrix Inversion)," *Computing*, Vol. 34, 1985, pp. 192-219.

[ScB84] D. A. Schwartz and T. P. Barnwell, III, "A Graph Theoretic Technique for the Generation of Systolic Implementations for Shift-Invariant Flow Graphs," *Proc. ICASSP*, 1984, pp. 8.3.1-8.3.4.

[ScK85] R. Schreiber and P. J. Kuekes, "Application of Systolic Array Technology to Recursive Filtering," in *VLSI and Modern Signal Processing*, S. Y. Kung, et al., ed., Prentice-Hall, 1985.

[SuM77] I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," *Scientific American*, Vol. 237, September 1977, pp. 210-228.

[Tho80] C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Dissertation, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[TrG82] P. C. Treleaven and L. Gouveia, "Japan's Fifth Generation Computer Systems," *IEEE Computer*, Vol. 15,

[WaA88] B. W. Wah , M. A. Aboelaze, and W. shang , "Systematic Design of Buffers in in Macropipelining of Systolic Arrays," *Journal of Parallel and Distributed Computing*.

[WaS85] B. W. Wah, W. J. Shang, and M. Aboelaze, "Buffering in Macropipelines of Systolic Arrays," *Proc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 1985, pp. 1-8.

[Wei81] Z. A. Weinberg, *Polysilicon Recrystallization by CO2 Laser Heating of SiO2*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981.

[Wis81] D.S. Wise, "Compact Layouts of Banyan/FFT Networks," in *VLSI Systems and Computations*, B. Sproull, and G. Steele H.T. Kung, ed., Computer Science Press, 1981.

VITA

VITA

Mokhtar A. Aboelaze was born in Cairo, Egypt on April 18, 1956. He received B.Sc. in Electrical Engineering from Cairo University in June 1987, M.Sc. in Electrical and Computer Engineering in August 1982 from the University of South Carolina