Master of Engineering Thesis

Parallel Processing Architecture for Linear Programming

Department of Electrical and Communication Engineering,
Graduate School of Engineering,
Tohoku University,
Sendai, JAPAN

Shinhaeng LEE

Contents

Chapter 1 Introduction **1**

1.1 Background 1

1.2 Purpose of This Thesis 7

1.3 Organization of This Thesis 8

Chapter 2 Linear Programming **9**

2.1 Introduction 9

2.2 Standard-form Linear Programming 10

2.3 An Example of a Problem 11

2.4 Several Different Solving Methods 12

2.5 Revised Simplex Method 16

2.5.1 Computational Steps 16

Chapter 3 Systolic Arrays **21**

3.1 Introduction 21

3.2 Space-time Mapping 22

3.2.1 Models for VLSI Arrays and Algorithms 22

3.2.2 Mapping Algorithms into VLSI Arrays 24

3.3 Optimum Transformation Matrix 25

3.3.1 Conditions for Optimum Transformation Matrix 26

3.3.2 Cost Function for Optimum Transformation Matrix 27

Chapter 4 Parallel Processing Architecture for Linear Programming **31**
4.1 Introduction 31
4.2 Simple Architecture for Parallel Processing 32
4.2.1 Systolic Array of Step 8 32
4.2.2 Systolic Arrays for the Revised Simplex Method 35
4.2.3 Systolic Array for Parallelism Improvement 44
4.3 Architecture for Large Scale Linear Programming 45
4.3.1 Modified Systolic Array for Large Scale Linear Programming 47
4.3.2 Distributing Data of Problems to Chips 50
4.3.3 Simulation of Cell 3 and the Unified Cell 52

Chapter 5 Concluding Remarks **57**
5.1 Conclusion 57
5.2 Further Works 58

Acknowledgments **59**

Works **62**

List of Figures

1.1 Functional structure of pipeline computer. 3
1.2 Functional structure of SIMD array processor. 4
1.3 Functional structure of MIMD multiprocessor system. 5
1.4 The systolic array for the matrix product. 6

2.1 Graphical method of example 2.1. 14
2.2 Fundamental difference between simplex and interior-point method. 15

3.1 Array with 8-neighbor connections. 23

4.1 The cells of the four mappable matrices. 34
4.2 The systolic array of T_4. 35
4.3 The structure of cell of T_4. 36
4.4 The structure of array and the cell of step 1. 37
4.5 The structure of array and the cell of step 2. 38
4.6 The structure of array and the cell of step 4. 39
4.7 The structure of array and the cell of step 8. 40
4.8 The systolic array for revised simplex method in the case of $m = 3$, $n = 9$. 41
4.9 The structure of the unified cell. 43
4.10 Efficient usage of an array of the revised simplex method. 45
4.11 Three patterns of the switching network. 46
4.12 Pipeline diagram of application of the array. 46
4.13 The modified architecture which consists of three chips. 48
4.14 The connections of chips. 49

4.15 The example of data distribution in the case of $s = 3$. 51
4.16 The timing wave of cell 3. 54
4.17 The timing wave of the unified cell. 54

List of Tables

2.1 The computation result of example 2.1. 20

4.1 The best results of an evaluation of the dependence matrix of step 8. . . . 33
4.2 The values for the constant parameters in the cost functions. 33
4.3 Comparison of sequential computation with computation using systolic array. 44
4.4 The relationship of control signals to function of Chip A; 'x' is the Don't care signal. 47
4.5 The number of chips that are required to solve an $m \times n$ size linear programming problem. 50
4.6 The specification of behavioral models of cell 3. 52
4.7 The specification of behavioral models of the unified cell. 53
4.8 The result of automatic synthesis for cell 3. 55
4.9 The result of automatic synthesis for the unified cell. 55
4.10 The number of cells that are implemented in a chip. 55
4.11 The computation time of each module in the case that the frequency of operation of the cells is 1 MHz. 56

Chapter 1

Introduction

1.1 Background

Over the past four decades the computer industry has experienced four generations of development [18]:

- Relays and vacuum tubes (1940 - 1950s)
- Discrete diodes and transistors (1950 - 1960s)
- Small- and medium-scale integrated (SSI/MSI) circuits (1960 - 1970s)
- Large- and very-large-scale integrated (LSI/VLSI) devices (1970s and beyond)

Increases in device speed and reliability and reductions in hardware cost and physical size have greatly enhanced computer performance. However, better devices are not the sole factor contributing to high performance. A modern computer system is really a composite of such items as processors, memories, functional units, interconnection networks, compilers, operating systems, peripheral devices, communication channels and database banks. A good computer architecture should master all these disciplines. It is the revolutionary advances in integrated circuits that have contributed to the significant improvement of computer performance during the past 50 years.

From an operating system point of view, computer systems have improved chronologically in four phases:

- Batch processing
- Multiprogramming
- Time sharing
- Multiprocessing

In these four operating modes, the degree of parallelism increases sharply from phase to phase. *Parallel processing* is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process.

Parallel computers are those systems that emphasize parallel processing. Parallel computers are divided into three architectural configurations:

- Pipeline computers
- Array processors
- Multiprocessor systems

The typical structures of above three parallel computers are depicted in Figure 1.1, 1.2, and 1.3.

A pipeline computer performs overlapped computations to exploit *temporal parallelism.* An array processor uses multiple synchronized arithmetic logic units to achieve *spatial parallelism.* A multiprocessor system achieves *asynchronous parallelism* through a set of interactive processors with shared resources. The rapid progress in the VLSI (Very Large Scale Integrated) technology has made these approaches possible.

We are interested in high-performance parallel algorithms that can be implemented directly on low-cost hardware devices. The concept of *systolic algorithm* is a general methodology for mapping high-level computations into hardware structures [7] [8]. In a systolic array, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, such as blood circulates to and from the heart. Figure 1.4 is the systolic array for the matrix product $\mathbf{C} = \mathbf{A} \times \mathbf{B}$.

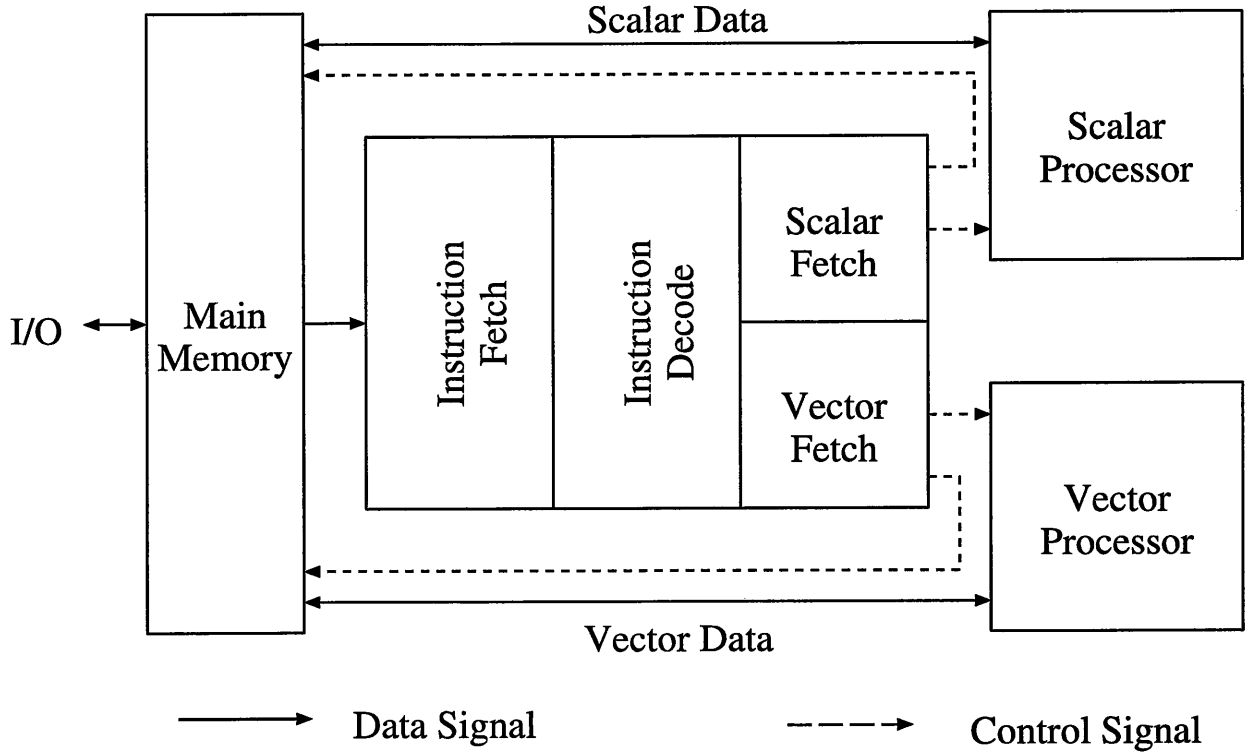


Figure 1.1: Functional structure of pipeline computer.

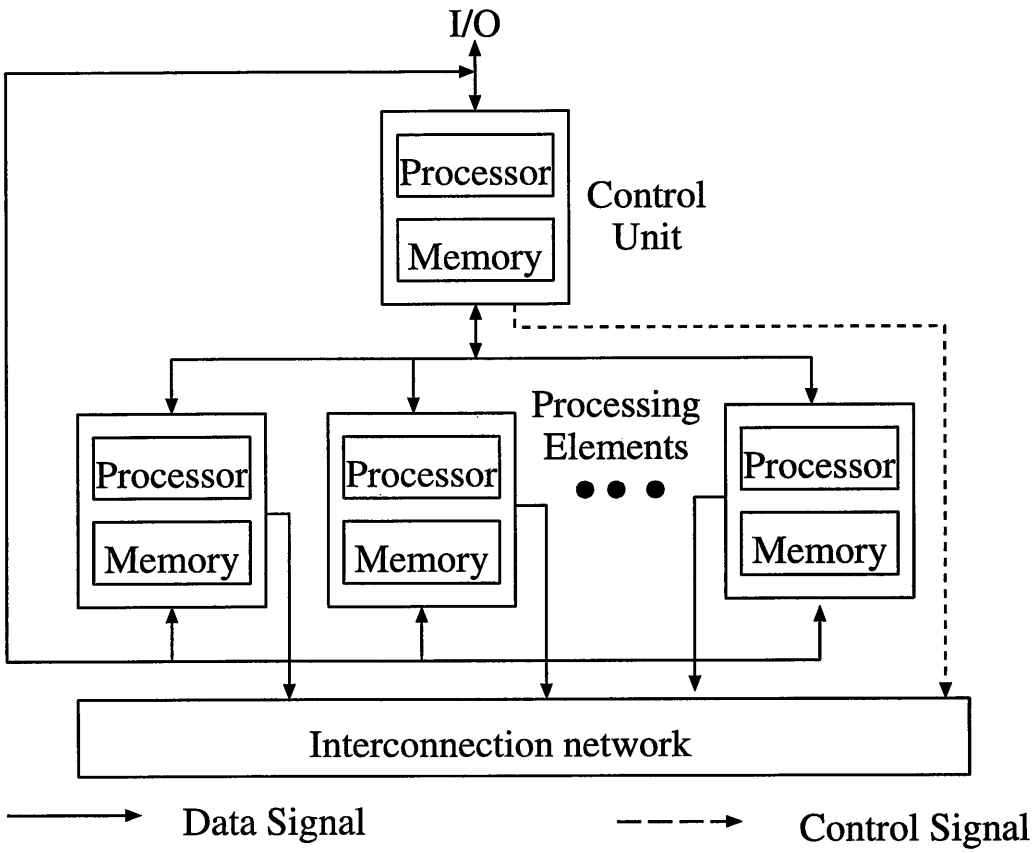


Figure 1.2: Functional structure of SIMD array processor.

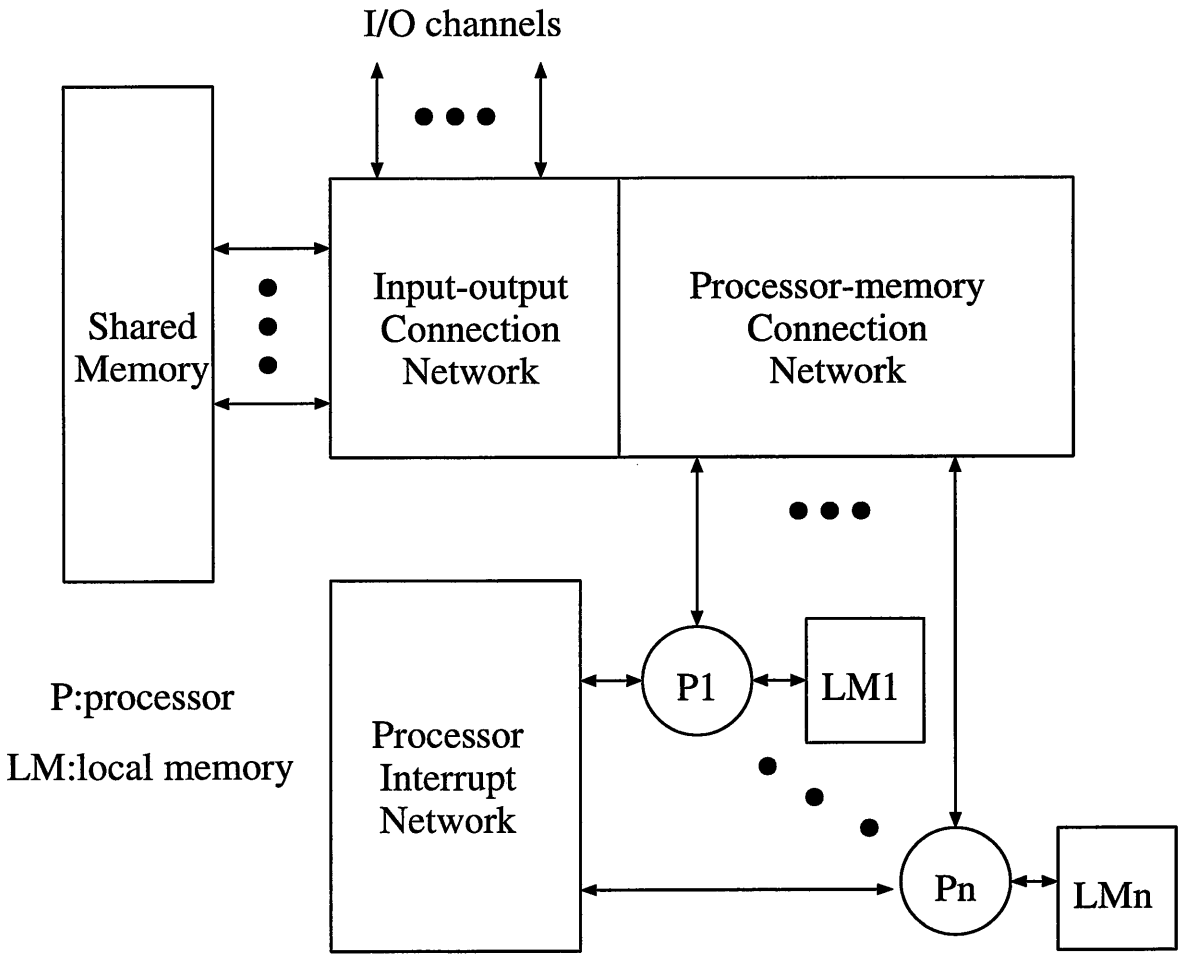


Figure 1.3: Functional structure of MIMD multiprocessor system.

