A cellular computer to implement the Kalman filter algorithm
by Lynn Elliot Cannon

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY in Electrical Engineering
Montana State University


Abstract:
The subject of this thesis is the development of the design for a specially-organized, general-purpose computer which performs matrix operations efficiently.

The content of the thesis is summarized as follows: First, a review of the relevant work which has been done with microcellular and macrocellular techniques is made, Second, the discrete Kalman filter is described as an example of the type of problem for which this computer is efficient. Third, a detailed design for a cellular, array-structured computer is presented. Fourth, a computer program which simulates the cellular computer is described. Fifth, the recommendation is made that one cell and the associated control circuits be constructed to determine the feasibility of producing a hardware realization of the entire computer.

# A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM

by

LYNN ELLIOT CANNON

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Head, Major Department

Chairman, Examining Committee

Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

August, 1969

# ACKNOWLEDGMENT

The author wishes to thank Professor R. C. Minnick for his encouragement and helpful suggestions during the course of this graduate work and thesis research.

The financial support of the graduate work provided by the Electrical Engineering Department and by a National Science Foundation Traineeship is gratefully acknowledged, as is the thesis support furnished by the Office of Naval Research Contract No. N00014-67-C-0477.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABSTRACT

The subject of this thesis is the development of the design for a specially-organized, general-purpose computer which performs matrix operations efficiently.

The content of the thesis is summarized as follows: First, a review of the relevant work which has been done with microcellular and macrocellular techniques is made. Second, the discrete Kalman filter is described as an example of the type of problem for which this computer is efficient. Third, a detailed design for a cellular, array-structured computer is presented. Fourth, a computer program which simulates the cellular computer is described. Fifth, the recommendation is made that one cell and the associated control circuits be constructed to determine the feasibility of producing a hardware realization of the entire computer.

# Chapter 1

# INTRODUCTION AND REVIEW OF CELLULAR RESEARCH

## 1.1 Introduction

Designers of computers are often confronted with the development of a computer for a particular application such as vehicle navigation, signal processing or the like. Many of these applications require that computations be performed as rapidly as possible and that the computer be as small as possible. In order to meet the requirements for speed, cost and physical size, the designer will sometimes resort to the development of a special purpose computer. The disadvantages of this approach are the large initial design costs and the possible necessity of a complete redesign to account for a change in the computation algorithm. On the other hand, general purpose computers, while they do not have to be redesigned to account for algorithm changes, do have disadvantages. Those general purpose computers which are fast enough for some applications are usually not small enough; and those which are small enough are usually not fast enough or are too expensive. These arguments suggest that some thought should be given to the design of general purpose computers which are capable of performing certain types of operations efficiently. Although efficiency may be gained by using faster components, this method is usually expensive and leads to other complications, such as poor reliability. Another method of obtaining greater efficiency for certain operations is to build a computer which takes advantage of the structure inherent in these operations. This type of machine can be called a specially organized, general purpose computer. A computer of this type could be highly efficient for some

types of operations as is a special purpose computer, but would be easy to adapt to changes in computation algorithms as is a general purpose computer.

One class of problems whose inherent structure may be incorporated into the design of a computer is that class which deals with operations on matrices and vectors. An example of problems of this type is the discrete Kalman filter.[3] For certain applications of the Kalman filter many operations must be performed on matrices as rapidly as possible. Thus, the basis for this research is to develop a design for a small specially organized, general purpose computer which is particularly efficient for performing matrix operations.

Incorporating matrix operations into the structure of a computer suggests the use of arrays of logic elements. Therefore, it is natural to consider some of the work which has been done with arrays of such elements.

Arrays of similar or identical logic circuits together with some interconnection structure constitute cellular arrays. If the cells have a low complexity, the arrays of these cells are referred to as microcellular. If not, they are referred to as macrocellular. Low-complexity cells contain no more than a few gates. In either case, the logic circuits, or cells as they are called, are usually arranged in some rectangular fashion and the interconnections between the cells usually forms a uniform pattern.

In the next section some of the work which has been done on micro-

cellular arrays will be discussed, and in the following section some of the pertinent work on macrocellular arrays will be considered.

## 1.2 Survey of Microcellular Work

One of the earliest arrays which used identical cells and a uniform interconnection was originated in 1961 by Brooking[4,8] at the Air Force Cambridge Research Laboratories (AFCRL). The cells in this array are eight-input NOR gates with each input of a cell being connected to the output of one of the eight neighboring cells in the array. The array is known as an eight-neighbor array. Each cell in the array has inputs from outside the array which are used to provide an electrical disconnect for any of the eight inputs. Several variations of the eight-neighbor arrays were produced at AFCRL by Giusti.

Methods for the logical design of eight-neighbor cellular arrays of NAND cells were developed by Spandorfer and Murphy[21] in 1963. These workers also developed methods of avoiding faulty cells in an array.

Much work has been done on arrays whose cells may perform any one of several different functions. In 1962, Maitra[11] considered one-dimensional arrays in which each cell can be any of the 16 functions of two variables. Such an array is called a Maitra cascade. Additional work on Maitra cascades eventually led to Minnick's cutpoint array[16] in 1964. This array contains cells which can produce any of eight combinational switching functions or an S-R flip-flop. Four parameters

specify the function to be produced for each cell. In a later array by Minnick[15], called the cobweb array, cell parameters specify the interconnection as well as the cell function. Techniques have been developed for the logical design of both cutpoint and cobweb arrays[14-16].

For a much more complete survey of the development of microcellular work the reader is referred to reference (10).

## 1.3 Survey of Macrocellular Work

Arrays of large or complex cells are known as macrocellular arrays. Most computers which have been designed with an array structure use some form of a macrocellular array. In this section, some of the work which has been done in the design of array structured computers is described.

### 1.3.1 Unger's Machine

In 1958 Unger[23] described a stored-program computer which was specially organized for pattern detection. The computer consists of a master control and a rectangular array of logical modules as shown in Figure 1.1. The master control contains a clock, decoding circuits and a random access memory. It accesses instructions from memory, decodes them and issues commands which go simultaneously to all of the modules.

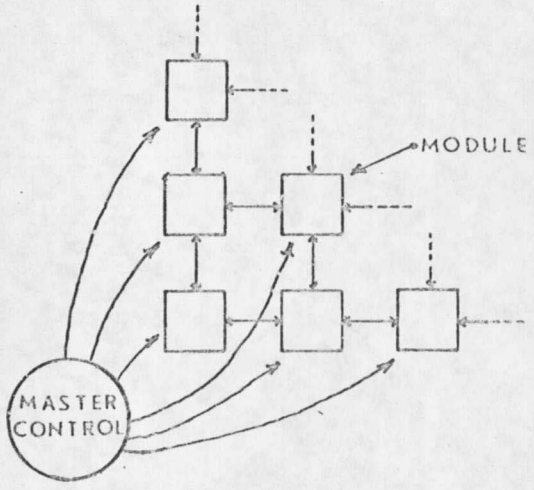



Figure 1.1. Unger's Machine

Each module (or macro-cell in the nomenclature of Section 1.1) contains a one-bit accumulator, six one-bit words of random access memory and associated logic. Each has inputs from the master control and from the accumulators of its four immediate neighbors. Each accumulator also has an input from outside the machine.

A logical OR circuit with inputs from each of the accumulators informs the master control if all of the accumulators contain zeros. This information is used for a transfer-on-zero command which is the only decision-dependent command used.

Other commands for Unger's machine are shown in Table 1.1. These commands are described in detail in reference (23).

Table 1.1. Command Table for Unger's Machine

| Command | Meaning |
|---|---|
| tr x | Transfer to instruction x. |
| trz x | Transfer on zero to instruction x. |
| in | Logical invert (NOT). |
| add | Logical add (OR). |
| mpy | Logical multiply (AND). |
| adm | Logical add to memory (OR). |
| mpm | Logical multiply to memory (AND). |
| st | Store. |
| wr | Write (actually a "LOAD" command). |
| sL(sR,sU,sD) | Shift left (right, up, down). |
| Add ref | Add reference. |
| Link | Link. |
| exp | Expand. |
| sA | Shift around. |

Inputs to the array can be made in parallel by associating an input device such as a photocell with each module or by using the shift around command.

A typical program for Unger's machine might locate lower-left corners of a pattern. Lower-left corners are located by placing a 1 in all cells (here a cell refers to the accumulator of a module)

which satisfy the conditions: (a) there is a 1 in the cell, (b) there are 1's in the cells immediately above and immediately to the right of the cell, and (c) there are 0's in the cells immediately to the left, immediately below and diagonally adjacent to the lower left. The shaded cell of Figure 1.2 satisfies these conditions.

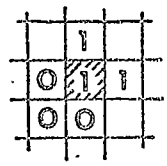



Figure 1.2. A Lower-Left Corner.

In order to illustrate Unger's machine, a program to find lower-left corners is given below. The first command of this program loads the original pattern into the a-register of all cells. In the following steps, the operands R and U refer to the a-register of adjacent cells to the right and above the cell in question, respectively. In Figure 1.3 an example set of data are shown in order to make the execution of the program more clear. From an examination of Figure 1.3(a) it is clear that cells (5,2) and (4,5) are the only two lower-left corners.

1) st a — Result in a-register is Figure 1.3(a).

2) mpm R,U,a — Result in a-register is Figure 1.3(b).

3) in — Result in accumulator is Figure 1.3(c).

4) mpy R,U — Result in accumulator is Figure 1.3(d).

5) sU — Result is not shown.

6) sR — Result in accumulator is Figure 1.3(e).

7) mpy a — Result in accumulator is Figure 1.3(f).

It should be noted that the result, shown in Figure 1.3(f), has 1's at the locations which are lower-left corners in the original pattern of Figure 1.3(a).

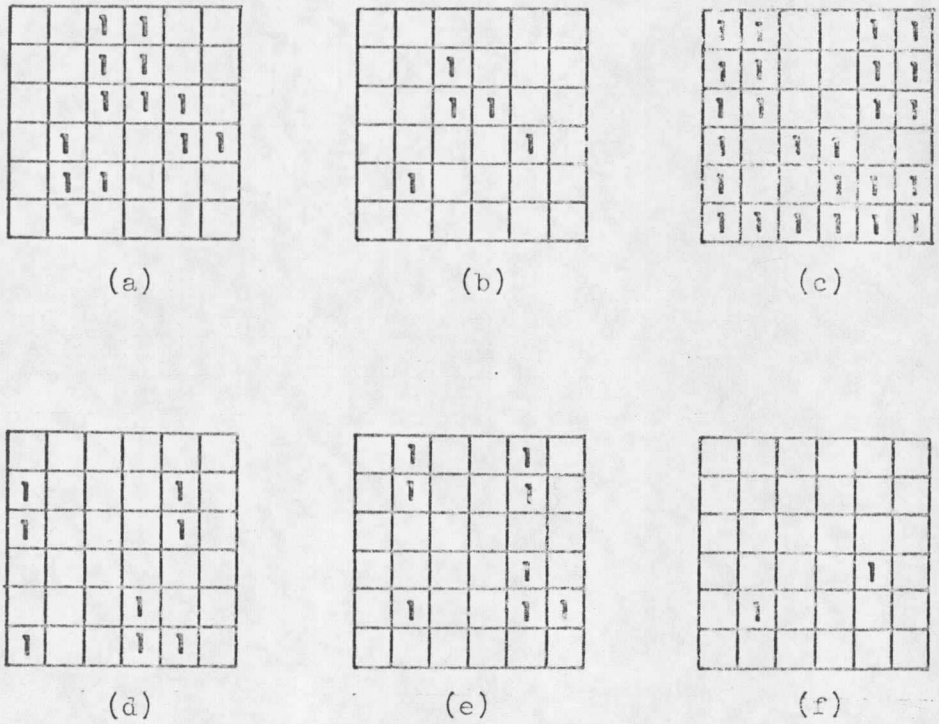



Figure 1.3. Lower-Left Corner Program Results.

Several programs have been written for pattern detection and recognition using the Unger machine. Some of these are described in reference (24).

### 1.3.2 Holland's Machine

In 1959 Holland[7] described a machine organization to provide a basis for investigations in computability and the theory of automata.

The machine is a two-dimensional array of modules. Each module contains a storage register, some associated logic and some auxiliary registers. At a given time a module may be active or inactive. If the module is active it treats the content of its storage register as an instruction and executes the instruction. After a module has executed its instruction it passes its active status on to its successor, which may be any of its four nearest neighbors in the array. Thus, sequences of instructions are arranged spatially throughout the array of modules, with an arbitrary number of sequences being executed at any given time.

Each cycle of the machine consists of three phases. In the first phase, module storage registers may be set to values provided by an external source. In the second phase, active modules determine the locations of their operands by causing paths to be gated open to them. In the third phase, the instructions in the storage registers of all active modules are executed.

In general, the Holland machine requires a large amount of hardware to accomplish reasonable computing tasks. It is extremely difficult to program the machine so that more than a very small percentage of the

modules are active at one time. In light of these problems a modified version of the Holland machine was described by Comfort[2] in 1963.

### 1.3.3 Comfort's Machine

Comfort's version had basically the same organization as the Holland machine. The machine described by Comfort is a fixed-size rectangular array of modules. At one side of the array are a set of arithmetic units called A-boxes. The array modules provide storage for data and instructions, communication with and between the A-boxes, and sequencing of instructions. The A-boxes do all arithmetic and logical computations. There is no central control unit; each module executes its own instruction when it is placed in the active state. The execution of a sequence of instructions causes the activation and deactivation of successive modules as in the Holland machine.

Comfort concluded that compared with Holland's machine, his organization results in:

1. Programmability improved by several orders of magnitude.
2. Reduction in amount of hardware by a factor of five.
3. Hardware utilization improved by a factor of three.
4. System performance degraded somewhat. (Because only one program sequence per A-box can be executed at one time.)

### 1.3.4 Gonzales' Machine

Another modified Holland machine was proposed by Gonzales[5]. It consists of three layers of modules with each layer being a

rectangular array similar to Holland's. A program layer stores data and instructions, a control layer decodes instructions and a computing layer performs arithmetical, logical and geometrical operations.

The programming is basically the same as the Holland machine with each instruction specifying the module containing the next instruction. While the computing layer is executing an instruction, the control and program layers can be working on the next two instructions, respectively.

As in the Holland machine, programming is difficult and hardware is not efficiently utilized.

### 1.3.5 The SOLOMON Computer

The SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) is a parallel computer which was introduced by Slotnick, Borck and McReynolds[19] in 1962.

The SOLOMON consists basically of an array of processing elements and a central control as shown in Figure 1.4.

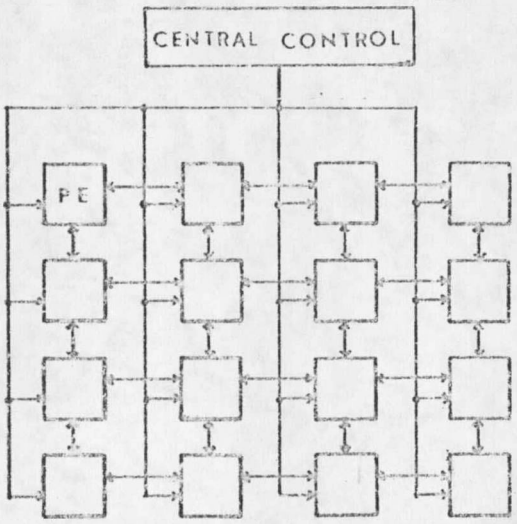



Figure 1.4. The SOLOMON Computer

Each of the processing elements has 4096 bits of core storage and the capabilities required to perform serial-by-bit arithmetic and logic. Words in the processing elements may be varied from one to 128 bits. A processing element may transfer data serially to and from its four nearest neighbors in the array.

A later version of the SOLOMON[17] had a faster clock rate, faster multiply logic and a fixed word length. The changes provided more computing capability for about the same cost.

1.3.6 The ILLIAC IV Computer

A direct descendent of the SOLOMON, the ILLIAC IV computer[1], consists of 256 processing elements arranged in four SOLOMON-type arrays of 64 processors each. Each processing element has 2048 words of 64-bit memory and extensive arithmetic capability. There is a common control unit which decodes instructions and generates control signals for all processing elements in the array.

The ILLIAC IV is supervised by a large general purpose computer which controls the execution of array programs, conducts input and output processes and performs independent data processing tasks.

The ILLIAC IV is currently under construction by the Burroughs Corporation. A large number of reports which describe various aspects of the ILLIAC IV have been issued by the University of Illinois where much of the design work is being done.

## 1.4 Relevance of Previous Work

The specially-organized computer to be described in this thesis draws both on the microcellular work and the macrocellular work which has been done.