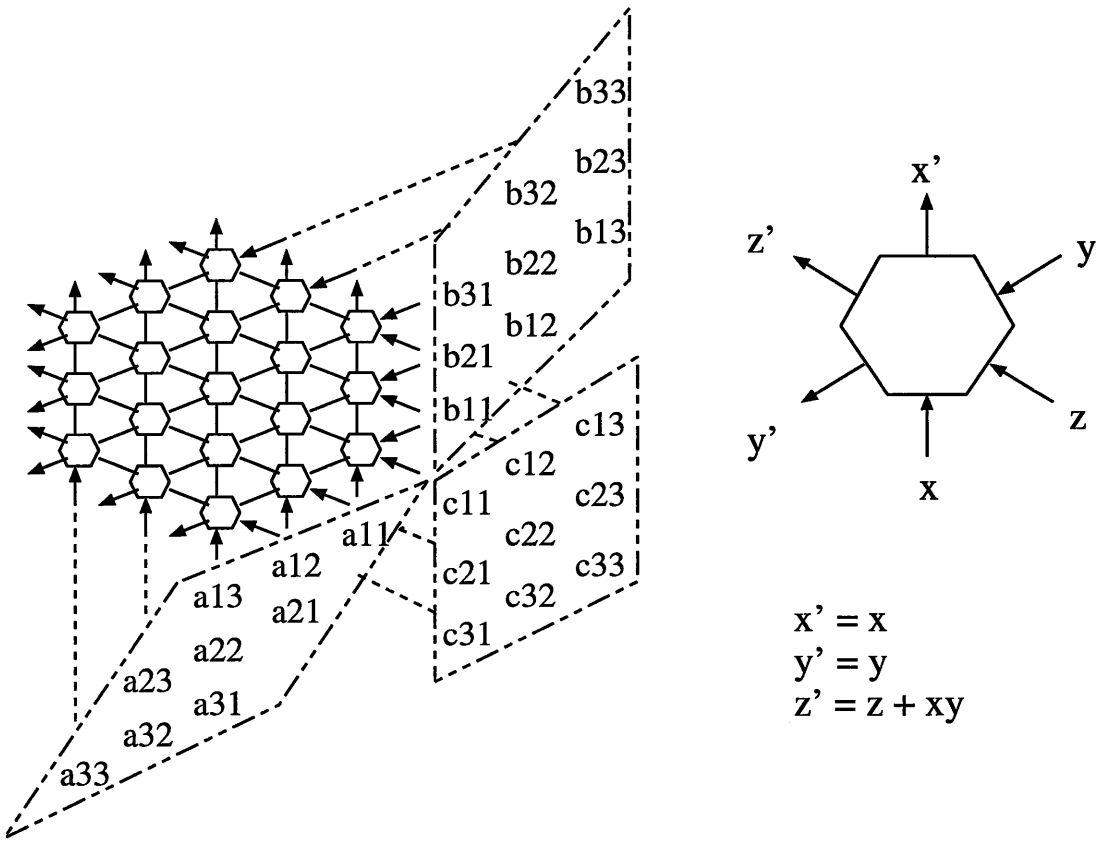


Figure 1.4: The systolic array for the matrix product.

1.2 Purpose of This Thesis

In this thesis, we propose an architecture which can solve any large size linear programming problem.

The aim of *linear programming* is to derive an optimum use of resources in industry and in organizations [4] [12] [13]. The purpose is to minimize the costs in organizations and in projects.

Linear programming is characterized by linear constraints equations and a cost function. Because the number of equations is usually very large, it takes a lot of computation time to find a solution [1] [2] [3]. Therefore, *special-purpose* hardware is sought for high-speed linear programming [6].

In this thesis, we design special-purpose hardware for high-speed linear programming.

Since most special-purpose chips will be made in relatively small quantities, the design cost must be kept low [5]. Systolic algorithm has several advantages which help reduce the cost:

- One may design and test only a few different, simple cells, since most of the cells on the chip are copies of a few basic ones
- Regular interconnection implies that the design can be made modular and extensible, so one can design a large chip by combining the designs of small chips
- By pipelining and multiprocessing, one can meet the performance requirement of a special-purpose chip simply by including many identical cells on the chip.

An enormous number and variety of designs have been developed. Formal methods for designing systolic arrays have also been developed [9] [10] [11] [17]. One such method is the *space-time mapping* given by Moldovan [15] [16]. In the space-time mapping method, an algorithm is mapped into a systolic array using data dependences which are obtained from the loop program of a given algorithm.

Although many kinds of systolic arrays can be generated for a given algorithm using the space-time mapping, few are optimal [19] [20].

In this thesis, we obtain optimal systolic architecture by optimal transformation matrix which has minimum $area \times time^2$.

In practice, the sizes of linear programming problems are not fixed and are very large, but special-purpose hardware can solve only fixed size linear programming problems. Therefore, we have to design an architecture that can solve any large size linear programming problems.

Cells in systolic arrays are very simple circuits and interconnections between cells are regular, so circuits are separated into several kinds of chips. If the size of linear programming problems is large, we can add systolic array chips whose types are the same. Then we can solve any large size linear programming problems.

Finally, we propose an architecture which can solve any large size linear programming problems.

1.3 Organization of This Thesis

First, we present an overview of linear programming in Chapter 2, followed by a summary of designing systolic arrays using the space-time mapping in Chapter 3. In Chapter 4, we propose an architecture for linear programming and the modified architecture which can solve any large size linear programming. Finally, the concluding remarks are presented in Chapter 5.

Chapter 2

Linear Programming

2.1 Introduction

Linear programming is concerned with problems in which a linear objective function in terms of decision variables is to be optimized (i.e., either minimized or maximized) while a set of linear equations, inequalities, and sign restrictions are imposed on the decision variables as requirements [3] [4]. Linear programming is a quite young and yet very active branch of applied mathematics [12].

The linear programming problem was conceived by G.B.Dantzig around 1947 while he was working as a Mathematical Advisor to the United States Air Force Comptroller on developing a mechanized planning tool for a deployment, training, and logistical supply program [13]. The work led to his 1948 publication, "Programming in a Linear Structure." An effective *simplex method* for solving linear programming problems was proposed by Dantzig in 1949.

Linear programming techniques are extremely useful in many diverse applications such as [14]:

- Agricultural applications
- Procurement of contract awards

- Economic aids
- Industrial applications
- Military applications
- Personnel assignment
- Production scheduling and inventory control
- Structural design
- Traffic analysis
- Transportation problems and network theory
- Traveling salesman problem
- Statistics, combinatorial analysis and graph theory
- Design of optical filters

2.2 Standard-form Linear Programming

In general, a linear programming is a problem of minimizing or maximizing a linear objective function with restricted or unrestricted decision variables in the presence of linear equality and/or inequality constraints. Therefore, we convert any general linear programming problem into *standard-form.*

The standard-form of linear programming deals with a linear minimizing problem with nonnegative decision variables and linear equality constraints. A standard-form linear programming problem can be described as the follows:

Minimize

$$z = \boldsymbol{c}^t \boldsymbol{x} \quad \left(z = \sum_{j=1}^{n} c_j x_j \right) \tag{2.1}$$

Subject to

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}^{(0)} \quad \left(\sum_{j=1}^{n} a_{ij}x_j = b_i^{(0)},\ i = 1,\ldots,m\right) \tag{2.2}$$

$$\boldsymbol{x} \geq 0 \quad (x_j \geq 0,\ j = 1,\ldots,n)$$

where $\boldsymbol{A}$ is an $m \times n$ constraints matrix ($m < n$), $\boldsymbol{c}$ is the cost vector with n elements, $\boldsymbol{b}^{(0)}$ is called the right-hand-side vector with m elements, and $\boldsymbol{x}$ is the decision variables vector with n elements. $\boldsymbol{c}^t$ denotes transpose of $\boldsymbol{c}$.

We search the optimal value of objective function for $\boldsymbol{x}$ such that the objective value $\boldsymbol{c}^t\boldsymbol{x}$ is minimum, while the constraints equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}^{(0)}$ and the bound constraints inequalities $\boldsymbol{x} \geq 0$ are satisfied.

2.3 An Example of a Problem

Although linear programming has long proved its merit as an effective model of numerous applications, still there is no fixed rule of modeling.

Each decision variable is associated with a certain activity of interest, and the value of a decision variable may represent the level of the associated activity. Once the decision variables are defined, the objective function usually represents the gain or loss of taking these activities at different levels, and each technological constraint depicts certain interrelationships among those activities.

To understand linear programming problems, consider the following problem:

EXAMPLE 2.1. (Production Scheduling)

LEE company produces two kinds of chips (Chip-1 and Chip-2). The unit selling price is \$ 1 for Chip-1 and \$ 2 for Chip-2. To make one Chip-1, LEE company has to invest 1 hour of skilled labor and 1 hour of unskilled labor. To make one Chip-2, it takes 2 hours of skilled labor and 1 hour of unskilled labor. LEE company has 40 hours of skilled labor and 60 hours of unskilled labor. How can LEE company determine its optimal product mix.

This problem is converted to the following form:

$$\begin{aligned} \text{Maximize} \quad & x_1 + 2x_2 \\ \text{Subject to} \quad & x_1 + x_2 \leq 40 \\ & 2x_1 + x_2 \leq 60 \\ & x_1, x_2 \geq 0 \end{aligned}$$

However, the standard form of linear programming deals with a linear minimization problem, then we convert a maximization form into minimization form as that:

$$\begin{aligned} \text{Minimize} \quad & -x_1 - 2x_2 \\ \text{Subject to} \quad & x_1 + x_2 \leq 40 \\ & 2x_1 + x_2 \leq 60 \\ & x_1, x_2 \geq 0 \end{aligned}$$

We convert this problem into standard-form using *slack variable* $x_3, x_4 \geq 0$:

$$\begin{aligned} \text{Minimize} \quad & -x_1 - 2x_2 \\ \text{Subject to} \quad & x_1 + x_2 + x_3 = 40 \\ & 2x_1 + x_2 + x_4 = 60 \\ & x_1, x_2, x_3, x_4 \geq 0 \end{aligned}$$

Although it has four variables, the feasible domain can be represented as a dimensional graph defined by

$$x_1 + x_2 \leq 40, 2x_1 + x_2 \leq 60, x_1 \geq 0, x_2 \geq 0 \tag{2.3}$$

2.4 Several Different Solving Methods

The fundamental theorem of linear programming shows that one of the extreme points of the *feasible domain* P is an optimal solution to a consistent linear programming problem unless the problem is unbounded.

There are several ways in solving linear programming problems.

- Graphical method
- Enumeration method
- Simplex method
- Interior-point approach

Graphical method is one of the most intuitive ways of solving a linear programming problem. First, we draw a graph of the feasible domain P by Equation (2.3). Then at each extreme point v of P, using the negative cost vector $-\boldsymbol{c}^t$ as the normal vector, we draw a hyper-plane H. If P is contained in the half-space H_L, then H is a desired supporting hyper-plane and v is an optimal solution. Figure 2.1 shows graphical representation of Example 2.1.

Enumeration method is a straight-forward method. Since an extreme point corresponds to a basic feasible solution, it must be a basic solution. We can generate all basic solutions by choosing m linearly independent columns from the columns of the constraint matrix. Among all basic solutions, we identify feasible ones and take the optimal one as our solution. It becomes impractical when the number $\frac{n!}{m!(n-m)!}$ becomes large.

Focusing on finding an optimal extreme point, the *simplex method* starts with one extreme point, hops to a better neighboring extreme point along the boundary, and finally stops at an optimal extreme point. Because the method is well designed, rarely do we have to visit too many extreme points before an optimal one is found. But in the worst case, this method may still visit all non-optimal extreme points.

Unlike the simplex method, the *interior-point method* stays in the interior of P and tries to position a current solution as the "center of universe" in finding a better direction for the next move. By properly choosing step lengths, an optimal solution is finally achieved after a number of iterations. This approach takes more effort, hence more computational time, in finding a moving direction than the simplex method, but better moving directions result in fewer iterations. Figure 2.2 shows the fundamental difference between simplex method and interior-point approach.

In this thesis, since the simplex method is well suited for hardware implementation [6], we present parallel architecture for the simplex method.

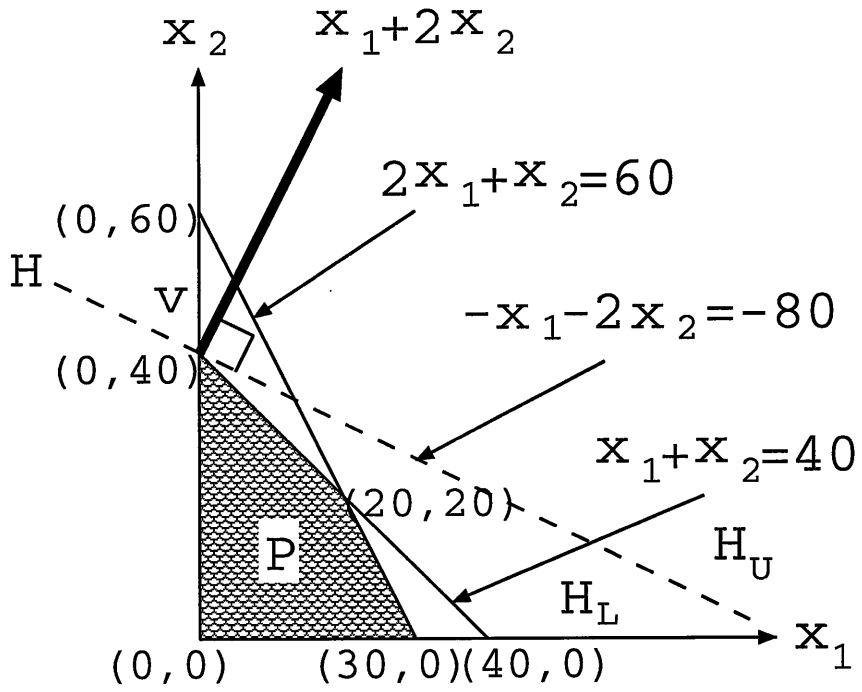


Figure 2.1: Graphical method of example 2.1.

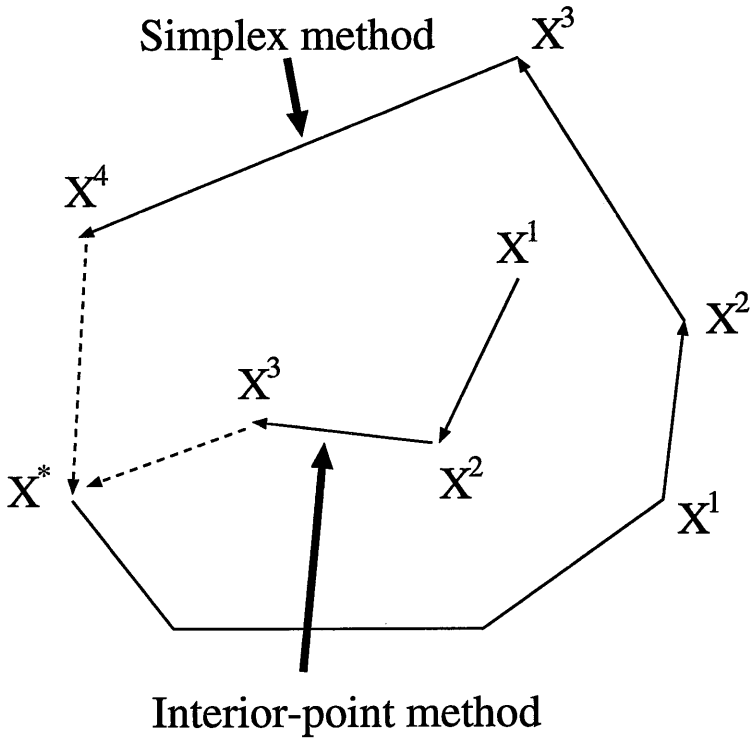


Figure 2.2: Fundamental difference between simplex and interior-point method.