Previous microcellular work is responsible for the concept of incorporating as much as possible of the structure of the problems, for which the computer is designed, into the array interconnection. The use of parameters to modify the interconnection structure is suggested by Minnick's cobweb array.

The contributions due to the macrocellular work are somewhat more obvious than those for the microcellular. Unger's machine suggests an array structure controlled by a central control unit. However, Unger's cells had limited arithmetic capability since his machine was intended for pattern detection. Holland introduced the idea of local control in the cells. While some local control seems advantageous, the use of purely local control has not proved to be very effective.

The basic structure of the proposed computer is similar to that of the SOLOMON with an array of processing cells under the control of a central control unit. Differences arise in the types of cells used, the interconnection structure for the array and the size of the system. The SOLOMON and its descendent the ILLIAC IV are large-scale systems. The designers of the ILLIAC IV, for example, are now considering the use of one of the largest computers currently available (the CDC 6600) for their central control unit. Although the ideas presented for

the proposed computer may be extended to much larger systems, the computer is intended to be roughly the size of some of the smallest computers now available.

## 1.5 Organization of Remaining Chapters

Chapter 2 discusses the Kalman filter. This discussion is presented to acquaint the reader with the types of operations which a specially organized computer should be able to perform efficiently. No new material is presented on the Kalman filter because it is intended to serve only as an example. Also included in this chapter are algorithms for the multiplication and inversion of matrices within an array-structured computer.

Chapter 3 presents a detailed design of a specially organized, general-purpose computer which has an array structure so that it is capable of performing matrix operations efficiently.

A computer program is given in Chapter 4 which provides a detailed simulation of the computer described in Chapter 3.

The concluding Chapter 5 gives a comparison of the computer described in Chapter 3 to more conventional computers.

# Chapter 2

# THE KALMAN FILTER

## 2.1 The Estimation Problem

A uniformly sampled discrete model of a linear dynamic system without control or noise inputs is given by

$$x(k+1) = \phi(k+1,k)x(k) \tag{2.1}$$

where $x(k)$ is a state vector representing the state of the system at time $kT$, $T$ being the sampling period, $\phi(k+1,k)$ is the system transition matrix and $x(k+1)$ represents the state of the system at time $(k+1)T$.

Equation (2.1) represents the case of perfect predictability. The state $x(k+n)$ at time $(k+n)T$ may be determined exactly based only on the state vector $x(k)$ and the transition matrix $\phi(k+n,k)$. No information about states preceeding or following $x(k)$ is required.

If external additive noise is applied to the system, equation (2.1) must be modified to

$$x(k+1) = \phi(k+1,k)x(k) + \Gamma(k+1,k)w(k) \tag{2.2}$$

where $w(k)$ is a random vector sequence representing noise and $\Gamma(k+1,k)$ is a known matrix indicating how $w(k)$ affects each component of $x(k+1)$.

In most systems the state vector $x(k)$ is not directly observed. Instead, a measurement $z(k)$ which is related to $x(k)$ by the measuring system $H(k)$ is observed. The relation between $z(k)$ and $x(k)$ is described by

$$z(k) = H(k)x(k) \tag{2.3}$$

where $H(k)$ is a known matrix describing the operation of the measuring system.

Since no measuring system is free of noise, equation (2.3) must be modified to

$$z(k) = H(k)x(k) + v(k) \qquad (2.4)$$

where $v(k)$ is the noise vector sequence associated with the measurement of $x(k)$.

The problem is: Given the system and measurement models, (equations (2.2) and (2.4)), some statistical properties of the noise and the past history of measurements, what is the best estimate of the state at some given time? If some past state is to be estimated, the process is called smoothing. If the present state is being estimated, it is called filtering. Estimation of the future is called predicting.

## 2.2 The Discrete Kalman Filter

### 2.2.1 Introduction

The Kalman or related filters such as least squares, maximum likelihood of Bayesian estimators, are often used for state estimation of a dynamic system. Since all reduce to the Kalman form under assumptions of gaussianess of random sequences and first order Markovian properties, attention will be restricted to the Kalman filter. Also, since smoothing is not often done and the filtering or prediction problems involve an identical algorithm, that algorithm will be employed.

Very breifly, the state of a system is a vector quantity which encodes all of the system history that needs to be known for a particular purpose ( ?2 ).

In general, it cannot be determined or predicted exactly due to measurement noise, random inputs to the system or incomplete knowledge of the system parameters. To avoid the identification problem and its exposition, system parameters are assumed to be known exactly in this discussion.

Although the state cannot be precisely determined, it is possible to estimate it in some optimal fashion by the appropriate processing of available data. Thus, the problem is that of estimating the state of a system based on the available measurements and on knowledge about system parameters and noise statistics.

2.2.2 The System Model

The discrete system model is described by

$$x(k+1) = \phi(k+1,k)x(k) + \Gamma(k+1,k)w(k) \tag{2.5}$$

and

$$z(k) = H(k)x(k) + v(k) \tag{2.6}$$

where $x(k)$ is the n-dimensional state vector, $\phi(k+1,k)$ is the system transition matrix, $z(k)$ is the m-dimensional observation vector, $m \leq n$, $H(k)$ is a matrix describing the operation of the measuring process, and $v(k)$ and $w(k)$ are vector Gaussian noise sequences.

The means and covariances of v and w are

$$E[v(k)] = 0 \quad \text{for all } k \tag{2.7}$$

$$E[w(k)] = 0 \quad \text{for all } k \tag{2.8}$$

$$\text{cov}[w(k)] = E[w(k)w'(n)] = \delta(k,n)Q(k) \quad \text{for all } k,n \tag{2.9}$$

$$\text{cov}[v(k)] = E[v(k)v'(n)] = \delta(k,n)R(k)$$

$$\text{for all } k,n \qquad (2.10)$$

where $\delta(k,n)$ is the Kronecker delta.

It is usually assumed that no correlation exists between noise driving the plant and noise in the observation system. Should such correlation exist, a covariance matrix is given as

$$\text{cov}[v(k)w(n)] = E[v(k)w'(k)] = \delta(k,n)C(k)$$

$$\text{for all } k,n. \qquad (2.11)$$

Appropriate filter equations for non-zero $C(k)$ can be found in reference (3). For the discussion and example to follow, no correlation is assumed and $C(k) = 0$. Also $\phi(k,k-1)$, $\Gamma(k,k-1)$, $H(k)$, $Q(k)$, and $R(k)$ are known since the identification problem is not to be dealt with.

Suppose that a sequence of observations $z(0)$, $z(1)$, ...$z(k)$ is made. Based on these observations and a knowledge of the system and noise parameters, it is desired to make an estimate of the state vector $x(n)$. Notationally this is shown as $\hat{x}(n|k)$ and is read as "the optimal estimate of $x(n)$ given observations up through time $kT$". For filtering, $n=k$; for prediction, $n>k$, most usually $n=k+1$. The prediction problem for $n=k+1$ (which is computationally identical to the filtering problem) will be discussed.

The estimation error is defined as

$$\tilde{x}(k+1|k) = x(k+1) - \hat{x}(k+1|k) \qquad (2.12)$$

and the measure of performance is the mean squared error, which is the sum of variances in the case of an unbiased estimator such as the Kalman filter.

$$\overline{e^2} = E[\tilde{\hat{x}}'(k+1|k)\tilde{\hat{x}}(k+1|k)].$$

In terms of the covariance matrix of $\tilde{\hat{x}}$,

$$\Sigma(k+1|k) = E[\tilde{\hat{x}}(k+1|k)\tilde{\hat{x}}'(k+1|k)] \tag{2.14}$$

the mean squared error can be expressed as

$$\overline{e^2} = tr\Sigma(k+1|k). \tag{2.15}$$

2.2.3 The Solution

Kalman (3) has solved the problem posed in Section 2.2.2 by using the projection theorem from the theory of linear spaces. The equations describing the optimal filter are

$$\hat{x}(k+1|k) = \phi(k+1,k)\hat{x}(k|k) \tag{2.16}$$

$$\hat{x}(k|k) = \hat{x}(k|k-1) - G(k)[H(k)\hat{x}(k|k-1) - z(k)] \tag{2.17}$$

$$G(k) = \Sigma(k|k-1)H'(k)[H(k)\Sigma(k|k-1)H'(k) + R(k)]^{-1} \tag{2.18}$$

$$\Sigma(k|k-1) = \phi(k,k-1)\Sigma(k-1|k-1)\phi'(k,k-1) + \Gamma(k,k-1)Q(k-1)\Gamma'(k,k-1) \tag{2.19}$$

$$\Sigma(k|k) = [I - G(k)H(k)]\Sigma(k|k-1) \tag{2.20}$$

where G(k) is a matrix called the optimal filter gain matrix. The filtered estimate of x(k) is $\hat{x}(k|k)$ and the predicted estimate is $\hat{x}(k+1|k)$. It is worthwhile to interpret 2.17 as showing the filtered value to be the predicted value modified by the gain matrix times the difference between predicted and actual observation.

As indicated in the equations, $\hat{x}(k+1|k)$ is a processed version of $\hat{x}(k|k-1)$, updated by the most recent observation z(k). It is not necessary, therefore, to store the sequence of previous observations, z(0), z(1), ..., z(k-1). All relevant information contained in these

observations is contained in the vector $\hat{x}(k|k-1)$ and matrix $\Sigma(k|k-1)$. It is this recursive nature that makes the Kalman and related filters computationally attractive.

### 2.2.4 Computation

To start the computation, initial value $\hat{x}(0|0)$ for the state vector and $\Sigma(0|0)$ for the covariance matrix are assumed. These choices having been made, $\Sigma(1|0)$ is calculated according to equation (2.20) and $G(1)$ is calculated according to equation (2.19). The estimation $\hat{x}(1|0)$ is then determined using equations (2.17) and (2.18). $\Sigma(1|0)$ and $\hat{x}(1|0)$ are then used along with the next observation $z(1)$ to determine $\Sigma(2|1)$ and $\hat{x}(2|1)$, and so on.

The computations involve several matrix additions, subtractions and multiplications and the inversion of one matrix for each observation time. For this reason a machine which is used to perform the Kalman filter algorithm should be able to do these operations on matrices efficiently. The next section describes an algorithm for the multiplication of two matrices using an array-structured computer. An algorithm for inverting a matrix in an array-structured computer is given in the succeeding section.

## 2.3 Matrix Multiplication Algorithm

In this section an algorithm for the multiplication of two matrices using an array-structured computer is given.

It is desired to multiply two n x n matrices using the conventional definition of matrix multiplication. Assume, for the sake of demonstrating the multiplying scheme, that two 3 x 3 matrices A and B are defined by

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_4 & b_7 \\ b_2 & b_5 & b_8 \\ b_3 & b_6 & b_9 \end{bmatrix}$$

where the numbering scheme for the elements was chosen to provide a matrix product whose elements show some form of symmetry as will be seen later.

By the conventional definition of matrix product, if A is multiplied by B, the result, call it C, is given by

$$C = AxB = \begin{bmatrix} a_1b_1+a_2b_2+a_3b_3 & a_1b_4+a_2b_5+a_3b_6 & a_1b_7+a_2b_8+a_3b_9 \\ a_4b_1+a_5b_2+a_6b_3 & a_4b_4+a_5b_5+a_6b_6 & a_4b_7+a_5b_8+a_6b_9 \\ a_7b_1+a_8b_2+a_9b_3 & a_7b_4+a_8b_5+a_9b_6 & a_7b_7+a_8b_8+a_9b_9 \end{bmatrix}.$$

The symmetry of this product can be seen by comparing the $ij^{th}$ element with the $ji^{th}$ element and noticing that one is obtained from the other by inter-changing the subscripts on the a's and the b's.

The first steps in the multiplication scheme are to clear all registers and load the matrices A and B into the array. After the matrices are loaded into the array they are shifted according to the following rules.

A. 1. The first row of A is left alone.

2. The second row of A is shifted left one column.

3. The third row of A is shifted left two columns.

(Note, in general the $i^{th}$ row of A is shifted left i-1 columns for i = 1, ..., n).

B. 1. The first column of B is left alone.

2. The second column of B is shifted up one row.

3. The third column of B is shifted up two rows.

(Note, in general the $j^{th}$ column of B is shifted up j-1 rows for j = 1, ..., n)

Once the registers have been shifted the multiplication process can begin. The contents of the A register of a cell are multiplied by the contents of the B register of the cell and the result is added to the contents of the C register and stored there. This operation will be called element multiplication. The contents of the A and B registers remain unchanged. Now the contents of all A registers are shifted right one cell. In a similar fashion the contents of all B registers are shifted down one cell. The cycle is now repeated. A is element multiplied by B and the result is added to C. A is shifted right one cell, B is shifted down one cell. The cycle is repeated n times in general (three in the example).

The operation can be seen more clearly as the 3 x 3 example is gone through. The registers are first loaded with A and B

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_4 & b_7 \\ b_2 & b_5 & b_8 \\ b_3 & b_6 & b_9 \end{bmatrix}.$$

A and B are shifted using the rules given. Note that A, B and C represent the contents of the A, B, and C registers respectively.

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_5 & a_6 & a_4 \\ a_9 & a_7 & a_8 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_5 & b_9 \\ b_2 & b_6 & b_7 \\ b_3 & b_4 & b_8 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

A is element multiplied by B and the result is added to C.

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_5 & a_6 & a_4 \\ a_9 & a_7 & a_8 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_5 & b_9 \\ b_2 & b_6 & b_7 \\ b_3 & b_4 & b_8 \end{bmatrix}$$

$$C = \begin{bmatrix} a_1b_1 & a_2b_5 & a_3b_9 \\ a_5b_2 & a_6b_6 & a_4b_7 \\ a_9b_3 & a_7b_4 & a_8b_8 \end{bmatrix}$$

A is shifted right and B is shifted down one cell.

$$A = \begin{bmatrix} a_3 & a_1 & a_2 \\ a_4 & a_5 & a_6 \\ a_8 & a_9 & a_7 \end{bmatrix} \qquad B = \begin{bmatrix} b_3 & b_4 & b_8 \\ b_1 & b_5 & b_9 \\ b_2 & b_6 & b_7 \end{bmatrix}$$

$$C = \begin{bmatrix} a_1b_1 & a_2b_5 & a_3b_9 \\ a_5b_2 & a_6b_6 & a_4b_7 \\ a_9b_3 & a_7b_4 & a_8b_8 \end{bmatrix}$$

A is element multiplied by B and the result is added to C.

$$A = \begin{bmatrix} a_3 & a_1 & a_2 \\ a_4 & a_5 & a_6 \\ a_8 & a_9 & a_7 \end{bmatrix} \qquad B = \begin{bmatrix} b_3 & b_4 & b_8 \\ b_1 & b_5 & b_9 \\ b_2 & b_6 & b_7 \end{bmatrix}$$

$$C = \begin{bmatrix} a_1b_1+a_3b_3 & a_2b_5+a_1b_4 & a_3b_9+a_2b_8 \\ a_5b_2+a_4b_1 & a_6b_6+a_5b_5 & a_4b_7+a_6b_9 \\ a_9b_3+a_8b_2 & a_7b_4+a_9b_6 & a_8b_8+a_7b_7 \end{bmatrix}$$

A is shifted right and B is shifted down one cell.

$$A = \begin{bmatrix} a_2 & a_3 & a_1 \\ a_6 & a_4 & a_5 \\ a_7 & a_8 & a_9 \end{bmatrix} \qquad B = \begin{bmatrix} b_2 & b_6 & b_7 \\ b_3 & b_4 & b_8 \\ b_1 & b_5 & b_9 \end{bmatrix}$$

$$C = \begin{bmatrix} a_1b_1+a_3b_3 & a_2b_5+a_1b_4 & a_3b_9+a_2b_8 \\ a_5b_2+a_4b_1 & a_6b_6+a_5b_5 & a_4b_7+a_6b_9 \\ a_9b_3+a_8b_2 & a_7b_4+a_9b_6 & a_8b_8+a_7b_7 \end{bmatrix}$$

A is element multiplied by B and the result is added to C.

$$A = \begin{bmatrix} a_2 & a_3 & a_1 \\ a_6 & a_4 & a_5 \\ a_7 & a_8 & a_9 \end{bmatrix} \qquad B = \begin{bmatrix} b_2 & b_6 & b_7 \\ b_3 & b_4 & b_8 \\ b_1 & b_5 & b_9 \end{bmatrix}$$

$$C = \begin{bmatrix} a_1b_1+a_3b_3+a_2b_2 & a_2b_5+a_1b_4+a_3b_6 & a_3b_9+a_2b_8+a_1b_7 \\ a_5b_2+a_4b_1+a_6b_3 & a_6b_6+a_5b_5+a_4b_4 & a_4b_7+a_6b_9+a_5b_8 \\ a_9b_3+a_8b_2+a_7b_1 & a_7b_4+a_9b_6+a_8b_5 & a_8b_8+a_7b_7+a_9b_9 \end{bmatrix}$$

C now contains the matrix product of the original A and B matrices.

## 2.4 Matrix Inversion Algorithm

In this section an algorithm is presented for obtaining the inverse of a matrix. The algorithm is Gauss's algorithm modified to make efficient use of an array processor. The modifications consist of performing operations on all of the elements of a row, column or matrix simultaneously wherever possible.

The algorithm is presented in the form of an APL program. The program assumes a machine with two memory registers (M), a right-shifting matrix register (A) with an augmented column vector register (R), a down-shifting matrix register (B) with an augmented row vector register (C), an adder-subtractor with three matrix registers (AA, AB and AC) and a multiplier with three matrix registers (MA, MB and MC). In the above context a "matrix register" means an array of registers used to store a matrix. The "vector register" means a string of registers used to store a vector.

The program, which is shown in Figure 2.1, is intended to demonstrate the algorithm, not to produce an optimum realization of it. A numerical example which demonstrates the algorithm follows.

```
      ∇ INVERT
[  1] N←(⍴MATRIX)[1]
[  2] C←N⍴0
[  3] M←(2,N,N)⍴0
[  4] M[1;;]←MATRIX
[  5] K←1
[  6] LOOP:A←M[1;;]
[  7] R←N⍵1
[  8] T←(0 1+⍴A)⍴0
[  9] T[;1]←R
[ 10] T[;1+⍳N]←A
[ 11] A←T[;⍳N]
[ 12] R←T[;N+1]
[ 13] R[N]←1÷R[N]
[ 14] M[1;;]←A
[ 15] A←R∘.×N⍴1
[ 16] M[2;;]←A
[ 17] MA←M[1;;]
[ 18] MB←M[2;;]
[ 19] MC←MA×MB
[ 20] M[1;N;]←MC[N;]
[ 21] B←M[1;;]
[ 22] T←(1 0+⍴B)⍴0
[ 23] T[1;]←C
[ 24] T[1+⍳N;]←B
[ 25] B←T[⍳N;]
[ 26] C←T[N+1;]
[ 27] B←(N,N)⍴C
[ 28] MA←M[2;;]
[ 29] M[2;;]←B
[ 30] MB←M[2;;]
[ 31] MC←MA×MB
[ 32] M[2;;]←MC
[ 33] AA←M[1;;]
[ 34] AB←M[2;;]
[ 35] AC←AA-AB
[ 36] M[1;⍳N-1;]←AC[⍳N-1;]
[ 37] B←M[1;¯1↑⍳N;]
[ 38] M[1;;]←B
[ 39] K←K+1
[ 40] →(K≤N)/LOOP
[ 41] INVERSE←M[1;;]
      ∇
```

Figure 2.1. Parallel Matrix Inversion Algorithm

As an example of obtaining the inverse of a matrix consider Gauss's algorithm modified for the array structure. A 3x3 array is used for brevity.

Find the inverse of the matrix

$$\begin{bmatrix} 3.0 & 0.5 & 1.0 \\ 0.5 & 1.0 & 0.5 \\ 1.0 & 0.5 & 0.5 \end{bmatrix}.$$

Load the matrix into the routing A register and set the row buffers as indicated giving

$$\left[\begin{array}{c|ccc} 0.0 & 3.0 & 0.5 & 1.0 \\ 0.0 & 0.5 & 1.0 & 0.5 \\ 1.0 & 1.0 & 0.5 & 0.5 \end{array}\right].$$

Rotate the array, including the row buffers, one cell to the right to get

$$\left[\begin{array}{c|ccc} 1.0 & 0.0 & 3.0 & 0.5 \\ 0.5 & 0.0 & 0.5 & 1.0 \\ 0.5 & 1.0 & 1.0 & 0.5 \end{array}\right]$$

Invert the last-row element of the row buffers giving

$$\left[\begin{array}{c|ccc} 1.0 & 0.0 & 3.0 & 0.5 \\ 0.5 & 0.0 & 0.5 & 1.0 \\ 2.0 & 1.0 & 1.0 & 0.5 \end{array}\right].$$

Store the routing A register in memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 3.0 & 0.5 \\ 0.0 & 0.5 & 1.0 \\ 1.0 & 1.0 & 0.5 \end{bmatrix} .$$

Broadcast (by routing) the row buffers across the routing A register to get

$$\left[\begin{array}{c|ccc} 1.0 & 1.0 & 1.0 & 1.0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 2.0 & 2.0 & 2.0 & 2.0 \end{array}\right] .$$

Store the routing A register in memory location 2. Location 2 now contains

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 0.5 & 0.5 & 0.5 \\ 2.0 & 2.0 & 2.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the multiply A register and the contents of memory location 2 into the multiply B register. Multiply, and store the last row only back into memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 3.0 & 0.5 \\ 0.0 & 0.5 & 1.0 \\ 2.0 & 2.0 & 1.0 \end{bmatrix}$$

Load the contents of memory location 1 into the routing B register. Rotate the array, including the column buffers, down one

cell to place the bottom row in the column buffers. Then broadcast (by routing) the column buffers down the routing B register to get

$$\begin{bmatrix} 2.0 & 2.0 & 1.0 \\ 2.0 & 2.0 & 1.0 \\ 2.0 & 2.0 & 1.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply A register. Store the routing B register in memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & 2.0 & 1.0 \\ 2.0 & 2.0 & 1.0 \\ 2.0 & 2.0 & 1.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply B register. Multiply and store the result back in memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & 2.0 & 1.0 \\ 1.0 & 1.0 & 0.5 \\ 4.0 & 4.0 & 4.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the adder A register. Load the contents of memory location 2 into the adder B register. Subtract, and store all but the last row of the result back in memory location 1. Location 1 now contains

$$\begin{bmatrix} -2.0 & 1.0 & -0.5 \\ -1.0 & -0.5 & 0.5 \\ 2.0 & 2.0 & 1.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing B register. Rotate the array down one cell giving

$$\begin{bmatrix} 2.0 & 2.0 & 1.0 \\ -2.0 & 1.0 & -0.5 \\ -1.0 & -0.5 & 0.5 \end{bmatrix} .$$

Store the routing B register back into memory location 1. Location 1 now contains

$$\begin{bmatrix} 2.0 & 2.0 & 1.0 \\ -2.0 & 1.0 & -0.5 \\ -1.0 & -0.5 & 0.5 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing A register and set the row buffers as indicated giving

$$\left[\begin{array}{c|ccc} 0.0 & 2.0 & 2.0 & 1.0 \\ 0.0 & -2.0 & 1.0 & -0.5 \\ 1.0 & -1.0 & -0.5 & 0.5 \end{array}\right] .$$

Rotate the array, including the row buffers, one cell to the right to get

$$\left[\begin{array}{c|ccc} 1.0 & 0.0 & 2.0 & 2.0 \\ -0.5 & 0.0 & -2.0 & 1.0 \\ 0.5 & 1.0 & -1.0 & -0.5 \end{array}\right] .$$

Invert the last-row element of the row buffers giving

$$\left[\begin{array}{c|ccc} 1.0 & 0.0 & 2.0 & 2.0 \\ -0.5 & 0.0 & -2.0 & 1.0 \\ 2.0 & 1.0 & -1.0 & -0.5 \end{array}\right] .$$

Store the routing A register in memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 2.0 & 2.0 \\ 0.0 & -2.0 & 1.0 \\ 1.0 & -1.0 & -0.5 \end{bmatrix} .$$

Broadcast (by routing) the row buffers across the routing A register to get

$$\left[\begin{array}{c|ccc} 1.0 & 1.0 & 1.0 & 1.0 \\ -0.5 & -0.5 & -0.5 & -0.5 \\ 2.0 & 2.0 & 2.0 & 2.0 \end{array}\right] .$$

Store the routing A register in memory location 2. Location 2 now contains

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 \\ -0.5 & -0.5 & -0.5 \\ 2.0 & 2.0 & 2.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the multiply A register and the contents of memory location 2 into the multiply B register. Multiply, and store the last row only back into memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 2.0 & 2.0 \\ 0.0 & -2.0 & 1.0 \\ 2.0 & -2.0 & -1.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing B register. Rotate the array, including the column buffers, down one cell to place the bottom row in the column buffers. Then broadcast (by routing) the column buffers down the routing B register to get

$$\begin{bmatrix} 2.0 & -2.0 & -1.0 \\ 2.0 & -2.0 & -1.0 \\ 2.0 & -2.0 & -1.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply A register. Store the routing B register in memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & -2.0 & -1.0 \\ 2.0 & -2.0 & -1.0 \\ 2.0 & -2.0 & -1.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply B register. Multiply and store the result back in memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & -2.0 & -1.0 \\ -1.0 & 1.0 & 0.5 \\ 4.0 & -4.0 & -2.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the adder A register. Load the contents of memory location 2 into the adder B register. Subtract, and store all but the last row of the result back in memory location 1. Location 1 now contains

$$\begin{bmatrix} -2.0 & 4.0 & 3.0 \\ 1.0 & -3.0 & 0.5 \\ 2.0 & -2.0 & -1.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing B register. Rotate the array down one cell giving

$$\begin{bmatrix} 2.0 & -2.0 & -1.0 \\ -2.0 & 4.0 & 3.0 \\ 1.0 & -3.0 & 0.5 \end{bmatrix} .$$

Store the routing B register back into memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & -2.0 & -1.0 \\ -2.0 & 4.0 & 3.0 \\ 1.0 & -3.0 & 0.5 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing A register and set the row buffers as indicated giving

$$\left[\begin{array}{c|ccc} 0.0 & 2.0 & -2.0 & -1.0 \\ 0.0 & -2.0 & 4.0 & 3.0 \\ 1.0 & 1.0 & -3.0 & 0.5 \end{array}\right] .$$

Rotate the array, including the row buffers, one cell to the right to get

$$\left[\begin{array}{c|ccc} -1.0 & 0.0 & 2.0 & -2.0 \\ 3.0 & 0.0 & -2.0 & 4.0 \\ 0.5 & 1.0 & 1.0 & -3.0 \end{array}\right] .$$

Invert the last-row element of the row buffers giving

$$\left[\begin{array}{c|ccc} -1.0 & 0.0 & 2.0 & -2.0 \\ 3.0 & 0.0 & -2.0 & 4.0 \\ 2.0 & 1.0 & 1.0 & -3.0 \end{array}\right] .$$

Store the routing A register in memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 2.0 & -2.0 \\ 0.0 & -2.0 & 4.0 \\ 1.0 & 1.0 & -3.0 \end{bmatrix} .$$

Broadcast (by routing) the row buffers across the routing A register to get

$$\left[\begin{array}{c|ccc} -1.0 & -1.0 & -1.0 & -1.0 \\ 3.0 & 3.0 & 3.0 & 3.0 \\ 2.0 & 2.0 & 2.0 & 2.0 \end{array}\right] .$$

Store the routing A register in memory location 2. Location 2 now contains

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 \\ 3.0 & 3.0 & 3.0 \\ 2.0 & 2.0 & 2.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the multiply A register and the contents of memory location 2 into the multiply B register. Multiply, and store the last row only back into memory location 1. Location 1 now contains

$$\begin{bmatrix} 0.0 & 2.0 & -2.0 \\ 0.0 & -2.0 & 4.0 \\ 2.0 & 2.0 & -6.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing B register. Rotate the array, including the column buffers, down one cell to place the bottom row in the column buffers. Then broadcast (by routing) the column buffers down the routing B register to get

$$\begin{bmatrix} 2.0 & 2.0 & -6.0 \\ 2.0 & 2.0 & -6.0 \\ 2.0 & 2.0 & -6.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply A register. Store the routing B register in memory location 2. Location 2 now contains

$$\begin{bmatrix} 2.0 & 2.0 & -6.0 \\ 2.0 & 2.0 & -6.0 \\ 2.0 & 2.0 & -6.0 \end{bmatrix} .$$

Load the contents of memory location 2 into the multiply B register. Multiply and store the result back in memory location 2. Location 2 now contains

$$\begin{bmatrix} -2.0 & -2.0 & 6.0 \\ 6.0 & 6.0 & -18.0 \\ 4.0 & 4.0 & -12.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the adder A register. Load the contents of memory location 2 into the adder B register.

Subtract, and store all but the last row of the result back in memory location 1. Location 1 now contains

$$\begin{bmatrix} 2.0 & 4.0 & -8.0 \\ -6.0 & -8.0 & 22.0 \\ 2.0 & 2.0 & -6.0 \end{bmatrix} .$$

Load the contents of memory location 1 into the routing B register. Rotate the array down one cell giving

$$\begin{bmatrix} 2.0 & 2.0 & -6.0 \\ 2.0 & 4.0 & -8.0 \\ -6.0 & -8.0 & 22.0 \end{bmatrix} .$$

The routing B register now contains the inverse of the original matrix.

## 2.5 Kalman Filter Example

To provide an example of a use of the Kalman filter algorithm, a problem which has been solved by Kolb and Hollister [9], using the algorithm is considered.

The problem concerns the estimation of the motion of a target in a plane, given only bearing measurements corrupted by zero-mean Gaussian noise. The observer is assumed to be moving in the samp plane as the target. Sequential observations of the bearing of the target from the observer are made and the problem is to estimate the position and velocity of the target. The observations are assumed to be disturbed by zero-mean Gaussian distributed errors. While the target velocity is assumed constant, the Kalman theory allows the system being estimated to be excited by a ramdon sequence with known mean and variance. Therefore, the filter is able to tolerate some maneuvering of the target.

The target motion relative to the observer is approximated by

$$x(k+1) = \phi x(k) + \Gamma[w(k) - u(k)], \qquad (2.21)$$

where $\phi$ is the state transition matrix, $w(k)$ is a vector of random disturbances with covariance matrix $Q$, $\Gamma$ is the distribution matrix and $u(k)$ is a vector of deterministic accelerations of the observer.

An observation $z(k)$ is determined by

$$z(k) = H(k)x(k) + v(k) \qquad (2.22)$$

where H(k) describes the measurement system and v(k) represents the measurement noise and has covariance matrix R.

The solutions given in ( 9 ) can be written as

$$\hat{x}(k|k-1) = x^{*}(k-1|k-1) - \Gamma u(k) \qquad (2.23)$$

$$\Sigma(k|k-1) = \phi\Sigma(k-1|k-1)\phi' + \Gamma Q\Gamma' \qquad (2.24)$$

$$G(k) = \Sigma(k|k-1)H'(k)[H(k)\Sigma(k|k-1)H'(k) + R]^{-1} \qquad (2.25)$$

$$\Sigma(k|k) = [I - G(k)H(k)]\Sigma(k|k-1) \qquad (2.26)$$

$$\hat{z}(k|k) = H(k)\hat{x}(k|k-1) \qquad (2.27)$$

$$x^{*}(k|k) = G(k)[z(k) - \hat{z}(k|k-1)] + \hat{x}(k|k-1) \qquad (2.28)$$

where $\hat{x}(k|k-1)$ is the estimate of the state x(k) based on k-1 observations, $x^{*}(k|k)$ is the estimate of x(k) based on k observations, $\Sigma(k|k-1)$ is the error covariance matrix for $\hat{x}(k|k-1)$, $\Sigma(k|k)$ is the error covariance matrix for $x^{*}(k|k)$, and G(k) is the Kalman gain matrix. The identity matrix is represented by I.

A FORTRAN IVH computer program was written to perform the Kalman filter algorithm as used in this problem. A listing of this program is given in Appendix A and a flow chart describing the program is shown in Figure 2.2. Table 2.1 shows the correspondence between the identifiers used in the program and those used in Equations 2.21 through 2.28.