2.5 Revised Simplex Method

For computational efficiency, we focus on the *revised simplex method*, which is a procedure for implementing the steps of the simplex method in a smaller array.

For simplicity without loss of generality, $\boldsymbol{A}$,$\boldsymbol{c}$ and $\boldsymbol{x}$ can be partitioned as

$$\boldsymbol{A} = [\boldsymbol{B}\ \boldsymbol{N}], \boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}_b \\ \boldsymbol{c}_n \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_b \\ \boldsymbol{x}_n \end{bmatrix} \tag{2.4}$$

where $\boldsymbol{B}$ is an $m \times m$ nonsingular matrix, and $\boldsymbol{N}$ is an $m \times (n-m)$ matrix.

An initial solution to linear programming problem is supposed to be known as such that $\boldsymbol{x}$ is feasible so that it holds: $\boldsymbol{x}_b = \boldsymbol{B}^{-1}\boldsymbol{b}^{(0)} > 0$ and $\boldsymbol{x}_n = 0$. The matrix $\boldsymbol{B}$ is called the *basis* and the variables $\boldsymbol{x}_b$ associated with $\boldsymbol{B}$ are called the *basic variables.*

Accordingly the matrix $\boldsymbol{N}$ is called the *non-basis* and the variables $\boldsymbol{x}_n = 0$ associated with $\boldsymbol{N}$ are the *non-basic variables.* The revised simplex algorithm starts from the initial solution.

In general, the so-called *Two Phase* strategy is used. *Phase 1* entails finding a basic feasible solution as the initial solution for *Phase 2. Phase 2* entails finding the optimal solution to the original linear programming problem, by starting from the solution obtained from *Phase 1.* The computational steps for both *Phase 1* and *Phase 2* are the same, except the data set being worked on. Therefore, in this thesis, we consider one computational step in Section 2.5.1

2.5.1 Computational Steps

In the following, we summarize the main computational steps for the revised simplex algorithm.

step0: Compute $\boldsymbol{B}^{-1}$ and Equation (2.5).

$$\boldsymbol{b} = \boldsymbol{B}^{-1}\boldsymbol{b}^{(0)} \tag{2.5}$$

step1: Compute the pricing vector, $\boldsymbol{w}^t$.

$$\boldsymbol{w}^t = \boldsymbol{c}_b^t\boldsymbol{B}^{-1} \tag{2.6}$$

step2: Price out the non-basic columns from the non-basis.

$$\boldsymbol{r} = \boldsymbol{c} - (\boldsymbol{w}^t \boldsymbol{A})^t \tag{2.7}$$

step3: If no element of $\boldsymbol{r}$ is negative, then terminate and the current solution is optimal.

$$\boldsymbol{r} \geq 0 \rightarrow OPTIMAL \tag{2.8}$$

Otherwise select an index q such that $r_q < 0$

The corresponding non-basic variable x_q will become basic.

step4: Update the entering column.

$$\boldsymbol{d} = \boldsymbol{B}^{-1} \boldsymbol{A}_q \tag{2.9}$$

Here $\boldsymbol{A}_q$ is the q-th column of matrix $\boldsymbol{A}$.

step5: Check for unboundedness. If no element of $\boldsymbol{d}$ is positive, then terminate and this problem is unbounded.

$$\boldsymbol{d} \leq 0 \rightarrow UNBOUNDED \tag{2.10}$$

step6: Select an index p such that

$$\frac{b_p}{d_p} = \min_{1 \leq i \leq m} \{ \frac{b_i}{d_i} | d_i > 0 \} \tag{2.11}$$

The leaving basic variable from the basis is corresponding variable to the p-th entry of $\boldsymbol{b}$.

step7: Compute so-called eta vector, η.

$$\eta_i = \begin{cases} -\dfrac{d_i}{d_p} & i \neq p \\ \dfrac{1}{d_p} & i = p \end{cases} \tag{2.12}$$

step8: Perform the pivoting

$$\hat{\boldsymbol{B}}^{-1} = \boldsymbol{B}^{-1} + (\boldsymbol{\eta} - \boldsymbol{e}^p)(\boldsymbol{B}^{-1})_p = \boldsymbol{B}^{-1} + \boldsymbol{\eta}^* (\boldsymbol{B}^{-1})_p \tag{2.13}$$

$$\hat{\boldsymbol{b}} = \boldsymbol{b} + (\boldsymbol{\eta} - \boldsymbol{e}^p) b_p = \boldsymbol{b} + \boldsymbol{\eta}^* b_p$$

$$\hat{\boldsymbol{c}}_b = \boldsymbol{c}_b + \boldsymbol{e}^p(c_q - c_{b,p})$$

where $\boldsymbol{e}^p$ denotes a vector which has '1' in p-th entry and '0' elsewhere, $\boldsymbol{\eta}^* = \boldsymbol{\eta} - \boldsymbol{e}^p$ and $(\boldsymbol{B}^{-1})_p$ is the p-th row of $\boldsymbol{B}^{-1}$.

Therefore, we can modify step 7 as the following

$$\eta_i^* = \begin{cases} -\dfrac{d_i}{d_p} & i \neq p \\ \dfrac{1}{d_p} - 1 & i = p \end{cases} \tag{2.14}$$

The same sequence of computations starts again from step 1 until the final optimal solution is found.

To understand in detail, compute Example 2.1 with this procedure.

Step 0

Assume that

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{bmatrix}, \boldsymbol{B} = \boldsymbol{B}^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \boldsymbol{N} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \tag{2.15}$$

and

$$\boldsymbol{b} = \begin{bmatrix} 40 \\ 60 \end{bmatrix}, \boldsymbol{c}^t = \begin{bmatrix} -1 & -2 & 0 & 0 \end{bmatrix}, \boldsymbol{c}_b^t = \begin{bmatrix} 0 & 0 \end{bmatrix}, \boldsymbol{c}_n^t = \begin{bmatrix} -1 & -2 \end{bmatrix} \tag{2.16}$$

Step 1

$$\boldsymbol{w}^t = \boldsymbol{c}_b^t \boldsymbol{B}^{-1} = \begin{bmatrix} 0 & 0 \end{bmatrix} \tag{2.17}$$

Step 2

$$\boldsymbol{r} = \boldsymbol{c} - (\boldsymbol{w}^t \boldsymbol{A})^t = \begin{bmatrix} -1 \\ -2 \\ 0 \\ 0 \end{bmatrix} \tag{2.18}$$

Step 3

$$\exists r < 0 \rightarrow q = 1 \tag{2.19}$$

Step 4

$$\boldsymbol{d} = \boldsymbol{B}^{-1}\boldsymbol{A}_q = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{2.20}$$

Step 5

$$\exists d > 0 \rightarrow BOUNDED \tag{2.21}$$

Step 6

$$\frac{b_p}{d_p} = \min_{1 \leq i \leq m} \{\frac{b_i}{d_i} | d_i > 0\} = min\{\frac{40}{1}, \frac{60}{2}\} \rightarrow p = 2 \tag{2.22}$$

Step 7

$$\boldsymbol{\eta}^* = \begin{bmatrix} -\frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} \tag{2.23}$$

Step 8

$$\hat{\boldsymbol{B}}^{-1} = \begin{bmatrix} 1 & -\frac{1}{2} \\ 0 & \frac{1}{2} \end{bmatrix}, \hat{\boldsymbol{b}} = \begin{bmatrix} 10 \\ 30 \end{bmatrix}, \hat{\boldsymbol{c}}_b = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{2.24}$$

This finishes one iteration of the revised simplex method. The new solution $(x_1, x_2) = (30, 0)$ is adjacent to the original vertex $(x_1, x_2) = (0, 0)$. In this solution, the objective value $\boldsymbol{c}^t\boldsymbol{x}$ is obtained by the follows:

$$\boldsymbol{c}^t\boldsymbol{x} = \begin{bmatrix} -1 & -2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 40 \\ 0 \\ 20 \end{bmatrix} = -80 \tag{2.25}$$

The same sequence of computations starts again from step 1 until the final optimal solution is found. Table 2.1 shows the procedure until the last iteration.

iteration	(x_1, x_2)	objective value
0	(0, 0)	0
1	(30, 0)	−30
2	(20, 20)	−60
3	(0, 40)	−80 (optimum)

Table 2.1: The computation result of example 2.1.

Therefore, LEE company has to produce only 40 Chip-2 according to the result of Table 2.1. LEE company benefits $ 80 by this production scheduling.

The characteristics of linear programming problems are linear constraints equations and a cost function. Since there is usually a very large number of equations, the computation time required to find a solution is also large. Consequently, *special-purpose* hardware is sought for high-speed linear programming.

Chapter 3

Systolic Arrays

3.1 Introduction

Since costs of parts are less important than design costs, special-purpose design costs can be reduced by the use of appropriate architecture [5]. In addition, special-purpose systems based on simple, regular designs are likely to be modular and therefore adjustable to various performance goals.

The design of a special-purpose system should be modular so that its structure can be easily adjusted to match a variety of I/O bandwidths.

As a solution to the above special-purpose architecture, we introduce *systolic architectures*, an architectural concept originally proposed for VLSI implementation [8]. The systolic architectural concept was developed at Carnegie-Mellon university and versions of systolic processors are being designed and built by several industrial and governmental organizations [5] [7] [8].

The concept of systolic architecture is a general methodology for mapping high-level computations into hardware structures. In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, such as blood circuits to and from the heart.

Formal methods of design of systolic arrays have also been developed. One such

method is the *space-time mapping* given by Moldovan [15] [16] [17].

3.2 Space-time Mapping

In the space-time mapping method, an algorithm is mapped into a systolic array using data dependences. First, the algorithm is expressed in terms of its dependence matrix and then, using a transformation matrix, dependence matrix is transformed to a mappable matrix where the elements of mappable matrix give all the information required for the array design.

In Section 3.2.1, computational models are introduced for VLSI arrays and algorithms. The models are used in Section 3.2.2 where we present the mapping of algorithms into systolic arrays.

3.2.1 Models for VLSI Arrays and Algorithms

We use symbol I to denote the set of non-negative integers and Z to denote the set of all integers. The nth powers of I and Z are denoted as I^n and Z^n, respectively.

A mesh connected array processor is a $(J^n, \boldsymbol{P})$ where J^n ($\subset Z^n$) is the index set of the array and $\boldsymbol{P}$ is a*matrix of interconnection primitives.*

The matrix of interconnection primitives is

$$\boldsymbol{P} = [\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_j, \ldots, \bar{p}_s] \tag{3.1}$$

where $\bar{p}_j$ is a column vector indicating a unique direction of communication links and s is the number of kinds of directions of a communication link.

For example, consider the array shown in Figure 3.1. Its model is described as $(J^2, \boldsymbol{P})$ where

$$J^2 = \{(j_1, j_2) : 0 \leq j_1 \leq 2, 0 \leq j_2 \leq 2\} \tag{3.2}$$

$$\boldsymbol{P} = \begin{bmatrix} 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \end{bmatrix} \tag{3.3}$$

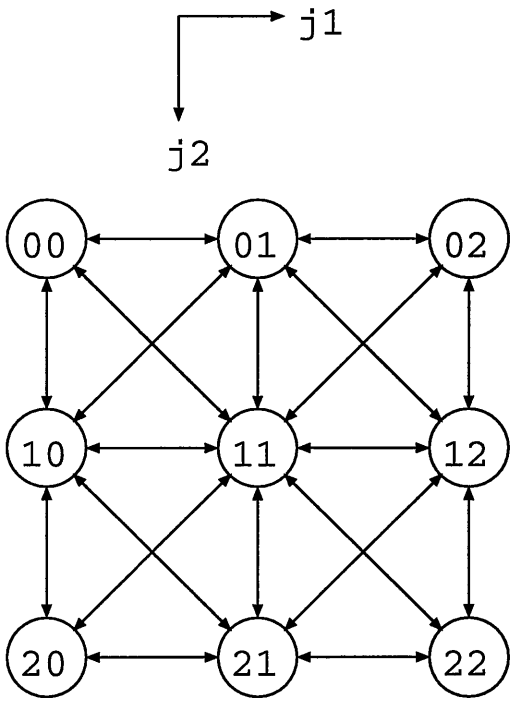


Figure 3.1: Array with 8-neighbor connections.

The important information about an algorithm which we want to include in the model is the data dependences.

The data dependencies can be described as difference vectors of index points where a variable is used and where that variable was generated. With these dependencies we form a *dependence matrix* D from loop program.

$$D = \left[\begin{array}{cccc} \bar{d}_1 & \bar{d}_2 & \cdots & \bar{d}_k \end{array} \right] \tag{3.4}$$

where k is the number of kinds of dependence vectors.

3.2.2 Mapping Algorithms into VLSI Arrays

A *transformation matrix* $\boldsymbol{T}$ which transforms an algorithm A into an algorithm $\hat{A}$ is defined as:

$$\boldsymbol{T} = \left[\begin{array}{c} \boldsymbol{\Pi} \\ \boldsymbol{S} \end{array} \right] = \left[\begin{array}{ccccc} t_{11} & t_{12} & t_{13} & \cdots & t_{1n} \\ t_{21} & t_{22} & t_{23} & \cdots & t_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{n1} & t_{n2} & t_{n3} & \cdots & t_{nn} \end{array} \right] \tag{3.5}$$

where *time mapping function* $\boldsymbol{\Pi}$ and *space mapping function* $\boldsymbol{S}$ are defined as $\boldsymbol{\Pi} : J^n \rightarrow \hat{J}^1$ and $\boldsymbol{S} : J^n \rightarrow \hat{J}^{(n-1)}$. In this paper, we consider only *linear transformation matrix* $\boldsymbol{T}$, i.e., $\boldsymbol{T} \in Z^{n \times n}$.

Thus, using a transformation matrix $\boldsymbol{T}$, $\boldsymbol{D}$ is transformed to a *mappable matrix* $\boldsymbol{\Delta}$:

$$\boldsymbol{\Delta} = \boldsymbol{T}\boldsymbol{D} = \left[\begin{array}{c} \boldsymbol{\Pi} \\ \boldsymbol{S} \end{array} \right] \left[\begin{array}{cccc} \bar{d}_1 & \bar{d}_2 & \cdots & \bar{d}_k \end{array} \right] \tag{3.6}$$

And the index set of algorithm, $\bar{j}^n$ ($\in J^n$) is mapped into $\hat{j}^n$. The first coordinate of $\hat{j}^n$ indicates the time at which the computation indexed by corresponding $\bar{j}^n$ is computed and the rest coordinates of $\hat{j}^n$ indicate the processors where that computations are performed.

Since the mapping $\boldsymbol{\Pi}$ ensures a valid execution ordering, $\boldsymbol{\Pi}$ must satisfy the following condition:

$$\boldsymbol{\Pi}\bar{d}_i > 0 \tag{3.7}$$

where each element indicates the number of time units allowed for its respective variable to travel from the processor where it is generated to processor where it is used.