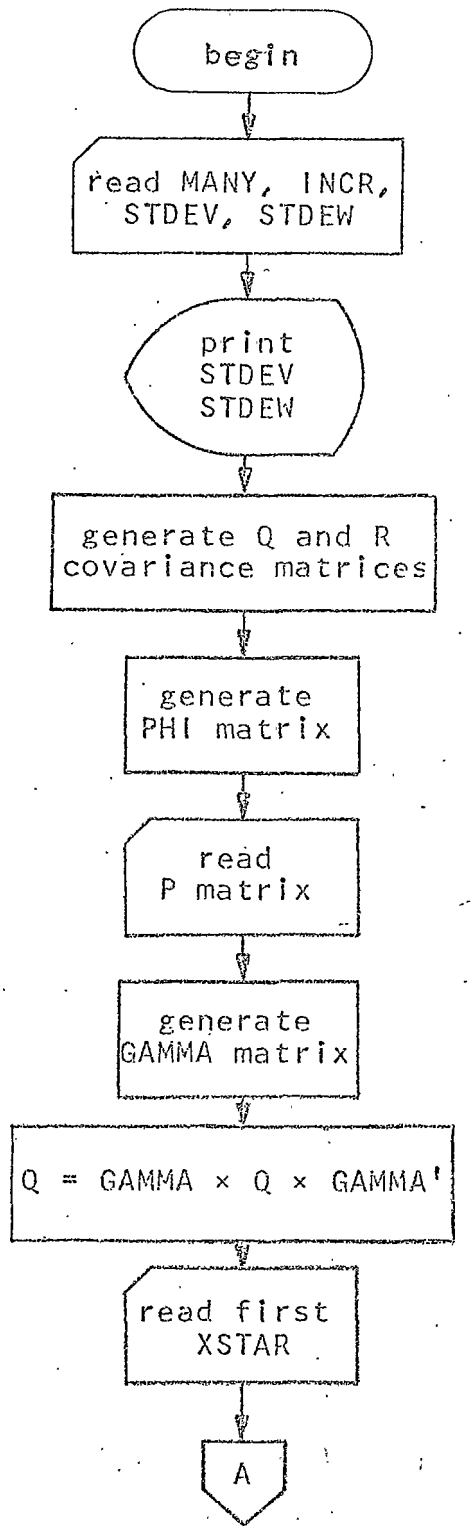



Figure 2.2. Flow Chart for Example Kalman Filter Program

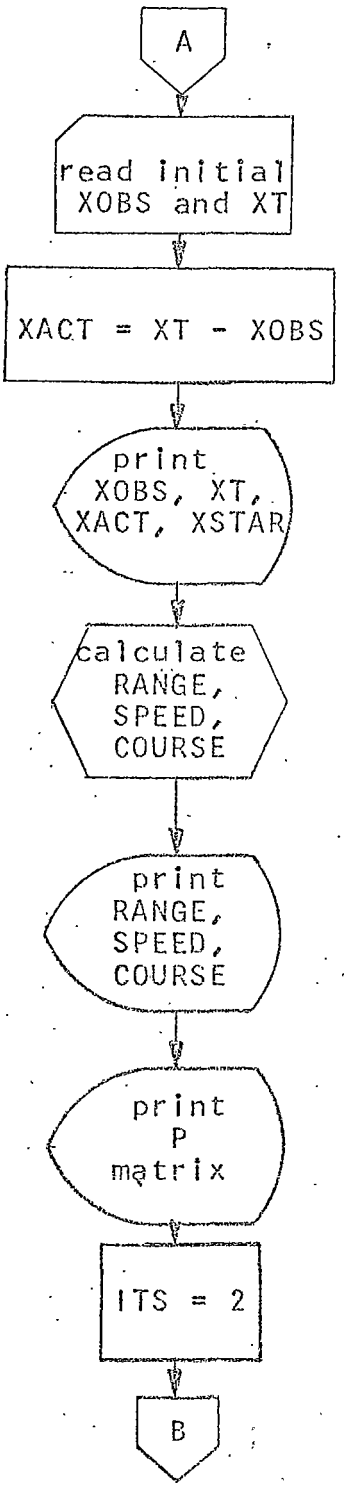



Figure 2.2. (Continued)

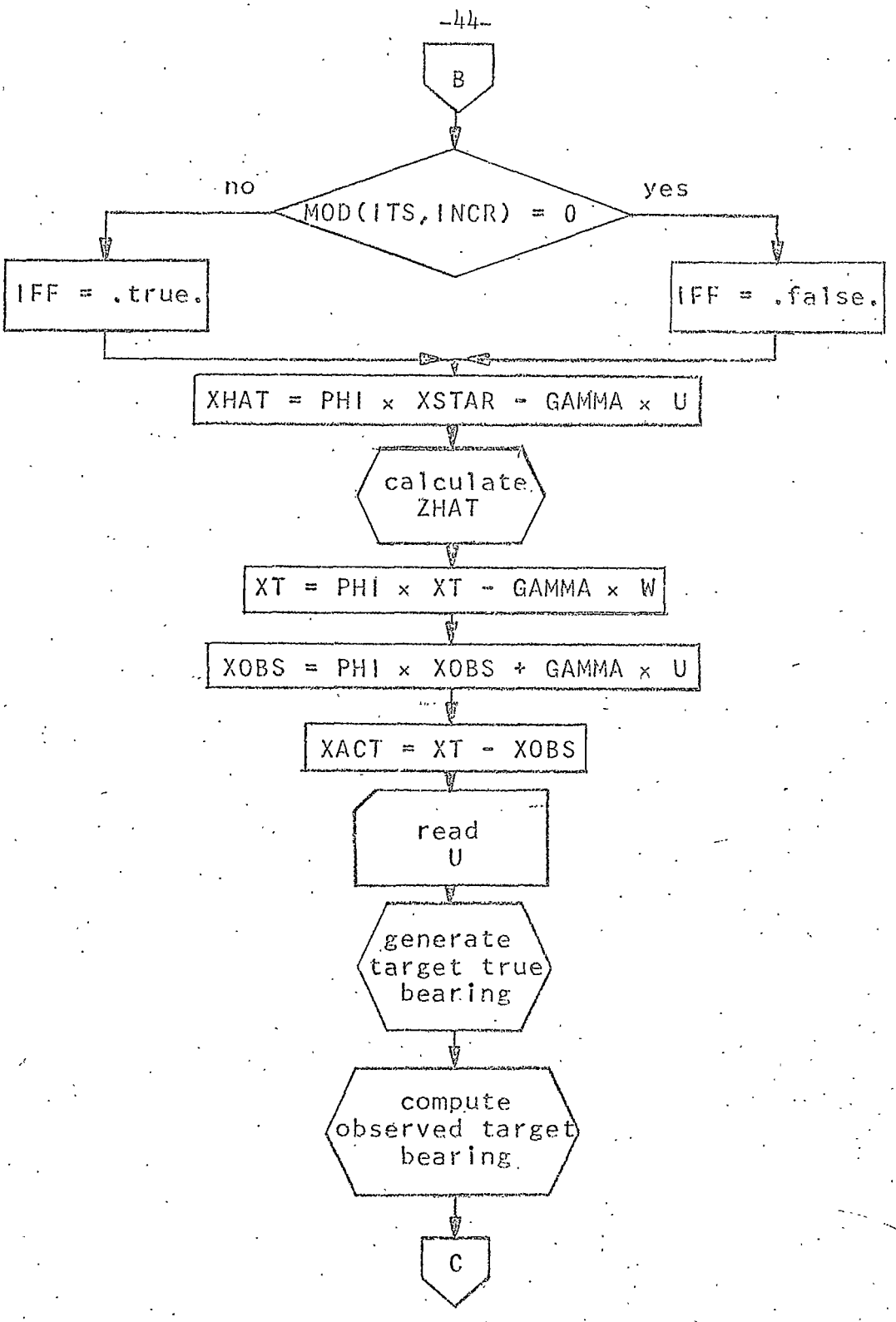



Figure 2.2. (Continued)

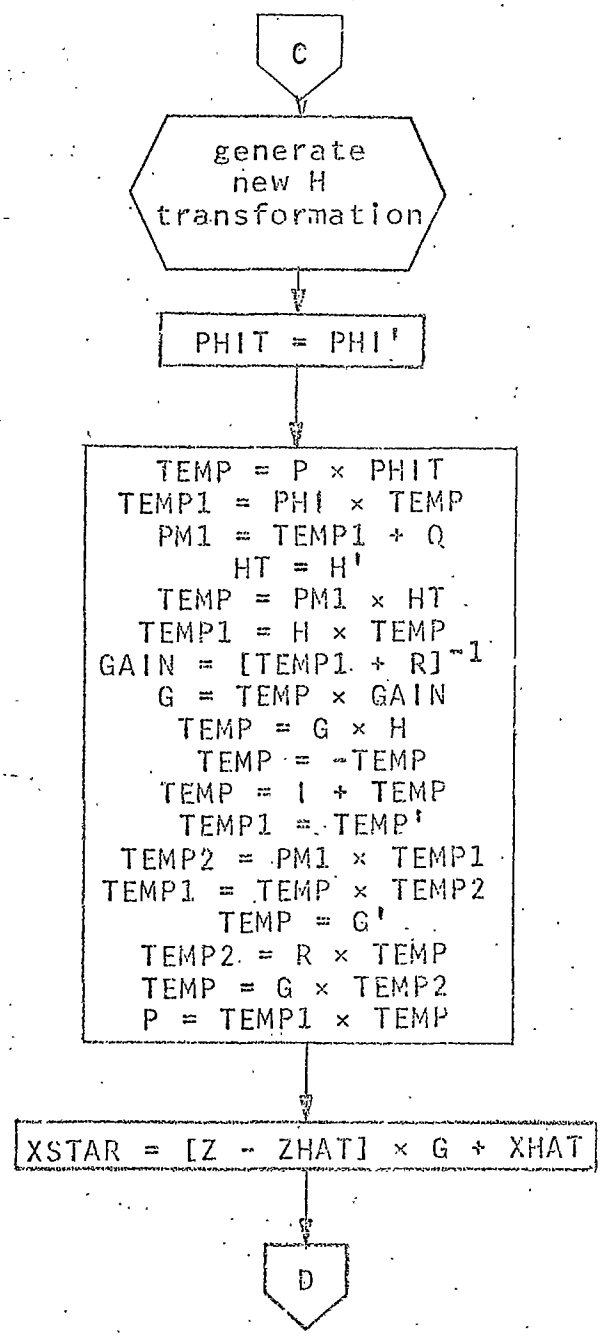



Figure 2.2. (Continued)

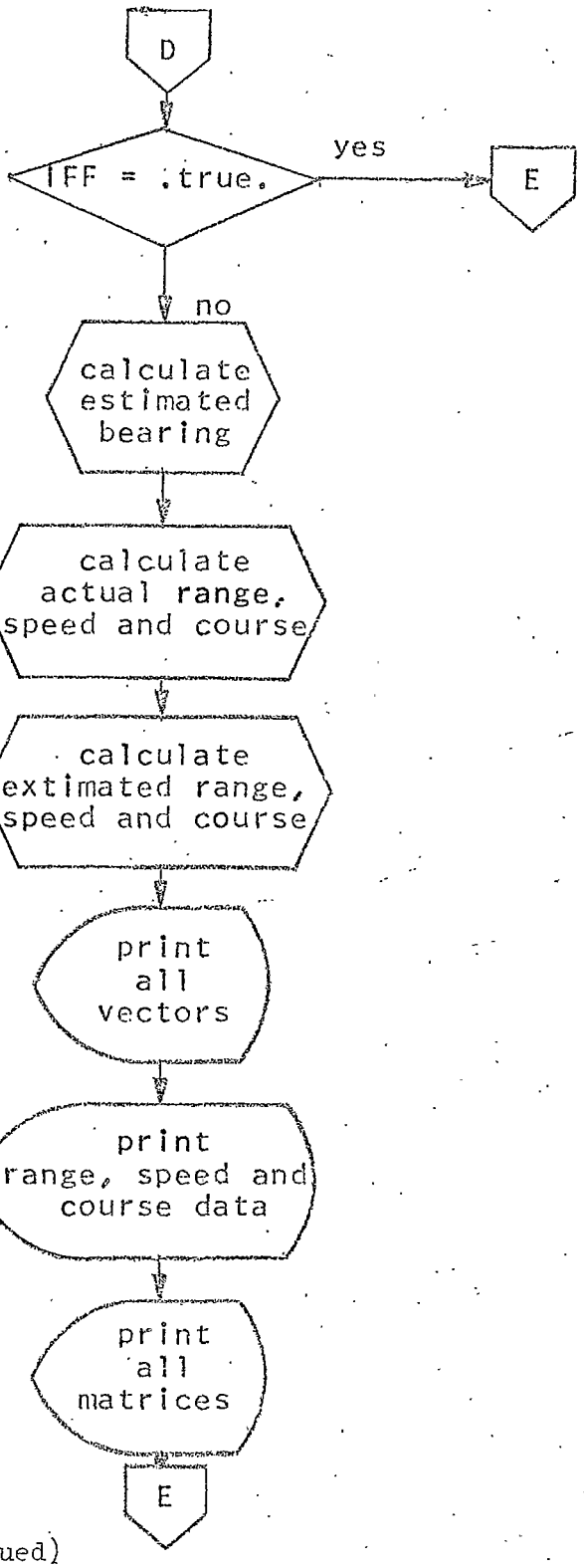



Figure 2.2. (Continued)

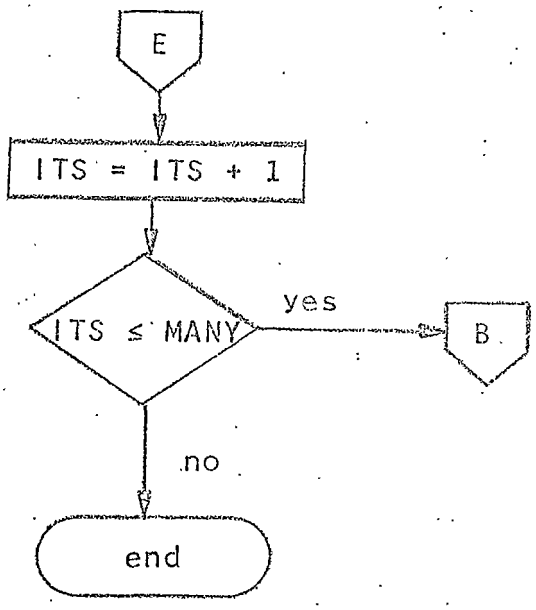



Figure 2.2. (Concluded)

Table 2.1. Identifiers for Kalman Filter Example

| Identifier used in program | Purpose | Identifier used in equations |
|---|---|---|
| STDEV | Standard deviation of bearing error | - |
| STDEW | Standard deviation of target exciting noise | - |
| Q | Covariance matrix of target exciting noise | $Q$ |
| R | Covariance matrix of bearing error | $R$ |
| PHI | State transition matrix | $\phi$ |
| P | Error covariance matrix of $x^*(k\ k)$ | $\Sigma(k\|k)$ |
| PM1 | Error covariance matrix of $\hat{x}(k\|k-1)$ | $\Sigma(k\|k-1)$ |
| GAMMA | Distribution matrix | $\Gamma$ |
| XSTAR | Estimate of x(k) based on k observations | $x^*(k\|k)$ |
| XHAT | Estimate of x(k) based on k-1 observations | $\hat{x}(k\|k-1)$ |
| XOBS | State of observer | - |
| XT | State of target | - |
| XACT | Relative state of target | $x(k)$ |
| Z | Actual observation | $z(k)$ |
| ZHAT | Estimated observation | $\hat{z}(k\|k-1)$ |

Table 2.1. (Continued)

| | | |
|---|---|---|
| W | Target exciting noise | w(k) |
| V | Bearing noise | v(k) |
| U | Observer accelerations | u(k) |
| H | Linearized observation system | H(k) |
| HT | Transpose of H | H'(k) |
| PHIT | Transpose of PHI | $\phi'$ |
| G | The optimal filter gain | G(k) |

Figure 2.3 shows a plot of the tracks of the observer, the target and the estimate of the target for a typical run of the Kalman filter example program. An x denotes the initial states of the target and of the observer and the initial estimation. Each dot on the tracks indicates 20 iterations. This is done to provide some time relationship between the tracks. The axes are marked in kilometers. It can be seen that the estimation quickly converges to the target track.

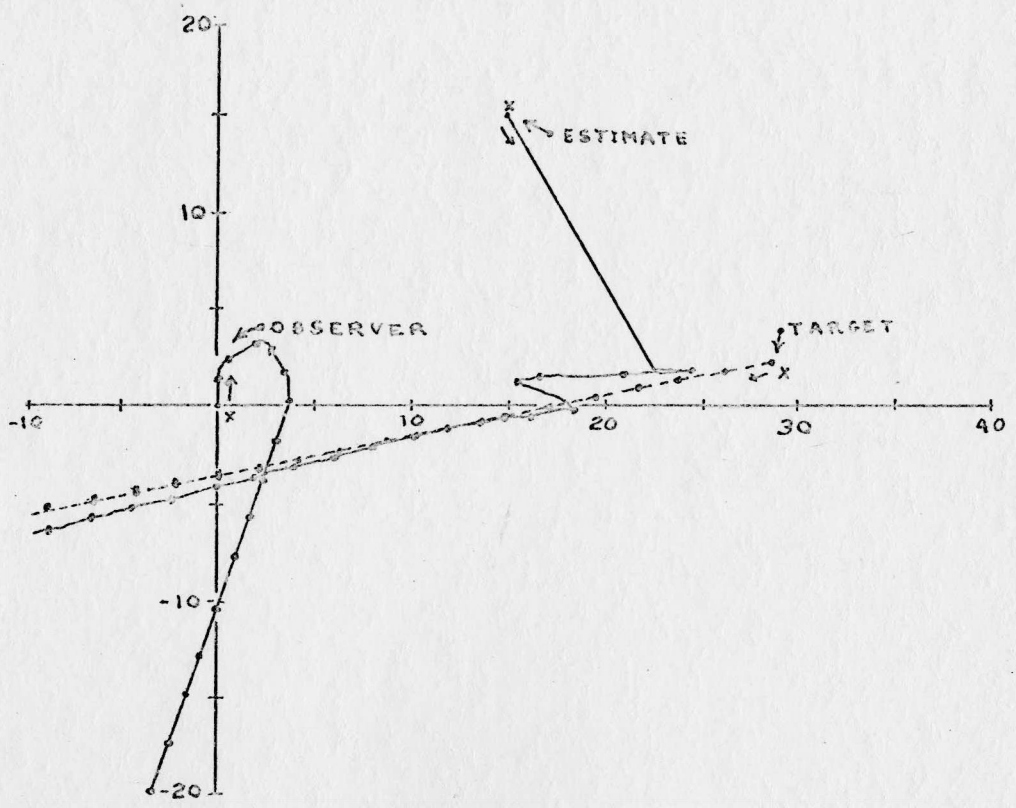



Figure 2.3. Plot of Observer, Target and Estimate Tracks

# Chapter 3

# DESIGN OF A CELLULAR COMPUTER

## 3.1 Introduction

In this chapter a design is presented for a computer which is specially structured for the types of operations used in the Kalman filter algorithm. The basic structure of the computer is given followed by detailed descriptions of the various parts. This computer will hereafter be referred to as the KF machine.

As indicated in Section 2.2 the Kalman filter algorithm involves a large number of operations on matrices and vectors. These operations include the addition, subtraction, multiplication, inversion and transposing of several matrices. Since many applications where the Kalman filter might be used require the calculations to be done as rapidly as possible, a computer used to perform these calculations should be able to perform them efficiently. Therefore, a computer with an array structure is proposed. Since not all operations deal with matrices or vectors, two types of instructions are suggested: (1) array instructions which perform operations on the matrices or vectors and (2) non-array instructions which perform operations on scalars.

## 3.2 KF Machine Structure

The proposed KF machine as shown in Figure 3.1 has four sections: an array of processing cells, row and column data and control registers and a global control unit.

The array of processing cells performs computations involving matrices and vectors. This type of data is stored in the array of processing cells with each $ij^{th}$ cell containing the $ij^{th}$ element of a matrix.

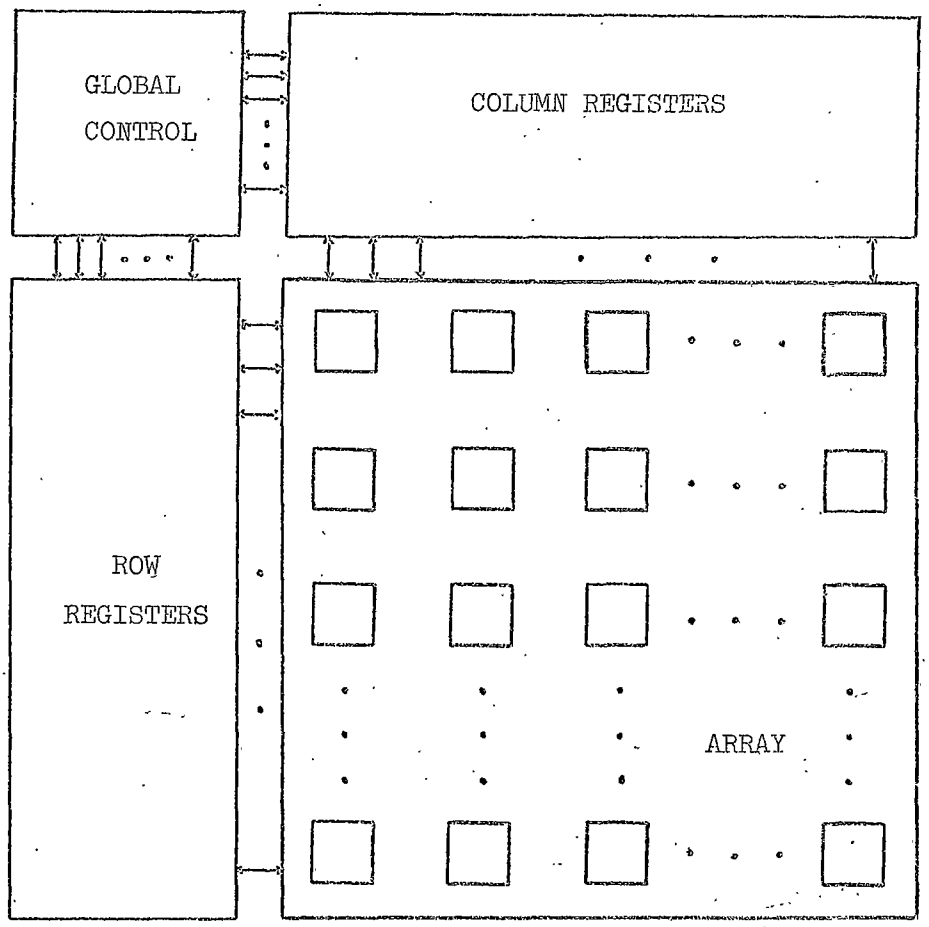



Figure 3.1. KF Machine Structure

Row and Column data and control registers provide an interface between the array of processing cells and the global control unit.

The purpose of the global control unit is to fetch instructions from its memory, decode the instructions and execute them. Non-array instructions are executed by the global control unit within its structure while array instructions are executed by the array of processing cells under the control of the global control unit.

The remainder of this chapter is devoted to the detailed description of component parts of the proposed KF machine.

## 3.3 Special Logic Units

In the descriptions of portions of the KF machine some specialized logic units are used extensively. This section describes some of these units and the conventions used for logic gates and for logic equations.

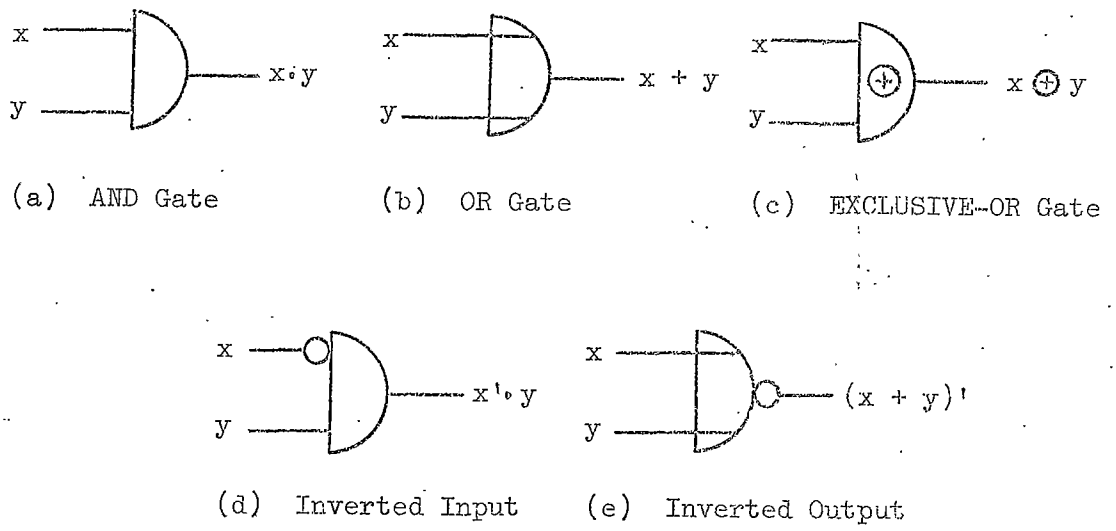
### 3.3.1 Logic Conventions

The specialized logic units to be described are assumed to be constructed of basic logic gates such as are shown in Figure 3.2. The inverted convention for inputs or outputs is indicated by Figures 3.2(d) and 3.2(e) respectively. For logic equations inverted variables are represented by an apostrophe (') following the variable. The logical OR operation is represented by a plus sign (+), the logical EXCLUSIVE-OR by a circled plus sign ($\oplus$) and the logical AND by a dot ($\cdot$).

### 3.3.2 Line-Select Gate

The line-select gate shown in Figure 3.3(a) is a three-input gate which realizes the function

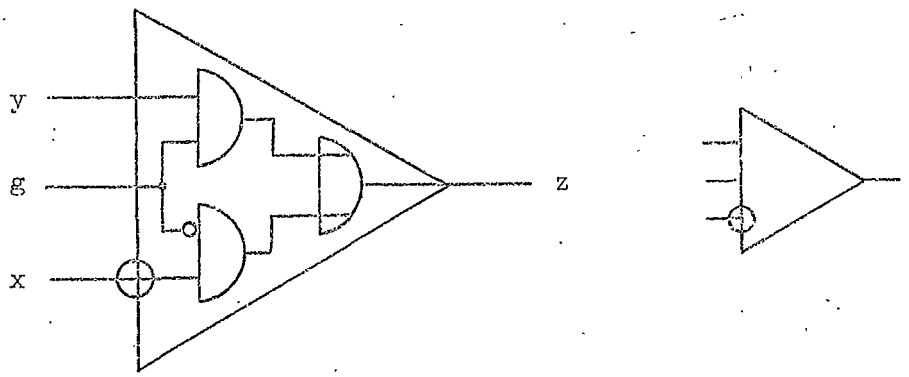
$$z = g' \cdot x + g \cdot y$$




(a) AND Gate (b) OR Gate (c) EXCLUSIVE-OR Gate

(d) Inverted Input (e) Inverted Output

Figure 3.2. Logic Gate Symbols.




(a) (b)

Figure 3.3 The Line-Select Gate

Its purpose is to select, as output z, the x input line if the gate line g is 0 or the y input line if the gate line g is 1. When used in other figures the gate will be represented as in Figure 3.3(b). The center input is always assumed to be g and the circled input is always assumed to be x.

3.3.3 Add-One Cell

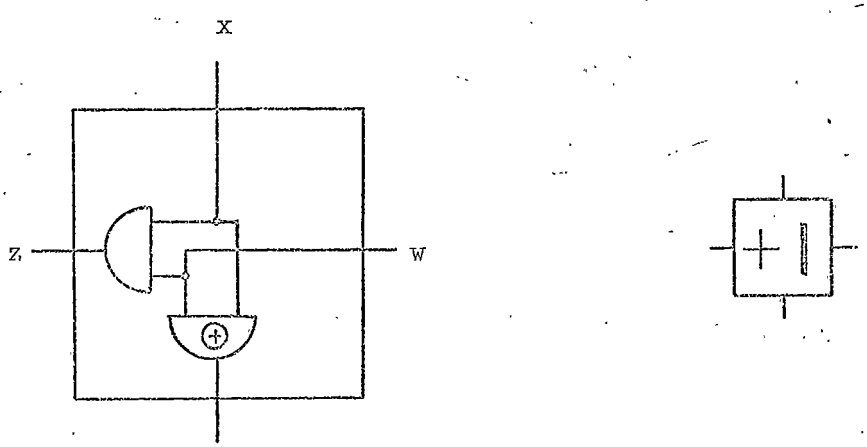
The add-one cell shown in Figure 3.4(a) is a two-input, two-output cell which produces the functions

$$z = x \cdot w$$

and

$$y = x \oplus w .$$

This cell was mentioned by Hennie [6] and later by Minnick [14].




(a) (b)

Figure 3.4. The Add-One Cell

When used in other figures the cell will be represented as in Figure 3.4(b).

The cell can be used to form a parallel add-one circuit for adding one to an N-bit binary number. N cells are connected in a cascade with

parallel inputs from the N-bit binary number as shown in Figure 3.5. This circuit is the one-dimensional decoder array discussed by Minnick (14). If the w input to the low-order add-one cell is supplied with a 0, the outputs $y_i$ will be equal to their corresponding $x_i$'s. However, if the input is a 1 the outputs $y_i$ will represent a binary number whose value is one more than the input number.

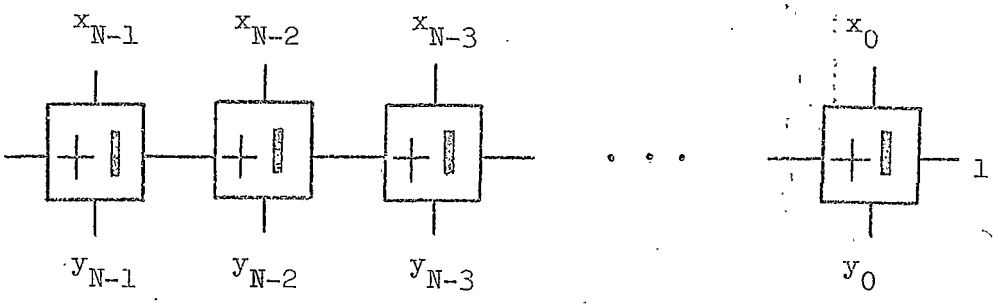



Figure 3.5. The Add-One Cascade

It should be noted that the cascade can be used to add any power of two if the 1 input is injected into the cascade at a higher-order bit position as indicated in the add-four cascade shown in Figure 3.6.

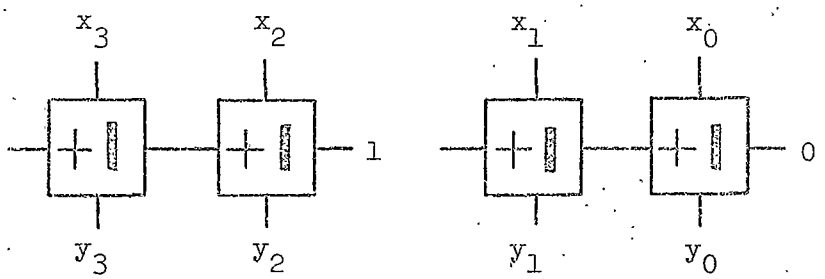



Figure 3.6. The Add-Four Cascade

### 3.3.4 One's-Two's Complementor Cell

The one's-two's complementor cell shown in Figure 3.6(a) is a two-input, two-output cell which produces the functions

$$c = a + b$$

and

$$d = a \oplus b.$$

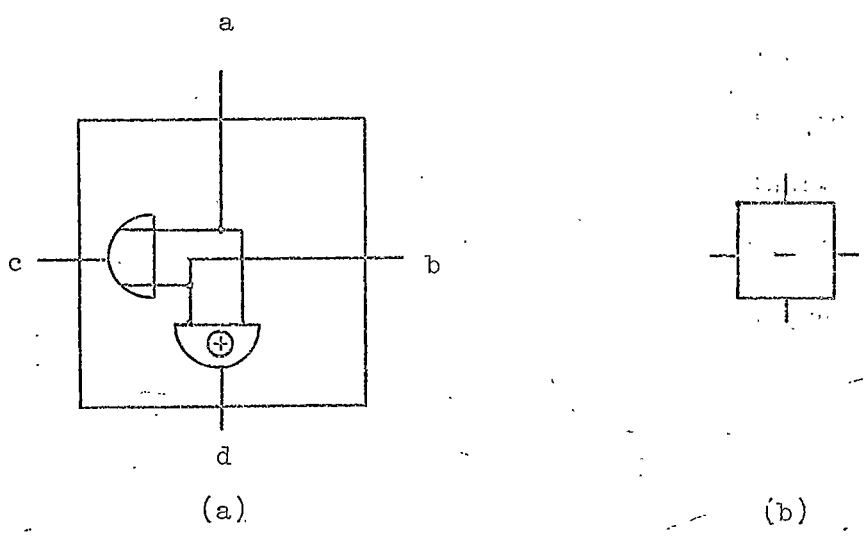



Figure 3.7. The One's-Two's Complementor Cell

When used in other figures the cell will be represented as in Figure 3.6(b).

The cell can be used to form a parallel one's-two's complementor circuit for N-bit binary numbers. N cells are connected in a cascade with parallel inputs $a_i$ from the N-bit binary number as shown in Figure 3.8. If the b input of the low-order cell is supplied with a 0 the $d_i$ outputs will represent the two's complement of the

binary number. However, if the input is supplied with a 1 the $d_i$ outputs will represent the one's complement of the binary number.