The mapping $\boldsymbol{S}$ can be selected such that the mappable matrix is mapped into a VLSI array modeled as $(\hat{J}^{(n-1)}, \boldsymbol{P})$. This can be written as:

$$\boldsymbol{SD} = \boldsymbol{PK} \tag{3.8}$$

where $s \times k$ matrix $\boldsymbol{K}$ indicates the utilization of primitive interconnections in matrix $\boldsymbol{P}$.

$$k_{ji} \geq 0 \tag{3.9}$$

$$\sum_j k_{ji} \leq \boldsymbol{\Pi} \bar{d}_i \tag{3.10}$$

Equation (3.9) simply requires that all elements of $\boldsymbol{K}$ matrix are nonnegative and Equation (3.10) required that communication of data associated with dependence vector $\bar{d}_i$ must be done using some primitives $\bar{p}_j$ exactly $\sum_j k_{ji}$ times. This flexibility apparently complicates matters, but in fact, it gives the designer the possibility to choose between a large number of arrays with different characteristics.

We select one transformation matrix which satisfies Equations (3.7) and (3.8) from many candidates.

Once a transformation matrix is selected, then the algorithm is mapped into systolic arrays.

3.3 Optimum Transformation Matrix

The generated matrix $\boldsymbol{\Delta}$ may or may not be the optimum matrix. Then we select the matrix $\boldsymbol{\Delta}$ that will give the optimal or near-optimal systolic array.

We have found that an advantage of using space-time mapping is that the matrix $\boldsymbol{\Delta}$ can be related to the architectural features of implementation of the matrix $\boldsymbol{\Delta}$. Therefore, we do not compare various implementations in a pure abstract terms of area and time, but extract other information such as routing topology, routing complexity, interconnection delays, storage requirements, etc., and use this information in the cost function to compare algorithms.

3.3.1 Conditions for Optimum Transformation Matrix

Since there are a large number of available matrices $\boldsymbol{\Delta}$ for an algorithm, we can reduce candidates of matrices $\boldsymbol{\Delta}$ by addition of the conditions of matrices.

In the following, we describe the conditions of matrix $\boldsymbol{\Delta}$.

Fault-tolerance

To improve the fault-tolerant capability of the array, it is desired that all the data variables in a given algorithm be propagated from one cell to the other and not be stored in the cells.

$$\boldsymbol{S}\bar{d}_i \neq 0 \tag{3.11}$$

No Delay Elements

We define

$$T_{compt} = \sum_{i=1}^{k} |\boldsymbol{\Pi}\bar{d}_i| \tag{3.12}$$

as a measure of number of delay units required within each processor for synchronization, where k is the number of dependence vectors. If $T_{compt} = k$, there are no delay elements. As T_{compt} becomes much larger than k, delay units give a slow and bulky architecture.

Interconnection Factor

Each element of $\boldsymbol{S}\bar{d}_i$ indicates the direction of data which travel from the current processor to another processors. Thus, any $|\boldsymbol{S}\bar{d}_i|$ that is greater than 1 is not desirable, since this indicates absence of nearest-neighbor communication, which is costly in time and area.

$$|\boldsymbol{S}\bar{d}_i| \leq 1 \tag{3.13}$$

This means that the rows of matrix $\boldsymbol{\Delta}$ that represent the space function is fixed from the set as $\{-1, 0, 1\}$.

Valid Execution Ordering

For valid execution ordering,

$$\boldsymbol{\Pi}\bar{d}_i > 0 \tag{3.14}$$

Each element indicates the number of time units allowed for its respective variable to travel from the processor where it is generated to processor where it is used.

Normal Transformation Matrix

Assume that $\boldsymbol{T}$ is a normal matrix. If $\boldsymbol{T}$ is not a normal matrix, a statement of a program needs its own processing element and the other statement needs another processing element.

$$det(\mathbf{T}) \neq 0 \tag{3.15}$$

In this thesis, we select five conditions for optimizing individual cost function. Then we were successful in reducing the search space by defining a practical upper limit.

3.3.2 Cost Function for Optimum Transformation Matrix

We can define the several cost functions which have *silicon area* and *throughput* factors, since matrix $\boldsymbol{\Delta}$ have information about implementation. Therefore, we can compare various arrays of any algorithm without practical data.

Silicon Area

The silicon area A required to implement any given algorithm is obtained by combining the area of processing elements, the area of delay units and the area required for routing the interconnection.

Area of Processing Elements

The area of processing elements A_{PE} is given by

$$A_{PE} = n_{PE} \cdot K_{PE} \tag{3.16}$$

where K_{PE} is the area of one processor and n_{PE} is the number of processing elements required to implement any algorithm.

Area of Delay Units

In Equation (3.12), k is the number of columns in $\boldsymbol{\Delta}$. For example, when $\boldsymbol{\Pi}\bar{d}_i \leq 1$, at least one delay unit is required for data to propagate from one cell to another. Therefore, the number of delay units in a processor is given by $\sum_{i=1}^{k} |\boldsymbol{\Pi}\bar{d}_i - 1|$ and the total number of delay units n_D in the array is given by

$$n_D = n_{PE} \cdot \sum_{i=1}^{k} |\boldsymbol{\Pi}\bar{d}_i - 1| \tag{3.17}$$

Therefore, the total area of the delay units A_D in the array is

$$A_D = n_D \cdot K_D \tag{3.18}$$

where K_D is the area of a delay unit.

Area for Routing the Interconnection

The complexity of the interconnection pattern is

$$C = \sum_{i=1}^{k} (|\boldsymbol{S}_1\bar{d}_i| + 2|\boldsymbol{S}_2\bar{d}_i|) \tag{3.19}$$

where $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are the first and second row vector of $\boldsymbol{S}$ in the case that the size of $\boldsymbol{S}$ is 2×3.

Therefore, the area required for routing the interconnection lines A_L is given by

$$A_L = C \cdot n_{PE} \cdot K_L \tag{3.20}$$

where K_L is the area of a unit length of interconnection line.

The silicon area A required to implement any given algorithm is obtained by Equations (3.16), (3.18) and (3.20).

$$A = n_{PE} \cdot K_{PE} + n_{PE} \cdot \sum_{i=1}^{k} |\boldsymbol{\Pi} \bar{d}_i - 1| \cdot K_D + C \cdot n_{PE} \cdot K_L \tag{3.21}$$

The number of processing elements n_{PE} required to implement an algorithm in a systolic array can be obtained using the following procedure:

for each (i,j,k)
begin
$\hat{j}_{i,j,k} = t_{21}i + t_{22}j + t_{23}k$
$\hat{k}_{i,j,k} = t_{31}i + t_{32}j + t_{33}k$
end
begin

$$n_{PE} = \sum_{\hat{k}_{i,j,k}=\min_{J2}\{\hat{k}_{i,j,k}\}}^{\max_{J2}\{\hat{k}_{i,j,k}\}} (\max_{J^1}\{\hat{j}_{i,j,k}\} - \min_{J^1}\{\hat{j}_{i,j,k}\} + 1)$$

end

where $\hat{j}_{i,j,k}$, $\hat{k}_{i,j,k}$ are the transformed indices of j, k and J^1, J^2 are the sets of $\hat{j}_{i,j,k}$, $\hat{k}_{i,j,k}$ respectively.

Throughput

The throughput of the systolic array is defined as the number of results that can be completed by the array per unit time. This can be measured by the total number of clock cycles required to complete the computation of any given algorithm.

The total number of clock cycles C required for the computation of the algorithm can be determined in the following Equation (3.22).

$$\begin{aligned} C &= \max_{i,j,k}\{\hat{i}_{i,j,k}\} - \min_{i,j,k}\{\hat{i}_{i,j,k}\} + 1 \\ &= \max_{i,j,k}\{t_{11}i + t_{12}j + t_{13}k\} - \min_{i,j,k}\{t_{11}i + t_{12}j + t_{13}k\} + 1 \end{aligned} \tag{3.22}$$

where $\hat{i}_{i,j,k}$ is the transformed index of i.

Interconnection Line Delay

To minimize the delay required for the data to propagate through the systolic architecture, it is desired that the routing between the cells be to the nearest neighbors. This means that the elements in $\boldsymbol{S}_1\bar{d}_i$ and $\boldsymbol{S}_2\bar{d}_i$ should be as small as possible.

The delay of the interconnection line, assuming the Manhattan geometry, is given by

$$t_L = \max_i \sum_{j=1}^{2} |\boldsymbol{S}_j\bar{d}_i| \cdot K_{INT} \tag{3.23}$$

where K_{INT} is the delay of an interconnection line with unit length.

Optimization with Respect to $Area - Time$

One cost function is the product of the silicon area and the total computation time of the algorithm.

$$AT = A \cdot C \cdot (t_C + t_L) \tag{3.24}$$

where t_C is the operating cycle of processor.

Optimization with Respect to $Area - Time^2$

The important factor in this cost function is the computation time factor. We can obtain cost function as Equation (3.25).

$$AT^2 = A \cdot C^2 \cdot (t_C + t_L)^2 \tag{3.25}$$

We can select one function from two cost functions, Equations (3.24) and (3.25), according to our purpose. The computation time factor in Equation (3.25) is more important than the computation time factor in Equation (3.24).

Chapter 4

Parallel Processing Architecture for Linear Programming

4.1 Introduction

In this chapter, we propose the parallel processing architecture for linear programming. As a parallel processing method, we use systolic architecture and the space-time mapping method.

First, in Section 4.2, the architecture for fast parallel processing was designed, which consists of three modules. In this configuration, we can start computations from three starting points. Then we can improve parallelism. This architecture can solve fixed size linear programming problems.

In Section 4.3, the architecture that can solve any large size linear programming was presented. That architecture consists of three kinds of chips. Then if we add chips which contain systolic arrays, we can solve any large size linear programming problems.

4.2 Simple Architecture for Parallel Processing

4.2.1 Systolic Array of Step 8

One example will be given to help the explanation of the space-time mapping method. To use the space-time mapping method, first, the algorithm of step 8 (Equation (2.13)) is expressed as follows:

for k=1 to K
  for i=1 to m
    for j=1 to m
      $(B^{-1})^{k}_{i,j} = (B^{-1})^{k-1}_{i,j} + (\eta^*)^{k-1}_{i}(B_p^{-1})^{k-1}_{j}$

where $(B_p^{-1})^{k-1}_{j} = (B^{-1})^{k-1}_{p,j}$ and the k-th term of $(B^{-1})_{i,j}$ is denoted as $(B^{-1})^{k}_{i,j}$ and K is the number of iterations when this computations are finished. Really, $K = 1$ but for space-time mapping, index k is shown explicitly.

The dependence matrix is obtained from data dependences as follows:

$$\boldsymbol{D} = \begin{array}{c} \\ k \\ i \\ j \end{array} \begin{array}{c} \begin{array}{ccc} B^{-1} & B_p^{-1} & \eta^* \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \tag{4.1}$$

where columns of $\boldsymbol{D}$ are corresponding to data in right-hand-side of the assignment statements of nested loop program.

Because there are many possible transformation matrices, we have to choose one transformation matrix among them using the conditions (Section 3.3.1) and the cost function for optimum transformation matrix (Section 3.3.2). Table 4.1 shows only the best results of evaluation of dependence matrix of Equation (4.1). This result uses the Northern Telecom CMOS 3 μm double metal layer technology [19], which determine the constant parameters in the cost functions that are shown in Table 4.2.

Transformation matrix	Area-Time ($\mu m^2 \cdot s$)	Area-Time^2 ($\mu m^2 \cdot s^2$)
T_1	6.4051E+01	3.2570E-05
T_2	6.4051E+01	3.2570E-05
T_3	6.4051E+01	3.2570E-05
T_4	6.4051E+01	3.2570E-05
$\vdots$	$\vdots$	$\vdots$

Table 4.1: The best results of an evaluation of the dependence matrix of step 8.

Parameters	Values
Area of a processor (K_{PE})	$2.5 \times 10^6 \mu m^2$
Area of a delay unit (K_D)	$5 \times 10^4 \mu m^2$
Area of a unit length of interconnection line (K_L)	$4.8 \times 10^3 \mu m^2$
Unit length interconnection delay (K_{INT})	$1.7ns$
Frequency of operation of the processor (t_C)	$10MHz$

Table 4.2: The values for the constant parameters in the cost functions.

The four transformation matrices are very similar to each other. They are:

$$\boldsymbol{T}_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ -1 & 0 & 0 \end{bmatrix}, \boldsymbol{T}_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \boldsymbol{T}_3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ -1 & 0 & 0 \end{bmatrix}, \boldsymbol{T}_4 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix} \quad (4.2)$$

Then the mappable matrices are obtained by Equations (4.1) and (4.2).

$$\boldsymbol{\Delta} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ -1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ -1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix} \quad (4.3)$$

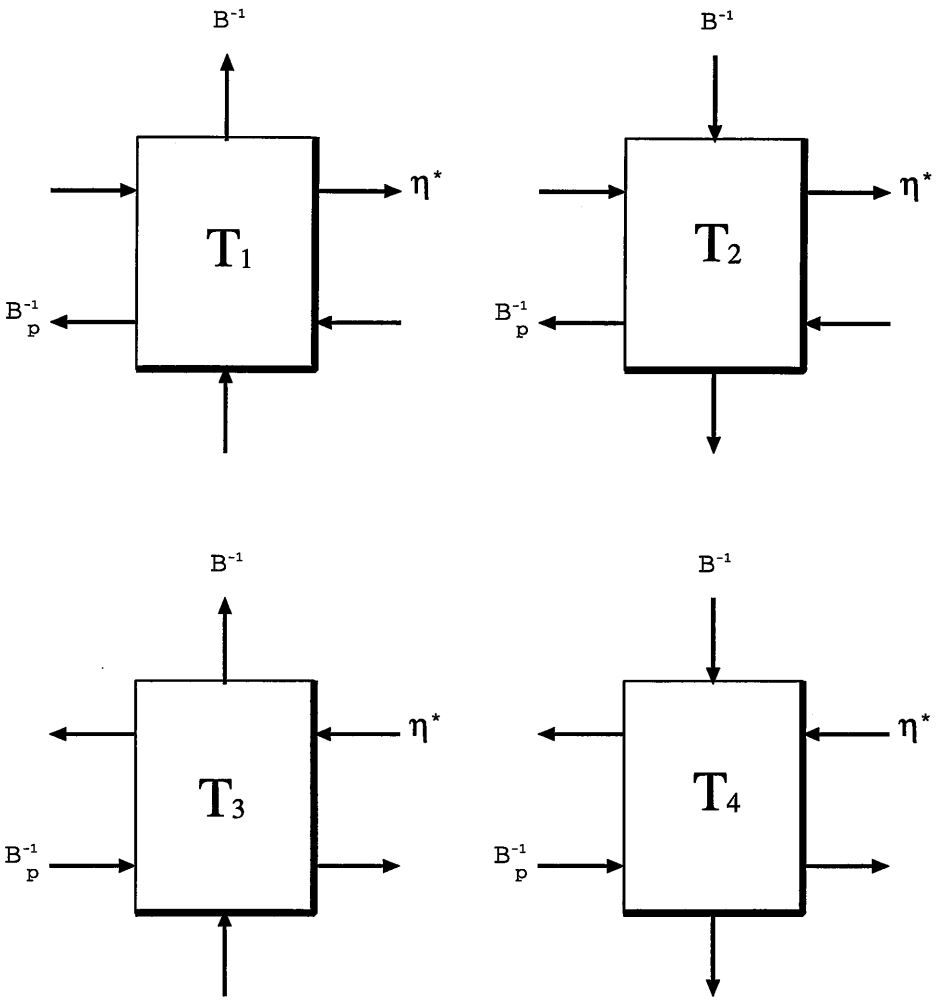


Figure 4.1: The cells of the four mappable matrices.

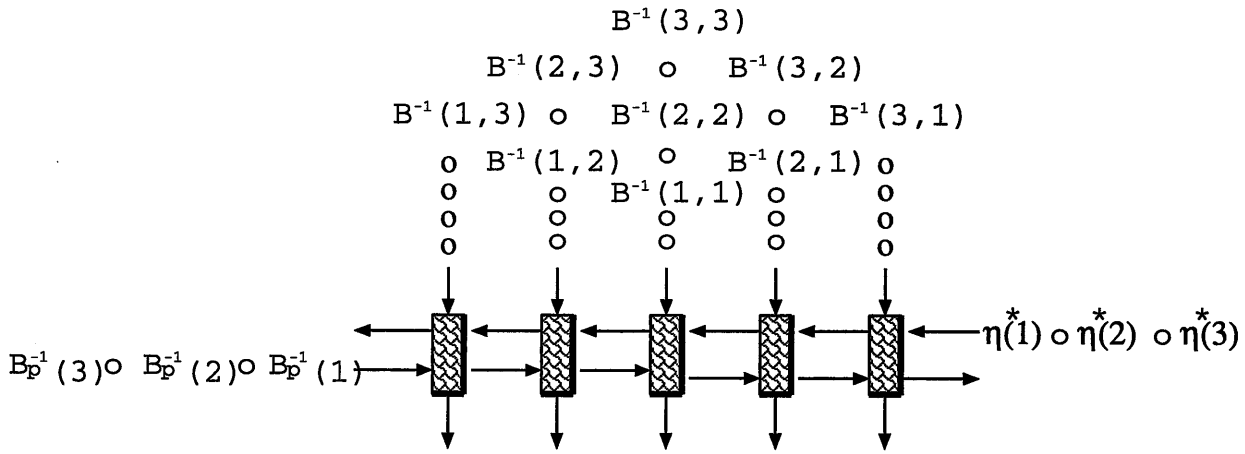


Figure 4.2: The systolic array of T_4.

The cells of the four mappable matrices are shown in Figure 4.1. In the four transformation matrices, we select $\boldsymbol{T}_4$. Then the mapping of the index set is given as follows:

$$\begin{bmatrix} t \\ x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} k \\ i \\ j \end{bmatrix} = \begin{bmatrix} k+i+j \\ i-j \\ k \end{bmatrix} \tag{4.4}$$

where t is the time at which the computation indexed by (k,i,j) is computed and (x,y) indicates the processor where that computation is performed.

The systolic array corresponding to this transformation matrix is shown in Figure 4.2 and Figure 4.3 is the structure of the cell. Each cell has simple circuits such as adder, multiplier, and delay units.

4.2.2 Systolic Arrays for the Revised Simplex Method

Other steps of algorithm are also designed in similar manner. Figures 4.4, 4.5, 4.6 and 4.7 show the structures of arrays and these cells. These figures are the results of revised simplex method in the case of $m = 3$, $n = 9$.

The whole result of this case is Figure 4.8. This array consists of three modules. The

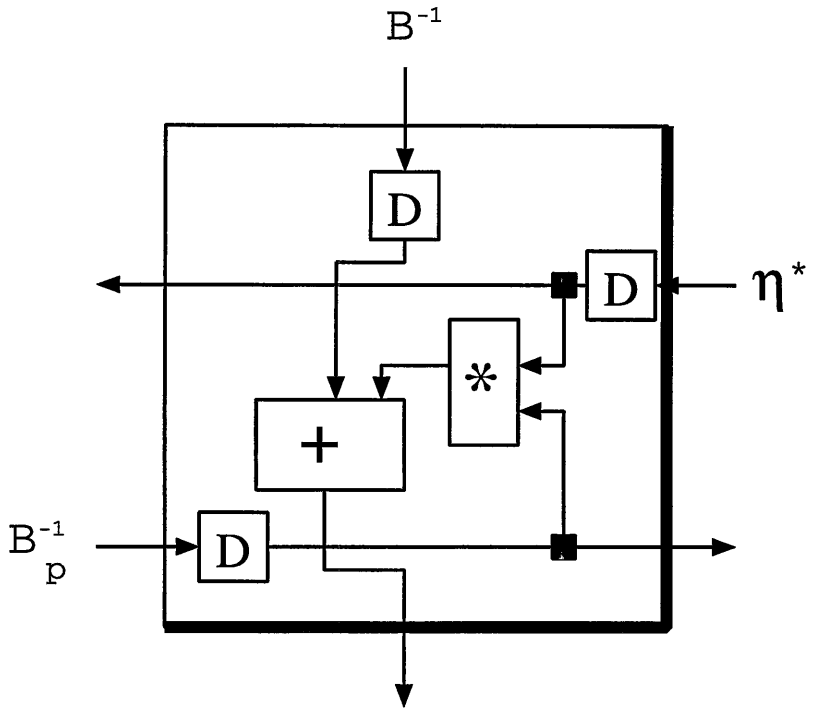


Figure 4.3: The structure of cell of T_4.

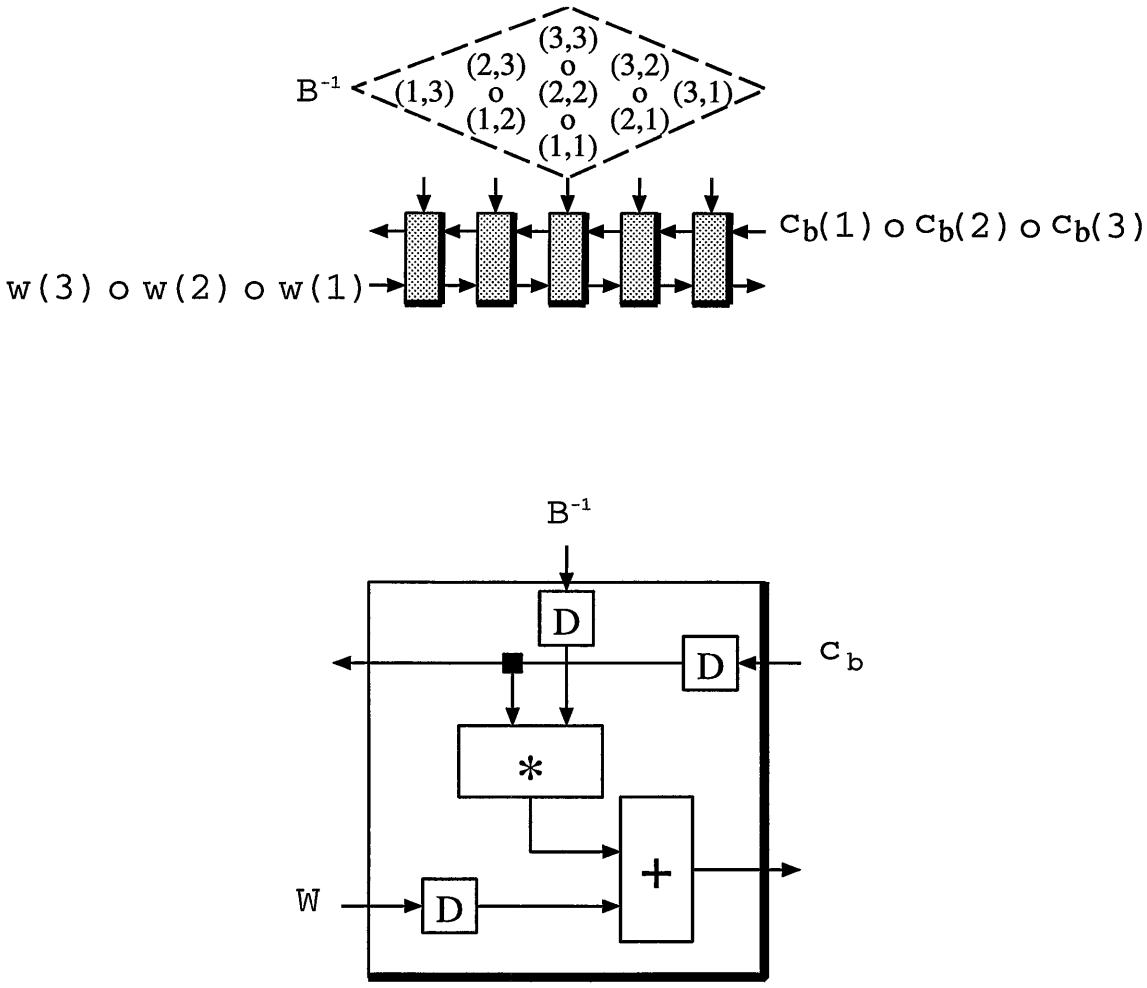
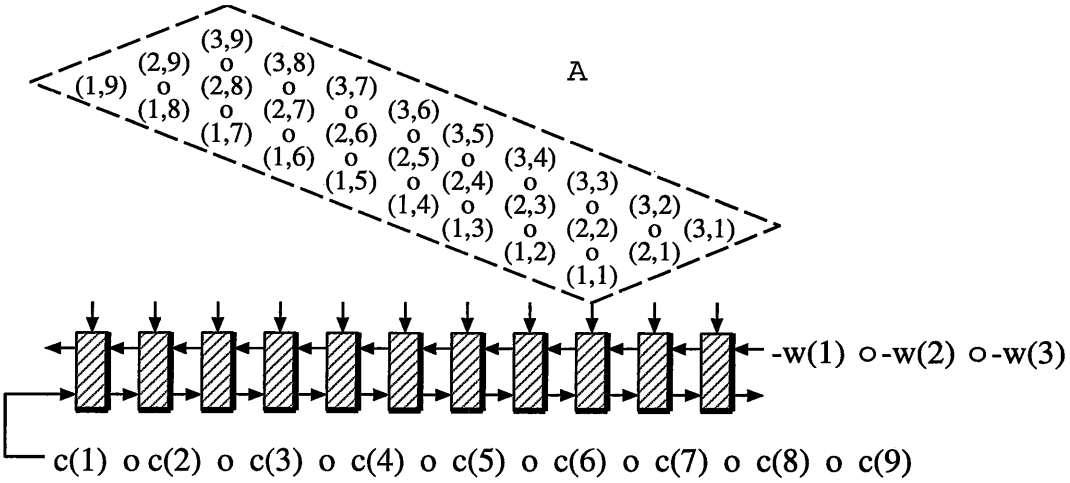


Figure 4.4: The structure of array and the cell of step 1.



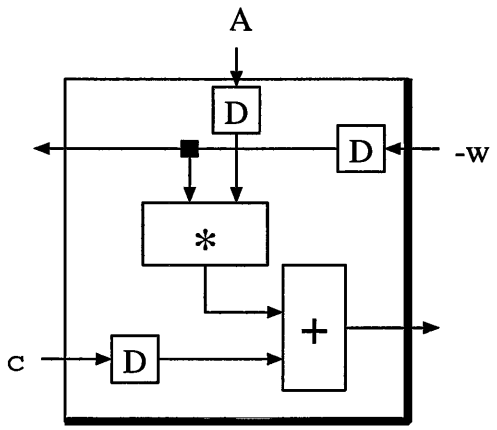


Figure 4.5: The structure of array and the cell of step 2.

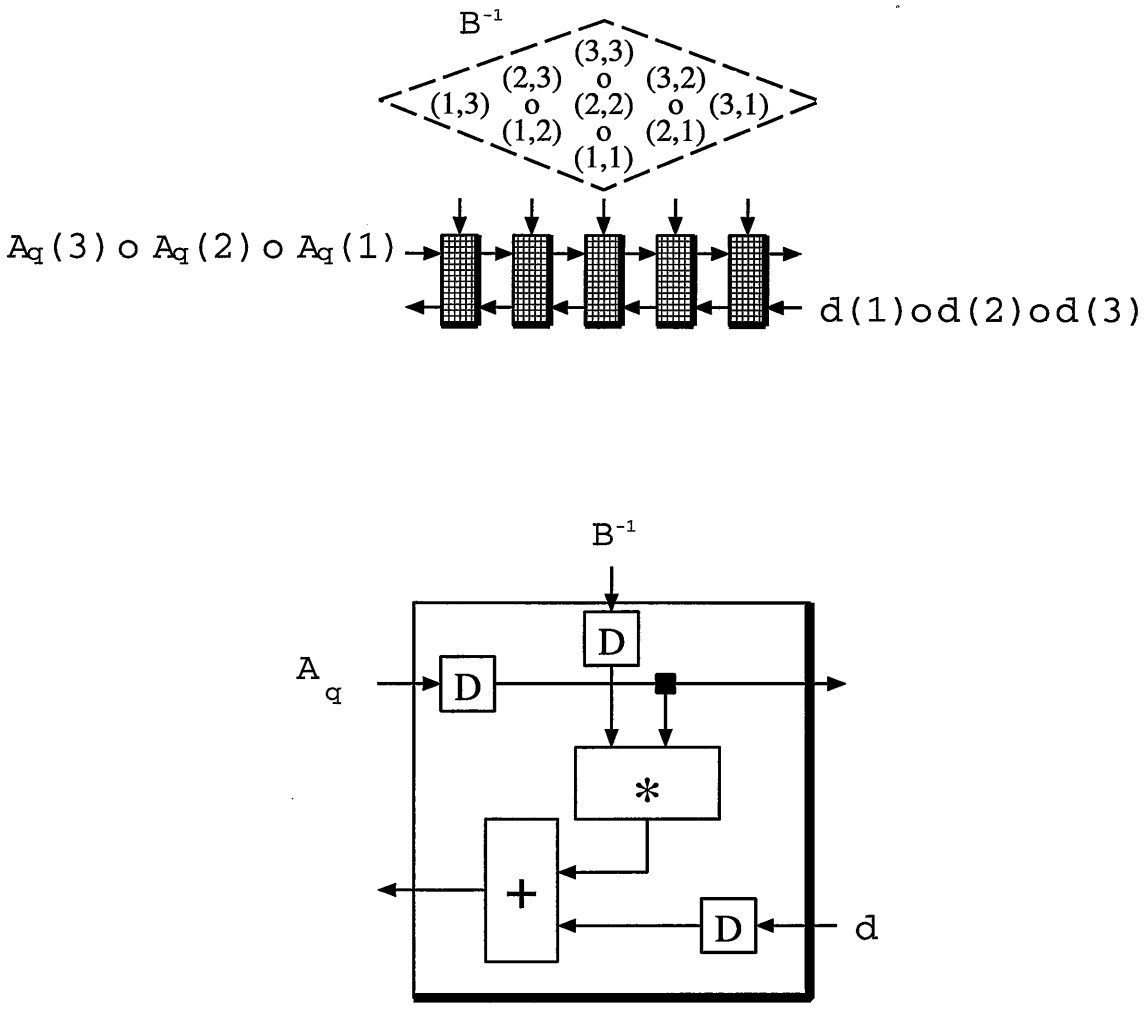
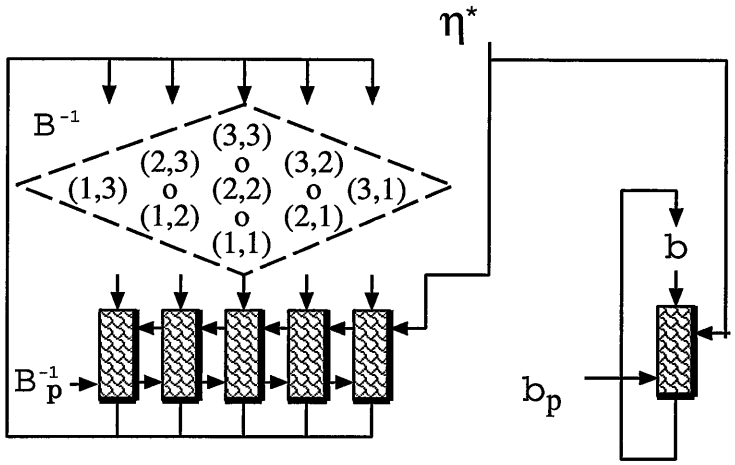


Figure 4.6: The structure of array and the cell of step 4.