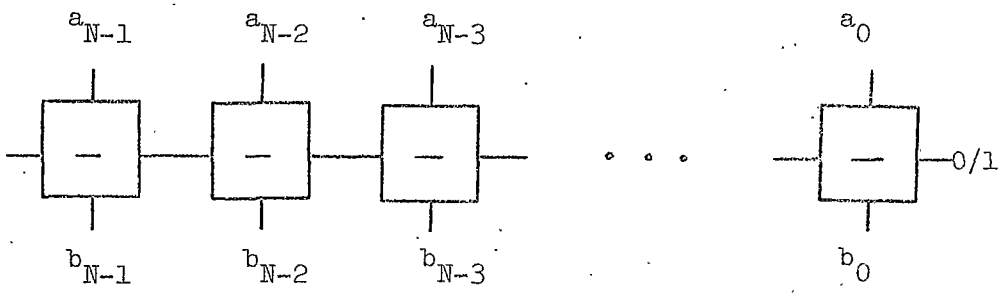



Figure 3.8. The One's-Two's Complementor Cascade

### 3.3.5 Comparator Cell

Much of the work on comparator circuits has been done by people such as Lee ( 10 ) and McKeever ( 12 ) who were concerned with associative memories.

The comparator cell shown in Figure 3.9(a) is a four-input, two-output cell which produces the functions

$$e = c + a \cdot b' \cdot d'$$

and

$$f = d + b \cdot a' \cdot c'$$

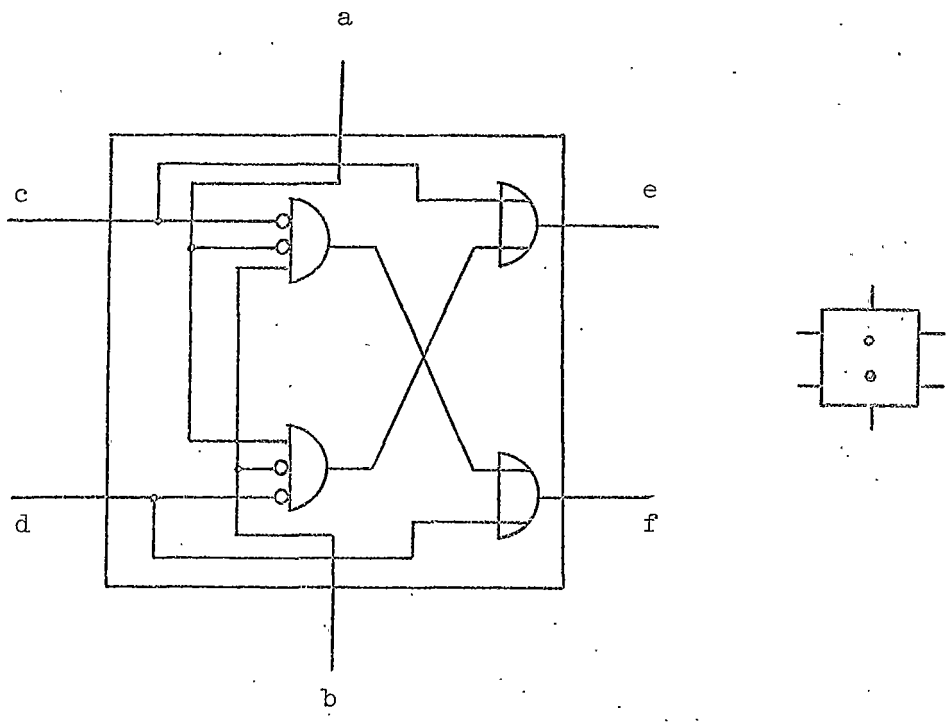



Figure 3.9. The Comparator Cell

When used in other figures the cell will be represented as in Figure 3.9(b).

The cell can be used to form a parallel comparator for comparing two N-bit binary numbers A and B. N cells are connected in a cascade with parallel inputs $a_i$ from the binary number A, and $b_i$ from the binary number B, as shown in Figure 3.10. The inputs c and d to the high-order comparator cell are supplied with 0's.

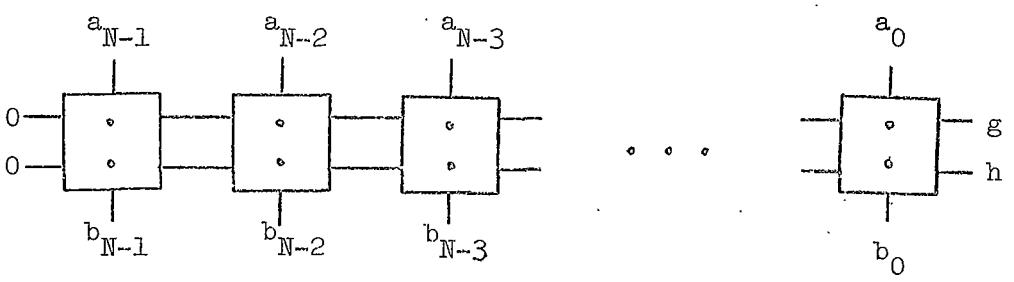



Figure 3.10. The Comparator Cascade

If the binary number A is larger than the binary number B the output line g will be a 0 and the output line h will be a 1. Similarly, if B is larger than A, the output line g will be a 1 and the output line h will be a 0. If the numbers are equal, both output lines will be 0. Thus, it is possible to detect the conditions A>B, A = B or A<B. The condition A = B is given by the function

$$k = g' \cdot h'.$$

### 3.3.6 Adder Cell

The adder cell shown in Figure 3.11(a) is a three-input, two-output cell which produces the functions

$$z = x \oplus y \oplus w$$

and

$$n = x \cdot y + y \cdot w + x \cdot w$$

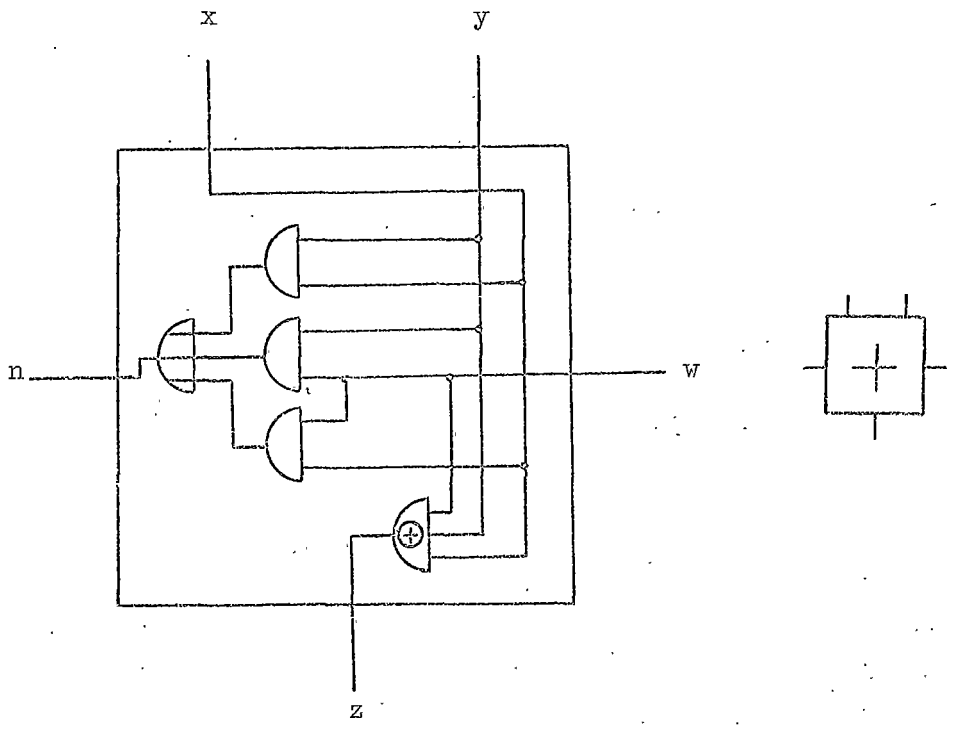



Figure 3.11. The Adder Cell

When used in other figures the cell will be represented as in Figure 3.11(b).

The cell can be used to form a parallel ripple-carry adder for obtaining the sum Z of two N-bit binary numbers X and Y. N cells are connected in a cascade with parallel inputs $x_i$ from the binary number X and $y_i$ from the binary number Y, as shown in Figure 3.12. The outputs $z_i$ represent a binary number Z whose value is the sum of the input numbers X and Y. The w input to the low-order cell of the cascade is supplied with a 0.

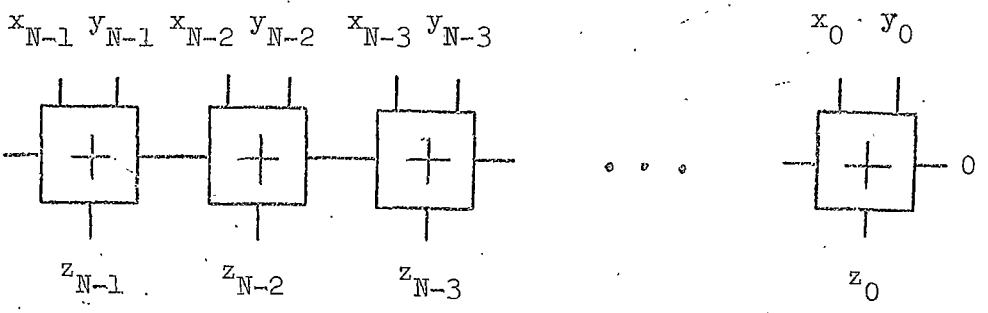



Figure 3.12. The Adder Cascade

### 3.3.7 Adder-Subtractor Cell

The adder-subtractor cell shown in Figure 3.13(a) is a four-input, two-output cell which produces the functions

$$z = x \oplus y \oplus w$$

and

$$n = y \cdot w + (x \oplus g) \cdot (w + y).$$

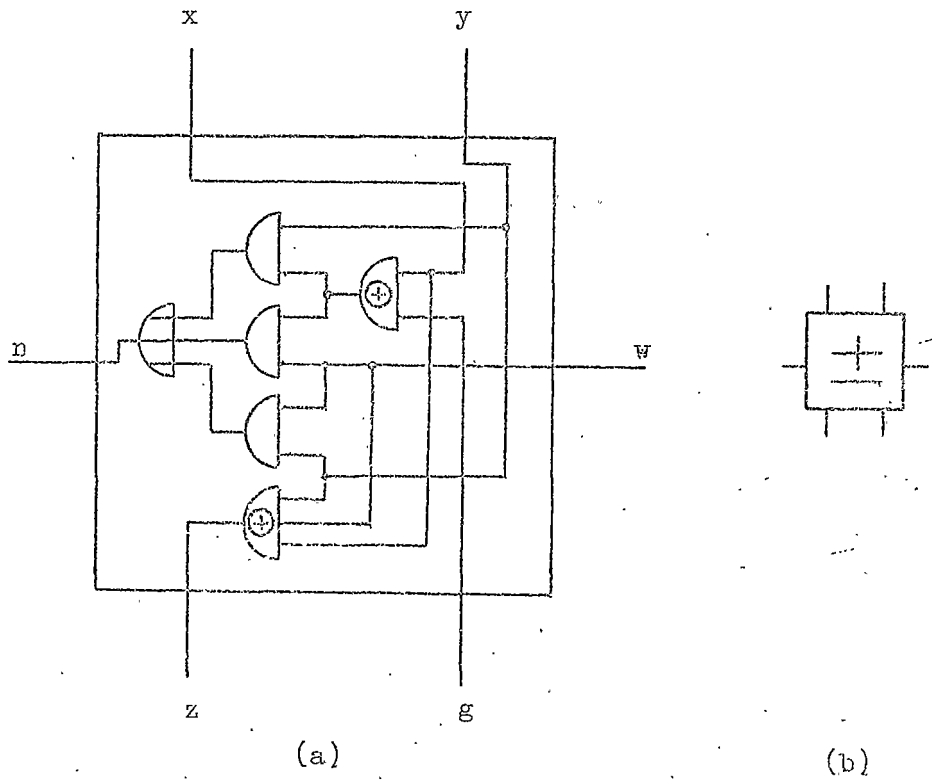



Figure 3.13. The Adder-Subtractor Cell

When used in other figures the cell will be represented as in Figure 3.13(b).

The cell can be used to form a parallel ripple-carry adder-subtractor for obtaining the sum or difference of two N-bit binary numbers X and Y. N cells are connected in a cascade with parallel inputs $x_i$ from the binary number X and $y_i$ from the binary number Y as shown in Figure 3.14.

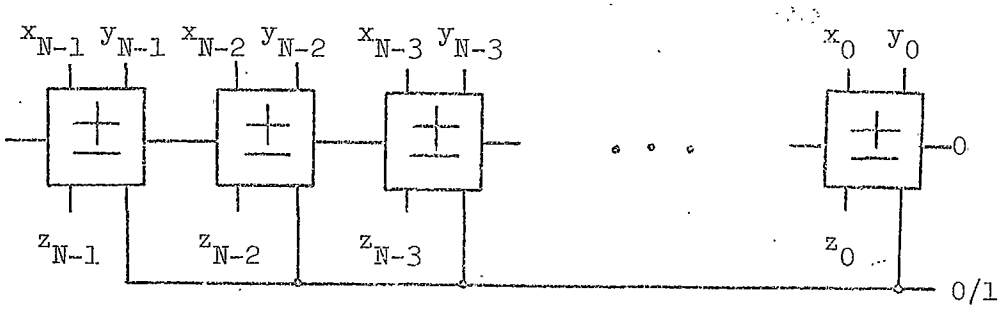



Figure 3.14. The Adder-Subtractor Cascade

The w input to the low-order cell is supplied with a 0. If the g input of each cell is supplied with a 0, the $z_i$ outputs will represent a binary number Z whose value is the sum of the input numbers X and Y. If the g input of each cell is supplied with a 1; the $z_i$ outputs will represent the difference of the input numbers X and Y. That is, X - Y.

### 3.3.8 General Register Cell

The shift and other registers can be thought of as being made of a cascade of identical cells. Such a cell is shown in Figure 3.15.

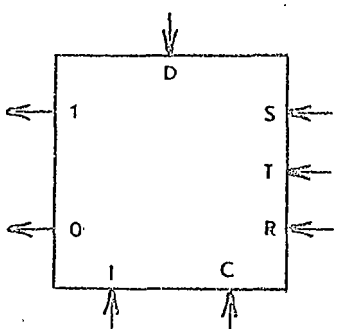



Figure 3.15. The General Register Cell

The cell contains a one-bit register whose contents and its complement appear on lines 1 and 0 respectively. An input on the toggle line T causes the register to switch to 1 if it was 0 and to 0 if it was 1. The switch occurs on the transition of the input from a 0 to a 1.

An input on the clock line C causes the data to be transferred into the register from the set line S and the reset line R. The change occurs on the transition of the input C from a 1 to a 0. Thus, a string of cells connected as shown in Figure 3.16 acts as a shift register when the C input is applied.

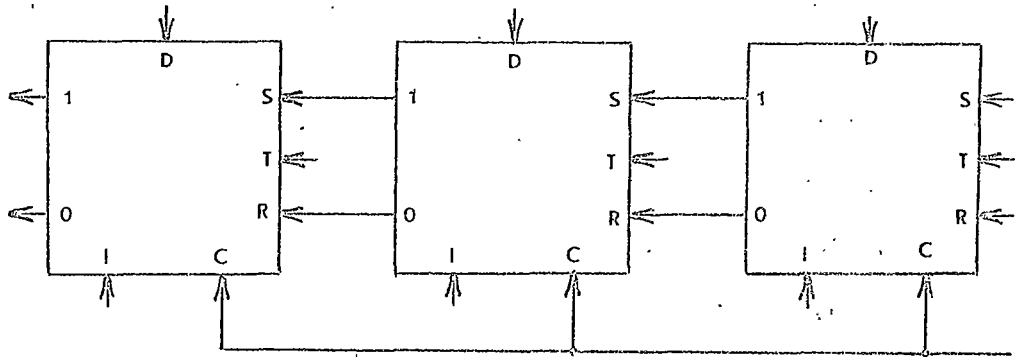
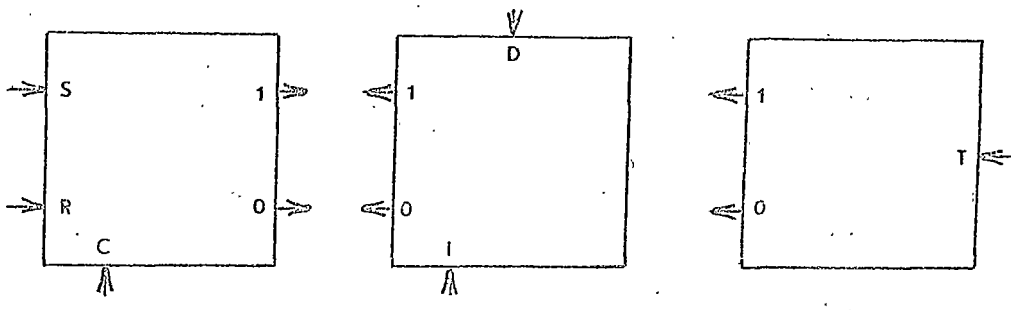


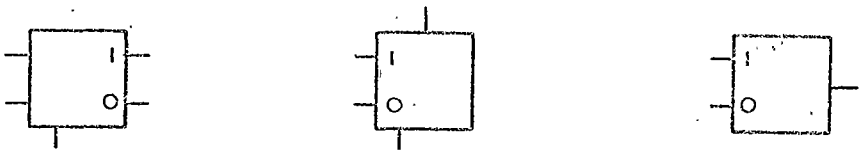

Figure 3.16. A Shift Register Cascade

An input on the input line I causes the register to be set to the value of the data line D. This input is used to transfer data in parallel into a string of cells.