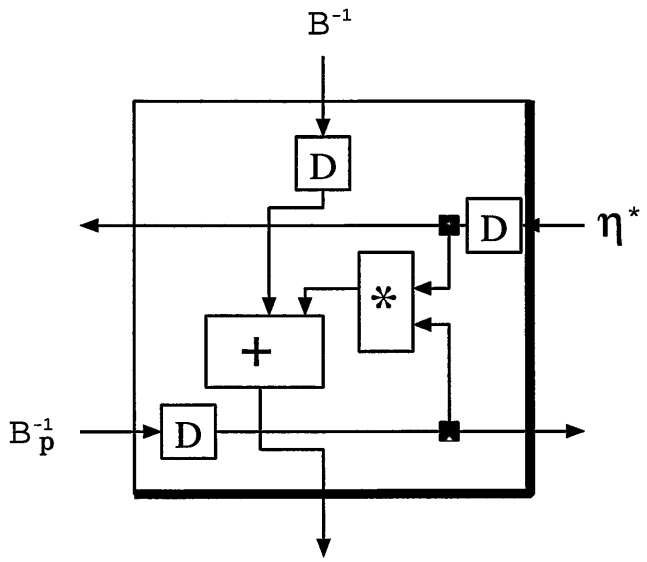


Figure 4.7: The structure of array and the cell of step 8.

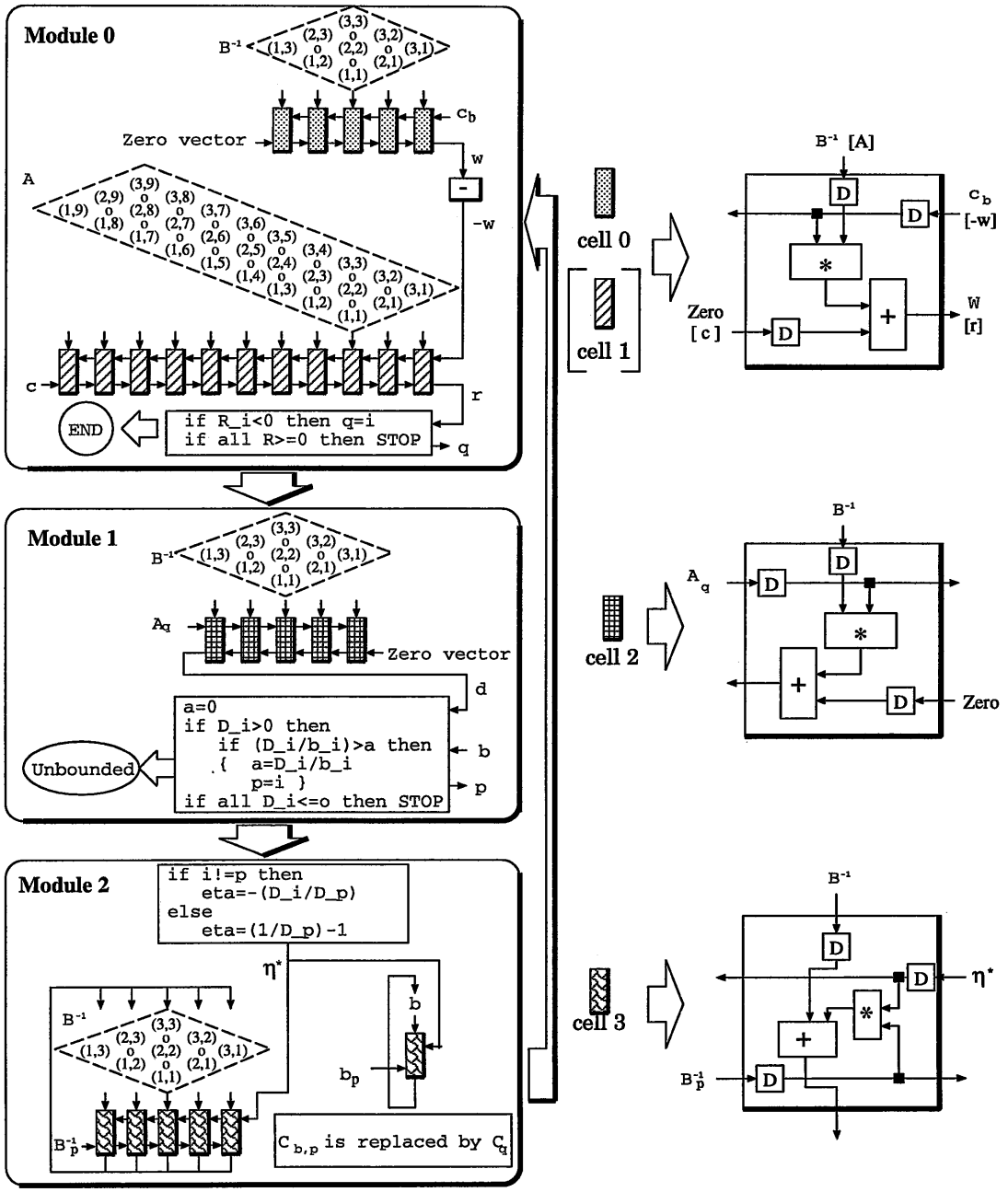


Figure 4.8: The systolic array for revised simplex method in the case of $m = 3$, $n = 9$.

module 0 has circuits for step 1 (Equation (2.6)), step 2 (Equation (2.7)) and step 3 (Equation (2.8)). For step 1, an array that has $(2m-1)$ cells (cell 0) are used. Elements of $\boldsymbol{B}^{-1}$ and $\boldsymbol{c}_b$ are inputed into the array and the vector $\boldsymbol{w}$ is put out. $\boldsymbol{w}$ is converted to $-\boldsymbol{w}$. Next, $-\boldsymbol{w}$ is inputed into the array for step 2, which has $(n-1)$ cells (cell 1) and calculates the vector $\boldsymbol{r}$. Then the index q is obtained if some elements of $\boldsymbol{r}$ are negative, and the computation is terminated if no element of $\boldsymbol{r}$ is negative (step 3).

The module 1 has circuits for step 4 (Equation (2.9)), step 5 (Equation (2.10)) and step 6 (Equation (2.11)). A circuit for step 4 has $(2m-1)$ cells (cell 2). Elements of $\boldsymbol{B}^{-1}$ and $\boldsymbol{A}_q$ are inputed into the circuit and the vector $\boldsymbol{d}$ is put out. If no element of $\boldsymbol{d}$ is positive, then the computation is terminated and this problem is unbounded (step 5). Otherwise an index p is obtained (step 6). To simplify initializing of step 6, we have designed the process of searching for the maximum of the reciprocal number instead of the minimum.

The module 2 has circuits for step 7 (Equation (2.12)) and step 8 (Equation (2.13)). The modified eta vector is computed (step 7). Circuit for step 8 whose design is shown in the previous section, has $(2m-1)$ cells for pivoting $\boldsymbol{B}^{-1}$ and one cell for pivoting $\boldsymbol{b}$ (cell 3). And it has a circuit for updating basic cost vector.

After the computation of module 2, the process starts again from module 0 until the final optimal solution is found.

Cell 0 and cell 1 in module 0 are the same. Cell 2 in module 1 is obtained by mirror transformation of cell 0 or cell 1. Cell 3 in module 2 is very similar to cell 0 and cell 1. Therefore we can design the revised simplex method with only three kinds of cells. Each cell has simple circuits such as adder, multiplier and delay circuits.

In order to use the same cells without changing the data stream of $\boldsymbol{B}^{-1}$, we can unify cell 0 (or cell 1) and cell 2 as Figure 4.9. Two 2:1 multiplexers and 1:2 decoder are included in the unified cell. If the signal Ctrl is '0', then the unified cells operate as cell 0. However, if the signal Ctrl is '1', then they operate as cell 2.

This systolic architecture can reduce the time required for solving problem as Table 4.3. Therefore, this systolic architecture can solve a linear programming problem faster than sequential computation.

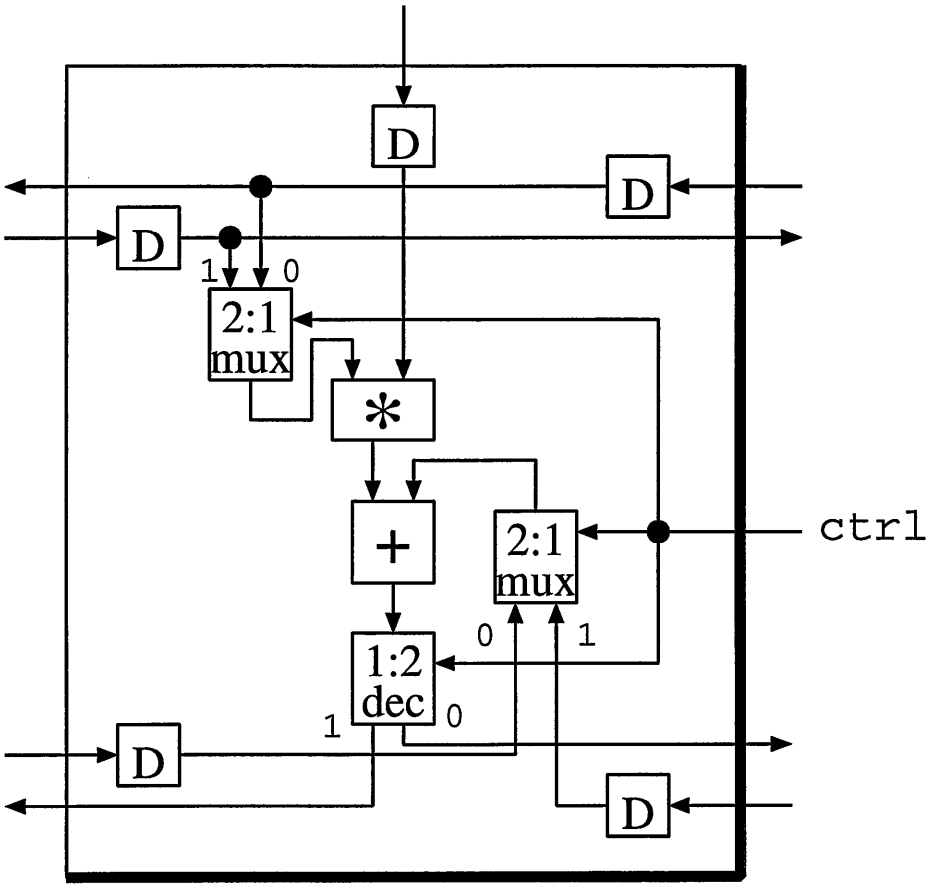


Figure 4.9: The structure of the unified cell.

Steps	Sequential Computation	Using Systolic Array
step 1	$O(m^2)$	$O(4m-2)$, $(2m-1)$ cells
step 2	$O(mn)$	$O(3n-m-2)$, $(n-1)$ cells
step 4	$O(m^2)$	$O(4m-2)$, $(2m-1)$ cells
step 8	$O(m^2)$	$O(4m-2)$, $(2m-1)$ cells

Table 4.3: Comparison of sequential computation with computation using systolic array.

Assume that τ_0 is the operating time required by the cells. Then the total time of module 0 is $(3n+5m)\tau_0$. Module 1 is $4m\tau_0$ and module 2 is $4m\tau_0$.

Because module 0 takes the most time, data between modules has to be changed after computations of module 0 are finished. Because index q of the result of module 0 is used in module 1, module 0, and module 1 can't be executed simultaneously. Also module 2 uses index p of the result of module 1. So only one module among three modules can perform computations at a time.

4.2.3 Systolic Array for Parallelism Improvement

In Section 4.2.2, we proposed the systolic array to solve linear programming (Figure 4.8). However in the design, only one module among three modules can be performed. Here we propose a more efficient model.

In order to improve parallelism, we start computations from three starting points. In other words, after the computation of module 0 is finished, we can obtain three starting points which have negative values of $\boldsymbol{r}$. Figure 4.10 shows the efficient usage of the array of Figure 4.8. Each controller stores information ($\boldsymbol{B}^{-1}$, $\boldsymbol{b}$, $\boldsymbol{c}_b$, p, q) in its local memory. The switching network connects three modules to three controllers.

The computations of three modules are finished and then the results of computations are stored in each controller. Then three controllers are connected to next three modules by the switching network. The switching network has three patterns as Figure 4.11 shows. If any module among three modules finds the optimal solution, the computations

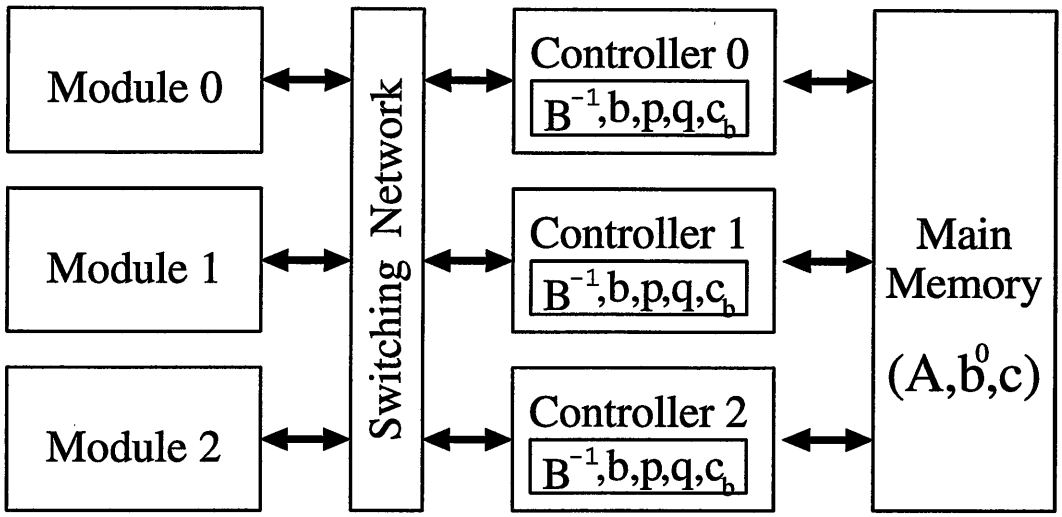


Figure 4.10: Efficient usage of an array of the revised simplex method.

are finished.

Figure 4.12 shows the pipeline diagram of the application of the array of the revised simplex method. Therefore, we can obtain the optimal solution faster than the initial design.

4.3 Architecture for Large Scale Linear Programming

We proposed the simple architecture in Section 4.2, but that design can solve only fixed size linear programming problems. In practice, the sizes of linear programming problems are very large. Then we propose an architecture which can solve any large size linear programming problems using separating into several chips.