Not all registers will require cells with the capability of the general register cell. However, this model serves to represent all of the types of cells required. When the general cell is used in a register, only those inputs or outputs required for that particular use will be shown. A convention will be adopted in order to avoid the necessity of labeling all inputs and outputs for every use: the outputs will be marked as 1 and 0 and the inputs will be identified by their position with respect to the outputs. Some examples of cells in which not all inputs and outputs are used are given in Figure 3.17(a); their corresponding equivalent forms are given in Figure 3.17(b).




(a)




(b)

Figure 3.17. Special Forms of General Register Cell

It should be noted that the register cell outputs may be directed either to the left or to the right. In either case the I input line is the line closest to the output side and the C line is the line closest to the side with the S, R and T inputs.

## 3.4 Array Interconnection Structure

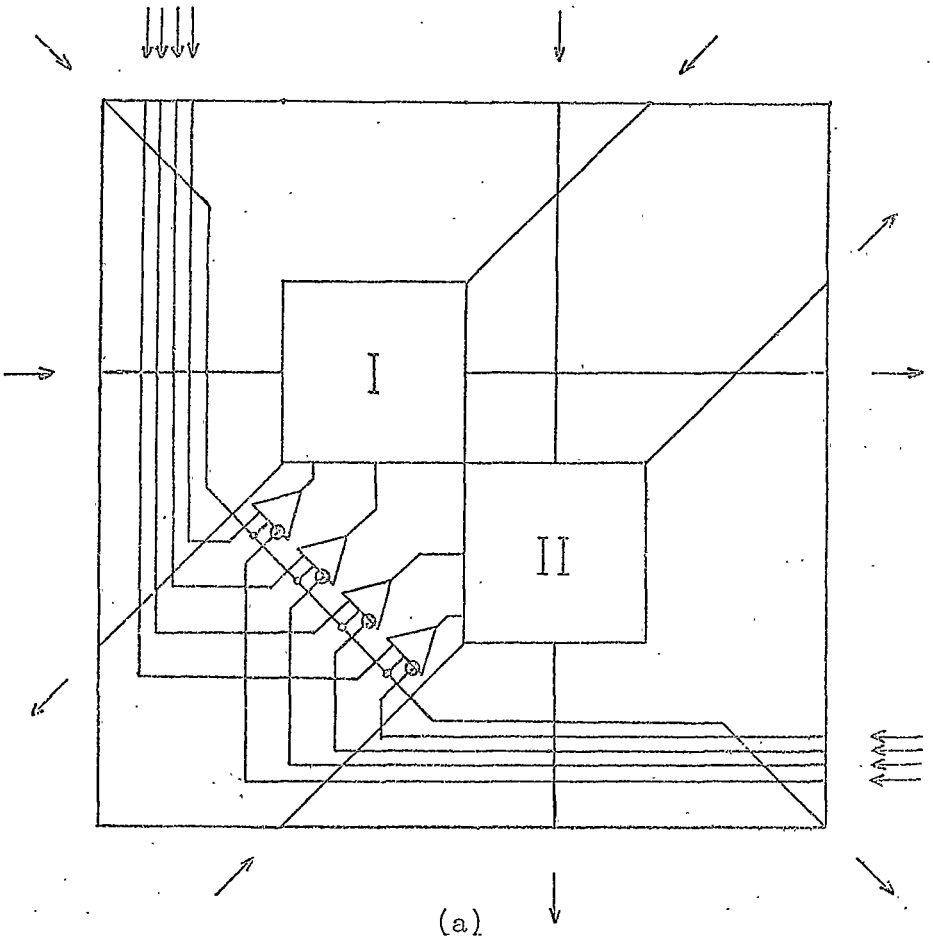
In order to describe the array interconnection structure for the proposed KF machine, it will be helpful first to consider just the routing portion of the cells. An equivalent treatment is to develop

the interconnection structure for cells with just a routing capability.

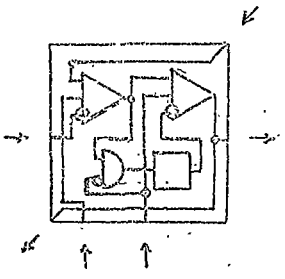
Such a routing cell is shown in Figure 3.18(a). The cell consists of two inner register cells I shown in Figure 3.18(b) and II shown in Figure 3.18(c) and some control logic. The inner cells I and II each have two data inputs, two data outputs and two control inputs. The data output lines are tied together internally so there is really only one output for each inner cell. Inner cells I have data inputs from corresponding inner cells of the routing cells west and northeast of their routing cell. Inner cells II have data inputs from the corresponding inner cells of the routing cells north and southwest of their routing cell.

One of the control lines into each of the inner cells selects the data line to be used as input to that cell. The other line selects whether the incoming data is to be routed to the register within the cell or around it. If it is routed to the register, the output of the cell corresponds to the low-order bit of the register and the input goes to the high-order bit of the register. If the data is routed around the register, the output of the cell corresponds to the data on the selected input line.

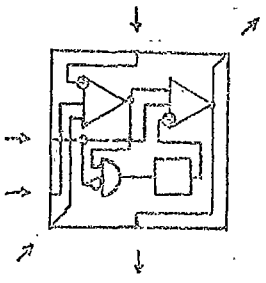
The routing cell has two sets of control line inputs for the inner cells: the set entering the cell from the top comes from the row control bus and the set entering the cell from the right comes from the column control bus. The control line passing through the cell diagonally from the upper-left to the lower-right selects which set of




(a)



(b) I Inner Cell



(c) II Inner Cell

Figure 3.18. The Routing Cell

control inputs the cell is to use. The column control lines are used if this line is 0, the row control lines are used if it is 1. This control goes to all cells in the array so either all cells are using row control lines or all cells are using column control lines. The arrangement of control lines is shown for a 4x4 array in Figure 3.19.

Each cell is capable of 16 different routing configurations depending on the values of the selected control lines. These configurations are shown in Figure 3.20. Cells in which the register is bypassed are shown with the input passing through to the output. Only the output line corresponding to the selected input line is shown although the same output would appear on the other output line as well.

Some of the basic routing operations can now be demonstrated using 4x4 arrays of the cells just described. No particular interconnection pattern is assumed at this time. When a sufficient set of basic routing patterns has been developed an interconnection pattern incorporating all of these can be constructed.

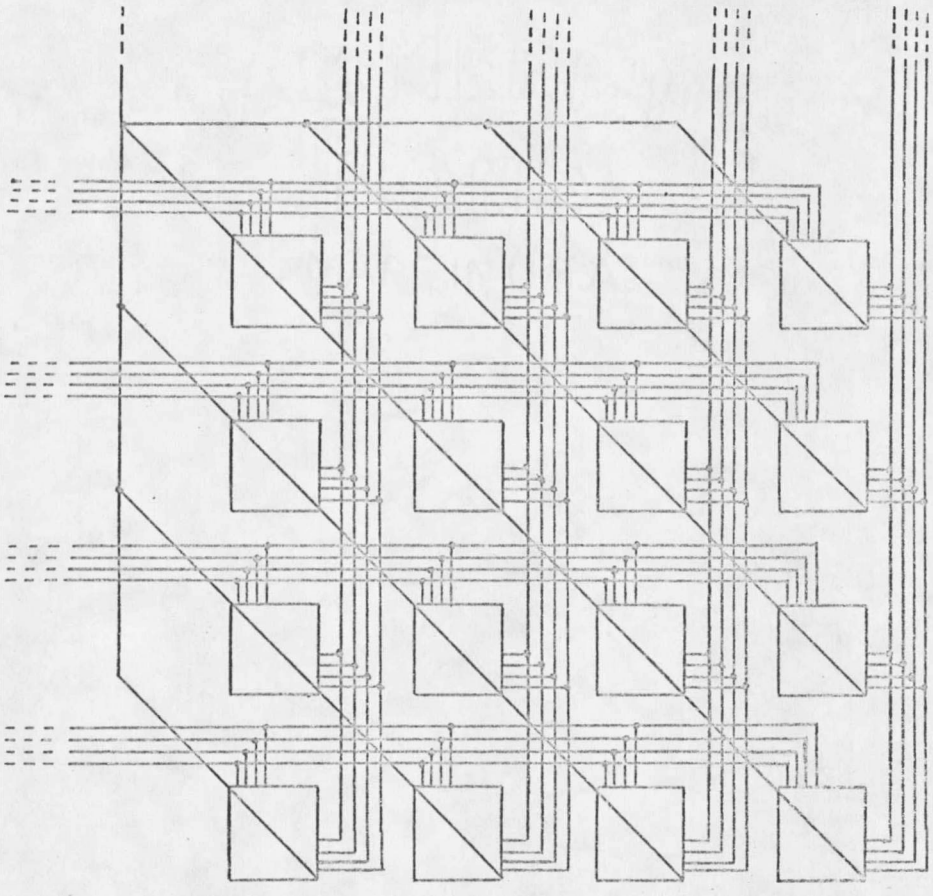


Figure 3.19. Routing Array Control-Line Interconnection

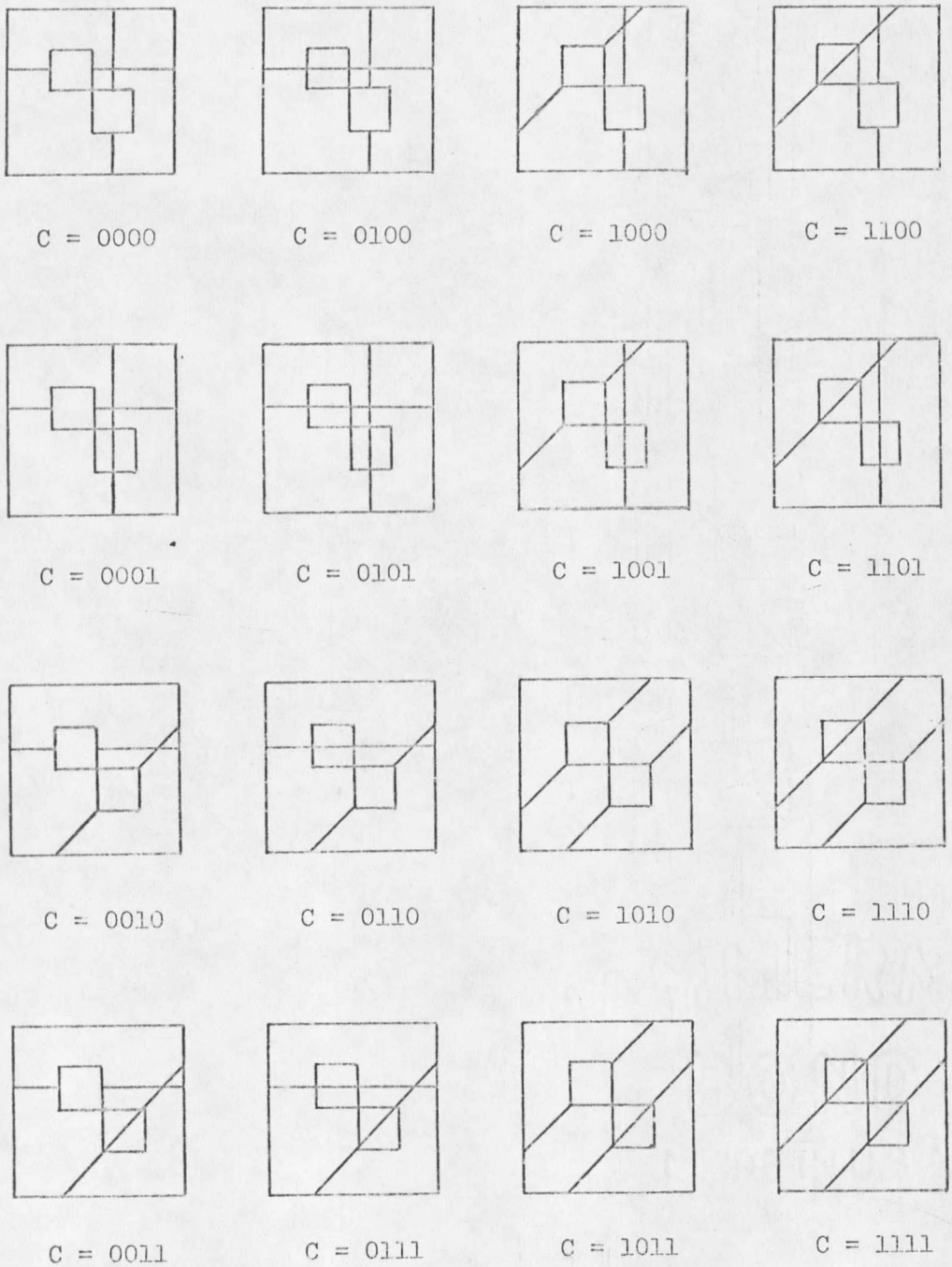



Figure 3.20. Possible Routing Cell Configurations

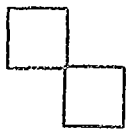


Figure 3.21. A Routing Cell Representation

The routing cells are drawn as shown in Figure 3.21 which represents the two inner cells of Figure 3.18(a). Control lines are not shown. For all of the interconnection patterns the registers are connected together to form long shift-registers. There is assumed to be one element of an array of data stored in each of the inner cells. The data is shifted from cell to cell along the interconnection paths.

Figure 3.22 shows an interconnection pattern for rotating the data array, stored in the registers of the I inner cells, to the right. The outputs of the right edge cells connect to the inputs of the left edge cells forming a cylindrical interconnection.

The interconnection pattern of Figure 3.23 is similar to that of Figure 3.22 except that cells in the second and fourth columns are bypassed. Thus only the first and third columns take an active part in the routing process. This scheme could be used to perform a column interchange operation of a sub-array rotation.

Figure 3.24 shows an interconnection pattern for rotating the data array, stored in the II inner cells, down. The outputs of the bottom

edge cells connect to the inputs of the top edge cells forming a cylindrical interconnection as before. Also as before some of the rows of cells may be by-passed; however, this is not shown in the figure.