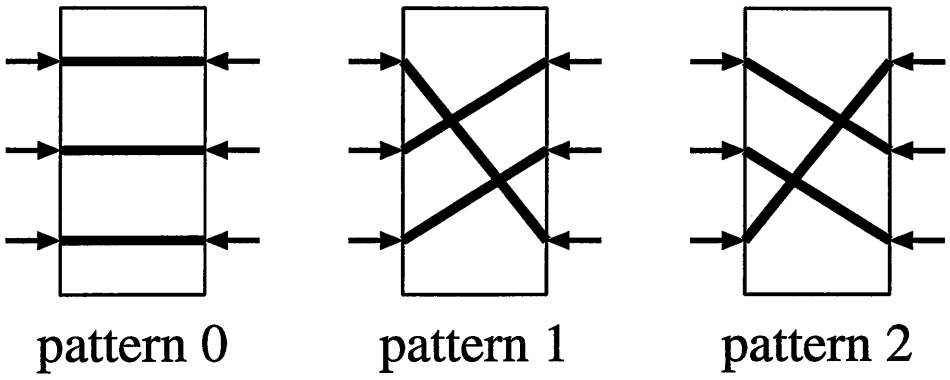
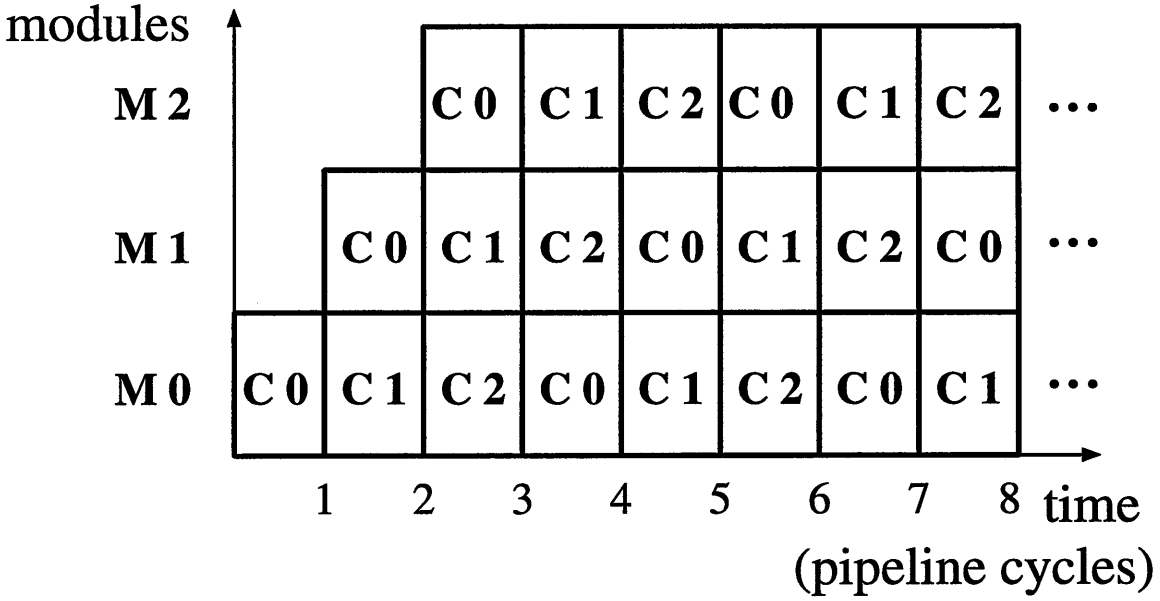


Figure 4.11: Three patterns of the switching network.



C i : computation of i-th starting point
(for i=0,1,2)

Figure 4.12: Pipeline diagram of application of the array.

4.3.1 Modified Systolic Array for Large Scale Linear Programming

As the sizes of linear programming problems are larger, cells are increased. Therefore, we separate circuits into three kinds of chips which are called Chip A, Chip B, and Chip C.

Chip A has the unified cells, cell 3, 1:2 decoder, and 2:1 multiplexer. Chip B has a cell 3, a 1:2 decoder, and several circuits. Chip C has 1:2 decoder and cell 1. And each chip has local memory.

Figure 4.13 shows the modified design in the case that Chip A, Chip B, and Chip C can solve linear programming problem of $m = 3$, $n = 9$. We can convert the function of Chip A by utilizing control signals such as Ctrl1 and Ctrl2 as Table 4.4.

Control signals (Ctrl1, Ctrl2)	Function of Chip A
00	Step 1
01	Step 4
1x	Step 8

Table 4.4: The relationship of control signals to function of Chip A; 'x' is the Don't care signal.

Signal Select is utilized to select chips. If signal Select is '0', then data passes through the chip and goes to next chip. If signal Select is '1', then that is the final chip and data can't go to next chip.

First, in local memory of each chip, we store all of information about a problem such as $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c_b}$, $\boldsymbol{c}$. In Chip A, Equations (2.6), (2.9), and (2.13) are computed. Equations (2.8), (2.10), (2.11), and (2.12) are computed in Chip B. In Chip C, Equation (2.7) is computed.

If the sizes of linear programming problems are large, we can solve problems by appending more chips of Chip A and Chip C. Figure 4.14 shows connections of chips in the case of k Chip As and k Chip Cs. Each bit of Select selects a Chip A and a Chip C. Assume that Chip A can include s unified cells and s cell 3s and Chip C can include q

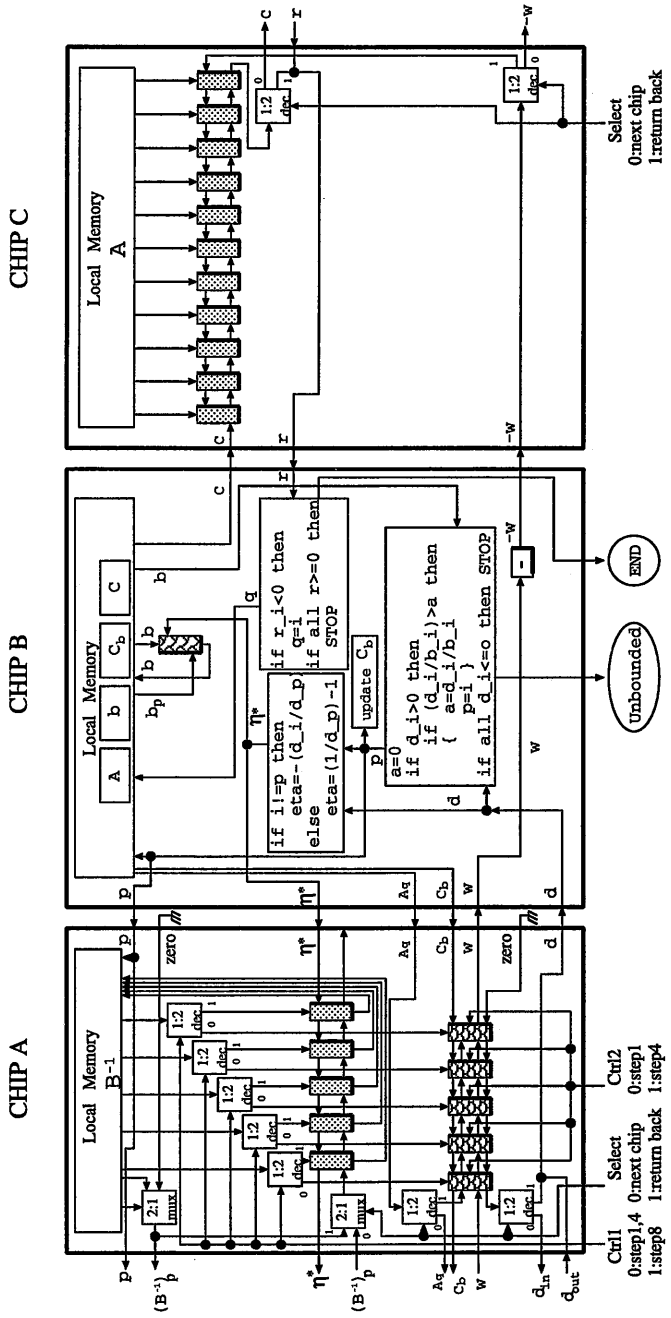


Figure 4.13: The modified architecture which consists of three chips.

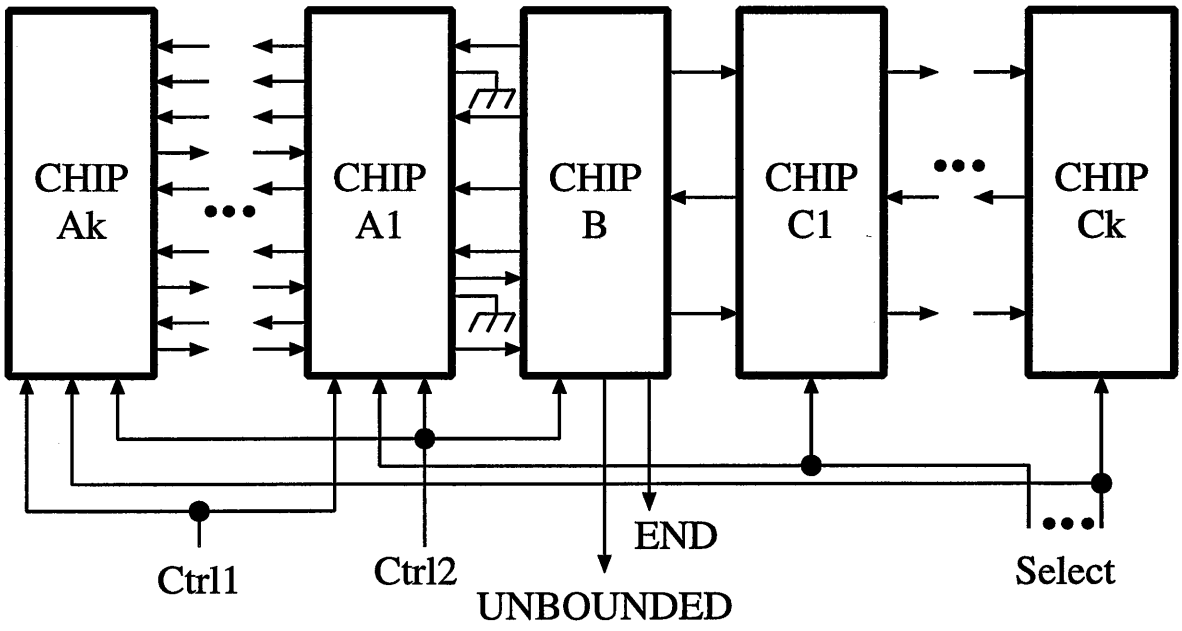


Figure 4.14: The connections of chips.

cell 1s by current VLSI technology. The number of chips which are required to solve an $m \times n$ size linear programming problem is evaluated in Table 4.5.

Name of Chips	The Number of Chips
Chip A	$\lceil \frac{2m-1}{s} \rceil$
Chip B	$\lceil \frac{n-1}{q} \rceil$

Table 4.5: The number of chips that are required to solve an $m \times n$ size linear programming problem.

4.3.2 Distributing Data of Problems to Chips

Each chip has its local memory which has information about problems such as $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c_b}$, $\boldsymbol{c}$. Before the computation is started, we have to store the data of problem in each chip.

Assume that the size of problem is $m \times n$ and each chip can include r cells. Equation (4.5) shows the relation the data of a problem and the cells of a chip.

$$rx - \lfloor \frac{r}{2} \rfloor \leq i - j \leq rx + \lfloor \frac{r}{2} \rfloor \tag{4.5}$$

The data of a problem are distributed into chips by the following equation.

$$x = \lfloor \frac{i - j + \lfloor \frac{r}{2} \rfloor}{r} \rfloor \tag{4.6}$$

$$c = (i - j + \lfloor \frac{r}{2} \rfloor) \ mod \ r \tag{4.7}$$

$$for \begin{cases} 1 \leq i \leq m, 1 \leq j \leq m & ChipA \\ 1 \leq i \leq m, 1 \leq j \leq n & ChipC \end{cases}$$

where i and j are indices of matrix $\boldsymbol{B}^{-1}$ and $\boldsymbol{A}$, and x is the number of chips. c means the number of cells in chips.

Figure 4.15 shows an example in the case of $r = 3$. Equation (4.6) distributes the data of problems, $\boldsymbol{A}$ and $\boldsymbol{B}^{-1}$, into c-th cells of x-th chips.

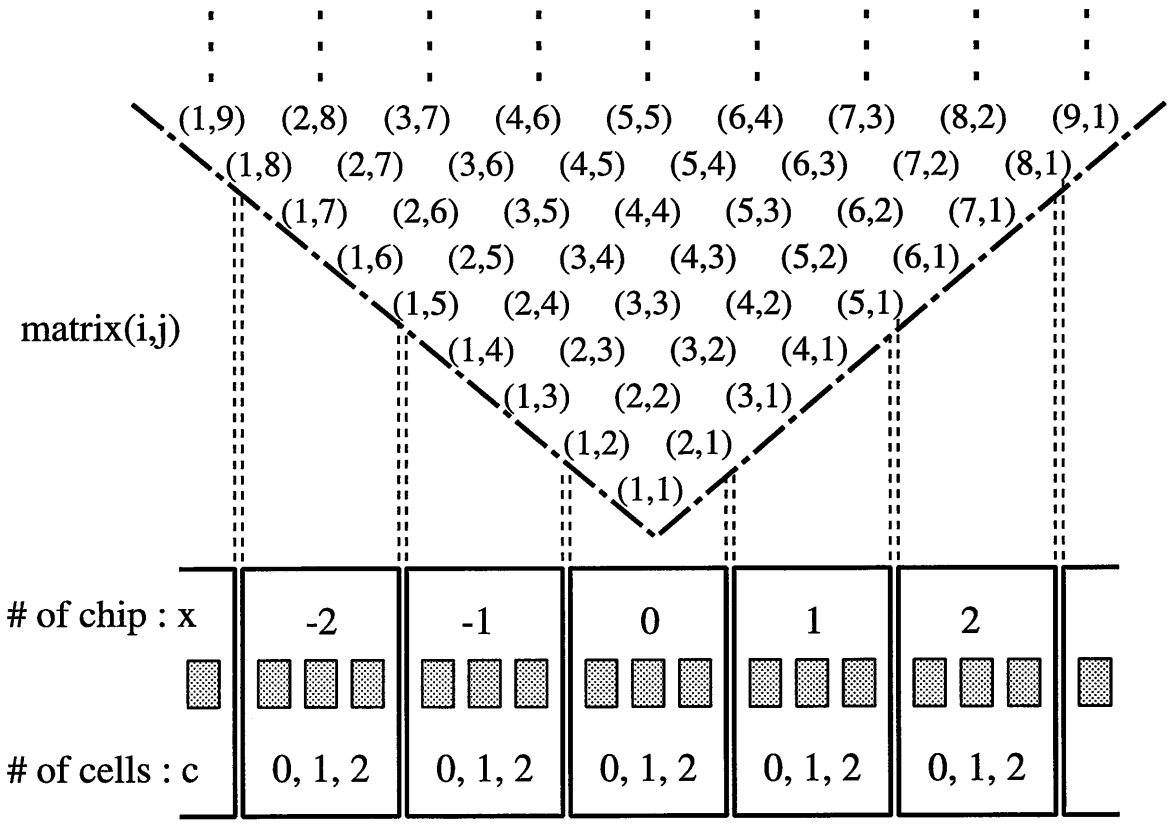


Figure 4.15: The example of data distribution in the case of $s = 3$.

4.3.3 Simulation of Cell 3 and the Unified Cell

In this section, we simulate cell 3 (Figure 4.8) and the unified cell (Figure 4.9). Two cells have several units such as adder, multiplier, and delay units.

In mathematics, there are infinitely many positive and negative integers. However, in a practical hardware system only a fixed number of integers can be represented, based on the number of bits allocated to the representation. In this thesis, we use 8-bit binary quantities.