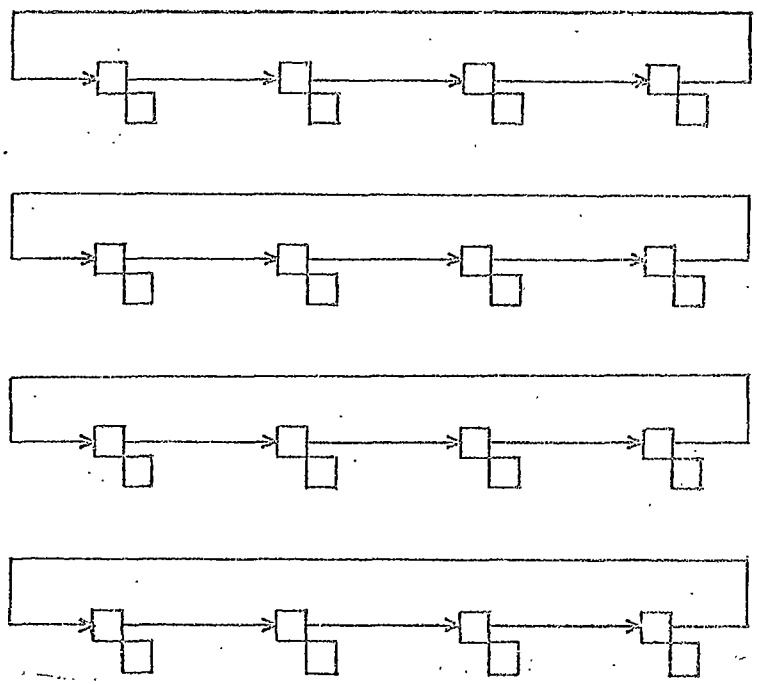
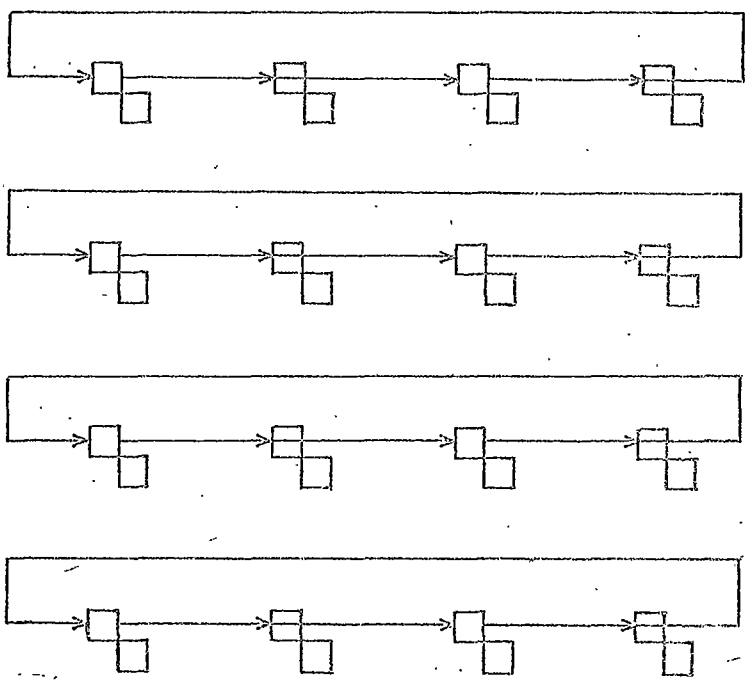


Figure 3.22. Rotate-Right Interconnection



Fogire 3.23. Interchange-Column Interconnection

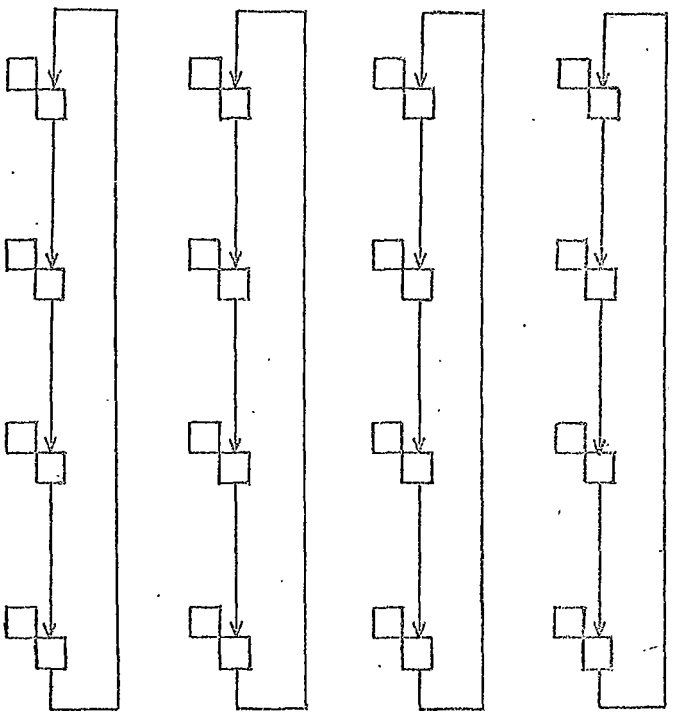


Figure 3.24. Rotate-Down Interconnection.

Figure 3.25 shows in interconnection pattern for replacing the data array, stored in the registers of the II inner cells (Figure 3.18) by a skewed-up version of the data array stored in the registers of the I inner cells.

Similarly Figure 3.26 shows an interconnection pattern for replacing the data array, stored in the registers of the I inner cells by a skewed-left version of the data array stored in the registers of the II inner cells.

Finally, Figure 3.27 shows an interconnection pattern for replacing the data array stored in the registers of the I inner cells by the transpose of the data array stored in the registers of the II inner cells.

It should be noted that in some cases more than one type of routing may occur at once. For example it would be possible to rotate the data array, stored in the registers of the I inner cells, to the right and at the same time rotate the data array, stored in the registers of the II inner cells, down.

A 4x4 routing array incorporating all of the interconnection patterns just discussed is shown in Figure 3.28. The array uses routing cells as shown in Figure 3.18.

Line-select gates along the left edge of the array are controlled by control line f; these gates are used to direct data to the west input of the I inner cells of the left edge routing cells. The data comes from either the external row inputs $b_i$ or from the right-edge line-select gates.

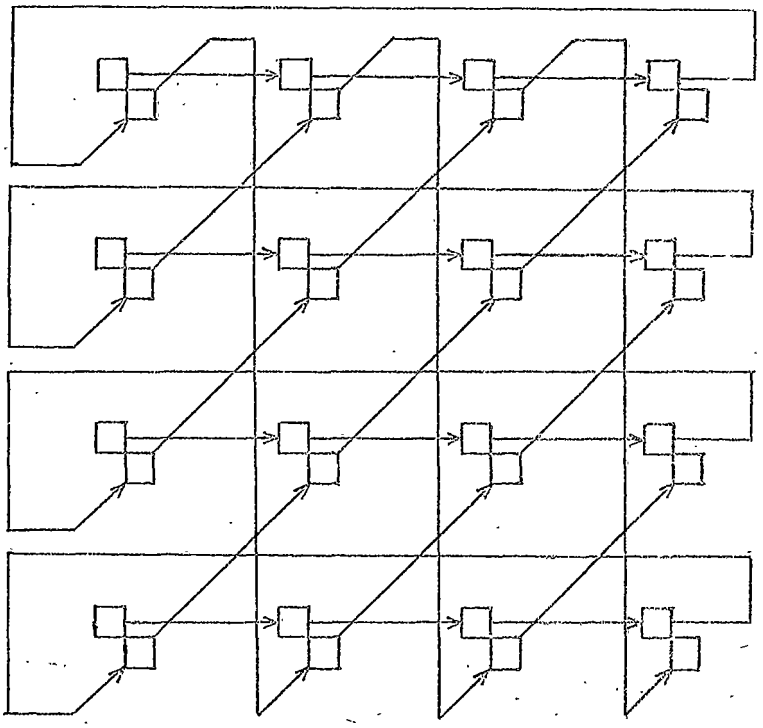


Figure 3.25. Skew-Up Interconnection

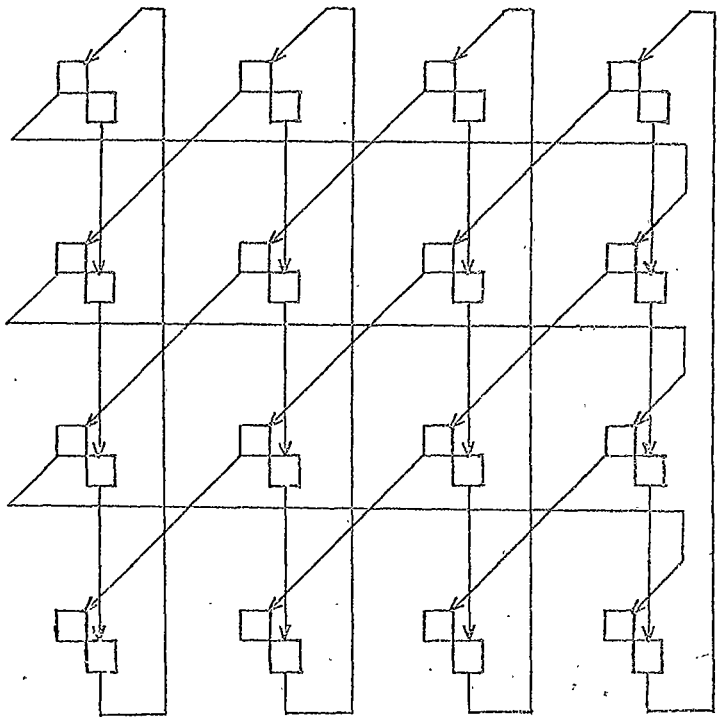


Figure 3.26. Skew-Left Interconnection

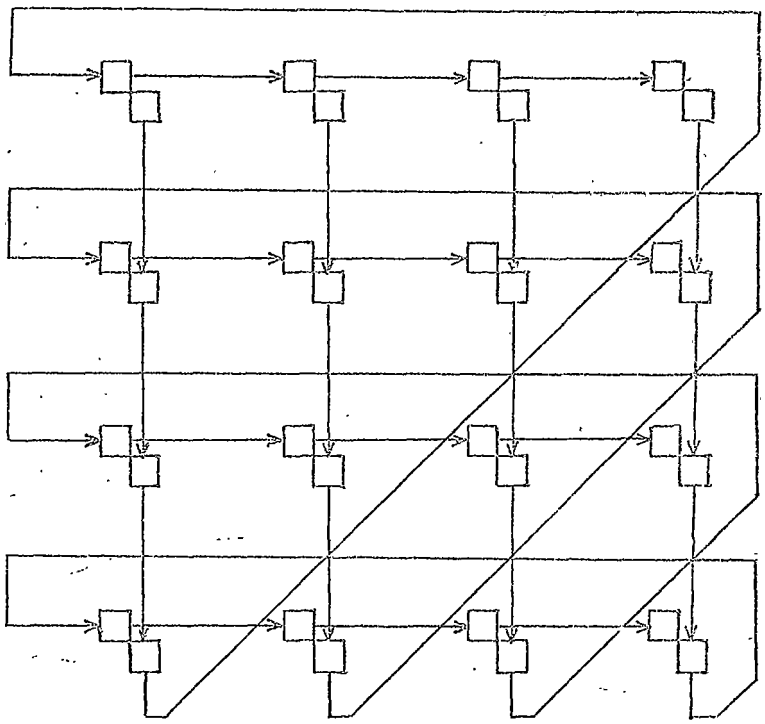


Figure 3.27. Transpose Interconnection

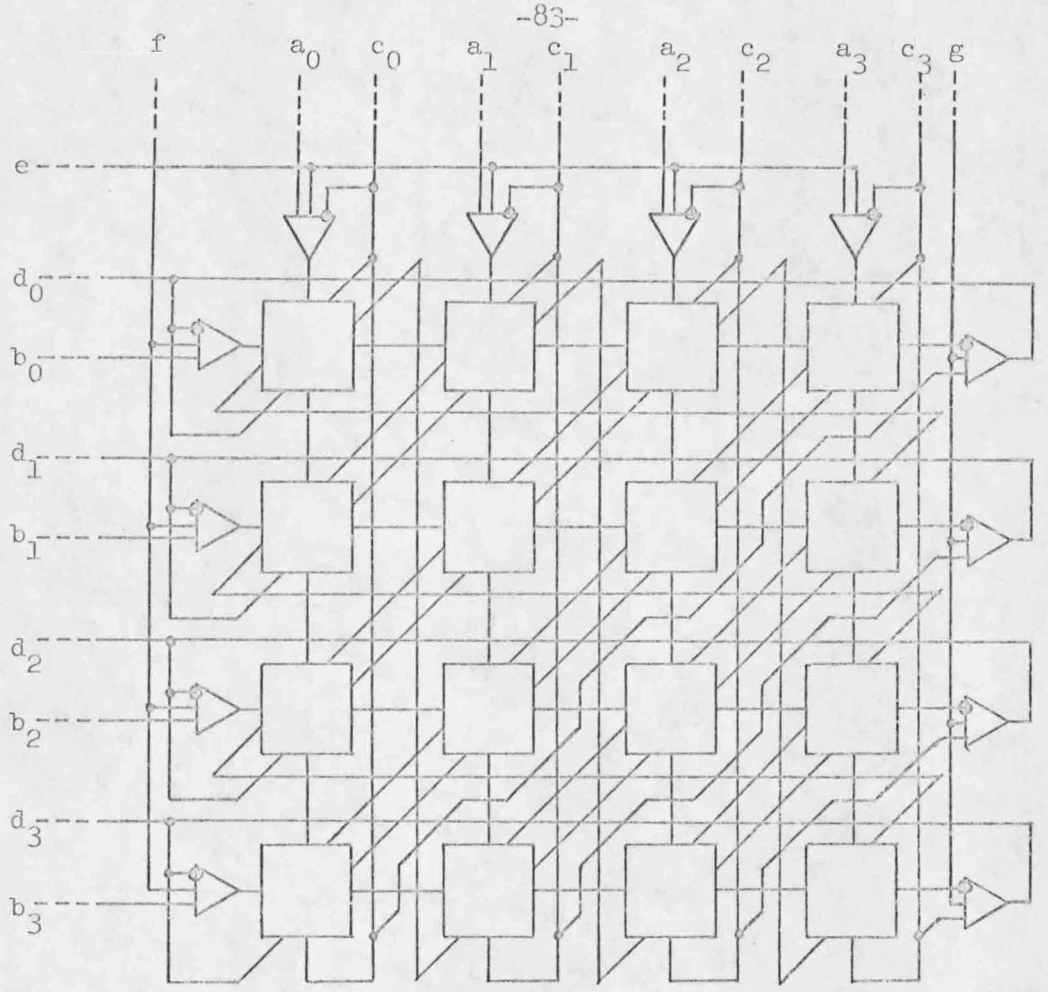



Figure 3.28. Routing Array Interconnection

Line-select gates along the right edge of the array of Figure 3.28 are controlled by control line g, and these gates are used to direct data to lines which go to the external row outputs $d_i$, to the left-edge line-select gates and to the south-east input of the II inner cells of the left-edge routing cells. The data comes either from the outputs of the I inner cells of the right-edge routing cells or from the outputs of the II inner cells of the bottom-edge routing cells.

Line-select gates along the top edge of the array of Figure 3.28 are controlled by control line e; these gates are used to direct data to the north inputs of the II inner cells of the top-edge routing cells. The data comes either from the external column inputs $a_i$ or from the outputs of the II inner cells of the bottom-edge routing cells.

The outputs from the II inner cells of the bottom-edge routing cells also connect to the right-edge line-select gates, to the northeast inputs to the I inner cells of the top-edge routing cells and to the external column outputs $c_i$.

Within the array the outputs of the I inner cells connect to the northeast inputs of I inner cells located southwest of the given cell and to the west input of the I inner cells located east of the given cell. The outputs of the II inner cells connect to the southwest inputs of II inner cells located northeast of the given cell and to the north input of II inner cells located south of the given cell.

On the left edge of the array in Figure 3.28 the southwest outputs of the I inner cells are connected to the northeast inputs

of the I inner cells on the right edge of the array and down one row from the given cell. An exception is the lower-left cell of the array; the output of this cell is not used. The northeast outputs of the II inner cells on the top edge of the array are connected to the southwest inputs of II inner cells on the bottom edge of the array and right one column from the given cell. An exception is the upper-right cell of the array, which has an unused output.

The southwest outputs of the I inner cells on the bottom edge of the array are not used and the south outputs of the II inner cells are directed to the top and right line-select gates and to the top northeast inputs as mentioned before.

On the right edge of the array the northeast outputs of the II inner cells are not used and the east outputs of the I inner cells are directed to the right line-select gates as mentioned before.

## 3.5 Data Representation and Arithmetic

Before describing the structure of a processing cell it will be helpful to discuss the form in which numbers are represented and how the arithmetic is handled in the cells.

### 3.5.1 Data Representation

The data are represented in floating-point form. A 32-bit floating-point number consists of a sign (bit 0), a 24-bit binary proper fraction (bits 1-24) and an excess-64 base-2 exponent (bits 25-31). The binary point is assumed to precede bit 1. The floating-point format is given by Figure 3.29

3 !" % T U G& % % T U

: ?; ?C @

) @1?F1 # ! %)< % !

,8 -!# 7 ' % ) - % $

< G ?mn5A ' )% :

< <T < U GmYY5; ' )% 1

% )! - % )% !% ' !" % % 8 A

" &# % ' 1

G& % % G" <=;1 ! / G& % % & %

$ %! $ ( -!# % ' G& % % -!# &# =;1 + '

G! &# / ' G& % % ' # " % ! ! >< %! $ % )

4 / % ' !" % # &# $ ?:[=; < ?Y=; / '

G& % % ' # ?> % ' !" % # &# $