For an n-bit multiplicand and an n-bit multiplier, the resulting product will be $2n$ bits. Thus, the product of two 8-bit numbers required 16 bits. Since the resulting products are entered into the 8 bits input ports of next cells, we simulate cells in which data bus is 8 bits.

We describe the cells in Verilog-HDL to evaluate the features of the architecture in relation to its implementation. We can obtain behavioral Verilog-HDL descriptions from the previous sections. A verification of the description correctness was performed by Verilog-XL of Cadence company. Tables 4.6 and 4.7 show the specification of behavioral models of two cells.

Kinds of Ports	bits	Names of Ports	Operations
input	8	IN_1	η^*
input	8	IN_2	B_p^{-1}
input	8	IN_3	B^{-1}
input	1	CLK	clock signal
output	8	OUT_1	IN_1
output	8	OUT_2	IN_2
output	8	OUT_3	IN_1 * IN_2 + IN_3

Table 4.6: The specification of behavioral models of cell 3.

Kinds of Ports	bits	Names of Ports	Operations
input	8	IN_1_1	c_b
input	8	IN_1_2	A_q
input	8	IN_2_1	w
input	8	IN_2_2	d_{in}
input	8	IN_3	B^{-1}
input	1	CLK	clock signal
input	1	CTRL	control signal
output	8	OUT_1_1	IN_1_1 (if CTRL==0)
output	8	OUT_2_1	IN_1_1 * IN_3 + IN_2_1 (if CTRL==0)
output	8	OUT_1_2	IN_1_2 (if CTRL==1)
output	8	OUT_2_2	IN_1_2 * IN_3 + IN_2_2 (if CTRL==1)

Table 4.7: The specification of behavioral models of the unified cell.

Figures 4.16 and 4.17 show the timing waves of cell 3 and the unified cell.

The cursor mark shows the current values of computation such that OUT_1=IN_1=6, OUT_2=IN_2=6, OUT_3=IN_1*IN_2+IN_3=6*6+0=36 in Figure 4.16. 'x' value of output signals (black region) in Figure 4.17 means that the value is unknown.

The next step is the synthesis of two cells. Table 4.8 summarizes the result of the automatic synthesis process of Design Analyzer of Synopsys company, based on NEL (NTT Electronic Technology) library in which 0.5 μm CMOS technology is used. In this automatic synthesis process, there are two modes such as area-minimum mode (Case-1) and delay-minimum mode (Case-2).

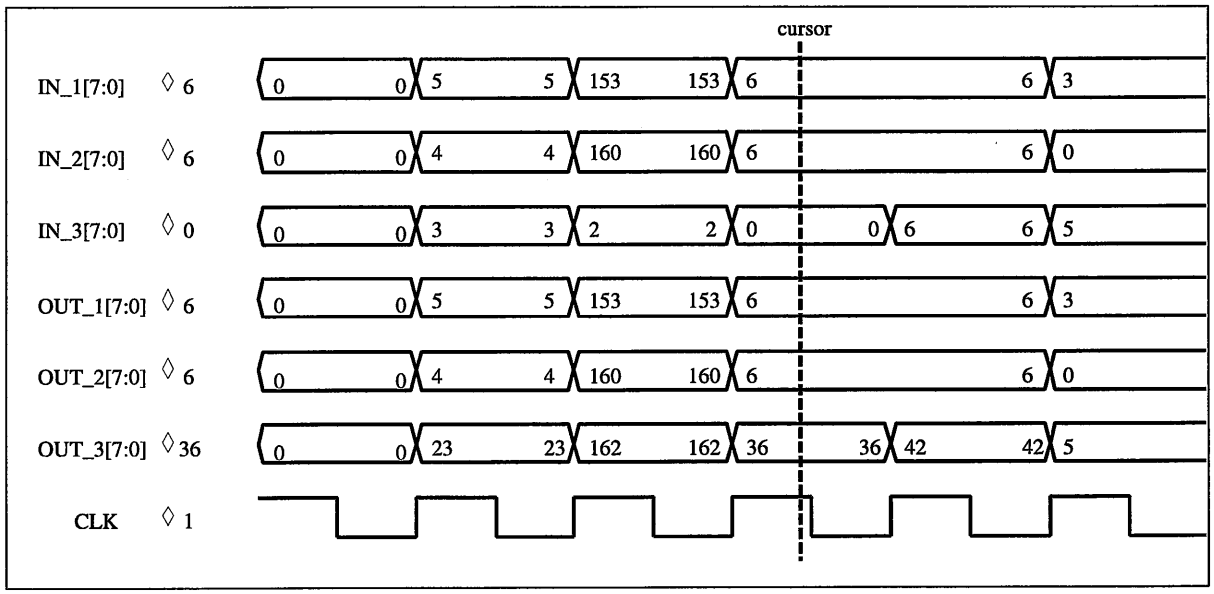


Figure 4.16: The timing wave of cell 3.

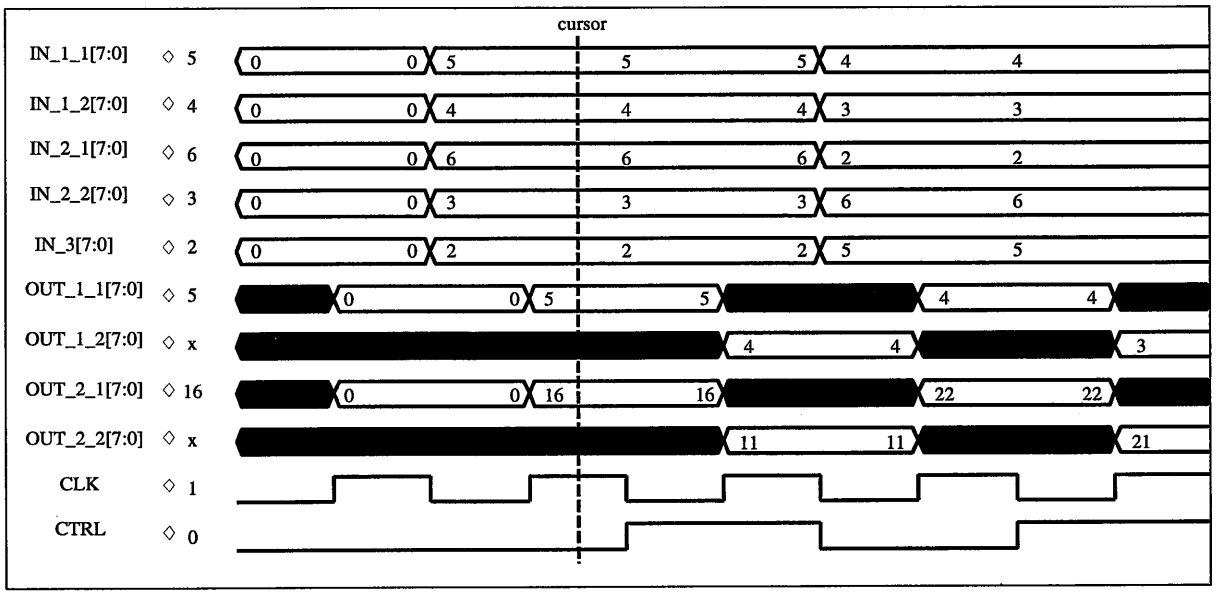


Figure 4.17: The timing wave of the unified cell.

Case (Factor)	Area [μm^2]	Delay [ns]	Power [mW]	Standard Cells
Case-1 (Area)	215130	0.78	136.5686	223
Case-2 (Delay)	213000	0.76	130.2947	208

Table 4.8: The result of automatic synthesis for cell 3.

Case (Factor)	Area [μm^2]	Delay [ns]	Power [mW]	Standard Cells
Case-1 (Area)	386950	0.78	260.5641	428
Case-2 (Delay)	382690	0.76	251.1768	398

Table 4.9: The result of automatic synthesis for the unified cell.

Tables 4.8 and 4.9 show clearly that the result of Case-2 is better than Case-1 at the points of area and delay.

How many cells will be implemented in a chip? We can roughly compute by Tables 4.8 and 4.9 as follows:

Size of Chip	Cell 3 (Case-2)	Unified Cell (Case-2)
2.3 × 2.3 mm^2	24	13
4.8 × 4.8 mm^2	108	60
7.2 × 7.2 mm^2	243	135

Table 4.10: The number of cells that are implemented in a chip.

Assuming that the frequency of operation of the cells is 1 MHz and the size of a problem is m=100, n=1000, the computation time of each module is obtained as Table 4.11.

Modules	Computation Time [ns]
module 0	3500
module 1	400
module 2	400
Total	4300

Table 4.11: The computation time of each module in the case that the frequency of operation of the cells is 1 MHz.

Table 4.11 shows whole steps (from step 1 to step 8) can be computed in a few millisecond.

Chapter 5

Concluding Remarks

5.1 Conclusion

In this thesis, we have proposed an architecture for linear programming. The aim of linear programming is to derive an optimum use of resources in industry and in organizations. Linear programming is concerned with problems in which a linear objective function in terms of decision variables is to be optimized while a set of linear equations, inequalities, and sign restrictions are imposed on the decision variables as requirements. As the number of equations is usually very large, it takes a lot of computation time to find a optimal solution. Therefore, special-purpose hardware is sought for high-speed linear programming.

As a solution to the special-purpose architecture, we introduce systolic architecture which have been proposed for VLSI implementation. A large number of systolic architectures have been developed and formal methods of design of systolic arrays have been proposed. In this thesis, we used space-time mapping method given by Moldovan and could obtain optimal systolic architecture by optimal transformation matrix which has minimum $area \times time^2$.

In this architecture (Section 4.2.2), only one module among three modules can be performed and then we proposed a parallel processing architecture in which all three

modules are working (Section 4.2.3). Each module has information of a starting point, then we can obtain the optimal solution faster than the initial architecture.

In practice, the sizes of linear programming problems are very large. Therefore, we proposed an architecture which can solve linear programming problems of any large size by separating circuits into three kinds of chips which are called Chip A, Chip B, and Chip C. Chip A and Chip C are consisted of systolic cells and another circuits are contained in Chip B. If the sizes of linear programming problems are large, we can solve problems by appending more chips of Chip A and Chip C.

In addition, we simulated cell 3 and the unified cell using 0.5 μm CMOS technology. By the result of simulation, we can compute how many cells in a chip and how long it takes.

5.2 Further Works

In this thesis, we proposed a parallel processing architecture. In order to use this architecture to solve linear programming problems, we have to implement this architecture using VLSI technology.

Another further work in this direction is to find fast and efficient algorithms. For example, the interior-point method finds an optimal solution faster than simplex method by finding a better direction for the next move. However, this approach takes more computational time in finding a moving direction than the simplex method. Therefore, it is necessary to consider parallel processing of another method.

Acknowledgments

I am indebted to a number of individuals who, either through discussions or by providing the facilities for my study, have contributed to the preparation of this thesis.

I express my appreciation to Prof. Hirotomo Aso in Tohoku University and I express my appreciation to Prof. Ken-ichi Abe and Prof. Masayuki Kawamata whose comments about the preliminary examination have been so helpful.

I also wish to thank Dr. Shin'ichiro Omachi and parallel group members of Aso lab. in Tohoku University.

In addition, I express my appreciation to our students at Aso lab. in Tohoku University.

Finally, I want to express my special thanks to God.

Bibliography

[1] G.L.Reijns, J.Luo, and F.Bruggeman, "Parallel Processing of Linear Programming Problems," *T.S.I.*, vol.10, no.4, 1991.

[2] J.Luo and G.L.Reijns, "Parallel Solution of the Revised Simplex Method and Interior Point Methods for Linear Programming," *International Symposium,* vol.2, pp.723-736, 1991.

[3] J.K.Ho, T.C.Lee, and R.P.Sundarraj, "Decomposition of Linear Programs Using Parallel Computation," *Mathematical Programming* 42, pp.391-405, 1988.

[4] G.B.Dantzig and P.Wolfe, "The Decomposition Principle for Linear Programs," *Operations Research* 8, pp.101-111, 1960.

[5] M.J.Foster and H.T.Kung, "The Design of Special-Purpose VLSI Chips," *IEEE Comput.*, vol.13, no.1, pp.26-40, Jan.1980.

[6] B.Schutz and A.Klindworth, "A VLSI-Chip-Set for a Hardware-Accelerator for the Simplex-Method," *Proc. of Fifth Annual IEEE International ASIC Conference and Exhibit,* pp.553-556, 1992.

[7] C.A.Meal and L.A.Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.

[8] H.T.Kung, "Why Systolic Architectures," *IEEE Comput.*, vol.15, no.1, Jan.1982.

[9] W.-M.Lin and V.K.P.Kumar, "A Note on the Linear Transformation Method for Systolic Array Design," *IEEE Comput.*, vol.39, no.3, Mar.1990.

[10] J.-P.Sheu and C.-Y.Chang, "Synthesizing Nested Loop Algorithms Using Nonlinear Transformation Method," *IEEE Comput.*, vol.2, no.3, July 1991.

[11] M.T.O'Keefe, J.A.B.Fortes and B.W.Wah, "On the Relationship Between Two Systolic Array Design Methodologies," *IEEE Comput.*, vol.41, no.12, Dec.1992.

[12] V.Chvatal, *Linear Programming,* W.H.Freeman and company, pp.425-454.

[13] S.-C.Fang and S.Puthenpura, *Linear Optimization and Extensions,* Prentice-Hall, 1993.

[14] D.J.Evans and G.M.Megson, "A Systolic Simplex Algorithm," *Intern. J. Computer Math.*, vol.38, pp.1-30, 1991.

[15] D.I.Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. Comput.*, vol.C-31, pp.1121-1126, Nov.1982.

[16] D.I.Moldovan and J.A.B.Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. Comput.*, vol.C-35,pp.1-12, Jan.1986.

[17] G.J.Li and B.W.Wah, "The Design of Optimal Systolic Arrays," *IEEE Trans. Comput.*, vol.C-34, no.1, pp.66-77, Jan.1985.

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

[19] M.O.Esonu, A.J.Al-Khalili, S.Hariri and D.Al-Khalili, "Systolic Arrays:How to choose them," *IEE Proc. E, Comput. & Digital Tech.*, vol.139, no.3, pp.179-188, May 1992.

[20] C.N.Zhang, J.H.Weston and Y.-F.Yan, "Determining Objective Functions in Systolic Array Designs," *IEEE Trans. VLSI systems,* vol.2, no.3, pp.357-360, Sep.1994.

[21] M.O.Esonu, A.J.Al-Khalili, S.Hariri and D.Al-Khalili, "Fault-Tolerant Design Methodology for Systolic Array Architectures," *IEE Proc. E, Comput. & Digital Tech.*, vol.141, no.1, pp.17-28, Jan.1994.

Work

- "Hardware for Linear Programming Using Systolic Array (in Japanese)"
  Shinhaeng Lee and Hirotomo Aso
  IEICE Fall Conference, D-67, 1996.

- "Special-Purpose Hardware for Linear Programming using Systolic Arrays (in English)"
  Shinhaeng Lee and Hirotomo Aso
  IEICE computer system technical report, CPSY96-71, Oct.1996.

- "Special Purpose Hardware for Large Scale Linear Programming (in Japanese)"
  Shinhaeng Lee and Hirotomo Aso
  IEICE Spring Conference, D-6-25, 1997.

- "Special-Purpose Hardware Architecture for Large Scale Linear Programming (in English)"
  Shinhaeng Lee, Shin'ichiro Omachi and Hirotomo Aso
  IEICE Trans. Information and Systems, vol.E80-D, no. 9, Sep. 1997.
  (to be published)